A Status Report on XXL—a Software Infrastructure for Efficient Query Processing

Michael Cammert, Christoph Heinz, Jürgen Krämer, Martin Schneider, Bernhard Seeger
Department of Mathematics and Computer Science, University of Marburg, Germany
http://www.mathematik.uni-marburg.de/DBS/xxl

Abstract

XXL is a Java library that contains a rich infrastructure for implementing advanced query processing functionality. The library offers low-level components like access to raw disks as well as high-level ones like a query optimizer. On the intermediate levels, XXL provides a demand-driven cursor algebra, a framework for indexing and a powerful package for supporting aggregation. The library is publicly available under GNU LGPL and comes with a full documentation.

1 Introduction

This paper describes the most important components of XXL, the eXtensible and fleXible Library for efficient query processing [1, 2, 3]. Multiple reasons have driven the design and implementation of the library:

- Many of the algorithms developed for query processing have been implemented in an ad-hoc manner. The software design of these algorithms is poor and therefore, their application is quite complicated and limited to specific (operating) systems. Furthermore, the modification of existing code is more complicated since the documentation is often not complete or even not available.
- XXL should support experimental evaluations within a uniform testbed that is freely available. It is difficult to compare two access methods, for example, when the underlying platform is not the same. The usage of different programming languages and compilers already results in substantial differences in the runtime. Consequently, most of the comparisons are based on a simplistic computing model where the number of I/Os is the only criterion.
- A more ambitious goal has been that XXL could serve as a repository for algorithms and use-cases. We recognized that many algorithms are published in papers, but only a few implementations are freely available. Wouldn't it be great to have a collection of algorithms implemented under a uniform framework? At least in our research group, XXL has served as a repository where the code of our most important research results is transparently present.

The XXL project started four years ago. One of our first design decisions implied to use Java as the underlying programming language. With respect to our goals mentioned above, we were convinced that the advantages

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

of Java will outweigh its disadvantages. In particular, XXL benefits from the rich functionality available in other Java libraries like the API of the SDK [4] and Colt [5]. We also agreed that the design of the library should be based on popular design patterns [6]. This improves the readability and reusability of the code, and leads to a comprehensive documentation. XXL uses design patterns like factory, iterator and decorator. In order to improve code reusability, functional concepts had a strong influence on the design of the library. Java's anonymous classes are an excellent mechanism to provide functional abstraction in Java with very little overhead.

Another important aspect of our library is that classes should be well documented and equipped with use-cases. These are important for inexperienced users to get familiar with the mechanisms and the handling of the library. A simple use-case is therefore attached to every class in its corresponding main method. For more complex application scenarios, we also provide use-cases in separate classes.

The rest of the paper gives a brief outline of the functionality of XXL, placing emphasis on the new concepts that have been developed recently. In Section 2, we first provide an introduction of the basics like our functional approach, containers and cursors. The principles for query processing like indexing and join processing are presented in Section 3. Another new important component is our native XML storage. In Section 4, advanced concepts are presented where metadata has to be taken into account. In particular, we introduce our object-relational package and show how to provide query optimization within our library.

2 Basic Components

2.1 Functions and Predicates

Functional abstraction is a powerful mechanism for writing compact code. Since functions are not first class citizens in Java, XXL provides the interface `Function` which has to be implemented by a functional class. A new functional object is declared at runtime using one of the following methods:

1. An anonymous class is implemented by extending `Function` and overriding a method `invoke` that should contain the executable code. An example for declaring a new function is given as follows:

   ```
   Function maxComp = new Function() {
     public Object invoke (Object o1, Object o2) {
       return (((Comparable) o1).compareTo(o2) > 0) ? o1 : o2;
     }
   }
   ```

2. The method `compose` of a functional object can be called to declare a new function by composition of functional objects.

The code of a functional object is executed by calling the method `invoke` with the expected number of parameters. Note that functional objects in XXL may have a status and therefore, are more powerful than pure mathematical functions.

Due to its importance in database systems, we decided to provide a separate interface (`Predicate`) for Boolean functions. This improves the readability of the code as well as its performance since expensive casts are avoided. Relevant to databases are particularly predicates like `exist` for specifying subqueries and the predicates for supporting a three-value logic.

2.2 Containers

A container is an implementation of a map that provides an abstraction from the underlying physical storage. If an object is inserted into a container, a new ID is created and returned. An object of a container can only be retrieved via the corresponding ID. Since a container is generally used for bridging the gap between levels of a storage hierarchy, mechanisms for buffer management are already included in a container.

There are many different implementations of containers in XXL. The class `MapContainer` refers to a container where the set of objects is kept in main memory. The purpose of this container is to run queries fast in memory and to support debugging. The class `BlockFileContainer` represents a file of blocks, where a block refers to an array of bytes with a fixed length. This is for instance useful when index-structures like R-trees are implemented.

Java does not support operations on binary data and therefore, a block has to be serialized into its object representation. Java's serialization mechanism is however not appropriate since it has to be defined at compile time. It is also too inflexible because there is only at most one serialization method for a class. XXL overcomes these deficiencies by introducing the class `ConverterContainer` that is a decorated container, i.e., an object of this class is a container and consists of a container. In addition, this class provides a converter that transforms an object into a different representation. A `BufferedContainer` is also a decorator. Its primary task is to support object buffering in XXL.

In order to run experiments on external storage without interfering with the underlying operating systems, XXL contains classes that support access to raw devices. There are two possibilities:

1. The class `NativeRawAccess` offers native methods on a raw device. By using `NativeRawAccess` the class `RawAccessRAF` extends the class `java.io.RandomAccessFile`, which is the storage interface of `BlockFileContainer`.

2. XXL offers an implementation of an entire file system that runs on a raw device. This is able to deliver files as objects of a class that extends `java.io.RandomAccessFile`. Therefore, an object of the class `BlockFileContainer` can store its blocks in files of XXL's file system.

2.3 Cursor

A cursor is an abstract mechanism to access objects within a stream. Cursors in XXL are independent from the specific type of the underlying objects. The interface of a cursor is given by:

```
interface Cursor extends java.util.Iterator {
  Object peek();
  void update(Object o);
  void reset();
  void close();
}
```

A cursor extends the functionality of the iterator provided in the package `java.util`. The `peek` method reports the next object of the iteration without changing the state of the iteration. A call of `reset` sets the cursor to the beginning of the iteration. The method `close` stops the iteration and releases resources like file handles. The method `update` modifies the current object of the iteration.

XXL offers an algebra for processing cursors, i. e., there are a set of operations that require cursors as input and return a cursor as output. We distinguish among three kinds of cursors:

- *Input cursors* are wrappers for transforming a data source into a cursor. For example, XXL provides an input cursor for transforming `java.sql.ResultSet` into a cursor.
- *Processing cursors* are the ones that modify the input cursor. Examples for such cursors are `Join`, `Grouper`, `Mapper` whose semantics are similar to the ones of the corresponding relational operators.
- *Flow cursors* do not change the objects within the input stream, but they are restricted to change the underlying data flow. For example, an instance of the class `TeeCursor` duplicates the input cursor.

3 Query Processing

3.1 Indexing

One of the most important packages of XXL is `indexStructures` that consists of a high-level framework for index-structures. The purpose of this package is twofold: First, it contains many different index-structures that are ready-to-use. Second, the implementation of new ones should be simplified.

Let us give an example for using an index-structure like an M-tree [7]:

```
MTree mTree = new MTree(MTree.HYPERPLANE_SPLIT);
mTree.initialize(getDescriptor, container, minCap, maxCap);
```

The first step is to call a constructor. In this example we used the one with a parameter where the split strategy is specified. The second step is an initialization of the M-tree. The first parameter `getDescriptor` refers to a functional object that computes a so-called *descriptor* for a given data item. In case of the M-tree, a descriptor corresponds to a bounding sphere. Our M-tree is able to manage any kind of objects as long as such a functional object is available. The next parameter is the container object which is responsible for managing the nodes of the tree. The other two parameters specify the minimum and maximum number of items within a node. Thereafter, the tree is ready for receiving operations like insertions and queries.

An implementation of a new index-structure requires a fundamental understanding of our framework that is a direct implementation of grow-and-post trees [8]. An index-structure is primarily determined by the inner class `Node`, which does not only describe the structure of the tree nodes, but also provides essential functionality for splitting and searching. The main task when implementing a new index-structure is to code a specialized class for the nodes. For example, a function is required to serialize a node of an index-structure. More details about the implementation can be found in our Java sources [3], where B-trees are probably the best starting point.

3.2 Join Processing

Joins are among the most important operators in a database system. While relational systems basically rely on equi-joins, new applications like spatial databases require new types of join predicates. The goal of our join processing framework was to provide a single implementation with the intention to support a bunch of different join predicates efficiently. Furthermore, our framework is sufficiently generic to cover both sort-merge joins and hash-based joins. It keeps a small subset, a so-called *sweep-area*, for each input source in main-memory where the join is processed on. Elements from the input are inserted into the associated sweep-area one by one. After the insertion of an element, the other sweep-area is checked for join partners. A sweep-area can periodically reorganized to remove the elements not producing join results anymore.

The interface `SweepArea` is the top class of all sweep-areas in XXL. The most important functionality looks as follows:

```
public interface SweepArea {
  public void insert(Object o);
  public void reorganize(Object curStatus, int id);
  public Iterator query(Object o);
  ...
}
```

The operations refer to the basic steps of join processing as described above. Note that every input has a unique identifier which has to be specified when calling `reorganize`. There is a large number of different classes that implement the interface `SweepArea`. We refer the interested reader to the documentation of XXL [3].

A join in XXL is called by the following statement:

```
Iterator it = new Join(input1, input2, HashBagSweepArea.FACTORY_METHOD,
                       Tuplify.DEFAULT_INSTANCE);
```

The first two parameters refer to the two input sources. The third parameter is a factory method for creating a sweep-area. In our example, the sweep-area is organized as a hash-table [15]. The last parameter is a functional object that specifies how to construct the output tuple of the join.

If a user of XXL is interested in implementing a new kind of join, she/he basically has to implement an appropriate class that satisfies the interface `SweepArea`. This is substantial easier than implementing a join from scratch.

3.3 Aggregation

Aggregate operations are important in large database systems to deliver a quick overview of the response set. In contrast to a relational DBMS, XXL supports functions as results of aggregate operations. This allows returning a histogram or other more advanced statistical data structures directly to the user (without producing an intermediate relation). In the following, we briefly describe the basic structures of our package `statistics`.

This package is based on a generic aggregator cursor that applies a user-defined functions to aggregate the objects of a given iterator. This cursor returns the intermediate value of the aggregate among the input that has been consumed so far. The final value can be reported by a call to *aggregator.last()*, which consumes the entire iterator. An example of such an aggregator is given below:

```
Aggregator aggregator = new Aggregator(
  new RandomIntegers(100, 50),
  new Function () { // the aggregation function
    public Object invoke (Object agg, Object next) {
      return (agg == null) ? next : maxComp.invoke(agg, next);
    }
  }
);
```

In our example, the source consists of 50 random integers in the range [0,100). The anonymous function computes the maximum of two elements where `agg` represents the aggregated value up to the previous element of the input and `next` is the current element of the input.

Our statistics package provides different implementations of selectivity estimators with histograms and kernels as well as estimators based on query feedback. We refer the interested reader to our documentation [3], in order to get more familiar with these concepts.

3.4 XML Storage

XXL contains functionality for processing queries on XML data. In addition to wrappers that transform XML input into Java objects, XXL also provides a class that implements native XML storage. A brief description of this class will be given in the following.

The native storage of XXL is an implementation of Natix [9] that performs quite similar to a B-tree. The basic idea is to keep adjacent nodes of an XML-object physically close to each other in one page of the tree, in order to support insertions and updates efficiently. An insertion of an XML node first determines the page where it has to be stored. This might result in an overflow of the page which then has to be split into two. This triggers a split of the XML document into smaller pieces which fit into pages.

4 Advanced Features

In this section, we present two packages of XXL that goes beyond the pure query processing techniques presented so far. Both of these packages rely on the availability and maintenance of metadata, whereas the functionality is inherited from the core packages. In order to deliver metadata, a class has to satisfy the interface

`MetaDataProvider` that only offers the method `getMetaData`. The class `MetaDataCursor` combines for example the two interfaces `MetaDataProvider` and `Cursor`.

Relational Connectivity XXL's package relational offers the functionality for processing on object-relational data sources. The functionality of the package is similar to the one of cursors, but the operators are enhanced by the corresponding metadata. In addition, the operators are processing tuples rather than Java objects. Consequently, there are operators for join processing, grouping, projection, ...

An important functionality of this package is the availability of wrappers for transforming an object of the class `java.sql.ResultSet` into an object of class `MetaDataCursor` and vice versa. This enables us to process data from relational sources directly without storing them in a local database. Database systems like Cloudscape have increased the functionality of SQL by accepting cursors in the from-clause. This yields an easy approach to extending the functionality of a database system. In [1], we presented an implementation of a similarity join in Cloudscape using XXL's join operator.

Query Optimization The recent version of XXL also includes a query optimizer for transforming relational operator trees into more efficient ones. In analogy to the optimizer of a DBMS, we first check for semantically correctness of the operator tree. Then, the optimizer starts transforming the operator tree by using a set of rules and a cost model. Eventually, the optimizer selects the specific algorithms for the implementation of the operators.

Since our query optimizer is part of a library, we require metadata being attached to data sources, operators, algorithms, functions and predicates. For operators, there are the interfaces `OperatorInputMetaData` and `OperatorOutputMetaData`, which extend the functionality of `java.sql.ResultSetMetaData`. These interfaces include methods that estimate the selectivity of an underlying operator and its costs. Our functional metadata (`FunctionMetaData`) offer methods that specify the attributes of the input stream. Metadata on predicates also return an estimation of the predicate's selectivity. Moreover, the algorithms considered in the physical optimization step have to deliver metadata like the associated logical operator.

Important to the design of the optimizer was its extensibility and flexibility. In our architecture, it is easy to add new predicates, operators and algorithms. Moreover, the underlying cost model is not fixed and might be replaced by a different one. As an extra feature, we support an XML format for queries, i. e. operator trees can be transformed into XML and vice versa.

5 Related Work

There has been only little work on the design and development of query processing libraries in the database literature up to date. Most of the work published in the database community presents a system-oriented architecture.

Our work has been largely inspired by the pioneering work of Graefe and his Volcano system [10]. Both, Volcano and XXL, use a tree-structured query evaluation strategy, represented by algebra expressions, that is used to execute queries by demand-driven dataflows. Volcano already used so-called *support functions* for manipulating individual data objects in the dataflow. XXL however goes beyond the functionality of Volcano. First, it offers a richer query processing infrastructure, many different index-structures and more support for statistics. Second, XXL also contains wrappers for diverse data sources. Third, the object-oriented design of XXL allows an easy extension of its functionality.

The work on GiST [11] is closely related to our indexing framework, but GiST is actually a system that is tightly coupled with its storage system. The focus of GIST is only on index-structures, whereas other functionality is missing. It is notable that the grid-file implementation [12] had already great abstraction mechanisms like iterators.

The design of libraries is more related to the area of algorithms and data structures, where libraries like LEDA [13] are well known. The focus of LEDA is more on data structures for main memory rather than on the management of very large data sets. Many of the abstraction mechanisms like functional classes are not available in LEDA. TPIE [14] is designed to assist programmers in writing high performance I/O-efficient programs. However, the operators of TPIE cannot pass data directly between each other, but have to use a temporary storage area. In addition, TPIE does not represent a pure library, because it relies on a special memory manager for organizing the physical memory. This also implies that TPIE is not platform independent.

6 Conclusions and Future Work

XXL is a query processing library implemented in Java that includes the most important ingredients for efficient query processing. The design of the library was determined by two goals: the functionality of XXL should be extended easily and XXL should be flexible enough for being customized fast to specific problems. Due to its powerful methods, XXL is also an excellent platform for experimental work. Coding of new algorithms and data structures requires substantial less time than beginning from scratch.

XXL is a live project! We are currently working on improving our indexing framework and strive for a realization of a processing algebra on data streams.

Acknowledgements We are grateful for the great contributions of the other and previous members of the database group to the current version of XXL. This work has been supported by the German Research Society (DFG) under SE 553/2-2 and SE 553/4-1.

References

[1] J. van den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, B. Seeger: XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. VLDB Conf. 2001: 39-48

[2] J. van den Bercken, J.-P. Dittrich, B. Seeger: javax.XXL: A prototype for a Library of Query processing Algorithms. SIGMOD Conf. 2000: 588

[3] The XXL Project, http://www.mathematik.uni-marburg.de/dbs/xxl, 2003

[4] JavaTM 2 Platform, Standard Edition, v 1.4.1 API Specification, http://java.sun.com/j2se/1.4.1/docs/api/, 2002

[5] The Colt Distribution - Open Source Libraries for High Performance Scientific and Technical Computing in Java, http://hoschek.home.cern.ch/hoschek/colt/V1.0.3/doc/index.html, 2002

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley. October 1994.

[7] P. Ciaccia, M. Patella, P. Zezula: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. VLDB Conf. 1997: 426-435

[8] D. Lomet: Grow and Post Index Trees: Roles, Techniques and Future Potential. Proc. Symp. on Spatial Databases 1991: 183-206

[9] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann: Natix: A Technology Overview. Web, Web-Services, and Database Systems 2002: 12-33

[10] G. Graefe: Volcano - An Extensible and Parallel Query Evaluation System. TKDE 6(1): 120-135 (1994)

[11] J. Hellerstein, J. Naughton, A. Pfeffer: Generalized Search Trees for Database Systems. VLDB Conf. 1995: 562-573

[12] K. Hinrichs: Implementation of the Grid File: Design Concepts and Experience. BIT 25(4): 569-592 (1985)

[13] K. Mehlhorn, S. Näher: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press 1999

[14] L. Arge, O. Procopiuc, J. Vitter: Implementing I/O-efficient Data Structures Using TPIE. ESA 2002: 88-100

[15] A. Wilschut, P. Apers: Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77