STUDIA INFORMATICA 2011
Volume 32 Number 2B (97)

Krzysztof MIODEK
Uniwersytet Łódzki, Centrum Informatyki
Krzysztof PODLASKI, Ścibór SOBIESKI, Bartosz ZIELIŃSKI
Uniwersytet Łódzki, Wydział Fizyki i Informatyki Stosowanej

SOME REMARKS ON OPTIMIZATION IMPACT ON DATABASE SECURITY

Summary. One of the most important factors of real live business applications are speed and reliability. The question that arises during development states: what is more important: efficiency of servers or security of database/application. One of the biggest databases used in the University of Łódź for its applications must have restricted access to data. On the other hand, although it is used by many people concurrently cannot be overloaded. Security rules are based on views created for every user, which gives scalability and flexibility. Unfortunately this approach has security vulnerabilities which is presented in this article.

Keywords: database security, database optimization

PEWNE SPOSTRZEŻENIA NA TEMAT WPŁYWU OPTYMALIZACJI NA BEZPIECZEŃSTWO BAZ DANYCH

Streszczenie. W zastosowaniach biznesowych bardzo często, jako najważniejsze wskaźniki jakości rozwiązania, wskazuje się szybkość działania oraz niezawodność. W trakcie tworzenia takich rozwiązań pojawia się dylemat: wydajność serwera czy też jego bezpieczeństwo? Przed podobnym dylematem stanęli twórcy jednej z największych baz danych użytkowanych na Uniwersytecie Łódzkim, gdyż aplikacje ją używające musiały posiadać bardzo ograniczony dostęp do danych, a ponieważ aplikacje te używane są przez wiele osób, to istnieje problem przeciążenia bazy danych. Reguły bezpieczeństwa zostały oparte na widokach tworzonych dla każdego użytkownika, co daje dużą skalowalność i elastyczność rozwiązania. Niestety, takie rozwiązanie posiada pewne niedostatki związane z bezpieczeństwem, które zostały omówione w niniejszej publikacji.

Słowa kluczowe: bezpieczeństwo baz danych, optymalizacja baz danych

1. Previous works and our contribution

The need for fine grained database access control for sensitive business, personal or medical data, on the level of single rows, as opposed to crude table level control available in most database systems, is already well recognized. Among mechanisms proposed for implementing such access control, query rewriting [1, 2, 3, 4, 5] seems to be most promising. The basic idea is to modify the user's queries by adding appropriate WHERE conditions to every reference to the protected table in the query. The added conditions are to select only those rows the user is allowed to access. The most available basic mechanism to force such rewriting is to create views for each user and each protected table as well as to give users access to views only, instead of tables. Of course for systems with hundreds of tables and thousands of users such basic mechanisms are useless without some framework to easy administrative and storage burden but such systems were created both as in-house built systems (e.g. *Rektorat*) and database mechanisms for commercial databases (e.g. [5]).

Unfortunately query rewriting systems are susceptible to leakage of secret information through hidden communication channels [2]. The basic problem is that while the user obtains as query result only rows he is authorized to see, the expressions in the WHERE part of user query can be pushed down by optimizer in the query plan, and thus get to be executed before the selection of authorized rows [2]. Then information can leak because the user expressions may not be side effect free – either because they contain user defined functions which dump their arguments to some log table or because the execution of the expression throws an exception and the value of arguments can be inferred from the error message [2].

There are two most obvious ways to prevent such leaks: to modify the optimizer in order to prevent generating unsafe query plans [2] which is difficult and sometimes impossible when one does not have access to the database source code, or one can cut off the optimizer from bare tables, for instance by using table functions in the definition of a view – table functions are in most cases opaque row sources from the point of view of the optimizer. Unfortunately the second solution is very degrading for the efficiency – not only it prevents the optimizer from rewriting the query into more efficient but possibly unsafe one, but it also makes it impossible to use the indexes on the protected tables.

Our contribution is the analysis of the problem in the context of PostgreSQL database and the working University of Łódź database system with an in-house created row level access control system [1].

2. Experiences with two approaches to row level access management

In [1] were presented two versions of row level access management system developed for applications created in the University of Lodz based on PostgreSQL database. First version of that system was based on a tight access control provided by a function invoked in views to every table in the database. The function returned only rows, to which client had an access. That approach had one main disadvantage – it was very time consuming. Using function in every view meant, that indexes were not taken into account by the database optimizer and the optimizer could not approximate the number of rows returned by the functions. The result was that query plans were far from optimal, which degraded database performance significantly.

Second approach loosened some security restrictions in order to gain better response times for each query. Instead of one view for each table providing access control by a function, static views were created. As every user might have had different permissions and each application option might have required different data ranges and also varied permissions, many views for each table and user were required, one for every option, to which user had an access.

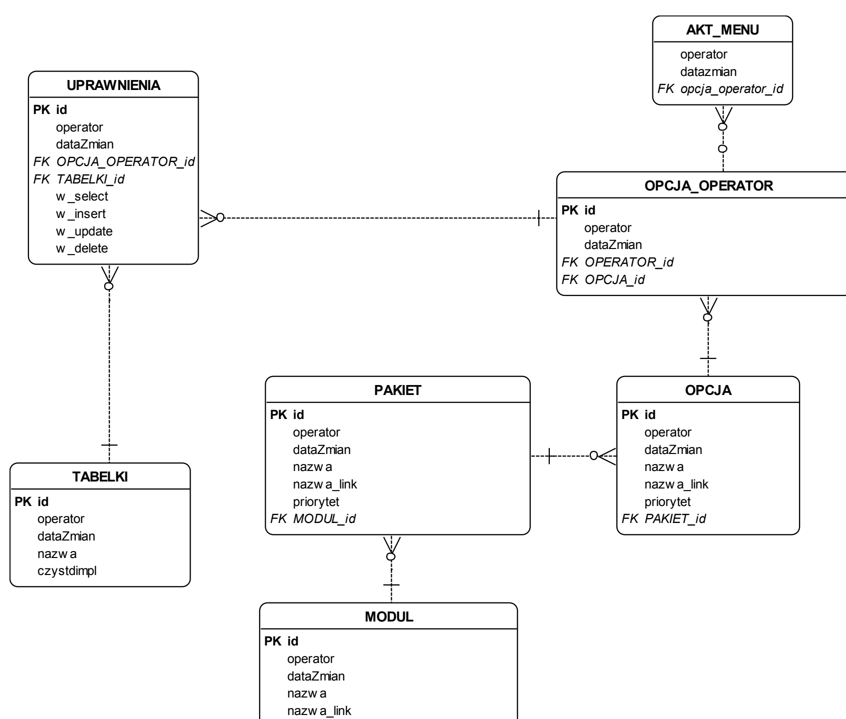

Fig. 1. Access restriction mechanism
Rys. 1. Mechanizm ograniczenia dostępu

PostgreSQL unique functionality allowing modification of search path variable was used. For every option a given user has access, a separate schema is created. In that schema for every table needed, a view is created with appropriate WHERE clause restricting access to data rows. Application for every user action starts a new transaction. At the beginning of the transaction, application provides option id in which action takes place by updating special

table akt_menu. A trigger on that table sets a proper search_path providing an access management by selecting right schema with proper set of views. At first all views were to be created while setting user permissions. This approach proved to be ineffective as the number of views to create for every user were in hundreds. Updating permissions for all users of the system took hours. It was decided to delay creating actual views to the moment, when they are really needed, ie. when a user selects a given option for the first time. Trigger on the mentioned akt_menu table checks whether views are already created and invokes function creating them if needed. This increases the time user waits for the response when using some options for the first time, but that was acceptable. Another gain was in the fact, that some options are never used by the users, so unnecessary views were not created in the database. The figure below shows tables providing access restriction mechanism.

Table “opcja_operator” contains information about which options are available for each user. Table “uprawnienia” contains permissions to specific tables for every pair user-option. Table akt_menu contains triggers governing access restriction mechanism, providing proper search path and ensuring that needed views exist. In a default search path there are only main user schema and public schema. If proper search path is not set, user has no access to data at all.

Application is required to set akt_menu to NULL before the end of transaction, which resets search path to default.

2.1. Performance comparison

For performance tests two representative options were selected. To avoid external influence on performance monitoring, tests were conducted on separated system. Prepared scripts were run 100 times.

Table 1

Performance test results

No access control	Function based views	Static views in schemas
62s	880s	208s

“No access control” tests were conducted on a user with access granted to tables containing data without any views covering them and all triggers turned off, simulating plain database without row level access control.

Performance of functions based on an access control was unacceptable. Rewriting the system to static views provided over four times better performance, which has given acceptable response times for most options. It is worth noting, that users with broad permissions (ie. to all rows in tables) have better performance than users with more complicated permissions. Tests above were conducted for a typical user with permissions narrowed to some subset of data in most of the tables. Permissions on some tables relied on permissions to correlated data in other tables.

3. Unauthorized data access

Let us explore the security weaknesses of current [1] system.

3.1. Looking for forbidden data

At first we select data (row) which our test user cannot see, and try to obtain any information about it using simple SQL query:

```
SELECT * FROM harmon WHERE id=16105; (1)
```

This gives us, 0 rows as result. It is worth to note that usually conditions defining the view harmon do not allow test user to see the chosen row. The only possibility is to work "silently" before view restrictions rules are applied. First let us try to create an error prepared specially for this row. If security rules worked as intended no error would occur. Unfortunately the query:

```
SELECT * FROM harmon WHERE (id=16105 AND 1/(id-16105)=0);
```
(2)

gives back the message: `ERROR: division by zero.`

From this, user knows that there is something he is not allowed to see. Now let us try to specify when we could expect potentially dangerous situation.

Optimizer implemented in PosgreSQL database will always try to apply first most narrow condition on an indexed column. Consider the following WHERE conditions supplied by the user:

1. `id=16105 AND 1/(id-16105)=0` (3)
2. `id in (15, 33, 16105, 16210) AND 1/(id-16105)=0` (4)
3. `id>16103 AND 1/(id-16105)=0` (5)

Cases 1,2 cause errors and 3 is safe and returns 0 rows. We should remember that the definition of the view harmon has complicated restriction rules and id is an indexed column for this view. To understand the mechanism we should look at the query plans using SQL command EXPLAIN.

For query (1) we obtain:

```
Nested Loop IN Join  (cost=719.16..1152.63 rows=1 width=655)
   Join Filter: ("outer".zatrudn_id = "inner".id)
   ->  Index Scan using harmon_pkey on harmon  (cost=0.00..5.75 rows=1
width=655)
         Index Cond: (id = 16105)
   ->  Hash IN Join  (cost=719.16..1143.53 rows=268 width=4)
         Hash Cond: ("outer".stanowis_id = "inner".id)
         ->  Seq Scan on zatrudnienie  (cost=0.00..400.68 rows=2802 width=8)
               Filter: ((s_grupprac_id = 1) OR (s_grupprac_id = 2))
         ->  Hash  (cost=714.44..714.44 rows=1885 width=4)
               ->  Hash IN Join  (cost=38.30..714.44 rows=1885 width=4)
```

```
                     Hash Cond: ("outer".jednostka_id = "inner".id)
                     ->  Seq Scan on stanowisko  (cost=0.00..558.86 rows=19686
width=8)
                     ->  Hash  (cost=38.02..38.02 rows=115 width=4)
                           ->  Seq Scan on jednostka  (cost=0.00..38.02 rows=115
width=4)
                                 Filter: (((kod)::text ~~ '__________'::text)
AND czyaktualne AND ((kod)::text ~~ '__________'::text))
```

On the other hand for query (2) we have:

```
 Nested Loop IN Join  (cost=719.16..1152.64 rows=1 width=655)
   Join Filter: ("outer".zatrudn_id = "inner".id)
   ->  Index Scan using harmon_pkey on harmon  (cost=0.00..5.76 rows=1
width=655)
         Index Cond: (id = 16105)
         Filter: ((1 / (id - 16105)) = 0)
   ->  Hash IN Join  (cost=719.16..1143.53 rows=268 width=4)
         Hash Cond: ("outer".stanowis_id = "inner".id)
         ->  Seq Scan on zatrudnienie  (cost=0.00..400.68 rows=2802 width=8)
               Filter:((s_grupprac_id = 1) OR (s_grupprac_id = 2))

         ->  Hash  (cost=714.44..714.44 rows=1885 width=4)
               ->  Hash IN Join  (cost=38.30..714.44 rows=1885 width=4)
                     Hash Cond: ("outer".jednostka_id = "inner".id)
                     ->  Seq Scan on stanowisko  (cost=0.00..558.86 rows=19686
width=8)
                     ->  Hash  (cost=38.02..38.02 rows=115 width=4)
                           ->  Seq Scan on jednostka  (cost=0.00..38.02 rows=115
width=4)
                               Filter: (((kod)::text ~~ '__________'::text) AND
czyaktualne AND ((kod)::text ~~ '__________'::text))
```

It is worth to note that in the query plan for query (2) index condition is taken before view filters are applied. In the case where our security rules work well (5), the query plan shows that view rules are checked before user conditions.
Query plan for (5)

```
Nested Loop  (cost=1144.20..3086.41 rows=86 width=655)
   ->  HashAggregate  (cost=1144.20..1146.88 rows=268 width=4)
         ->  Hash IN Join  (cost=719.16..1143.53 rows=268 width=4)
               Hash Cond: ("outer".stanowis_id = "inner".id)
               -> Seq Scan on zatrudnienie  (cost=0.00..400.68 rows=2802 wid-
th=8)
                  Filter: (((s_grupprac_id = 1) OR (s_grupprac_id = 2))
               ->  Hash  (cost=714.44..714.44 rows=1885 width=4)
                     ->  Hash IN Join  (cost=38.30..714.44 rows=1885 width=4)
                           Hash Cond: ("outer".jednostka_id = "inner".id)
                           ->  Seq Scan on stanowisko  (cost=0.00..558.86
rows=19686 width=8)
                           ->  Hash  (cost=38.02..38.02 rows=115 width=4)
                                 ->  Seq Scan on jednostka  (cost=0.00..38.02
rows=115 width=4)
                                       Filter: (((kod)::text ~~
'__________'::text) AND czyaktualne AND ((kod)::text ~~ '__________'::text))
   ->  Index Scan using harmon_zatrudn_id_idx on harmon  (cost=0.00..7.22 rows=1
width=655)
         Index Cond: (harmon.zatrudn_id = "outer".id)
         Filter: ((id > 16103) AND ((1 / (id - 16105)) = 0))
```

From query plans shown above we can conclude that if user conditions are very narrow, they can be applied before view filters.

3.2. Obtaining hidden data using special tricks

As shown previously, using some specially created query we can obtain information that we do not have access to. The question arises how we can extract forbidden data. There are two ways:

1. User defined stored functions.
2. Build-in functions.

3.2.1. User defined stored functions

With using specially prepared functions user can dump all data from forbidden rows.

What is more important even if SQL language allows forbidding a user to call a function using REVOKE command, PostgreSQL does not implement this. If a user has GRANT to schema where function is stored he has right to use it. On the other hand most users are not allowed to create stored functions.

3.2.2. Build-in functions

Second approach is to use build-in functions. We can use for example logarithm function and obtain error message `ERROR: cannot take logarithm of a negative number.`

The most dangerous function we can use is CAST operation, because it gives back very verbose error message. For example the query:

```
SELECT * FROM harmon WHERE
   (id=16105 AND cast( id||'_'||rok||'_'||d2 as integer)<0);
```

yields error message:

```
ERROR:  invalid input syntax for integer: "16105_2004_8"
```

Hence we know that:

Table 2

Obtained forbidden data

id	Rok	d2
16105	2004	8

No matter how we restrict creation of stored functions for user, if he has possibility to send SQL queries to database, he can obtain forbidden data. Even if we could revoke permission to use a function for a user we can't do it in case of cast function. Cast function must be allowed for everyone because it is used often implicitly, even without programmer knowledge.

4. Conclusions

It is well known that views used as security mechanisms can leak information due to side effects of WHERE clauses in the user queries when those clauses are pushed by the optimizer below the view filter in the query plan. We have analyzed the problem in the context of a view based security framework [1] of a working University of Łódź human resources database implemented in PostgreSQL database management system. Since our human resource database is a good representative example of a large business class database system containing sensitive information, we believe that some of the conclusions presented below can be relevant for implementers and users of other database applications, in particular those which are build on PostgreSQL.

The overall message is that while the standard leak mechanisms do work, the potential burglar needs an authorized access to a fairly large number of rows in a table he is interested in before he can use those mechanisms to leak information from the remaining (secret) rows:

1. The information cannot leak from the tables where the user is not authorized to see any rows – the views for such tables will not be generated in the user schema at all.
2. In order to force the rearranging of clauses into an unsafe query plan the user query against a view has to be much more selective and simple then the query defining the view. It follows that the view filter cannot be too selective in the first place.

For instance, we were not able to create the leak using the authorizations allowing only access to records related only to a single person. Note that this is far from obvious. While one cannot beat the selectivity of a WHERE clause like `pers_id=1000` (where there is an index on `pers_id` column), the actual filters of some of the views generated in this case were much more complex than that and involved complicated sub-queries and joins. In particular, it seems that un-trusted users (the ones which have some access only to their own data) are unable to acquire secret information from our system.

More generally, it is possible to leak only information of a similar kind to the one the user has already some access to. Only users which have authorized access to information about salaries of some subset of the university staff can acquire illegitimate access to some of the remaining salaries.

However even such limited leaks can be damaging in case of sensitive personal, medical or financial data, and much care should be taken to prevent it.

Our university database is not directly accessible to the users, but only through the web based applications residing on a trusted application server. Unfortunately this prevents the attacks of the kind described in this paper only if the application does not allow sending arbitrary user queries. It follows that using a database server side fine grained access control relieves the application writer from implementing the details of a security policy (which was

the purpose of our security framework in the first place), but the application must still be proofed against SQL injection.

It would be advantageous to close the hidden communication channels in the database itself, but unless one is willing to modify the optimizer, perhaps along the lines of [2], it is impossible without incurring severe efficiency penalty as described in the Section 2. However in case of tables storing the most sensitive data, like credit card numbers or medical information, one could argue that security is more important than efficiency. In this case it might be reasonable to use table functions in the definition of security views, which prevents the optimizer from creating unsafe plans even if it causes queries to execute an order of magnitude slower.

Usually common users are not able to create functions anyway, but in the systems where security is based on views (or more generally query rewriting) a special care should be taken to ensure that this is indeed the case, as well as to severely limit the access to any existing functions which log somehow their arguments.

Preventing returning of the error messages (together with what is described in the paragraph above) would make it impossible to use techniques described in the previous section. Unfortunately:

1. It is impractical as it would interfere with transactions, making it impossible for an application to know when a transaction should be aborted.
2. There are more exotic hidden communication channels which do not require functions or error messages – like specially engineered queries, where the execution time depends on the value of some secret row.

On the other hand the PostgreSQL error messages are much more revealing than necessary in the production environment and there is no possibility of reducing their verbosity. The most egregious example are conversion error messages which allow direct leakage of values.

BIBLIOGRAPHY

1. Miodek K., Pychowski J.: Elastyczny system uprawnień użytkowników w systemie zarządzania bazą danych PostgreSQL. [in:] Bazy Danych – Modele, Technologie, Narzędzia. WKŁ, Warszawa 2006, p. 309-314.
2. Kabra G., Ramamurthy R., Sudarshan S.: Redundancy and Information Leakage in Fine-Grained Access Contol. SIGMOD, 2006.
3. Stonebraker M., Wong E.: Access control in relational database management system by query modification. Procs of the ACM Annual Conference, 1974, p. 180-186.
4. Rivizi S., Mendelzon A., Sudarshan S., Roy P.: Extending query rewriting techniques for fine-grained access control. SIGMOD, 2004.
5. The Virtual Private Database in Oracle9ir2. An Oracle Technical White Paper.

Recenzenci: Dr hab. inż. Andrzej Chydziński, prof. Pol. Śląskiej
Dr inż. Paweł Kasprowski

Wpłynęło do Redakcji 14 stycznia 2011 r.

Omówienie

W zastosowaniach biznesowych bardzo często, jako najważniejsze wskaźniki jakości, rozwiązania, wskazuje się szybkość działania oraz niezawodność. W trakcie tworzenia takich rozwiązań pojawia się dylemat: wydajność serwera czy też jego bezpieczeństwo? Przed podobnym dylematem stanęli twórcy jednej z największych baz danych użytkowanych na Uniwersytecie Łódzkim, gdyż aplikacje ją używające musiały posiadać bardzo ograniczony dostęp do danych, a ponieważ z aplikacji tych korzysta wiele osób jednocześnie, to istnieje problem przeciążenia bazy danych. Reguły bezpieczeństwa zostały oparte na widokach tworzonych dla każdego użytkownika, co daje dużą skalowalność i elastyczność rozwiązania. Zastosowana metoda pozwala na ograniczenie dostępu do poszczególnych wierszy w tabelach nie zmniejszając jednocześnie wydajności serwera baz danych. Niestety, takie rozwiązanie posiada pewne niedostatki związane z bezpieczeństwem. Optymalizatory wykorzystane w silnikach baz danych często zmieniają kolejność wykonywanych operacji. Powoduje to zwykle przyspieszenie działania, jednocześnie dając możliwość spreparowania zapytania w taki sposób, aby uzyskać dostęp do niedozwolonych wierszy. W artykule pokazano, iż wykorzystując informacje o błędach funkcji cast, można uzyskać dostęp do dowolnego wiersza. Należy jednak zauważyć, że dostęp do danych może uzyskać jedynie osoba mająca uprawnienia do samodzielnego tworzenia zapytań SQL, więc zagrożenia tego typu można wyeliminować w innych warstwach wykorzystywanych aplikacji.

Addresses

Krzysztof MIODEK: Uniwersytet Łódzki, Centrum Informatyki, ul. Lindleya 3, 90-131 Łódź, Polska, krzysztof.miodek@uni.lodz.pl.
Krzysztof PODLASKI: Uniwersytet Łódzki, Wydział Fizyki i Informatyki Stosowanej, ul. Pomorska 149/153, 90-236 Łódź, Polska, podlaski@uni.lodz.pl.
Ścibór SOBIESKI: Uniwersytet Łódzki, Wydział Fizyki i Informatyki Stosowanej, ul. Pomorska 149/153, 90-236 Łódź, Polska, scibor.sobieski@uni.lodz.pl.
Bartosz ZIELIŃSKI: Uniwersytet Łódzki, Wydział Fizyki i Informatyki Stosowanej, ul. Pomorska 149/153, 90-236 Łódź, Polska, bzielinski@uni.lodz.pl.