Using PostgreSQL to develop biological and academic databases

Peter St. Onge
Department of Economics, University of Toronto
Alexandr Ignachenko
Toronto General Hospital
Paul Osman
eval.ca

http://pete.seul.org/talks/

Overview

- Rationale
- Research ... The Final fronter! (Maybe)
- Functions
- Views
- Stored Procedures
- Rules and Triggers
- Materialized Views
- SQL Data Prep Scripts
- Future work
- Summary

Goal

- Place more data-handling logic inside database
- Using strengths of modern RDBMS to protect data: AAA / CIA / ACID

Authentication, Authorization, Accountability

Confidentiality, Integrity, Availability

Atomicity, Consistency, Isolation, Durability

Rationale

- Many research apps are LAMP-based
- Simple, functional, effective
- Web apps are relatively simple to write
- But database is often only a simplistic persistance mechanism
- SQL is pretty standard
- So why is transition to LAPP stack increasingly common in research?

Why Bother?

Research needs for data handling are as varied as the individual projects:

- Different researchers, different needs
- Browse only vs. data entry
- Results entry vs. sample prep
- Status reports vs. prep list

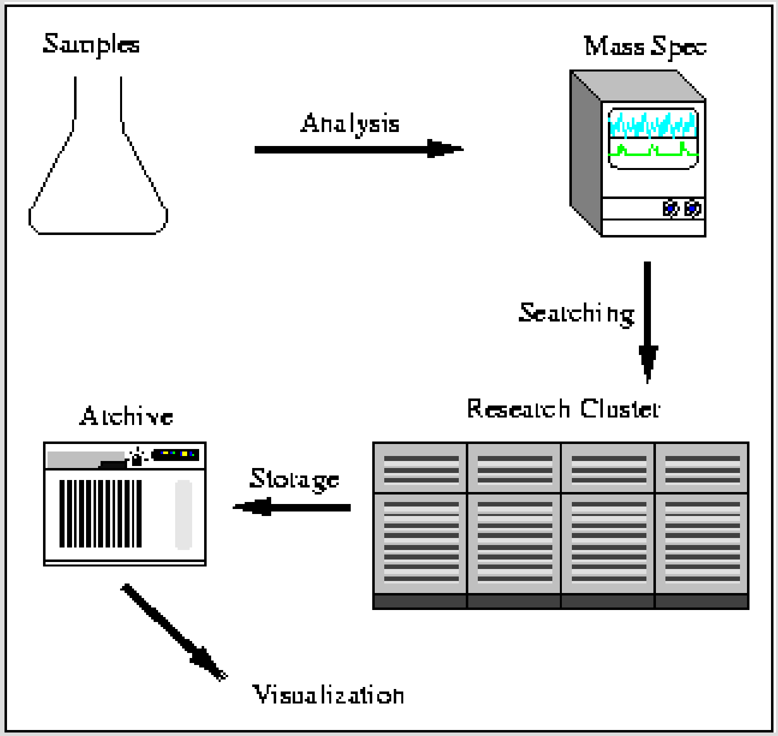
Data paths...

- Access through web app
- Direct access to tables
- Scripts for pre- & post-processing
- Export to stats package (eg. R)
- Export via ODBC to reporting package
- Export via ODBC to desktop tools (spreadsheet, etc)

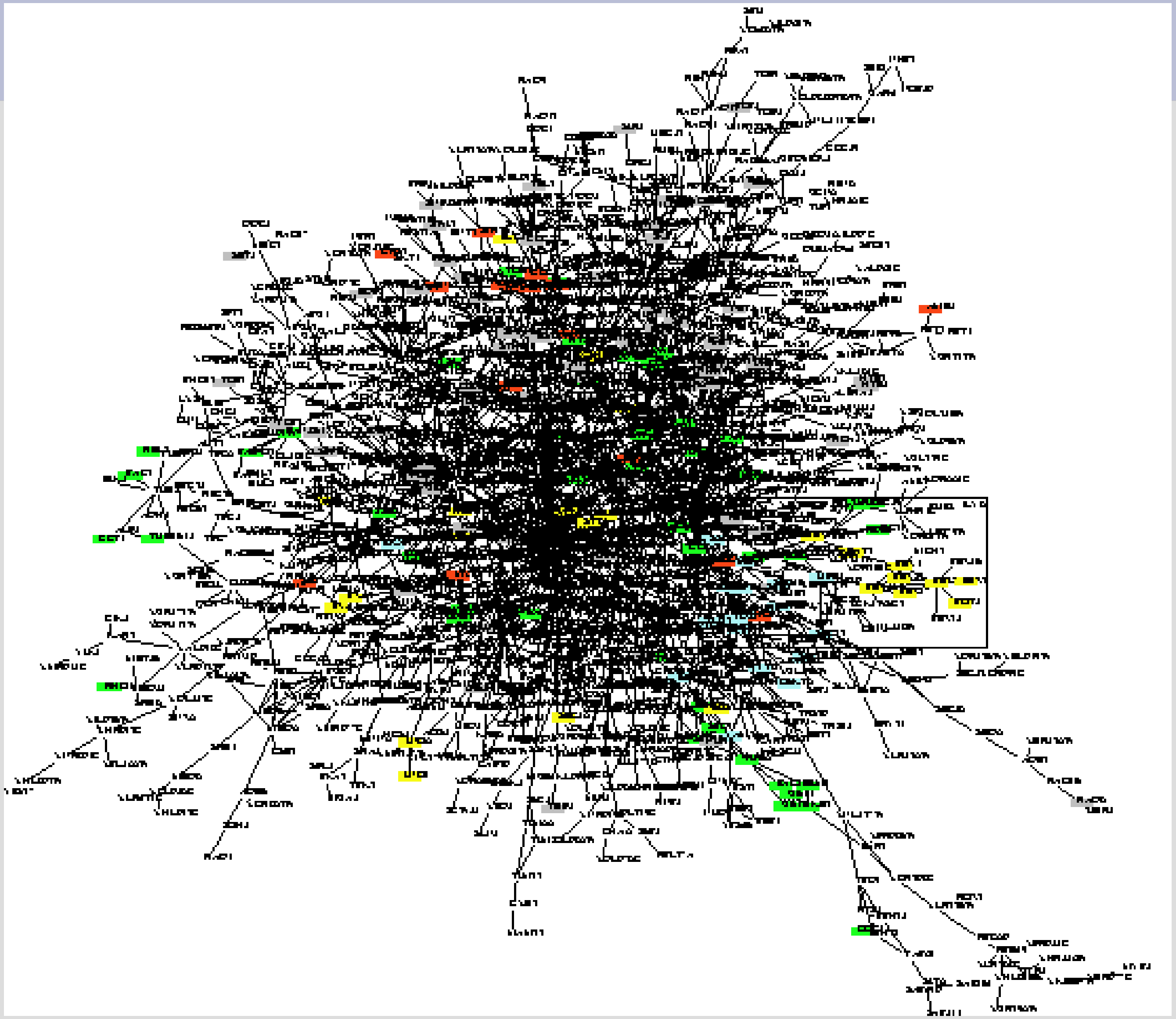
These are not typical of conventional web applications...

Is anything typical?

- Simplified overview of data paths



Still not simple...



FUNCTIONS

Synonyms

- Typical biological problem
- Mapping 'researcher-friendly' identifier to 'database-friendly' identifier
- eg. Lake identifier, watershed identifier, species identifier
- E. coli:

Blattner <-> Strain ID <-> Research ID

Synonyms

```
CREATE TABLE synonym (
  synonym_id   SERIAL PRIMARY KEY,
  blattner     VARCHAR,
  level        INTEGER
);
```

0 = Blattner
1 = Common Synonym
2 = EC record

get_blattner()

```
CREATE FUNCTION get_blattner(TEXT) RETURNS TEXT AS '    DECLARE
        testgene     ALIAS FOR $1;
        gene_rec      RECORD;
        blattner_rec synonym.blattner%TYPE;
    BEGIN
        IF length(testgene) = 0
        THEN
          RETURN ''NULL'';
        ELSE
          SELECT INTO gene_rec *
          FROM synonym
          WHERE synonym = testgene;
          IF NOT FOUND
          THEN
            RETURN ''NULL'';
          ELSE
            SELECT INTO blattner_rec blattner
            FROM synonym
            WHERE synonym = testgene;
            IF NOT FOUND
            THEN
              RETURN NULL;
            ELSE
              RETURN blattner_rec;
            END IF;
          END IF;
        END IF;
    END;'
LANGUAGE 'plpgsql';
```

•

get_ec_number()

```
CREATE FUNCTION get_ec_number(TEXT) RETURNS TEXT
AS 'DECLARE
        testgene ALIAS FOR $1;
        gene_rec RECORD;
        ec_rec   synonym.synonym%TYPE;
    BEGIN
        IF length(testgene) = 0
        THEN
          RETURN NULL;
        ELSE
          SELECT INTO gene_rec *
          FROM synonym
          WHERE blattner = testgene
          AND level = 2;

          IF NOT FOUND THEN RETURN ''NULL'';
          ELSE
            SELECT INTO ec_rec synonym
            FROM synonym
            WHERE blattner = testgene
            AND level = 2;
            IF NOT FOUND
            THEN
              RETURN NULL;
            ELSE
              RETURN ec_rec;
            END IF;
          END IF;
        END IF;
    END;'
LANGUAGE 'plpgsql';
```

get_synonym()

```
CREATE  FUNCTION get_synonym(TEXT) RETURNS TEXT
AS 'DECLARE
        blattner_rec  ALIAS FOR $1;
        synonym_rec   synonym.synonym%TYPE;
    BEGIN
        IF LENGTH(blattner) = 0
                THEN RETURN 'NULL'';
        ELSE
                SELECT INTO synonym_rec *
                FROM synonym
                WHERE blattner = blattner_rec
                AND level = 1;
                IF NOT FOUND
                THEN
                        RETURN 'NULL'';
                ELSE
                        RETURN synonym_rec;
                END IF;
        END IF;
    END;'
LANGUAGE 'plpgsql';
```

•

Conversion functions

- Another typical biological problem
- Formalizing typical conversions when bringing disparate data together
- Ensure consistancy, auditability

SQL is good, but slow, so ...

C is the way to go!

Conversion functions

```
#include <math.h>
#include "postgresql/server/postgres.h"

double *plt_convert_f_to_c(double *deg)
{
	double *ret = palloc(sizeof(double));

	*ret = (*deg + 32.0) * 5.0 / 9.0;
	return ret;
}
```

(Yeah, I know it's Version 0...)

Conversion functions

```
#include <math.h>
#include "postgresql/server/postgres.h"

double *plt_compute_freshwaterdensity_temp(double *temp)
{
	double *ret = palloc(sizeof(double));
	/* yay CRC handbook */
	*ret = 999.83952 + 16.945176 * *temp
		- (7.9870401 / 1000 * pow(*temp, 2))
		- (46.170461 / 1000000 * pow(*temp, 3))
		- (105.56303 / 1000000000 * pow(*temp, 4))
		- (280.54273 / 1000000000000 * pow(*temp, 5)) /
			(1 + 16.87985 / 1000 * *temp);

	return ret;
}
```

VIEWS

Using views...

As persisted queries, views can be used to:

Abstract complexity of non-obvious queries from users

Limit user access to data

Keep difficult SQL query logic in database, not in web interface

Set up once, they keep updating...

Economics Grad Admissions

Data sources:

- External data from School of Graduate Studies (SGS)
- Internal data from Grad Coordinators

Programs:

- M.A., Ph.D., & Non-degree – Dep't Grad Coordinator
- MFE – MFE Program Coordinator

Needs:

- Simple interface for data entry
- Able to integate data from both remote and local
- Present via common desktop tools
- Segregate data by grad coordinator
- Departmental and program-specific stats for applications

Economics Grad Admissions

Data entry interface:

- Prospective student enters personal info via SGS web site
- Internal data from Grad Coordinators via web page

Backend information:

- Local database has SGS data updated via script (called by cron)
- New data added, changes to data updated to database

Local interface:

- Desktop access to data by ODBC, by user authentication
- Linked data export to Access / Excel
- No local coding necessary! :)

Using views...

Perennial problem with complex views -

Accessing a view causes query to be executed; complex queries can cause significant contention on database server when query is hit often

View is good for simple data that changes often, but not for complex data that requires frequent access.

Oh no! What can we do now...?!

STORED PROCEDURES

NCBI Taxonomy Tree

National Center for Biotechnology Information – US NIH
Primary data store for many resources, including:

- PubMed
- Blast Dbs
- Taxonomy Data

“Taxonomy”

- is a hierarchical organization of species
- Kingdom, phylum, class, order, family, genus, species
- represented as node list for a graph
- Large dataset, very minor changes

NCBI Taxonomy Tree

Data description

- **nodes** – taxonomy nodes*
- **names** – species name (scientific, common names)
- **divisions** – broad classiciation of species
- **gencode** – genetic code info
- **delnode** – deleted nodes (for changes)
- **merged** – old & new id of nodes when merged
- **citations** – list of citations for particular nodes*

* Generally useful!

NCBI Taxonomy Tree

Database functions

- **nodes** – taxonomy_import_ncbi_nodes()
- **names** – taxonomy_import_ncbi_names()
- **divisions** – taxonomy_import_ncbi_divison()
- **gencode** – taxonomy_import_ncbi_gencode()
- **delnode** – taxonomy_import_ncbi_deleted()
- **merged** – taxonomy_import_ncbi_merged()
- **citations** – taxonomy_import_ncbi_citations()
  - fairly quickly done (< 1 minute or so)

- **Taxonomy tree** - build_ncbi_taxonomy_tree()
  - Sloooow (20+ minutes to generate tree!)

RULES & TRIGGERS

Facilitating user processing

Difficult to do **everything** via stored procedures
Sometimes best to externalize some data prep, esp for specialized needs
Protect core data via group membership & perms on namespaces / schema
Allow users to do their own processing!

BUT

How to keep intensive user scripts from running unnecessarily, causing contention problems?

Facilitating user processing

Core data tables have rule for UPDATE INSERT DELETE to update 'tables' table

'Tables' table has rule to UPDATE 'scripts' table (set BOOLEAN to TRUE)

User scripts query 'scripts' table via function ..

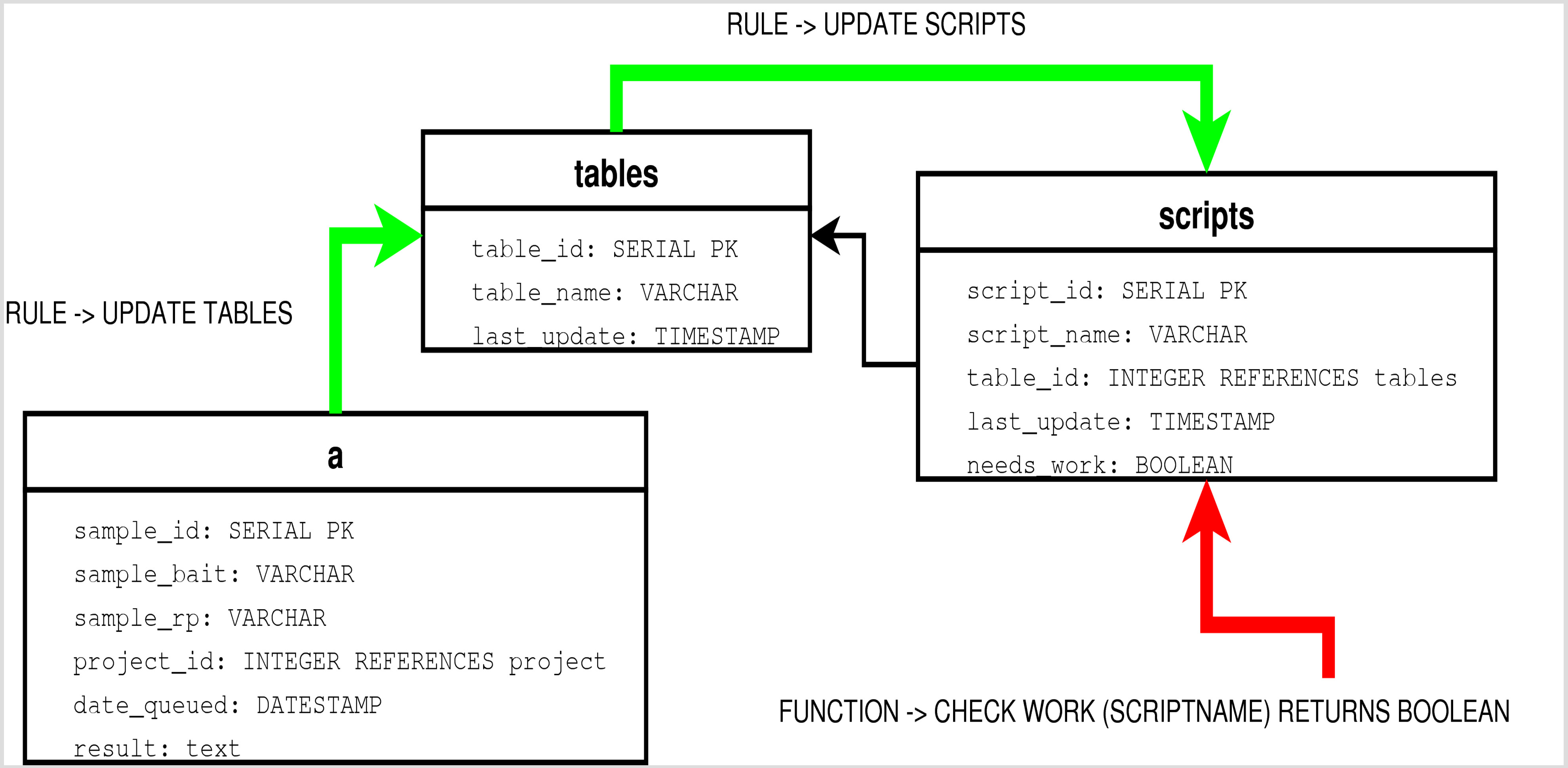
```
SELECT work_check(scriptname);
```

TRUE means one or more underlying tables have changed and user script should run. Once this is done ...

```
SELECT work_done(script_name);
```

... will reset the BOOLEAN flag for the next set of changes.

Facilitating user processing



Facilitating user processing

Advantages

- Useful by any database-aware scripting language (Perl, Python, Ruby, VB/VBA via ODBC, Bash/psql, R, Stata, etc)
- Ability to automate aspects of data maintenance (via cron)
- Play to strengths: RDBMS to manage / manipulate data, stats package to do stats

Disadvantages

- Minor increase in script complexity
- Potential for redundancy in user scripts created by different users
- VERY important to ensure proper privileges set up (eg. read-only access to core data tables by user scripts)

Auditing & Accountability
(or 'things I learned the hard way')

Maintaining a core or project data pool can have problems...

- Errors, mistakes requiring corrections
- Unauthorized ('creative') data changes

Finding the source of irregularities is difficult after the fact.

Importance of keeping the data pool 'clean'

Data entry and 'corrections' should be tracked by individual

Steps in processing should similarly be tracked by individual

Auditing & Accountability
(or 'things I learned the hard way, cont.')

Each member should have their own database account
- Permissions to be given primarily by group membership
- 'Sharing' accounts is not appropriate and unnecessary
- Accounts are cheap!

Web interface to do pass through authentication (based on db user / pass) – NOT stored in web form!

Changes to core data tables should be backed up into parallel tables along with timestamp and CURRENT_USER at change time; keep the backup tuples in a namespace not accessible to 'normal mortals'

Review changes made to all tables daily via script called via cron – identify issues before they become problems.

MATERIALIZED VIEWS

Materialized Views

When is a view not a view?

When it's a table!

MV are tables re-created via stored procedure in response to data changes

General approach:

1) Create table (easy enough)
2) Create procedure to (re)populate table
3) Create trigger(s) to call procedure

```
CREATE FUNCTION mv_tracking_update() RETURNS "trigger"
    AS ' DECLARE
  BEGIN
    /* Clear out table */
    DELETE FROM results.mv_tracking;

    /* Now populate pendings */
    INSERT INTO results.mv_tracking
    SELECT <foo>
      FROM <bar>
     WHERE data.status_id <> ''P'';

    /* and add last 15 completed */
    INSERT INTO results.mv_tracking
    SELECT <foo>
      FROM <bar>
     WHERE data.status_id = ''C''
  ORDER BY last_update DESC
     LIMIT 15;
 RETURN NULL;
 END;'
    LANGUAGE plpgsql;
```

2) Create procedure to (re)populate table

3) Create trigger(s) to call procedure

```
CREATE TRIGGER t_mv_tracking_i
    AFTER INSERT ON data
    FOR EACH STATEMENT
    EXECUTE PROCEDURE
  public.mv_tracking_update();

CREATE TRIGGER t_mv_tracking_u
    AFTER UPDATE ON data
    FOR EACH STATEMENT
    EXECUTE PROCEDURE
  public.mv_tracking_update();
```

Same for delete ...

Materialized Views

Why bother?

Overhead in table repopulation is comparable to running query via view

Procedure only runs when change to data occurs, so lower overhead for table queries

Processing done in 'computer' time, not 'user' time

SQL DATA PREP SCRIPTS

SQL Data Prep Scripts
(or 'more things I learned the hard way')

As the data is being collected, many research users want to 'play' with live data – ODBC + Desktop tools to the rescue!

BUT!

GUI data workup is often useful for exploratory data analysis (EDA) but generally not repeatable (or amenable to automation)

Spreadsheets get very complicated, very quickly with large datasets, and difficult to audit post-priori (= BAD) – also 3 papers on numeric inaccuracies in Excel

Difficult to track errors from changes and unit magnitudes

SQL Data Prep Scripts
(or 'more things I learned the hard way, cont')

Write an SQL script!

SQL scripting is relatively simple (compared to most stats packages)

SQL scripting is EASILY auditable (data manipulations via SQL are very clear)

SQL scripts are EASILY commented – inclusion of description, logic, revision history & amenable to CVS/SVN

Namespaces can be used to maintain multiple versions or approaches of data work up (snapshots in time)

FUTURE WORK

What's Next?

Some things we're looking at ...

Creation of compound UDTs, a 'profile' data type for limnological research + supporting functions

Database-mediated distributed analysis system for high-resolution time-series data

Kerberos-based authentication for student / staff / faculty information system

SUMMARY

In Summary ...

In our experience,PostgreSQL's inherent flexibility, extensibility and speed make it an excellent research and academic database system

Views, Stored Procedures, Rules & Triggers, MVs all offer tremendous opportunities for web and research developers to maximize application capability while minimizing nLOC required

'Don't be afraid of put logic in the DB rather than everything in the web form'

THANKS!

To all the PostgreSQL developers for making this possible!

Ger Cagney – RIS
Thomas Kislinger – MARS / UHN

Peter Dillon – OMEE / Trent University
Martyn Futter – OMEE / Laurentian University

Department of Economics, U of T
Genome Canada
NCE-SFN Aquatic Group
Colleagues from Emili Lab – U of T