# dotNetRDF Design Document

## dotNetRDF 0.4.3

Library: dotNetRDF.dll  
Version: 0.4.3  
Target Date: August 2011  
Author: Rob Vesse  
Proposed Implementer: Rob Vesse  
Last Updated:

## Required Features

* Reworked SQL backend support

## Time Permitting Features

* Virtualised Triples and Nodes

## Known Issues/Bugs to Fix

* None at present

# Design

## Reworked SQL backend support

### Database Schema

The new database schema will be as similar as possible to the old schema to ease migration. Main proposed changes are the following:

* Remove all the hash fields as these reduce the data portability and with better server side indexing and use of stored procedures will become defunct
* Add an additional field to NODES table for storage of Language Specifier/Datatype for Literals
* Remove the GRAPH\_TRIPLES table and instead just add a graph field to the TRIPLES table. While this increases storage requirements slightly it will make for faster IO operations on Triples as 1 less insert/delete is required to add/remove a Triple. This will also reduce the amount of defunct triples that remain in the database. Plus it makes it easier to compact the database by removing unused nodes since now only in-use nodes will be mentioned in the TRIPLES table

### General Implementation

While ideally we should remove the defunct ISqlIOManager interface and all existing implementations and related classes in practise this will not be immediately possible. Obsolete messages should be changed to indicate the classes are legacy and no longer fully support and **will** be removed at a future date TBC.

Design a new abstract base implementation that is a general ADO.Net implementation. Define the protected abstract implementation in terms of methods that get connections, commands, DataTable and DataReader objects. New implementations should have new class names to distinguish them from the old implementations, old implementations must detect when a user attempts to use them with an old format database and throw an error.

To distinguish the old and new store the new store will be referred to as the ADO Store since it is reliant only on ADO and stored procedures and not directly on SQL like the existing store.

Define a new abstract implementation deriving from this ADO.Net implementation that is heavily based on stored procedures i.e. offload much of the heavy lifting of insertion onto the database server. Proposed procedures:

* CreateGraph
* DeleteGraph
* AssertTriple
* RetractTriple
* GetGraphTriples
* GetTriplesWith – all variants
* GetNode – gets the actual values needed to create a node based on an ID

Then define implementations of this for both Microsoft SQL Server and MySQL. Test against both local SQL Server instances and against SQL Server Azure (to see whether it can fulfil the Microsoft use case for which dotNetRDF is being considered).

### Specialised Implemention

If the time permitting virtualised triple and nodes feature is both implemented and deemed viable then allow the implementation to operate in either materialised/virtual mode. In materialised mode retrieving Triples gets the full Nodes values for all Nodes. In virtual mode it takes advantage of the new virtualised storage implementation so that only IDs are retrieved and the values only get retrieved when needed.

Ensure that appropriate caching is used at the API end to avoid both unnecessary trips to the database and so that when operating in virtual mode a Node ID always returns the same virtual node instance (as far as possible).

### Upgrade Path

Provide an upgrade tool or path (separately to the core library) that allows existing stores to be cleanly upgraded to new stores. Best way to do this is probably to have an import script that can read the legacy database format and import into a new clean database. This is better than trying to upgrade the database in place as it avoids any issues with destroying existing data

## Virtualised Triples and Nodes

Define new INode implementations that are entirely virtual, they should be generic where the type parameter is the type of the ID that they use to refer to nodes and graphs in their underlying storage. These implementations **must not** inherit from the classes they are virtualising as they must have explicit casts available that let them be cast to the appropriate type. Once a cast has happened an implementation should cache the cast of itself internally so it can be reused for future casts.

Define a generic interface IVirtualisedRdfProvider (probably needs a better name?) which all the above must be instantiated with an instance of. The virtual INode implementations will then use this interface to load their actual values when required.

*Note – It may be better not to define a general purpose version of this and instead just create ADO store specific node implementations. The main benefit of these will be the ability to do equality and comparison based on database IDs wherever possible which with actual values be lazily loaded as late as possible.*