Explainable Security for Relational Databases (Extended Experimental Evaluation)

Gabriel Bender, Łucja Kot, Johannes Gehrke

February 24, 2014

This appendix contains an extended version of the experimental evaluation from Section 5 of the paper *Explainable Security for Relational Databases*, which will appear in SIGMOD 2014. In order to keep this document self-contained, we have duplicated material from Section 5 of the paper in addition to including further experiments.

The goal of our experimental evaluation was three-fold. First, we wanted to verify that SQL queries containing a wide range of commonly used features could be correctly handled by the compiler discussed in Section 4. Second, we wanted to determine whether the language of filter-project queries described in Section 4.2 was powerful enough to represent a variety of practical security constraints. And third, we wanted to determine whether policy formulas could be generated quickly enough to be used in practical systems.

We implemented a prototype system in Java. The prototype's architecture is depicted in Figure 1. Our system consisted of three main components: (i) a compiler for translating SQL queries into filter-project queries, (ii) a module for representing and reasoning about filter-project queries, and (iii) a module for representing and evaluating policy formulas and generating why-so and why-not explanations. The implementation of our compiler consisted of just over 5,500 lines of Java code, while filter-project queries were implemented in about 1,000 lines, and policy formulas were implemented in 250.

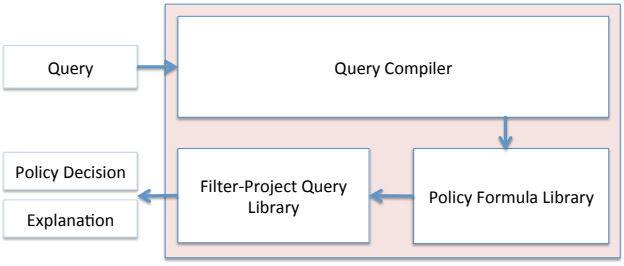

Figure 1: Architecture of Prototype System

For the first goal, we began by writing end-to-end tests for our compiler based on several dozen SQL queries taken from a standard undergraduate database textbook. (See Chapter 5 of [2].) We gradually added more tests over time to keep track of the border cases we encountered. At the time of publication, we had 110 of our 115 end-to-end tests running successfully, and the remaining five were rejected as being semantically invalid. Of these, three were rejected due to limitations

of JSqlParser. The last two were valid according to the SQL standard, but were considered to be invalid by both MySQL and PostgreSQL. The results gave us confidence that our compilation-based approach to disclosure control could be successfully applied to a broad range of real-world SQL features.

For the second and third goals, we ran experiments on a database inspired by the one used by the Facebook Query Language (FQL). [1] The schema has a rich and diverse set of security requirements. For example, albums containing media such as photos and videos can be made visible to friends of the current user, to friends of friends, or to the public at large. The membership of a group can be made world-visible, can be restricted to members of that group, or can be completely hidden. Lists of attendees for events can similarly be made world-visible or can be restricted to other attendees. All of these contingencies were accounted for using filter-project security views.

Our test database schema contained 18 relations with a grand total of 269 distinct columns – or an average of 15 columns per relation. We deliberately selected relations that had nontrivial security policies. The number of columns per relation ranged from two for the `Friend` relation (which stores information about friendship relations between users) to 62 for the `Application` table (which stores information about third-party Facebook Apps). We ended up defining a total of 75 security views. Equality constraints appeared in all but one of our security views; the exception was a view over the `Application` table that listed certain types of publicly available information about Facebook Apps. Filter atoms appeared in 59 security views. Column projections were used in just four of the security views.

A useful but unexpected consequence of the restrictions we impose on our handling of *filter-project* queries was that it is possible to define a form of inheritance for cases where the visibility of rows in one table depends on the visibility of rows in a different table. For instance, access to a `photo` is determined by the `album` which contains that photo. Any given album can be made world-visible or else its visibility can be restricted to the owner's friends or to friends of friends. We defined multiple security views over the `album` relation to handle the latter contingencies. However, we only needed to define one security view over the `photo` relation:

```
SELECT * FROM photo
 WHERE aid IN (SELECT aid FROM album);
```

In effect, the security view above forces any principal who queries the `photo` relation to perform a join on `aid` (Album ID) with the corresponding row of the `album` relation. Since our system performs a separate policy check for every table instance of the input query, this means that a principal can access information about a `photo` only if he or she is granted access to a view containing information about the corresponding `album`. In this way, the visibility of `photo`s is inherited from the visibility of the `album`s to which they belonged.

In addition to the qualitative experiments above, we also evaluated the performance of our prototype system. Our experiments were run on a workstation with two 4-core 2.13 GHz Intel Xeon processors and 50 GB of RAM. Our code was run on the OpenJDK 1.7 VM on Ubuntu 12.04.

Our experiments made use of a synthetic query generator that worked by recursively stringing together predefined templates. Our use of templates allowed us to generate realistic-looking queries whose joins and equality predicates encoded meaningful security-related constraints. Two sample queries generated by our system (chosen completely at random) are shown below:

```
SELECT R1.owner, R1.video_count FROM album AS R1
 WHERE R1.visible = 'friends-of-friends' AND EXISTS
```

```
 (SELECT DISTINCT 1 FROM friend AS R2, friend AS R3
   WHERE R2.uid1 = 4 AND R2.uid2 = R3.uid1
     AND R3.uid2 = 4);

SELECT R1.size, R1.src, R1.width
  FROM photo_src AS R1 WHERE EXISTS
  (SELECT DISTINCT 1 FROM photo AS R2, album AS R3
    WHERE R1.photo_id = R2.pid
      AND R2.aid = R3.aid
      AND R3.visible = 'everyone');
```

Each of the queries we generated contained between one and six table instances; the median was 2 and the mean was 2.5.

We measured the wall time needed to (i) randomly generate queries, (ii) lex and parse them, (iii) extract filter-project queries from the parsed output, (iv) compute policy formulas from the resulting filter-project queries, and (v) generate a policy decision for each query, a why-so explanation for each permitted query, and a why-not explanation for each denied query. Since each stage depended on the output of the previous stage, these measurements were cumulative: we measured the time for (i) alone, the time for (i) + (ii), and so on. We also varied the number of execution threads between 1 and 8. All of our algorithms are trivially parallelizable, and the threads operated completely independently of each other. In each case, we measured the time needed to process 1,000,000 queries. All numbers are averaged across five runs; the variation across runs was less than one second in almost all cases.

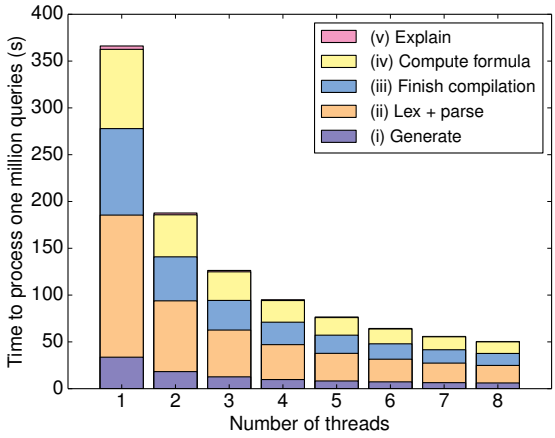

Figure 2: Performance for ordinary queries

The results are shown in Figure 2. As expected, throughput scaled nearly linearly with the number of cores. The analysis pipeline took a total of 366 seconds on a single core or 50 seconds on 8 cores. If we exclude the time needed to randomly generate queries, these numbers change to 333 and 44 seconds respectively. If we focus on the last three stages (which are where we claim algorithmic contributions), the numbers fall to 181 and 26 seconds respectively. All computation was CPU-bound, and memory consumption ranged from 60 MB to 220 MB per thread. The time

needed to perform policy checks and generate explanations was negligible compared to the rest of the system – about 3.5 seconds on a single core or 0.5 seconds on eight cores – and is barely visible in Figure 2.

Since query compilation time dominated the benchmark numbers in Figure 2, we speculated that throughput could be significantly improved by using prepared statements. The filter-project queries associated with prepared statements only needed to be calculated once (as a preprocessing step), and could be retrieved on demand when the prepared statements were executed. Parameters passed to prepared statements at execution time could be substituted directly into the compiled filter-project queries, bypassing the need for a heavy-weight compilation step.

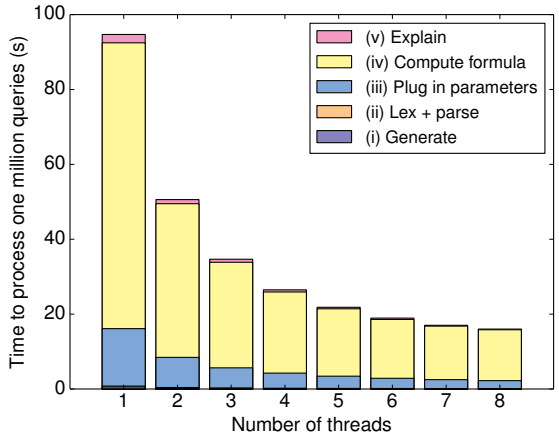

Figure 3: Performance for prepared statements

Results are shown in Figure 3. Our use of prepared statements yielded about a 3x improvement in the system's overall throughput. The time spent on query generation, lexing and parsing was negligible, and the time needed to retrieve filter-project queries and plug in parameter values paled in comparison with the time needed to perform end-to-end query compilation. As a result, computation of policy formulas dominated the running time.

The experiments above focus on the compiler's performance for semijoin queries. These types of queries effectively stress steps (iv) and (v) of our query analysis pipeline because they tend to yield large filter-project queries that have many filter atoms. Our final experiment was designed to get a better idea of the compiler's performance on real SQL queries. Our workload consisted of queries drawn from a list of 86 predefined strings, most of which were taken from Chapter 5 of [2]. The workload incorporated a diverse set of features, including aggregates, arithmetic comparisons, `GROUP BY` statements and aggregates, inner, outer, and semijoins, set operations, and temporary tables. The security constraints that naturally arose for our sample database were simpler than for our Facebook case study. Because the third experiment focused on the compilation phase of our pipeline, we defined only nine security views: three views for each of three base relations.

Results for the third experiment are shown in Figure 4. Query generation time was essentially zero because the queries were selected from a predefined list. The total time needed to compile 1,000,000 queries (which was the main focus of our experiment) was around 111 seconds on a single core, or around 15 seconds when distributed across eight cores. The number is significantly lower than in earlier experiments, where compilation time averaged 244 seconds on a single core

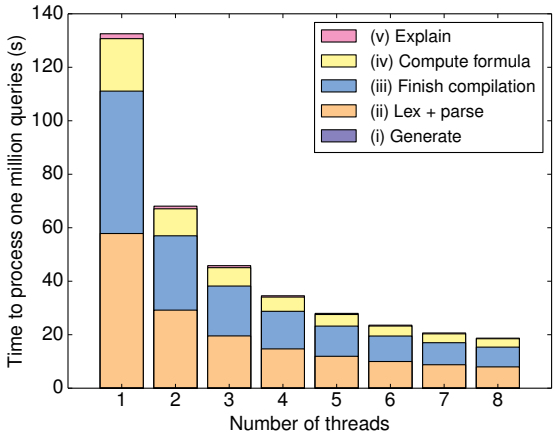

Figure 4: Performance for complex SQL features

or 31 seconds on eight cores. This is not surprising because queries in the synthetically generated workload from our previous experiment tended to yield relatively large ASTs and well-connected condition graphs.

References

[1] Facebook query language. `https://developers.facebook.com/docs/reference/fql/`.

[2] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.