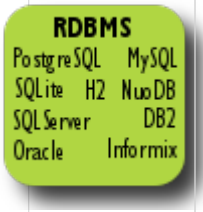
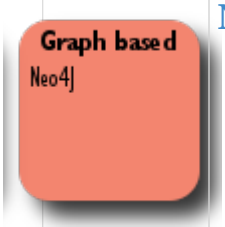
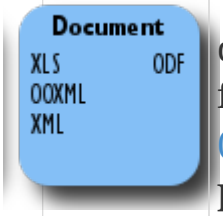
Datastores Guide (v5.0)

Table of Contents

The DataNucleus AccessPlatform is designed for flexibility to operate with any type of datastore. We already support a very wide range of datastores and this will only increase in the future. In this section you can find the specifics for particular supported datastores over and above what was already addressed for JDO and JPA persistence.

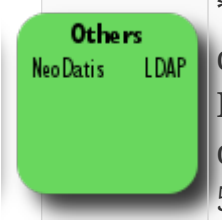
	RDBMS : tried and tested since the 1970s, relational databases form an integral component of many systems. They incorporate optimised querying mechanisms, yet also can suffer from object-relational impedance mismatch in some situations. They also require an extra level of configuration to map from objects across to relational tables/columns.
	* HBase : HBase is a map-based datastore originated within Hadoop, following the model of BigTable. * Cassandra : Cassandra is a distributed robust clustered datastore.
	Neo4J : plugin providing persistence to the Neo4j graph store



* Open Document Format (ODF) : ODF is an international standard document format, and its spreadsheets provide a widely used form for publishing of data, making it available to other groups. * Excel (XLS) : Excel spreadsheets provide a widely used format allowing publishing of data, making it available to other groups (XLS format). * Excel (OOXML) : Excel spreadsheets provide a widely used format allowing publishing of data, making it available to other groups (OOXML format). * XML : XML defines a document format and, as such, is a key data transfer medium.

* JSON : another format of document for exchange, in this case with particular reference to web contexts. * Amazon S3 : Amazon Simple Storage Service. * Google Storage : Google Storage.

MongoDB : plugin providing persistence to the MongoDB NoSQL datastore



* LDAP : an internet standard datastore for indexed data that is not changing significantly. * *NeoDatis* : an open source object datastore. Fast persistence of large object graphs, without the necessity of any object-relational mapping (no longer supported, use DataNucleus 5.0).

If you have a requirement for persistence to some other datastore, then it would likely be easily provided by creation of a DataNucleus *StoreManager*. Please contact us so that you can provide this and contribute it back to the community.

RDBMS Datastores

DataNucleus supports persisting objects to RDBMS datastores (using the datanucleus-rdbms plugin). It supports the vast majority of RDBMS products available today. DataNucleus communicates with the RDBMS datastore using JDBC. RDBMS systems accept varying standards of SQL and so DataNucleus will support particular RDBMS/JDBC combinations only, though clearly we try to support as many as possible.

The jars required to use DataNucleus RDBMS persistence are `datanucleus-core`, `datanucleus-api-jdo` /`datanucleus-api-jpa`, `datanucleus-rdbms` and `JDBC driver`.

There are tutorials available for use of DataNucleus with RDBMS for JDO and for JPA

By default when you create a PersistenceManagerFactory or EntityManagerFactory to connect to a particular datastore DataNucleus will automatically detect the *datastore adapter* to use and will use its own internal adapter for that type of datastore. If you find that either DataNucleus has incorrectly detected the adapter to use, you can override the default behaviour using the persistence property **datanucleus.rdbms.datastoreAdapterClassName**.

Using an RDBMS datastore DataNucleus allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- SQL - language found on alomst all RDBMS.
- JPQL - language defined in the JPA specification which closely mirrors SQL.

The following RDBMS have support built in to DataNucleus. Click on the one of interest to see details of any provisos for its support, as well as the JDBC connection information

- MySQL/MariaDB
- PostgreSQL Database
- PostgreSQL+PostGIS Database
- HSQL DB
- H2 Database
- SQLite
- Apache Derby
- Microsoft SQLServer
- Sybase
- SQL Anywhere
- Oracle
- IBM DB2
- IBM Informix
- Firebird
- NuoDB
- SAPDB/MaxDB
- Virtuoso
- Pointbase
- Oracle TimesTen

Note that if your RDBMS is not listed or currently supported you can easily write your own Datastore Adapter for it raise an issue in GitHub when you have it working and attach a patch to contribute it. Similarly if you are using an adapter that has some problem on your case you could use the same plugin mechanism to override the non-working feature.

DB2

To specify DB2 as your datastore, you will need something like the following specifying (where "mydb1" is the name of the database)

```
datanucleus.ConnectionDriverName=com.ibm.db2.jcc.DB2Driver
datanucleus.ConnectionURL=jdbc:db2://localhost:50002/mydb1
datanucleus.ConnectionUserName='username'     (e.g db2inst1)
datanucleus.ConnectionPassword='password'
```

With DB2 Express-C v9.7 you need to have db2jcc.jar and db2jcc_license_cu.jar in the CLASSPATH.

MySQL

MySQL and its more developed drop in replacement MariaDB are supported as an RDBMS datastore by DataNucleus with the following provisos

- You can set the table (engine) type for any created tables via persistence property **datanucleus.rdbms.mysql.engineType** or by setting the extension metadata on a class with key *mysql-engine-type*. The default is INNODB
- You can set the collation type for any created tables via persistence property **datanucleus.rdbms.mysql.collation** or by setting the extension metadata on a class with key *mysql-collation*
- You can set the character set for any created tables via persistence property **datanucleus.rdbms.mysql.characterSet** or by setting the extension metadata on a class with key *mysql-character-set*
- JDOQL.isEmpty()/contains() will not work in MySQL 4.0 (or earlier) since the query uses EXISTS and that is only available from MySQL 4.1
- MySQL on Windows MUST specify *datanucleus.identifier.case* as "LowerCase" since the MySQL server stores all identifiers in lowercase BUT the mysql-connector-java JDBC driver has a bug (in versions up to and including 3.1.10) where it claims that the MySQL server stores things in mixed case when it doesnt
- MySQL 3.* will not work reliably with inheritance cases since DataNucleus requires UNION and this doesn't exist in MySQL 3.*
- MySQL before version 4.1 will not work correctly on JDOQL Collection.size(), Map.size() operations since this requires subqueries, which are not supported before MySQL 4.1.
- If you receive an error "Incorrect arguments to mysql_stmt_execute" then this is a bug in MySQL and you need to update your JDBC URL to append "?useServerPrepStmts=false".
- MySQL throws away the milliseconds on a Date and so cannot be used reliably for Optimistic locking using strategy "date-time" (use "version" instead)
- You can specify "BLOB", "CLOB" JDBC types when using MySQL with DataNucleus but you must turn validation of columns OFF. This is because these types are not supported by the MySQL JDBC driver and it returns them as LONGVARBINARY/LONGVARCHAR when querying the column type

To specify MySQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=com.mysql.jdbc.Driver
datanucleus.ConnectionURL=jdbc:mysql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

SQL Server

Microsoft SQLServer is supported as an RDBMS datastore by DataNucleus with the following proviso

- SQLServer 2000 does not keep accuracy on *datetime* datatypes. This is an SQLServer 2000 issue. In order to keep the accuracy when storing *java.util.Date* java types, use *int* datatype.

To specify SQLServer as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

Microsoft SQLServer 2005 JDBC Driver (Recommended)

```
datanucleus.ConnectionDriverName=com.microsoft.sqlserver.jdbc.SQLServerDriver
datanucleus.ConnectionURL=jdbc:sqlserver://'host':'port';DatabaseName='db-
name';SelectMethod=cursor
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Microsoft SQLServer 2000 JDBC Driver

```
datanucleus.ConnectionDriverName=com.microsoft.jdbc.sqlserver.SQLServerDriver
datanucleus.ConnectionURL=jdbc:microsoft:sqlserver://'host':'port';DatabaseName='db-
name';SelectMethod=cursor
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Oracle

To specify Oracle as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc) ... you can also use 'oci' instead of 'thin' depending on your driver.

```
datanucleus.ConnectionDriverName=oracle.jdbc.driver.OracleDriver
datanucleus.ConnectionURL=jdbc:oracle:thin:@'host':'port':'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Sybase

To specify Sybase as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=com.sybase.jdbc2.jdbc.SybDriver
datanucleus.ConnectionURL=jdbc:sybase:Tds:'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

SAP SQL Anywhere

To specify SQL Anywhere as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=sybase.jdbc4.sqlanywhere.IDriver
datanucleus.ConnectionURL=jdbc:sqlanywhere:uid=DBA;pwd=sql;eng=demo
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

HSQLDB

HSQLDB is supported as an RDBMS datastore by DataNucleus with the following proviso

- Use of batched statements is disabled since HSQLDB has a bug where it throws exceptions "batch failed" (really informative). Still waiting for this to be fixed in HSQLDB
- Use of JDOQL/JPQL subqueries cannot be used where you want to refer back to the parent query since HSQLDB up to and including version 1.8 don't support this.

To specify HSQL (1.x) as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.hsqldb.jdbcDriver
datanucleus.ConnectionURL=jdbc:hsqldb:hsql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Note that in HSQLDB v2.x the driver changes to *org.hsqldb.jdbc.JDBCDriver*

H2

H2 is supported as an RDBMS datastore by DataNucleus.

To specify H2 as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.h2.Driver
datanucleus.ConnectionURL=jdbc:h2:'db-name'
datanucleus.ConnectionUserName=sa
datanucleus.ConnectionPassword=
```

Informix

Informix is supported as an RDBMS datastore by DataNucleus.

To specify Informix as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=com.informix.jdbc.IfxDriver
datanucleus.ConnectionURL=jdbc:informix-
sqli://[{ip|host}:port][/dbname]:INFORMIXSERVER=servername[;name=value[;name=value]...
]
datanucleus.ConnectionUserName=informix
datanucleus.ConnectionPassword=password
```

For example

```
datanucleus.ConnectionDriverName=com.informix.jdbc.IfxDriver
datanucleus.ConnectionURL=jdbc:informix-
sqli://192.168.254.129:9088:informixserver=demo_on;database=buf_log_db
datanucleus.ConnectionUserName=informix
datanucleus.ConnectionPassword=password
```

Note that some database logging options in Informix do not allow changing autoCommit dinamically. You need to rebuild the database to support it. To rebuild the database refer to Informix documention, but as example,

```
run $INFORMIXDIR\bin\dbaccess and execute the command "CREATE DATABASE mydb WITH
BUFFERED LOG".
```

INDEXOF: Informix 11.x does not have a function to search a string in another string. DataNucleus defines a user defined function, DATANUCLEUS_STRPOS, which is automatically created on startup. The SQL for the UDF function is:

```
create function DATANUCLEUS_STRPOS(str char(40),search char(40),from smallint)
returning smallint
    define i,pos,lenstr,lensearch smallint;
    let lensearch = length(search);
    let lenstr = length(str);

    if lenstr=0 or lensearch=0 then return 0; end if;

    let pos=-1;
    for i=1+from to lenstr
        if substr(str,i,lensearch)=search then
            let pos=i;
            exit for;
        end if;
    end for;
    return pos;
end function;
```

PostgreSQL

To specify PostgreSQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgresql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

PostgreSQL with PostGIS extension

To specify PostGIS as your datastore, you will need to decide first which geometry library you want to use and then set the connection url accordingly.

For the PostGIS JDBC geometries you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgresql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

For Oracle's JGeometry you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgres_jgeom://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

For the JTS (Java Topology Suite) geometries you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgres_jts://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Apache Derby

To specify Apache Derby as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=org.apache.derby.jdbc.EmbeddedDriver
datanucleus.ConnectionURL=jdbc:derby:'db-name';create=true
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Above settings are used together with the Apache Derby in embedded mode. The below settings are used in network mode, where the default port number is 1527.

```
datanucleus.ConnectionDriverName=org.apache.derby.jdbc.ClientDriver
datanucleus.ConnectionURL=jdbc:derby://'hostname':'portnumber'/'db-name';create=true
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

org.apache.derby.jdbc.ClientDriver

ASCII: Derby 10.1 does not have a function to convert a char into ascii code. DataNucleus needs such function to converts chars to int values when performing queries converting chars to ints. DataNucleus defines a user defined function, DataNucleus_ASCII, which is automatically created on startup. The SQL for the UDF function is:

```
DROP FUNCTION NUCLEUS_ASCII;
CREATE FUNCTION NUCLEUS_ASCII(C CHAR(1)) RETURNS INTEGER
EXTERNAL NAME 'org.datanucleus.store.rdbms.adapter.DerbySQLFunction.ascii'
CALLED ON NULL INPUT
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

String.matches(pattern): When pattern argument is a column, DataNucleus defines a function that allows Derby 10.1 to perform the matches function. The SQL for the UDF function is:

```
DROP FUNCTION NUCLEUS_MATCHES;
CREATE FUNCTION NUCLEUS_MATCHES(TEXT VARCHAR(8000), PATTERN VARCHAR(8000)) RETURNS
INTEGER
EXTERNAL NAME 'org.datanucleus.store.rdbms.adapter.DerbySQLFunction.matches'
CALLED ON NULL INPUT
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

Firebird

Firebird is supported as an RDBMS datastore by DataNucleus with the proviso that

- Auto-table creation is severely limited with Firebird. In Firebird, DDL statements are not auto-committed and are executed at the end of a transaction, after any DML statements. This makes "on the fly" table creation in the middle of a DML transaction not work. You must make sure that "autoStartMechanism" is NOT set to "SchemaTable" since this will use DML. You must also make sure that nobody else is connected to the database at the same time. Don't ask us why such limitations are in a RDBMS, but then it was you that chose to use it ;-)

To specify Firebird as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=org.firebirdsql.jdbc.FBDriver
datanucleus.ConnectionURL=jdbc:firebirdsql://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

NuoDB

To specify NuoDB as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=com.nuodb.jdbc.Driver
datanucleus.ConnectionURL=jdbc:com.nuodb://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
datanucleus.Schema={my-schema-name}
```

SAPDB/MaxDB

To specify SAPDB/MaxDB as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=com.sap.dbtech.jdbc.DriverSapDB
datanucleus.ConnectionURL=jdbc:sapdb://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

SQLite

SQLite is supported as an RDBMS datastore by DataNucleus with the proviso that

- When using sequences, you must set the persistence property **datanucleus.valuegeneration.transactionAttribute** to **UsePM**

To specify SQLite as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=org.sqlite.JDBC
datanucleus.ConnectionURL=jdbc:sqlite:'db-name'
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

Virtuoso

To specify Virtuoso as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=virtuoso.jdbc.Driver
datanucleus.ConnectionURL=jdbc:virtuoso://127.0.0.1/{dbname}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

Pointbase

To specify Pointbase as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=com.pointbase.jdbc.jdbcUniversalDriver
datanucleus.ConnectionURL=jdbc:pointbase://127.0.0.1/{dbname}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

JDBC Driver parameters

If you need to pass additional parameters to the JDBC driver you can append these to the end of the

datanucleus.ConnectionURL. For example,

```
datanucleus.ConnectionURL=jdbc:mysql://localhost?useUnicode=true&amp;characterEncoding
=UTF-8
```

RDBMS : Statement Batching

When changes are required to be made to an underlying RDBMS datastore, statements are sent via JDBC. A statement is, in general, a single SQL command, and is then executed. In some circumstances the statements due to be sent to the datastore are the same JDBC statement several times. In this case the statement can be *batched*. This means that a statement is created for the SQL, and it is passed to the datastore with multiple sets of values before being executed. When it is executed the SQL is executed for each of the sets of values. DataNucleus allows statement batching under certain circumstances.

The maximum number of statements that can be included in a *batch* can be set via a persistence property **datanucleus.rdbms.statementBatchLimit**. This defaults to 50. If you set it to -1 then there is no maximum limit imposed. Setting it to 0 means that batching is turned off.

It should be noted that while batching sounds essential, it is only of any possible use when the exact same SQL is required to be executed more than 1 times in a row. If a different SQL needs executing between 2 such statements then no batching is possible anyway.. Let's take an example

```
INSERT INTO MYTABLE VALUES(?,?,?,?)
INSERT INTO MYTABLE VALUES(?,?,?,?)
SELECT ID, NAME FROM MYOTHERTABLE WHERE VALUE=?
INSERT INTO MYTABLE VALUES(?,?,?,?)
SELECT ID, NAME FROM MYOTHERTABLE WHERE VALUE=?
```

In this example the first two statements can be batched together since they are identical and nothing else separates them. All subsequent statements cannot be batched since no two identical statements follow each other.

The statements that DataNucleus currently allows for batching are

- Insert of objects. This is not enabled when objects being inserted are using *identity* value generation strategy
- Delete of objects
- Insert of container elements/keys/values
- Delete of container elements/keys/values

Please note that if using MySQL, you should also specify the connection URL with the argument *rewriteBatchedStatements=true* since MySQL won't actually batch without this

RDBMS : Datastore Schema API

JDO/JPA are APIs for persisting and retrieving objects to/from datastores. They don't provide a way of accessing the schema of the datastore itself (if it has one). In the case of RDBMS it is useful to be able to find out what columns there are in a table, or what data types are supported for example. DataNucleus Access Platform provides an API for this.

The first thing to do is get your hands on the DataNucleus *StoreManager* and from that the *StoreSchemaHandler*. You do this as follows

```
import org.datanucleus.api.jdo.JDOPersistenceManagerFactory;
import org.datanucleus.store.StoreManager;
import org.datanucleus.store.schema.StoreSchemaHandler;

[assumed to have "pmf"]
...

StoreManager storeMgr = ((JDOPersistenceManagerFactory)pmf).getStoreManager();
StoreSchemaHandler schemaHandler = storeMgr.getSchemaHandler();
```

So now we have the *StoreSchemaHandler* what can we do with it? Well start with the javadoc for the implementation that is used for RDBMS 

Datastore Types Information

So we now want to find out what JDBC/SQL types are supported for our RDBMS. This is simple.

```
import org.datanucleus.store.rdbms.schema.RDBMSTypesInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTypesInfo typesInfo = schemaHandler.getSchemaData(conn, "types");
```

As you can see from the javadocs for *RDBMSTypesInfo* 

we can access the JDBC types information via the "children". They are keyed by the JDBC type number of the JDBC type (see java.sql.Types). So we can just iterate it

```
Iterator jdbcTypesIter = typesInfo.getChildren().values().iterator();
while (jdbcTypesIter.hasNext())
{
    JDBCTypeInfo jdbcType = (JDBCTypeInfo)jdbcTypesIter.next();

    // Each JDBCTypeInfo contains SQLTypeInfo as its children, keyed by SQL name
    Iterator sqlTypesIter = jdbcType.getChildren().values().iterator();
    while (sqlTypesIter.hasNext())
    {
        SQLTypeInfo sqlType = (SQLTypeInfo)sqlTypesIter.next();
        ... inspect the SQL type info
    }
}
```

Column information for a table

Here we have a table in the datastore and want to find the columns present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableInfo tableInfo = schemaHandler.getSchemaData(conn, "columns",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableInfo*  we can access the columns information via the "children".

```
Iterator columnsIter = tableInfo.getChildren().iterator();
while (columnsIter.hasNext())
{
    RDBMSColumnInfo colInfo = (RDBMSColumnInfo)columnsIter.next();

    ...
}
```

Index information for a table

Here we have a table in the datastore and want to find the indices present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableIndexInfo tableInfo = schemaHandler.getSchemaData(conn, "indices",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableIndexInfo* Javadoc we can access the index information via the "children".

```
Iterator indexIter = tableInfo.getChildren().iterator();
while (indexIter.hasNext())
{
    IndexInfo idxInfo = (IndexInfo)indexIter.next();

    ...
}
```

ForeignKey information for a table

Here we have a table in the datastore and want to find the FKs present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableFKInfo tableInfo = schemaHandler.getSchemaData(conn, "foreign-keys",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableFKInfo* Javadoc we can access the foreign-key information via the "children".

```
Iterator fkIter = tableInfo.getChildren().iterator();
while (fkIter.hasNext())
{
    ForeignKeyInfo fkInfo = (ForeignKeyInfo)fkIter.next();

    ...
}
```

PrimaryKey information for a table

Here we have a table in the datastore and want to find the PK present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTablePKInfo tableInfo = schemaHandler.getSchemaData(conn, "primary-keys",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTablePKInfo* Javadoc we can access the foreign-key information via the "children".

```
Iterator pkIter = tableInfo.getChildren().iterator();
while (pkIter.hasNext())
{
    PrimaryKeyInfo pkInfo = (PrimaryKeyInfo)pkIter.next();

    ...
}
```

Cassandra Datastores

DataNucleus supports a limited for of persisting/retrieving objects to/from Cassandra datastores (using the datanucleus-cassandra plugin, which utilises the DataStax Java driver). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=cassandra:[{host1}[:{port}] [,{host2} [,{host3}]]]
```

where it will create a Cassandra *cluster* with contact points of *host1* (*host2*, *host3* etc), and if the port is specified on the first host then will use that as the port (no port specified on alternate hosts).

For example, to connect to a local server

```
datanucleus.ConnectionURL=cassandra:
```

The jars required to use DataNucleus Cassandra persistence are `datanucleus-core`, `datanucleus-api-jdo`/`datanucleus-api-jpa`, `datanucleus-cassandra` and `cassandra-driver-core`.

There are tutorials available for use of DataNucleus with Cassandra for JDO and for JPA

Things to bear in mind with Cassandra usage :-

- Creation of a PMF/EMF will create a *Cluster*. This will be closed then the PMF/EMF is closed.
- Any PM/EM will use a single *Session*, by default, shared amongst all PM/EMs.
- If you specify the persistence property **datanucleus.cassandra.sessionPerManager** to *true* then each PM/EM will have its own *Session* object.
- Cassandra doesn't use transactions, so any JDO/JPA transaction operation is a no-op (i.e will be ignored).
- This uses Cassandra 3.x (and CQL v3.x), not Thrift (like the previous unofficial attempts at a datanucleus-cassandra plugin used)
- Specify persistence property **datanucleus.cassandra.metrics** to enable/disable metrics
- Specify persistence property **datanucleus.cassandra.compression** to enable/disable compression
- Specify persistence property **datanucleus.cassandra.ssl** to enable/disable SSL
- Specify persistence property **datanucleus.cassandra.socket.readTimeoutMillis** to set the timeout for reads (in ms)
- Specify persistence property **datanucleus.cassandra.socket.connectTimeoutMillis** to set the

timeout for connecting (in ms)

- You need to specify the "schema" (*datanucleus.mapping.Schema*)
- Queries are evaluated in-datastore when they only have (indexed) members and literals and using the operators ==, !=, >, >=, <, <=, &&, ||.
- You can query the datastore using JDOQL, JPQL, or CQL

Queries : Cassandra CQL Queries

If you choose to use Cassandra CQL Queries then these are not portable to any other datastore. Use JDOQL/JPQL for portability

Cassandra provides the CQL query language. To take a simple example using the JDO API

```
// Find all employees
PersistenceManager persistenceManager = pmf.getPersistenceManager();
Query q = pm.newQuery("CQL", "SELECT * FROM schema1.Employee");
// Fetch 10 Employee rows at a time
query.getFetchPlan().setFetchSize(10);
query.setResultClass(Employee.class);
List<Employee> results = (List)q.execute();
```

You can also query results as List<Object[]> without specifying a specific result type as shown below.

```
// Find all employees
PersistenceManager persistenceManager = pmf.getPersistenceManager();
Query q = pm.newQuery("CQL", "SELECT * FROM schema1.Employee");
// Fetch all Employee rows as Object[] at a time.
query.getFetchPlan().setFetchSize(-1);
List<Object[]> results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the Cassandra "CQL".

If you wanted to use CQL with the JPA API, you would do

```
// Find all employees
Query q = em.createNativeQuery("SELECT * FROM schema1.Employee", Employee.class);
List<Employee> results = q.getResultList();
```

Note that the last argument to *createNativeQuery* is optional and you would get *List<Object[]>* returned otherwise.

Excel Datastores

DataNucleus supports persisting/retrieving objects to/from Excel documents (using the datanucleus-excel plugin, which makes use of the Apache POI project). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=excel:file:myfile.xls
```

replacing `myfile.xls` with your filename, which can be absolute or relative. This connects to a file on your local machine. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus Excel persistence are `datanucleus-core`, `datanucleus-api-jdo` /`datanucleus-api-jpa`, `datanucleus-excel` and `apache-poi`.

There are tutorials available for use of DataNucleus with Excel for JDO and for JPA

Things to bear in mind with Excel usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed **in-memory**
- Relations : A spreadsheet cannot store related objects directly, since each object is a row of a particular worksheet. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell.

OOXML Datastores

DataNucleus supports persisting/retrieving objects to/from OOXML documents (using the datanucleus-excel plugin) which makes use of the Apache POI project. Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=excel:file:myfile.xlsx
```

replacing `myfile.xlsx` with your filename, which can be absolute or relative. This connects to a file on your local machine. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus OOXML persistence are `datanucleus-core`, `datanucleus-api-jdo`/`datanucleus-api-jpa`, `datanucleus-excel` and `apache-poi`.

There are tutorials available for use of DataNucleus with Excel for JDO andfor JPA

Things to bear in mind with OOXML usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed **in-memory**
- Relations : A spreadsheet cannot store related objects directly, since each object is a row of a particular worksheet. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell.

ODF Datastores

DataNucleus supports persisting/retrieving objects to/from ODF documents (using the datanucleus-odf plugin, which makes use of the ODFDOM project). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=odf:file:myfile.ods
```

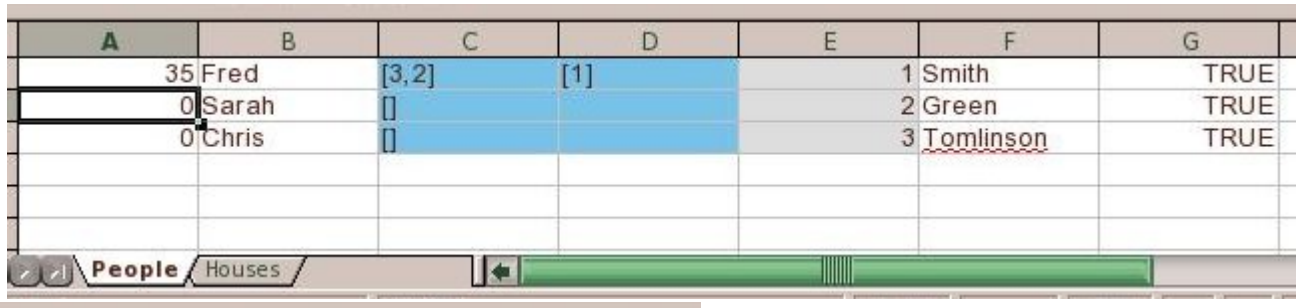
replacing `myfile.ods` with your filename, which can be absolute or relative. This connects to a file on your local machine. You then create your PMF/EMF as normal and use JDO/JPA as normal.

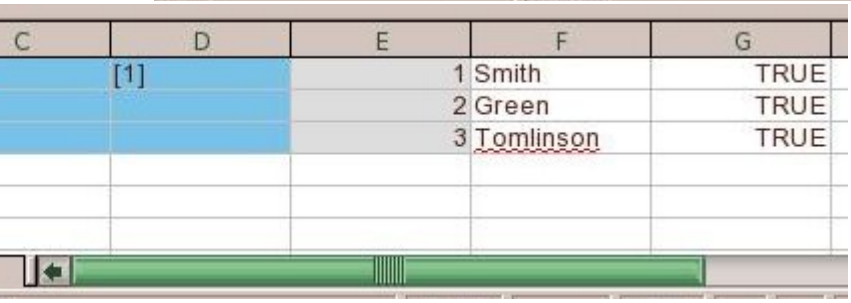
The jars required to use DataNucleus ODF persistence are `datanucleus-core`, `datanucleus-api-jdo` /`datanucleus-api-jpa`, `datanucleus-odf` and `odftoolkit`.

There are tutorials available for use of DataNucleus with ODF for JDO and for JPA

Things to bear in mind with ODF usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed **in-memory**
- Relations : A spreadsheet cannot store related objects directly, since each object is a row of a particular worksheet. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell. See this

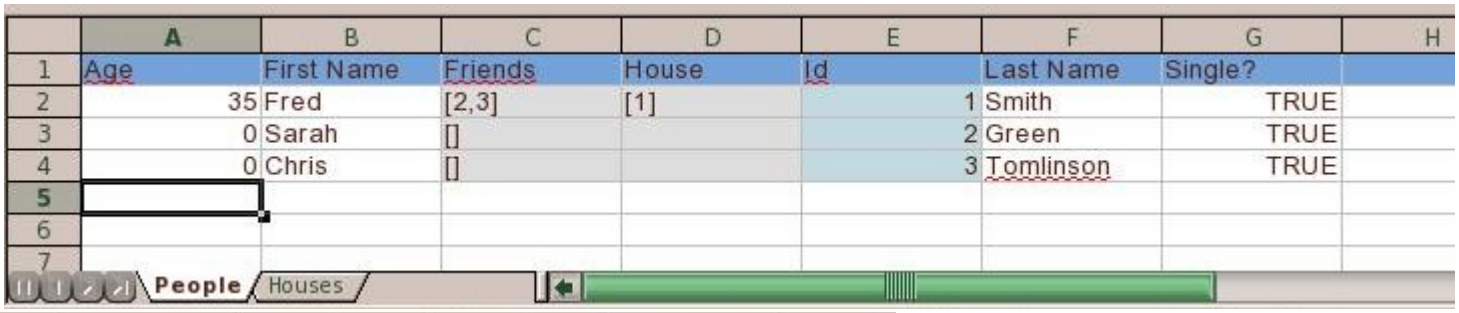




Worksheet Headers

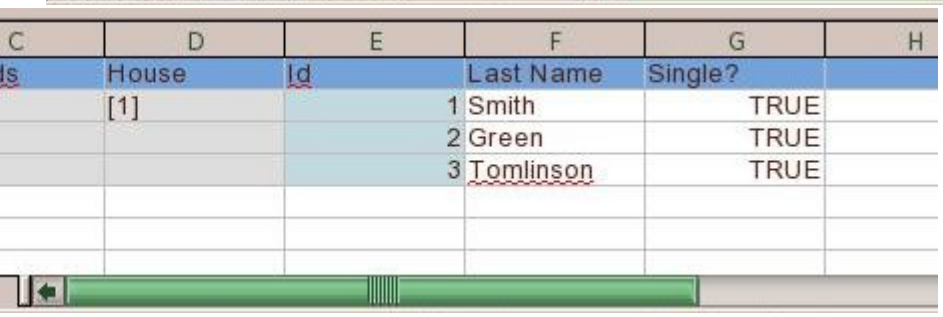
A typical spreadsheet has many rows of data. It contains no names of columns tying the data back

to the input object (field names). DataNucleus allows an extension specified at *class* level called **include-column-headers** (should be set to true). When the table is then created it will include an extra row (the first row) with the column names from the metadata (or field names if no column names were defined). For example



	A	B	C	D	E	F	G	H
1	Age	First Name	Friends	House	Id	Last Name	Single?	
2	35	Fred	[2,3]	[1]	1	Smith	TRUE	
3	0	Sarah	[]		2	Green	TRUE	
4	0	Chris	[]		3	Tomlinson	TRUE	
5								
6								
7								

People Houses



C	D	E	F	G	H
ls	House	Id	Last Name	Single?	
	[1]	1	Smith	TRUE	
		2	Green	TRUE	
		3	Tomlinson	TRUE	

XML Datastores

DataNucleus supports persisting/retrieving objects to/from XML documents (using the datanucleus-xml plugin). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=xml:file:myfile.xml
```

replacing `myfile.xml` with your filename, which can be absolute or relative.

It makes use of JAXB, and the jars required to use DataNucleus XML persistence are `datanucleus-core`, `datanucleus-api-jdo`/`datanucleus-api-jpa`, `datanucleus-xml` and `JAXB API`, `JAXB Reference Implementation`. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us.

Things to bear in mind with XML usage :-

- Indentation of XML : the persistence property **datanucleus.xml.indentSize** defaults to 4 but you can set it to the desired indent size
- Querying using JDOQL/JPQL will operate in-memory currently.
- Application identity is supported but can only have 1 PK field and must be a String. This is a limitation of JAXB
- Persistent properties are not supported, only persistent fields
- Out of the box it will use the JAXB reference implementation. You could, in principle, provide support for other JAXB implementations by implementing *org.datanucleus.store.xml.JAXBHandler* and then specify the persistence property **datanucleus.xml.jaxbHandlerClass** to the JAXBHandler implementation. If you do manage to write a JAXBHandler for other JAXB implementations please consider contributing it to the project

Mapping : XML Datastore Mapping

When persisting a Java object to an XML datastore clearly the user would like some control over the structure of the XML document. Here's an example using JDO XML MetaData

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Person" detachable="true" schema="/myproduct/people"
table="person">
            <field name="personNum">
                <extension vendor-name="datanucleus" key="XmlAttribute" value="true"/>
            </field>
            <field name="firstName" primary-key="true"/> <!-- PK since JAXB requires
String -->
            <field name="lastName"/>
            <field name="bestFriend"/>
        </class>
    </package>
</jdo>
```

Things to note :

- **schema** on class is used to define the "XPath" to the root of the class in XML. You can also use the extension "xpath" to specify the same thing.
- **table** on class is used to define the name of the element for an object of the particular class.
- **column** on field is used to define the name of the element for a field of the particular class.
- **XmlAttribute** : when set to true denotes that this will appear in the XML file as an attribute of the overall element for the object
- When a field is primary-key it will gain a JAXB "XmlID" attribute.
- When a field is a relation to another object (and the field is not embedded) then it will gain a JAXB "XmlIDREF" attribute as a link to the other object.
- **Important : JAXB has a limitation for primary keys** : there can only be a single PK field, and it must be a String!

What is generated with the above is as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<myproduct>
    <people>
        <person personNum="1">
            <firstName>Bugs</firstName>
            <lastName>Bunny</lastName>
            <bestFriend>My</bestFriend>
        </person>
    </people>
</myproduct>
```

Here’s the same example using JDO Annotations

```
@PersistenceCapable(schema="/myproduct/people", table="person")
public class Person
{
    @XmlAttribute
    private long personNum;

    @PrimaryKey
    private String firstName;

    private String lastName;

    private Person bestFiend;

    @XmlElementWrapper(name="phone-numbers")
    @XmlElement(name="phone-number")
    @Element(types=String.class)
    private Map phoneNumbers = new HashMap();

    ...
```

Here's the same example using JPA Annotations (with DataNucleus @Extension annotation) TODO Add this

JSON Datastores

DataNucleus supports persisting/retrieving objects to/from JSON documents (using the datanucleus-json plugin). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=json:{url}
```

replacing "{url}" with some URL of your choice (e.g "http://www.mydomain.com/somepath/"). You then create your PMF/EMF as normal and use JDO/JPA as normal.

Things to bear in mind with JSON usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed **in-memory**
- Relations : DataNucleus stores the id of the related object(s) in the element of the field. If a relation is bidirectional then it will be stored at both ends of the relation; this facilitates easy access to the related object with no need to do a query to find it.

Mapping : HTTP Mapping

The persistence to JSON datastore is performed via HTTP methods. HTTP response codes are used to validate the success or failure to perform the operations. The JSON datastore must respect the following:

Method
Operation
URL format
HTTP response code
PUT
update objects
/{primary key}
HTTP Code 201 (created), 200 (ok) or 204 (no content)
HEAD
locate objects
/{primary key}
HTTP 404 if the object does not exist

POST
insert objects
/
HTTP Code 201 (created), 200 (ok) or 204 (no content)
GET
fetch objects
/{primary key}
HTTP Code 200 (ok) or 404 if object does not exist
GET
retrieve extent of classes (set of objects)
/
HTTP Code 200 (ok) or 404 if no objects exist
DELETE
delete objects
/{primary key}
HTTP Code 202 (accepted), 200 (ok) or 204 (no content)

Mapping : Persistent Classes

Metadata API
Extension Element Attachment
Extension
Description
JDO
/jdo/package/class/extension
url
Defines the location of the resources/objects for the class

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Person" detachable="true">
            <extension vendor-name="datanucleus" key="url" value="/Person"/>
        </class>
    </package>
</jdo>
```

In this example, the *url* extension identifies the Person resources/objects as */Person*. The persistence operations will be relative to this path. e.g /Person/{primary key} will be used for PUT (update), GET (fetch) and DELETE (delete) methods.

Amazon S3 Datastores

DataNucleus supports persisting/retrieving objects to/from Amazon Simple Storage Service (using the datanucleus-json plugin). Simply specify your connection details as follows

```
datanucleus.ConnectionURL=amazons3:http://s3.amazonaws.com/
datanucleus.ConnectionUserName={Access Key ID}
datanucleus.ConnectionPassword={Secret Access Key}
datanucleus.cloud.storage.bucket={bucket}
```

You then create your PMF/EMF as normal and use JDO/JPA as normal.

Things to bear in mind with Amazon S3 usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed **in-memory**

References

Below are some references using this support

- Simple Integration of Datanucleus 2.0.0 + AmazonS3

Google Storage Datastores

DataNucleus supports persisting/retrieving objects to/from Google Storage (using the datanucleus-json plugin). Simply specify your connection details as follows

```
datanucleus.ConnectionURL=googlestorage:http://commondatastorage.googleapis.com/
datanucleus.ConnectionUserName={Access Key ID}
datanucleus.ConnectionPassword={Secret Access Key}
datanucleus.cloud.storage.bucket={bucket}
```

You then create your PMF/EMF as normal and use JDO/JPA as normal.

Things to bear in mind with GoogleStorage usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed **in-memory**

HBase Datastores

DataNucleus supports persisting/retrieving objects to/from HBase datastores (using the datanucleus-hbase plugin, which makes use of the HBase/Hadoop jars). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=hbase[:{server}:{port}]
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

If you just specify the URL as *hbase* then you have a local HBase datastore, otherwise it tries to connect to the datastore at *{server}:{port}*. Alternatively just put "hbase" as the URL and set the zookeeper details in `hbase-site.xml` as normal. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus HBase persistence are `datanucleus-core`, `datanucleus-api-jdo` /`datanucleus-api-jpa`, `datanucleus-hbase` and `hbase-client`.

There are tutorials available for use of DataNucleus with HBase for JDO and for JPA

Things to bear in mind with HBase usage :-

- Creation of a PMF/EMF will create an internal *HBaseConnectionPool*
- Creation of a PM/EM will create/use a *HConnection*.
- Querying can be performed using JDOQL or JPQL. Some components of a filter are handled in the datastore, and the remainder in-memory. Currently any expression of a field (in the same table), or a literal are handled in-datastore, as are the operators &&, ||, >, >=, <, <=, ==, and !=.
- The "row key" will be the PK field(s) when using "application-identity", and the generated id when using "datastore-identity"

Field/Column Naming

By default each field is mapped to a single column in the datastore, with the family name being the name of the table, and the column name using the name of the field as its basis (but following JDO/JPA naming strategies for the precise column name). You can override this as follows

```
@Column(name="{familyName}:{qualifierName}")
String myField;
```

replacing *{familyName}* with the family name you want to use, and *{qualifierName}* with the column name (qualifier name in HBase terminology) you want to use. Alternatively if you don't

want to override the default family name (the table name), then you just omit the "{familyName}:" part and simply specify the column name.

MetaData Extensions

Some metadata extensions (*@Extension*) have been added to DataNucleus to support some of HBase particular table creation options. The supported attributes at Table creation for a column family are:

- **bloomFilter** : An advanced feature available in HBase is Bloom filters, allowing you to improve lookup times given you have a specific access pattern. Default is NONE. Possible values are: ROW → use the row key for the filter, ROWKEY → use the row key and column key (family+qualifier) for the filter.
- **inMemory** : The in-memory flag defaults to false. Setting it to true is not a guarantee that all blocks of a family are loaded into memory nor that they stay there. It is an elevated priority, to keep them in memory as soon as they are loaded during a normal retrieval operation, and until the pressure on the heap (the memory available to the Java-based server processes)is too high, at which time they need to be discarded by force.
- **maxVersions** : Per family, you can specify how many versions of each value you want to keep.The default value is 3, but you may reduce it to 1, for example, in case you know for sure that you will never want to look at older values.
- **keepDeletedCells** : ColumnFamilies can optionally keep deleted cells. That means deleted cells can still be retrieved with Get or Scan operations, as long these operations have a time range specified that ends before the timestamp of any delete that would affect the cells. This allows for point in time queries even in the presence of deletes. Deleted cells are still subject to TTL and there will never be more than "maximum number of versions" deleted cells. A new "raw" scan options returns all deleted rows and the delete markers.
- **compression** : HBase has pluggable compression algorithm, default value is NONE. Possible values GZ, LZO, SNAPPY.
- **blockCacheEnabled** : As HBase reads entire blocks of data for efficient I/O usage, it retains these blocks in an in-memory cache so that subsequent reads do not need any disk operation. The default of true enables the block cache for every read operation. But if your use-case only ever has sequential reads on a particular column family, it is advisable that you disable it from polluting the block cache by setting it to false.
- **timeToLive** : HBase supports predicate deletions on the number of versions kept for each value, but also on specific times. The time-to-live (or TTL) sets a threshold based on the timestamp of a value and the internal housekeeping is checking automatically if a value exceeds its TTL. If that is the case, it is dropped during major compactions

To express these options, a format similar to a properties file is used such as:

```
hbase.columnFamily.[family name to apply property on].[attribute] = {value}
```

where:

- attribute: One of the above defined attributes (inMemory, bloomFilter,...)
- family name to apply property on: The column family affected.
- value: Associated value for this attribute.

Let's take an example applying column family/qualifiers, setting the bloom filter option to ROWKEY, and the in-memory flag to true would look like. Firstly JDO Annotations:-

```
@PersistenceCapable
@Extension(vendorName = "datanucleus", key = "hbase.columnFamily.meta.bloomFilter",
value = "ROWKEY")
@Extension(vendorName = "datanucleus", key = "hbase.columnFamily.meta.inMemory", value
= "true")
public class MyClass
{
    @PrimaryKey
    private long id;

    // column family data, name of attribute blob
    @Column(name = "data:blob")
    private String blob;

    // column family meta, name of attribute firstName
    @Column(name = "meta:firstName")
    private String firstName;

    // column family meta, name of attribute firstName
    @Column(name = "meta:lastName")
    private String lastName;

   [ ... getter and setter ... ]
}
```

or using XML

```
<class name="MyClass">
    <extension vendor-name="datanucleus" key="hbase.columnFamily.meta.bloomFilter"
value="ROWKEY"/>
    <extension vendor-name="datanucleus" key="hbase.columnFamily.meta.inMemory"
value="true"/>
    <field name="id" primary-key="true"/>
    <field name="blob">
        <column name="data:blob"/>
    </field>
    <field name="firstName">
        <column name="meta:firstName"/>
    </field>
    <field name="lastName">
        <column name="meta:lastName"/>
    </field>
</class>
```

Now JPA Annotations:-

```
@Entity
@org.datanucleus.api.jpa.annotations.Extensions({
    @org.datanucleus.api.jpa.annotations.Extension(key =
"hbase.columnFamily.meta.bloomFilter", value = "ROWKEY"),
    @org.datanucleus.api.jpa.annotations.Extension(key =
"hbase.columnFamily.meta.inMemory", value = "true")
})
public class MyClass
{
    @Id
    private long id;

    // column family data, name of attribute blob
    @Column(name = "data:blob")
    private String blob;

    // column family meta, name of attribute firstName
    @Column(name = "meta:firstName")
    private String firstName;

    // column family meta, name of attribute firstName
    @Column(name = "meta:lastName")
    private String lastName;

   [ ... getter and setter ... ]
}
```

or using XML

```
<entity class="mydomain.MyClass">
    <extension vendor-name="datanucleus" key="hbase.columnFamily.meta.bloomFilter"
value="ROWKEY"/>
    <extension vendor-name="datanucleus" key="hbase.columnFamily.meta.inMemory"
value="true"/>
    <attributes>
        <id name="id"/>
        <basic name="blob">
            <column name="data:blob"/>
        </basic>
        <basic name="firstName">
            <column name="meta:firstName"/>
        </basic>
        <basic name="lastName">
            <column name="meta:lastName"/>
        </basic>
    </attributes>
</entity>
```

References

Below are some references using this support

- Apache Hadoop HBase plays nicely with JPA
- HBase with JPA and Spring Roo
- Value Generator plugin for HBase and DataNucleus

MongoDB Datastores

DataNucleus supports persisting/retrieving objects to/from MongoDB datastores (using the datanucleus-mongodb plugin, which utilises the Mongo Java driver). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=mongodb:[{server}][/{dbName}] [,{server2}[,server3}]]
```

For example, to connect to a local server, with database called "myMongoDB"

```
datanucleus.ConnectionURL=mongodb:/myMongoDB
```

If you just specify the URL as *mongodb* then you have a local MongoDB datastore called "DataNucleus", otherwise it tries to connect to the datastore *{dbName}* at *{server}*. The multiple *{server}* option allows you to run against MongoDB replica sets. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus MongoDB persistence are `datanucleus-core`, `datanucleus-api-jdo`/`datanucleus-api-jpa`, `datanucleus-mongodb` and `mongo-java-driver`.

There are tutorials available for use of DataNucleus with MongoDB for JDO and for JPA

Things to bear in mind with MongoDB usage :-

- Creation of a PMF/EMF will create a *MongoClient*. This will be closed then the PMF/EMF is closed.
- Creation of a PM/EM and performing an operation will obtain a *DB* object from the *MongoClient*. This is pooled by the MongoClient so is managed by MongoDB. Closing the PM/EM will stop using that *DB*
- You can set the number of connections per host with the persistence property **datanucleus.mongodb.connectionsPerHost**
- Querying can be performed using JDOQL or JPQL. Some components of a filter are handled in the datastore, and the remainder in-memory. Currently any expression of a field (in the same table), or a literal are handled **in-datastore**, as are the operators &&, ||, >, >=, <, <=, ==, and !=. Note that if something falls back to being evaluated **in-memory** then it can be much slower, and this will be noted in the log, so people are advised to design their models and queries to avoid that happening if performance is a top priority.
- If you want a query to be runnable on a slave MongoDB instance then you should set the query extension (JDO) / hint (JPA) **slave-ok** as *true*, and when executed it can be run on a slave instance.

- All objects of a class are persisted to a particular "document" (specifiable with the "table" in metadata), and a field of a class is persisted to a particular "field" ("column" in the metadata).
- Relations : DataNucleus stores the id of the related object(s) in a field of the owning object. When a relation is bidirectional both ends of the relation will store the relation information.
- Capped collections : you can specify the extension metadata key *mongodb.capped.size* as the number of bytes of the size of the collection for the class in question.
- If you want to specify the max number of connections per host with MongoDB then set the persistence property **datanucleus.mongodb.connectionsPerHost**
- If you want to specify the MongoDB *threadsAllowedToBlockForConnectionMultiplier*, then set the persistence property **datanucleus.mongodb.threadsAllowedToBlockForConnectionMultiplier**

Mapping : Embedded Persistable fields

When you have a field in a class that is of a persistable type you sometimes want to store it with the owning object. In this case you can use JDO / JPA embedding of the field. DataNucleus offers two ways of performing this embedding

- The default is to store the object in the field as a sub-document (nested) of the owning document. Similarly if that sub-object has a field of a persistable type then that can be further nested.
- The alternative is to store each field of the sub-object as a field of the owning document (flat embedding). Similarly if that sub-object has a field of a persistable type then it can be flat embedded in the same way

For JDO this would be defined as follows (for JPA just swap @PersistenceCapable for @Entity)

```
@PersistenceCapable
public class A
{
    @Embedded
    B b;

    ...
}
```

This example uses the default embedding, using a nested document within the owner document, and could look something like this

```
{ "name" : "A Name" ,
  "id" : 1 ,
  "b" : { "b_name" : "B name" ,
          "b_description" : "the description"}
}
```

The alternative for JDO would be as follows (for JPA just swap @PersistenceCapable for @Entity)

```
@PersistenceCapable
public class A
{
    @Embedded
    @Extension(vendorName="datanucleus", key="nested", value="false")
    B b;

    ...
}
```

and this will use *flat embedding*, looking something like this

```
{ "name" : "A Name" ,
  "id" : 1 ,
  "b_name" : "B name" ,
  "b_description" : "the description"
}
```

Mapping : Embedded Collection elements

When you have a field in a class that is of a Collection type you sometimes want to store it with the owning object. In this case you can use JDO / JPA embedding of the field. So if we have

```
@PersistenceCapable
public class A
{
    @Element(embedded="true")
    Collection* bs;

    ...
}
```

and would look something like this

```
{ "name" : "A Name" ,
  "id" : 1 ,
  "bs" :
      [
        { "name" : "B Name 1" ,
          "description" : "desc 1"} ,
        { "name" : "B Name 2" ,
          "description" : "desc 2"} ,
        { "name" : "B Name 3" ,
          "description" : "desc 3"}
      ]
}
```

References

Below are some references using this support

Sasa Jovancic - Use JPA with MongoDb and Datanucleus

Neo4j Datastores

DataNucleus supports persisting/retrieving objects to/from **embedded** Neo4j graph datastores (using the datanucleus-neo4j plugin, which utilises the Neo4j Java driver). Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=neo4j:{db_location}
```

For example

```
datanucleus.ConnectionURL=neo4j:myNeo4jDB
```

You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus Neo4j persistence are `datanucleus-core`, `datanucleus-api-jdo` /`datanucleus-api-jpa`, `datanucleus-neo4j` and `neo4j`.

Note that this is for embedded Neo4j. This is because at the time of writing there is no binary protocol for connecting Java clients to the server with Neo4j. When that is available we would hope to support it.

There are tutorials available for use of DataNucleus with Neo4j for JDO and for JPA

Things to bear in mind with Neo4j usage :-

- Creation of a PMF/EMF will create a *GraphDatabaseService* and this is shared by all PM/EM instances. Since this is for an embedded graph datastore then this is the only logical way to provide this. Should this plugin be updated to connect to a Neo4J server then this will change.
- Querying can be performed using JDOQL or JPQL. Some components of a filter are handled in the datastore, and the remainder in-memory. Currently any expression of a field (in the same 'table'), or a literal are handled in-datastore, as are the operators &&, ||, >, >=, <, <=, ==, and !=. Also the majority of ordering and result clauses are evaluatable in the datastore, as well as query result range restrictions.
- When an object is persisted it becomes a Node in Neo4j. You define the names of the properties of that node by specifying the "column" name using JDO/JPA metadata
- Any 1-1, 1-N, M-N, N-1 relation is persisted as a Relationship object in Neo4j and any positioning of elements in a List or array is preserved via a property on the Relationship.
- If you wanted to specify some neo4j.properties file for use of your embedded database then specify the persistence property **datanucleus.ConnectionPropertiesFile** set to the filename.
- This plugin is in prototype stage so would welcome feedback and, better still, some contributions to fully exploit the power of Neo4j. Please contact us.

Persistence Implementation

Let's take some example classes, and then describe how these are persisted in Neo4j.

```
public class Store
{
    @Persistent(primaryKey="true", valueStrategy="identity")
    long id;

    Inventory inventory;

    ...
}

public class Inventory
{
    @Persistent(primaryKey="true", valueStrategy="identity")
    long id;

    Set<Product> products;

    ...
}

public class Product
{
    @Persistent(primaryKey="true", valueStrategy="identity")
    long id;

    String name;

    double value;

    ...
}
```

When we persist a Store object, which has an Inventory, which has three Product objects, then we get the following

- **Node** for the *Store*, with the "id" is represented as the node id
- **Node** for the *Inventory*, with the "id" is represented as the node id
- **Relationship** between the *Store* Node and the *Inventory* Node, with the relationship type as "SINGLE_VALUED", and with the property *DN_FIELD_NAME* as "inventory"
- **Node** for *Product* #1, with properties for "name" and "value" as well as the "id" represented as the node id
- **Node** for *Product* #2, with properties for "name" and "value" as well as the "id" represented as the node id

- **Node** for *Product* #3, with properties for "name" and "value" as well as the "id" represented as the node id
- **Relationship** between the *Inventory* Node and the *Product* #1 Node, with the relationship type "MULTI_VALUED" and the property *DN_FIELD_NAME* as "products"
- **Relationship** between the *Inventory* Node and the *Product* #2 Node, with the relationship type "MULTI_VALUED" and the property *DN_FIELD_NAME* as "products"
- **Relationship** between the *Inventory* Node and the *Product* #3 Node, with the relationship type "MULTI_VALUED" and the property *DN_FIELD_NAME* as "products"
- **Index** in "DN_TYPES" for the *Store* Node with "class" as "mydomain.Store"
- **Index** in "DN_TYPES" for the *Inventory* Node with "class" as "mydomain.Inventory"
- **Index** in "DN_TYPES" for the *Product* Node with "class" as "mydomain.Product"

Note that, to be able to handle polymorphism more easily, if we also have a class *Book* that extends *Product* then when we persist an object of this type we will have two entries in "DN_TYPES" for this Node, one with "class" as "mydomain.Book" and one with "class" as "mydomain.Product" so we can interrogate the Index to find the real inheritance level of this Node.

Query Implementation

In terms of querying, a JDOQL/JPQL query is converted into a generic query compilation, and then this is attempted to be converted into a Neo4j "Cypher" query. Not all syntaxis are convertable currently and the query falls back to in-memory evauation in that case.

LDAP Datastores

DataNucleus supports persisting/retrieving objects to/from LDAP datastores datanucleus-ldap plugin). If you wish to help out development of this plugin either by contributing or by sponsoring particular functionality please contact us.

Datastore Connection

The following persistence properties will connect to an LDAP running on your local machine

```
datanucleus.ConnectionDriverName=com.sun.jndi.ldap.LdapCtxFactory
datanucleus.ConnectionURL=ldap://localhost:10389
datanucleus.ConnectionUserName=uid=admin,ou=system
datanucleus.ConnectionPassword=secret
```

You create your PersistenceManagerFactory or EntityManagerFactory with these properties. Thereafter you have the full power of the JDO or JPA APIs at your disposal, for your LDAP datastore.

Queries

DataNucleus allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using SQL-like syntax

Queries are evaluated in-memory.

Mapping : LDAP Datastore Mapping

When persisting a Java object to an LDAP datastore clearly the user would like some control over where and how in the LDAP DIT (directory information tree) we are persisting the object. In general Java objects are mapped to LDAP entries and fields of the Java objects are mapped to attributes of the LDAP entries.

The following Java types are supported and stored as single-valued attribute to the LDAP entry:

- String, primitives (like int and double), wrappers of primitives (like java.util.Long), java.util.BigDecimal, java.util.BigInteger, java.util.UUID
- boolean and java.lang.Boolean are converted to RFC 4517 "boolean" syntax (TRUE or FALSE)
- java.util.Date and java.util.Calendar are converted to RFC 4517 "generalized time" syntax

Arrays, Collections, Sets and Lists of these data types are stored as multi-valued attributes. Please note that when using Arrays and Lists no order could be guaranteed and no duplicate values are allowed!

Mapping : Relationships

By default persistable objects are stored as separate LDAP entries. There are some options how to persist relationship references between persistable objects:

- DN matching
- Attribute matching
- LDAP hierarchies (DEPRECATED)

It is also possible to store persistable objects embedded. Note that there is inbuilt logic for deciding which of these mapping strategies to use for a relationship. You can explicitly set this with the metadata extension for the field/property *mapping-strategy* and it can be set to **dn** or **attribute**.

Examples

Here's an example using JDO XML MetaData:

```
<jdo>
    <package name="org.datanucleus.samples.models.company">
        <class name="Group" table="ou=Groups,dc=example,dc=com"
schema="top,groupOfNames" detachable="true">
            <field name="name" column="cn" primary-key="true" />
            <field name="users" column="member" />
        </class>

        <class name="Person" table="ou=Users,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson" detachable="true">
            <field name="personNum" column="cn" primary-key="true" />
            <field name="firstName" column="givenMame" />
            <field name="lastName" column="sn" />
        </class>
    </package>
</jdo>
```

For the class as a whole we use the **table** attribute to set the *distinguished name* of the container under which to store objects of a type. So, for example, we are mapping all objects of class Group as subordinates to "ou=Groups,dc=example,dc=com". You can also use the extension "dn" to specify the same thing.

For the class as a whole we use the **schema** attribute to define the object classes of the LDAP entry. So, for example, all objects of type Person are mapped to the common "top,person,organizationalPerson,inetOrgPerson" object classes in LDAP. You can also use the extension "objectClass" to specify the same thing.

For each field we use the **column** attribute to define the *LDAP attribute* that we are mapping this field to. So, for example, we map the Group "name" to "cn" in our LDAP. You can also use the extension "attribute" to specify the same thing.

Some resulting LDAP entries would look like this:

```
dn: cn=Sales,ou=Groups,dc=example,dc=com
objectClass: top
objectClass: groupOfNames
cn: Sales
member: cn=1,ou=Users,dc=example,dc=com

dn: cn=1,ou=Users,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: 1
givenName: Bugs
sn: Bunny
```

Here's the same example using JDO Annotations:

```
@PersistenceCapable(table="ou=Groups,dc=example,dc=com", schema="top,groupOfNames")
public class Group
{
    @PrimaryKey
    @Column(name = "cn")
    String name;

    @Column(name = "member")
    protected Set<Person> users = new HashSet<Person>();
}

@PersistenceCapable(table="ou=Users,dc=example,dc=com", schema
="top,person,organizationalPerson,inetOrgPerson")
public class Person
{
    @PrimaryKey
    @Column(name = "cn")
    private long personNum;

    @Column(name = "givenName")
    private String firstName;

    @Column(name = "sn")
    private String lastName;
}
```

Here's the same example using JPA Annotations:

```
@Entity
@Table(name="ou=Groups,dc=example,dc=com", schema="top,groupOfNames")
public class Group
{
    @Id
    @Extension(key="attribute", value="cn")
    String name;

    @OneToMany
    @Extension(key="attribute", value="member")
    protected Set users = new HashSet();
}

@Entity
@Table(name="ou=Groups,dc=example,dc=com", schema
="top,person,organizationalPerson,inetOrgPerson")
public class Person
{
    @Id
    @Extension(key="attribute", value="roomNumber")
    private long personNum;

    @Extension(key="attribute", value="cn")
    private String firstName;

    @Extension(key="attribute", value="sn")
    private String lastName;
}
```

Known Limitations

The following are known limitations of the current implementation

- Datastore Identity is not currently supported
- Optimistic checking of versions is not supported
- Identity generators that operate using the datastore are not supported
- Cannot map inherited classes to the same LDAP type

LDAP : Relationship Mapping by DN

A common way to model relationships between LDAP entries is to put the LDAP distinguished name of the referenced LDAP entry to an attribute of the referencing LDAP entry. For example entries with object class groupOfNames use the attribute *member* which contains distinguished names of the group members.

We just describe 1-N relationship mapping here and distinguish between unidirectional and bidirectional relationships. The metadata for 1-1, N-1 and M-N relationship mapping looks identical, the only difference is whether single-valued or multi-valued attributes are used in LDAP to store the relationships.

- Unidirectional
- Bidirectional

Mapping by DN : 1-N Unidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com                                       public class Department {
|                                                           String name;
|-- ou=Departments                                          Set<Employee> employees;
|   |-- cn=Sales                                        }
|   |-- cn=Engineering
|   |-- ...                                             public class Employee {
|                                                           String firstName;
|-- ou=Employees                                            String lastName;
|   |-- cn=Bugs Bunny                                       String fullName;
|   |-- cn=Daffy Duck                                   }
|   |-- cn=Speedy Gonzales
|   |-- ...
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. The **Employee** knows nothing about the **Department** it belongs to.

There are 2 ways that we can persist this relationship in LDAP because the DN reference could be stored at the one or at the other LDAP entry.

Owner Object Side

The obvious way is to store the reference at the owner object side, in our case at the department entry. This is possible since LDAP allows multi-valued attributes. The example department entry looks like this:

```
dn: cn=Sales,ou=Departments,dc=example,dc=com
objectClass: top
objectClass: groupOfNames
cn: Sales
member: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
member: cn=Daffy Duck,ou=Employees,dc=example,dc=com
```

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,groupOfNames">
            <field name="name" primary-key="true" column="cn" />
            <field name="employees" column="member">
                <extension vendor-name="datanucleus" key="empty-value"
value="uid=admin,ou=system"/>
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
        </class>
    </package>
</jdo>
```

So we define that the attribute *member* should be used to persist the relationship of field *employees*.

Note: We use the extension *empty-value* here. The groupOfNames object class defines the member attribute as mandatory attribute. In case where you remove all the employees from a department would delete all member attributes which isn't allowed. In that case DataNucleus adds this empty value to the member attribute. This value is also filtered when DataNucleus reads the object from LDAP.

Non-Owner Object Side

Another possible way is to store the reference at the non-owner object side, in our case at the employee entry. The example employee entry looks like this:

```
dn: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: cn=Sales,ou=Departments,dc=example,dc=com
```

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,groupOfNames">
            <field name="name" primary-key="true" column="cn" />
            <field name="employees">
                <element column="departmentNumber" />
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
        </class>
    </package>
</jdo>
```

We need to define the relationship at the department metadata because the employee doesn't know about the department it belongs to. With the *<element>* tag we specify that the relationship should be persisted at the other side, the *column* attribute defines the LDAP attribute to use. In this case the relationship is persisted in the *departmentNumber* attribute at the employee entry.

Mapping by DN : 1-N Bidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|
|-- ou=Departments
|   |-- cn=Sales
|   |-- cn=Engineering
|   |-- ...
|
|-- ou=Employees
|   |-- cn=Bugs Bunny
|   |-- cn=Daffy Duck
|   |-- cn=Speedy Gonzales
|   |-- ...
```

```
public class Department {
    String name;
    Set<Employee> employees;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Department department;
}
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. Now each **Employee** has a reference to its **Department**.

It is possible to persist this relationship on both sides.

```
dn: cn=Sales,ou=Departments,dc=example,dc=com
objectClass: top
objectClass: groupOfNames
cn: Sales
member: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
member: cn=Daffy Duck,ou=Employees,dc=example,dc=com
```

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,groupOfNames">
            <field name="name" primary-key="true" column="cn" />
            <field name="employees" column="member">
                <extension vendor-name="datanucleus" key="empty-value"
value="uid=admin,ou=system"/>
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="department" mapped-by="employees" />
        </class>
    </package>
</jdo>
```

In this case we store the relation at the department entry side in a multi-valued attribute *member*. Now the employee metadata contains a department field that is *mapped-by* the employees field of department.

Note: We use the extension *empty-value* here. The groupOfNames object class defines the member attribute as mandatory attribute. In case where you remove all the employees from a department would delete all member attributes which isn't allowed. In that case DataNucleus adds this empty value to the member attribute. This value is also filtered when DataNucleus reads the object from LDAP.

LDAP : Relationship Mapping by Attribute

Another way to model relationships between LDAP entries is to use attribute matching. This means two entries have the same attribute values. An example of this type of relationship is used by posixGroup and posixAccount object classes were posixGroup.memberUid points to posicAccount.uid.

We just describe 1-N relationship mapping here and distinguish between unidirectional and bidirectional relationships. The metadata for 1-1, N-1 and M-N relationship mapping looks identical, the only difference is whether single-valued or multi-valued attributes are used in LDAP to store

the relationships.

- Unidirectional
- Bidirectional

Mapping by Attribute: 1-N Unidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com                                       public class Department {
|                                                           String name;
|-- ou=Departments                                          Set<Employee> employees;
|   |-- ou=Sales                                        }
|   |-- ou=Engineering
|   |-- ...                                             public class Employee {
|                                                           String firstName;
|-- ou=Employees                                            String lastName;
|   |-- uid=bbunny                                          String fullName;
|   |-- uid=dduck                                           String uid;
|   |-- uid=sgonzales                                   }
|   |-- ...
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. The **Employee** knows nothing about the **Department** it belongs to.

There are 2 ways that we can persist this relationship in LDAP because the reference could be stored at the one or at the other LDAP entry.

Owner Object Side

One way is to store the reference at the owner object side, in our case at the department entry. This is possible since LDAP allows multi-valued attributes. The example department entry looks like this:

```
dn: ou=Sales,ou=Departments,dc=example,dc=com
objectClass: top
objectClass: organizationalUnit
objectClass: extensibleObject
ou: Sales
memberUid: bbunny
memberUid: dduck
```

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,organizationalUnit,extensibleObject">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees" column="memberUid">
                <join column="uid" />
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">

            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="uid" column="uid" />
        </class>
    </package>
</jdo>
```

So we define that the attribute *memberUid* at the department entry should be used to persist the relationship of field *employees*

The important thing here is the *<join>* tag and its *column*. Firstly it signals DataNucleus to use attribute mapping. Secondly it specifies the attribute at the other side that should be used for relationship mapping. In our case, when we establish a relationship between a **Department** and an **Employee**, the *uid* value of the employee entry is stored in the *memberUid* attribute of the department entry.

Non-Owner Object Side

Another possible way is to store the reference at the non-owner object side, in our case at the employee entry. The example employee entry looks like this:

```
dn: uid=bbunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: bbunny
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: Sales
```

Our JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees">
                <element column="departmentNumber" />
                <join column="ou" />
            </field>
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="uid" column="uid" />
        </class>
    </package>
</jdo>
```

We need to define the relationship at the department metadata because the employee doesn't know about the department it belongs to.

With the *<element>* tag we specify that the relationship should be persisted at the other side and the *column* attribute defines the LDAP attribute to use. In this case the relationship is persisted in the *departmentNumber* attribute at the employee entry.

The important thing here is the *<join>* tag and its *column*. As before it signals DataNucleus to use attribute mapping. Now, as the relation is persisted at the <u>other</u> side, it specifies the attribute at <u>this</u> side that should be used for relationship mapping. In our case, when we establish a relationship between a **Department** and an **Employee**, the *ou* value of the department entry is stored in the *departmentNumber* attribute of the employee entry.

Mapping by Attribute : 1-N Bidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com                                       public class Department {
|                                                           String name;
|-- ou=Departments                                          Set<Employee> employees;
|   |-- ou=Sales                                        }
|   |-- ou=Engineering
|   |-- ...                                             public class Employee {
|                                                           String firstName;
|-- ou=Employees                                            String lastName;
|   |-- uid=bbunny                                          String fullName;
|   |-- uid=dduck                                           String uid;
|   |-- uid=sgonzales                                       Department department;
|   |-- ...                                             }
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, **Department** and **Employee**. The **Department** class contains a Collection of type **Employee**. Now each **Employee** has a reference to its **Department**.

It is possible to persist this relationship on both sides.

```
dn: uid=bbunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: bbunny
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: Sales
```

```
<jdo>
    <package name="com.example">
        <class name="Department" table="ou=Departments,dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees" mapped-by="department" />
        </class>
        <class name="Employee" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="uid" column="uid" />
            <field name="department" column="departmentNumber">
                <join column="ou" />
            </field>
        </class>
    </package>
</jdo>
```

In this case we store the relation at the employee entry side in a single-valued attribute *departmentNumber*. With the *<join>* tag and its *column* we specify that the *ou* value of the department entry should be used as join value. Also note that *employee* field of **Department** is *mapped-by* the *department* field of the **Employee**.

LDAP : Relationship Mapping by Hierarchy (DEPRECATED)

As LDAP is a hierarchical data store it is possible to model relationships between LDAP entries using hierarchies. For example organisational structures like departments and their employees are often modeled hierarchical in LDAP. It is possible to map 1-1 and N-1/1-N relationships using LDAP hierarchies.

The main challenge with hierarchical mapping is that the distinguished name (DN) of children depends on the DN of their parent. Therefore each child class needs a reference to the parent class. The parent class metadata defines a (fixed) LDAP DN that is used as container for all objects of the parent type. The child class metadata contains a dynamic part in its DN definition. This dynamic part contains the name of the field holding the reference to the parent object, the name is surrounded by curly braces. This dynamic DN is the indicator for DataNucleus to use hierarchical mapping. The reference field itself won't be persisted as attribute because it is used as dynamic parameter. If you query for child objects DataNucleus starts a larger LDAP search to find the objects (the container DN of the parent class as search base and subtree scope).

Child objects are automatically dependent. If you delete the parent object all child objects are automatically deleted. If you null out the child object reference in the parent object or if you remove the child object from the parents collection, the child object is automatically deleted.

Mapping by Hierarchy : N-1 Unidirectional (DEPRECATED)

This kind of mapping could be used if your LDAP tree has a huge number of child objects and you only work with these child objects.

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|
|-- ou=Sales
|   |-- cn=Bugs Bunny
|   |-- cn=Daffy Duck
|   |-- ...
|
|-- ou=Engineering
|   |-- cn=Speedy Gonzales
|   |-- ...
|
|-- ...
```

```
public class Department {
    String name;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Department department;
}
```

In the LDAP tree we have departments (Sales and Engineering) and each department holds some associated employees. In our Java classes each **Employee** object knows its **Department** but not vice-versa.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
        </class>

        <class name="Employee" table="{department}"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="department"/>
        </class>
    </package>
</jdo>
```

The **Department** objects are persisted directly under *dc=example,dc=com*. The **Employee** class has a dynamic DN definition *{department}*. So the DN of the Department instance is used as container for Employee objects.

Mapping by Hierarchy : N-1 (1-N) Bidirectional (DEPRECATED)

If you need a reference from the parent object to the child objects you need to define a bidirectional relationship.

The example LDAP tree and Java classes looks like this:

```
dc=example,dc=com                                       public class Department {
|                                                           String name;
|-- ou=Sales                                                Set<Employee> employees;
|   |-- cn=Bugs Bunny                                   }
|   |-- cn=Daffy Duck
|   |-- ...                                             public class Employee {
|                                                           String firstName;
|-- ou=Engineering                                          String lastName;
|   |-- cn=Speedy Gonzales                                  String fullName;
|   |-- ...                                                 Department department;
|                                                       }
|-- ...
```

Now the **Department** class has a Collection containing references to its *Employee*s.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Department" table="dc=example,dc=com"
schema="top,organizationalUnit">
            <field name="name" primary-key="true" column="ou" />
            <field name="employees" mapped-by="department"/>
        </class>

        <class name="Employee" table="{department}"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="department"/>
        </class>
    </package>
</jdo>
```

We added a new *employees* field to the Department class that is *mapped-by* the department field of the Employee class.

Please note: When loading the parent object all child object are loaded immediately. For a large number of child entries this may lead to performance and/or memory problems.

Mapping by Hierarchy : 1-1 Unidirectional (DEPRECATED)

1-1 unidirectional mapping is very similar to N-1 unidirectional mapping.

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com                                       public class Person {
|                                                           String firstName;
|-- ou=People                                               String lastName;
|   |-- cn=Bugs Bunny                                       String fullName;
|   |   |-- uid=bbunny                                  }
|   |
|   |-- cn=Daffy Duck                                   public class Account {
|   |   |-- uid=dduck                                       String uid;
|   |                                                       String password;
|   |-- ...                                                 Person person;
                                                        }
```

In the LDAP tree we have persons and each person has one account. Each **Account** object knows to which **Person** it belongs to, but not vice-versa.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Person" table="ou=People,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
        </class>

        <class name="Account" table="{person}"
schema="top,account,simpleSecurityObject">
            <field name="uid" primary-key="true column="uid" />
            <field name="password" column="userPasword" />
            <field name="person" />
        </class>
    </package>
</jdo>
```

The **Person** objects are persisted directly under *ou=People,dc=example,dc=com*. The **Account** class has a dynamic DN definition *{person}*. So the DN of the Person instance is used as container for the Account object.

Mapping by Hierarchy : 1-1 Bidirectional (DEPRECATED)

If you need a reference from the parent class to the child class you need to define a bidirectional

relationship.

The example LDAP tree and Java classes looks like this:

```
dc=example,dc=com                                       public class Person {
|                                                           String firstName;
|-- ou=People                                               String lastName;
|   |                                                       String fullName;
|   |-- cn=Bugs Bunny                                       Account account;
|   |   |-- uid=bbunny                                  }
|   |
|   |-- cn=Daffy Duck                                   public class Account {
|   |   |-- uid=dduck                                       String uid;
|   |                                                       String password;
|   |-- ...                                                 Person person;
                                                        }
```

Now the **Person** class has a reference to its **Account**.

The JDO metadata looks like this:

```
<jdo>
    <package name="com.example">
        <class name="Person" table="ou=People,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="account" mapped-by="person" />
        </class>

        <class name="Account" table="{person}"
schema="top,account,simpleSecurityObject">
            <field name="uid" primary-key="true column="uid" />
            <field name="password" column="userPasword" />
            <field name="person" />
        </class>
    </package>
</jdo>
```

We added a new *account* field to the Person class that is *mapped-by* the person field of the Account class.

LDAP : Embedded Objects

With JDO it is possible to persist field(s) as embedded. This may be useful for LDAP datastores where often many attributes are stored within one entry however logically they describe different objects.

Let's assume we have the following entry in our directory:

```
dn: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
postalCode: 3578
l: Hollywood
street: Sunset Boulevard
uid: bbunny
userPassword: secret
```

This entry contains multiple type of information: a person, its address and its account data. So we will create the following Java classes:

```
public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Address address;
    Account account;
}

public class Address {
    int zip;
    String city
    String street;
}

public class Account {
    String id;
    String password;
}
```

The JDO metadata to map these objects to one LDAP entry would look like this:

```
<jdo>
    <package name="com.example">
        <class name="Person" table="ou=Employees,dc=example,dc=com"
schema="top,person,organizationalPerson,inetOrgPerson">
            <field name="fullName" primary-key="true" column="cn" />
            <field name="firstName" column="givenName" />
            <field name="lastName" column="sn" />
            <field name="account">
                <embedded null-indicator-column="uid">
                    <field name="id" column="uid" />
                    <field name="password" column="userPassword" />
                </embedded>
            </field>
            <field name="address">
                <embedded null-indicator-column="l">
                    <field name="zip" column="postalCode" />
                    <field name="city" column="l" />
                    <field name="street" column="street" />
                </embedded>
            </field>
        </class>
        <class name="Account" embedded-only="true">
            <field name="uid" />
            <field name="password" />
        </class>
        <class name="Address" embedded-only="true">
            <field name="zip" />
            <field name="city" />
            <field name="street" />
        </class>
    </package>
</jdo>
```

NeoDatis Datastores

NeoDatis is an object-oriented database for Java and .Net. It is simple and fast and supports various query mechanisms.

DataNucleus supports persisting/retrieving objects to Neodatis datastores (using the datanucleus-neodatis plugin).

The jars required to use DataNucleus NeoDatis persistence are `datanucleus-core`, `datanucleus-api-jdo`/`datanucleus-api-jpa`, `datanucleus-neodatis` and `neodatis`.

Datastore Connection

DataNucleus supports 2 modes of operation of *neodatis* - file-based, and client-server based. In order to do so and to fit in with the JDO/JPA APIs we have defined the following means of connection.

The following persistence properties will connect to a **file-based** Neodatis running on your local machine

```
datanucleus.ConnectionURL=neodatis:file:neodatisdb.odb
```

Replacing `neodatisdb.odb` by your filename for the datastore, and can be absolute OR relative.

The following persistence properties will connect to **embedded-server-based** NeoDatis running with a local file

```
datanucleus.ConnectionURL=neodatis:server:{my_neodatis_file}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

The filename {my_neodatis_file} can be absolute OR relative.

The following persistence properties will connect as a client to a **TCP/IP NeoDatis Server**

```
datanucleus.ConnectionURL=neodatis:{neodatis_host}:{neodatis_port}/{identifier}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

Neodatis doesn't itself use such URLs so it was necessary to define this DataNucleus-specific way of addressing Neodatis.

So you create your PersistenceManagerFactory or EntityManagerFactory with these properties. Thereafter you have the full power of the JDO or JPA APIs at your disposal, for your NeoDatis datastore.

Queries

AccessPlatform allows you to query the objects in the datastore using the following

- JDOQL - language based around the objects that are persisted and using Java-type syntax
- JPQL - language based around the objects that are persisted and using SQL-like syntax
- Native - NeoDatis' own type-safe query language
- Criteria - NeoDatis' own Criteria query language

Queries : NeoDatis Native Queries

If you choose to use NeoDatis Native Queries then these are not portable to any other datastore. Use JDOQL/JPQL for portability

NeoDatis provides its own "native" query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

```
// Find all employees older than 31
Query q = pm.newQuery("Native", new NativeQuery()
    {
        public boolean match(Object e)
        {
            if (!(e instanceof Employee))
            {
                return false;
            }
            return ((Employee)e).getAge() >= 32;
        }
        public Class getObjectType()
        {
            return Employee.class;
        }
    });

List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the NeoDatis "NativeQuery".

Queries : NeoDatis Criteria Queries

If you choose to use NeoDatis Criteria Queries then these are not portable to any other datastore. Use JDOQL/JPQL for portability

NeoDatis provides its own "criteria" query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

```
// Find all employees older than 31
Query q = pm.newQuery("Criteria", new CriteriaQuery(Employee.class, Where.ge("age",
32)));

List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the NeoDatis "CriteriaQuery".

Known Limitations

The following are known limitations of the current implementation

- NeoDatis doesn’t have the concept of an "unloaded" field and so when you request an object from the datastore it comes with its graph of objects. Consequently there is no "lazy loading" and the consequent impact that can have on memory utilisation.