IBM®

developerWorks.

Migrate from EJB 2 container-managed persistence to pureQuery for IBM Master Data Management Server, Part 2: Proving the pureQuery technology

Skill Level: Intermediate

Wei Zheng (wzheng@ca.ibm.com)
Product Architect - MDM Server
IBM

Paul van Run (pvanrun@ca.ibm.com)
Senior Technical Staff Member - MDM Architect
IBM

Tim Thorpe (tthorpe@ca.ibm.com)
Product Architect - Master Data Management
IBM

Kathy Zeidenstein (krzeide@us.ibm.com)
Senior Software Engineer, Data Studio Enablement
IBM

10 Jul 2008

If you've been curious about the new release of WebSphere® Customer Center (now named IBM® InfoSphere™ Master Data Management Server), then this series is for you! This series describes how and why pureQuery technology was used in the new release, the implementation and migration to pureQuery, and the results of performance and capability testing to validate this critical decision. Part 2 focuses on the productivity and performance measurements in making the decision to use pureQuery and also provides some hints and tips for working with pureQuery.

Introduction

© Copyright IBM Corporation 1994, 2007. All rights reserved.

In 2007 the IBM Master Data Management team was planning for a major new release of their WebSphere® Customer Center product which was to be renamed IBM InfoSphere Master Data Management Server. One critical architectural decision we had to make was what to do about the existing persistence mechanism, which was based on a mixture of EJB 2 CMP entity beans and native JDBC calls, and was becoming outdated.

This two-part series describes the migration of IBM InfoSphere MDM Server, a fairly typical J2EE application, from EJB 2 CMP to pureQuery. In the first article, we described why we wanted to migrate the data persistence mechanism. This article describes how we did it. The IBM MDM development team performed a proof of concept (POC) to validate that we could successfully migrate the MDM Server product code to use pureQuery. By reviewing this article, you can learn from our experiences if faced with the same task for another project.

Important: This project was an internal project to migrate our product code from EJB 2 CMP to pureQuery. It does NOT in any way reflect the steps required for an IBM customer to migrate over from WCC to MDM Server. That information can be found in the MDM Server product documentation (see Resources).

Proof of concept overview and methodology

As Part 1 describes, we knew we had a decision to make regarding updating our persistence technology that was being used in WebSphere Customer Center (WCC). We investigated some different persistence technologies and, as a result of that investigation, decided that Data Studio pureQuery was a strong candidate for the next version of our product –- IBM InfoSphere Master Data Management Server.

We developed a plan for a proof of concept with the following three goals in mind:

- Confirm whether pureQuery could provide the capability we needed for our data access.
- Understand the development and delivery environment and process, and estimate the amount of work it would take to migrate to Data Studio pureQuery.
- Ensure there are no backward compatibility issues, and that performance will be as good or better than the prior implementation.

For our proof of concept, we decided to cover the following areas:

- Select a subset of entities used in WCC. We chose to convert the following person- related entities to pureQuery entities:
  - Contact entity

© Copyright IBM Corporation 1994, 2007. All rights reserved.

    - Person entity
    - PersonName entity
    - Identifier entity
    - LocationGroup entity
    - AddressGroup entity
    - Address entity
- Run a performance benchmark on person-related transactions with the above entities.
- Mix converted entities and non-converted entities in one transaction for co-existence and global transaction support testing.

Development and delivery environment

Here is the development environment we used for pureQuery in this proof of concept:

- Platform:
    - Windows™
    - pureQuery runtime run on J2SE1.5 (it doesn’t require any JEE5 library)
    - WebSphere Application Server (WAS) 6.1, available now on J2SE 1.5
- Tools:
    - RAD7.0 is built on top of Eclipse3.2 with J2SE1.5 and WAS6.1 development support. Data Studio Developer is an Eclipse-based tool, which we installed with our RAD 7.0 to give us access to the full capability of pureQuery development tooling.

Here is our delivery platform for pureQuery in the proof of concept:

- AIX
- WebSphere V6.1

Migrating the CMP beans and JDBC to pureQuery

© Copyright IBM Corporation 1994, 2007. All rights reserved.

As you may recall from Part 1, we used both CMP entity beans (for CRUD operations) and JDBC (for our complex queries). In this section, we describe how we migrated both to pureQuery. We migrated our CMP entity beans to use the pureQuery annotated method style interface and migrated our JDBC calls to use pureQuery inline style. For more information on the different styles in pureQuery, see Resources.

Migrating CMP beans

Here are the steps we used to convert CMP entity beans to pureQuery annotated entities:

1. Used a meet-in-the-middle approach to create the mapping annotations for our existing EObjs to tables in the database. See Resources for detailed information on how to do meet-in-the-middle approach using Data Studio Developer tools. Listing 1 shows an example of the mapped address entity.
   Listing 1. Annotated entity object

```
@Table(name="ADDRESS")
public class EObjAddress extends EObjCommon {
    @Column(name = "address_id")
    public java.lang.Long addressIdPK;

    @Column(name = "addr_line_one")
    public java.lang.String addrLineOne;

        ...

    @Column(name = "residence_tp_cd")
    public java.lang.Long residenceTpCd;
}
```

2. After the entity object EObj is annotated with object-to-relational mapping information, we generated the pureQuery annotated method style SQL interface with basic CRUD SQL. One of the advantages of using Data Studio pureQuery is that it allows users to change the interface after it is generated. Users can regenerate if their mapping changes; however, the manual change will be overridden. Listing 2 shows our generated interface example; we appended the update SQL with an optimistic concurrency control WHERE clause `and LAST_UPDATE_DT = :oldLastUpdateDt`.
   Listing 2. Generated annotated method style SQL interface

```
public interface EObjAddressData {

  // Get ADDRESS by parameters
  @Select(sql="select * from ADDRESS where ADDRESS_ID = ? ")
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
  Iterator<EObjAddress> getAddress(long address_id);

  // Create ADDRESS by Address Object
  @Update(sql="insert into ADDRESS values( :addressIdPK, :countryTpCd,
        ……., :postalBarCode)")
  int createAddress(EObjAddress a);

  // Update one ADDRESS by Address object
  @Update(sql="update ADDRESS set ADDRESS_ID = :addressIdPK, COUNTRY_TP_CD =
      ………… where ADDRESS_ID = :addressIdPK and LAST_UPDATE_DT = :oldLastUpdateDt")
  int updateAddress(EObjAddress a);

  // Delete ADDRESS by parameters
  @Update(sql="delete from ADDRESS where ADDRESS_ID = ? ")
  int deleteAddress(long address_id);
}
```

3. Next we built the project to trigger the creation of the implementation class for the annotated method style SQL interface. For our address interface, a class called `EObjAddressDataImpl` is created.
4. After the CMP entity beans were replaced, we had to change the related calls to the CMP entity beans in the business component layer as well. **Note:** To avoid dealing with the implementation class directly, we used a factory class to isolate the creation of the implementation class, for example, `EObjAddressDataImpl`
   - For an ADD operation, Listing 3 shows sample code.
     Listing 3. Create entity pureQuery call

```
connection = DataManager.getInstance().getQueryConnection();
EObjAddressData addressData = (EObjAddressData)
   DataAccessFactory.getQuery(EObjAddressData.class, connection);
addressData.createEObjAddress(theTCRMAddressBObj.getEObjAddress());
```

   - For an UPDATE operation, Listing 4 shows sample code.
     Listing 4. Update entity pureQuery call

```
connection = DataManager.getInstance().getQueryConnection();
EObjAddressData addressData = (EObjAddressData)
   DataAccessFactory.getQuery(EObjAddressData.class, connection);
addressData.updateEObjAddress(theTCRMAddressBObj.getEObjAddress());
```

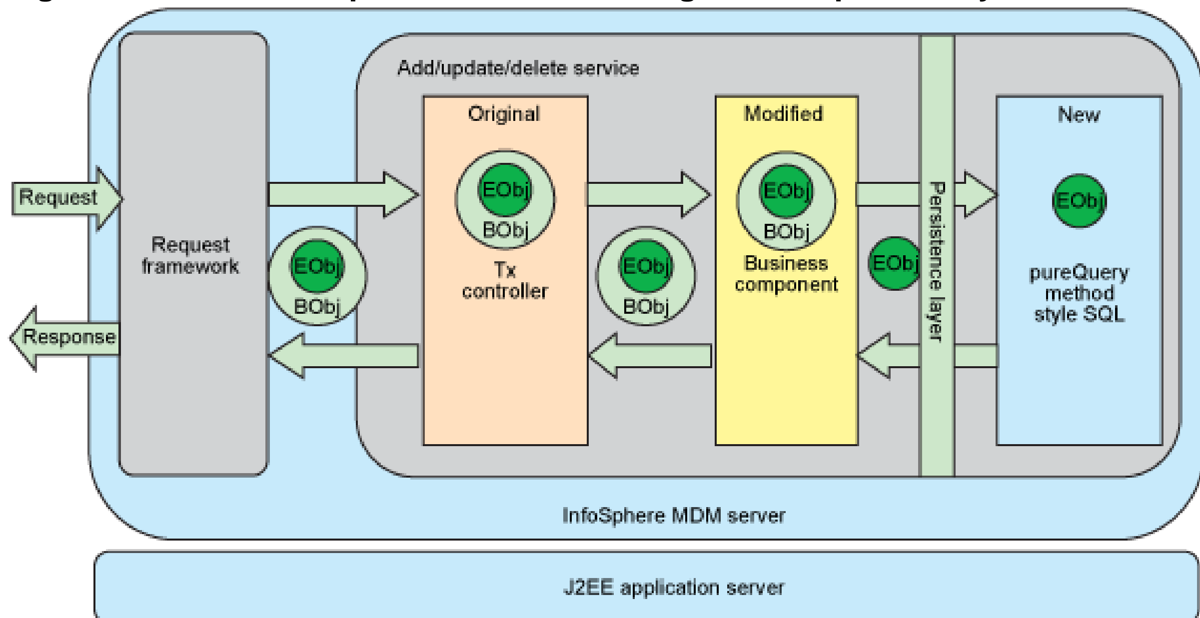
   - For a DELETE operation, Listing 5 shows sample code.
     Listing 5. Delete entity pureQuery call

© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
connection = DataManager.getInstance().getQueryConnection();
EObjAddressData addressData = (EObjAddressData)
   DataAccessFactory.getQuery(EObjAddressData.class, connection);
addressData.deleteEObjAddress(addressBObj.getEObjAddress().getAddressIdPK());
```

Figure 1 shows the new and changed components in our architecture.

Figure 1. Add/Delete/Update service after migration to pureQuery



Additional changes to our entity objects

We made other changes to handle entity lifecycle events, callback, and primary key identification:

- To have a common primary key generation facility, each entity object EObj is required to identify its primary key. The Parent entity object defined the abstract `setPrimaryKey()` and `getPrimaryKey()` methods for the child object to implement.
- Parent entity objects have common lifecycle event handling, but can be enhanced by the child entity object.
- To support optimistic concurrency control in our product, `getOldLastUpdateDate()` and `setOldLastUpdateDate()` methods are added to the parent entity object.

© Copyright IBM Corporation 1994, 2007. All rights reserved.

- A hook class is created and attached to the pureQuery Data class to call lifecycle events

Listing 6 shows the sample code to handle the additional changes.

Listing 6. Primary key and lifecycle events

```
public abstract class EObjCommon extends DWLEObjCommon {

        public abstract Object getPrimaryKey();

        public abstract void setPrimaryKey(Object value);

        public void beforeAdd() {
                handlePluggableKey();
                handleOptimisticLocking();
                handleLastUpdateUser();
                handleLastUpdateTx();
                beforeAddEx();
        }

        public void beforeUpdate() {
                handleOptimisticLocking();
                handleLastUpdateUser();
                handleLastUpdateTx();
                beforeUpdateEx();
        }

        public void afterAdd() {
                afterAddEx();
        }

        public void afterUpdate() {
                afterUpdateEx();
        }

        public Timestamp getOldLastUpdateDt() {
                return oldLastUpdateDt;
        }

        public void setOldLastUpdateDt(Timestamp oldLastUpdateDt) {
                this.oldLastUpdateDt = oldLastUpdateDt;
        }
}

public class EObjAddress extends EObjCommon {
        public void setPrimaryKey(Object aUniqueId) {
                this.setAddressIdPK((Long)aUniqueId);
        }

        public Object getPrimaryKey() {
                return this.getAddressIdPK();
        }
}

public class EObjCommonHook implements Hook   {
        public void pre(String methodName, Data data, SqlStatementType type ,Object[]
                parameters) {
                if (methodName.indexOf("create") > -1) {
                        EObjCommon anEObj = (EObjCommon) parameters[0];
                        anEObj.beforeAdd();
                } else if (methodName.indexOf("update") > -1) {
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
                        EObjCommon anEObj = (EObjCommon) parameters[0];
                        anEObj.beforeUpdate();
                }
        }


        public void post(String methodName, Data data, Object returnValue,
                SqlStatementType type, Object[] parameters) {
                if (methodName.indexOf("create") > -1) {
                        EObjCommon anEObj = (EObjCommon) parameters[0];
                        anEObj.afterAdd();
                } else if (methodName.indexOf("update") > -1) {
                        int count = ((Integer) returnValue).intValue();
                        EObjCommon anEObj = (EObjCommon) parameters[0];
                        if (count == 0){
                                throw new RuntimeException(new
                                        LastUpdateDateException(getErrorInfo(anEObj)));
                        }
                        anEObj.afterUpdate();
                }
        }
}

//attach hook to Data class
Hook hook = new EObjCommonHook();
DataFactory.getData(getDataSource(dataSourceName), hook)



}
```

Migrating direct JDBC calls

Here are the steps we used to convert direct JDBC calls to use pureQuery:

1. We modified each business object query (BObjQuery) to call pureQuery inline SQL API instead of generic SQLQuery to get a set of entity objects (EObj). There is a query framework in MDM Server where, depending on the request and the configuration settings, you may modify the WHERE clause of SELECT statements in different ways. So the “method style” query approach can't be used. The inline pureQuery style for query is more flexible than the annotated method style SQL where the SQL cannot be changed at runtime.

   The sample code in Listing 7 shows an example of an SQL modification of the WHERE clause of a query to enable record-level access control.

   We did something a little advanced here. The RowHandler shown in Listing 7 is used to convert a SQL ResultSet to an entity object. Even though we are using inline style for our queries, we used the RowHandler that was generated in annotated method style SQL data interface implementation class from the object-relational mapping annotation. That RowHandler code is capable of handling a query scenario in which the WHERE clause of a query is constructed on the fly, and the SELECT cause is known at development time. We also retrieved the SQL defined

© Copyright IBM Corporation 1994, 2007. All rights reserved.

in the annotated method style SQL data interface implementation for modification later on.

If you have the case where the SELECT clause of a query is also constructed on the fly, then you can't use the pre-generated object-relational mapping code, and you'll have to create your own customized RowHandler.

Listing 7. Use Row Handler for object-relational mapping

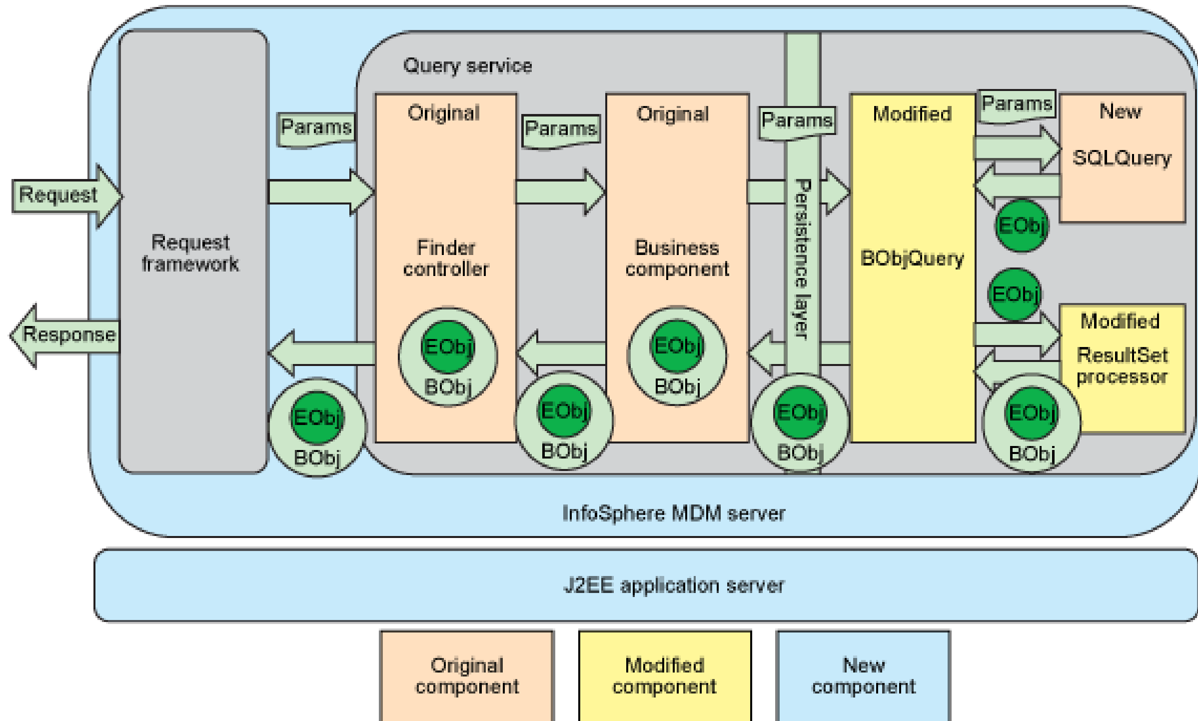
```
//get Row Handler from data interface implementation
RowHandler rowHandler = getRowHandler(dataInterfaceImpl);

//get SQL from data interface implementation
String sql = getSQLStatement(dataInterfaceImpl);

//Modify the SQL statement
 String modifiedSQL = handleAccessControl(sql);

//invoke pureQuery inline SQL API to get a set of entity objects
Iterator<Object> eObjs = getData().queryIterator(modifiedSQL,
     rowHandler, parameters);
```

2. The set of entity objects (EObj) is passed to the ResultSetProcessor to create a set of business objects (BObj). The ResultSetProcessor class is also changed to take the input of the EObjs instead of a ResultSet.

Figure 2 shows these changes.

Figure 2. Query service after migration to pureQuery

© Copyright IBM Corporation 1994, 2007. All rights reserved.



Functional testing for backward compatibility and global transaction support

As part of the proof of concept, we needed to verify that pureQuery could co-exist with existing client code using JDBC and EJB2/CMP for backward compatibility and global transaction support.

Co-existence of direct JDBC query and pureQuery

We verified the following scenarios for backward compatibility:

- pureQuery and direct JDBC query code within one transaction to query different tables.
- pureQuery and direct JDBC query code within one transaction to access the same table.

Both of the above scenarios returned the correct results.

Co-existence of EJB2/CMP and pureQuery

We verified the following scenarios for backward compatibility:

- pureQuery and CMP code within one transaction to access different tables. This is covered by mixing updates using a pureQuery entity and a

© Copyright IBM Corporation 1994, 2007. All rights reserved.

CMP entity in one transaction to different tables.

- pureQuery and CMP code within one transaction to access the same table. This is covered by mixing an ADD using a pureQuery entity and an UPDATE using CMP entity to the same table within one transaction.

Both of the above scenarios succeeded.

Global Transaction (XA) support testing

To execute the test, we set up the client to invoke InfoSphere MDM Server with two transactions that are wrapped in one Global Transaction. When the second transaction failed, we had to verify that the first transaction was rolled back through XA support.

The Global Transaction (XA) only commits the two transactions if both of them succeed; otherwise, it rolls back both transactions. EJB CMP entities have the XA support from the EJB container. pureQuery has XA support from the XA JDBC driver, and direct JDBC access can have XA support from XA JDBC driver, too. We verified the XA support in the following scenarios:

- The two transactions cover pureQuery code only.
- The two transactions cover both direct JDBC query and pureQuery code.
- The two transactions cover both EJB2/CMP and pureQuery code.

All of the above scenarios tested successfully, and the first transaction was rolled back in each.

Performance benchmarks

One important goal of this proof of concept is to prove that the performance is as good or (preferably) better after using pureQuery. Our migration decision was heavily dependent upon the results of these tests.

Important: Our performance measurements were done in a controlled laboratory environment. Your results will vary depending on overall system resources and other variables.

Environment setup and test measurement

The proof of concept application was deployed on WebSphere Application Server 6.1 running on one AIX machine, and DB2 9.1 running on another similar AIX machine as the database server. Rational® Performance Tester was used as the load generator, which was running on separate machines that were not part of the

© Copyright IBM Corporation 1994, 2007. All rights reserved.

system under test (SUT).

Measurements were collected during the steady phase of each run after a sufficient warm-up phase. Different types of MDM service transactions were used to represent insert, update and select operations. Each type of transaction was evaluated individually by running multiple concurrent simulated users without think time.

The following performance metrics were recorded:

- Response time, which is measured as milliseconds elapsed for each transaction (ms/txn)
- Throughput, which is measured as transactions finished per second (TPS)
- CPU utilization percentage on the WebSphere machine and DB2 machine

Results: Query transaction

We expected the same or slightly better performance using pureQuery over straight JDBC for query transactions. Our JDBC was already highly optimized, so we didn't expect improvement using pureQuery at execution time.

The results in Figure 3 show comparable results between JDBC and pureQuery using dynamic execution of the SQL. Response time, throughput and transactions per second differences are all under 2% and are so small as to fall within the acceptable test variation from run to run.

Figure 3. Query transaction performance details

			ms/txn	TPS	WAS CPU %	DB2 CPU %
test case No.	#OfUsers	Duration (s)	avg	avg	avg total	avg total
getParty baseline 29	4	1200	79.98	44.46	55.55	12.90
getParty pureQuery 30	4	1200	80.87	43.81	55.85	12.95
30 vs 29			1.11%	-1.47%	0.54%	0.39%

Results: Add transaction

We expected to have better performance by using pureQuery because we removed the multi-layer CMP code, and thus the CPU and memory cost to manage entities should be saved.

Our test results depicted in Figure 4 show significant performance improvements made by using pureQuery:

- Response time is 11% better.

© Copyright IBM Corporation 1994, 2007. All rights reserved.

- Throughput is 9% better.

Figure 4. Add transaction performance details

			ms/txn	TPS	WAS CPU %	DB2 CPU %
test case No.	#OfUsers	Duration (s)	avg	avg	avg total	avg total
addParty baseline 33	4	1200	107.70	32.60	66.00	8.50
addParty pureQuery 34	4	1200	95.69	35.66	60.65	9.40
34 vs 33			-11.16%	9.37%	-8.11%	10.59%

Results: Update transaction

We expected to have better performance by using pureQuery because we removed the multi-layer CMP code and thus the CPU and memory cost to manage entities should be saved. In addition, there is one less database read operation required. (The CMP update required both a read and update to the database.)

Again, our test results depicted in Figure 5 show the significant performance improvements made by using pureQuery:

- Response time is 17.85% better.
- Throughput is 27% better.

Figure 5. Update transaction performance details

			ms/txn	TPS	WAS CPU %	DB2 CPU %
test case No.	#OfUsers	Duration (s)	avg	avg	avg total	avg total
updateParty baseline 31	4	1200	423.32	8.61	56.95	12.20
updateParty pureQuery 32	4	1200	347.78	10.93	60.10	12.55
32 vs 31			-17.85%	26.97%	5.53%	2.87%

Lessons learned

Productivity using pureQuery

After our developers had become familiar with the conversion process using Data Studio Developer, each developer was converting about 6 EJB2 CMP entities per day. They were converting around 12 simple statements or 4 complicated SQL statements per day. All of these numbers include unit testing.

Avoiding regeneration of implementation classes

© Copyright IBM Corporation 1994, 2007. All rights reserved.

When a developer enables pureQuery support for a project, pureQuery determines when it needs to regenerate the implementation classes for the data interfaces. When we would synchronize our code with the repository and then perform a build, pureQuery would decide that all of our implementation classes needed to be regenerated. The best way to avoid this was to disable pureQuery support for the project and re-enable it after the synchronized code had been built.

Database connection using pureQuery

When RSA/RAD is started, by default there is not an active database connection. In the version of Data Studio Developer we were using, if we started a build before opening a connection, all the implementation classes that require regenerating were deleted, and all of the corresponding data interfaces were marked with an error because pureQuery was not able to regenerate its implementation class.

After opening a database connection, building again does not generate the implementation classes nor does it remove the error from the data interfaces. In order to get the implementation classes to be regenerated, either each affected data interface needs to be modified (to flag it as requiring to be built), or a clean and build needs to be started.

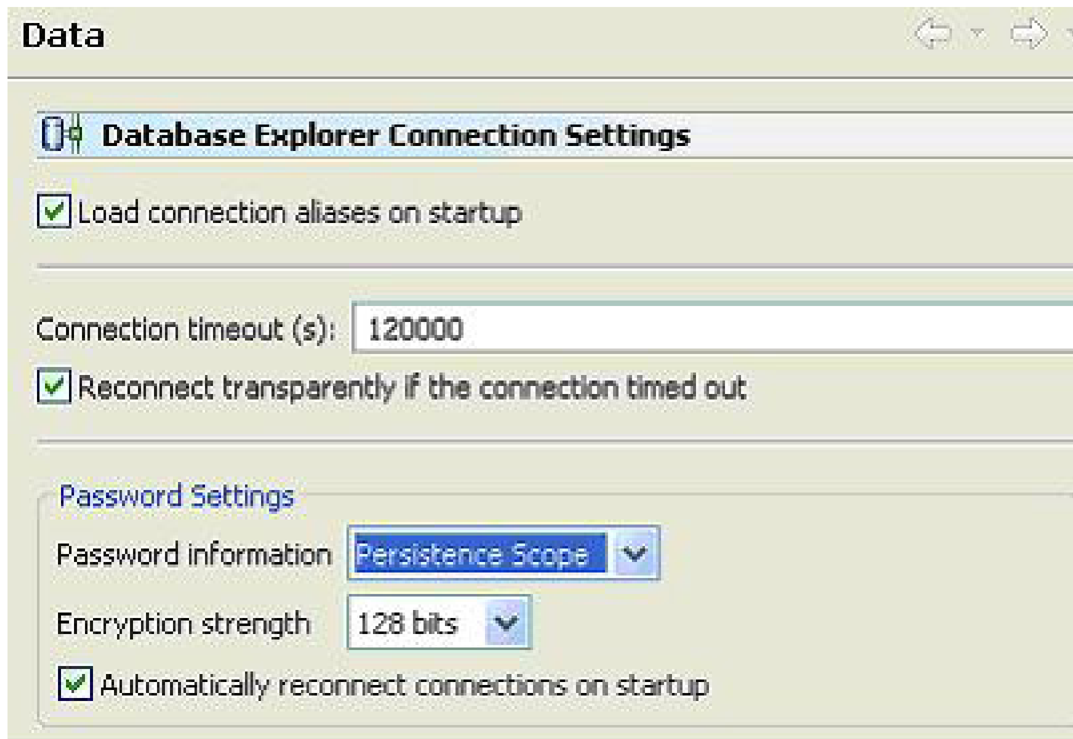
Depending on how many implementation classes need to be generated, modifying all the data interfaces could be time consuming. Running a clean and build can be equally time consuming if there are many implementation classes in the projects which are being built.

Note: In Data Studio Developer 1.2, the implementation classes will no longer be deleted because of the lack of a live database connection. It gives an error on the interface file that implementation wasn't regenerated due to lack of a live connection, and it also gives an error or warning on the implementation indicating that the implementation may be out of sync with interface.

Hint: To avoid forgetting to open a connection after starting RSA/RAD, it is possible to modify the “Password Settings” to “Persistence Scope” on the Database Explorer Connection Settings page (**Windows->Preferences->Data**).

Figure 6. Database connection setting

© Copyright IBM Corporation 1994, 2007. All rights reserved.



The nice thing about using an active database connection is that the SQL statements are validated at compile time. We ended up discovering a few issues with our statements during the build rather than having to run our testing to discover the issues.

Team development with pureQuery

It takes some time to get used to working in a team environment where multiple people may regenerate the same implementation classes and commit code to the repository. The tooling was regenerating all the implementation classes in any project that was enabled; developers working in the same project would regenerate the same implementation classes. To avoid overriding the changes of another developer, there were two ways of properly checking in the code. A developer could check in the implementation classes for any data interface that they had modified, or they could check in all implementation classes with the exception of any which corresponded to a data interface that has been changed by others at check in time. The first approach was safest and easiest they were only checking in a small amount of code changes. If they were checking in a large number of code changes, it was often easier to follow the second approach.

pureQuery's bean collation feature

We had a lot of SQL statements that returned multiple objects. pureQuery's bean collation feature generates RowHandlers that return multiple objects per row in a collection. Listing 10 is an example of one of these statements:

© Copyright IBM Corporation 1994, 2007. All rights reserved.

Listing 10. Return multiple objects in a query

```
@Select(sql="select person.name, …, address.city from person, address")
Iterator<ResultQueue2<Person, Address>>
    getPersonWithAddress(Object[] parameters);
```

We created a series of ResultQueue classes to handle between 1 to 8 return objects. For example, two return objects ResultQueue is defined as follow:

Listing 11. Return multiple objects in a query

```
public class ResultQueue2<A, B> extends LinkedList
```

The only reason for using the generics `<A, B>` is to allow the pureQuery generator to see what type of objects were supposed to be generated and put in the list.

Next steps

In the future, we plan to look into using the static SQL execution capabilities of pureQuery as well as the ability to query in-memory unmanaged objects.

Conclusion

This proof of concept revealed significant possibilities to improve our internal development processes by using pureQuery in our persistence mechanism. It proved the pureQuery assets could co-exist with the present CMP and JDBC code to provide backward compatibility for client extensions and showed support for Global Transactions. We demonstrated better performance when using pureQuery compared with EJB2/CMP. The Data Studio pureQuery Developer tools also proved to be a very productive development environment. Moving the persistence mechanism of WCC to pureQuery was therefore a natural choice when creating the InfoSphere Data Management Server. In February of 2008, MDM Server Version 8 was released with pureQuery.

Acknowledgements

We would like to thank the pureQuery development team and YongLi An from the MDM performance team for their help during the proof of concept and review of this article.

© Copyright IBM Corporation 1994, 2007. All rights reserved.

Resources

Learn

- IBM Master Data Management Server product page: See how the IBM InfoSphere Master Data Management Server can benefit your business by managing your critical master data across customer, product and account domains.
- IBM Data Studio product page: Learn how IBM Data Studio Gives you a comprehensive data management solution to design, develop, deploy and manage data-driven applications.
- "The Easy Way to Quick Data Access" (IBM Database Magazine, Q3 2007): Get a great overview of pureQuery technology and learn how it relates to existing frameworks.
- " No Excuses Database Programming for Java: Make your programs fly with pureQuery and static SQL" (IBM Database Magazine, May 2008) clarifies the benefits of static SQL and how easy it is to try with pureQuery.
- View these demos to see the developer tools in action.
- Data Studio pureQuery tools series (developerWorks): Discover how pureQuery tools make Java programming with SQL more productive than ever before.
- Introducing pureQuery Annotated Method Style (developerWorks, April 08): Get an introduction to the pureQuery annotated method coding style -- a simple, flexible style falling under the named-query paradigm, capable of executing SQL statically or dynamically.
- Introducing pureQuery Inline Style (developerWorks, May 08): Explore the benefits as well as some of the key features of using the inline method programming style.
- Browse the technology bookstore for books on these and other technical topics.

Get products and technologies

- Download IBM Data Studio development tools.

Discuss

- WebSphere Customer Center Forum : Chat with other developers about the WebSphere Customer Center.
- Data Studio Forum: Collaborate with other Data Studio users.
- Visit the Data Studio Community Space on developerWorks to participate in a community devoted to Data Studio.

© Copyright IBM Corporation 1994, 2007. All rights reserved.

About the authors

Wei Zheng
Wei Zheng is a member of Master Data Management Server Architecture team. He joined IBM in 2005 from acquisition of DWL and works on various area of the product. He graduated from Zhejiang University, China with Master Degree in Electronic Engineering.

Paul van Run
Paul van Run is a lead architect and STSM on the IBM Master Data Management team. He came to IBM through the DWL acquisition on 2005. Paul leads the architecture teams for MDM Server and WPC. He has a Master degree in Mathematics (Artificial Intelligence) from the University of Waterloo (Canada) and a Masters degree from the Technical University of Eindhoven in the Netherlands

---

Tim Thorpe
Tim Thorpe is a member of the MDM architecture team. He joined IBM in 2002 as part of the Tarian Software acquisition. He has a Bachelor of Science degree in Computer Science and a Bachelor of Arts degree in Music both from Carleton University.

Kathy Zeidenstein
Kathy Zeidenstein has worked at IBM for a bazillion years. She currently works on the IBM Data Studio enablement team focusing on community development. Before taking on this role, she was the product marketing manager for IBM OmniFind Analytics Edition.

Trademarks

© Copyright IBM Corporation 1994, 2007. All rights reserved.

This is the first trademark attribution statement.
This is the second trademark attribution statement.

© Copyright IBM Corporation 1994, 2007. All rights reserved.