Scalable Store of Java Objects Using Range Partitioning

Mariusz Bedla[1] and Krzysztof Sapiecha[2]

[1] Kielce University of Technology
25-314 Kielce, al. 1000-lecia Państwa Polskiego 7
m.bedla@tu.kielce.pl
[2] Cracow University of Technology
31-155 Kraków, ul. Warszawska 24
pesapiec@cyf-kr.edu.pl

Abstract. Scalable Store of Objects (SSO) should allow for storing and maintaining a huge number of objects distributed over many nodes of a network. RP*N is a structure which belongs to a family of order-preserving, Range Partitioning Scalable Distributed Data Structures (RP* SDDS). The architectures of SDDS were designed to store records. Different structures of objects and complicated dependences between objects are the cause that a new architecture for RP* is needed. The paper describes a new object-oriented version of RP*N architecture and its implementation for Java objects. This version can be used in a fast and scalable store of Java objects. Performance of the implementation is evaluated and compared with serialization of objects on a disk and storing objects as a main-memory collection.
Keywords: object store, scalability, SDDS, RP*, Java

1 Introduction

Object Store (OS) is one of the most important components of Object-Oriented Database Management System (OODBMS) [1]. OS manages physical objects. These are records and may be seen as tables of bytes.

OS should allow for storing and maintaining huge number of objects. As such OS requires powerful and scalable computer platform. A multicomputer could be such a platform. A multicomputer applies parallel architecture shared-nothing [2] which is the most scalable for very large databases [3]. In such an architecture every computer owns local memory and disk, and acts as a server for data [4]. All the computers are connected through a fast network. A communication between the computers is based on message passing.

The question arises how to distribute data among servers of a multicomputer. There are three basic partitioning schemes that might be used [4]:

- round-robin partitioning where data are partitioned according to function: *id mod n*, where *n* is the number of servers,
- hash partitioning where data are partitioned according to a hash function, and
- range partitioning where data are partitioned according to their ranges.

SDDS is a file of records, a file which expands on servers of a multicomputer. The main goal of SDDS file is to store a huge number of records and also to make an access to these records fast and dependable for many clients. Records are stored in so called buckets localized in main memories of the servers. A capacity of each of the buckets is limited. If a bucket's load reaches some critical level, it performs a *split*. A new bucket is created then on another server and typically a half of data from the splitting bucket is moved into a new one.

Distributed Object-Oriented Database Management Systems use fragmentation and allocation schemes. These schemes reduce data transfers, not-local data references, irrelevant data access and increase concurrency [5,6]. They may rely on many factors like a structure of a class, a frequence and a type of an access, etc. On the contrary in the SDDS a placement of an object depends only on a key of the object.

There are numerous architectures of SDDS file such as RP*[7], LH*[8], DDH [9], etc. The paper concerns a development of object oriented version of RP*N architecture to store Java objects in serialized form (OORP*N). Such an implementation may be used as a part of distributed store of Java objects

The paper is organized as follows. Motivation to this research is given in the next section. An architecture of object-oriented version of RP* (OORP*) is introduced in Section 3. In Section 4 its implementation for Java objects is given. In Section 5 a comparison of the implementation with object serialization on a disk and a main-memory collection are presented. Section 6 concludes the research.

2 Motivation

Not all SDDS architectures which are suitable for relational databases could be useful for object-oriented databases. These architectures in which it is assumed that a size of every record is the same are useless, because objects can contain tables that can results in various sizes of objects of the same class. Selection of data from a relational database bases on operations on primary and foreign keys. Relations between objects can be more complicated. SDDS usually stores all data in main memory, which is not durable. To achieve persistence of objects a backup on hard drive is required.

On the other side, scalability, high speed of operation and fault tolerance are important advantages of SDDS [8,7,10,11]. Storing an object-oriented database in SDDS should increase performance of OODBMS and make OODB applications scalable. Developing an architecture preserving all advantages of SDDS but efficiently applicable to store of objects would be very useful.

There are few well known implementations of SDDS:

– AMOS-SDDS: scalable distributed system combining AMOS II and SDDS-2000, object-relational DMBS with RP* file used as an external storage [12,13],
– prototype of LH*g run in a single multi-threaded computer process which uses integers as data [14],
– Distributed Dynamic Hashing [9],
– "actor databases" where actors manipulate and store data in SDDS based on CTH* (distributed Compact Trie Hashing) [15] and others.

However, in none of the above mentioned papers [9,12,13,14,15] an object-oriented version of RP* architecture was presented.

3 RP* for Objects

In RP* architecture an address of a bucket where a record should be stored is calculated on the basis of ranges of the buckets [7]. Hence, the algorithm for calculation of an address for a record in SDDS file (address of a bucket) is quite simple.

In OORP* architectures an address of the bucket where an object should be stored is calculated on the basis of ranges of the buckets, too. Objects of the same class can contain tables of different sizes what makes their sizes different. Hence, the buckets should store various numbers of objects.

For this reason a split of the bucket should be done when:

1) an object is to insert and
   a) load factor of the bucket is above threshold for controlled split; the load factor is measured as a quotient of sum of sizes of all objects in the bucket to the capacity of the bucket,
   b) there is no enough free space for next object for uncontrolled split, or
2) an object is updated. An updated object may change its size. If updated object is bigger then it may happen what follows:
   a) there is enough free space in the bucket and the updated object can be stored directly in the bucket, or
   b) there is not enough free space in the bucket and the bucket must split using the same algorithm as during insertion of an object.

If updated object is smaller or has the same size it is stored directly in the bucket.

The split algorithm for OORP* (Algorithm 1) [16] is similar to that for original RP* [7] and also quite simple. However, it is assumed that $bucketSize \gg maximum(sizeOf(object))$. Steps 2, 3, 4 and 6 are as in [7]. In OORP* *bucketSize* must be actualized when an object is added, updated or deleted.

4 Implementation of RP* for Java Objects

An implementation of OORP* architecture for Java objects should allow for storing, updating, retrieving and deleting individual objects of a class defined by a user, with such restrictions on classes which might be easy accepted (i.e. classes must be serializable).

Objects stored in OORP* are distributed among many servers. Hence, they should have some extra features. This can be achieved in different ways. These are as follows:

- modification of a source code of a class, what requires an access to this source code,
- modification of Java Virtual Machine (JVM), but not all licenses allow to do that, additionally permanent modification of JVM affects other applications,
- modification of compiled byte code to gain required features, what means that the source code is not necessary and is left unchanged.

Algorithm 1 OORP* split algorithm

```
1.  Determine middle key $c_m$ in the overflowing bucket $B$.
    $size \leftarrow 0$;
    $halfBucketSize \leftarrow bucketSize/2$;
    while $size < halfBucketSize$ do
        $o \leftarrow nextObject(B)$;
        $size \leftarrow size + sizeOf(o)$;
        $c_m \leftarrow c(o)$;
    end while
2.  Attempt the creation of bucket $M$. Wait for ack, or denial if bucket $M$ exists already.
3.  If creation is denied, then $M \leftarrow M + 1$;  go to Step 2.
4.  Copy to bucket $M$ the following content:

      – The header with :
        $\lambda \leftarrow c_m(B)$;
        $\Lambda \leftarrow \Lambda(B)$;
      – every object from $B$ with $c > c_m$.

5.  Decrease the maximal key in $B$:
    $\Lambda \leftarrow c_m(B)$;
    remove objects moved to bucket $M$, and actualize $bucketSize$.
6.  Set $M \leftarrow M + 1$,
    where:

      – $size$ and $halfBucketSize$ denote integer variables,
      – $o$ denotes an object variable,
      – $M$ denotes a number of a new bucket,
      – $bucketSize$ denotes a sum of sizes of all objects in $B$,
      – $c$ and $c_m$ denote a key and middle key in $B$ respectively,
      – $\lambda$ and $\Lambda$ denote minimal and maximal values of keys of $B$ respectively,
      – $nextObject$ and $sizeOf$ denote a next object in $B$ and a size of an object respectively.
```

To store objects of some class in OORP* the class must contain some extra methods and attributes. A solution based on inheritance can not be applied here because Java class can only extend one base class. A programmer could probably add these method and attributes manually but it excludes transparency. For these reasons the last from the above options that is byte code modification is chosen.

There are distinguished two types of objects, which can be added to OORP*:

- primary objects which can be directly added to OORP*, these are objects of classes created by a user,
- secondary objects which can be added to OORP* only with primary objects, for example objects of class Integer.

Objects may contain references to other primary or secondary objects organized as a list, for example. Managing a list as one single object may lead to many problems related with performance and dependency of the store. Every time when the first element of the list is required, the whole list would be retrieved. This is very time consuming (and not efficient). To avoid such problems the references to primary objects are treated in special way. When an object is added to OORP* every primary object is added separately. When the object is retrieved from OORP* no other primary object is retrieved. When a field containing reference to primary object is accessed (read or written) an autogenerated method (getter or setter) is invoked. Then the object is retrieved from (stored to) OORP*. A class containing references to primary objects is modified:

– autogenerated table with identifiers of the primary objects is added,
– every access (read or write) to such field is replaced with invocation of autogenerated method (the getter or setter),
– autogenerated getter and setter for every field containing the references to primary objects are added.

The program called SDDSModifier modifies compiled class and adds all required features. Objects are stored in serialized form. They are converted into tables of bytes, transmitted and then stored in the buckets on servers. Because of its popularity and universality Java collection is chosen as a method of accessing objects. The collections, besides tables, are probably the second primary method of arranging objects. All standard Java collections implement one of the two following interfaces: Collection which is a base for lists and sets, and Map which is used to map keys to values. The interface SDDSCollection, which extends Collection, and class SDDSFile, which implements it were developed. SDDSFile does not support queries based on values of fields of objects, but the objects may be iterated.

Summarizing, the development of scalable, distributed store of Java objects consists of the three following steps:

1. First a programmer develops an application which uses SDDSFile to store objects. The application may use classes which are stored in OORP* and other classes not related with OORP*.
2. Next, the classes are compiled using a standard Java compiler and then modified by SDDSModifier.
3. Finally, after starting servers the application may be launched. Every server may work in textual or graphical mode.

5 Performance Evaluation

In the experiment objects were inserted and retrieved from:

– a collection (java.util.HashSet) which is stored in main memory,
– a file which is stored in secondary memory,
– a RP*N file composed of 4 or 8 servers. The only one client of RP*N file was assumed.

Performances of these operations were measured and compared. During the experiment a class containing tables of bytes (512 kB) was used. The number of stored objects was from 1000 up to 6000 (from about 512 MB up to about 3 GB). All classes used for tests have not overloaded method hashCode(). Every test was repeated three times

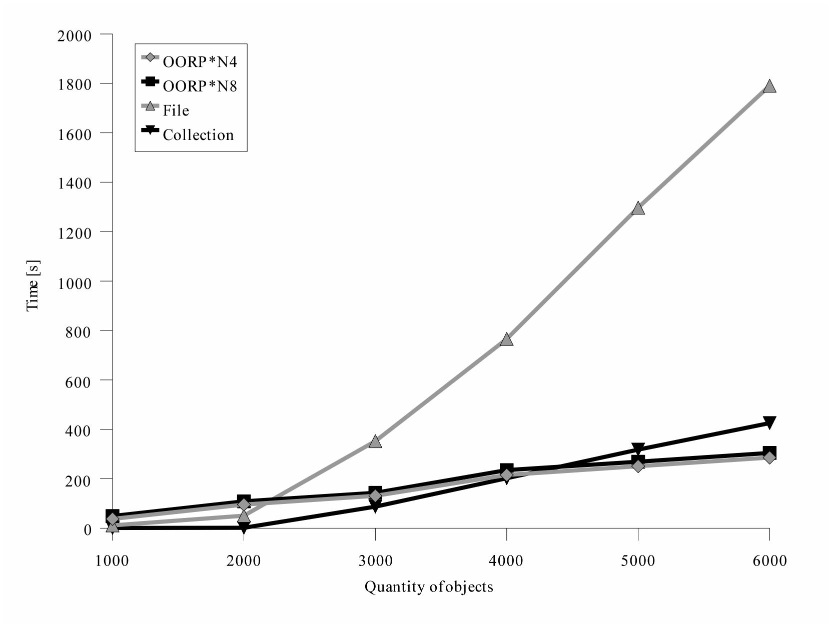

Fig. 1. Total time of storing and retrieving all objects.

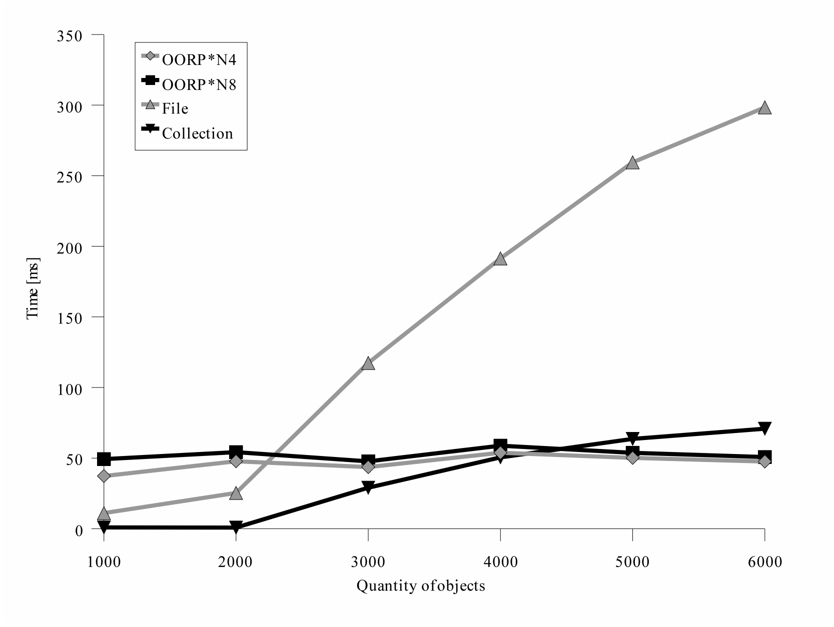

Fig. 2. Average time of storing and retrieving single objects.

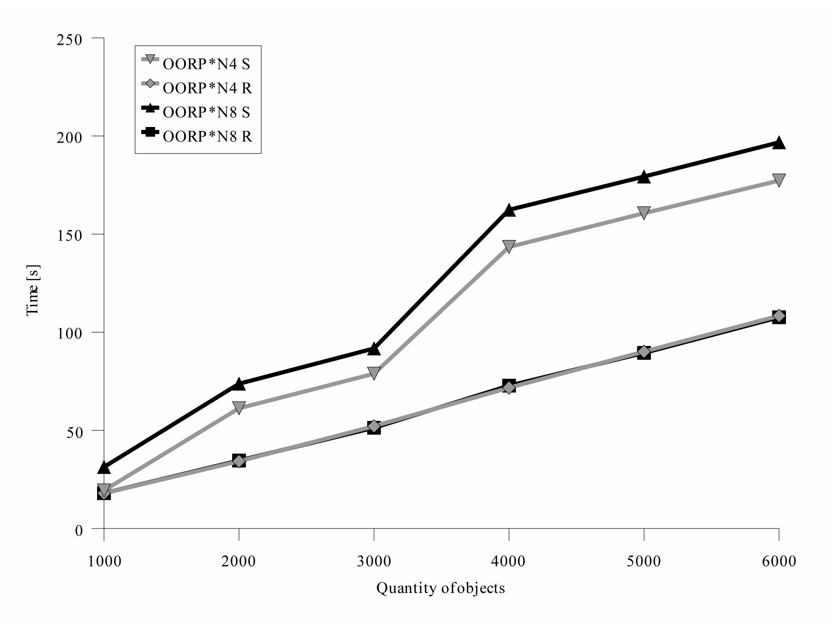

Fig. 3. Time of storing all objects to (S) and retrieving from (R) OORP*N for 4 and 8 servers.

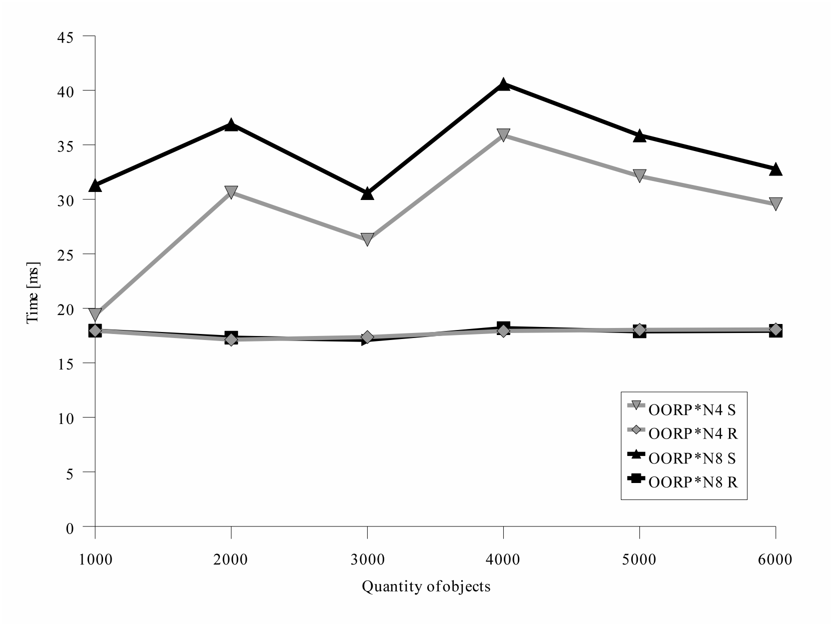

Fig. 4. Average time of storing single objects to (S) and retrieving from (R) OORP*N for 4 and 8 servers.

on the same computers: Athlon 3.0 GHz, 1,5 GB RAM and hard disk – ST380211AS, connected through gigabit Ethernet network. Results of the experiment are shown on Fig. 1, 2, 3 and 4.

The OORP*N has almost constant average time of storing and retrieving single objects. It does not depend directly on the number of objects. The difference between maximum and minimum values of the average time is about 44% for OORP*N consisted of 4 servers and about 23% for OORP*N consisted of 8 servers. For serialization into a file this difference is more than 2000% and for objects stored in the main memory is more than 8000%. The serialization into a file is the slowest from evaluated methods if total size of the objects excess 1.5 GB (that is the size of the main memory of the computer used in the experiment). Moreover, the serialized objects in a file must be accessed sequentially.

For OORP*N average time of retrieving objects is almost constant and it is about 17,7 ms. Average time of storing all objects is more variable and varies from about 19,4 ms to 40,6 ms. It is a sum of average time of sending objects to the servers and average time related with splits.

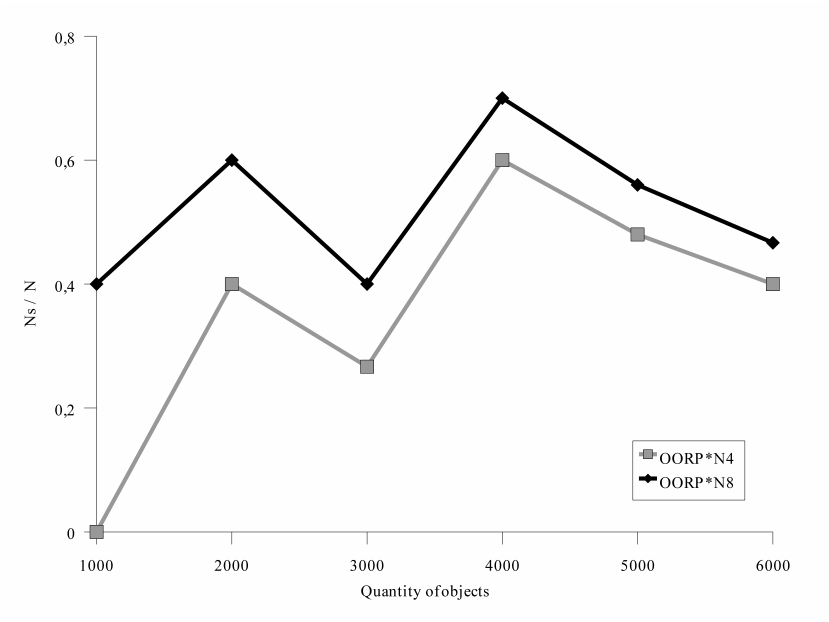

Fig. 5. Ratio of the number of objects moved during a split (Ns) to the number of all objects (N).

Let us assume that average time of sending single object is always the same and equals average time of storing single object when split does not happen. If *s* is the ratio of the number of objects moved during a split to the number of all objects then average time related with splits can be calculated as arithmetic product of *s* and some constant

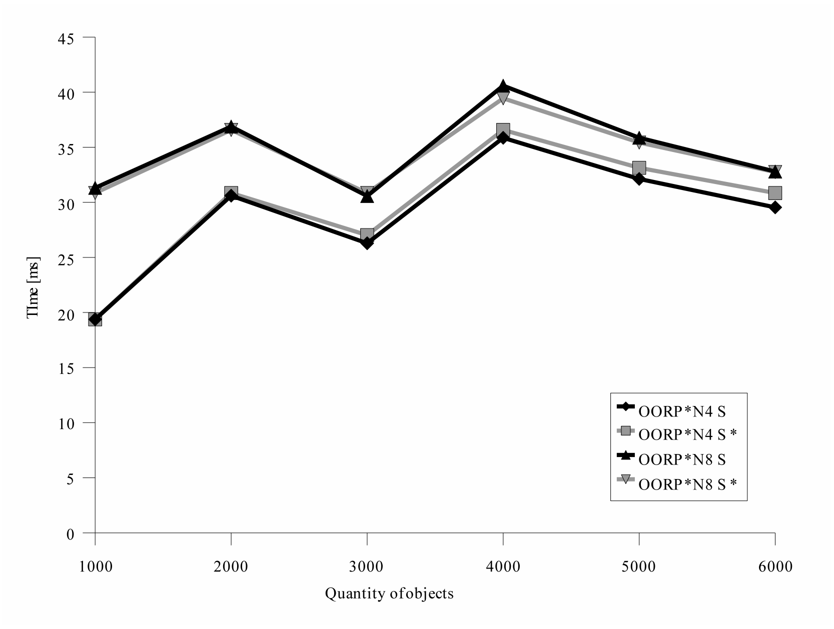

Fig. 6. Comparison of theoretical calculations of average time of storing single objects to OORP*N (marked *) with experimental results.

value related with a speed of the network. Figure 5 presents ratio of the number of objects moved during a split to the number of all objects. A comparison of theoretical calculations of average time of storing single objects to OORP*N (marked *) with results obtained experimentally is presented on Fig. 6. The curves are almost identical.

6 Conclusions

The OORP* can be used as a part of distributed object store. It can store Java objects in serialized form. In OORP* addressing algorithm is same as in RP*. However, buckets can store various numbers of objects. The split algorithm must be modified and not based on the number of objects but on the size of objects. The experiments proved that average times of storing and retrieving single objects are more constant for the OORP*N than for the collection and for the file. The difference between maximal and minimal values of the average times is almost twice smaller for OORP*N composed of 4 servers (44%) than for OORP*N composed of 8 servers (23%). Average time of retrieving single objects is almost constant and does not depend on the number of objects or servers. Time of storing all objects is a sum of time of sending objects to servers and time of splits. The time of splits depends on the number of objects moved during a split what, in turn, depends on the size of the bucket. The bigger buckets the better results because less objects are moved. The results from the experiment confirmed theoretical calculations.

Object-oriented versions of other architectures of the SDDS will be the subject of our further research.

References

1. Lobry, O., Collet, C., Déchamboux P.: The VIRTUOSE Distributed Object Store. In: Proceedings of 8th International Workshop on Database and Expert Systems Applications (DEXA '97), pp. 482–487. Toulouse, France (1997)
2. Stonebraker, M.: The Case for Shared Nothing. Database Engineering 9, 4–9 (1986)
3. Lo, Y.-L., Hua, K.A., Young, H.C.: GeMDA: A Multidimensional Data Partitioning Technique for Multiprocessor Database Systems. Journal of Distributed and Parallel Databases 9, 211–236 (2001)
4. DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. Communications of the ACM 35, 85–98 (1992)
5. Ezeife, C.I., Barker, K.: Distributed Object-based Design: Vertical Fragmentation of Classes. Distributed and Parallel Databases, Springer Netherlands 6, 317–350 (1998)
6. Barker. K., Bhar S.: A Graphical Approach to Allocating Class Fragments in Distributed Objectbase Systems. Distributed and Parallel Databases, pp. 207–239. Kluwer Academic Publishers (2001)
7. Litwin, W., Neimat, M.-A., Schneider, D.: RP*: A Family of Order Preserving Scalable Distributed Data Structures. In: Proceedings of the 20th International Conference on Very Large Databases, pp. 342–353 (1994)
8. Litwin, W., Neimat, M.-A., Schneider D.A.: LH*–A Scalable, Distributed Data Structure. ACM Transactions on Database Systems 21, 480–525 (1996)
9. Devine, R.: Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In: Proc. of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO), pp. 101–114. Chicago (1993)
10. Litwin, W., Neimat, M.-A.: High-Availability LH* Schemes with Mirroring. International Conference on Cooperating Information Systems, pp. 196–205. Brussels (1996)
11. Sapiecha, K., Łukawski, G.: Fault-tolerant Protocols for Scalable Distributed Data Structures. LNCS, vol. 3911, pp. 1018–1025. Springer, Heidelberg (2006)
12. Ndiaye, Y., Diéne, A.W., Litwin, W., Risch T.: Scalable Distributed Data Structures for High-Performance Databases. WDAS 9, 45–69 (2000)
13. Diéne A.W., Litwin, W.: Performance Measurements of RP*: A Scalable Distributed Data Structure For Range Partitioning. Intl. Conf. on Information Society in the 21st Century: Emerging Techn. and New Challenges, Aizu City, Japan (2000)
14. Lindberg, R.: A Java Implementation of a Highly Available, Scalable and Distributed Data Structure LH*g. Master's Thesis No: LiTH–IDA–Ex–97/65 (1997)
15. Hidouci, W.K., Zegour, D.E.: Actor Oriented Databases. WSEAS Transaction on Computers 3, 653–660 (2004)
16. Bedla, M., Sapiecha, K.: A Store of Java Objects on a Multicomputer. In: Proc. of the 10th International Conference on Enterprise Information Systems (ICEIS), pp. 374–379. Barcelona, Spain (2008)