Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1447–1454

Java Interface for Relaxed Object Storage

Michal Danihelka, Michal Kopecký, Petr Švec and Michal Žemlička
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
Email: Michal.Danihelka@seznam.cz, kopecky@ksi.mff.cuni.cz, petsvec@tiscali.cz, zemlicka@sisal.mff.cuni.cz

***Abstract*—Most development tools manipulate objects by changing values of their attributes. If the object should change more radically, problems arise. The amount of available information can vary from instance to instance and can be collected incrementally. It can happen that there exists no class suitable for all known attributes and so even movement of the instance to another class can be complicated. We can create exhaustive number of classes to cover all predicted variants, but still some other combinations of data can occur. To solve this situation, appearing often during processing of heterogeneous and mutable data, the model of relaxed objects was invented. It is based on the idea that object classes should be defined loosely in form of conditions – presumptions on data content or availability – and that instances should belong implicitly to all classes that are currently met. Methods associated with such classes assure that each instance is provided by all currently executable methods and its behavior change dynamically with changes of its content. The paper describes the Java-based object interface for this model, its effectivity, and the domain index suitable for efficient data searching.**

I. Introduction

MOST of the object models are designed for situations when we work with groups of objects described by a known set of attributes – with classes. This approach can handle changes of attribute values but can get in troubles when new attributes should be stored and handled. The practice shows that it is sufficient for many cases. The real life is however more complicated and thus the real-life objects can fit to various classes during the time. Sometimes there could be collected some additional data, while other data could be missing. Handling such situations in most class-based object models is very complicated.

Some of the situations could be expected and supported by the application (it could be expected that person can become both parent and driver at once), some could stay hidden (person being a child under 15 and pensioner at once) and could appear during the applications service. An example of an additional attribute could be an e-mail or ICQ for an application being designed sooner than these media got known enough.

The original motivation was the need to implement a historical map that should present given area as it looked like at given time in the past. All data had to be stored temporally together with the period of their validity. The obtaining of historical information from various sources was extremely complicated. A lot of information is lost and thus amount of available/known data differed from object to object and from period to period.

Therefore we wanted an application able storing and processing such data including the originally unexpected ones. Current models and tools were not matching our needs sufficiently. We therefore developed a corresponding object model together with corresponding store ourselves [1].

The *Relaxed object model* was introduced in [2], [3]. It has been developed for handling data of such extensible applications. We have tested it also for prototyping of applications handling complex data and compared it with the currently dominating approach (the use of relational or object-relational database supported by an object-oriented language). We have focused on the development speed, variability of the solution and its resulting efficiency (throughput; it could be interesting in the case when we create single-purpose applications processing large amount of non-trivial data).

II. Background

The relaxed object model presents inner content of objects as an arbitrary set of values assigned to a subset of predefined set of attributes $A = \{a_1, a_2, \ldots, a_n\}$. While the class-based object models define strict set of classes C and assigns each object instance to one of them, the relaxed object model define classes using arbitrary Boolean condition on attribute values. We can thus imagine class *Person* as any object defining values for attributes *Name*, *Surname* and *BirthDate*. Similarly as a *Driver* can be considered any object defining values for attributes *Name*, *Surname*, *BirthDate* and *DriversLicenseNr*, i.e. any *Person* with assigned driver's license. On the other hand, class *Child* can be defined as a *Person* having *BirthDate* greater than current date minus eighteen years. We can see that both *Driver* and *Child* classes are subclasses of *Person* class.

An important difference is the very loose coupling between object instances and classes. The class or classes are assigned to object instances not explicitly by its declaration, but implicitly according to their inner content. If the object gain value of another attribute or current values of the object are changed, the object can be immediately considered belonging to different set of classes.

Methods could be formally expressed as a finite set $F = \{f_1, f_2, \ldots, f_m\}$ of data manipulation functions.

Each method $f_i \in F$ defines a condition $req(f_i)$ of its executability on a given object instance. So the method *RevokeDriversLicense* can require objects having attribute *DriversLicenseNr* set. As the condition on class *Driver* implicates the condition required by a *RevokeDriversLicense* method, this method belongs to the *Driver* class interface.

978-1-4673-4471-5/$25.00

Thanks to the concept of Relaxed Object, where the interface $iface(o)$ of given object instance o contains all methods $f_i \in F$ for whose its content fulfills the requirement $req(f_i)$, it is easy to access and use all methods available for a given object at a given moment. The availability can depend not only on physical availability of data, but also on their accessibility with respect to users' rights to individual attributes. Again, the client application can still provide users with the maximum available functionality.

The concept of Relaxed Object promises solving of some problems arising during transformation of real world entities to their abstract object model.

This paper describes basics of first implementation of these concepts that provides programmers writing in Java language with an extension allowing easy access and manipulation with relaxed object stored in the Oracle relational database. The environment selection has been made with respect to its potential enterprise deployment.

Following section discusses related work. Section 3 concerns on main issues of implementing relaxed objects in Java language and presents its results from performance point of view. Section 4 discusses enhancements currently available for the database layer that allows further performance enhancements. The last chapter gives the final conclusion.

III. Related Work

The model presumes data storage by individual columns. The *C-Store* database system [4] deals with this idea. It shows that this approach can be under some conditions quite efficient for data retrieval. To optimize reading operations, *C-Store* keeps data copies organized with respect to particular queries and so slows-down data manipulation. The *Datapile* system [5] targets to be a backup system for heterogeneous operational databases. It stores attribute values including their time validity into a single table. Data can be only imported from or exported to the classical operational databases. The approach does not allow direct data manipulation. A Java interface for relaxed objects [6] was designed to allow direct manipulation with stored relaxed objects, using classical relational database storage and still provide acceptable response time.

Software development method named Design by contract [7] tries to decrease complexity of implemented applications. It is based on the concept of a contract between two modules. Each module guarantees that if all requirements for input parameters are met, the output fulfills declared output conditions. This approach is not much used in the practice today. The problem is that definitions of requirements are not mandatory. Relaxed objects will require definitions of input conditions, as this test checks if the object belongs to supposed class or not and if the method is available. Still the condition check should be optional on production systems for performance reasons. It should be possible to switch the checking off in the code if it is obvious, that the condition is already met. The conditions could be, for example, already checked and the object was not changed afterwards.

As the implementation had to use relational database store, it was necessary to implement object-relational mapping between Java representation of relaxed objects and the database. Most of the known frameworks (for example *Hibernate*[1] for Java), provides more mapping methods or their combinations, as well chosen mapping can substantially increase the application performance and decrease the lag caused by SQL communication. The relaxed objects use quite different approach to data storage, none of common mappings – neither *vertical*, *horizontal* nor *filtered* [8] is applicable. It appeared more efficient to implement a special mapping, optimized for its specific object manipulation. One of the advantages is the possibility to add new attributes at the runtime. As the mapping performance was one of major risks, it was tested in detail.

IV. Relaxed Object Implementation in Java

This chapter describes basic ideas of relaxed object interface in Java. Among others, the solution handles:

- Optimal description of conditions for methods and their parameters.
- Optimal way of checking conditions at runtime.
- Possibility to find a correct implementation of a given method.
- Finding relaxed objects having required characteristics, i.e. fulfilling of a given condition.

A. Relaxed Object Representation

The representation of relaxed objects in the memory has to be chosen first. The focus was held on the speed of in-memory operations and the size of the data representation as we wanted to add as little overhead as possible.

We distinguish primitive data types and object data types in the further text. Primitive data types as int contain data value directly and do not need any space overhead. On the other hand, object data types store data wrapped into objects and reference to them using four byte references.

From the test results shown in the table I is obvious that the creation time depends on the object representation memory size. The object representation sizes are everytime rounded up to 8 Bytes. Class *Object* itself requires 8 Bytes of memory. *Integer* value represented as an Integer instance requires 16 Bytes of memory plus 4 Bytes for each reference. Fortunately, the memory overhead for objects with more attributes is relatively smaller. The results for *String* types have shown to be comparable with int objects (not counting memory for string characters), and so we tested mainly *int* parameters.

The results show average values from five runs, where each run executes the operation one hundred thousand times. The first two rows correspond to the cases where attributes were declared as public and so the manipulation can be done directly, not through getters and setters. A *final* modifier denies further changing of a value once it is already set.

The last three columns compare object creation time without attribute initialization, with attributes set by constructor

[1] http://www.hibernate.org

Table I
IN-MEMORY MANIPULATION WITH JAVA OBJECTS ([6])

Class Content	Size (B)	Get Time (ns)	Set Time (ns)	Creation Time (ns)		
					+Constr.	+Setters
public final int	16	10	n/a	—	25	n/a
public int	16	11	13	—	25	26
Int	16	11	17	18	25	42
2 x int	16	12	16	19	26	44
8 x int	40	20	29	46	51	98
32 x int	136	25	52	—	196	402
32 x int + sync. crit. section	136	89	91	—	189	417
32 x int + sync. method	136	92	95	—	194	428

parameters and with attribute initialization using setters after the object is created. The required initialization time increased with the growing number of parameters.

The last two rows show the time consumption in the case when the attribute access is exclusive and when it forbids parallel processing. The recommended variant that uses synchronized access only for critical section is better than the variant that requires synchronization for the whole method. In both cases the throughput of the initialization phase decreases significantly.

The relaxed objects allow any combination of set attributes what leads to at least $2^{|A|}$ theoretically possible classes. To avoid this, the relaxed object instance representation should allow arbitrary number of attributes and adapt its behavior according to actual inner content. We supposed that from the speed point of view the *HashMap* representation of attribute name – value pairs will be optimal while representation in array will provide more efficient storage. The *HashMap* representation was – surprisingly – 2–5 times slower than array representation for tens of attributes. Moreover the *HashMap* representation takes 80B plus additional 24B for each pair, while arrays require 24B plus 8B for each new pair.

Table II
MANIPULATION WITH RELAXED OBJECT INSTANCES ([6])

Class Content nr. of int	Get Time (ns)	Set Time New (ns)	Set Time Known (ns)	Creation +constr. Time (ns)	Creation +setters Time (ns)
1	30/30	106/106	57/57	30/58	156
2	36/37	107/152	56/60	29/71	296
4	41/54	107/181	59/61	31/89	623
8	53/65	106/322	68/75	30/133	1416
16	80/95	109/650	92/107	32/230	4527
32	104/188	109/650	120/208	30/502	11818
32*Integer	147/190	115/641	162/201	32/677	13060

Table II shows time required by the operations for an array representation. The values were of primitive type int. The last row shows a comparison with the values stored as an object type *Integer*. Most of the results is shown in the format best/worst case. Setting values distinguishes between changing an already existing value and a (slower) setting value for an unknown attribute. Again, the creation of an object instance distinguishes between the creation of the initialized instance and the creation of an empty object instance and the additional setting attributes one by one. The first case corresponds to the situation where attribute values are known in advance while the other models are more probable in the situation when the attributes are set after the instance creation one by one. This approach is up to 30 times slower than the manipulation with classical Java objects.

The overall memory consumption is approximately 2 times larger than in the case of classical object models.

It is possible to suppose that the users will spend significantly more time by reading and processing information than by its modification. Under such condition it was reasonable to implement two different interfaces of the relaxed object instance – *RFObject*[2] and its extension *RMObject*[3]. The first of them represents immutable object instance and provide methods *find*, *get*, *isSet* and *setCallMode*. The second one inherits it and adds additional methods *set*, *setValue*, *unset*, *amend*, *store*, and *delete*.

RFObject instance has all its values set during initialization, which makes checking conditions easier, as they can be checked only once. Methods of this class are thread-safe. The *RMObject* implementation is thread-unsafe and so it was implemented a wrapper that overrides all methods as synchronized. The programmer then can easily choose whether he/she prefers more efficient or safer object manipulation.

Attributes are defined as members of an enumerated type. Their data type is determined by the interface it implements. This way the compiler can check the type consistency. The description of the type is done using annotation. The definition then can look like:

```
// Example 1 - Attribute definition
public enum String implements RString {
  @Info ("State description")
  StateDescription
}
```

Getters and setters take attribute identification as their first parameter. So getters are declared as *get(RBytes): byte[]*, *get(RBoolean): boolean*, *set(RLong,long): void*, *set(RString,String): void*, etc. This allows compiler to check type consistency. Getters return primitive types to decrease memory and time overhead. The *amend(String): void* method loads attributes of object from the database. The *store(): void* method stores object including referenced objects to the database. The *unset(...): void* method marks an instance attribute as deprecated and the subsequent store invocation removes its value from the database.

The most important is the method *find(class<Type>): Type* that allows finding proper implementation of the object according to its inner state and required type. Return value is stored in the object cache. Method *setCallMode* allows programmer to choose if the next method invocation should:

- evaluate the conditions and execute the method, (the default behavior), or

[2]Relaxed Final Object
[3]Relaxed Mutable Object

- execute the method without checking, or
- only checks the executability condition.

B. Sets and Object Lists

As proposed in [2], the sets and lists are represented as collections of references to relaxed objects. Again, the implementation was split to the final *RFSet* and *RFList* interfaces, and their mutable *RMSet* and *RMList* extensions. The *unset* method marks the reference to the object as unused. To remove the referenced object as well, the implementation extends the interface by the method remove. The method *filter* unsets all objects that do not fulfill given condition.

C. Object persistency

Attribute declaration can add other requirements as uniqueness constraint or index enforcement for the attribute. An application can contain a method that goes through all the available classes, creates missing tables for the attributes in the database and registers all the attributes in the catalog. The implementation supports also transient attributes of instances that can hold temporary data during object stay in the memory and that are not stored in the database. To define a transient attribute, *Info* annotation from the example above should be changed to *Transient* annotation.

Transaction support allows setting of isolation level of the transaction or set the transaction as read only. Attributes and/or instances can be explicitly locked in either shared or exclusive mode. An application can request locking objects in the order of the increasing object ID's, which decreases probability of deadlock occurrence.

D. Methods and Class Definitions using Conditions

One of the main goals was to implement suitable way for declaration of conditions for method executability and for their parameters. As a solution the annotations were chosen similarly to attribute declaration. The next interesting problem was the enforcing of a transparent condition checking. The instrumentation of bytecode was chosen as the most feasible implementation. This special Java feature allows us to modify the bytecode of the class in the time of loading it to the memory. The last main problem was the way allowing relaxed object instances calling any method available according to their inner state. The solution through proxies was chosen here. Each method defines its interface. The Java then allows generation of an implementation for a given set of interfaces.

```
// Example 2 - Method definition
public interface Foo {
  Object bar (Object obj) throws BazException;
}
public class FooImpl implements Foo {
  Object bar (Object obj) throws BazException {
    // ...
  }
}

// Example 3 - Proxy Object Implementation
public class DebugProxy implements
  java.lang.reflect.InvocationHandler {
  private Object obj;
  public static object newInstance(Object obj) {
  return java.lang.reflect.Proxy.newProxyInstance (
    obj.getClass().getClassLoader(),
    obj.getclass().getInterfaces(),
    new DebugProxy(obj));
  }
  private DebugProxy(Object obj) { this.obj = obj; }
  public Object
  invoke(Object proxy, Method m, Object[] args)
    throws Throwable
  {
    Object result;
    try {
      system.out.println("before method " +
          m.getName());
      result = m.invoke(obj, args);
    } catch (InvocationTargetException e) {
      throw e.getTargetException();
    } catch (Exception e) {
      throw new RuntimeException(
            "unexpected invocatÝon exception: "
            + e.getMessage());
    } finally {system.out.println("after method "
            + m.getName());
    }
  return result;
  }
}

// Example 4 - Proxy Object Usage
Foo foo =
  (Foo) DebugProxy.newInstance (new FooImpl());
  foo.bar(null);
```

E. Queries

The relaxed object interface in Java tries to hide the database and provides to the developer an object oriented interface. Queries are thus provided through methods taking the conditions and returning either *RFList* or *RMList* implementations according to the necessity to modify results programmatically.

F. System Catalogue

The library keeps information about defined attributes in a class defined as one of relaxed object classes. The *RSetAttributes* attribute represents set of attributes having a defined value in a given instance. Due to the reflexivity of the system catalogue definition, the application can query the catalogue using the same way as other stored data.

```
//Example 5 - System Catalogue Class Definition
public class systemCatalogue {
  public enum String implements RString {
      @Info(value="Name of attribute", unique=true)
    RAttribute,
      @Info("Type of attribute")
    RType,
      @Info("Unit of attribute (e.g. ms,s,CZK,USD)")
    RUnit,
      @Info(
        value="Description of attribute",
        uniqu=true
        )
    RDescription
  }
  public enum Integer implements RInteger {
      @Info(
         value="Index of attribute
                for RSetAttributes",
        unique=true)
    RSetIndex
  }
```

```
  public enum Boolean implements RBoolean {
      @Info(
      "Information if values of attribute
       have to be unique"
      )
    RUnigue
      @Info(
      "Information if index is created
       for attribute"
      )
    RIndex
  }
  public enum Bytes implements RBytes {
      @Info(
     "Bit array of set attributes of relaxed object"
      )
    RSetAttributes
  }
}
```

The main weakness of the provided implementation is the performance of a persistent layer due to splitting of objects to more tables for individual attributes. Searching for k attributes of a given relaxed object requires joining of at least k tables according to equal ID. The search effectiveness in relaxed object storage without support of any specially designed index is shown on figures 1 and 2.

Results show that the time needed for searching grows significantly with the number of stored objects. The querying is complicated also due to the fact, that search can be done according to at most one indexed column and other conditions have to be checked against data stored in other tables and joined by object ID.

To allow effective searching, special type of index had to be designed and implemented. We needed index embedded to some widespread enterprise database management system to support large number of applications and application environments. We choose an Oracle database for its support of extensible indexing through data cartridges[4]. Its first implementation was described in [9]. The index has been significantly evolved and further optimized later.

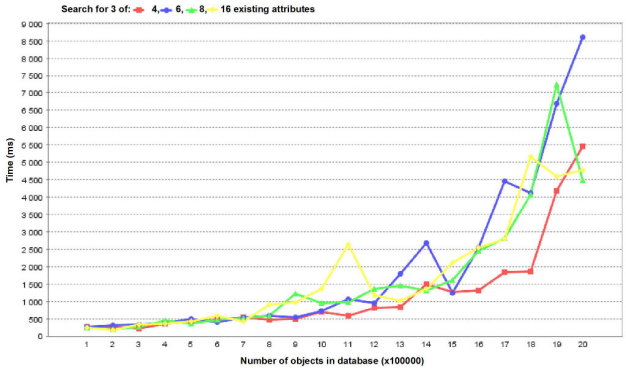

Figure 1. Object search in Relaxed Object Database - 3 attributes

[4]http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14289/toc.htm

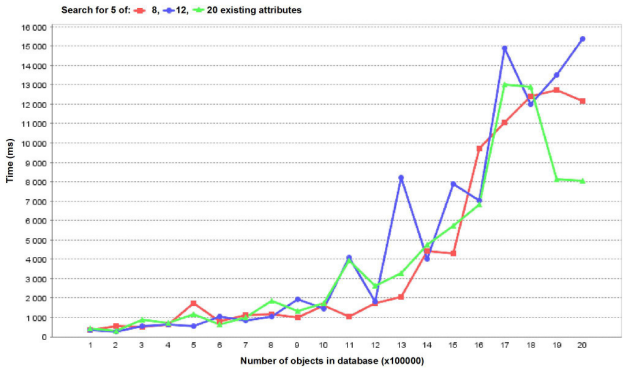

Figure 2. Object search in Relaxed Object Database - 5 Attributes

V. Temporal Relaxed Object Index

Due the restriction, given by predefined domain index interface in Oracle our indexes are created in two phases. First, they are created as *abstract* ones. In this phase the index holds only the metadata, not the data from the indexed attribute set (table set). In this state it is possible to attach new attributes to it, or detach unnecessary ones. At this moment the index is switched to *real* index, it becomes available for query optimization. Real indexes store data in R-tree structure [10] with parameters M and N. The parameters can be set at the creation time. Otherwise the optimal values are set automatically using built-in heuristics. The heuristics tries to find such a largest N that one node can be read by one I/O operation.

Let us begin with an example over Relaxed Object model storing information about people in attributes *Name*, *Surname*, *Degree*, *DrivingLicense*, *NrOfChildren*, and *Income*. The statistical office would like regularly check relation between education degree and the income, while some company would like to find potential customers. For these purposes two different multi-table indexes could be created:

```
-- Abstract index COMPANY creation
EXECUTE INX.CONSTRUCT('COMPANY');

-- Adding attributes
EXECUTE INX.ADD_ATTRIBUTE(
  'COMPANY','DrivingLicense');
EXECUTE INX.ADD_ATTRIBUTE(
  'COMPANY','NrOfChildren');
EXECUTE INX.ADD_ATTRIBUTE(
  'COMPANY','Income');

-- Switch index to real state and index data
EXECUTE INX.MAKE_REAL('COMPANY');

-- Create and populate index for statistics
EXECUTE INX.CONSTRUCT('STATISTICS');
EXECUTE INX.ADD_ATTRIBUTE(
  'STATISTICS', 'Degree');
EXECUTE INX.MAKE_REAL('STATISTICS', 20, 50);

-- Switch index back to abstract one,
-- add new attribute
-- and switch back to real index
EXECUTE INX.MAKE_ABSTRACT('STATISTICS');
EXECUTE INX.ADD_ATTRIBUTE(
  'STATISTICS', 'Income');
```

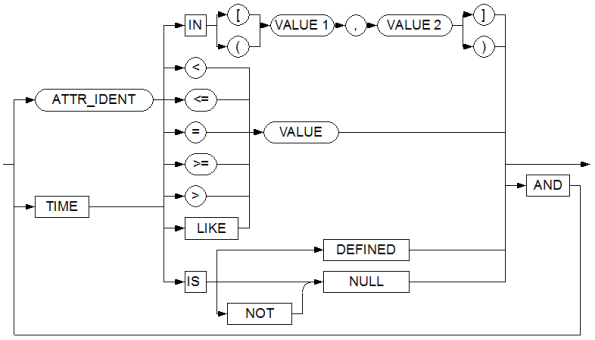

Figure 3. Query language grammar

```
EXECUTE INX.MAKE_REAL('STATISTICS');
```

All R-trees over all sets of columns are held within one domain index instance built on column ID of table $OBJECTS$. The select statements then looks like

```
SELECT * FROM OBJECTS
WHERE MATCH(ID,
  'conjunction of conditions on attribute values'
  ) = 1.
```

The query language used within the string parameter of this operator was defined to be as similar to SQL language as possible. With the exception for standard Boolean operators and comparison operators it allows easy definition of interval queries.

Index search then choose the best R-tree available to search object instances according to known attribute values. This is advantageous as the user need neither know all sets of indexed columns nor decide the specific one to be used.

Queries can either let the datastore choose optimal index, hint appropriate indexes, or force it to explicitly use given R-tree index. The SELECT statement then could look as follows:

```
SELECT *
FROM OBJECTS
WHERE MATCH(ID,
'Income >= 8000 AND NrOfChildren IN[1, 99)
 AND DrivingLicense IS NOT NULL') = 1;

-- Hinting index quality
-- by adding one point to COMPANY
-- and decreasing two points to STATISTIC
SELECT *
FROM OBJECTS
WHERE MATCH(ID,
'/* +COMPANY, --STATISTICS */ Income >= 8000
 AND NrOfChildren IN[1, 999)
 AND DrivingLicense LIKE "%B%"
 AND DEGREE = "MGR."') = 1;

-- Forcing search by index STATISTICS
SELECT *
FROM OBJECTS
WHERE MATCH(ID,
'/* STATISTICS */ Income >= 8000
 AND NrOfChildren IN[1, 999)
 AND DrivingLicense = "B" AND DEGREE = "MGR."') = 1
```

A. Search Language Grammar

The Figure 3 shows the grammar schema. The n-dimensional query is represented by a conjunction of conditions over more attributes – dimensions. Each atomic conjunction consists of the name of attribute or keyword *TIME*, of the comparison operator and of the value (1a). Each dimension can set at most one lower and at most one upper limit (1b). If the boundary for given attribute is not set, the value *MIN_VAL*, respectively *MAX_VAL*, defined internally inside the library for each database type and representing $+\infty$ and $-\infty$ is used.

The equality operator sets both upper and lower limit for given attribute (1c). To simplify interval queries, the *IN* operator was introduced. It defines both limits for the dimension delimited by a comma. Square brackets are used for closed interval boundaries, while the rounded brackets define open boundaries (1d). The value of *VALUE1* has to be less or equal to the value of *VALUE2*. The operator *IS NULL* is used to test equality with *NULL* value as it is usual in the SQL language. The *IS NOT NULL* operator is equivalent to search over closed interval *[MIN_VAL, MAX_VAL]* (1e). The *IS DEFINED* operator is used to find all objects with the value set to any value including the *NULL* value. It is equivalent to "*IS NULL* or *IS NOT NULL*". The *LIKE* operator is allowed only for textual attributes and works accordingly to its SQL equivalent. The following examples show different forms of queries.

```
-- (1a)
'NrOfChildren >= 1 AND NrOfChildren < 999
AND DrivingLicence = "B"'
-- (1b) - error: lower limit set twice
'NrOfChildren >= 1 AND NrOfChildren > 1'
-- (1c) - equiv. to 'DrivingLicence = "B" '
'DrivingLicence >= "B" AND DrivingLicence <= "B" '
-- (1d) - equiv. to 'NrOfChildren IN[1,999)'
'NrOfChildren >= 1 AND NrOfChildren < 999'
-- (1e) - equiv. to 'Degree IN[MIN_VALUE,MAX_VALUE]'
'Degree IS NOT NULL'
```

Syntax of values depends on their data type family – *integer*, *float*, *text* or *datetime*. Details for different database types are shown in Table III.

Strings have to be enclosed in double quotas ". The ordering and comparison is defined by Oracle function *NLSSORT* and is case insensitive. The date and timestamp values are enclosed in double quotas as well. If the length of the literal is shorter than expected, it is completed automatically with respect to the required format. So e.g. the value "2013-05" is by default completed to the timestamp value "2013-05-01 00:00:00.000000000".

B. Heuristics Language Grammar

All hints affecting the heuristics are optional and if present, they must be enclosed in comment brackets /* and */ at the beginning of the query string. The hint grammar is declared on Figure 4. Hint *NONE* explicitly forbids the use of any existing R-tree and data are searched programmatically. If the user wants to use one particular R-tree index, he or she can force its usage by writing its name to the comment. It is possible to write down the list of more R-tree indexes separated by commas. Each index name can be prefixed by a sequence of plus or minus signs. The heuristics then favors, respectively suppresses their usage accordingly to the prefix lengths.

Table III
SUPPORTED DATA TYPES

DB Type	Oracle format	Max	Min_Val	Max_Val
NUMBER(9)	FM999999999	10	-999999999	999999999
NUMBER(12,3)	FM999999999.000	14	-99999999.999	+99999999.999
TIMESTAMP(6)	FXYYYY-MM-DD HH24:MI:SS.FFF	21	0000-01-01 00:00:00.000	9999-12-30 23:59:59.999
VARCHAR2(16)		16	CHR(0)	16xCHR(255)
VARCHAR2(32)		32	CHR(0)	32xCHR(255)
VARCHAR2(64)		64	CHR(0)	64xCHR(255)
VARCHAR2(128)		128	CHR(0)	128xCHR(255)

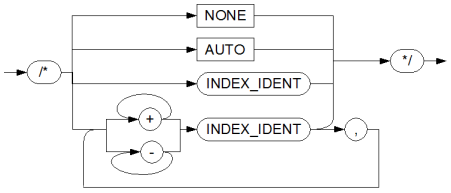

Figure 4. Hint comment grammar

C. The Embedded Heuristics

The optimal R-tree that allows optimal evaluation of given query is chosen by the embedded heuristics. It takes into account two basic factors. First it tries to avoid filter search phase. Second it tries to minimize the number of I/O operations between the R-tree stored in the BLOB and the operational memory. More formally, the heuristics tries to use such a suitable R-tree that provides maximal value of expression (1). Weights H_1 to H_4 are still subject of further research and change.

$$H_1 \frac{N_{Ind}}{N_{Attr}} + H_2 \frac{P_{Ind}}{P_{Tot}} + H_3 \frac{N}{S_{Ch}} + H_4 U_{Pref} \qquad (1)$$

The first expression $H_1(N_{Ind}/N_{Attr})$ tries to identify the biggest subset of indexed attributes. N_{Ind} represents the number of indexed attributes in the query. N_{Attr} stands for the number of attributes in the query. Optimally the number of indexed attributes corresponds with the number of attributes in the query and so $N_{Ind}/N_{Attr} = 1$. The H_1 value is currently set to 3.

The second expression $H_2(P_{Ind}/P_{Tot})$ searches for the index with as much pages read into the cache as possible. P_{Ind} denominates the number of pages already read into the index cache. P_{Tot} then holds the number of all pages that forms the index. Current value of H_2 is set to 2.

The expression $H_3(N/S_{Ch})$ prefers indexes with smaller sizes of attributes, as they can store more rows of indexed data into one index node. N is a parameter of the R-tree, while S_{Ch} stands for system chunk-size. H_3 is set to 1.

The last expression $H_4 U_{Pref}$ takes into account the user preferences. H_4 is currently set to 1. U_{Pref} is defined by the length of plus/minus prefix in the hint. For example, in the case of the hint /* ++COMPANY, -STATISTICS */ the U_{Pref} value is equal to +2 for R-tree *COMPANY* and to -1 for R-tree *STATISTICS*.

Following examples show the influence of hints and further aspects to the R-tree selection:

```
-- index COMPANY will be used (by heuristics)
SELECT * FROM OBJECTS
WHERE MATCH(ID,
'Income >= 8000 AND NrOfChildren IN[1, 9)
  AND DrivingLicense IS NOT NULL') = 1;

-- with LIKE, the STATISTICS will be used;
-- without it, the COMPANY one
SELECT * FROM OBJECTS
WHERE MATCH(ID,
'/* ++COMPANY, -STATISTICS */ Income >= 8000
  AND NrOfChildren IN[1, 9)
  AND DrivingLicense LIKE "%B%"
  AND DEGREE = "MGR."') = 1;
```

D. Combined Search of Data

All searches supported by domain index are primarily done through R-tree indexes. As R-tree indexes are not and cannot be built on all possible subsets of attributes, it is necessary to combine index search with additional comparisons of found object instances with the query.

In this case the search runs in two phases. The first index search phase – *index search* – uses the best available real R-tree index to find out object instance candidates. In the second phase – *filter search* – compares remaining attributes if they match to the user query or not. The obtained result set is then propagated to the application. The filter search evaluation is more time-consuming, but fortunately it is necessary to use it usually for relatively small amount of objects candidates only. If the sufficient R-tree index exists, it is not used at all.

E. Performance Tests

We tested the search speed using real domain indexes with optimal value of N parameter against searches, written in standard SQL both without any index support as well as using a join over *ID* columns supported by standard B-tree indexes on them. Tests run on *Oracle* 11gR2 XE on the PC with the dual-core processor Intel® Pentium® at 2.4 GHz and 4GB RAM DDR3. Third we tried to measure the search with *NONE* hint, so all instances were searched by filter search. Search times achieved over the database are shown in Table IV. Times in seconds spent by domain index search are listed in the column *REAL*. Search times needed by standard SQL search are in column *ORACLE*. The last variant is shown in column *NONE*.

```
-- REAL index STATISTICS
SELECT * FROM OBJECTS
```

```
WHERE MATCH(ID,
'/* STATISTICS */ Income >= 8000 AND DEGREE = "MGR."
AND TIME < "2100"') = 1;

-- ORACLE search
SELECT COUNT(DISTINCT Income.ID)
FROM Income I INNER JOIN DEGREE D
  ON (I.ID = D.ID
      AND ((I.VALID_TO > D.VALID_FROM
            AND I.VALID_FROM < D.VALID_TO)
           OR
           (D.VALID_TO > I.VALID_FROM
            AND D.VALID_FROM < I.VALID_TO)
     ))
WHERE I.VALUE >= TO_NUMBER('8000',
                           TYPES.GET_FORMAT('INT'))
  AND D.VALUE = "MGR."
  AND I.VALID_FROM < TYPES.TO_DATETIME(2100)
  AND D.VALID_FROM < TYPES.TO_DATETIME(2100);

-- NONE R-tree index
SELECT * FROM OBJECTS
WHERE MATCH(ID,
'/* NONE */ Income >= 8000 AND DEGREE = "MGR."
AND TIME < "2100"') = 1;
```

The database was populated with 10, 20, respectively 30 thousands of object instances with two attributes each. The attributes have had 8 to 15 unique values. Queries returned approximately 25% of the database content.

The table shows that the time spent by index creation is relatively large due to tree balancing. Optimal balancing has then positive influence to further index search times.

Table IV
SEARCH TIMES COMPARISON

Objects	Items	Create time (s)	REAL (s)	ORACLE (s)	NONE (s)
10 000	118.582	174.9	0.510	13.730	25.625
20 000	237.695	370.5	1.160	28.020	51.640
30 000	356.974	574.2	1.410	41.950	80.640

The same results are shown graphically on 5. It can be easily seen the major differences in times. Note that the results for Oracle SQL search were obtained in database without additional B-tree indexes on attribute values. Introducing them speeds up the Oracle SQL queries 3 to 4 times in the case of low dimensional queries. The queries on more attributes where the SQL join takes most of the time the speed-up was not so evident. The graph shows also an apparent increase of time between 10 and 20 thousands of object instances. It is caused by returning results in batches what implied more time-consuming calls from the PL/SQL to the C++ code. The tests were run with the batch size of 2000 objects.

VI. CONCLUSION

Using technologies as *Annotations*, *Annotation Processing* and *Java Instrumentation API*, available in Java 6.0 language, the concept of relaxed objects was successfully implemented including polymorphism that was originally not considered. Future planned extensions of Java that suppose storing method names together with parameter names and types in the byte-code in the retrievable way would make condition definitions

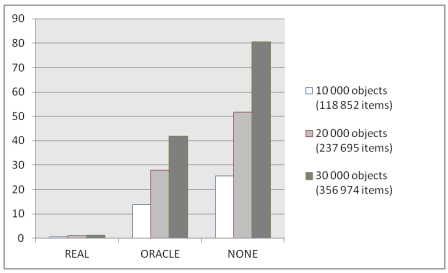

Figure 5. Search Times Comparison in Graphical Form

even simpler. The suitability of the above described implementation was proved by the pilot application – storage for software test management.

Advantage of the model is its ability of maintain heterogeneous data. The checking of method availability on given instance made the application more fault-tolerant. It appears that relaxed objects seem to be a good choice for prototyping applications having complex or heterogeneous data. Current state of implementation of specialized domain index promises even better performance and greater usability of this concept in the near future.

ACKNOWLEDGMENT

This research was partially supported by Charles University research funds PRVOUK as program P46.

REFERENCES

[1] M. Žemlička, J. Anděl, M. Bělocký, A. Buble, R. Doulík, P. Daněček, and D. Veselý, "gmap," FreeGIS CD v1.1 by Intevation GmbH, 2001.
[2] M. Kopecký and M. Žemlička, "Rozvolněné objekty (in Czech: Relaxed objects)," in *DATAKON 2004*, K. Ježek, Ed. Brno, Czech Republic: Masaryk University, 2004, pp. 243–252.
[3] ——, "Relaxed Objects - Object Model for Context-Aware Applications," in *2009 IEEE 33RD International Computer Software and Applications Conference, Vols 1 and 2*, ser. Proceedings - International Computer Software & Applications Conference, IEEE. 345 E 47th st, New York, NY 10017 USA: IEEE Computer Society, 2009, Proceedings Paper, pp. 898–903, IEEE 33rd International Computer Software and Applications Conference, Seattle, WA, JUL 20-24, 2009.
[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-Store: A Column-Oriented DBMS," in *VLDB*, Trondheim, Norway, 2005, pp. 553–564.
[5] D. Bednárek, D. Obdržálek, J. Yaghob, and F. Zavoral, "Data integration using datapile structure," in *Proceedings of the 9 th East-European Conference on Advances in Databases and Information Systems, ADBIS 2005*, Tallinn, Estonia, 2005, pp. 178–188.
[6] M. Danihelka, "Úložiště pro rozvolněné objekty (in Czech: Data Store for Relaxed Objects)," Master's thesis, Charles University, Prague, 2009.
[7] J. Rieken, "Design by contract for java - revised," Master's thesis, Department für Informatik, Universität Oldenburg, Apr. 2007.
[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey 07632: Prentice-Hall, 1991.
[9] P. Švec, "Datové úložiště pro temporální rozvolněné objekty (in Czech: Data Store for Temporal Relaxed Objects)," Master's thesis, Charles University, Prague, 2010.
[10] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '84. New York, NY, USA: ACM, 1984, pp. 47–57. [Online]. Available: http://doi.acm.org/10.1145/602259.602266