Zorba Store API

1. Goals

- Make it possible for Zorba to query a variety of different data sources; e.g., Web pages in the browser, a relational database, an RSS stream. Provide an interface that allows these data sources to produce input data for Zorba.

- Shield all main memory management from the Zorba runtime. Have a single place that controls (and possibly optimizes) main memory management, disk IO, and garbage collection. Furthermore, shield all security and remote data access issues from the Zorba runtime system and its clients. Again, all these issues are encapsulated in the implementation of a store.

- Implement the complete XQuery data model. At the same time, provide implementors of stores the flexibility to use the freedom that the XQuery data model provides; e.g., only store lexical values, only store binary values, or store both.

- Provide functionality for collection and document management.

- Updates to existing data (preserving identity) and creation of new data (node construction).

- Make it possible to implement simple stores as well as sophisticated, highly-tuned stores (e.g., fully streamed, sophisticated pooling of strings, URIs, qnames, clever garbage collection and main memory management, chained IO from disk, etc.).

- Support versioning of data (branching and time travel). Nevertheless, provide well-defined semantics for stores that do not support such versioning.

- Support common XML APIs in order to load XML data into a store:
    1. DOM
    2. SAX
    3. Binary and plain XML as an iostrema
    4. ItemIterator

- Integrate stores that support multi-threading and transactions as well as stores that do not provide such a support. Stores that are not meant to be used in a concurrent computing environment (e.g., the Web browser) should not be penalilzed by a heavy pre-defined design to implement synchronization.

2. Assumptions

- Zorba engine (runtime + clients consuming XQuery results) will never hold more data than what fits in virtual main memory. Stores do not necessarily provide functionality to "swap" data in and out of main memory.

- There is only one store instance. All new data (nodes and atomic values) must be created using the ItemFactory of that store. Parsing of input data is tightly integrated into the store. If this assumption is violated, it is no longer possible to compare nodes (i.e., different stores have different implementations of node-ids).

  This assumption does not rule out that the store can keep items in different physical representations (e.g., implement items from a database differently than items from an RSS input). What it means is that the store is responsible to orchestrate these different implementations.

- All access to XML data must carried out using the „item“ API provided by the store. In other words, XML items are opaque to clients. In particular, comparisons between qnames and URIs must be implemented by the implementation of items inside the store.

- The store is the only point (at runtime) at which concurrent read/write access to shared data occurs. Iterator trees and contexts are potentially shared, but they are not updated at runtime. As a result, we do not need to deal with thread-safety and synchronization anywhere else in the runtime system, except in the store.

Todo (known problems, open questions):

- Store API currently does not provide a good way to integrate converters for input data; e.g., CSV -> XML mappings. In order to write such a converter, clients must write an ItemIterator in order to implement the conversion and pass that ItemIterator to the store.

- Indexing: The hooks to implement indexes at compile-time and run-time are not yet implemented.

3. Details

The code of the store API can be found in the following directory:

- .../zorba/xquery/store/api

Example implemenations of the API can be found in the following directories:

- .../zorba/xquery/store/dom
- .../zorba/xquery/store/native

The following (fully abstract) classes are important:

- Store (in store.h)
- Collection (in collection.h)
- Item (in item.h)
- ItemFactory (in item_factory.h)
- ThreadIdFactory (in store.h)
- ThreadId (in store.h)
- TransactionId (in store.h)
- StoreAccessContext (in store.h)
- Requester (in store.h)
- TimeTravel (in store.h)

3.1

From the user's perspective, a store provides the functionality to manage collections. That is, the store provides the methods to put a new collection into the store (using various APIs such as DOM, SAX, iostream, and ItemIterator), associate a URI with to a collection, and remove a collection from a store. All this functionality is provided in the „Store" API (see „store.h").
The „Collection" API allows to read (via an ItemIterator) and update a collection (e.g., add and remove items from the collection. The „Item" API allows to read XDM properties from specific items (e.g., the items returned by an ItemIterator as part of a query result). The „ItemFactory" API allows to create new data (i.e., items).

N.B. The URIs associated to collections are „internal" names used by the C++ API in order to implement the „doc" and „collection" functions. The real URI resolution and management of bindings of external variables is done by the C++ API which is implemented on top of the store.

In essence, thus, a „Store" is kind of „Map" from (internal) URI to collections and a collection is a „Vector" of items. Both can live in main memory, disk, or on some network device – all this is implementation defined and encapsulated by the store API. For implementers of a store, the most challenging part is the implementation of the „Item" and „ItemFactory" APIs.   There are two scenarios:

a.) Implement a new store from scratch: In this case, „items“ are implemented according to the requirements of the application and scenario for which the store is designed. For instance, it would be possible to design a non-updateable store that is highly tuned to process (transient) messages. Examples of stores that are built from scratch are the DOM and „native“ stores which a shipped with Zorba.

b.) Integrate an existing data source (e.g., an RDBMS): In this case, there is already an existing implementation of „items“ as part of the existing data source. In this case, „glue“ (or wrapper) items must be implemented which implement the „Item“ and „ItemFactory“ on top of the existing instances of the data source.

3.2 Threads

a.) We will introduce a new concept: thread-id. The "thread-id" is a store-defined object which will give the store the possibility to implement thread-safety and synchronize concurrent accesses from different threads. As part of the store API, there will be a "thread-id" factory - i.e., an API which creates new thread-ids. Calls to the thread-id factory must be atomic which must be guaranteed by the store implementation.

b.) A new thread will need to call the store's thread-id factory in order to get a new thread-id. This thread-id will be stored in the thread-local context of that thread. How this is done will be designed by Daniel. When a Zorba query is executed by this thread, then the iterators executed by this will need to have access to that thread-id.

c.) Every access to the store needs to get the thread-id as an additional parameter. Examples are calls to accessors of an item (item.h), creation of new items (item_factory.h), creation of new URIs (store.h), collection management and updates (store.h and collection.h). Using the thread-id, the store (e.g., Oracle) will be able to do all necessary synchronizations necessary. Clearly, different stores will have different ways to deal synchronize accesses from different threads. Some might not use it at all (e.g., the Internet Explorer Store).

Note: There is a n:1 relationship between thread and thread-id. That is, it is possible that all thread run using the same thread-ids. That is, the thread-id factory may return the same thread-id. For example, it is likely that the store to integrate Zorba into the Internet Explorer will do that because it does not support multi-threading (as far as I know).

3.3 Transactions

a.) Transactions are implemented by the store, too. Again, there might be stores (e.g., Internet Explorer) who do not provide any transaction support. They will have dummy implementations (e.g., "empty") for some of the transaction-related methods defined below.

The store API is ammended for transaction support just as for multi-threading. In the same way as (store-defined) "thread-id" objects are used in order to disambiguate concurrent accesses from different threads, a (store-defined) "transaction-id" object is used in order to disambiguate concurrent accesses from different transactions. A "transaction-id" object is passed as a parameter to (almost) all store methods in addition to the "thread-id" parameter. In order to simplify matters, Zorba aggregates the two ids into a StoreAccessContext. Calls will look like this: e.g., item.getName(StoreAccessContext&). New transaction-id objects are generated by the beginOfTransaction method provided by the store (see below). The "beginOfTransaction" method must be called with a valid "thread-id". (That is the only asymmetry between thread-ids and transaction-ids. A thread-id must be created first in order to create a transaction-id via the "beginOfTransaction() method.)

The implementation of transactions (e.g., long-term synchronization of operations of a transaction) is orthogonal to the synchronization of individual concurrent operations as part of multi-threading. There is an n:m relationship between transaction-ids and tread-ids. That is, it is possible that the implementation of a transaction forks several threads (e.g., in order to carry out multiple Web services calls in parallel and asynchronously). Likewise is it possible that a single thread executes several transactions sequentially.

b.) The store API supports the following operations for transactions:

- void beginOfTransaction(StoreAccessContext&)
    . StoreAccessContext.threadId is an input parameter
    . StoreAccessContext.transactionId is an output parameter

- void commit(StoreAccessContext & const)

- void abort(StoreAccessContext & const)

4. Example Stores

The following lists a number of examples that the designers of the store APIs kept in mind and that demonstrate the wide range of stores that are supposed to be pluggable into the Zorba architecture.

- Native, virtual main memory store: (shipped with Zorba)
- DOM store implementation (shipped with Zorba)
- A relational database (e.g., MySQL, SQLite, Oracle, ...)
- Client-server / networked access to remote data (relational database fall into this class)

- DOM in the Web browser (shipped as part of the XBrowser project)
- An XML message stream
- The file system

5. Store and the Rest of the World

- 1:n relationship between static context and a query
- n:m relationship between static context and dynamic context
- 1:n relationship between query and a query invocation
- 1:n relationship between dynamic context and query invocation
- **n:1 relationship between query invocation and store**

Treads and transactions:

- n:1 relationship between thread and thread-id
- 1:1 relationship between transaction and transaction-id
- n:m relationship between thread and transaction
- 1:n relationship between thread and query invocation
- n:1 relationship between thread and store
- n:1 relationship between transaction and store

Contact:

- David Graf: david.graf@28msec.com
- Donald Kossmann: donaldk@ethz.ch
- Tim Kraska: tim.kraska@inf.ethz.ch

First Version: September 9, 2007
Last Change: September 17, 2007