Metatron Technology Consulting 's MySQL to PostgreSQL migration guide

Chris Travers

October 13, 2005

Contents

1 Why Migrate?

Information is the life-breath of any business in the way that money is its life-blood. Many businesses today use Relational Dabase Management Systems (RDBMS's) to manage this critical asset. As such, it is important that the RDBMS chosen to manage this information is as robust and powerful as possible. PostgreSQL offers many powerful tools to provide information from your database for a

variety of uses, including integration of diverse applications, reporting, and a variety of other uses. PostgreSQL is also quite scalable and extremely robust.

PostgreSQL is also the most standards-compliant open source database around, implimenting more SQL-99 features than MySQL or FirebirdSQL. It has a very vibrant community, and is free from many of the licensing issues that have, as of the time of this writing, surfaced with MySQL. Use or even distribution of PostgreSQL will never require an additional license from a commercial entity, for example.

PostgreSQL also performs better than MySQL in many real-world scenarios. While MySQL does perform better for simple read-only operations when transactional control is disabled, PostgreSQL handles more complex queries with ease, and provides better performance under high load than MySQL, especially when some users are writing to the database. Therefore, while MySQL is quite well suited for simple content management tools where there is no need to integrate it with other line of business applications, PostgreSQL is better for any application of reasonable complexity.

Finally, as of version 4.x, MySQL does not strictly check the validity of information written to the database. Numbers may be silently truncated, for example, or invalid dates, such as Feb 31, 2005, could be entered into the database. While this is possibly acceptible where only a single application is using the database, it becomes unacceptable quickly when several independent applications must access the same data because a bug in any one application could allow invalid (or worse, erroneous) data to be saved. Even when strict mode debuts in 5.x, it is unlikely that this will be enabled by default as MySQL will be largely required to be backwards compatible.

This document is not all-encompassing. In many areas, it will reference the main PostgreSQL and/or MySQL manuals for more information. However, before beginning a migration, you will want to at least familiarize yourself with the material in this guide and where to go for more information.

2 Differences Between PostgreSQL and MySQL

MySQL and PostgreSQL are not "100% compatible." From an administration and development perspective there are important differences which must be kept in mind. These differences range from session variables to how queries are executed, large object interfaces, and database maintenance requirements.

2.1 MySQL features missing in PostgreSQL

2.1.1 Session Variables

MySQL allows the use of session variables from SQL commands. This is usually only really required in the command line client, so it should not be a big deal, though some applications may use it to decrease the round-trips to the server. In these cases, you will want to port the queries using these into a user defined function in PostgreSQL or use client-side variables.

In MySQL, session variables might be used like:
SELECT @myvar:= id from mytable where colname = 'test';

In PostgreSQL session variables do exist in some procedural languages, most notably PL/Perl(U). It is also possible to create something similar using temporary tables and a couple of functions (there are some landmines here, btw).

2.2 PostgreSQL features missing in MySQL

2.2.1 MVCC

MVCC allows for better performance under load because readers never block writers. It also allows for a larger number of transaction isolation levels. For example the READ COMMITTED isolation level allows for transactions to read the values of all rows in the database as they were when the read operations began.

MVCC also makes rollbacks as inexpensive as commits. The read operation only needs to know that the transaction rolled back, and can then safely ignore the values it wrote to the database.

The downside of MVCC is that obsolete information remains stored in the database after it is updated because of the possibility that it may be required by a running transaction. Therefore, databases must be periodically vacuumed to remove these dead tuples.

InnoDB tables do use MVCC, but lack any way of removing dead tuples from the database. PostgreSQL requires periodic vacuuming of the database to remove dead tuples. This creates a number of problems including table bloat/sparceness, and in some cases serious performance issues. Several of our customers have run into this problem.

With the recent Oracle acquisition of Innobase, it is unclear whether InnoDB will continue to be available as part of future versions of MySQL. The problem is that although InnoDB table types are available under the GPL, MySQL cannot then relicense this code to customers without an agreement with Oracle. The contract comes up for renewal in 2006, and it will be interesting to see what transpires at that point (Oracle has voiced an intent to "renegotiate" the terms of the contract, whatever that means).

2.2.2 WAL and Point in Time Recovery

The Write-Ahead Log (WAL) allows PostgreSQL to provide absolute data consistance in the event of an expected shutdown (such as a power outage) without sacrificing much performance. After every write operation, the WAL is flushed to disk. If the database detects an unclean shutdown of the database server, it will attempt to run through the transactions on the WAL in the order that they were performed in order to bring the database up to the exact state it was before the shutdown.

The WAL also allows the database to be recovered to where it was at a point in time other than the shutdown point. In the event where one accidently deletes all records from a table, this can be undone using the point in time recovery features in PostgreSQL. However, this does require some modifications to the default setup of PostgreSQL so that WAL segments are retained for such a recovery. Please see the main PostgreSQL documentation for details.

This is different than the statement-based transaction log in MySQL being replayed in two important ways. The most important one is that the log is flushed to disk after every write. This allows for a shutdown to recover to disk exactly where things were before the unclean shutdown, assuming that the hardware does not lie about writes completing (as is common on some types of IDE drives). The second one is that the log is tuple-level rather than statement level and is thus able to provide point in time recovery information.

2.2.3 Domains and Check Constraints

Check constraints allow you to specify which conditions the database should check to ensure that the data is valid. For example, lets say you have a column specifying number of children an employee

has. This is likely to be an int(2) column in PostgreSQL. However, since PostgreSQL has no unsigned data types, and since it is not possible for an employee to have a negative number of children, you can specify that the column must be greater or equal to 0. Check constraints can also handle any other arbitrary requirement. For example, one can specify that a number minus 5 must be a multiple of 26, that a Varchar must either be "Chuck" or "Someone Else," that a date must be after the year 1990, or any other possible constraint one can think of.

Domains allow for centralized management of constraints on data input. For example, if I have several tables which have a "Month" column which is an ENUM in MySQL, I can use a domain in PostgreSQL. Furthermore, I can also use domains to check for other arbitrary characteristics of the data, such as its length, etc. It can also be used to emulate other data types.

Domains are actually quite a bit more flexible than any single mechanism they can emulate in MySQL. Basically they allow for a check constraint to be managed one place and then used many places. If the requirements change, the check constraint can be altered for all columns simultaneously.

2.2.4 User Defined Functions

PostgreSQL offers a number of options for user defined functions. New databases out of the box include the capacity to link to C modules or store user defined functions written in a basic form of SQL. One can also add a large number of languages for stored procedures including an Oracle-like PLPGSQL, Perl, PHP, Python, and TCL which are included with the basic distribution. Additional languages, such as Bash, Ruby, R, and Java can be installed after downloading the language handler from the project site, and it is not too difficult to write your own handler for your own language.

User-defined functions can add a lot of power to a database. They can be used in custom check constraints, triggers, and other things. If used in an "untrusted" mode, the languages can even access any other part of the system that the database process can access and, for example, send emails, instant messages, write log files, and more. Note that such external actions take place outside of transactional controls so it is probably safer to use something like the notify mechanism (see below) to let applications know to process new rows.

2.2.5 Notifications

PostgreSQL provides two commands: Listen and Notify which allow applications to notify other applications when a transaction commits which requires further action. This provides an alternative to adding such procedures directly into the database backend.

2.2.6 Triggers

Triggers provide a standards-compliant way to enforce referential integrity, and to require arbitrary actions when a row is changed in a database. PostgreSQL is not quite standards-compliant in this area because it requires that triggers call functions rather than simple SQL statements.

Triggers operate on the row level when a row is inserted, updated, or deleted. For more on this topic, see the online documentation.

Although PostgreSQL's CREATE TRIGGER syntax is nearly standards-compliant, the actual triggers are not as of 8.0. And there is a bit of a learning curve in writing such triggers.

Although MySQL 5.0 is expected to have some rudamentary trigger capability, it will likely be some time before it will be anywhere near as powerful as the capabilities in PostgreSQL. For example,

one cannot have two before triggers on the same table, etc. With PostgreSQL, one can have as many triggers as desired on any table.

2.2.7 Inherited Tables

PostgreSQL allows data to be partitioned into tables which extend the schema of a parent table. The entire table tree can then be queried by a single SELECT command (SELECT by default scans all child tables). This can be used for data partitioning, i.e. to keep historical data separate from current transactional data, or in a number of other ways. However, index scans are not inherited, so unique constraints are limited to a single table, not a whole tree of inherited tables. See the online documentation for more on this topic.

In order to limit select statements only to the current table and exclude all child tables, PostgreSQL supports a non-standard SELECT ... ONLY FROM syntax. See the online documentation for details.

2.3 Data Handling Differences

2.3.1 Dates and Time

MySQL will allow 0000-00-00 00:00:00 to be a valid date time. PostgreSQL will not. Instead, you must use NULL for dates and times.

Also, PostgreSQL will strictly check all dates and times to ensure their validity. Entering 2004-02-30 will result in an error: date/time field value out of range: ”2000-02-30.” Such values can be entered in MySQL without the database complaining.

If you have to use the MySQL types of dates, then having a constraint forcing a varchar field to fit a certain format would be necessary.

2.3.2 Numbers

MySQL will truncate numbers to fit in fields without so much as a warning. PostgreSQL is far more conservative and will refuse to insert a number if it won't fit. Even with an explicit cast, it will refuse to truncate. This behavior on PostgreSQL's part is standards-compliant.

Also PostgreSQL does not have any unsigned number types. If you need unsigned numbers, you will probably want to use a domain and the next higher size. I.e. bigint instead of int, and if you have lots of huge numbers, numeric instead of bigint.

MySQL allows integers to be set to autoincrement. PostgreSQL handles this differently by having special entities called ”sequences” that exist outside of transactions and are used to generate numbers in sequential order for the field. Sequences can be shared between tables a single linear sequence can number a set of related entites. Inherited tables (see above) use this mechanism internally.

See the online documentation for more.

2.3.3 Strings

Like numbers, MySQL will silently truncate strings if they are too long, while PostgreSQL will not. However unlike numbers, an explicit cast can be added to truncate a string (select 'abcde'::char2 will produce 'ab' as a result).

2.3.4 Enums

PostgreSQL has no ENUM type. Use a domain with a check constraint instead. If the options must be open (say colors for a part), you can use a foreign key pointing to a colors table.

2.3.5 Bools

MySQL treats bools as int1's. PostgreSQL treats them more like an enum(true, false). 1, 0, True, False, t, and f are all valid inputs.

2.3.6 Binary Data

Mysql includes a BLOB datatype which allows one to store large amounts of arbitrary binary data in a field in the database. PostgreSQL has two such interfaces, each has specific advantages and issues to consider.

PostgreSQL includes a ByteA datatype where arbitrary data is saved in the database as a field in the table. It is presented to the client as a text string with embedded escaped octals. Therefore to use this type, one must first unescape it (functions for doing this are provided by libpq).

Secondly, PostgreSQL includes a Large Object (lob) interface where arbitrary data can be stored in separate files linked to in the database. lobs can be chunked out to the client in any way the client requires, which can help free up memory for large operations. However, lobs have not only an odd interface, but they also are not backed up by a textual dump of the database. One must use binary dumps instead.

2.4 SQL Syntax Differences

There are several important areas where MySQL and PostgreSQL differ in SQL syntax. MySQL is not as standards-compliant as PostgreSQL, and both support a different set of extensions to the ISO standard.

2.4.1 The || Operator

The ANSI SQL and ISO SQL-92 and SQL-99 standards specify that the || operator will be used to concatenate fields. MySQL by default treates this as a logical or operator and relies on the concat() function instead. MySQL does treat this in the standard way in ANSI mode, however, but many applications expect that this will not be enabled. Therefore this is one of the major porting headaches when moving from MySQL to PostgreSQL.

2.4.2 Case sensitivity

In MySQL string comparisons are case insensitive and table names are case sensitive. Tables names are returned in their original case. In PostgreSQL, string comparisons are (as specified in the standards) case sensitive, but table naming is not. Table names are assumed to be lower case unless quoted.

Thus SELECT * FROM MyTable will select all columns from ”mytable” but SELECT * FROM ”MyTable” will select all columns from ”MyTable.” This difference is the source for many headaches.

We suggest you rename all tables to lower case tablenames before migrating your applications as this will make the process much easier.

2.4.3 Quoting Data

MySQL makes no distinction between single and double quotes. In PostgreSQL, these are given their standard meanings: single quotes identify data, double quotes identify areas where the reserved meanings of words should be ignored and case preserved.

So SELECT NOW FROM mytable is different from SELECT "NOW" FROM mytable and SELECT 'NOW' FROM mytable is different still. The first statement will select from a column named now (lower case) while the second will select from a column named NOW (upper case). The third will fill a column in the output with NOW.

To escape single quotes, the standard way to do it is to use two of them. So SELECT "'now"' will produce a column filled with 'now' values.

2.4.4 Obtaining Last Insert Info

In MySQL, one generally uses the last_insert_id() function to determine the id of the autoincremented integer primary key that was inserted. PostgreSQL has no equivalent function. Instead one must use currval('sequence_name') or pg_get_serial_sequence(table, column). This generally requires that the application must know the sequence id number or at least the table and column used in the query. This is one of the larger issues in migrating from MySQL to PostgreSQL.

2.5 Maintenance Differences

MySQL doesn't require frequent maintenance. However because of MVCC and the fact that dead versions of a row remain in the database, PostgreSQL databases must be vacuumed from time to time. One can use the externam program pg_autovacuum or schedule vacuum commands via Cron. Also, it is important to give the planner reasonable estimates for row counts in tables, etc. so usually one also analyzes the tabes at the same time. This is done via the VACUUM ANALYZE command or the vacuumdb -z 'database_name' from the shell or Cron.

2.6 Replication Differences

The list of PostgreSQL replication options is by no means exhaustive. Replication is one area where different solutions are better in different environments, and one would do well to seriously investigate all options that seem promising.

2.6.1 MySQL Replication

MySQL provides a single replication function which replicates SQL statements on the statement level. There are a number of serious limitations with this approach particularly the fact that one cannot be sure that the same database tuples are saved on all the replicas.

2.6.2 Slony-I

Slony-I is an open source replication solution for PostgreSQL sponsored in part by Affilias, Inc. It provides a fairly robust and powerful asynchronous replication solution for master/slave setups. It currently does not support Windows as it requires pthreads.

2.6.3 DBMirror

DBMirror is a simple asynchronous replication system for PostgreSQL (distributed with the product) which provides a simple ability to mirror the database over any connection, even if that connection is slow and/or unreliable. It is also master/slave.

2.6.4 PGPool

PGPool is not really a replication solution, but it allows one to offer better load-balancing solutions in areas where mixed read and write access are needed. I.e. one can switch to the master server transparently at the start of transaction blocks or for insert/update/delete statements.

2.6.5 Mammoth Replicator

Command Prompt Inc. maintains a product called Mammoth Replicator which provides an asynchronous replication solution. It is somewhat similar to Slony, but runs on Windows and is neither Free nor Open Source.

3 Function Porting Reference

3.1 Operators

Most operators are the same. However in some cases, there may be differences.

In general if an operator does not exist in PostgreSQL it may be possible to create it fairly easily. See more in the PostgreSQL manual for this topic.

3.1.1 String to Number conversion

PostgreSQL only converts ints and floating points from strings if the numbers are represented properly. Also one should must watch single quotes as they indicate data of an unknown type.

SELECT 1 >'0.13';
ERROR: invalid input syntax for integer: "0.13"

This fails because the parser sees 1 as an integer and tries to convert '0.13' into the same type.

SELECT '1' >'0.13';
?column?
————-
t
(1 row)

In this case, both numbers are of unknown types and are converted to numbers before the statement is run.

SELECT 1 >0.13;
?column?
————-
t
(1 row)

In this case the comparison is between two floating point numbers (the int is converted into a float before the conversion takes place).

3.1.2 Comparisons: GREATEST, LEAST, INTERVAL, and <=>

The upcoming 8.1 release of PostgreSQL will include GRATEST and LEAST functions.

There is no "GREATEST" function in PostgreSQL as of 8.0.x. PostgreSQL does not support variable number of arguments in function definitions, so you would have to define something like:

CREATE FUNCTION GREATEST(numeric, numeric, numeric)
RETURNS numeric AS ' SELECT CASE WHEN ($1 >$2) AND ($1 >$3)
THEN $1
WHEN ($2 >$1) AND ($2 >$3)
THEN $2
WHEN ($3 >$1) AND ($3 >$2)
THEN $3
END
' LANGUAGE SQL;

Or you can use a similar query in place of GREATEST().

Similar, the LEAST and INTERVAL functions will require either a change in coding or the creation of such a wrapper function. Also note that "interval" is a reserved word in SQL and a non-standard extension for MySQL.

<=>

In MySQL you might write:

SELECT a <=>b;

This is similar to:

SELECT a IS NULL AND b IS NULL OR a = b;

One can wrap this in a custom operator in PostgreSQL fairly easily. See the online documentation for details.

3.1.3 Bitwise Operators

In MySQL the bitwise XOR operator is ŵhile in PostgreSQL the same symbol is used to express exponential operations (i.e. 3 $\hat{2}$ = 8). The bitwise XOR operator in PostgreSQL is #.

3.1.4 ISNULL

In MySQL, you might write:

SELECT ISNULL(a >b);

The standard way to write this is:

SELECT (a >b) IS NULL;

3.1.5 Logical Operators

PostgreSQL sticks with the standards-defined logical operators of AND, OR, and NOT. It does not recognize && in this way. Also note that ! and || are used in other ways in other contexts so it is not advisable to overload these operators (though it is certainly possible). ! is used as a factorial operator (4! = 24), and || is for concatenating strings ('Hi' || ' ' || 'There' = 'Hi There'). Overloading these operators may cause some bugs which may be difficult to track down in your applications.

3.2 Control Flow Functions

IF and IFNULL are not standard SQL constructs. Use CASE instead. If your application makes extensive use of IF or IFNULL, you can create a wrapper function.

3.3 String Functions

BIN() is not supported in PostgreSQL. Writing a wrapper function should be trivial.

CHAR() does not function as it does in MySQL. In PostgreSQL, it tried to define a CHAR(n) datatype. Use CHR() instead. This only accepts one argument, so you may have to change char(a, b, c) to chr(a) || chr(b) || chr(c). This could be wrapped in a function if necessary.

COMPRESS() does not exist in PostgreSQL. Adding it would be somewhat difficult, but could be done with anyone with a reasonable knowledge of C. The same goes for UNCOMPRESS.

CONCAT does not exist in PostgreSQL. Use the || operator instead.

CONV does not exist. One would need to create a function in PostgreSQL.

ELT does not exist in PostgreSQL. Move such code onto the client side.

EXPORT_SET does not exist in PostgreSQL.

FIELD and FIELD_IN_SET do not exist in PostgreSQL. One could wrap specific uses of this into a PL/Perl function without too much trouble.

HEX is not recognized. Use TO_HEX instead. If you are heavily dependent on this function, you can write a wrapper.

INSERT is not recognized. Instead use substring and the || operator to accomplish the same tasks.

INSTR does not exist in PostgreSQL. Use position() instead. A wrapper function is possible in this case.

LCASE does not exist. You can create a wrapper for lower() if you need it. Same with UCASE.

LEFT and RIGHT don't exist. These are reserved words, so you could create

LENGTH returns the same as char_length. If you need to use this functionality, use bit_length()/8 instead.

LOCATE does not exist but is similar to position. One could write a wrapper function for both forms.

MAKE_SET does not exist. Use the || operator instead and if you need to you can wrap specific uses.

OCT does not exist in PostgreSQL. You would have to write a function to do this.

OCTET_LENGTH does not exist. You could use bit_length()/8 instead or write a wrapper function.

ORD does not exist. Would take some work to replace.

Quote does not exist. use QUOTE_LITERAL instead.

REVERSE does not exist. One would need to write a wrapper function instead.

SOUNDEX does not exist. One would have to create the function from scratch.

SOUNDS LIKE does not exist. Look for other searching packages like tsearch2.
SPACE does not exist. Use repeat instead.
SUBSTRING_INDEX does not exist. A wrapper function should be possible.
UNHEX does not exist. Write a wrapper function instead.

3.4 Numeric Functions

ATAN(Y,X) is called ATAN2 in PostgreSQL. It should be trivial to create a wrapper function.
CRC32() does not exist in PostgreSQL. Creating a function in C should not be that difficult.
LOG() in PostgreSQL defaults to base 10 instead of base e.
LOG10() does not exist in PostgreSQL. Use log(n) instead.
LOG2() does not exist in PostgreSQL. Use log(n, 2) instead.
RAND() is called RANDOM() instead in PostgreSQL.
TRUNCATE() is called TRUNC() in PostgreSQL.

3.5 Date and Time Function

ADDDATE/DATE_ADD don't exist in PostgreSQL. Use numeric + operator instead.

ADDTIME doesn't exist. This is equivalent to DATETIME + (TIME - '00:00:00'). A wrapper function should be easy to write.

CONVERT_TZ doesn't exist in PostgreSQL. It is equivalent to (DATETIME AT TIMEZONE tz1) AT TIME ZONE tz2. A wrapper function should be pretty trivial to write.

CURDATE() is called CURRENT_DATE in PostgreSQL (no parentheses). No reason why a wrapper funciton would be hard to write.

CURRENT_DATE does not work with parenteses in PostgreSQL. A wrapper function is not possible to write because of parser issues.

CURTIME() does not exist in PostgreSQL. Use Cthe CURRENT_TIME variable instead.

CURRENT_TIME doesn't work with parentheses.

CURRENT_TIMRSTAMP doesn't work with parentheses.

DATEDIFF() does not exist. It is equivalent to EXTRACT('DATE' FROM DATE1 - DATE2). A wrapper function would be trivial to write.

DATE_FORMAT dfoes not exist in PostgreSQL. One might be able to write a wrapper function in PL/Perl.

DAY() and DAYOFMONTH() are equivalent to EXTRACT(DAY FROM TIMESTAMP). Wrapper function should be trivial to write.

DAYOFWEEK() is equivalent to EXTRACT(DOW FROM TIMESTAMP).

DAYOFYEAR() is equivalent to EXTRACT(DOY FROM TIMESTAMP).

Note that extract is defined in the SQL Standard.

FROM_DAYS does not exist in PostgreSQL. A wrapper function might have to be written.

FROM_UNIXTIME does not exist. One can simulate it using SELECT TIMESTAMP WITH TIME ZONE unixtime + 982384720 * INTERVAL '1 second'; A wrapper function would be easy to write. If a format string was required, this would probably require PL/Perl.

GET_FORMAT doesn't exist. One could write a function to do this.

HOUR is equivalent to EXTRACT(HOUR FROM TIMESTAMP)

LAST_DAY does not exist. Something could be done like EXTRACT(DAY FROM (TIMESTAMP + INTERVAL '1 month' - (EXTRACT (DAY FROM (TIMESTAMP + INTERVAL '1 month')) + 1)) and a wrapper function created.

LOCALTIME and LOCALTIMESTAMP don't work with parentheses.

MAKEDATE is equivalent to (YEAR || '-01-01')::date + DOY. A function could easily be written to do this.

MAKETIME is equivalent to '00:00:00'::time + (HOUR || ' hours ' || MINS || ' minutes ' || SECS || ' seconds')::interval. A wrapper function could be used.

MICROSECOND is equivalent to EXTRACT(microsecond FROM TIMESTAMP).

MONTH is equivalent to EXTRACT(MONTH FROM TIMESTAMP).

MONTHNAME does not exist. One would have to add a wrapper function if you depend on this.

PERIODADD and PERIODDIFF do not exist in PostgreSQL and the syntax may be problematic. Probably one would want to create a wrapper function using interval operations.

QUARTER is equivalent to EXTRACT(QUARTER FROM TIMESTAMP).

SEC_TO_TIME is equivalent to '00:00:00'::time + (SEC || ' seconds')::interval

STR_TO_DATE does not exist and might need to be reimplimented in PL/Perl.

SUBDATE is equivalent to DATE - INTERVAL or DATE - INTEGER. A wrapper function could be written.

SUBTIME is equivalent to DATETIME - TIME::interval.

SYSDATE() does not exist. Use NOW() or a wrapper function.

TIME() is equivalent to DATETIME::TIME. You could write a wrapper function easily enough.

TIMEDIFF is equivalent to simple arithmatic operations (TIME1 - TIME2).

TIMESTAMP and TIMESTAMPADD are equivalent to DATE::TIMESTAMP or DATETIME + INTERVAL. Adding one date to another is probably not that meaningful anyway. If you have need fro DATE + DATE, you can add a custom operator,

TIMESTAMPDIFF is equivalent to EXTRACT(field FROM (timestamp1 - timestamp2))

TIME_FORMAT does not exist and might have to be reimplimented in PL/PERL.

TO_DAYS might have to be reimplimented.

UNIX_TIMESTAMP is equivalent to EXTRACT(EPOCH FROM timestamp).

UTC_TIME is equivalent to TIME::time with time zone AT TIME ZONE 'UTC'

UTC_TIMESTAMP is equivalent to TIMESTAMP AT TIME ZONE 'UTC'

WEEK is equivalent to EXTRACT(WEEK FROM TIMESTAMP)

YEARWEEK is equivalent to EXTRACT(YEAR FROM TIMESTAMP) || EXTRACT WEEK FROM TIMESTAMP

WEEKDAY is equivalent to (extract(DOW FROM '1998-02-03 22:23:00'::timestamp) + 6)::integer % 7

WEEKOFYEAR is equivalent to EXTRACT(WEEK FROM TIMESTAMP)

YEAR() is equivalent to extract(YEAR FROM TIMESTAMP)

3.6 Full Text Search

PostgreSQL has no full text search included by default. Instead there are several add-on packages such as tsearch2.

3.7 Encryption Functions

MD5() is the only encryption function provided by PostgreSQL by default. For more encryption functions, please see the pg_crypto extension.

The upcoming release of PostgreSQL 8.1 will include functions to generate SHA* hashes.

3.8 Information Functions

Most information functions are not the same on PostgreSQL. Many of these functions are unnecessary in PostgreSQL for the most part and are designed to be debugging aids.

BENCHMARK may give some similar hits as to performance as EXPLAIN ANALYZE.

CHARSET() is not supported by PostgreSQL.

COERCIBILITY() does not exist.

COLLATION() does not exist.

CONNECTION_ID does not exist. Version 8.1 might have some similar information available in terms of host and port info.

CURRENT_USER() is similar to the CURRENT_USER and SESSION_USER variables in PostgreSQL.

DATABASE() is equivalent to CURRENT_DATABASE()

FOUND_ROWS() is not an SQL command but may be accessed via client libraries.

LAST_INSERT_ID() does not exist. One must use CURRVAL('sequence_name') instead. Alternatively, one could use PG_GET_SERIAL_SEQUENCE(table, column) to get the same information.

ROW_COUNT() is not accessible as an SQL command but may be accessible via client libraries.

4 Migration Tools

4.1 Database Conversion Scripts

The primary tool used for this is mysql2pgsql.pl.

4.2 Porting Frameworks

Porting frameworks may depend on the language your application is written in. In many languages, the database access functions are sufficiently abstracted to make porting fairly simple.

For PHP, Metatron Technology Consulting offers a simple porting toolkit which can be used along with sed to solve most porting issues.

4.3 Abstraction Layers

In genreal, it is a good idea to use an abstraction layer when writing an application to start with. Generic (language-independent) abstraction layers include ODBC, for example.

4.4 Misc. Tools

4.4.1 DBI-Link

DBI-Link is a partial implimentation of SQL/MED (Management of External Data). It allows one to access any data source reachable via Perl and DBI as if it were a PostgreSQL table. This could be used, for example, to access MySQL data from a PostgreSQL system, etc.

DBI-Link can be found at http://pgfoundry.org/projects/dbi-link/.

5 Common Problems

5.1 Common Error Messages

5.1.1 relation "tablename" does not exist

This means that the table could not be found as named. PostgreSQL treats all tablenames as lower case unless quoted. Try renaming the table to lower case.

5.1.2 integer out of range

MySQL will try to truncate your integer. PostgreSQL will refuse to write it to the database. Alter the type of the column of check what your application is doing.

5.1.3 value too long for type character varying(n)

MySQL will truncate strings by default. If you want PostgreSQL to truncate the strings, cast it as a string of a specific length.

5.1.4 numeric field overflow

MySQL will truncate your numeric field to fit, while PostgreSQL refuses to proceed. Increase the size of the field or check to see what your application is trying to insert.

6 Service Offerings

Metatron Technology Consulting provides a full range of services to help migrate your application to PostgreSQL. We can help with planning, testing, and migration. Please email sales@metatrontech.com for a quote.