Integrity Coded Relational Databases (ICRDB) - Protecting Data Integrity in Clouds

Jyh-haw Yeh
Dept. of Computer Science, Boise State University,
Boise, Idaho 83725, USA

Abstract

1 Introduction

Database-as-a-service (DaaS) has been commercialized just recently in cloud industry such as Salesforce's Database.com, Amazon's Relational Database Service, Heroku's SQL DaaS, and Google's Google Cloud SQL, etc. However, outsourcing data to clouds, in addition to data privacy, data integrity is another important concern that slows down the adoption of this new emerging DaaS technology.

This paper describes an Integrity Coded Relational Database (ICRDB) that was designed to protect data integrity for the outsourced database in clouds. ICRDB ensures clients the detection of three types of attacks from malicious clouds. These three attacks are

1. Incomplete attack: For a query that returns a set of data, the cloud doesn't honestly return all matched data.

2. Forgery attack: The cloud returns some forged data. This type of attacks includes fabrication attacks and substitution attacks.

3. Unfresh data attack: The cloud returns unfresh data (has been removed) rather than up-to-date data.

Clients may request clouds to update or remove data. Untrusted clouds can maliciously keep these removed data along with their integrity codes and launch unfresh data attack later. The unfresh data and their integrity codes may bypass the integrity check. Thus, the better way to deal with the unfresh data attack will be a scheme that is similar to X.509 standard with Certificate Revocation List (CRL) used in public key cryptosystems.

Applying the x.509 concept here, each generated integrity code for the database will have an incremental serial number. The integrity code thus can be an ID-based signature with the serial number embedded inside the signature. The client requires to maintain an Integrity Code Revocation List (ICRL) that keeps track of the integrity codes of those removed/replaced data. In this way, the client will have enough information to ensure the freshness of returned data.

Any query returns a group of entities/values that share some properties, specified by the conditions in SQL WHERE clause. Ideally, if each such group in a database is assigned an unforgeable integrity code, then the incomplete and forgery attacks can be easily detected. However, the number of possible groups in a relational database is exponential to the number of data items (attribute values). Even worse, it will be extremely expensive to update data since all the group integrity codes related to the updated data will need to be re-assigned. Obviously, this approach is not practical.

We proposed an approach that only assigns three integrity codes SVC (Single-Value Code), SGC (Single-Group code) and $3VC$ (3-Value Code) to each attribute value. In addition, another integrity code CGC (Column Group Code) is assigned to each column. Thus, the number of integrity codes in our approach will be just linear to the number of data items in the database.

The client generates all the integrity codes, attaches all of them (except CGC) to their corresponding attribute values and then outsources the whole database to the clouds. Let $A(e)$ be the value of attribute A for an entity e. These four integrity codes are described as below:

1. Single-Value Code $SVC(A(e))$ is an unforgeable code that couples the attribute value $A(e)$ to its owner entity e together.
2. Single-Group Code $SGC(A(e))$ is an unforgeable code that groups all entities e' whose attribute value $A(e') = A(e)$.
3. 3-value Code $3VC(A(e))$ is an unforgeable code that couples three values $A(e_1)$, $A(e_2)$, and $A(e_3)$ together, where $A(e_1)$ is the next lower value than $A(e)$ in the column, $A(e_2) = A(e)$, and $A(e_3)$ is the next higher value than $A(e)$ in the column.
4. Column-Group Code $CGC(A)$ is an unforgeable code that groups all values of the entire column under the attribute A.

The first three codes are associated with each attribute value. The number of SVC's, SGC's and $3VC$'s grows linearly as data accumulated. It's not practical to store these integrity codes locally. Thus, these three kinds of integrity codes will be outsourced to the clouds along with data. Different from these three codes, a CGC is associated with a column rather than each attribute value. The number of CGC's in a database is fixed and which is equal to the number of attributes in the database. Therefore, CGC's can actually be kept in client's side to reduce the complexity of data updates.

2 Example Database

Throughout this document, we use the company database in Elmasri/Navathe's book "Fundamentals of Database Systems" as example to illustrate our integrity algorithms. The example database is shown in the last page of the document.

3 Communication Architecture

Since the ICRDB requires clients performing integrity code generation and verification, as well as modifying SQL queries to request extra information for integrity verification (described in next section), a proxy server taking care of all these work is desired. The proxy server should provide a GUI API for the end users for friendly usage. Figure 1 shows the communication architecture between clients and clouds.

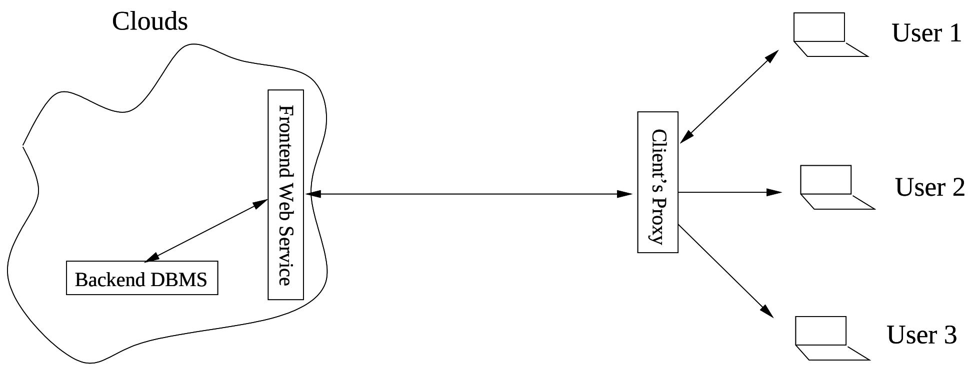

Figure 1: ICRDB communication architecture between clients and clouds

4 ICRDB

4.1 Integrity Codes Construction

We choose to use the RSA signature scheme to generate the integrity codes. The reason of using RSA is because of its multiplication homomorphism. With this property, re-assigning new integrity codes will be much more efficient in case of inserting, updating or removing data. The integrity code generation needs to ensure the unforgeability of grouping but also needs to make sure the unforgeability of each attribute value to its owner (i.e., the entity owns the attribute value). In relational databases, each table has a primary key attribute K that can uniquely identify each entity. Thus, the following describes the format of the integrity codes, where $SIG\{v\}$ stands for the RSA signature on v. That is,

$$SIG\{v\} = v^d \bmod N \tag{1}$$

where d, e and N are RSA keys.

1. $SVC(A(e))$ is a pair of quantities

$$SVC(A(e)) = (SIG\{s \times A \times A(e) \times K(e)\}, s) \tag{2}$$

   where $\times$ means multiplication, s is a unique serial number for each integrity code, A is the attribute name, and $K(e)$ is the primary key value of the entity e.

2. $SGC(A(e))$ is a pair of quantities

$$SGC(A(e)) = (SIG\{s \times A \times A(e) \times K(e_1) \times K(e_2) \times \ldots \times K(e_n)\}, s) \tag{3}$$

   where $e_1, \ldots e_n$ are all entities (including e) in the table whose attribute value $A(e_i) = A(e)$.

3. $3VC(A(e))$ is a pair of quantities

$$3VC(A(e)) = (SIG\{s \times (A + x + A(e) + z)\}, s) \tag{4}$$

   where $+$ means concatenation and the size of the concatenated string must be smaller than the size of RSA's modulo N. x and z in Equation (4) are next lower and next higher values than $A(e)$ in the column, respectively. Both x and z could be NULL if no next lower or no next higher values. If $A(e)$ is the MIN (or MAX) in the column, then x (or z) will be NULL. This $3VC(A(e))$ is different from the above two codes in that it does not tight the value

$A(e)$ to its key attribute value $K(e)$. This code, along with either $SVC(A(e))$ or $SGC(A(e))$, will be used together to check the completeness for a range query. Another difference is that we use concatenation rather than multiplication to construct the signature. The four concatenated quantities can be recovered by computing

1) $s \times (A + x + A(e) + z) = (SIG\{s \times (A + x + A(e) + z)\})^e \bmod N$, where e is the RSA public key. Note that even (e, N) is the RSA public key, but it will be only known by the client since in this application, only the client generates and verifies signatures.
2) $A + x + A(e) + z = s^{-1} \times s \times (A + x + A(e) + z) \bmod N$, where s^{-1} is the multiplicative inverse of $s \bmod N$.

4 CGC(A) is a single quantity

$$CGC(A) = SIG\{A \times A(e_1) \times A(e_2) \times \ldots \times A(e_n)\} \quad (5)$$

where $A(e_1), A(e_2), \ldots, A(e_n)$ inside the SIG function are all values under attribute A. No serial number is required for CGC's since they are stored locally in the client's side.

4.2 Reducing number of integrity codes

All attribute values (except the key values) need to have the integrity code SVC. Theoretically each non-key attribute value can have up-to three integrity codes. This full assignment is overkill in some cases. The following lists some attribute values which do not need to have three integrity codes.

If an attribute has a "unique" keyword in its DDL definition, any such attribute value's SVC and SGC actually provide the same integrity checking function. For example, if mgrssn is defined as unique, then $SVC(mgrssn)$ is the same as $SGC(mgrssn)$ since both codes couple a single mgrssn to its department number (key value).

Values of an attribute that have no natural ordering often do not need to have the $3VC$ integrity codes since $3VC$ is used for completeness checking in a range query. It doesn't make sense to have attributes without natural ordering in a range condition. For example, mgrssn has no natural ordering (yes, sometimes we may need to sort the mgrssn values in a report, but such alphabetical sorting is not a natural ordering of mgrssn) and thus mgrssn does not need to have $3VC$. For example, it doesn't make sense to have a query "retrieve the names of departments, of which the manager's social security number is greater than '123456789';" There are some other attributes in the company database may not need to have $3VC$ either such as address, sex, superssn and dno in the employee table, both dnumber and dlocation in the dept_locations table, and plocation and dnum in the project table. Of course, whether to assign a $3VC$ to an attribute needs to be pre-determined by clients because they are the data owners and will know whether range queries are applicable to that attribute.

Based on the above guidelines, Table 1 to Table 3 shows a possible integrity code assignment for some tables in the company database.

4.3 Integrity code verification

Clients are able to directly verify the integrity of returned query results if the query is the most basic query with none or one condition. Using the company database as an example, consider the following two queries:

Q1:"retrieve the names of all employees in the company;" and
Q2:"retrieve the names of employees who work for department 5."

Table 1

EMPLOYEE								
	name	ssn	bdate	address	sex	salary	superssn	dno
SVC	x		x	x	x	x	x	x
SGC	x		x	x	x	x	x	x
$3VC$	x		x			x		
CGC	x	x	x	x	x	x	x	x

Table 2

DEPARTMENT				
	dname	dnumber	mgrssn	mgrstartdate
SVC	x		x	x
SGC				x
$3VC$	x			x
CGC	x	x	x	x

Table 3

DEPENDENT					
	essn	dependent_name	sex	bdate	relationship
SVC	x	x	x	x	x
SGC	x	x	x	x	x
$3VC$		x		x	
CGC	x	x	x	x	x

For Q1, if the returned result is Table 4 below, with the column integrity code CGC(name) = $SIG\{$ name $\times$ Smith $\times$ Wong $\times$ Zelaya $\times$ Wallace $\times$ Narayan $\times$ English $\times$ Jabbar $\times$ Borg $\}$, the client should be able to verify the completeness and non-forgery of the result.

Table 4

name
Smith
Wong
Zelaya
Wallace
Narayan
English
Jabbar
Borg

For Q2, if the returned result is something like Table 5 below, the client can then check the integrity

Table 5

name	ssn	dno
Smith, SVC	123456789	5, SGC
Wong, SVC	333445555	5, SGC
Zelaya, SVC	666884444	5, SGC
English, SVC	453453453	5, SGC

as follows:

1. Verify the completeness of the returned result: Compute the $SIG\{s \times \text{dno} \times 5 \times 123456789 \times 333445555 \times 453453453 \times 666884444\}$ and check whether it is equal to the one in any SGC of the return result.
2. Check forgery attacks: For each name (say Smith), compute $SIG\{s \times \text{name} \times \text{Smith} \times \text{ssn}\}$ and compare it to the corresponding SVC in the returned result.
3. Check the freshness of data: For each data item inserted, the client will generate an integrity code with a new serial number. Use the similar scheme as modified X.509 to keep track of removed items's integrity codes so that if a returned result containing already removed data, it can be detected.

4.4 Parsing a query (query transformation)

Using example queries Q1 and Q2 above, their original SQL queries would look like

Q1: select name from employee;
Q2: select name from employee where dno=5;

With CGC's stored in the client's side, Q1 actually returns enough information for the integrity checking. However, for Q2, the query asks the cloud to return names of employees in department 5, which does not contain enough information for integrity checking. In order to get the required information, the SQL query needs to be modified to something like

```
Q2': select name, SVC(name), ssn, dno, SGC(dno)
     from employee
     where dno=5;
```

Thus, in the client proxy server, a software or an application programming interface API (can be developed by either the client, the clouds or a third-party software developer) is needed for parsing (converting) a standard query to a modified query as above.

4.5 Range queries

The design of $3VC$ integrity code is for checking the completeness of returning results in range queries. Query 3 below shows a range query example.

```
Q3: select name from employee where bdate > '1965-12-31';
```

For the example company database, three employees were born after 1965, who are Zelaya (1968-01-19), English (1972-07-31) and Jabbar (1969-03-29). If the cloud is honest, the cloud would return the three employees plus some information for integrity checking as shown in Table 6.

Table 6

name	ssn	bdate
Zelaya, SVC	999887777	1968-01-19, SGC , $3VC$
English, SVC	453453453	1972-07-31, SGC , $3VC$
Jabbar, SVC	987987987	1969-03-29, SGC , $3VC$

In order to include necessary integrity codes as in Table 6, the query API should modify Q3 to

```
Q3': select name, SVC(name), ssn, bdate, SGC(bdate), 3VC(bdate)
     from employee
     where bdate > '1965-12-31';
```

Now, the client's API checks the completeness using the returned $3VC$'s. The checking starts at the oldest employee Zelaya's $3VC$, which can recover the birthdates of Smith, Zelaya, and Jabbar. This checking ensures that no other employees were born within this range 1965-01-09 (Smith's birthdate) to 1969-03-29. This implies no other employees were born from 1965-12-31 to 1969-03-29. Next $3VC$ needs to be checked is English's $3VC$, which ensures that no other employees (except English) were born after 1969-03-29.

After the completeness checking, check the SGC's to ensure that each bdate is correctly associated with its ssn. In this example, the check of SGC's also ensures that no un-returned employee was born on the same date as those employees in Table 6. Finally, check the SVC's to ensure that each name is correctly associated with its ssn. Successful checking of both SGC and SVC, we know that each name is correctly associated with the bdate in the returned table.

4.6 Queries with multiple conditions

Let's consider a query Q4: "retrieve the names of employees who work for department 5 and were born after 1965;"

```
Q4: select name
    from employee
    where bdate > '1965-12-31' and dno = 5;
```

The result of this query can actually be derived by finding an intersection of two sets: the set of all employees in department 5 and the set of employees who were born after 1965. Table 7 shows the correct result.

Table 7

name	ssn	bdate	dno
English, SVC	453453453	1972-07-31, SGC , $3VC$	5, SGC

However, only returning the above result is not enough for completeness checking since the client will need all ssn who work in department 5 to verify the returned SGC, but the table above does not contain those employees in department 5 who were born before or in 1965. Similarly, the returned bdate data is also not enough for completeness checking since there are some other employees not included in the above table but who were born after 1965.

Therefore, more information is required for clients to perform integrity checking. In order to do so, the query API in the client side should transform the query to

Q4': select name, SVC(name), SSN, bdate, SGC(bdate), 3VC(bdate), dno, SGC(dno)
from employee
where bdate > '1965-12-31' OR dno = 5;

Note that the above modified query changes the condition operator from AND to OR so that the modified query would return the UNION of two sets rather than the INTERSECT of two sets. For the modified query, a trustworthy cloud would return the information shown in Table 8 below:

Table 8

name	ssn	bdate	dno
Smith, SVC	123456789	1965-01-09, SGC , $3VC$	5, SGC
Wong, SVC	444335555	1955-12-08, SGC , $3VC$	5, SGC
Zelaya, SVC	999887777	1968-01-19, SGC , $3VC$	4, SGC
Narayan, SVC	666884444	1962-09-15, SGC , $3VC$	5, SGC
English, SVC	453453453	1972-07-31, SGC , $3VC$	5, SGC
Jabbar, SVC	987987987	1969-03-29, SGC , $3VC$	4, SGC

After receiving the above information, the client's API performs the steps below :

1. Checking the completeness and non-forgery of bdate by verifying the bdate's $3VC$ and SGC of all employees who were born after 1965. If all the $3VC$'s and SGC's are valid, it indicates all employees who were born after 1965 are returned. In this example, the bdate $3VC$'s of Zelaya and English will be checked. By checking each SGC of bdate, we can ensure that no other not-returned employees whose bdates are the same as those returned, as well as each bdate is correctly associated with its key value.

2. Checking the completeness and non-forgery of dno by verifying the dno's SGC of department 5. If the SGC is valid, it indicates all employees in department 5 are returned.

3. Select tuples that satisfy both conditions, bdate $>' 1965-12-31'$ AND $dno = 5$, from Table 8, which results in Table 9 below:

4. Checking the non-forgery of each name by verifying its SVC. If the SVC is valid, it indicates the result is correct. The API then returns the result as Table 10 below to the client.

Table 9

name	ssn	bdate	dno
English, SVC	453453453	1972-07-31, SGC , $3VC$	5, SGC

Table 10

name
English

4.7 Queries across multiple tables with JOIN operations (key and foreign key relationship)

Usually, to access data across two tables, it requires a "JOIN" condition that specify the relationship between the primary key of one table and a foreign key of another table. Use query 5 below as an example: "retrieve the names of managers and their department names ;"

Q5: select name, dname
from employee, department
where ssn = mgrssn;

To have enough information for integrity checking, the modified query may look like:

Q5': select name, SVC(name), ssn, mgrssn, SVC(mgrssn), dnumber, dname, SVC(dname)
from employee, department
where ssn = mgrssn;

which would return information as in Table 11.

Table 11

name	ssn	mgrssn	dnumber	dname
Wong, SVC	333445555	333445555, SVC	5	Research, SVC
Wallace, SVC	987654321	987654321, SVC	4	Administration, SVC
Borg, SVC	888665555	888665555, SVC	1	Headquarter, SVC

The client's proxy receives the above information and then performs

1. Completeness checking: Verify the $CGC(mgrssn)$ to ensure the returned result containing all managers.
2. Query condition checking: Loop through the result to check all tuples having the same ssn and mgrssn.
3. Non-forgery checking: Verify all returned SVC's to ensure that each returned name is indeed the name of the corresponding ssn and each returned dname and mgrssn are indeed the dname and mgrssn of the corresponding dnumber.
4. After all integrity codes checked, the query API returns the result shown in Table 12 to the client.

Table 12

name	dname
Wong	Research
Wallace	Administration
Borg	Headquarter

4.7.1 Which CGC code should be checked in a join condition?

From the above example, you may notice we only check one of the two *CGC* codes of the two attributes in the join condition. The general guidelines of which *CGC* to be checked are

1. If only one attribute is a total participation (TP) in the relationship, then check the *CGC* of that attribute. For example,

   employee.ssn = department.mgrssn, where department.mgrssn is a TP.
   employee.ssn = dependent.essn, where dependent.essn is a TP.

2. If both attributes are total participation in the relationship and the cardinality ratio is 1: N, then check the *CGC* of the N side attribute. For example,

   employee.dno = department.dnumber, where employee.dno is the N side.

3. If both attributes are partial participation in the relationship, then check the *CGC*'s of both attributes. For example,

   Q6: select e.name s.name
   from employee e, employee s
   where e.superssn = s.ssn;

   Another example, query 7: "retrieve the department names whose managers have dependents."

   Q7: select distinct dname
   from department, dependent
   where mgrssn = essn;

   For the query 7, in order to use *CGC*'s to verify the completeness of result, all values of mgrssn and essn need to be returned. Thus, the modified query is

   Q7': select dname, SVC(dname), dnumber, mgrssn, SVC(mgrssn), essn
   from department, dependent
   where mgrssn]=[essn;

   where]=[means full outer join. Note that the modified query does not have the "distinct" key word. Table 13 shows the returning result of Q7'.

   The query API verifies the integrity by

   1) Checking completeness: Verify $CGC(mgrssn)$ and $CGC(essn)$ to make sure the table contains all mgrssn and all essn.

Table 13

dname	dnumber	mgrssn	essn
Research, SVC	5	333445555, SVC	333445555
Research, SVC	5	333445555, SVC	333445555
Research, SVC	5	333445555, SVC	333445555
Administration, SVC	4	987654321, SVC	987654321
Headquarter, SVC	1	888665555, SVC	null
null	null	null	123456789
null	null	null	123456789
null	null	null	123456789

Table 14

dname	dnumber	mgrssn	essn
Research, SVC	5	333445555, SVC	333445555
Research, SVC	5	333445555, SVC	333445555
Research, SVC	5	333445555, SVC	333445555
Administration, SVC	4	987654321, SVC	987654321

2) Checking the original query condition: Loop through the table and get rid of tuples which do not have matching mgrssn and essn. The remaining table is shown in Table 14.

3) Checking non-forgery: Verify all SVC's to make sure that each pair of mgrssn and dname in each tuple are not forged, and thus, are indeed for the same department.

4) Performing the selection part of the original query and then return the result to the client.

4.8 Aggregate functions

4.9 Aggregate functions over a whole table

With the construction of the integrity code $3VC$, the integrity checking of query results with aggregate functions MIN and MAX over a whole table can be solved easily.

Q8: select name, MIN salary
from employee;

$\Longrightarrow$

Q8': select name, SVC(name), ssn, MIN salary, SGC(salary), 3VC(salary)
from employee;

From the returned $3VC$, the salary can be recovered and it can be ensured that it is indeed a min salary if the next lower salary is a NULL. The $SGC(salary)$ ensures the returned ssn is the only employee with the minimum salary. Similarly, the MAX function works the same way.

Q9: select name, MAX salary
from employee;

$\Longrightarrow$

Q9': select name, SVC(name), ssn, MAX salary, SGC(salary), 3VC(salary)
from employee;

For COUNT, SUM, or AVE functions, the entire column needs to be returned for integrity checking.

Q10: select COUNT *
from employee;
$\Longrightarrow$
Q10': select any attribute A
from employee;

Q11: select AVE salary
from employee;
$\Longrightarrow$
Q11': select salary
from employee;

After the completeness check of the returned table using an appropriate CGC, the API returns the user the number of tuples or the average salary of the whole table, respectively. Similar techniques can be applied to the function SUM.

4.9.1 Aggregate functions over a partial table

If the aggregate functions MIN, MAX, COUNT, SUM and AVE are applied to a partial table, usually the SGC and/or $3VC$ can ensure the completeness.

Q12: select name, MIN salary
from employee
where dno = 5;
$\Longrightarrow$
Q12': select name, SVC(name), ssn, salary, SVC(salary), dno, SGC(dno)
from employee
where dno = 5;

The returned $SGC(dno)$ in Q12' can verify the completeness.

Q13: select COUNT *
from employee
where bdate > '1965-12-31';
$\Longrightarrow$
Q13': select ssn, bdate, SGC(bdate), 3VC(bdate)
from employee
where bdate > '1965-12-31';

After the query API verifies the integrity codes for completeness and non-forgery of the returned table, it performs the original MIN, MAX, COUNT, SUM or AVE function over the table and returns the result to the client.

4.9.2 Aggregate functions with grouping attributes

In order to verify the completeness and non-forgery, the entire column of the grouping attribute and the attribute applying the aggregate function, along with integrity codes, need to be returned.

Q14: select dno, AVE(salary)
from employee
group by dno
having (COUNT *) > 2;

$\Longrightarrow$

Q14': select dno, SVC(dno), ssn, salary, SVC(salary)
from employee;

After checking the completeness (using $CGC(dno)$) and non-forgery (using SVC's) of the returned table, the query API applies the aggregate function to each group and return the result to the client.

4.10 Nested Queries

First of all, the clients should avoid to use nested SQL queries if single level SQL queries can do the job. In this document, we proposed two approaches to deal with nested queries:

1. The query API converts the nested query to a single level query if possible. The integrity codes should be included in the converted query for the completeness and non-forgery checking as described in previous sections.

2. The query API breaks a nested query to several basic queries. Each basic query will return a set of values. The result of the nested query can actually be derived by performing some set operations on these sets of values. To ensure completeness, set operations on these sets will be performed by the client's query API rather than by the clouds. Thus, the query API needs to store all these returned sets until the final result of the original nested query is generated. The integrity codes should be included in each basic query for the completeness and non-forgery checking as described in previous sections.

Approach 2 always works for any nested query. However, this approach is less efficient. We would suggest using approach 1 if possible. If a nested query cannot be converted to a single level query, approach 2 will be applied.

In this section, we would use some examples to demonstrate these two approaches, as well as some algorithms of converting nested queries to single level queries.

4.10.1 Examples to use approach 2

This section gives two examples to show how to use approach 2 to take care of nested queries.

Q15: retrieve the names of employees whose salary is more than all employees in department 5.

Q15: select name
from employee
where salary > all (select salary
from employee
where dno = 5);

The above request cannot be done by a single level query. Thus, we would use approach 2 to break it to two basic queries as follows: The query API first issues Q15' below to the clouds and waiting for the result.

Q15': select salary, SVC(salary), ssn, dno, SGC(dno)
from employee
where dno = 5;

After the query API verifies the completeness and non-forgery of the result, the API finds the MAX salary of the result and let it be MAX_SALARY. A second basic query Q15" below will be issued to the clouds.

Q15": select name, SVC(name), ssn, salary, SGC(salary), 3VC(salary)
from employee
where salary > MAX_SALARY;

Again the query API verifies the integrity codes for the returned result and then report the names to the clients.

The second example query 16 using approach 2 is described below:

Q16: retrieve the names of employees who work on all projects located in 'Houston'.

This query cannot be represented by a single level query. The following is the nested query for it:

```
Q16: select name
     from employee
     where (select pno
             from works_on
             where ssn = essn)
            contains
            (select pnumber
             from project
             where plocation = 'Houston');
```

The query API breaks it to two basic queries Q16A and Q16B as follows, and sends them both to the clouds.

```
Q16A: select pnumber, plocation, SGC(plocation)
      from project
      where plocation = 'Houston';
```

```
Q16B: select name, SVC(name), ssn, essn, pno, SGC(essn)
      from employee, works_on
      where ssn = essn;
```

Two sets of results will be returned from the clouds, the query API verifies the integrity of both sets first. Let the two sets are set1 and set2, which are the results of Q16A and Q16B, respectively. The API can then perform the following query over these two sets:

```
Q16C: select distinct name
      from set2 B
      where (select pno
              from set2 C
              where B.ssn = C.ssn)
             contains
             (select pnumber
              from set1);
```

The above query actually can be done by a procedure with a nested loop if the API does not have the query processing capability.

4.10.2 Examples to use approach 1

We use nested queries 17 to 20 below to demonstrate how to use approach 1.

```
Q17: select name
     from employee
     where ssn in (select essn
                   from dependent);
⟹
Q17': select name
     from employee, dependent
     where ssn = essn;
⟹
Q17'': select name, SVC(name), ssn, essn
     from employee, dependent
     where ssn = essn;
```

The returned essn will be CGC checked for completeness.

```
Q18: (select pnumber
      from project, department, employee
      where dnum = dnumber and mgrssn = ssn and name = 'Smith')
     UNION
     (select pno
      from works_on, employee
      where essn = ssn and lname = 'Smith');
⟹
Q18': select pnumber, dnum, SGC(dnum), dnumber, mgrssn, SVC(mgrssn),
            ssn, name, SVC(name), essn, pno, SGC(essn)/SGC(pno)
     from project, department, employee, works_on
     where (dnum = dnumber and mgrssn = ssn and name = 'Smith')
            OR
            (pnumber = pno and essn = ssn and name = 'Smith');

Q19: select essn
     from works_on a
     where exists (select pno, hours
                   from works_on b
                   where a.pno = b.pno and a.hours = b.hours
                         and b. essn = '123456789');
⟹
Q19': select a.essn, a.pno, SGC(a.essn)/SGC(a.pno), a.hours, SGC(a.hours),
            b.essn, b.pno, SGC(b.essn)/SGC(b.pno), b.hours, SGC(b.hours),
     from works_on a, works_on b
     where a.pno = b.pno and a.hours = b.hours and b.essn = '123456789';

Q20: select name
     from employee
     where salary > any (select salary
                         from employee
```

where dno = 5);

$\Longrightarrow$ This query is equivalent to

Q20': select distinct a.name
from employee a, employee b
where a.salary > b.salary and b.dno = 5;

$\Longrightarrow$ API convert to

Q20": select distinct a.name, SVC(a.name), a.ssn, a.salary, SGC(a.salary),
b.ssn, b.salary, SGC(b.salary), b.dno, SGC(b.dno)
from a.employee, b.employee
where a.salary > b.salary and b.dno = 5;

In summary, the following lists some algorithms for converting nested queries to single level queries:

1. If a nested query uses "exists", or "in" in conditions, it usually can be converted to a single-level query. For example: Q17, Q19.

2. Queries with set operations UNION/INTERSECT may be converted to conditions with OR/AND respectively. For example, Q18.

3. Queries using "> (=, <) any (some)" can be converted by the following rule:

   "select a from T where b >=< any (select c from R)"
   $\Longrightarrow$ "select distinct a from T, R where $b >=< c$"

   "> (= *or* <) some": equivalent to "> (= *or* <) any" , For example, Q20.

4.11 Insert A Tuple

Inserting new tuples may affect some existing integrity codes and thus code re-assignment is required. Before the insertion, the client needs to request the cloud to return information listed below for new integrity codes generation.

1. Re-assignment of all CGC's in the table: All CGC codes will be affected by adding a new tuple. Because we choose to use RSA signature scheme with multiplication homomorphism, the re-generation of CGC's do not need to reference the data stored in clouds. The client's API assigns a new CGC for each attribute A. The new CGC will be

$$CGC(A)_{new} = SIG\{A(new)\} \times CGC(A)_{original} \bmod N \tag{6}$$

   where $A(new)$ is the value of attribute A of the to-be inserted tuple and $CGC(A)_{original}$ and $CGC(A)_{new}$ are the CGC codes of A before and after the insertion.

2. The SVC's for the new tuple can be directly computed by the API. No existing SVC's stored in clouds will be affected.

3. For each attribute A having SGC , the client needs to request clouds to return the affected existing $SGC(A(e))$'s that have the same attribute value as $A(new)$, i.e., $A(e) = A(new)$. For example, if the new tuple to be inserted is an employee in department 5, then the cloud will be asked to return all SGC's (for dno=5). All these $SGC(A(e))$'s need to be renewed based on Equations (7) and (8).

$$SGC(A(e))_{new} = (SIG_{new}, s_{new}) \tag{7}$$

and

$$SIG_{new} = (SIG\{s_{new} \times K(new) \times (s_{old}^{-1} \bmod N)\} \times SIG_{original}) \bmod N \tag{8}$$

where s_{new} is the new serial number, $K(new)$ is the key value of the to-be inserted tuple and $s_{old}^{-1} \bmod N$ is the inverse of old serial number mod$N$. The client then sends back clouds the new SGC's to replace those original SGC's. All replaced SGC's will be revoked.

4. For each attribute A having $3VC$, the client issues a query asking clouds to return the affected $3VC$'s:

   select A, 3VC(A)
   from the table containing attribute A
   where A is the smallest but > A(new) OR
   A is the largest but < A(new) OR
   A = A(new);

   If one or more tuples returned having $A(new)$ value, then no existing $3VC$'s will be affected, except the one to-be inserted. Otherwise, let the two groups of $3VC$'s returned are

$$3VC(y) = (SIG\{s \times (A + x + y + z)\}, s) \tag{9}$$

   where y is the attribute A's value and y is the largest but $< A(new)$ in the column;

$$3VC(z) = (SIG\{s \times (A + y + z + w)\}, s) \tag{10}$$

   where z is the attribute A's value and z is the smallest but $> A(new)$ in the column;

   To re-assign each of the $3VC(y)$'s, we just have to compute the following steps:

   1) Recover $A + x + y + z$, given that the size of this concatenated string is not bigger than the RSA modulo N.
   2) If the returned y and z are substrings of the recovered string, then replace z by $A(new)$ in the recovered string
   3) Generate the new $3VC(y)_{new} = (SIG\{s_{new} \times (A + x + y + A(new))\}, s_{new})$

   Similarly, we can use the same technique to re-assign the new $3VC(z)$'s. Finally, all replaced $3VC$'s will be revoked.

4.12 Delete A Tuple

1. Re-assignment of all CGC's in the table: All CGC codes will be affected by deleting a tuple. The client's API assigns a new CGC for each attribute A. The new CGC will be

$$CGC(A)_{new} = SIG\{A(old)^{-1} \bmod N\} \times CGC(A)_{original} \bmod N \tag{11}$$

   where $A(old)$ is the value of attribute A of the to-be deleted tuple, $CGC(A)_{original}$ and $CGC(A)_{new}$ are the CGC codes before and after the deletion.

2. The SVC's of the to-be deleted tuple will be revoked. No other SVC's will be affected.

3. For each attribute A having SGC, the client needs to request clouds to return the affected existing $SGC(A(e))$'s that have the same attribute value as $A(old)$, i.e., $A(e) = A(old)$. If there is only one affected SGC with the same attribute value as $A(old)$, or in other words, the group has only one member that is going to be deleted, all we need to do is to revoke the only SGC. If more than one affected $SGC(A(e))$'s due to the deletion of $A(old)$, they need to be re-assigned. For example, if the to-be deleted tuple is an employee ($ssn = 123456789$) in department 5, then the cloud will be asked to return all SGC's (for $dno = 5$). All these affected $SGC(A(e))$'s need to be renewed based on Equations (12) and (13).

$$SGC(A(e))_{new} = (SIG_{new}, s_{new}) \tag{12}$$

and

$$SIG_{new} = SIG\{s_{new} \times (K(old)^{-1} \bmod N) \times (s_{old}^{-1} \bmod N)\} \times SIG_{original} \bmod N \tag{13}$$

where $K(old)$ is the key value for the to-be deleted tuple. The client then sends back clouds the new SGC's to replace those original SGC's. All replaced SGC's will be revoked.

4. For each attribute A having $3VC$, the client issues a query asking clouds to return the affected $3VC$'s:

select A, 3VC(A)
from the table containing attribute A
where A is the smallest but > A(old) OR
        A is the largest but < A(old) OR
        A = A(old);

If more than one tuples returned having $A(old)$ value, then no existing $3VC$'s will be affected, except the one to-be deleted. Otherwise, let the two groups of $3VC$'s returned are

$$3VC(y) = (SIG\{s \times (A + x + y + A(old))\}, s) \tag{14}$$

where y is the attribute A's value and y is the largest but $< A(old)$ in the column;

$$3VC(z) = (SIG\{s \times (A + A(old) + z + w)\}, s) \tag{15}$$

where z is the attribute A's value and z is the smallest but $> A(old)$ in the column;

To re-assign each of the $3VC(y)$'s, we just have to compute the following steps:

1) Recover $A + x + y + A(old)$, given that the size of this concatenated string is not bigger than the RSA modulo N.
2) If the returned y and $A(old)$ are substrings of the recovered string, then replace $A(old)$ by z in the recovered string.
3) Generate the new $3VC(y) = (SIG\{s_{new} \times (A + x + y + z)\}, s_{new})$.

Similarly, we can use the same technique to re-assign a new $3VC(z)$ by replacing $A(old)$ by y. All replaced $3VC$'s will be revoked.

4.13 Update A Value

Updating a value is functionally equivalent to first delete the value and then insert back a new value. Thus, the techniques used in both deletion and insertion can be used to update a value. For each updated value, the following integrity codes will be affected

1. The corresponding CGC .
2. SVC of the data to be updated.
3. All grouping code SGC's that the new data becomes a member.
4. All grouping code SGC's that the old data was a member.
5. At most four groups of $3VC$'s plus the $3VC$ for the data itself will be affected. Updating a data is just like removing the data first and then inserting a new data back. Thus, at most two groups of $3VC$'s will be affected by removing the old data and at most two groups of $3VC$'s will be affected by inserting the new data.

4.14 Reduce the size of ICRL

The size of Integrity Code Revocation List (ICRL) grows after each insertion, deletion or update. If the list gets too long, the client may want to reduce the size of the ICRL for storage and search efficiency. Similar to the modified X.509, the ICRL used here has a first valid serial number s_f, followed by a list of revoked serial numbers as below.

Table 15

s_f	all revoked serial numbers ($> s_f$) in an ascending order

The client performs the following steps to reduce the size of ICRL:

1. Identify s'_f ($> s_f$) as the new first valid serial number.
2. All valid integrity codes whose serial number $s < s'_f$ need to be renewed. To renew an integrity code SVC, SGC, or $3VC$,
    1) The client assigns a new serial number s_{new} (must be $> s'_f$) to replace the old serial number s_{old}.
    2) The client sends a triple of values $(SIG\{s_{new} \times (s_{old}^{-1} \bmod N)\}, s_{old}, s_{new})$ back to the clouds for the to-be-renewed integrity code.
    3) The cloud replaces the original integrity code $(SIG_{original}, s_{old})$ by $(SIG_{original} \times SIG\{s_{new} \times (s_{old}^{-1} \bmod N)\} \bmod N, s_{new})$.
3. The new ICRL will be

Table 16

s'_f	all revoked numbers ($> s'_f$) in an ascending order

5 Performance Evaluation

6 Conclusion

Reference

Figure 7.6 One possible relational database state corresponding to the COMPANY schema.

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John		Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin		Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia		Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer		Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh		Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce		English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad		Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James		Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
		Houston
		Stafford
		Bellaire
		Sugarland

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Figure 2: Example company database