XQJ – XQuery Java API is Completed

Marc Van Cappellen, Zhen Hua Liu, Jim Melton, Maxim Orgiyan

Progress DataDirect
14 Oak Park Drive
Bedford, MA
marc.van.cappellen@datadirect.com

Oracle
500 Oracle Parkway
Redwood Shore, CA
{zhen.liu,jim.melton,maxim.orgiyan}@oracle.com

ABSTRACT

Just as SQL is a declarative language for querying relational data, XQuery is a declarative language for querying XML. JDBC provides a standard Java API to interact with variety of SQL engines to declaratively access and manipulate data stored in relational data sources. Similarly, XQJ provides a standard Java API to interact with a variety of XQuery engines to declaratively access and manipulate XML data in variety of XML data sources. XQJ, also known as JSR 225, is designed through the Java Community Process (JCP) [20]. The XQJ specification defines a set of Java interfaces and classes that enable a Java program to submit XQuery expressions to an XQuery engine operating on XML data sources and to consume XQuery results. In this article, we discuss the XQJ API's technical details with its similarities and differences from JDBC, the design philosophies and goals for XQJ, the implementation strategies of XQJ in variety of XQuery engines and their operating environments, and the possible future of XQJ.

1. INTRODUCTION

Observing the widely successful deployment of JDBC [19] as a standard API for Java applications to plug and play with a variety of SQL engines with different relational backend data sources, we believe the same requirements and use cases exist for XQuery with XML data sources. Furthermore, due to the existence of many XQuery implementations designed for operating in variety of environments managing both persistent and transient XML data, it is self-evident that Java applications should have a standard uniform API to interact with different XQuery engines and their operating environments. The XQJ efforts were started in late 2003, with its initial API draft available in 2004 [1]. Although the final release of XQJ was completed in 2009 as JSR225 [2], the XQJ core API has been stable since XQuery [5] became a W3C recommendation in 2007. The XQJ reference implementation (RI) and technology conformance kit (TCK) for the publication of the API have been stable since 2007. Commercial and open source implementations for the early releases of XQJ have been available since 2005.

The rest of this paper is organized as follows. Section 2 gives a motivating example showing XQJ usage. Section 3 presents key concepts in XQJ by discussing the details of the main interfaces and their conceptual similarities and differences compared to JDBC. Section 4 discusses the XQJ design philosophy. Section 5 discusses the implementation and design choices of XQJ for a variety of XQuery and XML data source environments. Section 6 concludes the article with discussing the possible future of XQJ. Section 7 acknowledges the primary contributors to XQJ.

2. MOTIVATING EXAMPLES

Consider Example 1, which represents the typical basic steps of using XQJ in a Java program with the following key concepts.

Obtaining XQDataSource and XQConnection objects: *XQDataSource* is an interface from which *XQConnection* interface objects are obtained. The initial *XQDataSource* object can be created through a typical data source instantiation mechanism in Java. For example, an *XQDataSource* object can be obtained via JNDI lookup or Java property file lookup, or can be explicitly created via calling XQJ specific implementation class for *XQDatasource* interface as shown in the example above. The concepts of *XQDataSource* and *XQConnection* are similar to the concepts of *DataSource* and *Connection* in JDBC respectively.

```
XQDataSource ds = null;
XQConnection conn = null;
XQPreparedExpression expr = null;
XQResultSequence result = null;
try
```

```
{
// obtaining XQDataSource instance
ds = (XQDataSource)Class.forName(
                  "com.jsr225.DataSourceImpl").newInstance();
// obtaining connection
 conn = ds.getConnection("usr", "passwd");
// preparing XQuery expression
String xqry = "declare variable $dname as xs:string external;
 for $i in fn:collection('dept')
 where $i/deptname = $dname
 return
 <dinfo>
   <dname>{$dname}</dname>
   <empcnt>{ count($i/employees)}</empcnt>
 </dinfo>";
expr = con,prepareExpression(xqry);
//bind variable with value
expr.bindString(new QName("dname"), "engineering", null);
// execute the XQuery Expression
XQResultSequence rs = expr.executeQuery();
// Consume results
while (rs.next())
{
  Node domNode = rs.getNode();
// do something with the DOM node
}
} catch (XQException e)
{
  e.printStackTrace();
}
finally
{
// clean up resource
  if (rs != null)
 {
  try {rs.close();} catch (XQException e1) {...}
}
  if (expr != null)
 {
  try {expr.close();} catch (XQException e2) {...}
}
  if (conn != null)
 {
  try {conn.close();} catch (XQException e3) {...}
}
```

Example 1 - XQJ Motivating Example

Preparing and Executing an XQuery: Once an *XQConnection* object is obtained, the XQJ application can execute XQuery using either *XQExpression* or *XQPreparedExpression* interfaces. The difference between the two is that *XQPreparedExpression* is designed to enable users to prepare one XQuery expression and execute it multiple times, each time with possibly different bind values. As shown in Example 1, XQJ applications may pass in a department name as a bind variable so that the same XQuery prepared expression can be re-used to compute different department employee count values with different bind values for the department name. *XQExpression*, on the other hand, is designed to execute an XQuery expression once. That is, a given *XQExpression* object can only be used to evaluate exactly one XQuery expression whereas a given object *XQPreparedExpression* can be reused to execute different XQuery expressions. The concepts of *XQPreparedExpression* and *XQExpression* are similar to the concepts of *PreparedStatement* and *Statement* in JDBC, respectively.

Consuming an XQuery Result: Execution of an XQuery results in an XQuery data model (XDM) [6] instance. The *XQResultSequence* interface allows applications to iterate through each item of the result sequence. XQJ applications can obtain each item as needed, which can be either an atomic value or an XML node. This step is similar to the process of iterating through rows in a JDBC *ResultSet*.

Releasing Resources: Once the XQuery results have been consumed, XQJ applications release the resources by calling corresponding close methods on *XQResultSequence*, *XQPreparedExpression*, *XQExpression* and *XQConnection* interfaces. Use of Java try/catch constructs to catch *XQException* objects and to ensure proper closing of resources are important to avoid resource leakage.

3. XQJ Requirements and Key Concepts

While Section 2 presented aspects of XQJ that are similar to JDBC, there are many key differences between the two. The following is a set of unique requirements for the design of XQJ:

- Providing support for static and dynamic context concepts that are unique to the XQuery language.
- Providing a deferred variable binding mode for binding an XML stream as an external variable input.

- Providing XDM-specific factory methods that allow creation and destruction of XQuery item and sequence objects and XQuery Sequence Type objects. These XDM objects have an independent lifecycle that is unrelated to the lifetime of *XQConnection* and *XQResultSequence* objects.
- Providing mappings for conversion of Java objects to XDM instances in the case of binding XQuery external variables and context item, and XDM instances to Java objects in the case of consumption of XQuery results. In particular, XQJ supports ways of consuming XQuery results using the common XML-related Java interfaces, such as DOM, SAX, and StAX.
- Providing fine-grain exception classes for better diagnosability.

3.1 XQuery Static Context & Dynamic Context Interface

XQuery has concepts of static context and dynamic context. XQJ provides *XQStaticContext* and *XQDynamicContext* interfaces to model them. The *XQStaticContext* provides methods that allow applications to get and set various XQuery static context components such as the Base URI, statically known namespaces, and default collation. The *XQStaticContext* object can be obtained by calling *getStaticContext()* on an *XQConnection* object. However, for a single *XQConnection* object, there can be different XQuery expressions prepared and executed, and each of these expression may need to set different values for certain static context components. Therefore, the association of *XQStaticContext* objects with *XQConnection,* *XQPreparedExpression*, and *XQExpression* objects is passed by value. In other words, a separate copy of the *XQStaticContext* object is made whenever an *XQPreparedExpression* or an *XQExpression* object is created from an *XQConnection* object, so that changes of the static context components in one particular *XQStaticContext* object are isolated from another. Modifications of *XQStaticContext* object retrieved from *XQConnection* object do not affect static context components associated with the *XQConnection* object until the *setStaticContext()* method is invoked on the *XQConnection* object.

The *XQDynamicContext* interface allows XQJ applications to retrieve and set the implicit time zone and bind values for the context item and the external variables of an XQuery. *XQDynamicContext* interface provides *bindString(), bindInt(), bindNode(), etc.* methods to bind XQuery external variable values with a variety of Java built-in primitive types and objects. Both *XQPreparedExpression* and *XQExpression* extend the *XQDynamicContext* interface.

3.2 Deferred Variable Binding Mode

To scale with large data size, mature SQL implementations use iterator-based lazy execution models [17] in which the full SQL query result set is not materialized at once but rather produced one row or a set of rows at a time. Applications use an iterator-based interface to obtain the result of such execution. This is reflected in the *ResultSet.next()* fetching method in JDBC. XQuery can be evaluated in the same iterator manner to scale with large data size [15]. This naturally justifies the *XQResultSequence.next()* JDBC-like fetching method. However, for the case of variable binding, the XQJ and JDBC requirements are different. In JDBC, SQL variable binding supports only simple scalar values. There is no concept of binding relational result sets that can be potentially large in size. However, in XQJ, an XQuery variable binding can be an XML document or an XQuery sequence of any size. Furthermore, the XQuery sequence object can be obtained from an XML stream API providing an iterator-like fetch interface. In such a use case, it makes sense for XQJ implementations to defer the binding of the input XML data stream until the XQuery execution time when the input XML data stream is actually consumed. Therefore, besides the default immediate binding mode, a deferred binding mode can be set in the *XQStaticContext*. In the deferred binding mode, the bind value might not be consumed until the variable is actually accessed by the underlying XQuery engine. This enables lazy value consumption and improves XQuery performance and scalability.

3.3 XDM Data Factory Support

XQJ needs to model XQuery Data Model [6] concepts. An XDM instance is a sequence of XQuery items. Each item can be an atomic value or an XML node (document, element, attribute, comment, processing instruction, or text). *XQSequence* is the XQJ interface that models XQDM instances. It contains zero or more *XQItem* interface objects. The *XQItem* interface in XQJ represents an XDM item. XDM is used both as input to an XQuery expression and output from the evaluation of that XQuery expression.

The XDM goes hand in hand with the XQuery type system. The *XQSequenceType* and *XQItemType* are two interfaces in XQJ enabling applications to work with the XQuery type system. The *XQSequenceType* interface represents the sequence type defined in XQuery. The *XQItemType* interface represents an item type defined in XQuery. The *XQItemType* interface extends the *XQSequenceType* interface, but restricts its occurrence indicator to be exactly one. The *XQItemType* interface provides methods to obtain information such as the item kind, the base type, the name of the node (if any), the type name of the node (if any), and the XML schema URI associated with the type (if any).

There are two kinds of *XQSequence* and *XQItem* objects in XQJ, and they differ in how they are obtained. The first kind of XDM object is obtained from the result of XQuery execution. Recall that an object of class *XQResultSequence* (the interface extending *XQSequence*) is obtained by invoking the *executeQuery()* method of an *XQExpression* or an *XQPreparedExpression* object. An object of class *XQResultItem* (the interface extending *XQItem*) is obtained by calling the *getItem()* method of an *XQResultSequence* object. The second kind of XQDM object- is obtained by explicit creation via the *XQDataFactory* interface from which the *XQConnection* interface extends. Once created, these objects have lifetimes that are independent of the lifetime of the *XQConnection* object that created them. They remain valid until the *close()* method is called on them. In contrast, the lifetime of *XQResultSequence* and *XQResultItem* objects obtained from XQuery execution depends on the lifetime of the *XQExpression* or *XQPreparedExpression* objects that created them. The lifetime of the *XQExpression* and *XQPreparedExpression* objects, in turn, depends on the lifetime of the *XQConnection* object that created them. Closing an *XQConnection* object implicitly closes all the XDM instances resulting from the execution of XQueries in the context of this *XQConnection* object.

3.4 XDM & Java Object Type Conversion

As stated earlier, an XDM item can be either an atomic value or an XML node. Atomic values can be of a variety of XQuery built-in datatypes, such as *xs:decimal, xs:boolean, xs:integer*, all of which have default Java data type mappings defined by XQJ to facilitate conversion between Java built-in type objects and XDM instances of built-in XQuery types.

For binding XML nodes, XQJ provides methods to interact with different Java interfaces that represent XML documents. The *XQDynamicContext* interface provides various bind methods to create XQuery document nodes from the following Java objects: *java.lang.String*, *java.io.Reader*, *java.io.InputStream*, *java.xml.stream.XMLStreamReader*, and *javax.xml.transform.Source*. There is also a *bindNode()* method for binding *org.w3c.dom.Node* DOM nodes.

For consuming XML nodes, *XQItemAccessor* interface (from which both *XQItem* and *XQSequence* extend) provides various "get" methods to convert XDM nodes into the following Java objects: *java.lang.String, java.io.Writer, java.io.OutputStream, org.w3c.dom.Node, javax.xml.stream.XMLStreamReader*, *javax.xml.transform.Result,* and *org.xml.sax.ContentHandler.* In addition, the *XQSequence* interface provides methods to convert an entire XDM sequence into Java objects representing XML nodes.

3.5 Exception Handling

XQJ allows the user to distinguish XQuery static or dynamic errors from other non-XQuery related errors through two exception classes: *XQException* and *XQueryException*. The *XQueryException* class gives access to the error code as defined by XQuery, error location information, and various other attributes.

To potentially report multiple errors during the static analyses or dynamic evaluation phase, *XQException* instances are chained and the XQJ application can invoke *getNextException()* method to retrieve all of the exceptions.

4. XQJ Design Philosophy

4.1 Support for Multiple XQuery Engines Deployment Environments

Since SQL and XQuery share many commonalities (such as a declarative language, amendability for iterator based set at-a-time processing model, support of bind variables, static compilation and dynamic evaluation phases), XQJ reuses those JDBC concepts that are applicable to XQuery as much as possible. However, we recognize that there are various XQuery implementations targeting different operating environments. While it is true that major RDBMS vendors [7,8,9,10] support XQuery for querying XML document content stored in an RDBMS and in XML views over relational data through the SQL/XML standard [4], XQuery is also supported by XML content server vendors to query pure XML content [11, 18]. XQuery is supported in mid-tier servers to provide uniform query language access to query different backend XML data sources, relational sources, and XML messages [16]. XQuery is also supported via standalone libraries to be embeddable indifferent types of applications [12, 13]. Therefore, XQJ has to be an API separate from JDBC. It cannot assume that XQuery and SQL coexist in one environment. Instead, XQJ provides an API to handle the variety of XQuery deployment environments.

4.2 Stylistically Consistency with JDBC drivers

XQJ requires establishment of an *XQConnection* before executing an XQuery. This appears unnatural for single-tier collocated XQuery deployment environments in which the

XQuery engine is embedded into the Java application. However, the *XQConnection* interface does not entail a physical network connection object but rather a handle that needs to be created before executing XQuery. Therefore, an *XQConnection* implementation can be either a light-weight handler type object for a single-tier XQuery embedded environment, or can be a heavy-weight network connection object for a multi-tier XQuery server environment. In the latter case, the connection pooling technique can be facilitated via the *PooledXQConnection* interface defined by XQJ.

The *XQSequence* interface ties the XQDM concept and the iterator based access to items within a single *XQSequence* object. Although it is debatable whether modeling the two separately is conceptually clean, here XQJ follows the design of JDBC *ResultSet* that represents both a set of relational rows and its iterator accessor as one object, because both of these tend to be accessed together.

4.3 XQDM Interoperability

Unlike relational result column values, which are scalar values mapped to common built-in Java types, XDM item types can be associated with user defined XML schemas. Ensuring a consistent XML schema repository among all tiers in a distributed environment is a complex and expensive task that applications may not be willing to bear. Furthermore, interpreting XML nodes may require the full context of the XML tree of which the node is a part. That is, XQuery types and XDM instances are not *standalone,* but rather context dependent. Thus, interoperability of XDM among XQJ drivers is an issue. Even for the same XQJ implementation in a client/server architecture, the XDM node exchanged between the XQJ client and the XQuery engine on the server can be passed by value or by reference. Passing by reference could be expensive and requires communication of the XML schema information, and full tree node context information among different tiers. Even for the same XQJ driver, the XQuery engine and the XQJ clients may run in different tiers. So the XML schema information and full tree node context information need to be communicated among XQJ tiers. Although there are mechanisms to support such interoperability, it requires proprietary design among XQJ implementations that is beyond the scope of XQJ. Therefore, XQJ only guarantees interoperability of *XQItem* and *XQSequence* instances having built-in XML Schema types, and *XQItemType* and *XQSequenceType* instances of built-in XML Schema types. It is implementation-defined whether an XDM node is passed by value or by reference. However, even for XDM nodes exchanged among different XQJ drivers and XQJ tiers with pass-by-value semantics, the node preserves all of its descendants, with all nodes being untyped.

5. XQJ Driver Architecture & Implementation Choices

In this section, we discuss various implementation choices to effectively and efficiently support XQJ drivers.

5.1 XQJ Driver Architecture Choices

Embeddable XQJ driver for collocated XQuery engine: in this architecture, the XQuery engine and XQJ driver are collocated in the same JVM that is also shared by the Java application that uses the XQJ API, and there is no physical network connection among the XQJ driver, the XQuery engine, and the application. XDM instances can preserve their full tree node context and type context. Passing XDM instances by reference is supported effortlessly.

Mid-tier based XQJ Driver for pure XML Content Server: in this architecture, the XQuery engine runs in an XML content server that provides management of and query over XML content. The client applications that access and query the XML content typically run in different tiers than that of the XML content server. The XQJ driver runs on each client tier and communicates with the XML content server using implementation-specific protocols. It is performance-critical for such an XQJ driver to implement *XQConnection* pooling to scale with a large number of clients. The XQJ client and XQuery server may choose a loosely coupled or a tightly coupled architecture, depending on the application requirements. For the tightly coupled choice, the XML content server and its client can share a common XML schema repository, and XML nodes can be passed by reference (much like in the distributed object database architecture). For the loosely coupled choice, the XML content server and all of its clients may not share the same XML schema repository. XML nodes exchanged between tiers are passed by value. The loosely coupled choice typically gives better scalability than the tightly coupled choice.

Mid-tier based XQJ Driver for SQL/XML enabled RDBMS: An SQL/XML-enabled RDBMS supports XQuery and XML via new concepts in SQL, such as the XML type, XMLQuery() functions, and XMLTable table function[14] that are defined in the SQL/XML standard[4]. RDBMS servers already provide support for JDBC drivers that run SQL/XML queries. An XQJ driver can be built on top of the JDBC driver, to leverage all the underlying plumbing from JDBC. Submitting an XQuery using the JDBC driver boils down to essentially running the '*SELECT * FROM XMLTABLE(xquery PASSING BY REF COLUMNS '.' XML BY REF)*' SQL/XML query. However, the XML type implementation in the JDBC driver has to be sophisticated enough to support the XDM model with an optional XML schema attached. For a query returning persistently stored XML nodes, passing nodes by reference can be feasibly supported.

Mid-tier based XQJ driver for data integration: in this architecture, XQuery is implemented in the mid-tier with the XQJ driver. Its backend data sources can be relational data sources, XML data sources, or other data sources whose data content can be converted into XML or even XML messages, local XML files, *etc*. The XQJ driver can open different kinds of connections to its underlying data sources and push down connection-specific queries to fetch the data content so that the XQuery engine in the XQJ driver can then further filter and assemble the result, based on the original user XQuery.

5.2 Facilitate efficient XQuery Evaluation

XML content has different shapes and sizes. While small to medium size XML documents can be processed as DOM objects, it is generally not a scalable solution to process large XML documents, or large number of XML nodes, as an XDM instance from an XQuery result. Thus, the most efficient XQuery implementations leverage iterator-based streaming evaluation strategies as much as possible, to cope with large XDM instances. XQJ recognizes this and provides constructs to facilitate such lazy evaluation strategies. An efficient XQJ driver implementation can consider the following design ideas:

- When retrieving an XDM sequence from XQuery execution, invoke the *getNext()* call of the underlying XQuery engine to consume one item at a time. Certain XQuery engines [15] may even work at sub-XQItem level by making *getNext()* operate at the same level of event as that of an XML stream reader API (StAX) [3]. In this case, implementing the StAX API for XDM nodes can be in sync with the underlying XQuery engine, thus yielding better memory utilization and performance. This streaming principle can be applied to cases when XDM is consumed as other streams as well.
- Support a deferred binding mode when the XDM result of one XQuery needs to be passed in as a bind variable value for another XQuery, especially when the intermediate XDM result is large in size. This support can enable end-to-end streaming evaluation among XQuery engines.

6. Conclusion & Future Work

As XQJ finishes its first release, there are multiple concurrent, ongoing XQuery specification efforts, including the XQuery Update Facility, XQuery Full Text, XQuery 1.1, and the XQuery Scripting Extension. The current XQJ specification only supports XQuery 1.0. Since XQuery Full Text expressions, XQuery 1.1 expressions, and XQuery Update Facility transform expressions are all read-only expressions, supporting them using the current XQJ model requires minimal work. However, supporting full power of the XQuery Update Facility in XQJ requires additional infrastructure. Furthermore, it is also debatable whether the XQuery Scripting Extension should be supported via XQJ, because XQuery scripting is an approach of mixing declarative query and imperative procedural manipulation of XML in one language. This contradicts the approach of embedding a declarative query language like XQuery into an imperative programming language like Java.

Nevertheless, the current release of XQJ provides a simple, easy-to-use, and portable API to support the use of XQuery with XML data sources on the Java platform. We believe its impact will be similar to that of JDBC in the years to come.

7. ACKNOWLEDGMENTS

We express our thanks to all JSR225 working group members. The following individuals deserve special acknowledgements: Per Bothner, Andrew Eisenberg, Muralidhar Krishnaprasad, Jason Hunter, Jan-Eike Michels, Karuna Muthiah, and Basuki Soeterman.

8. REFERENCES

[1] Andrew Eisenberg, Jim Melton: An Early Look at XQuery API for Java (XQJ). SIGMOD Record 33(2): 105-111 (2004)

[2] JSR225: http://www.jcp.org/en/jsr/detail?id=225

[3] JSR173: http://www.jcp.org/en/jsr/detail?id=173

[4] International Organization for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML)

[5] XQuery: http://www.w3.org/TR/xquery/

[6] XDM: http://www.w3.org/TR/xpath-datamodel/

[7] Zhen Hua Liu, Muralidhar Krishnaprasad, Vikas Arora: Native Xquery processing in oracle XMLDB. SIGMOD Conference 2005: 828-833

[8] Zhen Hua Liu, Sivasankaran Chandrasekar, Thomas Baby, Hui J. Chang: Towards a physical XML independent XQuery/SQL/XML engine. PVLDB 1(2): 1356-1367 (2008)

[9] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, Eugene Kogan: XQuery Implementation in a Relational Database System. VLDB 2005: 1175-1186

[10] Kevin S. Beyer, Roberta Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong C. Truong, Bert Van der Linden, Brian Vickery, Chun Zhang: System RX: One Part Relational, One Part XML. SIGMOD Conference 2005: 347-358

[11] Mary Holstege: Big, Fast XQuery: Enabling Content Applications. IEEE Data Eng. Bull. 31(4): 41-48 (2008)

[12] Michael Kay: Ten Reasons Why Saxon XQuery is Fast. IEEE Data Eng. Bull. 31(4): 65-74 (2008)

[13] Marc Van Cappellen, Wouter Cordewiner, Carlo Innocenti: Data Aggregation, Heterogeneous Data Sources and Streaming Processing: How Can XQuery Help? IEEE Data Eng. Bull. 31(4): 57-64 (2008)

[14] F Zemke, M. Rys, K. Kulkarni, J. Michels, B. Reinwald, F. Oczan, Zhen. Hua. Liu, I. Davis, K. Hare, "XMLTable" , ISO/IEC JTC1/SC32 WG3:SIA-051 ANSI NCITS H2 2004-039 http://www.wiscorp.com/H2-2004-039-xmltable.pdf

[15] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan: The BEA streaming XQuery engine. VLDB J. 13(3): 294-315 (2004)

[16] Michael J. Carey: Data delivery in a service-oriented world: the BEA aquaLogic data services platform. SIGMOD Conference 2006: 695-705

[17] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 25(2):73–170, 1993.

[18] EMC-XHIVE http://www.emc.com/domains/x-hive/index.htm

[19] JDBC: http://www.jcp.org/en/jsr/detail?id=221

[20] Java Community Process: http://jcp.org/en/home/index