Inside DODS

Inside DODS

Together Teamlösungen EDV-Dienstleistungen GmbH

Elmargasse 2-4
A-1190
Vienna
Austria
+43 (0) 5 04 04 - 122
+43 (0) 5 04 04 - 11 122
<office@together.at>
http://www.together.at/together/index.html

Table of Contents

1. Introduction 1
    Conventions used in this book 1
2. Simple Access 2
    Select statement 2
    Insert statement 2
    Update statement 3
    Delete statement 3
3. Lazy Loading 4
    Select statement with Lazyloading 4
    Further optimization ("pointer lazy loading") 4
4. Cache Transformation 6
5. Caching 8
    Table configuration 9
    Cache configuration 10
    Table and cache statistics 13
6. Data Caching 17
    Select statement 17
    Insert statement 17
    Update statement 17
    Delete statement 17
7. Query Caching 18
    Select statement 18
    Insert statement 18
    Update statement 19
    Delete statement 19
8. Caching And Lazy Loading 20
    Overview 20
    Select statement with Lazyloading and Caching 20
9. Security 21
    Select statement 25
    Lazy Loading 27
10. unique attribute 28
11. maxDBrows attribute 29
12. databaseLimit attribute 30
13. databaseLimitExceeded attribute 31
14. readSkip attribute 32
15. Read-only per Table 33
16. Global Read-only 34
17. Delete cascade 35
18. Multi Database Support 36
19. Fetch size 38
20. Cache Initialization 39
21. Reserve factor 41
22. New Parameters in Configuration and DOML Files 43
    TransactionCheck 43
    DirtyDOs 44
    DeleteCheckVersion 44
    AutoWrite 44
    TransactionCaches 45
    AutoSave 45
    AutoSaveCreateVirgin 46

DefaultFetchSize 46
MainCacheLockTimeout 47
CacheLockTimeout 47
CacheLockRetryCount 47
QueryTimeout 48
SelectOids 48
IncrementVersions 48
MaxConnectionUsages 49
MaxWaitingConnections 49
initAllCaches 49
ChangeAutocommit 50
MassUpdates and MassDeletes 50
UseCursorName 50
CaseSensitive 51
ClassList 51
ClassType 51
TransactionFactory 52
FullCacheCountLimit 52
InitialDSCacheSize 52
InitialCacheFetchSize 53
InitCachesResultSetType 53
InitCachesResultSetConcurrency 53
DisableConnectionPool 53
DataSourceName 54
ConnectionFactory 54
ConnectionAllocator 55
ConnectionIdleTimeout 55
RollbackOnReset 56
QueryCacheImplClass 56
SQLBatch 56
CaseInsensitiveDatabase 57
ShutDownString 57
NextWithPrefix 57
OidTableName 57
NextColumnName 58
NextColumnType 58
QueryTimeLimit 58
PrimaryLogicalDatabase 58
DODSCacheFactory 59
ConnectionAllocateCheckSql 59
MaxPreparedStatements 60
AllocationScope 60
ObjectIdAllocationSource 60
asynchLoadThreadNum 61
asynchLoadPriority 61
ClassName 61
JTA 62
cachePersistenceRoot 62
cachePersistenceOnShutdown 62
cachePersistenceOnDisableCaching 63
cursorless 63
defaultMaxRows 63
23. Database Vendor and Driver Specific Parameters 65
UseCursorName 65

SplitSQLPrimary 66
OrderedResultSet 66
DisableFetchSizeWithMaxRows 66
ResultSetType 67
ResultSetConcurrency 67
WildcardEscapeClause 67
SetNullAsVarchar 67
SetBytesAsLongvarbinary 68
CustomNotEqualSqlOperator 68
SetBytesAsBinaryStream 68
SetBooleanAsString 68
UsePrefixWithUpdate 69
EnableCreateStatistics 69
CreateStatistics 69
NamedStatistics 70
FullColumnNames 70
SupportAttribs 70
EndString 70
IncludeIndexColumns 70
DriverDependenciesClass 71
UseTopSyntax 71
24. Transactions 72
DOs in Transactions 72
Status of DOs 72
Creating DOs 72
Using DOs 73
Save and Delete Operations in Transactions 73
Insert, Update and Delete Operations on the Database 73
Queries 74
Caching 75
Commit of Transactions 75
Extended Trasaction 77
First attempt of using JTA API Implementation in DODS 77
Current JTA API Implementation in DODS 79
25. Mass Modifications 83
DODS's duality 83
Generated classes 83
26. Using database generated identity columns in DODS 85
27. Using "OID per Table" feature in DODS. 87
28. Statistics 88
Creating statistics statements 88
29. Additional coulumns in index 89
Including columns in index 89
30. Advanced Access 90
Creating union of ResultSets. 90
31. Database Configurations 91
Driver configuration 91
Using DODS with javax.sql.DataSource 91
Oracle 92
Informix 93
Sybase 93
QED 94
MySQL 94
PostgreSQL 95

InstantDB 96
Mckoi 96
P6SPY 97
DB2 97
HSQLDB (HypersonicSQL) 99
Microsoft SQL Server 99
    JTurbo JDBC driver 99
    jTDS JDBC driver 100
    MS-JDBC driver 100
Microsoft Access 101
InterBase 103
    InterClient 103
C-JDBC 105

Chapter 1. Introduction

This document describes the main features of DODS (Data Object Design Studio).

In the first part is described caching in DODS: the types of caching, the caching levels and a lot of features that involve caching.

In the second part are described transactions in DODS: new parameters of configuration and doml files and basics of transactions like the use of DO and Query objects, save and delete operations in transactions, insert, update and delete operations on the database, caching in transactions, the transaction DO cache and write and commit methods of transactions.

Conventions used in this book

The use of parameters with [ ] brackets indicates their optional use (in syntax lines).

For example: [a] means you can choose a or nothing.

Chapter 2. Simple Access

Select statement

For select statements, it is used <table_name>Query.java class. The query is formed using methods of <table_name>Query.java and QueryBuilder.java class. The query is executed with method runQuery() of the <table_name>Query.java class. In this method, the query is executed on the database, and the results are retrieved as a resultSet object. Then, the method uses protected method:

```
createExisting([String logicalDatabase],
ResultSet rs,
[HashMap queryRefs ],
[DBTransaction dbTrans])
```

of the <table_name>DO.java class. This method calls constructor which calls method:

```
initFromResultSet(ResultSet rs)
```

which sets in result DO all columns retrieved from the database. If any column is a reference, first is made referenced object with the method

```
createExisting([String dbName], BigDecimal bd )
```

and then is set DO's attribute to this referenced object.

Insert statement

New DO can be created using <table_name>DO.java class method createVirgin with the one of the parameter combinations:

```
createVirgin([DBTransaction dbTrans])
```

or

```
createVirgin(String dbName)
```

This method creates a DO that has no data set. Such a DO is used to insert a new database entry after its data had been set.

The parameter dbTrans can also be optional, but this option depends on DirtyDOs parameter value.

DirtyDOs is parameter of 'database' and 'table' tag in DOML file. DODS source code generating depends on its value.

If DirtyDOs is set to "Deprecate", methods without dbTrans parameter are deprecated.

If DirtyDOs is set to "Omit" methods without dbTrans parameter are not generated at all.

If DirtyDOs is set to "Compatible", the methods will be generated as before.

Default value for DirtyDOs is "Compatible".

After the DO is created using the method createVirgin, and its data set, method of <table_name>DO.java class:

```
public void save([DBTransaction dbt],[boolean references])
```

is used for inserting DO to the transaction.

Update statement

For updating a DO, are used set methods for table columns of the class <table_name>DO.java. When the DO is updated, the same method is used for updating the database as for inserting:

```
public void save([DBTransaction dbt],[boolean references])
```

When updating is started only those columns that are changed will be updated, and only data for changed columns will be sent to database, reducing dataflow to database server.

When DO is updated, its ‘version’ column is incremented.

Version of DO object is incremented only when the update of DO is executed.

Delete statement

To delete a DO, use method of the <table_name>DO.java class:

```
public void delete([DBTransaction dbt])
```

This method deletes the DO from its table in the database.

Chapter 3. Lazy Loading

The lazyLoading attribute is one of the table (TableConfiguration object) attributes. The object tableConfiguration can be retrieved from <table_name>DO.java class by calling method:

```
getConfigurationAdministration().getTableConfiguration()
```

This methods returns org.enhydra.dods.cache.TableConfiguration object. The lazyLoading attribute can be handled with the following methods of TableConfiguration object:

- isLazyLoading() - Returns current value of lazyLoading attribute.

Lazy loading is a mechanism that postpones the loading of DO's data (DataStruct objects) until they are needed. This mechanism is used in the protected method:

```
createExisting(ObjectId id)
```

or the same method with the combinations of parameters:

```
createExisting( String dbName, ObjectId id, [HashMap refs])
```

or

```
createExisting( ObjectId id, [HashMap refs], [DBTransaction dbTrans])
```

This method creates a DO with specified Oid (DO without data), and than, if it is not lazy loading, calls method

```
loadData()
```

to load the fields for the DO from the database.

Select statement with Lazyloading

For, example, DODS uses this method in select clauses (runQuery() method) when are created referenced objects - when is used method

```
createExisting( [String dbName], BigDecimal bd )
```

which calls mentioned method createExisting with the ObjectId parameter and combination of dbName,refs and dbTrans parameters. So, if lazy loading is used, we always select T.oid and create DO without data. Later if this referenced object is needed data is loaded using method:

```
checkLoad()
```

(using select T.* queries) of <table_name>DO.java class. This method checks whether the DO's original data is loaded, and if it isn't, this method loads colums data and version by calling:

```
loadData()
```

method.

Further optimization ("pointer lazy loading")

While resolving some nasty bugs with transactions and cache, we saw that DataStruct objects cannot know anything about DO references. These references were surviving the end of original transaction that made them, and were reused in another one, which got them from cache.

This situation made us remove all xxxDO references from DataStuct, leaving there BigDecimal instead. Now get/set methods of generated DO object are a bit different, set method stores DO object reference into map that is known to DO only, and extracts OId to store into DataStruct; get method after getting BigDecimal form DataStruct looks up the map for previously used (stored) DOs, then if look up fails creates one (with createExisting). Further optimization represents conditional use of DO map, which isn't used if transaction and its cache is on.

For oid based tables (references - DO objects), now, new public setXXX and getXXX methods are created in DO classes:

```
oid_getXXX() - returns oid value of referenced DO as a BigDecimal
```

```
oid_setXXX(BgDecimal), oid_setXXX(ObjectId), oid_setXXX(String)- sets oid value of referenced
 DO
```

Effect of these change is that application will create only DOs actually used, leaving all other out of a loop.

Chapter 4. Cache Transformation

Since DODS 5.1 final, the DO cache is transformed into DataStruct cache. Instead of the whole DOs, only their original DataStructs are added to new DataStruct cache.

DO has had only one data (DataStruct object) and all transformations were done on this object. DataStruct object contains values of columns of one table row. Now, DO holds 2 DataStruct-references:

- originalData
- data

The originalData holds original data (that was read from the database). This is never modified till commit, and this DataStruct object is added to DataStruct cache, if this cache exists.

The second, data, is only created (by copying the first one) if data is modified. If the second DataStruct exists, the DO's attribute isDirty is set to true. Even if after some modifications the new DataStruct holds exactly the same values as the original one, the DO is still dirty. So there is no way back from isDirty=true to isDirty=false (except during commit of the transaction). If the transaction is committed, the new DataStruct is moved in the place of the original one. The new DataStruct is NULL again, so the attribute isDirty becomes false again.

A newly created DO (in memory, not from the database) will just have a DataStruct object data. Data values in DataStruct object originalData is null before the commit().

The oid and the version attributes are moved from DO to DataStruct object.

New attributes added in DataStruct object are:

- isEmpty

- type: boolean

- default value: true

Since originalData is being constructed for every DO, this flag "knows" if DataStruct has any useful content. If there is no data in DataStructs - except oid, this attribute is true, otherwise false.

- databaseName

- type: String

- default value: null

The logical database to which this DataStruct belongs to.

New methods added in DataStruct object are:

- getOId()

Returns DataStruct's identifier.

- setDatabase(String dbName)

Sets attribute databaseName.

- getDatabase()

Returns attribute databaseName.

- getHandle()

Returns this DataStruct's handle (identifier as a string).

- getCacheHandle()

Returns this DataStruct's cache handle (String in the form: "<database_name>.<indentifier_as_String>").

- get and set methods for every table column

In DO class are added new methods that work with originalData:

- originalData_get<column_name>()

Returns the row value of the column <column_name> of the DO's originalData object.

- originalData_set(Object data)

Sets the DO's originalData object.

- getData()

Returns DO's DataStruct object. If DO's data object exists, returns that object, otherwise returns DO's originalData object.

- originalData_get()

Returns DO's originalData object.

- getOriginalVersion()

Returns the current version of DO's originalData object.

Chapter 5. Caching

DODS provides the possibility for every table to have its cache.

The possible cache types are:

1.None

No caching is available.

2.LRU

The size of the cache is limited by the maximal number of objects that can be stored in it. When the cache is full, the objects in it are being replaced by new objects according to LRU (least recently used) algorithm. This algorithm says that the object which had been used the least recently (in the scale of time, the object to which had been accessed the longest time ago, which is on the end of LRU list) is removed from list and new one is put in front of the LRU list. If maximal number of objects is set to 0, it means that caching is not available (None type) at the moment.

3.Complete

This cache extends HashMap and is unbounded. This cache type is defined by the negative number of maximal cache size.

4.Full (special case of complete cache)

This is a complete cache (HashMap), for which is the entire table queried and cached when the application starts (initial condition is "*"). This is appropriate for tables of "static" data which are accessed frequently.

There is a method, *isComplete()*, in the cache class that checks if the cache (DataStruct cache) is complete or not. If the cache was not complete at the start, it is not checked if it becomes complete or not. But, if the cache was complete, it is than calculated whether the cache is still complete. The method for setting max cache size (in the situation when cache is not null and new cache size is not zero) for DataStruct cache changes cache implementation (form complete to LRU), only if the cache was complete and the new maxCacheSize is positive. In all other cases, the implementation stays as it was.

It is a little bit different with query caches. They don't define the global caching type, so any change form negative to positive max cache size (and vice versa) changes the cache implementation (Complete or LRU).

When any of the caches (DataStruct or any of query caches) is created from scratch, the procedure is the same. Based on max cache size, the proper implementation is used. The same goes for methods for cache refreshing and enabling.

DODS has two levels of caching:

1.Data Caching level

There is only one LRU cache: cache with DataStruct objects. The keys of this cache are cache handles - Strings in the following form:

"<DataStruct_database_name>.<Table_name>.<String_presentation_of_DataStruct_oid>"

and cache values are, as mentioned before, DataStruct objects.

2.Query caching level

Beside DataStruct object cache, there is a possibility of using three query caches (simple, complex and multi-join). Multi-join cache is included since DODS 6.0. All query caches are also LRU caches. The keys of these caches are Strings in the following form:

"<query_database_name>.<String_presentation_of_query>",

and cache values are Query objects. Query objects are objects of the org.enhydra.dods.cache.QueryCacheItem class.

The QueryCacheItem object stores one query and its necessary data:

- Database of the query
- List of oids of DataStruct objects that are results of the query. This list can contain all query results, or just some of them.
- Number of cached query results
- Information whether all results are in result list or not
- Information whether the query results are modified (if there have been performed inserts, updates or deletes, the results are modified)
- Time needed for query execution
- Array of conditions declared in WHERE part of the query (array of org.enhydra.dods.cache.Condition objects). This is needed only for simple queries.
- Queries that are supported by DataStruct cache are simple queries. Simple query is query for which cache mechanisms can determine whether DataStruct object is query result or not. Other queries are complex queries.

The default values for maximal cache sizes for DataStruct, simple and complex query cache are 0 (no caching).

Table configuration

Table configuration is explained on DiscRack example (directory <DODS_HOME>/examples/discrack). The table parameters are defined on three levels.

The first level is DatabaseManager level. On this level can be defined the following parameters (all information are optional):

```
DatabaseManager.defaults.lazyLoading = true
DatabaseManager.defaults.maxExecuteTime = 200
DatabaseManager.defaults.AllReadOnly = false
```

The second level is Database level. On this level can be defined the following parameters (all information are optional):

```
DatabaseManager.DB.<database_name>.lazyLoading = false
DatabaseManager.DB.<database_name>.maxExecuteTime = 350
DatabaseManager.DB.<database_name>.AllReadOnly = false
```

The third level is table level. In the case of DiscRack example, there are two tables: Disc and person. The tables can have the following parameters:

```
#
# Table Disc - table configuration
```

```
# DatabaseManager.DB.DiscRack.Disc.readOnly = false
DatabaseManager.DB.DiscRack.Disc.lazyLoading = false
DatabaseManager.DB.DiscRack.Disc.maxExecuteTime = 150
```

```
#
# Table Person - table configuration
# DatabaseManager.DB.DiscRack.person.readOnly = true
DatabaseManager.DB.DiscRack.person.lazyLoading = false
# DatabaseManager.DB.DiscRack.person.maxExecuteTime = 150
```

Table defaults on DatabaseManager and Database are default values for all application's tables. If any of these parameters is defined on the Database level, that value is used as a default for all tables. If any of the parameters is not defined on the Database level, then, if it is defined on the DatabaseManager level, this value is used. If any of these parameter is not defined neither on the Database, nor on DatabaseManager level, DODS uses its own program defaults. For lazyLoading, program default is false, for maxExecuteTime 0 and for readOnly and AllReadOnly false.

If any of parameters lazyLoading or maxExecuteTime is defined on the table level, that value is used. If not, the default value for all tables is used (explained in previous paragraph).

If parameter AllReadOnly is defined and set to true (it can be defined on DatabaseManager or Database level), all applications will be read-only. In that case, readOnly parameter is ignored. Only, If AllReadOnly is set to true and readOnly attribute of the table is set to false, warning is written to log during table initialization. In runtime exception is thrown on attempt of writing to that table.

Parameter maxExecuteTime is time for query execution. Every query that is executed longer than maxExecuteTime is printed (SQL statement, execution time and maxExecutionTime) in application's log file.

All other parameters are explained later in this document.

Cache configuration

Cache configuration is explained on DiscRack example (directory <DODS_HOME>/examples/discrack). The cache parameters are defined on three levels.

The first level is DatabaseManager level. On this level can be defined the following parameters (all information are optional):

```
# DatabaseManager.defaults.cache.maxCacheSize = 100
DatabaseManager.defaults.cache.maxSimpleCacheSize = 20
DatabaseManager.defaults.cache.maxComplexCacheSize = 5
DatabaseManager.defaults.cache.maxMultiJoinCacheSize = 3
DatabaseManager.defaults.cache.reserveFactor = 0.1
DatabaseManager.defaults.cache.CachePercentage = -1
# DatabaseManager.defaults.cache.initAllCaches = true
DatabaseManager.defaults.cache.asynchLoadThreadNum = 2
DatabaseManager.defaults.cache.simpleCacheRowCountLimit = 300
DatabaseManager.defaults.cache.synchLoadRowCountLimit = 5000
DatabaseManager.defaults.cache.maxExecuteTimeCacheInit = 300
DatabaseManager.defaults.cache.queryTimeoutCacheInit = 10
DatabaseManager.defaults.cache.queryTimeLimitCacheInit = 12000
```

The second level is database level. On this level can be defined the following parameters (all information are optional):

```
DatabaseManager.DB.<database_name>.cache.maxCacheSize = 1100
# DatabaseManager.DB.<database_name>.cache.maxSimpleCacheSize = 10
# DatabaseManager.DB.<database_name>.cache.maxComplexCacheSize = 5
# DatabaseManager.DB.<database_name>.cache.maxMultiJoinCacheSize = 3
DatabaseManager.DB.<database_name>.cache.reserveFactor = 0.1
```

```
DatabaseManager.DB.<database_name>.cache.CachePercentage = -1
DatabaseManager.DB.<database_name>.cache.initAllCaches = true
DatabaseManager.DB.<database_name>.cache.simpleCacheRowCountLimit = 400
DatabaseManager.DB.<database_name>.cache.synchLoadRowCountLimit = 6000
DatabaseManager.DB.<database_name>.cache.maxExecuteTimeCacheInit = 400
DatabaseManager.DB.<database_name>.cache.queryTimeoutCacheInit = 15
DatabaseManager.DB.<database_name>.cache.queryTimeLimitCacheInit = 20000
```

The third level is table level. In the case of DiscRack example, there are two tables: Disc and person. The tables can have the following parameters:

```
#
# Table Disc - cache configuration
# DatabaseManager.DB.DiscRack.Disc.cache.maxCacheSize = 10000
DatabaseManager.DB.DiscRack.Disc.cache.maxSimpleCacheSize = 2000
DatabaseManager.DB.DiscRack.Disc.cache.maxComplexCacheSize = 250
DatabaseManager.DB.DiscRack.Disc.cache.maxMultiJoinCacheSize = 100
DatabaseManager.DB.DiscRack.Disc.cache.reserveFactor = 0.1
DatabaseManager.DB.DiscRack.Disc.cache.CachePercentage = 0.5
DatabaseManager.DB.DiscRack.Disc.cache.asynchLoadPriority = 2
DatabaseManager.DB.DiscRack.Disc.cache.simpleCacheRowCountLimit = 1000
DatabaseManager.DB.DiscRack.Disc.cache.synchLoadRowCountLimit = 10000
DatabaseManager.DB.DiscRack.Disc.cache.maxExecuteTimeCacheInit = 500
DatabaseManager.DB.DiscRack.Disc.cache.queryTimeoutCacheInit = 10
DatabaseManager.DB.DiscRack.Disc.cache.queryTimeLimitCacheInit = 12000
#
# Table Person - cache configuration
# DatabaseManager.DB.DiscRack.person.cache.maxCacheSize = -1
DatabaseManager.DB.DiscRack.person.cache.maxSimpleCacheSize = 2000
DatabaseManager.DB.DiscRack.person.cache.maxComplexCacheSize = 250
DatabaseManager.DB.DiscRack.person.cache.maxMultiJoinCacheSize = 75
DatabaseManager.DB.DiscRack.person.cache.initialCondition = *
DatabaseManager.DB.DiscRack.person.cache.asynchLoadPriority = 1
DatabaseManager.DB.DiscRack.person.cache.simpleCacheRowCountLimit = 100
DatabaseManager.DB.DiscRack.person.cache.synchLoadRowCountLimit = 5000
DatabaseManager.DB.DiscRack.person.cache.maxExecuteTimeCacheInit = 200
DatabaseManager.DB.DiscRack.person.cache.queryTimeoutCacheInit = 5
DatabaseManager.DB.DiscRack.person.cache.queryTimeLimitCacheInit = 7000
```

Cache defaults on DatabaseManager and Database are default values for all application's table caches. If, any of these parameters is defined on the Database level, that value is used as a default for all tables. If any of the parameters is not defined on the Database level, then, if it is defined on the DatabaseManager level, this value is used. If any of these parameters is not defined neither on the Database, nor on DatabaseManager level, DODS uses its own program defaults. For maxCacheSize, maxSimpleCacheSize, maxComplexCacheSize, maxMultiJoinCacheSize, reserveFactor, asynchLoadThreadNum, simpleCacheRowCountLimit, synchLoadRowCountLimit program default value is 0, for CachePercentage is -1.0, for initAllCaches is false and for maxExecuteTimeCacheInit, queryTimeoutCacheInit and queryTimeLimitCacheInit program default is value defined for parameters maxExecuteTime, QueryTimeout and QueryTimeLimit.

If any of table level parameters maxCacheSize, maxSimpleCacheSize, maxComplexCacheSize, maxMultiJoinCacheSize, reserveFactor, CachePercentage, simpleCacheRowCountLimit, synchLoadRowCountLimit, maxExecuteTimeCacheInit, queryTimeoutCacheInit or queryTimeLimitCacheInit is defined on the table level, that value is used. If not, the default value for all tables is used (explained in previous paragraph).

The parameter initialCondition can be defined only on the table level. It contains "where" part of select clause. With this select clause is DataStruct cache of specified table initialized. If initialCondition = '*', the entire table will be added to the DataStruct cache in DataStruct cache initialization. If the parameter is NULL or not defined, no objects are added to the Data cache during the cache initialization.

If, for any table parameter initialCondition is not defined and the initAllCaches parameter is set to 'true' (on DatabaseManager or Database level, as explained before), the default value of initialCondition parameter for the table is "*".

Parameter maxCacheSize contains information about maximal size of DataStruct cache. Parameter maxSimpleCacheSize contains information about maximal size of simple query cache. Parameter maxComplexCacheSize contains information about maximal size of complex query cache. Parameter maxMultiJoinCacheSize contains information about maximal size of multi-join query cache.

Parameter CachePercentage is used for query to make decision what type of query will be executed: select t.* or select t.oid. If no lazy loading and caching is turned on and value of CachePercentage is less then currently used cache (in percents), t1.* is used for query statement. Otherwise select t.oid. Parameter value 0 means use always t1.oid if cache is turned on, -1 (default) means never if not lazyloading but cached. If lazy loading is on always is used t1.oid query.

In <table_name>Query.java class are added new methods:

- setLoadData(boolean newValue)

  If parameter newValue set to true, query select t.* will be executed no matter what are the values of parameters lazyLoading and CachePercentage.

- getLoadData()

  Returns true if query select t.* will be executed, otherwise false.

Reserve factor is constant used in query caching. It is percent of how many more object are taken for evaluation. If num is number of needed results, then it is used

```
num + reserveFactor * num
```

objects for estimating what is quicker: go to database for all object that are not in the cache, or run again query on database. This value is given in percents, as number between 0 and 1 (0.25 means 25%).

For example, if reserveFactor is 0.5, and wanted number of results is 50, the estimation will be done on 75 (50 + 0.5 * 50) objects.

In the following text are explained maximal cache sizes (for DataStruct cache and query caches). The parameters maxCacheSize, maxSimpleCacheSize, maxComplexCacheSize and maxMultiJoinCacheSize of application's configuration file define these sizes.

- maxCacheSize > 0

This cache is limited. The maximal number of elements in the cache is maxCacheSize. This is LRU cache type.

- maxCacheSize = 0

This means that there is no cache available. This value excludes cache from use.

- maxCacheSize < 0

This cache is unlimited. This is complete type of cache (HashMap).

The parameter asynchLoadThreadNum is only defined on DatabaseManager level. This is the number of threads used for asynchronous cache load during application startup. The default value is 0 (asynchronous cache load is not used).

The parameter asynchLoadPriority is only defined on table level. It is the priority of asynchronous cache load for the table. The table that has the lowest value for this parameter will be first asynchronous loaded during application startup. When a thread finishes cache load of a table, it takes the next table from the

priority list and loads its cache, and so on. The default value for this parameter is -1. This means that the cache for that table will not be asynchronous loaded.

The parameter simpleCacheRowCountLimit defines max number of rows in the table for which simple cache is still used. If the table has more rows than defined by this parameter, complex cache is used for simple queries. The default value is 0 (simple cache is used for all simple queries).

The parameter synchLoadRowCountLimit defines the max number of rows in the table for which the synchronous cache load is performed if defined by configuration. If the number of rows is greater, the table's cache will be loaded asynchronous and this number will be taken for the asynchLoadPriority . The default value is 0 (asynch cache load is not performed if configuration parameters for asynch cache load are not defined).

The parameter maxExecuteTimeCacheInit is similar to table parameter maxExecuteTime, but defined for cache initialization. It defines the max time for which the query is not printed in application's log file during the cache initialization. If the time is greater, query (SQL statement, execution time and maxExecutionTime) is printed. The default value is value defined for parameter maxExecuteTime (whose default is 0 - nothing is printed).

The parameter queryTimeoutCacheInit is similar to table parameter QueryTimeout, but defined for cache initialization. It defines max number of seconds for which the query for cache initialization should be executed. If the limit is exceeded, an exception is thrown. The default value is value defined for parameter QueryTimeout (whose default is 0 - no limit).

The parameter queryTimeLimitCacheInit is similar to table parameter QueryTimeLimit, but defined for cache initialization. It defines max number of milliseconds for which the query for cache initialization should be executed and the resulset read from ResultSet. If the limit is exceeded, an Exception is thrown. The default value is value defined for parameter QueryTimeLimit (whose default is 0 - no limit).

In the previous mentioned DiscRack example for cache configuration, DataStruct cache for table person has type full, because maxCacheSize is negative and initialCondition is "*". This combination of parameters values forms special case of complete cache: *full* cache.

DODS has class org.enhydra.dods.cache.UpdateConfigurationAdministration. This class has public synchronized methods that provide possibility of run-time setting some cache and table parameters. This class is used by Enhydra application CacheAdmin. It is not recommended to be used by user applications.

Table and cache statistics

DODS has the possibility of providing table and cache statistics.

The public method

```
get_statistics()
```

of the <table_name>DO.java class returns the statictics object (statistics object must implement org.enhydra.dods.statistics.Statistics interface). This object provides the following methods for the table statistics and one method for retrieving cache statistics:

- getStatisticsType()

  Returns type of the statistics. It returns 0 if statistics is for table that has no caching, 1 if statistics is for table with only Data caching, and 2 if statistics is for table with Query caching.

- getInsertNum()

Returns number of insert statements performed on the table.

- setInsertNum(int newInsertNum)

  Sets number of insert statements performed on the table to value newInsertNum.

- incrementInsertNum()

  Increases number of insert statements performed on the table for one.

- getUpdateNum()

  Returns number of update statements performed on the table.

- setUpdateNum(int newUpdateNum)

  Sets number of update statements performed on the table to value newUpdateNum.

- incrementUpdateNum()

  Increases number of update statements performed on the table for one.

- getDeleteNum()

  Returns number of delete statements performed on the table.

- setDeleteNum(int newDeleteNum)

  Sets number of delete statements performed on the table to value newDeleteNum.

- incrementDeleteNum()

  Increases number of delete statements performed on table for one.

- getDMLNum()

  Returns number of DML operations (inserts, updates and deletes) performed on the table.

- getLazyLoadingNum()

  Returns number of lazy loadings performed on the table.

- setLazyLoadingNum(int newLazyLoadingNum)

  Sets number of lazy loadings performed on the table to value newLazyLoadingNum.

- incrementLazyLoadingNum()

  Increases number of lazy loadings performed on the table for one.

- getStartTime()

  Returns time when the statistics was started.

- setStartTime(Date startTime)

  Sets time when the statistics starts to value startTime.

- getStopTime()

  Returns time when the statistics was stopped.

- setStopTime(Date stopTime)

  Sets time when the statistics stops to value stopTime.

- stopTime()

  Sets stop time to current time.

- getQueryNum()

  Returns total number of non-oid queries performed on the table. Query by oid is query which "where" clause contains request for DO with specified oid. Non-oid query is any other query.

- setQueryNum(int newQueryNum)

  Sets total number of non-oid queries performed on the table to value newQueryNum.

- incrementQueryNum()

  Increases total number of non-oid queries performed on the table for one.

- getQueryByOIdNum()

  Returns total number of queries by oid performed on the table.

- setQueryByOIdNum(int newQueryByOIdNum)

  Sets total number of queries by oid performed on the table to value newQueryByOIdNum.

- incrementQueryByOIdNum()

  Increases total number of queries by oid performed on the table for one.

- getQueryAverageTime()

  Returns average time needed for executing non-oid query.

- updateQueryAverageTime(int newTime)

  Updates average time needed for executing non-oid queries to value newTime.

- getQueryByOIdAverageTime()

  Returns average time needed for executing query by oid.

- updateQueryByOIdAverageTime(int newTime, int no)

  Updates average time for executing OId queries with time newTime and increments number of them for paramether no.

- clear()

  Clears DO, simple query and complex query statistics.

- getCacheStatistics(int type)

Returns cache statistics (objects must implement interface org.enhydra.dods.statistics.CacheStatistics) for :

- DataStruct cache when parameter type equals 0
- simple query cache when parameter type equals 1
- complex query cache when parameter type equals 2
- multi-join query cache when parameter type equals 3

Cache statistics objects have the following methods:

- getCacheAccessNum()

  Returns total number of times the cache was accessed.

- setCacheAccessNum(int num)

  Sets total number of times the cache was accessed to value num.

- incrementCacheAccessNum(int num)

  Increases total number of times the cache was accessed for value num.

- getCacheHitsNum()

  Returns number of cache accesses that were successful.

- setCacheHitsNum(int cacheHitsNum)

  Sets number of of cache accesses that were successful to value cacheHitsNum.

- incrementCacheHitsNum(int num)

  Increases number of cache accesses that were successful for value num.

- getUsedPercents()

  Returns how much cache is currently used. This value is given in percents. If cache is unbounded, method returns 100%.

- getCacheHitsPercents()

  Returns how many cache accesses were successful. This value is given in percents.

- clearStatistics()

  Clears statistics.

Chapter 6. Data Caching

Select statement

For query by oid (query by oid is query which "where" clause contains request for DO with specified oid), first is checked in the DataStruct cache if there is DataStruct object with desired oid. If DataStruct object is not found in the cache, hitting the database is performed, and the retrieved DataStruct object is added to the DataStruct cache.

For full caching also, for query by oid, first is checked in the DataStruct cache if there is DataStruct object with desired oid. If DataStruct object is not found in the cache, hitting the database is not performed (all rows from the table are in the cache, so there is no result of this query).

For all other queries, hitting the database is immediately performed, and the query results are added to the DataStruct cache.

Insert statement

Data object is inserted in the database and first time the data is moved to original DataStruct, it is added to the DataStruct cache, after successful commit.

Update statement

Data object is updated in the database and first time the data is moved to original DataStruct, it is added to the DataStruct cache if commit was successful (the old DataStruct object is removed from the DataStruct cache if it was there).

Delete statement

Deletes data object from the database and removes its original DataStruct object originalData from the DataStruct cache (if it is there).

Chapter 7. Query Caching

Select statement

For query by oid (query by oid is query which "where" clause contains request for DO with specified oid), first is checked in the DataStruct cache if there is DataStruct object with desired oid. If DataStruct object is not found in the cache, hitting the database is performed, and the retrieved DataStruct object is added to the DataStruct cache. Queries by oid are not added in the query cache (they are trivial).

For full caching also, for query by oid, first is checked in the DataStruct cache if there is DataStruct object with desired oid. If DataStruct object is not found in the cache, hitting the database is not performed (all rows from the table are in the cache, so there is no result of this query).

For non-oid queries, for full caching, if the query is simple query, the query's result can be retrieved from the DataStruct cache, so there is no need to retrieve results from the database. In any other case of full caching, everything is done the same as for any other query (this is explained in the next paragraph).

For all other queries, it is checked if the query is already in the Query cache (simple, complex or multi-join). Query object has one attribute called "orderRelevant" which is true if query results must not be modified (no DO can be inserted, updated or deleted from cached query results). With the method isOrderRelevant() is checked whether the results of select can be modified or not.

If query is in the cache and the isOrderRelevant() returns false, result oids are retrieved from QueryCache. If query is in the cache and the isOrderRelevant() returns true, and the result oids are not modified, the result oids are also retrieved from query cache. But, if query is in the cache and the isOrderRelevant() returns true, but the result oids are modified, the result oids from the QueryCache are not used. Instead of that, hitting the database is performed.

If the result is found in the query cache, for every result oid, it is checked whether there is that object is in the DataStruct cache. Then, when is counted number of results that are not in the DataStruct cache, the time needed for performing queries by oid on database for all oids from the result that are not in the cache is compared against the time needed for performing the whole query.

If the time needed for performing queries by oid on database is less or equal to query execution time, results are retrieved from the cache, and those that are not there, from database (using queries by oid).

If the time is longer, or the query is not in the query cache, or the query supports joins with other tables, or cached query results are modified but for this query is order relevant, the query is performed on the database.

If the results are retrieved from the database, the query and its necessary data are put in the Query cache (simple, complex or multi-join).

If there was already that query in the query cache, but the query was executed again (because there were not enough result oids in the result list, or because the old query was modified, and for the new query isOrderRelavant is true), the old query is replaced by the new one (this query is not modified).

Insert statement

Data object is inserted in the database and first time the data is moved to original DataStruct, it is added to the DataStruct cache, after successful commit.

All complex and multi-join queries of the table that are for the database of inserted DO, are removed from the query caches.

For every simple query of the table (with the inserted DO's database) from query cache it is checked whether inserted DO is query result or not.

If new DO is query result, in the query cache is this query marked as "modified".

If its cached results are complete (all are in the query cache), oid of this inserted DO is added to query cached result list. If cached results are not complete oid is not added to the list.

Update statement

Data object is updated in the database and first time the data is moved to original DataStruct, it is added to the cache if commit was successful (the old DataStruct object is removed from the DataStruct cache if it was there).

All complex and multi-join queries of the table that are for the database of inserted DO are removed from the query caches.

For every simple query of the table (with the inserted DO's database) from query cache it is checked whether updated DO is the query result or not.

If yes, this query is marked as "modified" in the query cache, and the DO is included in query results only if it wasn't in the cache and the cached result list is complete.

If no, if DO's oid exists in the query results, it is removed from there and because of this change of the results, this query is marked as "modified" in the query cache.

Delete statement

Deletes DO from the database and removes its original DataStruct object originalData from the DataStruct cache (if it is there).

Goes through the query cache (simple, complex and multi-join) and wherever finds this DO, removes it from the query results and marks that query as "modified".

Chapter 8. Caching And Lazy Loading

Overview

As mentioned before, this mechanism is used in the method createExisting with one of the parameter combinations:

```
createExisting(ObjectId id)
```

or

```
createExisting( String dbName, ObjectId id, [HashMap refs])
```

or

```
createExisting( ObjectId id, [HashMap refs], [DBTransaction dbTrans]).
```

for retrieving DO with specified oid from the database.

When caching is used, the only difference is that this method first checks whether there is DataStruct object with the specified oid in the DataStruct cache. If yes, this DataStruct object is used for creating DO, and this DO is returned.

If the specified oid doesn't exist in the DataStruct cache, the rest of the method is the same: DO with specified oid (DO without data) is created using constructor, and then, if lazy loading is off, the method

```
loadData()
```

is called to load the fields for the DO from the database.

Lazy data objects (whose original data originalData is empty) are not added to the DataStruct cache. After they are loaded (with loadData() method), their original data originalData is added to the DataStruct cache.

Select statement with Lazyloading and Caching

For, example, in select clauses (runQuery() method) when are created referenced objects is used method

```
createExisting( [String dbName], BigDecimal bd )
```

which calls mentioned method createExisting with the ObjectId parameter and combination of dbName,refs and dbTrans parameters.

If both lazy loading and caching are used, if object exists in the DataStruct cache, it is retrieved from there, and if not, the object is formed, but its data is empty (because of lazy loading).

Later, if object is needed, data is loaded using method:

checkLoad() or loadData().

of <table_name>DO.java class.

Chapter 9. Security

Security concerns users and their rights to access to details of DO or the whole class. Security is used if parameter generateSecure (attribute of <database>, <table> and <column> tag in doml file) is set to true.

In this case, <table_name>DO.java class extends:

```
org.webdocwf.dods.access.SecureDO
```

class. This class is abstract and implements some methods of the class:

```
com.lutris.dods.builder.generator.dataobject.GenericDO
```

adding the security (check of user access rights).

They is one more parameter for security used in doml file: generateInsecure (also attribute of <database>, <table> and <column> tag). If generateInsecure is set to true then DODS while generate data access methods without user access check.

Parameters generateSecure and generateInsecure are not mutually exclusive, they can be added independently of each other. In that case both groups of data access methods (with or without users access check) can be generated according to parameters values.

Default value for DODS generator is generateInsecure=true and generateSecure=false.

The method of <tableName>DO that use user access rights are:

- createVirgin(org.webdocwf.dods.access.User usr)
- createVirgin(DBTransaction dbTrans, org.webdocwf.dods.access.User usr)
- createVirgin(String dbName, org.webdocwf.dods.access.User usr)

This method creates new clean DO with user acess concerns.

- createExisting(BigDecimal bd, org.webdocwf.dods.access.User usr)
- createExisting(BigDecimal bd, DBTransaction dbTrans, org.webdocwf.dods.access.User usr)
- createExisting(String dbName, BigDecimal bd, org.webdocwf.dods.access.User usr)
- createExisting(String handle,org.webdocwf.dods.access.User usr)
- createExisting(String handle, DBTransaction dbTrans, org.webdocwf.dods.access.User usr)
- createExisting(String dbName, String handle,org.webdocwf.dods.access.User usr)

This method creates new DO object based on data from existing DO with user access checks.

Other methods of DO object that use security are:

- createCopy(<table_name>DO orig, org.webdocwf.dods.access.User usr)
- createCopy(<table_name>DO orig, DBTransaction dbTrans, org.webdocwf.dods.access.User usr)
- createCopy(String dbName, <table_name>DO orig, org.webdocwf.dods.access.User usr)

  Method createCopy creates a DO that has no ObjectId but has a copy of an existing DO's data. Such a DO is used to insert a new database entry that is largely similar to an existing entry.

- originalData_get<column_name>([User usr])

  Returns the row value of the column <column_name> of the DO's originalData object.

- findTransactionCachedObjectByHandle(String cacheHandle,org.webdocwf.dods.access.User usr)

  Gets DO with key cacheHandle from the cache.

- findCachedObjectByHandle(String cacheHandle,org.webdocwf.dods.access.User usr)

  Gets DataStruct object with key cacheHandle from the cache.

- get and set methods for table columns

The constructors of <tableName>Query that use user access rights are:

- <tableName>Query(org.webdocwf.dods.access.User usr)
- <tableName>Query(DBTransaction dbTrans, org.webdocwf.dods.access.User usr)
- <tableName>Query(String dbName, org.webdocwf.dods.access.User usr)
- <tableName>Query(org.webdocwf.dods.access.User usr)

These constructors craete new <tableName>Query object with user acess checks.

Other methods of <tableName>Query class that use security are setQuery<column_name> methods, setUserMatch<column_name> method and setDBMatch<column_name> method.

The methods of SecureDO class that check user access rights are:

- public void assertDODeleteAccess(User usr)

Ensures that the given user is allowed to delete the DO.

- public boolean hasDODeleteAccess(User usr)

Checks if the given user is allowed to delete the DO.

- public void assertDOCopyAccess(User usr)

Ensures that the given user is allowed to copy the DO.

- public boolean hasDOCopyAccess(User usr)

Checks whether the given user is allowed to copy the DO.

- public boolean hasDOGetAttrAccess(String attrName, Object value, User usr)
- public boolean hasDOGetAttrAccess(String attrName, boolean value, User usr)
- public boolean hasDOGetAttrAccess(String attrName,byte value, User usr)
- public boolean hasDOGetAttrAccess(String attrName, short value, User usr)
- public boolean hasDOGetAttrAccess(String attrName, int value, User usr)

- public boolean hasDOGetAttrAccess(String attrName, long value, User usr)
- public boolean hasDOGetAttrAccess(String attrName, float value, User usr)
- public boolean hasDOGetAttrAccess(String attrName, double value, User usr)
- public boolean hasDOGetAttrAccess(String attrName, byte[] value, User usr)

Checks whether the given user is allowed to read the attribute and the value.

- protected boolean hasDOSetAttrAccess(String attrName,Object oldValue, Object newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, boolean oldValue, boolean newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, byte oldValue, byte newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, short oldValue, short newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, int oldValue, int newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, long oldValue, long newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, float oldValue, float newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, double oldValue, double newValue, User usr)
- protected boolean hasDOSetAttrAccess(String attrName, byte[] oldValue, byte[] newValue, User usr)

Checks whether the given user is allowed to update the attribute and the value.

- public void assertDOGetDOValueAccess(String attrName, SecureDO value, User usr)

Ensures that the given user is allowed to read the object in a given pointer.

- protected boolean hasDOGetDOValueAccess(SecureDO value, User usr)

Checks whether the given user is allowed to read the object in a given pointer.

- public boolean hasQueryFindAccess(User usr)

Checks whether the given user is allowed to find the object using a query.

- public void assertDOGetVersionAccess(User usr)

Ensures that the given user is allowed to access the version number.

- public boolean hasDOGetVersionAccess(User usr)

Checks whether the given user is allowed to access the version number.

- public void assertDOIsReadOnlyAccess(User usr)

Ensures that the given user is allowed to read the read-only flag.

- public boolean hasDOIsReadOnlyAccess(User usr)

Checks whether the given user is allowed to read the read-only flag.

- public void assertDOGetAccess(User usr)

Ensures that the given user is allowed to read the DO existence.

- public boolean hasDOGetAccess(User usr)

Checks whether the given user is allowed to read the DO existence.

- public void assertDOMakeReadOnlyAccess(User usr)

Ensures that the given user is allowed to set the object readonly.

- public boolean hasDOMakeReadOnlyAccess(User usr)

Checks whether the given user is allowed to set the object readonly.

- public void assertDOMakeReadWriteAccess(User usr)

Ensures that the given user is allowed to set the object readwrite.

- public boolean hasDOMakeReadWriteAccess(User usr)

Check whether the given user is allowed to set the object readwrite.

The User (org.webdocwf.dods.access.User) mentioned in all these methods is DODS Access User Interface.

```
org.webdocwf.dods.access.User
```

It defines user's access rights and its methods are used in previously mentioned methods. Applications that use security should implement this User interface.

The methods of User interface that should be implemented are:

- public void restrictQuery (Query query)

Restricts the Query for DODS Query access using SQL.

- public boolean hasDOCopyAccess (GenericDO obj)

Decides whether the User is allowed to copy the DO.

- public boolean hasDOGetAccess (GenericDO obj)

Decides whether the User is allowed to read the existance of the DO.

- public boolean hasDOGetVersionAccess (GenericDO obj)

Decides whether the User is allowed to read the version of the DO.

- public boolean hasDOIsReadOnlyAccess (GenericDO obj)

Decides whether the User is allowed to read the readOnly attribute of the DO.

- public boolean hasDOMakeReadOnlyAccess (GenericDO obj)

Decides whether the User is allowed to set the readOnly attribute to true of the DO.

- public boolean hasDOMakeReadWriteAccess (GenericDO obj)

Decides whether the User is allowed to set the readOnly attribute to false of the DO.

- public boolean hasDODeleteAccess (GenericDO obj)

Decides whether the User is allowed to delete the DO.

- public boolean hasDOCreateAccess (String className)

Decides whether the User is allowed to create the DO of a certain class.

- public boolean hasQueryAccess (String className)

Decides whether the User is allowed to query the DOs of a certain class.

- public boolean hasQueryFindAccess (GenericDO obj)

Decides whether the User is allowed to find the DO during a query.

- public boolean hasQueryAttrAccess (String className, String attrName, Object queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, boolean queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, byte queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, short queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, int queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, long queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, float queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, double queryValue, String cmp_op)
- public boolean hasQueryAttrAccess (String className, String attrName, byte[] queryValue, String cmp_op)

Decides whether the User is allowed to query DOs for an attribute with a certain query-value and comparison operator.

Select statement

For every query, in runQuery() method of the <table_name>Query.java class, the User's method restrictQuery(Query query) is called. It restricts the Query for DODS Query access using SQL.

For query by oid (query by oid is query which "where" clause contains request for DO with specified oid), first is checked in the DataStruct cache if there is DataStruct object with desired oid. If DataStruct object

is not find in the cache, hitting the database is performed, and the retrieved DataStruct object is added to the DataStruct cache. Queries by oid are not added in the query cache (they are trivial).

For found DO (in the cache or in the database), it is checked if the user has rights to assert this object. For this is used method of SecureDO.java class:

```
hasQueryFindAccess(User user)
```

This method calls User's method:

```
hasQueryFindAccess(GenericDO obj)
```

for the DO. This method returns true if the User is allowed to find the DO during a query. If user can find this object, it is added in query result.

For full caching also, for query by oid, first is checked in the DataStruct cache if there is DataStruct object with desired oid. If DataStruct object is not find in the cache, hitting the database is not performed (all rows from the table are in the cache, so there is no result of this query). After that, on the way described before, it is checked if the User is allowed to find the DO during a query. If yes, the object is added in query result.

For non-oid queries, for full caching, if the query is simple query, the query's result can be retrieved from the DataStruct cache, so there is no need to retrieve results from the database and for every result, the check of user rights is performed on the same way. In any other case of full caching, everything is done the same as for any other query (this is explained in the next paragraph).

For all other queries, it is checked if the query is already in the Query cache (simple, complex or multi-join). Query object has one attribute called "orderRelevant" which is true if query results must not be modified (no DO can be inserted or updated or deleted from cached query results). With the method isOrderRelevant() is checked whether the results of select can be modified or not.

If query is in the cache and the isOrderRelevant() returns false, result oids are retrieved from QueryCache. If query is in the cache and the isOrderRelevant() returns true, and the result oids are not modified, the result oids are also retrieved from query cache. But, if query is in the cache and the isOrderRelevant() returns true, but the result oids are modified, the result oids from the QueryCache are not used. Instead of that, hitting the database is performed.

If the results are found in the query cache, for every result oid, it is checked whether there is that object is in the DataStruct cache. Then, when is counted number od results that are not in the DataStruct cache, the time needed for performing queries by oid on database for all oids from the result that are not in the cache is compared against the time needed for performing the whole query.

If the time needed for performing queries by oid on database is less or equal to query execution time, results are retrieved from the cache, or, if they are not there, from database (using queries by oid).

If the time is longer, or the query is not in the query cache, or the query supports joins with other tables, or cached query results are modified but for this query is order relevant, the query is performed on the database.

If the results are retrieved from database, the query and its necessary data are put in the Query cache (simple, complex or multi-join).

For every result data object, (from the cache or the database), it is checked if the User is allowed to find the DO during a query. If yes, the object is added in query results.

If there was already that query in the query cache, but the query was executed again (because there were not enough result oids in the result list, or because the old query was modified, and for the new query isOrderRelavant is true), the old query is replaced by the new one (this query is not modified).

If caching is not used, query is executed on the database, and for every result data object, it is checked if the User is allowed to find the DO during a query. If yes, the object is added in query results.

Lazy Loading

In the case of lazy loading, while DO's original data originalData is empty, User access rights can't be checked. For this reason, security methods of SecureDO.java class which check assert rights about DOs, call method:

checkLoad()

which loads DO's original data originalData, if not loaded, and then, Users access rights methods are called for loaded DOs.

Chapter 10. unique attribute

This is attribute of <table_name>Query.java class. It is a flag that indicates whether the returned query results should be unique or not. If true, only unique results are returned, otherwise, all results are returned.

This attribute is private. The public methods that work with this attribute are also in the <table_name>Query.java class and they are:

- public void setUnique (boolean newUnique)

Sets the unique attribute to value newUnique.

- public boolean getUnique()

Returns the current value of unique attribute.The default value of this attribute is false.

Chapter 11. maxDBrows attribute

This is attribute of <table_name>Query.java class. It is used in execution of select statements. It defines maximal number of rows that can be retrieved when query is executed.

If query is executed on the database, this is maximal number of rows retrieved from database. If query is retrieved from the query cache, this is the number of oids from result list (starting from the beginning of the list) that will be retrieved.

This attribute is private. The public methods that work with this attribute are also in the <table_name>Query.java class and they are:

- public void setMaxRows(int maxRows)

Sets attribute maxDBrows to value maxRows.

- public int getMaxRows()

Returns current value of the attribute maxDBrows.

The default value of this attribute is 0.

If unique attribute or security is used, we recommend using (instead of this attribute) databaseLimit attribute, and its methods:

- public void setDatabaseLimit(int newLimit)

- public int getDatabaseLimit()

because databaseLimit attribute respects unique attribute and security. If unique attribute and security are not used, the query automatically sets maxDBrows parameter:

```
if ((user == null) && (databaseLimit != 0) && (! unique))
  setMaxRows(databaseLimit + 1);
```

For example, databaseLimit is set to 20. If unique attribute is set to true, the query will return 20 unique results (if there are so many results). If security is used, the query will return 20 results which the user can read (if there are so many results).

If instead of databaseLimit attribute is used maxDBrows attribute when unique parameter or security is used, we do not know for sure how many results we will get as query results.

For example, maxDBrows is set to 20. If unique attribute is set to true, the query will return from the database (or cache) 20 results, and then filter them (remove ununique results). So, the query results will return 20 or less results (if there were ununique results in those 20 results). The story is the same for the security: The query will return from the database (or cache) 20 results, and then filter them (remove results which the user can not read). So, the query results will return 20 or less results (if there were results in those 20 results to which user did not have acess rights).

Chapter 12. databaseLimit attribute

This is attribute of <table_name>Query.java class. It is used in execution of select statements. It defines maximal number of DOs that will be returned as query result when query is executed.

For example, if query should return only unique results, DatabaseLimit is maximal number of unique query results that will be returned as query result.

For example, if security is used in queries, DatabaseLimit is maximal number of query results (for which user has access) that will be returned as query result.

When the query is executed, always will be retrived databaseLimit+1 results from database to get information is there are any more results that are not retrieved with the query because of databaseLimit attribute.

This attribute is private. The public methods that work with these attributes are also in the <table_name>Query.java class and they are:

- public void setDatabaseLimit(int newLimit)

Sets the attribute DatabaseLimit to value newLimit.

- public int getDatabaseLimit()

Returns the current value of the DatabaseLimit attribute.

The default value of this attribute is 0.

Chapter 13. databaseLimitExceeded attribute

This is attribute of <table_name>Query.java class. It is a flag that indicates whether there are more query results beside the ones returned as query result. It is useful in queries that use databaseLimit and readSkip attributes.

This attribute is private. The public method that work with this attribute is also in the <table_name>Query.java class and it is:

- public boolean getDatabaseLimitExceeded()

Returns the current value of the databaseLimitExceeded attribute. The <table_name>Query.java class sets this attribute to true, if needed.

The default value is false.

Chapter 14. readSkip attribute

This is attribute of <table_name>Query.java class. It is used in execution of select statements. It defines how many first results will be skipped (will not be returned as query results) when query is executed.

For example, if query should return only unique results, readSkip is the number of first unique query results that will be skipped, and not returned as query result.

For example, if security is used in queries, readSkip is the number of first query results (for which the user has access) that will be skipped and not returned as query result.

If query should return unique results and security is used, readSkip is the number of first unique query results, for which the user has access, that will be skipped, and not returned as query result.

This attribute is private. The public methods that work with this attribute are also in the <table_name>Query.java class and they are:

- public void setReadSkip(int newReadSkip)

Sets the attribute readSkip to value newReadSkip.

- public int getReadSkip()

Returns the current value of readSkip attribute.

The default value of this attribute is 0.

Chapter 15. Read-only per Table

Read-only attribute is one of the table (tableConfiguration object) attributes. The object tableConfiguration can be retrieved from <table_name>DO.java class by calling method: getConfigurationAdministration().getTableConfiguration(). If table is read-only, insert, update and delete operations are not allowed on the table, only selects are allowed. This attribute can be handled from <table_name>DO.java class as following:

- getTableConfiguration().isReadOnly()

Returns the current value of readOnly attribute.

If AllReadOnly is set on DatabaseManager or Database level to true, the value for readOnly on table level can not be overridden – in all other cases readOnly attribute behaves usually. If AllReadOnly is set to true and readOnly attribute of table is set to false, warning is written to log during table initialization. In runtime exception is thrown on attempt of writing to that table.

Chapter 16. Global Read-only

The AllReadOnly attribute is one of the Database/DatabaseManager attributes.

The DatabaseManager configuration for AllReadOnly parameter can be retrieved by calling methods:

```
DODS.getDatabaseManager().getDatabaseManagerConfiguration().getAllReadOnly()
```

The Database configuration for AllReadOnly parameter can be retrieved by calling methods:

```
DODS.getDatabaseManager().findLogicalDatabase(<DbName>).getDatabaseConfiguration().getAllReadOnly()
```

This attribute is a flag that indicates whether all tables are read-only. If true, all tables are read-only. This means that insert, update and delete operations are not allowed on any of tables, only selects are allowed.

Since only selects are allowed when this attribute is set to true, in this case, queries with possible joins with other tables are also cached in the query cache (query with possible joins with other tables means that the query was built with methods that allow joins between tables).

This attribute can be handled as static method from <table_name>DO.java class as following :

- isAllReadOnly()

Returns the current value of (global) AllReadOnly attribute.

The value for AllReadOnly attribute is read from application's configuration file as:

```
DatabaseManager.defaults.AllReadOnly
```

```
DatabaseManager.DB.<database_name>.AllReadOnly
```

If AllReadOnly is set on DatabaseManager or Database level, the readOnly attribute on table level can not be overridden – in all other cases the readOnly attribute behaves usually. If AllReadOnly is set to true and readOnly attribut of table is set to false, warning is written to log during table initialization. In runtime exception is thrown on attempt of writing to that table.

If not defined in configuration file, the default value is used. The default value is false.

Chapter 17. Delete cascade

The class <table_name>DO.java provides java code for cascade delete of DOs.

The database that the application uses can support delete cascade. If supports, in its configuration file (in <dods_output>/build/conf directory) is the tag <DeleteCascade> set to true:

```
<DeleteCascade>true</DeleteCascade>.
```

If the database supports the delete cascade, the sql that handles delete cascade is generated. The java code in <table_name>DO.java class is always generated.

When delete cascade happens, if database supports delete cascade, the sql code is executed, the java code is not executed, and the cascade deleted DOs are marked as deleted.

If database does not support delete cascade, the sql code is neither generated, nor executed. The java code is executed, and cascade references are deleted.

Some database vendors (Microsoft SQLServer2000) have restrictions to delete cascade feature. Microsoft SQL Server 2000 support delete cascade only if there is no closed reference cycle between tables. Server will not allow creating reference cycle if they are set to delete cascade. In that case best solution is to set <DeleteCascade> tag to 'false', and leave DODS to do all delete cascade operation.

Chapter 18. Multi Database Support

DODS has the possibility of working with more than one database at the same time. This means that, when the application is started, it doesn't have to be stopped in order to change the database the application uses.

The table supports multi databases if the attribute multidb of <table> tag in DOML file is set to true for that table.

To take advantage of simultaneous use of multiple tables, DODS requires separate DOML file for every distinct database.

Example:

```
<doml>
  <database database="Standard">
    .........
    <package id="multibase.data.employee">
        <table id="multibase.data.employee.Employee"  multidb="true">
             <column id="firstName" usedForQuery="true">
                 <type dbType="VARCHAR" javaType="String"/>
             </column>
             ........
        </table>
        .........
    </package>
    .........
</doml>
```

```
<doml>
  <database database="Standard">
        ............
        <package id="multibase.data.employee.programer">
          <table id="multibase.data.employee.programer.Programer"  multidb="true">
            <column id="firstName" usedForQuery="true">
              <type dbType="VARCHAR" javaType="String"/>
            </column>
            ............
          </table>
          ............
        </package>
        ............
  </database>
</doml>
```

For that kind of table, in <App_name>.conf file must be defined all logical databases the application will use on this table. For each of these databases must be configured all needed parameters.

Example:

```
#-------------------------------------------------------------------------
#                   Database Manager Configuration
#-------------------------------------------------------------------------

DatabaseManager.Databases[] = "programer", "employee"
DatabaseManager.DefaultDatabase = "employee"

DatabaseManager.DB.programer.Connection.User = ""
DatabaseManager.DB.employee.Connection.User = ""

DatabaseManager.DB.programer.Connection.Password = ""
DatabaseManager.DB.employee.Connection.Password = ""

.................................................

DatabaseManager.DB.programer.Connection.Logging = false
DatabaseManager.DB.employee.Connection.Logging = false
```

```
DatabaseManager.DB.programer.ObjectId.CacheSize = 20
DatabaseManager.DB.programer.ObjectId.MinValue = 1000000

DatabaseManager.DB.employee.ObjectId.CacheSize = 20
DatabaseManager.DB.employee.ObjectId.MinValue = 1000000
```

If the table doesn't support multi databases, the default database will be used for this table.

When the <App_name>.conf file (with information about all databases) is updated, and the application is started, it uses the default database. The definition of the new (desired) database is being done in the stage of creation of DO and Query objects.

When a Query object is created for a database (given or default), the results of this Query are only DOs from that database, not from any other database.

If caching is used, there is only one cache for all <table_name>DO's original data originalData (these DataStruct objects can belong to different databases, but are all placed in the same DataStruct cache).

DODS takes care of referential integrities within the database which means that DODS searches referenced object in the same database in which is the object that referenced it . If you want to use referenced objects from any other database, you must yourself take care of referential integrities.

In the <table_name>DO class public constructors and methods (query, createVirgin, createCopy, createExisting) are now defined and with the database parameter.

Here are some examples of using these constructors and methods.

Query example:

```
ProgramerDO[] programers;
ProgramerQuery pQuery = new ProgramerQuery("programer");
programers = pQuery.getDOArray();
```

Create example:

```
EmployeeDO newE=EmployeeDO.createVirgin("employee");
newE.setFirstName(employees[i][0]);
.................................
newE.save();
```

Example of transferring data from one database to another

```
ProgramerDO[] programers;
ProgramerQuery pQuery = new ProgramerQuery("programer");
programers = pQuery.getDOArray();
for(int i=0; i< programers.length; i++) {
   EmployeeDO newEmployee=EmployeeDO.createVirgin("employee");
   newEmployee.setFirstName(programers[i].getFirstName());
   newEmployee.setLastName(programers[i].getLastName());
   newEmployee.setOccupation("programer");
   newEmployee.setDepartment("IT");
   newEmployee.save();
}
```

You can use these methods with this parameter, and then the object will be created for the given logical database. or you can use these constructors and methods without database parameter. In this case, they will be created for default database. If the methods with database parameter are used, and the parameter is set to null, the default database is used.

Chapter 19. Fetch size

Fetch size gives the JDBC driver a hint to the number of rows that should be fetched from the database when more rows are needed.

Fetch size is defined in QueryBuilder.java class (package com.lutris.dods.builder.generator.query). Two values are defined:

- iDefaultFetchSize - default fetch size (static variable) - depeds on value from configuration file
- iCurrentFetchSize - current fetch size (non-static variable) - value for current query

DefaultFetchSize is one of the parameters (optional) of the applications configuration file. This parameter (among some other parameters) is read by the constructor:

```
StandardDatabaseManager(Config config)
```

of

```
StandardDatabaseManager.java class.
```

This parameter can be accessed with method:

```
DODS.getDatabaseManager().getDatabaseManagerConfiguration().getFetchSize()
```

The default values for iDefaultFetchSize and iCurrentFetchSize are:

- private static int iDefaultFetchSize = -1
- private int iCurrentFetchSize = -1;

This value means that default and current fetch sizes are not defined.

The fetch size is used in QueryBuilder's method:

```
private void prepareStatement( DBConnection conn )
```

This method generates a JDBC PreparedStatement using the values passed for the where-clauses. If current fetch size is defined, that value is used. If not, the default fetch size is used. If this value is also not defined, the fetch size is not set for that JDBC PreparedStatement.

There are two methods in Query class that work with fetch size for current query. They are:

- set_FetchSize (int iCurrentFetchSizeIn)

Sets the current fetch size (does not update configuration file), overrides default value.

- get_FetchSize()

Reads the current value of fetch size. If new value is not set using set_FetchSize method, returns default value.

Chapter 20. Cache Initialization

For every DataStruct cache, it is possible to define initial query statement which contains "where" clause which is used during DataStruct cache initialization. When cache is created, query with this "where" condition is performed on the database, and the results are put in the DataStruct cache.

Before the query is executed, parameter maxDBrows is set to maxCacheSize using method

setMaxRows(int max)

of <table_name>Query.java class.

Using this method, maximum maxCacheSize DOs will be retrieved from the database and their original DataStruct objects (originalData) will be added to DataStruct cache.

If a table is fully cached, simple queries are done in the memory (even the first time this is done in the cache and not in the Database)

If initial query statement is set to "*", all rows of the table from the database (up to maxCacheSize) will be put in the DataStruct cache.

If initial query statement is set to null, no rows from the table in database will be put in the DataStruct cache during the cache initialiyation (cache would be empty).

The parameter ClassList defines the absolute path to "DODSClassList.xml" file that contains the list of data layer classes names whose caches should be initialized. There are two types of cache initialization: synchronous and asynchronous.

Synchronous cache initialization is performed during the application startup.

Beside ClassList parameter, for aynchronous cache initialization, two more parameters must be defined: *asynchLoadThreadNum* and *asynchLoadPriority*.

If parameter *asynchLoadThreadNum* is defined and has positive value, the asynchronous cache load would be performed with the number of threads defined by this parameter. If the parameter is not defined, or has negative or 0 value, the asynchronous cache initialization will not be performed during the application startup.

The parameter *asynchLoadPriority* (must be number equal or greater than zero) defines the priority (order) for the table's asynchronous cache load. The lower the number, the sooner the cache for the table will be loaded.

As mentioned, if parameter *asynchLoadThreadNum* is not defined (or negative) asynchronous cache load will not be performed for any of tables. Caches of the tables that have defined *asynchLoadPriority*, will not even be initialized synchronous because of this parameter. But, class com.lutris.appserver.server.sql.StandardDatabaseManager has method:

```
public void asynchInitCaches(int threadNum)
```

that can be call later. It defines the number of threads for asynchronous cache initialization and calls the initialization. This method can have effect only on tables whose caches were not loaded before.

Beside this main parameters for cache initialization, there are more parameters that can be used during the cache initialization (all are explained in the chapter *Caching*, in the section about *Cache configuration*):

- *synchLoadRowCountLimit* - max number of rows in the table for which the synchronous cache load is performed if defined by configuration. If the number of rows is greater, the table's cache will be loaded

asynchronous and this number will be taken for the *asynchLoadPriority*. The default value is 0 (asynch cache load is not performed if configuration parameters for asynch cache load are not defined).

- *maxExecuteTimeCacheInit* - max time for which the query is not printed in application's log file during the cache initialization. If the time is greater, query (SQL statement, execution time and maxExecutionTime) is printed. The default value is value defined for parameter *maxExecuteTime* (whose default is 0 - nothing is printed).

- *queryTimeoutCacheInit* - max number of seconds for which the query for cache initialization should be executed. If the limit is exceeded, an exception is thrown. The default value is value defined for parameter QueryTimeout (whose default is 0 - no limit).

- *queryTimeLimitCacheInit* - max number of milliseconds for which the query for cache initialization should be executed and the resulset read from ResultSet. If the limit is exceeded, an Exception is thrown. The default value is value defined for parameter QueryTimeLimit (whose default is 0 - no limit).

When synchronous cache load is performed, the application waits for it to be over before it starts its work. The other case is with asynchronous cache load: the application is running while the caches are being asynchronous loaded.

Chapter 21. Reserve factor

The reserveFactor attribute is one of the cache parameters of the applications configuration file. This attribute can be handled from <table_name>DO.java class as following:

```
getConfigurationAdministartion().getReserveFactor()
```

This method returns the value of reserveFactor attribute.

Reserve factor is constant used in query caching. It is percent of how many more objects are taken for evaluation. If num is number of needed results, then it is used

```
num + reserveFactor * num
```

of objects for estimating what is quicker: go to database for all object that are not in the cache, or run again query on database.

This value is given in percents, as number between 0 and 1 (0.25 means 25%).

Example:

Let us assume that there is a PersonQuery query pQuery, and that are set databaseLimit and readSkip attributes to values:

```
pQuery.setReadSkip(5);
pQuery.setDatabaseLimit(15);
```

When the query is executed, first is checked if this query is cached in the query cache. The reserve factor is used when the query is cached in query cache and the security is used (user != null).

Let us assume that the query is found in the query cache. The list of oids that are results of this query is retrieved from the query cache. For example, the result list has 50 oids.

If security is used, we don't know how many DOs from results will be skipped because the user can not read them, but, it has to be decided what to do (what is quicker): to execute the query on the database again, or to get only DOs from the database that are not in the DataStruct cache.

For this decision some kind of calculation must be done, and this is the reason why the reserveFactor is needed.

For example, reserveFactor = 0.25. Now calculation is performed: needed number of oids for calculation is:

```
(readSkip + databaseLimit) + (readSkip + databaseLimit) * reserveFactor
```

So, it is needed readSkip + databaseLimit = 5+15 = 20 results, but this may not be first 20 because of filtering (user rights may skip some oids), but, until the DO is not loaded from the database it is unknown whether user has rights for this DO or not.

For this reason, a little more oids must be taken and when they are filtered (only DOs whose original data is in the DataStruct cache can be checked for user rights), it is decided what might be better: to execute the query on the database again, or to get only DOs from the database that are not in the DataStruct cache.

In our case, that number is: needed number of oids for calculation is: 5 + 15 + (5 + 15) * 0.25 = 25. The first 25 oids from query result list are taken and they are filtered: it is checked for every oid if the DataStruct is in the DataStruct cache. If yes, the DO is created using this DataStruct object and then checked user rights for that DO. If user can read this DO, it is left in the result list, and if user can not read this DO, it is

removed from the result list. If DataStruct with this oid is not in the DataStruct cache, the user rights can not be checked for that DO, so it is left in the result list for the moment.

When the filtering of chosen 25 oids is finished, there can happen two situations:

1. in the result list is less than 20 (needed number) results; in this case, base to the calculations, the query is again executed on the database

2. in the result list is more than 20 (needed number) results, or exacly 20;

   in this case, if the time needed for getting only DOs that are not in the cache is less than the time needed for executing the whole query on the database, from the database are just retrieved DOs that are not in the DataStruct cache, checked user rights for them, and removed if neccessary

If, after this filtering there are still 20 results, they are returned as query result.

If not, the query is again executed on the database.

Chapter 22. New Parameters in Configuration and DOML Files

Many of parameters explained in this chapter can be applied both on global level (DatabaseManager) or on local level (for each database separately). Also, some parameters can be applied on table directly.

When they are defined on DatabaseManager level they apply on all databases defined in configuration file. Parameters defined on separate databases override those defined on global level. If any of parameters is defined for the table directly, this value is used for that table.

Only exception of this rule is readOnly parameter of table: if AllReadOnly is set on DatabaseManager or Database level to true the readOnly parameter of the tables can not override this value - in all other cases parameters behave usually.

To set parameters on DatabaseManager level, use the following syntax in configuration file:

```
DatabaseManager.defaults.<parameter_name> = parameterValue
```

To set parameters on Database level, use the following syntax in configuration file:

```
DatabaseManager.DB.<database_name>.<parameter_name> = parameterValue
```

To set parameters on table level, use the following syntax in configuration file:

```
DatabaseManager.DB.<database_name>.<table_name>.<parameter_name> = parameterValue
```

Cache parameters can be set on all levels: DatabaseManager, Database and table level.

To set parameters related to cache, use the following syntax in configuration file:

```
DatabaseManager.defaults.cache.<parameter_name> = parameterValue
```

```
DatabaseManager.DB.<database_name>.cache.<parameter_name> = parameterValue
```

```
DatabaseManager.DB.<database_name>.<table_name>.cache.<parameter_name> = parameterValue
```

To get values of parameter from databaseManager level (in runtime) use following methods:

```
DODS.getDatabaseManager().getDatabaseManagerConfiguration().get<CapitalizeParameterName>()
```

To get values of parameter from database level (in runtime) use following methods:

```
DODS.getDatabaseManager().findLogicalDatabase(<DbName>).getDatabaseConfiguration().get<CapitalizeParameterName
```

TransactionCheck

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to true, whenever a DO without a transaction context is created in the memory, a warning is created in the log. The warning message contains information about database name, class name, oid, version and the stack trace. If the parameter is set to false or not defined, nothing happens.

Sample code:

```
#values{true,false} default{false}
DatabaseManager.defaults.TransactionCheck=false;
```

DirtyDOs

Possible values: "Compatible", "Deprecate" and "Omit".

Default value: "Compatible".

File: DOML file.

Context: Database, table.

This parameter is attribute of <database> and <table> tag in DOML file. Optionally, "dirty" methods (methods that can create DOs in memory without transactions) can be marked as "deprecated" or even not be generated at all.

If set to "Compatible", "dirty" methods will be generated (as before), if set to "Deprecate", "dirty" methods will be generated as deprecated, and if set to "Omit", "dirty" methods will not be generated at all.

If parameter is set in <table> tag, it overrides default value and value in <database> tag.

DeleteCheckVersion

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

Before the delete of a DO, if this parameter is set to true, it is checked the DO's version. If the DO had been changed in the meantime, DODS throws an SQLException with message ("Delete failed: Table TABLE id=OID has version DB_VERSION where object has version DO_VERSION.").

Sample code:

```
#values{true,false} default{false}
DatabaseManager.defaults.DeleteCheckVersion=true;
```

AutoWrite

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to true, the row will be written to the database when the update of another row occurs. This way all changes of the row will be accumulated and written together (operation called Aggregation). If parameter is set to true, the row will be written before every new query. If this parameter is set to false, or is not defined, no writing will be done until the transaction's commit.

Modifications of single DO (row in table) are optimized - aggregated at transaction level. Aggregation works for successive modifications of the same DO, provided no queries (in same transaction) were executed between them:

- INSERT/UPDATE - update won't be executed, instead corrected INSERT is performed,
- UPDATE/UPDATE - successive updates, result in one update execution only,
- UPDATE/DELETE - update won't be executed, only DELETE is performed,
- INSERT/DELETE - nothing is executed,
- INSERT/UPDATE/DELETE - nothing is executed, because from scenario 1 first two produce one INSERT, and scenario 4 produces nothing.

Either query or modification of other DO stops aggregation for previous one. DO.delete() may execute cascade delete java code which runs queries, and these queries will stop aggregation of modifications for the first DO.

Sample code:

```
# values{true,false} default{false}
DatabaseManager.defaults.AutoWrite=true
```

TransactionCaches

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to true, the transaction cache exists, otherwise does not exist. The transaction cache is HashMap of DOs (data objects).

Sample code:

```
# values{true,false} default{false}
DatabaseManager.defaults.TransactionCaches=true
```

AutoSave

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If AutoSave is true and a DO belongs to a transaction and DO.setXXX(value) is done, the DO modification is automatically saved into the transaction.

Delete of DO objects is always implicitly written to transaction.

New DOs created with createVirgin method is saved regarding to the AutoSaveCreateVirgin parameter.

If AutoSave is false, DO.save() is mandatory before the transaction's commit.

AutoSave has no meaning for DOs without transaction.

Sample code:

```
#values{true,false} default{false}
DatabaseManager.defaults.AutoSave=true
```

AutoSaveCreateVirgin

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If AutoSaveCreateVirgin is true and a DO belongs to a transaction and DO.createVirgin() is done, the created DO is automatically saved into the transaction.

If AutoSaveCreateVirgin is false, DO.save() is mandatory before the transaction's commit.

AutoSaveCreateVirgin has no meaning for DOs without transaction.

Sample code:

```
#values{true,false} default{false}
DatabaseManager.defaults.AutoSaveCreateVirgin=true
```

DefaultFetchSize

Possible values: integer values.

Default value: -1 (means that default fetch size is not used).

File: configuration file.

Context: DatabaseManager, Database, table.

Fetch size gives the JDBC driver a hint to the number of rows that should be fetched from the database when more rows are needed.

You can use methods from Query class to access this parameter value for one query:

- public void set_FetchSize (int iCurrentFetchSizeIn)
- public int get_FetchSize()

MainCacheLockTimeout

Possible values: long (milliseconds).

Default value: 100.

File: configuration file.

Context: DatabaseManager.

In unlikely case when transaction's thread dies before unlocking main cache (Wrapper), this timeout expiration allows next transaction to gain new lock on it.

CacheLockTimeout

Possible values: integer values.

Default value: 0.

File: configuration file.

Context: DatabaseManager, Database.

Time for transaction to acquire a lock on global cache (Wrapper), before it can hide certain DOs from cache. Hiding is necessary, because during commit, there is a point in time in which some DOs in the cache are becoming out of sync with database. After hidding, the lock is released, so other transactions could hide their own DOs.

CacheLockRetryCount

Possible values: integer values.

Default value: 0.

File: configuration file.

Context: DatabaseManager, Database.

This parameter contains number of transaction's retries caused by expired CacheLockTimeout while attempting to lock the Global Cache (Wrapper).

QueryTimeout

Possible values: integer values.

Default value: 0.

File: configuration file.

Context: DatabaseManager, Database, table.

Sets the number of seconds for which the query should be executed. If the limit is exceeded, an exception is thrown. Value 0 means that there is no limit.

In <table_name>Query class exist methods to access this parameter value:

- public void set_QueryTimeout (int iQueryTimeout)
- public int get_QueryTimeout()

SelectOids

Possible values: true/false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database, table.

This Parameter is used in mass Updates or Deletes (in xxxUpdate, xxxDelete classes). Value true means that before mass Updates(Deletes) SQL statement SELECT oid with the same WHERE clause will be executed, to pick a list of oids - which should be objects of mass Updates or Deletes.

Examples:

```
DatabaseManager.defaults.SelectOids=true
```

```
DatabaseManager.DB.<database_name>.SelectOids=true
```

IncrementVersions

Possible values: true/false.

Default value: true.

File: configuration file.

Context: DatabaseManager, Database, table.

This Parameter is used in mass Updates(in xxxUpdate). Value true means that values in version column are increment during mass Updates.

Examples:

```
DatabaseManager.defaults.IncrementVersions=true
```

```
DatabaseManager.DB.<database_name>.IncrementVersions=false
```

MaxConnectionUsages

Possible values: integer values.

Default value: -1.

File: configuration file.

Context: Connection.

Enhydra DODS uses a connection pooling mechanism. So, if a request-thread needs a database connection, it is taken from the pool instead of re-created every time. If the thread finishes, the connection is put back into the pool for re-use by another thread.

This Parameter is maximum number of how many times a connection can be re-used by a new thread (and put in the connection pool again) before it is closed and re-created (to avoid situation where connection holds too much memory but does not release these resources again because the connection is never closed).

MaxWaitingConnections

Possible values: integer values.

Default value: 10.

File: configuration file.

Context: Connection.

Enhydra DODS uses a connection pooling mechanism. So, if a request-thread needs a database connection, it is taken from the pool instead of re-created every time. If the thread finishes, the connection is put back into the pool for re-use by another thread.

This Parameter is maximum number of open (unused) waiting connection in pool (rest are closed on release).

initAllCaches

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager , Database.

If initAllCaches parameter is set to true (on DatabaseManager or Database level in configuration file), the default value of initialCondition parameter for all tables is "*". This default value can also be overriden if initialCondition for table is set to any value.

Examples:

```
DatabaseManager.defaults.cache.initAllCaches=true
```

```
DatabaseManager.DB.<database_name>.cache.initAllCaches=true
```

ChangeAutocommit

Possible values: true and false.

Default value: true.

File: configuration file.

Context: Database.

Disables DODS to change value of, database connection, 'autocommit' property.

Prior to version 3.23, MySQL does not support transactions, and therefore does not support explicit commit (they use autocommit by default after any SQL command). To use MySQL versions 3.22 and earlier, you have to make change to application configuration file. You will need to set parameter ‘ChangeAutocommit’, of logical database, to ‘false’ .

Example:

```
DatabaseManager.DB.<database_name>.ChangeAutocommit=false
```

MassUpdates and MassDeletes

Possible values: true and false.

Default value: false.

File: DOML file.

Context: Database, table.

When turned on, allows you to build data layer including two classes *xxxUpdate* and *xxxUpdate*. These classes provide QueryBuilder speed in massive update operations, while maintaining caches (both global and transaction) valid.

See chapter Mass Modifications.

UseCursorName

Possible values: true and false.

Default value: true.

File: configuration file.

Context: DatabaseManager, Database.

When turned on, allows DODS to use named cursors, this is default value. Some jdbc compliant drivers, like Microsoft JDBC driver (msjdbc), don't implement this feature and need to disable it (set parameter to false) to avoid runtime Exceptions.

Note

"UseCursorName"

This parameter can also be set in dbVendorConf.xml file see section UseCursorName in Chapter "Database Vendor and Driver Specific Parameters".

CaseSensitive

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager , Database, Table.

When turned 'on', DODS uses case sensitive version of database operation (on specified level).

ClassList

Default value: not set.

File: configuration file.

Context: Database.

During application startup, DODS reads "DODSClassList.xml" file, and for all data layer classes listed in this file initializes caches.

This parameter contains absolute path to "DODSClassList.xml" file.

This file contains list of data layer classes names that need to be initialized during application startup. If this parameter is not set then default file path is used.

If DODSClassList.xml does not exist even on application class path, data layer caches initialization will be skipped.

ClassType

Default value: depend on "JdbcDriver" parameter.

File: configuration file.

Context: Database.

This parameter in version prior of DODS 5.1-9 was mandatory. From DODS 5.1.8 and later version, if this parameter is not set, DODS will use value of "JdbcDriver" (driver class) to try to find database vendor name. If "JdbcDriver" is "unknown" to DODS (list of all driver that are "known" to dods are in dods.conf file), it will use "Standard" as value for database vendor name.

TransactionFactory

Default value: none (DODS will use StandardTransactionFactory).

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to full class name of class that implements AbstractDBTransactionFactory interface, DODS will use this class to create database transaction factory. If parameter is not set, DODS will use default AbstractDBTransactionFactory implementation - StandardDBTransactionFactory.

Current possible values (implementations classes) for this parameter are:

1. "com.lutris.appserver.server.sql.standard.StandardDBTransactionFactory" (default)
2. "org.enhydra.dods.dbtransaction.ExtendedTxFactory" - needed to support reordering tables modifications based on object relations defined in doml file. This implementation is created as an attempt of reducing deadlocks.
3. "org.enhydra.dods.jta.SyncDBTransactionFactory" - needed to support JTA standard (see the section describing JTA in DODS).

FullCacheCountLimit

Default value: -1 (not used).

File: configuration file.

Context: DatabaseManager,Database,Table.

If a table has more rows then specified in 'FullCacheCountLimit', the result shouldn't be recalculated in the memory at first time of excution. The Query has to be executed at the first time against the database. The result of this query can be cached and recalculated afterwards.

InitialDSCacheSize

Default value: -1 (not used).

File: configuration file.

Context: cache parameter on DatabaseManager,Database,Table level.

This parameter represents cache size during first loading data into cache - if initialCondition is enabled.

InitialCacheFetchSize

Default value: 0 (not used).

File: configuration file.

Context: cache parameter on DatabaseManager,Database,Table level.

This parameter represents value for fetch size which is used in query for cache initialization.

InitCachesResultSetType

Possible values: TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE.

File: configuration file.

Context: DatabaseManager,Database,Table level.

Result set type. This type will be used for ResultSets created by PreparedStatemens only for SQL queries which will be executed during cache initialization. Default value is defined in dbVendor conf file - driver section - if not TYPE_FORWARD_ONLY.

InitCachesResultSetConcurrency

Possible values: CONCUR_READ_ONLY or CONCUR_UPDATABLE.

File: configuration file.

Context: DatabaseManager,Database,Table level.

Concurrency type: This type will be used for ResultSets created by PreparedStatemens only for SQL queries which will be executed during cache initialization. Default value is defined in dbVendor conf file - driver section - if not CONCUR_READ_ONLY.

DisableConnectionPool

Default value: false.

File: configuration file.

Context: Connection.

This parameter is used only if ConnectionAllocator parameter is set to "com.lutris.appserver.server.sql.datasource.DataSourceConnectionAllocator" or to "com.lutris.appserver.server.sql.datasource.SimpleDataSourceConnectionAllocator".

If this parameter is set to "true", DODS will disable internal connection pool.

DataSourceName

Default value: none.

File: configuration file.

Context: Connection.

Parameter defines JNDI name (in "jndi:<dataSourceName>" formath) of externally defined DataSource that can be used by DODS to establish connection to database.

This parameter is used by DODS only if ConnectionAllocator parameter is set to:

1.) "com.lutris.appserver.server.sql.datasource.DataSourceConnectionAllocator", or

2.)"com.lutris.appserver.server.sql.datasource.SimpleDataSourceConnectionAllocator"

Example:

```
DatabaseManager.DB.<database_id>.Connection.DataSourceName = jndi:<DataSourceName>
```

Note

See section called "Using DODS with javax.sql.DataSource" and reference for "ConnectionAllocator" parameter.

ConnectionFactory

Default value: "com.lutris.appserver.server.sql.StandardDBConnectionFactory".

File: configuration file.

Context: Connection.

If this parameter is set to full class name of class that implements AbstractDBConnectionFactory interface, DODS will use this class to create database connection factory. If parameter is not set, DODS will use default AbstractDBConnectionFactory implementation - StandardDBConnectionFactory.

Current possible values (implementations classes) for this parameter are:

1. "com.lutris.appserver.server.sql.StandardDBConnectionFactory" (default),
2. "com.lutris.appserver.server.sql.SimpleDBConnectionFactory" - needed to support "SimpleConnectionAllocator" (see "ConnectionAllocator" parameter reference),
3. "com.lutris.appserver.server.sql.DataSourceDBConnectionFactory" - needed to support "DataSourceConnectionAllocator" (see "ConnectionAllocator" parameter reference),
4. "com.lutris.appserver.server.sql.SimpleDataSourceDBConnectionFactory" - needed to support "SimpleDataSourceConnectionAllocator" (see "ConnectionAllocator" parameter reference).

Example:

```
DatabaseManager.DB.<LogicalDatabaseName>.Connection.ConnectionFactory =
 "com.lutris.appserver.server.sql.StandardConnectionAllocator"
```

ConnectionAllocator

Default value: "com.lutris.appserver.server.sql.standard.StandardConnectionAllocator".

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to full class name of class that implements ExtendedConnectionAllocator interface, DODS will use instance of this class as a connection pool.

Current possible values (implementations classes) for this parameter are:

1. "com.lutris.appserver.server.sql.standard.StandardConnectionAllocator" (default) - faster, in many occasions,

2. "com.lutris.appserver.server.sql.standard.SimpleConnectionAllocator" - usually slower but can be faster then "StandardConnectionAllocator" in some situations with very high transactions allocate/ deallocate frequency. To use "SimpleConnectionAllocator" in application, "ConnectionFactory" parameter needs to be set to "com.lutris.appserver.server.sql.SimpleDBConnectionFactory" (see "ConnectionFactory" parameter reference),

3. "com.lutris.appserver.server.sql.datasource.DataSourceConnectionAllocator" - this implementation is "StandardConnectionAllocator" clone that internally uses DataSource (javax.sql.DataSource) as connection source. To use "DataSourceConnectionAllocator" in application, "ConnectionFactory" parameter needs to be set to "com.lutris.appserver.server.sql.DataSourceDBConnectionFactory" (see "ConnectionFactory" parameter reference),

4. "com.lutris.appserver.server.sql.datasource.SimpleDataSourceConnectionAllocator" - this implementation is "SimpleConnectionAllocator" clone that internally uses DataSource (javax.sql.DataSource) as connection source. To use "SimpleDataSourceConnectionAllocator" in application, "ConnectionFactory" parameter needs to be set to "com.lutris.appserver.server.sql.SimpleDataSourceDBConnectionFactory" (see "ConnectionFactory" parameter reference).

Example:

```
DatabaseManager.DB.<LogicalDatabaseName>.ConnectionAllocator=
 "com.lutris.appserver.server.sql.standard.StandardConnectionAllocator"
```

Note

See also section called "Using DODS with javax.sql.DataSource" and parameter "DataSourceName" reference.

ConnectionIdleTimeout

Default value: none (DODS will not track time that connection spend in pool).

File: configuration file.

Context: Connection.

This parameter represents maximum time that connection can stay idle in connection pool. If connection stays longer, DODS will close that connection and throw it from connection pool.

RollbackOnReset

Default value: false

File: configuration file.

Context: Connection.

Makes posibly additional rollback on JDBC connection inside reset method - prior to putting connection back into pool.

QueryCacheImplClass

Default value: none (DODS will use QueryCacheImpl class).

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to full class name of class that extends abstract class DataStructCache, DODS will use this class to create data struct cache for xxxDO class If parameter is not set, DODS will use class QueryCacheImpl as a default.

SQLBatch

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to “true”, DODS will use SQLBatch, feature of JDBC PrepareStatement Object, during database operations.

Warning

If you set this parameter to *true*, and parameter **.Connection.MaxPreparedStatements* has low value, you may get errors at runtime - JDBC driver complaining about closed statements.

SQLBatch depends heavily on `PreparedStatements` cache (whose size is set with MaxPreparedStatements), and if it's not big enough some of batched statements may be closed by cache algorithm.

CaseInsensitiveDatabase

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to “true”, DODS will assume that database is case insensitive and will not use "LOWER" function in SQL statements (patch for MS Access).

ShutDownString

Default value: none (DODS will not use "ShutDownString").

File: configuration file.

Context: Connection.

Added to support new version of HSQL database (1.7.2-7 and newer). If this parameter is set to some value, than during application shutdown, DODS will send this string to database (as SQL command).

To stop HSQL (1.7.2-7 ...) database on application shutdown, and to force HSQL engine to write data to database (file) set this value in application configuration file to value "SHUTDOWN" or "CHECKPOINT".

Example:

```
DatabaseManager.DB.<databaseName>.Connection.ShutDownString = "SHUTDOWN"
```

NextWithPrefix

Default value: false.

File: configuration file.

Context: ObjectId.

f this parametre set to true, following statament will be used for update object id table: update OID_TABLE set OID_TABLE.next = ? where OID_TABLE.next = ?. Default value for this parameter is false.

Example:

```
DatabaseManager.DB.<databaseName>.ObjectId.NextWithPrefix = true
```

OidTableName

Default value: "objectid".

File: configuration file.

Context: ObjectId.

Name of table in database that contains next available value for "oid".

Example:

```
DatabaseManager.DB.<databaseName>.ObjectId.OidTableName = "objectid"
```

NextColumnName

Default value: "next".

File: configuration file.

Context: ObjectId.

Column name in <OidTableName> that contains next available value for "oid".

Example:

```
DatabaseManager.DB.<databaseName>.ObjectId.NextColumnName = "next"
```

NextColumnType

Default value: "BigDecimal".

File: configuration file.

Context: ObjectId.

Type of column in <OidTableName> that contains next available value for "oid".

Example:

```
DatabaseManager.DB.<databaseName>.ObjectId.NextColumnType = "BigDecimal"
```

QueryTimeLimit

Possible values: integer values.

Default value: 0.

File: configuration file.

Context: DatabaseManager, Database, table.

Sets the number of milliseconds for which the Query should be executed and the resulset read from ResultSet. If the limit is exceeded, an Exception is thrown. Value 0 means that there is no limit.

PrimaryLogicalDatabase

Default value: not used.

File: configuration file.

Context: ObjectId.

In case of multibase DODS application, it is possible to tell DODS to use unique (only from one logical database) objectId table to establish unique objectId(OID) across all logical databases. If "PrimaryLogicalDatabase" parameter is set to logical database name that is defined in current configuration, DODS will use objectid table from that database to create oids for all defined logical databases. If this parameter is not set, DODS will use separate objectId table from every database (one placed in that particular database).

Example:

```
DatabaseManager.DB.<databaseName>.ObjectId.PrimaryLogicalDatabase = <LogicalDatabaseName>
```

DODSCacheFactory

Default value: "org.enhydra.dods.cache.lru.DODSLRUCacheFactory".

File: configuration file.

Context: cache parameter on DatabaseManager and LogicalDatabase level.

If this parameter is set to full class name of class that implements DODSCacheAbstractFactory interface, DODS will use this class to create internal, cache, data storage objects used in DataStructCache, QueryCache and TransactionQueryCache implementation to store cache items.

Current possible values (implementations) for this parameter are:

1. "org.enhydra.dods.cache.lru.DODSLRUCacheFactory" (default) - uses LRU cache mechanism, java1.3 compatibile,
2. "org.enhydra.dods.cache.hash.DODSLinkedHashCacheFactory" - uses LinkedList (unordered), java1.4 compatible and less synchronized (faster).

Examples:

```
DatabaseManager.defaults.cache.DODSCacheFactory =
 "org.enhydra.dods.cache.lru.DODSLRUCacheFactory"
```

or

```
DatabaseManager.DB.<LogicalDatabaseName>.cache.DODSCacheFactory =
 "org.enhydra.dods.cache.lru.DODSLRUCacheFactory"
```

ConnectionAllocateCheckSql

Default value: none (DODS will not use "ConnectionAllocateCheckSql").

File: configuration file.

Context: Connection.

The SQL query statement that will be used to validate connections from connection pool, just before returning them. If specified, this query MUST be valid SQL SELECT statement for current database/driver.

Example:

```
DatabaseManager.DB.<databaseName>.Connection.ConnectionAllocateCheckSql = "select 1"
```

MaxPreparedStatements

Default value: 0 (DODS will not use PreparedStatement Cache).

File: configuration file.

Context: Connection.

Size of PreparedStatement Cache. If this parameter is set to 0 (null), PreparedStatement Cache is disabled.

Example:

```
DatabaseManager.DB.<databaseName>.Connection.MaxPreparedStatements = 50
```

Note

If you use PreparedStatement cache in your configuration and connection pooling, you should also use parameter *MaxConnectionUsages* to define how many times a connection can be re-used by a new thread (and put in the connection pool again).

AllocationScope

Values: "table", "database".

Default value: "table".

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to "database" (default), DODS generates ObjectId values that are unique in the scope of whole database. If parameter is set to "table", generated ObjectId values are unique in the scope of every single table, but distinct tables can contain same ObjectId values.

Example:

```
DatabaseManager.DB.<databaseName>.ObjectId.AllocationScope = "table"
```

ObjectIdAllocationSource

Values: "dods", "database".

Default value: "dods".

File: configuration file.

Context: DatabaseManager, Database, Table.

If this parameter is set do "dods", DODS will generate values for ObjectId according to rule specified by "AllocationScope" parameter value. If "ObjectIdAllocationSource" is set to "database", DODS will try to use "autogenerated columns" capabilities of database engine to get unique ObjectId values.

Example:

```
DatabaseManager.defaults.ObjectIdAllocationSource = database
```

asynchLoadThreadNum

Possible values: integer values.

Default value: 0.

File: configuration file.

Context: DatabaseManager.

Defines the number of threads used for asynchronous cache load during application startup.

Example:

```
DatabaseManager.defaults.asynchLoadThreadNum = 3
```

asynchLoadPriority

Possible values: integer values.

Default value: -1.

File: configuration file.

Context: Table.

Defines the order of caches that will be asynchronous initialized on cache startup. The lower the value, the sooner the cache will be asynchronous loaded for the table during application startup.

Example:

```
DatabaseManager.DB.sid1.PERSON.cache.asynchLoadPriority = 5
```

ClassName

Default value: "com.lutris.appserver.server.sql.standard.StandardObjectIdAllocator".

File: configuration file.

Context: ObjectId.

If this parameter is set to full class name of class that implements ObjectIdAllocator interface, DODS will use this class to create object id allocator that manages the allocation of unique object ids. If parameter is not set, DODS will use default ObjectIdAllocator implementation - StandardObjectIdAllocator.

Current possible values (implementations classes) for this parameter are:

1. "com.lutris.appserver.server.sql.standard.StandardObjectIdAllocator" (default)
2. "org.enhydra.dods.jta.JTAObjectIdAllocator" - needed to support ObjectIdAllocator in JTA environment" (see the section describing JTA in DODS).

Example:

```
DatabaseManager.DB.<LogicalDatabaseName>.ObjectId.ClassName="org.enhydra.dods.jta.JTAObjectIdAllocator"
```

JTA

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If this parameter is set to “true”, DODS will provide that DBTransaction objects are aware of JTA environment and obey TransactionManager. It will also make generated Query and Data Objects work inside *UserTransaction* implicitely acquiring SyncDBTranscation instance to work within.

Example:

```
DatabaseManager.defaults.JTA=true
```

For more information, see section describing JTA in DODS.

cachePersistenceRoot

Default value: not set.

File: configuration file.

Context: DatabaseManager

This parameter contains absolute path to directory where cache will be saved.

cachePersistenceOnShutdown

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager

If cachePersistenceOnShutdown parameter is set to true (on DatabaseManager level in configuration file) and cachePersistenceRoot parameter is set, than during application shutdown, DODS will save the cache.

This saved cache will be loaded during application startup. This value can be overriden(set to false) if cachePersistenceRoot is not set.

Examples:

```
DatabaseManager.defaults.cache.cachePersistenceOnShutdown=true
```

cachePersistenceOnDisableCaching

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager

If cachePersistenceOnDisableCaching parameter is set to true (on DatabaseManager level in configuration file) and cachePersistenceRoot parameter is set,than when DO.disableCaching() is done, the cache is saved. This saved cache will be loaded when DO.enableCaching() is done. This value can be overriden(set to false) if cachePersistenceRoot is not set.

Examples:

```
DatabaseManager.defaults.cache.cachePersistenceOnDisableCaching=true
```

cursorless

Possible values: true and false.

Default value: false.

File: configuration file.

Context: DatabaseManager, Database.

If cursorless is true, DODS will ignore all custom settings on PreparedStatement to avoid cursors on servers side. In this case, CachedPreparedStatementExt class which override methods setMaxRows and setFetchSize is used instead of CachedPreparedStatement class.

defaultMaxRows

Possible values: integer values.

Default value: -1 (means that default max rows is not used).

File: configuration file.

Context: DatabaseManager, Database, table.

The maximum number of rows that can be present in a result set.

You can use methods from Query class to access this parameter value for one query:

- public void setMaxRows(int maxRows)
- public int getMaxRows()

Chapter 23. Database Vendor and Driver Specific Parameters

Parameters that are explained in this section are placed in database vendor configuration files (in build/conf/<dbVendorName>Conf.xml files and also in dbmanager.jar file as part of 'org.enhydra.dods.conf' package).

This parameters tune-up DODS to work with distinct database vendors and jdbc drivers, and all parameters are not appropriate for all vendors and drivers.

Some of this parameter can also be set in application *.conf file, in that case they override values defined in vendor configuration file.

To set some of database vendor specific parameters you simply go to appropriate build/conf/<dbVendorName>Conf.xml file and set value to tag:

```
<paramName>paramValue</paramName>
```

To set some of driver specific parameters you need to follow some syntax rules:

```
<Drivers>
   <fullClassNameOfJDBCDriver>
      <ParamName>ParamValue</ParamName>
      ...
   </fullClassNameOfJDBCDriver>
   … …
   <fullClassNameOfJDBCDriver>
      <ParamName>ParamValue</ParamName>
      ...
   < /fullClassNameOfJDBCDriver>
   ... …
</Drivers>
```

Example:

```
<Drivers>
   <com.microsoft.jdbc.sqlserver.SQLServerDriver>
      <UseCursorName>false</UseCursorName>
      <OrderedResultSet>withPrefix</OrderedResultSet>
   </com.microsoft.jdbc.sqlserver.SQLServerDriver>
</Drivers>
```

Parameter are set on jdbc driver level (can be distinct for distinct driver on same database engine).

UseCursorName

Possible values: true, false

Default value: true

File: dbVendorConf.xml

Context: Drivers

When turned on, allow DODS to use named cursors, this is default value. Some jdbc compliant drivers, like Microsoft JDBC driver (msjdbc), don't implement this feature and need to disable it (set parameter to false) to avoid runtime Exceptions.

Note

This parameter can also be set in app.conf file see UseCorsorName section in Chapter 22.

SplitSQLPrimary

Possible values: true, false

Default value: true

File: dbVendorConf.xml

Context: vendor

When turned on , if splitSQL is enabled, allow DODS to remove (split in separate file 'xxxPrimary.sql') 'PRIMARY' clause (from generated 'CREATE TABLE' statement). If turned off (and splitSQL is enabled) xxxPrimary.sql file will still be generated, but 'PRIMARY' clause will be not removed from 'CREATE TABLE' statement

OrderedResultSet

Possible values: oldStyle, withPrefix ,noPrefix

Default value: oldStyle

File: dbVendorConf.xml file

Context: Drivers

Fix problem with some JDBC compilant drivers (msjdbc) that use order relevant column sequence in returned ResultSet.

To avoid this problem DODS now use SQL SELECT statements with explicit column names.

- oldStyle - DODS don't use explicit column names in SQL SELECT statements (default).
- withPrefix - DODS explicit use column names with table name as prefix (in SQL SELECT statements).
- noPrefix - DODS explicit use column names with no prefix (in SQL SELECT statements).

DisableFetchSizeWithMaxRows

Possible values: true, false

Default value: false

File: dbVendorConf.xml file

Context: Drivers

Disable DODS to use PreparedStatement FetchSize feature, if MaxRows parameter is set in application conf. file. Added to avoid problems with some.

ResultSetType

Possible values: TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE

File: dbVendorConf.xml file

Context: Drivers

Result set type. This type will be used for ResultSets created by PreparedStatemens

ResultSetConcurrency

Possible values: CONCUR_READ_ONLY or CONCUR_UPDATABLE

File: dbVendorConf.xml file

Context: Drivers

Concurrency type: This type will be used for ResultSets created by PreparedStatemens.

WildcardEscapeClause

Possible values: database dependent, none

File: dbVendorConf.xml file

Context: dbVendorConf.xml

Define SQL wildcard clause that will be added to SQL query statements generated by DODS. To disable adding this clause to generated statements set parameter value to none. .

Example:

```
<WildcardEscapeClause>ESCAPE '&#167;'</WildcardEscapeClause>
```

Example:

```
<WildcardEscapeClause>none</WildcardEscapeClause>
```

SetNullAsVarchar

Possible values: true, false

Default value: false

File: dbVendorConf.xml file

Context: Drivers

Tell DODS to explicitly use "Varchar" data type when PreparedStatement parameters to "Null" value (patch for MS Access).

SetBytesAsLongvarbinary

Possible values: true, false

Default value: false

File: dbVendorConf.xml fil

Context: Drivers

Tell DODS to explicitly use "Longvarbinary" data type when set's data for PreparedStatement parameters of "bytes" type (patch for MS Access).

CustomNotEqualSqlOperator

Possible values: database dependent, none

Default value: "!="

File: dbVendorConf.xml file

Context: Drivers

Tell DODS to use alternative (user defined) operator for "Not Equal" in SQL expressions (patch for MS Access).

SetBytesAsBinaryStream

Possible values: true, false

Default value: false

File: dbVendorConf.xml file

Context: Drivers

Tell DODS to explicitly use Binary Stream when set's data for PreparedStatement parameters of "bytes" type (patch for MS Access).

SetBooleanAsString

Possible values: true, false

Default value: true

File: dbVendorConf.xml file

Context: Drivers

Tell DODS to explicitly use String when set's data for PreparedStatement parameters of "Boolean" type (patch for PostgreSQL).

UsePrefixWithUpdate

Possible values: true, false

Default value: true

File: dbVendorConf.xml file

Context: Drivers

Tell DODS to explicitly use full column name (with table name prefix eg. TableName.ColumnName) or just column name, during QueryBuilder.addUpdateColumn(RDBColumn column, Object value) method call (patch for PostgreSQL).

EnableCreateStatistics

Possible values: true, false

Default value: false

File: dbVendorConf.xml file

Some vendors support creating statistics for tables, views. In SQL Server 2005, server can creates a histogram and associated density groups (collections) over the supplied column or set of columns of a table or indexed view. Default value for EnableCreateStatistics parameter is false.

CreateStatistics

Possible values: Beginning part of SQL statement for creating statistics

File: dbVendorConf.xml file

In SQL Server 2005, syntax is 'CREATE STATISTICS'.

Example:

```
CREATE STATISTICS stat_employeeName ON Computers(computerName, compOwner)
```

In Oracle, syntax is 'ASSOCIATE STATISTICS WITH COLUMNS'.

Example:

```
ASSOCIATE STATISTICS WITH COLUMNS Computers.computerName, Computers.compOwner USING NULL
```

NamedStatistics

Possible values: true, false

File: dbVendorConf.xml file

Some vendors support named statistics. If set to true, DODS generates SQL statement for creating statistics with name of the statistics. In SQL Server 2005, you have to set this value to true.

FullColumnNames

Possible values: true, false

File: dbVendorConf.xml file

If set to true, DODS generates SQL statement for creating statistics with full column name (with table name prefix eg. TableName.ColumnName)

SupportAttribs

Possible values: true, false

File: dbVendorConf.xml file

Some vendors support attribs(options) for statistics. If set to true, DODS generates SQL statement with attribs.

EndString

Possible values: Ending part of SQL statement for creating statistics

File: dbVendorConf.xml file

Define clause that will be added at the end of SQL statements (patch for Oracle). Supported value is 'USING NULL'.

IncludeIndexColumns

Possible values: true, false

Default value: false

File: dbVendorConf.xml file

Some vendors support to INCLUDE additional columns in index, so on that way you can extend your indexes with additional columns In SQL Server 2005 and DB2, you can extend the functionality of nonclustered indexes by adding nonkey columns to the leaf level of the nonclustered index. By including nonkey columns, you can create nonclustered indexes that cover more queries. In DB2, INCLUDE can only be specified for indexes that are defined with UNIQUE. Default value for IncludeIndexColumns parameter is false.

DriverDependenciesClass

Possible values: Name of class that implements *DriverDependencies* interface

Default value: "org.enhydra.dods.dependencies.StandardDriverDependencies"

File: dbVendorConf.xml file

Context: Drivers

According to JDBC 3.0 specification the way to work with "autogenerated columns" is standardized on database (JDBC) driver level, but many today available jdbc drivers have incomplete implementation of JDBC 3.0 or are not JDBC 3.0 compliant at all.

To support "autogenerated columns" in production environment DODS use separate implementation of driver DriverDependencies interface for each database driver/vendor.

Full name of class that implements DriverDependencies interface for concrete database driver/vendor is determined by "DriverDependenciesClass" parameter which is placed inside "Driver" section of DODS database vendor configuration file.

Current implementations:

```
org.enhydra.dods.dependencies.MsjdbcDriverDependencies
org.enhydra.dods.dependencies.PgSqlDependencies
org.enhydra.dods.dependencies.StandardDriverDependencies
```

UseTopSyntax

Possible values: true, false

Default value: false

File: dbVendorConf.xml file

Context: Drivers

If set to true, DODS generates "TOP" keyword in SQL statement with value defined by maxDBrows attribute of Query object. With "TOP" keyword is defined maximum numbers of rows that can be retrieved from database (patch for Microsoft SQL Server 2005).

For example, if attribute maxDBrows is set to value 25, and <UseTopSyntax> tag of dbVendorConf.xml file is set to true, DODS will generate SQL statement that begins like:

```
SELECT TOP 25 * from . . .
```

Chapter 24. Transactions

A transaction exactly belongs to one database.

DOs in Transactions

Every DO should belong to a transaction. DOs without transaction exist only for the compatibility with the old DODS.

If a DO belongs to a transaction, the transaction in the DO can not be changed, and the DO can only be used in this transaction. Otherwise, DODS throws an exception.

A DO that belongs to no transaction can not be "attached" to a transaction. This kind of a DO at least belongs to a database and can only be used in this database. Otherwise, DODS throws an exception.

A DO has getTransaction() method that returns its own transaction if has one, or returns null otherwise.

Status of DOs

The status of a DO can be checked with the following methods:

- isDeleted()

  Returns true if the object is deleted in the transaction, andh the transaction was not yet committed.

- get_transaction()

  Returns transaction to which the DO belongs to. If the DO does not have a transaction, NULL is returned.

- isLoaded()

  Returns true if DO's data is loaded from database (DO is not empty), otherwise false.

- isReadOnly()

  Return true if the data for this DO has been marked read-only, otherwise false.

- isVirgin()

  Returns true for a DO that is created virgin, and hasn't been commited.

- isDirty()

  Returns true if this object has been modified (needs to be updated to the database).

Creating DOs

If a row is read from the database (or from the cache) and the DO is created in memory using method with a transaction parameter, the DO is cached in the transaction's DO cache, if exists.

If a row is read from the database (or from the cache) and the DO is created in memory using method without a transaction parameter, the DO is not cached in the transaction DO cache. This way, DODS can not guarantee that if the same row is read again, the same DO will be returned from the transaction cache (it would always be a different DO instance in memory) and data could be inconsistent (because already changed in the database).

In new DODS are added new methods for memory creation of DOs for an existing database row (createExisting), for creating new DOs (createVirgin) and DO constructors that support transaction parameter. The DOML attribute DirtyDOs defines whether the old ("dirty") methods (methods without transaction parameter) will be generated or not. If this parameter is set to "Compatible", "dirty" methods will be generated (as before), if set to "Deprecate", "dirty" methods will be generated as deprecated, and if set to "Omit", "dirty" methods will not be generated at all.

Only for methods that support multi databases are not added new methods with transaction parameter because the transaction contains the information about the database, so the methods with the transaction parameter work only with transaction's database.

Using DOs

A DO that belongs to a transaction can be used together with DOs that belong to no transaction (setXXX(DOyyyy)).

A DO that belongs to no transaction can be used together with DOs that belong to a transaction.

Save and Delete Operations in Transactions

A DO that belongs to a transaction can only be saved into / deleted in its own transaction. Otherwise, DODS throws an exception. Even if the DO is saved /deleted without a transaction parameter, it is saved into / deleted in its own transaction. A DO that belongs to no transaction can be saved into / deleted in an explicit transaction. A DO without a transaction that is saved /deleted without a transaction parameter is saved into / deleted in an implicitly created transaction.

Sample code:

```
DBTransaction  dbTrans = DODS.getDatabaseManager().createTransaction();
PersonDO person1 = PersonDO.createVirgin(dbTrans);
person1.setFirstname("Person1Name");
person1.setLastname("Person1LastName");
person1.setLogin("p1");
person1.setPassword("p1");
person1.save();
```

Insert, Update and Delete Operations on the Database

A DO that is newly created (insert operation) with a transaction parameter, holds its transaction information.

A DO that is newly created (insert operation) without a transaction parameter is created in an implicit DB transaction, but DO holds no transaction in the memory.

If DOs are deleted with DO.delete(), they are marked as deleted (DO attribute deleted set to true). These DOs can not be returned any more from the local transaction DO cache, and methods DO.getXXX() throw an exception (because the DO is already deleted in the transaction).

Added new methods: unDelete() and unDelete(DBTransaction dbt). These methods undelete deleted DO by setting DO's attribute deleted to false and inserting the DO to the database.

If the unDelete method is called after cascade delete, only the root DO is undeleted.

Added new method: undo(). This method returns DO in the state in which it had been before the last commit, if there was any commit performed. If there was not any commit performed, DO is returned in the "empty" state (it has no data loaded).

The use of undo() method has only sense if the DO belongs to a transaction. It this method is called for a DO without a transaction, the method throws DataObjectException with a message indicating that an error has occured during the undo operation.

Repeated calls of undo() method must be separated with at least one call of commit method.

Old DODS code always updated rows, making SQL Update statement that contained all columns of a DO. This kind of operation tends to be a bit inefficient, when only a few of a dozen of columns have been changed.

Worst case scenario is: only one column changed, yet all columns get updated.

New feature in DODS is that a DO makes update statement that only includes columns that have been changed. For every column there is new boolean data member of <table_name>DO class: changedColumnName. Its value represents current state of that column. If it is true, column needs update, otherwise it doesn't.

Methods for writing to the database (insert, update and delete) reset these values to false, while methods for changing column value set appropriate changedColumnName to true.

All updates of the same DO (that are done one after another) are aggregated into one update. When insert, update or delete of another DO occurs, the aggregation is finished. If AutoWrite is true, the agregated update is performed, the DO version is increased. For AutoWrite set to false, DO remains waiting (in transaction) for explicit write() or commit().

Sample code:

```
DBTransaction dbTrans = DODS.getDatabaseManager().createTransaction();
DiscDO disc1 = DiscDO.createVirgin(dbTrans);
disc1.setArtist("Artist1");
disc1.setTitle("Disc test1");
disc1.setGenre("pop1");
disc1.setOwner(person1);
disc1.setIsLiked(false);
disc1.save(); person1.delete();
/*undelete PersonDO */
person1.unDelete();
```

Queries

In new DODS are added new Query constructors that support transaction parameter. The DOML attribute DirtyDOs defines whether the old ("dirty") constructors (constructors without transaction parameter) will be generated or not. If this parameter is set to "Compatible", "dirty" constructors will be generated (as before), if set to "Deprecate", "dirty" constructors will be generated as deprecated, and if set to "Omit", "dirty" constructors will not be generated at all.

Queries with a transaction create all DOs within this transaction. Oppositely, Queries without transactions create DOs without transactions.

Sample code:

```
<table id="discRack.data.person.Person" dirtyDOs="Omit" dbTableName="person">
    <column id="login" usedForQuery="true">
       <type dbType="VARCHAR" javaType="String"/>
    </column>
    ..........................................
    <column id="lastname" usedForQuery="true">
          <type dbType="VARCHAR" javaType="String"/>
    </column>
</table>
```

Caching

A local transaction has a DO transaction cache. This cache is a HashMap.

The local transaction cache can be switched "on" or "off" in configuration file (attribute TransactionCaches, default value is "off"). So, transactions are also able to work without local DO cache.

When the transaction starts, all caching activities whose result forms DO modifications (create, delete, modify) must be done locally in the transaction cache. The reading activities are first done in the local transaction DO cache. If the DO is not found there, it is searched in the DataStruct cache, and if the DO is not find even there, it is retrieved from the database. After the transaction's commit (explained in the next chapter), the DataStruct cache is re-synchronized and query caches are updated (with the changes performed in the transaction).

After first transaction.write() operation, query caches can not be used in this transaction any more (they are not consistent with the transaction). The transaction.write() operation includes writes caused by parameter AutoWrite when set to true.

Commit of Transactions

When transaction's commit() starts, there is a certain point in time when other transactions can start to see changes in the database (if database does not allow dirty reads, otherwise they could see changes before that). So, exactly at this point in time the DataStruct and Query caches would be out of synchronization with the database.

Following the principle that the caches always have to show the same data as if it was accessed the database, it should be avoided that another transaction can read or modify (add entries to the DataStruct cache) until database commit is completed and DataStruct cache and Query caches re-synchronized with the database (DataStructs of modified DOs replaced, simple queries re-evaluated, complex and multi-join queries removed).

The cache re-synchronization can only happen after the successful commit, because there could be errors in the database during the commit and there should not be inconsistent cache after such a failure. And the commit() can take some time...

This problem with transaction's commit() is solved with the Global Cache (Wrapper) and the Negative lists in the DataStruct caches.

The Global cache is a Singleton. It contains and synchronizes all applications DataStruct caches. It has the following methods:

- getInstance() - static method

Returns Wrapper object if exists, otherwise creates it and returns it.

- registerCache(DataStructCache dc)

Registers (adds) QueryCacheImpl cache (implementation of all caches per table) to the Wrapper.

- lock() - synchronized method

Returns 0 if the Wrapper is already locked. If not, locks all DataStruct caches (so that they can not be used) and query caches if needed, and returns time (in ms) when this lock expires. This way nobody can lock Wrapper indefinitely.

- unlock() - synchronized method

Unlocks DataStruct caches (they can be used again) and query caches if needed.

The negative list is contained in every DataStruct cache. It blocks access (read and modify) of just some parts of the caches. It contains DataStructs that are in the cache, but at the moment can not be read from (and modified in) the cache because that DataStructs may not be consistent with the database. So, the DataStructs that are in the negative list are not visible for read and modify methods. Since more than one transaction can add DataStructs to the negative list, the list counts the number of times (for every DataStruct) the DataStruct was made invisible. When the counter becomes zero, the DataStruct object is removed from the negative list.

The transaction's commit() method uses the Wrapper and the negative lists of DataStructs caches in the following way:

- makeQueryCachesInvisible()

Locks all query caches (simple, complex and multi-join) for all classes (tables) whose DOs are modified in transaction. QueryCaches are locked before commit to database until cache re-evaluation.

- makeQueryCachesVisible()

Unlocks Query caches.

The negative list is also used for locking QueryCaches similarly to DataStructCache.

- After the successful executed inserts, updates and deletes of DOs, the Wrapper must be locked, so that used DOs would be hidden for use (put in the negative lists in DataStructs caches). Due to a possible changes of DOs, the query caches are also hidden for use.
- If the Wrapper is already locked, the method waits for CacheLockTimeout time, CacheLockRetries number of times (two parameters in the application's configuration file). If even after that number of tries with that amount of time the Wrapper stays locked, the method throws SQL Exception with the message suggesting that the method could not wait any more and the rollback() is performed.
- If the method managed to lock the Wrapper, it makes invisible DOs that were changed in the DataStructs cache (it puts them in the DataStruct cache negative list) and makes invisible query caches. After that, the Wrapper is unlocked and the DataStruct caches can be used again (except some cache parts - invisible DataStructs that are in the negative lists can't be used).
- Then, the transaction is committed.
- If an exception occurred during the commit, the rollback() is performed and the used DOs are reloaded from the database.

- If the database commit was successful, all objects are notified that the transaction succeeded and the changed DOs (DataStructs) and changed cached queries are written back to the global cache (changes are re-evaluated in queries).
- No matter the commit was successful or not, the update of negative lists must be performed. For this, the locking of the Wrapper is again needed (DataStruct caches synchronization is needed).
- If the Wrapper had been locked before, the method must wait until it becomes unlocked, and then locks it again, updates the negative lists (the DataStruct objects that were put to negative lists by this method are removed from there) and makes query caches again visible. After the update of the negative lists, the Wrapper is unlocked again.
- When a transaction is committed, the DataStruct cache is re-synchronized with the database, in all DOs is attribute dirty set back to false, DataStruct objects are moved from data to originalData pointers, newly created, rows/DOs are set to "existing" (can be derived because originalData was null before).

New method DO.doLock() is added: a DO can get locked (even if no data is changed). This way a row that is not updated at all can still be ensured that will not be changed in the database till the commit (pessimistic locking). It gets executed against database immediately, with no regard for AutoWrite parameter.

New method DO.doTouch() is added: a DO can get locked (even if no data is changed). This way a row that is not updated at all can still be ensured that won't be changed in the database till the commit (pessimistic locking). It gets executed against database immediately, with no regard for AutoWrite parameter, and increments version.

New method DO.doCheck() is added: it marks a DO for locking just before the commit. This provides that this row will not be changed during the commit (optimistic locking). This type of locking is executed in commit() method and locks DOs which were marked (for locking) and modified in this transaction.

Extended Trasaction

Prolonged transaction times, enforced by *DirtyDO="Omit"* option introduced in **DODS v5.1**, brought in another DB feature into our focus - database locks, and their result *deadlocks*.

In attempt to reduce deadlocks happening, application should honor order in tables modifications. This DBTransaction type should help coping with the task.

Using alredy described configuration feature, you specify

```
DatabaseManager.DB.<LogicalDatabaseName>.TransactionFactory=org.enhydra.dods.dbtransaction.ExtendedTxFactory
```

and you're set to go.

This type has mandatory TransactionCache, and different collection algorithm to standard implementation.

First attempt of using JTA API Implementation in DODS

First version of JTA in DODS created transactions that implemented *javax.transaction.xa.XAResource* interface. This interface is a java mapping of the industry standard XA interface based on the X/Open CAE Specification (Distributed Transaction Processing: The XA Specification).

This solution brought hierachy problems between DODS and XADataSource connections. Instead of this implementation, new version of JTA in DODS created transactions that implemented *javax.transaction.Synchronization* interface. This implementation is explained in the next section.

Using XATransaction in DODS

To use JTA API Transaction implementation in DODS (XATransaction) first step is to set TransactionFactory parameter in app. configuration file to "org.enhydra.dods.xa.XATransactionFactory" :

```
DatabaseManager.DB.<LdbName>.TransactionFactory  = "org.enhydra.dods.xa.XATransactionFactory"
```

This parameter tell DODS to create (JTA) XATransaction instead of StandardDBTransaction. And then in same file set additional properties requested by XATransactionFactory:

1. XADefaultTimeout is timeout of distributed transation.

```
DatabaseManager.DB.<LdbName>.XADefaultTimeout = 60
```

2. XAUsageCase is parameter that tell DODS how to behave when application performe commit(),rollback() or release() against XATransaction. XATransaction are always controlled by (JTA) UserTransaction and should never explicit perform any of this operations.

Values are : 0(INGORE), 1(WARN), 2(WARN_WITH_TRACE), 3(THROW_EXCEPTION), 4(THROW_ERROR).

```
DatabaseManager.DB.<LdbName>.XAUsageCase=0
```

3. Factory class of Wrapped transaction (transaction encapsulated inside XATransaction, default = StandardDBTransactionFactory)

```
DatabaseManager.DB.<LdbName>.XAWrappedTransImplFactory   =
 "com.lutris.appserver.server.sql.standard.StandardDBTransactionFactory"
```

4. JNDI lookup name of (JTA) UserTransaction object default = "java:comp/UserTransaction"

```
DatabaseManager.DB.<LdbName>.XaUserTransactionLookupName    = "java:comp/UserTransaction"
```

5. JNDI lookup name of (JTA) TransactionManager object

```
DatabaseManager.DB.<LdbName>.XATransactionManagerLookupName = "java:comp/UserTransaction"
```

Using (JTA) UserTransaction.

To force DODS to create (JTA) UserTransaction instead (JTA) XATransaction set TransactionFactory parameter to "XAUserTransactionFactory". Eg:

```
DatabaseManager.DB.<LdbName>.TransactionFactory  =
 "org.enhydra.dods.xa.XAUserTransactionFactory
```

In case of using XAUserTransactionFactory there are also one additional parameter named 'JTASupport', this parameter tell DODS how to handle requests for new Transactions in different JTA environment. Eg:

```
DatabaseManager.DB.<LdbName>.JTASupport  = MANDATORY
```

Parameter values are: MANDATORY, REQUIRED (by default), REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER.

- NOT_SUPPORTED: If the DODS transaction factory is called within a user transaction scope, this user transaction is suspended during the time of the new transaction execution (transaction factory return simple DODS transaction object defined by XAWrappedTransImplFactory parameter).

- REQUIRED: If the DODS transaction factory is called within a user transaction, the transaction is created in the scope of this user transaction (transactions factory returns instance of org.enhydra.dods.xa.XATransaction class), else, a new user transaction is started (transactions factory returns instance of org.enhydra.dods.xa.XAUserTransaction class).
- REQUIRES_NEW: DODS transaction factory will always create new user transaction. If the transaction constructor is called within existing user transaction, this transaction is suspended before the new one is started and resumed when the new transaction has completed. (transactions factory returns instance of org.enhydra.dods.xa.XAUserTransaction class).
- MANDATORY: The DODS transaction factory should always be called within the scope of a user transaction (transactions factory returns instance of org.enhydra.dods.xa.XATransaction class), else the DODS will throw exception.
- SUPPORTS: DODS transaction factory is invoked within the caller transaction scope (transactions factory returns instance of org.enhydra.dods.xa.XATransaction class), if the caller does not have an associated user transaction , DODS transaction is invoked without a transaction scope (transaction factory return simple DODS transaction object defined by XAWrappedTransImplFactory parameter).
- NEVER: The client is required to call the DODS transaction factory without any transaction context (transaction factory return simple DODS transaction object defined by XAWrappedTransImplFactory parameter), if it is not the case, a exception is thrown by the DODS.

All other parameters (1-5) have same meaning like in case of XATransaction. Detail explanation of Java Transaction API (JTA) see reference documentation from Sun Microsystems Inc. site.

Note

All parameters described in this chapter can also be added on 'DatabaseManager' level.

Current JTA API Implementation in DODS

Introduction

Typical DODS application deals with Transaction and Query objects. The later type covers *R* of the *CRUD* acronym, while the former is responsible for *C*, *U* & *D* operations. Grouping of multiple operations together is done via explicit usage of DBTransaction objects, and that basically covers everything DODS application has to care about:

```
DBTransaction dbt;
try {
    dbt = DODS.getDatabaseManager().createTransation();
    DiscQuery qry = new DiscQuery(dbt);
    qry.setQueryOwner(person);
    qry.addOrderByArtist();
    DiscDO[] arr = qry.getDOArray();
    // ...
    dbt.commit();
} catch (Exception e) {
    dbt.rollback();
} finally {
    dbt.release();
}
```

DODS was extracted out of Enhydra Server, and for the quite same time it tended to be complete self sufficient solution for O/R mapping. DODS has it’s own connection pooling, transaction control, cache

implementations,… Since Enhydra was stripped off of the multi-server part, and recent versions can be plugged into various servlet containers, DODS had to follow the lead. It learned to recognize and use *DataSources*, their pools also, but remained its own boss with *DBTransaction* control, oblivious of *JTA standard*.

UserTransaction (JTA)

JTA introduces lots of features, two phase commit being the most visible, but API usage remains rather simple: begin an *UserTransaction*, get connections out of a *DataSource*, do work on those connections and close them, then either commit or roll back instance of *UserTransaction*. Transaction manager will keep you from getting the second *UserTransaction* active in the same thread, you cannot reuse transaction object until beginning it again.

```
UserTransaction ut;
try {
    ut = ((UserTransaction) new InitialContext()
                    .lookup("java:comp/UserTransaction"));
    DataSource ds = ((DataSource)new InitialContext()
                    .lookup("java:comp/datasource/app1"));
    Connection conn = ds.getConnection();
    conn.executeUpdate(“DELETE FROM TEMPTABLE”);
    // ...
    conn.close();
    ut.commit();
} catch (Exception e) {
    ut.rollback();
}
```

After some lessons learned on the XA path, DODS 7 introduces compatibility mode with the JTA compliant environments:

- generate layer with **dirtyDO=”Compatible”**,
- use the **Transaction** and **Query** objects like before,
- forget about **DBTransaction**, since it’s not in charge anymore, there is
- **UserTransaction** to control transaction's behavior.

```
UserTransaction ut;
try {
    ut = ((UserTransaction) new InitialContext()
                    .lookup("java:comp/UserTransaction"));
    DiscQuery qry = new DiscQuery();
    qry.setQueryOwner(person);
    qry.addOrderByArtist();
    DiscDO[] arr = qry.getDOArray();
    // ...
    ut.commit();
} catch (Exception e) {
    ut.rollback();
}
```

- set parmeters in configuration file to appropriate values (explanation is following),
- behind the scenes, generated layer will keep creating transactions the old style, but set of configuration parameters and DODS' transaction factory will make things work. Mapping of active *UserTransaction* to DODS' *DBTransactions*, factory will choose whether to really construct *DBTransaction* anew, or to serve back one previously created.

Configuration parameters

To use JTA in DODS, the following appicatation parameters are important (in configuration file):

- *JTA* - must be set to true, in order to use JTA scenario in DODS.

  Example:

  ```
  DatabaseManager.defaults.JTA=true
  ```

- *TransactionFactory* - must be set to class SyncDBTransactionFactory, which provides DBTransaction objects that are aware of JTA environment, and also obey TransactionManager.

  Instances returned by this factory implement both *javax.transaction.Synchronization* and *DBTransaction*, thus solve hierachy problem between DODS and XADataSource connections.

  Example:

  ```
  DatabaseManager.defaults.TransactionFactory=org.enhydra.dods.jta.SyncDBTransactionFactory
  ```

- *XATransactionManagerLookupName*

  Example:

  ```
  DatabaseManager.defaults.XATransactionManagerLookupName="java:comp/UserTransaction"
  ```

- *ConnectionAllocator* - must be set to a class that implements *com.lutris.appserver.server.sql.ExtendedConnectionAllocator* interface, and uses *DataSource* (javax.sql.DataSource) as a connection source.

  Example:

  ```
  DatabaseManager.defaults.ConnectionAllocator="com.lutris.appserver.server.sql.datasource.DataSourceConnecti
  ```

- *DisableConnectionPool* - must be set to true if parameter *ConnectionAllocator* is set to value *com.lutris.appserver.server.sql.datasource.DataSourceConnectionAllocator* or to value *com.lutris.appserver.server.sql.datasource.SimpleDataSourceConnectionAllocator*.

  Example:

  ```
  DatabaseManager.DB.<LogicalDatabaseName>.Connection.DisableConnectionPool=true
  ```

- *ClassName* - must be set to *JTAObjectIdAllocator* implementation of *ObjectIdAllocator* interface.

  Example:

  ```
  DatabaseManager.DB.<LogicalDatabaseName>.ObjectId.ClassName=org.enhydra.dods.jta.JTAObjectIdAllocator
  ```

- *ConnectionFactory* - at the moment, it is set to *com.lutris.appserver.server.sql.DataSourceDBConnectionFactory* class.

  Example:

  ```
  DatabaseManager.DB.<LogicalDatabaseName>.Connection.ConnectionFactory="com.lutris.appserver.server.sql.data
  ```

- *DataSourceName* - must be to JNDI name (in "jndi:<dataSourceName>" formath) of externally defined DataSource that can be used by DODS to establish connection to database.

  Example:

  ```
  DatabaseManager.DB.<LogicalDatabaseName>.Connection.DataSourceName="jndi:java:comp/
  datasource/discRackdb"
  ```

- *XAWrappedTransImplFactory* - this parameter is optional. It defines factory class of wrapped transaction (transaction encapsulated inside *UserTransaction*). If not defined, the default value is *StandardDBTransactionFactory*).

Example:

```
DatabaseManager.defaults.XAWrappedTransImplFactory="org.enhydra.dods.dbtransaction.ExtendedTxFactory"
```

References

- JTA Specification [http://java.sun.com/products/jta/ttp://www.objectweb.org]
- A Java Open Transaction Manager - JOTM [http://jotm.objectweb.org/]
- Enhydra XAPool project [http://xapool.experlog.com/]

Chapter 25. Mass Modifications

DODS's duality

A problem

DODS gives you the option to choose how you want to modify rows of table in database. Obvious one is to use instances of generated DO classes:

```
SomeDOClass sgDO = SomeDOClass.createExisting(oid);
sgDO.delete();
```

Other option is QueryBuilder, which may be used to build not only select queries, but update or delete statements, as in:

```
QueryBuilder qb = getQueryBuilderForClass("SomeDOClass");
qb.setDeleteQuery();
qb.addWhereClause("oid", Integer.parseInt(enumoid), QueryBuilder.EQUAL);
qb.executeUpdate();
```

There could be a situation where you may want to touch many rows at once. First approach must be encompassed by loop which would iterate value of *oid*, thus producing many separate SQL statements. This isn't efficient at all, and it gets slower as number of rows raises.

Second approach, using QueryBuilder produces *one* SQL statement, and executes much faster. But:

Cache implementation in DODS includes caching DataStructs and queries for table globally, and caching DO objects in transaction. Both caches are implemented in generated classes only, so using QueryBuilder won't touch caches.

Warning

Direct use of QueryBuilder is NOT recommended, since it doesn't affect any of the caches, and your application may work erroneously.

Generated classes

A solution

New options in `.doml` file are massUpdates and massDeletes. They're implemented as boolean attributes of doml and table tags. Default values are false.

When turned on new options allow you to build data layer including two classes *xxxUpdate* and *xxxDelete*. These classes provide you QueryBuilder speed in massive update operations, while maintaining caches (both global and transactions) valid.

Classes *xxxUpdate* and *xxxDelete* have constructor that takes *xxxQuery* as parameter. This instance of query builds WHERE clause of a statement, while *setCOLUMN* methods provide contents of SET part.

```
xxxQuery  query = new xxxQuery(dbt);
query.setQueryCOLUMN_NAME(value);
xxxUpdate update = new xxxUpdate(query);
update.setANOTHER_COLUMN(another_column_value);
update.save();
dbt.commit();
```

In order to keep caches valid, global query caches (for affected table) are cleared, because we cannot compute their consistency without all DataStructs. DataStructs in global cache and DO objects in transaction caches are removed (for delete) or emptied (they will be loaded as with lazy load feature, for update). When using true (default value) for IncrementVersions entries in cache are removed also. Only SelectOIds parameter gives the cache enough information (list of OIds) to precisely update its contents.

New `.conf` parameter *SelectOids* introduced at DatabaseManager, LogicalDatabase, and table level, specifies whether there would be additional select statement executed to collect oids that would be affected by mass modification. Default value for *SelectOids* is false, and usual override method is applied too (table level overrides database which in turn overrides manager's value). If parameter is `true`, before actually executing massive modifications `select` statement will be run to collect list of OId's. This is then used to update cache for listed DataStructs/DOs only. Otherwise (*SelectOids* is `false`), all instances of xxxDataStruct/xxxDO will be updated.

Both *xxxUpdate* and *xxxDelete* have method `setSelectOIds(boolean)` for developer to prevent configuration parameter *SelectOids* effects. If certain mass modification will affect many rows, developer may choose to prevent collecting oids, so even if administrator sets parameter to true, application doesn't lose on its speed.

```
xxxQuery  query = new xxxQuery(dbt);
query.setQueryCOLUMN_NAME(value);
xxxDelete delete = new xxxDelete(query);
delete.setSelectOIds(false);
delete.save();
dbt.commit();
```

Chapter 26. Using database generated identity columns in DODS

In 6.5-1 and prior version, DODS explicitly use self-generated (by DODS framework) values for object identification (OID), and uses it as base values for all primary and foreign keys columns in whole database. Although this mechanism have advantage in simplicity and easy of use, in case that we have database that is totally dedicated to DODS application(s) and built "form the ground", but they introduces some very strong constraints in case when we have already existing database or we need to use same database simultaneously from DODS and non-DODS application(s).

Since version 6.6.1 DODS have capabilities to use database generated identity (identity, auto-generated, serial .. name depends on DB vendor) values as base for DO object identification. Because now database engine is the main generator of object identity, and this values are used as primary and foreign key column values this enables DODS usage in case when we have already existing database or mix of DODS/non-DODS application that use same database.

To work with this "autogenerated" values DODS relays on JDBC driver / DB engine capabilities to retrieve this values from database. According to JDBC 3.0 specification the way to work with "autogenerated columns" is standardized on database (JDBC) driver level, but many today available jdbc drivers have incomplete implementation of JDBC 3.0 or are not JDBC 3.0 compliant at all. So, to enable work with "autogenerated columns" independent on JDBC driver specification level, and to support this feature on broader range of JDBC drivers and database vendors, DODS introduces several different mechanism implementation for retrieving autogenerated values from database.

Base for all of this "custom" implementation is DriverDependencies interface. There are several implementation of DriverDependencies interface and DODS in runtime decide which to use, based on current active database driver.

Mapping between database driver and DriverDependencies implementation is placed in DatabaseVendor configuration file (<DODS_HOME>/build/conf folder) inside <Drivers> section (DriverDependenciesClass parameter) together with other driver dependent parameters eg. (in PostgreSQLConf.xml):

```
<Drivers>
  <org.postgresql.Driver>
    <DriverDependenciesClass>org.enhydra.dods.dependencies.PgSqlDependencies</
DriverDependenciesClass>
  </org.postgresql.Driver>
</Drivers>
```

Default value for this parameter are:

```
org.enhydra.dods.dependencies.StandardDriverDependencies
```

that implements access to autogenetrated columns trough standard JDBC 3.0 compliant way.

There are also several other implementations:

- org.enhydra.dods.dependencies.HsqlDbDependencies.java - for HSQL database/driver.
- org.enhydra.dods.dependencies.PgSqlDependencies – for Postgers database engine.
- org.enhydra.dods.dependencies.MsjdbcDriverDependencies – for MS SQL Server2000 database and Microsoft JDBC driver (msjdbc)

For all details about parmeter see " DriverDependenciesClass" section in "Database Vendor and Driver Specific Parameters." In Ch.23.

DriverDependenciesClass parameter value just maps specific implementation of DriverDependencies to specific database driver, but to enable DODS to use this implementation in runtime we need to set " ObjectIdAllocationSource" parameter in application configuration file to "database" eg.

```
DatabaseManager.defaults.ObjectIdAllocationSource = database
```

For all details about parmeter see " ObjectIdAllocationSource" section in " New Parameters in Configuration and DOML Files." In Ch.22.

Chapter 27. Using "OID per Table" feature in DODS.

In 6.5-1 and prior versions, all DODS generated values for object identification (OIDs), where unique in scope of whole database. In case when we need to insert some new values to database table, outside of DODS (and potentially link them to some other table) or in case we need to do some "migration" from one database model to another one, this OID uniqueness are making difficulties.

Since version 6.6-1 DODS supports creating of OIDs that are unique in context of table, but distinct tables can contains same OID values ("OID per Table").

To enable this feature in DODS, there are two important steps:

- First we need to add new column to database "objectid" table, this new column need to be called "table_name" and be some of variable length character type with length of 255 eg. (on MS SQL Server2000)

```
table_name NVARCHAR(255)
```

- Then we need to set "AllocationScope" parameter in application configuration file to "table" value. Eg.

```
DatabaseManager.DB.<databaseName>.ObjectId.AllocationScope = "table"
```

After this two steps all newly generated OIDs (ObjectId) values will be unique in scope of database table independently of values in other tables.

For all details about parmeter see " AllocationScope " section in " New Parameters in Configuration and DOML Files." In Ch.22.

Chapter 28. Statistics

Creating statistics statements

DODS generator can create statistics statements. New element in doml STATISTICS is introduced. It creates a histogram and associated density groups (collections) over the supplied column or set of columns of a table or indexed view. String summary statistics are also created on statistics built on char, varchar, varchar(max), nchar, nvarchar, nvarchar(max), text, and ntext columns. The query optimizer uses this statistical information to choose the most efficient plan for retrieving or updating data. Up-to-date statistics allow the optimizer to accurately assess the cost of different query plans, and choose a high-quality plan.

Attributes and subelements

id - is the name of the statistics group to create. Statistics names must comply with the rules for identifiers and must be unique to the table or view on which they are created.

statisticsColumn - is subelement and it is the column or set of columns on which to create statistics. Any column that can be specified as an index key can also be specified for statistics, with the following exceptions: xml columns cannot be specified. The maximum allowable size of the combined column values can exceed the 900-byte limit that is imposed on the index key value. Computed columns can be specified only if the ARITHABORT and QUOTED_IDENTIFIER database options are set to ON. CLR user-defined type columns can be specified if the type supports binary ordering. Computed columns defined as method invocations off a user-defined type column can be specified if the methods are marked deterministic.

fullScan - Specifies that all rows in table or view should be read to gather the statistics. Specifying FULLSCAN provides the same behavior as SAMPLE 100 PERCENT. This option cannot be used with the SAMPLE option.

sample(sampleNo,sampleType) - Specifies that a percentage, or a specified number of rows, of the data should be read by using random sampling to gather the statistics. Number must be an integer. If PERCENT is specified, number should be from 0 through 100; if ROWS is specified, number can be from 0 to the n total rows.

noRecompute - Specifies that the Database Engine should not automatically recompute statistics. If this option is specified, the Database Engine continues to use previously created (old) statistics, even as the data changes. The statistics are not automatically updated and maintained by the Database Engine. This may produce suboptimal plans.

Only the table owner can create statistics on that table. The owner of a table can create a statistics group (collection) at any time, whether or not there is data in the table. If the AUTO_UPDATE_STATISTICS database option is set to ON (this is the default setting) and the NORECOMPUTE clause is not specified, the Database Engine will automatically update any statistics that are manually created. CREATE STATISTICS can be executed on an indexed view. Statistics on indexed views are used by the optimizer only if the view is directly referenced in the query and the NOEXPAND hint is specified for the view. Otherwise, statistics are derived from the underlying tables before the indexed view is substituted into the query plan. This substitution is supported only on Microsoft SQL Server 2005 Enterprise and Developer editions.

Chapter 29. Additional coulumns in index

Including columns in index

DODS generator can create create index statements with INCLUDE option. New subelement in doml for index, includeIndexColumn is introduced.

Include index column option enables the nonkey columns to be added to the leaf level of the nonclustered index. The nonclustered index can be unique or nonunique. The maximum number of included nonkey columns is 1,023 columns; the minimum number is 1 column (for SQL server 2005).Column names cannot be repeated in the INCLUDE list and cannot be used simultaneously as both key and nonkey columns. All data types are allowed except text, ntext, and image. In DB2, INCLUDE can only be specified for indexes that are defined with UNIQUE.

Chapter 30. Advanced Access

Creating union of ResultSets.

In some cases is useful to make result set that are union of two or more distinct result sets, this case is supported trough "UNION [ALL]" part of "SELECT" sql statement. To support this feature of SELECT statement, DODS introduced "addUnion(QueryBuilder, boolean all)" method in QueryBuilder class. To avoid possibilities of loosing data integrity there are some restrictions of using "UNION [ALL]" part of “SELECT” statement in QueryBuilder. If "addUnion(QueryBuilder, boolean all)" method of QueryBuilder is called then that query can not be used to directly retrieve xxxDO objects from database (you can not call runQuery() method of xxxQuery class) and only way to get some data is to allocate connection and directly call "executeQuery(DBConnection conn)" method of QueryBuilder class to get ResultSet.

Example:

```
            DBConnection conn=DODS.getDatabaseManager().allocateConnection();
            ResultSet rs=null;
            try {

                QueryBuilder tempQB1 = new QueryBuilder("PersonTable1","firstname,lastname");
           QueryBuilder tempQB2 = new QueryBuilder("personSelfRef","firstname,lastname");
                QueryBuilder tempQB2 = new QueryBuilder("PersonTable2","firstname,lastname");

                tempQB1.addWhere(" firstname='mark' ");
                tempQB2.addWhere(" firstname='leo' ");

                tempQB1.addUnion(tempQB2, false);
                rs=tempQB1.executeQuery(conn);
                printResultSet(rs);
                rs.close();

            }catch (Exception ex){
                ex.printStackTrace();
            }finally {
                conn.release();
            }
```

Or You can use instance of QueryBuilder related to one or more other QueryBuilder instances (trough call of "addUnion(QueryBuilder, boolean all)" method, as part of "WHERE" statement in another instance of QueryBuilder.

Example:

```
   query1 = new Table1Query();
   qb1 = query1.getQueryBuilder();
   qb1.setSelectClause(TABLE1.TABLE2OID);

   query2 = new Table2Query();
   qb2 = query2.getQueryBuilder();
   qb2.setSelectClause(TABLE2.OID);
   qb2.addUnion(qb1);

   query3 = new Table3Query();
   qb3 = query3.getQueryBuilder();
   qb3.addWhereIn((new RDBColumn(new RDBTable("TABLE3"),"TABLE2OID"), qb2);
   Table3Query.getDOArray();
```

Chapter 31. Database Configurations

This chapter provides information on connecting DODS applications to specific database types. In general, you need to add the database configuration information to the application configuration file (e.g., simpleApp.conf). Configurable items in the code snippets that you need to specify, such as path names or database identifier, are enclosed in brackets and italicized (for example, <path_name> or <database_id>).

Driver configuration

Important DODS connects to databases using a JDBC driver. DODS has its own class loader, but the JDBC driver must be loaded by the system class loader. Therefore, it is important to specify the path to the JDBC driver in your system CLASSPATH and not in the DODS application's CLASSPATH.

A common way to specify the path to the JDBC driver is to save the driver in a lib directory in the project and define the CLASSPATH in the run script. To do this, follow these steps:

- 1 Create a lib directory in the top level of your project and copy your JDBC driver to this directory.
- 2 Edit your application's run file template, start.in, (in the <appName>/input directory) to place the driver in your CLASSPATH. For example:

```
...
#
# Build up classpath.
#
CLASSPATH="../lib/idb.jar/;../lib/jta-spec1_0_1.jar"
APPCP="${DODS_LIB}${PS}../classes"
...
```

- 3 Build the project with ant, which will copy the run script to the directory <appName>/output. Use this script to start your application.

Be careful to keep the right driver with your application. For example, there are multiple versions of the Oracle JDBC driver, classes111.zip. When your application goes into production, make sure that the project administrator knows to reference the correct driver when the database is upgraded in the future.

Using DODS with javax.sql.DataSource

DODS can use separate ConnectionAllocator implementation to support javax.sql.DataSource as database connections source for connection allocation (see "ConnectionAllocator" and "ConnectionFactory" parameters reference).

In case that DODS is configured to use DataSource then connection level parameters "Url", "User" and "Password" are sufficient. Parameters named "ClassType" and "JdbcDriver" are not mandatory (DODS will try collect this information's directly from DataSource and internal dodsConf.xml file) but this information's can be useful in situations when DODS don"t recognize specific driver/database.

DataSource is defined externally to DODS by some environment dependent mechanism (eg. inside Servlet/JSP Container ..) and DODS use JNDI api to access this object. DODS references DataSource

by name that is defined in application configuration parameter called DataSourceName (see reference for "DataSourceName").

eg.

```
DatabaseManager.DB.<database_id>.Connection.DataSourceName = jndi:<DataSourceName>
```

Example of full configuration:

```
#---------------------------------------------------------------------
# Database Manager Configuration
#---------------------------------------------------------------------
DatabaseManager.Databases[]     = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "false"
DatabaseManager.DB.<database_id>.ClassType  = "<dbTypeName>"
DatabaseManager.DB.<database_id>.JdbcDriver = "<JdbcDriverClassName>"

DatabaseManager.DB.<database_id>.ConnectionAllocator
                   = com.lutris.appserver.server.sql.datasource.DataSourceConnectionAllocator
DatabaseManager.DB.<database_id>.Connection.ConnectionFactory
                   = com.lutris.appserver.server.sql.DataSourceDBConnectionFactory
DatabaseManager.DB.<database_id>.Connection.DataSourceName = jndi:<DataSourceName>

# not used with DataSource
#   DatabaseManager.DB.<database_id>.Connection.Url =
 "jdbc:sybase:Tds:<hostname>.sybase.com:7100"
#   DatabaseManager.DB.<database_id>.Connection.User = "<name>"
#   DatabaseManager.DB.<database_id>.Connection.Password = "<password>"
# not used with DataSource

DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = "2"
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = "2"
DatabaseManager.DB.<database_id>.Connection.Logging = "true"
DatabaseManager.DB.<database_id>.Connection.MaxPreparedStatements = "2"
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 2
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1
```

Oracle

This section presents an example of an Oracle configuration, where <database_id> is your database identifier.

```
#---------------------------------------------------------------------
# Database Manager Configuration
#---------------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "false"
DatabaseManager.DB.<database_id>.ClassType = "Oracle"
DatabaseManager.DB.<database_id>.JdbcDriver = "oracle.jdbc.driver.OracleDriver"
DatabaseManager.DB.<database_id>.Connection.Url =
                        "jdbc:oracle:thin:@<server_name>:<port#>:<db_instance>"
DatabaseManager.DB.<database_id>.Connection.User = "<user>"
DatabaseManager.DB.<database_id>.Connection.Password = "<password>"
DatabaseManager.DB.<database_id>.Connection.MaxPreparedStatements = 10
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = 30
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = 10000
DatabaseManager.DB.<database_id>.Connection.Logging = false
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 20
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1
```

The driver used here is the Oracle thin driver, and <db_instance> is the name of the Oracle database instance.

This is the link where you can find all needed information and downloads for Oracle database:

http://www.oracle.com/products [http://www.oracle.com/products]

Informix

This section presents an example of an Informix configuration, where <database_id> is your database identifier.

```
#-------------------------------------------------------------------
# Database Manager Configuration
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "false"
DatabaseManager.DB.<database_id>.ClassType = "Informix"
DatabaseManager.DB.<database_id>.JdbcDriver = "com.informix.jdbc.IfxDriver"
DatabaseManager.DB.<database_id>.Connection.Url =
jdbc:informix-sqli://<hostname>:<port#>:INFORMIXSERVER=<db_instance>;
user=<user>;password=<password>
DatabaseManager.DB.<database_id>.Connection.User = "<user>"
DatabaseManager.DB.<database_id>.Connection.Password = "<password>"
DatabaseManager.DB.<database_id>.Connection.MaxPreparedStatements = 10
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = 30
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = 10000
DatabaseManager.DB.<database_id>.Connection.Logging = false
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 20
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1
```

This is the link where you can find all needed information and downloads for Informix database:

http://www-3.ibm.com/software/data/informix/ids [http://www-3.ibm.com/software/data/informix/ids]

Sybase

This section presents an example of a Sybase configuration, where <database_id> is your database identifier.

```
#-------------------------------------------------------------------
# Database Manager Configuration
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "true"
DatabaseManager.DB.<database_id>.ClassType = "Sybase"
DatabaseManager.DB.<database_id>.JdbcDriver = "com.sybase.jdbc2.jdbc.SybDriver"
DatabaseManager.DB.<database_id>.Connection.Url =
 "jdbc:sybase:Tds:<hostname>.sybase.com:7100"
DatabaseManager.DB.<database_id>.Connection.User = "<name>"
DatabaseManager.DB.<database_id>.Connection.Password = "<password>"
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = "2"
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = "2"
DatabaseManager.DB.<database_id>.Connection.Logging = "true"
DatabaseManager.DB.<database_id>.Connection.MaxPreparedStatements = "2"
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 2
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1
```

This is the link where you can find all needed information and downloads for Informix database:

http://www.sybase.com/products/databaseservers [http://www.sybase.com/products/databaseservers]

QED

QED, the Quadcap Embeddable Database. QED is a fast, small, pure Java, relational database, implementing the SQL 92 standard, with transactions and resilient failure recovery. QED has a novel open source license permitting free use of QED by all and free redistribution in other open source projects.

```
#-----------------------------------------------------------------
# Database Manager Configuration
#-----------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "true"
DatabaseManager.DB.<database_id>.ClassType = "Sybase"
DatabaseManager.DB.<database_id>.JdbcDriver = " com.quadcap.jdbc.JdbcDriver"
DatabaseManager.DB.<database_id>.Connection.Url = " jdbc:qed:<databaseFolderPath>"
DatabaseManager.DB.<database_id>.Connection.User = "<name>"
DatabaseManager.DB.<database_id>.Connection.Password = "<password>"
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = "2"
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = "2"
DatabaseManager.DB.<database_id>.Connection.Logging = "true"
DatabaseManager.DB.<database_id>.Connection.MaxPreparedStatements = "2"
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 2
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1
```

Where <databaseFolderPath> is path to folder that represents QED database.

This is the link where you can find all needed information and downloads for Informix database:

http://www.quadcap.com/products/qed/docs/index.html [http://www.quadcap.com/products/qed/docs/index.html]

MySQL

MySQL is an open source database that is lightweight and fast.

NOTE:

Prior to version 3.23, MySQL does not support transactions, and therefore does not support explicit commit (they use autocommit by default after eny SQL command). To use MySQL versions 3.22 and earlier, you have to make change to application configuration file. You will need to set parameter ‘ChangeAutocommit’, of logical database, to ‘false’ (this will disable DODS to change, database connection, autocommit property). Example:

Configuration:

This section presents an example of a MySQL configuration, where <database_id> is your database identifier.

```
#-----------------------------------------------------------------
# Database Manager Configuration
#-----------------------------------------------------------------
```

```
DatabaseManager.Databases[] = <database_id>
DatabaseManager.DefaultDatabase = <database_id>
DatabaseManager.Debug = true
DatabaseManager.DB.<database_id>.ClassType = Standard
DatabaseManager.DB.<database_id>.Connection.User = <username>
DatabaseManager.DB.<database_id>.Connection.Password = <password>
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = 5
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = 10000
DatabaseManager.DB.<database_id>.Connection.Logging = true
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 1024
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 100
DatabaseManager.DB.<database_id>.JdbcDriver = org.gjt.mm.mysql.Driver
DatabaseManager.DB.<database_id>.Connection.Url ="jdbc:mysql://<hostname>:<port#>/
<db_instance>"
DatabaseManager.DB.<database_id>.ChangeAutocommit = "true"
```

This is the link where you can find all needed information and downloads for MySQL database:

http://www.mysql.com/downloads/index.html [http://www.mysql.com/downloads/index.html]

PostgreSQL

Note:

Although other versions are available commercially, the Together company supports the open-source version of PostgreSQL for the Linux operating system for use with DODS

PostgreSQL is a popular open-source database used with DODS however, DODS requires a special column named OID in each table. However, OID is a reserved word in PostgreSQL.

Fortunately, the column names used for OID and VERSION are configurable. To configure these names, add the following lines to your application configuration file:

```
DatabaseManager.ObjectIdColumnName = "<ColName_for_ObjectId>"
DatabaseManager.VersionColumnName = "<ColName_for_Version>"
```

where <ColName_for_ObjectId> and <ColName_for_Version> are the column names you want to use instead of OID and VERSION.

Configuration example:

```
#---------------------------------------------------------------------
# Database Manager Configuration
#---------------------------------------------------------------------

DatabaseManager.Databases[] = <database_id>
DatabaseManager.DefaultDatabase = <database_id>
DatabaseManager.Debug = true
DatabaseManager.DB.<database_id>.ClassType = Standard
DatabaseManager.DB.<database_id>.Connection.User = <username>
DatabaseManager.DB.<database_id>.Connection.Password = <password>
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = 5
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = 10000
DatabaseManager.DB.<database_id>.Connection.Logging = true
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 1024
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 100
DatabaseManager.DB.<database_id>.JdbcDriver = "org.postgresql.Driver"
DatabaseManager.DB.<database_id>.Connection.Url ="jdbc:postgresql://<host>/<db_instance>"
```

To specify character encoding for Postgresql you can specify the 'charSet' parameter within the connection URL.

```
DatabaseManager.DB.example.Connection.Url =
       "jdbc:postgresql://192.168.1.1/yourDbName?charSet=iso-8859-1"
```

This is the link where you can find all needed information and downloads for PostgreSQL database:

http://www.postgresql.org [http://www.postgresql.org]

InstantDB

To use an InstantDB database with an DODS application:

1. In the application configuration file <appName>/output/conf/<appName>.conf (or better, in <appName>/input/conf/<appName>.conf.in) set the following line:

```
DatabaseManager.DB.<database_id>.Connection.Url = "jdbc:idb:<propFile>.prp"
```

where <propFile> is the full path to the database properties file, and <database_id> is the database identifier used in the configuration file.

2. In the same configuration file, identify the JDBC driver with the line:

```
DatabaseManager.DB.<database_id>.JdbcDriver = "org.enhydra.instantdb.jdbc.idbDriver"
```

3. Add the path to idb.jar to the setting for CLASSPATH in the application's run script, in <appName>/run, or better in . <appName>/run.in.

This is the link where you can find all needed information and downloads for InstantDb database:

http://instantdb.tripod.com/old-site/index-9.html [http://instantdb.tripod.com/old-site/index-9.html]

Mckoi

To use an Mckoi sql database with an DODS application

1. In the application configuration file <appName>/output/conf/<appName>.conf (or better, in <appName>/input/conf/<appName>.conf.in) set the following line:

```
DatabaseManager.DB.<database_id>.Connection.Url = " jdbc:mckoi:local://<confFilePath>"
```

where < confFilePath> is the full path to the database properties file, and <database_id> is the database identifier used in the configuration file.

2. In the same configuration file, identify the JDBC driver with the line:

```
DatabaseManager.DB.<database_id>.JdbcDriver = " com.mckoi.JDBCDriver"
```

3 Add the path to mckoidb.jar and mkjdbc.jar to the setting for CLASSPATH in the application's run script, in <appName>/run, or better in . <appName>/run.in.

This is the link where you can find all needed information and downloads for InstantDb database:

http://mckoi.com/database [http://mckoi.com/database]

P6SPY

P6Spy is an open source framework for applications that intercept and optionally modify database statements.

To use an p6Spy driver with DODS application:

1. In the application configuration file set the following line:

```
DatabaseManager.DB.<database_id>.Connection.Url = "<RealDatabaseURL>"
```

where < confFilePath> is the full path to the database properties file, <database_id> is the database identifier used in the configuration file, and <RealDatabaseURL> is URL of your database (real database driver URL)

2. Set p6spy jdbc driver class in same file:

```
DatabaseManager.DB.<database_id>.JdbcDriver = "com.p6spy.engine.spy.P6SpyDriver"
```

3 Add the path to p6spy.jar to the setting for CLASSPATH in the application's run script, in <appName>/run.

4. Edit

```
<DODS_HOME>/lib-ext/spy.properties
```

file, there are sample setings and additional informations, and put it in some folder in your application class path.

For more informations about p6spy there are a link where you can find all needed information and downloads.

http://www.p6spy.com/

DB2

To use an DB2 database with an DODS application

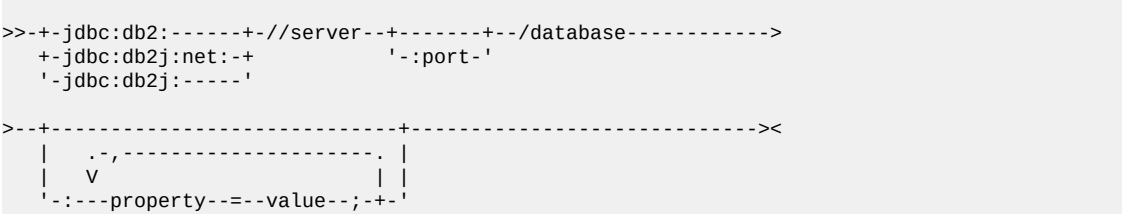
1. In the application configuration file <appName>/output/conf/<appName>.conf (or better, in <appName>/input/conf/<appName>.conf.in) set the following line:

```
DatabaseManager.DB.<database_id>.Connection.Url = “<url>”
```

The <url> argument represents a data source, and indicates what type of JDBC connectivity you are using.

For DB2 Universal JDBC Type 4 Connectivity, specify a URL of the following form:

Syntax for a URL for Universal Type 4 Connectivity:



For DB2 Universal JDBC Type 2 Connectivity, specify a URL of the following form:

Syntax for a URL for Universal Type 2 Connectivity:

```
>>-jdbc:db2:database--+----------------------------+---------><
                      |   .-,----------------------. |
                      |   V                        | |
                      '-:---property--=--value--;-+-'
```

The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2j: or jdbc:db2j:net:

jdbc:db2: indicates that the connection is to a server in the DB2 UDB family.

jdbc:db2j: indicates that the connection is to a for local Cloudscape access.

jdbc:db2j:net: indicates that the connection is to a remote IBM(R) Cloudscape server.

server

The domain name or IP address of the database server.

port

The TCP/IP server port number that is assigned to the database server. This is an integer between 0 and 65535. The default is 446.

database

The name of the database server. For a connection to a DB2 UDB for Linux, UNIX(R) and Windows(R) server, the name is the database name.

For a connection to an IBM Cloudscape server, the name is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

```
"c:/databases/testdb"
```

property=value;

A property for the JDBC connection. For the definitions of these properties, see

http://www-3.ibm.com/software/data/db2/udb/ad/v8/java/ad/rjvdsprp.htm

2 In the same configuration file, identify the JDBC driver with the line:

```
DatabaseManager.DB.<database_id>.JdbcDriver = ” com.ibm.db2.jcc.DB2Driver "
```

This is the link where you can find all needed information and downloads for DB2 database:

http://www-3.ibm.com/software/data/db2 [http://www-3.ibm.com/software/data/db2]

HSQLDB (HypersonicSQL)

HSQLDB is a relational database engine written in Java, with a JDBC driver, supporting a rich subset of ANSI-92 SQL (BNF tree format). It offers a small (less than 160k), fast database engine which offers both in memory and disk based tables. Embedded and server modes are available. Additionally, it includes tools such as a minimal web server, in-memory query and management tools (can be run as applets) and a number of demonstration examples.

To use an HSQLDB database with an Enhydra application

1. In the application configuration file <appName>/output/conf/<appName>.conf (or better, in <appName>/input/conf/<appName>.conf.in) set the following line:

```
DatabaseManager.DB.<database_id>.Connection.Url = “jdbc:hsqldb:hsql://<hostName>:<port>”
```

where <hostName> is the host name or IP adress of computer with runing database server, and <port> is port where database server wait for request (default: 9001).

2. In the same configuration file, identify the JDBC driver with the line:

```
DatabaseManager.DB.<database_id>.JdbcDriver = ”org.hsqldb.jdbcDriver "
```

3. Add the path to hsqldb.jar to the setting for CLASSPATH in the application’s run script, in <appName>/run, or better in . <appName>/run.in.

This is the link where you can find all needed information and downloads for HSQLDB database:

http://hsqldb.sourceforge.net

Microsoft SQL Server

The exact configuration settings for connecting to MS SQL server depend on the JDBC driver you are using. We do not recommend using the JDBC-ODBC bridge with MS SQL Server.

This is the link where you can find all needed information and downloads for MSQL database:

http://www.microsoft.com/sql/default.asp [http://www.microsoft.com/sql/default.asp]

JTurbo JDBC driver

We certified the JTurbo 2.0 JDBC driver, and the configuration settings for this are:

```
# JTurbo 2.0 JDBC Driver for MS SQL server
DatabaseManager.Databases [] = "my_db"
DatabaseManager.DefaultDatabase = "my_db"
DatabaseManager.DB.my_db.ClassType = "MSQL"
DatabaseManager.DB.my_db.JdbcDriver = "com.ashna.jturbo.driver.Driver"
DatabaseManager.DB.my_db.Connection.Url = "jdbc:JTurbo://<host>:<port>/<dbName>"
DatabaseManager.DB.my_db.Connection.User = "<user_name>"
DatabaseManager.DB.my_db.Connection.Password = "<password>"
```

If you are using another JDBC driver, you need to determine the driver package, for the DatabaseManager.DB.my_db.JdbcDriver setting, and connection string, for DatabaseManager.DB.my_db.Connection.Url setting.

jTDS JDBC driver

jTDS is an open source 100% pure Java (type 4) JDBC 3.0 driver for Microsoft SQL Server (6.5, 7, 2000 and 2005) and Sybase (10, 11, 12).

We certified the jTDS JDBC Driver 1.1, and the configuration settings for this are:

```
# jTDS JDBC Driver 1.1 for MS SQL server
DatabaseManager.Databases [] = "my_db"
DatabaseManager.DefaultDatabase = "my_db"
DatabaseManager.DB.my_db.ClassType = "MSQL"
DatabaseManager.DB.my_db.JdbcDriver = "net.sourceforge.jtds.jdbc.Driver"
DatabaseManager.DB.my_db.Connection.Url
            ="jdbc:jtds:sqlserver://<hostname>:<port>/<databaseName>;
               tds=8.0;lastupdatecount=true"
DatabaseManager.DB.my_db.Connection.User = "<user_name>"
DatabaseManager.DB.my_db.Connection.Password = "<password>"
```

http://jtds.sourceforge.net/

MS-JDBC driver

Configuration settings example for MS-JDBC driver are:

```
DatabaseManager.DB.my_db.JdbcDriver = "com.microsoft.jdbc.sqlserver.SQLServerDriver"
# NOTE: substitute your server's IP address (hostname)
# Substitute the port your DB is listening on for (default: 1433)
DatabaseManager.DB.my_db.Connection.Url
          ="jdbc:microsoft:sqlserver://<hostname>:<port>;
             DatabaseName= <databaseName>;SelectMethod=cursor "
DatabaseManager.DB.my_db.Connection.User = "<user_name>"
DatabaseManager.DB.my_db.Connection.Password = "<password>"
```

Configuration settings example for Microsoft SQL Server 2005 JDBC Driver driver are:

```
DatabaseManager.DB.my_db.JdbcDriver = "com.microsoft.sqlserver.jdbc.SQLServerDriver"
# NOTE: substitute your server's IP address (hostname)
# Substitute the port your DB is listening on for (default: 1433)
DatabaseManager.DB.my_db.Connection.Url
          ="jdbc:sqlserver://<hostname>:<port>;
             DatabaseName= <databaseName>;SelectMethod=cursor "
DatabaseManager.DB.my_db.Connection.User = "<user_name>"
DatabaseManager.DB.my_db.Connection.Password = "<password>"
```

If you are using another JDBC driver, you need to determine the driver package, for the DatabaseManager.DB.my_db.JdbcDriver setting, and connection string, for DatabaseManager.DB.my_db.Connection.Url setting.

Connection Parameters:

SendStringParametersAsUnicode

Determines whether string parameters are sent to the SQL Server database in Unicode or in the default character encoding of the database. True means that string parameters are sent to SQL Server in Unicode.

False means that they are sent in the default encoding, which can improve performance because the server does not need to convert Unicode characters to the default encoding. You should, however, use default encoding only if the parameter string data that you specify is consistent with the default encoding of the database. Default value is true.

SelectMethod

Determines whether database cursors are used for Select statements. Performance and behaviour of the driver are affected by the SelectMethod setting. Direct-The direct method sends the complete result set in one request to the driver. It is useful for queries that only produce a small amount of data that you fetch completely. You should avoid using direct when executing queries that produce a large amount of data, as the result set is cached completely on the client and constrains memory. In this mode, each statement requires its own connection to the database. This is accomplished by "cloning" connections. Cloned connections use the same connection properties as the original connection; however, because transactions must occur on a single connection, auto commit mode is required. Due to this, JTA is not supported in direct mode. In addition, some operations, such as updating an insensitive result set, are not supported in direct mode because the driver must create a second statement internally. Exceptions generated due to the creation of cloned statements usually return an error message similar to "Cannot start a cloned connection while in manual transaction mode." Cursor-When the SelectMethod is set to cursor, a server-side cursor is generated. The rows are fetched from the server in blocks. The JDBC Statement method setFetchSize can be used to control the number of rows that are fetched per request. The cursor method is useful for queries that produce a large amount of data, data that is too large to cache on the client. Performance tests show that the value of setFetchSize has a serious impact on performance when SelectMethod is set to cursor. There is no simple rule for determining the value that you should use. You should experiment with different setFetchSize values to find out which value gives the best performance for your application. The default is direct. DODS supports only cursor method.

Microsoft Access

Microsoft Access is not a true SQL database server; as such, it is suitable for development and testing, but not for a production database. Access does not have a JDBC driver. However, Access does support ODBC, and there is a JDBC-ODBC bridge in the Sun JDK, which enables Access to work with Enhydra.

Because Access cannot read-in files containing SQL commands, you must create tables in the Access GUI. See the Access documentation for more information. For the DiscRack example, you can also use the Access database provided in <dods_root>/examples/DiscRack/discRack.mdb.

To use DODS with Access:

- Register the database as an ODBC data source:
  - Go to Start|Settings|Control Panel and click ODBC Data Sources.
  - Click the Add button in the dialog box that comes up.
  - Select the Microsoft Access Driver in the Create New Datasource dialog box and click Finish.
  - The ODBC Microsoft Access Setup dialog box appears.
  - Choose a name, like discRack, for the Data Source Name. Under Database, click the Select button, browse to the *.mdb file, select it, and click OK.
- Set connection related parameters:

```
DatabaseManager.DB.<database_id>.ClassType = "Access"
DatabaseManager.DB.<database_id>.JdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver"
DatabaseManager.DB.<database_id>.Connection.Url = "jdbc:odbc:<dsnName>"
```

- Set names for "objectid" table name and "next" column of "objectid" table (parameters "OidTableName" and "NextColumnName"). This will caused that DODS don't use "next" when access to "objectid" table ("next" is reserved word in "Access")

  ```
  DatabaseManager.DB.<database_id>.ObjectId.OidTableName="<oidTableName>"
  DatabaseManager.DB.<database_id>.ObjectId.NextColumnName="<nextColumnName>"
  ```

  - or you can (have a same effect) set:

  ```
  DatabaseManager.DB.<database_id>.ObjectId.NextWithPrefix="true"
  ```

- Disable use of "FetchSize" in DODS - this feature are not supported by MS Access odbc driver.

  ```
  DatabaseManager.defaults.DefaultFetchSize=0
  ```

- Disable use of "QueryTimeout" in DODS - this feature are not supported by MS Access odbc driver.

  ```
  DatabaseManager.defaults.QueryTimeout=false
  ```

- To avoid use of "LOWER" function in DODS created SQL statements (this function is not supported by Access) set:

  ```
  DatabaseManager.defaults.CaseInsensitiveDatabase="true"
  ```

- To avoid problem with inconsistency of data read/write in different transaction, use clean transaction or use every connection (transaction) only once (disable connection pool). To force DODS to disable connection pool set:

  ```
  DatabaseManager.DB.<database_id>.Connection.MaxConnectionUsages = 1
  ```

Note

You don’t have to place the JDBC driver in the application’s CLASSPATH because the ODBC/JDBC bridge is in the JDK and thus is already in the system’s CLASSPATH.

This section presents an example of an Access configuration, where <database_id> is your database identifier.

```
#---------------------------------------------------------------
# Database Manager Configuration
#---------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "false"
DatabaseManager.DB.<database_id>.ClassType = "Access"
DatabaseManager.DB.<database_id>.JdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver"
DatabaseManager.DB.<database_id>.Connection.Url = "jdbc:odbc:<data_source>"
DatabaseManager.DB.<database_id>.Connection.User = "Admin"
DatabaseManager.DB.<database_id>.Connection.Password = ""
DatabaseManager.DB.<database_id>.Connection.MaxPreparedStatements = 10
DatabaseManager.DB.<database_id>.Connection.MaxConnectionUsages = 1
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = 30
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = 10000
DatabaseManager.DB.<database_id>.Connection.Logging = false
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 20
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1


DatabaseManager.DB.<database_id>.ObjectId.OidTableName="objectid"
DatabaseManager.DB.<database_id>.ObjectId.NextColumnName="next"
DatabaseManager.defaults.QueryTimeout=0
```

```
DatabaseManager.defaults.DefaultFetchSize=0
DatabaseManager.defaults.CaseInsensitiveDatabase="true"
```

This is the link where you can find all needed information for Microsoft Access:

http://www.microsoft.com/office/access/default.asp

Warning

FOR APPLICATION DEVELOPERS (related to application code)

1. Avoid "DIFFERENCE" clausule in SQL statements: "DIFFERENCE" is unsupported.

2. Don’t use "!=" in SQL expression: "!=" is not supported - use "<>"

3. Don't use "LOWER" function in SQL statements.

InterBase

InterBase® is an efficient and powerful RDBMS engine. Its vendor, Borland/Inprise, has released InterBase version 6.0 as an open-source product. For more information and product downloads see:

http://www.interbase.com [http://www.interbase.com]

InterClient

The JDBC driver for InterBase is called InterClient™ The InterClient system includes an all-Java thin client, and a server-side daemon (also known as a service on Microsoft Windows NT) called InterServer. This daemon accepts JDBC connection requests and in turn connects to the InterBase RDBMS daemon. The three processes (JDBC client, InterServer daemon, InterBase daemon) can run all on separate hosts, all on the same host, or in any other combination.

InterClient is a class 3 JDBC driver in that it has a separate daemon on the server to serve JDBC connections; however, it also matches the definition of a class 4 driver because the client component can connect only to one DBMS back-end, InterBase.

InterClient is installed separately from InterBase. On Windows, InterClient is commonly installed in:

```
C:\Program Files\Borland\InterClient\interclient.jar
```

Depending on the version of InterClient, it might instead be installed in:

```
C:\Program Files\InterBase Corp\InterClient\interclient.jar
```

Find the JAR file and append its location to your system CLASSPATH environment variable on the client host where you run Java applications.

Different versions of InterClient are available.

- InterClient version 1.50x works only with JDK 1.1x
- InterClient version 1.51x works only with JDK 1.2.x

Whichever version of InterClient you use, you must use the matching version of InterServer.

Configuration

You need to configure both the dods.conf and your <application>.conf to support InterClient.

DODS configuration

You should apply the following configuration edits to dods.conf to make the Standard_JDBC database class match InterBase features. This is necessary because there is not yet a specific com.lutris.appserver.server.sql.interbase package in the Enhydra sources.

```
Database.OidDbType.Standard_JDBC= "DECIMAL(9,0)"
Database.BitType.Standard_JDBC= "SMALLINT"
Database.TimeType.Standard_JDBC= "DATE"
Database.TimestampType.Standard_JDBC= "DATE"
Database.OnCascadeDelete.Standard_JDBC= true
Database.StringQuoteCharacter.Standard_JDBC= '
Database.StringMatch.Standard_JDBC= "LIKE"
Database.StringWildcard.Standard_JDBC= "%"
```

Application configuration

This section presents an example of an Interbase configuration, where <database_id> is your database identifier.

```
#--------------------------------------------------------------------
# Database Manager Configuration
# InterBase / InterClient
#--------------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "false"
DatabaseManager.DB.<database_id>.ClassType = "Standard"
DatabaseManager.DB.<database_id>.JdbcDriver = "interbase.interclient.Driver"
DatabaseManager.DB.<database_id>.Connection.Url =
"jdbc:interbase://loopback/<path_to_database>"
DatabaseManager.DB.<database_id>.Connection.User = "sysdba"
DatabaseManager.DB.<database_id>.Connection.Password = "masterkey"
```

Configuration notes

The JDBC driver class is interbase.interclient.Driver.

Server name

The general URL format for InterClient JDBC connections is as follows:

```
jdbc:interbase://servername/<path_to_database>
```

where <path_to_database> is the full path to the database file, including the name of the database (for example, /usr/local/data/inventory.gdb).

The servername is the hostname or IP address of the server running InterServer, the server-side daemon that accepts JDBC connection requests. If your Enhydra application runs on the same host where InterServer runs, you can use the special servername loopback.

Pathnames

The <path_to_database> is an absolute path to the InterBase database file on the server where the InterBase RDBMS server runs. InterBase does not have abstract handles to databases, like some database products

do (for example, Oracle SIDs or BDE aliases). You must specify the real path to the database. You cannot use mapped drives or NFS filesystems in this path.

Notice the literal slash character (/) following the server name. If the absolute path starts with a slash character (/), then you should have a pair of slash characters (//) together. For example:

```
jdbc:interbase://servername//usr/local/data/inventory.gdb
```

If the server is a Windows host, the path starts with a drive letter identifier:

```
jdbc:interbase://servername/C:/data/inventory.gdb
```

If InterServer runs on a different host than the InterBase RDBMS server, you must specify this host in the path to database, with the following syntax:

```
jdbc:interbase://<interserver_host>/<interbase_host>:<path_to_database>
```

Tip

Slash (/) and backslash (\) characters within path names are interchangeable to InterBase; the InterBase daemon translates these characters as needed to match the convention on the server platform. It is easier to use slashes in code, however, because escape sequences are required to represent backslashes in code.

Ports

nterBase does not take a port number argument in connection strings. InterClient and InterServer always communicate using the TCP/IP service named interserver, which defaults to port 3060. InterServer and InterBase always communicate using the TCP/IP service named gds_db , which defaults to port 3050. These services and port numbers are registered with IANA.

Username and password

The username sysdba and its default password masterkey are used in the example configuration above, but for security reasons it is recommended that you: (a) change the default sysdba password on your InterBase server, and (b) create a non-superuser login in the InterBase password database, and use that login for general database access.

C-JDBC

C-JDBC is a database cluster middleware that allows any Java application (standalone application, servlet or EJB container, ...) to transparently access a cluster of databases through JDBC. You do not have to modify client applications, application servers or database server software. You just have to ensure that all database accesses are performed through JDBC.

First you will need to install C-JDBC. The easiest way to install C-JDBC is to use the Java graphical installer. A Java Virtual Machine is of course needed in this case. Simply launch the installation program by typing:

```
java -jar c-jdbc-x.y.bin-installer.jar
```

(Check CJDBC_HOME environment variable)

Once you have installed the C-JDBC controller, you will find the driver JAR in the drivers/ directory of the controller installation location.

To install the C-JDBC driver, you just have to add the c-jdbc-driver.jar to the client application classpath. This driver replaces the database native driver in the client application.

The database native driver will be used by the C-JDBC controller to access your database. Therefore, the C-JDBC driver and controller can be seen as a proxy between your application and your database native driver.

DODS uses C-JDBC as any standard JDBC driver.

In the application configuration file <appName>/output/conf/<appName>.conf (or better, in <appName>/input/conf/<appName>.conf.in) identify the JDBC driver with the line:

```
DatabaseManager.DB.<database_id>.JdbcDriver=”org.objectweb.cjdbc.driver.Driver"
```

In the same configuration file, identify the database with the database URL .

The JDBC URL expected for the use with C-JDBC is the following:

```
jdbc:cjdbc://host1:port1,host2:port2/database.
```

Host is the machine name (or IP address) where the C-JDBC controller is running, port is the port where the controller is bound on this host.

At least one host must be specified but a list of comma separated hosts can be specified. If several hosts are given, one is picked up randomly from the list. If the currently selected controller fails, another one is automatically picked up from the list.

Default port number is 25322 (C-JDBC on the phone !) if omitted. Those two URL are equivalent:

```
"jdbc:cjdbc://localhost:/tpcw"
"jdbc:cjdbc://localhost:25322/tpcw"
```

So set URL in conf file to:

```
DatabaseManager.DB.<database_id>.Connection.Url = " jdbc:cjdbc://<host>:<port>"
```

Example:

```
#-------------------------------------------------------------------
# Database Manager Configuration
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.DB.<database_id>.ClassType = "Standard"
DatabaseManager.DB.<database_id>.JdbcDriver = =”org.objectweb.cjdbc.driver.Driver"
DatabaseManager.DB.<database_id>.Connection.Url = " jdbc:cjdbc://<host>:<port>"
DatabaseManager.DB.<database_id>.Connection.User = "<name>"
DatabaseManager.DB.<database_id>.Connection.Password = "<password>"
DatabaseManager.DB.<database_id>.Connection.MaxPoolSize = "2"
DatabaseManager.DB.<database_id>.Connection.AllocationTimeout = "2"
DatabaseManager.DB.<database_id>.Connection.Logging = "true"
DatabaseManager.DB.<database_id>.ObjectId.CacheSize = 2
DatabaseManager.DB.<database_id>.ObjectId.MinValue = 1
```

Details of creating, configuring and starting C-JDBC database is out of scope of this document. All details about these features can be found on http://c-jdbc.objectweb.org

C-JDBC is a free, open source ObjectWeb Consortium (http://www.objectweb.org)'s project. It is licensed under the GNU Lesser General Public License