XQuery meets SQL

Luis Rodriguez
@kijon77
Rodolfo Ochoa

XQuery meets SQL

http://dilbert.com/dyn/str_strip/000000000/00000000/0000000/000000/20000/1000/100/21168/21168.strip.gif

What is JSONiq?

- XQuery + JSON = JSONiq
- A way to exploit **decades** of research, standardization and implementation work around **processing flexible data**.

```
let $stats := s:execute-query($db,"SELECT * FROM
stats")for $access in $statsgroup by $url :=
$access("url")return{ "url": $url, "avg":
avg($access("response_time")), "hits": count($access)}
```

Why JSONiq?

- Why do we use JSONiq to represent relational data?
  - **Efficiency** of implementation
  - **Smaller payload** than XML
  - It is already **working on Zorba**!
  - **SQL and NoSQL** input output are **standardized** by using JSONiq.
- JSONiq **made a big splash** when first announced (NoSQL Now! and other conferences)

Why JSONiq?

- JSONiq **impact in the industry**?
  - Three papers have been written in the last couple of weeks about it
    - http://www.infoworld.com/d/application-development/the-time-nosql-standards-now-205109
    - http://www.zdnet.com/the-nosql-community-threw-out-the-baby-with-the-bath-water-7000010361/
    - http://nosql.mypopescu.com/post/4186634936 1/mongodb-is-abusing-json-with-its-query-language

XML & SQL Development

The development of apps that involves both XML and SQL includes the following parts:

- SQL to store their structured data
- XML technologies to gather information from XML sources (like web services)
- An SQL library to connect to Databases
- An intermediate Language to bind XML and SQL
- Some even XQuery over XML fields from SQL

XML & SQL Development

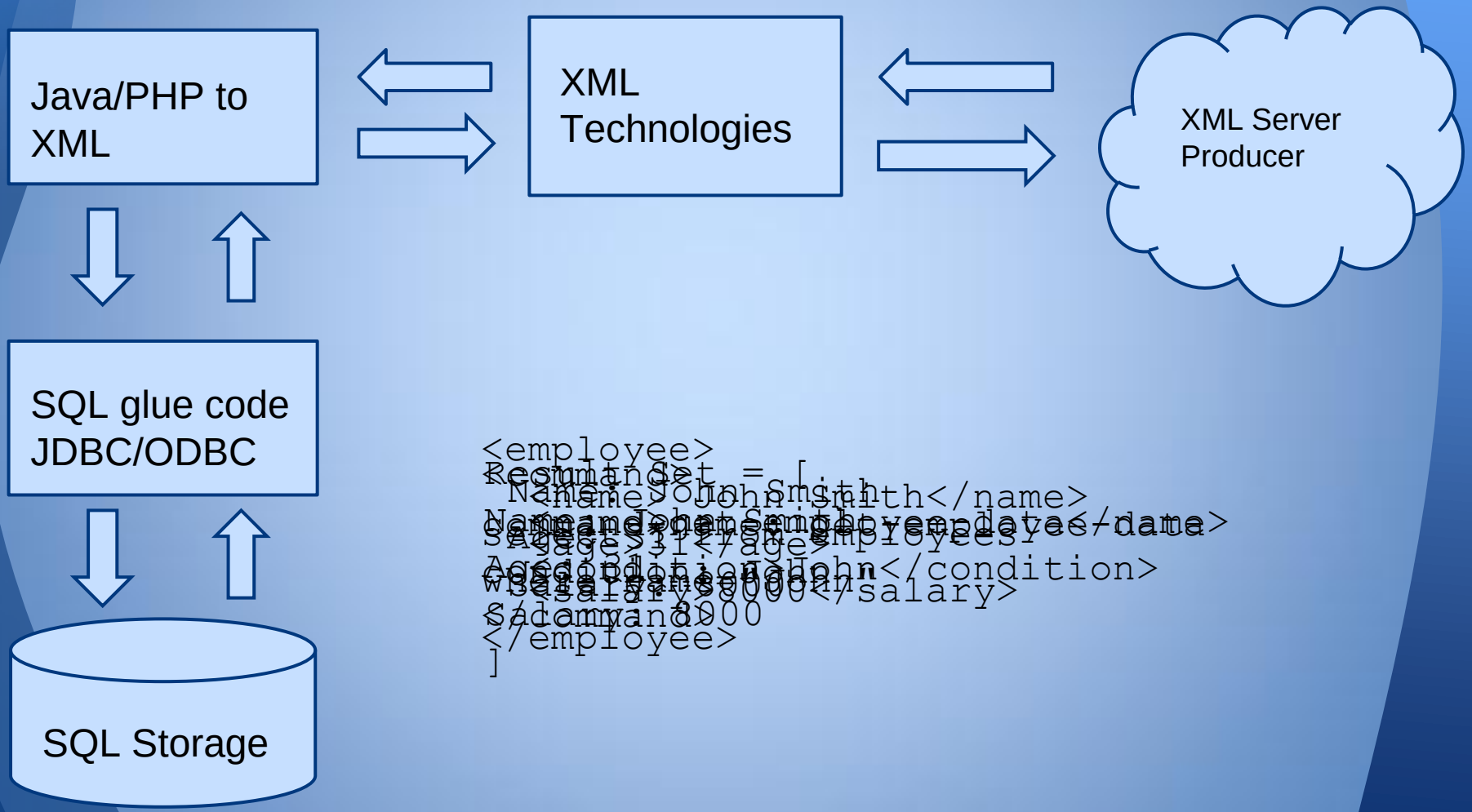
Why not get rid of the "middle man"?

Anyone said **SQLXML**? Why not?

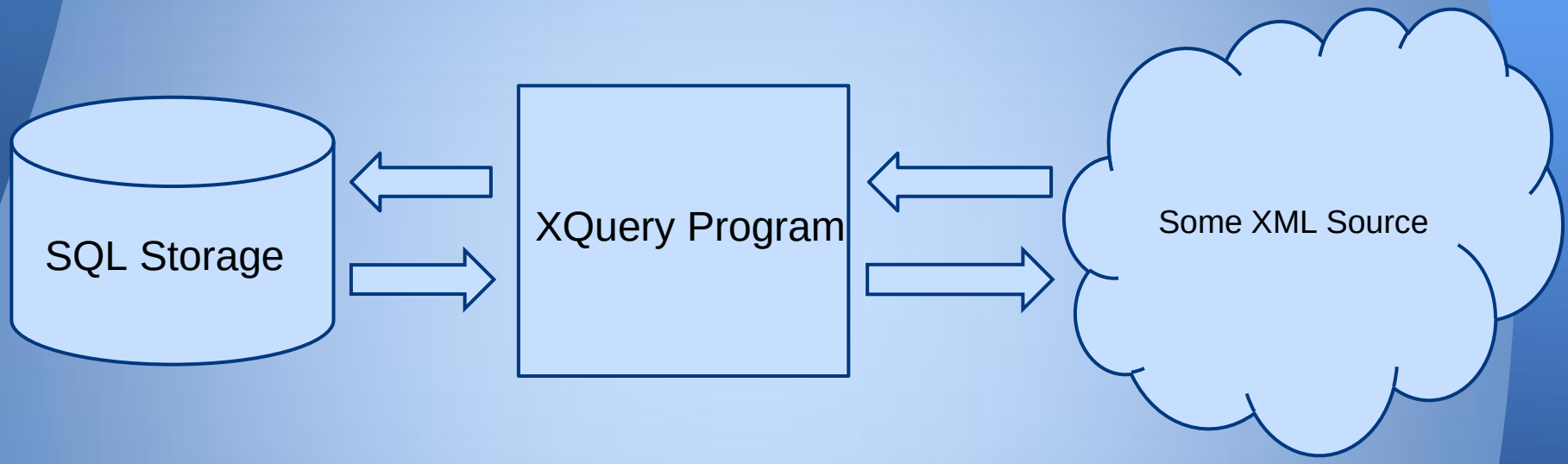
- **Dependent on DB implementation** (eg., MySQL doesn't implement it)
- Sometimes poorly implemented or incomplete (eg. SQL Server)
- **Way more complex than XQuery + SQL modules**

Enter the **SQL modules for XQuery**!!

XML + RDBMS


Java/PHP to XML
XML Technologies
XML Server Producer
SQL glue code JDBC/ODBC
SQL Storage
<employee>
<name>John Smith</name>
<age>31</age>
<salary>80000</salary>
</employee>
]

XML + RDBMS




```
<employee>
  <command>
    <name>John Smith</name>
    <name>getEmployeeData</name>
    <condition>John</condition>
    <salary>2000</salary>
  </command>
</employee>
```

No glue code required!

SQL Modules - Querying DBs

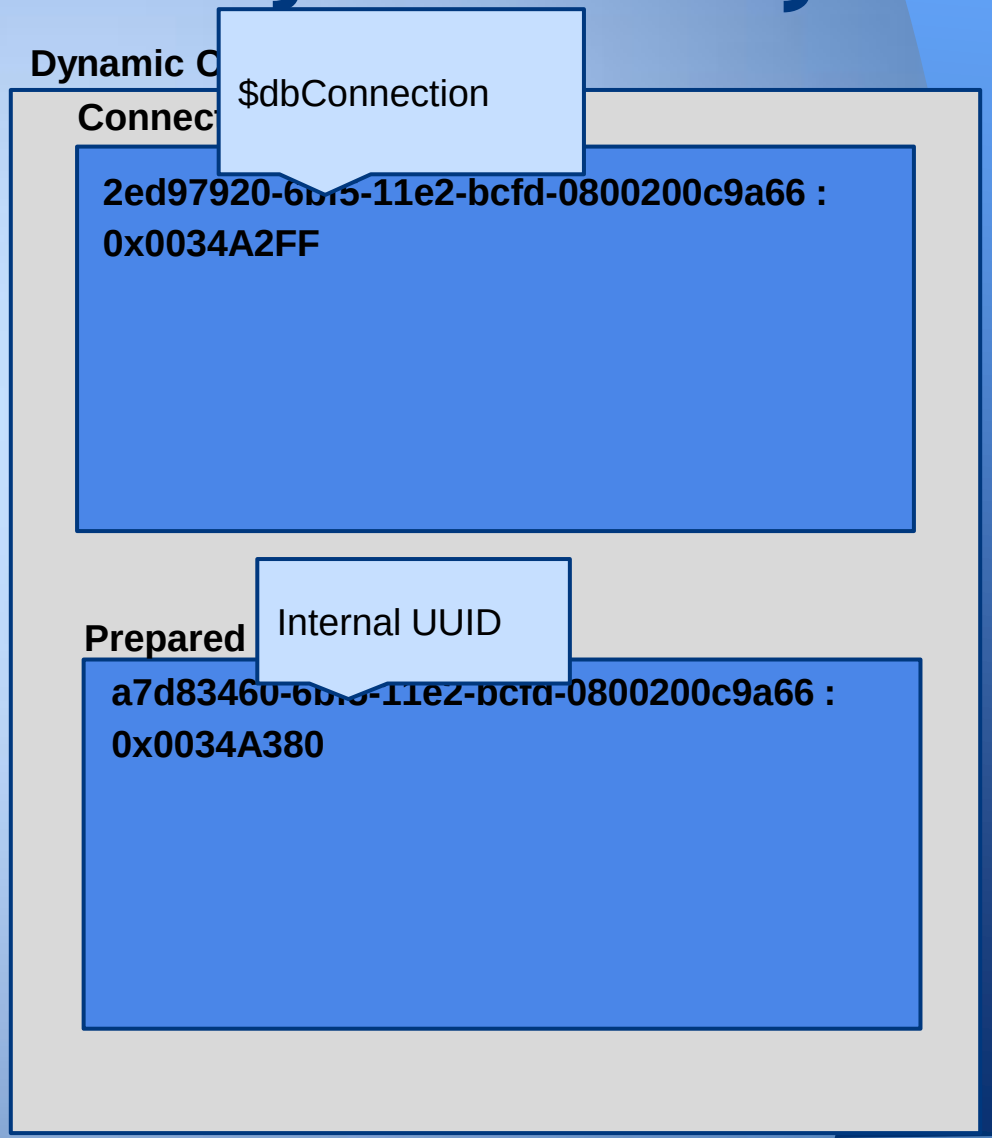
```
import m          espace s =
  "http:          ba-
  xquery.com/modules/sqlite";

let $db(          n :=
  s:connect("D:/smalldb.db")
let $query := "SELECT * FROM smalltable"
let $result := s:execute-
  query($dbConnection, $query)

return $result
```

xs:AnyURI

xs:AnyURI

SQL Modules - xs:anyURI as keys

```
let $dbConnection :=
s:connect("small.db")

let $query := "SELECT *
FROM smalltable"
let $result :=
s:execute-
query($dbConnection,
$query)
```



SQL Modules - Querying DBs

And the output is going to be like this:

```
{ "id" : 1, "name" : "apple", "calories" : 25 }
{ "id" : 2, "name" : "orange", "calories" : 35 }
{ "id" : 3, "name" : "fried egg", "calories" : 92 }
{ "id" : 4, "name" : "chocolate milk regular", "calories"
   : 210 }
```

SQL Modules - Inserting into DBs

```
import module namespace s =
"http://www.zorba-
  xquery.com/modules/sqlite";

let $dbConnection :=
  s:connect("D:/small.db")
let $query :="INSERT INTO smalltable (name,
  calories) VALUES ('carrot', 25)"
let $result := s:execute-
  update($dbConnection, $query)

return $result
```

SQL Modules - Inserting into DBs

And the output is not very interesting though:

```
1
```

SQL Modules - From XML to SQL

```
import module namespace s =
  "http://www.zorba-
  xquery.com/modules/sqlite";

let $xml :=
<root>
<food><name>carrot</name><calories>25</calor
  ies></food>
<food><name>tomato</name><calories>35</calor
  ies></food>
</root>

let $db := s:connect("")
let $inst := s:execute-update($db, "CREATE
```

SQL Modules - From XML to SQL

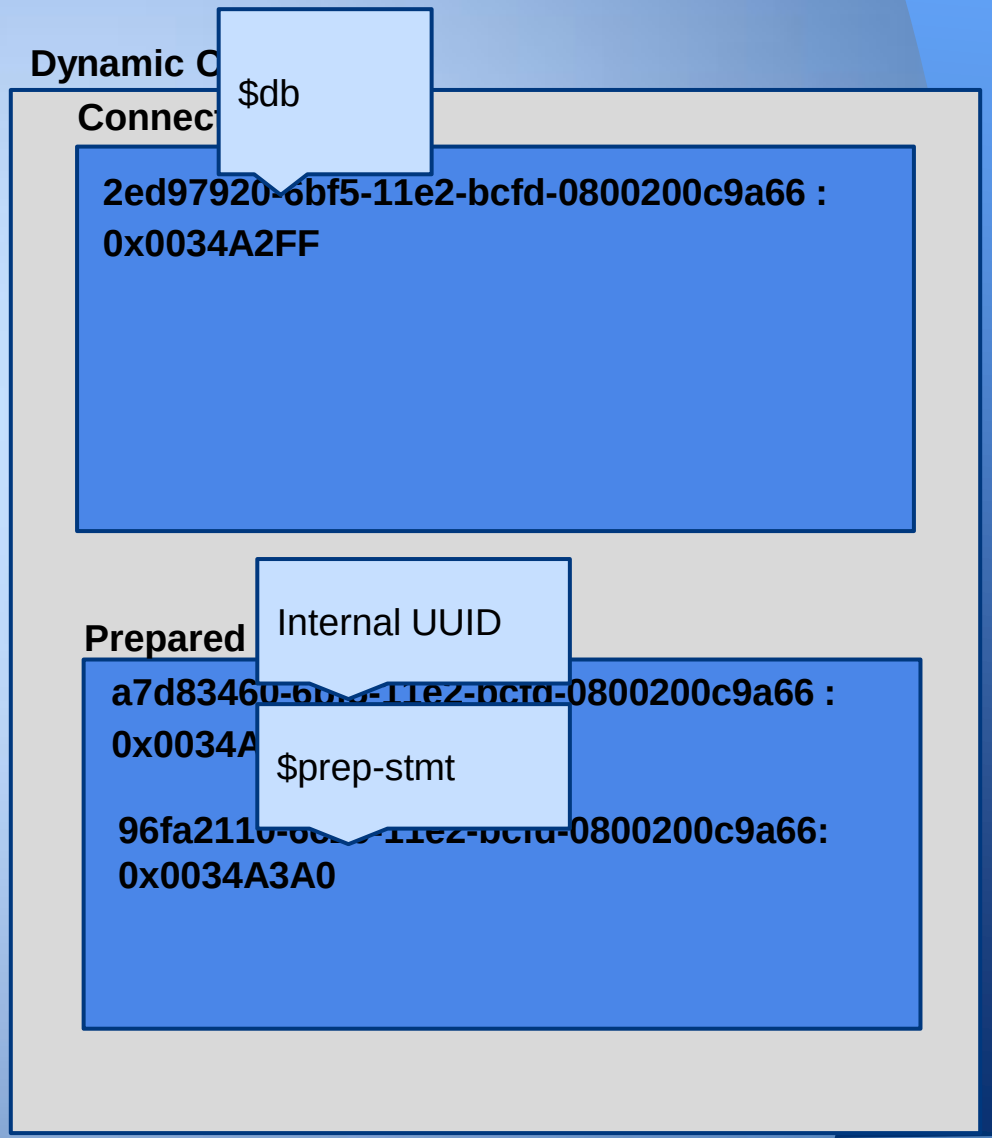
```
let $prep-stmt := s:prepare-statement($db,
  "INSERT INTO smalltable (name, calories)
  VALUES (?, ?)")

for $e in $xml/food
let $name := data($e/name)
let $calories := data($e/calories) cast as
  xs:integer
return {
 s:set-string($prep-stmt, 1, $name);
 s:set-numeric($prep-stmt, 2, $calories);
 s:execute-update-prepared($prep-stmt)
}
```

SQL Modules - Prepared Statements

```
let $db := s:connect("")

s:execute-update($db, "CREATE TABLE smalltable (id INTEGER primary key asc, name TEXT not null, calories INTEGER)")

let $prep-stmt := s:prepare-statement($db, "INSERT INTO smalltable (name, calories) VALUES (?, ?)")

s:set-string($prep-stmt, 1, $name);

s:set-numeric($prep-stmt, 2, $calories);

s:execute-update-prepared($prep-stmt)
```

name: carrot
calories: 25

name: tomato
calories: 35



SQL Modules - Prepared Statements

The output for this is the following table:

id	name	calories
1	carrot	25
2	tomato	35

SQL Modules - SQL to XML

```
let $dbConnection := s:connect("D:/small2.db")
let $query := "SELECT id, name, calories FROM smalltable"
let $results := s:exec                          nection, $query)

for $e in $results
let $id := $e("id")
let $name := $e("name")
let $calories := $e("calories")
return
 <food>
  <id>{$id}</id>
  <name>{$name}</name>
  <calories>{$calories}</calories>
 </food>
```

In JSONiq this means that we want the value associated with "calories"

SQL Modules - SQL to XML

The output is like this:

```
<food>
  <id>1</id>
  <name>carrot</name>
  <calories>25</calories>
</food>
<food>
  <id>2</id>
  <name>tomato</name>
  <calories>35</calories>
</food>
```

SQL Modules - Us

Namespace for JDBC module

connection string

```
import module namespace jdbc = "http://www.zorba-
   xquery.com/modules/jdbc";

let $connectionString :=
   "jdbc:mysql://myhost/human_resources?user=name&password=
   pass"
let $dbConnection := s:connect($connectionString)
let $query := "SELECT idEmployee, name, address FROM
   employees"
let $results := s:execute-query($dbConnection, $query)

for $employee in $results
 let $idEmployee := $employee("idEmployee")
 let $name := $employee("name")
 let $address := $employee("address")
return <employee idEmployee="{$idEmployee}">
       <name>{$name}</name>
       <address>{$address}</address>
```

Using JDBC

The output is similar to the others:

```
<employee idEmployee="1">
  <name>John Smith</name>
  <address>1st Avenue #123</address>
</employee>
<employee idEmployee="2">
  <name>Rebecca Hills</name>
  <address>Oak street #987</address>
</employee>
```

SQL Modules - Metadata

```
{
  variable $pstmt := s:prepare-statement($db, "SELECT * FROM
smalltable");
  variable $meta := s:metadata($pstmt);
  for $i in 1 to jn:size($meta("columns")
  return $meta("columns")($i)
}
```

```
{ "name" : “id”, "table" : “smalltable”, "database" :
“main”, "type" : “INTEGER”, "collation" : BINARY, "nullable"
: false, "primary key" : true, "autoincrement" : true }
{ "name" : “name”, "table" : “smalltable”, "database" :
“main”, "type" : “TEXT”, "collation" : BINARY, "nullable" :
true, "primary key" : false, "autoincrement" : false }
{  "name" : “calories”, "table" : “smalltable”, "database" :
“main”, "type" : “INTEGER”, "collation" : BINARY, "nullable"
```

SQL Modules - Implementation Notes

JDBC Module

- **100% C++ code** interfacing JNI, no extra JAR needed (except for the JDBC driver)
- **Internally delivering data row-by-row** through a **Zorba::Iterator**, this means minimal memory footprint. Drawbacks? No disconnect() available and we depend on the JDBC driver from the Database.
- **XQuery opaque data types** (connections, prepared statements and data sets) **modeled according to existing JDBC data types**
- Total size of the module is about 2200 lines of C++ code

SQL Modules - Implementation Notes

The **type conversion system** was **carefully designed** to fit into JSONiq specification (just xs:base64Binary is out of the simple JSONiq types); minimal data conversion means **less complexity** for the user.

JDBC type	XQuery type
NULL	jn:null
BIGINT - INTEGER - ROWID - TINYINT - BIT - SMALLINT	xs:Integer
DECIMAL - DOUBLE - FLOAT - NUMERIC - REAL	xs:double
BLOB - BINARY - LONGVARBINARY - VARBINARY - ARRAY - DATALINK - JAVA_OBJECT - OTHER - REF	xs:base64Binary
CHAR - CLOB - LONGVARCHAR - LONGNVARCHAR - LONGVARBINARY- NCHAR - NCLOB - NVARCHAR - VARCHAR - DATE - TIME - TIMESTAMP	xs:string

SQL Modules - Implementation Notes

SQLite Module

- Most of the functions are **direct calls** to the **C SQLite API**, a few are not (`metadata()`)
- **Data streaming is the same as JDBC module** (row-by-row through calling `sqlite3_step()`)
- **Only connections and prepared statements needed** (No dataset implementation required)
- Total size of the module is about 1350 lines of C++ code

SQL Modules - Implementation Notes

The **type conversion system** followed the **same process as the JDBC conversions** in order to keep the same **standard**.

SQLite Native type	XQuery type
SQLITE_NULL	jn:null
SQLITE_INTEGER	xs:Integer
SQLITE_FLOAT	xs:double
SQLITE_BLOB	xs:base64Binary
SQLITE_TEXT	xs:string

Conclusions

SQL Modules

- **Extend** the power of **XQuery** to reach the **SQL world**
- **Fill the gap** that has always existed between **JSON/XML world** and the **DB world**
- Provides a **simple** and **straightforward API**
- Are not perfect so **feedback** is expected and appreciated!

References & links

- JSONiq specification http://www.jsoniq.com/
- SQLXML http://en.wikipedia.org/wiki/SQL/XML
- SQLite home page http://www.sqlite.org/
- Scripting extensions for XQuery http://www.w3.org/TR/xquery-sx-10/
- Zorba Home Page http://www.zorba-xquery.com/
- JDBC specification http://www.oracle.com/technetwork/java/javase/jdbc/index.html
- Database modules http://www.zorba-xquery.com/html/modules/db/couchbase

SQL Modules

Thank you!