# On Being Prepared

Prepared Statements and all that.

Mark Kirkwood
Sep 20, 2009

# Background

- Web apps often do a lot of (often similar) queries to render a page e.g:
  - Drupal upto 5000 (!)
  - Moodle several 100s
  - audience suggest other perl based ones?
- This is expensive, even for database managers that are in theory optimized for this use case

# Expensive?

- Why expensive? Requires:
  - Db server parse query string
  - Produce execution plan
  - Start executor machinery
  - Run query and emit result rows
  - Shutdown executor machinery

# What are the costs?

- Rough figures for 10000 executions:
- (higher than a typical bad web page, but makes measurement easy...)
- First try to measure basic executor startup, trivial planning and emitting 1 row, shutdown...

# What are the costs?

```
my $sql = "SELECT 1";
my $maxi = 10000;

for ($i = 0; $i < $maxi; $i++) {
        $row = $dbh->selectrow_arrayref($sql);
}
```

# What are the costs?

- Approx 1 s for 10000 executions
  - Postgres 1.3 s (8.3.8 on Ubuntu 9.04 i386)
  - Mysql 1.0 s (5.4.1 on Ubuntu 9.04 i386)

# What are the costs?

- Now try measure planning + row extraction by accessing a real table

# What are the costs?

```
my $sql = "SELECT * FROM accounts WHERE aid = ";
my $maxi = 10000;
my $maxaid = 5000000;

for ($i = 0; $i < $maxi; $i++) {
        $aid = int(rand($maxaid));
        $row = $dbh->selectrow_arrayref($sql . $aid);
}
```

# What are the costs?

- Approx 2-5 s for 10000 executions
  - Postgres 2.6 s
  - Mysql (innodb) 2.4 s
  - Mysql (myisam) 1.8 s
- This approach is clearly not optimal, no matter what db engine is in use
- Is there another way?

# Reuse of similar statements

- The statements are often very similar – i.e only the values are different
- We can use the prepare API to reuse the statement(s) with different bind variables
- In theory saving the parsing and planning steps.

# Reusing statements

```
my $sql = "SELECT * FROM accounts WHERE aid = ?";
my $maxi = 10000;
my $maxaid = 5000000;

$sth = $dbh->prepare($sql);

for ($i = 0; $i < $maxi; $i++) {
        $aid = int(rand($maxaid));
        $sth->execute($aid);
        $row = $sth->fetchrow_arrayref();
}
```

# Reusing statements

- Approx 1-2 s for 10000 executions
  - Postgres 1.2 s
  - Mysql (innodb) 2.0 s
  - Mysql (myisam) 1.1 s
- Is clearly better in general
- Database engines that are “heavier” in plan and setup stages are helped more

# Some analysis

- Performing fewer queries would help more than anything...
- Some sort of partial cache for commonly queries rows:
  - Query cache (Mysql)
  - Pg Memcache (Postgres)
  - Simple hash stored in shared variable?

# Drawbacks of statement reuse

- Db engine must support SQL operations PREPARE, EXECUTE
  - Mysql 5.0.x or later
  - Postgres 7.4 or later
  - Others... need to check

# Drawbacks of statement reuse

- Plan for a query statement with unknown parameters may not be as optimal as the equivalent one without them
  - Optimal plan may depend on the value
  - Not so important for lookup via primary key
  - Vital for range (<) operations or non uniform data distributions

# Drawbacks of statement reuse

```
bench=# EXPLAIN SELECT * FROM accounts WHERE aid < 3000000;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using accounts_pkey on accounts
   Index Cond: (aid < 3000000)

bench=# EXPLAIN SELECT * FROM accounts WHERE aid < 3500000;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on accounts
   Filter: (aid < 3500000)
```

# Drawbacks of statement reuse

```
bench=# PREPARE s0 AS SELECT * FROM accounts WHERE aid < $1;
PREPARE

bench=# EXPLAIN EXECUTE s0(3000000);
                                QUERY PLAN
---------------------------------------------------------------------------------------
 Index Scan using accounts_pkey on accounts
   Index Cond: (aid < $1)

Time: 0.264 ms
bench=# EXPLAIN EXECUTE s0(3500000);
                                QUERY PLAN
---------------------------------------------------------------------------------------
 Index Scan using accounts_pkey on accounts
   Index Cond: (aid < $1)
```

# Drawbacks of statement reuse

- The complete PREPARE + EXECUTE combination is more expensive than a single simple query operation
- Show cost of this...

# Drawbacks of statement reuse

```
my $sql = "SELECT * FROM accounts WHERE aid = ?";
my $maxi = 10000;
my $maxaid = 5000000;

for ($i = 0; $i < $maxi; $i++) {
        $aid = int(rand($maxaid));

        $sth = $dbh->prepare($sql);
        $sth->execute($aid);
        $row = $sth->fetchrow_arrayref();
        $sth->finish;
}
```

# Drawbacks of statement reuse

- Approx 2-5 s for 10000 executions
  - Postgres 5.2 s
  - Mysql (innodb)  2.5 s
  - Mysql (myisam) 1.9 s
- Particularly bad for Postgres... why?
- Look at server log

preparation

# Drawbacks of statement reuse

## exec.pl:

```
LOG:  duration: 0.292 ms  statement: SELECT * FROM accounts WHERE aid = 3729
```

## execprepared.pl:

```
LOG:  duration: 0.162 ms  parse dbdpg_p19774_2: SELECT * FROM accounts WHERE aid = $1
LOG:  duration: 0.037 ms  bind dbdpg_p19774_2: SELECT * FROM accounts WHERE aid = $1
DETAIL:  parameters: $1 = '3729'
LOG:  duration: 0.059 ms  execute dbdpg_p19774_2: SELECT * FROM accounts WHERE aid =
$1
DETAIL:  parameters: $1 = '3729'
LOG:  duration: 0.040 ms  statement: DEALLOCATE dbdpg_p19774_2
```

# Drawbacks of statement reuse

- 4 separate steps
  - Extra overhead manage many named statements
  - Potentially more expensive in network latency

# Some Analysis

- Using prepared statements can improve performance markedly
  - Need separate logic so do few PREPAREs
  - ..and many EXECUTEs
- If you cannot do this, would you want to use the more expensive PREPARE + EXECUTE anyway? (continued...)

# Some Analysis

- Security...
- Simple method vulnerable to injection

- Consider a contrived example

# Injection

```
my $sql = "SELECT * FROM accounts WHERE aid = ";

for ($i = 0; $i < $maxi; $i++) {
        $aid = int(rand($maxaid));

        $row = $dbh->selectrow_arrayref($sql . $aid .
                "; CREATE TABLE secure" . $i .
                "(id INTEGER); " );
}
```

# Injection

- Creates 10000 tables
- Consider an equivalent prepared example

# Injection

```
my $sql = "SELECT * FROM accounts WHERE aid = ?";

for ($i = 0; $i < $maxi; $i++) {
        $aid = int(rand($maxaid));
        $sth = $dbh->prepare($sql);
        $sth->execute($aid .
                "; CREATE TABLE secure" . $i .
                "(id INTEGER); ");
        $row = $sth->fetchrow_arrayref();
        $sth->finish;
}
```

# Injection

- No tables created:
  - All statements fail, with invalid integer errors.
- Analogous examples with string variables fail too:
  - Strings are safely quoted
  - Prepare interface disallows multiple statements in 1 string

# Some Analysis

- Can we get the protection of PREPARE but the performance of simple statements?
  - Use programmatic prepare methods, but without server side PREPAREd statements.

# Injection

```
$dbh = DBI->connect($dsn, $user, "",
        {AutoCommit => 0,
         pg_server_prepare => 0})
```

# Injection

- The pg_server_prepare enables switching on/ off server side prepare, but lets you use the Prepare API for safety.
- pg_server_prepare can be applied to connection or individual statement objects
- Analgous parameters exist for other db engines.

# Final Analysis

- For performance:
  - 1 prepare, many executes
  - Best for lots of simple statements
- For safety:
  - 1 prepare + 1 execute
  - May need to set to pg_server_prepare (or similar) to 0 for performance in this case