Migrating from Microsoft SQL Server to MySQL

A MySQL® White Paper

Table of Contents

Executive Summary

While MySQL is famously known as the "M" of the popular LAMP stack (Linux, Apache, MySQL, PHP/Perl/Python), Microsoft Windows consistently ranks as the #1 development platform for MySQL users in our surveys. As a matter of fact, Windows also ranks higher than any Linux distribution as a deployment platform among the MySQL Community Edition users. Given that so many users deploy MySQL on Windows for production, it makes sense to explore the business and technical reasons for migrating from SQL Server to MySQL.

Of course, database migrations are not something to be taken lightly, and the majority of Windows-based MySQL applications are new applications; this being the case, many organizations are migrating from Microsoft SQL Server because they have reached the conclusion that the combination of cost-savings, platform freedom, and feature set of MySQL make for a compelling business case. This paper provides insight into what is needed for considering a move from SQL Server to MySQL and presents a number of options that help make the transition easier. Both the business and technical sides of migrating to MySQL will be dealt with, so whether you are a manager or a seasoned DBA, you will find many of the answers to questions that revolve around migrating to the world's most popular open source database.

.

Migrating from SQL Server makes Business Sense

Before committing to a technology, enterprises typically consider both the business and technical advantages of the given technology. While each organization follows its own methodology, the core set of factors that normally govern acceptance are calculating the Total Cost of Ownership (TCO) and validating the viability of the software vendor.

MySQL: The World's Most Popular Open Source Database

MySQL is the world's most popular open source database. With its proven ease-of-use, performance, reliability and scalability, MySQL has become the leading database choice for Web-based applications, used by high profile web properties including Facebook, Twitter, YouTube, Yahoo!, Wikipedia and thousands of mid-sized companies. MySQL is now part of Oracle Corporation, and backed by Oracle's world class support organization worldwide.

Figure 1: Industry Leaders Rely on MySQL

Measuring the Costs of using SQL Server vs. MySQL

In MySQL's annual surveys, lower cost is consistently shown to be the number one reason why users choose MySQL.

Microsoft changed the SQL Server pricing from Per Processor to Per Core, resulting in a major price increase for customers who want to take advantage of the latest multi-core hardware.
Below is a chart which compares the 3 year database TCO of MySQL Enterprise Edition vs. Microsoft SQL Server 2014 Enterprise Edition. In this configuration, Microsoft has doubled the license cost of SQL Server 2014 over SQL Server 2008.

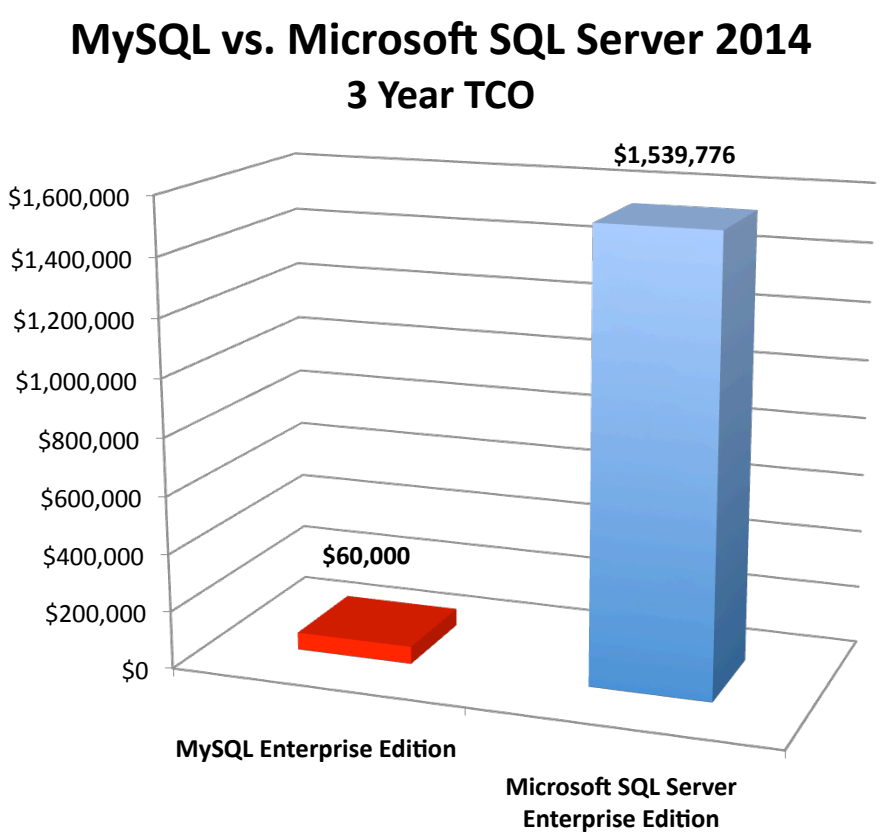

Figure 2: 3 Year TCO vs Microsoft SQL Server 2014

Hardware Configuration:
Intel x86_64 Servers: 4, CPUs/Server: 4, Cores/CPU: 8

Microsoft Price Increases
Microsoft increased prices of Microsoft SQL Server for customers who want to take advantage of the latest multi-core hardware. Microsoft SQL Server 2014 is licensed Per Core.

Microsoft SQL Server vs. MySQL: 3 Year Database TCO
The table below presents a simple comparison of licensing costs between MySQL Enterprise Edition and Microsoft SQL Server Enterprise Edition. Pricing is based on publicly available list price information:
Microsoft
http://www.microsoft.com/en-us/server-cloud/products/sql-server/
MySQL Enterprise Edition:
http://www.mysql.com/products/
Hardware Configuration:
- Web application (Unlimited Users)
- Windows
- Intel x86_64
- CPUs/Server: 4
- Cores/CPU: 8
- Total Servers: 4
- Total CPUs: 16
- Total Core: 128

List Prices	Microsoft SQL Server Enterprise Edition License + 3 Year Support	MySQL Enterprise Edition 3 Year Subscription
License List Price	$6,874/Core	N/A
Support List Price	25% of List Price	N/A
Subscription List Price	N/A	$5,000/Server/Year

Annual Savings	SQL Server Enterprise Edition	MySQL Enterprise Edition
Annual Support	$219,968	N/A
Annual Subscriptions	N/A	$20,000
Annual Savings ($)		$199,968
Annual Savings (%)		91%

3 Year TCO Savings	SQL Server Enterprise Edition License + 3 Year support	MySQL Enterprise Edition 3 Year Subscription
Total License:	$879,872	N/A
Total Support (3 Yrs):	$659,904	N/A
Total Subscription (3 Yrs)	N/A	$60,000
Total (3 Years)	$1,539,766	$60,000
Total Savings ($)		$1,479,766
Total Savings (%)		96%

Migrating from SQL Server makes Technical Sense

The next consideration with respect to migrating to MySQL from SQL Server is the technical review of the database feature set and capabilities.

Unlike Microsoft SQL Server, MySQL supports all major operating systems and platform combinations including Linux, Windows, Mac OS, Solaris and many more. MySQL users are not locked-in to a single operating system or platform and have the flexibility to deploy MySQL in a heterogeneous computing environment.

In addition, ease of use has been a design goal for MySQL since its inception. MySQL offers exceptional quick-start capability with an average time from software download to installation completion of less than

fifteen minutes. To make it even easier for Windows users to get going with MySQL, Oracle provides a unified, point and click MySQL Installer for Windows. This MySQL Installer radically simplifies the installation, update and configuration process for all MySQL users on the Windows platform. With the MySQL Installer, it takes only 3 minutes from downloading the MySQL database and supporting products to having a ready to use MySQL system up and running in your environment!

Once installed, self-management features like automatic space expansion, auto-restart, and dynamic configuration changes take much of the burden off already overworked database administrators. MySQL also provides visual database design, development, administration and monitoring tools that make Windows database developers and DBAs feel at home.

MySQL Features

MySQL was designed with a focus on high performance, reliability and ease of use, and continues to evolve along those lines. In terms of core features, DBAs and CIO's alike will be pleased to find that MySQL contains all the necessary capabilities to run many of their application needs, especially their web, custom enterprise, and embedded database applications:

Database Feature	Available in MySQL
Open Source	√
Available on all major platforms (32 and 64 bit) including Windows: (Oracle Linux, RedHat, SuSE, Fedora, Solaris, , FreeBSD, Mac OS, Windows)	√
Pluggable Storage Engine Architecture (InnoDB, MyISAM, Merge, Memory, Archive, Cluster)	√
High-Availability Clustered Database	√
ANSI SQL, SubQueries, Joins, Cursors, Prepared Statements	√
Stored Procedures, Triggers, SQL and User-Defined Functions	√
Updateable Views	√
ACID Transactions with Commit, Rollback	√
Distributed Transactions	√
Row-level Locking	√
Snapshot/Consistent Repeatable Reads (readers don't block writers and vice-versa)	√
Server-enforced Referential Integrity	√
Strong Data type support (Numeric, VARCHAR, BLOB, etc)	√
High-Precision Numeric Data types	√
Robust Indexing (clustered, b-tree, hash, full-text)	√
Dynamic Memory Caches	√
Unique Query Cache	√

Cost-Based Optimizer	√
Unicode, UTF-8	√
XML, XPath	√
Geospatial support	√
Replication (Row-based and Statement-based)	√
Hi gh Availability with auto failover, promotion of slaves to master	√
Partitioning (Range, List, Hash, Key, Composite)	√
VLDB (terabytes) capable	√
High-speed, parallel data load utility	√
Online Backup with Point-in-Time Recovery	√
Automatic Restart/Crash Recovery	√
Automatic Storage Management (auto-expansion, undo management)	√
Compressed and Archive Tables	√
Information Schema/Data Dictionary	√
Robust Security (SSL, fine grained object privileges)	√
Built-in data encryption and decryption	√
Built-in Job/Task Scheduler	√
Drivers (ODBC, JDBC, .NET, PHP, etc)	√
GUI Tools (Data Modeling, Administration, SQL Development, Migration)	√

Figure 3: MySQL Features

As shown above, MySQL contains a very strong feature set, and provides functionality that supports most applications that have been traditionally targeted for deployment on SQL Server.

MySQL Features Compared to SQL Server Express

Several years ago, Microsoft introduced SQL Server Express. The move was seen by some industry watchers as an attempt to take away some of the momentum of Open Source databases such as MySQL.

Microsoft SQL Server Express 2008 has a number of feature, support, and performance limitations you won't find with MySQL:

- Limited to 1 CPU socket whereas MySQL has the capability to scale efficiently on 16-way and 64-way systems depending on the chip design.
- Can only address up to 1 GB of RAM. MySQL imposes no such limitation and instead works within the capabilities of the operating system.
- Imposes a limit of 10 GB of user data per database. MySQL imposes no such limitation and can scale to support multi-terabyte configurations.
- Does not include support for Microsoft's SQL Profiler tool, which helps in locating problem SQL queries. The Community edition of MySQL includes the general and slow query logs, which can

capture either all or only 'slow' SQL, and MySQL Enterprise Edition includes the MySQL Enterprise Monitor with Query Analyzer that locates all problem SQL code across all monitored servers.

- Does not support the SQL Server Agent. MySQL natively supports an Event Scheduler as of version 5.1 and above.
- Offers replication only as a subscriber. MySQL supports both master and slave configurations straight out of the box at no additional cost or with any limitations.
- Does not support table and index partitioning. MySQL 5.1 and above supports a variety of data partitioning options.

MySQL Enterprise Edition for Production Deployments

For business critical applications that require the highest levels of security, performance and availability, backed by 24x7x365 technical support Oracle provides MySQL Enterprise Edition. MySQL Enterprise Edition includes the most comprehensive set of advanced features, management tools and technical support to help you achieve the highest levels of MySQL scalability, reliability, security and uptime. MySQL Enterprise Edition reduces the risk, cost and time required in developing, deploying and managing business-critical MySQL applications.

In addition to the MySQL Database, MySQL Enterprise includes:

The MySQL Enterprise Monitor:

The MySQL Enterprise Monitor provides at-a-glance views of the health of your MySQL databases. It continuously monitors your MySQL servers and alerts you to potential problems before they impact your system. It's like having a "virtual DBA" assistant at your side to recommend best practices and eliminate security vulnerabilities, improve replication, and optimize performance. As a result, DBAs and system administrators can manage more servers in less time.

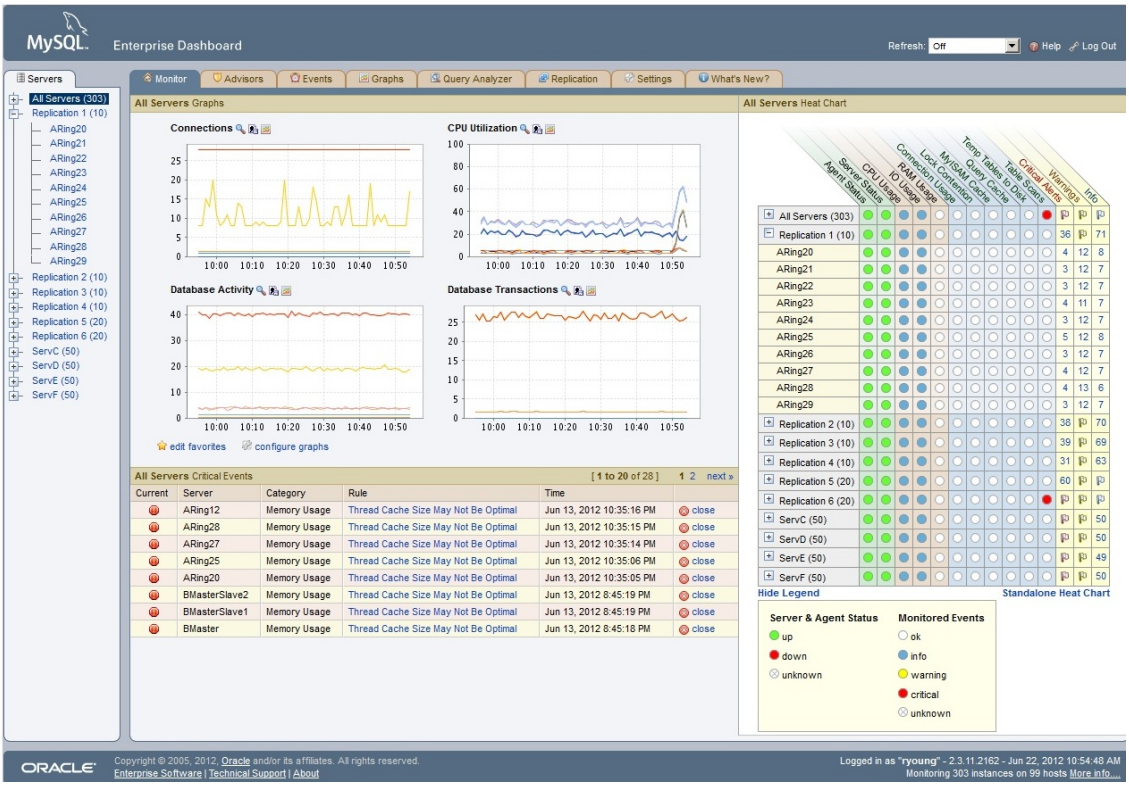

Figure 4: MySQL Enterprise Monitor

MySQL Replication Monitor

The MySQL Replication Monitor auto detects and monitors replication topologies and provides real-time replication health and status checks on master/slave performance and latency issues.

MySQL Advisors

The MySQL Enterprise Monitor includes more than 160 MySQL advisor rules and graphs that monitor more than 600 MySQL and operating system specific variables to help you automatically enforce MySQL recommended database best practices. The MySQL Advisors deliver expert advice that come straight from the database professionals who build the MySQL database and provide tailored step-by-step instructions to solve specific problems or to act on performance tuning opportunities. The MySQL Enterprise Monitor includes the following Advisors:

- **MySQL Enterprise Backup Advisor** – Monitors and advises on backup operations and advises on servers in need of backup.
- **Upgrade Advisor -** Monitors and advises on using the most up-to-date version of MySQL.
- **Administration Advisor -** Enforces DBA best practices to prevent costly outages.
- **Security Advisor** - Minimizes exposure to security vulnerabilities.
- **Replication Advisor** - Recommends solutions for replication setup and performance improvements.
- **Schema Advisor** - Assists in uncovering database design issues that reduce performance.
- **Performance Advisor** - Recommends changes to boost database performance.
- **Memory Usage Advisor** - Monitors dynamic memory-related server metrics and recommends configuration changes to improve performance.
- **MySQL Cluster Advisor** – Monitors performance, availability, memory usage thresholds of MySQL Cluster data nodes.

- **Custom Advisor -** Enables you to create custom best practice rules tailored to your needs.

MySQL Query Analyzer

The MySQL Query Analyzer helps developers and DBAs improve application performance by monitoring queries and accurately pinpointing SQL code that is causing a slow down. With the new MySQL Connector Plug-ins, performance of Java, Microsoft .NET, and PHP applications can be optimized more efficiently by communicating directly with the MySQL Query Analyzer.

Queries are presented in an aggregated view across all MySQL servers so DBAs and developers can filter for specific query problems and analyze their most expensive code. With the MySQL Query Analyzer, DBAs can improve the SQL code during active development and continuously monitor and tune the queries in production.

MySQL Workbench

MySQL Workbench is a unified visual tool that enables developers, DBAs, and data architects to design, develop and administer MySQL servers. MySQL Workbench provides advanced data modeling, a flexible SQL editor, and comprehensive administrative tools.

MySQL Workbench: Migration from SQL Server to MySQL made easy

MySQL Workbench also provides a complete, easy to use solution for migrating Microsoft SQL Server database tables, objects and data to MySQL. Developers and DBAs can quickly and easily convert existing applications to run on MySQL on Windows, or any of the other operating systems supported by MySQL.

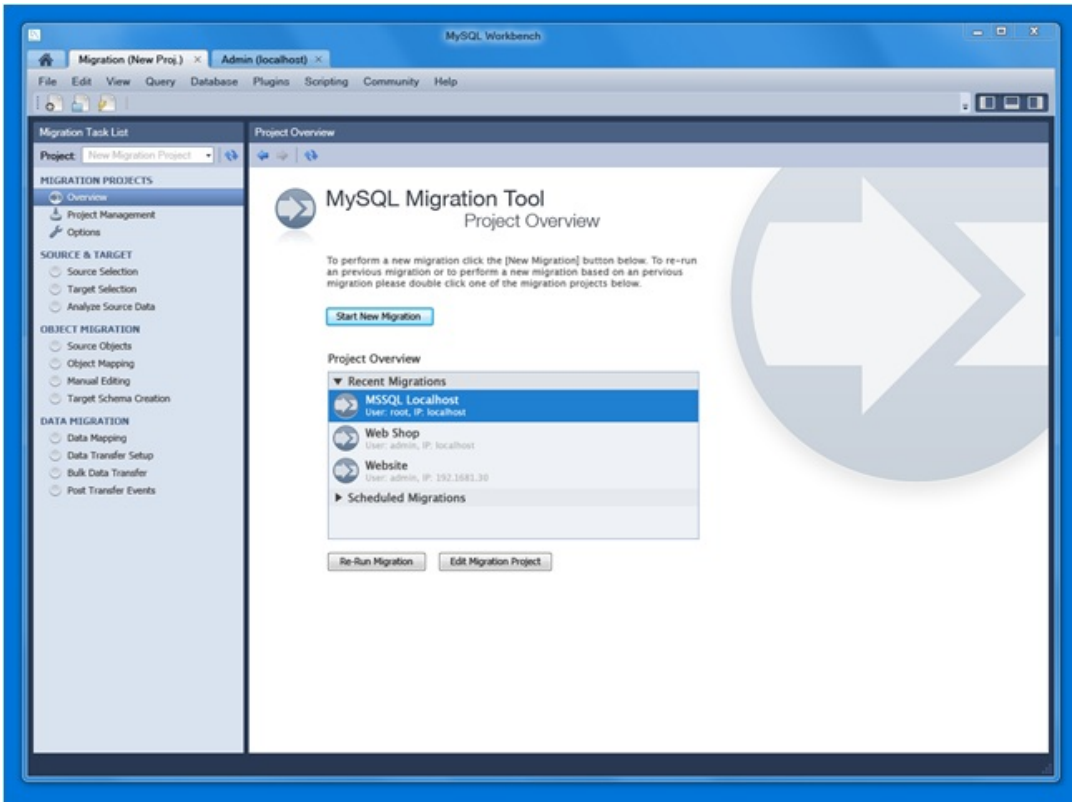

Figure 5: MySQL Workbench Migration Tool

The MySQL Workbench Migration Tool is designed to save DBA and developer time by providing visual, point and click ease of use around all phases of configuring and managing the SQL Server to MySQL migration process:

- Migration project management - allows migrations to be configured, copied, edited, executed and scheduled.
- Source and Target selection – allows users to define specific data sources and to analyze source data in advance of the migration.
- Object migration – allows users to select objects to migrate, assign source to target mappings where needed, edit migration scripts and create the target schema.
- Data migration – allows users to map source and target data and data types, set up data transfer and assign post data transfer events where needed.

With the MySQL Workbench Migration Tool users can convert an existing SQL Server database to MySQL in minutes rather than hours or days that the same migration would require using traditional, manual methods.

MySQL Enterprise Backup

MySQL Enterprise Backup performs online "Hot" backups of your MySQL databases. You get a consistent backup copy of your database to recover your data to a precise point in time. In addition, MySQL Enterprise Backup supports creating compressed backup files, and performing backups of subsets of InnoDB tables. Compression typically reduces backup size up to 90% when compared with the size of actual database files, helping to reduce storage costs. In conjunction with the MySQL binlog, users can perform point in time recovery.

MySQL Enterprise Scalability

MySQL Enterprise Scalability enables you to meet the sustained performance and scalability requirements of ever increasing user, query and data loads. The MySQL Thread Pool provides an efficient, thread-handling model designed to reduce overhead in managing client connections, and statement execution threads.

MySQL Enterprise Security

MySQL Enterprise Security provides ready to use external authentication modules to easily integrate MySQL with existing security infrastructures including PAM and Windows Active Directory. MySQL users can be authenticated using Pluggable Authentication Modules ("PAM") or native Windows OS services.

MySQL Enterprise High Availability

MySQL Enterprise High Availability enables you to make your database infrastructure highly available. MySQL provides you with certified and supported solutions, including MySQL Replication, Oracle VM Templates for MySQL and Windows Failover Clustering for MySQL.

Oracle Premier Support for MySQL

MySQL Enterprise Edition provides 24x7x365 access to Oracle's MySQL Support team, staffed by seasoned database experts ready to help with the most complex technical issues, and backed by the MySQL developers. Oracle's Premier support for MySQL provides you with:

- 24x7x365 phone and online support
- Rapid diagnosis and solution to complex issues

- Unlimited incidents
- Emergency hot fix builds forward compatible with future MySQL releases
- Access to Oracle's MySQL Knowledge Base
- Consultative support services
- The ability to get MySQL support in 29 languages

Migrating from SQL Server to MySQL Fundamentals

The figure below outlines the three basic steps to migrate from SQL Server to MySQL:

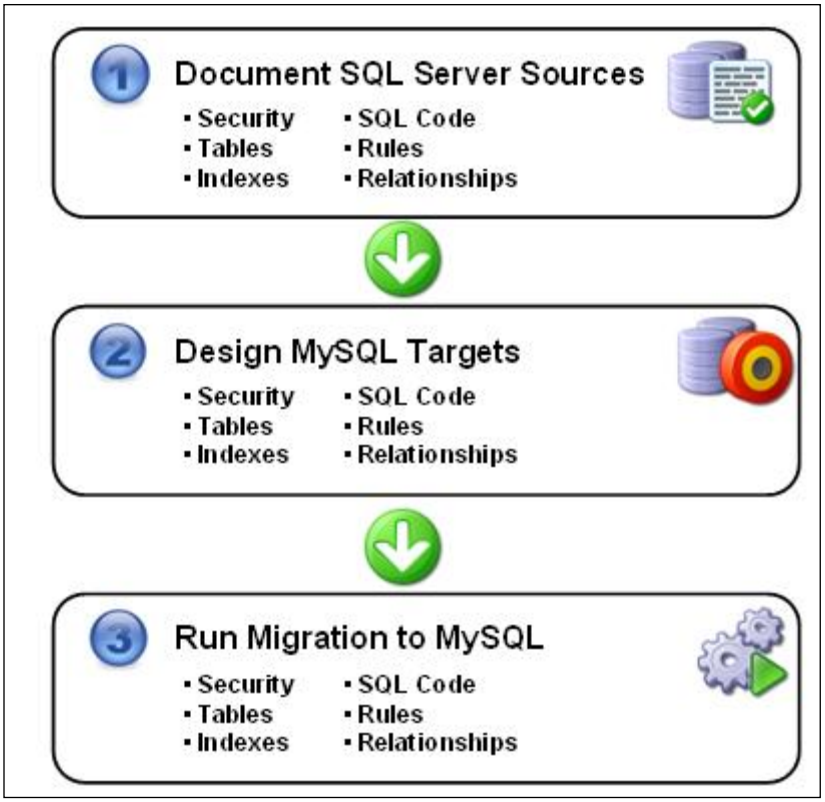

Figure 6: Three Basic Steps for Migrating from SQL Server to MySQL

All or Nothing?

Before working through each migration step, it is helpful to first point out that one very successful MySQL deployment strategy that customers have used is to move forward with MySQL for new development or migrating existing applications where the need for esoteric SQL Server features is not necessary, and use SQL Server when very specialized database needs are present. As previously stated, MySQL offers the perfect balance of ease-of-use and a strong enterprise feature set for handling many database applications requirements. Further, MySQL Workbench also enables easy schema and data migration from SQL Server to MySQL, but many or most of the steps below should be followed as there are limitations that must be handled outside of the Workbench migration capabilities.

Step 1: Document SQL Server Sources

Documenting an existing SQL Server database structure can be challenging if it is attempted by standard manual processes. While SQL Server has a very good metadata dictionary to work with, the process of manually pulling all metadata (table, column, index, etc.) can be very time and labor intensive.
The best approach is to use a computerized reverse engineering process that automatically catalogs all applicable metadata that is needed for conversion. A good third party data modeling tool can be used for this, and there are a number of good products on the market, such as Sybase's/Quest's PowerDesigner and Embarcadero's ER/Studio, that support the reverse engineering of multiple datasources such as SQL Server.

Moving data and index structures over to MySQL isn't typically a challenging task as MySQL supports all the important datatypes, table designs, and index structures. For a complete description of datatype comparisons and other like things in SQL Server and MySQL, please see Appendix A-C of this paper, and see Appendix D for a step-by-step example migration of an SQL Server schema to MySQL.

Outside of code-related objects such as stored procedures, a few SQL Server objects that cannot be migrated in a one-to-one move include:

- Synonyms
- Security Roles

The more challenging aspect of migrations is dealing with code objects. SQL Server's Transact-SQL language provides many developer features, many of which are not ANSI standard, and so a careful eye must be used when looking at stored procedures, triggers, views, user-defined functions, and the like. Besides general syntax functions and feature differences, the following items require special attention before they can be migrated completely from SQL Server to MySQL:

- Assemblies
- Types
- DDL and statement-based triggers (MySQL has row-based triggers)
- Proprietary SQL Server function calls
- Certain cases of dynamic T-SQL

Step 2: Design the MySQL Targets

Once the SQL Server source metadata has been obtained and digested, the next step is to design the MySQL target database. This basically involves translating the source objects and their properties (such as column datatypes) to MySQL complements. As one can imagine, this step can be extremely time consuming and error-prone if attempted by hand as most databases will have thousands of object properties that must be converted. Again, for a detailed listing of SQL Server to MySQL transformation datatypes and functions, please see Appendix A of this paper.

Note that many data modeling tools have the ability to convert a SQL Server schema to a MySQL schema with only a few mouse clicks. Naturally, the models can be tweaked if need be. The automatic conversion and migration performed by development and modeling tools such as MySQL Workbench, for SQL Server to MySQL data objects is a huge time saver and can result in lots of productivity gains for a database migration team.

The conversion of code objects is a different matter, however help is available in third party tools such as

SQLWays from Ispirer that can convert SQL Server T-SQL to MySQL stored procedure, trigger, and function code.

Step 3: Execute the Migration to MySQL

Once the source SQL Server metadata has been understood and the MySQL target database designed, the next step is to run the actual data migration process. The extract, transform, and load (ETL) stage can be quite elaborate depending on what one wants to accomplish. There are many heavy-duty third party ETL tools on the market that offer extreme flexibility in just how to move, aggregate, map, and transform data from SQL Server to MySQL databases. Further, Open Source ETL tools such as Pentaho and Talend have free versions of their products that can handle most data movement/transformation use cases.

Of course, Microsoft also makes available Data Transformation Services (DTS in SQL Server 2000) and Integration Services (SQL Server 2005-12), which can help facilitate any SQL Server migration to MySQL. Microsoft's built-in migration tools support moving SQL Server data to MySQL with little effort being required on the part of the DBA.

For those using SQL Server who are not familiar with Integration Services, it is possible to move data from SQL Server to MySQL using a combination of the SQL Server bulk copy program (BCP) and the MySQL LOAD DATA INFILE utility. A DBA can create data files with SQL Server BCP where the data is delimited by an appropriate character (such as a comma, semi-colon, etc.) and then load the data into MySQL with LOAD DATA INFILE with the same delimiter being specified.

A final way to load a SQL Server database into MySQL using Microsoft-supplied aids is to use the export capabilities of Microsoft Access. A DBA can export any dataset in Access to MySQL by clicking on an Access table and then using a combination of the export function and MySQL's ODBC driver. Note that indexes are not normally exported with the table structure and data.

Another data migration strategy is to migrate all the existing SQL Server objects and data from the source database to a MySQL staging database. Once safely in the MySQL database server, a DBA or developer can create stored procedures or other migration code that selectively moves and manipulates data from the staging database into another MySQL database that will be used for further development or production.

Step 4: Validate the Migration

Once everything has been extracted from the source SQL Server database into MySQL, it is wise to perform follow-up checks to ensure everything is as it should be with respect to matching object and row counts as well as ancillary items such as indexes and supporting code objects (views, etc.). It is also smart to perform a set of benchmark tests so acceptable performance can be validated. This step is perhaps the most neglected of all the migration lifecycle steps as it has traditionally not been easy to do.

Proper performance testing catches the performance problems that inadequate user and quality assurance testing miss. In a nutshell, performance testing simulates what is expected from real world use. It stresses the MySQL database in ways that could otherwise only be accomplished by opening the floodgates of the production user community.

Smart performance testing uses the following components to pull off a realistic simulation of what a database will experience during expected production usage:

- **Anticipated user presence –** it is critical that the test simulate the number of user connections that are expected during peak times and normal working hours. This is the major area where manual methods that pick a subset of users to test a database and application fail. The database may run just fine with 10 or so user connections, but may fall over when 400 connect to the system.
- **Repetitive user activity –** once the anticipated user sessions have connected to the database, they obviously have to "do something" for the system to be stressed. And they can't just "do something" once. Either all or a portion of the connected sessions need to repetitively perform tasks as they would occur during a normal workday. For an OLTP system, this may mean entering multiple orders. For a data warehouse, this may mean issuing long running analytical queries. The key is that the work is repetitive so repeated blows are dealt against the database.
- **Extended duration -** once you have a set number of sessions performing repetitive work, you next need to ensure that the work continues for a period of time that makes the test meaningful. What you are looking for is to unearth problems that take time to develop. For example, a MySQL table may not become fragmented after 30 minutes of OLTP work, but may surprisingly fragment in a dramatic fashion after 2 or more hours of repeated action.
- **Expected production data volume –** to have a truly valid test, you need to load your database with test data that is approximately the size you expect your database to be in the real world. This component is easily met if an existing production database has been migrated to MySQL.

MySQL 5.1 and higher provides an easy to use load testing utility – mysqlslap – that offers a command line driven way of benchmarking a MySQL server. The utility offers a quick way of creating a test schema, populating that schema with test data, creating any number of virtual users that execute a series of predefined or ad-hoc SQL statements against a test database, and reporting on the test results

Leveraging MySQL and SQL Server Together

If MySQL will coexist in a data center alongside SQL Server, what are some best practice tips that help a DBA who is charged with managing both platforms? Although not exhaustive, the following advice will help DBAs ensure they are being as productive as they possibly can be:

- Database tools can provide huge gains in productivity if the right products are chosen. Many third party database tools vendors today support MySQL and SQL Server together, so the DBA should investigate whether these tools make sense for them. For example, Quest TOAD is heavily used in many SQL Server shops by DBAs because of the time-savings benefits it offers. A DBA who is used to the TOAD interface can easily transfer to MySQL by using the TOAD for MySQL product offered by Quest. The same is true for users of Embarcadero's DBArtisan product that supports SQL Server. DBArtisan now supports MySQL, which means a DBA can use one tool to manage both the SQL Server and MySQL platforms. Of course, a DBA can also utilize tools supplied from the database vendor such as SQL Server's Management Studio (Enterprise Manager in SQL Server 2000 and below) product and of course MySQL Workbench is available from Oracle.

- SQL Server DBAs who are used to enabling SQL Server to automatically take care of itself should utilize the same complementary MySQL features as well so any additional management is avoided. This includes things like enabling InnoDB tablespace files to automatically grow (which corresponds to enabling the autogrowth property for SQL Server data and log files), setting up MySQL binary logging for point-in-time recovery (which corresponds to using a full recovery model in SQL Server, with periodic transaction log dumps), and other like items.

- Remember that in MySQL, unlike SQL Server, you have a number of different underlying database engines at your disposal that are designed for particular application scenarios. Make sure you choose the right engine for the right purpose.

- Generous memory allocations help MySQL in the areas of performance just like in SQL Server, so depending on the underlying engine being used, ensure enough RAM is provided for enabling memory access to data (as opposed to data being read from disk) as the defaults for MySQL are too low for most every enterprise application. Realize too, just as in SQL Server, just because data is always read in RAM, it doesn't mean the actual SQL code is efficient. Unnecessary logical I/O can hurt performance just like physical I/O can.

- Clustered indexes in MySQL differ slightly from those in SQL Server in that:

    - Clustered indexes are only available with the InnoDB storage engine.
    - Clustered indexes must be built on a table's primary key.
    - If a clustered index is not explicitly defined on an InnoDB table, MySQL will automatically create one on a primary key, a unique key (if a primary key is not available), or an internal row identifier if the table contains no unique constraint whatsoever.

- Partitioning in MySQL requires no creation of partition functions or partition schemes as in SQL Server. Instead, the partitions and type of partitioning (range, hash, key, list, composite) are defined in a table's creation DDL.

- Unlike SQL Server 2005-8, enabling 'snapshot read' (where readers don't block writers and vice versa) in MySQL is automatic and requires no setting of database options or prefacing SQL statements with special commands for snapshot isolation reads. However, snapshot read is currently only available for specific storage engines (like InnoDB, Archive, etc.) so check the current MySQL manual for each engine's supported locking abilities.

- Because inefficient SQL can have such a drastic negative impact on an otherwise well-running MySQL database, MySQL DBAs should enable the slow log to catch poorly-tuned SQL code and review the results of the log on a periodic basis. With MySQL, the slow log or the general log can be set up to mirror the results found in SQL Server's dynamic management views or SQL Profiler tool that are used to ferret out bad SQL code. Versions 5.1 and higher provide SQL performance tables that can be queried via SQL to get inefficient code metrics. In terms of other performance monitoring advice, the MySQL SHOW GLOBAL STATUS and SHOW INNODB STATUS commands can be used to examine raw MySQL performance metrics. MySQL Enterprise Monitor with Query Analyzer offers more advanced monitoring and advice functionality, and should be used in most production environments where MySQL is deployed.

- MySQL DBAs should use whatever schedule they use to check SQL Server Error/Notification Logs to also check the MySQL Error Log for any occurrences of abnormal behavior.

- Those used to using the SQL Server Agent to schedule and execute SQL/T-SQL jobs can utilize the Event Scheduler in MySQL 5.1 and above for the same purpose.

- DBAs with production applications in need of around-the-clock monitoring should evaluate MySQL Enterprise Edition as the certified software provides high degrees of reliability and the built-in software upgrade and monitoring/advisory tools help proactively guarantee uptime and assist in diagnosing and resolving performance issues.

Conclusion

From both a business and a technical perspective, the question of whether a migration to MySQL from SQL Server makes sense is easily answered. Countless organizations are enjoying huge cost savings with MySQL for their web, custom enterprise and embedded applications.

A summary of why to consider a move to MySQL from Microsoft SQL Server (besides cost-savings) includes the following:

- MySQL runs great on the Microsoft Windows platform, is extremely popular as evidenced by many developing and running production MySQL databases on Windows, but MySQL can be deployed on other platforms if desired, whereas SQL Server cannot.
- Regarding installation and configuration, MySQL installs faster, has a smaller footprint while still being able to manage fairly large databases, and has less configuration knobs that need turning than SQL Server.
- There are no size restrictions (CPU, RAM, database size, etc.) in any MySQL database offering unlike Microsoft's constraints that are placed on their standard, workgroup, compact, and express editions.
- MySQL storage engines provide more flexibility and offer more performance and custom application options over SQL Server's standard RDBMS table type.
- MySQL's feature set can handle the vast majority of RDBMS use cases (e.g. OLTP, Web, Hosting, SaaS, Cloud, etc.) and has simpler implementation models than SQL Server in some areas (e.g. partitioning, replication).
- In the area of high availability, MySQL has a number of proven solutions including replication, SANs, and MySQL Cluster, which equal or best SQL Server depending on the scenario.
- Although MySQL lacks some of SQL Server's optimizer sophistication and parallel features, MySQL's performance has been proven to deliver under heavy OLTP and web-scale workloads and can scale both up and out very well.
- MySQL's monitoring and query analysis methodology is global in nature and is better suited to more easily monitor and tune many servers at one time over SQL Server's server-at-a-time performance analysis paradigm.
- The ubiquity of MySQL, the Open Source nature of the product, and its great Community provide many benefits including a great developer and DBA network of everyone working together to help ensure a high-quality product and each other's success.

Although SQL Server database migrations are not something normally tackled with ease, following the simple data migration steps outlined in this paper and leveraging migration tools such as MySQL Workbench helps ensure a smooth transition and ultimate success in the end. Making the switch from SQL Server to MySQL – whether done in full or in a partial format where both MySQL and SQL Server are used for the proper application situation – can make great sense, both from a financial and a technology perspective.

Additional Resources

MySQL on Windows Resource Center:
http://www.mysql.com/why-mysql/windows/

MySQL on Windows Customer References
http://www.mysql.com/customers/operatingsystem/?id=109

MySQL Enterprise Edition Trial
http://www.mysql.com/trials/

MySQL Workbench and Migration Toolkit (download and demos)
http://www.mysql.com/products/workbench/

To contact an Oracle MySQL Representative:
http://www.mysql.com/about/contact/

Appendix A – SQL Server to MySQL - Datatypes

This section provides a quick listing of the compatibility between SQL Server and MySQL with respect to datatypes.

Comparable Datatypes

The following datatypes can be mapped in a one-to-one relationship from SQL Server to MySQL:

- BIGINT
- BINARY
- BIT
- CHAR
- CHARACTER
- DATETIME
- DEC, DECIMAL
- FLOAT
- DOUBLE PRECESION
- INT, INTEGER
- NCHAR, NATIONAL CHARACTER
- NVARCHAR, NCHAR VARYING
- NATIONAL CHAR VARYING, NATIONAL CHARACTER VARYING
- NUMERIC
- REAL
- SMALLINT
- TEXT
- TIMESTAMP
- TINYINT
- VARBINARY
- VARCHAR, CHAR VARYING, CHARACTER VARYING

Note that the DATETIME datatype in MySQL is a seven byte value that is currently down to the second precision (although a future release will allow greater granularity), while SQL Server is an eight byte value that has granularity down to the 3/100 second precision.

In addition, TIMESTAMP is a special datatype that is typically used to store the time a row was last created or updated (which MySQL handles automatically).

Datatypes Requiring Conversion

The following map can be used to convert SQL Server datatypes that do not map in 1-to-1 relationship to MySQL:

SQL Server	MySQL
IDENTITY	AUTO_INCREMENT
NTEXT, NATIONAL TEXT	TEXT CHARACTER SET UTF8
SMALLDATETIME	DATETIME
MONEY	DECIMAL(19,4)
SMALL MONEY	DECIMAL(10,4)
UNIQUEIDENTIFIER	BINARY(16)
SYSNAME	CHAR(256)

Appendix B – SQL Server to MySQL – Predicates

Overview

This section provides a list of the SQL predicates supported by SQL Server and compares them to equivalent predicates supported by MySQL version 5.1. The predicates are split into nine categories:

- Predicates for comparison conditions
- Predicates for exists conditions
- Predicates for floating-point conditions
- Predicates for in conditions
- Predicates for null conditions
- Predicates for pattern matching conditions
- Predicates for range conditions

Synopsis

SQL Server and MySQL 5.1 both support the use of conditions in the WHERE clause of the SELECT, DELETE, and UPDATE statements, as well as in the HAVING clause of the SELECT statement.

Boolean conditions

A Boolean condition combines two other conditions, to return a single result. SQL Server supports three Boolean conditions.

Summary

Each SQL Server Boolean condition has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server boolean conditions

expression1 AND expression2

Returns TRUE if both expressions return TRUE, UNKNOWN if either expression is NULL, FALSE otherwise.

MySQL equivalent: AND

NOT expression

Negates the result of expression; returns TRUE if expression is FALSE, UNKNOWN if expression is NULL, and FALSE if expression is TRUE.

MySQL equivalent: NOT

expression1 OR expression2

Returns TRUE if either expression returns TRUE, UNKNOWN if either expression is NULL, FALSE otherwise.

MySQL equivalent: OR

Comparison conditions

Comparison conditions compare two expressions. SQL Server supports eight comparison conditions.

Summary

Each SQL Server comparison condition has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server comparaison conditions

expression1 = *expression2*

Returns TRUE if the expressions are equal, UNKNOWN if either expression is NULL, and FALSE otherwise.

MySQL equivalent: =

expression1 <> *expression2*

Additional formats accepted (not all forms available on every platform): != ^= ¬=
Returns TRUE if the expressions are not equal, UNKNOWN if either expression is NULL, and FALSE otherwise.

MySQL equivalent: <> !=

expression1 < *expression2*
Returns TRUE if expression1 is less than expression2, UNKNOWN if either expression is NULL, and FALSE otherwise.

MySQL equivalent: <

expression1 <= *expression2*
Returns TRUE if expression1 is less than or equal to expression2, UNKNOWN if either expression is NULL, and FALSE otherwise.

MySQL equivalent: <=

expression1 > *expression2*

Returns TRUE if expression1 is greater than expression2, UNKNOWN if either expression is NULL, and FALSE otherwise.
MySQL equivalent: >

expression1 >= *expression2*

Returns TRUE if expression1 is greater than or equal to expression2, UNKNOWN if either expression is NULL, and FALSE otherwise.

MySQL equivalent: >=

expression1 comparison ANY expression2
expression1 comparison SOME expression2

Synonyms; return TRUE if the comparison condition returns TRUE for at least one value in expression2 and FALSE if the comparison condition returns either FALSE for every value in expression2 or returns an empty set (zero rows).

comparison must be one of: = <> < <= > >=.
expression2 is generally a subquery, but may resolve to any expression.

MySQL equivalent: ANY, SOME, provided expression2 is a subquery.

expression1 comparison ALL expression2

Returns FALSE if the comparison condition returns FALSE for at least one value in expression2 and TRUE if the comparison condition returns either TRUE for every value in expression2 or returns an empty set (zero rows).

comparison must be one of: = <> < <= > >=.

expression2 is generally a subquery, but may resolve to any expression.

MySQL equivalent: ALL, provided expression2 is a subquery.

Exists conditions

Exists conditions test for rows in a sub query. SQL Server supports two exists conditions.

Summary

Each SQL Server exists condition has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server exists conditions

EXISTS (subquery)

Returns TRUE if *subquery* returns at least one row, FALSE otherwise.

MySQL equivalent: EXISTS

NOT EXISTS (subquery)

Returns TRUE if *subquery* returns zero rows, FALSE otherwise.

MySQL equivalent: NOT EXISTS

In conditions

In conditions test whether a value is found in a set. SQL Server supports two in conditions.

Summary

Each SQL Server in condition has an exact MySQL equivalent, and therefore no action is necessary

when migrating from SQL Server to MySQL.

SQL Server in conditions

expression IN {value_list | (subquery)}

Returns TRUE if expression equals any value in value_list (or returned by subquery), UNKNOWN if either argument is NULL, FALSE otherwise.

MySQL equivalent: IN

expression NOT IN {value_list | (subquery)}

Returns TRUE if expression does not equal any value in value_list (or returned by subquery), UNKNOWN if either argument is NULL, FALSE otherwise.

MySQL equivalent: NOT IN

Null conditions

Null conditions test for null values. SQL Server supports two null conditions.

Summary

Each SQL Server null condition has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server null conditions

expression IS NULL

Returns TRUE if the result of expression is NULL, FALSE otherwise.

MySQL equivalent: IS NULL

expression IS NOT NULL

Returns TRUE if the result of expression is not NULL, FALSE otherwise.

MySQL equivalent: IS NOT NULL

Pattern matching conditions

Pattern matching conditions test whether a value matches a specific pattern. SQL Server supports nine pattern matching conditions.

SQL Server pattern matching conditions

string_expression LIKE pattern [ESCAPE escape_string]

Returns TRUE if string_expression matches pattern, UNKNOWN if any argument is NULL, FALSE

otherwise.
Uses the current (input) character set to interpret all arguments.

MySQL equivalent: LIKE

string_expression NOT LIKE pattern [ESCAPE escape_string]

Returns TRUE if string_expression does not match pattern, UNKNOWN if any argument is NULL, FALSE otherwise.
Uses the current (input) character set to interpret all arguments.

MySQL equivalent: NOT LIKE

Range conditions

Range conditions test for inclusion in a range of values. SQL Server supports two range conditions.

Summary

Each SQL Server range condition has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server range conditions

expression1 BETWEEN expression2 AND expression3

Returns TRUE if expression1 falls into the range of values specified by the other expressions (i.e. if expression1 is greater than or equal to expression2 and less than or equal to expression3), UNKNOWN if any expression is NULL, FALSE otherwise.

MySQL equivalent: BETWEEN

expression1 NOT BETWEEN expression2 AND expression3
Returns TRUE if expression1 does not fall into the range of values specified by expression2 and expression3, UNKNOWN if any expression is NULL, FALSE otherwise.

MySQL equivalent: NOT BETWEEN

Appendix C – SQL Server to MySQL – Operators and Date Functions

Overview

This paper provides a list of the built-in SQL operators supported by SQL Server Database 10g and compares them to equivalent operators supported by MySQL version 5.0. The operators are split into five categories:

- Arithmetic operators
- Concatenation operators

- Hierarchical Query operators
- Set operators

Arithmetic operators

Summary

Each SQL Server arithmetic operator has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server arithmetic operators

+
When used as a unary operator, indicates a positive numeric or temporal expression.
When used as a binary operator, adds two numeric or temporal values.

MySQL equivalent: +

-
When used as a unary operator, indicates a negative numeric or temporal expression.
When used as a binary operator, subtracts one numeric or temporal value from another.
MySQL equivalent: -

*
Binary operator; multiplies numeric expressions.

MySQL equivalent: *

/
Binary operator; divides numeric expressions.

MySQL equivalent: /

Concatenation operators

Summary

When the MySQL server is started with the –ansi option, the SQL Server || (concatenation) operator has an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

When the MySQL server is started in the default mode, map SQL Server || to MySQL CONCAT('string1', 'string2').

No other concatenation operators are supported by SQL Server, although SQL Server's CONCAT string function serves the same purpose.

SQL Server concatenation operators

+

Concatenates character strings and CLOB values.
MySQL equivalent (--ansi mode): ||
MySQL equivalent (default mode): CONCAT('string1', 'string2')

Hierarchical query operators

Summary

SQL Server LEVEL has no MySQL equivalent.

Set operators

Summary

SQL Server INTERSECT and EXCEPT have no MySQL equivalent.

All other SQL Server set operators have an exact MySQL equivalent, and therefore no action is necessary when migrating from SQL Server to MySQL.

SQL Server set operators

UNION
Combines two SELECT queries.
Returns all distinct (non-duplicate) rows found by either SELECT.

MySQL equivalent: UNION.

UNION ALL
Combines two SELECT queries.
Returns all rows found by either SELECT, including duplicates.

MySQL equivalent: UNION ALL.

Date Functions

Summary

The majority of date functions in SQL Server can be replicated in MySQL.

Date Formats

SQL Server	MySQL
YYYYMMDD	YYYYMMDD
YYYY MonthName DD	YY[YY]-MM-DD
MM/DD/YY[YY]	YY[YY]/MM/DD
MM-DD-YY[YY]	YY[YY]-MM-DD
MM.DD.YY[YY]	YY[YY].MM.DD

Date Functions

SQL Server	MySQL
DATEADD(day, 1, GETDATE())	DATE_ADD(NOW(), INTERVAL 1 DAY)
DATEDIFF(day, GETDATE(), GETDATE()-1)	DATEDIFF(NOW(), NOW() – INTERVAL 1 DAY)
DATENAME(month, GETDATE())	DATE_FORMAT(NOW(), '%M') MONTHNAME(NOW())
DATENAME(weekday, GETDATE())	DATE_FORMAT(NOW(), '%W') DAYNAME(NOW())
DATEPART(month, GETDATE())	DATE_FORMAT(NOW(), '%m')
DAY(GETDATE())	DATE_FORMAT(NOW(), '%d') DAY(NOW()) *(as of MySQL 4.1.1)* DAYOFMONTH(NOW())
GETDATE()	NOW() SYSDATE() CURRENT_TIMESTAMP CURRENT_TIMESTAMP()
GETDATE() + 1	NOW() + INTERVAL 1 DAY CURRENT_TIMESTAMP + INTERVAL 1 DAY
GETUTCDATE()	UTC_TIMESTAMP()
MONTH(GETDATE())	MONTH(NOW())
YEAR(GETDATE())	YEAR(NOW())

Appendix D – T-SQL Conversion Suggestions

Overview

As of March 2006 the MySQL migration tool does not migrate MS SQL Transact SQL code to MySQL. Because of this stored procedures and other code objects in SQL Server need to be migrated manually. This appendix provides suggestions in converting Microsoft T-SQL code to MySQL.

Getting T-SQL Code out of SQL Server

The stored procedures are extracted from SQL Server using the SQL Server 'create scripts' option with all Stored Procedure selected and the generate a file for each stored procedure flag selected.

Procedure Formats

While the general format for an MS stored procedure is:

```
CREATE PROCEDURE name
        Param1 type,
        Param2 type
AS
```

```
        Statement1
        Statement2
```

MySQL requires that a format of:

```
CREATE PROCEDURE name (
        Param1 type,
        Param2 type
)
BEGIN
        Statement1 ;
        StateMent2 ;
        …
END ;
```

With the major differences being that MySQL:

1. Requires that the parameter list be in '(' ')" pairs
2. Cannot take an AS after the parameter list
3. Requires a BEGIN END pair if there is more than one statement in the stored procedure
4. Requires that every statement is terminated by a ';'.
5. In MySQL stored procedures all variable declarations must come before the first non-declare statement.

Error Handling

In SQL Server, errors below a certain level are returned in @@ERROR and the Stored Procedure continues while errors above that level terminate the Stored Procedure immediately and return and error to the caller. In MySQL most errors terminate the Stored Procedure and return an error code. To get a MySQL Stored Procedure to behave more like a SQL Server Stored Procedure you must define an error handler as follows:

```
DECLARE "@ERROR" INT DEFAULT 0;
        DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
                        SET "@ERROR" = 1;
        END;
```

And then modify your Stored Procedure code to use the @ERROR variable instead of @@ERROR, for example, replace:

```
        IF @@ERROR <>0 GOTO ERROUT

        UPDATE MessageBoardEntries SET ReplyCount = ReplyCount - 1
        WHERE EntryID = @EntryID
        IF @@ERROR <>0  GOTO ERROUT

        UPDATE MessageBoardCategories SET PostCount = PostCount -1
        WHERE CategoryID = @CategoryID
        IF @@ERROR <>0 GOTO ERROUT
```

With:

```
DECLARE "@ERROR" INT DEFAULT 0 ;
```

```
        DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN
                        SET "@ERROR" = 1 ;
        END ;

        INSERT groupmessageboardreplies (
                parentid,
                authorid,
                body )
        VALUES (
                "@entryid",
                "@authorid",
                "@body"  ) ;

        IF "@ERROR" = 0 THEN
                UPDATE groupmessageboardentries
                set replycount = replycount + 1,
                        lastpostdate = NOW(),
                        lastposter = "@authorid"
                WHERE entryid = "@entryid" ;
        END IF

IF "@ERROR" = 0 THEN
                UPDATE groupmessageboards
                set lastpostdate = NOW(),
                        postcount = postcount + 1,
                        lastposterid = "@authorid",
                        lastpostentryid = "@entryid"
                WHERE groupid = "@groupid" ;
        END IF;
```

Use LIMIT instead of TOP

The MySQL, the LIMIT feature in a SELECT both replaces the SQL Server TOP function and adds new functionality not available in TOP. In its simplest form:

Replace:

```
SELECT TOP 100 * FROM TABLE
```

With:

```
SELECT * FROM TABLE LIMIT 100 ;
```

If you want to get 10 records starting at position 100, then use:

```
SELECT * FROM TABLE LIMIT 100,10 ;
```

Note that the values in the LIMIT clause must be constants, variables are not allowed.

You cannot use:

```
SELECT * FROM TABLE LIMIT @start,@rows ;
```

To get around this issue you can use a PREPARE statement as follows. Assuming the query you want to run is:

```
select c.classifiedid,
       c.subject,
       c.createddate,
       c.views,
       c.userid

from    schoolsclassifieds c
where   categoryid = "@categoryid"
and     schoolid = "@schoolid"
order by classifiedid desc
limit “@startrow","@maxrows" ;
```

Use the following code to implement the SELECT:

```
PREPARE STMT FROM 'select c.classifiedid,
       c.subject,
       c.createddate,
       c.views,
       c.userid

from    schoolsclassifieds c
where   categoryid = ?
and     schoolid = ?
order by classifiedid desc
limit ?,?' ;

EXECUTE STMT USING
"@categoryid","@schoolid","@startrow","@maxrows"  ;
```

LIMIT and Optimization

In some cases, MySQL handles a query differently when you are using LIMIT <*row_count*> and not using HAVING:

- If you are selecting only a few rows with LIMIT, MySQL uses indexes in some cases when normally it would prefer to do a full table scan.

- If you use LIMIT *row_count* with ORDER BY, MySQL ends the sorting as soon as it has found the first *row_count* rows of the sorted result, rather than sorting the entire result. If ordering is done by using an index, this is very fast. If a filesort must be done, all rows that match the query without the LIMIT clause must be selected, and most or all of them must be sorted, before it can be ascertained that the first *row_count* rows have been found. In either case, after the initial rows have been found, there is no need to sort any remainder of the result set, and MySQL does not do so.

- When combining LIMIT *row_count* with DISTINCT, MySQL stops as soon as it finds *row_count* unique rows.

- In some cases, a GROUP BY can be resolved by reading the key in order (or doing a sort on the key) and then calculating summaries until the key value changes. In this case, LIMIT *row_count*

does not calculate any unnecessary GROUP BY values.

- As soon as MySQL has sent the required number of rows to the client, it aborts the query unless you are using SQL_CALC_FOUND_ROWS.

- LIMIT 0 quickly returns an empty set. This can be useful for checking the validity of a query. When using one of the MySQL APIs, it can also be employed for obtaining the types of the result columns. (This trick does not work in the MySQL Monitor, which merely displays Empty set in such cases; you should instead use SHOW COLUMNS or DESCRIBE for this purpose.)

- When the server uses temporary tables to resolve the query, it uses the LIMIT *row_count* clause to calculate how much space is required.

IF THEN … ELSE … END IF ;

MySQL requires that an IF include a THEN and that a block with more than one statement must be terminated by an 'END IF ;'. Note that the ';' after the END IF is required.

IF <expr> THEN
Statement 1;
ELSE
Statement 2 ;
END IF;

DATETIME with default values

The MySQL migration tool cannot handle migrating DATETIME columns that have a default value of getdate(). For example:

UPLOADED DATETIME NULL DEFAULT GETDATE(),

Tables with DATETIME columns with the above default will not automatically be converted and must be manually modified as follows:

UPLOADED TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMTP,

Note that only the TIMESTAMP type can have a default value of CURRENT_TIMESTATMP so if you want to keep the default you must also change the type from DATETIME to TIMEDSTAMP.

Another difference between MySQL and SQL Server is that MySQL only allows 1 column per table to have a default value of CURRENT_TIMESTATMP. For tables in SQL Server with two or more columns with default values set to GETDATE(), one or the other column must have its default value removed for the table to create successfully. In addition, MySQL implicitly assigns a default value of CURRENT_TIMESTAMP to the first TIMESTAMP column in a create table statement unless it is explicitly assigned a default value other than NULL or CURRENT_TIMESTATMP. This means that if you are converting a table that has two columns with a default values of getdate() and you choose the second column as the one to have the default value you will get a error creating the table if you do not explicitly assign a default value, such as 0 or '0000-00-00 00:00:00' to the first column.

Index Name Length Differences

MySQL does not support index names longer than 64 bytes. Indexes with names longer than 64 bytes must be renamed.

Support for SQL Server Declared Tables

SQL Server supports a 'table' data type in an stored procedure. For example:

```
DECLARE "@Results" TABLE
(Pos int IDENTITY, ClassifiedID int)
```

MySQL does not support this. Instead, use a temporary in memory table:

```
CREATE TEMPORARY TABLE temp1
(ClassifiedID int) ENGINE = MEMORY;

        < other code >

DROP TEMPORARY TABLE temp1 ;
```

Obtaining Row Counts

In MySQL, the function ROW_COUNT() returns the number of rows effected by the last insert, update, or delete. Use this instead of the SQL Server @@ROWCOUNT variable.

Default Values for Parameters

In SQL Server a parameter can have the form:

```
@Param int default  0
```

MySQL does not support this. The application must be changed to pass in the appropriate default values.

Use of SELECT INTO

In SQL Server, if you want to set a variable based on the results of a SELECT you use:

```
SELECT INTO A=col1 …
```

In MySQL, the INTO will cause an error, so you will need to use:

```
SELECT A=col1 …
```

Uses of IFNULL

Use the IFNULL() function to check if a value is NULL and then return either the value or another value based on the results of the test. The IFNULL() function returns A if A is not NULL, and returns B otherwise.

Replace DATEADD with ADDDATE()

The MySQL function ADDDATE() replaces DATEADD(). For example:

Replace:

DATEADD(dd, -14, GETDATE())

With:

ADDDATE(NOW(),INTERVAL -14 DAY) ;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

This exact syntax works in MySQL and can be placed at the top of each Stored Procedure if desired. The better solution is to set the value of transaction-isolation in the my.cnf file:

transaction-isolation=READ-UNCOMMITTED

Replace SET ROWCOUNT = @rows with SQL_SELECT_LIMIT

Use SQL_SELECT_LIMIT instead of ROWCOUNT in MySQL to limit the rows returned in a select. This method allows the number of rows returned to be set by a variable without using a prepare statement.

Replace:

SET ROOWCOUNT = @rowcount

With

SET SQL_SELECT_LIMIT=@rowcount;

Perl Script to Help with Conversion of T-SQL to MySQL

The Perl script below assists in the conversion of SQL Server T-SQL to MySQL. Note that it performs rudimentary changes only, so many complex SQL Server code objects will still have to be massaged by hand. Many thanks go to Brian Miezejewski of MySQL Professional Services for providing the utility.

The method of using the Perl conversion utility is as follows:

1. Extract all desired SQL Server stored procedures into files and place them into a single directory.
2. Run the below perl script 'MSProcstoMySql.pl' as such: perl MSProcstoMySql.pl *.PRC
3. The procedure will create a subdirectory called 'converted' and place the converted stored procedures in that directory.

The following example shows a stored procedure before and after the above perl script is run:

```
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO

CREATE PROCEDURE DeleteMessageBoardReply
```

```
        @ReplyID int,
        @EntryID int,
        @CategoryID int
AS
        SET NOCOUNT ON
        SET LOCK_TIMEOUT 4000

        BEGIN TRAN

        DELETE FROM MessageBoardReplies WHERE ReplyID = @ReplyID
        IF @@ERROR <>0  GOTO ERROUT

        UPDATE MessageBoardEntries SET ReplyCount = ReplyCount - 1
        WHERE EntryID = @EntryID
        IF @@ERROR <>0  GOTO ERROUT

        UPDATE MessageBoardCategories SET PostCount = PostCount -1
        WHERE CategoryID = @CategoryID
        IF @@ERROR <>0  GOTO ERROUT

        COMMIT TRAN
        RETURN
        ERROUT:
        ROLLBACK TRAN
GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO



DELIMITER $$;
SET SQL_MODE = ANSI_QUOTES$$


DROP PROCEDURE IF EXISTS DeleteMessageBoardReply $$

CREATE PROCEDURE DeleteMessageBoardReply (
"@replyid" int,
"@entryid" int,
"@categoryid" int
)
BEGIN

        /*BEGIN tran*/

        delete from messageboardreplies WHERE replyid ="@replyid"
        IF @@error <>0  goto errout

        UPDATE messageboardentries set replycount = replycount - 1
        WHERE entryid ="@entryid"
        IF @@error <>0  goto errout
```

```
        UPDATE messageboardcategories set postcount = postcount -1
        WHERE categoryid ="@categoryid"
        IF @@error <>0  goto errout

        /*commit tran*/
        return
        errout:
        rollback tran
END;
$$
DELIMITER ;$$
```

The Perl conversion routine performs the following functions:

1. Renames the file from dbo.spName.PRC to spName.sql
2. Wraps the code in:
   ```
   DELIMITER $$;
   SET SQL_MODE = ANSI_QUOTES$$
   <Stored procedure code>
   $$
   DELIMITER ;$$
   ```
3. Removes all 'GO' statements from the code
4. Remove all SET statements for QUOTED_IDENTIFIER, ANSI_NULL, NOCOUNT, and LOCK_TIMEOUT.
5. Comments out both BEGIN TRAN and COMMIT TRAN
6. Replaces all @@IDENTITY references with a call to LAST_INSERT_ID()
7. Wraps every reference to a variable of the format @<name> in double quotes, i.e. @vari becomes "@vari"
8. Makes all table names lower case
9. Changes NVARCHAR types to VARCHAR
10. Places the parameter list in parentheses
11. Removes the AS after the parameter list
12. Wraps all code after the parameter list in a BEGIN END block
13. Replaces all call to GETDATE() with calls to NOW()

While the script makes many changes to the SQL Server stored procedures, it does not fully convert them. Each stored procedure will still need to be reviewed and modified before it can be used. The following list includes some of the more common issues that will still need to be fixed:

Place a ';' after every statement
Add an error handler
Remove any IF @@ERROR<>0 GOTO LABEL and replace them with IF … END IF blocks using the value set in the error handler.

Perl Conversion Script

```
#!/usr/bin/perl

if (! -d "converted" ) {
        mkdir "converted" ;
}
```

```
foreach my $inFile (@ARGV) {

        open INFILE, "<$inFile" ;
        my $outFile = $inFile ;
        $outFile =~ s/\.[^\.]*$/.sql/ ;
        $outFile =~ s/^dbo\.// ;
        open OUTFILE, ">converted/$outFile" ;

        print OUTFILE "DELIMITER \$\$;\n" ;
        print OUTFILE "SET SQL_MODE = ANSI_QUOTES\$\$" ;
        while( <INFILE> ) {
                s/\@\@identity/LAST_INSERT_ID()/i ;
                s/\s(\@[^\s\,\)\@]+)/"$1"/g ;
                /SET QUOTED_IDENTIFIER ON/ && next ;
                /SET NOCOUNT ON/ && next ;
                /SET\s*LOCK_TIMEOUT.*/ && next ;
                /^\s*GO\s*$/ && next ;
                /SET QUOTED_IDENTIFIER OFF/i && next ;
                /SET QUOTED_IDENTIFIER ON/ && next ;
                /SET ANSI_NULLS ON/ && next ;
                /SET ANSI_NULLS OFF/ && next ;
                s/nvarchar/varcharg/ ;
                if (/(CREATE\s+PROCEDURE)\s+([^\n\r]*)/) {
                        print OUTFILE "DROP PROCEDURE IF EXISTS $2 \$\$\n\n" ;
                        print OUTFILE "$1 $2 (\n"  ;
                        next ;
                };
                if (/^\s*AS\s*$/) {
                        print OUTFILE ")\n"  ;
                        print OUTFILE "BEGIN\n"  ;
                        next ;
                };
                s/(BEGIN\s+TRAN[^\n\r]*)/\/*$1*\// ;
                s/(COMMIT\s+TRAN[^\n\r]*)/\/*$1*\// ;
                tr/[A-Z]/[a-z]/;
                s/getdate\(\)/NOW()/i ;
                s/select\s/SELECT / ;
                s/update\s/UPDATE / ;
                s/insert\s/INSERT / ;
                s/declare\s/DECLARE / ;
                s/where\s/WHERE / ;
                s/values/VALUES/ ;
                s/begin/BEGIN/ ;
                s/if/IF/ ;
                /([^\r\n]*)/ && print OUTFILE "$1\n" ;
        }
        print OUTFILE "END;\n" ;
        print OUTFILE "\$\$\n" ;
        print OUTFILE "DELIMITER ;\$\$\n" ;

        close OUTFILE ;
        close INFILE ;
}
```

Appendix E – Sample Migration

This section provides a simple example of moving an SQL Server 2008 database schema over to MySQL using the MySQL Migration Toolkit.

Sample SQL Server Schema

The SQL Server 2008 schema is a small database used for tracking financial investments in a small brokerage company. The SQL Server schema DDL is as follows:

```
--
-- CREATE Tables
--
CREATE TABLE BROKER
(
   BROKER_ID            numeric(18,0) NOT NULL,
   OFFICE_LOCATION_ID   numeric(18,0) NULL,
   BROKER_LAST_NAME     varchar(40)   NOT NULL,
   BROKER_FIRST_NAME    varchar(20)   NOT NULL,
   BROKER_MIDDLE_INITIAL char(1)      NULL,
   MANAGER_ID           numeric(18,0) NULL,
   YEARS_WITH_FIRM      numeric(3,1)  NOT NULL,
   CONSTRAINT BROKER_PK
   PRIMARY KEY CLUSTERED (BROKER_ID)
)
go
IF OBJECT_ID('BROKER') IS NOT NULL
   PRINT '<<< CREATED TABLE BROKER >>>'
ELSE
   PRINT '<<< FAILED CREATING TABLE BROKER >>>'
go
CREATE TABLE CLIENT_TRANSACTION
(
   CLIENT_TRANSACTION_ID     numeric(18,0) NOT NULL,
   CLIENT_ID                 numeric(18,0) NOT NULL,
   INVESTMENT_ID             numeric(18,0) NOT NULL,
   [ACTION]                  varchar(10)   NOT NULL,
   PRICE                     numeric(12,2) NOT NULL,
   NUMBER_OF_UNITS           numeric(18,0) NOT NULL,
   TRANSACTION_STATUS        varchar(10)   NOT NULL,
   TRANSACTION_SUB_TIMESTAMP  datetime     NOT NULL,
   TRANSACTION_COMP_TIMESTAMP datetime     NOT NULL,
   DESCRIPTION               varchar(200)  NULL,
   BROKER_ID                 numeric(18,0) NULL,
   BROKER_COMMISSION         numeric(10,2) NULL,
   CONSTRAINT CLIENT_TRANSACTION_PK
   PRIMARY KEY CLUSTERED (CLIENT_TRANSACTION_ID)
)
go
IF OBJECT_ID('CLIENT_TRANSACTION') IS NOT NULL
   PRINT '<<< CREATED TABLE CLIENT_TRANSACTION >>>'
ELSE
   PRINT '<<< FAILED CREATING TABLE CLIENT_TRANSACTION >>>'
go
```

```
CREATE TABLE CLIENT
(
   CLIENT_ID            numeric(18,0) NOT NULL,
   CLIENT_FIRST_NAME      varchar(20)   NOT NULL,
   CLIENT_LAST_NAME       varchar(40)   NOT NULL,
   CLIENT_GENDER          char(1)       NOT NULL,
   CLIENT_YEAR_OF_BIRTH    numeric(4,0)  NOT NULL,
   CLIENT_MARITAL_STATUS   varchar(20)   NULL,
   CLIENT_STREET_ADDRESS   varchar(40)   NOT NULL,
   CLIENT_POSTAL_CODE      varchar(10)   NOT NULL,
   CLIENT_CITY           varchar(30)   NOT NULL,
   CLIENT_STATE_PROVINCE   varchar(40)   NOT NULL,
   CLIENT_PHONE_NUMBER     varchar(25)   NOT NULL,
   CLIENT_HOUSEHOLD_INCOME numeric(30,0) NULL,
   CLIENT_COUNTRY         varchar(40)   NULL,
   BROKER_ID            numeric(18,0) NOT NULL,
   CONSTRAINT CLIENT_PK
   PRIMARY KEY CLUSTERED (CLIENT_ID)
)
go
IF OBJECT_ID('CLIENT') IS NOT NULL
   PRINT '<<< CREATED TABLE CLIENT >>>'
ELSE
   PRINT '<<< FAILED CREATING TABLE CLIENT >>>'
go

--
-- Foreign Keys
--
ALTER TABLE CLIENT_TRANSACTION
   ADD CONSTRAINT CLIENT_TRANSACTION_CLIENT
   FOREIGN KEY (CLIENT_ID)
   REFERENCES CLIENT (CLIENT_ID)
   ON DELETE CASCADE
go

--
-- CREATE Indexes
--
CREATE NONCLUSTERED INDEX CLIENT_BROKER
   ON CLIENT(BROKER_ID)
go
IF EXISTS (SELECT * FROM sysindexes WHERE id=OBJECT_ID('CLIENT') AND name='CLIENT_BROKER')
   PRINT '<<< CREATED INDEX CLIENT.CLIENT_BROKER >>>'
ELSE
   PRINT '<<< FAILED CREATING INDEX CLIENT.CLIENT_BROKER >>>'
go

--
-- Foreign Keys
--
ALTER TABLE CLIENT_TRANSACTION
   ADD CONSTRAINT CLIENT_TRANSACTION_BROKER
   FOREIGN KEY (BROKER_ID)
   REFERENCES BROKER (BROKER_ID)
   ON DELETE CASCADE
go
ALTER TABLE CLIENT
   ADD CONSTRAINT FK__CLIENT__BROKER_I__0F975522
   FOREIGN KEY (BROKER_ID)
```

```
   REFERENCES BROKER (BROKER_ID)
go

--
-- CREATE Indexes
--
CREATE NONCLUSTERED INDEX CLIENT_TRANSACTION_BROKER
   ON CLIENT_TRANSACTION(BROKER_ID)
go
IF EXISTS (SELECT * FROM sysindexes WHERE id=OBJECT_ID('CLIENT_TRANSACTION') AND
name='CLIENT_TRANSACTION_BROKER')
   PRINT '<<< CREATED INDEX CLIENT_TRANSACTION.CLIENT_TRANSACTION_BROKER >>>'
ELSE
   PRINT '<<< FAILED CREATING INDEX CLIENT_TRANSACTION.CLIENT_TRANSACTION_BROKER >>>'
go
CREATE NONCLUSTERED INDEX CLIENT_TRANSACTION_CLIENT
   ON CLIENT_TRANSACTION(CLIENT_ID)
go
IF EXISTS (SELECT * FROM sysindexes WHERE id=OBJECT_ID('CLIENT_TRANSACTION') AND
name='CLIENT_TRANSACTION_CLIENT')
   PRINT '<<< CREATED INDEX CLIENT_TRANSACTION.CLIENT_TRANSACTION_CLIENT >>>'
ELSE
   PRINT '<<< FAILED CREATING INDEX CLIENT_TRANSACTION.CLIENT_TRANSACTION_CLIENT >>>'
go
CREATE NONCLUSTERED INDEX CLIENT_TRANSACTION_INVESTMENT
   ON CLIENT_TRANSACTION(INVESTMENT_ID)
go
IF EXISTS (SELECT * FROM sysindexes WHERE id=OBJECT_ID('CLIENT_TRANSACTION') AND
name='CLIENT_TRANSACTION_INVESTMENT')
   PRINT '<<< CREATED INDEX CLIENT_TRANSACTION.CLIENT_TRANSACTION_INVESTMENT >>>'
ELSE
   PRINT '<<< FAILED CREATING INDEX CLIENT_TRANSACTION.CLIENT_TRANSACTION_INVESTMENT >>>'
go

--
-- CREATE Tables
--
CREATE TABLE INVESTMENT
(
   INVESTMENT_ID       numeric(18,0) NOT NULL,
   INVESTMENT_TYPE_ID  numeric(18,0) NULL,
   INVESTMENT_VENDOR   varchar(30)   NULL,
   INVESTMENT_NAME     varchar(200)  NULL,
   INVESTMENT_UNIT     varchar(20)   NULL,
   INVESTMENT_DURATION varchar(10)   NULL,
   CONSTRAINT INVESTMENT_PK
   PRIMARY KEY CLUSTERED (INVESTMENT_ID)
)
go
IF OBJECT_ID('INVESTMENT') IS NOT NULL
   PRINT '<<< CREATED TABLE INVESTMENT >>>'
ELSE
   PRINT '<<< FAILED CREATING TABLE INVESTMENT >>>'
go

--
-- CREATE Indexes
--
CREATE NONCLUSTERED INDEX INVESTMENT_INVESTMENT_TYPE
   ON INVESTMENT(INVESTMENT_TYPE_ID)
```

```
go
IF EXISTS (SELECT * FROM sysindexes WHERE id=OBJECT_ID('INVESTMENT') AND
name='INVESTMENT_INVESTMENT_TYPE')
   PRINT '<<< CREATED INDEX INVESTMENT.INVESTMENT_INVESTMENT_TYPE >>>'
ELSE
   PRINT '<<< FAILED CREATING INDEX INVESTMENT.INVESTMENT_INVESTMENT_TYPE >>>'
go

--
-- CREATE Tables
--
CREATE TABLE INVESTMENT_TYPE
(
   INVESTMENT_TYPE_ID   numeric(18,0) NOT NULL,
   INVESTMENT_TYPE_NAME varchar(30)   NOT NULL,
   CONSTRAINT INVESTMENT_TYPE_PK
   PRIMARY KEY CLUSTERED (INVESTMENT_TYPE_ID)
)
go
IF OBJECT_ID('INVESTMENT_TYPE') IS NOT NULL
   PRINT '<<< CREATED TABLE INVESTMENT_TYPE >>>'
ELSE
   PRINT '<<< FAILED CREATING TABLE INVESTMENT_TYPE >>>'
go

--
-- Foreign Keys
--
ALTER TABLE INVESTMENT
   ADD CONSTRAINT FK__INVESTMEN__INVES__108B795B
   FOREIGN KEY (INVESTMENT_TYPE_ID)
   REFERENCES INVESTMENT_TYPE (INVESTMENT_TYPE_ID)
go
ALTER TABLE CLIENT_TRANSACTION
   ADD CONSTRAINT CLIENT_TRANSACTION_INVESTMENT
   FOREIGN KEY (INVESTMENT_ID)
   REFERENCES INVESTMENT (INVESTMENT_ID)
   ON DELETE CASCADE
go

--
-- CREATE Tables
--
CREATE TABLE OFFICE_LOCATION
(
   OFFICE_LOCATION_ID    numeric(18,0) NOT NULL,
   OFFICE_NAME           varchar(20)   NOT NULL,
   OFFICE_ADDRESS        varchar(50)   NOT NULL,
   OFFICE_CITY           varchar(30)   NOT NULL,
   OFFICE_STATE_PROVINCE varchar(40)   NOT NULL,
   OFFICE_POSTAL_CODE    varchar(10)   NOT NULL,
   OFFICE_COUNTRY        varchar(40)   NULL,
   CONSTRAINT OFFICE_LOCATION_PK
   PRIMARY KEY CLUSTERED (OFFICE_LOCATION_ID)
)
go
IF OBJECT_ID('OFFICE_LOCATION') IS NOT NULL
   PRINT '<<< CREATED TABLE OFFICE_LOCATION >>>'
ELSE
   PRINT '<<< FAILED CREATING TABLE OFFICE_LOCATION >>>'
```

```
go

--
-- Foreign Keys
--
ALTER TABLE BROKER
   ADD CONSTRAINT BROKER_OFFICE_LOCATION
   FOREIGN KEY (OFFICE_LOCATION_ID)
   REFERENCES OFFICE_LOCATION (OFFICE_LOCATION_ID)
   ON DELETE CASCADE
go

--
--
-- CREATE Indexes
--
CREATE NONCLUSTERED INDEX BROKER_LOCATION
   ON BROKER(OFFICE_LOCATION_ID)
go
IF EXISTS (SELECT * FROM sysindexes WHERE id=OBJECT_ID('BROKER') AND name='BROKER_LOCATION')
   PRINT '<<< CREATED INDEX BROKER.BROKER_LOCATION >>>'
ELSE
   PRINT '<<< FAILED CREATING INDEX BROKER.BROKER_LOCATION >>>'
go
```

Sample MySQL Schema Generated from SQL Server

The following MySQL database was generated from the sample SQL Server schema:

```
DROP TABLE IF EXISTS gim_dbo.BROKER;
CREATE TABLE  gim_dbo.BROKER (
  BROKER_ID decimal(18,0) NOT NULL,
  OFFICE_LOCATION_ID decimal(18,0) default NULL,
  BROKER_LAST_NAME varchar(40) NOT NULL,
  BROKER_FIRST_NAME varchar(20) NOT NULL,
  BROKER_MIDDLE_INITIAL char(1) default NULL,
  MANAGER_ID decimal(18,0) default NULL,
  YEARS_WITH_FIRM decimal(3,1) NOT NULL,
  PRIMARY KEY (BROKER_ID),
  KEY BROKER_LOCATION (OFFICE_LOCATION_ID),
  CONSTRAINT BROKER_OFFICE_LOCATION FOREIGN KEY (OFFICE_LOCATION_ID) REFERENCES OFFICE_LOCATION
(OFFICE_LOCATION_ID) ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS gim_dbo.CLIENT;
CREATE TABLE  gim_dbo.CLIENT (
  CLIENT_ID decimal(18,0) NOT NULL,
  CLIENT_FIRST_NAME varchar(20) NOT NULL,
  CLIENT_LAST_NAME varchar(40) NOT NULL,
  CLIENT_GENDER char(1) NOT NULL,
  CLIENT_YEAR_OF_BIRTH decimal(4,0) NOT NULL,
  CLIENT_MARITAL_STATUS varchar(20) default NULL,
  CLIENT_STREET_ADDRESS varchar(40) NOT NULL,
  CLIENT_POSTAL_CODE varchar(10) NOT NULL,
  CLIENT_CITY varchar(30) NOT NULL,
  CLIENT_STATE_PROVINCE varchar(40) NOT NULL,
  CLIENT_PHONE_NUMBER varchar(25) NOT NULL,
  CLIENT_HOUSEHOLD_INCOME decimal(30,0) default NULL,
  CLIENT_COUNTRY varchar(40) default NULL,
  BROKER_ID decimal(18,0) NOT NULL,
  PRIMARY KEY (CLIENT_ID),
  KEY CLIENT_BROKER (BROKER_ID),
  CONSTRAINT FK__CLIENT__BROKER_I__0F975522 FOREIGN KEY (BROKER_ID) REFERENCES BROKER (BROKER_ID)
```

```
ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS gim_dbo.CLIENT_TRANSACTION;
CREATE TABLE  gim_dbo.CLIENT_TRANSACTION (
  CLIENT_TRANSACTION_ID decimal(18,0) NOT NULL,
  CLIENT_ID decimal(18,0) NOT NULL,
  INVESTMENT_ID decimal(18,0) NOT NULL,
  ACTION varchar(10) NOT NULL,
  PRICE decimal(12,2) NOT NULL,
  NUMBER_OF_UNITS decimal(18,0) NOT NULL,
  TRANSACTION_STATUS varchar(10) NOT NULL,
  TRANSACTION_SUB_TIMESTAMP datetime NOT NULL,
  TRANSACTION_COMP_TIMESTAMP datetime NOT NULL,
  DESCRIPTION varchar(200) default NULL,
  BROKER_ID decimal(18,0) default NULL,
  BROKER_COMMISSION decimal(10,2) default NULL,
  PRIMARY KEY (CLIENT_TRANSACTION_ID),
  KEY CLIENT_TRANSACTION_BROKER (BROKER_ID),
  KEY CLIENT_TRANSACTION_CLIENT (CLIENT_ID),
  KEY CLIENT_TRANSACTION_INVESTMENT (INVESTMENT_ID),
  CONSTRAINT CLIENT_TRANSACTION_BROKER FOREIGN KEY (BROKER_ID) REFERENCES BROKER (BROKER_ID) ON
DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT CLIENT_TRANSACTION_INVESTMENT FOREIGN KEY (INVESTMENT_ID) REFERENCES INVESTMENT
(INVESTMENT_ID) ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT CLIENT_TRANSACTION_CLIENT FOREIGN KEY (CLIENT_ID) REFERENCES CLIENT (CLIENT_ID) ON
DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS gim_dbo.INVESTMENT;
CREATE TABLE  gim_dbo.INVESTMENT (
  INVESTMENT_ID decimal(18,0) NOT NULL,
  INVESTMENT_TYPE_ID decimal(18,0) default NULL,
  INVESTMENT_VENDOR varchar(30) default NULL,
  INVESTMENT_NAME varchar(200) default NULL,
  INVESTMENT_UNIT varchar(20) default NULL,
  INVESTMENT_DURATION varchar(10) default NULL,
  PRIMARY KEY (INVESTMENT_ID),
  KEY INVESTMENT_INVESTMENT_TYPE (INVESTMENT_TYPE_ID),
  CONSTRAINT FK__INVESTMEN__INVES__108B795B FOREIGN KEY (INVESTMENT_TYPE_ID) REFERENCES
INVESTMENT_TYPE (INVESTMENT_TYPE_ID) ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS gim_dbo.INVESTMENT_TYPE;
CREATE TABLE  gim_dbo.INVESTMENT_TYPE (
  INVESTMENT_TYPE_ID decimal(18,0) NOT NULL,
  INVESTMENT_TYPE_NAME varchar(30) NOT NULL,
  PRIMARY KEY (INVESTMENT_TYPE_ID)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS gim_dbo.OFFICE_LOCATION;
CREATE TABLE  gim_dbo.OFFICE_LOCATION (
  OFFICE_LOCATION_ID decimal(18,0) NOT NULL,
  OFFICE_NAME varchar(20) NOT NULL,
  OFFICE_ADDRESS varchar(50) NOT NULL,
  OFFICE_CITY varchar(30) NOT NULL,
  OFFICE_STATE_PROVINCE varchar(40) NOT NULL,
  OFFICE_POSTAL_CODE varchar(10) NOT NULL,
  OFFICE_COUNTRY varchar(40) default NULL,
  PRIMARY KEY (OFFICE_LOCATION_ID)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Sample Code Migration

As was mentioned, no automatic conversion exists in terms of migrating SQL Server T-SQL code objects to MySQL, although a prior Appendix in this paper does supply some

guidance and a simple Perl conversion utility. MySQL is in discussions with several partners on providing this functionality, but until a mechanized solution is reached, there are several options open to performing code migration:

- Contact MySQL Professional Services group to help with the code migration. Staff at MySQL have assisted in code conversions from a number of platforms to MySQL.
- Perform code migrations internally.

The difficulty level of migrating SQL Server T-SQL code to MySQL depends on the actual code itself. For example, the following SQL Server T-SQL procedure uses the sample schema presented above to create various business intelligence reports to the staff of the company using the database:

```
SET QUOTED_IDENTIFIER OFF
go
SET ANSI_NULLS OFF
go
CREATE PROCEDURE CORPORATE_ANALYSIS
   @START_DATE DATETIME,
   @END_DATE   DATETIME
AS
      BEGIN

      --
      -- display brokers ordered by highest commissions for time period
      --
      SELECT  A.BROKER_ID,
            A.BROKER_FIRST_NAME,
            A.BROKER_LAST_NAME,
            SUM(BROKER_COMMISSION) TOTAL_COMMISSIONS
      FROM    BROKER A,
            CLIENT_TRANSACTION B
      WHERE   B.TRANSACTION_COMP_TIMESTAMP BETWEEN @START_DATE AND
@END_DATE AND
            A.BROKER_ID = B.BROKER_ID
      GROUP BY A.BROKER_ID,
            A.BROKER_FIRST_NAME,
            A.BROKER_LAST_NAME
      ORDER BY 4 DESC

      --
      -- display offices ordered by highest commissions for time period
      --
       SELECT  C.OFFICE_NAME,
            SUM(BROKER_COMMISSION) TOTAL_COMMISSIONS
      FROM    BROKER A,
            CLIENT_TRANSACTION B,
            OFFICE_LOCATION C
      WHERE   B.TRANSACTION_COMP_TIMESTAMP BETWEEN @START_DATE AND
@END_DATE AND
            A.BROKER_ID = B.BROKER_ID AND
            A.OFFICE_LOCATION_ID = C.OFFICE_LOCATION_ID
      GROUP BY C.OFFICE_NAME
      ORDER BY 2 DESC

      --
      -- display top 20 invests ordered by highest invested dollars for time period
      --
      SELECT  TOP 20
```

```
            B.INVESTMENT_VENDOR,
            B.INVESTMENT_NAME,
            SUM(PRICE) * SUM(NUMBER_OF_UNITS) TOTAL_INVESTED_DOLLARS
      FROM    CLIENT_TRANSACTION A,
            INVESTMENT B
      WHERE   A.TRANSACTION_COMP_TIMESTAMP BETWEEN @START_DATE AND
@END_DATE AND
            B.INVESTMENT_ID = A.INVESTMENT_ID AND
            A.ACTION = 'BUY'
      GROUP BY B.INVESTMENT_VENDOR,
            B.INVESTMENT_NAME
      ORDER BY 3 DESC

      --
      -- display top invest types ordered by highest invested $$ for time period
      --

      SELECT  C.INVESTMENT_TYPE_NAME,
            SUM(PRICE) * SUM(NUMBER_OF_UNITS) TOTAL_INVESTED_DOLLARS
      FROM    CLIENT_TRANSACTION A,
            INVESTMENT B,
            INVESTMENT_TYPE C
      WHERE   A.TRANSACTION_COMP_TIMESTAMP BETWEEN @START_DATE AND
@END_DATE AND
            B.INVESTMENT_ID = A.INVESTMENT_ID AND
            C.INVESTMENT_TYPE_ID = B.INVESTMENT_TYPE_ID AND
            A.ACTION = 'BUY'
      GROUP BY C.INVESTMENT_TYPE_NAME
      ORDER BY 2 DESC

       --
      -- display top 20 clients ordered by highest invested dollars for time period
      --
      SELECT  TOP 20
            B.CLIENT_FIRST_NAME,
            B.CLIENT_LAST_NAME,
            SUM(PRICE) * SUM(NUMBER_OF_UNITS) TOTAL_INVESTED_DOLLARS
      FROM    CLIENT_TRANSACTION A,
            CLIENT B
      WHERE   A.TRANSACTION_COMP_TIMESTAMP BETWEEN @START_DATE AND
@END_DATE AND
            B.CLIENT_ID = A.CLIENT_ID AND
            A.ACTION = 'BUY'
      GROUP BY B.CLIENT_FIRST_NAME,
            B.CLIENT_LAST_NAME
      ORDER BY 3 DESC

      END
go
IF OBJECT_ID('CORPORATE_ANALYSIS') IS NOT NULL
   PRINT '<<< CREATED PROCEDURE CORPORATE_ANALYSIS >>>'
ELSE
   PRINT '<<< FAILED CREATING PROCEDURE CORPORATE_ANALYSIS >>>'
go
SET ANSI_NULLS OFF
go
SET QUOTED_IDENTIFIER OFF
go
```

The SQL Server T-SQL procedure can be translated to MySQL as follows:

```
CREATE PROCEDURE CORPORATE_ANALYSIS
(IN START_DATE DATETIME,
 IN END_DATE   DATETIME)

      BEGIN

      /*
      display brokers ordered by highest commissions for time period
      */
      SELECT  A.BROKER_ID,
            A.BROKER_FIRST_NAME,
            A.BROKER_LAST_NAME,
            SUM(BROKER_COMMISSION) TOTAL_COMMISSIONS
      FROM    BROKER A,
            CLIENT_TRANSACTION B
      WHERE   B.TRANSACTION_COMP_TIMESTAMP BETWEEN START_DATE AND
END_DATE AND
            A.BROKER_ID = B.BROKER_ID
      GROUP BY A.BROKER_ID,
            A.BROKER_FIRST_NAME,
            A.BROKER_LAST_NAME
      ORDER BY 4;

      /*
      -- display offices ordered by highest commissions for time period
      */
       SELECT  C.OFFICE_NAME,
            SUM(BROKER_COMMISSION) TOTAL_COMMISSIONS
      FROM    BROKER A,
            CLIENT_TRANSACTION B,
            OFFICE_LOCATION C
      WHERE   B.TRANSACTION_COMP_TIMESTAMP BETWEEN START_DATE AND
END_DATE AND
            A.BROKER_ID = B.BROKER_ID AND
            A.OFFICE_LOCATION_ID = C.OFFICE_LOCATION_ID
      GROUP BY C.OFFICE_NAME
      ORDER BY 2 DESC;

      /*
      -- display top 20 invests ordered by highest invested dollars for time period
      */
      SELECT  B.INVESTMENT_VENDOR,
            B.INVESTMENT_NAME,
            SUM(PRICE) * SUM(NUMBER_OF_UNITS) TOTAL_INVESTED_DOLLARS
      FROM    CLIENT_TRANSACTION A,
            INVESTMENT B
      WHERE   A.TRANSACTION_COMP_TIMESTAMP BETWEEN START_DATE AND
END_DATE AND
            B.INVESTMENT_ID = A.INVESTMENT_ID AND
            A.ACTION = 'BUY'
      GROUP BY B.INVESTMENT_VENDOR,
            B.INVESTMENT_NAME
      ORDER BY 3 DESC
      LIMIT 20;

      /*
      -- display top types ordered by highest invested dollars for time period
      */
```

```
    SELECT  C.INVESTMENT_TYPE_NAME,
            SUM(PRICE) * SUM(NUMBER_OF_UNITS) TOTAL_INVESTED_DOLLARS
    FROM    CLIENT_TRANSACTION A,
            INVESTMENT B,
            INVESTMENT_TYPE C
    WHERE   A.TRANSACTION_COMP_TIMESTAMP BETWEEN START_DATE AND
END_DATE AND
            B.INVESTMENT_ID = A.INVESTMENT_ID AND
            C.INVESTMENT_TYPE_ID = B.INVESTMENT_TYPE_ID AND
            A.ACTION = 'BUY'
    GROUP BY C.INVESTMENT_TYPE_NAME
    ORDER BY 2 DESC;

    /*
    -- display top 20 clients ordered by highest invested dollars for time period
    */
    SELECT  B.CLIENT_FIRST_NAME,
            B.CLIENT_LAST_NAME,
            SUM(PRICE) * SUM(NUMBER_OF_UNITS) TOTAL_INVESTED_DOLLARS
    FROM    CLIENT_TRANSACTION A,
            CLIENT B
    WHERE   A.TRANSACTION_COMP_TIMESTAMP BETWEEN START_DATE AND
END_DATE AND
            B.CLIENT_ID = A.CLIENT_ID AND
            A.ACTION = 'BUY'
    GROUP BY B.CLIENT_FIRST_NAME,
            B.CLIENT_LAST_NAME
    ORDER BY 3 DESC
    LIMIT 20;

    END;
```