Refactoring PostgreSQL databases

A note of warning

This uses advanced PostgreSQL features and is not portable to other databases.

Rules

- PostgreSQL feature allowing you to hack the query planner
- Rules are dangerous
- The canonical gun comparison:
  - C lets you shoot yourself in the foot
  - C++ lets you blow your whole foot away
  - Rules will take out the town
- Be very, very careful when writing rules.

Here's an example

```
CREATE RULE foo AS
        ON INSERT TO my_table
                DO ALSO
                INSERT INTO my_table_log NEW.*;
```

```
INSERT INTO my_table VALUES (random());
```

OOP

- OOP is a good idea.
  - As long as you apply it to code.
- Data is not object-oriented.
- Data is (or should be) described by the relational calculus.
- What happens when your database is designed by people who have drunk the OO koolaid?

In my case

- The better bits
  - 40 tables with the same structure.
  - One table per class.
- The worse bits.
- 160 tables with a very similar structure. And a table to dynamically generate SQL for these tables.
- Application development is stalled (including wanted features)!

```
CREATE TABLE domcno_cns (
	orderid		integer,
	ns		varchar(255),
	ip		varchar(50),
	creationdt	integer,
	timestamp	timestamp
);
```

```
CREATE TABLE	dombiz_cns (
	orderid		integer,
	ns		varchar(255),
	ip		varchar(50),
	creationdt	integer,
	timestamp	timestamp
);
```

Interesting data in the table:

- IP addresses containing ',' instead of '.'
- IP addresses containing 555 or 999
- Multiple IP addresses in the same line
- IPv6 addresses. The funny thing? We didn't even officially support v6 till this year

Fixing that one

- There's a rather major constraint
  - Uptime
- Actual change needed code fixing (including validation code).
- Writing the table design was easy.
  - Implementing it was more difficult

Actually implementing the change

```
-- Create the table
CREATE TABLE domain_cns (
	orderid				integer,
	nameserver			text,
	ip				inet,
	creation_time			timestamptz	default now(),
	last_modified_on		timestamptz	default now(),
	CONSTRAINT		domain_cns_pk
		PRIMARY KEY		(orderid, nameserver, ip),
	CONSTRAINT		domain_cns
		FOREIGN KEY		(orderid)
			REFERENCES		domorder(orderid)
);

-- Create a function which throws a general exception and a custom message
-- This will trickle up to the end user
CREATE FUNCTION fail(text) RETURNS text AS $$
BEGIN
		RAISE EXCEPTION '%', $1;
END;
$$ LANGUAGE 'plpgsql';
```

```
-- Define a trigger to ensure that last_modified_time is set since
-- this table may be modified by hand
CREATE OR REPLACE FUNCTION set_last_modification_time() RETURNS
        trigger AS $$
BEGIN
                NEW.last_modified_time = now();
                RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

-- And apply the trigger
CREATE TRIGGER fix_modification_time
         BEFORE UPDATE ON domain_cns
                  FOR EACH ROW
                           EXECUTE PROCEDURE set_last_modification_time();
```

```
-- We need to ensure that data in this table stays unchanged while we merge the tables
-- But since this table is read fairly often, an exclusive lock will block the application
-- leading to timeouts
-- We use rules to throw errors on any operations which will modify data
-- This isn't quite an exclusive lock since DDL would go right through, but for our purposes,
-- it's good enough.

CREATE RULE dombiz_cns_insert AS
	ON INSERT TO dombiz_cns
		DO INSTEAD SELECT fail('Database maintainance in progress. Try again later');

CREATE RULE dombiz_cns_update AS
	ON UPDATE TO dombiz_cns
		DO INSTEAD SELECT fail('Database maintainance in progress. Try again later');

CREATE RULE dombiz_cns_delete AS
	ON DELETE TO dombiz_cns
		DO INSTEAD SELECT fail('Database maintainance in progress. Try again later');
```

```
BEGIN;

-- Copy the data into the actual table
SELECT INTO domain_cns (orderid, nameserver, ip, creation_time, last_modified_on)
	SELECT orderid, ns, ip::inet, to_timestamp(creationdt), “timestamp”
		FROM dombiz_cns;

-- Now get rid of the old table and replace it with an updatable view
DROP TABLE dombiz_cns;

-- In theory this should be a join and have a where clause to limit the search
-- results. However, for my use case, adding more data doesn't matter
-- since the select query will always specify the orderid.

CREATE VIEW dombiz_cns AS
	SELECT orderid, nameserver ASV ns, ip, creation_date, last_modified_time
		FROM domain_cns;

-- These rules only work when the action is performed on a single row.
CREATE RULE dombiz_cns_insert AS
	ON INSERT TO dombiz_cns DOINSTEAD
		INSERT INTO domain_cns (orderid, nameserver, ip)
			VALUES (NEW.orderid, NEW.ns, NEW.ip);
```

```
CREATE RULE dombiz_cns_update AS
    ON UPDATE TO dombiz_cns DO INSTEAD
        UPDATE domain_cns SET
                orderid       = NEW.orderid,
                nameserver    = NEW.nameserver,
                ip            = NEW.ip::inet
        WHERE   orderid       = OLD.orderid
        AND     nameserver    = OLD.nameserver
        AND     ip            = OLD.ip;

CREATE RULE dombiz_cns_delete AS
    ON DELETE TO dombiz_cns DO INSTEAD
        DELETE FROM domain_cns
        WHERE   orderid       = OLD.orderid
        AND     nameserver    = OLD.ns
        AND     ip            = OLD.ip;

COMMIT;
```

Partitioning

Partitioning is good

- When your tables are log tables
  - Logs need archival
  - Partitioning saves the trouble of data deletion
    - Just drop the partition
- When you have a very large number of rows
  - Faster queries

Partitioning a logging table

- Create a new table with the appropriate structure and permissions
- Create an inheritance hierarchy under this table
- Apply a trigger to the parent table which inserts into the appropriate child table and then returns NULL so that no row is actually inserted into the parent
- Apply a rule to the old table which rewrites insert statements so that insertion happens on the new table

How to ...

- Move the old data over into the new table.
- In a transaction, drop the existing table and create a view with a rule redirecting inserts
- Write a small script which creates a new inheritance table every partitioning period and updates the trigger procedure to send data to the new table(s), and then put it into a cron job.
- Archive old tables every so often.