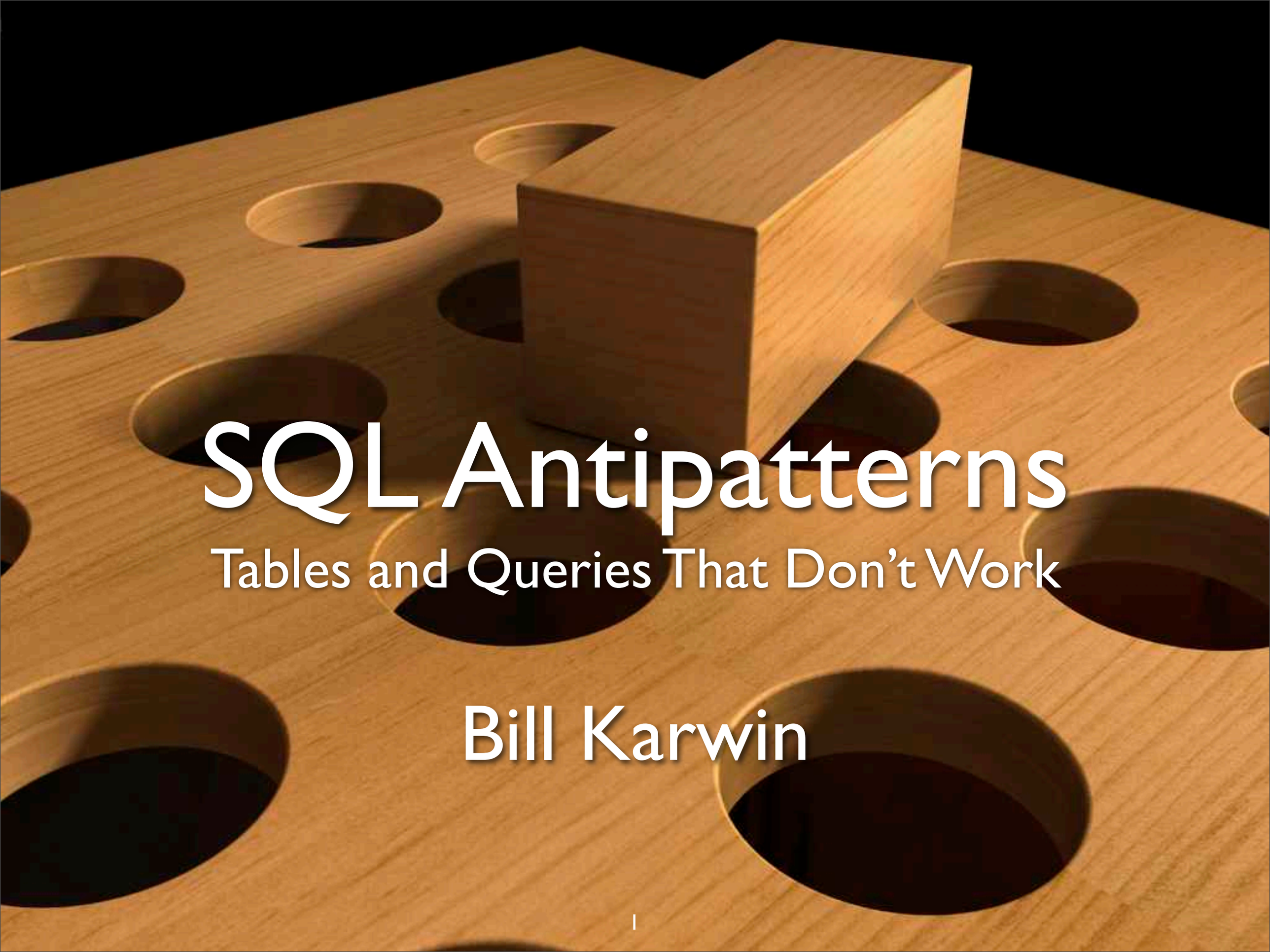


SQL Antipatterns

Tables and Queries That Don't Work

Bill Karwin

Antipattern Categories

Logical Database Antipatterns

Physical Database Antipatterns

```
CREATE TABLE BugsProducts (
  bug_id   INTEGER REFERENCES Bugs,
  product  VARCHAR(100) REFERENCES Products,
  PRIMARY KEY (bug_id, product)
);
```

Query Antipatterns

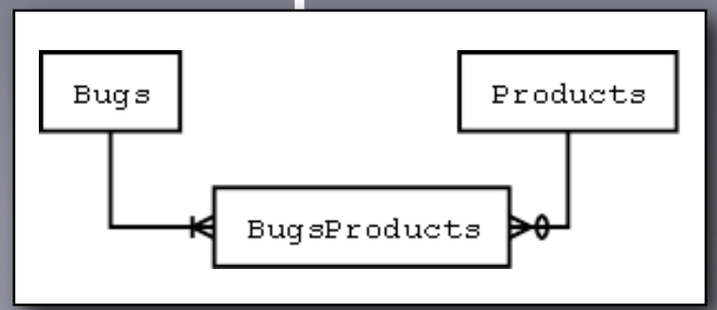
```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

```
$dbHandle = new PDO('mysql:dbname=test');
$stmt = $dbHandle->prepare($sql);
$result = $stmt->fetchAll();
```

Antipattern Categories

Logical Database Antipatterns



Physical Database Antipatterns

```
CREATE TABLE BugsProducts (
  bug_id   INTEGER REFERENCES Bugs,
  product  VARCHAR(100) REFERENCES Products,
  PRIMARY KEY (bug_id, product)
);
```

Query Antipatterns

```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

```
$dbHandle = new PDO('mysql:dbname=test');
$stmt = $dbHandle->prepare($sql);
$result = $stmt->fetchAll();
```

Logical Database Antipatterns

1. Comma-Separated Lists
2. Multi-Column Attributes
3. Entity-Attribute-Value
4. Metadata Tribbles

Comma-Separated Lists

Comma-Separated Lists

- **Example:** table BUGS references table PRODUCTS using a foreign key

Comma-Separated Lists

```
CREATE TABLE bugs (
        bug_id          SERIAL PRIMARY KEY,
        description     VARCHAR(200),
        product_id      BIGINT
                        REFERENCES products
)
```

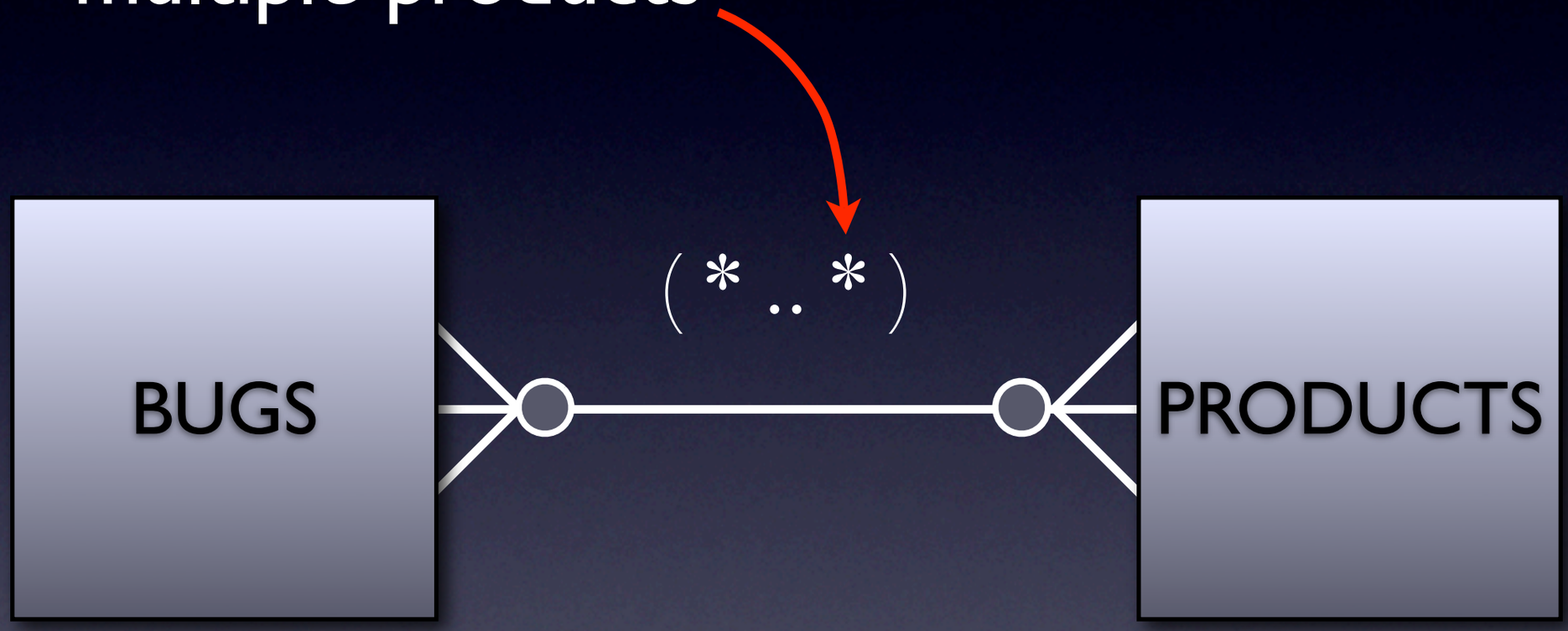
```
CREATE TABLE products (
        product_id      SERIAL PRIMARY KEY,
        product_name    VARCHAR(50)
)
```

bug_id	description	product_id
1234	crashes while saving	1
3456	fix performance	2
5678	support XML	3

product_id	product_name
1	Open RoundFile
2	Visual TurboBuilder
3	ReConsider

Comma-Separated Lists

- **Objective:** Allow a bug to reference multiple products



Comma-Separated Lists

- **Antipattern:** reference multiple products in a comma-separated list

```
CREATE TABLE bugs (
        bug_id          SERIAL PRIMARY KEY,
        description     VARCHAR(200),
        product_id      VARCHAR(50)
)
```

bug_id	description	product_id
1234	crashes while saving	1
3456	increase performance	1, 2, 3
5678	support XML	3

Comma-Separated Lists

- Difficult to add a value to the list:

```
UPDATE bugs
SET product_id = CONCAT(product_id, ',', 2)
WHERE bug_id = 3456
```

- Even more difficult to remove a value

Comma-Separated Lists

- Difficult to count products per bug:

```
SELECT LENGTH(product_id)
    – LENGTH(REPLACE(product_id, ',', ''))+1
    AS num_products
FROM bugs
```

Comma-Separated Lists

- Difficult to count bugs per product:

```
SELECT SUM( FIND_IN_SET($id, product_id) > 0 )
    AS num_bugs
FROM bugs
```

Comma-Separated Lists

- Difficult to search for bugs for a specific product:

```
SELECT *
FROM bugs
WHERE product_id RLIKE '[[:<:]]2[[:>:]]'
```

Comma-Separated Lists

- Difficult to join BUGS to PRODUCTS:

```
SELECT *
FROM bugs b JOIN products p ON b.product_id
    RLIKE CONCAT('[[:<:]]', p.product_id, '[[:>:]]')
```

Comma-Separated Lists

- Can't enforce referential integrity or data type integrity:

```
INSERT INTO bugs (product_id)
    VALUES ( '23, 57, 94' )

INSERT INTO bugs (product_id)
    VALUES ( '1, 2, banana, 3' )
```

do these values exist?

- How does your application detect and correct bogus data?

Comma-Separated Lists

- **Solution:** a many-to-many relationship *always* requires an intersection table.

Comma-Separated Lists

- The intersection table references both tables

```
CREATE TABLE bugs_prods (
    bug_id          BIGINT REFERENCES bugs,
    product_id      BIGINT REFERENCES products,
    PRIMARY KEY (bug_id, product_id)
)
```

Comma-Separated Lists

- The intersection table references both tables

bug_id	description
1234	crashes while saving
3456	fix performance
5678	support XML

bug_id	product_id
1234	1
3456	1
3456	2
3456	3
5678	3

product_id	product_name
1	Open RoundFile
2	Visual TurboBuilder
3	ReConsider

Comma-Separated Lists

Add a product:

```
INSERT INTO bugs_prods
  VALUES (1234, 2)
```

Delete a product:

```
DELETE FROM bugs_prods
WHERE bug_id = 1234
  AND product_id = 2
```

Query bugs for a product:

```
SELECT * FROM bugs
  JOIN bugs_prods USING (bug_id)
WHERE product_id = 2
```

Comma-Separated Lists

Join bugs to products:

```
SELECT * FROM bugs
  JOIN bugs_prods USING (bug_id)
  JOIN products USING (product_id)
WHERE bug_id = 1234
```

Count bugs per product:

```
SELECT product_id, COUNT(*)
FROM bugs_prods
GROUP BY product_id
```

Count products per bug:

```
SELECT bug_id, COUNT(*)
FROM bugs_prods
GROUP BY bug_id
```

Multi-Column Attributes

Multi-Column Attributes

- **Objective:** support an attribute with multiple values

bug_id	description	product_id
1234	crashes while saving	1
3456	fix performance	2
5678	support XML	3

Multi-Column Attributes

- **Antipattern:** add more columns

```
CREATE TABLE bugs (
    bug_id          SERIAL PRIMARY KEY,
    description     VARCHAR(200),
    product_id1     BIGINT REFERENCES products,
    product_id2     BIGINT REFERENCES products,
    product_id3     BIGINT REFERENCES products
)
```

bug_id	description	product_id1	product_id2	product_id3
1234	crashes while saving	1	NULL	NULL
3456	increase performance	1	2	3
5678	support XML	3	NULL	NULL

Multi-Column Attributes

- Search must reference all columns:

```
SELECT * FROM bugs
WHERE    product_id1 = 1
    OR   product_id2 = 1
    OR   product_id3 = 1
```

Multi-Column Attributes

- Search for multiple products (two forms):

```
SELECT * FROM bugs
WHERE ( product_id1 = 1
    OR     product_id2 = 1
    OR     product_id3 = 1 )
 AND (    product_id1 = 3
    OR     product_id2 = 3
    OR     product_id3 = 3 )

SELECT * FROM bugs
WHERE    1 IN (product_id1, product_id2, product_id3)
 AND     3 IN (product_id2, product_id3, product_id4)
```

Multi-Column Attributes

- Remove product 2 from a row:

	bug_id	product_id1	product_id2	product_id3
BEFORE	1234	1	3	2
AFTER	1234	1	3	NULL

```
UPDATE bugs
SET  product_id1 = NULLIF(product_id1, 2),
     product_id2 = NULLIF(product_id2, 2),
     product_id3 = NULLIF(product_id3, 2)
WHERE bug_id = 1234
```

Multi-Column Attributes

- Add product 3 to a row:

	bug_id	product_id1	product_id2	product_id3
BEFORE	1234	1	NULL	2
AFTER	1234	1	3	2

```
UPDATE bugs
SET     product_id1 = CASE
            WHEN 3 IN (product_id2, product_id3) THEN product_id1
            ELSE COALESCE(product_id1, 3) END,
        product_id2 = CASE
            WHEN 3 IN (product_id1, product_id3) THEN product_id2
            ELSE COALESCE(product_id2, 3) END,
        product_id3 = CASE
            WHEN 3 IN (product_id1, product_id2) THEN product_id3
            ELSE COALESCE(product_id3, 3) END
WHERE bug_id = 1234
```

Multi-Column Attributes

- How many columns are enough?
  - Add a new column for fourth product:

```
ALTER TABLE bugs ADD COLUMN product_id4
    BIGINT REFERENCES products
```

- Rewrite all queries searching for products:

```
SELECT * FROM bugs
WHERE   product_id1 = 2
   OR   product_id2 = 2
   OR   product_id3 = 2
   OR   product_id4 = 2
```

Multi-Column Attributes

- **Solution:** use an intersection table

Query for two products:

```
SELECT * FROM bugs
  JOIN bugs_prods AS p1 USING (bug_id)
  JOIN bugs_prods AS p2 USING (bug_id)
WHERE  p1.product_id = 1
  AND    p2.product_id = 3
```

Entity-Attribute-Value

Entity-Attribute-Value

- **Objective:** make a table with a variable set of attributes

bug_id	bug_type	priority	description	severity	sponsor
1234	BUG	high	crashes when saving	loss of functionality	
3456	FEATURE	low	support XML		Acme Corp.

Entity-Attribute-Value

- **Antipattern:** store all attributes in a second table, one attribute per row

```
CREATE TABLE eav (
    bug_id          BIGINT REFERENCES bugs,
    attr_name       VARCHAR(20) NOT NULL,
    attr_value      VARCHAR(100),
    PRIMARY KEY (bug_id, attr_name)
)
```

mixing data with metadata

Entity-Attribute-Value

bug_id	attr_name	attr_value
1234	priority	high
1234	description	crashes when saving
1234	severity	loss of functionality
3456	priority	low
3456	description	support XML
3456	sponsor	Acme Corp.

Entity-Attribute-Value

- Difficult to ensure consistent attribute names

bug_id	attr_name	attr_value
1234	created	2008-04-01
3456	created_date	2008-04-01

Entity-Attribute-Value

- Difficult to enforce data type integrity

bug_id	attr_name	attr_value
1234	created_date	2008-02-31
3456	created_date	banana

Entity-Attribute-Value

- Difficult to enforce mandatory attributes (i.e. NOT NULL)
- SQL constraints apply to columns, not rows
- No way to declare that a row must exist with a certain *attr_name* value (‘created_date’)
- Maybe create a trigger on INSERT for bugs?

Entity-Attribute-Value

- Difficult to enforce referential integrity for attribute values

bug_id	attr_name	attr_value
1234	priority	new
3456	priority	fixed
5678	priority	banana

- Constraints apply to all rows in the column, not selected rows depending on value in *attr_name*

Entity-Attribute-Value

- Difficult to reconstruct a row of attributes:

```
SELECT b.bug_id,
       e1.attr_value AS `created_date`,
       e2.attr_value AS `priority`
FROM bugs b
  LEFT JOIN eav e1 ON (b.bug_id = e1.bug_id
      AND e1.attr_name = 'created_date')
  LEFT JOIN eav e2 ON (b.bug_id = e2.bug_id
      AND e2.attr_name = 'priority')
```

need one JOIN per attribute

bug_id	created_date	priority
1234	2008-04-01	high

Entity-Attribute-Value

- **Solution:** use *metadata* for metadata
  - Define attributes in columns
  - ALTER TABLE to add attribute columns
  - Define related tables for related types

Entity-Attribute-Value

- Define similar tables for similar types: "Sister Tables"

```
CREATE TABLE bugs (
    bug_id          SERIAL PRIMARY KEY,
    created_date    DATE NOT NULL,
    priority        VARCHAR(20),
    description     TEXT,
    severity        VARCHAR(20)
)
```

```
CREATE TABLE features (
    bug_id          SERIAL PRIMARY KEY,
    created_date    DATE NOT NULL,
    priority        VARCHAR(20),
    description     TEXT,
    sponsor         VARCHAR(100)
)
```

Entity-Attribute-Value

- Define related tables for related types: "Inherited Tables"

```
CREATE TABLE issues (
    issue_id        SERIAL PRIMARY KEY,
    created_date    DATE NOT NULL
    priority        VARCHAR(20),
    description     TEXT
)
```

```
CREATE TABLE bugs (
    issue_id    BIGINT PRIMARY KEY
                REFERENCES issues,
    severity    VARCHAR(20)
)
```

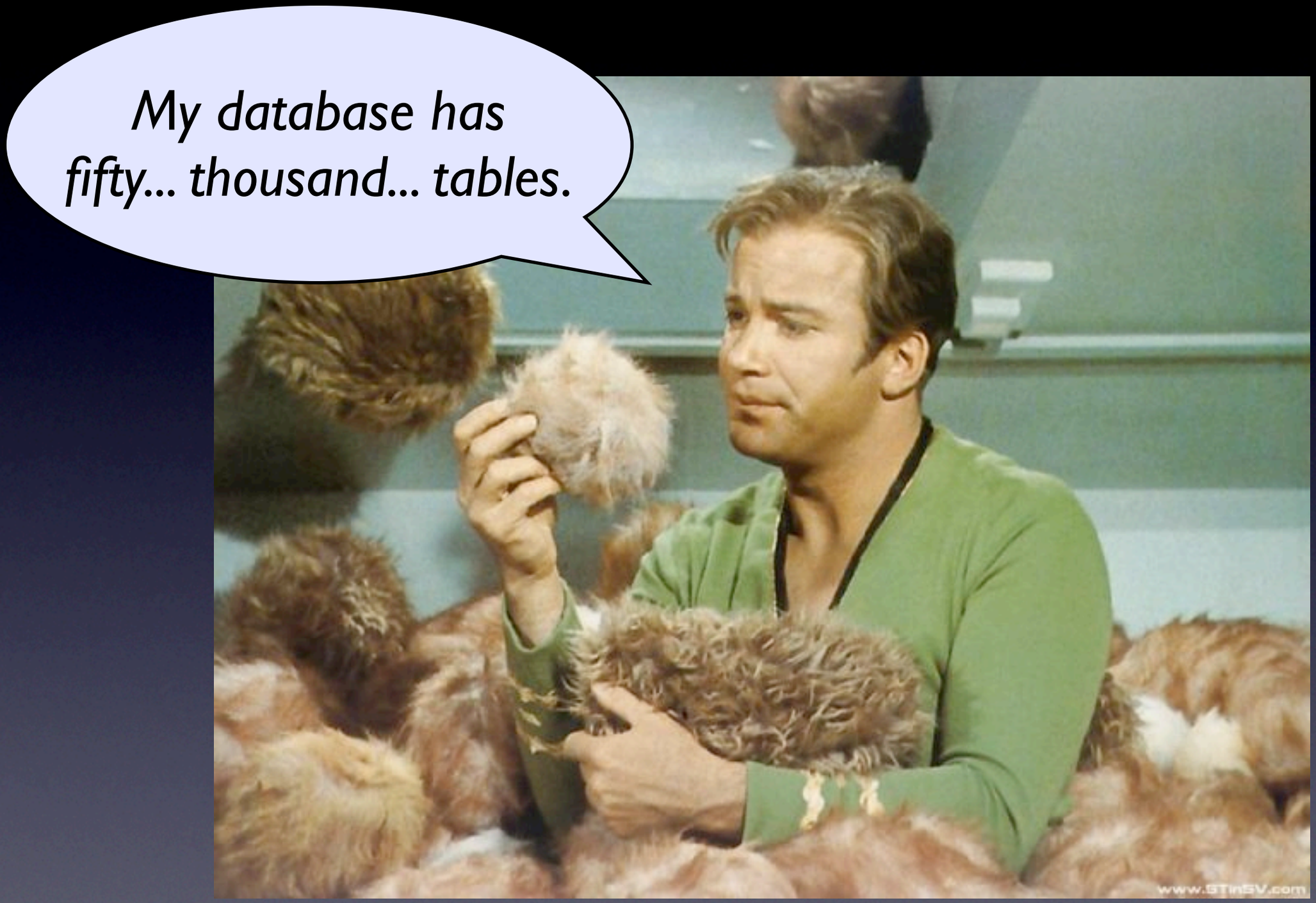
```
CREATE TABLE features (
    issue_id    BIGINT PRIMARY KEY
                REFERENCES issues,
    sponsor     VARCHAR(100)
)
```

Entity-Attribute-Value

- Appropriate usage of EAV
  - When attributes must be truly dynamic
  - Enforce constraints in application code
  - Don’t try to fetch one object in a single row
  - Consider non-relational solutions for semi-structured data, e.g. RDF/XML

Metadata Tribbles

Metadata Tribbles



Metadata Tribbles

- **Objective:** improve performance of a very large table

Metadata Tribbles

- **Antipattern:** separate into many tables with similar structure
- Separate tables per distinct value in atttribute
- e.g., per year, per month, per user, per postal code, etc.

Metadata Tribbles

- Must create a new table for each new value

```
CREATE TABLE bugs_2005 ( . . . );
CREATE TABLE bugs_2006 ( . . . );
CREATE TABLE bugs_2007 ( . . . );
CREATE TABLE bugs_2008 ( . . . );
. . .
```

mixing data with metadata

Metadata Tribbles

- Automatic primary keys cause conflicts

```
CREATE TABLE bugs_2005 (bug_id SERIAL . . . );
CREATE TABLE bugs_2006 (bug_id SERIAL . . . );
CREATE TABLE bugs_2007 (bug_id SERIAL . . . );
CREATE TABLE bugs_2008 (bug_id SERIAL . . . );
. . .
```

Metadata Tribbles

- Difficult to query across tables

```
SELECT b.status, COUNT(*) AS count_per_status
FROM (
        SELECT * FROM bugs_2008
         UNION
        SELECT * FROM bugs_2007
         UNION
        SELECT * FROM bugs_2006 ) AS b
GROUP BY b.status
```

Metadata Tribbles

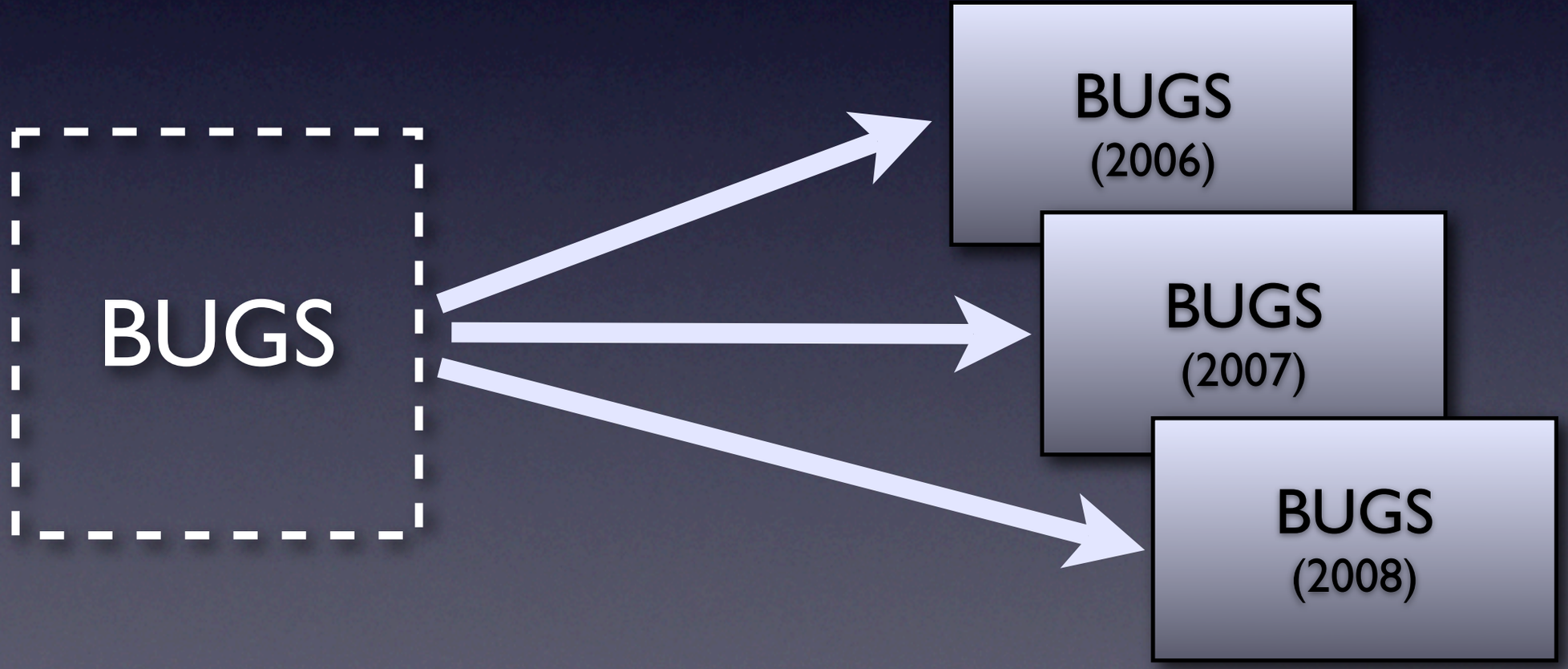
- Table structures are not kept in sync

```
ALTER TABLE bugs_2008
ADD COLUMN work_estimate_hrs NUMERIC
```

- Prior tables don't contain new column
- Dissimilar tables can't be combined with UNION

Metadata Tribbles

- **Solution #1:** use horizontal partitioning
  - Physically split, while logically whole
  - MySQL 5.1 supports partitioning



Metadata Tribbles

- **Solution #2:** use vertical partitioning
  - Move bulky and seldom-used columns to a second table in one-to-one relationship

Metadata Tribbles

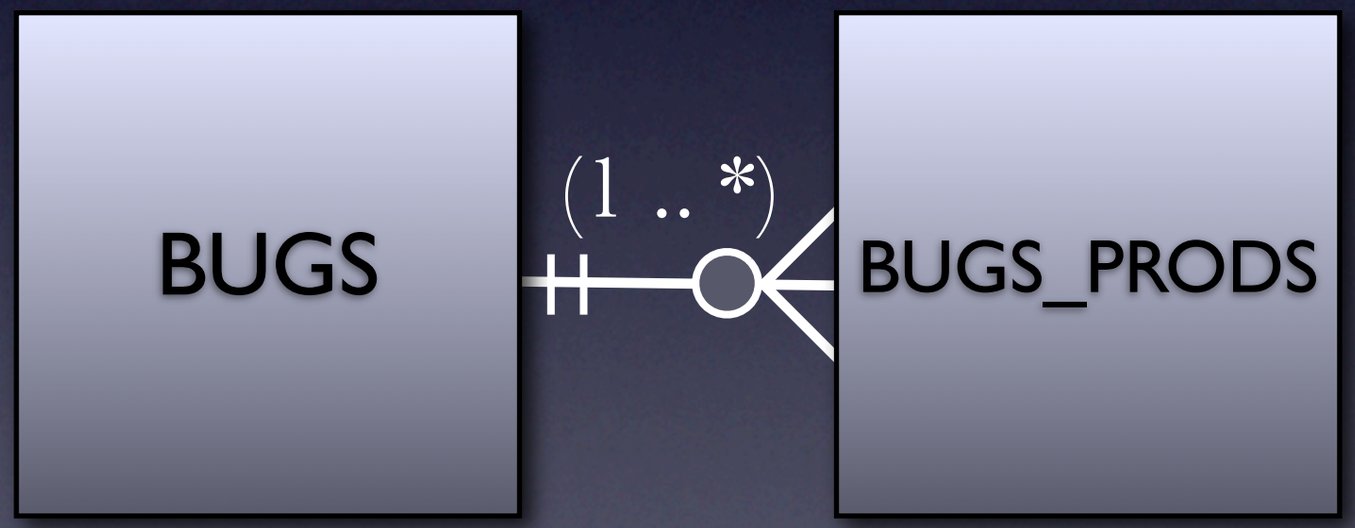
- Columns can also be tribbles:

```
CREATE TABLE bugs (
    bug_id          SERIAL PRIMARY KEY,
    ...
    product_id1     BIGINT,
    product_id2     BIGINT,
    product_id3     BIGINT
)
```

Metadata Tribbles

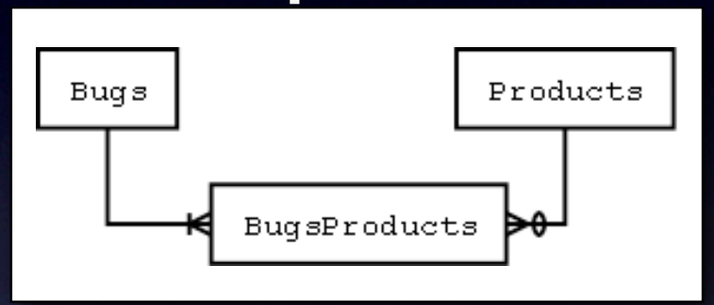
- **Solution #3:** add a dependent table

```
CREATE TABLE bugs_prods (
        bug_id          BIGINT REFERENCES bugs
        product_id      BIGINT REFERENCES products,
        PRIMARY KEY (bug_id, product_id)
)
```



Antipattern Categories

Logical Database Antipatterns



Physical Database Antipatterns

```
CREATE TABLE BugsProducts (
  bug_id   INTEGER REFERENCES Bugs,
  product  VARCHAR(100) REFERENCES Products,
  PRIMARY KEY (bug_id, product)
);
```

Query Antipatterns

```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

```
$dbHandle = new PDO('mysql:dbname=test');
$stmt = $dbHandle->prepare($sql);
$result = $stmt->fetchAll();
```

Physical Database Antipatterns

5. ID Required
6. Phantom Files
7. FLOAT Antipattern
8. ENUM Antipattern
9. Readable Passwords

ID Required

ID Required

- **Antipattern:** mandatory "id" column as primary key

```
CREATE TABLE bugs (
    id              BIGINT AUTO_INCREMENT
                    PRIMARY KEY,
    description VARCHAR(200),
    . . .
)
```

ID Required

- Even stranger: another candidate key already exists

```
CREATE TABLE bugs (
        id          BIGINT AUTO_INCREMENT
                    PRIMARY KEY,
        bug_id      VARCHAR(10) UNIQUE,
        description VARCHAR(200),
        . . .
)
```

ID Required

- Issues:
  - Natural keys
  - Compound keys
  - Duplicate rows
  - Obscure meaning
  - JOIN ... USING

ID Required

- Natural keys
  - Not auto-generated
  - Might not be integers
  - Have domain meaning

```
CREATE TABLE bugs (
    bug_id      VARCHAR(10) PRIMARY KEY,
    description VARCHAR(200),
    ...
)
```

ID Required

- Compound keys
  - More than one column
  - Might not be auto-generated
  - Might not be integers

intersection tables have compound keys

```
CREATE TABLE bugs_prods (
    bug_id          BIGINT NOT NULL REFERENCES bugs,
    product_id      BIGINT NOT NULL REFERENCES products,
    PRIMARY KEY (bug_id, product_id)
)
```

ID Required

- "id" allows duplicate rows

```
CREATE TABLE bugs_prods (
    id          BIGINT AUTO_INCREMENT
                PRIMARY KEY,
    bug_id      BIGINT NOT NULL,
    product_id  BIGINT NOT NULL
)
```

compound candidate key

ID Required

- “id” obscures meaning

```
SELECT *
FROM sp_bi_mt
WHERE id = 5678
```

“software project bugs & issues main table” of course!

ID Required

- "bug_id" clarifies meaning

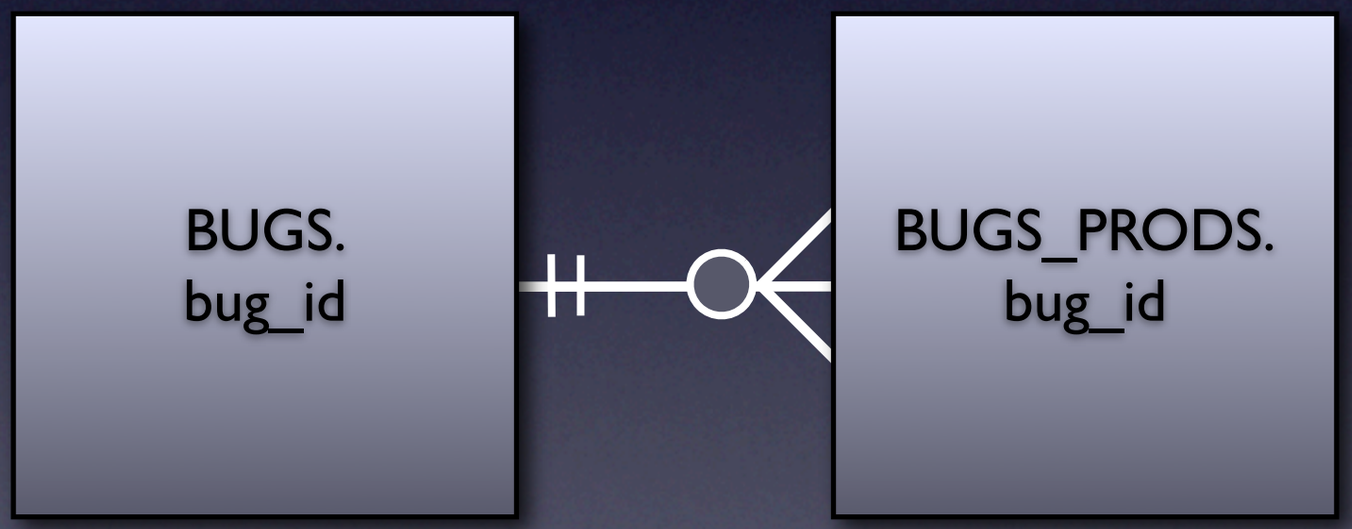
```
SELECT *
FROM sp_bi_mt
WHERE bug_id = 5678
```

must be something about bugs

ID Required

- “id” prevents JOIN...USING

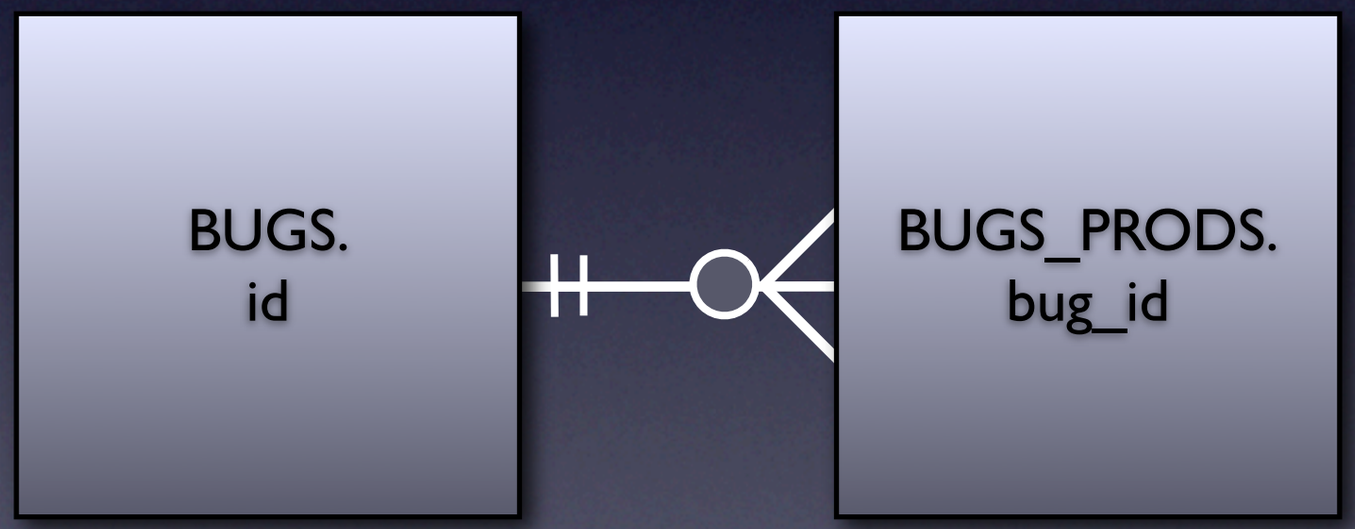
```
SELECT *
FROM bugs
     JOIN bugs_prods USING (bug_id)
```



ID Required

- “id” prevents JOIN...USING

```
SELECT *
FROM bugs b
     JOIN bugs_prods p ON (b.id = p.bug_id)
```



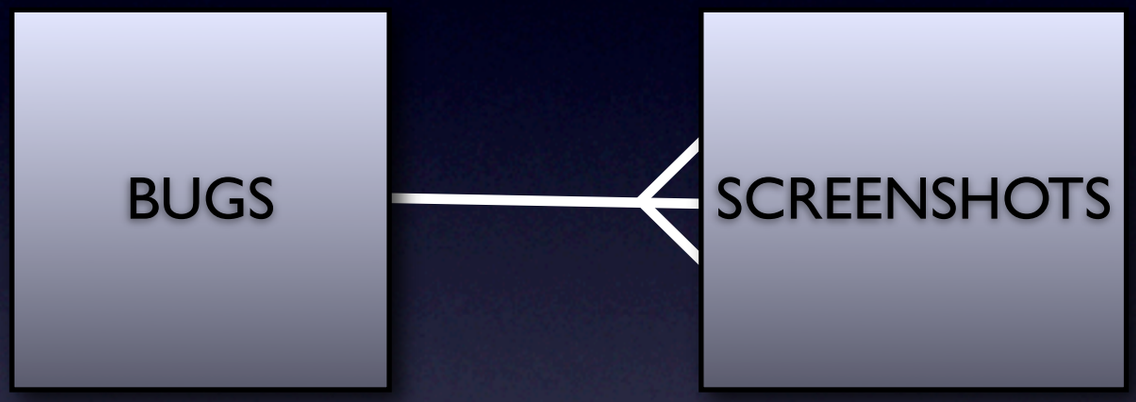
ID Required

- **Solution:**
  - Use natural keys when needed
  - Use compound keys when needed
  - Choose sensible names
  - Use same name in foreign keys

Phantom Files

Phantom Files

- **Objective:** store screenshot images



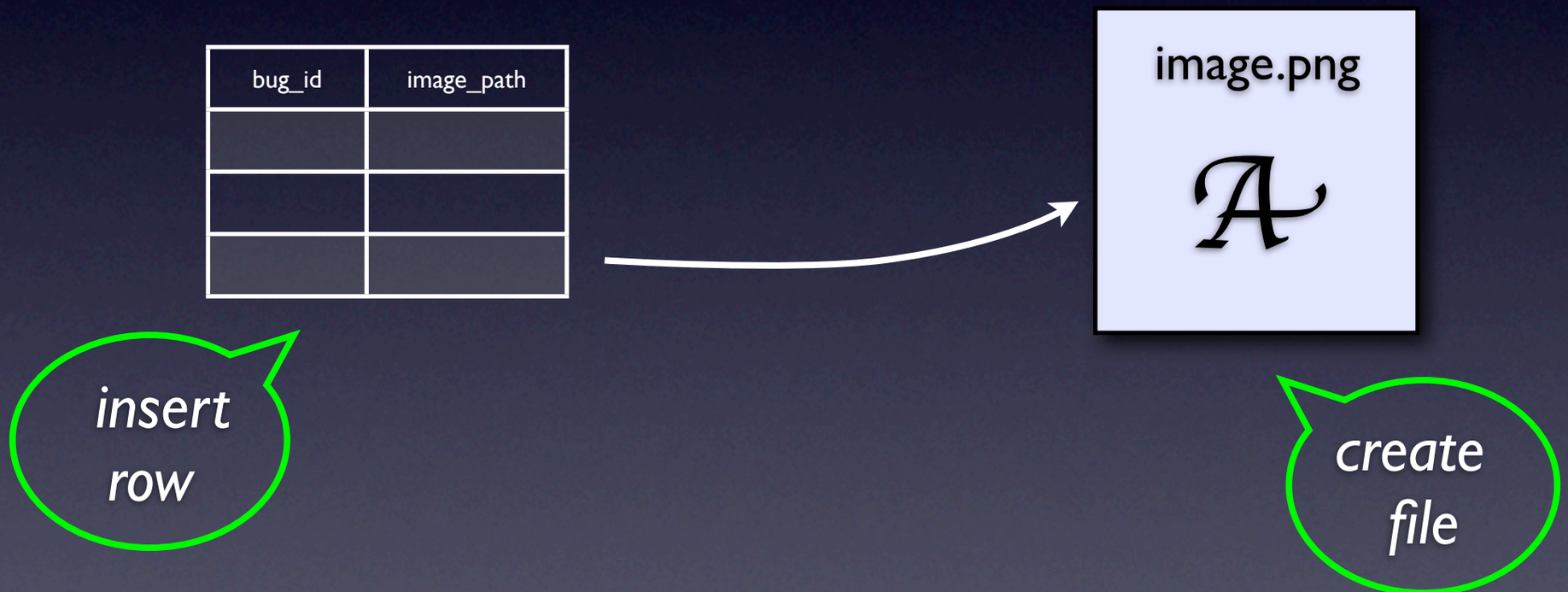
Phantom Files

- **Antipattern:** store path to image in database, image on filesystem


```
CREATE TABLE screenshots (
        bug_id          BIGINT REFERENCES bugs,
        image_path      VARCHAR(200),
        comment         VARCHAR(200)
)
```

Phantom Files

- Files don't obey DELETE



Phantom Files

- Files don't obey DELETE

bug_id	image_path

image.png

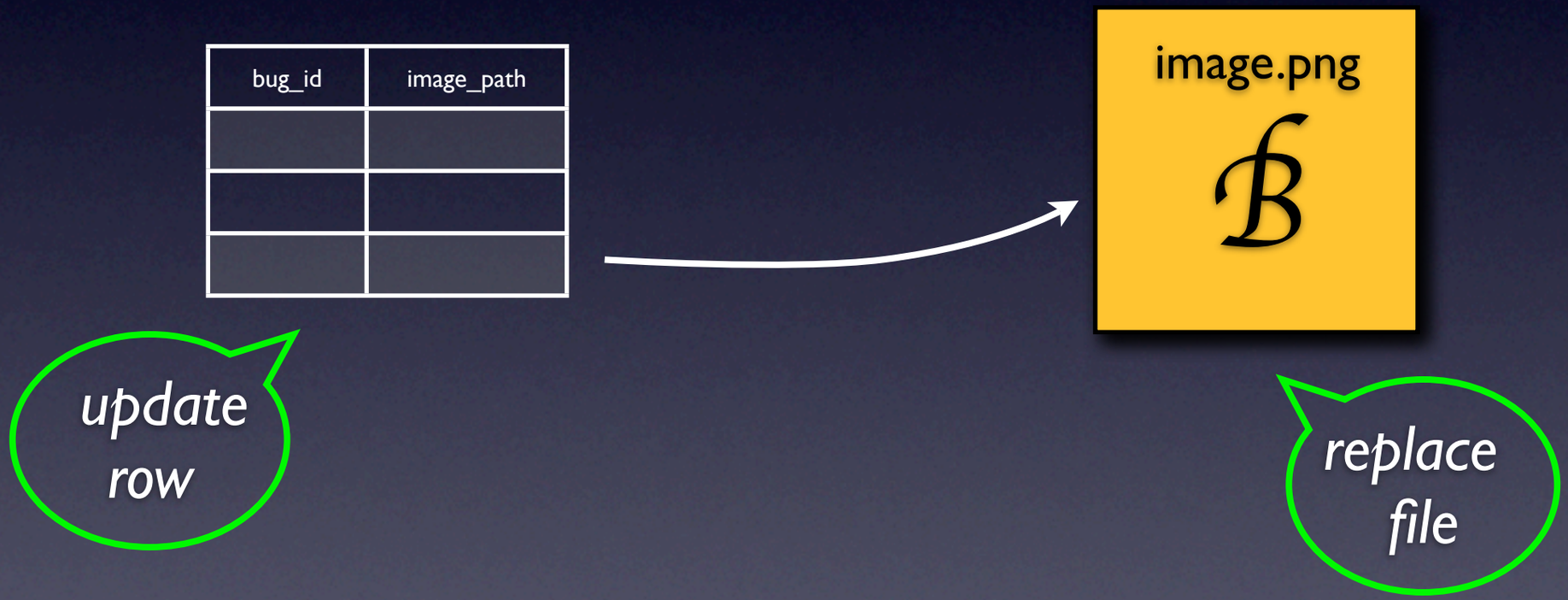
delete row

file is orphaned

Phantom Files

- Files don't obey UPDATE

1. Client #1 updates row, acquires row lock



Phantom Files

- Files don't obey UPDATE

2. Client #2 replaces image file, but not row

bug_id	image_path

image.png

C

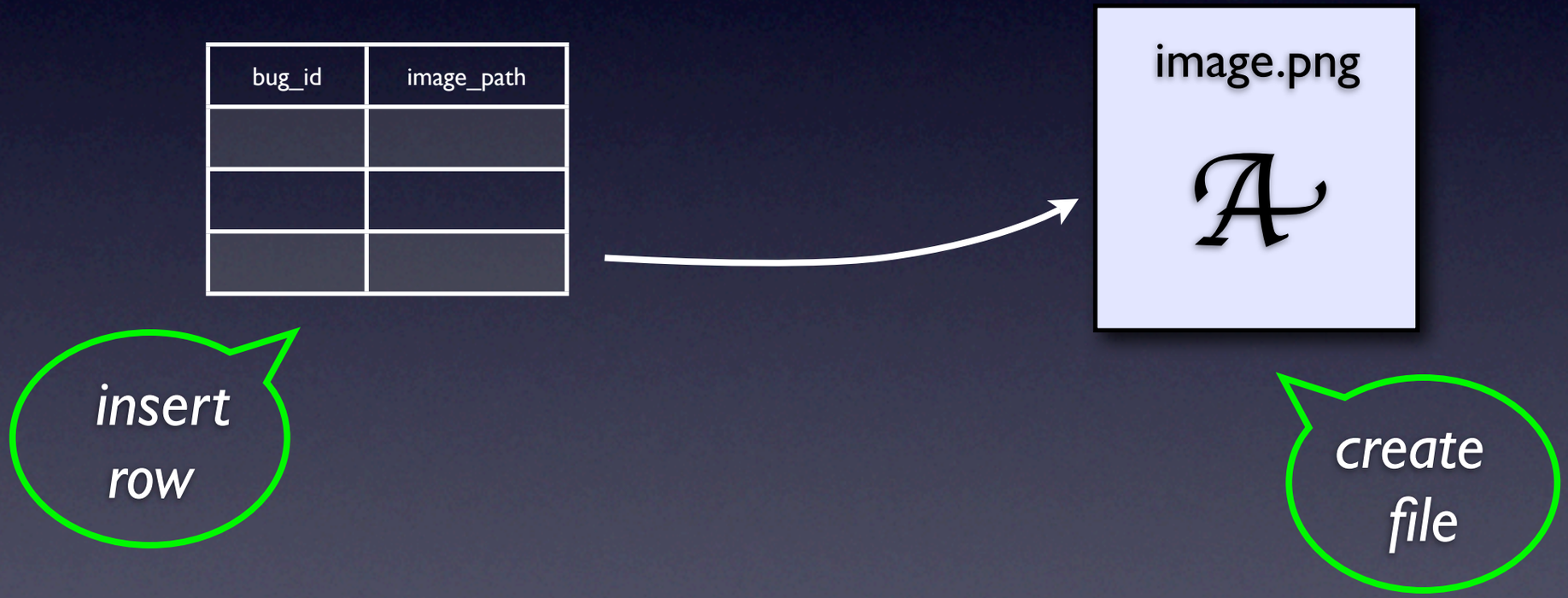
2nd update gets conflict error

replaces file anyway

Phantom Files

- Files don't obey ROLLBACK

1. Start transaction and INSERT



Phantom Files

- Files don't obey ROLLBACK

2. ROLLBACK

bug_id	image_path

image.png

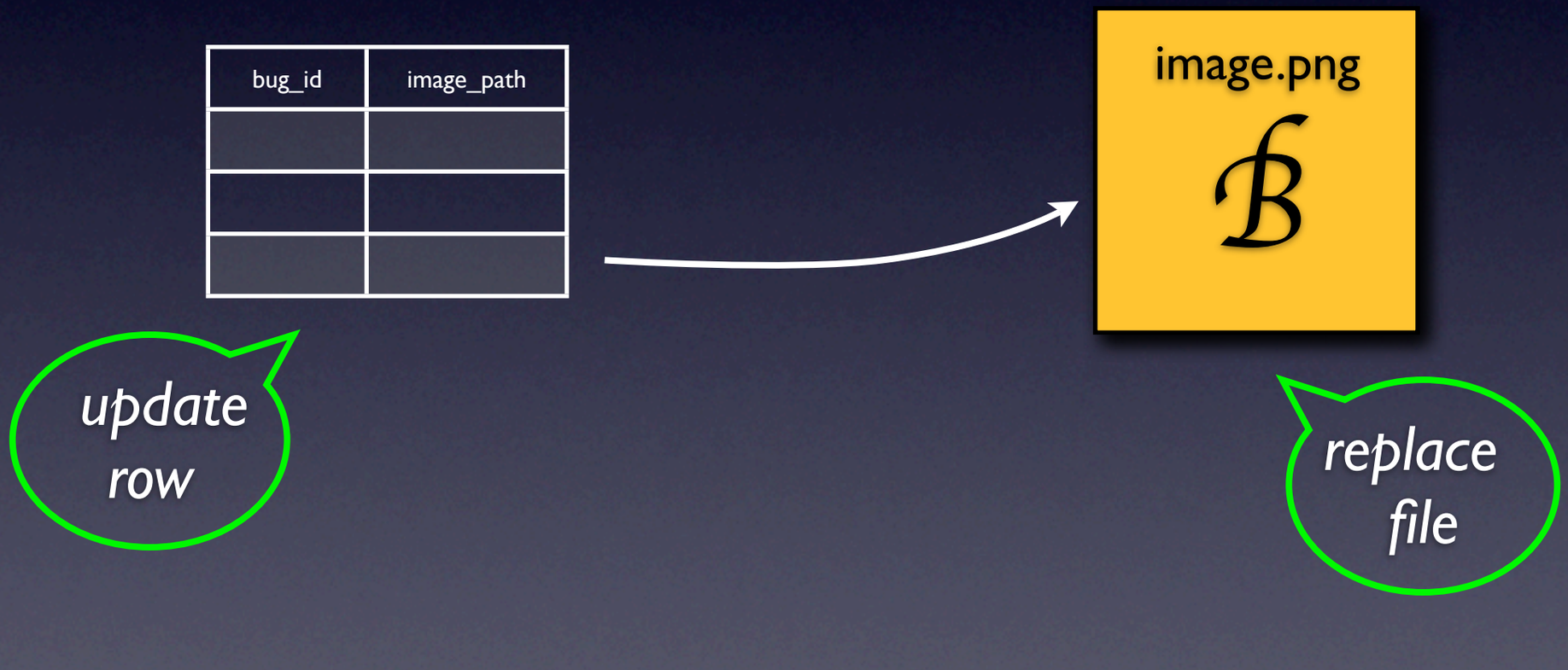
row is discarded

file is orphaned

Phantom Files

- Files don't obey ROLLBACK

1. Start transaction and UPDATE



Phantom Files

- Files don't obey ROLLBACK

2. ROLLBACK

bug_id	image_path

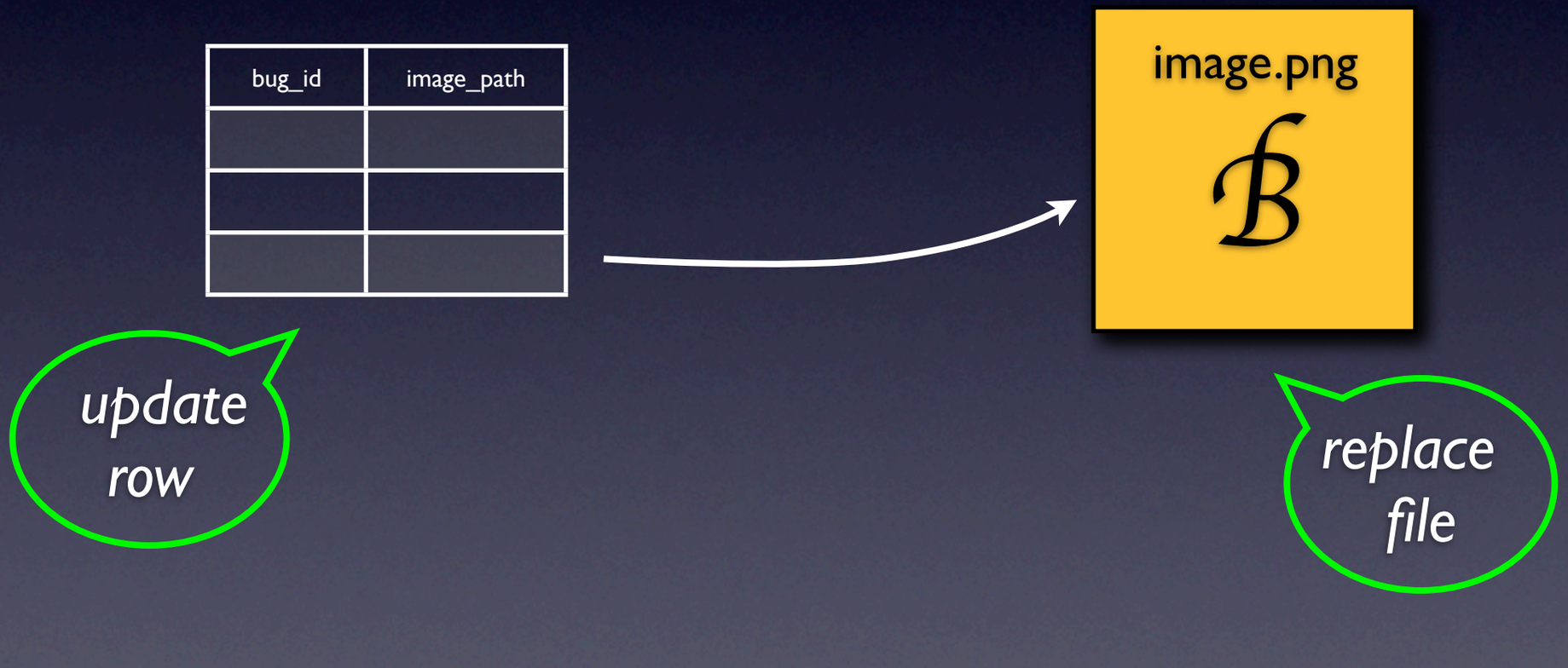
new row reverts to old

new file doesn't revert

Phantom Files

- Files don't obey transaction isolation

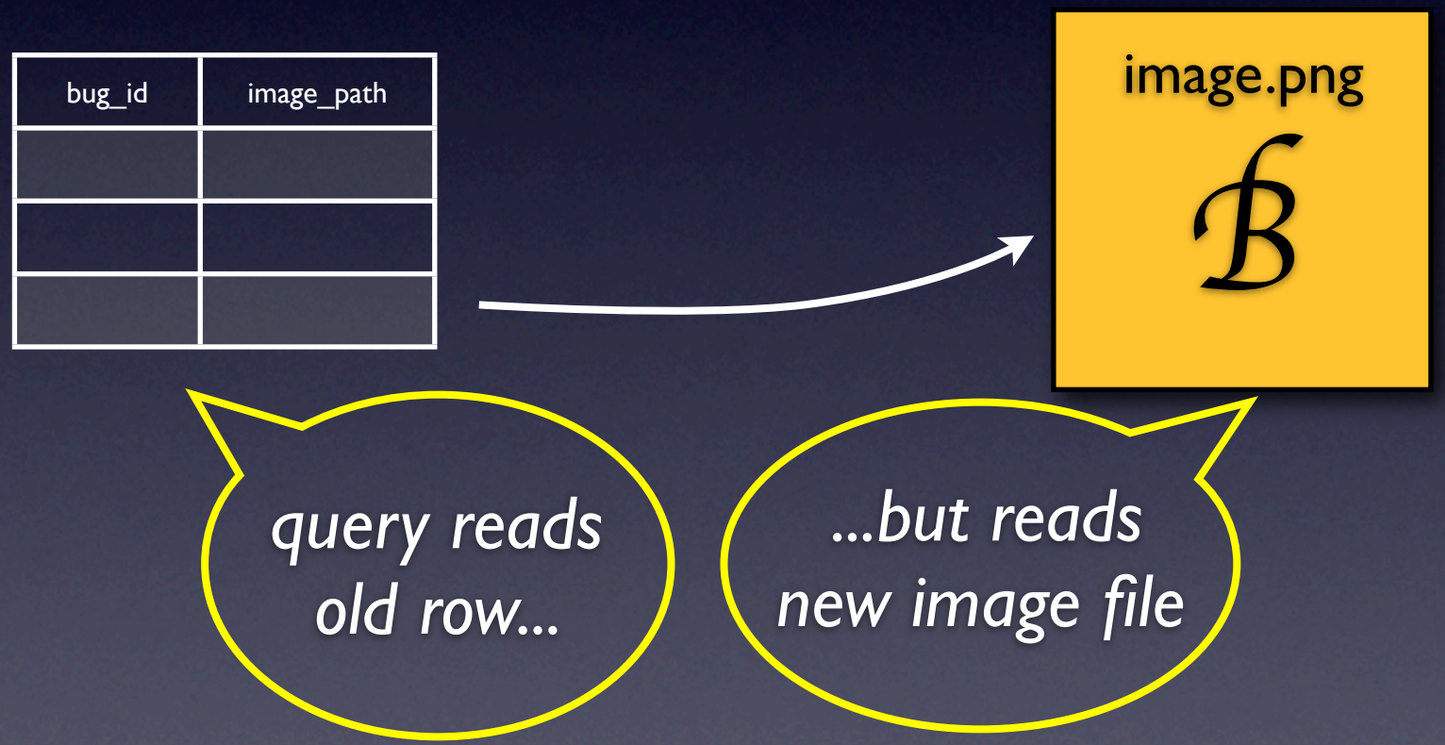
1. Client #1 starts transaction and UPDATE



Phantom Files

- Files don't obey transaction isolation

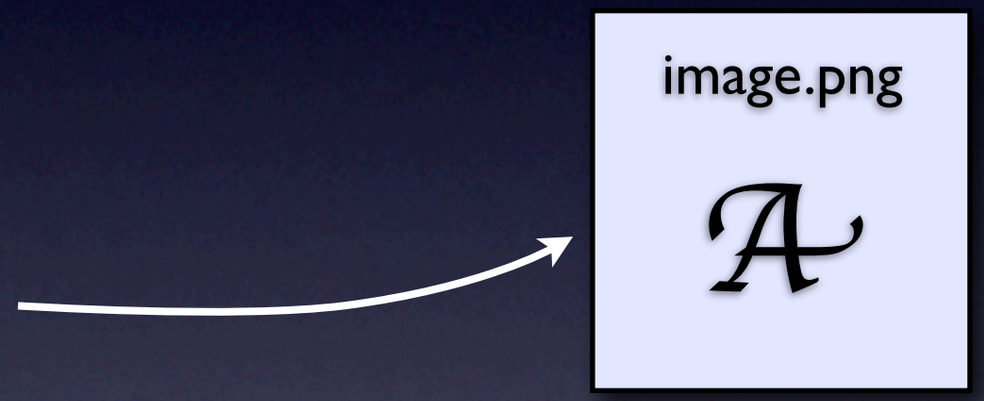
2. Client #2 queries before #1 COMMITs



Phantom Files

- Files don't obey database backup tools

bug_id	image_path

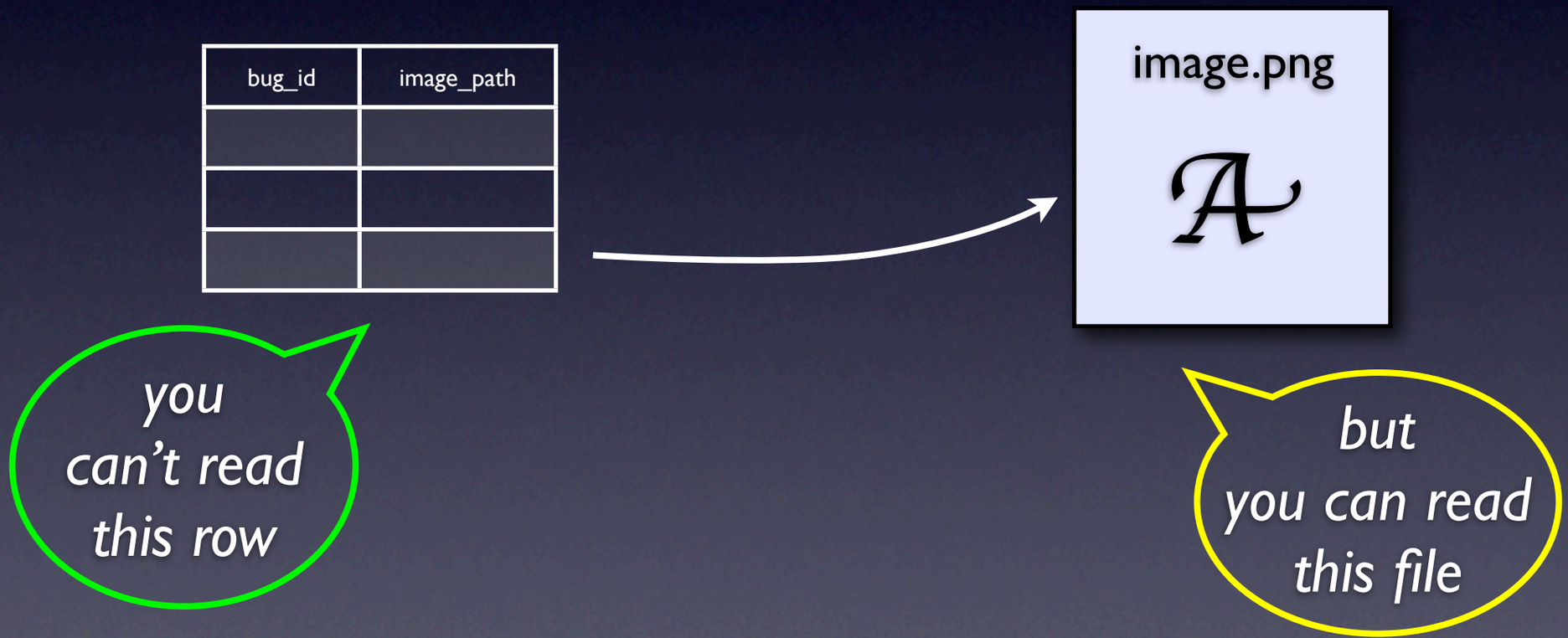


included in backup

excluded from backup

Phantom Files

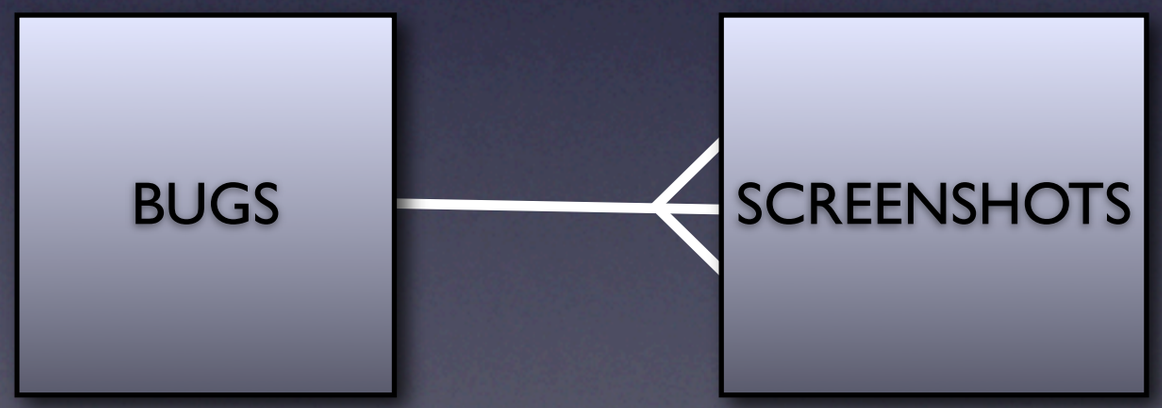
- Files don't obey SQL access privileges



Phantom Files

- **Solution:** consider storing images inside the database, in a BLOB column

```
CREATE TABLE screenshots (
        bug_id          BIGINT REFERENCES bugs,
        image_data      BLOB,
        comment         VARCHAR(200)
)
```



Phantom Files

- Legitimate advantages of filesystem storage:
  - Database is much leaner without images
  - Database backups are smaller, faster too
  - Easier to manage & preview images
- Make your own informed decision

FLOAT Antipattern

FLOAT Antipattern

- **Objective:** store real numbers exactly
  - Especially money
  - Work estimate hours

FLOAT Antipattern

- **Antipattern:** use FLOAT data type

```
ALTER TABLE bugs
    ADD COLUMN work_estimate_hrs FLOAT

INSERT INTO bugs (bug_id, work_estimate_hrs)
    VALUES (1234, 3.3)
```

FLOAT Antipattern

- FLOAT is inexact

```
SELECT work_estimate_hrs
FROM bugs WHERE bug_id = 1234
```

  - 3.3

```
SELECT work_estimate_hrs * 1000000000
FROM bugs WHERE bug_id = 1234
```

  - 3299999952.3163

FLOAT Antipattern

- Inexact decimals
- 1/3 + 1/3 + 1/3 = 1.0
- 0.333 + 0.333 + 0.333 = 0.999

assuming infinite precision

finite precision

FLOAT Antipattern

- IEEE 754 standard for representing floating-point numbers in base-2
- Some numbers round off, aren't stored exactly
- Comparisons to original value fail

```
SELECT * FROM bugs
WHERE work_estimate_hrs = 3.3
```

FLOAT Antipattern

- **Solution:** use NUMERIC data type

```
ALTER TABLE bugs
    ADD COLUMN work_estimate_hrs NUMERIC(9,2)

INSERT INTO bugs (bug_id, work_estimate_hrs)
    VALUES (1234, 3.3)

SELECT * FROM bugs
WHERE work_estimate_hrs = 3.3
```

comparison succeeds

ENUM Antipattern

ENUM Antipattern

- **Objective:** restrict a column's values to a fixed set of values

INSERT INTO bugs (status)
VALUES ('new')

OK

INSERT INTO bugs (status)
VALUES ('banana')

FAIL

ENUM Antipattern

- **Antipattern:** use ENUM data type, when the set of values may change

```
CREATE TABLE bugs (
        status        ENUM('new', 'open', 'fixed')
)
```

ENUM Antipattern

- Changing the set of values is a metadata alteration
- You must know the current set of values

```
ALTER TABLE bugs MODIFY COLUMN
    status ENUM('new', 'open', 'fixed', 'duplicate')
```

ENUM Antipattern

- Difficult to get a list of possible values

```
SELECT column_type
FROM information_schema.columns
WHERE table_schema = 'bugtracker_schema'
      AND table_name = 'bugs'
      AND column_name = 'status'
```

- You must parse the LONGTEXT value

"ENUM('new', 'open', 'fixed')"

ENUM Antipattern

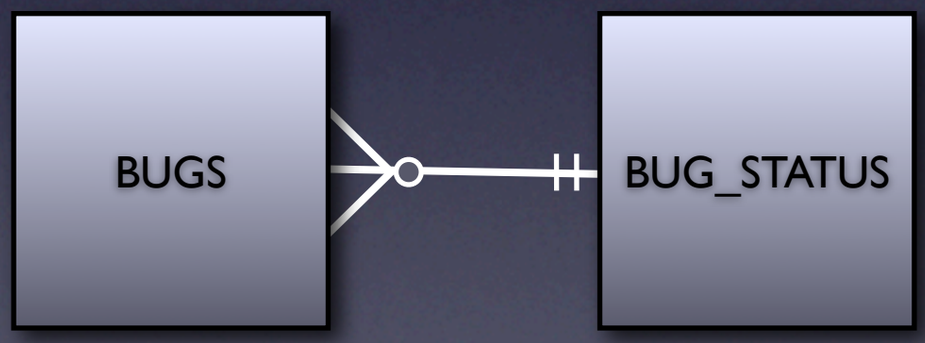
- **Solution:** use ENUM only if values are set in stone

```
CREATE TABLE bugs (
        . . .
        bug_type        ENUM('defect', 'feature')
        . . .
)
```

ENUM Antipattern

- Use a lookup table if values may change

```
CREATE TABLE bug_status (
        status  VARCHAR(10) PRIMARY KEY
)

INSERT INTO bug_status (status)
        VALUES ('new'), ('open'), ('fixed')
```



ENUM Antipattern

- Adding/removing a value is a data operation, not a metadata operation
- You don't need to know the current values

INSERT INTO bug_status (status) VALUES ('duplicate')

ENUM Antipattern

- Use an attribute to retire values, not DELETE

```
CREATE TABLE bug_status (
        status      VARCHAR(10) PRIMARY KEY,
        active      TINYINT NOT NULL DEFAULT 1
)

UPDATE bug_status
SET active = 0
WHERE status = 'duplicate'
```

Readable Passwords

Readable Passwords

- **Objective:**
  help users who forget their password

Readable Passwords

- **Antipattern:** store password in plain text (i.e. not encoded)

```
CREATE TABLE accounts (
        acct_name       VARCHAR(20) PRIMARY KEY,
        email           VARCHAR(100) NOT NULL,
        password        VARCHAR(30) NOT NULL
)

INSERT INTO accounts
        VALUES ('bill', 'bill@example.com', 'xyzzy')
```

Readable Passwords

- **Antipattern:** compare user input directly against stored password

```
SELECT (password = ‘xyzzy’) AS is_correct
FROM accounts
WHERE acct_name = ‘bill’
```

Readable Passwords

- **Antipattern:** send password in plain text in email to user upon request

From: daemon
To: bill@example.com
Subject: password request

Your account is “bill” and your password is “xyzzy”

Click the link below to log in.

. . .

Readable Passwords

- **Solution:** store password using a one-way message digest function

```
CREATE TABLE accounts (
        acct_name       VARCHAR(20) PRIMARY KEY,
        email           VARCHAR(100) NOT NULL,
        password_hash   CHAR(32) NOT NULL
)

INSERT INTO accounts (acct_name, password_hash)
        VALUES ('bill', MD5('xyzzy'))
```

Readable Passwords

- Good: compare hash of user input against stored hash

```
SELECT (a.password_hash = MD5('xyzzy'))
       AS is_correct
FROM accounts a
WHERE a.acct_name = 'bill'
```

Readable Passwords

- Better: concatentate a salt before using the hash function

```
CREATE TABLE accounts (
        . . .
        password_hash    CHAR(32) NOT NULL,
        salt             BINARY(4) NOT NULL
)

SELECT (a.password_hash = MD5(
        CONCAT('xyzzy', a.salt))) AS is_correct
FROM accounts a
WHERE a.acct_name = 'bill'
```

Readable Passwords

- Best: create hash of user input in your application, before sending to database

```
$salt =   SELECT salt FROM accounts
            WHERE acct_name = ‘bill’;

$hash = md5(‘xyzzy’ . $salt);

SELECT (a.password_hash = $hash) AS is_correct
FROM accounts a
WHERE a.acct_name = ‘bill’
```

Readable Passwords

- Reset password to a temporary random string, require user to change it

From: daemon
To: bill@example.com
Subject: password reset

Your password has been reset.

The password “p0trz3bie” will work for one hour.

Click the link below to log in and change your password.
. . .

Antipattern Categories

Logical Database Antipatterns

Physical Database Antipatterns

```
CREATE TABLE BugsProducts (
  bug_id   INTEGER REFERENCES Bugs,
  product  VARCHAR(100) REFERENCES Products,
  PRIMARY KEY (bug_id, product)
);
```

Query Antipatterns

```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

```
$dbHandle = new PDO('mysql:dbname=test');
$stmt = $dbHandle->prepare($sql);
$result = $stmt->fetchAll();
```

Query Antipatterns

10. Ambiguous GROUP BY
11. HAVING antipattern
12. Poor Man's Search Engine
13. Implicit columns in SELECT and INSERT

Ambiguous GROUP BY

Ambiguous GROUP BY

- **Objective**: perform grouping queries, and include some attributes in the result

```
SELECT prod_name, bug_id,
    MAX(created_date) as latest
FROM bugs
GROUP BY prod_name
```

Ambiguous GROUP BY

- **Antipattern:** *bug_id* isn't that of the latest per product

product_name	bug_id	created_date
Open RoundFile	1234	2007-12-19
Open RoundFile	2248	2008-04-01
Visual TurboBuilder	3456	2008-02-16
Visual TurboBuilder	4077	2008-02-10
ReConsider	5678	2008-01-01
ReConsider	8063	2007-11-09

product_name	bug_id	latest
Open RoundFile	1234	2008-04-01
Visual TurboBuilder	3456	2008-02-16
ReConsider	5678	2008-01-01

Ambiguous GROUP BY

- The **Single-Value Rule**: all columns in the select-list must be either:
  - Part of an aggregate expression
  - Referenced in the GROUP BY clause
  - A **functional dependency** of a column named in the GROUP BY clause

Ambiguous GROUP BY

- Functional dependency:
  - For a given *product_name*, there is guaranteed to be one value in a functionally dependent attribute

product_name	bug_id	created_date
Open RoundFile	1234	2007-12-19
Open RoundFile	2248	2008-04-01
Visual TurboBuilder	3456	2008-02-16
Visual TurboBuilder	4077	2008-02-10
ReConsider	5678	2008-01-01
ReConsider	8063	2007-11-09

multiple values per product name

bug_id is not functionally dependent

Ambiguous GROUP BY

- **Solution #1:** use only functionally dependent attributes in select-list

```
SELECT prod_name, ~~bug_id,~~
    MAX(created_date) as latest
FROM bugs
GROUP BY prod_name
```

product_name	latest
Open RoundFile	2008-04-01
Visual TurboBuilder	2008-02-16
ReConsider	2008-01-01

Ambiguous GROUP BY

- **Solution #2:** use GROUP_CONCAT() function in MySQL to get all values

```
SELECT prod_name,
    GROUP_CONCAT(bug_id) as bug_id_list,
    MAX(created_date) as latest
FROM bugs
GROUP BY prod_name
```

product_name	bug_id_list	latest
Open RoundFile	1234, 2248	2008-04-01
Visual TurboBuilder	3456, 4077	2008-02-16
ReConsider	5678, 8063	2008-01-01

Ambiguous GROUP BY

- **Solution #3:** use this OUTER JOIN query instead of GROUP BY

```
SELECT b.prod_name, b.bug_id,
    b.created_date AS latest
FROM bugs b LEFT OUTER JOIN bugs b2
    ON (b.prod_name = b2.prod_name AND
        b.created_date < b2.created_date)
WHERE b2.bug_id IS NULL
```

product_name	bug_id	latest
Open RoundFile	2248	2008-04-01
Visual TurboBuilder	3456	2008-02-16
ReConsider	5678	2008-01-01

HAVING Antipattern

HAVING Antipattern

- **Objective:** use column aliases in query criteria (the WHERE clause)

```
SELECT bug_id, YEAR(created_date) AS yr
FROM bugs
WHERE yr = 2008
```

HAVING Antipattern

- **Antipattern:** use a HAVING clause instead of WHERE clause

```
SELECT bug_id, YEAR(created_date) AS yr
FROM bugs
HAVING yr = 2008
```

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions
2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions

filter rows

2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions
2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

execute select-list for filtered rows

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions
2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

after expressions

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions
2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

reduce rows by groups

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions
2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

filter groups

HAVING Antipattern

- Sequence of query execution:

1. WHERE expressions
2. SELECT expressions
3. Define column aliases
4. GROUP BY
5. HAVING expressions
6. ORDER BY

postponing row filtering causes the SELECT expressions to execute for all rows

HAVING Antipattern

- Other functions are run on rows that will be filtered

```
SELECT bug_id,
       YEAR(created_date) AS yr,
       expensive_func(b.description)
FROM bugs
HAVING yr = 2008
```

HAVING Antipattern

- **Solution 1:** repeat the expression in the WHERE clause, instead of the column alias

```
SELECT bug_id,
       YEAR(created_date) AS yr,
       expensive_func(description)
FROM bugs
WHERE YEAR(created_date) = 2008
```

HAVING Antipattern

- **Solution 2:** use a derived table

```
SELECT b.bug_id, b.yr,
        expensive_func(b.description)
FROM
        (SELECT bug_id, description,
            YEAR(created_date) AS yr
        FROM bugs) AS b
WHERE b.yr = 2008
```

Poor Man's Search Engine

Poor Man's Search Engine

- **Objective:** search for specific words in larger text

```
SELECT * FROM bugs
WHERE MATCH(description) AGAINST ('crash')
```

Poor Man's Search Engine

- **Antipattern:** use LIKE with wildcards

```
SELECT * FROM bugs
WHERE description LIKE '%crash%'
```

- Or regular expressions

```
SELECT * FROM bugs
WHERE description RLIKE 'crash'
```

Poor Man's Search Engine

- Performance issues
- Indexes don't benefit from substring searches
- String-matching + full table scan = expensive

Poor Man's Search Engine

- Telephone book analogy:

```
CREATE TABLE telephone_book (
    full_name       VARCHAR(40),
    KEY             (full_name)
)

INSERT INTO telephone_book VALUES
    ('Riddle, Thomas'),
    ('Thomas, Dean')
```

Poor Man's Search Engine

- Search for all with last name "Thomas"

```
SELECT * FROM telephone_book
WHERE full_name LIKE 'Thomas%'
```

uses index

- Search for all with first name "Thomas"

```
SELECT * FROM telephone_book
WHERE full_name LIKE '%Thomas'
```

doesn't use index

Poor Man's Search Engine

- Accuracy issues
  - Substrings find irrelevant or false matches
  - LIKE '%one%' matches words 'one', 'money', 'prone', etc.

Poor Man's Search Engine

- **Solution:** use a real full-text search engine
  - MySQL FULLTEXT index (MyISAM only)
  - http://lucene.apache.org/
  - http://www.sphinxsearch.com/

Implicit Columns

Implicit Columns

- **Objective:** write shorter, more general SQL queries

```
SELECT *
FROM bugs
```

wildcard column list

```
INSERT INTO bugs (status, priority, description)
VALUES ( . . . )
```

implicit column list

Implicit Columns

- Difficult to detect errors
- Add/remove a column
- Reposition a column
- Rename a column

Implicit Columns

- SQL has no wildcard syntax for "all but one"

```
SELECT * EXCEPT description
FROM bugs
```

imaginary syntax; doesn't exist

Implicit Columns

- **Solution:** always name columns explicitly

```
SELECT bug_id, status, priority, description, . . .
FROM bugs

INSERT INTO bugs
      (status, priority, description, . . .)
VALUES ( . . . )
```

Implicit Columns

- Get informative errors immediately
  - if you use a removed column
  - if you use a renamed column
- Specify the order of columns in query
  - even if columns are repositioned

Antipattern Categories

Logical Database Antipatterns

Physical Database Antipatterns

```
CREATE TABLE BugsProducts (
  bug_id   INTEGER REFERENCES Bugs,
  product  VARCHAR(100) REFERENCES Products,
  PRIMARY KEY (bug_id, product)
);
```

Query Antipatterns

```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

```
$dbHandle = new PDO('mysql:dbname=test');
$stmt = $dbHandle->prepare($sql);
$result = $stmt->fetchAll();
```

Application Antipatterns

14. User-supplied SQL
15. SQL Injection
16. Parameter Façade
17. Pseudokey Neat Freak
18. Session Coupling
19. Phantom Side Effects

User-Supplied SQL

User-supplied SQL

- **Objective:** satisfy demand for flexibility to query database

User-supplied SQL

- **Antipattern:**
  - Let users write their own SQL expressions
  - Run user-supplied SQL verbatim

User-supplied SQL

- Crash database server with one SELECT:

```
SELECT *
  FROM bugs JOIN bugs  JOIN bugs
       JOIN bugs JOIN bugs
```

- 100 bugs = 10,000,000,000 row result set

User-supplied SQL

- Filesort uses temporary disk space

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

User-supplied SQL

- **Solution:** users should specify values, but not code
- You write SQL queries with parameter placeholders

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

plug in user's value here

SQL Injection

SQL Injection

- **Antipattern:** interpolating untrusted data into SQL expressions

http://bug-server/?id=1234

```
<?php
$id = $_GET[‘id’];
$sql = “SELECT * FROM bugs
        WHERE bug_id = $id”;
mysql_query($sql);
```

SQL Injection

- Exposes your application to mischief

http://bug-server/?id=1234 or 1=1

SELECT * FROM bugs
WHERE bug_id = 1234 or 1=1

SQL Injection

- Exposes your application to mischief

http://bug-server/set-passwd?user=bill' or 'x'='x

```
UPDATE accounts
SET password_hash = MD5($pass)
WHERE account_name = 'bill' or 'x'='x'
```

SQL Injection

- **Solution #1:** filter input

```
<?php
$id = intval( $_GET[‘id’] );
$sql = “SELECT * FROM bugs
        WHERE bug_id = $id”;
mysql_query($sql);
```

- For strings, use *mysql_real_escape_string()*.

SQL Injection

- **Solution #2:** parameterize query values

```
<?php
$query = "SELECT * FROM bugs
      WHERE bug_id = ? ";
$stmt = mysqli_prepare($query);
$stmt->bind_param('i', $_GET['id']);
$stmt->execute();
```

parameter placeholder

Parameter Façade

Parameter Façade

- **Objective:** include application variables in SQL statements

```
SELECT * FROM bugs
WHERE bug_id = $id
```

Parameter Façade

- **Antipatterns:**
  - Trying to use parameters to change syntax
  - Interpolating variables for simple values

Parameter Façade

- Interpolation can modify syntax

$list = '1234, 3456, 5678'

```
SELECT * FROM bugs
WHERE bug_id IN ( $list )
```

```
SELECT * FROM bugs
WHERE bug_id IN ( 1234, 3456, 5678 )
```

three values separated by commas

Parameter Façade

- A parameter is always a single value

$list = '1234, 3456, 5678'

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

EXECUTE USING $list
```

one string value

```
SELECT * FROM bugs
WHERE bug_id IN ('1234, 3456, 5678' )
```

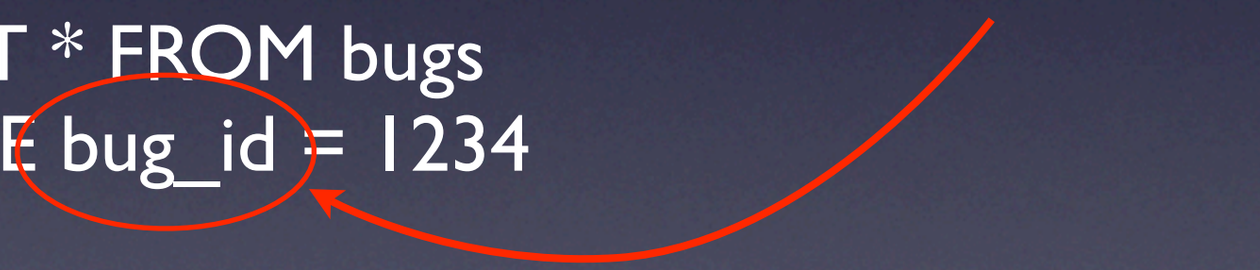
Parameter Façade

- Interpolation can specify identifiers

$column = 'bug_id'

```
SELECT * FROM bugs
WHERE $column = 1234
```

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



Parameter Façade

- A parameter is always a single value

```
$column = ‘bug_id’;
SELECT * FROM bugs
WHERE  ? = 1234

EXECUTE USING $column

SELECT * FROM bugs
WHERE ‘bug_id’ = 1234
```

one string value

Parameter Façade

- Interpolation risks SQL injection

$id = '1234 or 1=1'

```
SELECT * FROM bugs
WHERE bug_id = $id

SELECT * FROM bugs
WHERE bug_id = 1234 or 1=1
```

logical expression

Parameter Façade

- A parameter is always a single value

  $id = '1234 or 1=1'

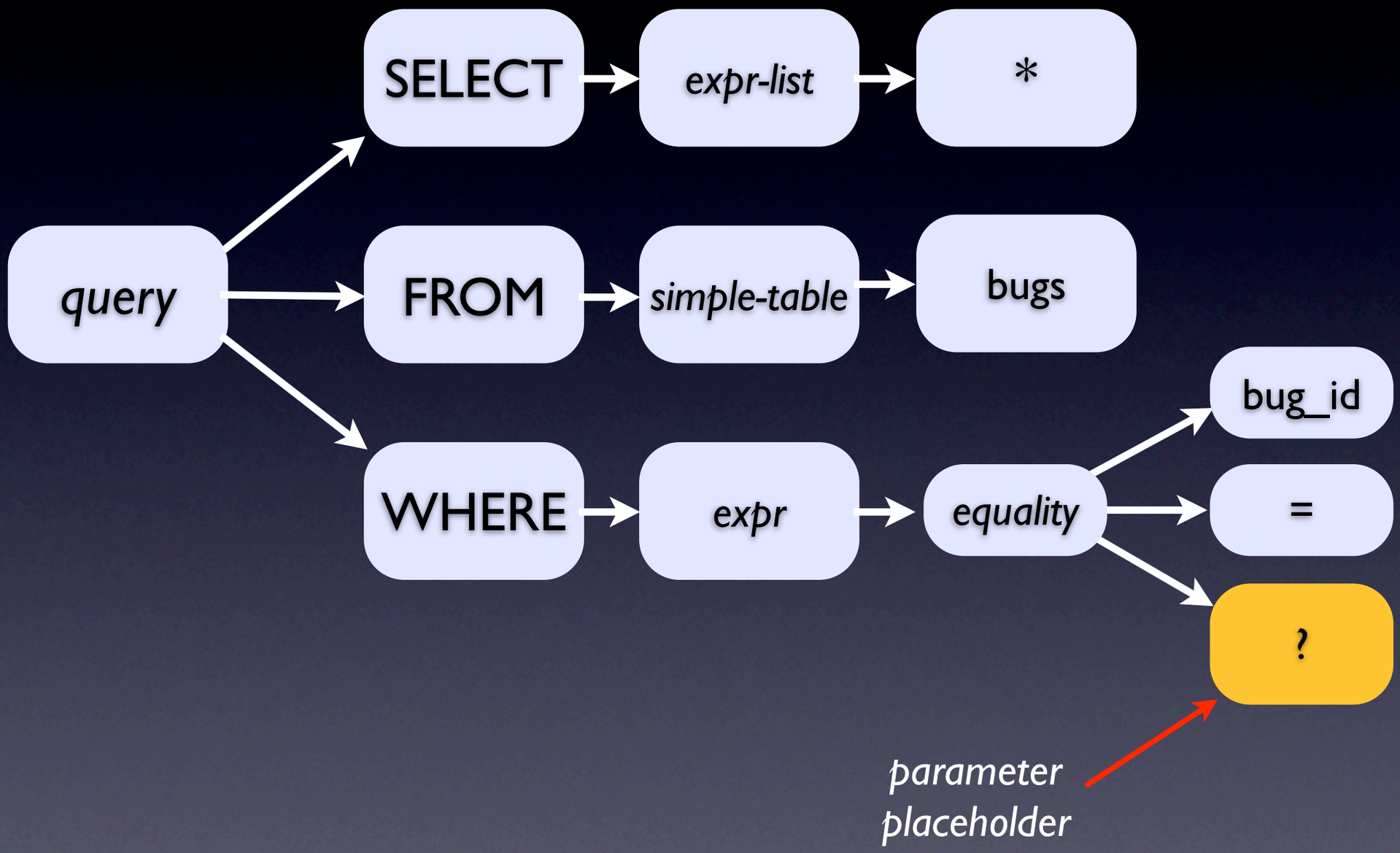
```
SELECT * FROM bugs
WHERE bug_id = ?

EXECUTE USING $id

SELECT * FROM bugs
WHERE bug_id = '1234 or 1=1'
```

one string value

Parameter Façade

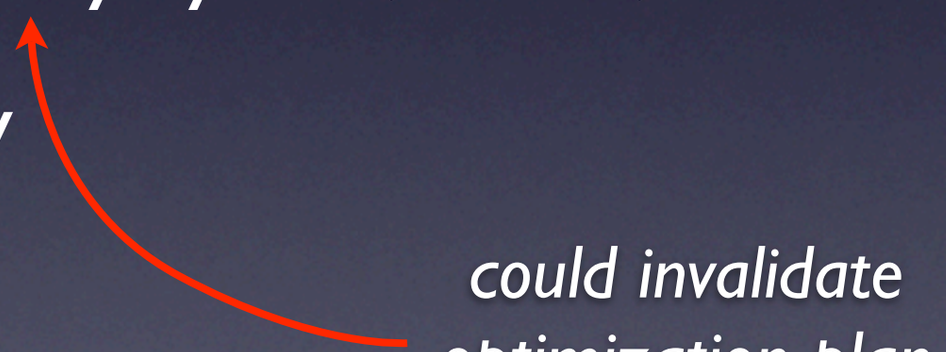
- Preparing a SQL statement
- Parses SQL syntax
- Optimizes execution plan
- Retains parameter placeholders

Parameter Façade

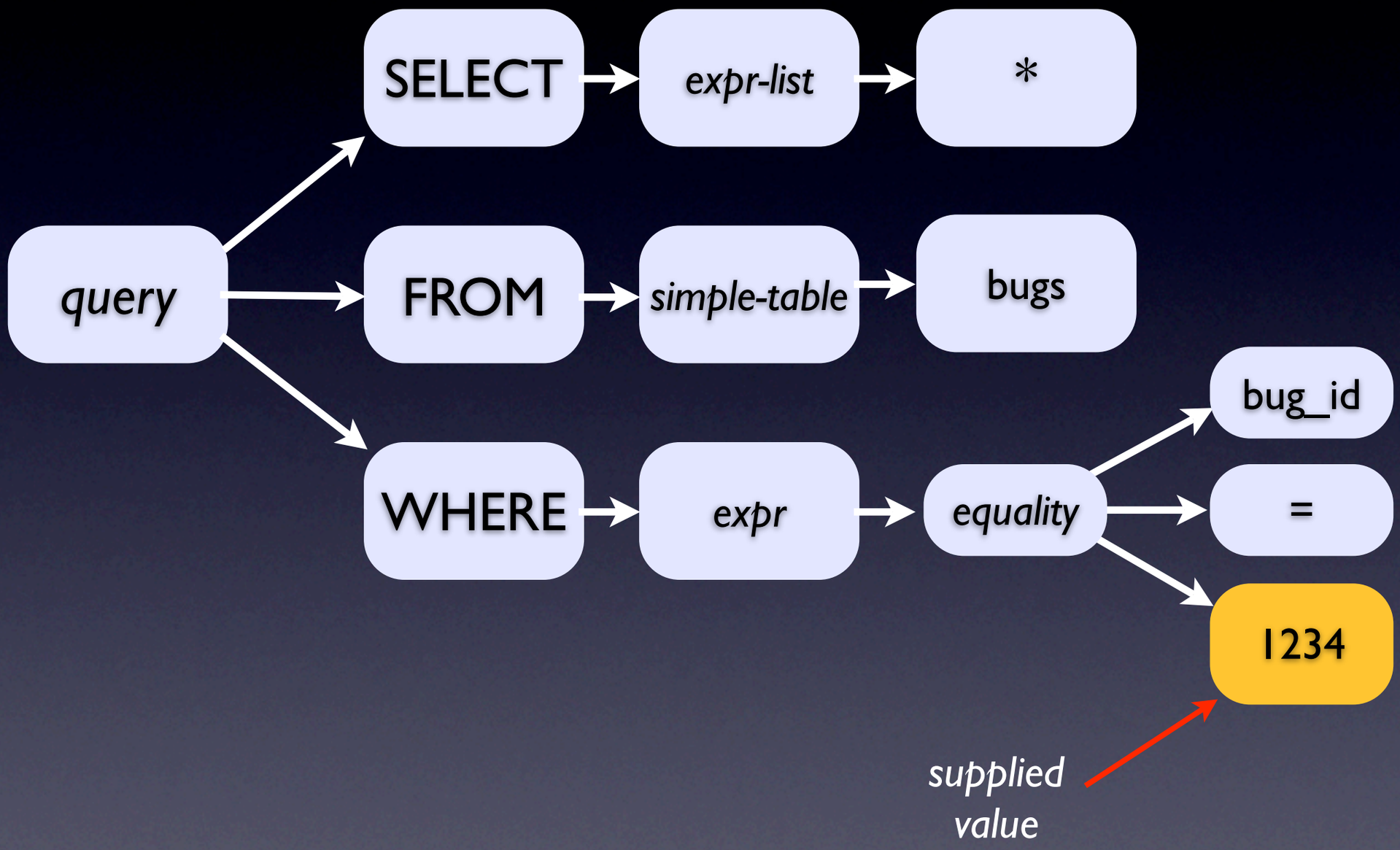


Parameter Façade

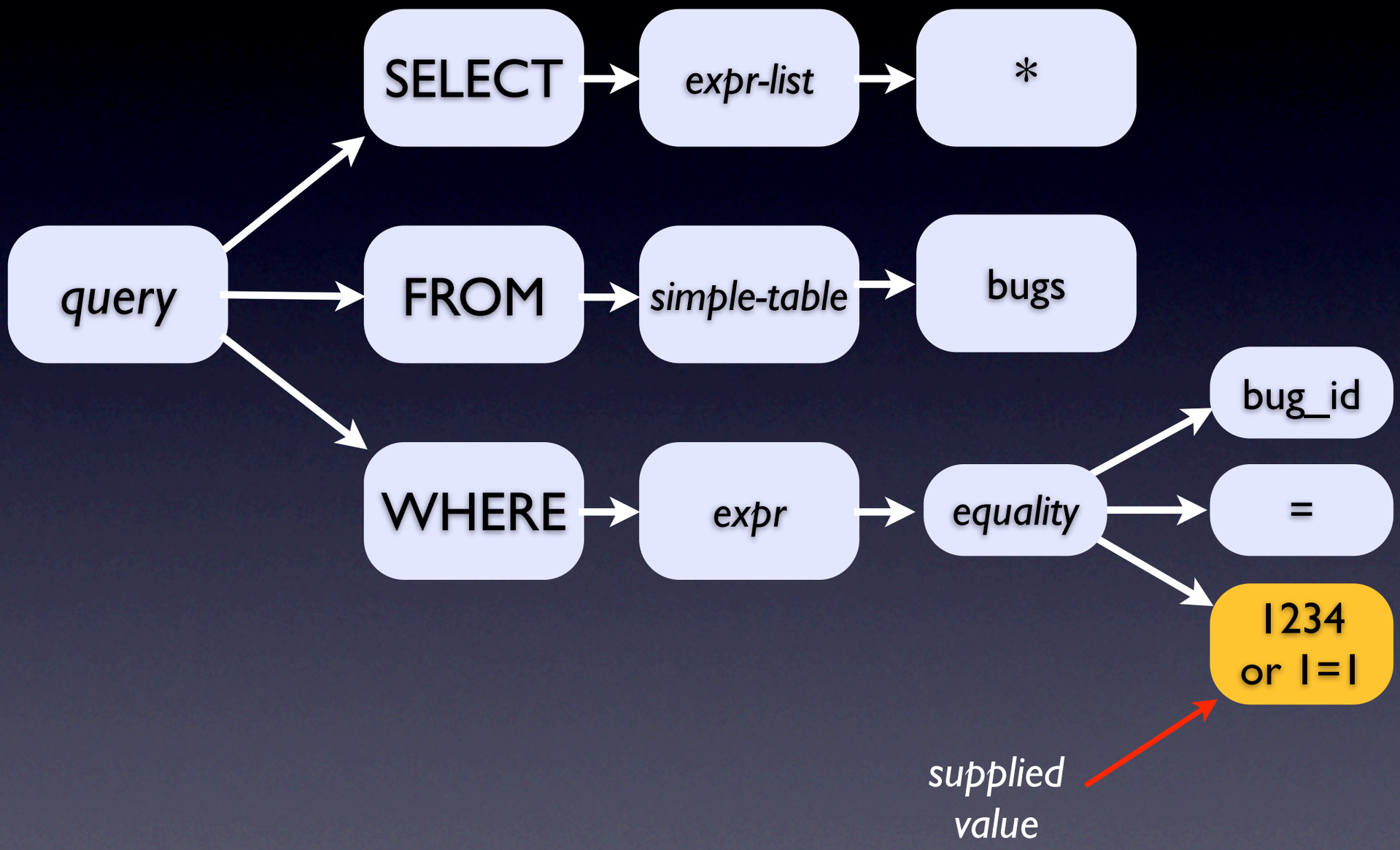
- Executing a prepared statement
- Combines a supplied value for each parameter
- *Doesn't* modify syntax, tables, or columns
- Runs query



Parameter Façade



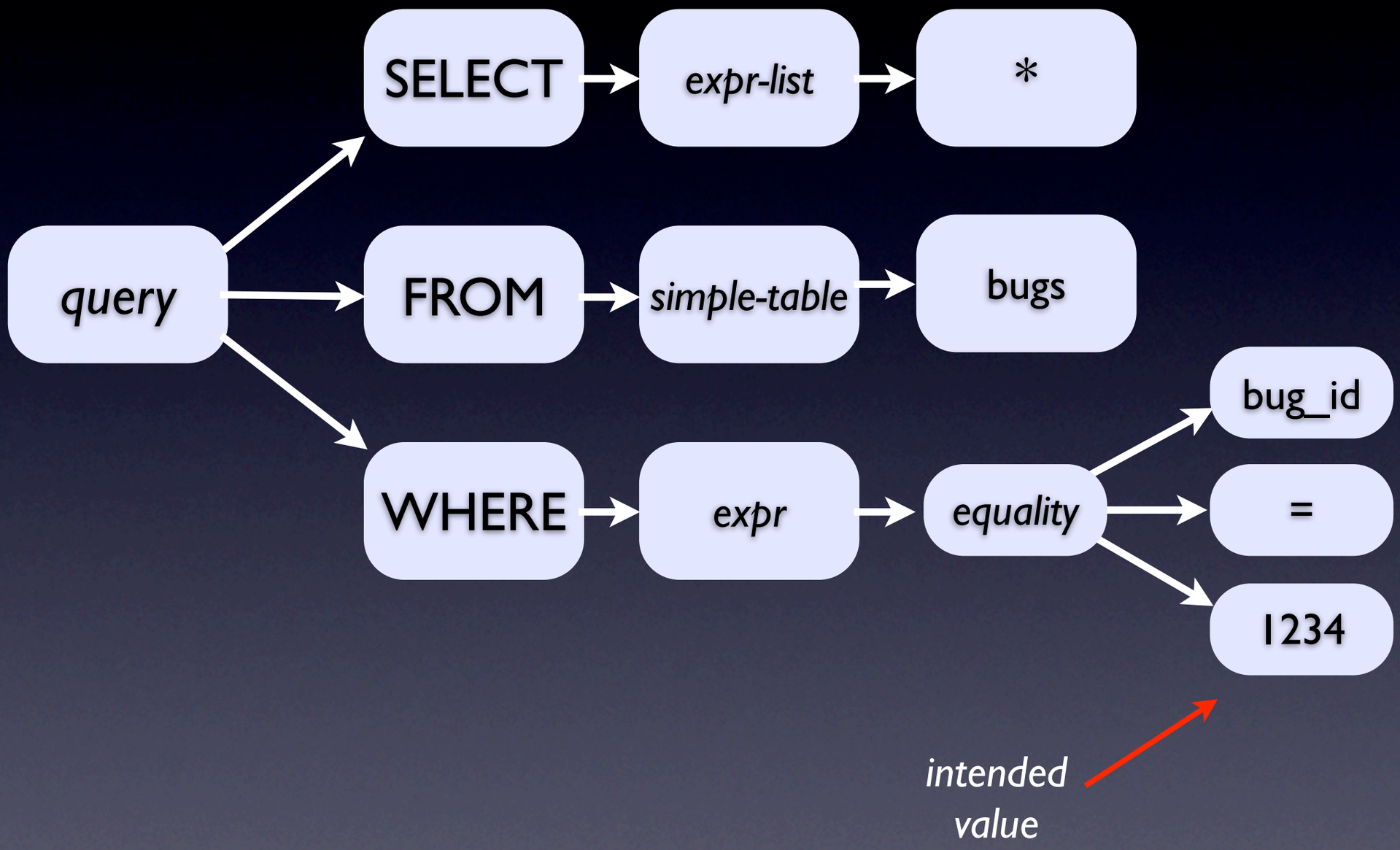
Parameter Façade



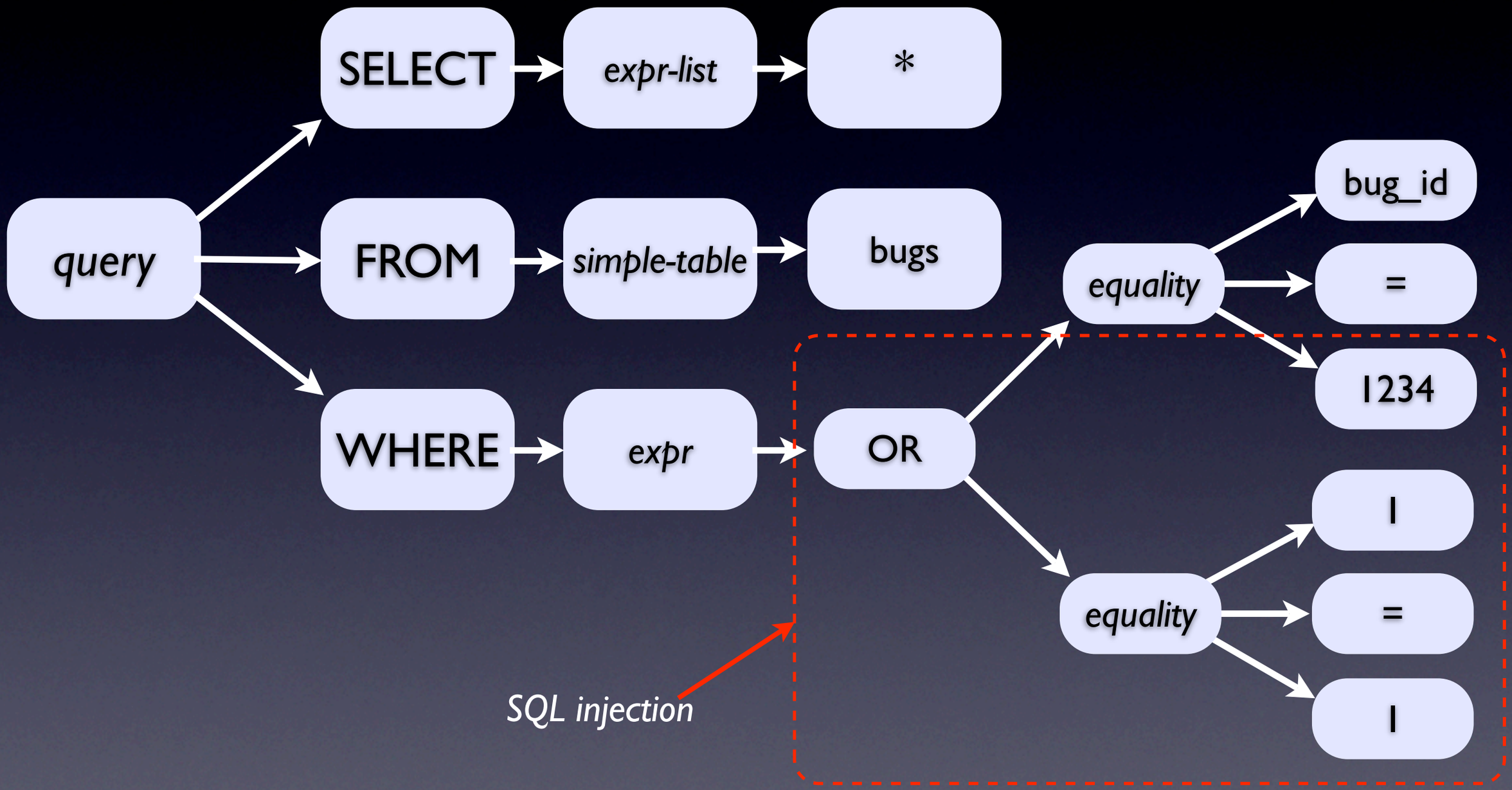
Parameter Façade

- Interpolating into a query string
- Occurs in the application, before SQL is parsed
- Database server can't tell what part is dynamic

Parameter Façade



Parameter Façade



Parameter Façade

- The Bottom Line:
  - Interpolation may change the shape of the tree
  - Parameters cannot change the tree
  - Parameter nodes may only be values

Parameter Façade

- Example: IN predicate

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

may supply only one value

```
SELECT * FROM bugs
WHERE bug_id IN ( ?, ?, ?, ? )
```

must supply exactly four values

Parameter Façade

Scenario	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 (?, ?, ?)
column name	'bug_id'	SELECT * FROM bugs WHERE $column = 1234	*NO*
table name	'bugs'	SELECT * FROM $table WHERE bug_id = 1234	*NO*
other syntax	'bug_id = 1234'	SELECT * FROM bugs WHERE $expr	*NO*

Parameter Façade

- **Solution:**
  - Use parameters only for individual values
  - Use interpolation for dynamic SQL syntax
  - Be careful to prevent SQL injection

Pseudokey Neat Freak

Pseudokey Neat Freak

- **Objective:** address the discomfort over the presence of "gaps" in the primary key

bug_id	bug_status	prod_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
5	DUPLICATE	ReConsider

Pseudokey Neat Freak

- **AntiPattern #1:**
  changing values to close the gaps:

bug_id	bug_status	prod_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

update 5 to 3

Pseudokey Neat Freak

- **AntiPattern #2:**
  recycling primary key values

bug_id	bug_status	prod_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
5	DUPLICATE	ReConsider
3	NEW	Visual TurboBuilder

Pseudokey Neat Freak

- Finding an unused value is complex:

```
SELECT b1.bug_id + 1
FROM bugs b1
LEFT JOIN bugs b2
       ON (b1.bug_id + 1 = b2.bug_id)
WHERE b2.bug_id IS NULL
ORDER BY b1.bug_id LIMIT 1
```

Pseudokey Neat Freak

- Finding an unused value: concurrency
  - Two clients can find the same value
  - Same problem as using MAX(bug_id)+1 to generate key values
  - Resolved only with a table lock

Pseudokey Neat Freak

- Reusing primary key values exposes data

bug_id	description

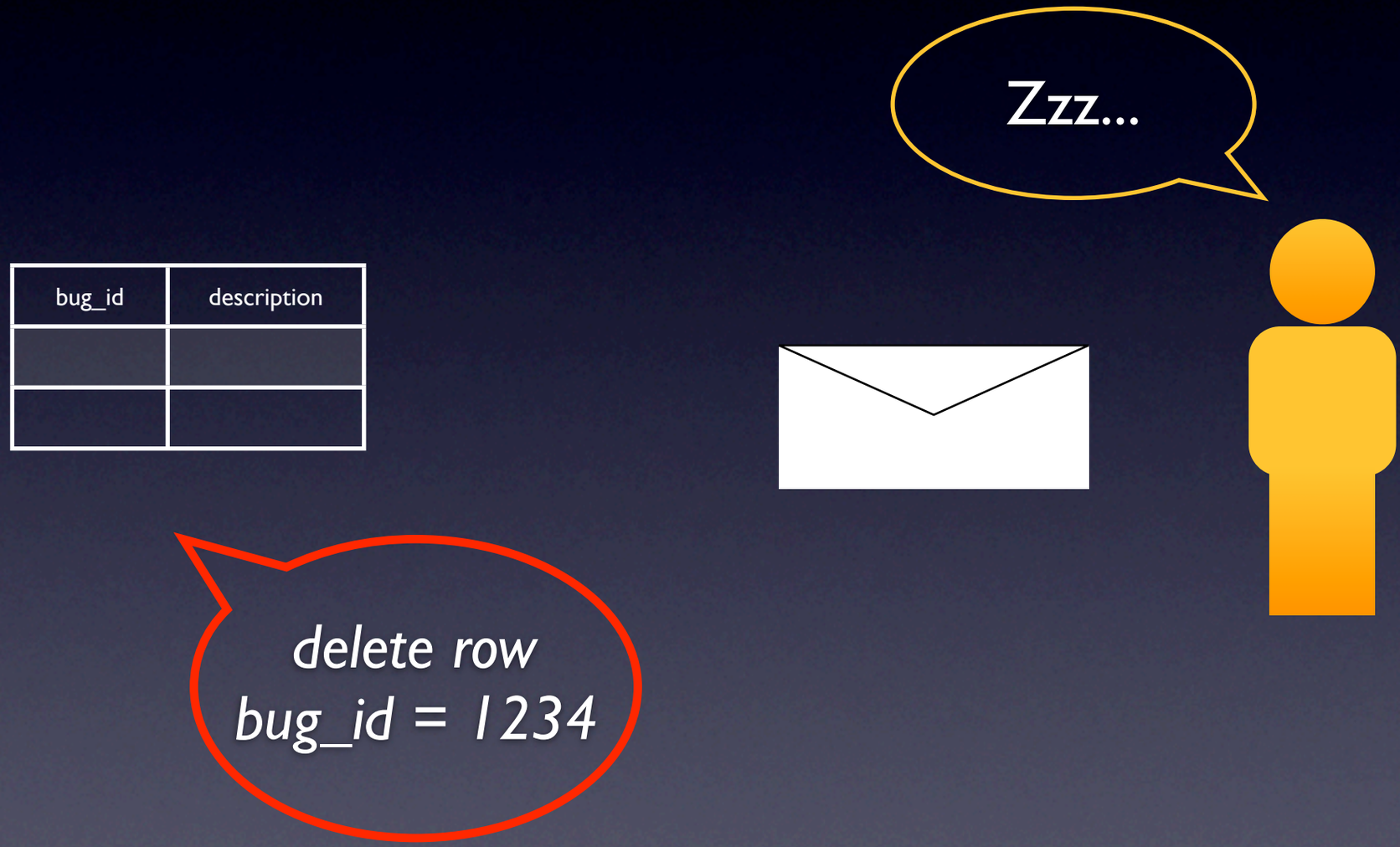
insert row
bug_id = 1234

Pseudokey Neat Freak

- Reusing primary key values exposes data

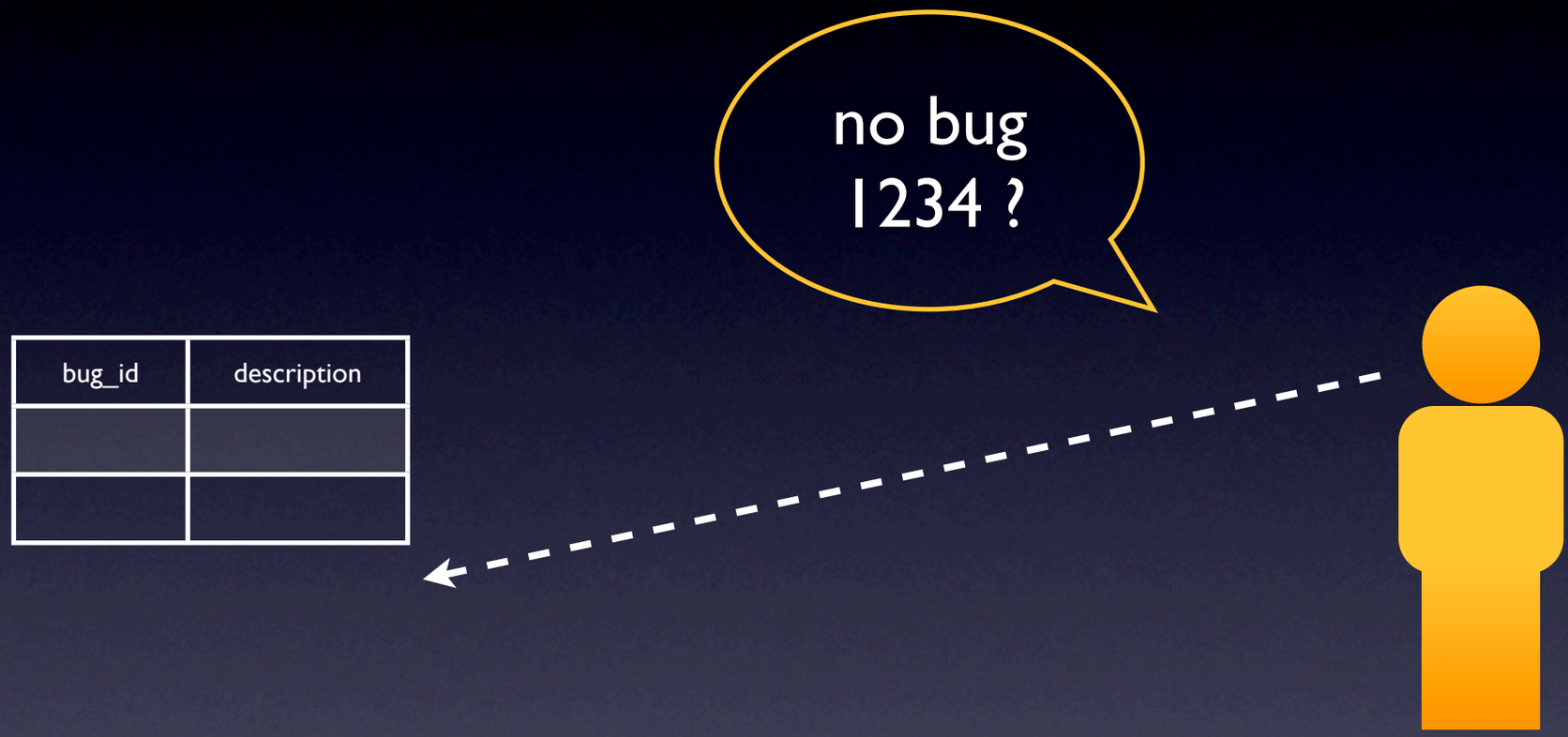
Pseudokey Neat Freak

- Reusing primary key values exposes data



Pseudokey Neat Freak

- Reusing primary key values exposes data



Pseudokey Neat Freak

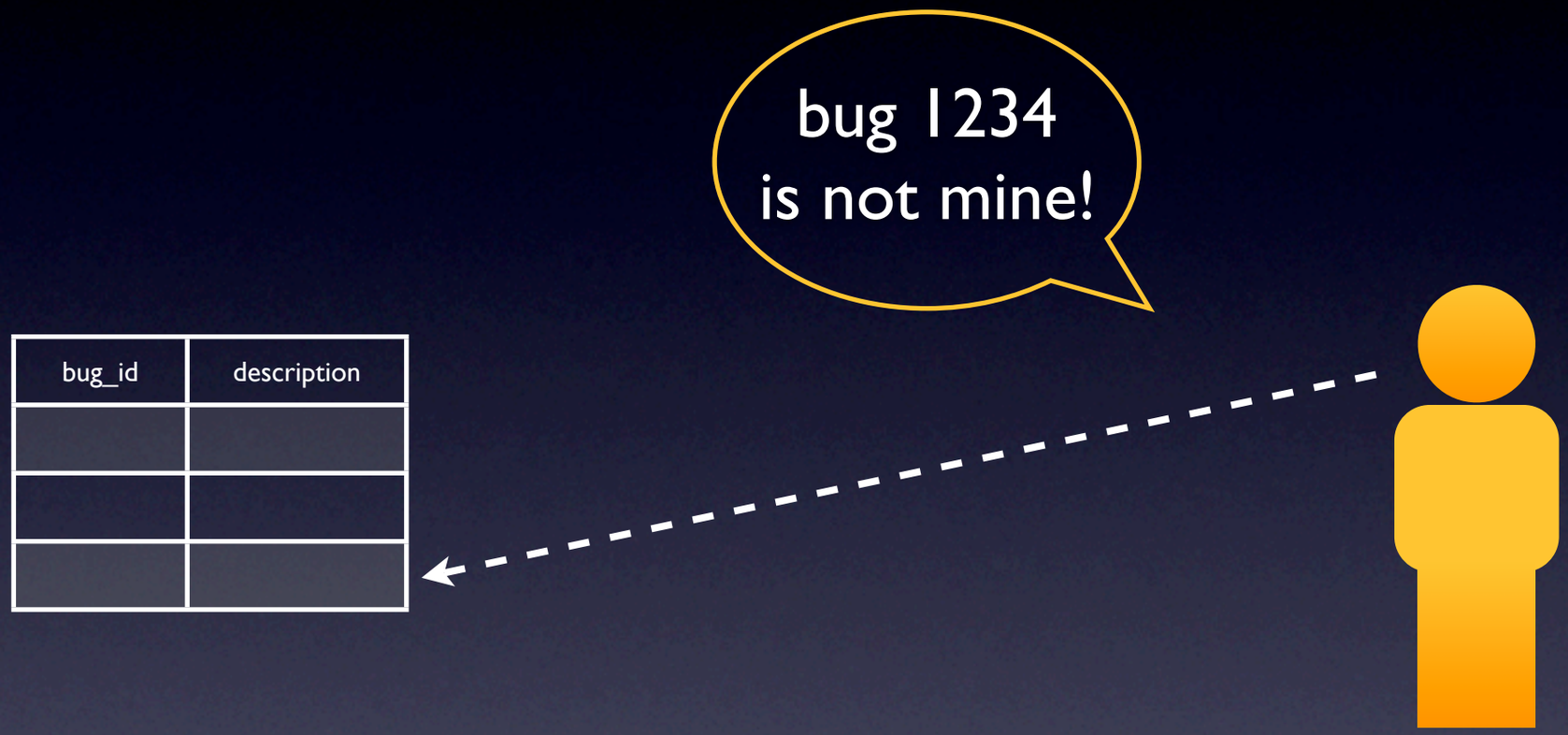
- Reusing primary key values exposes data

bug_id	description

re-insert row
bug_id = 1234

Pseudokey Neat Freak

- Reusing primary key values exposes data



Pseudokey Neat Freak

- **Solution:**
  - Don't re-use primary key values
  - Keys are unique, but not contiguous
  - You will not run out of integers!
    - UNSIGNED BIGINT has a maximum value of 18,446,744,073,709,551,615
    - Create 1,000 rows per second 24x7 for 584 million years

Session Coupling

Session Coupling

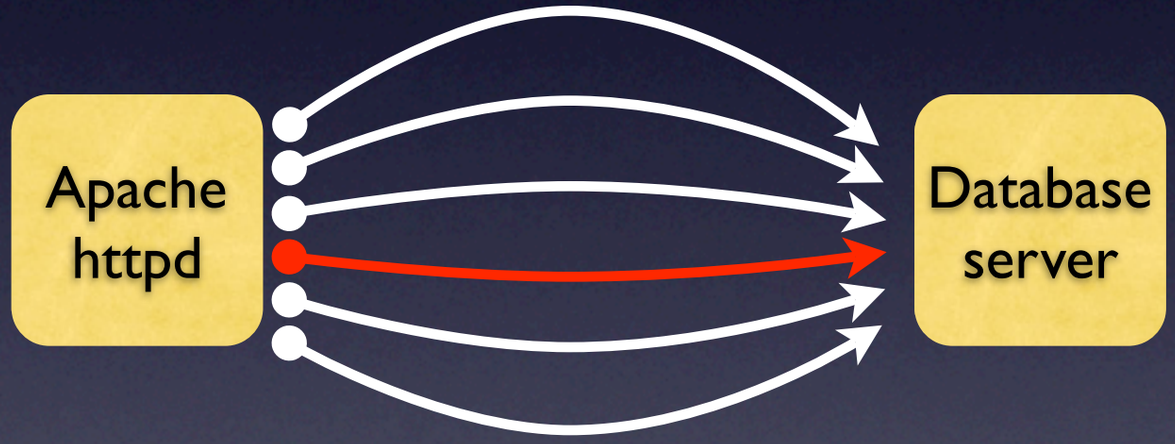
- **Objective:** decrease the overhead of making a database connection
- PHP "share nothing" architecture
- CGI applications

Session Coupling

- **Antipattern:** persistent connections

Session Coupling

- Scalability
  - One database connection per Apache thread
  - Most requests don't need database access



- Database server must maintains many idle connections, may exhaust resources

Session Coupling

- Uniform authentication
  - All apps must use identical user & password
  - Or else each Apache thread must have multiple persistent database connections

Session Coupling

- Connections have state:
  Uncommitted Transactions
- Request #1 starts transaction, but doesn't commit
- Request #2 acquires connection, executes changes, but rolls back current transaction

Session Coupling

- Connections have state: *Character Set*
  - Request #1:
    SET NAMES cp1251_koi8
  - Request #2:
    mysteriously inherits connection character set,
    uses Russian 8-bit encoding and collation

Session Coupling

- Connections have state: *Session Variables*
  - Request #1:
    SET SESSION @@autocommit = 1
  - Request #2:
    mysteriously inherits autocommit behavior,
    ROLLBACK ineffective

Session Coupling

- Connections have state: *Last Insert ID*
  - Request #1:
    INSERT INTO bugs (. . .)
  - Request #2:
    LAST_INSERT_ID() returns bug_id
    generated in previous request

Session Coupling

- **Solution:** reduce need for database connections
  - Create database connections lazily (when needed)
  - Redirect requests for static content (images, CSS, Javascript, downloads) to another web server, e.g. lighttpd

Session Coupling

- Wishlist:
  - Method to reset connection to default state
  - Database resident connection pooling (e.g. Oracle)

Phantom Side Effects

Phantom Side Effects

- **Objective:** execute application tasks with database operations

INSERT INTO bugs (. . .)

...and send email to notify me

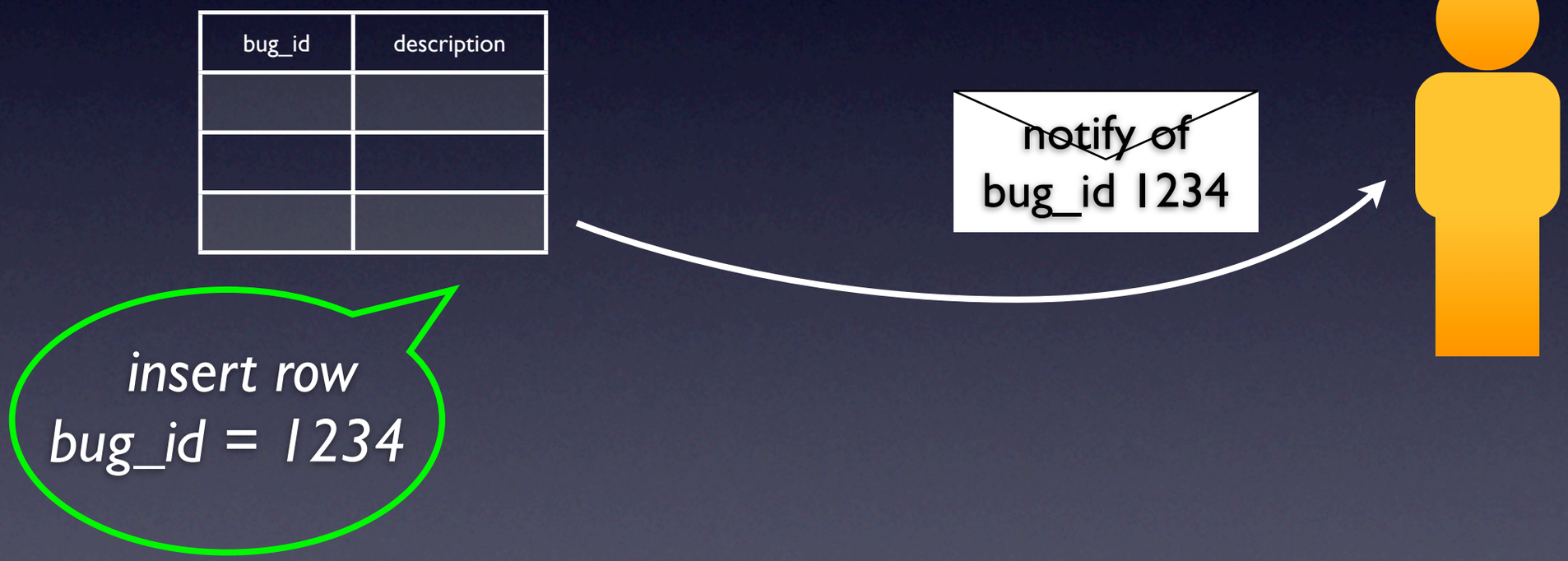
Phantom Side Effects

- **Antipattern:** execute external effects in database triggers, stored procedures, and functions

Phantom Side Effects

- External effects don't obey ROLLBACK

1. Start transaction and INSERT



Phantom Side Effects

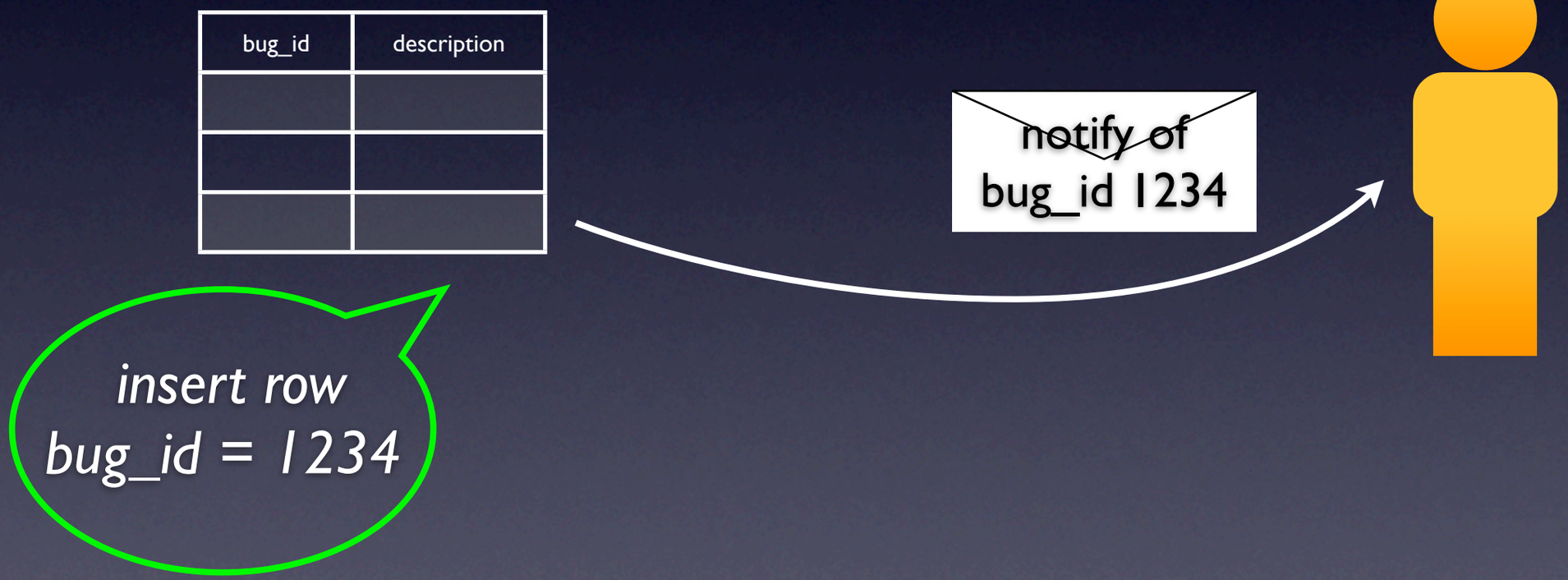
- External effects don't obey ROLLBACK

2. ROLLBACK

Phantom Side Effects

- External effects don't obey transaction isolation

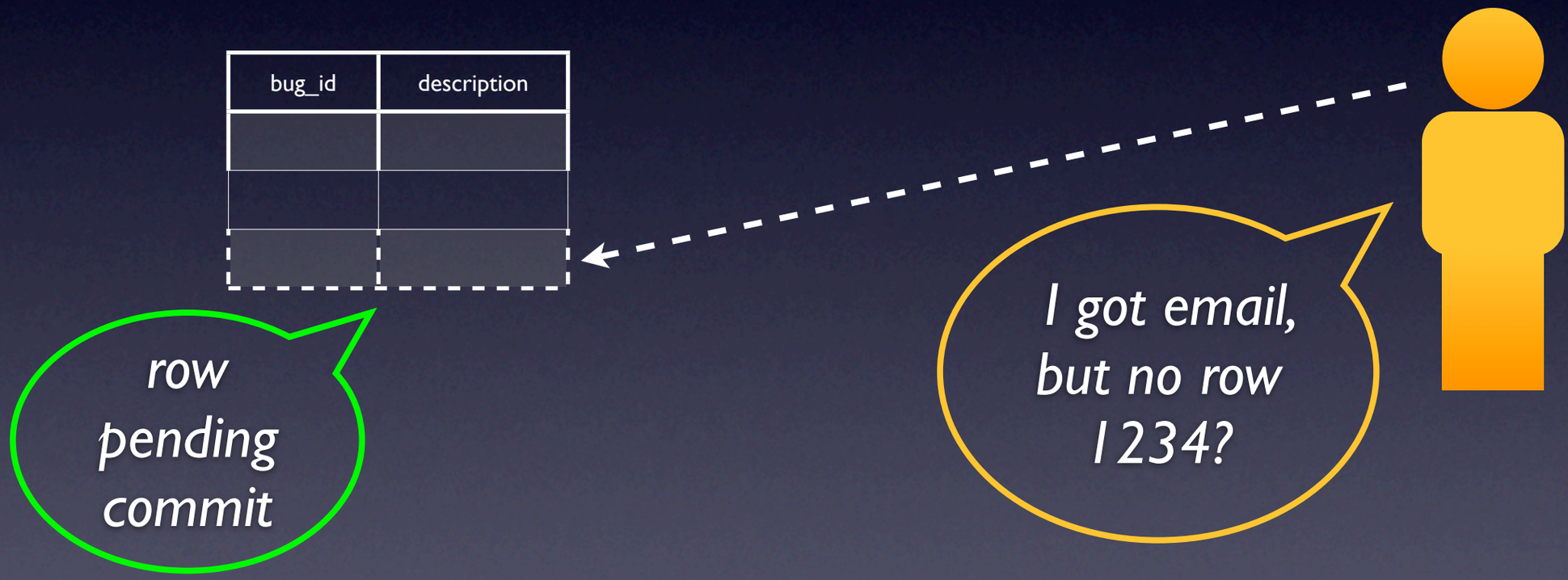
1. Start transaction and INSERT



Phantom Side Effects

- External effects don't obey transaction isolation

2. Email is received before row is visible



Phantom Side Effects

- External effects run as database server user
  - Possible security risk

SQL injection

```
SELECT * FROM bugs
WHERE bug_id = 1234
      or send_email('Buy cheap Rolex watch!')
```

- Auditing/logging confusion

Phantom Side Effects

- Functions may crash

missing file
causes fatal error

```
SELECT pk_encrypt(description,
    '/nonexistant/private.ppk')
FROM bugs
WHERE bug_id = 1234
```

Phantom Side Effects

- Long-running functions delay query
  - Accessing remote resources
  - Unbounded execution time

unresponsive website

```
SELECT libcurl_post(description,
       'http://myblog.org/ …')
FROM bugs
WHERE bug_id = 1234
```

Phantom Side Effects

- **Solution:**
  - Operate only on database in triggers, stored procedures, database functions
  - Wait for transaction to commit
  - Perform external actions in application code

Antipattern Categories

Logical Database Antipatterns

Physical Database Antipatterns

```
CREATE TABLE BugsProducts (
  bug_id   INTEGER REFERENCES Bugs,
  product  VARCHAR(100) REFERENCES Products,
  PRIMARY KEY (bug_id, product)
);
```

Query Antipatterns

```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

```
$dbHandle = new PDO('mysql:dbname=test');
$stmt = $dbHandle->prepare($sql);
$result = $stmt->fetchAll();
```

Thank You

Bill Karwin