◀ See all posts

Dynamic SQL in applications: how to handle dynamic WHERE clauses

by pkalliok | 29 Dec 2015
programming ● Clojure ● SQL ● security ● simplicity ● orthogonality ● Python

What is so difficult about using SQL in applications?

If you're like me, you might have thought that there is something smelly if not outright wrong in the way applications interact with SQL databases. The ubiquitous practice of sending practically executable code – SQL statements – to the server means that we will have to construct executable code programmatically in our application. And that means that we have to be very careful to prevent security problems in that generated code. The reason we have SQL injections in the first place is that there's no other way to interact with SQL databases than to construct code on the fly.

> ```
> SELECT phone, email
> FROM people
> WHERE name LIKE '%$namepart%'
> ```
>
> Whoopsie, what happens if $namepart contains a quote character?

Of course, every modern database interface provides some facilities to interpolate values from our application language to the generated SQL in a safe way. This handles the vast majority of cases where you need to take care that your dynamically generated SQL doesn't end up being something completely different you meant it to be. The SQL is generated from some kind of (textual) SQL templates, and the templating library – often integrated with the programming

language's database interface library – takes care that the generated SQL is always structurally sound.

```
SELECT users.username, people.email
FROM people, users
WHERE users.person = people.id
  AND people.name LIKE :namepattern
  AND users.last_login > :lastdate
```

This is what it usually looks like, with modern database interfaces.

The problem of dynamic WHERE clauses

But, there is a very common need for dynamic SQL that textual SQL templates do *not* cover very well, or at all. That is the case where we want to have a varying number of WHERE conditions in a database query depending on whether or not the listing is constrained in some specific way. For instance, the user might want to list all cities in some specific area; then, having seen there are too many to browse through, they want to restrict the search to only big cities with more than half a million people. The two queries are essentially the same, except that the latter adds a new AND condition to the WHERE clause of our generated SQL.

```
query = """SELECT * FROM cities
        WHERE (lng - :x) * (lng - :x) + (lat - :y) * (lat - :y) < 100"""
params = { "x": x, "y": y }
if minpopulation:
  query += " AND population > :minpop"
  params["minpop"] = minpopulation
```

One (bad) way to dynamically construct WHERE clauses (in Python).

There are a couple of solutions to this situation without inducing code duplication between the two queries. One of them (and sadly common) is to construct SQL *templates* by hand, adding more AND conditions when needed, and also updating the list of template parameters. This is error prone and wastes working time every time one needs to update the query construction logic.

Because code generation is error prone, some have solved this problem by making a more powerful templating language. HoneySQL, for instance, is a Clojure library that converts templates, expressed by native data structures, into executable SQL. This is a working solution, but sometimes it feels stupid to learn yet another database language – the data structure language used to express SQL. It might be

more portable across databases, but it also requires you to extend the template language if you want to use some database specific features.

```
(-> (select :*)
    (from :cities)
    (merge-where (if-not (nil? namepattern)
      [:like :name namepattern]))
    (merge-where (if-not (nil? minpopulation)
      [:> :population minpopulation]))
    sql/format)
```

Dynamic SQL construction in Clojure and HoneySQL.

Yet another approach to dynamic SQL generation is the ORM, or object-relational mapper. ORM is a technique which adorns native data structures with the ability to be database-backed. All changes to those objects' state will be synced into the database, and vice versa. ORMs are superb for data updates, but outright horrible for complex queries. They shift data query logic from the database side to the application side, which makes it harder to use the database for what it excels in, and in practice results in all kinds of performance problems. However, when and how to use ORMs is a very complicated question well worthy its own blog post or several.

Solving the dynamic WHERE clause problem in SQL

However, there is a simpler solution – so simple that it is easy to overlook. Usually, we can trust the SQL server to behave sensibly when we shift the condition logic to the SQL side. Every database I know optimises away conditions whose truth value can be proved (such as `3 < 5` or, more usefully, `NULL IS NULL`). In practice, this means that unwanted search parameters can be passed in a NULLs, and their value can be checked in the SQL so that they never affect the search when they are NULL.

```
SELECT * FROM cities
WHERE (lng - :x) * (lng - :x) + (lat - :y) * (lat - :y) < 100
  AND (:minpop IS NULL OR population > :minpop)
```

Handling the dynamic part on the SQL side.

As a sidenote, I recently found out about Yesql, which very well appeals to my æsthetic taste. Because SQL is a domain-specific language, I don't want to embed it in strings in another language; rather I would like to keep it in a separate file, so that I can tell my text editor to use SQL syntax highlighting for editing that file, and I

won't need to bother with the indentation of my host language (currently Clojure) when I write longish SQL excerpts. Yesql embeds query metadata in SQL comments. This is an actual example of a PostgreSQL query in my Yesql query file.

```
-- name: db-points-near
-- Return points in order of proximity to :point, along with their tag(s).
SELECT loc.id, loc.coord, tag.name, tag.ns
FROM (SELECT id, coord, modtime, mergedto
        FROM location
        WHERE mergedto IS NULL
        ORDER BY coord <-> (:point)::point
        LIMIT (:limit)::integer
        OFFSET (:page)::integer * :limit) AS loc, location_tag l, tag
WHERE loc.id = l.location
  AND l.tag = tag.id
  AND ((:mindate)::date IS NULL OR loc.modtime > :mindate)
  AND ((:maxdate)::date IS NULL OR loc.modtime < :maxdate)
  AND ((:maxdist)::float IS NULL OR (loc.coord <-> :point) < :maxdist)
  AND ((:tagpat)::text IS NULL OR tag.name LIKE :tagpat)
  AND ((:username)::text IS NULL OR tag.ns = :username)
ORDER BY loc.coord <-> :point;
```

Neat, right? Although I find it somewhat worrying that I have to type-annotate parameters in almost all contexts, having the full query logic in SQL is very pleasing and makes my application code more straightforward. After having the query defined thus, I only need to construct the parameter map to pass to the query.

Further reading

- Common Cases when (not) to Use Dynamic SQL: The need for dynamic SQL generation arises from various reasons, some good and some bad. This article, even if written for Microsoft SQL Server, has a lot of ideas and analysis that applies quite well to other SQL backends.
- The two top performance problems caused by ORM tools: Usually, you do not want to optimise before you can verify that performance problems exist (by profiling or similar techniques). However, ORM performance problems are especially difficult to tackle and sometimes require restructuring of your application. Here are also two articles about ORM optimisation for nHibernate (Java) and SQLAlchemy (Python).

JOCA • 4 years ago

why not use Prepared Statements? :)

in Clojure:

```
(def q "SELECT phone, email
FROM people
WHERE name LIKE? ")

(require '[clojure.java.jdbc :as sql])

(sql/query dbinfo [q "JOCA"])
```

that simple and more secure than your sample

regards

pkalliok ➚ JOCA • 4 years ago

I actually advocate prepared statements, it's just not all databases offer them. Note however, that it's easier to use prepared statements if you don't construct them on the fly in your application, so my point and yours are actually a good match.

Re "more secure than your sample": which sample are you referring to? If it's the first one, yes, that's meant to be an example of bad programmatic SQL, and the next (second) one is an example of how to fix that, and really similar to your example code.

However, AFAICT your example code doesn't actually prepare the statement q and so isn't actually an example of prepared statements. Your point, on the other hand, is a sound one: once you have your SQL template, preparing it for execution is usually a good practice.

integrated in Xcode) for Swift

Avatar Reminded me of this book:
http://shop.oreilly.com/pro...

Euroclojure 2016 Review

1 comment • 3 years ago

Continuous delivery for Product Owners and UX

/dev/solita

Writings from our fine developers.

Work and write with us ►

More about Solita

- RSS /dev/solita
- solita.fi
- Facebook
- LinkedIn
- Twitter
- Instagram
- YouTube

Tags

AI (2) AWS (16) Active Directory (1) Ansible (2) Api (2) Arduino (2) Automation (1) Azure (2) Azure Data Lake (1) Beercraft (1) Burp (1) C++ (1) CI (3) CMS (4) CircleCI (1) ClamAV (1) Clojure (15) ClojureScript (5) Cloud (2) CloudFormation (1) Code quality (1) Conference (2) Continuous inspection (1) DOTNET (14) Data factor (1) Data pipeline (1) Database (1) Datomic (1) Deep Learning (1) DevOps (11) DevSec (3) DevSecOps (2) Digitalization (1) Distributed (1) Docker (4) Elasticsearch (1) Electronics (1) Enabling platform (1) Environment (1) Episerver (15)

External table (1) Full stack (1) GDPR (1) GIS (2) Git (1) GitHub API (1) GraalVM (1) Haskell (1) Haveged (1) Hystrix (1) IntelliJ IDEA (1) IoT (2) JBehave (1) JMeter (1) JSON (1) Java (15) Java 11 (1) Java 9 (7) Java EE (5) JavaOne (5) Javascript (2) Jenkins (4) Jigsaw (1) Jira (1) Kibana (1) Kubernetes (2) LDAP (1) Lambda (2) Linux (1) Lisp (1) LogStash (1) MVC (1) Machine Learning (1) Microservices (1) Monitoring (1) Msbuild (1) NP (1) Neural Networks (1) Node.JS (1) OWASP (1) OpenCV (1) PRNG (1) Polybase (1) PostgreSQL (1) PowerShell (7) Privacy (1) Project (1) Prolog (1) Python (3) REPL (1) REST (1) Raspberry Pi (1) React (1) Rust (1) S3 (2) SQL (3) SQL Data Warehouse (1) SQL Korma (1) Schemaspy (1) Security (2) Security Pipeline (1) Selenium (1) Servant (1) Serverless (3) Slush (1) Software development (1) Solita (6) SonarQube (1) Spring (1) Spring Boot (1) Squirrel (1) TUT (2) TensorFlow (1) Testing (1) Travis CI (1) Turing (1) TypeScript (1) Vert.x (1) Virus (1) Vulnerabilities detection (1) WebSocket (1) Wicket (1) Wildfly Swarm (1) XML (1) ZAP (1) agile retrospectives (1) ai (1) algorithm (1) apps (1) appstore (1) architecture (4) artificial intelligence (1) association analysis (1) automated documentation (1) automation (3) autonomy (1) brevity (1) bugs (1) business (1) business logic (1) clojure (4) clojurescript (3) cloud (3) comic (1) command line (1) complexity (1) composability (2) computer science (2) conference (1) continuous delivery (8) continuous deployment (2) continuous integration (3) cross language programming (1) culture (9) data science (3) database (1) deployment pipeline (3) developer culture (1) developers (1) development (3) devsecops (3) disobey (3) docker (1) documentation (1) documentation pipeline (1) domain (1) domain modeling (1) e2e testing (1) encoding (1) entropy (1) episerver composer (1) exploit (1) feedback (1) frontend (2) functional programming (4) graph (2) hackathon (1) hacker (2) hacking (5) infosec (4) integration (1) iterative development (1) language (1) legacy code (1) lens (1) machine learning (2) maintainability (3) maven (1) meetup (3) methodologies (1) microservices (4) microsoft (1) mobile (1) modularity (1) multi language programming (1)

multithreading (1) mumps (1) network programming (1) neural network (1)
normal day (1) normal form (1) normipäivä (1) open development (1) open source (5)
orthogonality (1) pagetypebuilder (1) parenthood (1) performance (2)
poly paradigm programming (1) polyglot programming (1) postgres (1) powerapps (1)
predictive analytics (1) procurement (1) product owner (1) productisation (1)
productivity (1) professional development (1) programming (12) prototyping (3)
quality (1) random number (1) randomness (1) re frame (1) re:Inforce (1)
re:Invent (6) reagent (1) real time delivery (1) real time deployment (1)
recruitment (2) rngd (1) security (6) simplicity (3) slack (2)
software architecture (1) software design (2) software development process (3)
software processes (1) software security (5) software testing (1) specql (1)
startup (1) state (1) stylefy (1) supervised learning (1) support (1) tendering (1)
test data (1) testing (7) training (1) traversal (1) tuck (1) user experience (1)
virus scanner (1) visualization (1) web workers (1) windows containers (1)
workshop (1)