

Database refactoring

Nathalie Tang
Pascal Van Cauwenberghe

Mayima



Outline

- Team and project setup
- Application – DB interface
- Database design
- Column refactoring
- Table refactoring
- Data refactoring
- Refactoring in production

Team and project setup (1)

- My private database server
- Or `CREATE SCHEMA TANGNAT_BPI ...`
- Different environments
 - DEV, TEST, ACPT, PROD
- Scripts in SCM
 - DB version table
 - `upgrade_v1.sql`, `upgrade_v2.sql`
 - Check version in script

Team and project setup (2)

- Automatic build + test
 - Create DB, `upgrade_vX.sql`, unit test
 - Create / upgrade database
- Be friendly to the DBA
 - If they are not in your team...
 - ...make them feel part of the team

Application-DB Interface The Goal

- No duplication
- Unit testing
 - Each test is a transaction
 - Set up data, perform test, Rollback
- 2 techniques: Implicit or Explicit interface

Explicit DB Interface

- Metadata: Schema + Column objects
- Generate SQL statements
 - `Builder(Schema,Row) => SQL`
 - Once and only once
- - This code has to be written and kept in sync
 - This code can be generated

Implicit DB interface

- Metadata: retrieve from DB
- Generate SQL statements
 - Builder(Schema,Row) => SQL
 - But: you have to write more column names
 - Esp. in WHERE clauses
- (almost) Nothing to write, always in sync
 - If not... error at runtime (better: in tests!)

Database design

- Database first, Application later
 - Entities, Relations not Objects, Pointers, NULLS
- Encapsulation with
 - Explicit names (not system generated)
 - Constraints
 - Computed columns
 - Views
 - Stored procedures
 - Trigger
 - if you really, really need to, for a short while

Column refactoring (1)

- Add column
 - Simple, if you give it a default value
 - ALTER TABLE people ADD COLUMN fax VARCHAR(20) NOT NULL DEFAULT " ;

Column refactoring (2)

- Alter column
 - Compatible changes
 - Incompatible change
 - Read-only: computed column for old column
 - R/W: add new column,
 - keep consistent (code + trigger),
 - delete old column later

Column refactoring (3)

- Delete column
 - When nobody is looking
 - Check in for next version
- Split/Merge column
 - Read-only: computed column for old column
 - R/W: add new column,
 - keep consistent (code + trigger),
 - delete old column later

Table refactoring (1)

- Add table

```
CREATE TABLE people
(
  id serial NOT NULL,
  firstname varchar(64) NOT NULL DEFAULT "",
  lastname varchar(128) NOT NULL DEFAULT "",
  email varchar(128) NOT NULL DEFAULT "",
  CONSTRAINT "PK_people" PRIMARY KEY (id),
  UNIQUE(id)
);
```

Table refactoring (2)

- Rename table
 - Update dependencies (auto/manual)
 - CREATE SYNONYM old FOR new (oracle)
 - CREATE VIEW old as select * from new (ms)
 - so you don't have to update code (yet)
 - Remove old in next release

Table refactoring (3)

- Delete table
 - When nobody is looking
 - Check in for next version

Table refactoring (4)

- Split table
 - Split table A in X and Y
 - Create view A: X join Y
 - But: not all views are updatable
 - Use rules (PGSQL) or triggers (Oracle, MSSQL)

Data refactoring

- Acceptance test
 - Select results that don't match new criteria
 - Write test first
 - Call before and after migration
- Rollbacks for tryouts on production copy
- Create new data if old data is still used
 - New data = f(old data)
 - UPDATE users SET data = new data
 - WHERE data = old data ;
 - DELETE old data ; IF NOT USED !

Refactoring in production (1)

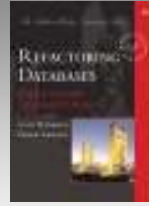
- Repeatable: everything in scripts
 - Rehearse, rehearse, rehearse!
- Upwards compatible changes
 - Keep the application running
- Rolling upgrades

Refactoring in production (2)

- Multiple applications
 - Multiple concurrent versions
 - Keep track of who's using what
 - And when it's no longer used
- Release often! Small releases!
 - In any case, small scripts

WHY ?

- Design the database story per story
- Small releases
- Max 5 minutes application down
- Users happy



Questions?
Remarks ?

<http://www.agiledata.org>