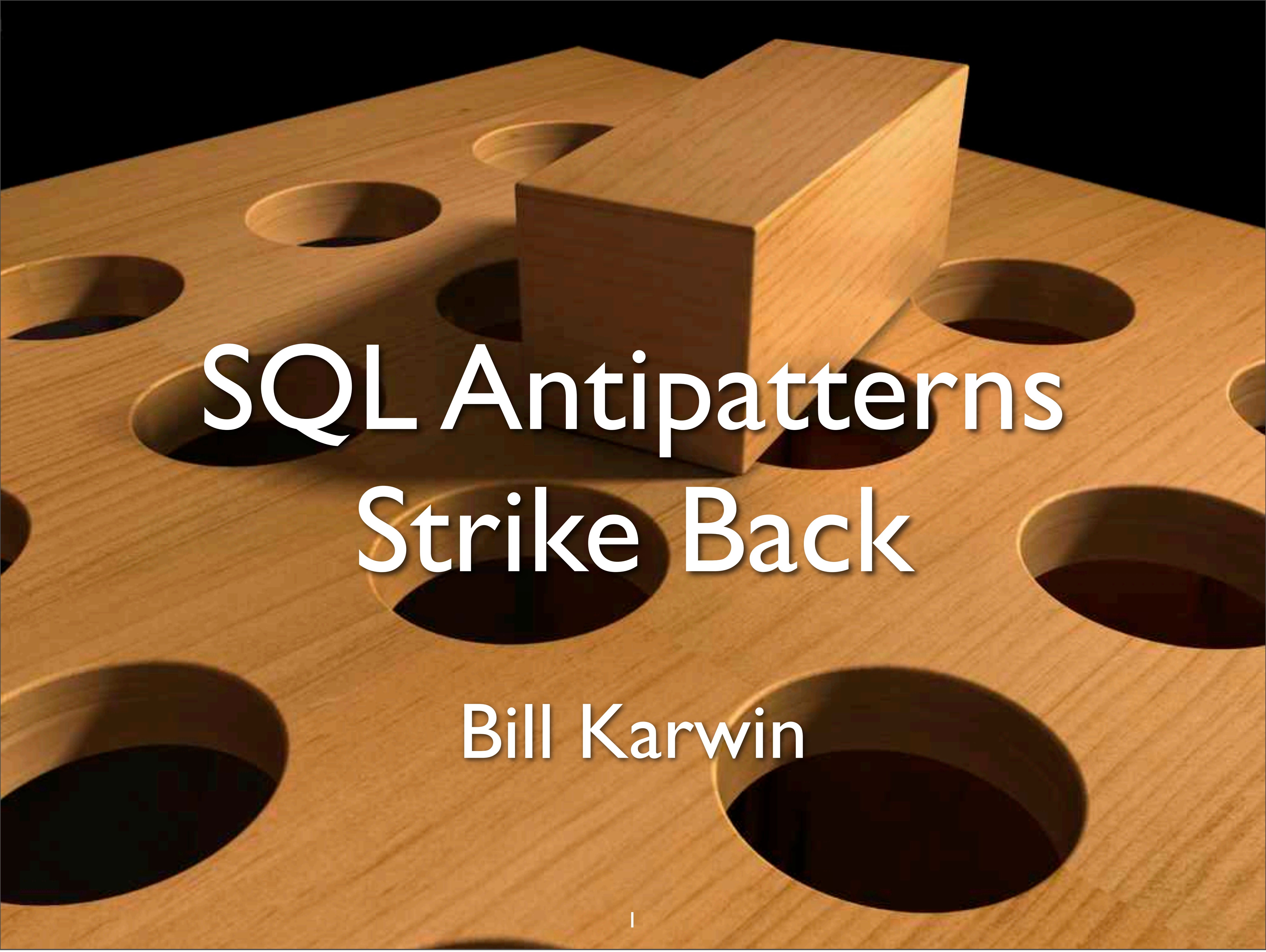
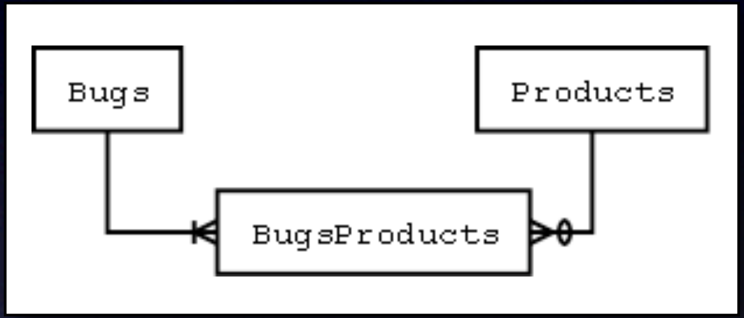


SQL Antipatterns Strike Back

Bill Karwin

Antipattern Categories

Database Design Antipatterns



Database Creation Antipatterns

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

Query Antipatterns

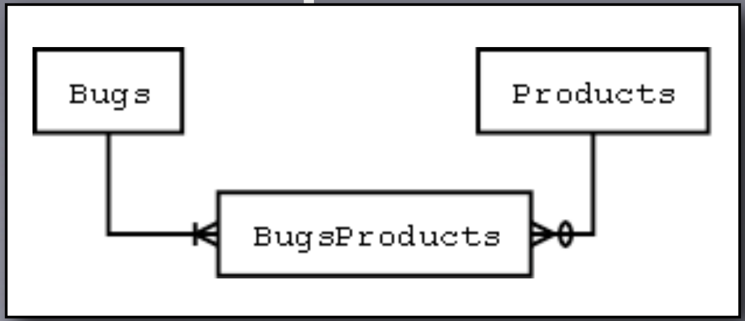
```
SELECT b.product, COUNT(*)
FROM BugsProducts AS b
GROUP BY b.product;
```

Application Antipatterns

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

Antipattern Categories

Database Design Antipatterns



Database Creation 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();
```

Database Design Antipatterns

1. Metadata Tribbles
2. Entity-Attribute-Value
3. Polymorphic Associations
4. Naive Trees

Metadata Tribbles

I want these things off the ship. I don't care if it takes every last man we've got, I want them off the ship.
— James T. Kirk

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 attribute
  - 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 . . . );
. . .
```

same values allocated
in multiple tables

Metadata Tribbles

- Difficult to query across tables

```
SELECT b.status, COUNT(*) AS count_per_status
FROM (
      SELECT * FROM Bugs_2009
       UNION
      SELECT * FROM Bugs_2008
       UNION
      SELECT * FROM Bugs_2007
       UNION
      SELECT * FROM Bugs_2006 ) AS b
GROUP BY b.status;
```

Metadata Tribbles

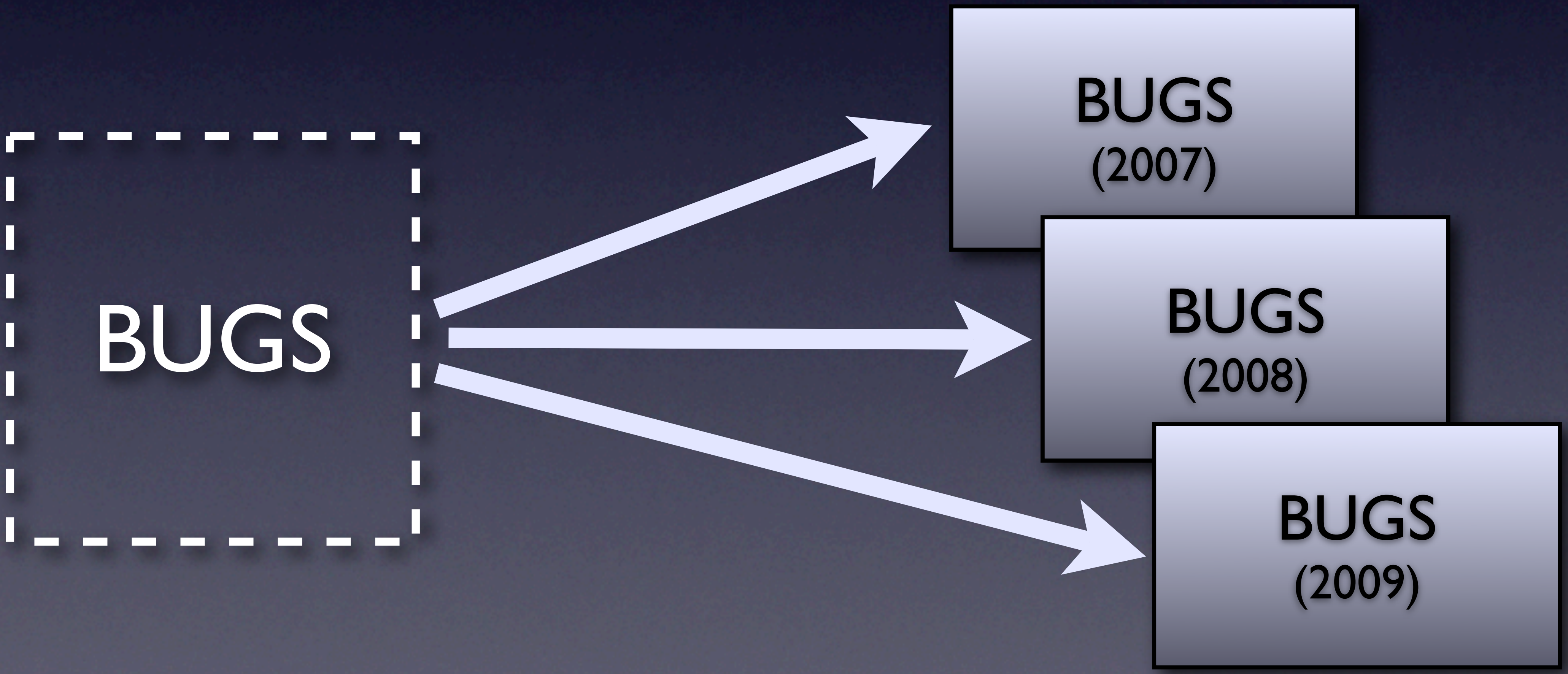
- Table structures are not kept in sync

  ```
  ALTER TABLE Bugs_2009
      ADD COLUMN hours NUMERIC;
  ```

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

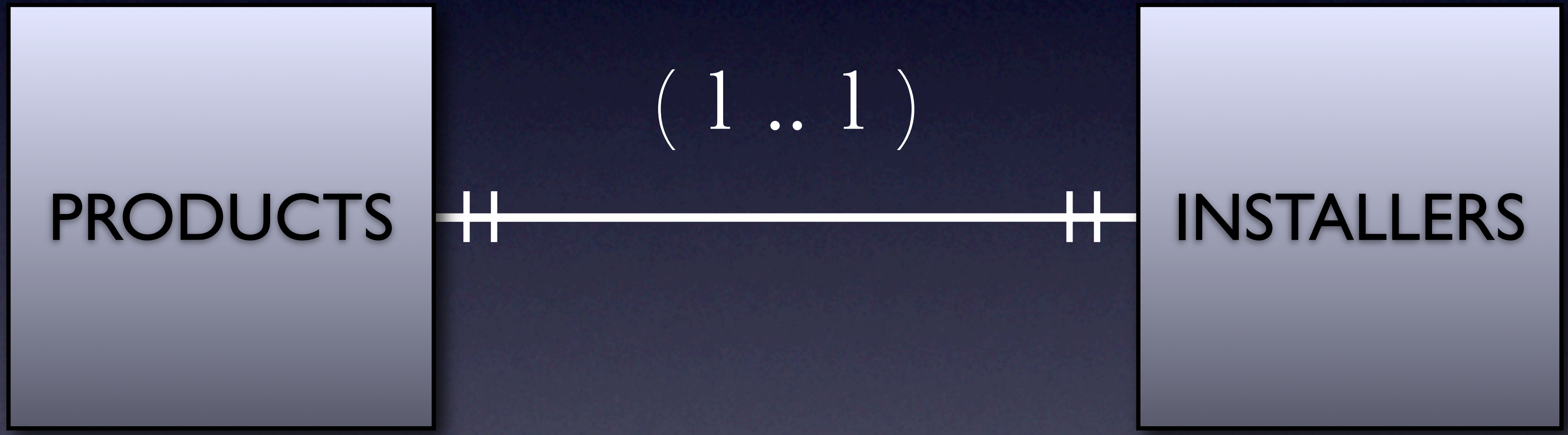
Metadata Tribbles

- **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



PRODUCTS

(1 .. 1)

INSTALLERS

Metadata Tribbles

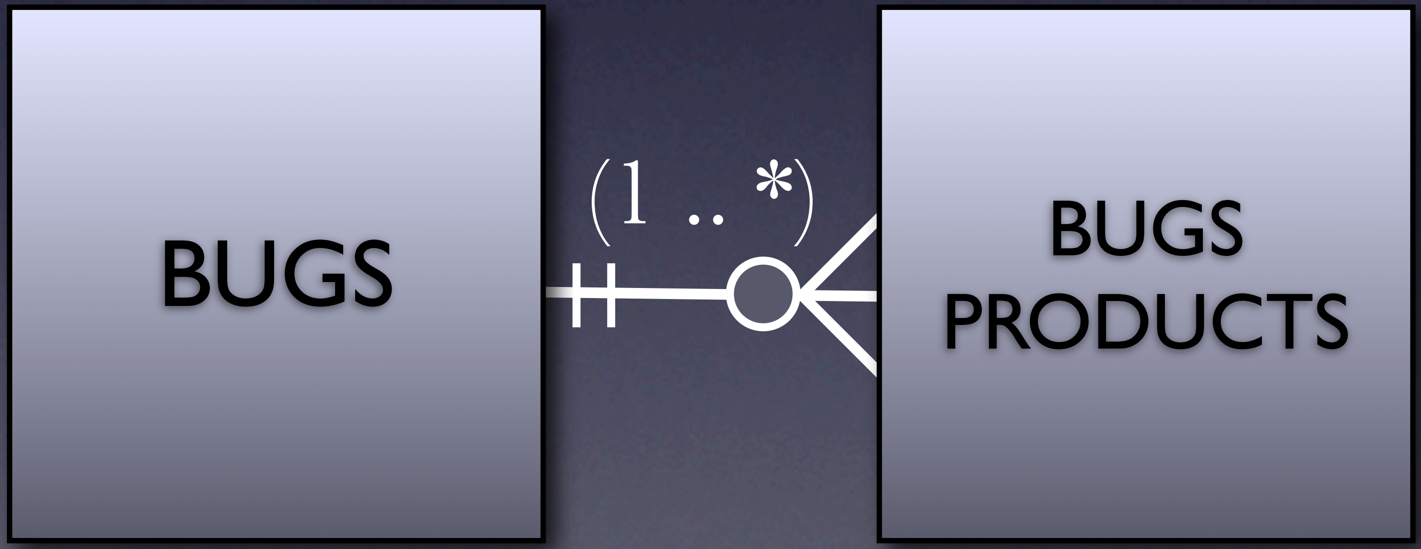
- Columns can also be tribbles:

```
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 BugsProducts (
    bug_id        BIGINT REFERENCES bugs,
    product_id    BIGINT REFERENCES products,
    PRIMARY KEY (bug_id, product_id)
);
```



BUGS

(1 .. *)

BUGS PRODUCTS

Entity-Attribute-Value

If you try and take a cat apart to see how it works,
the first thing you have on your hands is a non-working cat.
— Richard Dawkins

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 NOT NULL,
    attr_name       VARCHAR(20) NOT NULL,
    attr_value      VARCHAR(100),
    PRIMARY KEY (bug_id, attr_name),
    FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

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 rely on 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:

need one JOIN per attribute

```
SELECT b.bug_id,
        e1.attr_value AS created_date,
        e2.attr_value AS priority,
        e3.attr_value AS description,
        e4.attr_value AS status,
        e5.attr_value AS reported_by
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’)
LEFT JOIN eav e3 ON (b.bug_id = e3.bug_id AND e3.attr_name = ‘description’)
LEFT JOIN eav e4 ON (b.bug_id = e4.bug_id AND e4.attr_name = ‘status’)
LEFT JOIN eav e5 ON (b.bug_id = e5.bug_id AND e5.attr_name = ‘reported_by’);
```

bug_id	created_date	priority	description	status	reported_by
1234	2008-04-01	high	Crashes when I save.	NEW	Bill

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

- **Solution #1:** Single Table Inheritance
  - One table with many columns
  - Columns are NULL when inapplicable

```
CREATE TABLE Issues (
        issue_id            SERIAL PRIMARY KEY,
        created_date        DATE NOT NULL,
        priority            VARCHAR(20),
        description         TEXT,
        issue_type          CHAR(1) CHECK (issue_type IN (‘B’, ‘F’)),
        bug_severity        VARCHAR(20),
        feature_sponsor     VARCHAR(100)
);
```

Entity-Attribute-Value

- **Solution #2:** Concrete Table Inheritance
  - Define similar tables for similar types
  - Duplicate common columns in each table

```
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)
);
```

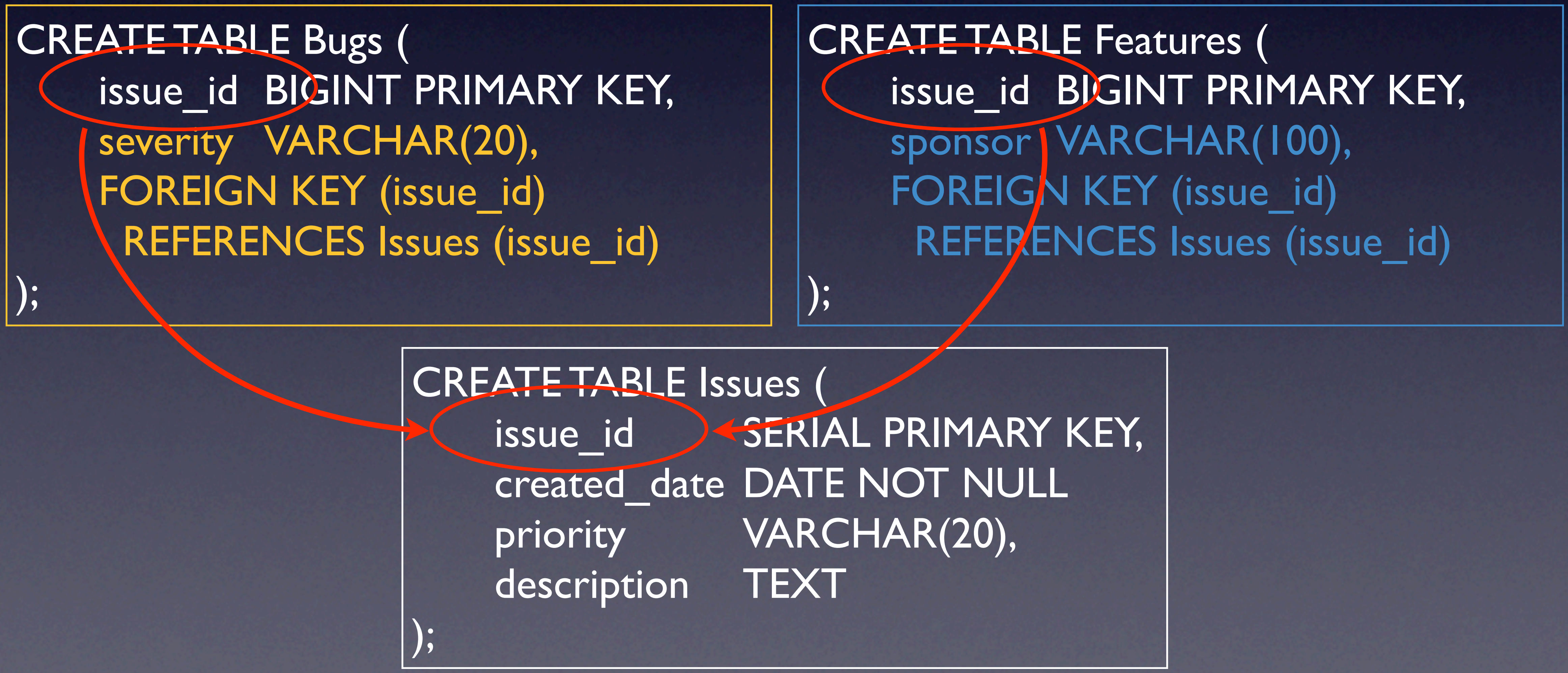
Entity-Attribute-Value

- **Solution #2:** Concrete Table Inheritance
  - Use UNION to search both tables:

```
SELECT * FROM (
    SELECT issue_id,  description FROM Bugs
    UNION ALL
    SELECT issue_id, description FROM Features
) unified_table
WHERE description LIKE ...
```

Entity-Attribute-Value

- **Solution #3:** Class Table Inheritance
  - Common columns in base table
  - Subtype-specific columns in subtype tables



```
CREATE TABLE Bugs (
    issue_id  BIGINT PRIMARY KEY,
    severity   VARCHAR(20),
    FOREIGN KEY (issue_id)
     REFERENCES Issues (issue_id)
);
```

```
CREATE TABLE Features (
    issue_id  BIGINT PRIMARY KEY,
    sponsor   VARCHAR(100),
    FOREIGN KEY (issue_id)
     REFERENCES Issues (issue_id)
);
```

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

Entity-Attribute-Value

- **Solution #3:** Class Table Inheritance
  - Easy to query common columns:

    ```
    SELECT * FROM Issues
    WHERE description LIKE ...
    ```

  - Easy to query one subtype at a time:

    ```
    SELECT * FROM Issues
    JOIN Bugs USING (issue_id);
    ```

Entity-Attribute-Value

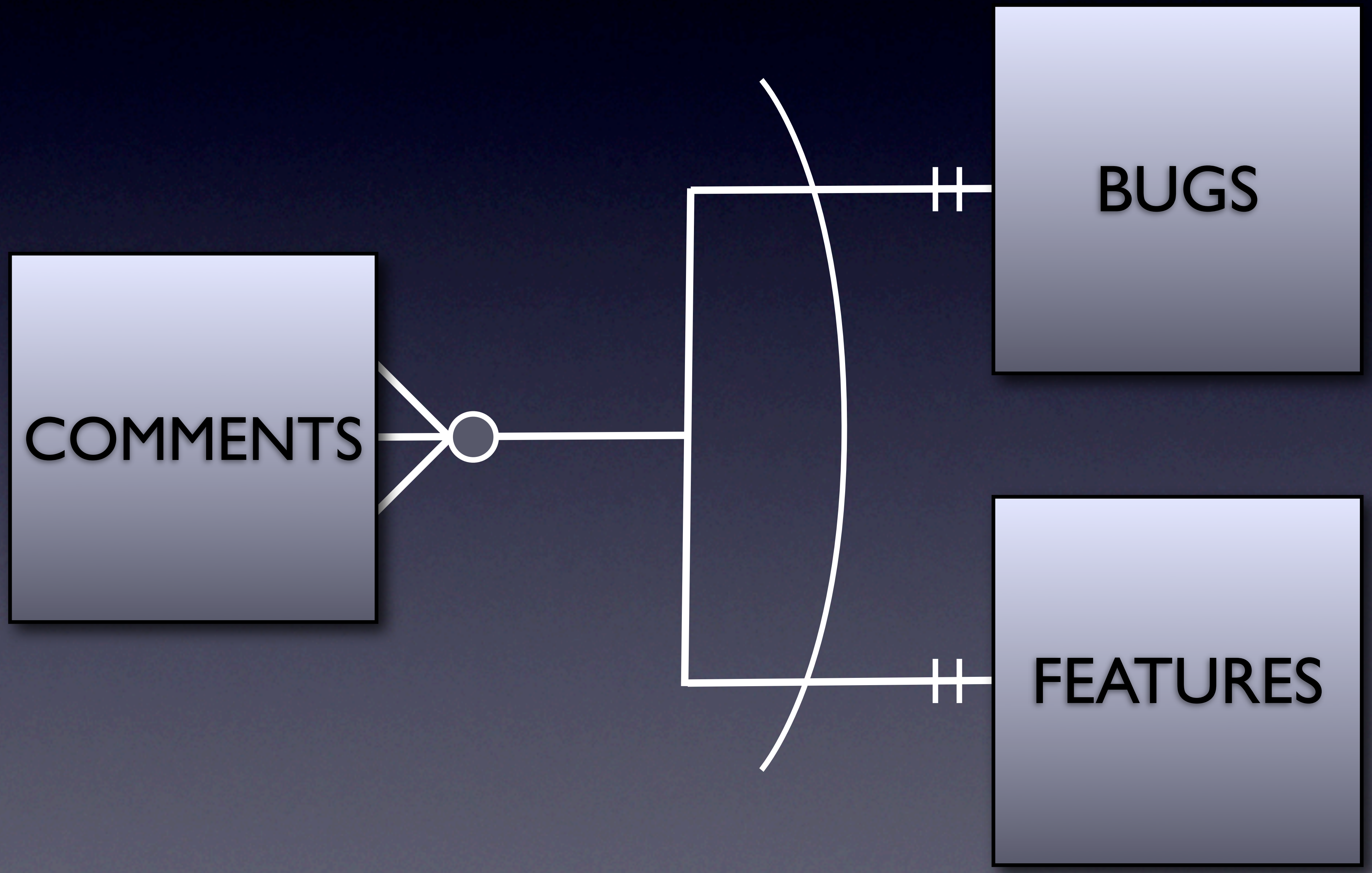
- Appropriate usage of EAV:
  - If attributes must be fully flexible and dynamic
  - You must enforce constraints in application code
  - Don’t try to fetch one object in a single row
  - Consider non-relational solutions
for semi-structured data, e.g. RDF/XML

Polymorphic Associations

Of course, some people do go both ways.
— The Scarecrow

Polymorphic Assocations

- **Objective:** reference multiple parents



Polymorphic Assocations

- Can’t make a FOREIGN KEY constraint reference two tables:

```
CREATE TABLE Comments (
    comment_id   SERIAL PRIMARY KEY,
    comment      TEXT NOT NULL,
    issue_type   VARCHAR(15) CHECK
        (issue_type IN (‘Bugs’, ‘Features’)),
    issue_id     BIGINT NOT NULL,
    FOREIGN KEY issue_id REFERENCES
);
```

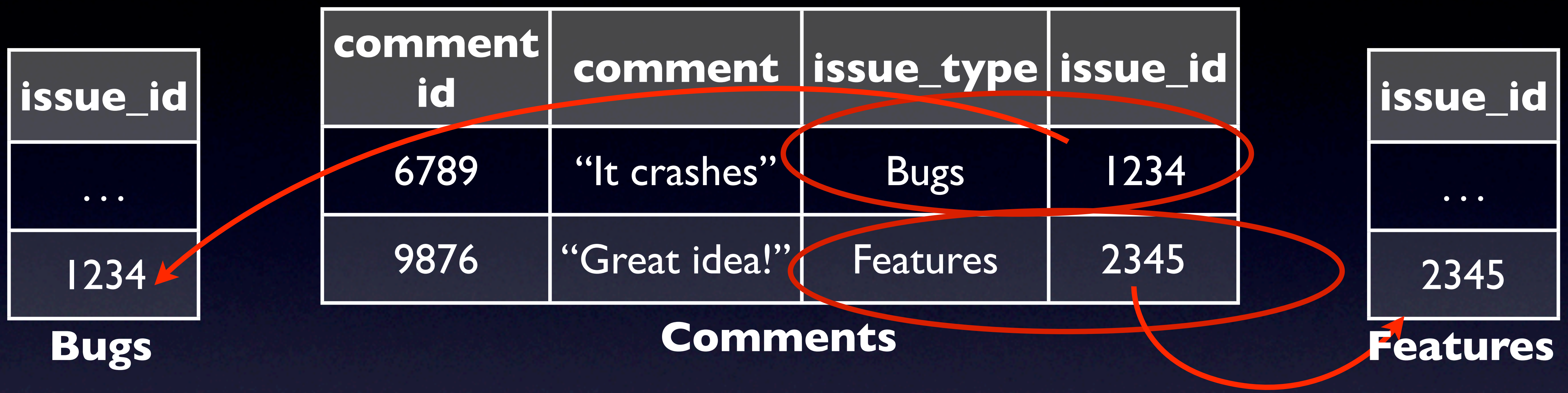
you need this to be Bugs or Features

Polymorphic Assocations

- Instead, you have to define table with no FOREIGN KEY or referential integrity:

```
CREATE TABLE Comments (
    comment_id   SERIAL PRIMARY KEY,
    comment      TEXT NOT NULL,
    issue_type   VARCHAR(15) CHECK
        (issue_type IN (‘Bugs’, ‘Features’)),
    issue_id     BIGINT NOT NULL
);
```

Polymorphic Assocations



issue_id
. . .
1234

Bugs

comment id	comment	issue_type	issue_id
6789	“It crashes”	Bugs	1234
9876	“Great idea!”	Features	2345

Comments

issue_id
. . .
2345

Features

Query result:

comment_id	comment	issue_type	c. issue_id	b. issue_id	f. issue_id
6789	“It crashes”	Bug	1234	1234	NULL
9876	“Great idea!”	Feature	2345	NULL	2345

Polymorphic Assocations

- You can’t use a different table for each row. You must name all tables explicitly.

```
SELECT * FROM Comments
JOIN            USING (issue_id);
```

you need this to be Bugs or Features

Polymorphic Assocations

- Instead, join to each parent table:

```
SELECT *
FROM Comments c
LEFT JOIN Bugs b     ON (c.issue_type = ‘Bugs’
    AND c.issue_id = b.issue_id)
LEFT JOIN Features f ON (c.issue_type = ‘Features’
    AND c.issue_id = f.issue_id);
```

you have to get these strings right

Polymorphic Assocations

- **Solution #1:** exclusive arcs

```
CREATE TABLE Comments (
    comment_id   SERIAL PRIMARY KEY,
    comment      TEXT NOT NULL,
    bug_id       BIGINT,
    feature_id   BIGINT,
    FOREIGN KEY bug_id
        REFERENCES Bugs(bug_id)
    FOREIGN KEY feature_id
        REFERENCES Features(feature_id)
);
```

both columns are nullable;
exactly one must be non-null

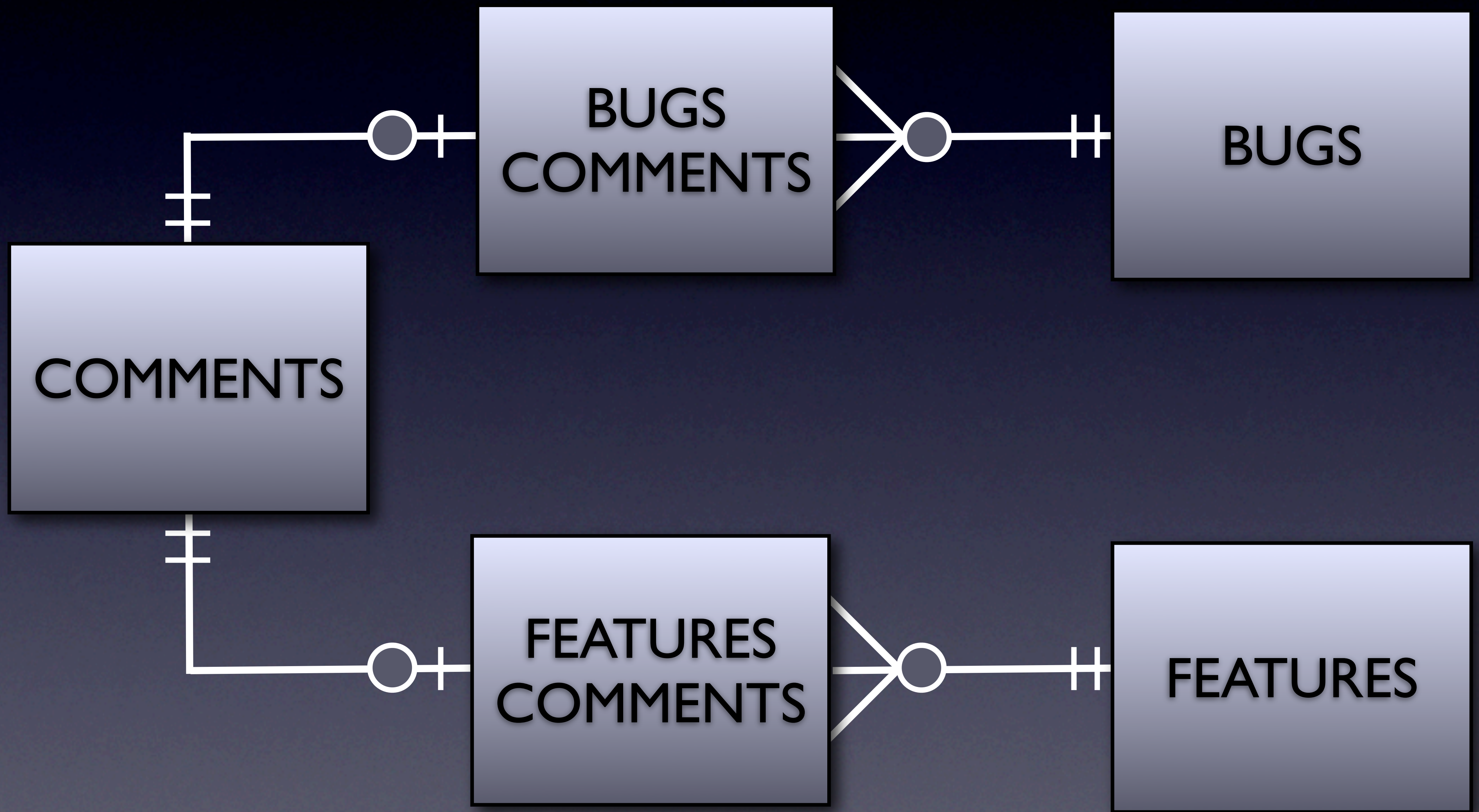
Polymorphic Assocations

- **Solution #1:** exclusive arcs
  - Referential integrity is enforced
  - But hard to make sure exactly one is non-null
  - Queries are easier:

```
SELECT * FROM Comments c
LEFT JOIN Bugs b  USING (bug_id)
LEFT JOIN Features f USING (feature_id);
```

Polymorphic Assocations

- **Solution #2:** reverse the relationship



Polymorphic Assocations

- **Solution #2:** reverse the relationship

```
CREATE TABLE BugsComments (
    comment_id   BIGINT NOT NULL,
    bug_id       BIGINT NOT NULL,
    PRIMARY KEY (comment_id),
    FOREIGN KEY (comment_id) REFERENCES Comments(comment_id),
    FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE FeaturesComments (
    comment_id   BIGINT NOT NULL,
    feature_id   BIGINT NOT NULL,
    PRIMARY KEY (comment_id),
    FOREIGN KEY (comment_id) REFERENCES Comments(comment_id),
    FOREIGN KEY (feature_id) REFERENCES Features(feature_id)
);
```

Polymorphic Assocations

- **Solution #2:** reverse the relationship
  - Referential integrity is enforced
  - Query comments for a given bug:

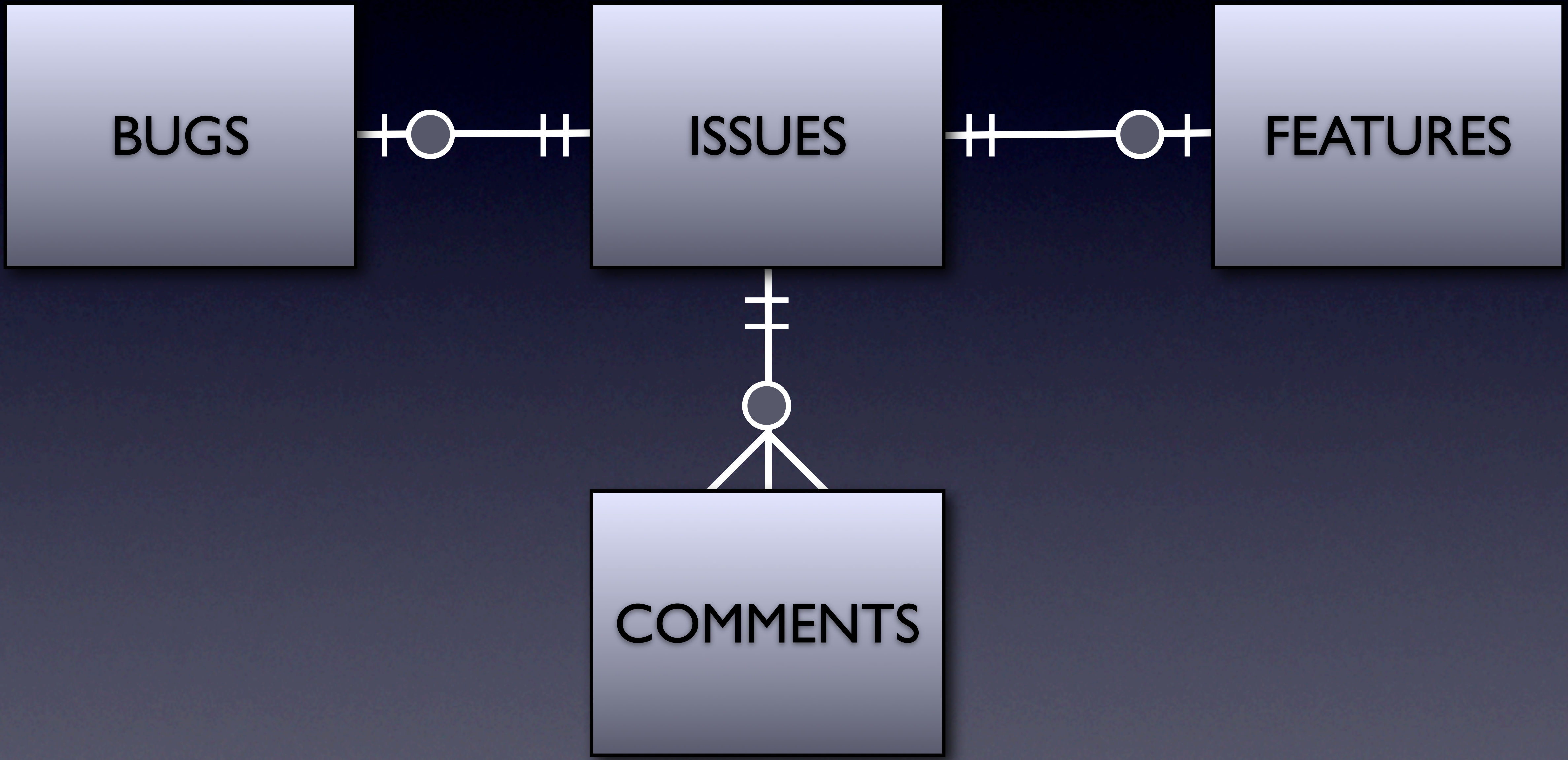
```
SELECT * FROM BugsComments b
JOIN Comments c USING (comment_id)
WHERE b.bug_id = 1234;
```

  - Query bug/feature for a given comment:

```
SELECT * FROM Comments
LEFT JOIN (BugsComments JOIN Bugs USING (bug_id))
    USING (comment_id)
LEFT JOIN (FeaturesComments JOIN Features USING (feature_id))
    USING (comment_id)
WHERE comment_id = 9876;
```

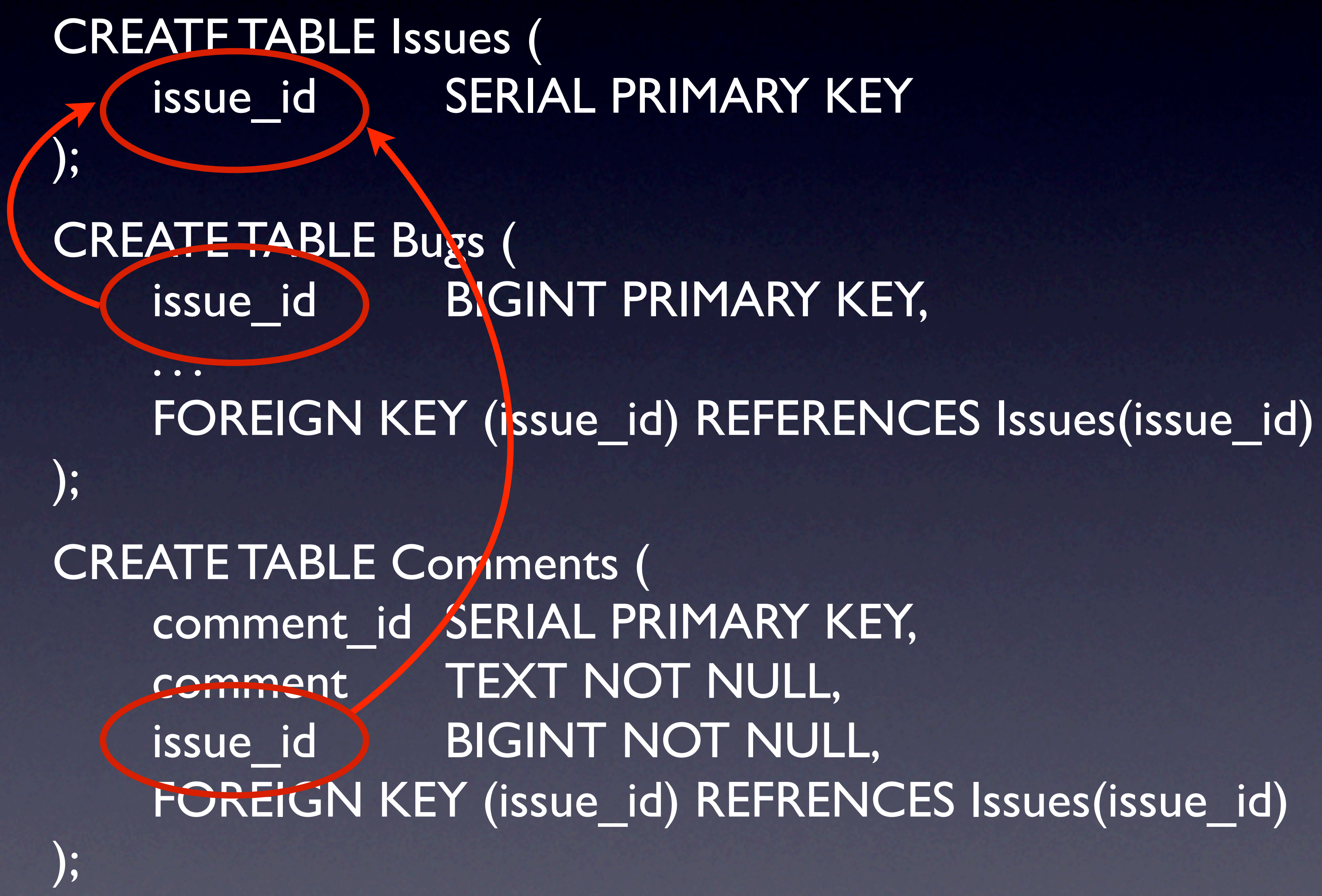
Polymorphic Assocations

- **Solution #3:** use a base parent table



Polymorphic Assocations

- **Solution #3:** use a base parent table



```
CREATE TABLE Issues (
        issue_id        SERIAL PRIMARY KEY
);
CREATE TABLE Bugs (
        issue_id        BIGINT PRIMARY KEY,
        . . .
        FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
CREATE TABLE Comments (
        comment_id  SERIAL PRIMARY KEY,
        comment         TEXT NOT NULL,
        issue_id        BIGINT NOT NULL,
        FOREIGN KEY (issue_id) REFRENCES Issues(issue_id)
);
```

Polymorphic Assocations

- **Solution #3:** use a base parent table
  - Referential integrity is enforced
  - Queries are easier:

```
SELECT * FROM Comments
JOIN Issues USING (issue_id)
LEFT JOIN Bugs USING (issue_id)
LEFT JOIN Features USING (issue_id);
```

Polymorphic Assocations

- Enforcing disjoint subtypes:

```
CREATE TABLE Issues (
    issue_id       SERIAL PRIMARY KEY,
    issue_type     CHAR(1) NOT NULL CHECK (issue_type IN (‘B’, ‘F’)),
    UNIQUE KEY (issue_id, issue_type)
);
```

referential integrity

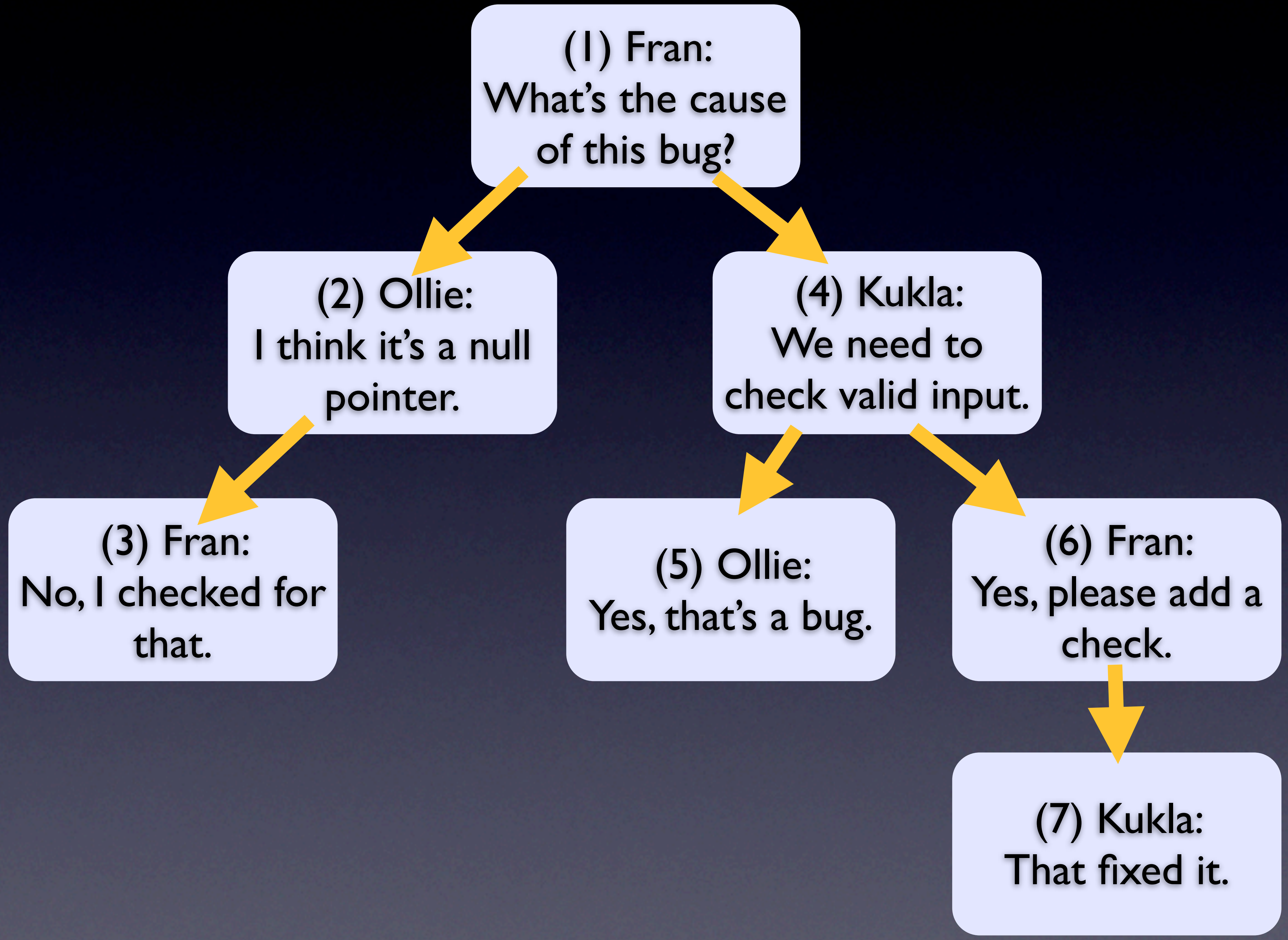
```
CREATE TABLE Bugs (
    issue_id       BIGINT PRIMARY KEY,
    issue_type     CHAR(1) NOT NULL CHECK (issue_type = ‘B’),
    . . .
    FOREIGN KEY (issue_id, issue_type)
        REFERENCES Issues(issue_id, issue_type)
);
```

Naive Trees

Naive Trees

- **Objective:** store/query hierarchical data
  - Categories/subcategories
  - Bill of materials
  - Threaded discussions

Naive Trees



(1) Fran:
What's the cause of this bug?

(2) Ollie:
I think it's a null pointer.

(3) Fran:
No, I checked for that.

(4) Kukla:
We need to check valid input.

(5) Ollie:
Yes, that's a bug.

(6) Fran:
Yes, please add a check.

(7) Kukla:
That fixed it.

Naive Trees

- Adjacency List
  - Naive solution nearly everyone uses
  - Each entry in the tree knows immediate parent

comment_id	parent_id	author	comment
1	NULL	Fran	What's the cause of this bug?
2	1	Ollie	I think it's a null pointer.
3	2	Fran	No, I checked for that.
4	1	Kukla	We need to check valid input.
5	4	Ollie	Yes, that's a bug.
6	4	Fran	Yes, please add a check
7	6	Kukla	That fixed it.

Naive Trees

- Adjacency List
  - Easy to inserting a new comment:

```
INSERT INTO Comments (parent_id, author, comment)
    VALUES (7, ‘Kukla’, ‘Thanks!’);
```

  - Easy to move a subtree to a new position:

```
UPDATE Comments SET parent_id = 3
WHERE comment_id = 6;
```

Naive Trees

- Adjacency List
  - Querying a node’s immediate children is easy:

    ```
    SELECT * FROM Comments c1
    LEFT JOIN Comments c2
      ON (c2.parent_id = c1.comment_id);
    ```

  - Querying a node’s immediate parent is easy:

    ```
    SELECT * FROM Comments c1
    JOIN Comments c2
      ON (c1.parent_id = c2.comment_id);
    ```

Naive Trees

- Adjacency List
  - Hard to query all descendants in a deep tree:

```
SELECT * FROM Comments c1
LEFT JOIN Comments c2 ON (c2.parent_id = c1.comment_id)
LEFT JOIN Comments c3 ON (c3.parent_id = c2.comment_id)
LEFT JOIN Comments c4 ON (c4.parent_id = c3.comment_id)
LEFT JOIN Comments c5 ON (c5.parent_id = c4.comment_id)
LEFT JOIN Comments c6 ON (c6.parent_id = c5.comment_id)
LEFT JOIN Comments c7 ON (c7.parent_id = c6.comment_id)
LEFT JOIN Comments c8 ON (c8.parent_id = c7.comment_id)
LEFT JOIN Comments c9 ON (c9.parent_id = c8.comment_id)
LEFT JOIN Comments c10 ON (c10.parent_id = c9.comment_id)
. . .
```

it still doesn't support
unlimited depth!

Naive Trees

- **Solution #1:** Path Enumeration
  - Store chain of ancestors as a string in each node
  - Good for breadcrumbs, or sorting by hierarchy

comment_id	path	author	comment
1	1/	Fran	What's the cause of this bug?
2	1/2/	Ollie	I think it's a null pointer.
3	1/2/3/	Fran	No, I checked for that.
4	1/4/	Kukla	We need to check valid input.
5	1/4/5/	Ollie	Yes, that's a bug.
6	1/4/6/	Fran	Yes, please add a check
7	1/4/6/7/	Kukla	That fixed it.

Naive Trees

- **Solution #1:** Path Enumeration
  - Easy to query all ancestors of comment #7:

    ```
    SELECT * FROM Comments
    WHERE ‘1/4/6/7/’ LIKE path || ‘%’;
    ```

  - Easy to query all descendants of comment #4:

    ```
    SELECT * FROM Comments
    WHERE path LIKE ‘1/4/%’;
    ```

Naive Trees

- **Solution #1:** Path Enumeration
  - Easy to add child of comment 7:

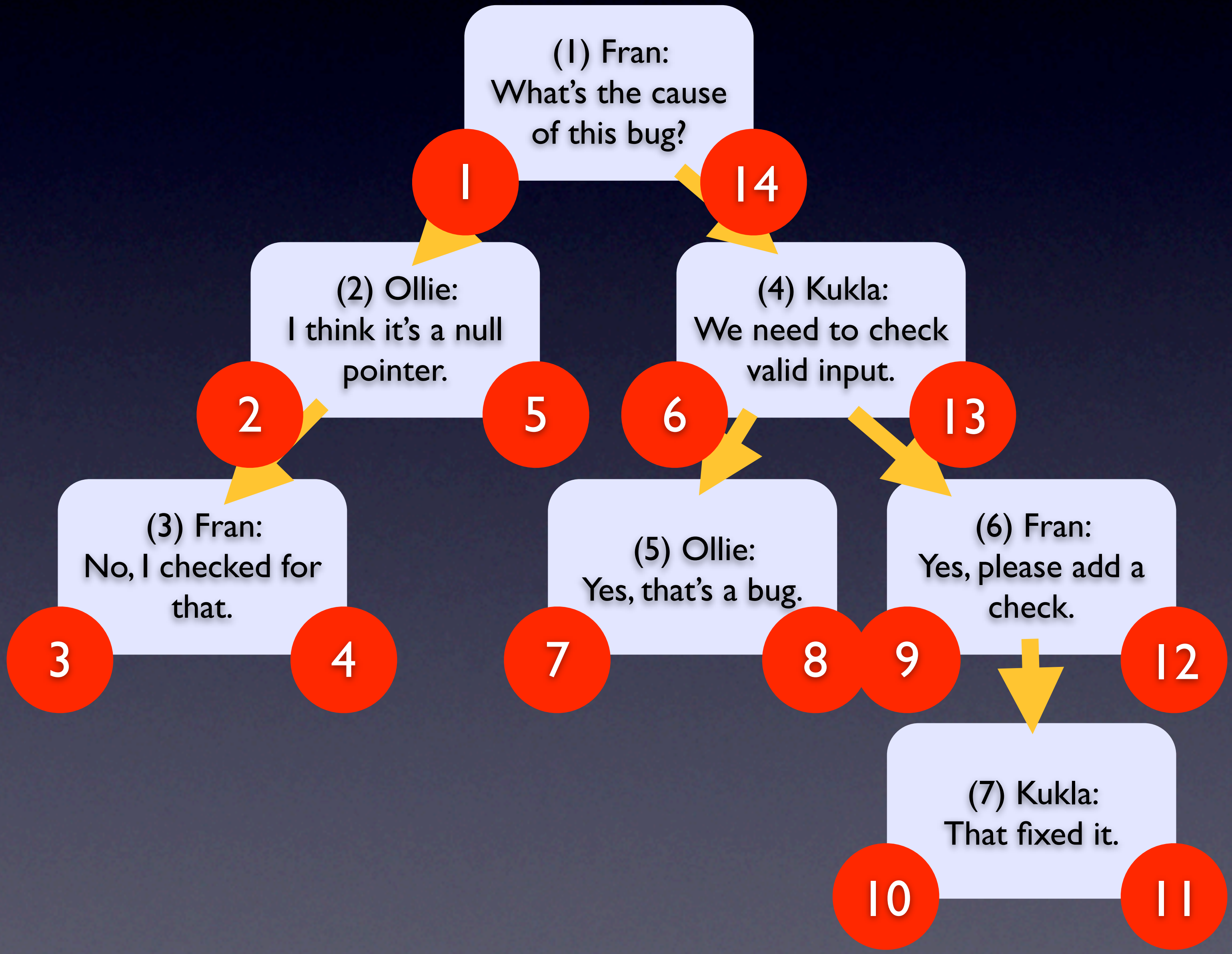
```
INSERT INTO Comments (author, comment)
VALUES (‘Ollie’, ‘Good job!’);

SELECT path FROM Comments
WHERE comment_id = 7;

UPDATE Comments
SET path = $parent_path || LAST_INSERT_ID() || ‘/’
WHERE comment_id = LAST_INSERT_ID();
```

Naive Trees

- **Solution #2:** Nested Sets
  - Each comment encodes its descendants using two numbers:
  - A comment's *right* number is *less than* all the numbers used by the comment's descendants.
  - A comment's *left* number is *greater than* all the numbers used by the comment's descendants.

Naive Trees

- **Solution #2:** Nested Sets



Naive Trees

- **Solution #2:** Nested Sets

comment_id	nsleft	nsright	author	comment
1	1	14	Fran	What's the cause of this bug?
2	2	5	Ollie	I think it's a null pointer.
3	3	4	Fran	No, I checked for that.
4	6	13	Kukla	We need to check valid input.
5	7	8	Ollie	Yes, that's a bug.
6	9	12	Fran	Yes, please add a check
7	10	11	Kukla	That fixed it.

these are not foreign keys

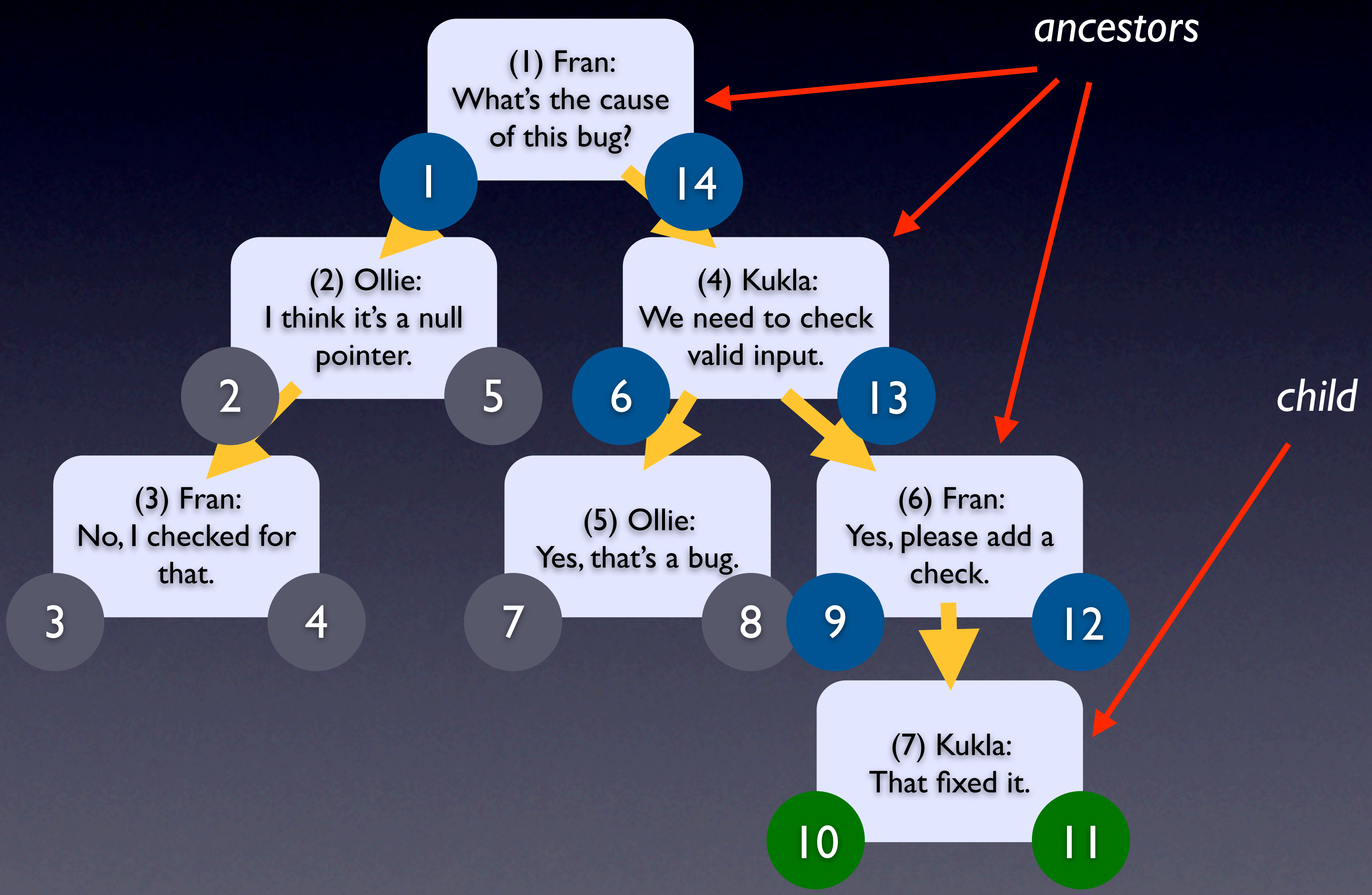
Naive Trees

- **Solution #2:** Nested Sets
  - Easy to query all ancestors of comment #7:

```
SELECT * FROM Comments child
JOIN Comments ancestor
  ON (child.left BETWEEN  ancestor.nsleft
                     AND  ancestor.nsright)
WHERE child.comment_id = 7;
```

Naive Trees

- **Solution #2:** Nested Sets



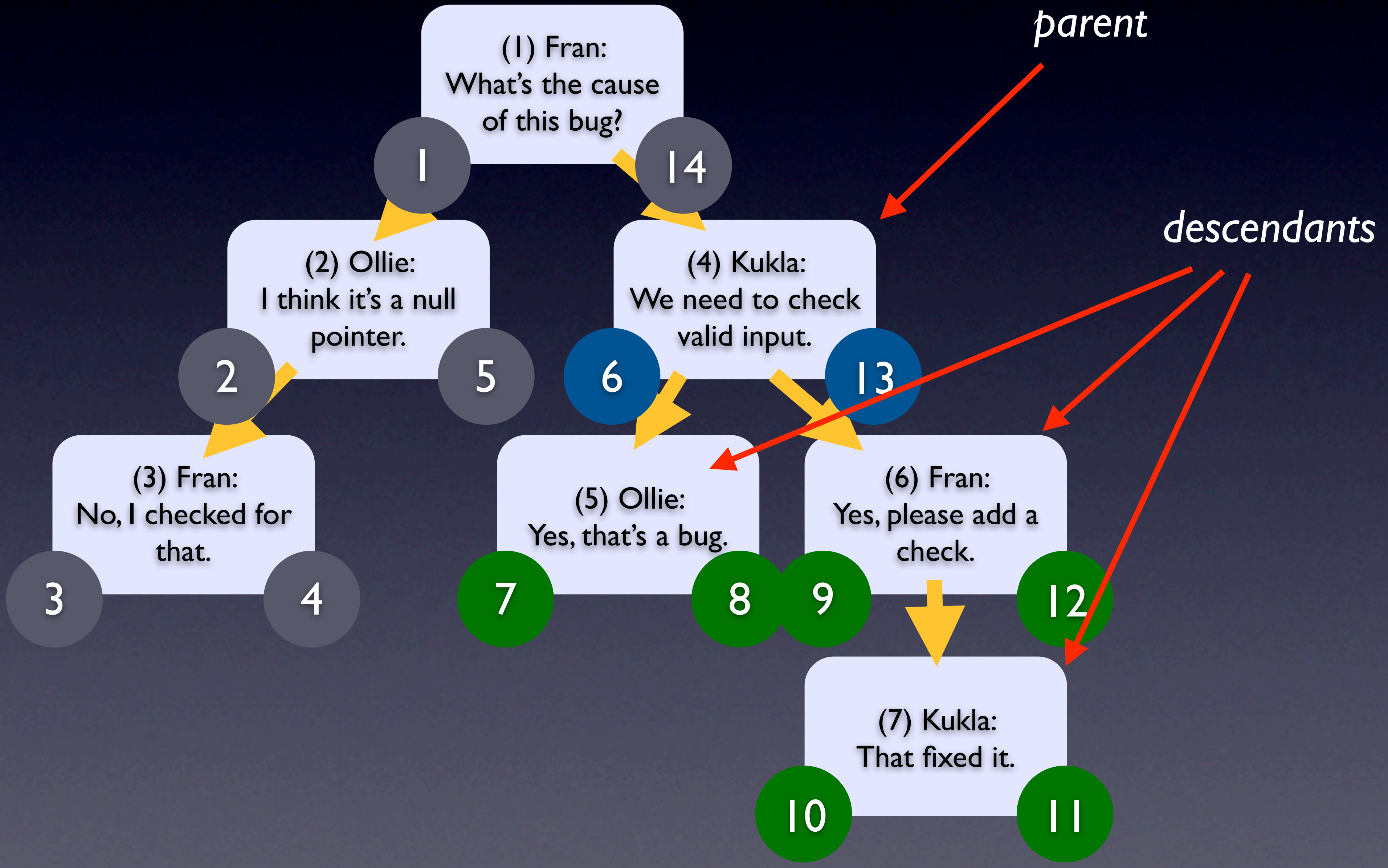
Naive Trees

- **Solution #2:** Nested Sets
  - Easy to query all descendants of comment #4:

```
SELECT * FROM Comments parent
JOIN Comments descendant
  ON (descendant.left BETWEEN   parent.nsleft
                          AND   parent.nsright)
WHERE parent.comment_id = 4;
```

Naive Trees

- **Solution #2:** Nested Sets



Naive Trees

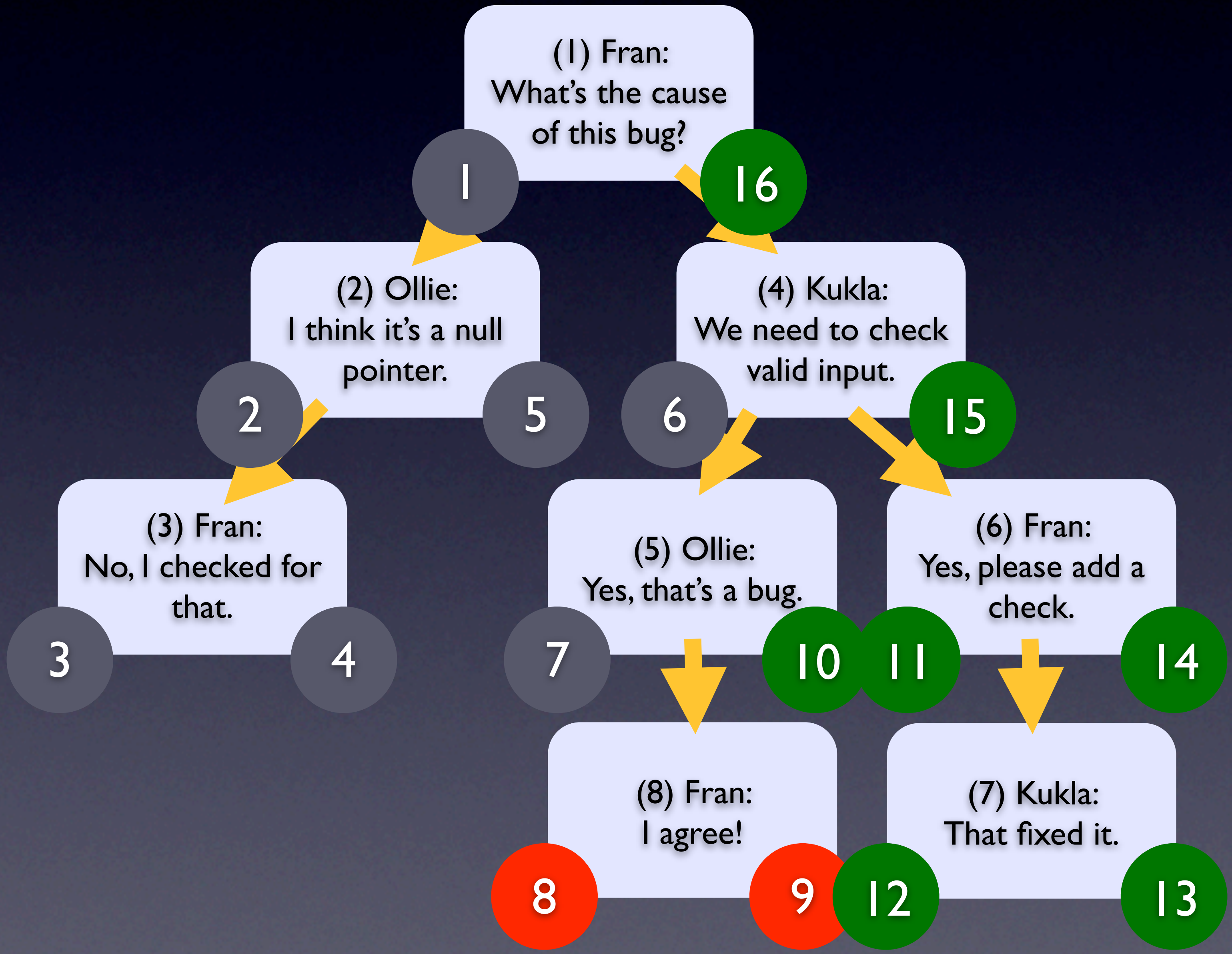
- **Solution #2:** Nested Sets
    - Hard to insert a new child of comment #5:

```
UPDATE Comment
SET  nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft END,
     nsright = nsright+2
WHERE nsright >= 7;

INSERT INTO Comment (nsleft, nsright, author, comment)
 VALUES (8, 9, 'Fran', 'I agree!');
```

    - Recalculate *left* values for all nodes to the right of the new child. Recalculate *right* values for all nodes above and to the right.

Naive Trees

- **Solution #2:** Nested Sets



(1) Fran: What's the cause of this bug?

(2) Ollie: I think it's a null pointer.

(3) Fran: No, I checked for that.

(4) Kukla: We need to check valid input.

(5) Ollie: Yes, that's a bug.

(6) Fran: Yes, please add a check.

(8) Fran: I agree!

(7) Kukla: That fixed it.

Naive Trees

- **Solution #2:** Nested Sets
  - Hard to query the parent of comment #6:

```
SELECT parent.* FROM Comments AS c
JOIN Comments AS parent
  ON (c.nsleft BETWEEN parent.nsleft AND parent.nsright)
LEFT OUTER JOIN Comments AS in_between
  ON (c.nsleft BETWEEN in_between.nsleft AND in_between.nsright
    AND in_between.nsleft BETWEEN parent.nsleft AND parent.nsright)
WHERE c.comment_id = 6 AND in_between.comment_id IS NULL;
```

  - Parent of #6 is an ancestor who has no descendant who is also an ancestor of #6.
  - Querying a child is a similar problem.

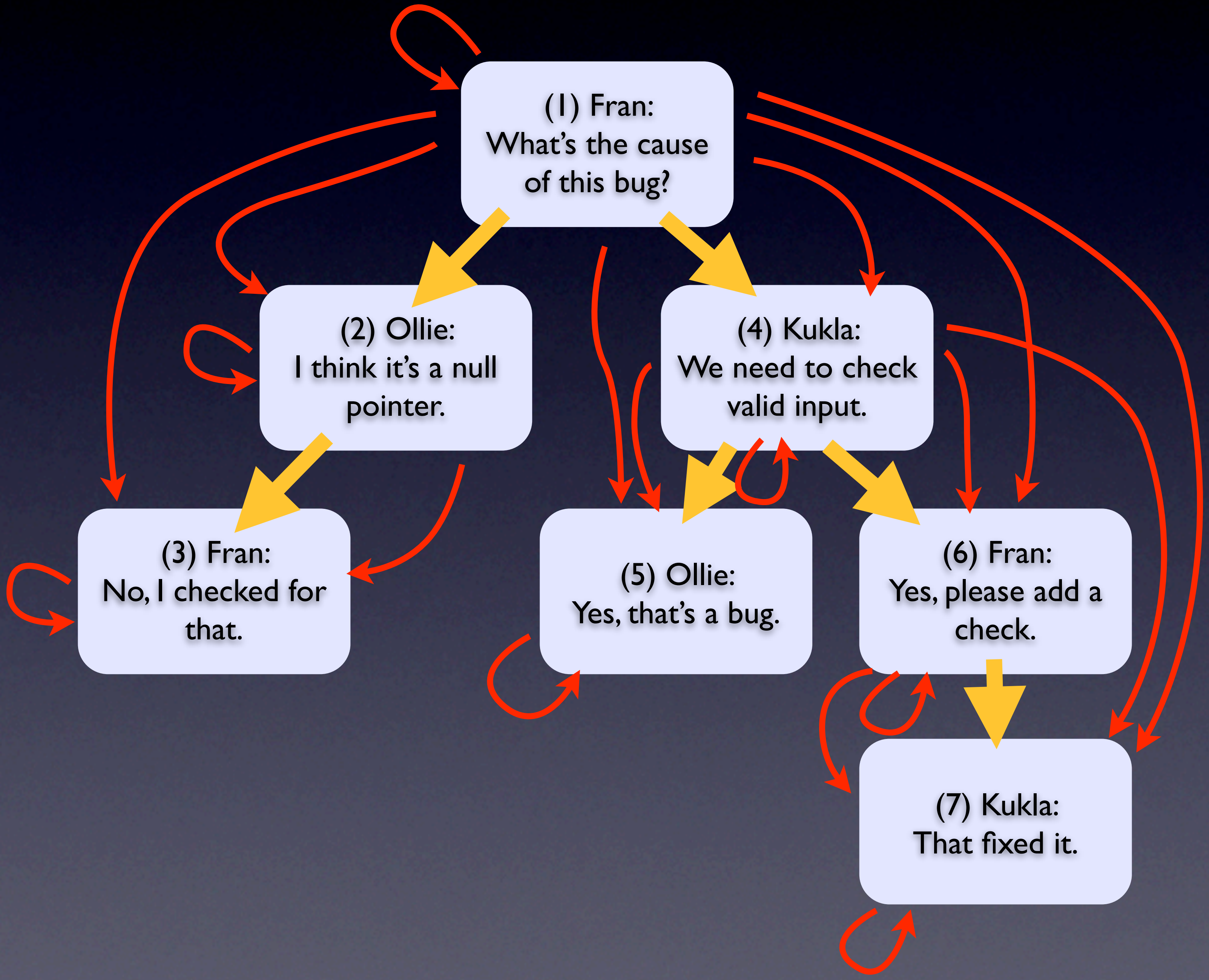
Naive Trees

- **Solution #3:** Closure Table
  - Store every path from ancestors to descendants
  - Requires an additional table:

```
CREATE TABLE TreePaths (
	ancestor	BIGINT NOT NULL,
	descendant	BIGINT NOT NULL,
	PRIMARY KEY (ancestor, descendant),
	FOREIGN KEY(ancestor) REFERENCES Comments(comment_id),
	FOREIGN KEY(descendant) REFERENCES Comments(comment_id),
);
```

Naive Trees

- **Solution #3:** Closure Table


(1) Fran:
What’s the cause of this bug?

(2) Ollie:
I think it’s a null pointer.

(3) Fran:
No, I checked for that.

(4) Kukla:
We need to check valid input.

(5) Ollie:
Yes, that’s a bug.

(6) Fran:
Yes, please add a check.

(7) Kukla:
That fixed it.

Naive Trees

- **Solution #3:** Closure Table

comment_id	author	comment
1	Fran	What’s the cause of this bug?
2	Ollie	I think it’s a null pointer.
3	Fran	No, I checked for that.
4	Kukla	We need to check valid input.
5	Ollie	Yes, that’s a bug.
6	Fran	Yes, please add a check
7	Kukla	That fixed it.

ancestor	descendant
1	1
1	2
1	3
1	4
1	5
1	6
1	7
2	2
2	3
3	3
4	4
4	5
4	6
4	7
5	5
6	6
6	7
7	7

requires $O(n^2)$
rows at most

but far fewer
in practice

Naive Trees

- **Solution #3:** Closure Table
  - Easy to query descendants of comment #4:

```
SELECT c.* FROM Comments c
JOIN TreePaths t
  ON (c.comment_id = t.descendant)
WHERE t.ancestor = 4;
```

Naive Trees

- **Solution #3:** Closure Table
  - Easy to query ancestors of comment #6:

```
SELECT c.* FROM Comments c
JOIN TreePaths t
  ON (c.comment_id = t.ancestor)
WHERE t.descendant = 6;
```

Naive Trees

- **Solution #3:** Closure Table
  - Easy to insert a new child of comment #5:

```
INSERT INTO Comments ...
```

generates comment #8

```
INSERT INTO TreePaths (ancestor, descendant)
    VALUES (8, 8);

INSERT INTO TreePaths (ancestor, descendant)
    SELECT ancestor, 8 FROM TreePaths
    WHERE descendant = 5;
```

Naive Trees

- **Solution #3:** Closure Table
  - Easy to delete a child comment #7:

    ```
    DELETE FROM TreePaths
    WHERE descendant = 7;
    ```

  - Even easier with ON DELETE CASCADE

Naive Trees

- **Solution #3:** Closure Table
  - Easy to delete the subtree under comment #4:

    ```
    DELETE FROM TreePaths WHERE descendant IN
        (SELECT descendant FROM TreePaths
         WHERE ancestor = 4);
    ```

  - For MySQL, use multi-table DELETE:

    ```
    DELETE p FROM TreePaths p
    JOIN TreePaths a USING (descendant)
    WHERE a.ancestor = 4;
    ```

Naive Trees

- **Solution #3:** Closure Table
  - Add a *depth* column to make it easier to query immediate parent or child:

```
SELECT c.* FROM Comments c
JOIN TreePaths t
  ON (c.comment_id = t.descendant)
WHERE t.ancestor = 4
 AND t.depth = 1;
```

ancestor	descendant	depth
1	1	0
1	2	1
1	3	2
1	4	1
1	5	2
1	6	2
1	7	3
2	2	0
2	3	1
3	3	0
4	4	0
4	5	1
4	6	1
4	7	2
5	5	0
6	6	0
6	7	1
7	7	0

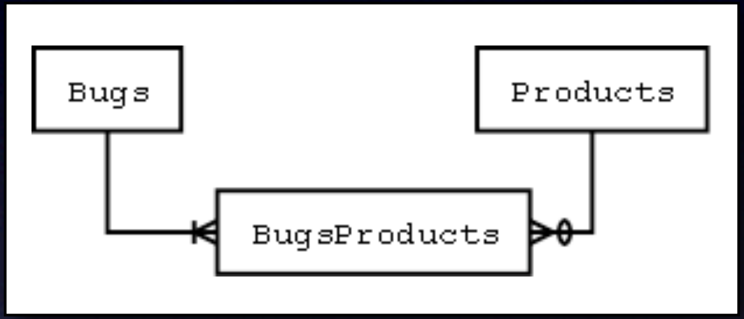
Naive Trees

- Summary of Designs:

Design	Number of Tables	Query Child	Query Subtree	Modify Tree	Referential Integrity
Adjacency List	1	Easy	Hard	Easy	Yes
Path Enumeration	1	Easy	Easy	Hard	No
Nested Sets	1	Hard	Easy	Hard	No
Closure Table	2	Easy	Easy	Easy	Yes

Antipattern Categories

Database Design Antipatterns



Database Creation 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();
```

Database Creation Antipatterns

5. ENUM Antipattern
6. Rounding Errors
7. Indexes Are Magical

ENUM Antipattern

ENUM Antipattern

- **Objective:** restrict a column 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’;
```

- Returns a LONGTEXT you must parse:

```
“ENUM(‘new’, ‘open’, ‘fixed’)”
```

ENUM Antipattern

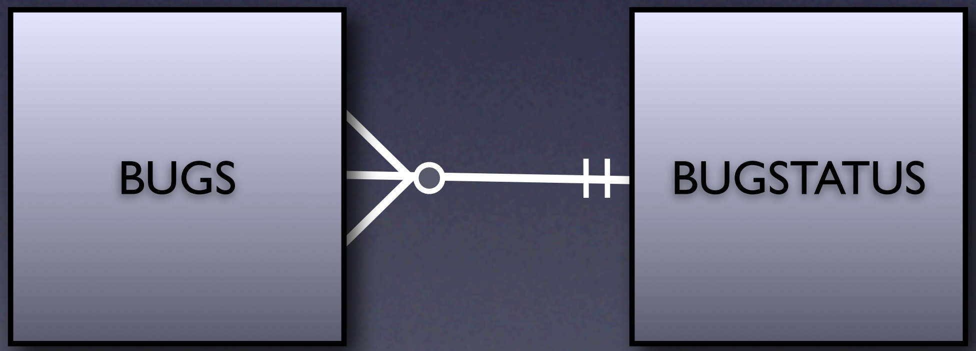
- **Solution:** use ENUM only if values are set in stone

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

ENUM Antipattern

- Use a lookup table if values may change

```
CREATE TABLE BugStatus (
        status  VARCHAR(10) PRIMARY KEY
);

INSERT INTO BugStatus (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 BugStatus (status)
  VALUES (‘DUPLICATE’);
```

ENUM Antipattern

- Use an attribute to retire values, not DELETE

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

UPDATE BugStatus
SET active = 0
WHERE status = ‘DUPLICATE’;
```

Rounding Errors

10.0 times 0.1 is hardly ever 1.0.
— Brian Kernighan

Rounding Errors

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

Rounding Errors

- **Antipattern:** use FLOAT data type

```
ALTER TABLE Bugs
  ADD COLUMN hours FLOAT;

INSERT INTO Bugs (bug_id, hours)
  VALUES (1234, 3.3);
```

Rounding Errors

- FLOAT is inexact

```
SELECT hours FROM Bugs
WHERE bug_id = 1234;
```

```
3.3
```

```
SELECT hours * 1000000000 FROM Bugs
WHERE bug_id = 1234;
```

```
3299999952.3163
```

Rounding Errors

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

assuming infinite precision

finite precision

Rounding Errors

- 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 hours = 3.3;
```

comparison fails

Rounding Errors

- **Solution:** use NUMERIC data type

```
ALTER TABLE Bugs
  ADD COLUMN hours NUMERIC(9,2)

INSERT INTO bugs (bug_id, hours)
  VALUES (1234, 3.3);

SELECT * FROM Bugs
WHERE hours = 3.3;
```

comparison succeeds

Indexes are Magical

Whenever any result is sought, the question will then arise — by what course of calculation can these results be arrived at by the machine in the shortest time?
— Charles Babbage

Indexes are Magical

- **Objective:** execute queries with optimal performance

Indexes are Magical

- **Antipatterns:**
  - Creating indexes blindly
  - Executing non-indexable queries
  - Rejecting indexes because of their overhead

Indexes are Magical

- Creating indexes blindly:

```
CREATE TABLE Bugs (
    bug_id          SERIAL PRIMARY KEY,
    date_reported DATE NOT NULL,
    summary         VARCHAR(80) NOT NULL,
    status          VARCHAR(10) NOT NULL,
    hours           NUMERIC(9,2),
    INDEX (bug_id),
    INDEX (summary),
    INDEX (hours),
    INDEX (bug_id, date_reported, status)
);
```

redundant index

bulky index

unnecessary index

unnecessary covering index

Indexes are Magical

- Executing non-indexable queries:
  - SELECT * FROM Bugs
    WHERE description LIKE ‘%crash%’;
  - SELECT * FROM Bugs
    WHERE MONTH(date_reported) = 4;
  - SELECT * FROM Bugs
    WHERE last_name = “...” OR first_name = “...”;
  - SELECT * FROM Accounts
    ORDER BY first_name, last_name;

*non-leftmost
string match*

*function applied
to column*

*no index spans
all rows*

*non-leftmost
composite key match*

Indexes are Magical

- Telephone book analogy
  - Easy to search for *Dean Thomas:*

```
SELECT * FROM TelephoneBook
WHERE full_name LIKE ‘Thomas, %’;
```

uses index to match

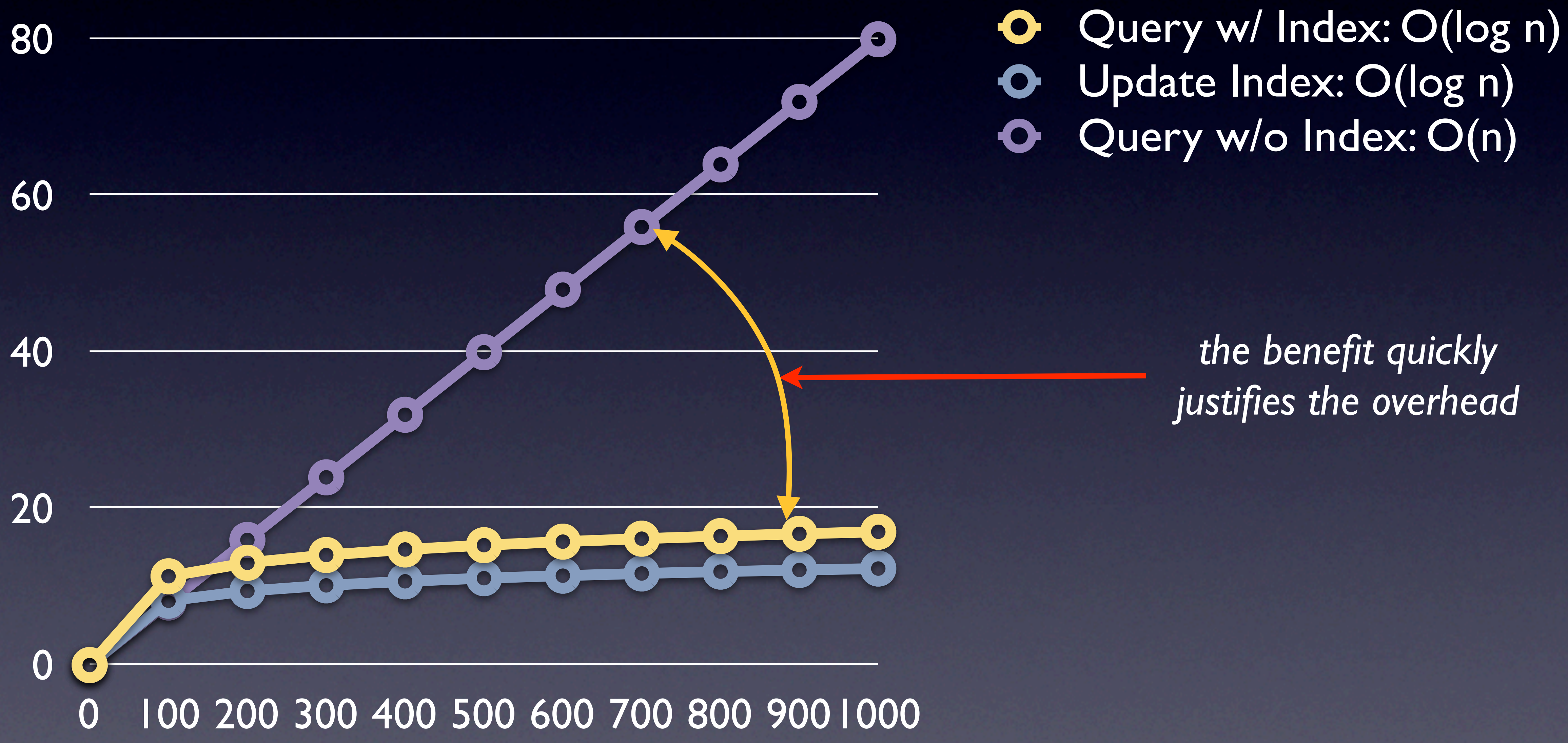
  - Hard to search for *Thomas Riddle:*

```
SELECT * FROM TelephoneBook
WHERE full_name LIKE ‘%, Thomas’;
```

requires full table scan

Indexes are Magical

- Rejecting indexes because of their overhead:



the benefit quickly justifies the overhead

Indexes are Magical

- **Solution:** “MENTOR” your indexes

  Measure

  Explain

  Nominate

  Test

  Optimize

  Repair

Indexes are Magical

- **Solution:** “MENTOR” your indexes

Measure

Explain

Nominate

Test

Optimize

Repair

- Profile your application.
- Focus on the most costly SQL queries:
  - Longest-running.
  - Most frequently run.
  - Blockers, lockers, and deadlocks.

Indexes are Magical

- **Solution:** “MENTOR” your indexes

Measure

Explain

Nominate

Test

Optimize

Repair

- Analyze the optimization plan of costly queries, e.g. MySQL’s EXPLAIN
- Identify tables that aren’t using indexes:
  - Temporary table
  - Filesort

Indexes are Magical

- **Solution:** “MENTOR” your indexes

Measure

Explain

Nominate

Test

Optimize

Repair

- Could an index improve access to these tables?
  - ORDER BY criteria
  - MIN() / MAX()
  - WHERE conditions
- Which column(s) need indexes?

Indexes are Magical

- **Solution:** "MENTOR" your indexes

Measure

Explain

Nominate

Test

Optimize

Repair

- After creating index, measure again.
- Confirm the new index made a difference.
- Impress your boss!

  "The new index gave a 27% performance improvement!"

Indexes are Magical

- **Solution:** “MENTOR” your indexes

Measure

Explain

Nominate

Test

Optimize

Repair

- Indexes are compact, frequently-used data.
- Try to cache indexes in memory:
  - MyISAM: key_buffer_size, LOAD INDEX INTO CACHE
  - InnoDB: innodb_buffer_pool_size

Indexes are Magical

- **Solution:** “MENTOR” your indexes

Measure

Explain

Nominate

Test

Optimize

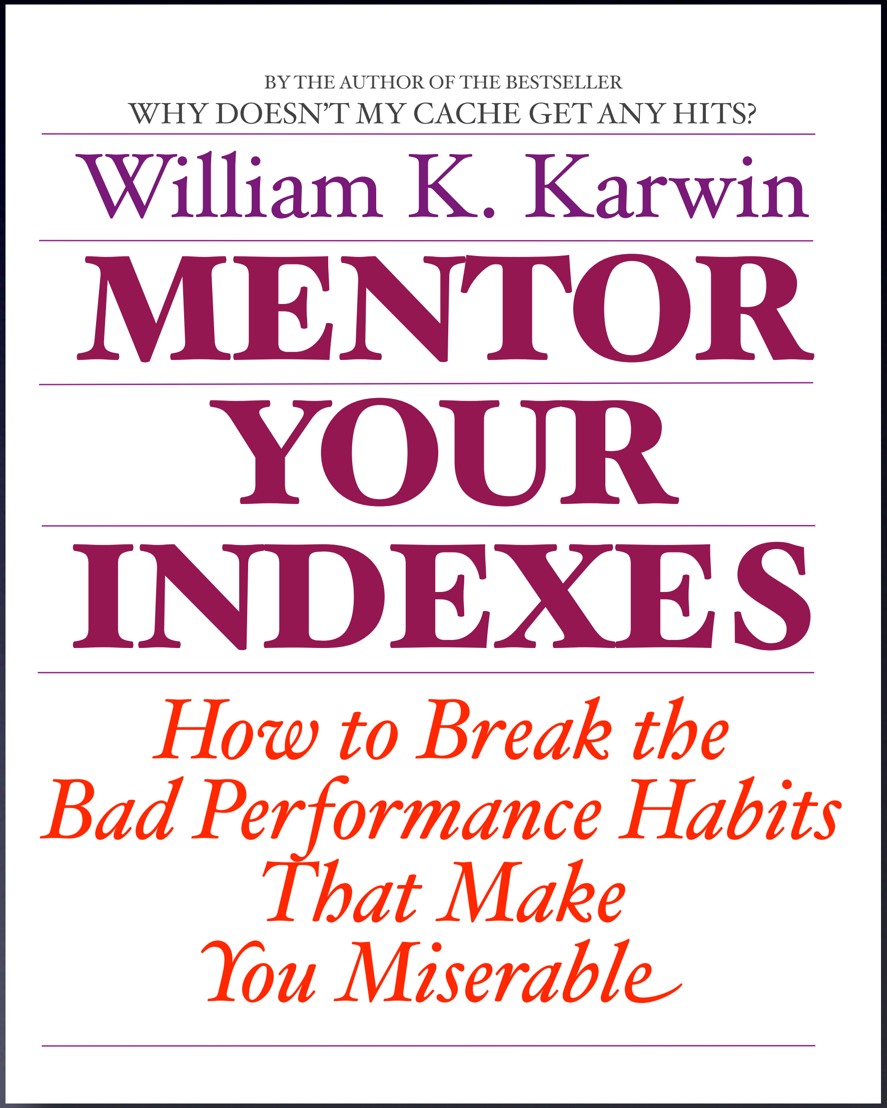
Repair

- Indexes require periodic maintenance.
- Like a filesystem requires periodic defragmentation.
- Analyze or rebuild indexes, e.g. in MySQL:
  - ANALYZE TABLE
  - OPTIMIZE TABLE

Indexes are Magical

- **Solution:** “MENTOR” your indexes
  - Sounds like the name of a “self-help” book!

just kidding!
please don’t ask
when it’s coming out!



BY THE AUTHOR OF THE BESTSELLER
WHY DOESN’T MY CACHE GET ANY HITS?

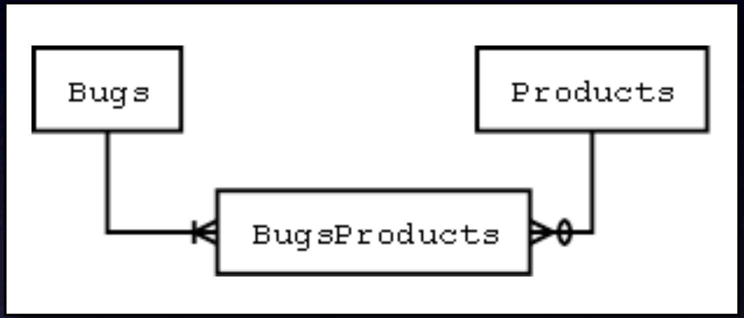
William K. Karwin

MENTOR
YOUR
INDEXES

How to Break the
Bad Performance Habits
That Make
You Miserable

Antipattern Categories

Database Design Antipatterns



Database Creation 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

8. NULL antipatterns
9. Ambiguous Groups
10. Random Order
11. JOIN antipattern
12. Goldberg Machine

NULL Antipatterns

As we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns — the ones we don't know we don't know.

— Donald Rumsfeld

NULL Antipatterns

- **Objective:** handle “missing” values, store them as missing, and support them in queries.

NULL Antipatterns

- **Antipatterns:**
  - Use NULL as an ordinary value
  - Use an ordinary value as NULL

NULL Antipatterns

- Using NULL in most expressions results in an *unknown* value.

SELECT NULL + 10;

NULL is not zero

SELECT ‘Bill’ || NULL;

NULL is not an empty string

SELECT FALSE OR NULL;

NULL is not FALSE

NULL Antipatterns

- The opposite of *unknown* is still *unknown.*

```
SELECT * FROM Bugs
WHERE assigned_to = 123;
```

```
SELECT * FROM Bugs
WHERE NOT (assigned_to = 123);
```

which query returns bugs that are not yet assigned?

neither query!

NULL Antipatterns

- Choosing an ordinary value in lieu of NULL:

```
UPDATE Bugs SET assigned_to = -1
WHERE assigned_to IS NULL;
```

assigned_to is a foreign key so this value doesn't work

NULL Antipatterns

- Choosing an ordinary value in lieu of NULL:

```
UPDATE Bugs SET hours = -1
WHERE hours IS NULL;

SELECT SUM(hours)
FROM Bugs
WHERE status = ‘OPEN’
AND hours <> -1;
```

this makes SUM() inaccurate

special-case code you were trying to avoid by prohibiting NULL

NULL Antipatterns

- Choosing an ordinary value in lieu of NULL:
  - Any given value may be significant in a column
  - Every column needs a different value
  - You need to remember or document the value used for “missing” on a case-by-case basis

NULL Antipatterns

- **Solution:** use NULL appropriately
  - NULL signifies “missing” or “inapplicable”
  - Works for every data type
  - Already standard and well-understood

NULL Antipatterns

- Understanding NULL in expressions

Expression	Expected	Actual
NULL = 0	TRUE	Unknown
NULL = 12345	FALSE	Unknown
NULL <> 12345	TRUE	Unknown
NULL + 12345	12345	Unknown
NULL || ‘string’	string’	Unknown
NULL = NULL	TRUE	Unknown
NULL <> NULL	FALSE	Unknown

NULL Antipatterns

- Understanding NULL in boolean expressions

Expression	Expected	Actual
NULL AND TRUE	FALSE	Unknown
NULL AND FALSE	FALSE	FALSE
NULL OR FALSE	FALSE	Unknown
NULL OR TRUE	TRUE	TRUE
NOT (NULL)	TRUE	Unknown

NULL Antipatterns

- SQL supports IS NULL predicate that returns *true* or *false*, never *unknown*:

```
SELECT * FROM Bugs
WHERE assigned_to IS NULL;

SELECT * FROM Bugs
WHERE assigned_to IS NOT NULL;
```

NULL Antipatterns

- SQL-99 supports IS DISTINCT FROM predicate that returns *true* or *false*:

```
SELECT * FROM Bugs
WHERE assigned_to IS DISTINCT FROM 123;
```

```
SELECT * FROM Bugs
WHERE assigned_to IS NOT DISTINCT FROM 123;
```

```
SELECT * FROM Bugs
WHERE assigned_to <=> 123;
```

MySQL operator works like IS NOT DISTINCT FROM

NULL Antipatterns

- Change NULL to ordinary value on demand with COALESCE():

```
SELECT COALESCE(
  first_name || ‘ ’ || middle_initial || ‘ ’ || last_name,
  first_name || ‘ ’ || last_name) AS full_name
FROM Accounts;
```

- Also called NVL() or ISNULL()
  in some database brands.

Ambiguous Groups

Please accept my resignation. I don't want to belong to any club that will accept me as a member.
— Groucho Marx

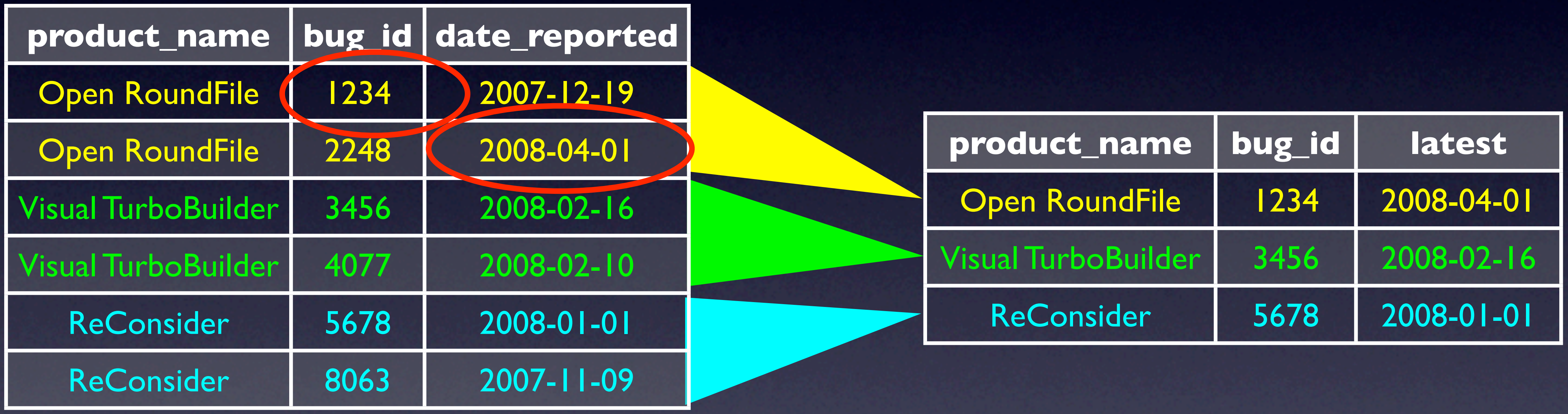
Ambiguous Groups

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

```
SELECT product_name, bug_id,
    MAX(date_reported) AS latest
FROM Bugs
GROUP BY product_name;
```

Ambiguous Groups

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



product_name	bug_id	date_reported
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 Groups

```
SELECT product_name, bug_id,
    MAX(date_reported) AS latest
FROM Bugs
GROUP BY product_name;
```

assume bug_id from the same row with MAX(date_reported)

Ambiguous Groups

```
SELECT product_name, bug_id,
    MAX(date_reported) AS latest
FROM Bugs
GROUP BY product_name;
```

what if two bug_id both match the latest date?

Ambiguous Groups

```
SELECT product_name, bug_id,
    MIN(date_reported) AS earliest,
    MAX(date_reported) AS latest
FROM Bugs
GROUP BY product_name;
```

what bug_id
has both the earliest
and the latest date?

Ambiguous Groups

```
SELECT product_name, bug_id,
    AVG(date_reported) AS mean
FROM Bugs
GROUP BY product_name;
```

what if no bug_id matches this date?

Ambiguous Groups

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

Ambiguous Groups

- For a given product_name, there is a single value in each *functionally dependent* attribute.

product_name	bug_id	date_reported
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 Groups

- **Solution #1:** use only functionally dependent attributes:

```
SELECT product_name, ~~bug_id~~,
    MAX(date_reported) AS latest
FROM Bugs
GROUP BY product_name;
```

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

Ambiguous Groups

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

```
SELECT b.product_name, b.bug_id, m.latest
FROM Bugs b
JOIN (SELECT product_name, MAX(date_reported) AS latest
      FROM Bugs GROUP BY product_name) m
  ON (b.product_name = m.product_name
     AND b.date_reported = m.latest);
```

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

Ambiguous Groups

- **Solution #3:** use an outer JOIN:

```
SELECT b1.product_name, b1.bug_id,
    b1.date_reported AS latest
FROM Bugs b1 LEFT OUTER JOIN Bugs b2
    ON (b1.product_name = b2.product_name
        AND b1.date_reported < b2.date_reported)
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

Ambiguous Groups

- **Solution #4:** use another aggregate:

```
SELECT product_name, MAX(date_reported) AS latest,
    MAX(bug_id) AS latest_bug_id
FROM Bugs
GROUP BY product_name;
```

if bug_id increases in chronological order

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

Ambiguous Groups

- **Solution #5:** use GROUP_CONCAT():

```
SELECT product_name,
     GROUP_CONCAT(bug_id) AS bug_id_list,
     MAX(date_reported) AS latest
FROM Bugs
GROUP BY product_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

Random Order

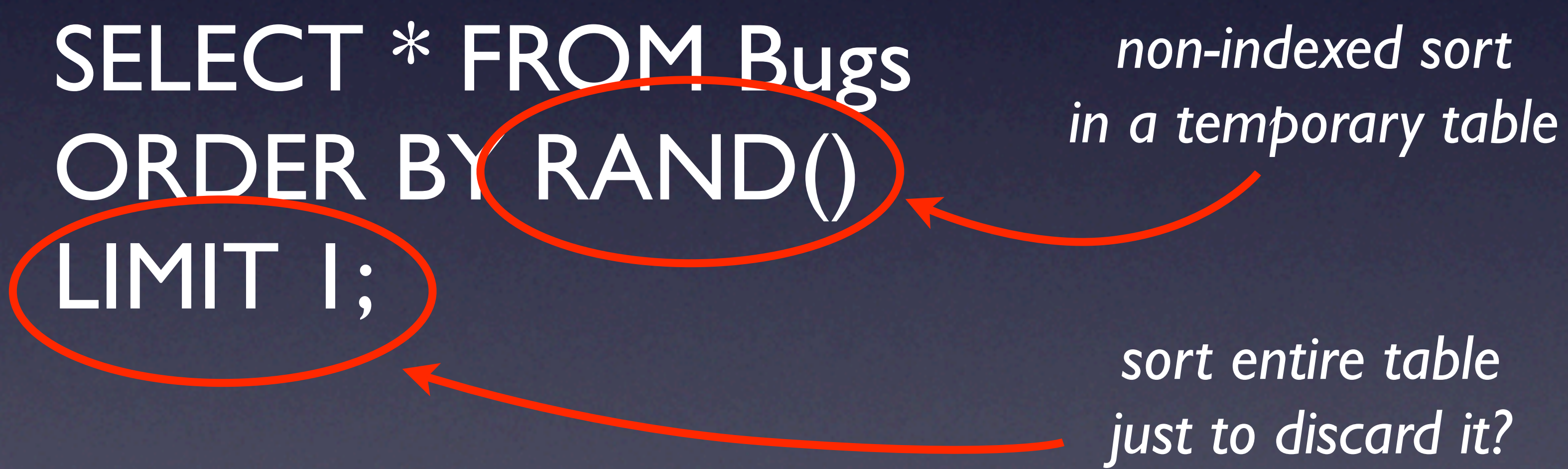
I must complain the cards are ill shuffled till I have a good hand.
— Jonathan Swift

Random Order

- **Objective:** select a random row

Random Order

- **Antipattern:** sort by random expression, then return top row(s)



```
SELECT * FROM Bugs
ORDER BY RAND()
LIMIT 1;
```

non-indexed sort in a temporary table

sort entire table just to discard it?

Random Order

- **Solution #1:** pick random primary key from list of all values:

```
$bug_id_list = $pdo->query(
      ‘SELECT bug_id FROM Bugs’ )->fetchAll();

$rand = random(count($bug_id_list));

$stmt = $pdo->prepare(
      ‘SELECT * FROM Bugs WHERE bug_id = ?’);
$stmt->execute( $bug_id_list[$rand][0] );
$rand_bug = $stmt->fetch();
```

Random Order

- **Solution #1:** pick random primary key from list of all values:

```
$bug_id_list = $pdo->query(
    'SELECT bug_id FROM Bugs' )->fetchAll();
```

- $bug_id_list may grow to an impractical size:

Fatal error: Allowed memory size of 16777216 bytes exhausted

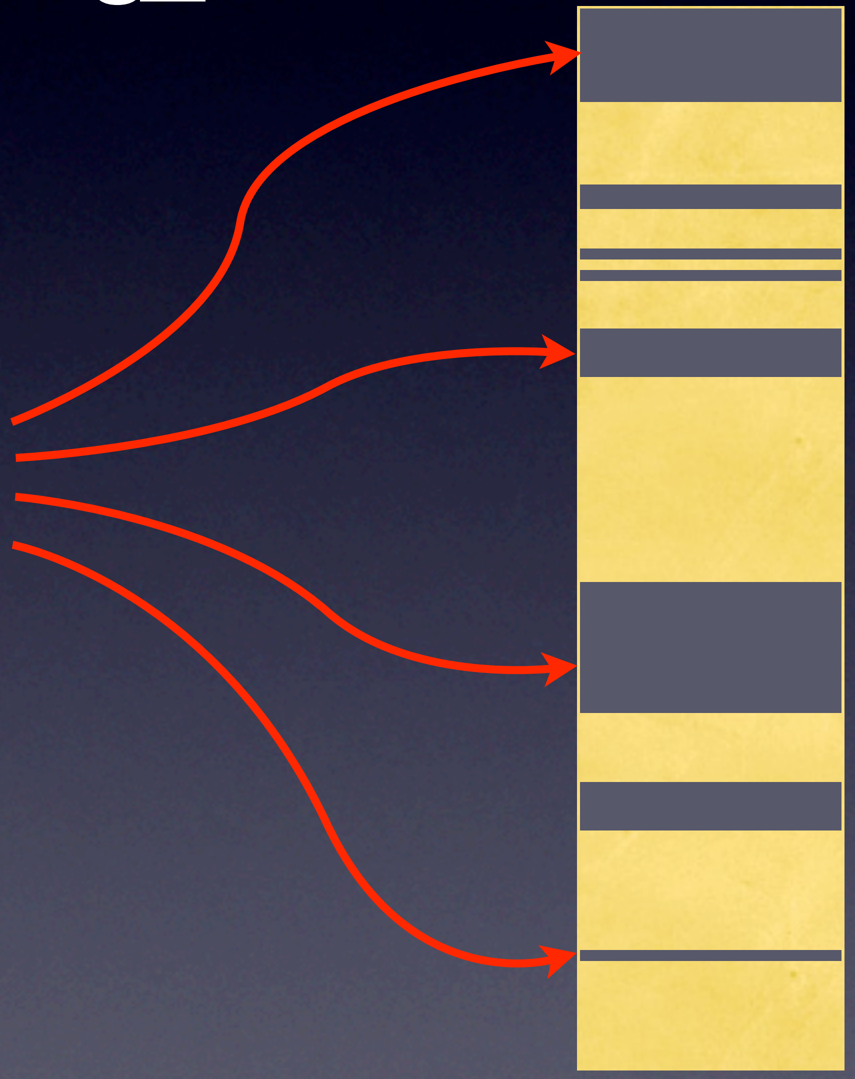
Random Order

- **Solution #2:** pick random value between 1...MAX(bug_id); use that bug_id:

```
SELECT b1.* FROM Bugs b1
JOIN (SELECT CEIL(RAND() *
     (SELECT MAX(bug_id) FROM Bugs)) rand_id) b2
  ON (b1.bug_id = b2.rand_id);
```

Random Order

- **Solution #2:** pick random value between 1...MAX(bug_id); use that bug_id:
  - Assumes bug_id starts at 1 and values are contiguous.
  - If there are gaps, a random bug_id may not match an existing bug.



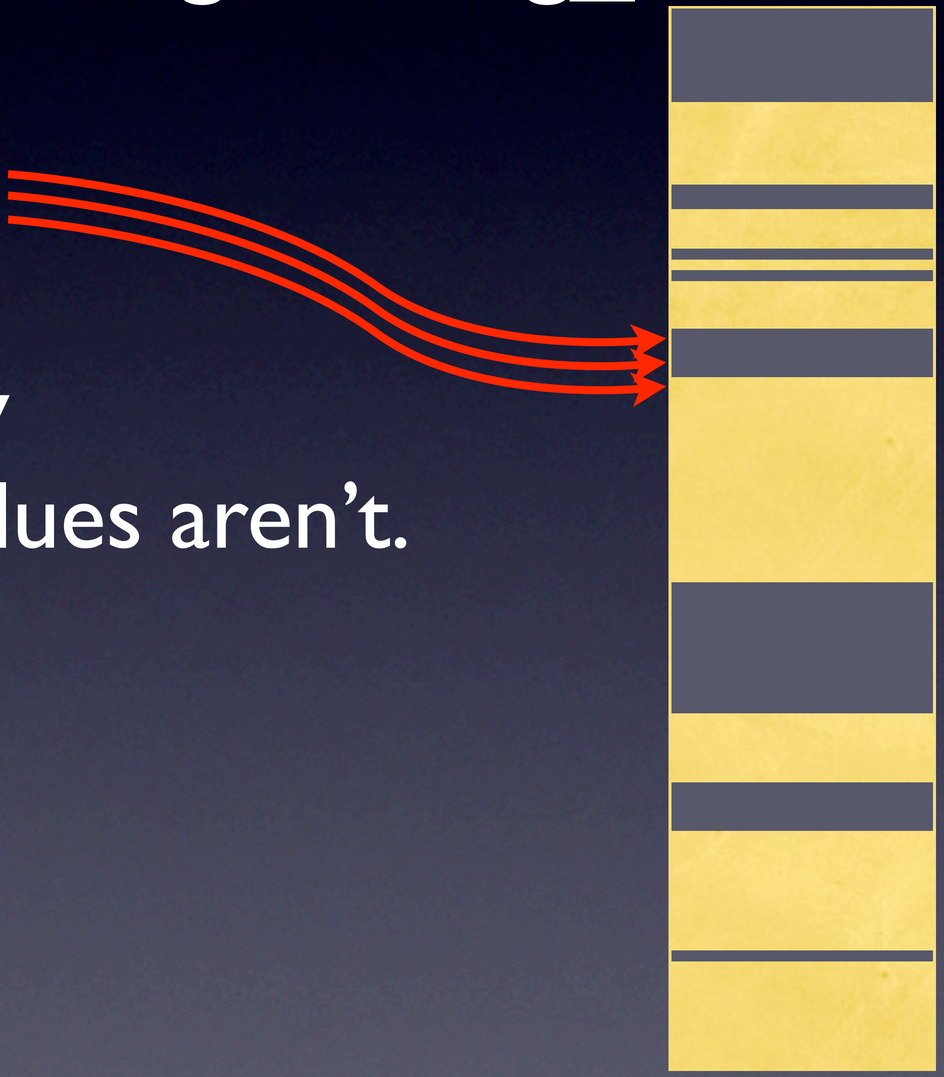
Random Order

- **Solution #3:** pick random value between 1...MAX(bug_id); use next higher bug_id:

```
SELECT b1.* FROM Bugs b1
JOIN (SELECT CEIL(RAND() *
    (SELECT MAX(bug_id) FROM Bugs)) AS bug_id) b2
WHERE b1.bug_id >= b2.bug_id
ORDER BY b1.bug_id
LIMIT 1;
```

Random Order

- **Solution #3:** pick random value between 1...MAX(bug_id); use next higher bug_id:
  - bug_id values after gaps are chosen more often.
  - Random values are evenly distributed, but bug_id values aren't.



Random Order

- **Solution #4:** pick random row from 0...COUNT, regardless of bug_id values:

```
$offset = $pdo->query(
    ‘SELECT ROUND(RAND() *
    (SELECT COUNT(*) FROM Bugs))’ )->fetch();
$sql = ‘SELECT * FROM Bugs LIMIT 1 OFFSET ?’;
$stmt = $pdo->prepare( $sql );
$stmt->execute( $offset );
```

JOIN Antipattern

JOIN Antipattern

- **Objective:** Design optimal queries.

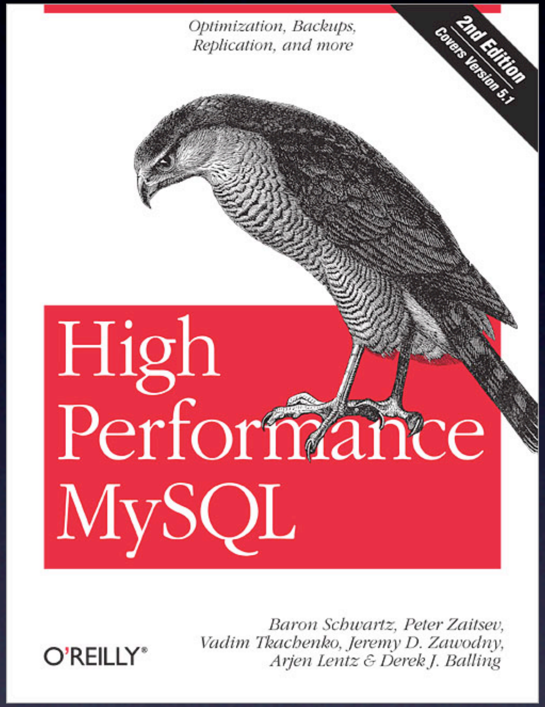
JOIN Antipattern

- **Antipatterns:**
  - Senseless avoidance of JOIN.
  - Overzealous JOIN decomposition.
  - *“Joins are slow!”*

compared to what?

JOIN Antipattern

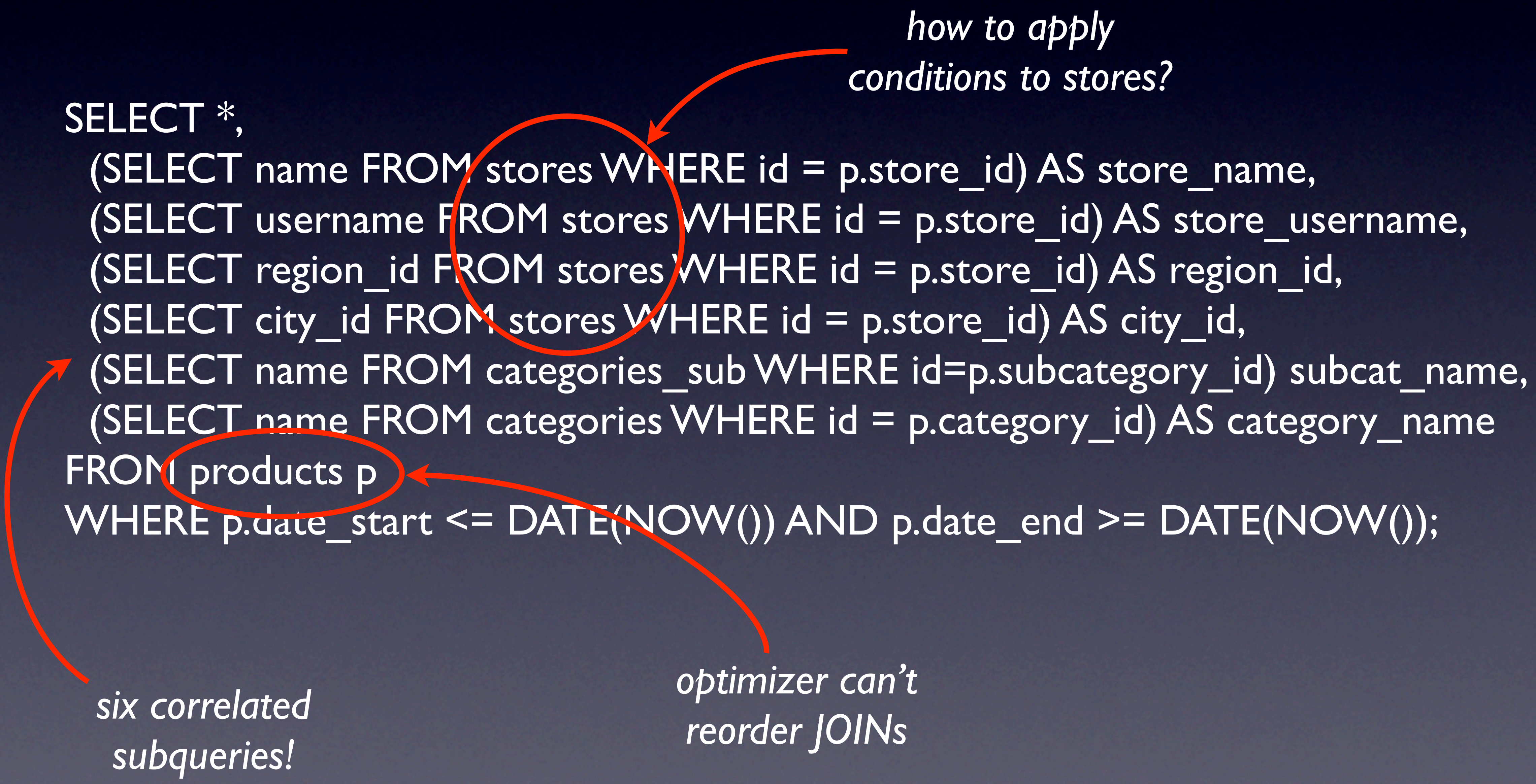
- Reasons for JOIN decomposition:
  - Cache and reuse earlier results
  - Reduce locking across multiple tables
  - Distribute tables across servers
  - Leverage IN() optimization
  - Reduce redundant rows
    (result sets are denormalized)
- Notice these are *exception cases!*


Optimization, Backups, Replication, and more
2nd Edition
Covers Version 5.1
High Performance MySQL
O'REILLY
Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz & Derek J. Balling

borrowed from this book

JOIN Antipattern

- Example from the web (2009-4-18):



how to apply conditions to stores?

```
SELECT *,
  (SELECT name FROM stores WHERE id = p.store_id) AS store_name,
  (SELECT username FROM stores WHERE id = p.store_id) AS store_username,
  (SELECT region_id FROM stores WHERE id = p.store_id) AS region_id,
  (SELECT city_id FROM stores WHERE id = p.store_id) AS city_id,
  (SELECT name FROM categories_sub WHERE id=p.subcategory_id) subcat_name,
  (SELECT name FROM categories WHERE id = p.category_id) AS category_name
FROM products p
WHERE p.date_start <= DATE(NOW()) AND p.date_end >= DATE(NOW());
```

six correlated subqueries!

optimizer can't reorder JOINs

JOIN Antipattern

- Example revised with JOINs:

```
SELECT p.*, s.name AS store_name, s.username AS store_username,
       s.region_id,  s.city_id, cs.name AS subcategory_name,
       c.name AS category_name
FROM products p
  JOIN stores s ON (s.id = p.store_id)
  JOIN categories c ON (c.id = p.category_id)
  JOIN categories_sub cs ON (cs.id = p.subcategory_id)
WHERE p.date_start <= DATE(NOW()) AND p.date_end >= DATE(NOW())
  AND s.store_category = ‘preferred’;
```

easier to optimize

easier to apply conditions

JOIN Antipattern

- Example: find an entry with three tags: HAVING COUNT solution:

```
SELECT b.*
FROM Bugs b
  JOIN BugsProducts p ON (b.bug_id = p.bug_id)
WHERE p.product_id IN (1, 2, 3)
GROUP BY b.bug_id
HAVING COUNT(*) = 3;
```

must match all three products

JOIN Antipattern

- Example: find an entry with three tags:: multiple-JOIN solution:

```
SELECT DISTINCT b.*
FROM Bugs b
  JOIN BugsProducts p1 ON ((p1.bug_id, p1.product_id) = (b.bug_id, 1))
  JOIN BugsProducts p2 ON ((p2.bug_id, p2.product_id) = (b.bug_id, 2))
  JOIN BugsProducts p3 ON ((p3.bug_id, p3.product_id) = (b.bug_id, 3));
```

three joins is slower than one, right?

not if indexes are used well

JOIN Antipattern

- **Solution:**
  - JOIN is to SQL as *while()* is to other languages.
  - One-size-fits-all rules (e.g. "joins are slow") don't work.
  - Measure twice, query once.
  - Let the SQL optimizer work.
  - Employ alternatives (e.g. JOIN decomposition) as exception cases.

Goldberg Machine

Enita non sunt multiplicanda praeter necessitatem
("Entities are not to be multiplied beyond necessity").
—William of Okham

Goldberg Machine

- **Objective:** Generate a complex report as efficiently as possible.

Goldberg Machine

- Example: Calculate for each account:
  - Count of bugs reported by user.
  - Count of products the user has been assigned to.
  - Count of comments left by user.

Goldberg Machine

- **Antipattern:** Try to generate all the information for the report in a single query:

```
SELECT a.account_name,
       COUNT(br.bug_id) AS bugs_reported,
       COUNT(bp.product_id) AS products_assigned,
       COUNT(c.comment_id) AS comments
FROM Accounts a
  LEFT JOIN Bugs br ON (a.account_id = br.reported_by)
  LEFT JOIN (Bugs ba JOIN BugsProducts bp ON (ba.bug_id = bp.bug_id))
       ON (a.account_id = ba.assigned_to)
  LEFT JOIN Comments c ON (a.account_id = c.author)
GROUP BY a.account_id;
```

expected: 3

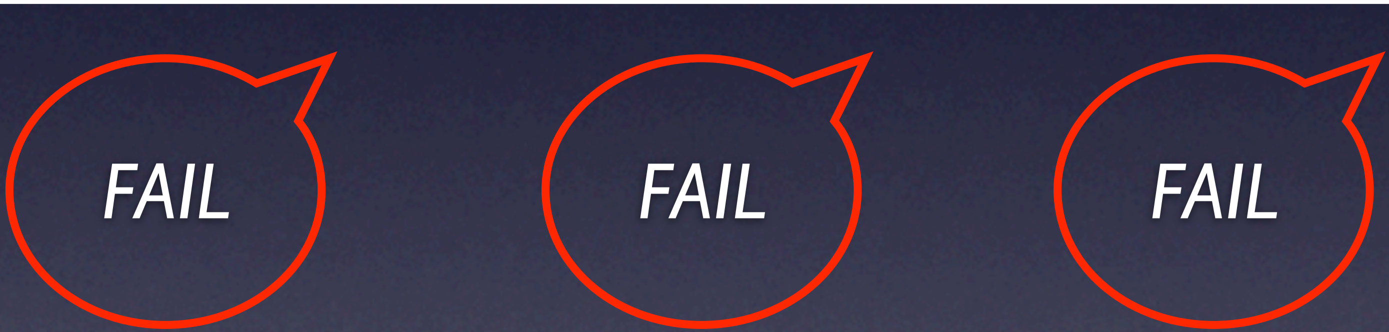
expected: 2

expected: 4

Goldberg Machine

- Expected result versus actual result:

account name	bugs reported	products assigned	comments
Bill	~~3~~ 48	~~2~~ 48	~~4~~ 48



FAIL

FAIL

FAIL

Goldberg Machine

- Run query without GROUP BY:

```
SELECT a.account_name,
       br.bug_id AS bug_reported,
       ba.bug_id AS bug_assigned,
       bp.product_id AS product_assigned
       c.comment_id
FROM Accounts a
  LEFT JOIN Bugs br ON (a.account_id = br.reported_by)
  LEFT JOIN (Bugs ba JOIN BugsProducts bp ON (ba.bug_id = bp.bug_id))
       ON (a.account_id = ba.assigned_to)
  LEFT JOIN Comments c ON (a.account_id = c.author);
```

Goldberg Machine

- Query result reveals a *Cartesian Product:*

account name	bug reported	bug assigned	product assigned	comment
Bill	1234	1234	1	6789
Bill	1234	1234	1	9876
Bill	1234	1234	1	4365
Bill	1234	1234	1	7698
Bill	1234	1234	3	6789
Bill	1234	1234	3	9876
Bill	1234	1234	3	4365
Bill	1234	1234	3	7698

Goldberg Machine

- Query result reveals a *Cartesian Product:*

account name	bug reported	bug assigned	product assigned	comment
Bill	1234	5678	1	6789
Bill	1234	5678	1	9876
Bill	1234	5678	1	4365
Bill	1234	5678	1	7698
Bill	1234	5678	3	6789
Bill	1234	5678	3	9876
Bill	1234	5678	3	4365
Bill	1234	5678	3	7698

Goldberg Machine

- Query result reveals a *Cartesian Product:*

account name	bug reported	bug assigned	product assigned	comment
Bill	2345	1234	1	6789
Bill	2345	1234	1	9876
Bill	2345	1234	1	4365
Bill	2345	1234	1	7698
Bill	2345	1234	3	6789
Bill	2345	1234	3	9876
Bill	2345	1234	3	4365
Bill	2345	1234	3	7698

Goldberg Machine

- Query result reveals a *Cartesian Product:*

account name	bug reported	bug assigned	product assigned	comment
Bill	2345	5678	1	6789
Bill	2345	5678	1	9876
Bill	2345	5678	1	4365
Bill	2345	5678	1	7698
Bill	2345	5678	3	6789
Bill	2345	5678	3	9876
Bill	2345	5678	3	4365
Bill	2345	5678	3	7698

Goldberg Machine

- Query result reveals a *Cartesian Product:*

account name	bug reported	bug assigned	product assigned	comment
Bill	3456	1234	1	6789
Bill	3456	1234	1	9876
Bill	3456	1234	1	4365
Bill	3456	1234	1	7698
Bill	3456	1234	3	6789
Bill	3456	1234	3	9876
Bill	3456	1234	3	4365
Bill	3456	1234	3	7698

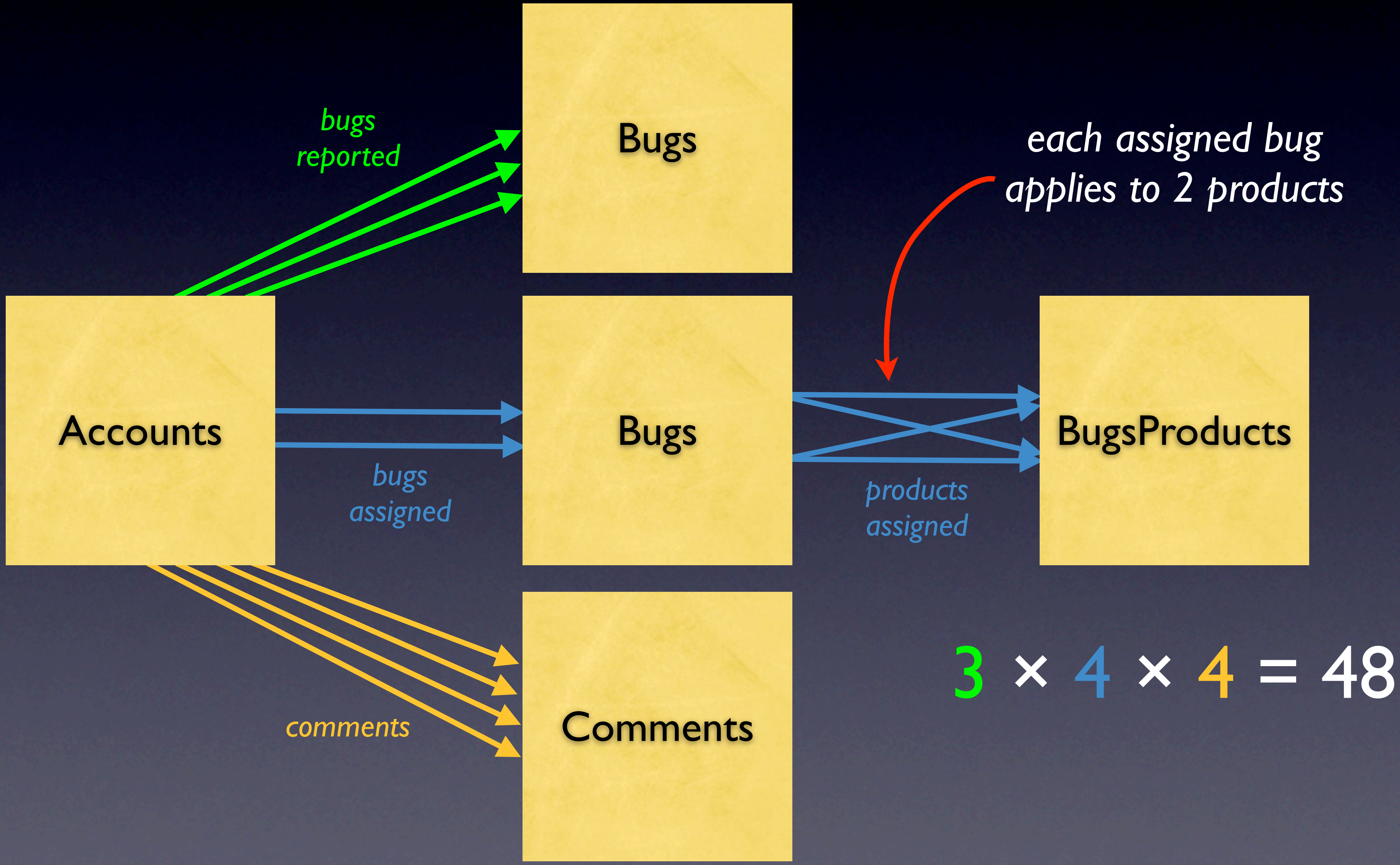
Goldberg Machine

- Query result reveals a *Cartesian Product:*

account name	bug reported	bug assigned	product assigned	comment
Bill	3456	5678	1	6789
Bill	3456	5678	1	9876
Bill	3456	5678	1	4365
Bill	3456	5678	1	7698
Bill	3456	5678	3	6789
Bill	3456	5678	3	9876
Bill	3456	5678	3	4365
Bill	3456	5678	3	7698

Goldberg Machine

- Visualizing a Cartesian Product:



172

Goldberg Machine

- **Solution:** Write separate queries.

result: 3

```
SELECT a.account_name, COUNT(br.bug_id) AS bugs_reported
FROM Accounts a LEFT JOIN Bugs br ON (a.account_id = br.reported_by)
GROUP BY a.account_id;
```

result: 2

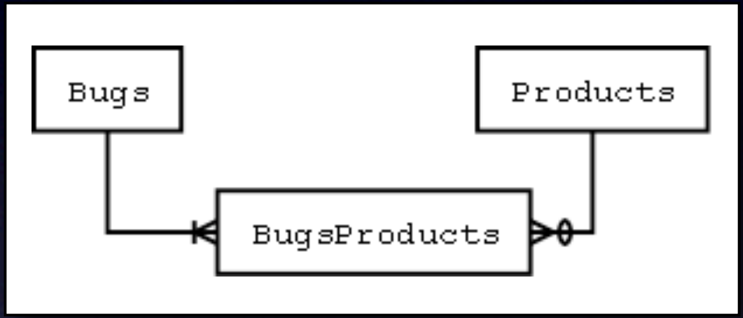
```
SELECT a.account_name,
    COUNT(DISTINCT bp.product_id) AS products_assigned,
FROM Accounts a LEFT JOIN
    (Bugs ba JOIN BugsProducts bp ON (ba.bug_id = bp.bug_id))
    ON (a.account_id = ba.assigned_to)
GROUP BY a.account_id;
```

result: 4

```
SELECT a.account_name, COUNT(c.comment_id) AS comments
FROM Accounts a LEFT JOIN Comments c ON (a.account_id = c.author)
GROUP BY a.account_id;
```

Antipattern Categories

Database Design Antipatterns



Database Creation 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

13. Parameter Facade
14. Phantom Side Effects
15. See No Evil
16. Diplomatic Immunity
17. Magic Beans

Parameter Facade

Parameter Facade

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

```
SELECT * FROM Bugs
WHERE bug_id IN ( $id_list );
```

Parameter Facade

- **Antipattern:** Trying to use parameters for complex syntax

Parameter Facade

- 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 Facade

- A parameter is always a single value

```
$list = ‘1234, 3456, 5678’

SELECT * FROM Bugs
WHERE bug_id IN ( ? );

EXECUTE USING $list;

SELECT * FROM Bugs
WHERE bug_id IN (‘1234, 3456, 5678’ );
```

one string value

Parameter Facade

- Interpolation can specify identifiers

```
$column = ‘bug_id’

SELECT * FROM Bugs
WHERE $column = 1234;

SELECT * FROM Bugs
WHERE bug_id = 1234;
```

column identifier

Parameter Facade

- 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 Facade

- 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 Facade

- A parameter is always a single value

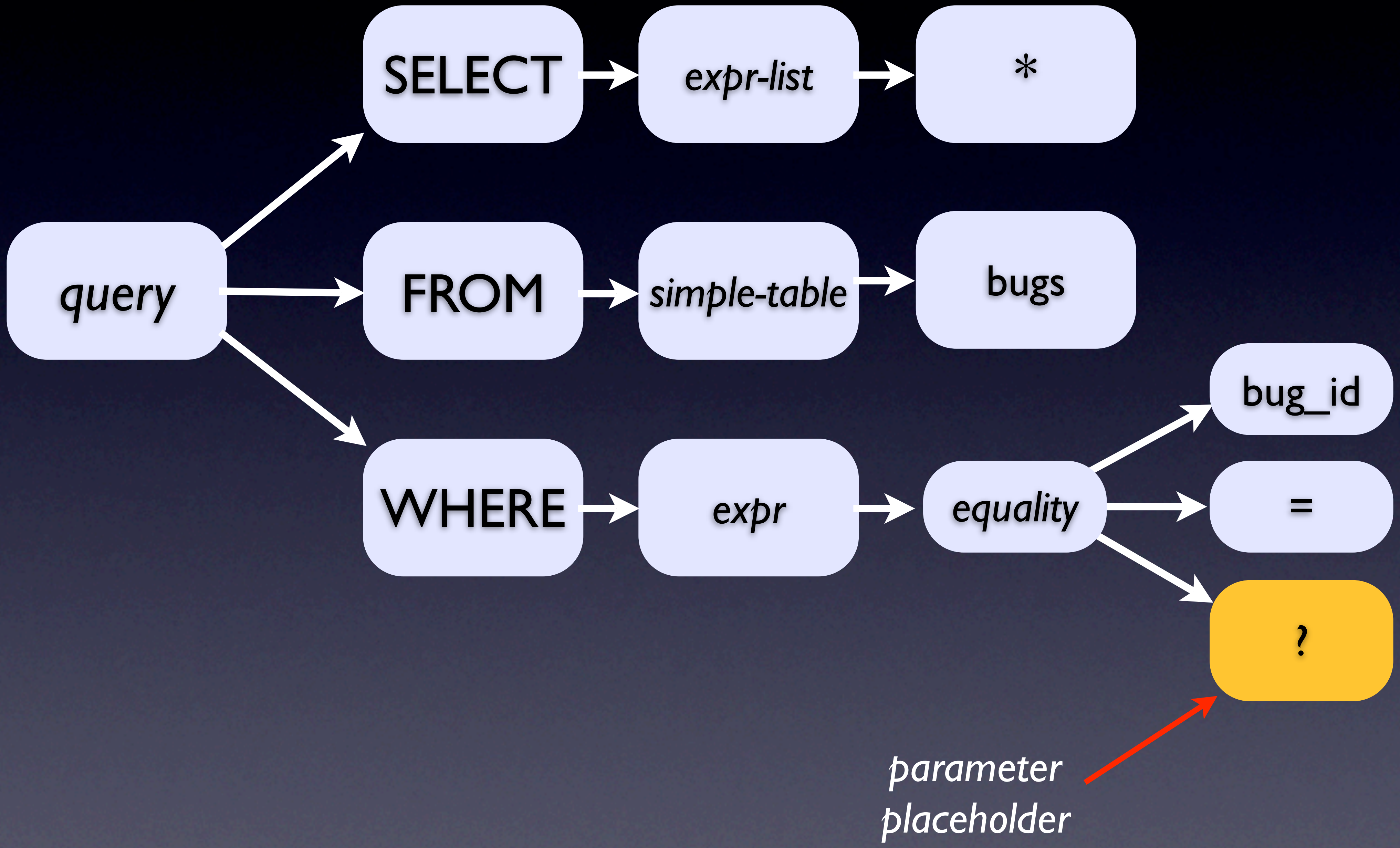
```
$id = ‘1234 or 1=1’;

SELECT * FROM Bugs
WHERE bug_id = ?;

EXECUTE USING $id;

SELECT * FROM Bugs
WHERE bug_id = ‘1234 or 1=1’;
```

one string value

Parameter Facade

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

Parameter Facade

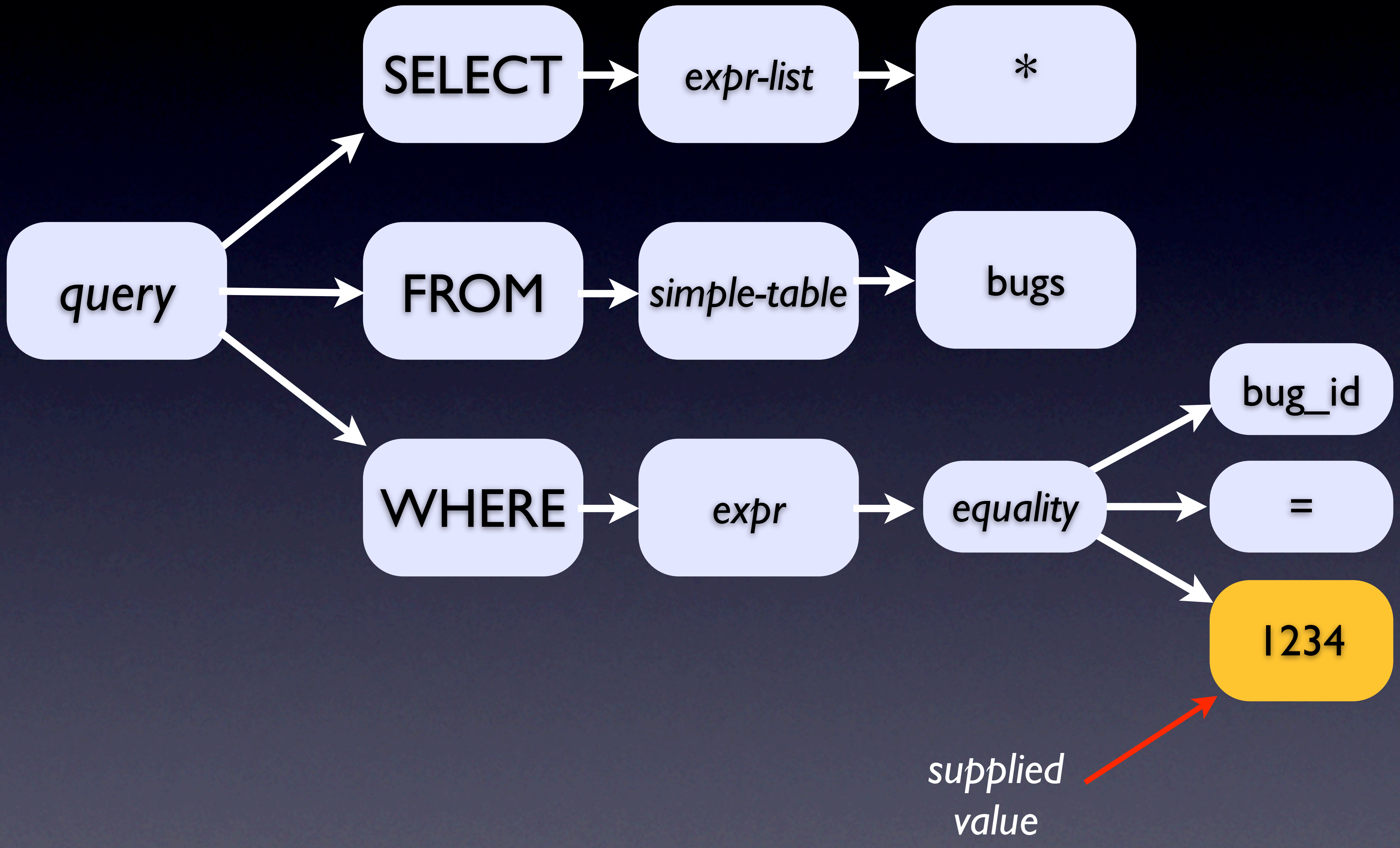


Parameter Facade

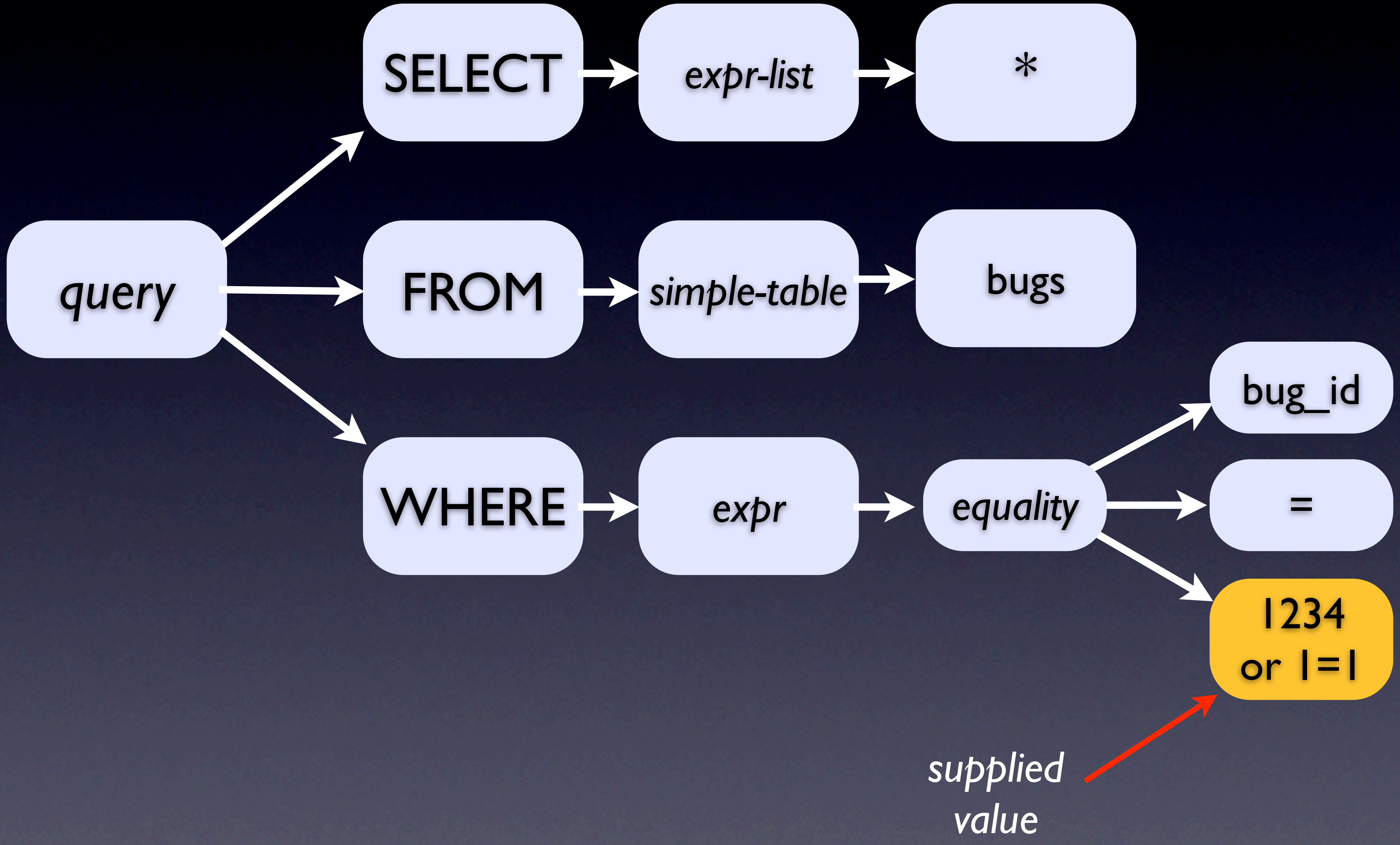
- Executing a prepared statement
  - Combines a supplied value for each parameter
  - *Doesn’t* modify syntax, tables, or columns
  - Runs query

could invalidate optimization plan

Parameter Facade



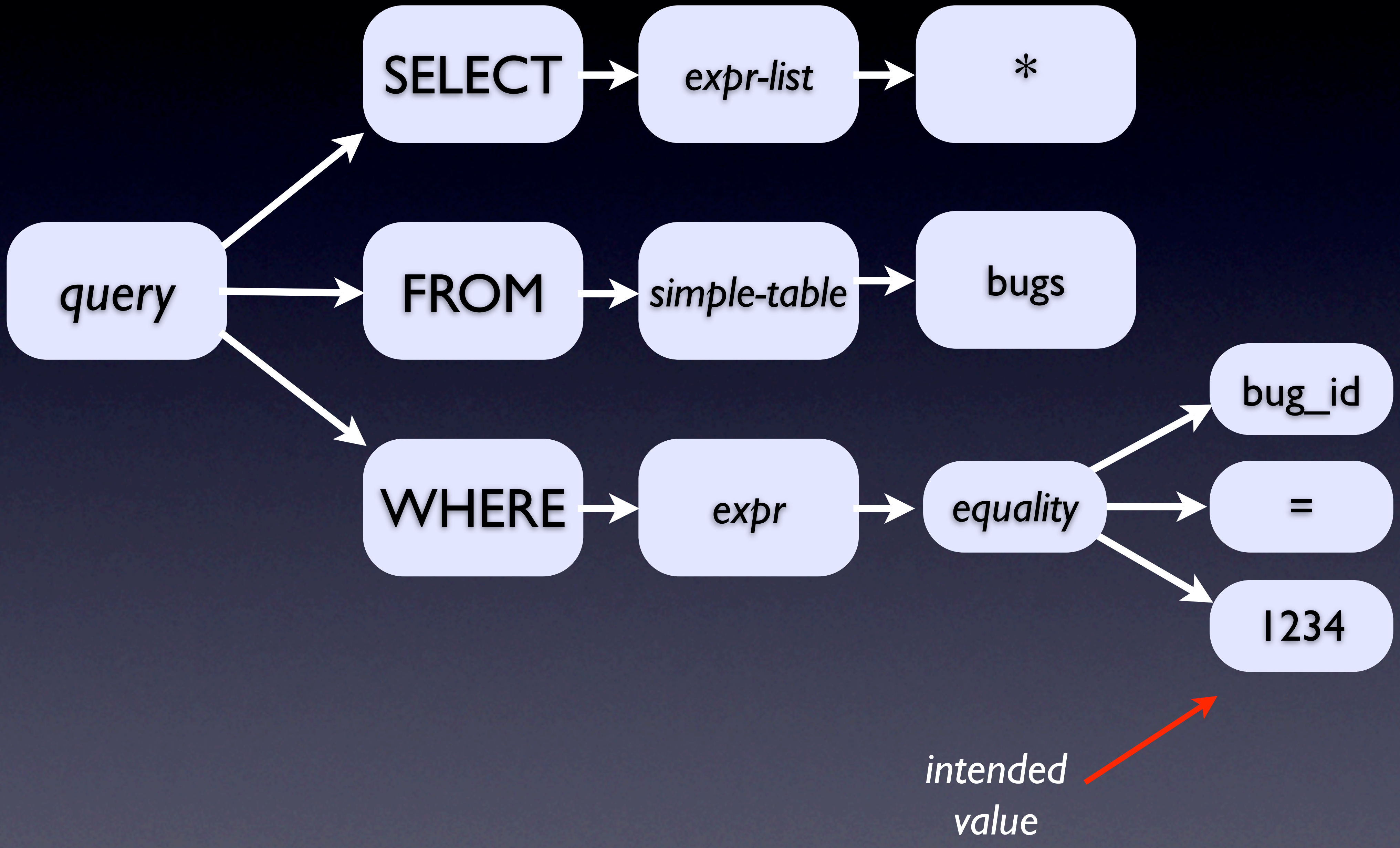
Parameter Facade



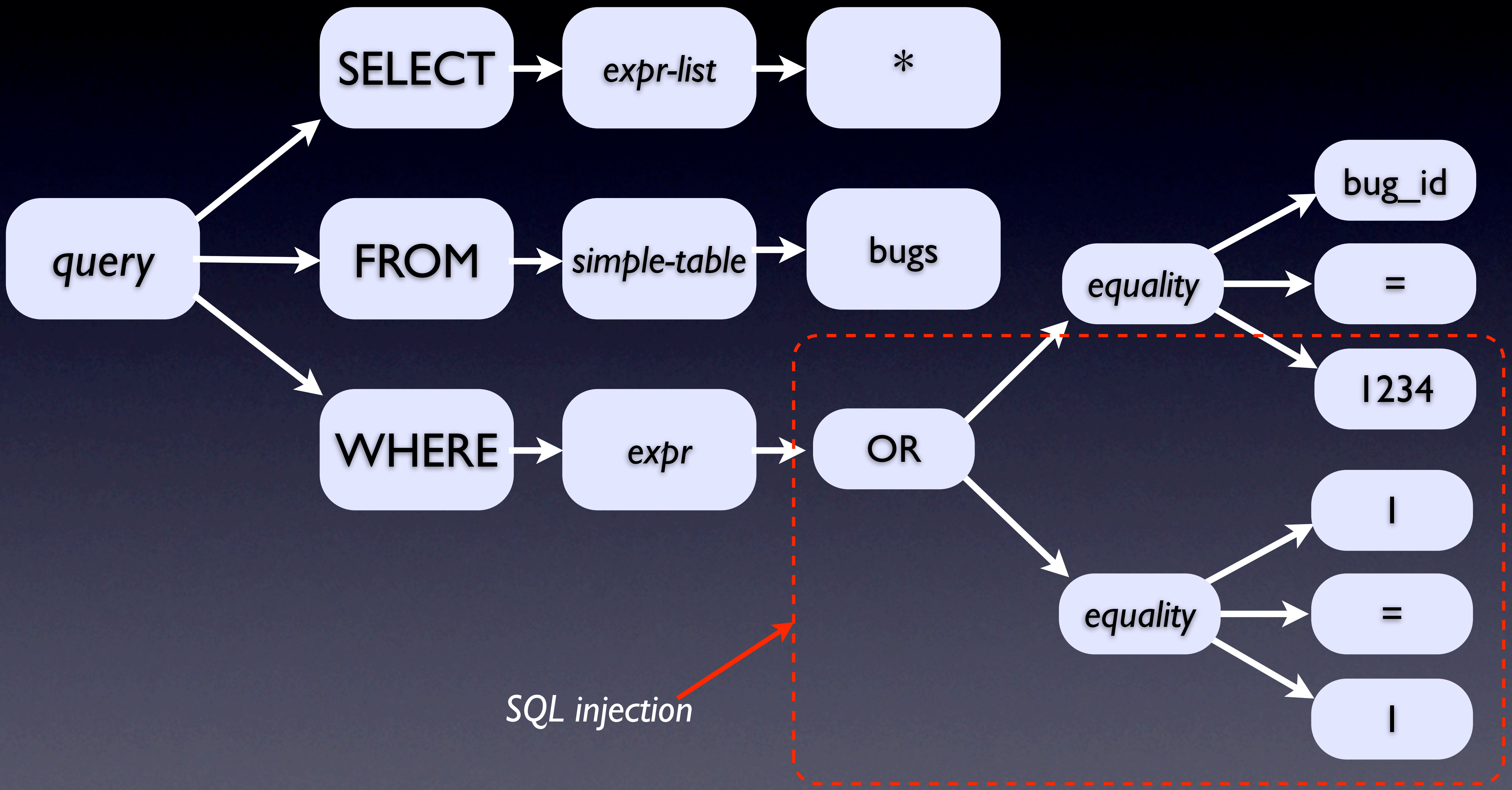
Parameter Facade

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

Parameter Facade



Parameter Facade



Parameter Facade

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

Parameter Facade

- 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 Facade

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 Facade

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

Phantom Side Effects

Every program attempts to expand until it can read mail.
— Jamie Zawinsky

Phantom Side Effects

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

  INSERT INTO Bugs . . .

  ...and send email to notify me

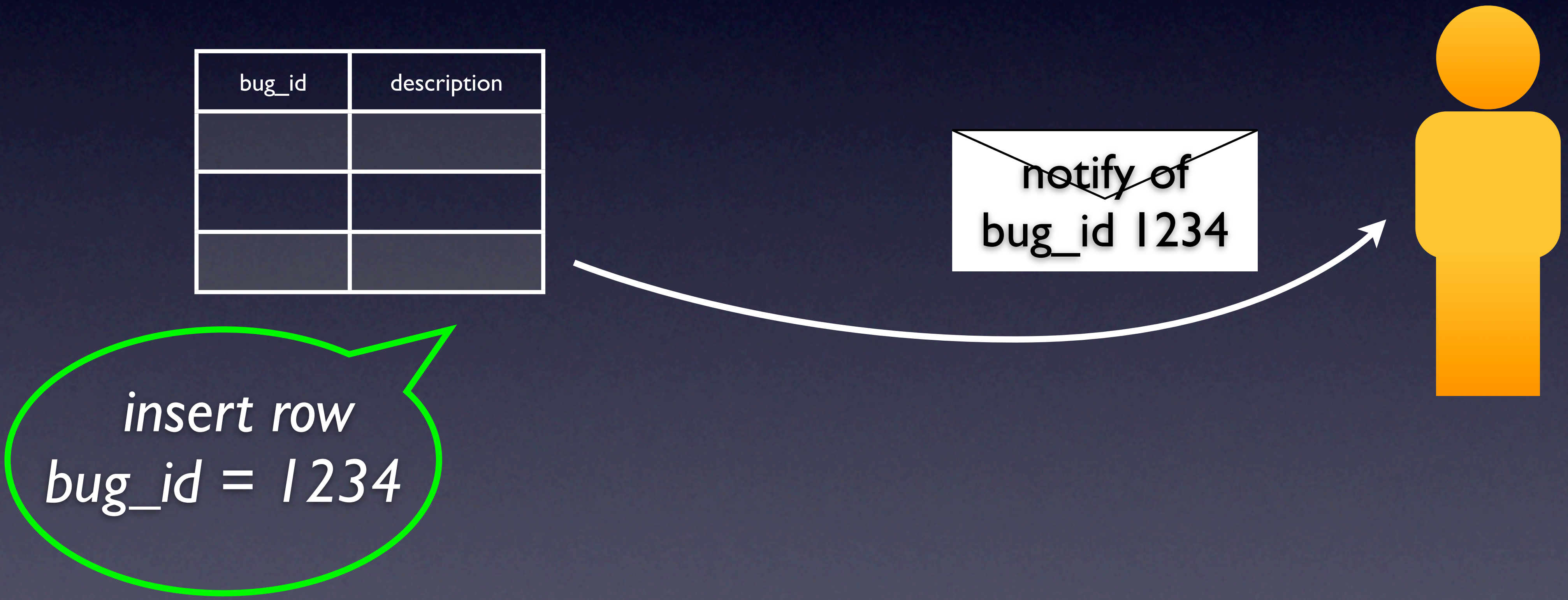
Phantom Side Effects

- **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



bug_id	description

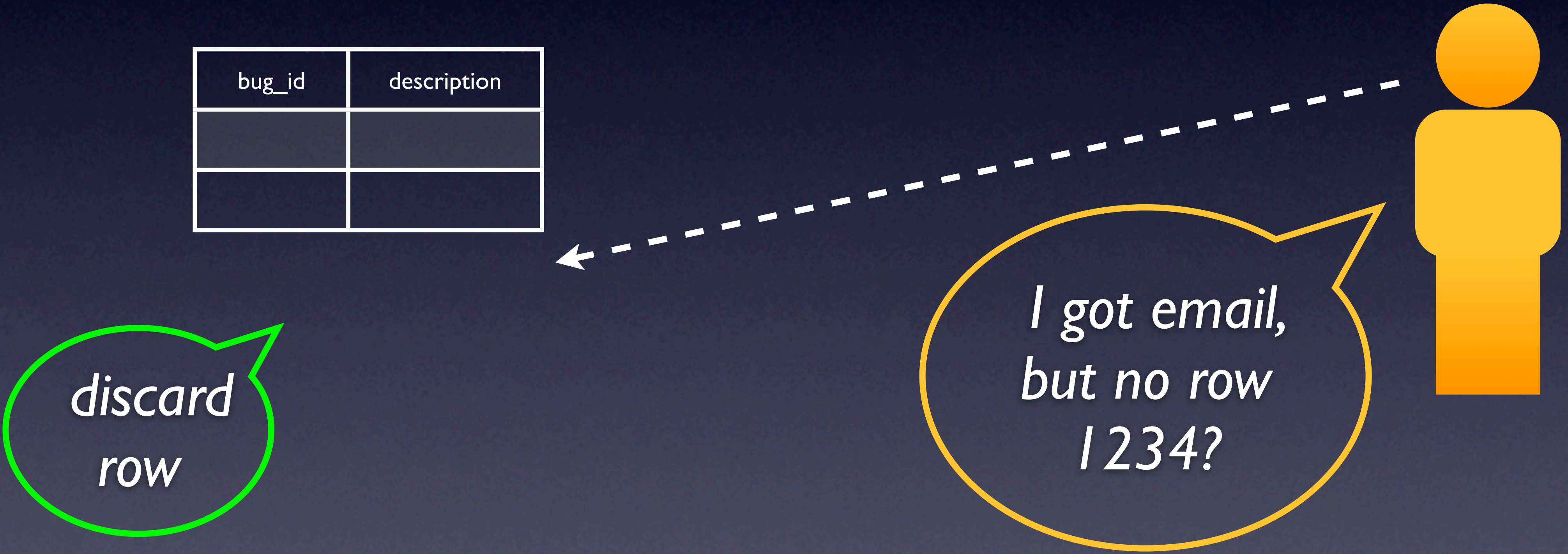
insert row bug_id = 1234

notify of bug_id 1234

Phantom Side Effects

- External effects don't obey ROLLBACK

2. ROLLBACK

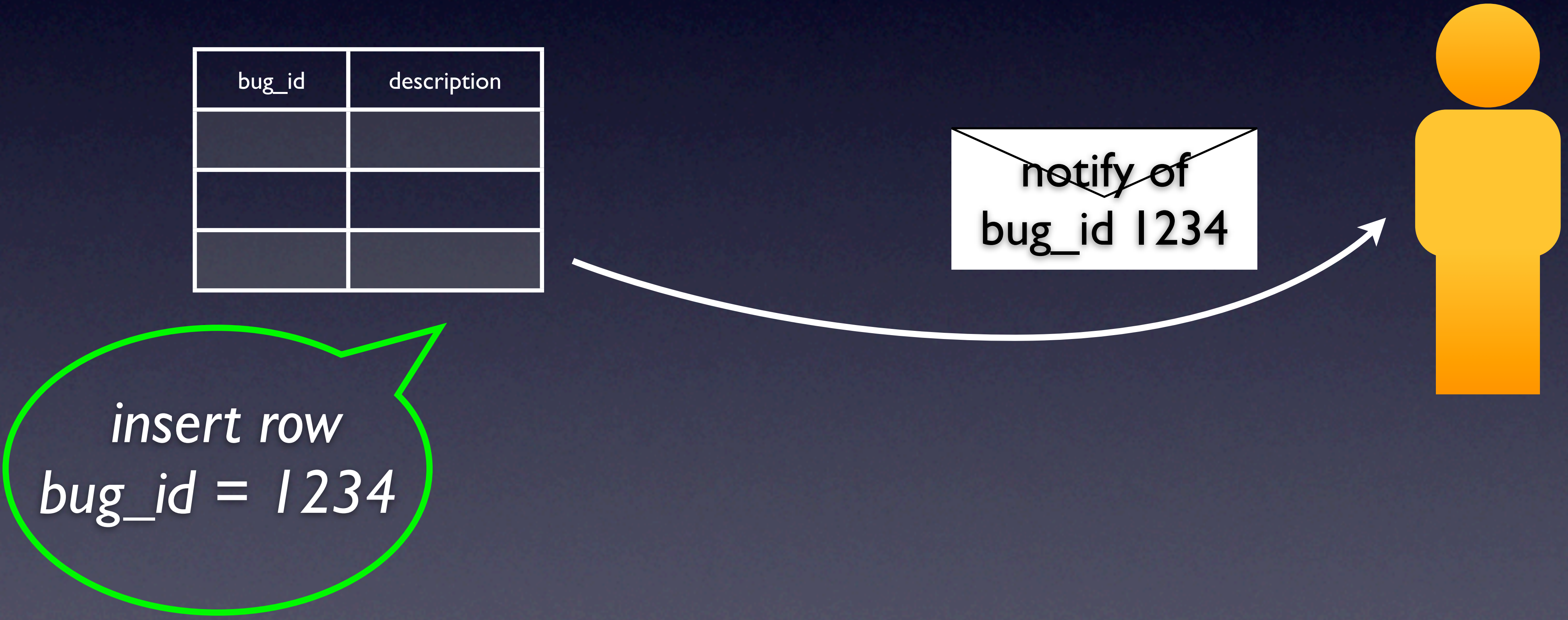


bug_id	description

Phantom Side Effects

- External effects don’t obey transaction isolation

1. Start transaction and INSERT

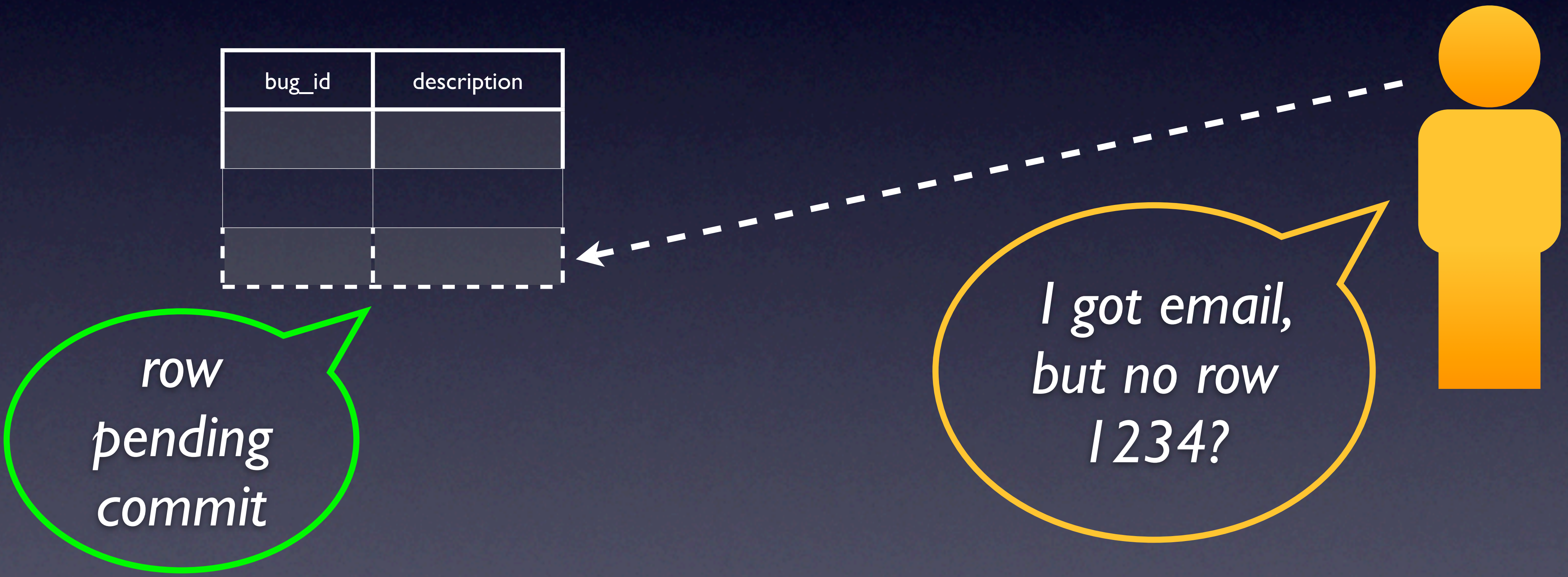


bug_id	description

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

```
SELECT * FROM bugs
WHERE bug_id = 1234
      OR send_email(‘Buy cheap Rolex watch!’);
```

SQL injection

  - Auditing/logging confusion

Phantom Side Effects

- Functions may crash

```
SELECT pk_encrypt(description,
    ‘/nonexistant/private.ppk’)
FROM Bugs
WHERE bug_id = 1234;
```

*missing file
causes fatal error*

Phantom Side Effects

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

```
SELECT libcurl_post(description,
     ‘http://myblog.org/ . . .’)
FROM Bugs
WHERE bug_id = 1234;
```

unresponsive website

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

See No Evil

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
>
> — Brian Kernighan

See No Evil

- **Objective:** Debug errors in queries.

See No Evil

- **Antipatterns:**
  - Ignore errors in return status or exceptions.
  - Troubleshoot code that builds queries.

See No Evil

- Ignoring errors in return status:

```
$sql = “SELECT * FROM Bugs”;

$result = $mysqli->query( $sql );

$rows = $result->fetch_all();
```

OK

See No Evil

- Ignoring errors in return status:

```
$sql = “SELECT * FROM Bugz”;

$result = $mysqli->query( $sql );

$rows = $result->fetch_all();
```

returns FALSE

FAIL

See No Evil

- Ignoring exceptions:

```
$sql = “SELECT * FROM Bugz”;
$stmt = $pdo->query( $sql );
$rows = $stmt->fetchAll();
```

throws PDOException

NOT REACHED

See No Evil

- **Solution:** check for error status.

```
$sql = “SELECT * FROM Bugz”;

$result = $mysqli->query( $sql );

if ($result === FALSE ) {
    log($mysqli->error());
    return FALSE;
}

$rows = $result->fetchAll();
```

don’t let it go this far!

See No Evil

- **Solution:** handle exceptions.

```
$sql = “SELECT * FROM Bugz”;

try {
    $stmt = $pdo->query( $sql );
} catch (PDOException $e) {
    log($stmt->errorInfo());
    return FALSE;
}

$rows = $stmt->fetchAll();
```

don’t let it go this far!

See No Evil

- Troubleshooting code:

```
$sql = ‘SELECT * FROM Bugs
    WHERE summary LIKE \’%’
    . $mysqli->quote( $feature )
    .‘ doesn\’t work 50\% of the time!%\”;

$result = $mysqli->query( $sql );

$rows = $result->fetchAll();
```

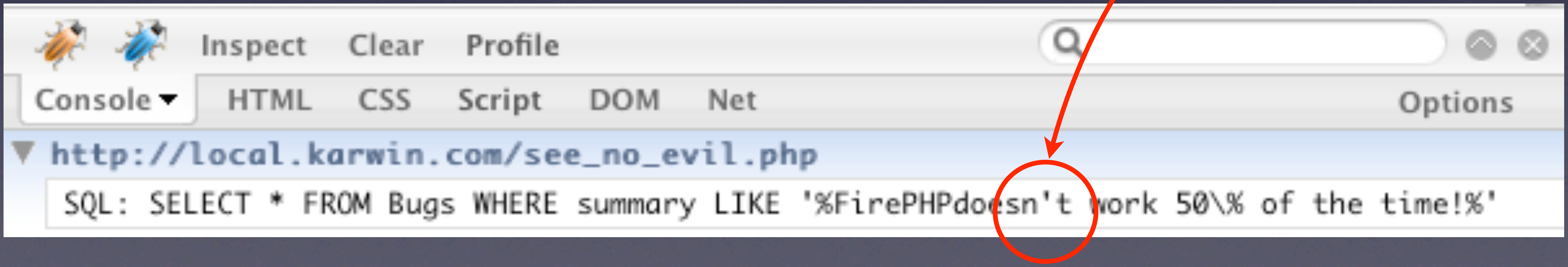
who wants to read this!?

See No Evil

- **Solution:** Look at the SQL, not the code!

```
$sql = ‘SELECT * FROM Bugs
    WHERE summary LIKE \’%’
    . $mysqli->quote( $feature )
    . ‘ doesn\’t work 50\% of the time!%\”;

$firephp = FirePHP::getInstance(true);
$firephp->log( $sql, ‘SQL’ );
```

the error is now clear!



```
http://local.karwin.com/see_no_evil.php
SQL: SELECT * FROM Bugs WHERE summary LIKE '%FirePHPdoesn't work 50\% of the time!%'
```

Diplomatic Immunity

Humans are allergic to change. They love to say, "We've always done it this way." I try to fight that.
—Adm. Grace Murray Hopper

Diplomatic Immunity

- **Objective:** Employ software development "best practices."

Diplomatic Immunity

- **Antipattern:** Belief that database development is “different” — software development best practices don’t apply.

Diplomatic Immunity

- **Solution:** Employ best practices, just like in conventional application coding.
  - Functional testing
  - Documentation
  - Source code control

Diplomatic Immunity

- Functional testing

  Tables, Views, Columns

  Constraints

  Triggers

  Stored Procedures

  Bootstrap Data

  Queries

  ORM Classes

- Verify presence of tables and views.
- Verify they contain columns you expect.
- Verify absence of tables, views, or columns that you dropped.

Diplomatic Immunity

- Functional testing

  Tables, Views, Columns

  Constraints

  Triggers

  Stored Procedures

  Bootstrap Data

  Queries

  ORM Classes

- Try to execute updates that ought to be denied by constraints.
- You can catch bugs earlier by identifying constraints that are failing.

Diplomatic Immunity

- Functional testing

  Tables, Views, Columns

  Constraints

  Triggers

  Stored Procedures

  Bootstrap Data

  Queries

  ORM Classes

- Triggers can enforce constraints too.
- Triggers can perform cascading effects, transform values, log changes, etc.
- You should test these scenarios.

Diplomatic Immunity

- Functional testing

  Tables, Views, Columns

  Constraints

  Triggers

  Stored Procedure

  Bootstrap Data

  Queries

  ORM Classes

- Code is more easily developed, debugged, and maintained in the application layer.
- Nevertheless, stored procedures are useful, e.g. solving tough bottlenecks.
- You should test stored procedure code.

Diplomatic Immunity

- Functional testing

  Tables, Views, Columns

  Constraints

  Triggers

  Stored Procedures

  Bootstrap Data

  Queries

  ORM Classes

- Lookup tables need to be filled, even in an “empty” database.
- Test that the required data are present.
- Other cases exist for initial data.

Diplomatic Immunity

- Functional testing

  Tables, Views, Columns

  Constraints

  Triggers

  Stored Procedures

  Bootstrap Data

  Queries

  ORM Classes

- Application code is laced with SQL queries.
- Test queries for result set metadata, e.g. columns, data types.
- Test performance; good queries can become bottlenecks, as data and indexes grow.

Diplomatic Immunity

- Functional testing

Tables, Views, Columns

Constraints

Triggers

Stored Procedures

Bootstrap Data

Queries

ORM Classes

- Like Triggers, ORM classes contain logic:
  - Validation.
  - Transformation.
  - Monitoring.
- You should test these classes as you would any other code.

Diplomatic Immunity

- Documentation

Entity Relationship Diagram

Tables, Columns

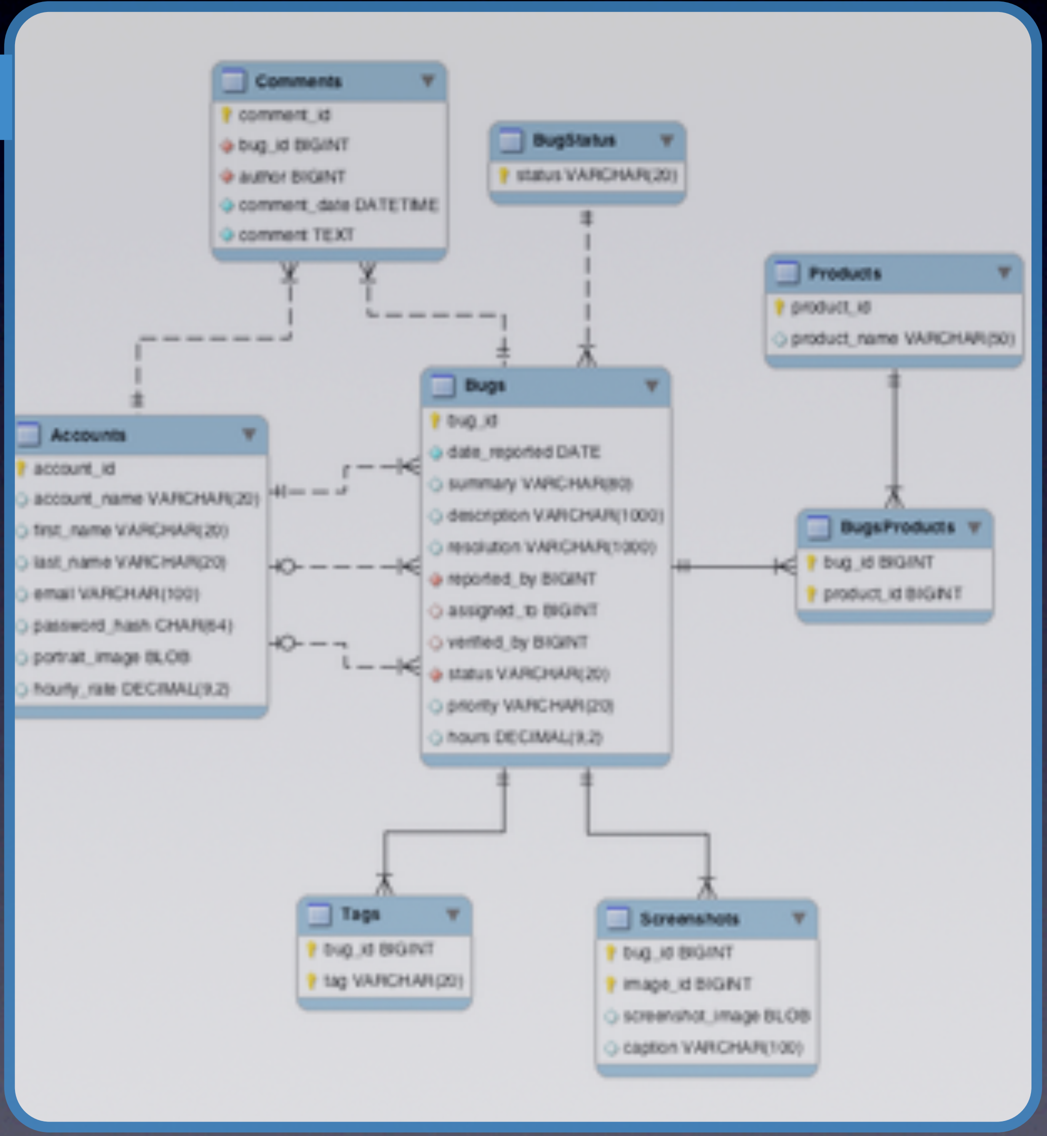
Relationships

Views, Triggers

Stored Procedures

SQL Privileges

Application Code



Diplomatic Immunity

- Documentation

  Entity-Relationship Diagram

  Tables, Columns

  Relationships

  Views, Triggers

  Stored Procedures

  SQL Privileges

  Application Code

- Purpose of each table, each column.
- Constraints, rules that apply to each.
- Sample data.
- List the Views, Triggers, Procs, Applications, and Users that use each.

Diplomatic Immunity

- Documentation

  Entity-Relationship Diagram

  Tables, Columns

  Relationships

  Views, Triggers

  Stored Procedures

  SQL Privileges

  Application Code

- Describe in text the dependencies between tables.
- Business rules aren't represented fully by declarative constraints.

Diplomatic Immunity

- Documentation

  Entity-Relationship Diagram

  Tables, Columns

  Relationships

  Views, Triggers

  Stored Procedures

  SQL Privileges

  Application Code

- Purpose of Views; who uses them.
- Usage of updatable Views.
- Business rules enforced by Triggers:
  - Validation
  - Transformation
  - Logging

Diplomatic Immunity

- Documentation

  Entity-Relationship Diagram

  Tables, Columns

  Relationships

  Views, Triggers

  Stored Procedure

  SQL Privileges

  Application Code

- Document the Stored Procedures as an API.
- Especially side-effects.
- What problem is the procedure solving?
  - Encapsulation
  - Performance
  - Privileged access

Diplomatic Immunity

- Documentation

  Entity-Relationship Diagram

  Tables, Columns

  Relationships

  Views, Triggers

  Stored Procedures

  SQL Privileges

  Application Code

- Logins with specific access purposes (e.g. backup, reports).
- Sets of privileges (roles) used for different scenarios.
- Security measures.

Diplomatic Immunity

- Documentation

  Entity-Relationship Diagram

  Tables, Columns

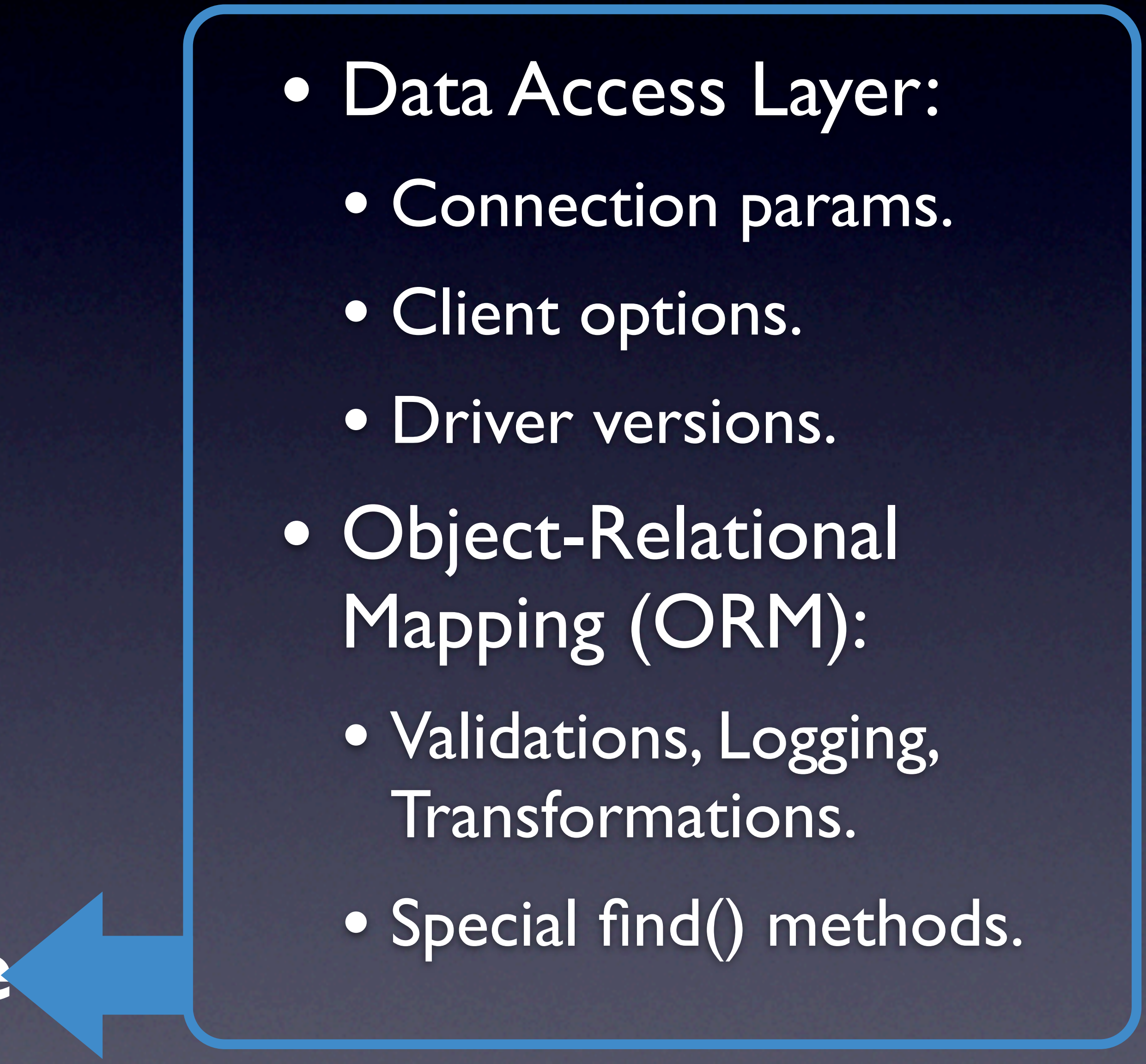
  Relationships

  Views, Triggers

  Stored Procedures

  SQL Privileges

  Application Code



- Data Access Layer:
  - Connection params.
  - Client options.
  - Driver versions.
- Object-Relational Mapping (ORM):
  - Validations, Logging, Transformations.
  - Special find() methods.

Diplomatic Immunity

- Source code control
  - Keep database in synch with application code.
  - Commit portable ".SQL" files, not binaries.
  - Create a separate database instance for each working set (each branch or revision you test).
  - Also commit bootstrap data and test data to source control.

Diplomatic Immunity

- Source code control: “Migrations.”
  - Migrations are like version-control for the database instance.
  - Incremental scripts for each milestone.
  - “Upgrade” script to apply new changes (e.g. CREATE new tables).
  - “Downgrade” script to revert changes (e.g. DROP new tables).
  - Database instance includes a “revision” table.

Magic Beans

Essentially, all models are wrong, but some are useful.
— George E. P. Box

Magic Beans

- **Objective:** simplify application development using Object-Relational Mapping (ORM) technology.

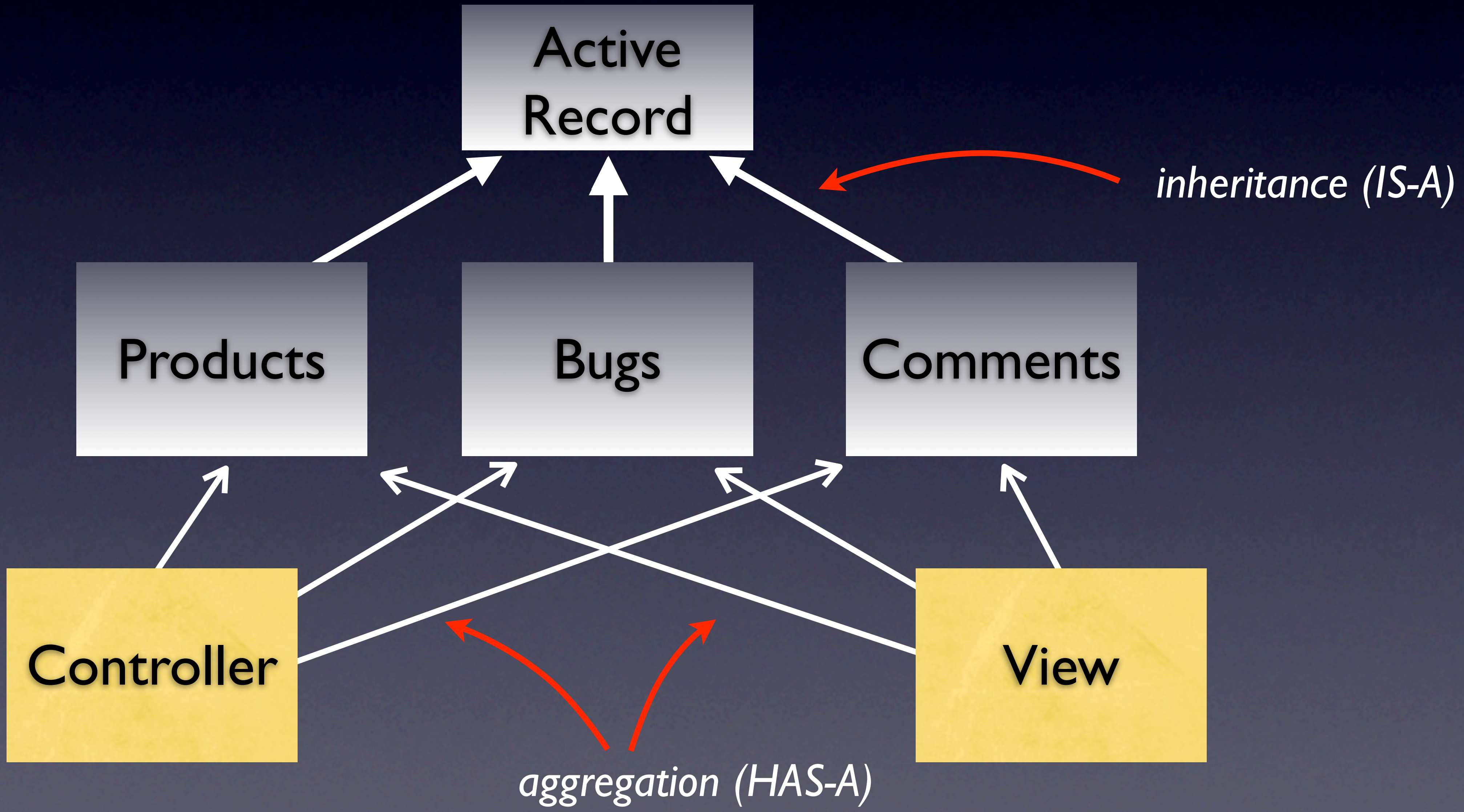
Magic Beans

- **Antipattern:** equating "Model" in MVC architecture with the Active Record pattern.
  - The *Golden Hammer* of data access.
  - "Model" used inaccurately in MVC frameworks:

RAILS®
symfony
CodeIgniter®
catalyst
Web Framework
SOLAR
Framework for PHP 5
CakePHP
Microsoft®
ASP.net
TURBOGEARS
django

Magic Beans

- **Antipattern:** Model *is-a* Active Record.



Magic Beans

- Bad object-oriented design:
  - “Model” → Active Record

    inheritance (IS-A)
  - Models tied to database structure.

    inappropriate coupling
  - Can a Product associate to a Bug, or does a Bug associate to a Product?

    unclear assignment of responsibilities
  - Models expose general-purpose Active Record interface, not model-specific interface.

    poor encapsulation

Magic Beans

- Bad Model-View-Controller design *“T.M.I.” !!*
    - Controller needs to know database structure.
    - Database changes cause code changes. ← *not “DRY”*
    - *“Anemic Domain Model,”* contrary to OO design.
      http://www.martinfowler.com/bliki/AnemicDomainModel.html
    - Pushing Domain-layer code into Application-layer, contrary to Domain-Driven Design.
      http://domaindrivendesign.org/

Magic Beans

- Bad testability design
  - Model coupled to Active Record; harder to test Model without database.
  - Database "fixtures" become necessary.
  - Business logic pushed to Controller; harder to test Controller code.

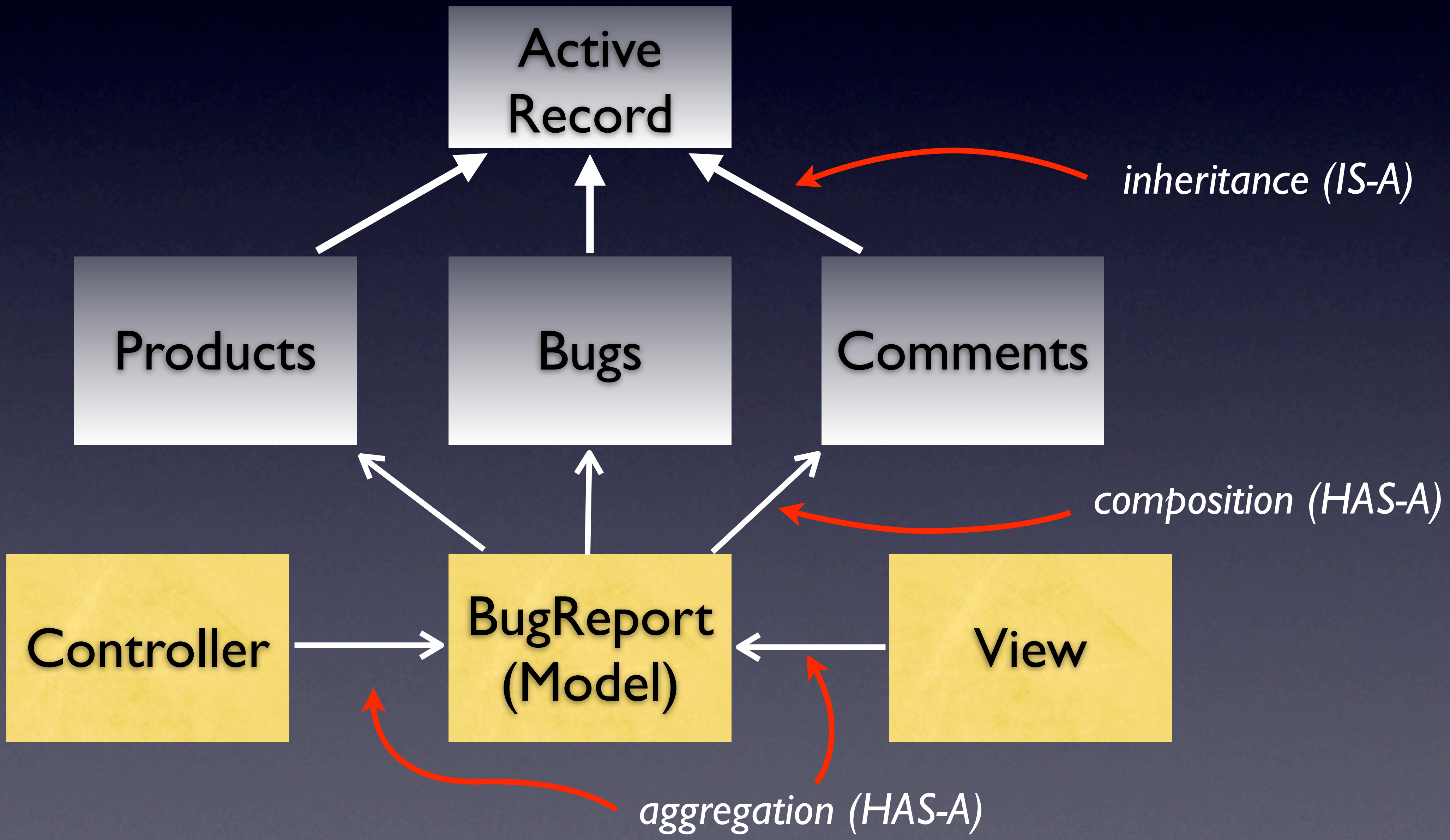
tests are slow

tests are even slower

mocking HTTP Request, scraping HTML output

Magic Beans

- **Solution:** Model *has-a* Active Record(s).



Magic Beans

- **Solution:** Model *has-a* Active Record(s).
  - Models expose only domain-specific interface.
  - Models encapsulate complex business logic.
  - Models abstract the persistence implementation.
  - Controllers and Views are unaware of database.

Magic Beans

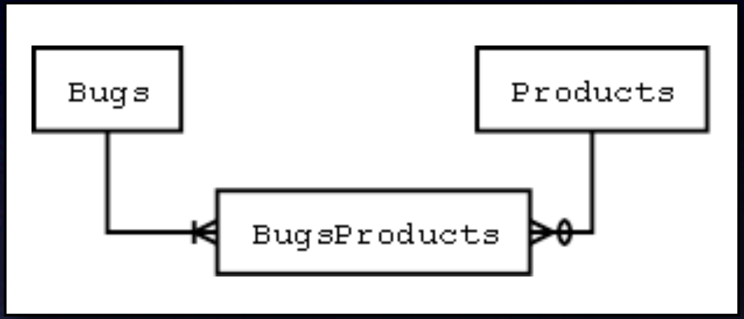
- **Solution:** Model *has-a* Active Record(s).
  - Models are decoupled from Active Record.
    - Supports mock objects.
    - Supports dependency injection.
  - Unit-testing Models in isolation is easier & faster.
  - Unit-testing thinner Controllers is easier.

Magic Beans

- **Solution:** Model *has-a* Active Record(s).
  - It's possible to follow this design, even in MVC frameworks that assume that Model *is-a* Active Record.

Antipattern Categories

Database Design Antipatterns



Database Creation 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

www.karwin.com