XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries*

Jochen van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer,
Tobias Schäfer, Martin Schneider, Bernhard Seeger

Philipps-Universität Marburg, D-35032 Marburg, Germany
xxl@informatik.uni-marburg.de

Abstract

Today's DBMS are still too inflexible to adapt fast enough to the query processing needs of new applications [CW00]. Instead of using the cumbersome functionality of a monolithic DBMS, it is not uncommon that users implement their own functionality on top of the system. For such a scenario, the implementation would be substantially facilitated through a powerful library.
This paper introduces XXL (eXtensible and fleXible Library), a high-level, easy-to-use, platform independent Java library supporting the implementation of new query functionality. XXL provides framework implementations as well as toolboxes whose applications are independent from the underlying data types and data structures.
We introduce the most important concepts of XXL and discuss different application scenarios where XXL has been used recently. In particular, we show how an implementation of an efficient algorithm for processing spatial joins can easily be integrated into a commercial database system (Cloudscape).

1 Introduction

Since the last decade object-relational DBMS have been emerged in order to support new application areas where complex data has to be managed in a user-suitable way. Today's DBMS provide different mechanisms to incorporate new functionality into the systems. In general, DBMS vendors offer specific languages which allow the implementation of user-defined functionality. Recently, they also support user-defined functionality being implemented directly in Java. This code can be reasonably fast since Java Virtual Machines (JVM) are part of the kernel of most DBMS. It seems that this approach is very appealing to users of DBMS and therefore, we expect an increase of its practical relevance in the near future.

In this paper, we address the problem of designing a Java library that facilitates the implementation of user-defined functionality. As our approach to this problem we present XXL (eXtensible and fleXible Library). XXL is freely available under the terms of the GNU Lesser General Public License [Dat01]. The library consists of the following components:

1. The cursor package provides an algebra of the most important query operators whose implementations are demand-driven whenever it is effective. The operators of the algebra require that the input as well as the output satisfy an iterator interface. In order to support the import of external database sources, XXL also contains an enriched algebra based on Java's interface *ResultSet*. Moreover, some database vendors [Inf01] have extended SQL in such a way that a *ResultSet* can be used within a SQL statement. This allows a seamless and easy integration of application code into SQL.

2. XXL offers a rich infrastructure of external data structures which are helpful for the implementation of new database functionality. In addition, XXL provides a very flexible buffer mechanism. In contrast to system-based approaches, specific data structures of XXL like the buffer can be used without knowledge of the other parts of XXL.

3. A broad class of index structures has been implemented as a framework in XXL, where many default implementations (R-tree, M-tree, X-tree) already exist and the implementations of new struc-

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

*This work has been supported by grant no. SE 5532-1 from DFG.

tures are largely facilitated. For example, different types of bulk operations on indexes are already efficiently implemented as part of the generic code. Moreover, the framework of index structures in XXL is not an isolated component, but is fully embedded into the library.

There is another purpose of XXL that is of great importance at least from a research point of view. We propose XXL as a library for conducting experiments. Results obtained from experiments are important indicators for evaluating the performance of new query processing techniques. We however feel that the quality of the experimental work has to be considerably improved. Due to the large gap between a high-level algorithmic description and a low-level implementation, we argue that in addition to the publication of the results, the code should be made publicly available, too. However, we assume that the quality of the code is generally poor. Therefore, a well documented and open library like XXL would be an ideal infrastructure for experiments. The library could provide reference implementations as well as building blocks (e.g. external algorithms and data structures) that are used in an experimental comparison of competitive approaches.

Research in the database area is largely driven by the idea of building a system, whereas the development of a library is exceptional. Volcano [Gra94] is probably the work closest to ours. It provides a data model independent query processing functionality similar to XXL, but also addresses the problem of query optimization and parallelism. Volcano can be viewed as a hybrid between a system and a library. Iterator-based processing is also addressed in object-oriented query languages like OFL [GMP95] and functional ones [BF79]. According to indexing, there has been only a few approaches. The grid-file system [Hin85] is an exceptional example of an elegant implementation of an index structure, but it is limited to one specific structure. GIST [HNP95] is basically an extensible system for indexing which supports the implementation of R-trees and related structures. The development of libraries is more popular in the algorithmic area. There is also an increasing interest on external problems where the data cannot be kept in main memory. TPIE [VV96] and Leda-SM [AC99] are among the most well known libraries with a rich source of external algorithms. However, both libraries do not support demand-driven query processing as it is required in a DBMS. Moreover, index structures are not supported in the libraries.

The paper is organized as follows. In Section 2, the implementation of the functional paradigm is outlined. In Section 3, the cursor algebra of XXL is introduced. A selection of the most important operators is presented and, it is shown how the functionality of a cursor can be extended to satisfy the interface *java.sql.ResultSet*. In Section 4, the different collections currently available in XXL are presented. The I/O infrastructure including buffering and different types of containers is discussed in Section 5. XXL's framework of index structures is presented in Section 6. Section 7 discusses the most important features of the spatial package. In Section 8, we present two use-cases of XXL where the one shows the connectivity of XXL and a commercial database system and the other presents approaches to spatial databases. Finally, Section 9 concludes the paper.

2 Functions

Functions are an extremely important concept to encapsulate abstractions. This generally holds, but we found that the functional paradigm is of utmost importance when designing and implementing a query processing library like XXL. So far, the functional paradigm received little attention in the database community, at least with respect to implementation issues of query processing algorithms. Therefore, we first give a brief review on the implementation of the functional concept in XXL.

Since Java does not support a function as a first-class citizen nor higher-order functions [Hug89], XXL contains an abstract class *Function* where these concepts are provided in an elegant way. A new function can be implemented by defining a subclass of *Function*. The desired functionality is brought in by redefining one of its abstract `invoke` methods. In order to reduce the number of explicit classes, subclasses of *Function* are frequently implemented as anonymous classes. This is a specific concept in Java where the implementation of a subclass occurs at the same position in the code where an object of the class is created. A call of the `invoke` method of the object eventually causes the evaluation of the target function.

Moreover, the class *Function* also supports higher-order functions in a similar way as it is known from Smalltalk's blocks. Within the class *Function* the method `compose` allows to create new functions through compositions of other functions at runtime. This is easy to support in Java because of its feature of anonymous classes. For example, consider that the mathematical functions `sin`, `cos` and `div` are implemented as objects of the class *Function*. Then,

```
Function tan = div.compose(sin, cos);
```

defines a new function whose evaluation is simply initiated by calling `invoke`. Moreover, our approach allows that a function object may have a state (similar to C where the static variables of a procedure survive a call). This powerful feature is used in XXL, for example, when aggregate functions are implemented in an incremental manner where the partial results are delivered through an iterator.

Since predicates are functions with a high relevance to query processing, we decided to introduce special

classes for creating predicate objects. An arbitrary WHERE clause of an SQL statement can be expressed in XXL as an object of the class *Predicate*. There are no limitations on the complexity of a predicate which also can cope with subqueries.

3 Cursors

A cursor is an abstract concept for manipulating objects within a stream. The cursors provided in XXL are independent from the specific structure of the objects as well as from the kind of storage representation of the underlying stream. This property is closely related to the notion of *physical independence* which is one of the key concepts of database systems. XXL offers an algebra of cursors which is suitable for defining complex queries in a declarative fashion similar to SQL. In addition, our cursor algebra is also beneficial for a very compact implementation of low-level core functionality (e.g. index structures).

In the following, we start with a brief review of the concept of iterators which is already available in the Java API. Thereafter, we present the interface *Cursor* of XXL that corresponds to a specialized kind of iterators.

3.1 Iterators

Cursors are closely related to iterators which are well known from different areas in computer science. Volcano [Gra94] is probably the most well known project within the database community where iterators have extensively been used for processing queries. Although iterators have been used within Volcano the semantics of the different iterators have been not published to the best of the authors' knowledge. Iterators also occur in large libraries like STL and the Java API. Iterators are also used in the algorithmic community [Wei01].

In the following section, we present how cursors and iterators are used for processing queries in the XXL library. In order to be compatible to the Java API, XXL adapts the iterator functionality from the Java API. The *Iterator* interface of the Java API consists of the following three methods:

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

The method `hasNext` asks whether the underlying stream contains another object, whereas `next` retrieves the next object from the stream. These two methods have to be implemented by a class that satisfies the *Iterator* interface. The third method is a so-called *optional* method which is actually not required to be implemented. This design-principle is frequently used within the Java API. If `remove` is implemented, a call simply removes the last object returned by an iterator.

3.2 Cursor Interface

The interface *Cursor* extends the functionality of an iterator by the following methods:

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

The `peek` method shows the next object of the iteration without changing its state. By calling `reset` the iteration is started again from the beginning. Important to cursors is the method `close` which allows to release resources (e.g. net-connections) associated with a cursor.

Our intention was to define a cursor algebra where the different operators of the algebra take a cursor as an input and deliver a cursor as an output. In addition, our algebra also contains a set of *input operators* where the input type does not fulfill the *Cursor* interface. A complex query is then represented as an operator tree where the nodes correspond to specific operators. The leaves of the nodes are input operators, whereas the internal nodes are *processing operators*. Processing operators can easily be customized by passing functional arguments. This idea is closely related to *support functions* [Gra94] and *algorithmic generators* [Wei01]. From a functional point of view [Hug89], a complex query is simply a higher-order function that can be defined dynamically during runtime.

3.3 Input Operators

There are different types of input operators which are briefly described in the following. Due to space limitations we only discuss a few examples in this paper and refer the interested reader to the online documentation of XXL [Dat01].

XXL provides a large set of input operators that are wrappers of specific sources. We just want to mention two of the most important ones:

- IteratorCursor: java.util.Iterator $\mapsto$ Cursor
- ResultSetCursor: java.sql.ResultSet $\mapsto$ Cursor

The first operator transforms an iterator into a cursor. This is important for processing sources which only satisfy the *Iterator* interface of the Java API. The second operator is used for processing sources that satisfy the *ResultSet* interface. This operator can particularly be useful for importing tables of relational databases.

XXL also provides virtual sources, e.g. the class *RandomIntegers* returns randomly created Integer objects on demand. Hence, the output of such an operator is not limited in size. In general, XXL supports query processing on infinite sources.

3.4 Processing Operators

Processing operators require at least one cursor as an input. We distinguish operators whose processing is *demand-driven*, *strict* and *hybrid*.

In case of demand-driven processing, an operator can deliver objects to its output without having processed its entire input, whereas in case of strict operators the entire input has to be consumed.

We introduce another group of so-called hybrid operators which are processed in (at least) two phases where the first phase is strict, but the second phase is demand-driven. One of the design goals of the library was to strive for demand-driven implementations whenever it is effective. Therefore, almost all operators are demand-driven and only a few of them are strict and hybrid.

3.4.1 Demand-driven Operators

In the following we present the most important demand-driven operations of XXL. We put our focus on those operators that are of particular interest for processing database queries.

Let us first consider those demand-driven operators whose input consists of an iterator and a function and whose output is again a cursor. Among these operators are *Filter*, *Mapper* and *Aggregator*. A filter simply delivers the objects from the input iterator which satisfy a user-defined predicate. In order to illustrate the declaration of a filter let us consider a simple example where `employeeList` is a list of objects of the class `Employee`. The following code deletes all persons older than 30 from the list:

```
removeAll( new Filter( employeeList.iterator(),
    new Predicate () {
      public boolean invoke(Object employee) {
        return ((Employee)employee).getAge() > 30;
      }
    }
)
```

The static method `removeAll` requires as its input an object of type iterator. In our example, the iterator corresponds to a filter. The constructor of the filter requires two input parameters where the first one is again an iterator and the second one is a function that implements a simple predicate. Note, that the semantics of the expression is equivalent to the following SQL command:

```
DELETE
FROM Employee
WHERE Age > 30
```

An object of the class *Mapper* maps each object of the input iterator to a new output object. The second parameter of a mapper is again a function which implements the mapping. The relational projection is obviously a special case of a mapper.

Another interesting operator of XXL is the *Aggregator*. An aggregator computes for an input iterator an aggregate (e.g. sum or average), but it also delivers the partial results after having consumed an object of the input iterator. This is interesting for computing approximations of aggregates as it has been proposed in [HHW97].

A second class of demand-driven operators provides powerful mechanisms to change the structure of an iterator. The class *Grouper* provides objects that transform an iterator into a cursor where each result of the operator is again a cursor. The constructor of a grouper requires a predicate comparing two consecutive objects of the input iterator. If the predicate is satisfied, the objects are in the same group. Otherwise, the second object belongs to a new group. Consequently, a grouper partitions the input iterator into disjoint groups. The class *Grouper* of XXL is again implemented in a demand-driven fashion. Note, that a grouper is more powerful than the GROUP BY operator known from relational DBMS (which is only applicable in combination with an aggregation). It is semantically equivalent to the *nest* operation known from object-oriented DBMS [AB95]. In addition, the class *Sequentializer* of XXL provides the inverse operator (*unnest*) of a grouper.

In order to combine the object of two iterators, XXL supports the computation of the Cartesian product. Different types of join operators are also efficiently implemented in XXL whose predicates can be relational, spatial and similarity-based. A more detailed discussion of the spatial functionality in XXL is given in Section 7.

3.4.2 Strict and Hybrid Operators

Almost all of the operators in XXL are implemented in a demand-driven manner, but there are a few exceptions where the processing is strict. The implementation of an operator is *strict*, when the total output of an operator is computed during its initialization. In general, this requires that the output is temporarily stored on disk. During the next-phase of a strict operator a result is delivered to the caller without any additional processing cost. In contrast to a strict operator, a hybrid operator shifts a substantial part of its computation from its initialization phase to the demand-driven phase.

The class *HashGrouper* can be viewed as a special form of a grouper whose implementation is strict. It partitions its input into hash buckets. A call of next delivers an iterator of a hash bucket to the caller. An example of a hybrid operator is the class *MergeSorter* that offers external sorting of the input iterator based on the sort-merge paradigm and *ReplacementSelection* [Knu73]. As suggested in [Gra94], our merge sorter delays the final merge to the next-phase.

3.5 ResultSet versus Cursor

Java's interface *ResultSet* provides access to database relations. A result set is usually generated by issueing a query to the database via a *Java Database Connectivity (JDBC)* call. A result set maintains a reference pointing to its current row of data. The `next` method moves the reference to the next row. Since this method returns `false` when there are no more rows in the result set, it can be used in a while loop to iterate over the result set. Thus, a result set and a cursor share both the same principle of demand-driven dataflow. The difference however is that the *ResultSet* interface provides a kind of type system. This mechanism is based on meta data information for retrieving column values from the current row. A JDBC driver attempts to convert the underlying data to the Java type specified by the meta data and returns a suitable Java value. XXL's *Cursor* interface does not contain such methods due to the lack of meta data information. In the future, a new cursor will be developed that wraps an arbitrary cursor by adding meta data information concerning the contained objects using the Java reflection mechanism. Thus, the integration of arbitrary data sources into database tables becomes seamless due to the possibility to use our cursors as result sets. A practical example of this integration is demonstrated in Section 8.1. Note, that XXL already contains a package providing a cursor-based implementation of the relational algebra's physical operators (*xxl.relational*).

4 Collections

xxl.collections is a package that contains interfaces and classes for storing objects. The Java API already contains the interface *Collection*. However, the *Collection* interface is a *fat* interface, i.e., it specifies many optional methods. This is not desirable because users are forced to implement methods that are not required. Another drawback of Java's *Collection* interface is the lack of a `close` method. This functionality however is needed to release resources when a collection is based on an external resource, e.g. a file or a JDBC connection. Furthermore, many important collections like *Queues* are not part of the Java API.

For these reasons, we provide a new, redesigned set of *collection* interfaces. The interface is broken up into several *small* interfaces containing only a few essential methods. This allows a fast implementation of these interfaces.

The package *xxl.collections* depends on three fundamental interfaces: *Bag*, *Queue* and *Container*.

4.1 Bags

A bag is an implementation of a mathematical multiset, i.e., it represents a set of objects that might contain duplicate objects. No order is specified on the objects of a bag. Objects can be added to the bag, but there is no efficient way to check whether a bag contains an object or not. The only way to access objects of the bag is to inspect its entire content using an iterator. The bag supports the insertion of single objects as well as bulk insertion.

For those cases when a bag is able to guarantee an order on its objects, it should implement a marker interface. The marker interface contains an additional method returning a cursor which iterates over the objects of the bag in the specified order (e.g. FIFO, LIFO, Priority). The order determines the relationship between insertion and iteration over the bag's objects. The package *xxl.collections* contains several implementations of the *Bag* interface, e.g. *ArrayBag*, *DynamicArrayBag* and *ListBag*.

4.2 Queues

A queue represents a multiset that behaves like an ordered bag. In comparison to a bag the queues' objects are removed when they are accessed. The most important implementations of the *Queue* interface are *ArrayQueue*, *Heap*, *ListQueue*, *StackQueue* and *RandomAccessFileQueue*.

4.3 Containers

A container is an implementation of a map. A container generates a unique *ID* for each object and stores a tuple, namely $(ID, object)$. 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. A container supports the removal and update of objects as well as the retrieval of its elements through an iterator.

The main intent of the abstract class *Container* is to provide buffered access to objects (typically stored in secondary memory). Therefore, every access method provides a flag that allows to fix and unfix the accessed object (to request the insertion of objects into the buffer and the removal, respectively). Furthermore, buffered objects can be flushed, i.e., the buffer is forced to write back any modified object to its underlying container.

The package *xxl.collections* contains only a few implementations of the *Container* interface, e.g. a *MapContainer*. Most of the interesting containers are designed for external data management. These implementations (e.g. *BlockFileContainer*, *BufferedContainer* and *BufferedRandomAccessFileContainer*) are in the package *xxl.io*.

5 I/O

The package *xxl.io* is a set of interfaces and classes dealing with external resources. It contains classes for serializing objects in order to read or write them. In addition, this package provides implementations of collections that manage objects in external memory

and input operators that read objects from external sources.

5.1 Converters

An important mechanism required for I/O operations is serialization, i.e., the conversion of objects to streams of binary data. The Java API already provides a serialization mechanism. However, this mechanism has drawbacks with respect to the usage in databases. When applying Java's serialization mechanism, additional meta data is written to the output stream. Java requires this meta data for deserialization. Note that the size of the meta data is not predictable.

This behaviour is generally not intended in the context of databases for the following reasons. First, storing meta data may cause a considerable storage overhead. Second, the size of a serialized object depends on the time of serialization. This is a particular problem when the number of objects fitting in a block of fixed size has to be computed. Third, objects can only be serializable when all of the subobjects are also serializable. For these reasons, XXL provides the interfaces *Convertable* and *Converter* as an alternative to the Java API's serialization mechanism.

Convertable classes have to implement a `read` and a `write` method. These methods offer complete control over the format and contents of the data being serialized. If a class does not implement *Convertable*, a converter object can still be used for serialization. Hence, a converter separates a class from its converting mechanism. The package *xxl.io* contains a large set of converters supporting primitive data types and arrays. Note that Java's serialization mechanism can also be used in conjunction with XXL.

5.2 Buffers & Blocks

Buffering is an important technique for making I/O operations efficient. In general, buffering means keeping some blocks of serialized data in main memory in order to prevent additional I/O if data is required again. However, it is also desirable to buffer deserialized objects in order to avoid expensive conversions. XXL provides a buffer class which allows the creation of multiple buffers. A buffer contains a certain number of slots. Every slot is able to store a reference to an object that is kept in main memory. Because of storing object references instead of serialized data, the associated data of a slot does not have a fixed size.

There is an m:n-relationship between buffers and containers where each instance corresponds to an object of the class *BufferedContainer*. An object of the class *BufferedContainer* satisfies a request by consulting the buffer first. If this is unsuccessful, the request is delegated to the underlying container. Therefore, different implementations of buffers can be exchanged for efficiency or debugging purposes. For using a *traditional* buffer that stores blocks of serialized data, XXL provides the *Block* class. Blocks are objects that offer to store a certain amount of binary data and to create input and output streams of the data. Currently, XXL supports the LRU buffer replacement strategy, but it is not difficult to implement other ones.

5.3 External Collections & Input Iterators

Collections that use external resources for storing their objects are very important for database applications. For this reason, XXL contains external implementations for the most important collections. Currently, there are external bags, queues and containers. Other data structures like external lists are in process. For debugging and testing purposes external collections can easily be exchanged by their corresponding (main memory) variants.

Another benefit of XXL is its rich reservoir of input iterators that read data from external sources, e.g. files or URLs. For example, XXL contains the class *InputIterator* that is able to wrap an arbitrary Java input stream to an iterator. Special input iterators are also the *FileInputIterator* and the *URLInputIterator*. Moreover, iterators can be wrapped in Java input streams using the class *IteratorInputStream*.

6 Index Structures

A framework of tree-based index structures as well as many implementations are gathered in the package *xxl.indexStructures*. This framework provides a skeleton implementation for so-called grow-and-post trees [Lom91] which is a broad class of index structures including the popular B-trees, R-trees, X-trees and hB-trees. In contrast to GIST [HNP95] where a similar approach has been pursued, the framework is more generic in the sense that a broader class of structures is supported. Furthermore, our framework offers efficient implementations of generic query processing algorithms and bulk operations [BSW97, BSW99, BSS00] that is not available in GIST. It is also notable that GIST is basically an isolated system that limits its focus solely to indexing, but does not provide a query processing infrastructure nor external connectivity. However, the index structures of XXL deliver the results of queries as cursors which allows further processing using the operators of the cursor algebra.

6.1 The Lower Interface of Index Structures

Tree-based index structures keep their data in nodes where each of them refers to a block that is generally of a fixed size. These blocks are managed by an object that satisfies the interface *Container*. Obviously, a user may implement his/her own container classes.

A typical scenario for the usage of containers is illustrated in Figure 1. The top container belongs to the class *BufferedContainer* where presumably a large number of nodes of the index structure are kept in the

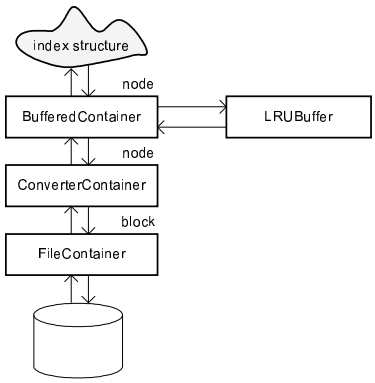

Figure 1: An example for using containers

buffer. If a desired node is not in the buffer, the node is requested from a *ConverterContainer*. The only task of a *ConverterContainer* is to convert a node into its block and vice versa. Therefore, it forwards a request to a *FileContainer* object that reads the desired block from disk.

6.2 Implementation of new Index Structures

Although the tree framework seems to be very intricate, new index structures can be implemented fast. In general, it is sufficient to implement the interface *Descriptor* for index entries and to complete the implementation of the *Node* class whose objects refers to the specific nodes used in the target index structure.

For sake of simplicity, let us consider an implementation of an R-tree. Then, the descriptor of an index entry refers to a window region of the multi-dimensional data space that covers all data objects stored in the corresponding subtree. In general, the interface *Descriptor* consists of the following methods:

```
interface Descriptor {
    boolean overlaps (Descriptor descriptor);
    boolean contains (Descriptor descriptor);
    Descriptor union (Descriptor descriptor);
    boolean equals (Object object);
}
```

The predicate `overlaps`, `contains` and `equals` are used for the comparison of descriptors, whereas `union` computes the minimum enclosing descriptor. The semantics of theses methods seems to be selfexplainable for a window region. Additionally to the implementation of a descriptor, the implementation of the abstract class *Node* has to be completed. Most of its functionality is already implemented in the generic predecessor classes within the class hierarchy. Therefore, only the functionality has to be provided that cannot be made available without specific knowledge about the target index structure. This basically consists of the methods `chooseSubtree` and `split`. `chooseSubtree` determines the subtree where the descriptor should be inserted in, using a given descriptor and a set of subtrees. The `split` method performs a node split and returns information about the location of the split in the tree.

Because our tree-based index structures generally keep their nodes as blocks in containers, converters are required for the descriptor and the nodes.

Our R*-tree implementation (without re-insertion) [BKSS90] is a good example for a compact implementation of an index structure. It only consists of less than 200 lines of code because the cursor algebra is heavily used again.

6.3 Available Functionality

In the following, we briefly describe the available functionality of an index structure that is based on our framework. First of all, the index structure supports insertions and deletions of objects. In addition, the index structure is automatically equipped with efficient algorithms for bulk-loading and bulk-insertion.

At a first glance, it may sound surprising that the programer of a new index structure has not to deal with the implementation of queries. This is generally true for standard queries, but we have to admit that exotic queries still have to be implemented by hand. The predicate of a standard query can be expressed by using the descriptor of the underlying index structure. For an R-tree, a standard query is the popular window query since the corresponding predicate refers to a rectangle which is also the data type of the descriptor. Moreover, our framework offers the efficient implementation of nearest-neighbor queries [HS99] and their combinations with standard queries.

7 Spatial

The package *xxl.spatial* provides support and building blocks for spatial, temporal and high-dimensional join processing. The classes belong to three main categories: data types, building blocks and algorithms.

The most important data types are d-dimensional points and rectangles. For these, we provide lp-metrics and operations like *overlap*, *perimeter*, *area*, etc.

This package also contains useful building blocks like z-code computation, data-conversion tools and a fixed-point double-arithmetic. The fixed-point arithmetic considerably facilitates the implementation of algorithms that deal with data that is normalized to the unit-cube $[0;1)^d$, e.g. computing a z-code for a given point is performed with a simple 'bit-zipper'.

Currently, several d-dimensional join-algorithms come ready-to-use. We provide implementations of plane sweep [APR+98], the z-code join [Ore91], S3J and MSJ [KS00] and hash-based algorithms [PD96]. In addition, this package contains new developments like the join techniques proposed in [DS00] and a new, powerful similarity-join algorithm [DS01]. The implementations of all the above algorithms are very compact

and flexible, e.g. the d-dimensional z-code join [Ore91] is implemented with only 20 lines of code. The high-level coding of the algorithms makes the integration of new functionality easy. Our experience has shown that implementing a new algorithm is reduced to implementing its delta to an existing approach. This does not only sharply reduce the coding time but also helps to classify new approaches.

8 Applications

The quality of a library is mainly determined by a broad range of applications that are supported. In order to illustrate XXL's flexibility, we present two use cases described in detail in the following sections.

8.1 Connectivity to Cloudscape

Cloudscape [Inf01], a commercial DBMS of Informix, is a Java- and SQL-based object-relational database management system, written in 100% pure Java. It can directly be embedded in a Java application, or used in a classical client-server or Web-server mode. Data access is realized via SQL-92E calls by using the standard Java Database Connectivity (JDBC) protocol. Cloudscape extends this functionality by the feature to store arbitrary serializable Java objects. In that way, it is possible to define an attribute of a table directly as a Java data type. Physically the Java object is stored in its byte format. However, when the Java object is read from and written into the database, it is automatically deserialized and serialized, respectively. Furthermore, Java is used for implementing stored procedures and triggers.

XXL communicates with Cloudscape in two directions. The first and more obvious direction is accessing the DBMS via JDBC calls and SQL statements. This functionality guarantees that XXL is able to create, drop and alter tables as well as to evaluate queries on tables. Executed SELECT statements return a *ResultSet* (see Section 3.5) whose rows are consumed by XXL's *ResultSetCursor*. Each output object of this cursor is created by calling a user-defined function on each row. Therefore, XXL processes arbitrary objects of the interface *ResultSet* in a demand-driven fashion and provides a smooth integration of them into the cursor algebra.

The second direction refers to the feature of Cloudscape that a *ResultSet* can directly be used in the FROM clause of a SQL statement. This is also termed the *Virtual Table Interface (VTI)*. As far as the other parts of the SQL statement are concerned, there is no difference between an ordinary table and an object satisfying VTI. This feature is beneficial for a seamless integration of external functionality into the DBMS. In particular, it is very appealing to XXL since an operator tree is able to deliver its results wrapped as a *ResultSet*. Currently, there is still the limitation that the FROM clause only accepts a constructor call of a *ResultSet* that makes it difficult to use a previously created *ResultSet*. Moreover, VTI only accepts those constructors with primitive parameter types. In order to alleviate these deficiencies, XXL provides the class *VirtualTable* which is a proxy for an object of the interface *ResultSet*. An example of an SQL statement employing the VTI is given by:

```
SELECT Emp.Name
FROM NEW VirtualTable() AS Emp
WHERE Emp.Salary > 100000
```

As shown in our example, attributes of a virtual table are treated like attributes of an ordinary table. In comparison to the overhead of implementing data blades, this is a very convenient way to integrate arbitrary data sources and user-defined functionality.

Currently, XXL is limited to read-only virtual tables. In the future XXL will also allow read-write access with the intention to support INSERT and DELETE statements based on virtual tables. Furthermore, a generic wrapper will be developed that transforms a cursor into a *ResultSet* by adding meta data information. Therefore, XXL will offer the possibility to integrate data from a variety of sources into a Cloudscape database as well as to handle these sources in SQL statements.

8.2 Experiments on Spatial Data

When developing new query processing techniques researchers strive for an experimental comparison of their approach to already established ones. There are at least two important requirements for an experimental comparison [ZMR96]: the experiment should be *fair* (i.e., the approaches should be based on the same building blocks) and *reproducible* (i.e., other researchers should be able to repeat the experiments easily). XXL is designed as a platform for experimental comparisons where the above mentioned requirements can be satisfied.

- **Fair:** XXL provides a rich infrastructure for the implementation of new query processing techniques. The developer will employ the infrastructure since it largely facilitates the burden of programming. As a side effect, this leads to a better comparability of different approaches and hence, fair comparisons are easily possible.

- **Reproducible:** Applications implemented in Java using XXL are running under different operating systems and hardware platforms. Moreover, XXL is available for download [Dat01] and includes a full documentation. Since XXL contains many useful building blocks, applications are likely be written in a high-level style that makes it easy to understand the underlying semantics. These are important properties for reproducible code.

data set	description	#MBRs
LA_RR	railways and rivers LA	128,971
LA_ST	streets LA	131,461

Table 1: Description of the data sets

algorithms	flat file	Cloudscape
SQL (nested loops)	–	> 2 days
plane-sweep (hybrid)	11.0	17.4
plane-sweep (strict)	11.3	19.0
Orenstein (kd-trie)	52.5	58.2
Orenstein (quadtree)	66.5	70.7

Table 2: Elapsed time of the algorithms in seconds

In the following, we present an experimental comparison where we examine the popular problem of processing a spatial join. Our comparison shows results that are unique with respect to the following issues. First, we examine the overhead of reading the input data from a database in comparison to reading from flat files. Second, our results are reported as a function of the elapsed time, i.e., the number of results that are delivered since the start of the algorithms. These results are seldom found in the literature, though they are important in a demand-driven query processor.

Each of the spatial tables corresponds to a set of rectilinear minimum bounding rectangles (MBR) represented by columns for the x- and y-coordinate of the lower left and the upper right corner. In our experiments we used two different real data sets of the TIGER database [Bur89] (see Table 1). The data sets LA_RR and LA_ST contain the MBRs of different types of line segments from the region of Los Angeles. In the following, we present the experimental evaluation of five different spatial join algorithms.

The first algorithm is an internal algorithm of Cloudscape, i.e., we executed an SQL statement on the two given tables specifying the overlapping condition directly in the WHERE clause. The other spatial joins correspond to algorithms provided by XXL. The second algorithm is a plane sweep algorithm by Arge et.al. [APR+98] where the input is sorted in a strict manner using the sort-merge routine of XXL. The third algorithm is an improved plane sweep approach based on a hybrid sort-merge operator, i.e., the sorted streams are created on demand by merging sorted runs [Gra94]. The fourth algorithm is the z-code join-algorithm by Orenstein [Ore91] using kd-trie splits. The fifth is a variant of this algorithm using quadtree splits.

Our experiments were performed on a PC with an AMD processor (Athlon 700 MHz, 1024 MB main memory) under Windows 2000. Since main memory was large enough, the entire join phase was performed without any disk access. Figure 2 depicts the number of computed results of the spatial joins as a function of the elapsed time where the input was from a Cloudscape database. Table 2 shows the total elapsed time of the algorithms. Since Cloudscape makes use of a simple nested-loops algorithm, the total elapsed time was more than 2 days. After that we stopped the process. The other algorithms performed the join in less than 71 seconds, whereas the plane sweep algorithms are more efficient than the ones that rely on multi-dimensional data structures. Results show that the use of hybrid sorting pays off.

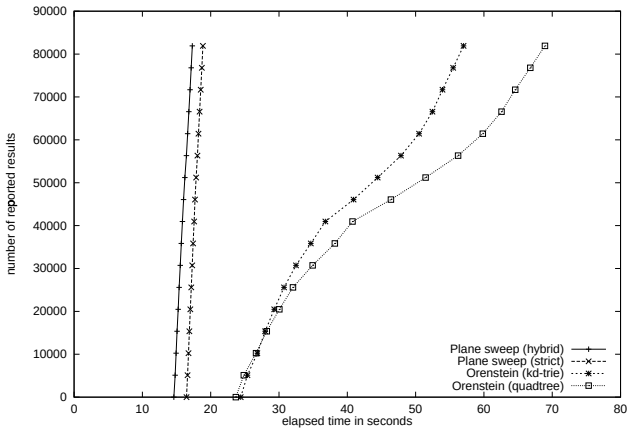

Figure 2: Elapsed time of spatial join algorithms in Cloudscape

Overall, the results of our experiments show the necessity of using efficient algorithms for advanced query operators. DBMS like Cloudscape are not able (and willing) to provide the right operators for specific applications. As a first solution, these operators can be implemented on top of the DBMS using a library like XXL which facilitates the coding. The overhead of keeping the data in a DBMS is surprisingly low in comparison to using flat files (see Table 2). This generally supports our library approach.

9 Conclusions

In this paper we outlined the design of XXL, a well documented Java library, suitable for rapid implementation of advanced query processing techniques. The software is freely available under the terms of the GNU Lesser General Public License [Dat01].

Key components of XXL are a powerful cursor algebra, a framework for a broad class of index structures and a toolbox of I/O data structures. XXL is not in competition to the popular Java API, but it provides seamless enhancements.

The focus of our future work is directed to supporting additional data sources including XML. We will also examine the connectivity to commercial database systems. XXL will be extended by a rich source of on-line aggregation functions as well as statistical estimators. Finally, we will proceed in providing further reference implementations of advanced query processing techniques.

References

[AB95] S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB Journal*, 4(4):727–794, 1995.

[AC99] K. Mehlhorn A. Crauser. LEDA-SM Extending LEDA to Secondary Memory. In *Algorithm Engineering*, pages 228–242, 1999.

[APR+98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable Sweeping-Based Spatial Join. In *VLDB*, pages 570–581, 1998.

[BF79] P. Buneman and R. E. Frankel. FQL - A Functional Query Language. In *SIGMOD*, pages 52–58, 1979.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.

[BSS00] J. van den Bercken, M. Schneider, and B. Seeger. Plug&Join: An easy-to-use Generic Algorithm for Efficiently Processing Equi and Non-Equi Joins. In *EDBT*, pages 495–509, 2000.

[BSW97] J. van den Bercken, B. Seeger, and P. Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *VLDB*, pages 406–415, 1997.

[BSW99] J. van den Bercken, B. Seeger, and P. Widmayer. The Bulk Index Join: A Generic Approach to Processing Non-Equijoins. In *ICDE*, page 257, 1999.

[Bur89] Bureau of Census. TIGER/Line precensus files (1990 Technical Documentation), 1989.

[CW00] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, pages 1–10, 2000.

[Dat01] Database Research Group. XXL. http://www.mathematik.uni-marburg.de/DBS/xxl, 2001.

[DS00] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, 2000.

[DS01] J.-P. Dittrich and B. Seeger. GESS: a Scalable Similarity-Join Algorithm for Minig Large Data Sets in High Dimensional Spaces. In *KDD*, 2001.

[GMP95] G. Gardarin, F. Machuca, and P. Pucheral. OFL: A Functional Execution Model for Object Query Languages. In *SIGMOD*, pages 59–70, 1995.

[Gra94] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE*, 6(1):120–135, 1994.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[Hin85] K. H. Hinrichs. *The Grid File System: Implementation and Case Studies.* PhD thesis, ETH Zürich, 1985.

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, pages 562–573, 1995.

[HS99] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Inf01] Informix. Cloudscape. Internet: http://www.cloudscape.com/, 2001.

[Knu73] D. E. Knuth. *The Art of Computing, vol. 3 – Sorting and Searching.* Addison-Wesley, 1973.

[KS00] N. Koudas and K. C. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *TKDE*, 12:3–18, 2000.

[Lom91] D. Lomet. Grow and Post Index Trees: Role, Techniques and Future Potential. In *SSD*, 1991.

[Ore91] J. A. Orenstein. An Algorithm for Computing the Overlay of k–Dimensional Spaces. In *SSD*, pages 381–400, 1991.

[PD96] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, pages 259–270, 1996.

[VV96] D. E. Vengroff and J. S. Vitter. I/O-Efficient Scientific Computation using TPIE. In *Proc. Goddard Conf. on Mass Storage Systems and Technologies*, pages 553–570, 1996.

[Wei01] K. Weihe. A Software Engineering Perspective on Algorithms. In *ACM Computing Surveys*, page to appear, 2001.

[ZMR96] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for Presentation and Comparison of Indexing Techniques. *SIGMOD Record*, 25(3):10–15, 1996.