

# Make your database work for you

Beth Skwarecki  
beth@loxosceles.org

Pittsburgh Perl Workshop, 2006

(If you're new to databases, welcome. These are the things that I wish I had learned when I was new to databases.)

- my name is Beth
- I'm here to talk to you about databases
- I do bioinformatics, here's a little about my db:
  - 37+ million rows (I lost count)
  - 250+ tables, but most apps only deal with say 20 at a time
    - old crufty "legacy" tables in there too
  - writes to the db are batchy; the website does lots of reads, including searches, which should be fast.
  - we use postgres with transactions, constraints, etc
    - formerly we used MySQL 4, MyISAM, with none of those. (MySQL 5 has many of these features now)
    - when we switched to a db that had these features we saw huge improvements in performance and were able to ensure data integrity much better. (This can happen to you!)
- My examples will be in postgres
  - # If a feature I describe sounds useful but your DB of choice doesn't support it, please try not to fall asleep
  - # someday it will come in handy

## **Why hand off work to the database?**

**1) Performance**

**2) Data integrity**

(Sometimes both!)

- I'm here to tell you some things about databases that maybe you didn't know before
  - your DB can do things easily that are slow or hard in runtime code
  - you can make your database much FASTER
  - you can ensure much better DATA INTEGRITY
  - this will make you (and everyone you work with) happy.

## A few lines of SQL can save you a lot of work in Perl

we're going to keep our perl simple, AND our sql simple,  
and the database is going to do all our dirty work.

- What does SQL do for you?

- it's declarative.

You don't have to know how to sort your results,  
or how to determine the maximum of a list of  
numbers, or how to quickly find the right rows, or  
which pieces of your query are irrelevant - you just  
ask for it and the database gives it to you

- you're just *\*asking\** for your query to be optimized

The database engine will first think about how to  
execute your query (there's this thing called a  
query optimizer). Then it will eliminate any of the  
tables or information it doesn't need, and make  
short work of the rest.

Ergo,

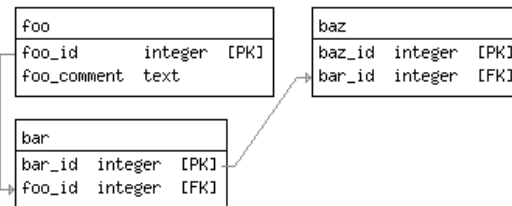
>>> HANDING WORK OFF TO THE DATABASE ALMOST

ALWAYS

RESULTS IN BETTER PERFORMANCE. <<<

## “Joining in software”

```
foreach my $foo $obj->get_foos() {
  foreach my $bar ($foo->get_bars()){
    foreach my $baz ($bar->get_baz()){
      if($baz->id() > 50){
        print $foo->id(),
              $bar->id(),
              $baz->id();
      }
    }
  }
}
```



- This I call "joining in software"

Your DB is actually really good at joining. Let it do its job!

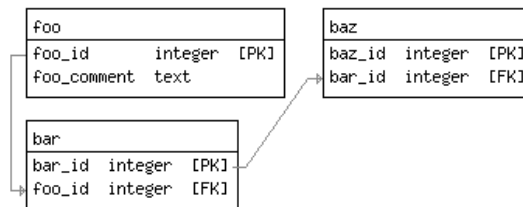
- Analogy: send husband to the store for milk. When he returns, send him to the store for eggs. When he returns, send him to the store for beer. You get the idea.

- Example: page at SGN that made two thousand database queries (or more) and two thousand objects. The page took a couple minutes to load when it wasn't cached. We rewrote it to make a small number of queries, fetching all 2000 rows at once, and the page started loading in a small number of seconds.

- Black-box objects hide database access, so you may not know whether there are easy optimizations to be done. (more later)

# Joining in SQL

```
SELECT
    foo.foo_id,
    bar.bar_id,
    baz.baz_id
FROM
    foo
    inner join bar using(foo_id)
    inner join baz using(bar_id)
WHERE
    baz.baz_id > 50;
```



- With this simple schema, it's easy to see that the first approach is kind of silly and the second approach is more efficient. In real life the objects will be more numerous and more complicated, and DB ACCESS MAY BE HIDDEN.

"Black box" objects can be dangerous to performance.

## Think of your DB as a collection of previously solved problems



Have you heard the one about the mathematician and the fire extinguisher?

So these scientists put a mathematician in a small office with a fire extinguisher on the desk. A fire starts in the trash can. So the mathematician picks up the fire extinguisher and puts out the fire.

In the next stage of the experiment, there is no fire extinguisher on the desk, so when the trash can catches on fire again, the mathematician runs out into the hall and finds a fire extinguisher there. He brings it back into the room and places it on the desk.

The scientists come in and ask him if he's going to put out the fire, now that he has the fire extinguisher. He says "nah. I reduced it to a previously solved problem."

## A simple search

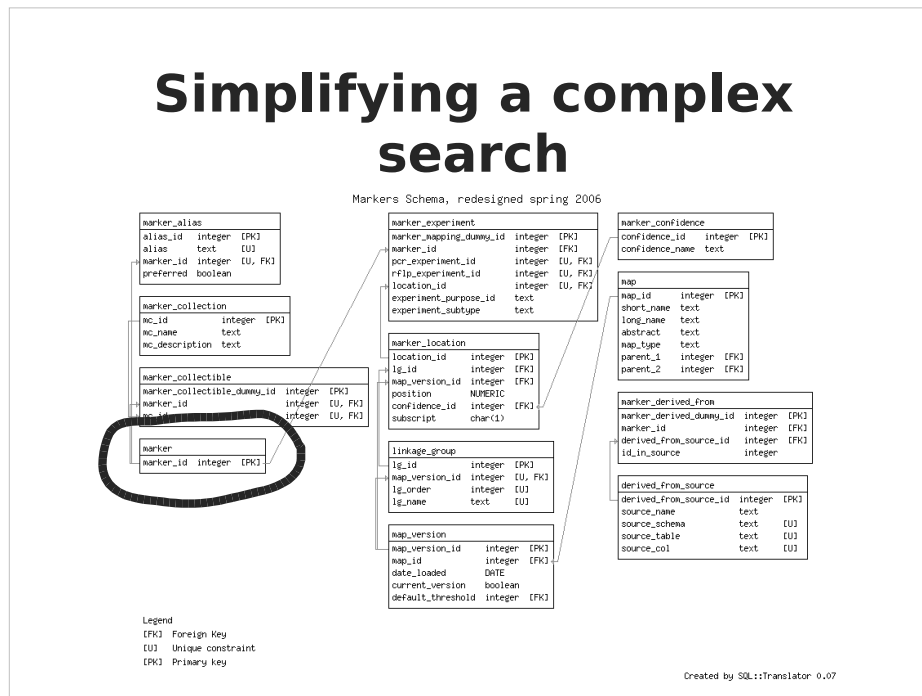
```
my $search = Marker::Search->new()

$search->name_exactly('TG123');
$search->in_species('Tomato',
                  'Eggplant');

$search->perform_search;
```

- This is how we'd like to do a search, from the programmer's point of view. Let's figure out a good way to implement it.
- A "marker" is a piece of biological data, the meaning of which isn't important here. It's a tiny snippet of DNA that typically has a known location within the DNA of some organism.
- The typical way to make a database search involves generating strings of SQL by gluing together bits of WHERE clauses, lists of tables, etc. I hereby present that this is a BAD IDEA. You can try to abstract away the complexity into big ugly objects, but that generally makes the problem worse. This makes debugging and testing very, very difficult; Adding a new feature to the search is a very tricky business.

# Simplifying a complex search

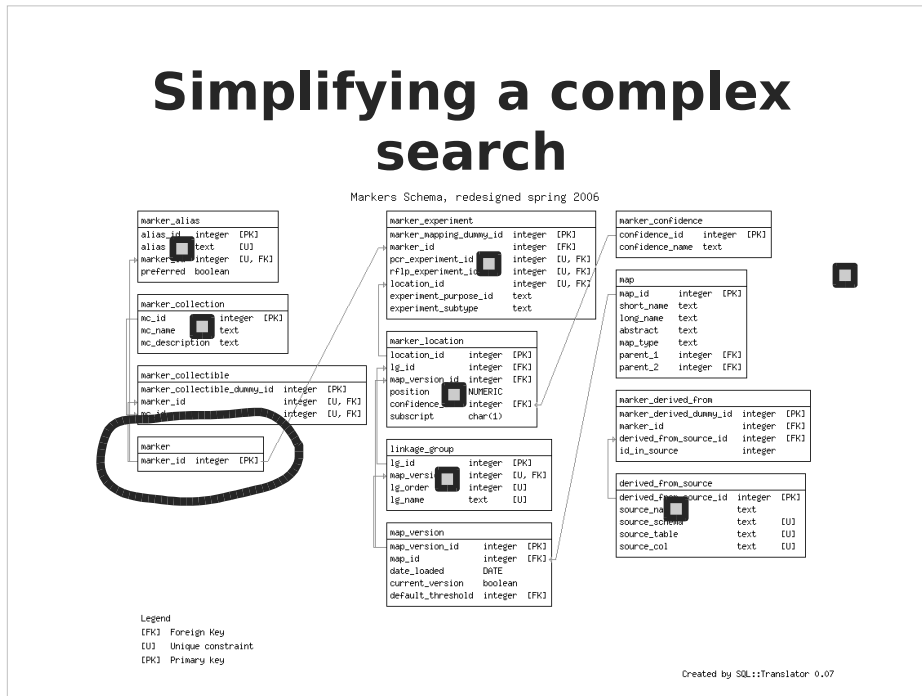


- This is where our simple search runs into the reality of a pretty complex database.
- This is a small piece of a biological database. The circled table is the "marker" table and contains a unique numeric ID for each marker.

The surrounding tables all represent data that has some relationship with the marker, such as

- its name
- its location
- the species it's studied in

# Simplifying a complex search



The squares are the pieces of information we might know, when we set out to search for a marker.

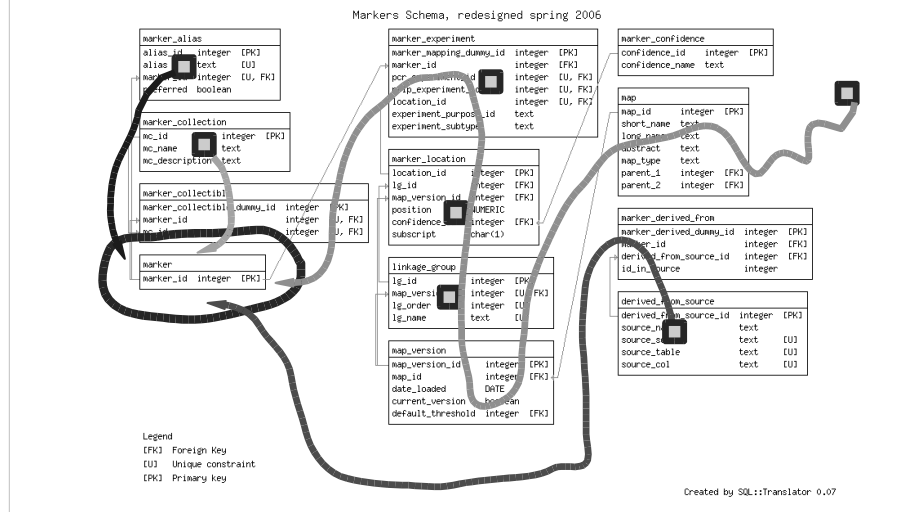
Analogy: book search.

circle is "books"

squares are author, publisher, year, subject, ...

The important thing is that we want to search by ANY OR ALL of the parameters. The more parameters, the more you narrow down your search.

# Yeah, that's pretty complex.



Getting from each search parameter to the markers table is different for the different cases.

Some have short "join paths", some long

## Seems like we really want the *intersection* of all those different things

Markers starting with 'T':

{1, 5, **72**, **388**, 509, **1124**, 1568, 2022, ... }

Markers studied in eggplant:

{2, 8, **72**, 120, 298, **388**, 785, **1124**, ...}

Result set:

{72, 388, 1124, ...}

Narrowing a search is like taking the intersection of many searches.

## That's a previously solved problem!



```
(SELECT marker_id FROM a,b,c WHERE ... )  
INTERSECT (SELECT marker_id FROM d...)
```

The database knows how to intersect. (It can also do unions and other fun things).

The database can intersect QUICKLY, too.

Code on the bottom of the slide shows the SQL for intersecting the two searches from the last slide.

## The DB knows how to do intersections

```
(SELECT marker_id FROM a,b,c ... )  
INTERSECT (SELECT marker_id FROM a,b ... )  
INTERSECT (SELECT marker_id FROM a ... )  
INTERSECT (SELECT marker_id FROM d...)  
INTERSECT (SELECT marker_id FROM e...)  
INTERSECT (SELECT marker_id FROM f...)
```

The last slide showed the intersection of two queries. This just shows the intersection of several.

Note that the blue joinpath appears once for every search parameter. Even though some of these queries use the same tables, it still makes sense to put them in separate queries

This means you can add a query without knowing what's in the other queries.

That's part of the secret to this method's success.

The code looks redundant but it is actually blazing fast.

If you don't believe me, TIAS!

This would still be my solution of choice even if it didn't have a speed advantage. Let's see why.

## The perl code

```
sub perform_search {  
    my ($self) = @_;  
  
    my $query = join ' INTERSECT ', map {"($_)" } @$self->{subqueries};  
  
    my $statement = $dbh->prepare($query);  
    $statement->execute(@{$self->{placeholder_values}});  
  
    # Now return the statement handle, or perhaps just the list of  
    # results.  
}
```

(SELECT marker\_id FROM a,b,c WHERE ... )  
INTERSECT (SELECT marker\_id FROM d...)

So how do we take advantage of this technique to make our Perl code shorter, easier to write, and more maintainable?

This is how we build the query. Assume we're writing an OO module.  
In the object we store a list of subqueries,  
and a list of associated placeholders.

Just join them together. What to do after that is up to us, let's say we execute the query here, maybe return the \$sth.

## The perl code

```
sub name_exactly {
    my ($self, $name) = @_;

    my $subquery = "SELECT marker_id FROM marker_alias WHERE alias = ?";

    push @{$self->{subqueries}}, $subquery;
    push @{$self->{placeholder_values}}, $name;
}

sub in_species {
    my ($self, @species) = @_;

    my $subquery = "SELECT marker_id FROM .... [blah blah].... WHERE "
        . join(' OR ', map {' common_name.common_name ILIKE ? ' } @species);

    push @{$self->{subqueries}}, $subquery;
    push @{$self->{placeholder_values}}, @species;
}
```

Now let's write those methods we saw in our simple interface.

We want to call methods that add parameters to the search.

Each method just adds a subquery to the list.

The big coding advantage is this: **YOU DON'T HAVE TO WRAP YOUR HEAD AROUND THE WHOLE SYSTEM JUST TO ADD A PARAMETER.**

Subquery pieces are logically separate from each other, so you can futz with them in isolation.

Isolating the pieces means you can debug them very simply:

## psql

It's easy to write and debug queries this way:

```
sandbox=> select organism_name from plants
where common_name = 'Petunia';
      organism_name
-----
Petunia axillaris parodii
Petunia hybrida
Petunia axillaris
Petunia integrifolia
(4 rows)

sandbox=> █
```

Type a query, see what happens.

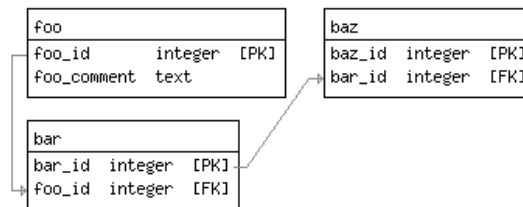
If there's no syntax errors and it returns the results you expect, most of your debugging work is behind you.

## How can we improve this schema definition?

```
create table foo ( foo_id integer,  
                  foo_comment text);
```

```
create table bar ( bar_id integer,  
                  foo_id integer );
```

```
create table baz ( baz_id integer,  
                  bar_id integer );
```



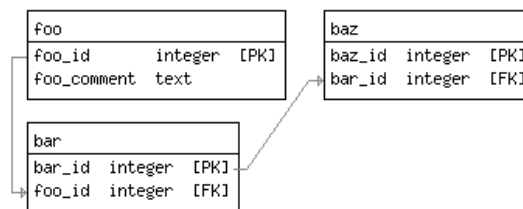
This is the schema from my example on joining.  
Each foo has many bars  
Each bar has many bazes

## Primary keys...

```
create table foo ( foo_id integer primary key,  
                  foo_comment text);
```

```
create table bar ( bar_id integer primary key,  
                  foo_id integer );
```

```
create table baz ( baz_id integer primary key,  
                  bar_id integer );
```



Unique identifier for each row.

A PK can be something naturally occurring,

- like an address if you're keeping track of houses,
- or a scientific name if you're keeping track of animals
- or a movie actor's name (the screen actors' guild requires members' stage names to be unique).

A PK must be unique, and not null.

(can't keep track of an actor with no name - how would you keep track of them?)

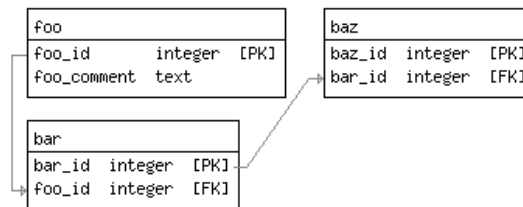
- In postgres, PK implies unique, not-null, and an index is made on the column.

Or it may work to assign numeric PK's as they're needed.

This is what the Social Security people do.  
And the people who assign ISBNs for books.

## ...and foreign keys

```
create table foo ( foo_id integer primary key,  
                  foo_comment text);  
  
create table bar ( bar_id integer primary key,  
                  foo_id integer references foo);  
  
create table baz ( baz_id integer primary key,  
                  bar_id integer references bar);
```



When we point to `foo_id`, that's not just an idea in some programmer's head.

We tell the DB that we mean for these tables to be linked.

The DB then makes sure that relationship stays intact.

## Foreign keys keep relationships intact

```
sandbox=> delete from foo where foo_id=1;
```

```
ERROR: update or delete on "foo" violates  
foreign key constraint "$1" on "bar"  
DETAIL: Key (foo_id)=(1) is still referenced  
from table "bar"
```

## Foreign keys keep relationships intact

```
sandbox=> delete from foo where foo_id=1;
```

```
ERROR: update or delete on "foo" violates  
foreign key constraint "$1" on "bar"  
DETAIL: Key (foo_id)=(1) is still referenced  
from table "bar"
```

...OR...

```
create table bar ( bar_id integer primary key,  
                  foo_id integer references  
                  foo on delete cascade);
```

What does this mean?

- you can't delete a foo without deleting its bars
- optionally, you can have bars and bazes be deleted when you delete their foo (ON DELETE CASCADE)

For cascade, the table that gets rows deleted must "volunteer" for it.

## Data integrity



"Those who preserve integrity remain unshaken by the storms of daily life."

### "referential integrity"

- most importantly, it means that six months from now you won't be scratching your head wondering why you have 600 bars without corresponding foos. Did you lose data by accident? Did your former coworker delete the data because it was incorrect? Which foos did those bars belong to anyway? Should I go and delete some bazes?
  - This can happen to you. (It happened to me last week.) # ests deleted but unigenes still there
  - Have you heard of "legacy code"? Sooner or later someone will inherit your "legacy database".

### "data integrity"

- In addition to relationships, you also want to keep your data intact.
  - Constraints to the rescue!

## Constraints

For example...

- UNIQUE
- NOT NULL
- CHECK status IN ('not submitted', 'in processing', 'done')
- CHECK rflp\_experiment\_id IS NOT NULL OR pcr\_experiment\_id IS NOT NULL

Some people like to check values in the perl code that loads data into the database. But what are the chances that that one program will always be the only thing ever used to insert data into the database? In my experience, data also comes from:

- scripts written by other people
- scripts written by you, after you forget about checking that field
- people making "quick fixes" at the psql prompt
- you don't have to remember, with each program you write, what you have to check for
- you can know that nobody else's programs or wacky psql antics are going to put in data that shouldn't be allowed
- you have an extra (though small) measure of security for INSERTS from the web user
- not scratching your head six months from now wondering how the wrong data got into that column.

You might think "but I'm a good programmer! I wouldn't put the wrong thing in a column!" ... but where does your data come from? For most of us, it comes from untrustworthy sources, such as ... HUMANS.

# Constraints

This can happen  
to you.

GAATCTCACA CCTGCTCCAC
GCGTTCTCGTTACTGGTGCT
GGAACACAACCAAGAAGTGGA
GCGGTTGATTCACATCGTAA
GGATTTCTCATGGAGAATCAGTC
CGTAAAGGGTTGTTCTTGTGC
TTCGGATAAAGCAATCCACC
TTCGGATAAAGCAATCCACC
ACAACAAAGGCAGCTGGTTC
ACTCACCATGGCTGCTTCTT
from Nancy
TGGCTGCCTTCTCTGTTT
CGGAGAGTGAAGATGCATTG
ACACCCGCTG TTTGTGACTT
TGCTTCTGAGGAGCCTGCAAAATAGACAAA
TGGCTCATCCTGAAGCTGATAGCGC
TGGATTTGATTAGCCGAAGG
TCTCAGTGGACTAAGGGGTCA

# Triggers

Even though triggers live in the database, we can write them in Perl!



- A trigger is a function that is called when you do something to a table (such as insert or delete). Use them to enforce things more complicated than simple foreign keys or constraints.

## A trigger in PL/Perl

```
CREATE FUNCTION check_if_tuesday()
  RETURNS trigger AS $$
  if((localtime)[6] != 2){
    die "You can only insert data on Tuesdays!";
  } else {
    return; # this means success (!)
  }
$$ LANGUAGE plperl;

CREATE TRIGGER only_insert_on_tuesdays BEFORE UPDATE
OR INSERT ON table_foo FOR EACH STATEMENT
EXECUTE PROCEDURE check_if_tuesday();
```

(PL = "procedural language", as opposed to declarative languages like ordinary SQL).

Create function. Make it a trigger by attaching it to a table.

Trigger notes.

- If you use '\$\$' for quoting, don't use double dollar signs in your perl code.
- a plain 'return' indicates success
- wrapped in a sub like mod\_perl does, so watch out for closure issues

## DB access in PL/Perl

### Old and new data (say, during an update)

```
$_TD->{old}{some_column_name}  
$_TD->{new}{some_column_name}
```

### Querying

```
my $result = spi_exec_query("SELECT...");  
my $rows_returned = $result->{processed};  
my $thing = $result->{rows}[0]{some_col};
```

- "old" and "new" data structures.
  - both for an update
  - old for deletes, new for inserts
- DB NULLs become Perl undefs
- spi\_exec\_query to get/retrieve data (no DBI, though there is a module that emulates DBI).
- constraints and triggers both slow things down
  - Does slowness really matter? Are we optimizing prematurely?
    - many users of databases have shitloads of data (If a script takes an hour to run, and your triggers make it take 2 or 3 hours, that might affect how much work you can do in a day.)
    - Many database apps are on the web  
The faster the better. You can't make the user wait more than a few seconds, or else they'll think your site is broken, click the submit button ten times, get cranky, etc.
  - row/statement triggers
  - deferred constraints
  - tradeoff: extra data integrity vs fast inserts: you decide

# Transactions

- a very fundamental feature of rdbms's. Many databases require that your statements be in transactions, but if you don't BEGIN a transaction, they will pretend you did. This is basically what AutoCommit does.

<http://www.postgresql.org/docs/8.1/interactive/tutorial-transactions.html>

On IRC a few days ago, somebody was asking if there was such a thing as a database test module that would keep track of the changes your tests made to the database (inserts, deletes, updates) and then have a "tear-down" function that would reverse all of those changes.

I don't know of a Perl module that does this, but no worry; your database has it built in!

Fun way to use transactions: Load a bunch of data. Then, run some tests on it to make sure it's all loaded correctly and you like the results. If it passes all the tests, commit.

Another use case: you're replacing a large amount of data (many tables, many rows) with updated versions. You wouldn't want users to see inconsistent data while this is happening.

Not just for inserts: you get a consistent view of the database the whole time. What if you're making queries and the database changes because of somebody else's insert?

## Transactions in SQL

```
BEGIN;  
INSERT INTO a_table VALUES ('garbage');  
DROP TABLE your_favorite_table;  
ROLLBACK; -- just kidding!
```

- If you forget to roll back, a rollback will occur when your code exits.

## Transactions in Perl

```
$dbh->{AutoCommit} = 0;

eval {
    # Your inserts and queries go here
};

if($@){
    $dbh->rollback;
} else {
    $dbh->commit;
}
```

(this is one way. DBI also has a "begin" method.)

- If an error occurs (such as a "die" from a trigger or a syntax error in SQL) the transaction is aborted and all further commands are ignored until you issue a rollback.
- Such an SQL error will set \$@ so you can treat it like any other Perl eval error
- Transactions can include savepoints, similar to savepoints in video games: you can rollback to a particular savepoint and continue from there as if nothing had happened.
- You can defer foreign key constraints until the end of a transaction, thus allowing circular dependencies (if for some reason those are a good idea for you).

SET CONSTRAINTS ALL DEFERRED

(or you can name constraints)

Opposite: SET CONSTRAINTS ALL IMMEDIATE

Currently this only applies to FK's and not unique or check constraints.

## Don't repeat yourself

```
$dbh->prepare("select asdf_info from foo inner  
join bar using(foo_id) inner join baz using  
(bar_id) inner join blorf using(baz_id) inner  
join asdf using(blorf_id) where foo.good=1  
and bar.current=1 and baz.censored=0 and  
foo_id=?");
```

All I have to say about this query is, it's pretty ugly.  
I don't want to read it.

I can see it has several tables

And some flag checking that we probably have to do  
almost every time we query these tables

Man, I don't want to have to type big ugly queries like  
that.

- A common complaint about coding SQL directly (rather than using object wrappers like CDBI) is that you'll have to repeat similar SQL throughout your program
- Views to the rescue!

## Use a view

```
psql=> CREATE VIEW foo_to_asdf AS
SELECT * FROM foo inner join bar using
(foo_id) inner join baz using(bar_id)
inner join blorf using(baz_id) inner
join asdf using(blorf_id) WHERE
foo.good=1 and bar.current=1 and
baz.censored=0;
```

It's still ugly, but that's the last time we have to type it!

- A view is a "fake table" in the database that contains the result of some query. It's generated automatically (by the rdbms) and is always up to date.
  - If you find yourself writing the same set of joins all the time, create a view

## Enjoy short, simple queries.

```
$dbh->prepare("SELECT asdf_info FROM  
foo_to_asdf WHERE foo_id=?");
```

```
$dbh->prepare("SELECT foo_info FROM  
foo_to_asdf WHERE asdf_id=?");
```

```
# etc.
```

- If you find yourself writing some really large and horrendous query more than once, make a view
- If you find yourself performing a simple but predictable sort of processing on the data that comes out of certain queries, write a view
  - Example: distance between cities.
- Fun facts:
  - \* views don't have to be read-only. You can define rules saying what should happen when you insert or update data in a view.
  - \* we just made views on tables, but you can make views on other views, if that helps you.
  - \* you can grant different permissions on a view than on the tables that make it up. (example: create view public\_foo as select from foo where censored=0;)
  - \* when you change your database's structure (it will happen someday), you won't have to update queries in programs that use views; instead you can just update the view definition. Similarly, you can create backward compatibility after a db redesign by creating a view that looks just like one of the old tables.

## Materialized views

- Since regular views are generated more-or-less dynamically, a big query that takes a long time to run will still be a PITA
  - although you do benefit from the query being prepared ahead of time
- A materialized view gets its results precomputed, so that querying it is like using a lookup table. This is like a cache of query results.
- Neither postgres nor mysql officially supports materialized views,  
but you can achieve the same effect by creating a table and then updating its contents either from a cron job or from an insert trigger

```
create table big_hairy_query_mview as ...;
```

## **Materialized views**

```
CREATE TABLE my_materialized_view AS  
SELECT...
```

For best results, update this from a trigger or a cron job.

**That's it! Thanks!**