periodica polytechnica

Electrical Engineering
55/3-4 (2011) 103–110
doi: 10.3311/pp.ee.2011-3-4.02
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

A study of various techniques for storing inhomogeneous descriptive data of digital maps

Roberto Giachetta / *Zsigmond* Máriás

Received 2012-07-09

Abstract

In contemporary Geographical Information Systems (GIS) the large variety of digital map sources requires the handling and storing of various descriptive data in a single storage facility. Although the number and type of attributes can vary among different maps, the storage of such inhomogeneous data in a single database is difficult, as querying is an essential task and requires fast retrieval of data based on any present attribute.

In this study, the authors compare three different approaches to this problem based on relational and object-oriented database systems by implementing and testing with massive inhomogeneous and altering descriptive data. Since this problem is not only typical in the field of GIS, the solution can be applied generally to any domain using inhomogeneous data, like e-commerce systems and document warehouses.

Keywords

geographical information systems · large-scale spatial data storage · document-oriented databases · object-oriented databases · performance analysis

Acknowledgement

The research is supported by the TÁMOP-4.2.1/B-09/1/KMR-2010-003 project.

Roberto Giachetta

Department of Software Technology and Methodology, Eötvös Loránd University, H-1111 Budapest,, Hungary

Zsigmond Máriás

Department of Algorithms and Applications, Eötvös Loránd University, H-1111 Budapest,, Hungary

1 Introduction

Digital maps are the most important data source of Geographical Information Systems, and publishing of maps is an essential governmental task. For several years, the *Eötvös Loránd Universiy* (ELTE) Faculty of Informatics has been developing a digital map database server known as EDIT (short for University Digital Map Server in Hungarian) with the foremost aim of assisting university scientific research projects and education by providing online access to various digital raster and vector maps, aerial and satellite imagery (see [1]). The EDIT system makes it possible to perform queries based on any descriptive data of the maps, and soon it will be able to perform spatial queries.

Our aim is to provide a database storage and retrieval system that has the highest performance to execute database operations, including the modification of data and structural information. For this reason, several implementations have been concerned, and the most promising three ones have undergone an extensive performance measure procedure. Their results are presented in the following.

The rest of the paper is arranged as follows. In Section 2 we will introduce the problem and the abstract solutions. In Section 3 our implementations are presented, with the performance measure procedure and results in Section 4. We will conclude in Section 5.

2 The database structure

2.1 Map database of the EDIT system

In EDIT system raster and vector maps are stored in different categories. Every map category has different set of descriptive data called map attributes. For example, every raster map and satellite image contains information about image resolution and spatial resolution, while vector maps do not contain this attribute.

In the database each map category may have several subcategories, for example we distinguish topographic maps, touristic maps and remote sensed images, and among remotely sensed images there are also different categories for multispectral and hyperspectral images, and so on. Each subcategory inherits all the descriptive data of its parent category and extends it with

several additional attributes. This structure is similar to the concept of object-oriented programming, where categories correspond to classes, subcategories are provided through inheritance, and each record is an instance of the class.

Sometimes we have to alter the categories by adding or removing attributes. This also affects the subcategories. For example if we want to add copyright information to raster maps, we have to alter raster map category and then have this information inherited into every subcategory.

Beyond the maintenance of this category taxonomy, we have to store and retrieve map instances for all categories, with all the descriptive data of a certain category. Since the database stores a huge amount of maps – at the moment over 30 000 –, we need the functionality to retrieve not just single objects but a set of maps based on different filter conditions. These filter queries contain conditions based on the descriptive data, such as retrieving maps with a specific attribute value, or class conditions, such as retrieving all the objects in a specific category and its subcategories.

2.2 Generalization

The goal was to design a database structure in which we can define class inheritance taxonomy and store objects that belong to the defined classes.

We need to store name and type information for every attribute in our database. This information is called attribute schema. It may also contain additional properties such as default values and measurement units, but since this information does not affect the way objects are stored or filtered, we do not consider them further on.

Every class holds a number of attributes, which can be modified any time, so we need the ability to add and remove attributes. Each class – except the base class – must have a parent class, and all attributes of the parent class are inherited by their descendant classes.

We want to store a large amount of objects possibly for each class and we want to perform different kinds of filter queries. We need the ability to reference and retrieve a single object and also to get a set of objects based on different conditions:

- get all the objects of a certain class,
- get all objects with specific attribute values,
- combination of the two filters above: get all objects of a certain class with specific attribute value(s).

Due to the large number of objects and operations, performance properties are crucial in finding an adequate solution. We have to consider not only filtering the stored objects, but also the modifications of database. In this paper we will describe three different approaches to store and query this kind of object database and measure their performance on several operations.

2.3 Other applications

Our basic goal is to design and analyze the performance of different solutions to store a large amount of maps with various descriptive data, but the results are rather general. This kind of hierarchy of classes and objects is very useful in applications like e-commerce systems or corporate document warehouses. If we consider e-commerce systems, classes are product categories, objects are products, and attributes are product features. In the case of document warehouses, classes are document schemes, objects are documents and attributes are the document fields. Searching facilities are very important in both cases.

Another current project of ELTE faces the same challenges and is another use case of the class taxonomy. The workflow-based ERP system known as Amnis handles all business processes as workflows, and all required information as documents. Workflows work on documents that can have any attributes and can be altered any time with the ability to create custom sequences and assemble any type of document used during the workflow process. Workflows and documents can be specialized, document attributes can be extended.

3 Implemented solutions

In previous research we have studied several kinds of relational database structures (presented in [3]), from which two have been chosen: the two highest performing solutions. We will compare them with a third, document-oriented database solution, which is a rather natural implementation of our structure.

3.1 Relational database

We will describe two different approaches to store classes and object instances. The main difference between the two solutions is the way they store objects. The attribute and class schema definitions and the inheritance are described the same way. Hence, first we describe the method of storing classes and attributes.

3.1.1 Class hierarchy

Class hierarchy is stored in three tables:

- *attribute* table defines the schema information of each attribute in the system. This table stores the type of a certain attribute and its name. It is possible to add measurement unit information and default values as well.
- *class* table defines classes and inheritance relations. This table stores the class id, the name of the class and the parent class id.
- *classHasAttributes* table defines the attributes belonging to a class. Each row of this table stores a class id and an attribute id.

Creating, removing or modifying functions of classes are quite simple and can be implemented in straightforward way. There is one point that needs to be analyzed. When retrieving the attributes of a class, we have to collect the attributes

of a given class and also its ancestors, which requires multiple queries. As we have to perform this frequently – even when retrieving a single object from the database –, we should do some improvements by denormalization.

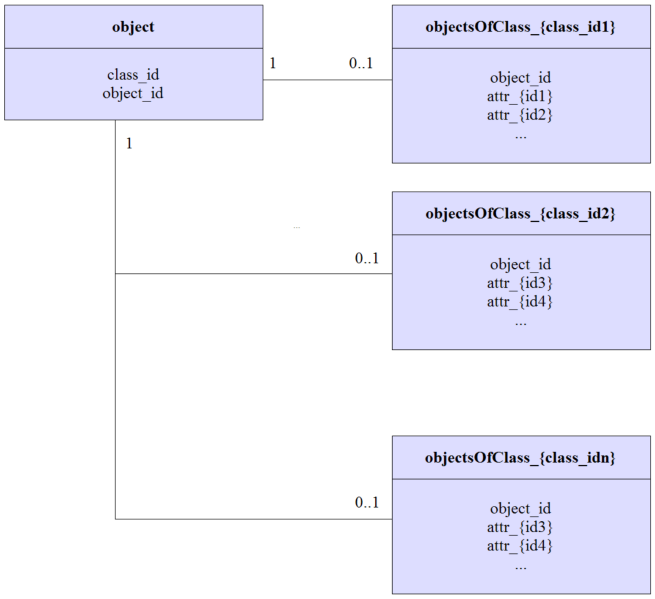

Fig. 1. On-the-fly created tables with relation to the object table

This improvement is done by adding a new field into classHasAttributes table that indicates whether an attribute is inherited or it is among the extension attributes of the given class. This results in storing the inherited attributes multiple times, causing redundancy in the database. However performance can be significantly improved. This improvement affects the class creating and modifying functions. The functions are still simple, but the database consistency has to be maintained, by adding or deleting the inherited attribute rows as well in the *classHasAttributes* table.

An example

The database consists of two classes. Class "A" is the base class, with one integer attribute called x, class "B" is a subclass of "A" and it has a text (y) and a double (z) attribute and it inherits the integer attribute x from class "A". The database will contain the following elements:

```
attribute [attr_id,attr_name,type]:
(attr_id_x, 'x','integer')
(attr_id_y, 'y','text')
(attr_id_z, 'z','double')
class [class_id, class_name, parent_id]:
(class_id_A, 'A', null)
(class_id_B, 'A', class_id_A)
classHasAttributes [class_id, attr_id, inherited]
(class_id_A, attr_id_x, false),
(class_id_B, attr_id_y, false),
(class_id_B, attr_id_z, false),
(class_id_B, attr_id_x, true)
```

3.1.2 Storing classes in on-the-fly created tables

In case of a fixed attribute schema and fixed number of classes the standard solution is to create tables for each class and store objects as records of the table. Our first solution is similar to this method, but we need to consider the frequent change of classes and attribute schema. Objects are stored as records, but a separate table is generated automatically for each class when new classes are added to the data. This implementation is similar to the one introduced in [4].

The name of the tables are *objectsOfClass_{class_id}*, and these tables contain an *object_id* and the attribute fields. For each attribute a separate column is created in the table. The name and type of the column is calculated after the attribute table's *attr_id* and type values: *attr_{attr_id}: baseTypeOf(attr_id)* This structure can be seen on Fig. 1. For the example described in Section 3.1.1 the following tables stores the objects:

```
objectsOfClass_{class_id_A}:
[object_id: int, attr_{attr_id_x}: integer]
objectsOfClass_{class_id_B\}:
[object_id: int, attr_{attr_id_x}: integer,
attr_{attr_id_y}: text, attr_{attr_id_z}: double]
```

When a class is altered by adding or deleting attributes, not only the class hierarchy needs to be changed but the tables of the class and its descendants are affected, which also needs to be considered for the altering functions. When a class is deleted, the tables of the descendants have to be dropped as well.

Although the class operations are quite complex, retrieving and filtering objects remain simple in this approach. If we want to retrieve a single object from the database, a simple select query has to be performed in the proper data table. To archieve that, an additional lookup table called objects is maintained which stores pairs of object and class identifiers. Using this table, objects can be retrieved by performing a simple query in the proper table.

Filtering objects by class is simple in this case. Retrieving a set of objects in a specific class or several specific classes can be done by simple "select" queries. If the class identifiers are given, the tables are determined in which the queries have to be performed. The queries are generated by string operations.

Filtering the objects by specific attribute value conditions can be done in two steps. These conditions are given by an attribute identifier, a relation and a value. First, the classes are determined that have the attribute with a query on the classHasAttributes table, and then a simple selection is performed in all class tables that contain the attribute. The "where" conditions are generated by string operations based on the attribute schemas. If several attribute conditions are given, then several sets of classes are calculated, and the intersection of these sets is used.

Filtering a class' objects with a specific attribute value is pretty simple; the select query has to be performed only on one table. This query is generated based on the class attribute schema and the given attribute conditions.

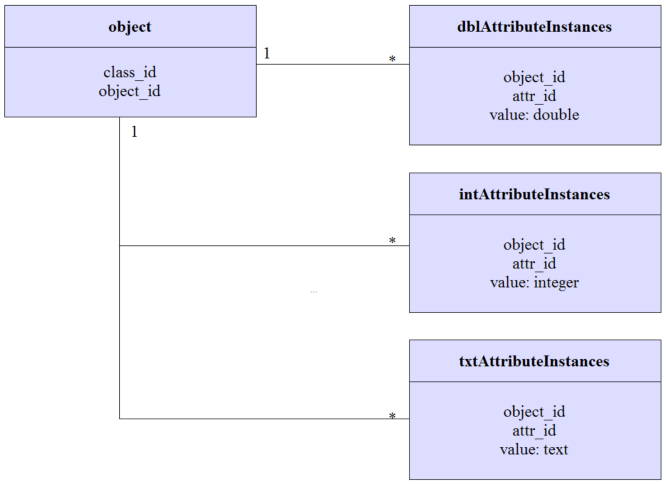

Fig. 2. Multiple attributeInstances tables with the object table

In this solution creating and modifying a class are quite complex, because these operations can have consequences (the corresponding data need to be transformed), but the filtering algorithms are quite simple, and are generated by the class' attribute schema and the filter conditions. Our expectation was that this solution will work well in searching and filtering, which is the most expensive part of usage in most applications.

3.1.3 Storing objects and attribute instances in separate tables

In our second approach, instead of generating tables for each class, attribute instances are stored in separate tables, according to two guidelines.

- Each attribute type has its own table, named *{base-type}AttributeInstances*, in which instances are stored. For example if we allow integer, text and double attributes in a system, three tables are created. These tables store an object identifier of the object the instance belongs to, an attribute identifier and the attribute value.
- The *objects* table stores the object and class identifiers as in the previous solution.

In this implementation, there is no need to create new tables, so creating a class need no further operations. When inserting an object, the attribute values are placed in the proper attribute instance tables in multiple records. Objects can be retrieved by first determining their attribute schema, then by queries in each attribute instance table and, and after that the result is processed using the attribute schema.

Filtering objects is a more complex operation. Class filter performs one query in the objects table, searching the objects with specific class identifier(s), and then, if we need the attributes of objects as well, we have to collect them from the attribute instance tables as discussed above. This structure can be seen on Fig. 2.

Filtering the objects by a specific attribute value conditions can be done in two steps. First, we get the value instances from attribute instance tables to obtain all object identifiers with the specific value. After that, the object attributes are collected if needed. If multiple filter conditions are given, several sets of object identifiers are calculated and the intersection of these sets will be the result.

Filtering a class' objects with attribute values is done by joining the attribute instance table and the object table on the object identifier with the specific class and attribute conditions. This way, the object identifiers are obtained, so attributes can be collected if needed. If several attribute conditions are given, then a set of objects is calculated via multiple joins and the result will be the intersection of these sets. This can be done with several quite complex queries.

Queries can be simplified using three enhancements of this solution.

Enhancements First, the attribute instance tables can be contracted into one single *attributeInstances* table that has a separate column for each attribute type as seen in Fig. 3. This tables stores columns for each attribute type, and in each record only the used column is filled, other fields contain *null* values. To store the attribute description (name, type, measurement unit, etc.), we use an *attributeSchema* table. With this improvement the required space for the database grows, but the queries become simpler and faster, due to faster joins.

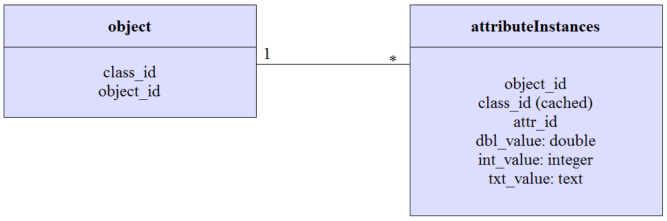

Fig. 3. A single attributeInstances tables with the object table

Second, the filters that contain both class and attribute conditions can be simplified, if we store the class information in this table as well. With this caching, these conditions result can be calculated by using only the attribute instance table. This technique creates redundancy in the database, so it's very important to create the new *class_id* field properly. The third possibility is to store the attribute description information also inside the *attributeInstances* table to spare another join operation. This operation also results in redundant data.

3.2 Document-oriented database

As opposed to relational databases, document-oriented databases do not have a schema that describes the structure of each record. Instead, records are stored as documents, which can have any number of fields, and each field can be of any type and can contain even multiple pieces of data. Documents are stored in collections, without any structural information, so each document can have a different structure inside a collection. Information can be added or removed any time without the need

of schema alternation; therefore it can provide an ideal solution for frequent structural changes. Fields can be of any complexity, it is even allowed to store complete documents inside the fields. Also, the documents' fields are named and typed, so they can be exactly parsed to objects of an object-oriented application.

In our project we have chosen *MongoDB* as the document-oriented database engine. *MongoDB* is known to be one of the fastest schema-free document-oriented solutions (see [5]), therefore it serves as a perfect candidate for our performance measure. It is based on *JSON*, allowing the storage of semi-structured data, and the nesting of data into complex structures that still can be queried and indexed. It also provides the full functionality of relational databases.

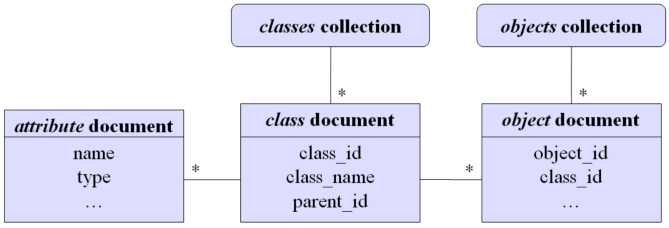

Fig. 4. Document-oriented storage of classes and objects

In our model it is easy to match objects against documents simply by transforming every attribute of an object to a field of the document. Due to the lack of structuring, objects can be stored in a single collection (named objects). However, since class data can only be obtained from each document separately, it is vital for performance to store also the class description data. We use a separate collection (classes) for that purpose. Due to the complex structuring abilities, attribute properties (like name, type, default value) can be nested inside the class information document. Inheritance is implemented using reference identifiers and recursive queries. Thus, we have a structure seen in Fig. 4.

4 Performance measure

To measure the performance of each approach, we have developed a testing environment to implement all three solutions using the *Microsoft .NET Framework* and the C# programming language. We have chosen C# because it's object-oriented and strongly typed, and most importantly it's the backbone of ASP.NET Web applications. Also, the EDIT project is developed in C# as well. We have chosen *MySQL*, as our relational database engine.

The test environment implements the abstract object and class concepts, and provides three classes for the different database types. Each operation's time is measured, including the transformation of the object or class from and to the solution, but not including any other environmental delays. All operations have been implemented including creation, insertion, altering and querying of objects and classes. Operations can be preformed multiple times, and with different weights for each operation time. Data can be imported from any database structured according to one of our implementations and generated by providing detailed class information, or simply just generating random names and values. Also the database size is constantly monitored. Time and size values can be output to charts or spreadsheets.

It must be noted, that our testing method relies on the performance of the .NET Framework, so the results can vary between implementations, but the ratio of the values should not change too much, since all used application programming interfaces (the *MySQL Connector* and the *MongoDB Driver*) use the same network connection protocols, data structures and therefore the same .NET facilities for retrieving and modifying information in the database, so the comparison should be accurate.

4.1 Performance results

Results have been gained by performing all operations several thousand times and summing runtimes. For simplification, the solutions are marked by numbers. In the first solution, tables are generated on-the fly for each class as described in Section 3.1.2. The second solution stores objects and attributes in separate tables as described in Section 3.1.3. We have implemented all combinations of enhancements. Measurements show that using a single attribute instance table enormously raises performance with only 5 to 20 percent in database size growth, and also the caching of identifiers and descriptions can have a speed advance of 20 to 40 percent. The third solution uses the document-oriented implementation described in Section 3.2.

This kind of empirical testing does not make use of any theoretical background. However our previous work with this approach (see [6]) shows promising practical results.

- **Class creation and removal**
  Without considering inheritance, class creation is the first operation to be executed, and also sometimes needed during operation. Class creation time is primary determined by the attribute count in case of the second implementation, but it does not significantly affect the runtimes for the other two solutions (as seen on Fig. 5). Class removal is quite fast in all cases, but usually determined by the number of objects also needed to be removed is case the relational database solutions. The removal is only slightly affected in the third solution, in case of large objects. It has been observed that the document-based implementation is about ten times faster in creation and removal than the first implementation. The second implementation takes at least two times as much time as the first, and the difference is even greater with a large number of attributes.
  When using inheritance the third solution is still not affected, the other solutions are affected as much as without inheritance, but with the same amount of attributes. Removal of objects is linearly affected in the first solution by the amount of child classes.

- **Object creation and removal**

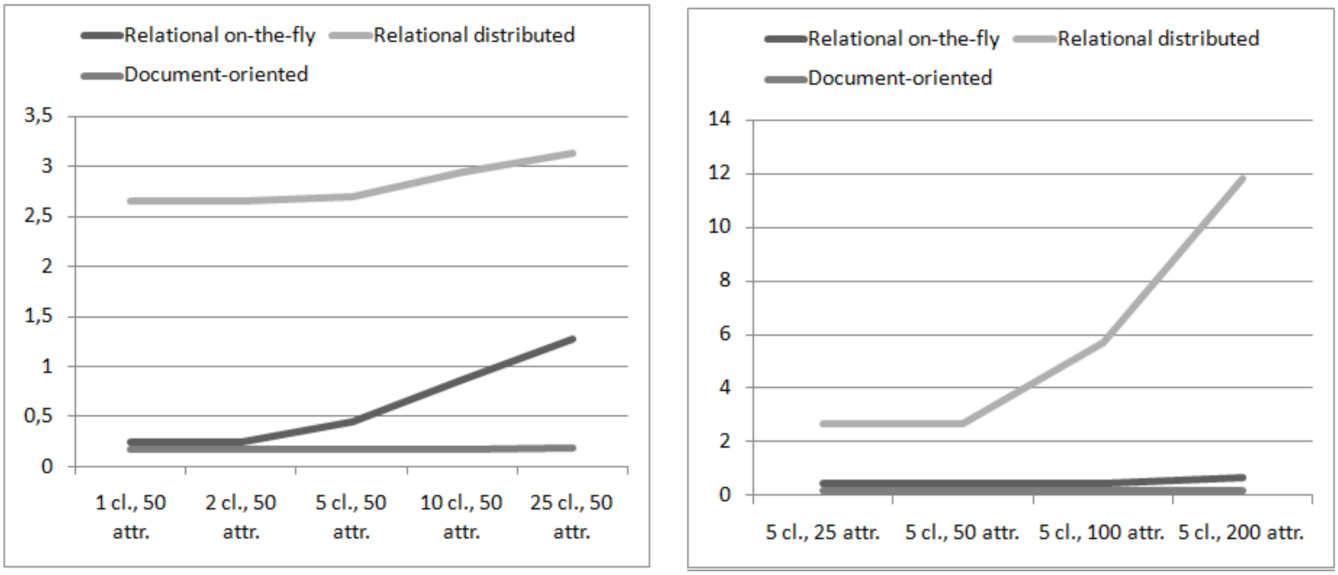

Fig. 5. Class creation time with fixed number of total attributes (50) and classes (5)

The number of attributes clearly affects object creation time, but with different amount. Our tests show that the first and third solutions only slightly drop in performance (logarithmically) by raising the attribute count of the classes, while the second solution is linearly affected (with a constant multiplier of 1/2, see Fig. 6).

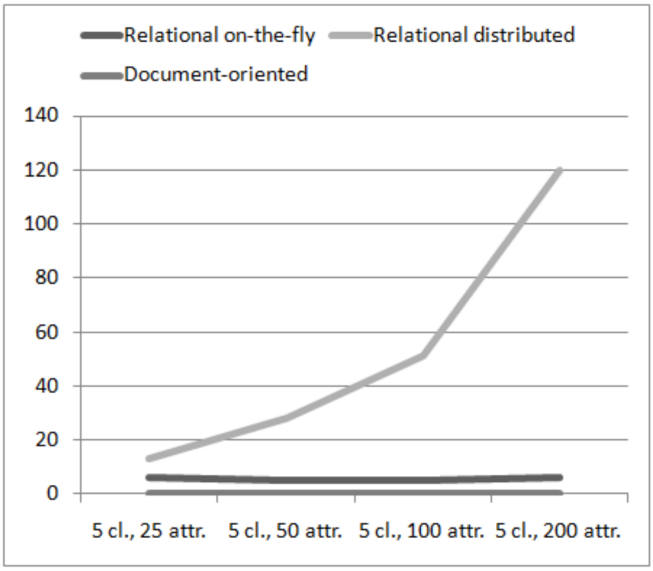

Fig. 6. Object creation time witch variable number of total attributes

Raising the number of classes linearly increases time taken by the second implementation (with the same total number of attributes), but does not affect the other two implementations, therefore the second solution generally requires at least three times as much for insertion and removal compared to the first solution (affected by the number of classes), while the third is again clearly much more efficient, producing the fraction of time used by the other two implementations.

We must note that the lack of speed of the second solution is mostly due to the number of insert and delete commands that need to executed by the test environment. This can be improved by creating strored procedures on the database server to execute these multiple operations, todignificantly shorten the communition time of the application.

- **Class queries**
  Quickly querying an entire class is the main promise of the first solution, as it only needs to fetch an entire table. This results between 1.2 and 4 times the speed of the third solution. The difference lies in the number of objects stored in the database. The second solution's query times exponentially grow with the number of objects, the total number of attributes, and the number of classes as well (as seen on Fig. 7).

- **Attribute queries**
  Somewhat unexpectedly of the second solution is an order of magnitude better than the first with attribute queries. In case of few (1-2) filter conditions, it is even double as fast as the document-oriented implementation. However, the number of attributes influences it in linear time, while the third solution is not affected by this number. Also, the first solution is pretty much resistant to the number of objects, but can be influenced by the number of total attributes. The advanatage of the third solution is even better when raising the number of filter conditions (due to less documents to be returned). This is shown in Fig. 8.
  When filtering for attributes of a certain class, the first solution proves to the best again, but the much less advantage, as with simpla class queries.

- **Class altering**
  As expected, adding or removing attributes from a class is quite slow with the first implementation with foremost the attribute count of the class influencing its speed. This can be seen in Fig. 9. Both the second and third implementation are only slightly affected by the number of (descendant) classes,

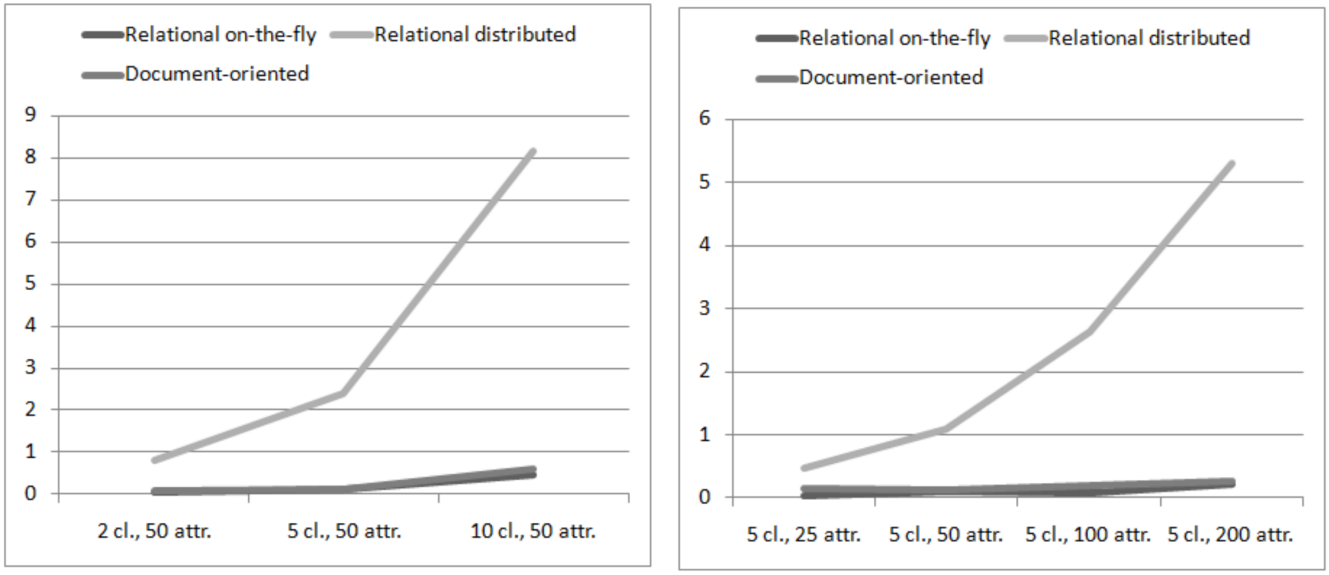

Fig. 7. Class query time with fixed number of total attributes (50) and classes (5)

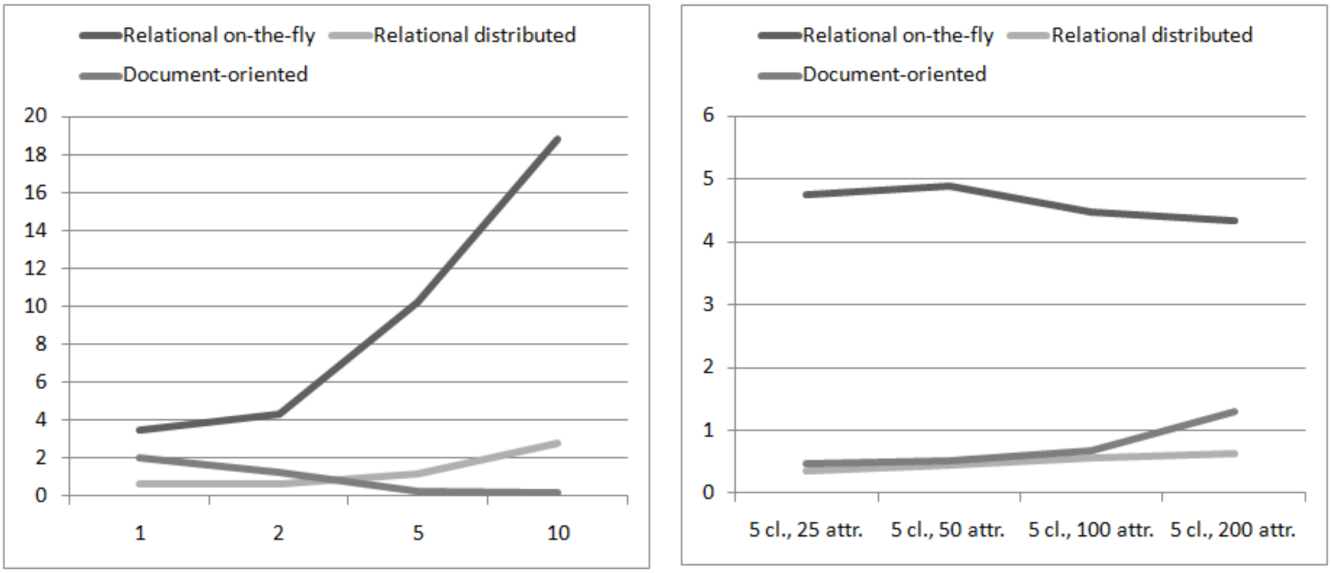

Fig. 8. Attribute query time with variable (1 to 10) and fixed filter count (2)

and the number of objects.

- **Balanced usage**
  It is not easy to determine the balanced operational load of a system using this architecture, because it can vary by usage. Our monitoring of the EDIT system has revealed the following aspects. About 90 to 95 percent of the operations are queries, equally distributed between class and attribute filters. Significant part of the remaining operations are object related (creation and removal), and a few are class related operations. In case of the Amnis project, where many documents are created during runtime, query times have less importance, but still take about 50% of the operations.
  When calculating with these times, we can see that the third solution outperformes both relational implementations, and the first implementation is somewhat better then the second in both cases.

- **Database size** It's easy to calculate that in terms of the relational implementations the first solution needs less storage space due to the redundant data storage of the second solution. Using all enhancements this difference can grow very fast, and the database size can become the multiple amount of the first solution. The MongoDB implementation is also sensitive to the amount and size of stored objects. With small object and attribute count (e.g. 1000 total attribute values) the size of the database can be less than that of first soltuion, but this advantage can quickly disappear as the object count goes up (with about 200 000 stored values it takes four times as much space as the first relational solution), see Fig. 10.

5 Conclusion and future work

In the previous sections we have presented three solutions to a database structure that implements class inheritance tax-

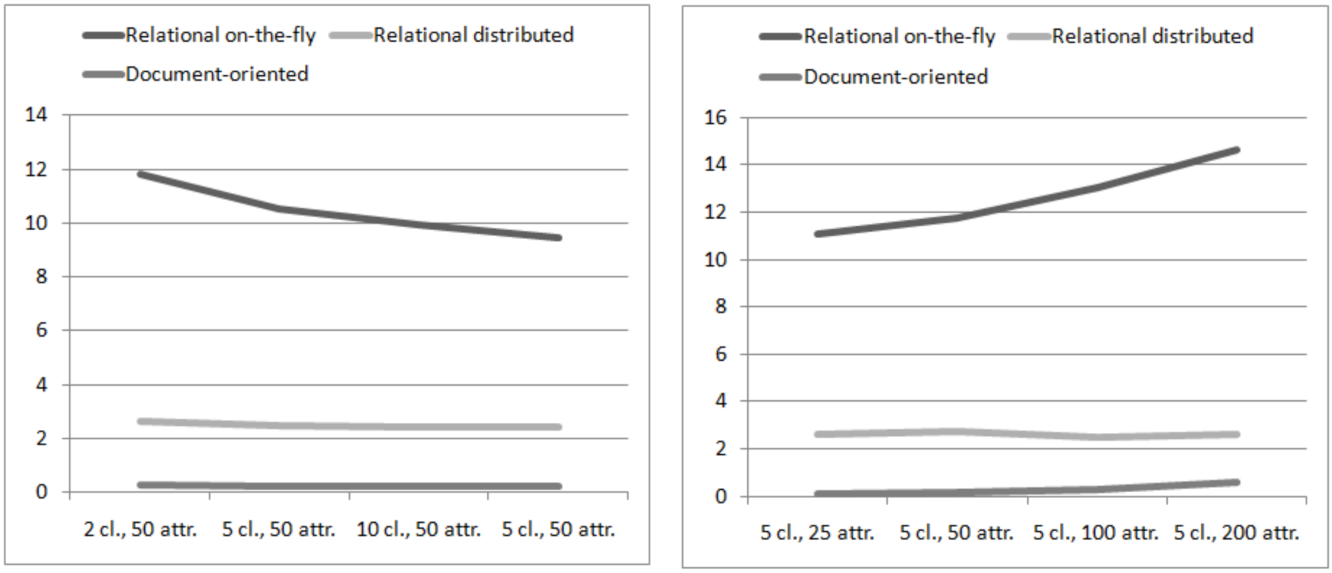

Fig. 9. Class altering time with fixed number of total attributes (50) and classes (5)

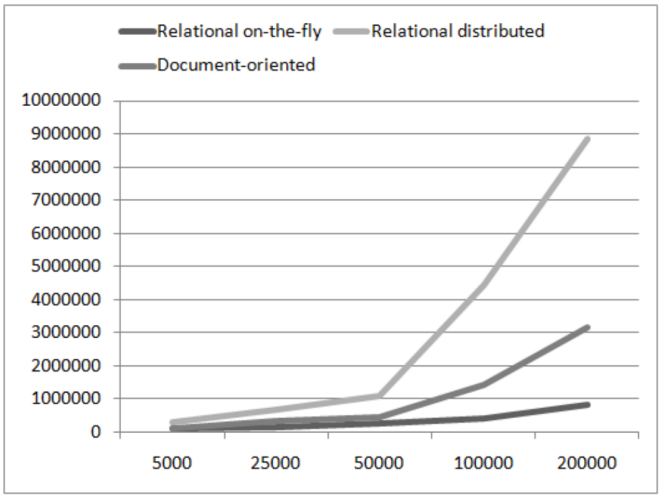

Fig. 10. Database size (in bytes) with respect to total attribute value count

onomy. We have developed a testing environment in .NET to study the performance of these solutions using the MySQL and MongoDB database engines. Our intention was not to generally give an opinion on which solution is better on any software and hardware platforms, but to gain results which we can work with in our projects, and to have an idea how our solutions perform against each other.

As expected we have not gained clear results in all fields, but in terms of general usage, the document-oriented solution seems to outperform relational solutions. This may be due to the rather natural compliance with our object-oriented model. In terms of the relational implementation, using on-the-fly generated tables provides faster class queries, creation and removal time and less disk space, while the distributed object model provides fast attribute based filtering, class alternations. Still, in overall performance one may favour the first solution, but in some situations the advantage of the second implemants can also come handy. Ultimately we can only say that much relies on the nature of the project being worked on.

In the future we will work further on developing and perfecting solutions for inheritance based database structuring to serve us in our next projects.

References

1 **Giachetta R., Elek. I.**, *Developing an Advanced Document Based Map Server*, 8th International Conference on Applied Informatics (ICAI) (2010).

2 **Ambler S. W.**, *Process Patterns - Building Large-Scale Systems Using Object Technology*, 8th International Conference on Applied Informatics (ICAI) (2010).

3 **Máriás Zs.**, *Design and Performance Analysis of Hierarchical Large-scale Inhomogeneous Databases*, 8th International Conference on Applied Informatics (ICAI) (2010).

4 **Nadkarni P. M.**, *Organization of Heterogeneous Scientific Data Using the EAV/CR Representation*, The Journal of the American Medical Informatics Association (JAMIA) **6** (1999), no. 6.

5 **Chodorow C.**, *Introduction to MongoDB*, Free and Open Source Software Developers' European Meeting (FOSDEM) (2010).

6 **Fekete I., Giachetta R., Kovács P.**, *To Balance or to Rebuild? - An Experimental Study of Randomly Built Binary Search Trees*, 8th International Conference on Applied Informatics (ICAI) (2010).