ObInject: a NoODMG Persistence and Indexing Framework for Object Injection

Luiz O. Carvalho[1,2], Thatyana F. P. Seraphim[1], Caetano Traina Júnior[2], Enzo Seraphim[1,*]

[1] Universidade Federal de Itajubá, Brazil
[2] Universidade de São Paulo, Brazil
olmes@icmc.usp.br, thatyana@unifei.edu.br, caetano@icmc.usp.br, seraphim@unifei.edu.br

Abstract. The object-oriented model has become standard for developing information systems. Consequently, more and more classes have been created to embed business rules and are been instantiated several times, generating a huge amount of objects. In several scenarios, objects must change their state from transient to persistent. Thus, several solutions have been created to meet this need for persistence. Many solutions use frameworks to perform the mapping from objects to relations in database schemas. However, few object persistence frameworks pursue efficiency as a target, for example dealing with index structures like B-Tree, M-Tree and R-Tree simultaneously. This article presents ObInject, a `NoODMG` framework to support object persistence and indexing (`www.obinject.org`). Its main feature is to allow objects to be persisted and indexed into any data structure. The framework is divided into four modules based on Design Patterns and provides abstractions for persistence and indexing controllers that are specialized from the user class. We performed extensive experiments evaluating several of its properties and target abstractions, and show that it allows the implementation to achieve significant performance improvements in relation to the existing alternatives.

Categories and Subject Descriptors: H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; E.2 [**Data Storage Representation**]: Object representation

Keywords: framework, indexing, noODMG, persistence

1. INTRODUCTION

Conceived at the beginning of the 1990s, the Object Database Management Group (ODMG) proposed a list of specifications for developers to write portable applications. Such applications should run on distinct products using common and portable data schemes, query languages, data manipulation and programming language bindings. Less than a decade later, contributions and discussions [Barry and Stanienda 1998; Carey and DeWitt 1996] to the ODMG standard [Cattell and Barry 2000] pushed the ODMG evolution to cover Object Database Management System (ODBMS) and the Object-Relational Mapping (ORM) standards. Early in the 2000s, when the standard reached enough maturity, the attention of the ODMG organization shifted to its implementation and promulgation as a standard for the industry. In addition, the ODMG community concentrated their efforts on the specification of the ODMG Java Language Binding. The current specification is known as the Java Data Objects (JDO). Currently, in its fourth generation, the ODMG standard aims at aiding the large use of standard-based object database technology.

Even with the innovation introduced by the ODMG standard, it was not widely disseminated and the reason ranges from differences between companies and applications that did not follow the standard. Nevertheless, the main reason is that it does not address data indexing. Thus, corporations, middle/final users and developers adopted solutions to add persistence resources that did not follow the ODMG standard.

* Corresponding author. Phone: +55 (35) 3629-1669

This material is based upon work partially supported by CAPES, CNPQ, FAPEMIG, FAPESP, and MEC.

Still in the 2000s, a number of non-relational database solutions known as NoSQL emerged. Such solutions relax the ACID properties in order to improve the scalability for specific data storage types [Thomson and Abadi 2010]. In a general view, SQL solutions follow the Relational Model [Codd 1970]. ODMG solutions are based on the model presented by Cattell and Barry [2000]. NoSQL solutions are not limited to the Relational Model. Their storage data types are classified by Leavitt [2010] as Key-Value, Big Table, Document and Graph Database. NoSQL was stimulated by works like the ones by Chang et al. [2006] and DeCandia et al. [2007]. NoODMG solutions do not follow the ODMG standard, relax the ACID to achieve performance and present a data model distinct from NoSQL models. In this work, we explore a technique based on flexible indexing for object persistence.

This article presents a NoODMG framework, named Object-Injection, to support object persistence and indexing. This framework contributes with the following features:

—**Loose coupling of classes**: User classes are directly handled to enable the persistence solution. The coupling between user classes and the framework is performed without information redundancy in order to save computational resources.

—**Storage flexibility**: The storage of persistent entities can be performed relaxing or not the ACID properties. Relaxing ACID properties ensures better performance because transaction processing is easier. For this purpose, storage media such as disk or memory might be used. ACID properties are available, although requiring lengthier transaction processing.

—**Multiple indexing**: Index structures are used to speed up query processing. Generally, user queries employ structures that handle data presenting the total order relation. However, queries using metric or spatial data have become popular among users. Once a new query type is required, the user can define indexes on the queried fields.

—**Object identification**: Objects are tagged with a unique identifier that allows retrieving objects regardless of their storage type.

—**Object serialization**: Object attributes are represented as a concatenated data string. The major advantage of a data string is to allow the data to move to different devices and operational environments.

—**Non-intrusive**: Intrusive solutions force the user to model classes that inherit a persistent class or that must be modified, with addition of methods, to achieve persistence. Our solution is non-intrusive and does not require modifications on the user-defined classes.

—**Not restricted to the ODMG standard**: The ODMG standard defines its own model for objects, language specification and language binding. However, several applications that do not follow the standard, or that just implement some parts of the standard, also require persistence and can benefit from the Object-Injection technique.

This article is organized as follows: Section 2 describes related work. Section 3 presents a basic description of Object-Injection Framework, hereafter named ObInject. Experiments, results and discussions are presented in Section 4. Finally, in Section 5, general considerations conclude the work.

2. RELATED WORK

Despite the existence of many studies and contributions such as the ones by Mattsson and Bosch [2000], Kienzle and Romanovsky [2002], Leist and Zellner [2006], Oliveira et al. [2012] and Levandoski et al. [2013] regarding framework development, few of them focus on combining indexing, persistence and all of the features described in Section 1. The first work that focuses on persistence was the Volcano system [Graefe 1994]. Volcano was an application, strongly coupled in C++, that implemented mechanisms for data persistence and query processing. Its query mechanism provided basic operators, such as select, project and join, and some extra functions to allow the users to define their own operators. In its first versions, Volcano did not provide indexing but, in subsequent versions, a B^+-Tree [Comer 1979] was included. In addition, the query mechanism was based only in queries using the attributes of the stored data.

The GiST library [Hellerstein et al. 1995] implements in C++ an indexing data structure known as Generalized Search Tree. This structure, conceptually based on the B^+-Tree and on the R-Tree [Guttman 1984], can be customized to index different data types, including multidimensional and metric data. The library also offers a basic query mechanism, where the processing is based on the attributes of the stored objects, but does not make these objects persistent.

With respect to data indexing, Ogasawara and Mattoso [1999] evaluated the use of indexes for query processing on ODMG database GOA++. As noted in that work, it was not possible to define indexes to the user application classes.

The work of Camargo et al. [2003] introduces a framework that explores the use of Aspect-Oriented Programming (AOP) to persist Java classes on RDBMS. This intrusive framework forces the user-application classes to inherit its persistence mechanism. This framework uses relational databases, therefore the ACID properties are always enforced and it does not present storage flexibility regarding transaction processing. However, modeling applications using AOP leads to classes that are loosely coupled and easy to extend. Still in the AOP line, Kienzle and Gélineau [2006] re-implement the ACID properties using AOP and applies its implementation to transactional object persistence.

Other approach for persistence is presented by Alia et al. [2004] and Weske and Kuropka [2001]. These works propose strategies and framework solutions for persistence in middleware environments.

Arboretum [Arboretum 2005] is a C++ framework that provides several index structures, such as M-Tree [Ciaccia et al. 1997] and Slim-Tree [Traina Júnior et al. 2000]. This framework implements facilities to use, to develop and to evaluate metric data structures and similarity search operations. Although some of its indexing abstractions have inspired the Object-Injection modeling, it does not provide object identification and most of its indexing mechanism targets the metric space. Arboretum also does not separate the user persistent objects in entities and keys.

The work of Batko et al. [2007] proposes a framework called Messif. This framework implements some metric data structures, such as M-Tree and Slim-Tree. Those authors describe Messif as a modularized and easily extensible framework, facilitating the development of similarity search algorithms. However, this framework does not support object persistence and, although some parts of its source code are freely available, its core engine is not open source.

Applications introduced by Achtert et al. [2008], Kaur et al. [2012], and Zhu et al. [2012] are frameworks that present some outline in indexing, but their main goal is to mine, respectively, general databases, spatial medical databases and string sequence databases.

Alternatives such as Hibernate [Bauer and King 2004], JDO[1] and JPA[2] are widely used to perform object persistence. Hibernate is a Java object-relational mapping framework which relates an object-oriented data domain, based on plain Java objects, to RDBMS. Hibernate does not provide indexing, serialization and storage flexibility. Its classes are strongly coupled and use a meta-scheme to create redundancy among objects and relations. Moreover, this framework is non-invasive and object identification is implemented using Java annotations.

Java Persistence API (JPA) is a Java framework to allow persistence management and object-relational mapping with Java SE and Java EE. Nevertheless, JPA does not follow the ODMG specification for Java binding. The ODMG Java binding standard is defined and specified by Java Data Objects (JDO). JDO is a standard model abstraction for persistence based on Java interfaces and constitutes an alternative to JPA. The main benefit of JDO technology is the direct storage of a Java domain model instance into the persistent store (database). JPA and JDO are dependent on RDBMS and do not perform multiple indexing and serialization. They also do not define object identification.

[1]JSR-12 (JDO): www.jcp.org/en/jsr/detail?id=12 Access: May 5, 2013
[2]JSR-220 (JPA): www.jcp.org/en/jsr/detail?id=220 Access: May 5, 2013

Perst Lite [Graves and Knizhnik 2009] and db4o [Breunig et al. 2013] are open source commercial solutions that do not depend on RDBMS in order to achieve object persistence (Java and .Net). Those solutions present multiple indexing mechanisms employing data structure such as B-Tree and R-Tree. However, none of them supports data indexing in metric spaces (M-Tree). Moreover, the abstractions that control persistence and indexing are not specialized from the user class.

The Extensible and Flexible Library (XXL) [Bercken et al. 2000] is the most related work to ours. XXL is a high level application, with its open source code written in Java. Its main goal is to support the development of data structures and algorithms for query processing. XXL provides collections of index structures, such as B-Tree, M-Tree, R-Tree, and others, as well as query operators, aiming at facilitate the evaluation of new query algorithms. It does not use RDBMS to persist objects, but its persistence mechanism is strongly based on Java serialization, forcing users to extend its class model and override the required methods for persistence, so it is an invasive framework.

In order to support object persistence, many of these solutions use external systems. In this way, they end up overloading the application. Our framework differs from them, being easily coupled to the applications to provide indexing and persistence. In addition to the features aforementioned in Section 1, its persistence mechanism does not rely on any (R)DBMS. Moreover, it offers a class architecture that can be easily extended to support new query algorithms and indexing data structure development.

3. OBJECT-INJECTION FRAMEWORK

This section presents the Object-Injection Framework (ObInject) in more details and explains the concepts that guided its design. The framework does not focus on any particular programming language and only uses object-oriented programming fundamentals. Most of its modeling is based on the Curiously Recurring Template Pattern (CRTP) [Coplien 1995] and Design Patterns, in order to abstract semantic issues. The Design Patterns, when mentioned, behave and refer to those described by Gamma et al. [1995].

ObInject has four abstraction modules, organized according to Figure 1. The `Metaclass` module defines persistent entities and the domain of indexed keys. The `Storage` module defines the primary indexes for persistent entities and secondary indexes for indexed keys. These indexes are responsible for managing and organizing data from the `Metaclass` module, which are kept by the `Block` module. The `Device` module defines computational resources in charge of managing the storage of the data structure. This module constructs blocks requested by and passed on to the `Storage` module. Lastly, the `Block` module defines how the persistent entities and the indexed keys are stored in blocks or pages. Then, these modules are detailed using UML Class Diagrams, but only the most important classes, attributes and methods are shown.

3.1 The `Metaclass` Module

The `Metaclass` module defines the bind between the user application classes and the framework, expressed by two class groups: persistent entities and indexed keys, as shown in Figure 2.

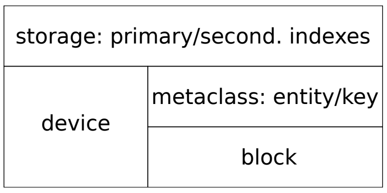


Fig. 1. Module organization

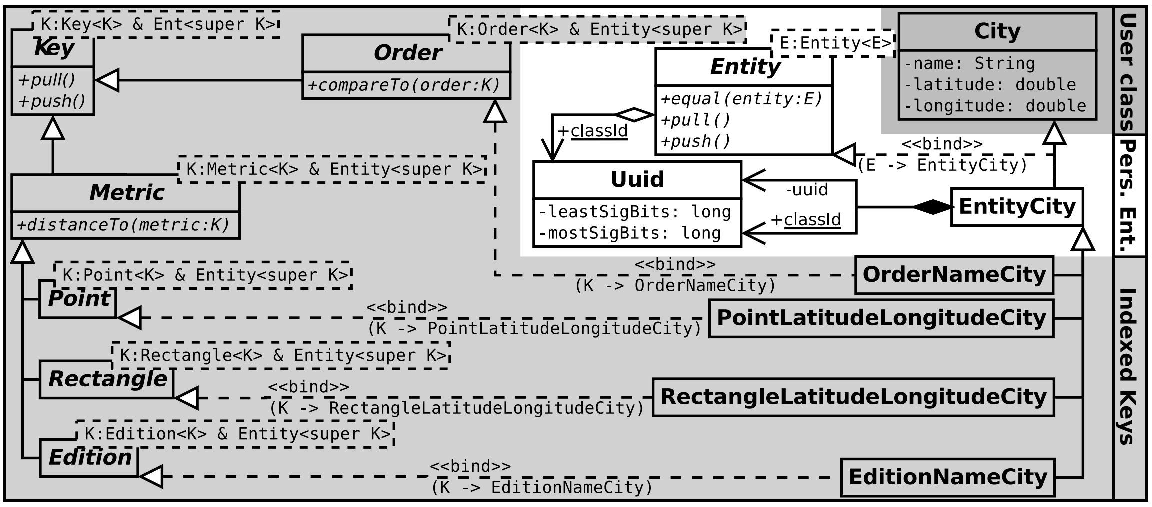


Fig. 2. The Metaclass module

The first group, the persistent entities, extends the user-defined classes and implements the *Entity* interface. The extension avoids modifications in the user-defined classes. The *Entity* implementation provides the methods required for persistence. An example of a persistent entity is *EntityCity*, as shown in Figure 2. This class implements the *Entity* interface and specializes the *City* class, here assuming that "cities" would be what the user wants to persist.

The *Entity* interface assigns four responsibilities to persistent entities: object serialization, object equivalence, object uniqueness and class identity.

Object serialization is performed by using auxiliary stream handler classes (omitted in Figure 2). These classes are used in *push* and *pull* methods in order to convert objects to their serialized format.

Object equivalence checks if two distinct entities have the same attribute values. To ensure that only objects from the same class are compared, *Entity* is a parameterized class and their specializations follow the CRTP. A Boolean method, called *equal* (Figure 2) performs the comparison.

Object uniqueness and class identity are achieved using two UUID [Leach et al. 2005] attributes. The first one, called *uuid*, is used by the data structure to retrieve instances of each stored object. Its value is always unique, ensuring the uniqueness of each object instance. The other attribute, named *classId*, is a class scope attribute whose value is unique among all *Entity* class implementations, regardless of the application. The *classId* is used to ensure that a retrieved object is indeed an instance of the correct class.

The second group, *indexed keys*, allows persistent entity attributes to be associated to a key domain. Key domains are classified as ordered, punctual, rectangular and edition.

The ordered domain satisfies the total order relation (transitivity, antisymmetry and totality) by using the *Order* interface, as shown in Figure 2. *Order* delegates to indexed keys three responsibilities: serialization, object equivalence and object ordering. Serialization and object equivalence work as aforementioned. Object ordering ensures the total order relation. It is implemented in the *compareTo* method, from the *Order* interface (Figure 2). This method compares two keys to establish a total order over a set of objects. Therefore, the *compareTo* implementation determines which attributes of an object will compose the index.

Figure 2 illustrates these concepts. The *OrderNameCity* class is a *Key* to a *City* object, in which the index attribute is the name of the city. Here, the *compareTo* method receives an *OrderNameCity* argument and evaluates if its own name comes lexicographically before the argument name. This class can be indexed in a secondary index such as a B^+-Tree.

The punctual domain associates a set of coordinates to a key, using the *Point* interface. *Point* delegates to indexed keys four responsibilities: serialization, object equivalence, metric calculation and spatial position definition. Metrics satisfy the postulates of a metric space (symmetry, non-negativity, identity and triangle inequality) and are implemented by the *distanceTo* method from the *Metric* interface. The most common metrics for points (L_p family) were implemented in auxiliary classes (omitted in Figure 2). Spatial position defines the number of dimensions and the coordinates on each dimension.

In Figure 2, the *PointLatitudeLongitudeCity* class represents a *Key* to a *City*. Here, a point, given by latitude and longitude values, defines cities, and the *distanceTo* method computes how far apart two cities are, according to the L_2 metric. The *PointLatitudeLongitudeCity* objects can be indexed in data structures like M-Tree or R-Tree.

The rectangular domain associates an area, volume or hyper-volume to a key by using *Rectangle* interface. *Rectangle* delegates to indexed keys the same responsibilities of the *Point* and a spatial extension, that is, the size in each dimension (width, height, depth...). Rectangles are used as a simplified geometry for line and polygon objects in spatial databases and such representation is referred as Minimum Bounding Rectangle (MBR).

In Figure 2, the main task of the *RectangleLatitudeLongitudeCity* class is to index cities like regions in space, using their latitude, longitude and dimensions to locate them. Generally, this class is indexed by a secondary index such as an R-Tree. However, it is possible to index *Rectangle* objects in an M-Tree, since the *Metric* interface defines a distance calculation to these objects. Roussopoulos et al. [1995] and Manolopoulos et al. [2006] briefly discuss how to compute distances between rectangles.

Finally, the edition domain establishes metric distances to chains of symbols by using the *Edition* interface. Its responsibilities are serialization, object equivalence and metric distance calculations, as previously described. The metric distances implemented for a chain of symbols are the Levenshtein family and their variations.

The *EditionNameCity* class in Figure 2 indexes cities according to a string distance between their names. This class is indexed in a secondary index such as an M-Tree.

Multiple indexing is achieved by the definition of several indices over an object. In Figure 2, there are different key domains associated to the same instance of a *City* object. Thus, this instance can be indexed in ordered, punctual, rectangular and edition domain, simultaneously.

3.2 The `Storage` Module

The `Storage` module specifies the implementation of primary and secondary index structures. These structures hold block collections in order to organize and store the previously defined objects.

The `Storage` module has two basic interfaces, which are depicted in Figure 3: *EntityStructure* and *KeyStructure*. The former defines primary index structures, responsible for storing the whole instance of an entity. The latter defines secondary indexes, storing only the indexed keys to these entities.

ObInject provides a *Sequential list* and an *Extendible Hash* [Fagin et al. 1979] for use as primary index structures. The secondary indexes can be either a *B^+-Tree* [Comer 1979], an *M-Tree* [Ciaccia et al. 1997; Zezula et al. 2006] or an *R-Tree* [Guttman 1984; Manolopoulos et al. 2006].

The `Storage` module enables new data structures to be easily created, by simply deriving them from the `Storage` classes and implementing the required methods. Moreover, this approach allows more than one data structure to share the same objects.

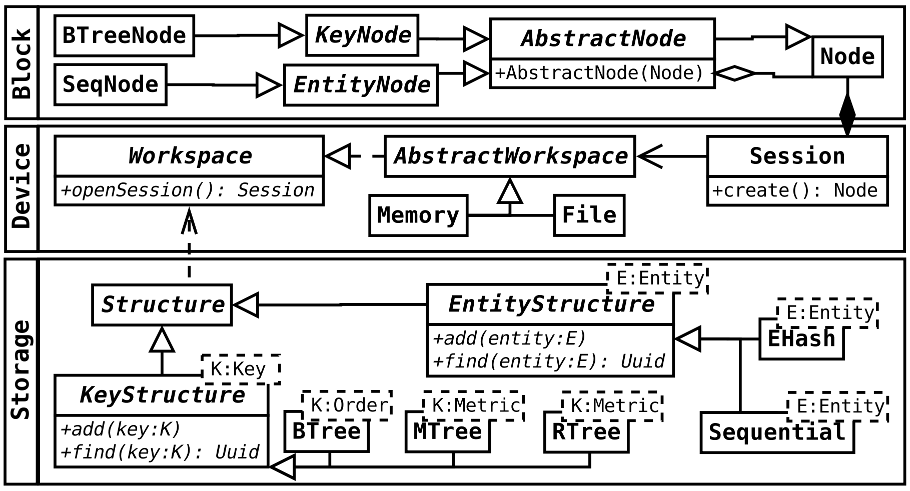


Fig. 3. The Block, Device and Storage hierarchy

The `Storage` module is directly linked to the `Device` module (Section 3.3) through a Bridge. This pattern decouples the data structures (B^+*-Tree*, *Sequential*, etc.) from their storage devices (*File*, *Memory*). Thus, they can be extended independently. According to the Bridge pattern, the *Structure* class is the 'abstraction'; in Device, the *AbstractWorkspace* class is the 'implementor', and its subclasses are the 'concrete implementor'.

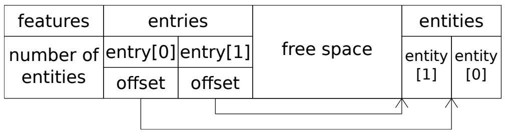
3.3 The `Device` Module

The `Device` module isolates persistent objects from storage abstractions. An object does not need to know about storage devices or about the data format used when it is persisted. Since different types of storage devices present particular properties, the natural approach to represent them is through an abstract class hierarchy. Therefore, each distinct storage type derives from this abstract hierarchy and implements the required operations. In this way, defining other storage types are easily performed.

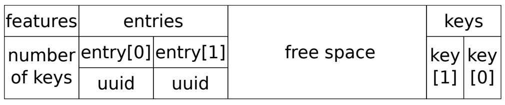
In Figure 3, the *AbstractWorkspace* class is the common interface from which all types of storage devices are specialized. It represents an area that any data structure can use to access and share data. This class is specialized into *File* and *Memory*. A *File* object handles data in a persistent storage and is keep saved even after application termination, whereas an *Memory* object handles data in a volatile storage, and these data are lost when the application finishes. These classes do not implement ACID properties, although the *AbstractWorkspace* can be extended to ensure ACID.

The *Session* class acts as a temporary storage area, keeping the recently used pages in an internal hash map to speed up access. Thus, *Session* connects `Device` and `Block` modules maintaining a hash of *Nodes*. Another responsibility of the *Session* class is to allocate space for a new block when requested by data structures.

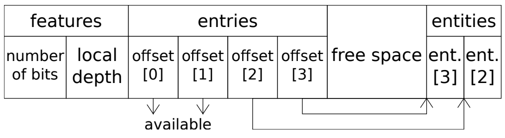
A *Session* ensures that the data structures will not access the *AbstractWorkspace* methods directly. For this purpose, the *Workspace* class provides the *openSession* method, which returns a *Session* instance. Using that instance, a data structure is able to perform storage operations regardless of the workspace, without knowing where the workspace is stored and without knowing how these operations are implemented in the corresponding workspaces. The *openSession* method follows the Factory Method pattern: the *Session* instances are its 'products' and the *Workspace* is the 'creator'.



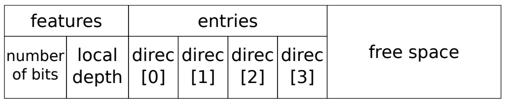
(a) Sequential Node



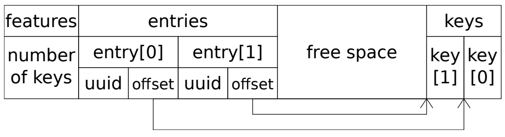
(b) R-Tree Leaf



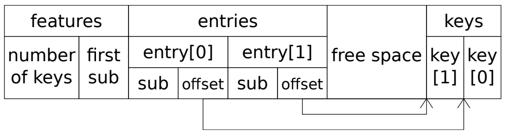
(c) E-Hash Bucket



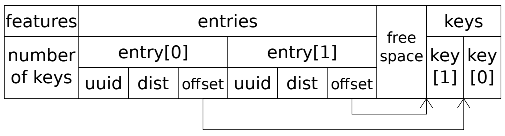
(d) E-Hash Directory



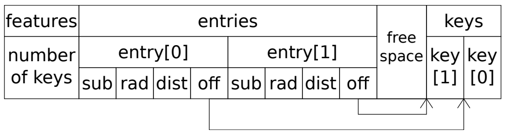
(e) B-Tree Leaf



(f) B-Tree Index



(g) M-Tree Leaf



(h) M-Tree Index

Fig. 4. Blocks organization

3.4 The Block Module

The Block module defines the format to store and retrieve data from pages (or blocks). A page is defined as an array of bytes. This is the usual format to store data on disk, memory or to move them through the network. Here, pages are part of the data structures and are represented by the *Node* class, as shown in Figure 3. This class provides methods allowing data structures to read and write primitive data types in a page.

The *Node* class defines a numeric identifier that allows the retrieval of a *Node* instance from a storage device. The identifier can be used as the reference to navigate to other *Nodes*.

As different data structures (e.g.: *B-Tree*, *Sequential*, etc.) may arrange a *Node* into different formats (respectively *B-Tree Node*, *Sequential Node*, etc.), it is required to abstract a *Node* in such a way that it can be specialized. For this purpose, the modeling of the node hierarchy follows the Decorator pattern. It provides a flexible alternative for the *Node* subclasses to extend their functionality. In this pattern, a *Node* acts both as a 'component' and as a 'concrete component' (as it can have instances), an *AbstractNode* is the 'decorator' and their subclasses are the 'concrete decorators'.

Data inside *Nodes* are organized into five sections: (i) *header*, containing the node identification (omitted in Figure 4); (ii) *features*, storing general data about blocks; (iii) *entries*, with information about objects; (iv) *entities/keys*, which are the objects themselves; and (v) a *free space* area. The *Node* customization for each data structure is shown in Figure 4 and described as follows.

Sequential: *SequentialNodes* stores persistent entities into blocks of a *Sequential* structure. This class derives from *EntityNode* and its block is modeled according to Figure 4(a). In this figure, the *features* section retains particular information about the block, such as the current number of entities or objects present in the node. Each object stored in a *SequentialNode* has an *offset* field, in the *entry* section. An *offset* points to the first byte of the object, at the end of the block. Between *entries* and *entities* sections, there is *free space*, used for the insertion of new objects. Since the objects are size-variant, all fixed-size information is stored at the beginning of the node, and non-fixed size information at the end. Thus, an object can be quickly located using the *offset*, besides being this the most convenient way to provide direct access to objects of any size inside a block. This approach is particularly practical because a node is not restricted to a defined number of entities, but only to the block size. In the insertion process, there must be sufficient free space in the node to insert a new object and an entry to this object. If there is not, the data structure splits the node, according to the split policy.

E-Hash: Extendible Hash nodes are specialized into leaves (buckets) and indexes (directories) and they derive from *EntityNode*. An *EHashBucket* (Figure 4(c)) corresponds to the last level of its primary index and stores persistent entities. In the *features* section, the *number of bits* parameter determines the number of entries in the block. The *local depth* indicates the initial position of bits-counting with respect to total number of bits in *EHash*. The *entries* section contains *offsets* for the objects; however, not all positions may have been mapped for the objects, therefore some *offsets* may be null. As it occurs in *SequentialNode*, there is a *free space* area between the *entries* and the *entities* sections. The *EHashDirectory* (Figure 4(d)) stores just the directory entries to navigate to the sub-levels. The *features* section contains the same fields described in *EHashBucket*.

B-Tree: The B-Tree Nodes derive directly from *KeyNode*, storing only the indexed keys. The B-Tree blocks are specialized into leaf and index blocks. A B-Tree Leaf, shown in Figure 4(e), stores keys inserted into the B-Tree, whereas a B-Tree Index stores routing objects to reach these keys. In B-Tree Leaves, the *features* section contains the number of keys inserted into the node. Each entry consists of an *offset* and a *uuid*, which is employed to retrieve the whole object associated to that key. The *entries* and *keys* sections are separated by a *free space* area. A B-Tree Index block is shown in Figure 4(f). Instead of storing a *uuid*, each entry has a *sub* value, which refers to the root of a subtree, which may be either another B-Tree Index or a B-Tree Leaf. This subtree contains keys equal or greater than the key of its entry. Since the index nodes of a B-Tree have a number of keys and an additional reference to subtrees, the *first sub* attribute in the *features* section refers to the subtree that stores the keys that are lower than the first key.

R-Tree: R-Tree Nodes also derive into leaf nodes and index nodes. Both of them are specialized from the *KeyNode* class. Figure 4(b) shows the R-Tree Leaf modeling. Unlike other data structures, keys in an R-Tree cannot be size variant. Since the Minimum Bounding Rectangle (MBR), whose size in bytes is fixed, represents the keys, they will always have the same size. Thus, its position can be directly calculated and an *offset* is not necessary. Each entry in an R-Tree Leaf stores just one *uuid*. Although the union of one *uuid* and one R-Tree key has the same size in bytes, and could be joined, the *uuids* and keys are kept separated, because *uuids* are not part of keys. The R-Tree Indexes are identical to the R-Tree Leaves, except for the *uuid* field. In an R-Tree Index, the *uuid* is replaced by the *sub* value pointing to a subtree.

M-Tree: The M-Tree Nodes follow the same design of the other nodes. The main difference of the M-Tree Nodes from the other structures is the *entry* section. Entries in an M-Tree Leaf have their fields adjusted to the peculiarities of the M-Tree, as shown in Figure 4(g). In this case, a leaf contains a *uuid*, the distance from the representative node (the *dist* attribute) and the *offset*. Unlike R-Trees, the M-Tree keys present a variant size. The entries in an M-Tree Index consist of a *sub*-value as shown in Figure 4(h), which points to the root of a subtree; a covering radius (*rad*), which is used in insertion and query operations; the distance to the representative object (*dist*) and the key *offset*.

4. EXPERIMENTS

Three datasets were used to perform the experiments. The first dataset – `Words` – consists of words extracted from OpenOffice dictionaries[3]. This dataset combines words from 28 different languages

[3]OpenOffice dictionary extension: http://extensions.openoffice.org/en/dictionaries Access: May 5, 2013

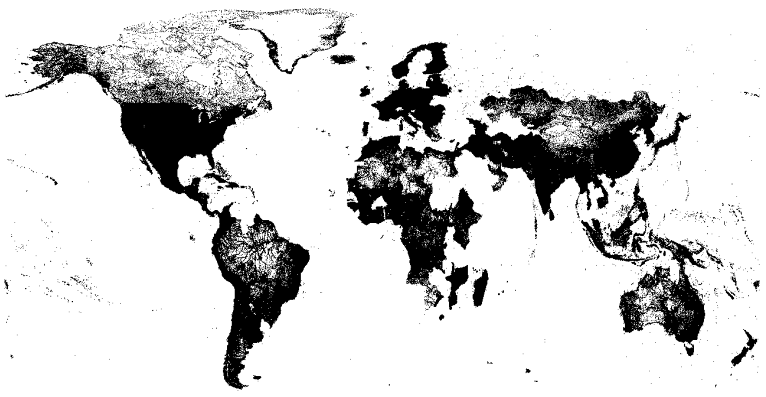


Fig. 5. GEONet points distribution

that use the Latin alphabet, totaling 4.621.240 distinct words. The dataset was pre-processed to eliminate duplicates and later shuffled at random.

The second dataset – `Points` – comes from the U.S. GEONet Names Server[4] and from the U.S. Board on Geographic Names[5]. This set contains information about geographic points (latitude and longitude) of several geopolitical areas. This file was pre-processed to eliminate duplicates and shuffled to avoid clusters, totaling 6.066.616 distinct points, distributed according to Figure 5.

The third dataset – `Proteins` – was obtained from UniProt Project[6] and consists of 78.221 proteins (chains of amino acids). This dataset was shuffled to avoid clusters and pre-processed to avoid duplicates. We retain only proteins whose length is less than or equal to 32 amino acids.

The experiments evaluated the execution time, the number of block accesses and the numbers of comparison operations on both insert operations and point query execution (i.e. retrieval of inserted objects) on secondary indexes. The experiments were performed using a computer with an Intel® Core$^{\text{TM}}$ i7-9200 processor, running at 2.67 GHz, with 6 GB of RAM, on the operating system GNU Linux, distribution Kubuntu 13.04 and HD SATA 1 TB 7200 RPM. ObInject was implemented in Java, using Oracle® JDK 7u21.

In order to evaluate the ObInject performance, the evaluations were conducted increasing the number of objects inserted into the framework. For the `Words` dataset, we considered intervals of 600.000 objects, i.e., 600.000 objects were inserted in the first measure, 1.200.000 in the second measure and so on, until completing the total size of this dataset. For the `Proteins` dataset, the interval was 10.000 objects and for the `Points`, it was 760.000 objects. We used 4096 bytes as block size for all datasets.

Each experiment was also executed on the XXL Framework. We consider only the XXL among the applications cited in Section 2 because this is the one entirely open source, it is also available in Java and, similar to our prototype, it does not perform object-relational mapping. Since XXL does not implement primary indexes, the comparison evaluations were run only with secondary indexes.

Once the XXL persistence is based on Java serialization, one of the difficulties in deal with XXL was adjust the block size. In XXL, the block size is defined as a balance between the minimum and maximum number of objects to be stored inside of them. When the objects have a fixed size, as is the case of the `Points` dataset, this balance ensures a proper utilization of the block. However, when the keys have varying sizes, their average length was used as a parameter to adjust that property.

[4]GEONet: http://earth-info.nga.mil/gns/html/namefiles.htm Access: May 5, 2013
[5]USGS: http://geonames.usgs.gov/domestic/download_data.htm Access: May 5, 2013
[6]UniProtKB/TrEMBL Fasta: http://www.uniprot.org/downloads Access: May 5, 2013

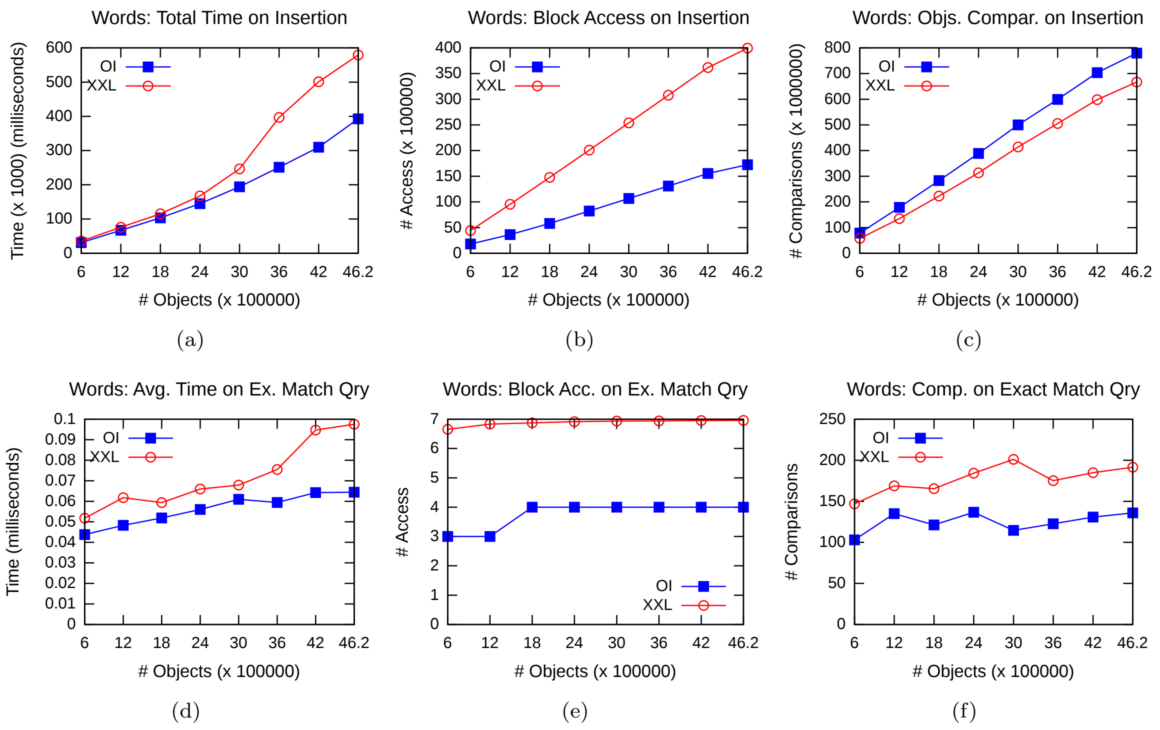


Fig. 6. Experiments over the `Words` dataset

ObInject does not need this setup, since the keys are inserted while there is available space in the block, as described in Section 3.4.

The XXL buffer cache was set to have the same height of the tree in order to minimize the gain of this resource. Since XXL does not clear its buffer cache, we perform this configuration to force XXL to retrieve a page from the disk, similar to our framework. ObInject uses a similar technique, the *Session* (see Section 3.3), but it is flushed and reinitialized for each operation invoked. Both mechanisms improve the bottom-up and top-down navigations over the tree.

The `Words` dataset was inserted in a B$^+$-Tree. In this experiment, to satisfy the total order relation, the lexicographical order of the words was considered. The results obtained are shown in Figure 6, in which "OI" indicates ObInject and "XXL" refers to XXL.

Figure 6(a) shows the total time for the insert operation and Figure 6(d) shows the average time for exact match queries. It can be seen that the behavior of both frameworks is sub-linear regarding processing time and both present an equivalent behavior. Anyway, ObInject is always at least 10% faster on both insertion and query operations.

Figure 6(b) presents the total number of disk accesses during insert operations. ObInject manages the blocks using the classes described in Section 3.3, whereas XXL delegates this responsibility to the Java serialization extensions. It can be seen that both frameworks presents a linear behavior regarding the number of objects inserted. Remarkably, ObInject performs 56% less disk access than XXL.

Figure 6(e) presents the average number of disk access for the exact match query operation. It can be noticed that for both 600.000 and 1.200.000 objects, the amount of disk accesses is the same. This occurs because both trees built by ObInject have the same height, but the leaves of the tree with 600.000 objects are sparser than the leaves of the tree with 1.200.000 objects. This also occurs in the range from 1.800.000 to 4.621.240 objects. This same figure suggests that XXL does not keep its B$^+$-Tree balanced, since the average number of disk access is fractional.

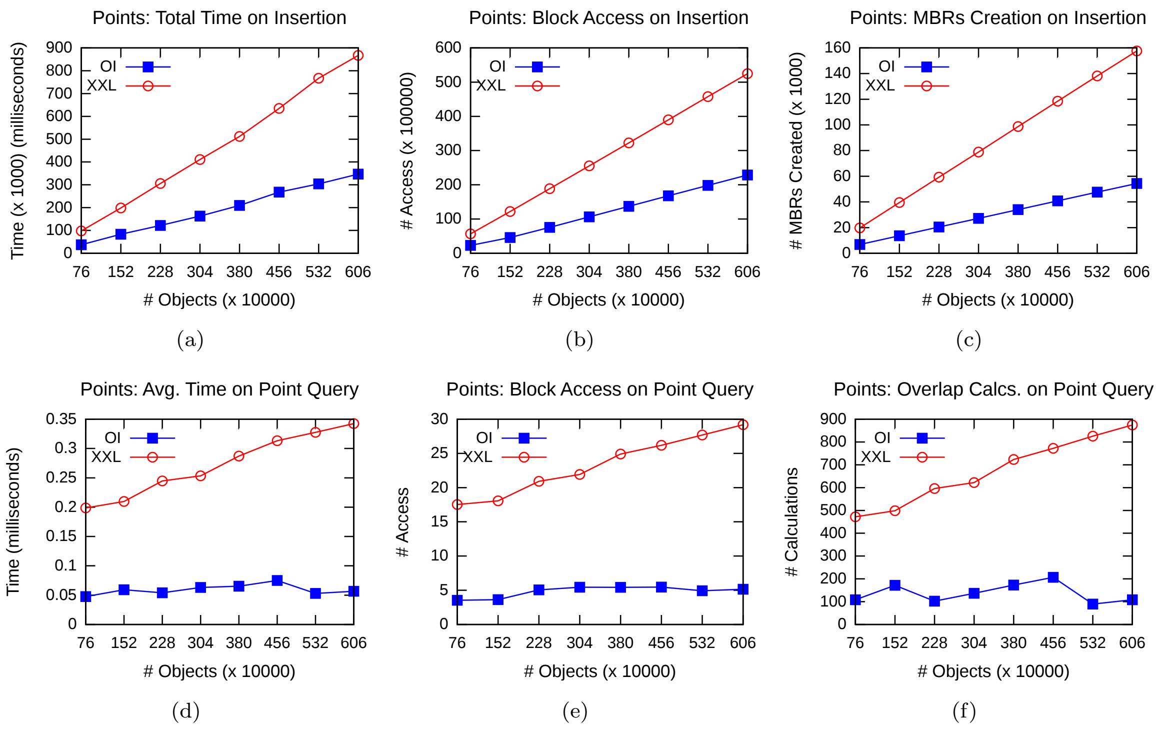


Fig. 7. Experiments over the `Points` dataset

Regarding comparison operations, Figure 6(c) shows the total number of object comparisons for the insert operation and Figure 6(f) the average number for the exact match query operation. It is important to know that ObInject uses the Insertion-Sort algorithm to add a new key into the block whereas XXL uses the Merge-Sort algorithm provided by Java. Both approaches keep the keys sorted inside each block even during split operations. Therefore, it is expected that ObInject will perform a larger number of comparisons for the insert operation when compared to XXL. This in fact happens, as it is shown in Figure 6(c). However, the average number of object comparisons performed by the query operations should not be affected by that approach, since both frameworks use binary search to locate the most suitable subtree to descend, and in fact this happens, as shown in Figure 6(f). Altogether, the tree that XXL constructs is higher than the one constructed by ObInject and XXL performs more comparisons to reach a leaf.

The `Points` dataset employed an R-Tree, and Figure 7 shows the results. For both frameworks, the quadratic split algorithm was used to divide nodes. Geographical coordinates are represented as double precision floating point numbers (`double`) in both frameworks, but ObInject is more flexible and is able to supports other types. Here, MBRs are collapsed to a point, i.e., the corners of the rectangles coincide. It can be seen that the behavior of both frameworks is sub-linear regarding processing time and both present equivalent behavior. However, ObInject is always at least 57% faster.

Figure 7(a) shows the total time required for the insert operation and, as expected, it confirms that the behavior of both frameworks is linear regarding processing time. The same behavior occurs for the query operation, as shown in Figure 7(d), but in a more pronounced way, since the quadratic split algorithm generates a reduced overlap among MBRs. Figure 7(d) also shows that the sub-linear scalability of the ObInject framework is even more remarkable for this dataset. ObInject creates a smaller number of MBRs during insertion as shown in Figure 7(c), and there are fewer intersections among them than in XXL, as shown in Figure 7(f), which shows the average number of overlap calculation in queries. Figure 7(b) shows that the total number of disk accesses to insert objects

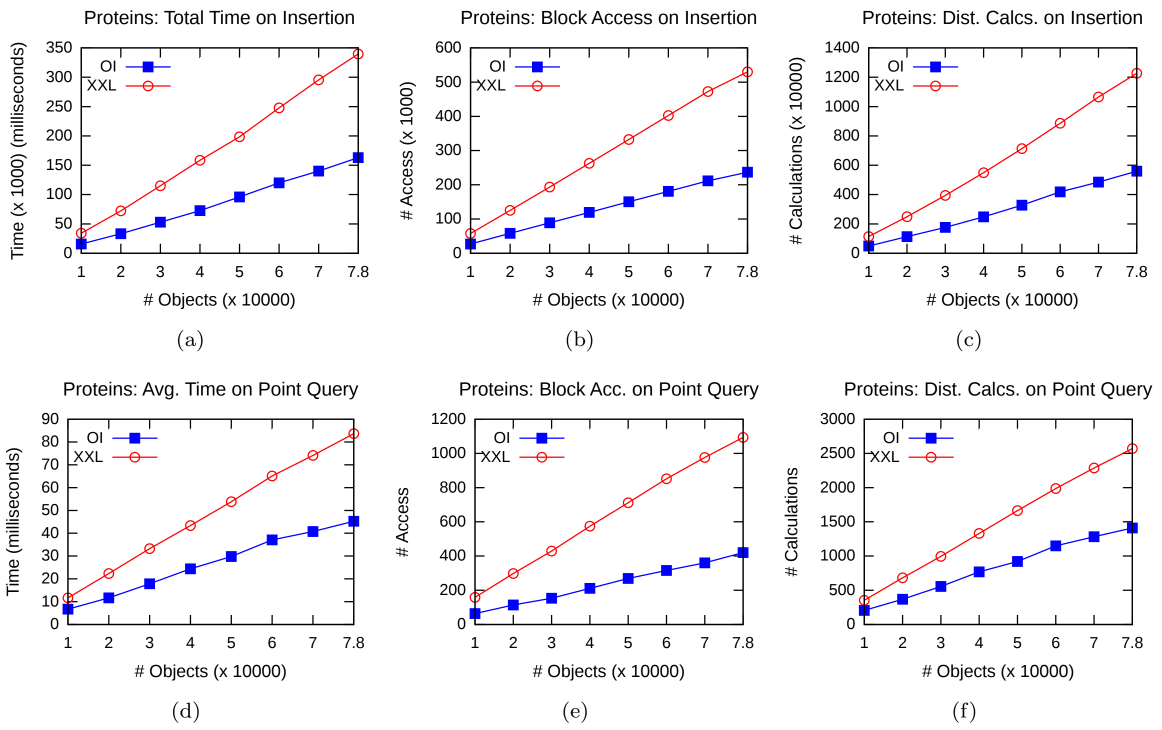


Fig. 8. Experiments over the `Proteins` dataset

is stable in both frameworks, and Figure 7(e) (average number of disk accesses in query operation) presents the same behavior as the amount of objects increases.

The `Proteins` dataset was indexed using an M-Tree. A Damerau Levenshtein metric variant was used to compare elements. This metric combines the cost of insertions, replacements, deletions and transposition with the cost of substitution of an amino acid by another, following the Matrix Point Accepted Mutation (mPAM) algorithm [Xu and Miranker 2004]. The gap penalty adopted for amino acids insertion and deletion is the highest value of the mPAM, which was set to 7. In both frameworks, the random promotion algorithm and the distribution of keys by generalized hyperplane were used. Since the original M-Tree employed in XXL Framework only computes distances by using L_p metric between points, adjustments had to be made in some methods in order to allow calculating the same metric previously cited in both frameworks.

The results of the M-Tree experiments are depicted in Figure 8. Figure 8(a) shows the total time and Figure 8(c) shows the average number of distance calculations required for the insert operation. As it can be seen, the behavior of both frameworks is linear with respect to processing time, and both present the equivalent behavior. However, ObInject now is always at least twice as faster than XXL.

It also can be seen that the total time required by the insert operation, shown in Figure 8(a), is more affected by the number of distance calculations (shown in Figure 8(c)) than by the average of number of block accesses, as shown in Figure 8(b). Figures 8(d), 8(e) and 8(f) shows that queries present the same behavior of insertions, which indicates that the measurements regarding number of disk accesses and object comparisons for query operations have equivalent impact on the time to perform point queries. In all measurements, it can be seen that ObInject always performs faster and requires a smaller number of computational resources — in fact at least half of those required by XXL.

As a final measurement, we fixed the amount of objects to be inserted and varied the block size. To generate the measurements shown in Figure 9 we used blocks of 2 KB, 4 KB, 8 KB and 16 KB.

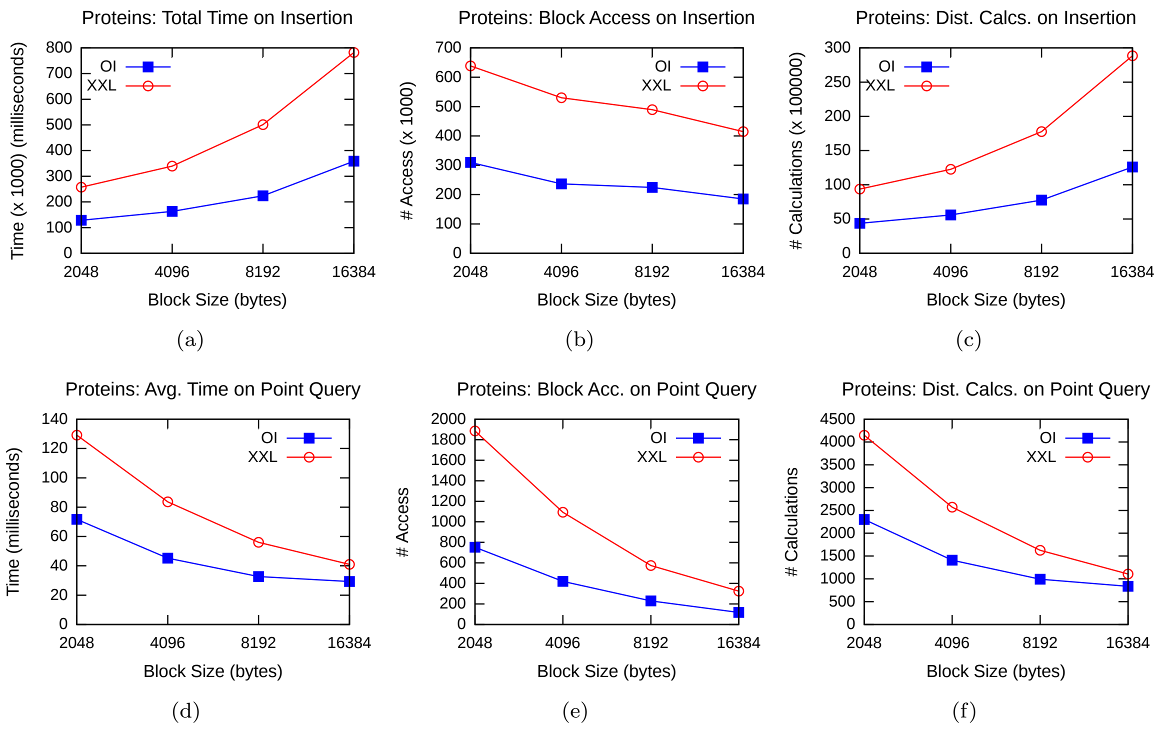


Fig. 9. Results of experiments varying the block size

Notice that every abscissa is in logarithmic scale and every ordinate is linear. The number of objects inserted in both frameworks corresponds to the entire `Proteins` dataset: 78.221. Here, due to space limitations, we only present results from the M-Tree. Figures 9(a), 9(b) and 9(c) confirm the previous results that insertion time is more influenced from the number of distance calculations than from the number of block accesses. Figure 9(e) shows that as the block size increases and a larger number of keys are stored in each block, the resulting tree has a smaller height and less block accesses are performed. Figure 9(f) shows the same, but here, a smaller number of distance calculations are required, once the tree does not have to handle splits. Both figures show that the block size causes the same impact on the query time (Figure 9(d)). In addition, Figure 9(c) points out that although the function employed to compare pairs of elements in the dataset is computationally costly, as is the case of the Damerau Levenshtein one, it is better to set the block size smaller.

Table I shows the amount of RAM memory required for both frameworks to fully insert and query each dataset, using block size of 4 KB. For the `Words` dataset, ObInject consumes 16.44% less memory than XXL. The results from the `Proteins` dataset show that both frameworks require equivalent memory, but XXL spends 04.84% less memory than ObInject for the `Points` dataset.

As a general overview, we can conclude that, for an ordered domain, ObInject executes more than 30% faster than XXL on the insertion and more than 20% faster on query answers. The results in a metric domain show that ObInject on average gains more than 50% for insertion and more than 40% for queries over XXL. Regarding punctual/rectangular key domains, ObInject is about 60% faster than XXL for insertion and 75% for queries.

Table I. Memory usage (Megabytes) measured using JVirtualVM.

Framework \ Dataset	Words	Points	Proteins
OI	370,240	508,861	548,397
XXL	443,103	485,351	549,270

5. CONCLUSIONS

The main goal of this article is to present a new object persistence mechanism, the ObInject framework. As a NoODMG solution, ObInject provides object persistence and indexing to aid in the development of user-application classes. It also provides resources for object identification, multiple indexing, serialization and storage flexibility.

The ObInject model uses the concepts of CRTP and Design Patterns in order to abstract semantic issues and allows extensions for its functionalities, allowing the support of new devices and data structures. The framework does not rely on any specific language or technique and thus it can be implemented in any object-oriented programming language.

To provide a weak dependence and coupling among classes, the framework follows a modular architecture. The `Metaclass` module provides a hierarchical class model that allows the system analyst to provide persistence to any user-application class. Moreover, this module classifies the diverse relations among the data domain, with respect to indexing, regardless of its data domain meeting either the total ordering relation, or metric or multidimensional properties.

The index implementation is performed in the `Storage` module. Distinctly from NoODMG-based solutions, the `Block` module introduces its own data organization scheme. Thus, the persistence mechanism replaces the need of a RDBMS. The `Device` module constitutes a factory of blocks, binding the `Storage` and `Block` modules. The `Device` module can also be extended to provide an easy-to-use way of defining other storage media types.

As experiments highlight, ObInject is a flexible and complete solution to provide persistence to object classes, also attaining good performance gains over competing alternatives. Furthermore, it is stable to handle multidimensional data and presents a significant and expressive performance gain to store data in metric space.

Now, the authors are planning to release the source code to the community following a free software license and building an online repository. As a work in progress, a new module for queries is been developed, that will supply several new query algorithms aimed at metric and spatial domains, as well as a new data structure and the syntax specification for a corresponding query language.

Finally, we highlight that a decentralized pool of storage types could improve the scalability and the connectivity of the data structures that provides the persistence. Examples of target types include the client-server, grid and cloud architectures. Moreover, although not strictly required, the ACID properties could be included and ensured in any of the storage types.

REFERENCES

Achtert, E., Kriegel, H. P., and Zimek, A. ELKI: a software system for evaluation of subspace clustering algorithms. In *Proceedings of the International Conference on Scientific and Statistical Databases Management*. Hong Kong, China, pp. 580–585, 2008.

Alia, M., Chassande-Barrioz, S., Déchamboux, P., Hamon, C., and Lefebvre, A. A Middleware Framework for the Persistence and Querying of Java Objects. In *Proceedings of the European Conference on Object-Oriented Programming*. Oslo, Norway, pp. 292–316, 2004.

Arboretum. www.gbdi.icmc.usp.br/arboretum, 2005.

Barry, D. and Stanienda, T. Solving the Java Object Storage Problem. *IEEE Computer* 31 (11): 33–40, 1998.

Batko, M., Novak, D., and Zezula, P. MESSIF: metric similarity search implementation framework. In *Proceedings of the International Conference on Digital Libraries: Research and Development*. Pisa, Italy, pp. 1–10, 2007.

Bauer, C. and King, G. *Hibernate in Action*. Manning Publications, 2004.

Bercken, J. v. d., Dittrich, J. P., and Seeger, B. Javax.XXL: a prototype for a library of query processing algorithms. *SIGMOD Record* 29 (2): 588–590, 2000.

Breunig, M., Butwilowski, E., Golovko, D., Kuper, P., Menninghaus, M., and Thomsen, A. Advancing DB4GeO. In *Progress and New Trends in 3D Geoinformation Sciences*. Vol. 6. Springer, pp. 193–210, 2013.

Camargo, V. V., Ramos, R. A., Penteado, R. A. D., and Masiero, P. C. Projeto Baseado em Aspectos do Padrão Camada de Persistência. In *Simpósio Brasileiro de Engenharia de Software*. Manaus, Brazil, pp. 114–129, 2003.

Carey, M. J. and DeWitt, D. J. Of Objects and Databases: a decade of turmoil. In *Proceedings of the International Conference on Very Large Data Bases*. Mumbai, India, pp. 3–14, 1996.

Cattell, R. G. G. and Barry, D. K. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: a distributed storage system for structured data. In *Proceedings of Symposium on Operating System Design and Implementation*. Seatle, USA, pp. 205–218, 2006.

Ciaccia, P., Patella, M., and Zezula, P. M-Tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Data Bases*. Athens, Greece, pp. 426–435, 1997.

Codd, E. F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* 13 (6): 377–387, 1970.

Comer, D. The Ubiquitous B-Tree. *ACM Computing Surveys* 11 (2): 121–137, 1979.

Coplien, J. O. Curiously Recurring Template Patterns. *C++ Report* 7 (2): 24–27, 1995.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating Systems Principles*. Stevenson, USA, pp. 205–220, 2007.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible Hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems* 4 (3): 315–344, 1979.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, 1995.

Graefe, G. Volcano: an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering* 6 (1): 120–135, 1994.

Graves, S. and Knizhnik, K. Simple as A, B, Tree? Applications Find Mobile Fruit in the R-Tree. *GeoWorld*, July, 2009.

Guttman, A. R-Trees: a dynamic index structure for spatial searching. *SIGMOD Record* 14 (2): 47–57, 1984.

Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. Generalized Search Trees for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases*. Zurich, Switzerland, pp. 562–573, 1995.

Kaur, H., Chauhan, R., Alam, M. A., Aljunid, S., and Salleh, M. SpaGRID: a spatial grid framework for high dimensional medical databases. In *Proceedings of the Hybrid Artificial Intelligence Systems*. Spain, pp. 690–704, 2012.

Kienzle, J. and Gélineau, S. AO Challenge - implementing the ACID properties for transactional objects. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. Bonn, Germany, pp. 202–213, 2006.

Kienzle, J. and Romanovsky, A. Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages. *IEE Proceedings - Software* 149 (3): 77–85, 2002.

Leach, P., Mealling, M., and Salz, R. A Universally Unique Identifier (UUID). IETF RFC 4122, 2005.

Leavitt, N. Will NoSQL Databases Live Up to Their Promise? *IEEE Computer* 43 (2): 12–14, 2010.

Leist, S. and Zellner, G. Evaluation of Current Architecture Frameworks. In *Proceedings of the ACM Symposium on Applied Computing*. Dijon, France, pp. 1546–1553, 2006.

Levandoski, J., Lomet, D., and Sengupta, S. The Bw-Tree: a B-Tree for new hardware. In *Proceedings of the International Conference on Data Engineering*. Brisbane, Australia, pp. 1–12, 2013.

Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N., and Theodoridis, Y. *R-Trees*. Springer, 2006.

Mattsson, M. and Bosch, J. Stability Assessment of Evolving Industrial Object-Oriented Frameworks. *Journal of Software Maintenance* 12 (2): 79–102, 2000.

Ogasawara, E. S. and Mattoso, M. Uma Avaliação Experimental Sobre Técnicas de Indexação em Bancos de Dados Orientados a Objetos. In *Simpósio Brasileiro de Banco de Dados*. Brazil, pp. 285–297, 1999.

Oliveira, A. L., Ferrari, F. C., Penteado, R. A. D., and Camargo, V. V. Investigating Framework Product Lines. In *Proceedings of the ACM Symposium on Applied Computing*. Riva del Garda, Italy, pp. 1177–1182, 2012.

Roussopoulos, N., Kelley, S., and Vincent, F. Nearest Neighbor Queries. *SIGMOD Record* 24 (2): 71–79, 1995.

Thomson, A. and Abadi, D. J. The Case for Determinism in Database Systems. *VLDB Journal* 3 (1): 70–80, 2010.

Traina Júnior, C., Traina, A. J. M., Seeger, B., and Faloutsos, C. Slim-Trees: high performance metric trees minimizing overlap between nodes. In *Proceedings of International Conference on Extending Database Technology*. Konstanz, Germany, pp. 51–65, 2000.

Weske, M. and Kuropka, D. Flexible Persistence Framework for Object-Oriented Middleware. Tech. Rep. 5, Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany, 2001.

Xu, W. and Miranker, D. P. A Metric Model of Amino Acid Substitution. *Bioinformatics* 20 (8): 1214–1221, 2004.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. *Similarity Search: the metric space approach*. Springer, 2006.

Zhu, H., Kollios, G., and Athitsos, V. A Generic Framework for Efficient and Effective Subsequence Retrieval. *VLDB Endowment* 5 (11): 1579–1590, 2012.