Migration to PostgreSQL
Preparation and Methodology

Joe Conway, Michael Meskes

credativ International

October 21, 2014
PostgreSQL (and Linux) user since 1999
  - Community member since 2000
  - Contributor since 2001
  - Commiter since 2003

PostgreSQL Features
  - PL/R
  - Set-returning (a.k.a. table) functions feature
  - Improved bytea and array datatypes, index support
  - Polymorphic argument types
  - Multi-row VALUES list capability
  - Original privilege introspection functions
  - pg_settings VIEW and related functions
  - dblink, connectby(), crosstab(), generate_series()
Joe Conway - Business

- Currently President/CEO of credativ USA
- Previously IT Director of large company
- Wide variety of experience, closed and open source
- Full profile: http://www.linkedin.com/in/josepheconway
Since 1993 Free Software

Since 1994 Linux

Since 1995 Debian GNU/Linux

Since 1998 PostgreSQL, mostly ECPG
Michael Meskes - Business

- 1992 - 1996 Ph.D
- 1996 - 1998 Project Manager
- 1998 - 2000 Branch Manager
- since 2000 President of credativ Group
Why migrate?

Free and Open Source Software

- No licence cost
- Open standards
- High quality software and support
- White box testing
- Tailor-made standard software
- Independence
- Protection of investment
Intro to Migration

- Choose a capable manager
- Create a solid planning basis
- Design top-down, implement bottom-up
- Consider all processes and data traffic
- No interim, temporary or isolated solutions
- Essential parts have to be redundant
- Remember training, maintenance and support
Porting projects are hard
SQL Standard and compatibility layers are not a panacea
You might be better off not migrating
Success can pay off big

Disclaimers:
Presentation written from perspective of PostgreSQL expert
Almost anything is possible; we are looking for reasonable options
3 hours is not nearly enough time to cover this topic in depth
Case Study - Best case

- Admins know PostgreSQL
- Middleware supports PostgreSQL
- Standard datatypes
- Standard SQL code

⇒ Only one hour of work
⇒ Instant Return on Investment!
Case Study - Lots of licenses

- 600 Installations
- $5,000 per database server
- $150,000 up-front migration costs
- $2,000 additional rollout costs
- 25 rollouts per month

⇒ Return on Investment: 2 months after begin of rollout!

Joe Conway, Michael Meskes
PGConf.EU 2014
Case Study - Lot of Migration Work

- 1800 installations with 2 servers each
- $2,000 per installation per year
- Migration costs $1,000,000
- $1,000 additional rollout costs
- 125 rollouts per month

⇒ Return on Investment: 8 months after begin of rollout!
Inventory Your Requirements

- What features of the incumbent database are in use by your application?
- Which of them are unique and likely need substitution?
- What PostgreSQL specific features would bring great benefits?
- What are your upcoming requirements?
Inventory Your Requirements

Requirements to consider

- Data Types
- Database Object Types
- SQL Syntax
- Stored Functions and/or Procedures
- Client libraries
- Encodings
- Replication and/or High Availability
- Extensions
Data types

PostgreSQL supported Data Types:

- INTEGER, NUMERIC, DOUBLE PRECISION
- CHARACTER (CHAR), CHARACTER VARYING (VARCHAR), TEXT
- TIMESTAMP WITH[OUT] TIME ZONE, INTERVAL
- BYTEA, BOOLEAN, BIT

Data types

PostgreSQL supported Data Types

- large object
- spatial, geometric
- full text
- JSON, XML, UUID, network address
- composite, array, enumerated
- others ...


http://www.postgis.org/documentation/manual-1.5/
Database Object Types

PostgreSQL supported Object Types

- DATABASE, SCHEMA
- USER, GROUP, ROLE
- TABLE, INDEX, SEQUENCE, VIEW, FOREIGN
- FUNCTION, AGGREGATE, TRIGGER, RULE, OPERATOR
- TYPE, DOMAIN, CAST, COLLATION, CONVERSION
- EXTENSION, LANGUAGE, TABLESPACE, TEXT SEARCH
SQL Syntax

- Identifiers
  - UPPER
  - lower
  - MiXeD_cAsE

- NULL value handling

- Sub-selects
  - target list
  - FROM clause
  - WHERE clause
  - correlated
  - uncorrelated
SQL Syntax

- Outer joins
- WITH clause
- WINDOW clause
- UPSERT/MERGE
Stored Functions and/or Procedures

- **PostgreSQL supports Stored Functions**
  
  ```
  SELECT a, foo(b) FROM bar;
  SELECT a, b FROM foo() AS t(a, b);
  ```

- **PostgreSQL does not support Stored Procedures**
  
  ```
  EXEC sp_foo(42);
  CALL sp_bar('abc');
  ```
Stored Functions

- PL/pgSQL similar to PL/SQL
- Also distributed with PostgreSQL
  - C, SQL, Perl, Python, Tcl
- Other languages available:
  - Java, PHP, Ruby, R, Shell, others . . .
Client libraries

PostgreSQL supported Client Libraries

- Interface available in virtually every programming language
  - Check syntax and semantics
  - Use Database agnostic interface, e.g. Perl DBI
- ODBC, .Net, JDBC
- ECPG
Encodings

PostgreSQL supported Encodings

- Too many to list
- Pay attention to:
  - server vs. client-only encodings
  - compatible conversions and locale settings

See:
Replication and/or High Availability

PostgreSQL supported HA and Replication Options
- Covered separately later in this presentation
Extensions

- Current "other" database extensions in use
  ⇒ Check equivalent PostgreSQL extension availability

- Existing PostgreSQL extensions
  ⇒ Leverage where it makes sense

- Missing PostgreSQL extensions
  ⇒ Write your own!
Database Conversion

General Thoughts

- Practice, practice, practice, ...
- Plan final conversion well in advance
- Convert
- Check
- Go live!
Practice

- **Script your conversion**
  - Figuratively: document the steps to be taken
  - Literally: automate the data processing and checking as much as possible
- **Identify criterion to declare success**
  - No unexpected errors
  - Time meets available window
  - One or more methods to check result for correctness
- **Execute your conversion script, beginning to end**
- **Rinse and repeat until consistently flawless**
Conversion - possible methodologies

- **Hard cutover**
  - Requires downtime
  - Provides cleanest result

- **Continuous cutover**
  - Use external replication or manual sync
  - Minimal downtime
  - Tricky to do
  - Very difficult to verify absolute correctness

- **Dual entry/overlap system operation**
  - No downtime
  - Laborious and error prone
  - Provides easy fallback
Check

- Logged ERRORs and WARNINGs
- Row counts
- Data sampling
- Data diffs
- A-B-A test
- Application regression testing
Both Oracle and PostgreSQL support plenty of SQL-conforming data types.

But usually the nonconforming ones are in wider use.

Thin compatibility layers can usually help, but that will make your PostgreSQL application ugly.

A big search-and-replace is usually in order.
Data Types: Specifics

- VARCHAR2 → VARCHAR or TEXT
- CLOB, LONG → VARCHAR or TEXT
- NCHAR, NVARCHAR2, NCLOB → VARCHAR or TEXT
- NUMBER → NUMERIC or BIGINT or INT or SMALLINT or DOUBLE PRECISION or REAL (bug potential)
- BINRAY_FLOAT/BINARY_DOUBLE → REAL/Doubles PRECISION
- BLOB, RAW, LONG RAW → BYTEA (additional porting required)
- DATE → DATE or TIMESTAMP
Null Values

- Infamous Oracle behaviour: `NULL = ''`
- Consequently, `' ' = ''` is not true
- Completely weird and inconsistent
- Usually, your data will just disappear in PostgreSQL
- `transform_null_equals` does not help here

http://www.postgresql.org/docs/9.4/interactive/runtime-config-compatible.html#

`GUC-TRANSFORM-NULL-EQUALS`

- If your application relies on any of this, you are in trouble.
Sequences: Creating

Sequences are somewhat compatible . . .

- Change NOCACHE to CACHE 1 (or omit).
- MAXVALUE 9999999999999999999999999 needs to be reduced.

Don’t rely on the caching behaviour.
Sequences: Using

- Oracle syntax: `sequence_name.nextval`
- PostgreSQL syntax: `nextval('sequence_name')`

Search-and-replace; but direct sequence calls are rare.
ROWNUM and ROWID

ROWNUM:
- Use row_number() WINDOW function
- Use generate_series()
- Rewrite and apply LIMIT
- Just handle in the client

ROWID:
- Analogous to ctid
- Good code should usually not use this.
- That does not prevent some from trying.
Identifiers  Oracle case folds to upper case, PostgreSQL to lower case. Big trouble if you mix quoted and unquoted identifiers.

MINUS  Change to EXCEPT.

SQL key words  Usually not a big problem, but should be kept in mind.

“FROM dual”  Easy to work around (or use orafce).
Outer Joins

- PostgreSQL only supports the SQL-standard outer join syntax.
- Oracle supports it since version 9.
- Much Oracle code uses the old, Oracle-specific syntax.
- Porting is usually straightforward, but requires manual work.
- Set up test queries to catch porting mistakes.
Functions: General

- Function compatibility is a bottomless pit.
- PostgreSQL (+ orafce) supports many Oracle compatibility functions.
- It’s easy to write your own.
- Only the special syntax spells trouble.
Functions: Compatibility

For example, the following common functions are supported by PostgreSQL as well:

- `substr`
- `to_char`
- `nvl`, `nullif` (orafce)
Functions: Specifics

Manual work required here:

- `sysdate → current_timestamp` or `localtimestamp`
Functions: decode

DECODE(expr, search, expr, ... [, default])

becomes

CASE WHEN expr THEN search .. ELSE default END
Client Libraries

- OCI ⇒ rewrite with libpq
- ODBC ✓
- JDBC ✓
- Perl-DBI ✓
- Pro*C ⇒ use ECPG
  Lot of additions for compatibility.
Usage

ecpg prog1.pgc
# (creates prog1.c)

cc -c -I/usr/include/postgresql prog1.c
# (creates prog1.o)

cc -o prog prog1.o ... -lecpg
# (creates prog)
ECPG

- Mostly works out of the box
- Parser
- Runtime: Pro*C as blueprint
Parser

- Connect database syntax
- EXEC SQL VAR
- EXEC SQL TYPE
- EXEC SQL IFNDEF
Host variables

EXEC SQL BEGIN DECLARE SECTION; /* needed for ECPG */
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;
...
do {
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
Encodings

- Both Oracle and PostgreSQL support the same ideas.
- But everything is named differently.
- Might be a good time to review the encoding and locale choices.
https://github.com/orafce/orafce

- Large set of Oracle compatibility functions
- “dual” table
- Debian and RPM packages available
- Invaluable
ora2pg

http://ora2pg.darold.net/

- Converts Oracle schema definitions
- Extracts data from Oracle database for import into PostgreSQL
- Packages available
- Invaluable
http://tora.sourceforge.net/

- GUI for PostgreSQL and Oracle
- Contains exploration and debugging facilities for Oracle
- Packages available, but usually without Oracle support
- Generally a bit outdated, but good for this purpose
Things That Won’t Work Directly

- CONNECT BY: Try contrib/tablefunc or WITH RECURSIVE.
- Materialized views: Exists in 9.3; improved in 9.4.
- Snapshots: Write your own wrapper.
- Database links: Use contrib/dblink plus views or FDW.
- Autonomous transactions: Try dblink.
- Synonyms: Try views or wrapper or schema path.
- Partitioning: Use inheritance, check constraints, and constraint exclusion.
Coincidence?

If you need help:

Oracle  Ask Tom: http://asktom.oracle.com/
PostgreSQL  Ask Tom: tgl@sss.pgh.pa.us
Datatype Mapping - Numeric Types

Numeric Datatypes in Informix are mostly compatible with PostgreSQL datatypes

- SERIAL present in PostgreSQL with different syntax
- SMALLINT
- INTEGER
- FLOAT
- SMALLFLOAT $\Rightarrow$ REAL or FLOAT4
- DECIMAL($p$, $s$) $\Rightarrow$ NUMERIC($p$, $s$)
Character datatypes

- `CHAR(n), NCHAR(n) ⇔ CHAR(n), CHARACTER(n)`
- `VARCHAR(n,r), NVARCHAR(n,r), CHARACTER VARYING(n,r) ⇔ VARCHAR(n)`
- Variables length types can be larger than 255 bytes in PostgreSQL
- No minimal length specifier `r` in PostgreSQL
- `TEXT` must be handled carefully: Informix allows arbitrary encoded literals in such columns ⇒ encoding issues
- `LVARCHAR` ⇒ `TEXT` or `VARCHAR`
Binary datatypes

- BYTE, BLOB, CBLOB ⇒ BYTEA
- Handling different: PostgreSQL allows direct access to bytea columns
- Different output formats: bytea_output
- TEXT ⇒ BYTEA or TEXT
Binary datatypes - Hints

1. Binary datatypes should be matched to BYTEA
2. Textual datatypes like TEXT must be carefully evaluated: they might contain different encodings, which can’t be used with PostgreSQL’s TEXT datatype
3. Handling of BYTEA is much easier in PostgreSQL
4. The old LOB interface in PostgreSQL should only be used when values larger than one GByte must be stored.
Complex datatypes

- **SET** $\Rightarrow$ array type, issues remain (e.g. uniqueness of elements aren’t checked in PostgreSQL arrays)
- Same with **MULTISET**, but it also allows duplicate entries in Informix
- **LIST** $\Rightarrow$ ENUM or array type
- **ROW** $\Rightarrow$ composite types in PostgreSQL (CREATE TYPE)
- No datatype inheritance in PostgreSQL (CREATE TYPE...UNDER())

Generally, migrating such types require deep investigation how they are used and implemented in the application.
User Defined Functions - SPL

SPL should be migrated to PL/PgSQL

- Named Parameters and default parameters are supported since PostgreSQL 9.0
- Syntax differences in declarations, conditional statements
- PROCEDURES with CALL have a different notion in PostgreSQL
- Parameter declaration DEFINE must be moved into DECLARE section.
- LET variable assignments are done with :=.
- Migrating cursor usage within a FOREACH statement
Client Libraries

- 4GL ⇒ Aubit (http://aubit4gl.sourceforge.net)
- ODBC ✓
- JDBC ✓
- ESQL/C ⇒ use ECPG
  Lot of additions for compatibility.
Overview
Oracle to PostgreSQL
Informix to PostgreSQL
MySQL to PostgreSQL
MSSQL to PostgreSQL
Replication and/or High Availability
Discussion

Data Types
Stored Functions and/or Procedures
Client Libraries

ECPG

- Mostly works out of the box
- Compatibility modes: INFORMIX, INFORMIX_SE
- Parser
- Runtime behaviour
- Compatibility library
Overview
Oracle to PostgreSQL
Informix to PostgreSQL
MySQL to PostgreSQL
MSSQL to PostgreSQL
Replication and/or High Availability
Discussion

Data Types
Stored Functions and/or Procedures
Client Libraries

Parser

- EXEC SQL \Rightarrow\$ 
- EXEC SQL IFDEF|IFNDEF|ELSE|ELIF|ENDIF 
- EXEC SQL VAR 
- EXEC SQL TYPE 
- EXEC SQL CLOSE database
Runtime

- NULL handling: risnull(), rsetnull()
- SQLDA handling
- Data conversion
- Error codes
- Decimal type
Compatibility Library

- ESQL/C Function Library ⇒ PGTypeslib
- Decimal: decadd(), ...
- Date: rdayofweek(), ...
- Datetime: dtcurrent(), ...
- Interval: intoasc(), ...
- Misc: rupshift(), ...
Overview
Oracle to PostgreSQL
Informix to PostgreSQL
MySQL to PostgreSQL
MSSQL to PostgreSQL
Replication and/or High Availability
Discussion

Resources

- **PostgreSQL Wiki:**
  
  http://wiki.postgresql.org/wiki/Converting_from_other_Databases_to_PostgreSQL#MySQL

- **mysql**
  
  `--compatible=postgresql`  
  $\Rightarrow$ Equivalent to PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS


- **MySQL built-in-function equivalents**


- **pgloader**

  http://pgloader.io/index.html
Cautions

Even when syntax matches, semantics can be different

- MySQL behavior of out-of-range/overflow/bad values with strict mode off

- Semantics of familiar operators, e.g.

  SELECT 10^3; --> 9 : In MySQL
  SELECT 10^3; --> 1000 : In Postgres
  SELECT '1' || '0'; --> 1 : In MySQL
  SELECT '1' || '0'; --> '10': In Postgres

- Therefore – test, test, test, …
Too many combinations/types to cover exhaustively
Data type aliases make this worse
Integers

- MySQL: 1, 2, 3, 4, 8 byte signed/unsigned integers
  ⇒ TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
- MySQL: supports attributes display width and ZEROFILL
  ⇒ INT(4) ZEROFILL column would display 42 as 0042
- Postgres: 2, 4, 8 byte signed integers, 1 byte ”char”
  ⇒ SMALLINT, INTEGER, BIGINT, ”char”
- BIGINT UNSIGNED ⇒ NUMERIC or DOUBLE PRECISION
- INT UNSIGNED and BIGINT ⇒ BIGINT
- Everything else ⇒ INT
- 2 byte integers and ”char” usually don’t save space (alignment)
Floating Point Numbers

- MySQL: 4, 8 byte, signed/unsigned floating point types
  $\Rightarrow$ FLOAT, DOUBLE

- MySQL: supports attributes precision and scale
  $\Rightarrow$ FLOAT(5,3) column would round 99.0009 as 99.001

- Postgres: 4 and 8 byte signed floating point types
  $\Rightarrow$ REAL, DOUBLE PRECISION

- FLOAT $\Rightarrow$ REAL
- DOUBLE $\Rightarrow$ DOUBLE PRECISION
- MySQL UNSIGNED max value is same as signed
MySQL: NUMERIC, DECIMAL

MySQL: supports attributes precision and scale
⇒ NUMERIC(5,3) column would round 99.0009 as 99.001

Postgres: NUMERIC

Postgres: supports attributes precision and scale
⇒ NUMERIC(5,3) column would round 99.0009 as 99.001

NUMERIC, DECIMAL ⇒ NUMERIC

PostgreSQL precision greater than MySQL so not out-of-range concern
Character

- MySQL: CHAR, VARCHAR, TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT
  ⇒ each has different max length
- Postgres: CHAR, VARCHAR, TEXT
  ⇒ all have the same max length
- CHAR, VARCHAR, TEXT ⇒ CHAR, VARCHAR, TEXT
- LONGTEXT can exceed maximum length allowed in PostgreSQL
- MySQL TEXT types have index/sorting differences from Postgres
**Date/Time**

- MySQL: DATETIME, DATE, TIMESTAMP, TIME, YEAR
- Postgres: DATE, TIMESTAMP and TIME (WITH/WITHOUT TIME ZONE), INTERVAL
- DATETIME, TIMESTAMP \(\Rightarrow\) TIMESTAMP
- DATE \(\Rightarrow\) DATE
- TIME \(\Rightarrow\) TIME, INTERVAL
- YEAR \(\Rightarrow\) no direct match
- Generally Postgres types have more range
- strict mode off/ALLOW_INVALID_DATES, expect errors
MySQL DATABASE similar to Postgres SCHEMA
If joining data across databases, Postgres SCHEMA best choice
But be careful security differences in multi-tenant situations
MySQL USER similar to Postgres

Postgres GROUP/ROLE provide additional capability

Wildcard GRANTs

⇒ PL/pgSQL function
⇒ DO
⇒ ALL TABLES IN SCHEMA schema_name
TABLE, VIEW, INDEX

- Basic syntax OK
- AUTO_INCREMENT ⇒ SERIAL
- Watch semantics of options
- Devil is in the details

EVENT

- No PostgreSQL equivalent
- Use cron
- FDW support expanded with PostgreSQL 9.1
- MySQL and many others quickly becoming available
  http://wiki.postgresql.org/wiki/Foreign_data_wrappers
- MySQL only supports mysql wrapper
TRIGGER

- MySQL trigger contains executed SQL
- PostgreSQL trigger refers to function
- Otherwise basic syntax similar
General

- Comments: `#` ⇒ `--` or `/* */`
- Literal Quoting: `'` or `"` ⇒ `'` or `$$`
- String Comparison: case-insensitive ⇒ case-sensitive
- Identifier Quoting: `′` (backtick) ⇒ `"`
- Identifier Comparison: case-insensitive ⇒ case-sensitive

http://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL
String Comparison

MySQL:
SELECT "a" = "A" AS t;
+-----+
| t   |
+-----+
| 1   |
+-----+
1 row in set (0.03 sec)

PostgreSQL:
SELECT 'a' = 'A' AS f, lower('a') = lower('A') as t;
f | t
-----+
| f   | t
+-----+
| f   | t
(1 row)

-- also consider citext
Identifier Comparison

MySQL:

CREATE TABLE Foo (id integer);
Query OK, 0 rows affected (0.13 sec)

CREATE TABLE foo (id integer);
Query OK, 0 rows affected (0.15 sec)

PostgreSQL:

CREATE TABLE Foo (id integer);
CREATE TABLE

CREATE TABLE foo (id integer);
ERROR: relation "foo" already exists
Example: Tables with Triggers

MySQL:

CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4(
    a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    b4 INT DEFAULT 0
);

PostgreSQL:

CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 SERIAL PRIMARY KEY);
CREATE TABLE test4(
    a4 SERIAL PRIMARY KEY,
    b4 INT DEFAULT 0
);
Example: Tables with Triggers (cont.)

MySQL:

delimiter |
CREATE TRIGGER testref BEFORE INSERT ON test1
FOR EACH ROW BEGIN
  INSERT INTO test2 SET a2 = NEW.a1;
  DELETE FROM test3 WHERE a3 = NEW.a1;
  UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
END;|
delimiter ;
Example: Tables with Triggers (cont.)

PostgreSQL:

CREATE OR REPLACE FUNCTION testref_tgf() returns trigger as $$
BEGIN
  INSERT INTO test2(a2) VALUES (NEW.a1);
  DELETE FROM test3 WHERE a3 = NEW.a1;
  UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  RETURN NEW;
END; $$ language plpgsql;
CREATE TRIGGER testref BEFORE INSERT ON test1
FOR EACH ROW EXECUTE PROCEDURE testref_tgf();
Example: Tables with Triggers (cont.)

MySQL and PostgreSQL:

```sql
INSERT INTO test3 (a3) VALUES
  (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT),
  (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT);
INSERT INTO test4 (a4) VALUES
  (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT),
  (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT);
INSERT INTO test1 VALUES (1), (3), (1), (7), (1), (8), (4), (4);
```
Example: Tables with Triggers (cont.)

MySQL:

SELECT * FROM test1;
SELECT * FROM test2;
SELECT * FROM test3;
SELECT * FROM test4;

PostgreSQL:

SELECT * FROM test1;
SELECT * FROM test2;
SELECT * FROM test3;
SELECT * FROM test4 order by 1;
REPLACE/UPsert

- REPLACE: Replaces existing row on duplicate key
- ON DUPLICATE KEY UPDATE: updates existing row on duplicate key
- In PostgreSQL use PL/pgSQL function
- Be careful about race behavior in high concurrency environments

http://www.postgresql.org/docs/9.1/static/plpgsql-control-structures.html
MySQL: use `LAST_INSERT_ID()` with `AUTO_INCREMENT`

PostgreSQL: use `INSERT INTO (...) RETURNING (...)`
PostgreSQL does not support procedures
⇒ Use a function where possible, or external SQL script

MySQL UDFs must be written in C or C++
⇒ Port to PostgreSQL C function

Consider replacing with PL/pgSQL, SQL, or other PL functions

Leverage significant flexibility of PostgreSQL functions
Client Libraries

- PostgreSQL has equivalent for virtually all MySQL
- Depending on library/language, some client conversion needed
  - JDBC, ODBC, DBI ⇒ probably minimal
  - Some (e.g. PHP) more extensive but straightforward
- Watch out for semantic differences
MySQL has somewhat more granular encoding and collation support

PostgreSQL has no option for per table or per column encoding

PostgreSQL does have option for per column collation
MSSQL: General Considerations

- Many considerations similar to Oracle and MySQL
- Simple database schemas should convert easily
- Semantic differences can still bite you, especially case-sensitivity
- Stored procedures likely to be significant issue

http://wiki.postgresql.org/wiki/Microsoft_SQL_Server_to_PostgreSQL_Migration_by_Ian_Harding
Numeric Types

- **IDENTITY** \(\Rightarrow\) **SERIAL**
- **SMALLINT, INTEGER, BIGINT** \(\Rightarrow\) **SMALLINT, INTEGER, BIGINT**
- **TINYINT** \(\Rightarrow\) possibly "char"
- **FLOAT, REAL, DOUBLE PRECISION** \(\Rightarrow\) **REAL, DOUBLE PRECISION**
- **NUMERIC, DECIMAL** \(\Rightarrow\) **NUMERIC**
Character datatypes

- CHAR, NCHAR → CHAR
- VARCHAR, NVARCHAR → VARCHAR
- TEXT, NTEXT → TEXT
Date and Time datatypes

- DATE, TIME, DATETIME  ⇒  DATE, TIME, TIMESTAMP
- DATETIMEOFFSET  ⇒  TIMESTAMMP WITH TIME ZONE
Binary datatypes

- BINARY, VARBINARY, IMAGE \(\Rightarrow\) BYTEA
PostgreSQL does not support procedures
⇒ Use a function where possible, or external SQL script

MSSQL FUNCTION somewhat similar to PostgreSQL
⇒ T-SQL port to PL/pgSQL function
⇒ CLR port to C function or other PostgreSQL PL
What’s In A Term?

- Replication
- Clustering
- High availability
- Failover
- Standby

Putting data on more than one computer
Solution Space

Narrowing the Range of Possibilities

- Goals
  - What do you want to achieve?
- Techniques
  - How to implement?
- Solutions
  - What software is available?
Possible Goals

- High availability
- Performance
  - Read
  - Write
- Wide-area networks
- Offline peers
Goal: High Availability

- Provisions for System Failures
  - Software Faults
  - Hardware Faults
  - External interference
Goal: Read Performance

- Applications with:
  - many readers (e.g. busy mostly read-only website)
  - resource intensive (e.g. data warehouse)
- Distribute the readers over more hardware
- Often one physical machine is sufficient
Goal: Write Performance

- Applications with:
  - many writers (e.g. busy social networking website)

- Distribute the writers over more hardware
  - constraint checking and conflict resolution are difficult

- Faster writing and replication contradict
  - Partition (shard), don’t replicate
  - RAID 0 is not replication
  - RAID 10 is good idea, but does not solve the problem
Goal: Optimizing for Wide-Area Networks

- Faster access across WANs
- Reading
  - Local copies
- Writing
  - Synchronization
Goal: Offline Peers

- Synchronize data with laptops, handhelds, . . .
- Road warriors
- May be considered very-high-latency WANs
Techniques

- Replication
- Proxy
- Standby system
Techniques: Replication

- Synchronous vs. Asynchronous
- Multi-Master vs. Master/Slave
- Shared Storage vs. Shared Nothing
- Mechanism for detecting update
  - Triggers
  - Logs
  - 'Updated’ Field
- Conflict Resolution
  - Master/Slave: unneeded
  - Synchronous Multi-Master: two-phase commit process
  - Asynchronous Multi-Master: rule based
Techniques: Proxy

- Connection pooling
- Load balancing
- Replication
- Sharding/Parallel Query
Techniques: Standby System

- File system level
- Log shipping
Solutions

- Replication
- Proxy
- Standby system
Overview
Oracle to PostgreSQL
Informix to PostgreSQL
MySQL to PostgreSQL
MSSQL to PostgreSQL
Replication and/or High Availability
Discussion

Solutions: Replication

- Hot standby
  http://www.postgresql.org/docs/9.4/static/hot-standby.html

- Slony-I
  http://www.slony.info/

- Bucardo
  http://bucardo.org/wiki/Bucardo

- Londiste
  http://pgfoundry.org/projects/skytools/
Solutions: Proxy

- **pgpool-II**

- **PL/Proxy**
Solutions: Standby System

- DRDB
- Continuous Archiving
  - 'Out of the box'
  - pg_standby
- OmniPITR
  - https://github.com/omniti-labs/omnipitr
- repmgr
  - http://projects.2ndquadrant.com/repmgr
Questions?