

Database Independence! Myth or Reality?

Edward K. Yu (ekyu@asg.sc.edu)

Senior Solutions Architect, Advanced Solutions Group
University of South Carolina

5/2/2003

Table of Contents

Table of Contents.....2
Database Independence! Myth or Reality?3
RDBMS - The lowest denominator3
 Database accessibility standards 3
 Transactional behavior..... 4
 Database view support using SQL..... 4
 Common data type and conversions 4
 Surrogate key generation methodology 5
 Database stored procedures/functions..... 5
 Database triggers..... 5
 Performance and Database specific features..... 5
Conclusions.....6
About the Author6
Additional Resources.....6

Database Independence! Myth or Reality?

One of the goals of software development is to create software that could potentially run on many different computing platforms. With the rising popularity of Java, we are coming close to achieving this goal. Unfortunately, most database applications are tied to a specific Relational Database Management System (RDBMS) even if it is written in Java using the Java Database Connectivity specification (JDBC). This is due to the fact that different RDBMS vendors created their own extensions to the ANSI SQL standard to handle primary key generation, database triggers and stored package programs (procedures and functions), etc. In addition to the SQL differences, transactional (records management) behaviors of different RDBMSs are usually incompatible (e.g. trigger firing scope). These differences created a huge challenge for system developers to create software that should run on different databases. In this article, we will explore how to architect and implement a software system using Java and JDBC that could potentially work with many backend RDBMSs.

RDBMS - The lowest denominator

While it is impossible to develop a software system that runs on all computing platforms (while Java comes close, it has its fair share of problems), it is not impossible to create software systems that run on multiple RDBMSs. Thanks to industry recognized standards such as ANSI SQL 92 (SQL92), Java 2 Enterprise Edition (J2EE), it is possible to develop software systems that conform to them to achieve compatibility. The key to developing database software systems that can run on multiple database systems from different vendors is through software architecture layering. When designing a software system, the database access should be encapsulated into its own layer (package) such that when we deploy the system using a different database vendor, we simply replace this database access layer for the new target database (either through automatic code generation or through manual re-coding). In addition, the target database must exhibit certain behaviors such that your application data store within the database can be created to look identical to your application even if it is on different operating systems and different RDBMS. The following are some considerations when developing database software applications that should potentially work for multiple databases (the lowest denominator):

- a) database accessibility standards
- b) transactional behavior
- c) database view support using SQL
- d) common data type and conversions
- e) surrogate key generation methodology
- f) database stored procedures/functions
- g) database triggers

If you develop your application using the above guidelines, the resulting software application will probably be compatible among a number of different databases.

Database accessibility standards

It is very important for your target database to support industry standards such as Open Database Connectivity (ODBC), Call Level Interface (CLI), and Java Database Connectivity (JDBC) to name a few. Supporting these standards means that you have a lot of off-the-shelf software that is available to you to maintain and manage your database system. Coding around these standards allows you to potentially deploy your code to other databases that supports these standards without too much change.

Transactional behavior

Transactional behavior is very important to any database application. It protects data integrity by using ‘2-phase commit’ such that all related transactions can be committed all at once or rolled back all at once. The majority of databases have good transactional support to protect data integrity. Coding around transactions is crucial since the industry is moving towards a distributed approach to databases and making sure all of your data changes are committed within a single transaction prevents it (the data) from going into an inconsistency state because of unexpected conditions such as temporary network outage (including timeout due to heavy network load).

Database view support using SQL

It is crucial for the target database to support views using SQL. Views are frequently used to control what a database user sees by providing a subset of columns from the original table or tables. Some database engines allow updateable views such that your application can insert or update directly against a database view, potentially joined with multiple tables. In addition, views can be used as another abstraction layer so that you can generate a common “view” to your data using database specific SQL with conversion functions to deal with data column formatting. This technique allows your application to maintain a different set of SQL scripts for each database, yet your application sees the underlying data in a uniform manner. One important thing to note is that some databases allow an *order by* clause within the view definition. Your application should not count on this feature and should explicitly order the query result dataset in your application to ensure maximum compatibility across different databases.

Common data type and conversions

The first rule to develop a cross database application is to properly choose your data types. Use common data types (char, varchar, integer, float, numeric, timestamp, date, time, etc.) and avoid specialized data types (money, bit, etc.). This ensures your application can properly deal with the underlying data types without relying on a specific database engine.

There are some issues when dealing with timestamp, date and time data types and their conversions. Some databases allow arbitrary date conversion using functions, some use the locale (setup on the database server) to determine the proper datetime format and some databases use a combination of the two. Unfortunately, there is not a consistent way how a database engine handles datetime column and its conversion. By isolating database access code into its own layer, we can change how we deal with datetime column formatting when inserting, updating and querying the underlying database table. One other technique when dealing with datetime conversion is to create a database view to properly format a datetime column so that it looks the same to your application even if the underlying database engine changes.

Another data type related issue is the empty string and null value equivalence on character (char, varchar, etc.) columns and non-character (integer, date, etc.) columns. Some databases convert empty string (‘’) to a null value when inserted or updated against a character type table column (e.g. Oracle), while others actually put an empty string (‘’) with zero length into the character type table column (e.g. Postgresql, DB2, MySQL, MSSQL). This affects the query ‘where clause’ in which the former only requires the ‘where clause’ to be ‘where column is null’, and the latter requires the ‘where clause’ to be ‘where column is null or length(column) <= 0’. Your database access layer should be developed to handle this in a uniform manner to achieve database independence.

Along the same line is the issue that the empty string and null value equivalence on non-character (integer, float, date, etc.). Similar to character columns, some databases actually put a null value into the non-character database column (Oracle) some actually translate the empty string into a default value (Postgresql, MSSQL, MySQL), and some databases will return an error when this situation occurs (DB2). Your database access layer should be developed to handle this in a uniform manner to achieve database independence.

Finally, some database engines support “object tables” to be created with some form of inheritance mechanism. Unfortunately, these object oriented database features do not have a widely adopted standard to ensure

compatibility. As a result, applications using these object oriented features of relational databases are pretty much database specific and cannot be ported to different database engines.

Surrogate key generation methodology

One of the most important aspects of database application programming is to automatically generate unique primary key column values for your database records. A majority of databases provide sequence object support to allow the generation of unique numbers used as primary key column values (e.g. Oracle, DB2, Postgresql). Some databases support the generation of a unique binary value to achieve the same goal (MSSQL), while others only support this using some kind of a column auto increment facility (MySQL).

It is not advisable to use auto increment facility to acquire surrogate key values since it usually has limitations (e.g. for MSSQL only a single auto id column is allowed per table and it is not updateable). Sometimes, using binary primary column values is not very efficient, especially when joining multiple tables (due to the size of the binary column value, 16 bytes for MSSQL).

In order to create a cross database application, you must develop a scheme to request sequence values for your database records. It is very important that your sequence value management must take into consideration that multiple clients could potentially grab the same sequence value at the exact same time. As a result, your application might error out if your database enforces primary key (or unique index) constraints on tables.

Database stored procedures/functions

Most of today's relational databases have the ability to store executable programs within the database. These database stored programs (whether it is in a vendor specific database procedural language or any standard programming language such as C or Java) are very efficient when dealing with multiple database table changes. Rather than bringing multiple database records back to the client, modifying them and committing the changes back to the database, all the application code has to do is to invoke a database stored program with a set of parameters related to the operation and the database stored program can take care of all the related transactions without requiring that any data go across the network. This is a very efficient way to program transactions involving multiple tables. In addition, side benefits includes the ability to grant execution privileges to the proper database role (or user) to improve security, facilitate access logging within stored programs and better structure your business logic.

Database stored programs also act as another abstraction layer such that your application can have a different set of stored programs for each database. As long as the stored program call API (the number of parameters and the types) remains the same, your application code can expect the same behavior from the database and remain unchanged.

Database triggers

Modern relational databases usually allow small pieces of code to be executed when a database table record is inserted/updated or deleted. These code fragments are referred to as 'triggers'. Triggers are very useful to detect database table changes and carry out additional database transactions. They can also be thought of as yet another database abstraction layer since you can have a different set of triggers for different database engines. One important thing to note is that some databases support row level triggers (Oracle) and some only supports table level triggers (MSSQL). In addition, some database engines actually report multiple record counts to the application when a transaction is executed because triggers are attached to database tables involved in the transaction (MSSQL).

Performance and Database specific features

You may have noticed that so far we have not yet addressed the issue of performance. Most database engines have their own SQL extensions (optimizer hints) to affect the query optimizer (rule based or cost based) to take different

paths to the data. Others provide a system level table allowing you to modify gathered statistics. Again, there is not a single way to handle hinting the optimizer among different database engines. If you are using SQL views (as an abstraction layer) as described above, you can embed optimizer hints into SQL statements at the script level (not within the application). With the help of indices (or indexes), you should be able to tune the database performance at the SQL level for different databases. Since these techniques are all database related, we can encapsulate them all in the SQL layer for different databases. The most important point for building applications that are database neutral is to encapsulate database specific features at the SQL script level so that you can tune them for a specific database engine and yet reuse the application without change. This also applies to special database features that you may want to use (rollup or cube functions that are available among advanced databases to support data warehousing).

Conclusions

In this article, we have discussed the important guidelines to design and develop database application systems that can be portable among different databases. We've looked at the "lowest denominator" of database features that can be supported across different database engines. We've looked at SQL standards, data type and their conversions; we've covered database transaction support; we've looked at primary key value generation mechanisms; we've talked about architectural layering of database access code and different techniques that form this database access layer (SQL scripts, views, stored programs and database triggers). Following the above guidelines and with the help of automated code generation tools, we can write database applications that can be ported among different databases.

About the Author

Ed Yu (ekyu@asg.sc.edu) is currently working with the Advanced Solutions Group at the University of South Carolina and has 20+ years of industry experience. His primary job focus includes information system consultations, design, development and deployment (hardware/software). In addition, he establishes technical system design and implementation guidelines within the organization to better support its primary business functions and to better serve clients in the private industries (oil industry, insurance industry, etc.) and public sectors (local, state and federal government agencies in criminal justice, law enforcement and education, etc.). In addition, he also mentors junior programmers on various different projects. One of his recent projects was to develop a database Object/Relational Mapper that works across all major databases supported by JDBC 2.0.

Additional Resources

The following are various open source/commercial database projects.

ANSI – <http://www.ansi.org>

Postgresql – <http://postgresql.org>

MySQL – <http://mysql.com>

SapDB – <http://www.sapdb.org>

The following are various commercial databases (their trademarks and copyrights).

Oracle – <http://oracle.com>

DB2 (UDB) – <http://www.ibm.com>

Microsoft SQL Server – <http://www.microsoft.com>

Java/JDBC/J2EE – <http://java.sun.com>