Life with Object-Relational Mappers

Or, how I learned to stop worrying and love the ORM.

Christophe Pettus
PostgreSQL Experts, Inc.
cpettus@pgexperts.com

Has this ever happened to you?

- "This query is running way too slowly. God, RDBMSes suck!"
- *"Well, you just need to change the WHERE clause…"*
- "I can't change the SQL. We're using an…"

ORM

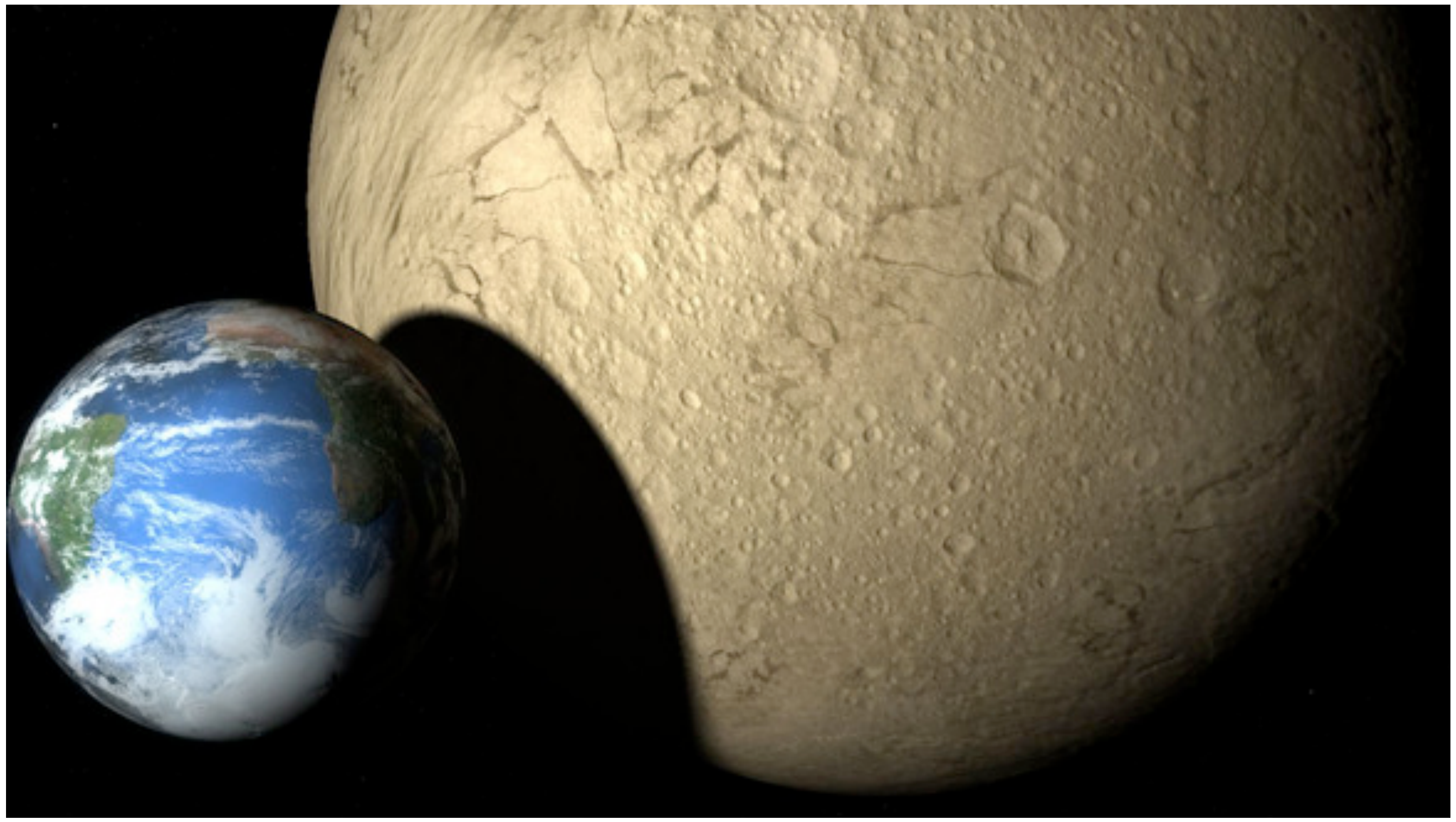
Let’s talk about ORMs.

- What is an ORM?
- Why do we have to put up with them?
- What are they good at?
- What are the problems?
- Can’t we just make them go away?
  - No. Sorry.
- How can we live with them?

Oh, right. Hi!

- Christophe Pettus
- Consultant with PostgreSQL Experts, Inc.
- PostgreSQL person since 1998.
- Application and systems architect.
- Designed a bunch of ORMs for various languages.

WHEN WORLDS COLLIDE.



The Two Worlds

- Object-Oriented Programming.
- Relational Database Management.

Object-Oriented Programming.

- Let’s ask Wikipedia!
    - “Object-oriented programming (OOP) is a programming paradigm using ‘objects’ …”
- Ask three programmers, get five answers.
- “… – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs.”

What is the critical OO feature?

- Data abstraction? Nope.
- Messaging? Nope.
- Modularity? Nope.
- Polymorphism? Nope, but getting warmer.
- Inheritance? Getting colder…
- Encapsulation.

Encapsulation

- Objects export behavior, not data.
- Many language expose the data — but that's a shortcut.
- Many objects don't have significant behavior — but that's a degenerate case.
- OO is all about wrapping up the behavior and the data into a single package, the object.

Object Relationships.

- Object models are collections of graphs.
- Pointers, references, swizzlers, strong refs, weak refs, lazy refs, blah blah woof woof.
- Ultimately, it is all derived from in-memory structures that point to each other using memory references.

The Reference Collection.

- Each object has its own list of references to other objects.
- The shape of the graph (as opposed to its contents) is generally an application architecture decision.
- Collections are not intrinsic to the objects, but are external structures they can be added to.

Object classes are static.

- Generally, object classes are static for the life of the application.
  - Dynamic languages blah blah woof woof.
- Adding new methods and members to an object is an application change.
- Run-time classes must be based on existing classes to allow existing code to make sure of them.

Objects are transient.

- Objects are first and foremost in-memory structures.
- Object persistence is a layer added on top of the object model.
  - No production OO language assumes persistence as the default condition.
- Even object databases required some kind of marking for object persistence.

The OO Paradigm.

- The objects export a set of behavior.
- The application supplies the data that is to exhibit that behavior.
- If you want different behavior, you need different objects.

The Relational Model, or Dr Codd Explains It All To Us.

- **The information rule:** This rule simply requires all information to be represented as data values in the rows and columns of tables. This is the basis of the relational model.
- **Physical data independence:** Application programs must remain unimpaired when any changes are made a storage representation or access methods.
- **Logical data independence:** Changes should not affect the user’s ability to work with the data.

Data is Primary.

- The RDBMS stores data, and makes it available to applications.
- It doesn’t know, or care, about the applications that access it.
  - Stored procedures, blah blah woof woof.
- What behavior it has is data-centric, not application-centric.

Relational Relationships.

- An RDMS has no pre-defined relationships.
- No, not foreign keys.
    - Foreign keys declare integrity constraints, and are only secondarily about “relationships” in a data sense.
- You can JOIN in any way you wish as long as you have compatible key types and can get at the data.

Relations are dynamic.

- CREATE TABLE (...);
- SELECT a, b, c FROM x JOIN y ...;
- These both create relations.
  - One has a longer lifetime, but there's otherwise nothing special about it (logically).
- An RDBMS can't work without throwing around anonymous relation types all the time.

Implicit Persistence.

- Databases don’t make much sense without persistence.
- The default operational model for RDMSes is to store data.
- Temporary and transient data is a special case.

The Two Worlds

- Mostly static typing system vs extremely dynamic typing system.
- Encapsulated data vs exposed data.
- Bound behaviors vs external behaviors.
- Explicit persistence vs assumed persistence.

When Worlds Collide.

- ORMs were designed to bridge these two worlds.
- With varying degrees of success.
- Different ORMs approach the problem differently.
  - RDBMS-up.
  - Application-down.

A VISIT TO PLANET ORM.

The Problem.

- Application programmer needs to get at data in relational database.
- Application programmer is handed an SQL manual.
- Application programmer starts writing code…

… that looks like this.

```
cursor* curs;
curs = db_connection->create_cursor();

customer_order *order = new(customer_order);

if (curs.execute(“SELECT * from customer_order WHERE order_id=123”)) {
    result_set* results;
    results = curs->fetch_results();

    customer_order->order_id = results->fetch_column(“order_id”);
    customer_order->customer_id = results->fetch_column(“customer_id”);
    customer_order->date_placed = results->fetch_column(“date_placed”);
    // ??? Need to finish.  First programmer quit to become
    // ??? a tour guide in Slovakia.
}
```

What we need is an interface layer.

- A tuple in a database is a collection of fields.
- An object has a collection of members.
- A table is a “type” that defines the fields in a tuple.
- A class is a type that defines the members in an object.
- This is all kind of starting to make sense!

I know! I know!

- We'll map a class to a table.
- Each of the columns of the table can be a member of the instances of that class.
- We can define create and save methods on a base class or something.
- We can keep some kind of flag as to whether or not the object maps to a row on disk yet.
- We'll figure the rest out later.

Who Wouldn’t Rather Write This?

```
customer_order* order = customer_order.retrieve(123);
order->cancel();
   // Didn’t want that loser’s business anyway.
order->save();
   // Off for a latte!
```

PROBLEM SOVLED!

- Just a few details. Really, just a few.
- How do we know how to map the tables to classes, columns to fields?
- Do we make the user specify whether to create a new row, or do we do it magically?
- How do we handle joins that are not persistent tables?
- We’ll figure that out later. How hard can it be?

And then came 1998.

- Java was the language of choice.
- RDBMSes were largely still in the hands of database architects and administrators.
- Battalions of application programmers starting writing DB-centric applications.
- And we needed a solution, fast.
- No one in their right mind was going to use J2EE. So, along came…

The RDBMS Up Approach.

- Pioneered by Hibernate.
- Design the database schema.
- Write mapping files that map columns into object fields.
- Use the ORM to convert results into collections of objects.
- Uses its own query language, HQL.

Problem solved, right?

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="eg">
        <class name="Cat"
            table="cats"
            discriminator-value="C">
                <id name="id">
                        <generator class="native"/>
                </id>
                <discriminator column="subclass"
                     type="character"/>
                <property name="weight"/>
                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>
                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>
                <property name="sex"
                    not-null="true"
                    update="false"/>
                <property name="litterId"
                    column="litterId"
                    update="false"/>
                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>
                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat"
                    discriminator-value="D">
                        <property name="name"
                            type="string"/>
                </subclass>
        </class>
        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>
</hibernate-mapping>
```

Yes and No.

- No tedious object copying.
  - Tedious XML files instead.
- Don’t have to learn SQL.
  - Do have to learn HQL — which is basically SQL.
- Can model joins in the XML file.
  - Have to create object classes for them.

Annotations!

```
@Entity
@Tuplizer(impl = DynamicEntityTuplizer.class)
public interface Cuisine {
    @Id
    @GeneratedValue
    public Long getId();
    public void setId(Long id);

    public String getName();
    public void setName(String name);

    @Tuplizer(impl = DynamicComponentTuplizer.class)
    public Country getCountry();
    public void setCountry(Country country);
}
```

Specify the mapping in the code!

- No nasty XML files to write.
- One less thing to get wrong.
- Uses introspection to calculate the schema.
- Of course, the schema has to match the object, or bad things happen.

Can’t we just create the class?

- Examine the schema, create the class from it.
- In the Java era, not easy.
- But then came the dynamic languages!
  - Python, Ruby.

The culture was shifting, too.

- More very small startups.
- Application programmers pressed into DBA roles.
- Even less time and interest in learning SQL.
- The database was increasingly viewed as an application object store rather than a shared data repository.

Active Record

- Got its name in Martin Fowler's 2003 book, *Patterns of Enterprise Application Architecture*.
- Exemplar: Active Record in Rails.
- Analyzes schema, produces classes.
- Clients of the class need to stay ahead of the interface.
- Requires a language that can extend classes on the fly.

Application-Down Approach.

- Exemplar: Django.
- The object model is defined in the application.
- The database is created by the application from the object model.
- Non-SQL-like query languages.

Look, Ma! No SQL!

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()
```

Problem Solved!

- The application writer does not need to learn SQL.
- Application programmers hate SQL.
- No, really. They hate hate `hate` SQL.
- They get a place to store their objects with minimum hassle.
- They get the demo up and running fast.

Even more good news!

- The application is “database independent.”
- You don’t have to hire any of those really expensive SQL people.
- And, hey, if we’re just stuffing objects into the database, why do we need SQL at all?
  - My dad used to listen to SQL on his 8-track in his Buick LeSabre.
- Let’s switch to MongoDB! It’s Web-Scale!

WHAT COULD POSSIBLY GO WRONG?

A real life case.

- Client complains DB is running too slow.
- Check batch process.
  - Does a BEGIN.
  - Does a SELECT.
  - Does an UPDATE.
  - Does a COMMIT.
- 1,235,000 times. Each night.

“I think we found your problem.”

```
for order in qs.all():
    order.days_open += 1
    order.save()
    transaction.commit()
```

Problem 1: Using the DB as Memory.

- Objects are an in-memory model.
- The database is generally not stored in memory.
  - By definition, a persistent store has to write to persistent storage.
- Just because it's easy, doesn't mean it's fast.

Transaction Mismanagement

```
address = Address(street_address="1112 E Broad St",
city="Westfield", state="NJ", zip="07090")

address.save()

order = Order(customer_name="Gomez Addams",
shipping_address=address)

order.save()
```

```
BEGIN;

INSERT INTO Address VALUES (...);

COMMIT;

BEGIN;

INSERT INTO Order VALUES (...);

COMMIT;
```

Problem 2: Weird Transaction Models

- ORMs generally have bizarre transaction models.
- “Each operation its own transaction” seems to be a typical default.
- Transaction management tools are often made to seem like a black art.

Index to Prohibited Features

- “Why don’t you create an index on these columns?”
- “Full-text search would be more appropriate here.”
- “PostgreSQL has a built-in POINT type.”
- “You need a trigger to enforce most multi-row constraints.”

Problem 3:
Limited Functionality

- Does not expose particular functionality.
- Especially if special syntax is required.
- Often claimed to be a feature.
  - “Database agnosticism.”
- Requires dropping to raw SQL.
  - Application programmers hate SQL.

Helping! I’m helping!

- Client was experiencing deadlocks.
- Deleting a record was deleting all dependent records across a foreign key.
- Normal right?
- Except that the relationship wasn’t marked ON CASCADE.

Problem 4: Excessive Help.

- Django (until the most recent version) did a manual ON DELETE CASCADE on foreign keys.
- And there was no way to turn it off.
- “Database agnosticism.”
- Why that particular feature? Who knows?

What you see is what you get, like it or not.

- Client complained a summarization operation was running too slow.
- Look at the database activity
  - SELECT about 125,000 records.

Sure Enough.

```
total = 0

for order in qs.all():

    total += order.amount
```

Problem 5: Bad Reporting Query Support

- Should do a SUM, right?
- Couldn’t return that from a query, because...
    - ... each row needs a primary key.
- Can drop down to raw SQL.
    - Application programmers hate... oh, you get the idea.

A Memory Disaster

- Client code queried for all records in a 12 million row table.
  - No problem! Django queries are lazy.
- Touched the first record.
  - BANG! Out of memory.
- Traced down through the code. What could be going on?

Problem 6: Naïve use of DB interface

- That ORM never uses named cursors.
- So, libpq happily sends over the entire result set when you ask for the first record.
- No clean way of getting around this…
  - … even though the language interface atop libpq fully supports named queries.
- If it's not this, it's something else.

“Don’t Do That, You’ll Kill Yourself!”

- Client was complaining about high log usage.
- Sure enough, >12GB/hour in logs being generated.
- Some individual queries were nearly 100,000 characters long.

```
list_of_values = [q.i for q in qs1.all()]

qs2 = Y.filter(z__in=list_o_values)

for y_value in qs2:

   ...
```

becomes

```
SELECT *

  FROM y

  WHERE z IN (insert a few thousand integers here);
```

Problem 7: Feature Mismatch

- Allows for creation of bad queries, easily.
- Without seeing underlying SQL, code looks very simple.
- A quick look at the log identifies the problem.
- But remember, this client was generating 12 gigabytes of log an hour…
  - And still didn't want to look at SQL.

Bad Idioms

- In Active Record (Rails), referred records in a foreign key relationship are updated before referring records.
- This behavior is difficult to override.
- Foreign key deadlocks, anyone?

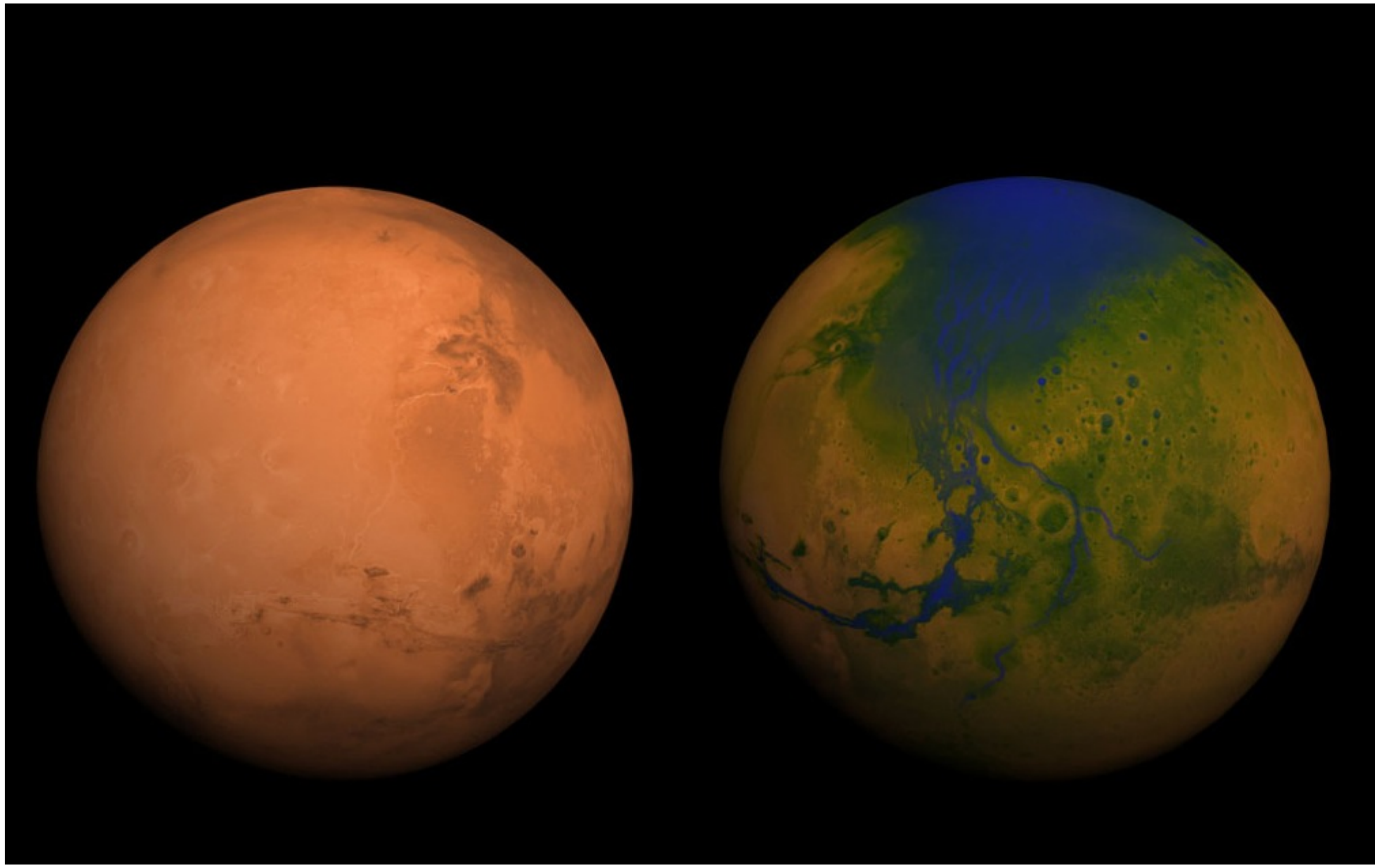
Problem 8: Unhelpful Standard Behavior

- The described pattern can cause deadlocks in PostgreSQL as of 9.x.
- Most application programmers think deadlocks are something that happens to someone else.
- Can be very difficult to track down.

Why should we care?

- These problems are blamed on the RDBMS, not the ORM.
- DB administrators and architects are routinely being called in to solve ORM-related problems.
- ORM-think is one of the primary driving forces behind the NoSQL movement.
    - If the only thing a DB is good for is an object store, why learn about an RDBMS?

REPAIRING THE DAMAGE



Fixing ORM Damage.

- Every ORM has its own idiosyncrasies.
- But the patterns of abuse are remarkably similar.
- Some changes will require substantial re-architecting.
- But some can be repaired quickly.

Pathological Iteration.

- Reading results in, processing them, writing them back out.
- Storing large result sets in application objects.
- SELECT / UPDATE loops.
- Replace with stored procedures or single UPDATE statements.

Transaction Maladies.

- Small transactions.
- Transactions left open between requests.
- Transactions that do not completely bracket atomic sequences.
- All modern ORMs have reasonable transaction primitives.
  - May require a bit of rearchitecture.

Query Train Wrecks.

- Queries with gigantic predicates.
- Bad, automatically-generated JOINs.
- Queries with very large SELECT lists.
- Replace with hand-crafted SQL or stored procedures, wrapped in an application API.

Join Landslides.

- JOINs done manually in the application.
- ORM syntax for joins tends to be horrible…
- … so application programmers don’t use it.
- Or, they are not thinking in SQL terms.

Cache Disasters.

- All ORMs cache.
- Almost no ORMs do intelligent cache invalidation.
- Do read-after-write if required (triggers, stored procedures, etc.).
- Replication lag?

Planner Phollies.

- Many ORMs love prepared statements.
- Java-based ORMs seem to particularly love them.
- PostgreSQL plans a prepared query once per session, and caches the plan...
- ... which is then often wrong for subsequent calls.
- DISCARD PLANS is your friend in these cases.

Index Incidents.

- Columns not indexed, because the ORM syntax for creating indexes is obscure.
- The wrong kind of indexes, because ORM can’t create multi-column or functional indexes.
- Too many indexes, because the application programmer just threw them on everything.

Debugging Tips

- ORM calls can be hard to correlate with database activity.
  - Be liberal with logging calls that indicate where in the application you are.
- Turn up PG logging. Use pgFouine.
- Remember that ORM operations are usually lazy, and rarely happen at the point of query.

IS THERE HOPE?

ORMs are not evil.

- They're invaluable for their core operation of object persistence.
  - We'd have to pry them out of their cold, dead hands anyway.
- Most of the problems come from the "hammer/nail" attitude.
- App programmers have been convinced that not learning SQL is a virtue.

Better ORMs?

- “Better ORMs” are not the answer.
- ORMs have been around since the early 1990s.
- If we could fix it that way, we would have by now.
- Virtually all production ORMs have ways of solving these problems.
- But we don’t take advantage of them.

Everything can be fixed.

- Technology Fixes.
- Educational Fixes.
- Management Fixes.

Better Application Architectures

- Don’t have the primary interface to the DB be the ORM.
  - The ORM is a relatively low-level component.
- Push the interface up a level to a more logical one.
  - Gives you a rug to sweep the SQL under.

Friendlier Database Design

- Turn the database into an application server.
  - Stored procedures.
  - Views.
- Use a familiar language for stored procedure implementation.
  - Wrap the nasty SQL up in a sugar coding of Python or Perl.
- Do background operations outside the ORM frameworks.

A Few Home Truths

- Web developers tend to be focused on the front-end OLTP.
  - Get them involved in data warehousing and analysis architecture.
- SQL experts are in-demand and well-compensated.
  - It’s a career development opportunity.

Teach the Controversy

- SQL is taught as a command language like bash.
- Teach the relational model instead.
- Programmers love efficiency.
    - Reduce the data, don’t ship it.
- Databases are a discipline, not a priesthood.

Education Fixes.

- Make developers use the production database system.
  - No SQLite on their laptops.
- tail -f the logs so they can see what is really happening.
  - Cheap profiling.
- Teach the relational model, not "SELECT gets the data."

Management Fixes

- No application is pure OLTP.
  - ORMs are not a data warehousing solution.
- An underused or poorly used RDBMS costs money.
  - Hardware, virtual server time…
- Remember those expensive SQL consultants?
  - Bring the skills in-house.

ORMs are great…

- … for the problem they were designed to solve.
- Creating objects out of database rows.
- The pathologies come from pushing them beyond their design center.
- So, don't do that.

LEARN TO LOVE THE ORM

ORMs are tools.

- Very useful in their proper place.
- Painful if you grab the wrong end.
- We need to confront ORMs as they are, not ORMs as we would like them to be (or not be).

Knowledge is Power.

- As DB experts, it’s our job to understand ORMs.
- Just like we need to understand SQL.
- The better we understand them, the more help we can provide to application programmers.
- And it’s one more valuable skill in <u>your</u> professional toolkit.

THANK YOU.