Using MQSeries from DB2 Applications

Dan Wolfson

August 06, 2001

Author Dan Wolfson of IBM’s Database Technology Institute describes how to use new features in DB2 UDB Version 7.2 to easily integrate MQSeries messaging with database applications in this article. Wolfson explains how the new functions provide seamless access to MQSeries messaging within standard SQL statements to support a broad range of applications -- from simple event notification to operational data store creation.

© 2001 International Business Machines Corporation. All rights reserved.

This paper describes how to use the new features available in DB2® UDB V7.2 to easily integrate MQSeries® messaging with database applications. These functions provide seamless access to MQSeries messaging within standard SQL statements to support a wide range of applications ranging from simple event notification to operational data store creation.

1. Introduction

Many DB2 customers also use messaging, queuing, or publish/subscribe systems in their application environments. Such systems have a wide variety of uses, from linking together disparate applications, to disseminating real-time information, to integrating data and communication with external partners. Often, such uses combine database operations with messaging operations in the same applications. Today, this requires that application programmers combine these two very different models together to craft an application. The focus of this paper is to explain how new features in DB2 7.2 can both simplify application development and leverage the combined power of DB2 and MQSeries.

DB2 7.2 provides many new MQSeries integration features. These include a set of MQSeries functions that are callable from within SQL statements, an MQSeries Assistant to simplify the mapping of queues to DB2 tables, support for XML messages using the XML Extender, and support for MQSeries as a new kind of data source from the Data Warehouse Center. This paper will primarily focus on the core of the MQSeries support, the integration of MQSeries into SQL. All of the other features leverage this fundamental capability. Subsequent papers will review these other capabilities.
2. MQSeries Overview

MQSeries is the core of the MQSeries product family. With overwhelming market share, MQSeries is the leading messaging middleware used throughout the corporate world. It is a flexible messaging system that allows applications to communicate in a distributed, heterogeneous environment. The MQSeries product family also includes MQSeries, MQSeries Integrator (MQSI), MQSeries Workflow, and the MQSeries Adapter Offering.

MQSI is a message broker and publish/subscribe server. A message broker is capable routing and transforming messages based on a set of rules. MQSI is often used in enterprise application solutions to integrate heterogeneous applications. The core of MQSI is the dataflow engine that accepts messages from a queue and processes them according to a sequence of operations described by a user-defined dataflow. Operations include the ability to transform message content, perform database operations, and route messages to one or more destinations.

MQSeries Workflow is a comprehensive workflow environment that supports both people-oriented and process-oriented workflows. Workflows are "statefull" sequences of operations. Each operation may be a standalone application that is triggered by an MQSeries message.

MQSeries Adapter Offering is a new product that simplifies the integration of packaged applications by providing some pre-defined integration libraries and by simplifying the construction of new integration libraries.

2.1 MQSeries Messaging Styles

MQSeries supports three messaging models: datagrams, publish/subscribe (p/s), and request/reply (r/r). Messages sent as datagrams are sent to a single destination with no reply expected. In the p/s model, one or more publishers sends a message to a publication service which distributes the message to interested subscribers. Request/reply is similar to datagram - but the sender expects to receive a response.

MQSeries is used in a wide variety of ways. Simple datagrams are exchanged to coordinate multiple applications, to exchange information, request services, and provide notification of interesting events. Publish/Subscribe is most often used to disseminate real-time information in a timely manner. The request/reply style is generally used as a simple form of pseudo-synchronous remote procedure call. More complex models can of course be constructed by combining these basic styles.

These fundamental messaging techniques are used in an exceptionally wide variety of ways. Because MQSeries is available across a very wide range of platforms it provides an important mechanism to link together disparate applications, from either similar or dissimilar environments. MQSeries is often used in Enterprise Application Integration (EAI) configurations to link packaged applications such as SAP into existing environments and to help extend such application packages with additional functionality. Because MQSeries offers guaranteed delivery of messages, many companies use it within business critical systems as the core mechanism by which key information is conveyed among applications. Quite often, these applications also interact with a database as either the source or destination of the information to be conveyed within the message.
2.2 Message Structure
MQSeries does not, itself, mandate or support any particular structuring of the messages it transports. Other products, such as MQSeries Integrator (MQSI) do offer support for messages formed as C or Cobol or as XML strings. Structured messages in MQSI are defined by a message repository. XML messages typically have a self-describing message structure and may also be managed through the repository. Messages may also be unstructured, requiring user code to parse or construct the message content. Such messages are often semi-structured - that is, they use either byte positions or fixed delimiters to separate the fields within a message.

2.3 MQSeries APIs
Three primary application programming interfaces are provided with MQSeries. The most ubiquitous is MQI, which has only a few simple interfaces but many options and parameters. MQI programmers typically intermix messaging requests with the information needed to control MQSeries. MQI has both procedural and object interfaces supporting many common programming languages.

The Application Messaging Interface (AMI) of MQSeries provides a clean separation between messaging actions and the definitions that dictate how those actions should be carried out. These definitions are kept in an external repository file and managed using the AMI Administration tool. This makes AMI applications very simple to develop and maintain. AMI is relatively new and is not yet completely supported on all platforms. The performance overhead of AMI is somewhat higher than MQI.

Finally, the Java ® Messaging Service (JMS) provides a standard Java interface to MQSeries. JMS is popular, but is limited to Java and also incurs somewhat more overhead than MQI.

It is important to note that each of these interfaces provides a mechanism to interact with MQSeries. Particular interfaces may be easier to use within certain environments or by developers with different backgrounds. However the messages constructed and sent by one kind of interface may be received by one or more other applications. Thus an AMI "C" Client program may well communicate with an MQI Cobol application.

3. Functional Overview
A set of MQSeries functions are provided with DB2 7.2 to allow SQL statements to include messaging operations. This means that this support is available to applications written in any supported language (C, Java, SQL, etc) using any of the database interfaces. The functions may be used either by a database client or by a stored procedure. To keep things simple, all examples shown below are in SQL. This SQL may be used from many programming languages in all the standard ways. All of the MQSeries messaging styles described above are supported.

In a basic configuration, as shown below, an MQSeries server is located on the database server machine along with DB2. The MQSeries functions are installed into DB2 and provide access to the MQSeries server. DB2 clients may be located on any machine accessible to the DB2 server.
Multiple clients can concurrently access the MQSeries functions through the database. Through the provided functions, DB2 clients may perform messaging operations within SQL statements. These messaging operations allow DB2 applications to communicate among themselves or with other MQSeries applications. This provides a simple way, for instance, for a DB2 application to publish database events to remote MQSeries applications, initiate a workflow through MQSeries Workflow or communicate with an application package such as SAP through MQSeries Integrator. Several gateways also exist that provide access to environments such as CICS® and IMS®.

MQSeries functionality is enabled in a DB2 database using the `enable_MQFunctions` command. This command configures DB2 for the MQSeries functions and establishes a simple default configuration that client applications may utilize with no further administrative action. The default configuration allows application programmers a quick way to get started and a simpler interface for development. Additional functionality may be configured incrementally as needed.

For instance, to send a simple message using the default configuration, the SQL statement would be:

**Example 1**

```
values MQSEND('simple message')
```

This will send the message `simple message` to the MQSeries queue manager and queue specified by the default configuration.

Please note that for simplicity, we do not show the full DB2 function name which is qualified by the schema name DB2MQ. To use the full function name in this example we would issue:

**Example 2**

```
values DB2MQ.MQSEND('simple message')
```

If you wish to use just the short function name in your SQL, you can define DB2MQ to be on your function path by issuing:

**Example 3**

```
set current function path = current function path, DB2MQ
```
The MQSeries functions provided with DB2 are based on the AMI MQSeries interface. AMI supports the use of an external configuration file, called the AMI Repository, to store configuration information. The default configuration includes an MQSeries AMI Repository configured for use with DB2.

Two key concepts in MQSeries AMI, service points and policies, are carried forward into the DB2 MQSeries functions. A service point is a logical end-point from which a message may be sent or received. In the AMI repository, each service point is defined with an MQSeries queue name and queue manager. Policies define the quality of service options that should be used for a given messaging operation. Key qualities of service include message priority and persistence. Default service points and policy definitions are provided and may be used by developers to further simplify their applications. Example 1 can be re-written as follows to explicitly specify the default service point and policy name as follows:

**Example 4**
```
values MQSEND('DB2.DEFAULT.SERVICE',
               'DB2.DEFAULT.POLICY',
               'simple message')
```

Queues may be serviced by one or more applications at the server upon which it resides. In many configurations, multiple queues will be defined to support different applications and purposes. For this reason, it is often important to define different service points when making MQSeries requests. This is demonstrated in the following example.

**Example 5**
```
Values MQSEND('ODS_Input',
              'simple message')
```

Here we are sending a message to a service point titled ODS_Input (ODS is a common abbreviation for Operational Data Store) rather than the default service point. Note that in this example, the policy is not specified and thus the default policy will be used.

### 3.1 Limitations
MQSeries provides an incredible wealth of features and options - not all of which are currently supported by this work. We have attempted to err on the side of simplicity rather than completeness. Similarly, MQSeries may be configured in a wide variety of ways. We have tested what we believe to be the most common. We welcome all feedback.

MQSeries provides the ability for message operations and database operations to be combined in a single unit of work as an atomic transaction. This feature is not initially supported by the MQSeries Functions on UNIX® and Windows. Plans are under consideration to support this feature in a future product release.

### 4. Usage Scenarios
The MQSeries Functions we describe may be used in a wide variety of scenarios. This section will review what we anticipate to be some of the more common scenarios. Those we will cover include
Basic Messaging, Application Connectivity and Data Publication. We will also use this section to provide examples of how the MQSeries DB2 Functions may be used, and common application configurations. Full details on the syntax of the MQSeries Functions may be found in the DB2 7.2 Release Notes.

4.1 Basic Messaging

The most basic form of messaging with the MQSeries DB2 Functions is when all database applications connect to the same DB2 server. This is demonstrated in the figure below where we have two DB2 client applications. Clients may be local to the database server or distributed in a network environment.

In the following simple scenario, Client A invokes the MQSEND function to send a user defined string to the default service location (step 1a). The MQSeries functions are then executed within DB2 on the database server. At some later time, Client B invokes the MQRECEIVE function to remove the message at the head of the queue defined by the default service and return it to the client (1b). Again, the MQSeries functions to perform this work are executed by DB2.

Database clients may use simple messaging in a number of ways. Among some common uses for messaging are:

* **Data Collection** - where information is received in the form of messages from one or more possibly diverse sources of information. Information sources may be commercial applications such as SAP or in-house developed applications. Such data may be received from queues and stored in database tables for further processing or analysis.

* **Workload distribution** - where work requests are posted to a queue shared by multiple instances of the same application. When an instance is ready to perform some work it receives a message from the top of the queue containing a work request to perform. Using this technique multiple instances can share the workload represented by a single queue of pooled requests.

* **Application signaling** - where several processes collaborate, messages are often used to coordinate their efforts. These messages may contain commands or requests for work to be performed. Typically, this kind of signaling is one-way; that is, the party that initiates the message does not expect a reply. Request/reply messaging is discussed in section 4.4.1.
**Application notification** - notification is similar to signaling in that data is sent from an initiator with no expectation of a response. Typically however, notification contains data about business events that have taken place. Publish/Subscribe, discussed in section 4.4.2 is a more advanced form of notification. The following scenario extends this basic scenario to incorporate remote messaging. That is, a message is sent between the machines Machine A and Machine B.

The sequence of steps is as follows:

1. The DB2 Client executes an MQSEND call, specifying a target service that has been defined to represent a remote queue on the Machine B.
2. The MQSeries DB2 functions perform the actual MQSeries work to send the message. The MQSeries Server on Machine A accepts the message and guarantees that it will deliver it to the destination defined by the Service Point definition and current MQSeries configuration of Machine A. The server determines that this is a queue on Machine B. It then attempts to deliver the message to the MQSeries Server on Machine B, transparently retrying as needed.
3. The MQSeries server on Machine B accepts the message from the server on Machine A and places it in the destination queue on Machine B.
4. An MQSeries client on Machine B requests the message at the head of the queue. [Note: MQSeries "client" is used here and throughout this document in the application sense of a program that uses the services of an MQSeries server. This is irrespective of whether the client application attaches to MQSeries using the MQSeries server or client bindings.]

As the above description shows, MQSeries transparently and reliably delivers messages between two or more MQSeries Servers. It is both easy to use and sophisticated enough to handle a wide range of configurations and requirements.

### 4.2 Sending Messages

Using MQSEND, a DB2 user or developer simply chooses what data to send, where to send it, and when it will be sent. In the industry, this is commonly called "Send and Forget" -- meaning that the sender just sends a message, relying on the guaranteed delivery protocols of MQSeries to ensure that the message reaches its destination. The following examples illustrate this. To send a user-defined string to the service point *myplace* with the policy *highPriority*:

**Example 6**

```sql
values MQSEND('myplace','highPriority','test')
```
Here, the policy highPriority refers to a policy defined in the AMI Repository that sets the MQSeries priority to the highest level and perhaps adjusts other qualities of service, such as persistence, as well. The message content may be composed of any legal combination of SQL and user specified data. This includes nested functions, operators, and casts. For instance, given a table EMPLOYEE, with varchar columns LASTNAME, FIRSTNAME, and DEPARTMENT, to send a message containing this information for each employee in DEPARTMENT 5LGA you would say:

**Example 7**

```sql
select MQSEND(LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT) from EMPLOYEE
where DEPARTMENT = '5LGA'
```

If this table also had an integer AGE column, we could include this as follows:

**Example 8**

```sql
select MQSEND(LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT || ' ' || char(AGE)) from EMPLOYEE
where DEPARTMENT = '5LGA'
```

Finally, the following example shows how message content may be derived using any valid SQL expression. Given a second table DEPT containing varchar columns DEPT_NO and DEPT_NAME, we can send messages containing employee LASTNAME and DEPT_NAME:

**Example 9**

```sql
select MQSEND(e.LASTNAME || ' ' || d.DEPTNAME)
from EMPLOYEE e, DEPT d
where e.DEPARTMENT = d.DEPTNAME
```

### 4.3 Retrieving Messages

The MQSeries DB2 Functions allow messages to be either received or read. The difference between reading and receiving is that reading returns the message at the head of a queue without removing it from the queue while receiving operations cause the message to be removed from the queue. So a message retrieved using a receive operation can only be retrieved once while a message retrieved using a read operation allows the same message to be retrieved many times. The following examples demonstrate this:

**Example 10**

```sql
values MQREAD()
```

This example returns a varchar string containing the message at the head of queue defined by the default service using the default quality of service policy. It is important to note that if no messages are available to be read, a null value will be returned. The queue is not changed by this operation.

**Example 11**

```sql
values MQRECEIVE('Employee_Changes')
```
The above example shows how a message can be removed from the head of the queue defined by the *Employee_Changes* service using the default policy.

One very powerful feature of DB2 is the ability to generate a table from a user-defined (or DB2 provided) function. We exploit this table function feature to allow the contents of a queue to be materialized as a DB2 table. The following example demonstrates the simplest form of this:

**Example 12**

```sql
select t.* from table (MQREADALL()) t
```

This query returns a table consisting of all of the messages in the queue defined by the default service and the metadata about these messages. While the full definition of the table structure returned is defined in the DB2 7.2 Release Notes, the first column reflects the contents of the message and the remaining columns contain the metadata. So to return just the messages, the example could be rewritten:

**Example 13**

```sql
select t.MSG from table (MQREADALL()) t
```

The table returned by a table function is no different from a table retrieved from the database directly. This means that we can use this table in a wide variety of ways. For instance we can join the contents of the table with another table or count the number of messages in a queue:

**Example 14**

```sql
select t.MSG, e.LASTNAME from table (MQREADALL()) t, EMPLOYEE e
where t.MSG = e.LASTNAME
```

**Example 15**

```sql
select count(*) from table (MQREADALL()) t
```

We can also hide the fact that the source of the table is a queue by creating a view over a table function. For instance, the following example creates a view called *NEW_EMP* over the queue referred to by the service named *NEW_EMPLOYEES*:

**Example 16**

```sql
create view NEW_EMP (msg) as
select t.msg from table (MQREADALL()) t
```

In this case the view is defined with only a single column containing an entire message. If messages are simply structured, for instance containing two fields of fixed length, it is straightforward to use the DB2 built-in functions to parse the message into the two columns. For instance, if we knew that messages sent to a particular queue always contained an 18 character last name followed by an 18 character first name, then we could define a view containing each field as a separate column as follows:
Example 17

create view NEW_EMP2 as
select left(t.msg,18) as LNAME, right(t.msg,18) as FNAME
from table(MQREADALL()) t

A more complete discussion of message parsing will be presented in a forthcoming white paper.

Finally, it is often desirable to store the contents of one or more messages in the database. This may be done using the full power of SQL to manipulate and store message content. Perhaps the simplest example of this is:

Example 18

insert into MESSAGES
select t.msg from table (MQRECEIVEALL()) t

Given a table MESSAGES, with a single column of varchar(2000), the statement above will insert the messages from the default service queue into the table. The general approach can be embellished to cover a wide variety of circumstances.

To summarize, in this section we have described how the MQSeries DB2 Functions can be used with the full power of SQL in a wide variety of ways. We have presented a basic set of building blocks by which messages can be sent and received from within SQL statements and where queues may be manipulated as database tables. Fundamentally, we have shown how basic messaging and database operations may be combined in a simple and natural manner.

4.4 Application to Application Connectivity

Application integration is a common element in many solutions. Whether integrating a purchased application into an existing infrastructure or just integrating a newly developed application into an existing environment, we are often faced with the task of gluing a heterogeneous collection of subsystems together to form a working whole. MQSeries is commonly viewed as an essential tool for integrating applications. Accessible in most hardware, software, and language environments, MQSeries provides the means to interconnect a very heterogeneous collection of applications.

In this section we will briefly discuss some application integration scenarios and how they may be used with DB2. As the topic is quite broad, a comprehensive treatment of Application Integration is beyond the scope of this work. Indeed, we will focus on just two simple topics: Request/Reply communications and a brief discussion on MQSeries Integrator and publish/subscribe.

4.4.1 Request/Reply Communications

The Request/Reply (R/R) communications method is a very common technique for one application to request the services of another. We have already shown that one way to do this is for the requester to send a message to the service provider requesting some work to be performed. Once the work has been completed, the provider may decide to send results (or just a confirmation of completion) back to the requestor. But using the basic messaging techniques described above, there is nothing that connects the sender's request with the service provider's response. Unless the requester waits for a reply before continuing, some mechanism must be used to associate each reply with its request. Rather than force the developer to create such a mechanism,
MQSeries provides a **correlation-id** that allows us to correlate one group of messages with another. While there are a number of ways in which this mechanism could be used, the simplest is for the requestor to mark a message with a known correlation identifier using, for instance, the following:

**Example 19**

```
MQSEND ('myRequester', 'myPolicy','SendStatus:cust1','Req1')
```

This statement adds a final parameter *Req1* to the MQSEND statement from above to indicate the correlation id for the request. Now to receive a reply to this specific request, we can use the corresponding MQRECEIVE statement to selectively retrieve the first message defined by the indicated service that matches this correlation id as follows:

**Example 20**

```
MQRECEIVE('myReceiver','myPolicy','Req1')
```

If the application servicing the request is busy and the requestor issues the above MQRECEIVE before the reply is sent, then no messages matching this correlation id will be found.

The diagram above illustrates this in more detail. In this diagram we show processing by two applications. One, the Service Requester and the other a Service Provider. A request message (M1) is initiated by the requester and sent to the queue described by *aServer*. When the service provider is ready to perform new work it receives the request (M1) and performs the requested service. To receive both the service request and the **correlation-id**, a statement like the following is used:

**Example 21**

```
Select msg, correlid from table (MQRECEIVEALL('aServer','myPolicy',1)) t
```

This returns the message and correlation identifier of the first request from the service *aServer*.

Once the service has been performed, it sends the reply message (M2) to the queue described by *aRequester*. Meanwhile, the service requestor could have been doing other work. In fact, there is
no guarantee that the initial service request will be responded to within a set time...application level timeouts such as this must be managed by the developer. Indeed, the requester must poll to detect the presence of the reply.

The advantage of such time-independent asynchronous processing is that the requester and service provider execute completely independently of one another. This can be used both to accommodate environments in which applications are only intermittently connected and more batch-oriented environments in which multiple requests or replies are aggregated before processing. This kind of aggregation is often used in data warehouse environments to periodically update a data warehouse or operational data store.

4.4.2 Publish/Subscribe

In this section we will describe implementing publish/subscribe (pub/sub) techniques with the DB2 MQseries functions. Pub/sub may be implemented in multiple ways, from simple distribution lists to full pub/sub with MQseries Integrator. In addition, the act of publishing messages could be performed either under explicit program control or through automated definitions. In this section we will discuss each of these techniques.

Simple Data Publication

Another common scenario in application integration is for one application to notify other applications about events of interest. This is easily done by sending a message to a queue monitored by another application. The contents of the message may be either a user defined string or may be composed from database columns. Often a simple message is all that needs to be sent using the MQSEND function. When such messages need to be sent concurrently to multiple recipients, the Distribution List facility of the MQSeries AMI may be used.

A distribution list is defined using the AMI Administration tool (NOTE: also by directly editing the AMT.XML file). A distribution list comprises a list of individual services. A message sent to a distribution list is forwarded to every service defined within the list. This is especially useful when it is known that a few services will always be interested in every message. The following example illustrates the sending of a message to the distribution list interestedParties:

Example 22

MQSEND('interestedParties', 'information of general interest');
When more control over which services should receive what messages is required, a publish/subscribe capability is needed. Publish/Subscribe systems typically provide a scalable, secure environment in which many subscribers can register to receive messages from multiple publishers. To support this capability the MQPublish interface can be used, in conjunction with MQSeries Integrator or the simple MQSeries Publish/Subscribe facility.

MQPublish allows users to optionally specify a topic to be associated with a message. Topics allow a subscriber to more clearly specify the messages to be accepted. The diagram above shows the flow of steps using MQSeries Integrator. The sequence of steps is as follows:

1. An MQSeries administrator configures MQSeries Integrator publish/subscribe capabilities.
2. Interested applications subscribe to subscription points defined by the MQSI configuration, optionally specifying topics of interest to them. Each subscriber may select relevant topics and, in addition may also utilize the content-based subscription techniques of MQSeries Integrator V2. [NOTE: Please see the MQSeries Integrator V2 manuals for details.] It is important to note that queues, as represented by service names, define the subscriber.
3. A DB2 application publishes a message to the service point Weather. Indicated that the weather is sleet with a topic of Austin, thus notifying interested subscribers that the weather in Austin is Sleet.
4. The mechanics of actually publishing the message are handled by the MQSeries functions provided by DB2. The message is sent to MQSeries Integrator using the service named Weather.
5. MQSI accepts the message from the Weather service, performs any processing defined by the MQSI configuration, and determines which subscriptions it satisfies. MQSI then forwards the message to the subscriber queues whose criteria it meets.
6. Applications that have subscribed to the Weather service, and registered an interest in Austin, will receive the message 'Sleet' in their receiving service.

Moving back to our employee example, if we wanted to publish the name, department and age for all employees in the department "5LGA", we would issue the following. This data uses the default publication service and policy and a null topic.

Example 23

```
select MQPUBLISH(LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT|| ' ' ||char(AGE)) from EMPLOYEE
where DEPARTMENT = '5LGA'
```

Fully specifying all the parameters and simplifying the message to contain only the LASTNAME the statement would look like:

Example 24

```
select MQPUBLISH('HR_INFO_PUB', 'SPECIAL_POLICY', 'MANAGER', 'ALL_EMP:5LGA', LASTNAME)
from EMPLOYEE where DEPARTMENT = '5LGA'
```

This statement publishes messages to the HR_INFO_PUB publication service using SPECIAL_POLICY indicated that the sender is the MANAGER service. The topic string
demonstrates that multiple topics, concatenated using a ':' may be specified. In this example, the use of two topics allows subscribers to register for either ALL_EMP or just 5LGA to receive these messages.

To receive published messages, you must first register your interest in messages containing a given topic and indicate the name of the subscriber service that messages should be sent to. It is important to note that an AMI subscriber service defines a **Broker Service** and a **Receiver Service**. The broker service is how the subscriber communicates with the publish/subscribe broker and the receiver service is where messages matching our subscription request will be sent to. The following statement registers an interest in the topic **ALL_EMP**.

**Example 25**

MQSUBSCRIBE('aSubscriber', 'ALL_EMP')

Once an application has subscribed, messages published with the topic **ALL_EMP** will be forwarded to the receiver service defined by the subscriber service. An application may have multiple concurrent subscriptions. To obtain the messages that meet your subscription any of the standard message retrieval functions can be used. For instance, if the subscriber service *aSubscriber* defines the receiver service to be *aSubscriberReceiver* then the following statement will non-destructively read the first message:

**Example 26**

MQREAD('aSubscriberReceiver')

To find out both the messages and the topics that they were published under, one would use one of the table functions. The following statement would receive the first 5 messages from *aSubscriberReceiver* and display both the message and the topic:

**Example 27**

Select t.msg, t.topic from table (MQRECEIVEALL('aSubscriberReceiver',5)) t

To read all of the messages with the topic **ALL_EMP**, we could leverage the power of SQL to issue:

**Example 28**

Select t.msg from table (MQREADALL('aSubscriberReceiver')) t where t.topic = 'ALL_EMP'

**NOTE:** It is important to realize that if we had used MQRECEIVEALL with a constraint then the entire queue would have been consumed, not just those messages published with topic **ALL_EMP**. This is because the table function is performed before the constraint is applied!

When we are no longer interested in subscribing to a particular topic we must explicitly unsubscribe using a statement such as:
Example 29:

MQUNSUBSCRIBE('aSubscriber', 'ALL_EMP')

Once this statement is issued, the publish/subscribe broker will no longer deliver messages matching this subscription.

Automated Publication

Another important technique in database messaging is automated publication. Using the trigger facility within DB2, one can automatically publish messages as part of a trigger invocation. While other techniques exist for automated data publication, the trigger based approach allows administrators or developers great freedom in constructing the message content and flexibility in defining the trigger actions. As with any use of triggers, attention must be paid to the frequency and cost of execution. The following examples demonstrate how triggers may be used with the MQSeries DB2 Functions.

The example below shows how easy it is to publish a message each time a new employee is hired. Any users or applications subscribing to the HR_INFO_PUB service with a registered interest in NEW_EMP will receive a message containing the date, name and department of each new employee.

Example 30

CREATE TRIGGER new_employee AFTER INSERT ON employee
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
VALUES MQPUBLISH('HR_INFO_PUB', 'NEW_EMP', current date || ' ' || LASTNAME || ' ' || DEPARTMENT)

You can also use the trigger facility to perform very simple data replication in a heterogeneous environment. While not a replacement for the power and sophistication of DB2 DataPropagator®, this technique may be useful in cases where MQSeries Integrator is used to transform the content of incoming messages and potentially to store the transformed data in one or more databases.

5. Summary

In this paper we have demonstrated how the combination of DB2 and MQSeries can be brought together to solve a wide variety of development needs. The new DB2 Messaging functions are easy to use and provide a powerful repertoire of capabilities and functions. From basic datagrams to more sophisticated publish/subscribe messaging, these functions can be used to extend the reach of your database applications, simplify application integration, and effectively share data across a heterogeneous set of platforms.