Annales Univ. Sci. Budapest., Sect. Comp. 37 (2012) 275–292

A STUDY OF STORING LARGE AMOUNT OF INHOMOGENEOUS DATA IN WORKFLOW MANAGEMENT SYSTEMS

Zsigmond Máriás, Tibor Nikovits,
Tamás Takács and Roberto Giachetta
(Budapest, Hungary)

Communicated by András Benczúr

(Received January 15, 2012; revised February 8, 2012;
accepted February 24, 2012)

Abstract. In workflow-driven Enterprise Resource Planning (ERP) systems a large variety of documents requires handling and storing of various descriptive data in a single storage facility. The number and type of attributes can vary among different kinds of documents. Storing such inhomogeneous data in a single database is difficult, as querying requires fast retrieval of data based on any present attribute.
In this paper the authors introduce three different approaches to this problem based on relational and document-oriented database systems. All the three solutions are compared by implementing and testing them with massive inhomogeneous data and by using a sophisticated cost model.
This problem is typical in ERP systems. Moreover, the solution can be generally applied to any domain using inhomogeneous data, like e-commerce systems, document warehouses and GIS systems.

Key words and phrases: Workflow management systems, object-oriented databases, inhomogeneous data, cost analysis.
2010 Mathematics Subject Classification: 68P20, 68U35.
1998 CR Categories and Descriptors: H.2.4, H.2.5, H.3.3.
The research is supported by KMOP-1.1.2-08/1-2008-0002 and European Regional Development Fund (ERDF).

1. Introduction

Business processes and data representation is an essential issue in the design of ERP systems [1]. Therefore, workflow based ERP systems have been an active field of research in the last few years. ELTE Soft Ltd. has been developing a system called AMNIS for two years, where the workflow model for a business process is defined by the documents created during the process. In the development process the most important issue was to create the application layer that makes it possible to perform queries based on any descriptive data of the documents.

Hence, the aim is to provide a database storage and retrieval system that has the highest performance to execute database filtering operations on a large amount of objects, including the possibility to modify the data and structural information. For this reason, several implementations have been concerned and tested, and the most promising three have undergone an extensive performance measurement procedure. Their results are presented in this paper. Furthermore, we will give a short literature review on the new approaches for document handling in databases, focusing especially on XML documents.

The rest of the paper is arranged as follows. In Section 2, the problem and the abstract solutions are introduced. In Section 3, the implementations are presented, with the performance measurement procedure and results in Sections 4. Section 5 concludes the paper.

2. The database structure

In the AMNIS system the documents belong to different document types. Every type category has different set of descriptive data called document attributes. For example, vehicle tracking workflows may use trucks and destinations as documents. Truck documents contain information about truck driver, current fuel level, average fuel consumption and license plate number, while destinations contain an address and GPS coordinates.

Each document type category may have several subcategories, for example cars, vans and trucks, and among trucks there are also different subcategories for light, medium and heavy trucks 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.

Beyond the maintenance of this category taxonomy, document instances have to be stored and retrieved for all categories, with all the descriptive data of a certain category. Since the database stores a huge amount of documents, functionality is needed to retrieve not just single objects but a set of documents based on different filter conditions. These filter queries contain conditions based on the descriptive data, such as retrieving documents with a specific attribute value, or class conditions, such as retrieving all the objects in a specific category and its subcategories.

This kind of hierarchy of classes and objects is very useful in different kinds of applications. If e-commerce systems are considered, classes are product categories, objects are products, and attributes are product features. Various data can also be found in geospatial data storage systems, for example the currently developed EDIT mapping system [2]. Searching facilities are very important in both cases.

The goal is to design a database structure in which class inheritance taxonomy can be defined and objects can be stored that belong to the defined classes. Name and type information have to be stored for every attribute in the 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, they are not considered further on.

Every class holds a number of attributes, which can be modified any time, so the ability is needed 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.

A large amount of objects have to be stored for each class and different kinds of filter queries have to be performed. The ability to reference and retrieve a single object or set of objects – based on different conditions – is needed. Therefore, the following operations are introduced for queries and filters:

- *getAttributeOfObject (attrId, objId)*: Retrieves one attribute value of a single object, based on the object identifier and attribute identifier.
- *getAllAttributesOfObject (objId)*: Retrieves every attribute of a single object based on the object identifier.
- *getObjectsOfClass (classId, descendants = true/false)*: Collects all objects for a class based on the class identifier with or without the objects of descendant classes.
- *getObjectsByAttributeValue (classId, attrId ⇒ value, operation, descendants = true/false)*: Collects all objects of a class with a specific attribute value or one attribute constraint with or without the objects of descendant classes.

- *getObjectsByAttributeValues (classId, filters[], descendants = true/false)* where $filters = array(attr\_id_1 \Rightarrow value, attr\_id_2 \Rightarrow value, ...)$: Collects all objects of a class with specific attribute values or several attribute constraints, with or without the objects of descendant classes.

Due to the large number of objects and operations, performance properties are crucial in finding an adequate solution [3]. In the AMNIS system, the filters and object queries are used much more often than class and object operations. Therefore, the solution must provide fast queries with these operations.

3. Implemented solutions

In previous research, several kinds of relational database structures have been studied [4], from which the two highest performing have been chosen. 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. They are compared with a third, document-oriented database solution, which is a rather natural implementation of the structure.

In the relational solutions, the class hierarchy is stored in three tables:

- *attributeSchema* table defines the schema information of each attribute in the system. This table stores the attribute id, the type of a certain attribute and its name.
- *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. However, when retrieving the attributes of a class, the attributes of a given class and also its ancestors have to be collected, which require multiple queries. As this has to be performed frequently – even when retrieving a single object from the database –, some improvements should be done by denormalization.

This improvement is done by adding a new field into the *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.

3.1. 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. The first solution is similar to this method, but the frequent change of classes and attribute schema needs to be considered. Objects are stored as records, but a separate table is generated automatically for each class when new classes are added.

The name of the tables are *objectsOfClass_{class_id}*, and these tables contain an *object_id* and the attribute fields, as shown in Figure 1. 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).*

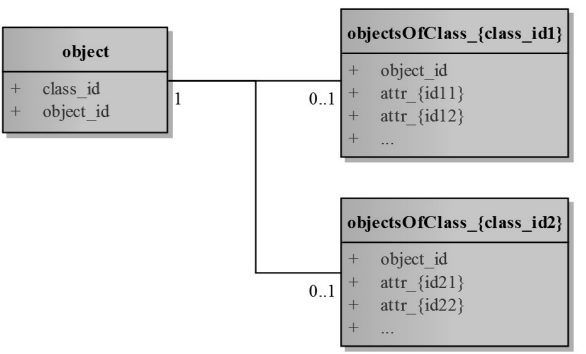

Figure 1. Objects in tables created on-the-fly

When a class is altered by adding or deleting attributes, not only the class hierarchy needs to be changed. The tables of the class and its descendants are also affected, which needs to be considered for the altering functions. When a class is removed, 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 a single object has to be retrieved from the database, a simple select query has to be performed in the proper data table. To achieve that, an additional lookup table called objects is maintained which stores pairs of object and class identifiers.

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 the object of a class 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.

Algorithm 1 getAttributeOfObject($attrId$, $objId$):
$classId \leftarrow getClassByObject(objId)$
$R \leftarrow Query(\text{SELECT } attr\_\{attrId\}$
FROM $objectsOfClass\_\{classId\}$
WHERE $object\_id = objId)$
return R

Algorithm 2 getAllAttributesOfObject($objId$):
$classId \leftarrow getClassByObject(objId)$
$R \leftarrow Query(\text{SELECT} * \text{FROM } objectsOfClass\_\{classId\}$
WHERE $object\_id = objId)$
return R

Algorithm 3a getObjectsOfClass($classId$, $descendants = \text{false}$):
$R \leftarrow Query(\text{SELECT} * \text{FROM } objectsOfClass\_\{classId\})$
return R

Algorithm 3b getObjectsOfClass($classId$, $descendants = \text{true}$):
$Push(R, Query(\text{SELECT} * \text{FROM } objectsOfClass\_\{classId\})$
$descList \leftarrow descOfClass(classId)$
for all $descId$ **in** $descList$ **do**
$Push(R, Query(\text{SELECT} * \text{FROM } objectsOfClass\_\{descId\})$
end for
return R

Algorithm 4 getObjectsByAttributeValue($classId$, $attrId \Rightarrow value$, $descendants = \text{false}$):
$R \leftarrow Query(\text{SELECT} * \text{FROM } objectsOfClass\_\{classId\}$
WHERE $attr\_\{attrId\} = value)$
return R

Algorithm 5 getObjectsByAttributeValues($classId$, $filters[]$, $descendants$ = false):

$constraintList \leftarrow buildConstraintList(filters[])$
$R \leftarrow Query(\text{SELECT} * \text{FROM } objectsOfClass\_\{classId\}$
WHERE $constraintList$)
return R

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 attribute schema of a class and the filter conditions. The expectation was that this solution would work well in searching and filtering, which is the most expensive part of usage in most applications.

3.2. Storing objects and attribute instances in separate tables

In the 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 {*basetype*}*AttributeInstances* in which attribute instances are stored. For example if integer, text and double attributes are allowed 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, as illustrated in Figure 2.
- The *objects* table stores the object and class identifiers as in the previous solution.

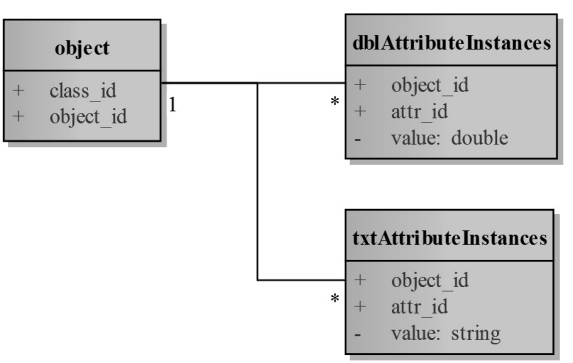

Figure 2. Multiple attributeInstances tables with the object table

The following modifications of this structure enhance the overall performance.

- The attribute instances tables can be contracted into one single attributeInstances table that has a separate column for each attribute type as can be seen in Figure 3. This table stores columns for each attribute type. Only the used column is filled in each record, other fields contain null values.
- The filters that contain both class and attribute conditions can be simplified, if the class information is stored in this table as well. With this caching, these conditions can be calculated by using only the attribute instance table. This technique creates redundancy in the database, so it is very important to maintain the new *class_id* field properly.

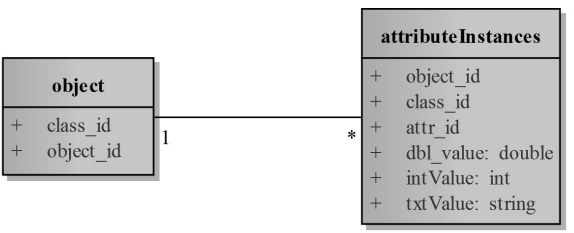

Figure 3. A single attributeInstances table with the object table

In this implementation, no new tables are created, so adding a class needs no further operations. When inserting an object, the attribute values are placed into the attribute instance table in multiple records. Objects can be retrieved by first determining their attribute schema, then by queries in the attribute instance table 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). If the attributes of objects are needed as well, they have to be collected from the attribute instance table as discussed above. The table structure can be seen in Figure 3.

Filtering the objects by a specific attribute value condition can be done in two steps. First, the value instances are collected from the attribute instance table to obtain all object identifiers with the specific value. 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 is the result.

Filtering the object of a class 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 multiple attribute conditions are given, the 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.

Algorithm 6 getAttributeOfObject($attrId$, $objId$):
$R \leftarrow Query($SELECT $*$ FROM $attributeInstances$
WHERE $object\_id = objId$ AND $attr\_id = attrId)$
return R

Algorithm 7 getAllAttributesOfObject($objId$):
$R \leftarrow Query($SELECT $*$ FROM $attributeInstances$
WHERE $object\_id = objId)$
return R

Algorithm 8 getObjectsOfClass($classId$, $descendants =$ false):
$R \leftarrow Query($SELECT $*$ FROM $objects$
WHERE $class\_id = classId)$
for all obj **in** R **do**
$Push(Q, Query($SELECT $*$ FROM $attributeInstance$
WHERE $object\_id = obj.objId))$
end for
return Q

Algorithm 9 getObjectsByAttributeValue($classId$,
$filterAttrId \Rightarrow filterAttrValue$, $descendants =$ false):
$R \leftarrow Query($SELECT $*$ FROM $attributeInstances$
WHERE $class\_id = classId$
AND $attr\_id = filterAttrId$
AND $attr\_value = filterAttrValue)$
for all $attrInstance$ **in** R **do**
$Push(Q, Query($SELECT $*$ FROM $objects$
WHERE $object\_id = attrInstance.object\_id)$
end for
return Q

Algorithm 10 getObjectsByAttributeValues($classId$, $filters[]$, $descendants$ = false):

```
filter_1 ← Pop(filters)
R ← Query(SELECT object_id FROM attributeInstances
            WHERE class_id = classId
            AND attr_id = filter_1.attr_id
            AND attr_value = filter_1.value)
for all filterArrId ⇒ filterAttrValue in filter do
  R_1 ← newList
  for all objectId in R do
    Q ← Query(SELECT object_id FROM attributeInstances
                WHERE object_id = objectId
                AND attr_id = filterAttrId
                AND attr_value = filterAttrValue)
    if numRows(Q) > 0 then
      Push(R_1, objectId)
    end if
  end for
  R ← R_1
end for
return R
```

3.3. Document-oriented database

In the third solution a document oriented database is used instead of a relational database system. An open source system, called MongoDB is chosen for this purpose [5]. As MongoDB is a schema-free database system, only two collections are needed, one for the classes, and one for the objects. In these collections, objects and classes are represented by documents that can be produced by transforming the data into JSON style arrays.

A class document consists of two parts. The header part contains the MongoDB identifier that is used as the class identifier, the name and the MongoDB identifier of the parent class. The body is a subarray that contains the list of attribute schemata for the class as a $fieldname \Rightarrow type$ pair.

The documents that represent objects are quite similar to class documents. An object document consists of two parts. The header part contains the object identifier and the identifier of the object's class. The body is an embedded subarray that contains the list of attribute values as a $fieldname \Rightarrow value$ pair.

Class:

```
array(
    _id ⇒ mongoDBObjectId
    classname ⇒ "classname",
    parent ⇒ mongoDBObjectId
    attributes ⇒ array(
        field1name ⇒ type,
        field2name ⇒ type,
        ...
    )
);
```

Objects of Class:

```
array(
    _id ⇒ mongoDBObjectId
    class ⇒ mongoDBobjectId,
    attribute_values ⇒ array(
        field1name ⇒ value,
        field2name ⇒ value,
        ...
    )
);
```

One advantage of this solution is that most of the filter queries can be simply transformed into MongoDB queries.

Algorithm 11 getAttributeOfObject($attrId$, $objId$):
return db.$objectofclass$.find($\{\_id : objId\}$, $\{attribute\_values.field\{attrID\}name\}$)

Algorithm 12 getAllAttributesOfObject($objId$):
return db.$objectofclass$.find($\{\_id : objId\}$)

Algorithm 13a getObjectsOfClass($classId$, $descendants = \text{false}$):
return db.$objectofclass$.find($\{class\_id : classId\}$)

Algorithm 13b getObjectsOfClass($classId$, $descendants = \text{true}$):
$Push(R, \text{db.}objectofclass\text{.find}(\{class\_id : classId\}))$
$descList := descOfClass(classId)$
for all $descId$ **in** $descList$ **do**
  $Push(R, \text{db.}objectofclass\text{.find}(\{class\_id : descId\}))$
end for
return R

Algorithm 14 getObjectsByAttributeValue ($classId$, $filterAttrId \Rightarrow filterAttrValue$, $descendants = \text{false}$):
return db.$objectofclass$.find(
$\{attribute\_values.field\{filterAttrId\}name :$
$filterAttrValue\}$)

Algorithm 15 getObjectsByAttributeValues ($classId$, $filters[]$, $descendants = \text{false}$):
$constraintList \leftarrow buildConstraintList(filters[], classId)$
return db.$objectofclass$.find($\{constraintList\}$)

4. Performance measurement

In this section we compare the efficiency of queries on the presented three storage models. We use the two most common performance measurement methods: the cost model and the Benchmark test. In the first one we make some simplifications and concentrate only on queries omitting all kinds of modifications.

To predict the performance of our solutions, a cost model is introduced that gives a detailed picture on the advantages of each solution. Based on this model, several performance measurement cases are presented.

4.1. Cost model

In a database environment the dominant part of the cost of a query is usually the cost of the I/O operations expressed in the number of data blocks read or written. Data block is the smallest amount of data a DBMS can read or write. In our model the concept of data block is not applicable, that's why the number of I/O operations is used instead. The data element length is assumed to be not significantly different, therefore, the content type of the field is not considered and there is only one read operation. In the following, the I/O costs of the most frequent operations are presented.

If the data is indexed, the read operation is performed after the search in the index. Normally the cost of the index search depends on the structure and size of the index. Our cost model assumes that each index search costs the same and only the number of index searches is counted.

Abbreviations:

nc: number of classes

na: average number of attributes per class

no: average number of objects per class

nf: number of filters

nmo: number of matching objects

is: index search operation

re: read operation

The cost calculations for the following six test cases are presented:

- Retrieving an attribute by object id. Table 1.
- Retrieving a single object with all atributes by object id. Table 2.
- Retrieving all objects of a class without descendants. Table 3.
- Retrieving all objects of a class and descendant classes. First, all descendant class ids need to be retrieved, which is independent of the storage model, then the objects of every class are collected. See Table 3.

- Retrieving all the objects of a class having a special value in a given attribute. If there are no indices on the attributes, then the cost is the same as in Table 3. If indexing values differ, see Table 4. If there are indices on the attributes, then the cost is the same as in Table 3. If we have more than one filter conditions, then the number of matching objects is much smaller. See Table 5.

getAttributeOfObject(attrId, objId)		Cost
Algorithm 1	Search is performed for the *class_id* in the object table, then for the *object_id* in the *objectsOfClass_classid* table.	$(no \cdot nc + no) \cdot re$
Algorithm 1 with indexing	With indices on both tables, the costs are two index searches and two read operations.	$2 \cdot is + 2 \cdot re$
Algorithm 6	A single row has to be searched in the *attributeInstances* table.	$no \cdot nc \cdot na \cdot re$
Algorithm 6 with indexing	With an index on the *attributeInstances* table.	$is + re$
Algorithm 11	A single record has to be searched in the *objectsOfClasses* collection.	$no \cdot nc \cdot na \cdot re$
Algorithm 11 with indexing	With an index on the collection elements.	$is + re$

Table 1. Retrieving an attribute by object id, attribute id

getAllAttributesOfObject(objId)		Cost
Algorithm 2	There is no difference in this storage model, retrieving 1 attribute or all attributes of an object.	$(no \cdot nc + no) \cdot re$
Algorithm 2 with indexing	Same as in Algorithm 1 with indexing.	$2 \cdot is + 2 \cdot re$
Algorithm 7	Same as in Algorithm 6.	$no \cdot nc \cdot no \cdot re$
Algorithm 7 with indexing	A separate search is performed for every attribute.	$na \cdot is + na \cdot re$
Algorithm 12	The whole record has to be found in the *objectsOfClasses* collection.	$no \cdot nc \cdot no \cdot re$
Algorithm 12 with indexing	Same as in Algorithm 11 with indexing.	$is + re$

Table 2. Retrieving a single object by object id

getObjectsOfClass(classId, descendants = false)		Cost
Algorithm 3a	In this case, there is no meaning of using indexes, because all rows of the class table have to be retrieved.	$no \cdot re$
Algorithm 3a with indexing	Same as in Algorithm 3a.	$no \cdot re$
Algorithm 8	Same as in Algorithm 6.	$no \cdot nc \cdot no \cdot re$
Algorithm 8 with indexing	A separate search is needed for every attribute of every object.	$no \cdot na \cdot (is + re)$
Algorithm 13a	Same as in Algorithm 11.	$no \cdot nc \cdot no \cdot re$
Algorithm 13a with indexing	A separate search is needed for every object.	$no \cdot is + no \cdot re$

Table 3. Retrieving all objects of a class (without descendants)

getObjectsByAttributeValue(classId, attrId $\Rightarrow$ *value, descendants = false)*		Cost
Algorithm 4	Same as Algorithm 3a.	$no \cdot re$
Algorithm 4 with indexing	The cost strongly depends on the number of matching objects.	$is + nmo \cdot re$
Algorithm 9	Same as Algorithm 8.	$no \cdot nc \cdot no \cdot re$
Algorithm 9 with indexing	The index size is much larger ($nc \cdot na$ times larger) than in the first model, so the index search is slower.	$is + nmo \cdot re$
Algorithm 14	Same as Algorithm 13a.	$no \cdot nc \cdot no \cdot re$
Algorithm 14 with indexing	With an index on the collection element values.	$is + nmo \cdot re$

Table 4. Retrieving all objects of a class by attribute value

4.2. Experimental performance measurement

To measure the performance of each approach, a testing environment has been developed to implement all three solutions using the Microsoft .NET Framework and the C# programming language. MySQL has been chosen as our relational database engine.

The test environment realises the abstract object and class concepts, and provides three classes for the different database types. Each filter 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. Operations can be performed multiple times; data can be imported from any of the implemented database structure into the other. Also the database size is constantly monitored.

It must be noted that the testing method relies on the performance of the .NET Framework. The results can vary between implementations, but the ratio

getObjectsByAttributeValues(classId, filters[], descendants = true/false)		Cost
Algorithm 5	Same as Algorithm 3a.	$no \cdot re$
Algorithm 5 with indexing	If the number of filters increases, the number of matching objects decreases.	$nf \cdot is + nmo \cdot re$
Algorithm 10	Same as Algorithm 8.	$no \cdot nc \cdot no \cdot re$
Algorithm 10 with indexing	The index size is much larger ($nc \cdot na$ times larger) than in the first model, so the index search is slower.	$nf \cdot is + nmo \cdot re$
Algorithm 15	Same as Algorithm 13a.	$no \cdot nc \cdot no \cdot re$
Algorithm 15 with indexing	If we have more filters we get less matching elements.	$nf \cdot is + nmo \cdot re$

Table 5. Retrieving all the objects of a class having special values in given attributes

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. Therefore, the comparison should be accurate. Results for performance measurement have been gained by performing all operations several thousand times and summing runtimes. The test configuration is an Intel Core i3 2.33GHz CPU with 4GB of RAM and a 5400 rpm SATAII hard disk.

4.2.1. 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 linearly grow with the number of objects, the total number of attributes, and the number of classes as well (as seen in Figure 4).

4.2.2. Attribute queries

Somewhat unexpectedly, the second solution is an order of magnitude better than the first one with attribute queries. In the case of few (1-2) filter conditions, it is twice 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.

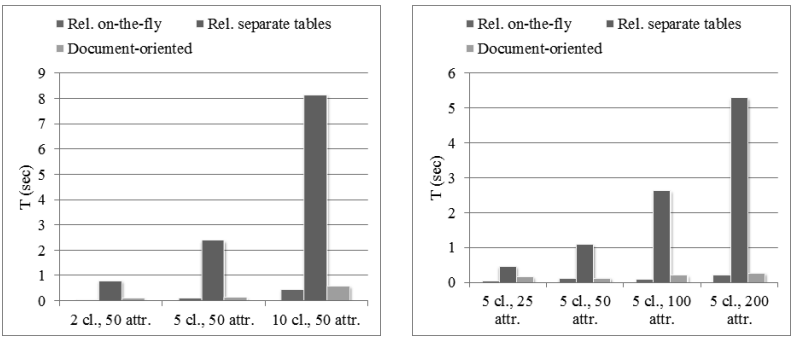

Figure 4. Class query time with fixed number of total attributes (50) and classes (5)

The advantage of the third solution is even better when raising the number of filter conditions (due to fewer documents to be returned). This is shown in Figure 5.

When filtering for attributes of a certain class, the first solution proves to be the best again, but with less advantage in the case of simple class queries.

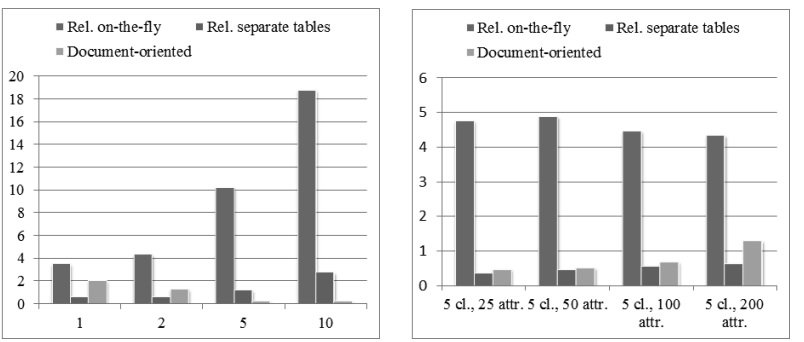

Figure 5. Attribute query time with variable (1 to 10) and fixed filter count (2)

5. Conclusion and future work

In the previous sections three solutions were presented for a database structure that implements class inheritance taxonomy. A testing environment has been developed to study the performance of these solutions using the MySQL

and MongoDB database engines. The 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 research projects, and to have an idea of how the solutions perform against each other.

As expected, none of the solutions has been proven to be the absolute winner, but in terms of the general usage in AMNIS, 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 imple- mentation, 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 favor the first solution, but in some situations the advantage of the second implementation can also come in handy. Ultimately, it can only be said that much relies on the nature of the project being worked on.

Besides the discussed techniques, most of the modern database engines provide document handling and querying capabilities through XML documents [6] and XQuery [7] query language. Hence, another implementation option is to use this model to represent the classes, objects and attributes. Similar EAV implementation techniques are often used in bioinformatics [8] and geoinformatics [9].

In the future we will work further on the development of these solutions for inheritance based database structures. This effort refers also to one of our running projects, the open-source geoinformational system AEGIS.

References

[1] **Cardoso, J, R.P. Bostrom and A. Sheth,** Workflow management systems and ERP systems: Differences, commonalities and application, in: Laudon, K. C., Turner, J. (eds.): *Information Technology and Management*, **5(3-4)**, 2004, 319–338.

[2] **Giachetta, R. and I. Elek,** Developing an advanced document based map server, in: A. Egri-Nagy, E. Kovács, G. Kovásznai, G. Kusper and T. Tómács (eds.), *Proceedings of the 8th International Conference on Applied Informatics*, Eger, Hungary, 2010.

[3] **Ambler, S.W.,** *Process Patterns - Building Large-Scale Systems Using Object Technology*, Cambridge University Press, 1998.

[4] **Máriás, Zs.,** Design and performance analysis of hierarchical large-scale inhomogeneous databases. Lecture at: *8th International Conference on Applied Informatics*, Eger, Hungary, 2010.

[5] **Padhy, R.P., M.R. Patra and S.C. Satapathy,** RDBMS to NoSQL: Reviewing some next-generation non-relational database's, in: R.P. Padhy (ed.), *International Journal of Advanced Engineering Sciences and Technology*, **11(1)** 2011, 15–30.

[6] **Lee, G.,** *Oracle Database 11g XML DB Technical Overview - An Oracle White Paper*, 36 pages, 2007.
`http://www.oracle.com/technetwork/database/features/xmldb/xmldb-11g-twp-132368.pdf`

[7] **Boag, S. et al. (eds.),** *XQuery 1.0: an XML query language*, 2005.
`http://www.w3.org/TR/xquery/`

[8] **Foping, F.S., I.M. Dokas, J. Feehan and S. Imran,** A new hybrid schema-sharing technique for multitenant applications, in: *Proceedings of the Fourth International Conference on Digital Information Management*, 2009, 1–6.

[9] **Räsinmaki, J.,** XQuery as a retrieval mechanism for longitudinal multi-scale forest resource data, in: Jakeman, A.J. (ed.), *Environmental Modelling and Software*, **24(10)**, 2009, 1153–1162.

Zs. Máriás, T. Nikovits and R. Giachetta
Faculty of Informatics
Eötvös Loránd Universtity
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
`zmarias@inf.elte.hu`
`nikovits@inf.elte.hu`
`groberto@inf.elte.hu`

T. Takács
Amnis Laboris Ltd.
Budapest
Hungary
`tamas.takacs@gmail.com`