SQL Injection Myths and Fallacies

Bill Karwin, Percona Inc.

Me

Software developer

C, Java, Perl, PHP, Ruby

SQL maven

MySQL Consultant at Percona

Author of *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*

What is SQL Injection?

```
SELECT * FROM Bugs
  WHERE bug_id = $_GET['bugid']
```

user input

What is SQL Injection?

```
SELECT * FROM Bugs
  WHERE bug_id = 1234 OR TRUE
```

unintended logic

Worse SQL Injection

```
UPDATE Accounts
  SET password = SHA2('$password')
  WHERE account_id = $account_id
```

Worse SQL Injection

```
UPDATE Accounts
  SET password = SHA2('xyzzy'), admin=('1')
  WHERE account_id = 1234 OR TRUE
```

changes account to administrator

changes password for all accounts

Myths and Fallacies

MYTH Based on a grain of truth, but derives a wrong conclusion

FALLACY Based on a false assumption, but derives a logical conclusion

MYTH

"SQL Injection is an old problem—so I don't have to worry about it."

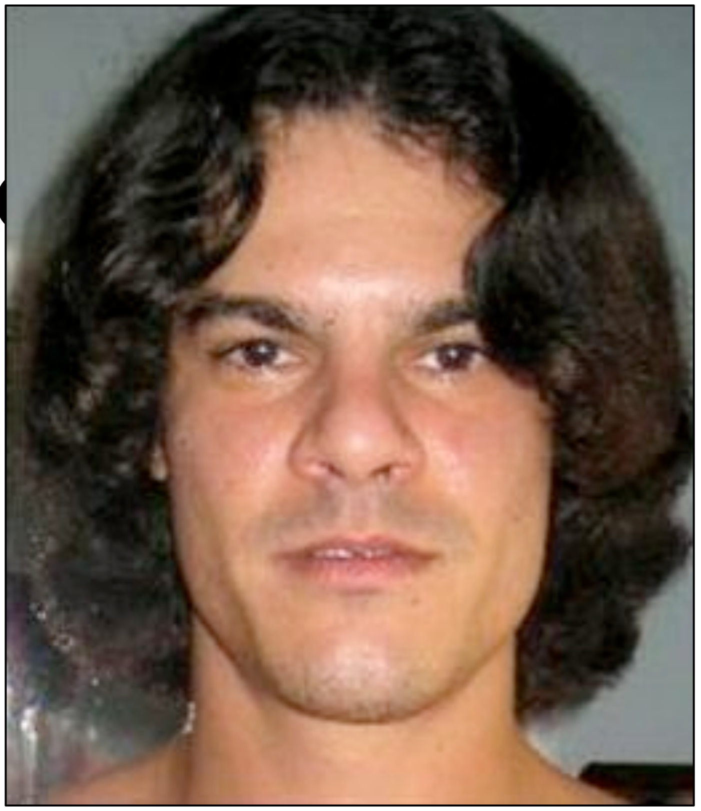
Identity Theft

130 million credit card numbers

Albert Gonzalez used SQL Injection to install his packet-sniff code onto credit-card servers

Sentenced 20 years in March 2010

Cost to victim company Heartland Payment Systems: $12.6 million



http://www.miamiherald.com/2009/08/22/1198469/from-snitch-to-cyberthief-of-the.html

http://www.cio.com/article/492039/Security_Breach_Cost_Heartland_12.6_Million_So_Far

Other Recent Cases

(April 2011) Sun.com and MySQL.com attacked by blind SQL Injection attack, revealing portions of the site's databases, including usernames and passwords.

http://techie-buzz.com/tech-news/mysql-com-database-compromised-sql-injection.html

http://seclists.org/fulldisclosure/2011/Mar/309

http://tinkode27.baywords.com/

(April 2011) LizaMoon scareware campaign infected hundreds of thousands of websites via SQL Injection.

http://www.informationweek.com/news/security/attacks/showArticle.jhtml?articleID=229400764

Experts Agree

2009 Data Breach Investigations Report, Verizon Business RISK Team

> “When hackers are required to work to gain access, SQL injection appears to be the uncontested technique of choice. In 2008, this type of attack ranked second in prevalence (utilized in 16 breaches) and first in the amount of records compromised (79 percent of the aggregate 285 million).”

http://www.verizonbusiness.com/resources/security/reports/2009_databreach_rp.pdf

MYTH

"Escaping input prevents SQL injection."

Escaping & Filtering

backslash escapes special characters

```
UPDATE Accounts
   SET password = SHA2('xyzzy\'), admin=(\'1')
   WHERE account_id = 1234
```

coerced to integer

Escaping & Filtering Functions

```
<?php
$password = $_POST["password"];
  $password_escaped =
        mysql_real_escape_string($password);
$id = (int) $_POST["account"];
$sql = "UPDATE Accounts
        SET password = SHA2(‘{$password_escaped}’)
        WHERE account_id = {$id}";
mysql_query($sql);
```

Escaping & Filtering Functions

```
<?php
$password = $_POST["password"];
  $password_quoted = $pdo->quote($password);
$id = filter_input(INPUT_POST, "account",
        FILTER_SANITIZE_NUMBER_INT);
$sql = "UPDATE Accounts
        SET password = SHA2( {$password_quoted} )
        WHERE account_id = {$id}";
$pdo->query($sql);
```

Identifiers and Keywords

```
<?php
$column = $_GET["order"];
  $column_delimited = $pdo->FUNCTION?($column);
$direction = $_GET["dir"];
$sql = "SELECT * FROM Bugs
        ORDER BY {$column_delimited} {$direction}";
$pdo->query($sql);
```

no API to support delimited identifiers

keywords get no quoting

MYTH

“If some escaping is good, more must be better.”

Overkill?

real function from a user's project

```
<?php
function sanitize($string){
    $string = strip_tags($string);
    $string = htmlspecialchars($string);
    $string = trim(rtrim(ltrim($string)));
    $string = mysql_real_escape_string($string);
    return $string;
  }
$password = sanitize( $_POST["password"] );
mysql_query("UPDATE Users
   SET password = '$password'
   WHERE user_id = $user_id");
```

“FIRE EVERYTHING!!”

Just the One Will Do

```
<?php

$password = mysql_real_escape_string(
  $_POST["password"] );

mysql_query("UPDATE Users
   SET password = '$password'
   WHERE user_id = $user_id");
```

MYTH

"I can write my own escaping function."

Please Don’t

addslashes() isn’t good enough in a multibyte world

Example:

- http://example.org/login.php?account=%bf%27 OR 1=1 --
- $account = addslashes($_REQUEST(“account”));
- Function sees a single-quote (%27) and inserts backslash (%5c). Result:

  $\boxed{\text{\%bf\%5c}}\boxed{\text{\%27}}$ OR 1=1 --

single-quote

valid multi-byte character in GBK: 縗

Grant Access to Any Account

Interpolating:

```
SELECT * FROM Accounts WHERE
   account = '{$account}' AND password =
   '{$password}'
```

Results in:

```
SELECT * FROM Accounts WHERE
   account = '縗' OR 1=1 -- ' AND password = 'guess'
```

http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string

http://bugs.mysql.com/bug.php?id=8378

Solutions

Use driver-provided escaping functions:

- mysql_real_escape_string()
- mysqli::real_escape_string()
- PDO::quote()

Use API functions to set the client character set:

- mysql_set_charset()
- mysqli::set_charset()

http://ilia.ws/archives/103-mysql_real_escape_string-versus-Prepared-Statements.html

Use UTF-8 instead of GBK, SJIS, etc.

Use SQL query parameters (more on this later)

MYTH

“Unsafe data comes from users—if it’s already in the database, then it’s safe.”

Not Necessarily

```
$sql = "SELECT product_name FROM Products";
  $prodname = $pdo->query($sql)->fetchColumn();


$sql = "SELECT * FROM Bugs
        WHERE MATCH(summary, description)
        AGAINST ('{$prodname}')";
```

not safe input

FALLACY

"Using stored procedures prevents SQL Injection."

Static SQL in Procedures

filtering by data type is a good thing

```
CREATE PROCEDURE FindBugById (IN bugid INT)
  BEGIN
    SELECT * FROM Bugs WHERE bug_id = bugid;
  END


CALL FindByBugId(1234)
```

Dynamic SQL in Procedures

```
CREATE PROCEDURE BugsOrderBy
    (IN column_name VARCHAR(100),
     IN direction VARCHAR(4))
  BEGIN
    SET @query = CONCAT(
      'SELECT * FROM Bugs ORDER BY ',
      column_name, ' ', direction);
    PREPARE stmt FROM @query;
    EXECUTE stmt;
  END

CALL BugsOrderBy('date_reported', 'DESC')
```

interpolating arbitrary strings = SQL injection

Worthy of *TheDailyWTF*

```
CREATE PROCEDURE QueryAnyTable
    (IN table_name VARCHAR(100))
  BEGIN
    SET @query = CONCAT(
      'SELECT * FROM ', table_name);
    PREPARE stmt FROM @query;
    EXECUTE stmt;
  END

CALL QueryAnyTable( '(SELECT * FROM ...)' )
```

http://thedailywtf.com/Articles/For-the-Ease-of-Maintenance.aspx

MYTH

"Conservative SQL privileges limit the damage."

Denial of Service

```
SELECT * FROM Bugs JOIN Bugs
  JOIN Bugs JOIN Bugs JOIN Bugs
  JOIN Bugs
```

100 bugs = 1 trillion rows

Denial of Service

```
SELECT * FROM Bugs JOIN Bugs
  JOIN Bugs JOIN Bugs JOIN Bugs
  JOIN Bugs
  ORDER BY 1
```

still requires only SELECT privilege

Just Asking for It

http://www.example.com/show.php?query=SELECT%20*%20FROM%20Bugs

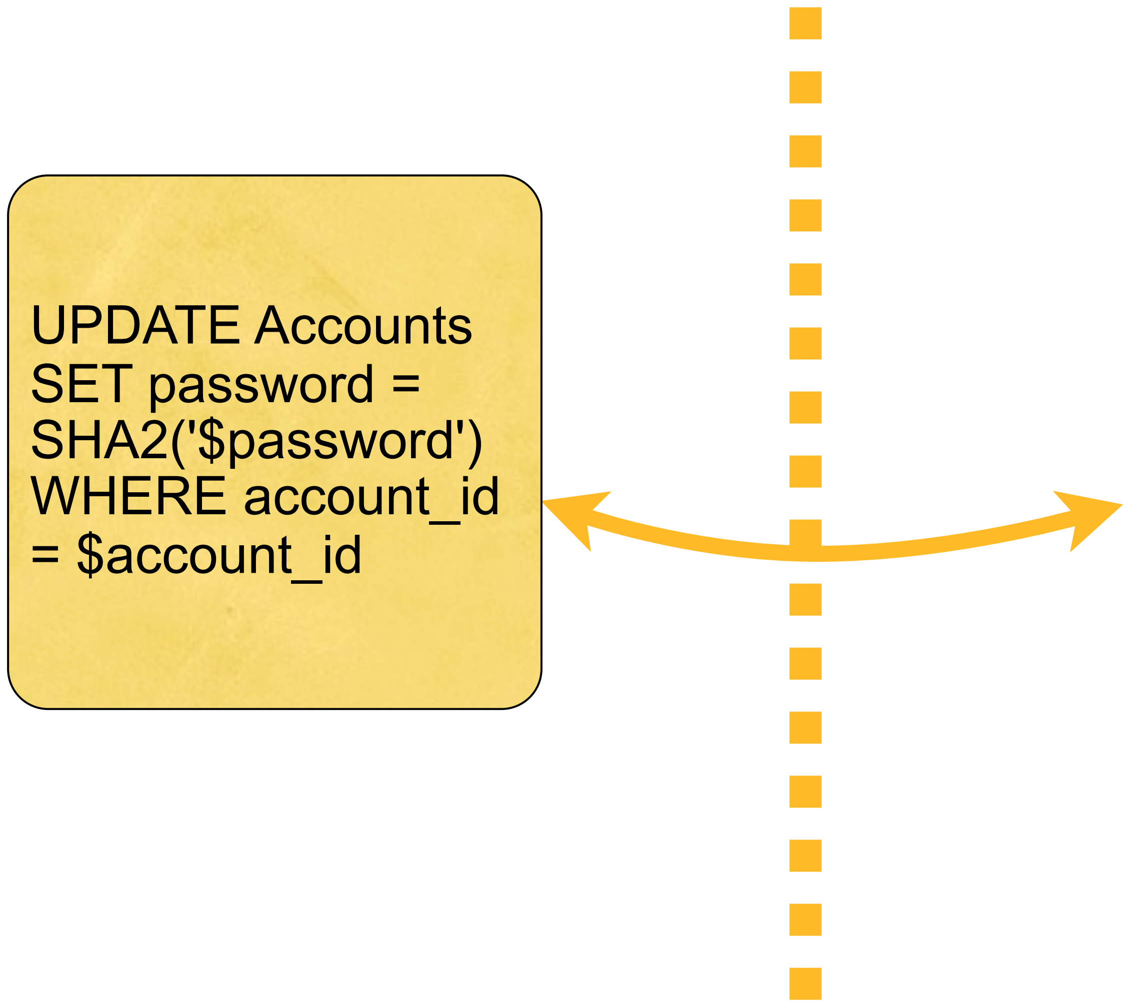
FALLACY

“It’s just an intranet application—it doesn’t need to be secure.”

Just Ask This Manager

What Stays on the Intranet?

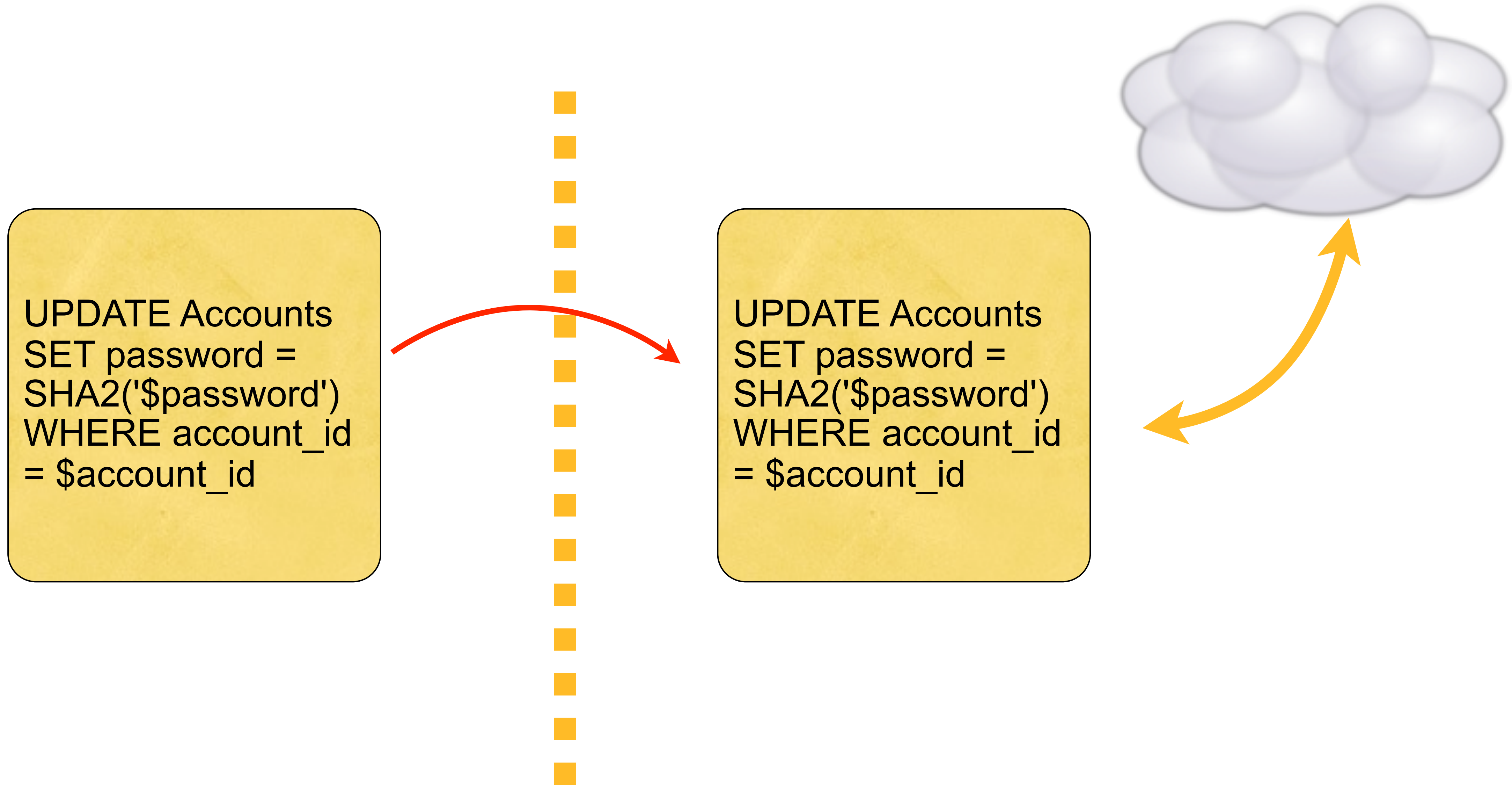
You could be told to give business partners access to an internal application



```
UPDATE Accounts
SET password =
SHA2('$password')
WHERE account_id
= $account_id
```

What Stays on the Intranet?

Your casual code could be copied & pasted into external applications

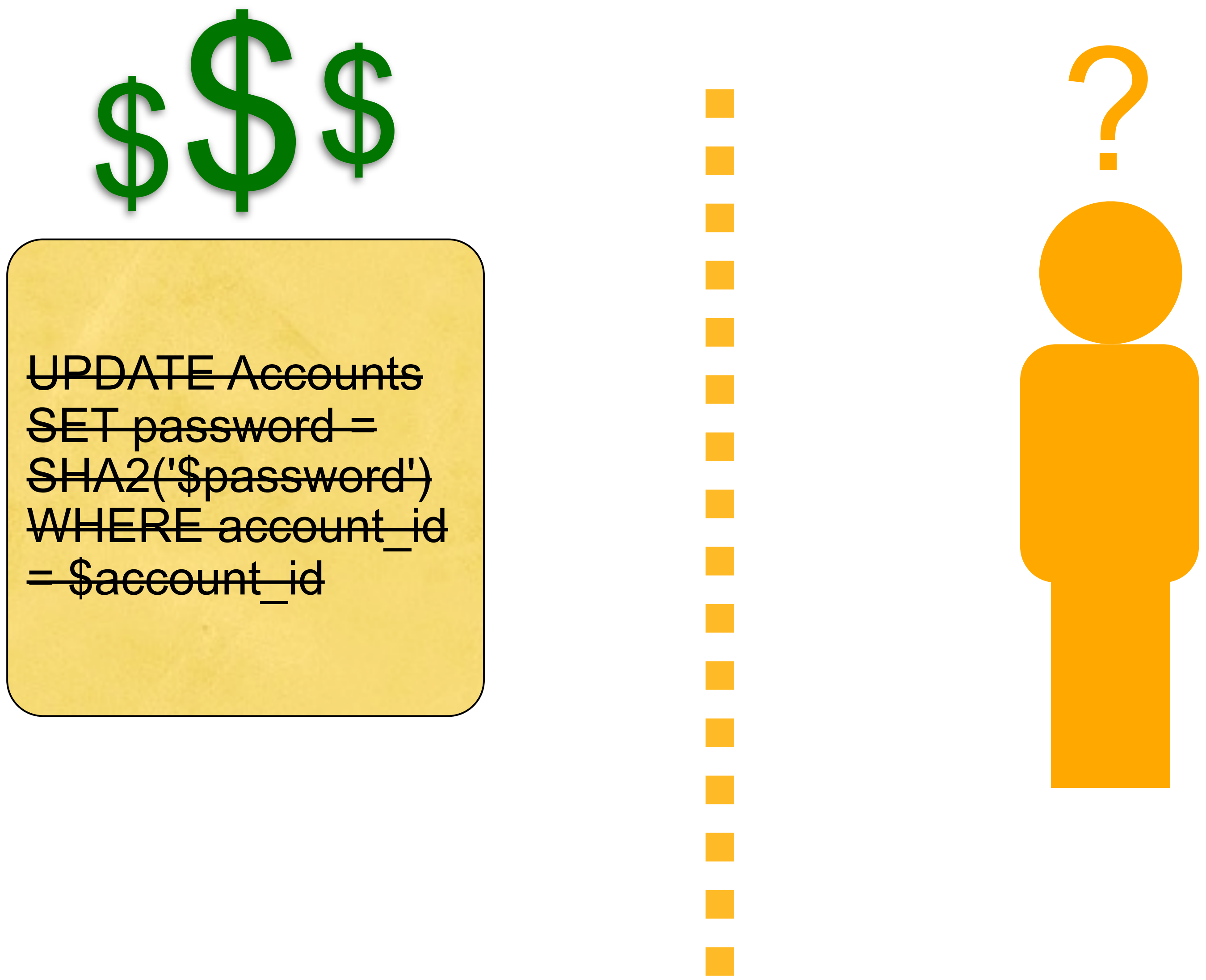


```
UPDATE Accounts
SET password =
SHA2('$password')
WHERE account_id
= $account_id
```

```
UPDATE Accounts
SET password =
SHA2('$password')
WHERE account_id
= $account_id
```

What Stays on the Intranet?

It's hard to argue for a security review or rewrite for a "finished" application



~~UPDATE Accounts~~
~~SET password =~~
~~SHA2('$password')~~
~~WHERE account_id~~
~~= $account_id~~

MYTH

“My framework prevents SQL Injection.”

ORMs Allow Custom SQL

Dynamic SQL always risks SQL Injection, for example Rails ActiveRecord:

```
Bugs.all(
    :joins => "JOIN Accounts
          ON  reported_by = account_id",

    :order => "date_reported DESC"
  )
```

any custom SQL can carry SQL injection

Whose Responsibility?

Security is the application developer's job

No database, connector, or framework can prevent SQL injection all the time

FALLACY

"Query parameters do quoting for you."

Interpolating Dynamic Values

Query needs a dynamic value:

```
SELECT * FROM Bugs
  WHERE bug_id = $_GET['bugid']
```

user input

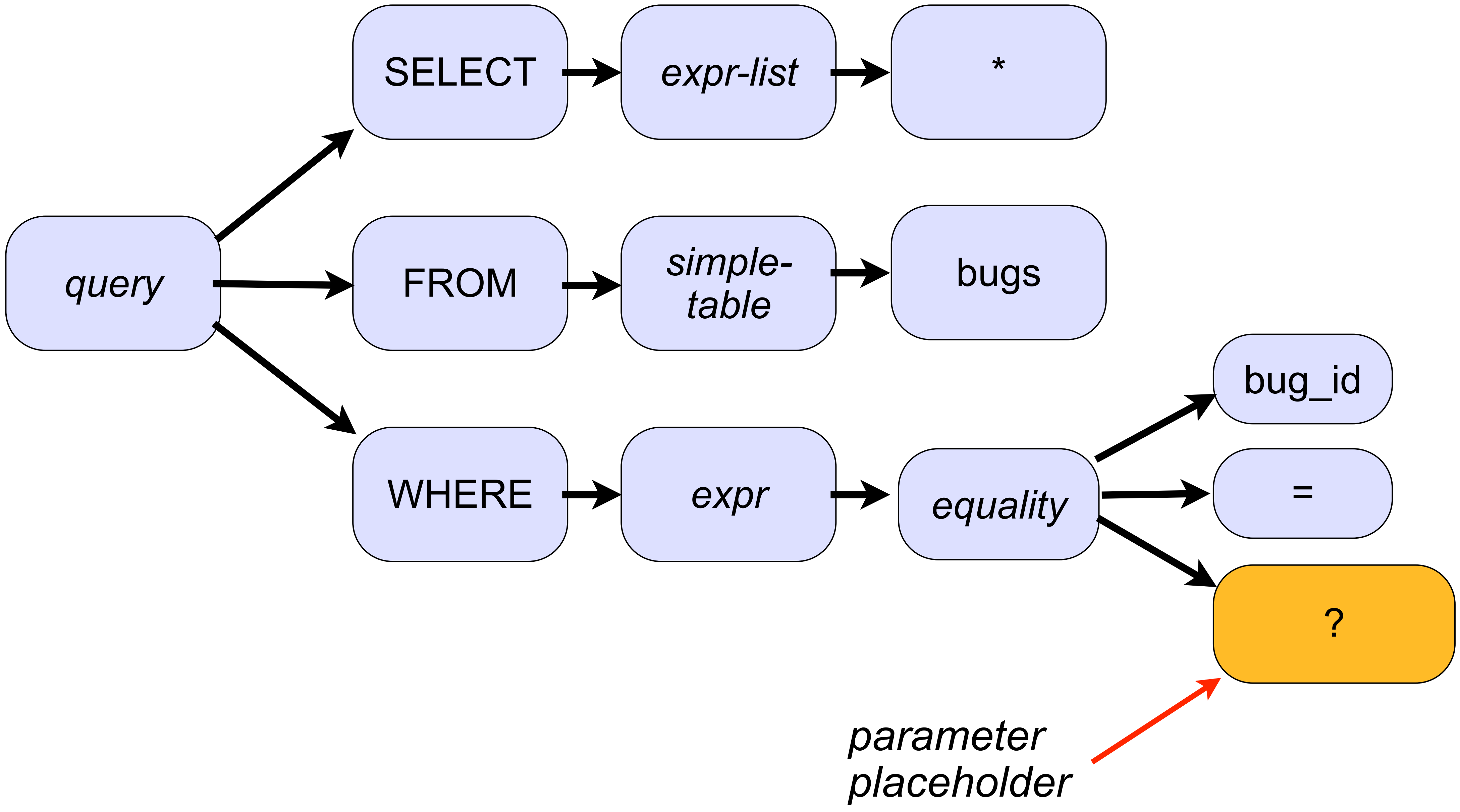
Using a Parameter

Query parameter takes the place of a dynamic value:

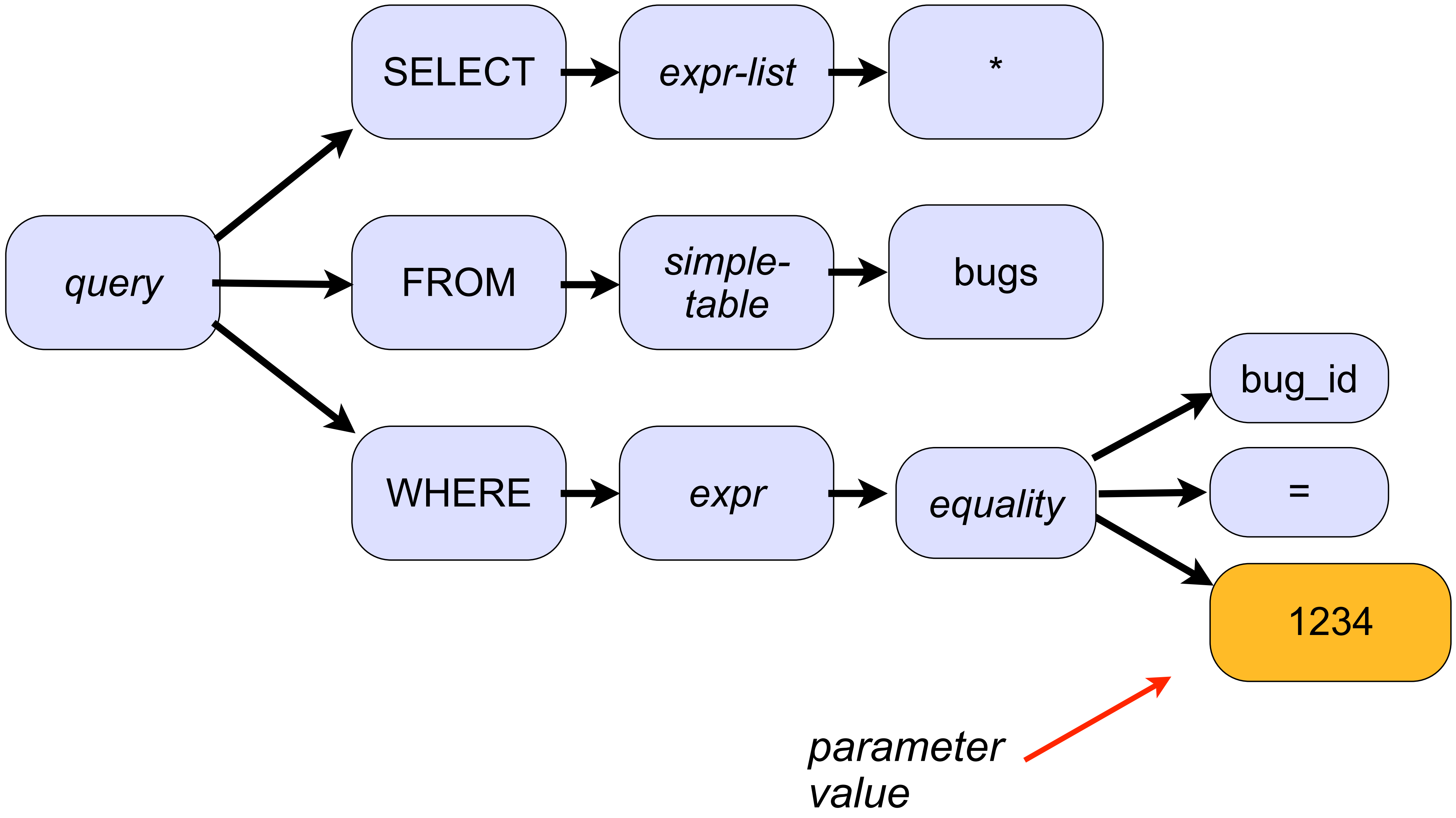
```
SELECT * FROM Bugs
  WHERE bug_id = ?
```

parameter placeholder

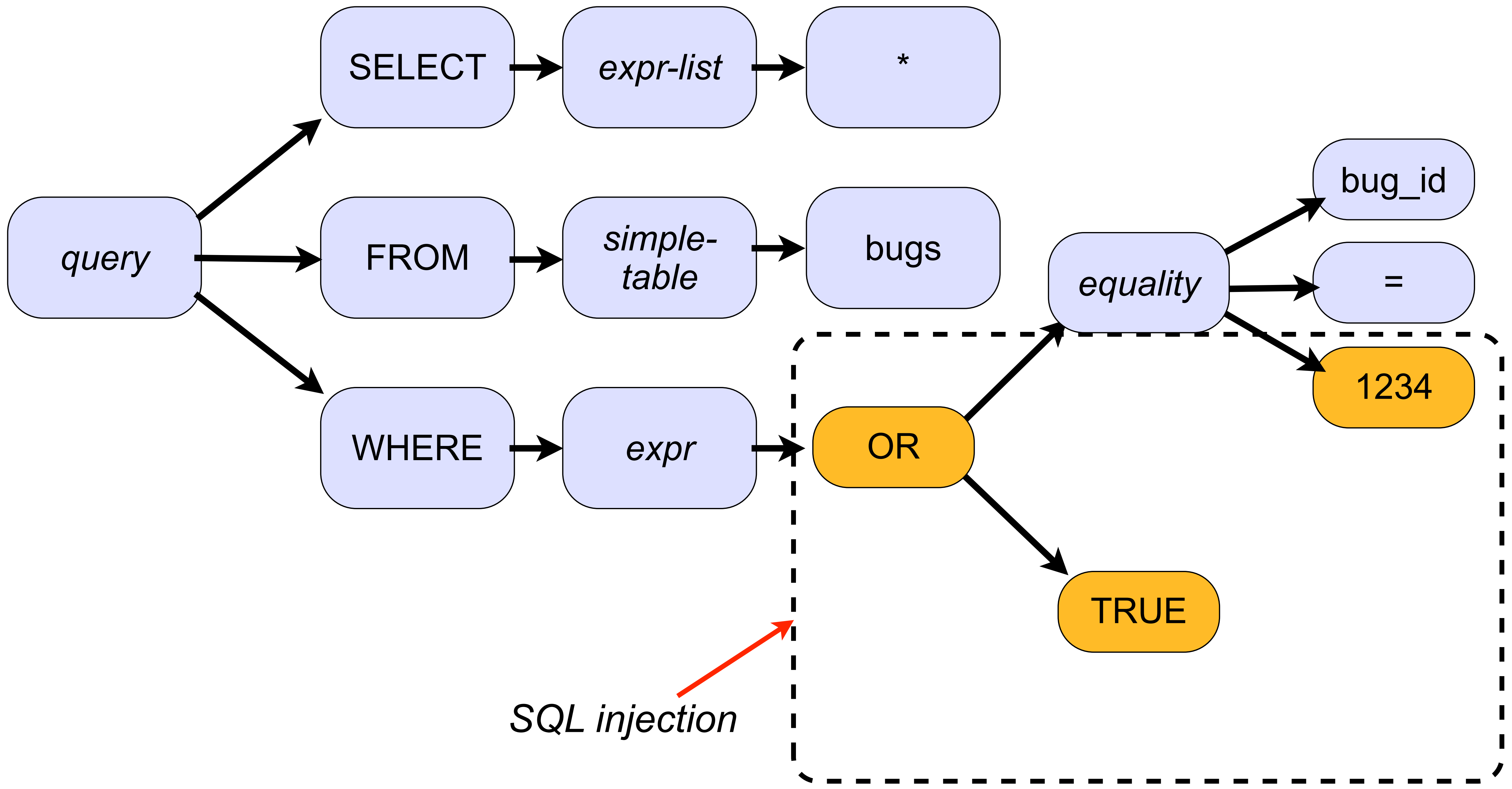
How the Database Parses It



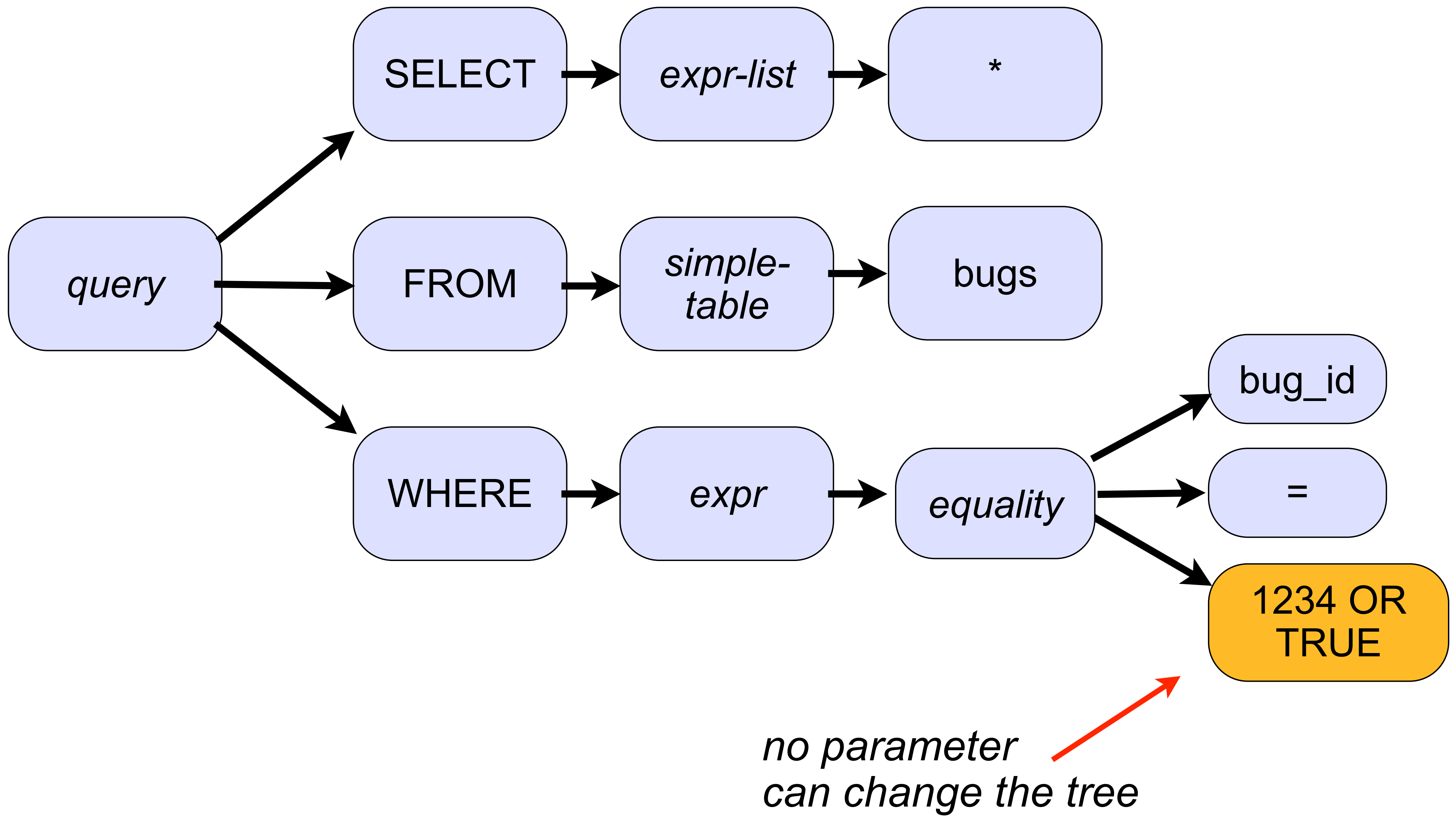
How the Database Executes It



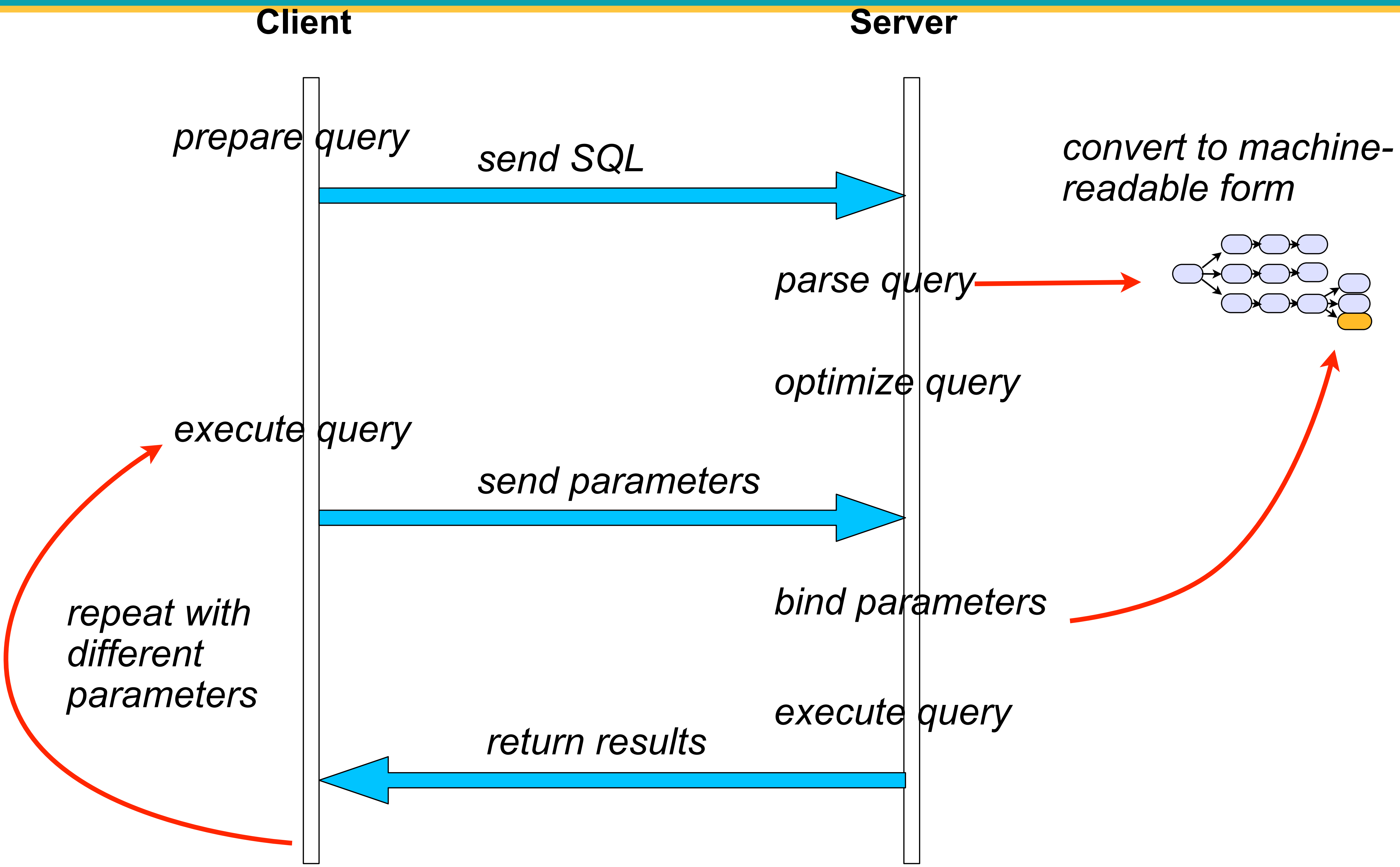
Interpolation



Parameterization



Sequence of Prepare & Execute



MYTH

"Query parameters prevent SQL Injection."

One Parameter = One Value

```
SELECT * FROM Bugs
  WHERE bug_id = ?
```

Not a List of Values

```
SELECT * FROM Bugs
  WHERE bug_id IN (  ?  )
```

Not a Table Name

```
SELECT * FROM   ?
  WHERE bug_id = 1234
```

Not a Column Name

```
SELECT * FROM Bugs
  ORDER BY  ?
```

Not an SQL Keyword

```
SELECT * FROM Bugs
  ORDER BY date_reported  ?
```

Interpolation vs. Parameters

Scenario	Example Value	Interpolation	Parameter
single value	‘1234’	SELECT * FROM Bugs WHERE bug_id = $id	SELECT * FROM Bugs WHERE bug_id = ?
multiple values	‘1234, 3456, 5678’	SELECT * FROM Bugs WHERE bug_id IN ($list)	SELECT * FROM Bugs WHERE bug_id IN (?, ?, ?)
table name	‘Bugs’	SELECT * FROM $table WHERE bug_id = 1234	*NO*
column name	‘date_reported’	SELECT * FROM Bugs ORDER BY $column	*NO*
other syntax	‘DESC’	SELECT * FROM Bugs ORDER BY date_reported $direction	*NO*

SOLUTION

Whitelist Maps

Example SQL Injection

http://www.example.com/?
   order=date_reported&dir=ASC

unsafe inputs

SQL Injection

```
<?php

$sortorder = $_GET["order"];
   $direction = $_GET["dir"];

$sql = "SELECT * FROM Bugs
        ORDER BY {$sortorder} {$direction}";

$stmt = $pdo->query($sql);
```

Fix with a Whitelist Map

application request values

SQL identifiers and keywords

```
<?php
$sortorders = array("DEFAULT"          => "bug_id",
                    "status"           => "status",
                    "date"             => "date_reported" );


$directions = array( "DEFAULT"         => "ASC",
                     "up"              => "ASC",
                     "down"            => "DESC" );
```

Map User Input to Safe SQL

```
<?php

if (isset( $sortorders[ $_GET["order"] ]))
{
    $sortorder = $sortorders[ $_GET["order"] ];
} else {
    $sortorder = $sortorders["DEFAULT"];
}
```

Map User Input to Safe SQL

PHP 5.3 syntax

```
<?php

$direction =  $directions[ $_GET["dir"] ]    ?:
              $directions["DEFAULT"];
```

Interpolate Safe SQL

http://www.example.com/?order=date&dir=up

```
<?php

$sql =  "SELECT * FROM Bugs
         ORDER BY {$sortorder} {$direction}";

$stmt = $pdo->query($sql);
```

whitelisted values

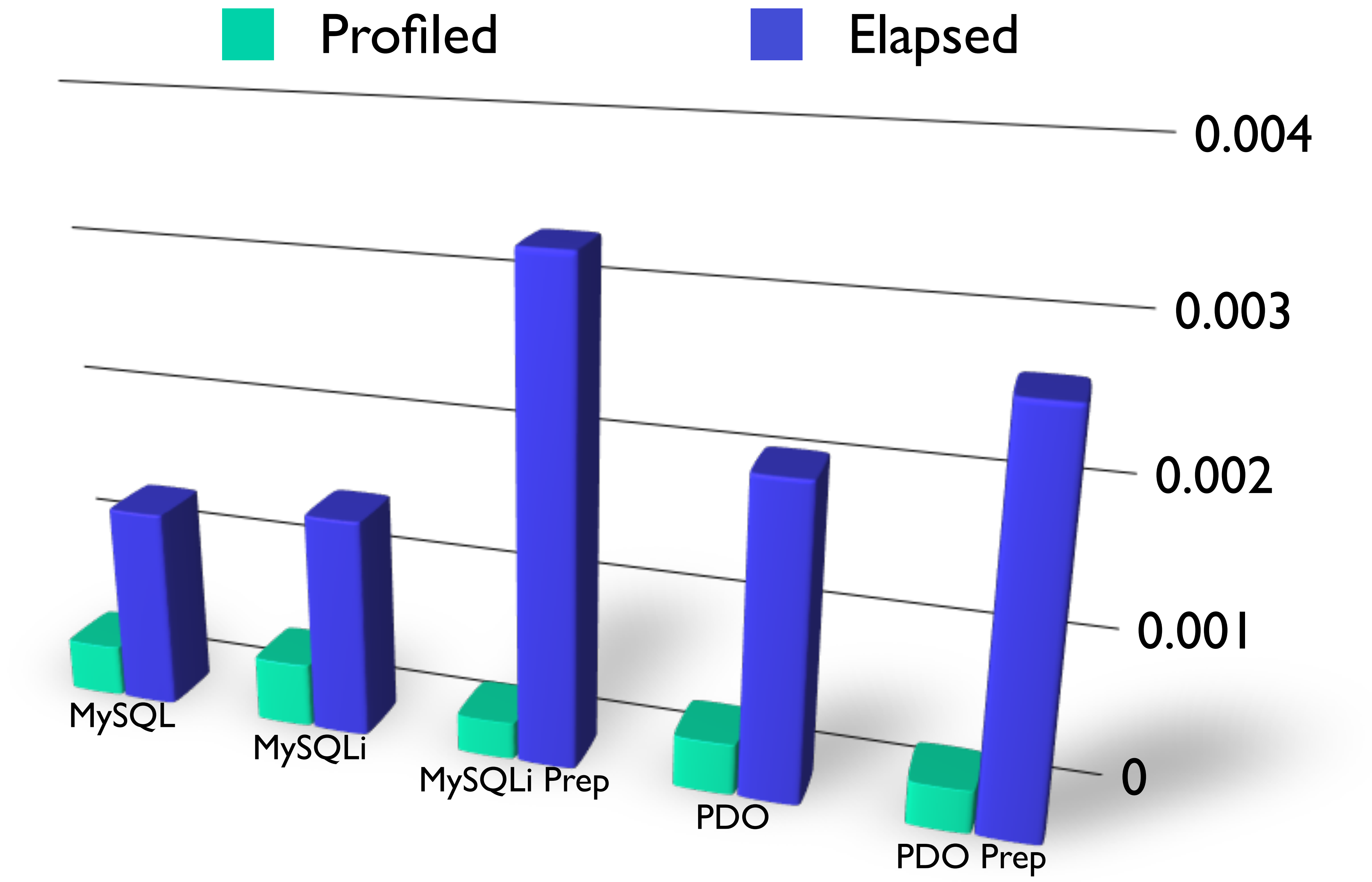
Benefits of Whitelist Maps

- Protects against SQL injection in cases where escaping and parameterization doesn’t help.
- Decouples web interface from database schema.
- Uses simple, declarative technique.
- Works independently of any framework.

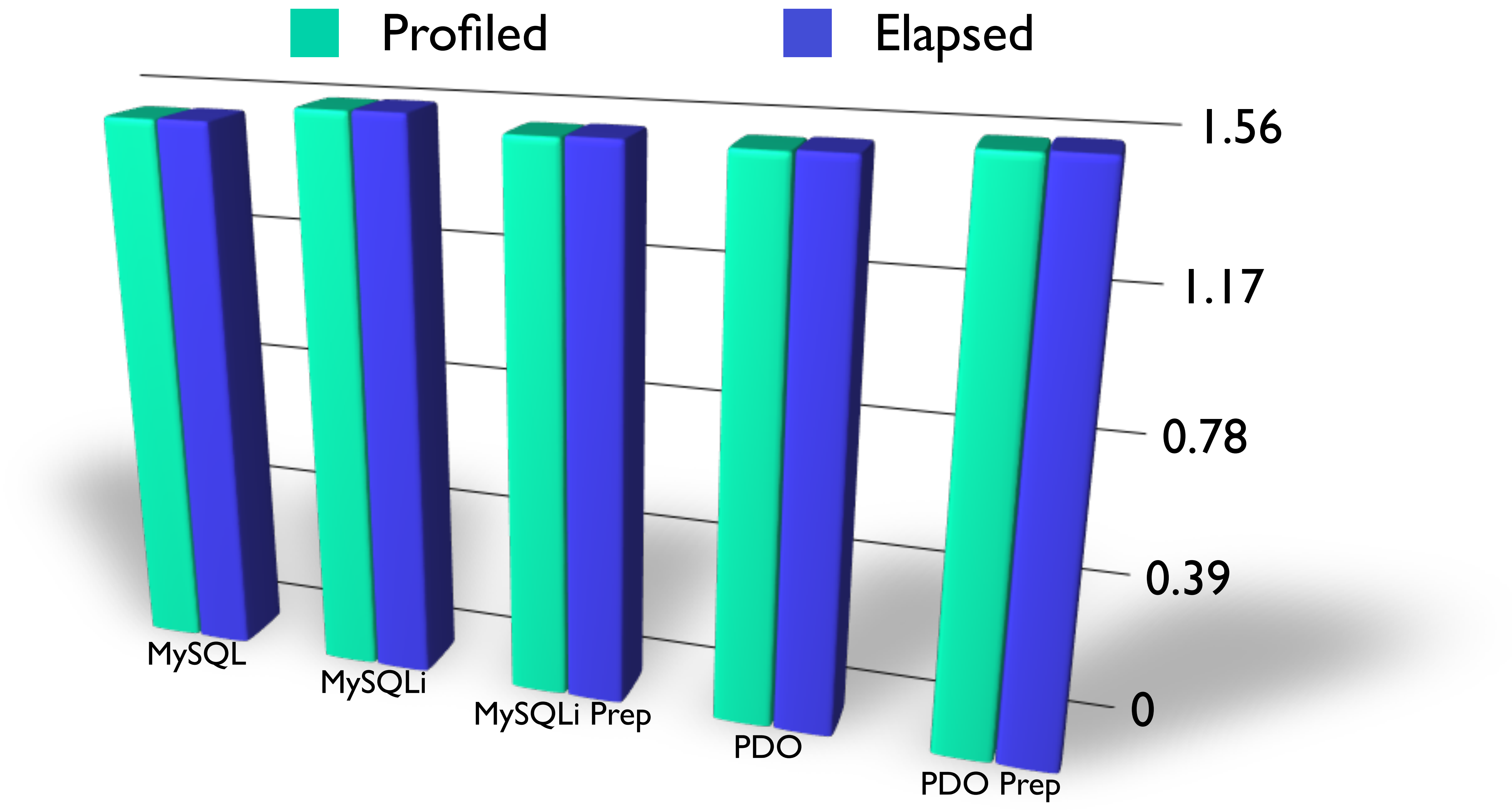
FALLACY

“Queries parameters hurt SQL performance.”

Simple Query



Complex Query



MYTH

“A proxy/firewall solution prevents SQL injection.”

Oracle Database Firewall

Reverse proxy between application and Oracle

- Whitelist of known SQL queries
- Learns legitimate queries from application traffic
- Blocks unknown SQL queries
- Also supports Microsoft SQL Server, IBM DB2, Sybase ASE, SQL Anywhere

http://www.oracle.com/technetwork/database/database-firewall/overview/index.html

GreenSQL

Reverse proxy for MySQL, PostgreSQL, Microsoft SQL Server

Detects / reports / blocks “suspicious” queries:

- Access to sensitive tables
- Comments inside SQL commands
- Empty password
- An ‘or’ token inside a query
- An SQL expression that always returns true

http://www.greensql.net/about

Still not Perfect

Vipin Samar, Oracle vice president of Database Security:

> “Database Firewall is a good first layer of defense for databases but it won't protect you from everything,”

http://www.databasejournal.com/features/oracle/article.php/3924691/article.htm

GreenSQL Architecture

> “GreenSQL can sometimes generate false positive and false negative errors. As a result, some legal queries may be blocked or the GreenSQL system may pass through an illegal query undetected.”

http://www.greensql.net/about

Limitations of Proxy Solutions

- False sense of security; discourages code review
- Gating factor for emergency code deployment
- Constrains application from writing dynamic SQL
- Doesn’t stop SQL injection in Stored Procedures

FALLACY

"NoSQL databases are immune to SQL injection."

“NoSQL Injection”

http://www.example.com?column=password

```
<?php

$map = new MongoCode("function() {
        emit(this." . $_GET["column"] . ",1);
  } ");

$data = $db->command( array(
        "mapreduce" => "Users",
        "map" => $map
  ) );
```

any string-interpolation of untrusted content is Code Injection

NoSQL Injection in the Wild

Diaspora wrote MongoDB map/reduce functions dynamically from Ruby on Rails:

```
def self.search(query)
    Person.all('$where' => "function() {
      return this.diaspora_handle.match(/^#{query}/i) ||
        this.profile.first_name.match(/^#{query}/i) ||
        this.profile.last_name.match(/^#{query}/i); }")
  end
```

did query come from a trusted source?

http://www.kalzumeus.com/2010/09/22/security-lessons-learned-from-the-diaspora-launch/

Myths and Fallacies

I don't have to worry anymore

Escaping is the fix

More escaping is better

I can code an escaping function

Only user input is unsafe

Stored procs are the fix

SQL privileges are the fix

My app doesn't need security

Frameworks are the fix

Parameters quote for you

Parameters are the fix

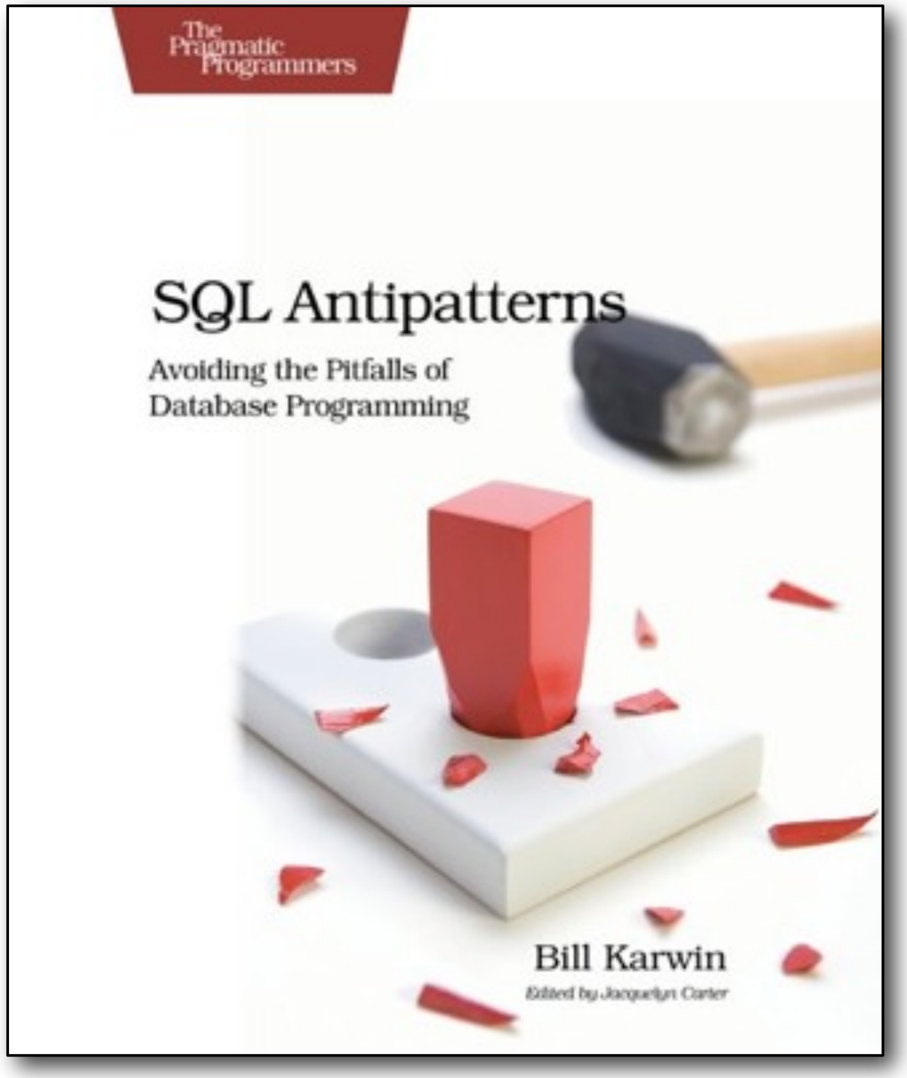
Parameters make queries slow

SQL proxies are the fix

NoSQL databases are the fix

there is no single silver bullet—
use all defenses when appropriate

SQL Antipatterns


The Pragmatic Programmers
SQL Antipatterns
Avoiding the Pitfalls of Database Programming
Bill Karwin
Edited by Jacquelyn Carter

http://www.pragprog.com/titles/bksqla/