

> PROGRESS<sup>®</sup>  
DATADIRECT  
XQUERY<sup>®</sup> AND  
THE INSURANCE  
INDUSTRY

---

## TABLE OF CONTENTS

<b>Preface</b> . . . . .	1
What Is DataDirect XQuery . . . . .	1
About the ACORD Messages in This Document. . . . .	1
Typographical Conventions. . . . .	2
<b>Relational Insurance Model Using ACORD Standards.</b> . . . . .	4
Simple Database . . . . .	4
Downloading the Database Tables. . . . .	5
SQL Scripts . . . . .	5
Example Operations . . . . .	5
<b>Request/Response Example</b> . . . . .	6
<b>Performing a Search Using Multiple Joins</b> . . . . .	10
<b>Validating Update Requests.</b> . . . . .	13
<b>Updating a Relational Database</b> . . . . .	15
<b>Managing Complex Update Requests</b> . . . . .	16
<b>XQuery and EDI.</b> . . . . .	18
Sample X12 Message . . . . .	18
<b>Using XQuery to Create PDF</b> . . . . .	20
Try It Yourself. . . . .	20
<b>Binding External Variables</b> . . . . .	20
Introduction to DataDirect XQuery® and XQJ. . . . .	21
Sample Java Code. . . . .	21
Using DataDirectXQuery and XQJ with EJB. . . . .	22
Sample Code. . . . .	22
Comparing Solutions: Using DataDirect XQuery versus SQL and DOM . . . . .	23
Processing an ACORD Message. . . . .	23
Processing an ACORD Party Inquiry. . . . .	24
<b>Installing and Running Examples.</b> . . . . .	25
Preparing the Database . . . . .	25
Running the XQuery Files. . . . .	26
About the Java Files . . . . .	27

---

## PREFACE

ACORD (Association for Cooperative Operations Research and Development, <http://www.acord.org>) is a global, nonprofit insurance association whose mission is to facilitate the development and use of standards for the insurance, reinsurance, and related financial services industries. ACORD standards are used by hundreds of insurance and reinsurance companies and by thousands of agents and brokers, related financial services organizations, software providers, and industry organizations worldwide.

This paper presents numerous examples that show you how you can take advantage of Progress® DataDirect XQuery® in your efforts to adhere to and comply with ACORD standards.

### *WHAT IS DATADIRECT XQUERY*

DataDirect XQuery is an implementation of XQuery that can query XML, relational data, SOAP messages, EDI, or a combination of data sources. DataDirect XQuery provides the fastest, most reliable, and most scalable XQuery support for all major relational databases, and it runs on any Java platform. DataDirect XQuery supports the XQuery for Java API (XQJ) and is easily embeddable in any Java program. It does not require any other product or application server, and it has no server of its own. DataDirect XQuery is recommended for applications that must manage XML, relational, and legacy data formats, including applications for data integration, Web services, Web publishing, and report generation.

### *ABOUT THE ACORD MESSAGES IN THIS DOCUMENT*

The ACORD XML standards for Life and Annuity are defined through a set of fairly extensive XML Schema, and companies dealing with ACORD often need to create and interpret messages defined according to those XML Schema. In particular, in the example introduced here and described in this document, we focus on `txLife` (Life Business Message Specification) and XMLife (Life Data Model Specification) messages.

Concepts similar to those used in this example can be adapted for different ACORD standards or even for different industry sectors.

```
<TXLife>
  <UserAuthRequest>...</UserAuthRequest>
<TXLifeRequest>...</TXLifeRequest>
< /TXLife>
```

or

```
<TXLife>
  <UserAuthResponse>...</UserAuthResponse>
  <TXLifeResponse>...</TXLifeResponse>
  <TXLifeNotify>...</TXLifeNotify>
</TXLife>
```

As long as the request message is involved, the core information is contained in the `<TXLifeRequest>` element, where the following values are specified:

- > `TransRefGUID`—transaction request identifier
- > `TransType`—transaction code that determines action (Phone Change Request tc=182, for example)
- > `oLife`—object(s) required to process the transaction

More details are available on the ACORD Web site (<http://www.acord.org>); you can also download documentation and related XML Schemas from the same location.

## **TYPOGRAPHICAL CONVENTIONS**

This document uses the following typographical conventions:

<b>Convention</b>	<b>Explanation</b>
<i>italics</i>	Introduces new terms with which you may not be familiar and is used occasionally for emphasis.

---

<b>bold</b>	Emphasizes important information. Also indicates button, menu, and icon names on which you can act. For example, click <b>Next</b> .
UPPERCASE	Indicates keys or key combinations that you can use. For example, press the ENTER key.
monospace	Indicates syntax examples, values that you specify, or results that you receive.
<i>monospaced italics</i>	Indicates names that are placeholders for values that you specify. For example, <i>filename</i> .
forward slash /	Separates menus and their associated commands. For example, Select File / Copy means that you should select Copy from the File menu.  The slash also separates directory levels when specifying locations under UNIX.
vertical rule	Indicates an “OR” separator used to delineate items.
brackets [ ]	Indicates optional items. For example, in the statement – SELECT [DISTINCT], DISTINCT is an optional keyword.  Also indicates sections of the Windows Registry.

braces { }

Indicates that you must select one item. For example, {yes | no} means that you must specify either yes or no.

ellipsis . . .

Indicates that the immediately preceding item can be repeated any number of times in succession. An ellipsis following a closing bracket indicates that all information in that unit can be repeated.

## RELATIONAL INSURANCE MODEL USING ACORD STANDARDS

This chapter introduces the relational model that serves as the data repository for the XQuery examples. Take a look at it before proceeding to the examples described in following chapters to gain some insight into the different tables and their relationships.

### SIMPLE DATABASE

The following illustration depicts an oversimplified model of a relational database.

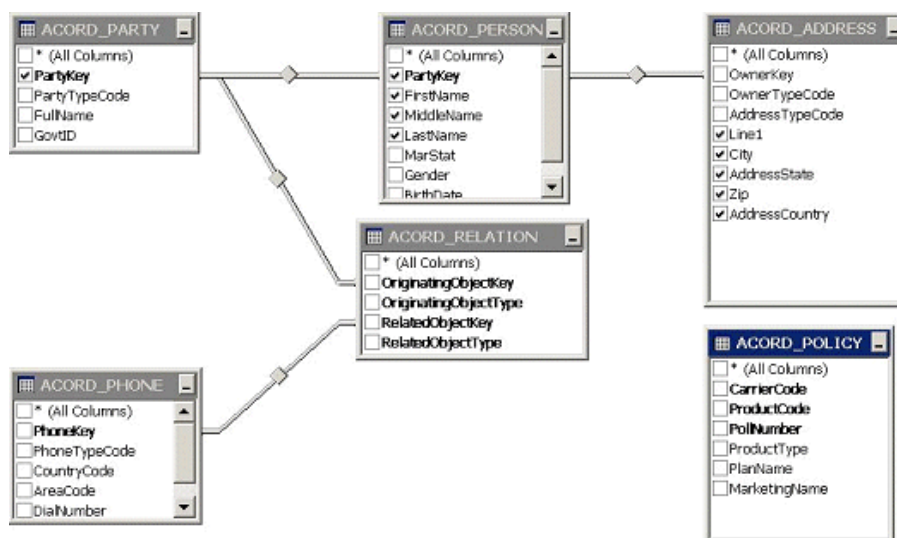


Figure 1-1. Simple Insurance Relational Database

---

Though over-simplified, it is complete enough to show the set of operations on ACORD messages that are at the core of our example. It consists of six tables:

- > ACORD\_PARTY
- > ACORD\_PERSON
- > ACORD\_ADDRESS
- > ACORD\_PHONE
- > ACORD\_POLICY
- > ACORD\_RELATION

### ***DOWNLOADING THE DATABASE TABLES***

You can create the set of tables shown in Figure 1-1 on your own system, which will allow you to run the examples on your own machine.

### ***SQL Scripts***

Two SQL scripts make it easy to create and populate the tables shown in Figure 1-1 in your database:

- > ACORD\_create.sql—creates and populates the tables described in Relational Insurance Model Using ACORD Standards. To execute this script, just open it in SQL Server Management Studio, and run it.
- > ACORD\_drop.sql—drops the tables added by ACORD\_create.sql from your database.

Both scripts assume that you want to create/remove the tables in the pubs database; you can easily change that default at the beginning of the script files.

### ***EXAMPLE OPERATIONS***

The subsequent chapters in this book demonstrate several operations that you might want to perform using XQuery

- > Message processing
- > Business validation

- > Updating data in a single table
- > Updating data in multiple tables
- > Integrating XML processing with EDI messages
- > Report generation

The XQuery used to perform these operations is structured to rely on a library of functions that implements most of the logic about how to deal with ACORD messages and how to query the relational model described in the section "Simple Database" on page 11. You can find the XQuery library here:

[http://www.xquery.com/ACORD/code\\_samples/acord\\_verbs.xquery](http://www.xquery.com/ACORD/code_samples/acord_verbs.xquery).

Each example is implemented by a specific XQuery that leverages the functions implemented in the `acord_verbs.xquery` imported module.

## REQUEST/RESPONSE EXAMPLE

Imagine we need to process a message requesting information about a specific policy product (message `tc=201`, as defined by ACORD standards). The ACORD message looks like this:

```
<tx:TXLife xmlns:tx="http://ACORD.org/Standards/Life/2">
  <tx:UserAuthRequest>
    <tx:UserLoginName>username</tx:UserLoginName>
    <tx:UserPswd>
      <tx:Pswd>password</tx:Pswd>
    </tx:UserPswd>
  </tx:UserAuthRequest>

  <tx:TXLifeRequest>
    <tx:TransRefGUID>2fac30d5-cd66-4eaa-833f9f4989db796b
    </tx:TransRefGUID>
    <tx:TransType tc="201">Policy Product
      Inquiry</tx:TransType>
    <tx:TransExeDate>2007-06-01</tx:TransExeDate>
    <tx:TransExeTime>17:00:13-05:00</tx:TransExeTime>
    <tx:InquiryLevel tc="1">OLI_INQUIRY_OBJ</
      tx:InquiryLevel>

    <tx:OLife Version="2.14.00">
      <tx:PolicyProduct id="PolicyProduct_1">
        <tx:CarrierCode>1002</tx:CarrierCode>
```

```

    <tx:ProductCode>VWL01</tx:ProductCode>
  </tx:PolicyProduct>
</tx:OLife>
</tx:TXLifeRequest>
</tx:TXLife>

```

You can see how the message is a Policy Product Inquiry concerning the PolicyProduct provided by carrier "1002" using the product code "VWL01."

The XQuery code is as simple as this:

```

declare namespace tx = "http://ACORD.org/Standards/Life/2";
import module namespace example =
"http://www.datadirect.com/xquery/examples" at
"acord_verbs.xquery";

<tx:TXLife> {
  for $request in doc("201-1.xml")/tx:TXLife/
  tx:TXLifeRequest return example:reply-201($request)
}
</tx:TXLife>

```

The example:reply-201 function analyzes the TXLifeRequest content and fetches the relevant information from the relational repository:

```

declare function example:reply-201 ($request as
element(tx:TXLifeRequest)) as element(tx:TXLifeResponse)? {
  let $results :=
    for $policyProduct at $pos in
      $request/tx:OLife/tx:PolicyProduct
    let $carrierCode :=
      xs:string($policyProduct/tx:CarrierCode)
    let $productCode :=
      xs:string($policyProduct/tx:ProductCode)
    for $policy in collection("ACORD_POLICY")/ACORD_POLICY
      [CarrierCode eq $carrierCode]
      [ProductCode eq $productCode]
    return <tx:PolicyProduct> {
      attribute id {fn:concat("PolicyProduct_",
        xs:string($pos))},
      $carrierCode,
      <tx:PlanName>{xs:string($policy/PlanName)}
      </tx:PlanName>,
      $productCode,

```

```

        <tx:MarketingName>{xs:string($policy/MarketingName)}
      </tx:MarketingName>
    } </tx:PolicyProduct>
  }
return
  example:create-response(
    $request,
    (<tx:TransResult>
      <tx:ResultCode tc=
        "1">Success</tx:ResultCode>
    </tx:TransResult>,
    <tx:OLife Version="2.12.00">
      {$results}
    </tx:OLife>
  )
);

```

Note how the relational repository is accessed by the XQuery function:

```

for $policy in
collection ("ACORD_POLICY")/ACORD_POLICY[CarrierCode eq
$carrierCode][ProductCode eq $productCode]

```

The first "for" instruction iterates over each PolicyProduct element from the request (there could be more than one); then, for each PolicyProduct the carrier and product code are used as keys to retrieve the information from the database.

The response is formatted according to the required ACORD structure:

```

(: Creates a TXLife message with non-empty TXLifeResponse :)
declare function example:create-response(
  $request as element(tx:TXLifeRequest),
  $payload as element(*) as
  element(tx:TXLifeResponse) {
    <tx:TXLifeResponse> {
      $request/tx:TransRefGUID,
      $request/tx:TransType,
      <tx:TransExeDate>{example:formatDate
        (fn:current-date())}</tx:TransExeDate>,
      <tx:TransExeTime>{example:formatTime
        (fn:current-time())}</tx:TransExeTime>,
      $payload
    }
  }
  </tx:TXLifeResponse>
);

```

From the XQuery author's point of view, the relational model is navigated as if it was an XML structure; under the covers, DataDirect XQuery will issue the proper SQL statements to fetch the entries in the ACORD\_POLICY table that match carrier and product codes.

A similar example is a "party search" (tc=301); in this case the message we receive is this one:

```
<tx:TXLife xmlns:tx="http://ACORD.org/Standards/Life/2">
  <tx:UserAuthRequest>
    <tx:UserLoginName>username</tx:UserLoginName>
    <tx:UserPswd>
      <tx:Pswd>password</tx:Pswd>
    </tx:UserPswd>
  </tx:UserAuthRequest>

  <tx:TXLifeRequest>

<tx:TransRefGUID>e6c00b0e-94d4-4e48-8857-1690945df1c4</
tx:TransRef GUID>
  <tx:TransType tc="301">Party Search</tx:TransType>
  <tx:TransExeDate>2007-06-01</tx:TransExeDate>
  <tx:TransExeTime>17:00:13-05:00</tx:TransExeTime>
  <tx:CriteriaExpression>
    <tx:CriteriaOperator tc="2">AND</tx:CriteriaOperator>
  <tx:Criteria>
    <tx:ObjectType tc="115">Person</tx:ObjectType>
    <tx:PropertyName>LastName</tx:PropertyName>
    <tx:PropertyValue tc=
      "+1">Roberts</tx:PropertyValue>
    <tx:Operation tc="1">Equal</tx:Operation>
  </tx:Criteria>
  <tx:Criteria>
    <tx:ObjectType tc="115">Person</tx:ObjectType>
    <tx:PropertyName>FirstName</tx:PropertyName>
    <tx:PropertyValue>Edward</tx:PropertyValue>
    <tx:Operation tc="1">Equal</tx:Operation>
  </tx:Criteria>
</tx:CriteriaExpression>

  <tx:OLife Version="2.14.00">
  </tx:OLife>
</tx:TXLifeRequest>
</tx:TXLife>
```

The search criteria could be more complicated than what is illustrated here (see 301-5.xml for example). The XQuery processing this message can be something like this (click here for the XQuery document):

```
import module namespace example = "http://www.datadirect.com/
xquery/examples" at "acord_verbs.xquery";

declare namespace tx="http://ACORD.org/Standards/Life/2";

declare variable $request as document-node(element(*,
xs:untyped)) external;

<tx:TXLife> {
  for $r in $request/tx:TXLife/tx:TXLifeRequest
  return example:reply-301($r)
}
</tx:TXLife>
```

You can browse `acord_verbs.xquery` to find details about how the request is processed. Note that in this case, to simulate a more realistic scenario, we are not processing the request message as a file (accessed through the `fn:doc()` function), but the request is bound externally to the `$request` variable. This means that the Java code invoking the XQuery is responsible for binding the proper ACORD request message to the variable before executing the XQuery; see Chapter 9, "Binding External Variables" for details about how that can be done.

## PERFORMING A SEARCH USING MULTIPLE JOINS

In this chapter, we consider a more complicated scenario, one in which we receive a message that contains a party inquiry, but one that also requests information about related objects (tc=204 with InquiryLevel set to 2). A "party" in ACORD terms refers to a generic entity that can be associated to people, organizations, or trusts; information like details about person/organization, address, phone and more are all related to a party.

In our case, the party inquiry message request looks like this:

```
<tx:TXLife xmlns:tx="http://ACORD.org/Standards/Life/2">
  <tx:UserAuthRequest>
    <tx:UserLoginName>username</tx:UserLoginName>
    <tx:UserPswd>
      <tx:Pswd>password</tx:Pswd>
```

```

    </tx:UserPswd>
  </tx:UserAuthRequest>

  <tx:TXLifeRequest>

<tx:TransRefGUID>f2965a64-903b-4616-aa9c-afa6ed04e1b0</
tx:TransRefGUID>
  <tx:TransType tc="204">Party Inquiry</tx:TransType>
  <tx:TransExeDate>2007-06-01</tx:TransExeDate>
  <tx:TransExeTime>17:00:13-05:00</tx:TransExeTime>
  <tx:InquiryLevel tc=
    "2">OLI_INQUIRY_OBJREL</tx:InquiryLevel>

  <tx:OLife Version="2.14.00">
    <tx:Party id="Party_1">
      <tx:PartyTypeCode tc=
        "1">Person</tx:PartyTypeCode>
      <tx:GovtID>222222222</tx:GovtID>
      <tx:Person>
        <tx:FirstName>John</tx:FirstName>
        <tx:LastName>Smith</tx:LastName>
      </tx:Person>
    </tx:Party>
  </tx:OLife>
</tx:TXLifeRequest>
</tx:TXLife>

```

The XQuery handling this request looks very similar to XQuery used for handling what we have seen before:

```

declare namespace tx = "http://ACORD.org/Standards/Life/2";
import module namespace example =
  "http://www.datadirect.com/xquery/examples" at
  "acord_verbs.xquery";

declare variable $request as document-node(element(*,
xs:untyped)) external;

<tx:TXLife> {
  for $r in $request/tx:TXLife/tx:TXLifeRequest
  return example:reply-204($r)
}
</tx:TXLife>

```

This time the code responsible for computing the response needs to join data from two tables in the relational database; we use the GovtID

information in the message to select the Party and then join with Person based on PartyKey:

```

declare function example:reply-204($request as
element(tx:TXLifeRequest)) as element(tx:TXLifeResponse)? {
  let $results :=
    for $p in $request/tx:OLife/tx:Party
    let $govtID := $p/tx:GovtID
    (: Party inquiry on GovtID :)
    for $party in collection("ACORD_PARTY")/ACORD_PARTY
      [GovtID eq $govtID]
    let $partyKey := $party/PartyKey
    (:
      Example of 1-1 relation.
      ACORD_PERSON.PartyKey must match with
      ACORD_PARTY.PartyKey
    :)
    for $person in collection("ACORD_PERSON")/ACORD_PERSON
      [PartyKey eq $partyKey]
    return example:create-person($party, $person, $request
      /tx:InquiryLevel/@tc)

  return
    example:create-response( $request,
      (
        <tx:TransResult>
          <tx:ResultCode tc="1">Success</tx:ResultCode>
        </tx:TransResult>,
        <tx:OLife Version="2.12.00">
          {$results}
        </tx:OLife>
      )
    )
};

```

Once again, note how the XQuery author is focusing only on navigating the various data sources as if they all were native XML structures. The underlying DataDirect XQuery engine handles the part of code that deals with relational data by issuing the proper SELECT statements to resolve the join condition to the relational database.

## VALIDATING UPDATE REQUESTS

Suppose now our system receives a message containing an address change request (tc=181 in ACORD parlance). Before even processing it, however, we want to make sure the message is semantically valid (not just valid in terms of the ACORD standards); in particular, we want to ensure that:

- > The person exists.
- > The old address exists.
- > The new ZIP code is valid.
- > The new ZIP code corresponds to the new state.

The XQuery implementing such validation is shown here:

```
declare namespace tx="http://ACORD.org/Standards/Life/2";

import module namespace example =
  "http://www.datadirect.com/xquery/examples" at
  "acord_verbs.xquery";

(: Example of business validation on an "Address Change"
request
:)
let $request := doc("181-1.xml")
return example:validate-181($request/tx:TXLife/tx:TXLifeRequest)
```

As you can see in `acord_verbs.xquery`, `example:validate-181()` fetches data from two relational tables (ACORD\_PERSON and ACORD\_ADDRESS), performs the various checks, and returns either an empty result (no errors; success), or a message describing the error condition (failure). (The `acord_verbs.xquery` file is part of an `examples.zip` file, which you can download from DataDirect. See Chapter 10, "Installing and Running Examples" for more information.)

The first part of the function is particularly interesting, as it is responsible for implementing the actual business validation rules:

```
declare function example:validate-181($request as
element(tx:TXLifeRequest)) as element(tx:TXLifeResponse)? {
  let $party := $request/tx:OLife/tx:Party[@id eq
$request/@PrimaryObjectID]
  let $key := xs:string($party/tx:PartyKey)
```

```

    let $person :=
collection("ACORD_PERSON")/ACORD_PERSON[PartyKey eq $key]
    let $oldAddress := $party/tx:Address[@DataRep eq "Removed"]
    let $newAddress := $party/tx:Address[@DataRep eq "Full"]
    let $row := collection("ACORD_ADDRESS")/ACORD_ADDRESS
        [OwnerKey eq $key]
        [AddressTypeCode eq
xs:integer($oldAddress/tx:AddressTypeCode/@tc)]
        [Line1 eq $oldAddress/tx:Line1]
        [City eq $oldAddress/tx:City]
        (: AddressState is nullable :)
        [AddressState eq $oldAddress/tx:AddressState
or
fn:empty(AddressState) and
fn:empty($oldAddress/tx:AddressState)]
        [Zip eq $oldAddress/tx:Zip]
        [AddressCountry eq $oldAddress/tx:AddressCountry]
    let $rowValid := fn:count($row) eq 1
    let $zipState :=
        if ( $newAddress/tx:AddressCountry eq "USA" )
        then
            example:getZipInfo($newAddress/tx:Zip)
        else
            ()
...

```

Validation of the ZIP code, when "AddressCountry" is "USA," is delegated to the `example:getZipInfo()` function, which, in turn delegates validation to a third-party public Web Service, published by <http://www.webservicesmart.com>.

```

declare function example:getZipInfo($zipCode as xs:string) as
element()? {
    let $response :=
        ddtek:wscall(
            <ddtek:location address=
"http://www.webservicesmart.com/uszip.asmx"
            soapaction=
"http://webservicesmart.com/ws/ValidateZip"/>,
            <ws:ValidateZip>
                <ws:ZipCode>{$zipCode}</ws:ZipCode>
            </ws:ValidateZip>
        )/ws:ValidateZipResponse/ws:ValidateZipResult
    return

```

```

    (: because the webservice returns the result as text
    (not xml) wrapped in a ValidateZipResult node we have to
    parse the content :)
    ddtek:parse($response/text()/*
};

```

Thanks to the `ddtek:wscall()` function, invoking an external Web Service from a DataDirect XQuery is extremely simple, as you can see.

## UPDATING A RELATIONAL DATABASE

Once we know that the address change request message (tc= 181) is valid (see Chapter 4, “Validating Update Requests”), we need to process that request and commit any changes to our relational database. DataDirect XQuery has the ability to perform updates against relational data (<http://www.xquery.com/examples/updates/>). The XQuery implementing that update operation is shown here:

```

declare namespace tx="http://ACORD.org/Standards/Life/2";

import module namespace example =
"http://www.datadirect.com/xquery/examples" at
"acord_verbs.xquery";

declare variable $request as document-node(element(*,
xs:untyped)) external;

(: Address Change :)
example:update-181($request/tx:TXLife/tx:TXLifeRequest)

```

The `example:update-181()` function is fairly simple, and it relies on the `ddtek:sql-update()` function that allows rows in relational tables to be modified directly from your XQuery:

```

declare function example:validate-181($request as
    element(tx:TXLifeRequest)) {
    let $party := $request/tx:OLife/tx:Party[@id eq
        $request/@PrimaryObjectID]
    let $key := xs:string($party/tx:PartyKey)
    let $oldAddress := $party/tx:Address[@DataRep eq "Removed"]
    let $newAddress := $party/tx:Address[@DataRep eq "Full"]
    let $row := collection("ACORD_ADDRESS")/ACORD_ADDRESS
        [OwnerKey eq $key]
        [AddressTypeCode eq

```

```

        xs:integer($oldAddress/tx:AddressTypeCode/@tc)]
    [Line1 eq $oldAddress/tx:Line1]
    [City eq $oldAddress/tx:City]
    (: AddressState is nullable :)
    [AddressState eq $oldAddress/tx:AddressState
    or
    fn:empty(AddressState) and
    fn:empty($oldAddress/tx:AddressState)]
    [Zip eq $oldAddress/tx:Zip]
    [AddressCountry eq $oldAddress/tx:AddressCountry]
return ddtek:sql-update( $row,
    "AddressTypeCode",
    xs:integer($newAddress/tx:AddressTypeCode/@tc),
    "Line1",
    $newAddress/tx:Line1,
    "AddressState",
    $newAddress/tx:AddressState,
    "Zip", $newAddress/tx:Zip,
    "AddressCountry",
    $newAddress/tx:AddressCountry
)
};

```

The next chapter describes how to use DataDirect XQuery to manage more complex update scenarios involving updates to multiple tables.

## MANAGING COMPLEX UPDATE REQUESTS

A more complicated update scenario (compared to the simple one described in *Updating a Relational Database*) might be one in which we receive a party update message (tc=186). Party update messages of this type allow for the possibility of inserting or deleting rows in multiple tables.

The sample XQuery file `acord_186-update.xquery` processes a party update message in XML format (available by [clicking here](#)) twice—inserting new entries the first time and deleting them the second time. (These files are part of an `examples.zip` file, which you can download from DataDirect. See Chapter 10, “Installing and Running Examples” for more information.)

```
declare namespace tx="http://ACORD.org/Standards/Life/2";
```

```

import module namespace example =
"http://www.datadirect.com/xquery/examples" at
"acord_verbs.xquery";

(:Party Update Transaction (delete + insert) :)
for $request in doc("186-5.xml")/tx:TXLife/tx:TXLifeRequest
return (
  example:delete-186($request),
  example:insert-186($request)
)

```

As the update message is dealing with "Party" information, multiple tables are affected when deleting or inserting entries; for example, this is the function that takes care of deleting all the information related to a specific Party stored in the database:

```

declare updating function example:delete-186($request as
element(tx:TXLifeRequest)) {
  if (
fn:not(fn:empty($request/tx:OLife/tx:Party/tx:Organization)))
  (: update of organization not supported in example :)
  then fn:error((), "update organization not supported")
  else
  (
  (: look up existing Party based on GovtID :)
  for $partyKey in
    collection("ACORD_PARTY")/ACORD_PARTY[GovtID eq
    $request/tx:OLife/tx:Party/tx:GovtID]/PartyKey
  return
  (
  (: Example delete of Party with 1 - 1 relation {Party ->
  Person}:)
  ddtek:sql-delete(collection("ACORD_PARTY")/ACORD_PARTY
  [PartyKey eq $partyKey] ),
  ddtek:sql-delete(collection("ACORD_PERSON")/ACORD_PERSON
  [PartyKey eq $partyKey] ),

  (: Example deleting an 1 - n relation {Party ->
  Address} :)
  ddtek:sql-delete(collection("ACORD_ADDRESS")/
  ACORD_ADDRESS[OwnerKey eq $partyKey] ),

  (: Example deleting an n - n relation {Party <-> Phone} :)
  for $relation in
  collection("ACORD_RELATION")/ACORD_RELATION
  [OriginatingObjectKey eq $partyKey]

```

```

    [OriginatingObjectType eq $example:OLI_PARTY]
    [RelatedObjectType eq $example:OLI_PHONE]
return (
    ddttek:sql-delete($relation),
    ddttek:sql-delete(collection("ACORD_PHONE")/
        ACORD_PHONE[PhoneKey eq
$relation/RelatedObjectKey])
    )
)
};

```

## XQUERY AND EDI

Organizations often need to be able to handle not only XML ACORD standards, but also older non-XML standards, like ACORD EDI, or even X12 EDI messages. Because EDI does not use XML grammars, it can be difficult to access using XQuery or other XML query languages. But thanks to the integration with Progress® DataDirect XML Converters®, even EDI messages can be easily accessed using DataDirect XQuery. DataDirect XQuery integration with DataDirect XML Converters allows organizations to handle difficult data integration scenarios, such as those in which some business partners are dealing with XML grammars, but others still rely on EDI formats.

Using DataDirect XQuery to access EDI messages can be achieved at the API level (by converting an EDI message into XML and then binding the resulting stream to an XQuery external variable), or by using the `fn:doc()` function and the proper URL format supported by DataDirect XML Converters™.

### *SAMPLE X12 MESSAGE*

Suppose we receive the following X12 Insurance Plan Description message:

```

ISA+00+          +00+
+01+abcdefghijkmno+01+onmlkjihgfedcba+070601+1217+U+00401+0000321
23+0+P+*'
GS+PG+9988776655+1122334455+20070601+1217+128+X+004010'
ST+100+00128001'
BGN+00+88200001+20070601'
NM1+GQ+1+Anderson+Anthony++Doctor+Senior'
N3+14 Oak Park+Room 1078'

```

```
N4+Bedford+MA+01730-1414+US+F'
SE+6+00128001'
GE+1+128'
IEA+1+000032123'
```

Our task is to extract information about the referenced party from our database. The following XQuery does that:

```
import module namespace example =
"http://www.datadirect.com/xquery/examples" at
"acord_verbs.xquery";

declare namespace tx="http://ACORD.org/Standards/Life/2";
declare option dtek:detect-XPST0005 "no";

let $request:= doc("converter:EDI:long=yes? 100.x12")
let $person := example:x12-to-party($request, "Party_1")
return
  example:insert-186 (
    <tx:TXLifeRequest> {
      <tx:OLife> {
        $person
      }
    }
  )
```

The `fn:doc()` function references the custom DataDirect XML Converters URL scheme, which triggers the on-the-fly conversion of the X12 message to XML; then, information about the referenced person and address is extracted and wrapped in a `tx:Party` element. This `tx:Party` element is then used to add to our database—the same way we do in the example Managing Complex Update Requests, which shows insertion of new items described in ACORD tc=186 messages. Similarly, the ability to reuse XQuery functions is clearly another good reason for using XQuery modules in your code.

Integration of DataDirect XML Converters with DataDirect XQuery allows organizations to handle difficult scenarios in which not all business partners are dealing with XML grammars. The underlying streaming architecture of XML Converters perfectly fits the streaming processing capabilities of DataDirect XQuery to ensure performance and scalability even when dealing with non-XML messages.

---

## USING XQUERY TO CREATE PDF

Suppose that instead of returning the result of an ACORD search request as an ACORD XML response, we need to return a report formatted for human consumption, say as PDF. In Chapter 5, “Updating a Relational Database,” we saw how to resolve an inquiry ACORD message and return a result in raw XML format; but how can we leverage DataDirect XQuery to create a well-formatted report instead?

XSL-FO (XSL Formatting Objects, <http://www.w3.org/TR/xsl>) is an XML vocabulary for specifying formatting semantics. In simpler words, XSL-FO specifies an XML structure that is used to describe richly formatted documents, in particular documents that can be easily converted into readily usable PDF or PostScript documents, for example. This implies that if the output of your XQuery executing an inquiry is not a ACORD XML document, but an XSL-FO document containing both formatting information and data, then that XSL-FO document can be converted into PDF. Many commercial and open source technologies are available for this purpose—for example, Apache FOP (<http://xmlgraphics.apache.org/fop/>). And development environments like Progress® Stylus Studio® provide visual tools that make creation of XQuery that generates XSL-FO based on dynamic data very easy.

### *TRY IT YOURSELF*

The XQuery example available for download (see Chapter 10, “Installing and Running Examples”) processes a tc-301 message “party search,” and then generates an XSL-FO document that can be converted into PDF.

## BINDING EXTERNAL VARIABLES

In the examples described earlier in this paper, we focused only on the kind of XQuery that is required to resolve processing requests received by our hypothetical ACORD-based system. But how do you execute an XQuery using DataDirect XQuery? How do you bind external XQuery variables to dynamic values?

DataDirect XQuery implements the XQJ (XQuery for Java) interface. XQJ is analogous to JDBC, which is used by Java applications in conventional

SQL environments. The XQJ API specification (JSR 225) is currently in Public Review, part of the Java Community Process (JCP). You can learn more about this emerging standard here: <http://www.jcp.org/en/jsr/detail?id=225>.

The examples described in this chapter show how you can use the DataDirect XQuery XQJ implementation to execute any of the XQuery examples described in XQuery Operation Examples for an ACORD Insurance Application and which require binding documents to external variables in the Java code. We also show how XQJ can be used in your EJB environment. Finally, we will try to answer a question that is probably on your mind: how difficult is it to achieve the same kinds of results using alternative technologies, like a conventional SQL+DOM approach in your Java application?

### *INTRODUCTION TO DATADIRECT XQUERY AND XQJ*

This Java example shows how simple it is to setup and execute an XQuery using DataDirect XQuery and its XQJ interface. The example references a 201 ACORD XML message used in Chapter 5, "Updating a Relational Database," but it can be easily modified/parameterized to execute other examples described in this book.

#### **Sample Java Code**

As you can see the core of the Java code is straightforward:

```
public void run() throws Exception {
    // configure datasource
    DDXQDataSource ds = new DDXQDataSource();
    ds.setBaseUri(Acord201.queryBaseDir);
    ds.setJdbcUrl(connectionUrl);

    // connect
    XQConnection c = ds.getConnection();

    // create expression
    XQExpression expr = c.createExpression();

    // bind input document
    String uri = new File(Acord201.queryBaseDir +
        Acord201.inputDoc).toURI().toString();
    SAXSource request = Acord201.getSAXSource(uri);
    expr.bindDocument(new QName("request"), request);
}
```

```

// execute XQuery
XQSequence reply = expr.executeQuery(new FileInputStream
(new File(Acord201.queryBaseDir + Acord201.inputQuery)));

// dump reply to stdout
Acord201.dumpSequence(reply);
expr.close();
c.close();
}

```

The example code dumps the XQuery result to stdout, but that can be changed to suit your needs (creating a file, streaming the result to another XQuery processing, creating an XML DOM representation of the result, or feeding the result to an XSL-FO processor, for example).

### ***USING DATADIRECT XQUERY AND XQJ WITH EJB***

The XQJ interface is designed to provide Java developers with an easy way to access XQuery functionality from virtually any environment, and to resemble the structure of the well known JDBC interface. DataDirect XQuery integrates smoothly in virtually any J2EE environment.

This Java example in particular shows how you can leverage DataDirect XQuery and XQJ to create a Message Driven Bean. The example references a 181 ACORD XML update request used in Chapter 5, "Updating a Relational Database," but it can be easily be modified/parameterized to execute other examples described in this book.

### ***Sample Code***

DataDirect XQuery is able to plug into the JDBC pooling mechanism supported by most application servers. The core of the required Java code, shown here, is quite simple:

```

public void onMessage(Message message) {
    Connection jdbc_c = null;
    try {
        Context initContext = new InitialContext();
        Context envContext = (Context)initContext.lookup
            ("java:/comp/env");
        DataSource jdbc_ds = (DataSource)envContext.lookup

```

```

        ("jdbc/DDXQExample");
jdbc_c = jdbc_ds.getConnection();

XQConnection xqj_c =
    XQueryConnection.getXQConnection(jdbc_c);
XQPreparedExpression xqj_p =
    xqj_c.prepareExpression(Acord181EJB.queryBaseDir +
        Acord181EJB.inputQuery);

if (message instanceof TextMessage) {
    // Get the XML request from the message
    String text = ((TextMessage)message).getText();

    // Bind the request
    xqj_p.bindDocument(new QName("request"), new
        InputSource(new StringReader(text)));

    // Execute; no need to handle the response; the XQuery is
    // performing update operations
    xqj_p.executeQuery();
}
}
...

```

### ***COMPARING SOLUTIONS: USING DATADIRECT XQUERY VERSUS SQL AND DOM***

The XQuery and Java examples described previously in this paper, and in this chapter in particular, are all very interesting and show how powerful XQuery and the DataDirect XQuery implementations are in manipulating ACORD XML messages, EDI messages, and relational databases. But why do you need them? Can you implement the same tasks writing traditional Java code that executes SQL queries over JDBC and that fetches/creates XML documents through XML DOM interfaces? Of course you can, but the cost of doing that compared to the combination of XQuery and Java is huge. The best way to describe that is to show a couple of examples that you can compare with the XQuery solutions described in the other sections.

### ***Processing an ACORD Message***

The first example is about processing an ACORD message requesting information about a specific policy product (tc=201). (The corresponding XQuery is described in Chapter 5, "Updating a Relational Database," and the necessary Java layer to execute the XQuery is described in "Introduction to DataDirect XQuery and XQJ.") If we want to solve the same problem using Java+SQL+XML DOM (already it seems more complicated), then you need a Java application similar to the one described in the file `acord201JDBC.java`, which you can find here: [http://www.xquery.com/ACORD/code\\_samples/acord201jdbc.java](http://www.xquery.com/ACORD/code_samples/acord201jdbc.java)

Just comparing the visual size of the two solutions makes the XQuery-based approach particularly appealing. Not only does the XQuery solution provide coding economy, but it does not suffer from any of the drawbacks of the Java+SQL+XML DOM approach. In the Java+SQL+XML DOM approach:

- > Most of the Java code is dedicated to fetching information from the incoming XML message and to formatting the resulting XML response following the ACORD standards.
- > The code dealing with XML is difficult to read, difficult to maintain, and error prone.
- > The SQL statement and its execution are fairly simple, but its use in conjunction with XML data makes it cumbersome. As you try to perform more complex operations with the resulting XML message (reporting and some of the more complex ACORD message types are typical examples of this), the SQL statements needed to meet these requirements become increasingly complicated.

### ***Processing an ACORD Party Inquiry***

The second example is based on an ACORD party inquiry, which requires joins across multiple tables, as described in Chapter 3, "Performing a Search using Multiple Joins." This makes the Java+SQL+XML DOM solution much more complicated, as you can see in the Java code `acord204JDBC.java`,

---

which you can find here: [http://www.xquery.com/ACORD/code\\_samples/acord204JDBC.java](http://www.xquery.com/ACORD/code_samples/acord204JDBC.java)

Now three SQL queries are needed, and the creation of the XML result gets significantly more complicated.

With its superior economy, ease of development (and ease of maintenance), simple access to and manipulation of relational and legacy data, and integration with XQJ, DataDirect XQuery provides an elegant solution for application developers and integration architects working with ACORD and other standards.

## INSTALLING AND RUNNING EXAMPLES

You can run the examples discussed in this book on your own machine. To get started, you need:

- > A copy of DataDirect XQuery 3.0 or later—either an evaluation copy or a fully licensed DataDirect XQuery will do.
- > Your local or remote Microsoft SQL Server (2000 or later) database.

The examples can be modified to be installed and run against different databases, but we have written and tested them against a Microsoft SQL Server database at this point.

### *PREPARING THE DATABASE*

Before you can run the examples, you need to create and populate the tables in your database. We have prepared two SQL scripts to make this task as easy as possible:

- > ACORD\_create.sql creates and populates the tables described in Relational Insurance Model Using ACORD Standards. To execute this script, just open it in SQL Server Management Studio, and run it. You can find the ACORD\_create.sql file here: [http://www.xquery.com/ACORD/code\\_samples/ACORD\\_create.sql](http://www.xquery.com/ACORD/code_samples/ACORD_create.sql)
- > ACORD\_drop.sql drops the tables added by ACORD\_create.sql from your database. You can find the ACORD\_drop.sql file here: [http://www.xquery.com/ACORD/code\\_samples/ACORD\\_drop.sql](http://www.xquery.com/ACORD/code_samples/ACORD_drop.sql)

Both scripts assume that you want to create/remove the tables in the pubs database; you can easily change that default at the beginning of the script files.

### ***RUNNING THE XQUERY FILES***

The XQuery files described in this book are all available in the ACORD-xquery.zip file, which you can download here:

[http://www.xquery.com/ACORD/code\\_samples/ACORD-xquery.zip](http://www.xquery.com/ACORD/code_samples/ACORD-xquery.zip)

Once downloaded and expanded on your local file system, you can run the XQuery files in at least two different ways:

- > From a command line. You can execute the sample XQuery files with DataDirect XQuery using the following command line template:

```
>java -jar c:\ddxq3.0\lib\ddxq.jar -jdbc
jdbc:xquery:sqlserver://localhost:1433;databaseN
ame=pubs;user=user;password=pass !indent=yes
acord_201.xquery
```

Make sure you use the proper location for ddxq.jar (depending on where you have installed DataDirect XQuery) and set the correct information about server location, database name, and user information. The template command line string shown here assumes the current directory is where the sample .xquery files are located on your local file system. The “!indent=yes” option is not necessary, but it makes the result much easier to read.

- > If the XQuery requires that a document is bound to an external variable, you can do that from the command line specifying the variable binding:

```
>java -jar c:\ddxq3.0\lib\ddxq.jar -jdbc
jdbc:xquery:sqlserver://localhost:1433;database
Name=pubs;user=user;password=pass
acord_201-extvar.xquery !indent=yes +request=
"201-1.xml"
```

- > From Stylus Studio  
You can run all sample files from the Stylus Studio development environment (<http://www.stylusstudio.com>). Before being able to

---

do so though, you need to configure the XQuery in Stylus Studio to connect to the proper database:

- > In Stylus Studio File Explorer, right click on the Relational DB entry, and choose New Server.
- > Enter the connection information to the server where you have installed tables and data as described earlier.
- > Connect to the server and navigate down to the database (pubs, if you chose the default installation) where the data is located.
- > Open any sample XQuery, and drag and drop the database (pubs, for example) to the panel to the right of the XQuery text.
- > Run the XQuery (click the green arrow at the top of the XQuery Editor or press Ctrl+F5).

Repeat these steps for each XQuery you want to run. Note, however, that once you save them, this information is available next time you open them in Stylus Studio.

### ***ABOUT THE JAVA FILES***

The Java files described in Chapter 9, “Binding External Variables,” can be compiled and run in your preferred Java development environment (and in Stylus Studio). These files are also available in ACORD-xquery.zip.

Note that you need to edit the top sections of the Java sources to reference the correct folder locations and database connection strings.

In addition, Acord181EJB.java and Acord201.java rely on DataDirect XQuery. These files need ddxq.jar in your classpath to compile and run. Finally, Acord201JDBC.java and Acord204JDBC.java rely on Progress® DataDirect Connect® for JDBC™ and require the proper DataDirect Connect for JDBC libraries in the classpath to compile and run.



---

## PROGRESS SOFTWARE

Progress Software Corporation (NASDAQ: PRGS) is a global software company that enables enterprises to be operationally responsive to changing conditions and customer interactions as they occur. Our goal is to enable our customers to capitalize on new opportunities, drive greater efficiencies, and reduce risk. Progress offers a comprehensive portfolio of best-in-class infrastructure software spanning event-driven visibility and real-time response, open integration, data access and integration, and application development and management—all supporting on-premises and SaaS/cloud deployments. Progress maximizes the benefits of operational responsiveness while minimizing IT complexity and total cost of ownership.

## WORLDWIDE HEADQUARTERS

Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA

Tel: +1 781 280-4000 Fax: +1 781 280-4095 On the Web at: [www.progress.com](http://www.progress.com)

For regional international office locations and contact information, please refer to the Web page below: [www.progress.com/worldwide](http://www.progress.com/worldwide)

Progress, DataDirect, DataDirect Connect, DataDirect Connect for JBDC, DataDirect XQuery, Stylus Studio, and Business Making Progress are trademarks or registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners. Specifications subject to change without notice.

© 2009 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

Rev. 12/09 | 6525-128340