

# Write high performance Java data access applications, Part 3: Data Studio pureQuery API best practices

Learn about pureQuery best practices by looking at code snippets and real-world scenarios.

Skill Level: Advanced

Vitor Rodrigues ([vrodrig@us.ibm.com](mailto:vrodrig@us.ibm.com))  
Software Engineer  
IBM Silicon Valley Lab

14 Aug 2008

pureQuery is a high-performance Java™ data access platform focused on simplifying the tasks of developing, managing and optimizing applications and services that access data. It consists of tools, APIs and a runtime engine. The previous articles in this [series](#) introduced the two programming styles to help users access the database through simple but powerful APIs. This article summarizes some best practices for development using the pureQuery API and gives you real-world scenarios to see how to implement these practices.

## Introduction

### Read important pureQuery articles

- The first two articles of this series describe the [inline style](#) and [annotated method style](#) provided by IBM pureQuery.
- If you are new to pureQuery, you should start with the [pureQuery overview article](#)
- Also, check out this series of articles on the [pureQuery tooling](#).

The first two articles of this series described in detail the two API styles provided by IBM pureQuery: inline style and annotated method style. This third article provides you with insight on various best practices of development using the pureQuery API. Most of these practices exploit the advanced features of the pureQuery API. Whenever applicable, a real world scenario is used to illustrate the usage of the described feature. The code snippets included are for illustrative purposes only, but should give you a good idea of how to use the API.

## Choose your style: Inline vs annotated method style

In the previous articles of this series, the authors presented the two programming styles available in pureQuery: the annotated method style and the inline style.

Both styles have their advantages, so here are some things to consider when trying to decide which style to use:

Use pureQuery annotated method style if you:

- Want to use static SQL at runtime
- Prefer to have your SQL separated from your business logic
- Want a simple data access layer code generated by the Data Studio tooling
- Like to use XML files to define your data access layer
- Have predefined queries

User pureQuery inline style if you:

- Like to have your SQL statements inline in your Java code, just like regular JDBC programming
- Have dynamically generated queries

Still can't decide? If you are still not sure what style you should use, I recommend using the annotated method style. Due to its flexibility and isolation from the business logic, the annotated method style simplifies tasks like refactoring, because code is in a single place, and code reutilization, by sharing your data access interfaces between projects.

## Query over collections

In addition to querying relational databases, you can use the pureQuery API to query in-memory Java collections using the same query language — SQL. This creates a

seamless integration between the database and the Java world. In a distributed environment, network trips to the database are usually one of the most expensive operations, so you can use this alternative query approach to avoid some of these expensive operations.

When you use query over collection, your query can be executed against existing result sets without the need to re-evaluate the query in the database and re-fetch all the data into your application. You can also use this feature to execute join operations between two or more Java collections.

### Scenario: Display product catalog

Suppose your Web application needs to display a catalog of products filtered by a specific brand. On the same Web page, there is a highlighted frame on the right side which shows the best selling products in the same category.

In a typical application, at least two database calls would be required to generate this page: one to fetch all the products to show in the main catalog and another to fetch the most viewed products for highlighting on the right side of the screen. With pureQuery's query over collections, you can improve your application performance by reusing the first result set which contains all the products for the selected category. pureQuery is able to execute a SQL statement on this existing result set, filtering only the products that have the status of bestseller.

In the example in Listing 1, pureQuery's inline style is used to achieve this filtering, but queries over collections are available in both the annotated method and inline method API styles.

To query a Java in-memory collection, you need to get an instance of a data interface that is not associated with a database connection. The fact that there is no associated connection tells pureQuery that you are querying in-memory data and not data stored in a database.

### Listing 1. Query an existing result set residing in memory

```
public void displayProducts(String category){
    Data db = DataFactory.getData(getConnection());
    List<Product> catalog = db.queryList("SELECT PID, NAME, DETAILS, "
        + " PRICE, WEIGHT, CATEGORY, BRAND, SIZE, "
    + " DESCRIPTION, BESTSELLER FROM PDQ_SC.PRODUCT "
    + " where CATEGORY = ?", Product.class, category);
    for (Product p : catalog){
        //list product on webpage
    }
    Data inMemData = DataFactory.getData();
    List<Product> bestsellers = inMemData.queryList("SELECT * FROM "
        + "?1.com.pureQuery.Product as prod WHERE prod.bestseller = 'Y'",
        Product.class, catalog);
    for (Product p : bestsellers){
        //list bestseller
    }
}
```

```
}
```

Line 7 of Listing 1 evaluates a SQL query over the Java collection *catalog* which holds all the products for a specific brand. Note that the API used to query this Java collection is the same one used to query a database. If you have read [Part 2](#) of this series, you should be familiar with the API method `queryList`. Also note the qualified class name in the SQL statement. Because queries using the inline style API are only evaluated at runtime, you need to specify the fully qualified class name in order to get content assist from pureQuery tooling when you are typing the SQL query and for the AI to know which object types are being used.

### Scenario: Generate shipping report

Consider the warehouse department of an e-retailing company. After an order's payment has been cleared, a request is sent to the warehouse to ship the contents of that order. The warehouse has management software that receives the order ID and uses that information to query the `ORDER_ITEMS` table to find out what products make up the order and should be shipped to the customer. After knowing all the products from the order, the software generates a list with the product name and location (aisle and bin) so that the warehouse employees can fetch the product from their locations and add them to the order package. Because the product location information is frequently used, it is kept in memory by the application, in the form of *locator* objects. The following code snippet shows how to join an order's products and locator information to generate the shipping report to be used by warehouse employees:

### Listing 2. Joining two in-memory collections using pureQuery API

```
public List<ProductInfo> generateShippingReport(String orderID){
    Data db = DataFactory.getData(getConnection());
    List<Locator> locators = LocatorUtil.getLocators();
    Iterator<Product> products = db.queryIterator("SELECT p.* from PRODUCT AS p, "
        + " ORDER_ITEMS AS po where p.pid = po.pid and po.poid = ?",
        Product.class, orderID);
    Data inMemData = DataFactory.getData();
    List<ProductInfo> shippingReport = inMemData.queryList("SELECT pr.pid, "
        + " pr.name, lr.aisle, lr.bin FROM ?1.com.pureQuery.Product AS pr, "
        + " ?2.com.pureQuery.Locator AS lr where pr.pid = lr.pid",
        ProductInfo.class, products, locators);
    return shippingReport;
}
```

Looking in more detail at Listing 2, you can see that the location information is generated by the business logic, while the product information is fetched from the database. As in [Listing 1](#), you need to create an instance of the `Data` interface not associated with a database connection in order to execute queries over in-memory objects.

After executing the join statement, `shippingReport` will contain information merged from `locators` and `products` collections.

## Pluggable callback mechanism using the Hook interface

pureQuery's Data interface lets you attach statement hooks to their connections. A hook is similar in functionality to a database trigger. It provides a way to define functionality that is executed before and/or after each API call is executed. You can take advantage of this feature in several ways:

- **Performance monitoring:** You can use hooks to measure API calls runtime aspects like execution time, network, and I/O.
- **Validate data:** Statement hooks let you validate parameter data before statement execution, providing the opportunity to do constraint check and data validation at the application level.
- **Auditing SQL:** If you need to audit all the SQL statements executed by your pureQuery application, hooks provide an easy way to do it.

The fact that hooks are simply attached to the data object makes your application oblivious to it (with exception of the code snippet where you create your data object). For this reason, you can achieve all of the above functionality without the need to refactor any of your code.

### Scenario: Implementing a performance monitor

Let's look at how to use a hook to implement a performance monitoring solution for your application.

The first step is to define a class that implements the pureQuery Hook interface. Listing 3 shows the code used to do this.

### Listing 3. SystemMonitorHook, used to monitor database access performance

```
public class SystemMonitorHook implements Hook
{
    DB2SystemMonitor systemMonitor;

    public void pre (String methodName, Data dataInstance,
        SqlStatementType sqlStatementType, Object... parameters)
    {
        try {
            systemMonitor = ((DB2Connection)dataInstance.getConnection()).getDB2SystemMonitor();
            systemMonitor.enable (true);
            systemMonitor.start (DB2SystemMonitor.ACCUMULATE_TIMES);
        }
        catch (SQLException e) {
            throw new RuntimeException ("Unable to start system monitor " + e.getMessage ());
        }
    }
}
```

```
}

public void post (String methodName, Data dataInstance, Object returnValue,
                 SqlStatementType sqlStatementType, Object... parameters)
{
    try {
        systemMonitor.stop ();
        System.out.println("Performance of method: " + methodName + ":");
        System.out.println ("Application Time " +
            systemMonitor.getApplicationTimeMillis () + "milliseconds");
        System.out.println ("Core Driver Time " +
            systemMonitor.getCoreDriverTimeMicros () + "microseconds");
        System.out.println ("Network Time " +
            systemMonitor.getNetworkIOTimeMicros () + "microseconds");
        System.out.println ("server Time " +
            systemMonitor.getServerTimeMicros() + "microseconds");
    }
    catch (SQLException e) {
        throw new RuntimeException
            ("Unable to stop system monitor " + e.getMessage ());
    }
}
}
```

The `SystemMonitorHook` class implements the `pureQueryHook` interface that declares the methods `pre()` and `post()`. These two methods are executed before and after a `pureQuery` API call is executed. [Listing 3](#) uses parts of the IBM JCC driver proprietary API. To learn more about this API, please refer to the DB2 Information Center page about the [IBM Data Server for JDBC](#).

This hook can be used by any of our `pureQuery` applications. To trigger its usage, you just need to attach it to your `Data` instance as [Listing 4](#) illustrates:

#### Listing 4. Associating a hook with a connection

```
//...
Connection con = getConnection();
SystemMonitorHook monitorHook = new SystemMonitorHook();
Data data = DataFactory.getData(CustomerData.class, con, monitorHook);
// ...

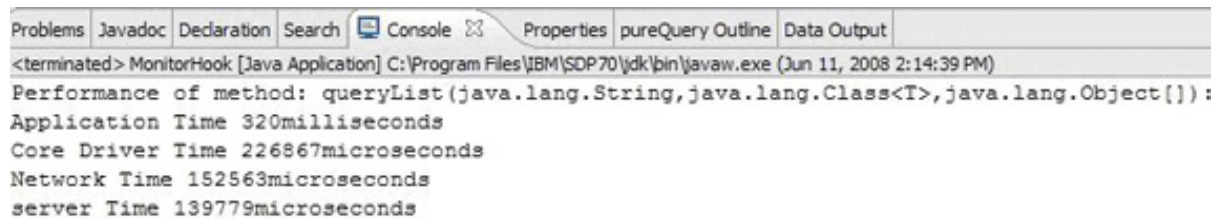
data.queryList("select * from pdq_sc.product", Product.class);

// ...
```

By attaching the hook to your `Data` instance, you activate your monitoring mechanism which is executed for every API call.

The output for the sample `queryList` call in [Listing 4](#) is shown in [Figure 1](#):

#### Figure 1. Output from our system monitor hook



```
Problems Javadoc Declaration Search Console Properties pureQuery Outline Data Output
<terminated> MonitorHook [Java Application] C:\Program Files\IBM\SDP70\jdk\bin\javaw.exe (Jun 11, 2008 2:14:39 PM)
Performance of method: queryList (java.lang.String, java.lang.Class<T>, java.lang.Object[]):
Application Time 320milliseconds
Core Driver Time 226867microseconds
Network Time 152563microseconds
server Time 139779microseconds
```

Several performance metrics are printed by the monitor, including application time, driver time, network time and server time.

## queryList and queryArray vs queryIterator

pureQuery provides three API methods that return collections of Java objects: *queryArray*, *queryList*, and *queryIterator*. The rule of thumb is that you should use the one method that best matches the collection type the application is waiting for, so that type conversions are avoided.

However, there is more to these methods than just different return types. The way these methods work under the covers is also important and should be taken into account when developing your application. While *queryArray* and *queryList* do eager materialization of the result set, *queryIterator* does lazy materialization, fetching data on demand as the method `Iterator.next()` is called.

Consider the following hints when deciding which API methods to use.

Use *queryArray* or *queryList* when:

- You want to be able to traverse the result set multiple times.
- Your application can allocate enough memory to load all the data into the collection.

User *queryIterator* if:

- You would like to do paging of results. Data is fetched on demand as you display new pages.
- Your application has a reduced amount of available memory.

## Take advantage of pureQuery batching

The next two sections describe how to take advantage of batching facilities provided by pureQuery to achieve both homogeneous and heterogeneous types of batching.

### Homogeneous batch updates

A common requirement in database applications is to insert several rows into the same table as part of the same operation. JDBC provides batching facilities; however, these facilities are very verbose (you need to set statement parameters manually) and somewhat complex to use.

To assist with the requirement for batch updates associated with an easy-to-use API, pureQuery inline style provides the `updateMany` method. This method can be used for homogeneous batching and receives only two parameters: the update SQL statement and a Java collection of objects to be batched into the update call. Under the covers, `updateMany` implements the JDBC best practice of batch updates, drastically reducing the number of network trips required for updating the data. It also ensures that all the updates occur in a single transaction.

### Scenario: Update product database

Every week, a backend application receives several updates from partners with the list of new products they have available for sale. Your application needs to update the database used by the online store application, so that these new products will show up when a user browses the catalog. The easiest and fastest way to do it is by using the API method `updateMany()` as Listing 5 illustrates:

#### Listing 5. Homogeneous batch update API call

```
//...
Data db = DataFactory.getData(getConnection());
List<Product> prods = getNewProducts();
db.updateMany("INSERT INTO PRODUCT (PRODUCTID, NAME, DETAILS, LISTPRICE, "
    + " WEIGHT, CATEGORY, BRAND, SIZE, DESCRIPTION)"
    + " VALUES (:productid, :name, :details, :listprice, :weight, :category, "
    + " :brand, :size, :description)", prods);
```

Note that only a single API call is needed in order to update several rows in the database table. pureQuery batches the insertion of the new products contained in the variable `prods`.

When using pureQuery annotated method style programming, homogeneous batch is implicit. If a method has a collection of beans as a parameter, pureQuery interprets that as a homogeneous batch call.

### Heterogeneous batch updates

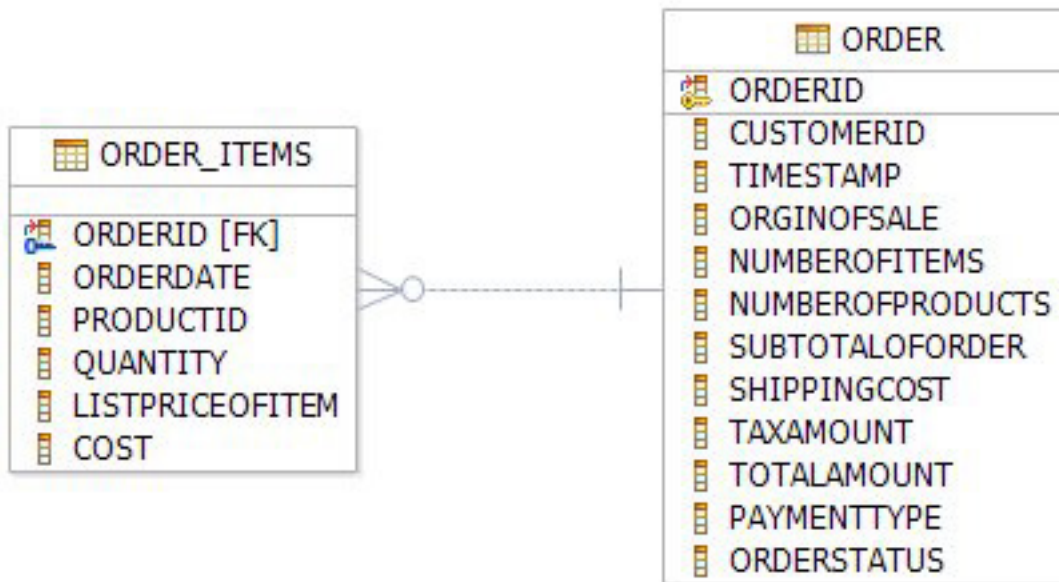
While pureQuery's `updateMany` method provides an optimized way to do parameter batching with a single SQL statement to a single table, sometimes applications require more complex operations than that, such as updating more than one table.

### Scenario: Updating purchase orders

Consider a scenario where a customer purchase order is stored in two tables: ORDER with the order summary and ORDER\_ITEMS with the list of items for each order. The SQL standard does not provide a way to insert/update/delete multiple tables in a single statement, so applications need to execute two separate statements, one for each table. Even if you use homogeneous batch to update the ORDER\_ITEMS table, your application still needs to execute two separate network calls — one to update the ORDER table and another to update the ORDER\_ITEMS table. Also, you will need to control the transaction yourself to make sure that the database is in a consistent state after you update both tables.

Figure 2 shows the one-to-many relationship between ORDER and ORDER\_ITEMS tables used for this scenario.

**Figure 2. One-to-many relationship between ORDER and ORDER\_ITEMS**



In Version 1.2, pureQuery introduces support for heterogeneous batch updates. A heterogeneous batch update lets you combine several SQL statements with or without parameter markers into a single network call. In this scenario, you could use a heterogeneous update to update the ORDER and ORDER\_ITEMS table in a single database operation. While existing JDBC batching only supports batching of statements with literals, pureQuery heterogeneous batch supports batching of parameterized statements. By supporting parameterized statements, you exploit the advanced features provided by these, like access path reuse and SQL injection prevention.

While homogeneous batch operations are achieved inside a single API call which operate only on one table, heterogeneous batch operations can include several API calls and even span across multiple data access objects, which impacts multiple tables. This becomes very useful when you need to have a mix of inline and method

style and/or calls to different method interfaces inside the same transaction.

Listing 6 uses pureQuery's heterogeneous batch to update several tables, executing multiple calls to the *OrderData* method style interface:

### Listing 6. Heterogeneous batch update using pureQuery API

```
public void insertPurchaseOrder(PurchaseOrder po, String poid){
    OrderData orderData = DataFactory.getData(OrderData.class, getConnection());
    //start batch
    ((Data)orderData).startBatch(HeterogeneousBatchKind.heterogeneousModify__);
    //create new order
    orderData.insertNewPurchaseOrder(po);
    //add items to order
    for (OrderItem oi : po.getItems())
    {
        orderData.addItemToPurchaseOrder(poid, oi);
    }
    // end batch
    ((Data)orderData).endBatch();
}
```

Note that the methods `startBatch()` and `endBatch()` belong to the `com.ibm.pdq.runtime.Data` interface, so you need to cast your `OrderData` object to `Data` before we can call those methods. Alternatively, you can have your `OrderData` interface extend the `Data` interface so that casting is not required. Between `startBatch()` and `endBatch()`, you have all the method style interface calls that you want to execute in a single batch transaction. In [Listing 6](#), your multiple API calls inside the batch block guarantee that all the information relative to a purchase order will be updated in a single transaction.

But there is more to pureQuery's heterogeneous batch! In addition to support the execution of several operations in the same `OrderData` object, pureQuery batch also supports aggregation of operations from different API styles, allowing for a mix between inline and annotated method styles inside the same batch operation. Suppose you want to reuse the example above and add another operation that updates the product inventory, decreasing the product quantity for each item in the purchase order. Furthermore, let's say you want to execute this operation using the inline method style, while inserting the purchase order into the database using the annotated method style. Listing 7 shows code similar to the one you would use to implement these changes.

### Listing 7. Heterogeneous batch update using different Data objects

```
public void insertPurchaseOrder(PurchaseOrder po, String poid){
    Data data = DataFactory.getData(getConnection());
    OrderData orderData = DataFactory.getData(OrderData.class, data);
    //start batch
    data.startBatch(HeterogeneousBatchKind.heterogeneousModify__);
    //create new order
    orderData.insertNewPurchaseOrder(po);
    //add items to order
```

```

for (OrderItem oi : po.getItems())
{
    orderData.addItemToPurchaseOrder(poid, oi);
}
//update inventory
data.updateMany("UPDATE INVENTORY SET " +
    " QUANTITY = QUANTITY - 1 WHERE PRODUCTID = :pid",
    po.getItems());
// end batch
data.endBatch();
}

```

As shown in Listing 7, both methods from `data` and `orderData` objects are being called inside the batch execution. Even though I am referring to two different objects, these calls are executed inside a single batch operation, since both objects reference the same underlying Data object (note second parameter of the call to `DataFactory.getData()`).

### Listing 8. OrderData interface

```

public interface OrderData {
    //insert a new purchaseOrder
    @Update(sql = "insert into DB2ADMIN.ORDER(orderid, customerid, numberofitems, " +
        " numberofproducts, subtotaloforder, taxamount, totalamount, timestamp) " +
        " values(:orderid, :customerid, :numberofitems, :numberofproducts, " +
        " :subtotaloforder, :taxamount, :totalamount, :timestamp)")
    void insertNewPurchaseOrder(PurchaseOrder po);

    //add product to PurchaseOrder
    @Update(sql="insert into DB2ADMIN.ORDER_ITEMS(orderid, productid, quantity, cost)"
        + " values(?1, ?2.pid, ?2.quantity, ?2.price)")
    void addItemToPurchaseOrder(String poID, OrderItem p )
}

```

Listing 8 shows the methods of the `OrderData` interface, detailing the calls `insertNewPurchaseOrder` and `addItemToPurchaseOrder`. Note that on the `addItemToPurchaseOrder` method, named parameters are used to specify which variable of the object `po` should be used as the parameter value. On the `insertNewPurchaseOrder` method, a combination of numbered and named parameters is used to refer to the parameter values.

In the same way that inline and method style APIs are used in the heterogenous batch example, you can also use several method style interfaces in the same heterogenous batch operation. For that, all the method style interfaces need to be created using the same base Data object.

## Customize your result sets using ResultHandlers and RowHandlers

`pureQuery` provides some basic object-table mapping functionality that can be very useful when you are developing your data access layer. Data Studio Developer

helps automate this step by providing tooling to generate Java beans that map to database tables, which can improve your productivity.

However, sometimes applications require more complex mapping than can be achieved with this simple object-table mapping. There are times when you only need to map a subset of a table to a Java object; while in other cases, you may want to map a table row into multiple objects.

Also, it is a common requirement to transform result sets into a non-relational format, like HTML, XML or custom Java objects.

pureQuery allows users to implement custom mapping patterns that can be used to meet the needs described above.

### Scenario: Display several results sets in HTML

Consider an application that needs to display several result sets in HTML format. One way to automate this task is to use pureQuery's result handlers. A result handler is used to convert a result set contents into an object. In the next example, the result handler processes a result set and returns an HTML page, displaying the result set contents as a table.

Here is a snippet of my HTMLHandler class:

#### Listing 9. Result Set handler that generates HTML pages

```
public class HTMLHandler implements ResultHandler<String>
{
    private DocumentBuilderFactory documentBuilderFactory_;
    private DocumentBuilder domBuilder_;
    private Transformer transformer_;

    public HTMLHandler ()
    {
        // ... initialize variables
    }
    //...

    public String handle (ResultSet resultSet)
    {
        Document document = domBuilder_.newDocument ();

        // Create root element
        Element rootElement = document.createElement ("html");
        rootElement.setAttribute ("xmlns", "http://www.w3.org/TR/REC-html40");
        document.appendChild (rootElement);
        Element headElement = document.createElement ("head");
        rootElement.appendChild (headElement);
        Element titleElement = document.createElement ("title");
        titleElement.setTextContent ("HTML Table Printer");
        rootElement.appendChild (titleElement);
        Element bodyElement = document.createElement ("body");
        rootElement.appendChild (bodyElement);
        generatedTable (resultSet, bodyElement, document);
    }
}
```

```

    return transformXML (document);
}

private void generatedTable (ResultSet resultSet, Element bodyElement,
    Document document)
{
    ResultSetMetaData resultSetMetaData = resultSet.getMetaData ();
    int columnCount = resultSetMetaData.getColumnCount ();
    Element tableElement = document.createElement ("TABLE");
    tableElement.setAttribute ("border", "1");
    bodyElement.appendChild (tableElement);
    Element headerRowElement = document.createElement ("TR");
    tableElement.appendChild (headerRowElement);

    for (int index = 0; index < columnCount; index++) {
        Element headerElement = document.createElement ("TH");
        headerElement.setTextContent (resultSetMetaData.getColumnLabel (index + 1));
        headerRowElement.appendChild (headerElement);
    }
    while (resultSet.next ()) {
        Element rowElement = document.createElement ("TR");
        tableElement.appendChild (rowElement);
        for (int index = 0; index < columnCount; index++) {
            Element columnElement = document.createElement ("TD");
            columnElement.setTextContent (resultSet.getString (index + 1));
            tableElement.appendChild (columnElement);
        }
    }
}
// ...
}

```

For simplicity, Listing 9 shows only a part of the `HTMLHandler.java` file.

To use this and other result handlers, the `pureQuery` API provides the `query()` method. This method receives several parameters, including a SQL statement and a result handler, and returns an object of generic type *T*. This type is defined by the runtime type *T* of the parameterized `RowHandler<T>` interface. In the example in Listing 10, the `HTMLHandler` processes a result set and returns an object of type `String`, which contains the textual representation of a Web page that lists all the result set rows.

To convert the result of the query into an HTML page, simply pass an `HTMLHandler` to the API call:

### Listing 10. Passing our `HTMLHandler` result handler to the API call

```

public String generateProductList(){
    Data db = DataFactory.getData(getConnection());
    String htmlpage = db.query("SELECT * from PRODUCT", new HTMLHandler());
    return htmlpage;
}

```

## Scenario: Handling addresses of different structures

Commonly, a row in a database table can store information from different objects in

your Java application. Consider the table ADDRESS, which contains addresses from my customers. Although I am using only one table to store this information, several countries have different address structure. This often causes unused table columns or the use of the same column to store different properties. For example, US state and Canadian provinces can both be saved in a column name “STATE;” although, in your Java beans, you need to clearly have variables named *state* and *province*.

Let’s define your *Address* interface. Your application retrieves customers’ addresses from the database and prints them in the typical format used on address labels so that they can be attached to the shipment boxes. The only method you need to implement is `printableFormat()` which returns the address as it should be printed.

### Listing 11. Sample Address interface

```
public interface Address {  
    public String printableFormat();  
}
```

Because you have customers in both US and Canada, you have two implementations of the Address interface:

### Listing 12. USAddress and CANAddress Java classes

```
public class USAddress implements Address {  
    protected String customerName;  
    protected String street;  
    protected String city;  
    protected String state;  
    protected String zipcode;  
    //...  
}  
  
public class CANAddress implements Address {  
    protected String customerName;  
    protected String street;  
    protected String city;  
    protected String province;  
    protected String postalCode;  
    //...  
}
```

At run-time, your `RowHandler` decides which one is the appropriate object to return for the current result set row:

### Listing 13. Address result handler

```

public class AddressHandler implements RowHandler<Address> {
    public Address handle(ResultSet rs, Address object) throws SQLException {
        Address addr = null;
        if (rs.getString(3).equals("United States")){
            USAddress us = new USAddress();
            us.setCustomerName(rs.getString(2));
            us.setStreet(rs.getString(4));
            us.setCity(rs.getString(5));
            us.setState(rs.getString(6));
            us.setZipcode(rs.getString(7));
            addr = us;
        } else if (rs.getString(3).equals("Canada")){
            CANAddress can = new CANAddress();
            can.setCustomerName(rs.getString(2));
            can.setStreet(rs.getString(4));
            can.setCity(rs.getString(5));
            can.setProvince(rs.getString(6));
            can.setPostalCode(rs.getString(7));
            addr = can;
        }
        return addr;
    }
}

```

In your application, refer to all objects as objects of type *Address*, instead of having to deal with both types of addresses:

#### Listing 14. Processing our result set using the AddressHandler

```

//...
Data db = DataFactory.getData(getConnection());
List<Address> addrs = db.queryList("SELECT * FROM CUSTOMERADDRESS",
                                new AddressHandler());
// process list of Address objects...

```

For the sake of simplicity, we do not show calls to any interface methods in this example. In a real world application, the *Address* interface would define several methods to work with addresses, regardless of whether it is a *USAddress* or *CANAddress* object.

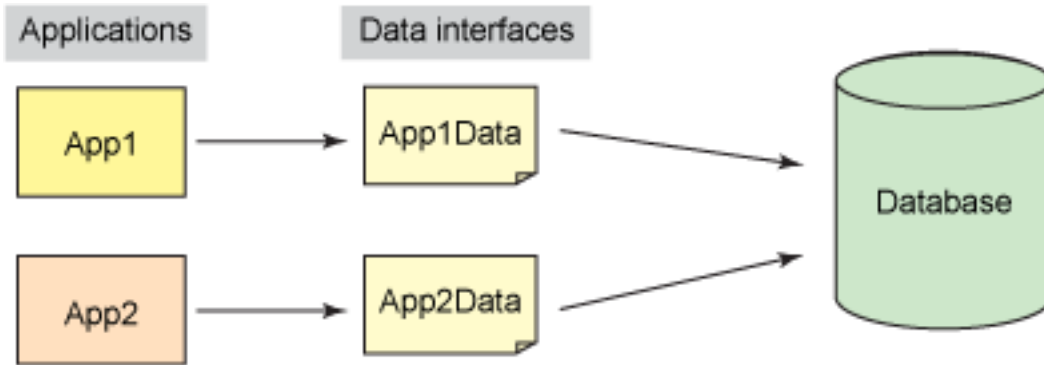
## Define the most appropriate granularity for your method style interfaces

When developing an application using pureQuery Method style, it is important to define the most appropriate granularity for your method style interfaces. Although there is no magic recipe for how this granularity should be defined, there are certainly some guidelines that can help with achieving the design that best matches your needs:

- If your data access layer is very application-specific, that is, it contains data access code that is only used by the one application you are currently developing, then it is a good idea to aggregate all the data

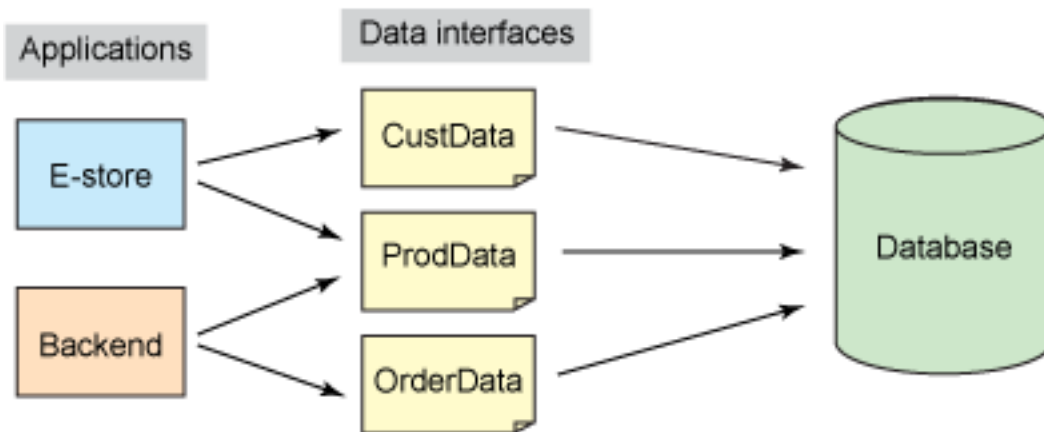
access code in the same interface. Using this approach, each one of your applications has its own method style interface, making it easier to manage. (See Figure 3 for an architectural view.)

**Figure 3. Application specific architecture**



- On the other hand, if you are developing data access code that will be required by several applications, you should separate that code into logical units and create an annotated method style interface for each logical unit (See Figure 4). With this approach, your applications are able to share data access interfaces, reusing code and reducing the amount of work needed to build the complete application.

**Figure 4. Logical unit architecture**



For some people, creating a method style interface for each database table may seem a very straightforward approach. However, I think that separating the data access into logical units instead of data units is a better approach for the following reasons:

- Rarely does an application access only one table, so by taking this approach, some operations would require instantiating and working with more than one interface. For example, saving an order in the database requires updating the ORDER and ORDER\_ITEMS table, thus an *OrderData* interface that works both with table ORDER and table

ORDER\_ITEMS is more appropriate than having two separate interfaces. The same is true for retrieving data; every time you retrieve data from the ORDER\_ITEMS table, you also want to get the order information, so ORDER table will be accessed too. Logical units should be created to aggregate database access to related objects.

- In pureQuery v1.1, each Data interface is bound to a different Static SQL package. Having one interface per table would require more DBA management to achieve the same results.
- If you are using static SQL, then the collection of SQL into packages is determined by the methods in the interface; therefore, you might want to consider what collection of SQL you want to have in your database package.
- If you are going to use heterogenous batch you might want to collect the SQL statements that you want to batch into one interface to make it simpler to use heterogeneous batch.

## Implement pagination using a Paging handler

Pagination is a common approach used to display large amounts of data. Applications like Web catalogs, multi-page sales reports and others make extensive use of paging, fetching a new block of data for each new page displayed.

The pureQuery API provides a result handler called `IteratorPagingResultHandler`. Just like any other `ResultHandler`, you pass this handler to the Data object methods when calling the API. The constructor of this handler allows you to specify several options, including the Class of the beans being returned from the result set, or even a `RowHandler` to handle each one of the returned rows. On top of specifying how the data is returned, you can also specify how much and which data is returned from the database, according to two different schemes:

- **Block retrieval:** All rows in the interval specified by the parameters `absoluteStartingRow` and `absoluteEndingRow` are retrieved.
- **Page retrieval:** The application specifies `pageSize` and `pageNumber` parameters. The handler returns the `pageNumber`<sup>th</sup> page of data containing `pageSize` rows.

Listing 15 is an example of how to use `IteratorPagingResultHandler`:

### Listing 15. Paginating results using the `IteratorPagingResultHandler`

```
//...
```

```
int pageNumber = this.getCurrentPage();
int pageSize = this.getPageSize();

Iterator<Product> prods = db.query("SELECT * from PRODUCT",
    new IteratorPagingResultHandler<Product>(pageNumber, pageSize, Product.class));

// display Product objects...
```

In the code sample above, if the `pageNumber` value was 2 and the `pageSize` value was 15, the variable `prods` would contain the products from rows 16 to 30 in our `PRODUCT` table.

## Stored procedure CallHandler

Stored procedures have a nature of their own when it comes to database objects having multiple return values. While SQL statements return result sets and UDFs return values (scalar or tabular), stored procedures can return multiple values in `OUT` and `INOUT` parameters, in addition to multiple result sets. This characteristic makes stored procedures a complex database resource to handle using JDBC. In order to access all the information returned by a stored procedure call, developers need to register all the output parameters beforehand and assign the result of the call to a `ResultSet` object. Fortunately for us developers, `pureQuery` provides a `StoredProcedureResult` object type that can be used to store all the output information generated by a stored procedure call, including output parameters and result sets.

The next code sample describes how to make use of the `StoredProcedureResult` object type to access all the output information of a stored procedure call:

### Listing 16. Processing stored procedure output using `StoredProcedureResult`

```
//...
int medianSalary = 75000;

StoredProcedureResult spr = db.call ("call TWO_RESULT_SETS (?)", medianSalary);

String[] outParms = (String[])spr.getOutputParms ();

System.out.println ("Output Parameter(s) length: " + outParms.length);
System.out.println ("List of Products");

Iterator<Product> prods = spr.getIterator(Product.class);
while (prods.hasNext()) {
    Product p = prods.next();
    System.out.println("Name: " + p.getName());
}

spr.close ();
// ...
```

By using the `StoredProcedureResult` object, you avoid the need to register output parameters before calling the stored procedure. This not only reduces the

amount of code you have to type, but it also simplifies the process, because you are dealing with only a single object, instead of handling a set of various output parameters and the result set returned by the stored procedure.

## Cursor processing

One of the aspects that makes pureQuery a unique approach to data persistence is that, although it provides several automated steps and mappings, it never deprives you of the same level of control achieved with more low level approaches like plain JDBC programming. You always have the option to take advantage of the automatic mappings and provided APIs of pureQuery, or take back control and implement your own result set handlers or connection hooks.

If you need to control how the data is fetched from the database, pureQuery provides the option to define the type, concurrency level and holdability of the database cursor used to fetch the data. These settings can be passed as parameters to the API methods or defined as pureQuery annotations when using the annotated method style. The valid values for these parameters reflect the values supported by the JDBC API (You can check the supported values by referring to the [JDBC ResultSet API page](#)).

Listing 17 shows a pureQuery API call used to set the cursor's parameters:

### Listing 17. Setting cursor attributes in a pureQuery API call

```
//...
Data db = DataFactory.getData(getConnection());
Iterator<Product> prods = db.queryIterator(java.sql.ResultSet.TYPE_FORWARD_ONLY,
    java.sql.ResultSet.CONCUR_READ_ONLY,
    java.sql.ResultSet.CLOSE_CURSORS_AT_COMMIT,
    "SELECT * FROM PRODUCT", Product.class);
//...
```

Listing 17 uses the cursor to fetch all the products from the database to be a forward-only cursor that executes in read-only mode and is closed when the transaction is committed.

## Closing resources

Although pureQuery does most of the resource management for you, improving your application's overall performance, there are a few cases when you need to be concerned about closing resources.

pureQuery helps you by automatically closing statements, result sets, and cursors when they are not in use anymore. In addition, it closes objects of type

`ResultIterator` and `StoredProcedureResult` when their contents have been consumed. If your API call returns one of these object types and you consume all of its contents, they are automatically closed by `pureQuery`.

If your application logic can lead to situations when the contents of the result objects may not have all been fetched, then you should explicitly close those resources. Data objects should also be closed once the data access part of your application is complete, so that the connection object associated with it is released too.

Listing 18 shows an example of how to close an *Iterator* object returned by `pureQuery`:

### Listing 18. Closing an ResultIterator object

```
//...
Iterator<Product> prods= db.queryIterator("SELECT * from DB2ADMIN.PRODUCT",Product.class);
// work with iterator variable "prods"
((ResultIterator<Product>)prods).close();
```

Because `pureQuery` returns an object of the generic type *Iterator*, you need to cast the `prods` variable into an object of type `ResultIterator<T>`, since this is the iterator type implemented by `pureQuery` and that provides the `close()` functionality.

## Summary

This article has described several best practices for `pureQuery` developers. The list included advanced API features, as well as rules of thumb for development decisions. By following these recommendations, you can expect to increase your productivity and write cleaner code, since most of the JDBC burden disappears when using `pureQuery` API.

I hope you find this article to be a useful resource to assist you in developing `pureQuery` applications and getting the most out of `pureQuery`. I am waiting for your feedback on the article and on `pureQuery`.

# Resources

## Learn

- Read the article [Write high performance, Java data access applications, Part 1: Introducing pureQuery annotated method style](#) (Heather Lamb, developerWorks, April 2008) to learn about the pureQuery API annotated method style.
- Read the article [Write high performance, Java data access applications, Part 2: Introducing pureQuery inline style](#) (Daya Vivek, developerWorks, May 2008) to learn about the pureQuery API inline method style.
- “No Excuses” [Database Programming for Java](#) (Kathy Zeidenstein and Bill Bireley, May 2008, IBM Database Magazine) is an excellent article that explains the benefits of Static SQL and how to exploit those benefits using Java.
- A wealth of Data Studio information is available in the [IBM Data Studio Information Center](#).
- For more information about pureQuery and its API, you can refer to the online [pureQuery documentation](#).
- Visit the [developerWorks resource page for Data Studio](#) to read articles and tutorials and connect to other resources to expand your Data Studio skills.

## Get products and technologies

- [IBM Data Studio](#) – download free of charge.

## Discuss

- [Participate in the discussion forum for this content](#).
- [Data Studio Community Space](#)
- [Managing the Data lifecycle: from design to deletion](#) – a blog from the Data Studio team
- [IBM Data Studio discussion forum](#) – general discussion about the Data Studio family of products.

## About the author

Vitor Rodrigues

Vitor Rodrigues is a software developer in the IBM Data Studio Developer team and works at the Silicon Valley Lab. He graduated from University of Minho, Portugal, in Computer Science and Systems Engineering. Vitor joined IBM in 2005 as an intern working on DB2 Everyplace and DB2 9 pureXML. Prior to joining Data Studio

developer team, he was a member of the Technical Enablement team for DB2 pureXML and IBM Data Studio, working out of the IBM Toronto and Silicon Valley labs.

## Trademarks

This is the first trademark attribution statement.

This is the second trademark attribution statement.