Tuning Performance of E-Business Applications Through Automatic Transformation of Persistent Database Structures

Janusz R. Getta
School of Computer Science and Software Engineering
University of Wollongong
Wollongong, NSW
Australia
Email: jrg@uow.edu.au

***Abstract*—Performance of e-business applications strongly depends on internal implementations of persistent database structures and on effective algorithms processing these structures. Commercial database systems, which are the basis of e-business applications typically implement logical database structures with *one-size-fits-all* persistent storage structure.**

This work investigates a new class of database systems where a conceptual and logical view of a database can be implemented in many different ways depending on the performance requirements of database applications. We consider the improvements to performance of e-business applications through automatic changes of the persistent database structures in the ways indicated by the performance statistics obtained from tracing of the applications. In particular, this work describes a language for formal specification of variable persistent database structures, it shows how to find the best implementation structures for a given set of e-business applications, and it shows how to automatically transform one persistent data structure into another. We also investigate a problem how to automatically adjust the implementations of applications to the new persistent database structures.

I. Introduction

Performance of database systems is one of the most important factors in the development of information systems that provide access to continuously growing, global, and widely distributed sources of information. In particular, many of e-business applications frequently require extremely good performance indicators due to the constraints imposed by the business processes conducted in a particular environment, e.g processing of e-business applications at stock market environment. Performance of e-business applications is always determined by the efficient algorithms and persistent data structures which are *logically consistent* with the algorithms. Logical consistency of persistent data structures and algorithms means, that algorithms processing data structures do not impose any additional costs required by the implementations of specialized access methods. For example, assume, that an application frequently performs vertical and horizontal traversals of data organized in a hierarchical way. Then, the respective persistent data structures must be implemented in such a way, that the algorithms traversing the hierarchies minimize the total number of physical operations on persistent data structures implementing hierarchically organized data.

The existing database systems are typically based on a single method of implementation of their persistent data structures. When a new application uses the algorithms that are logical inconsistent with the current implementation of a database system then a standard solution is to create and to maintain the additional persistent data structures, like indexes, clusters, and partitions or to change a database system. For example, the majority of relational database systems implement relational tables as sequences of records sequentially stored in data blocks. When an application performs an exhaustive scan of few selected columns from a relational table then implementation of the table as a sequence of rows stored in data blocks requires access to all data blocks even so, only few columns are needed. Implementation of a vertical partition on the selected columns or and index on the columns and vertical traversal of leaf level of an index solves the problem. Unfortunately, such solution creates the problems with efficiency of data manipulation applications due to the additional physical operations required to maintain the consistency of a vertical partition or an index with a relational table. The additional data manipulation operations are required because certain amounts of data are replicated in the additional persistent data structures, e.g. an index replicates the values of key attributes. The additional persistent data structures, that improve performance are the *band-aid* ideas, which try to repair persistent storage structures that are not logically consistent with the algorithms used by database applications.

A simple observation that *column-by-column* instead of *row-by-row* implementation of a relational table is more logically consistent with an algorithm that requires access to the individual columns leads to a conclusion that many performance problems can be eliminated through an appropriate implementation of the original persistent data structures. In a general case, whenever a database application runs to slow then it should be possible to automatically adjust the persistent database structures to the algorithms applied in the application. A class of database systems, which enable

automatic modification of persistent data structures is called as *database systems with variable implementation schemas*[1].

An idea of many different persistent data structures possible for implementation of one logical schema dramatically changes database performance tuning. In a traditional approach performance tuning of database applications is based on the automatic or semiautomatic improvements to the algorithms used by the applications, tuning the parameters of database servers, tuning database concurrency, etc. The idea of variable implementation schemas allows for the automatic modifications of persistent data structures implementing logical structures of a database in order to improve the performance indicators. Variable implementation schemas would make integration of different database systems much easier because there will be no need to deal with syntactical heterogeneity of the logical data models. Another useful application is an automated performance tuning through modifications of implemented data structures just before processing of an application. A typical example of such applications is dynamic creation of indexes, vertical projections of relational tables, or even denormalization of relational tables in a period of time when database processing load is low.

The objective of this work is to investigate the systems capable of automatic changes to persistent database structures depending on the performance statistics obtained from the traces of processing of database applications. In particular, we consider the following problems: a language for formal specification of persistent implementation structures, finding the best persistent data structures for a given set of user applications, mapping of data into linear persistent storage, automatic transformation of one data structure into another, and automatic adjustments of user applications to the new implementation structures.

The paper is organized in the following way. The next section overviews the previous works in this area. Section (3) presents a language for formal specification of implementation schemas, and mapping of data into persistent storage. Optimization of implementation schemas is discussed in section (4). The same section presents optimization of the processing costs for a given set of applications through creation of an appropriate implementation schema. Next, we discuss automatic transformation of user applications to reflect the changes to implementation schemas. Section (5) includes discussion of the proposed solution and presents the open problems. Finally, section (6) concludes the paper.

II. Previous works

The implementations of persistent database structures to large extent follow the views provided by the logical data models. For example, a tabular view of data in the relational database model is usually implemented either as sequences of flat records representing rows [2] or as sequences of long records representing columns [3]. Hierarchical structures of XML data model are implemented as nested persistent data structures [4]. Object-oriented and object-relational models are implemented as sets of data records linked through the references to their addresses in persistent storage [5].

Performance of database applications is strongly related to a logical view of data and persistent data structures implementing a logical view of data. For example, the performance of navigation through the linked structures was always quoted in favour of object-oriented or object-relational data models [6] when compared with the relational data model. Faster traversal through the hierarchical structures supports XML data model [4] over the tabular or linked organizations of data. Recently, performance of distributed database applications is used to promote $<$*key,value*$>$ data model [7].

The past research works the languages for specification of physical data structures are very rare. The best known is a database implementation language E [8]. The language allows to describe the persistent database structures and it provides a type manager to maintain state and location information about the types and procedures used in the implementation. A group of database programming languages like [9], [10], [11] attempted to link logical logical specification of a database with its physical structures.

Some of the existing relational database systems provide the options for different implementation of relational tables. For example, the tables can be implemented is B*-Tree indexes or ISAM files. The system with both row-by-row column-by-column implementation of relational tables like [3] or in Monet Db [12] to some extent implement an idea of variable implementation schemas.

A language presented in this work is based on the Object-Relationship-Attribute Semi-Structured (ORA-SS) data model presented in [13] for specification of implementation schemas. The objectives of the original ORA-SS model was to express richer semantics without loosing the properties of modeling of semistructured data.

III. Implementation schemas

A notation of extended Object-Relationship-Attribute Semi-Structured (XORA-SS) data model is used for formal specification of persistent structures of a database system, i.e. *implementation schema* of a database system. The extensions of the original ORA-SS [13] include the concepts of indexes, implementation of attributes, implementation techniques for classes of objects and associations, and multilevel vertical and horizontal partitioning of persistent storage. At the moment XORA-SS model can be used to describe a wide range of persistent data structures. An extended Object-Relationship-Attribute data model (XORA-SS) model is defined around the concepts of *implementation schema* and *instance of implementation schema*. An *implementation schema* is a set of pairs $\{<D_1, A_1>, \ldots, <D_n, A_n>\}$ where for all $i = 1, \ldots, n$ each D_i is a *data component* and each A_i is an *association component* of a data component D_i.

A *data component* D is a tuple $<d_1, \ldots, d_m>$ where D is a name of data component and for all $i = 1, \ldots, m$ all d_i are either (1) *vertical partitions* or (2) *horizontal partitions* or (3) *descriptions of attributes*. Note, that all d_is in a data

component must be of the same type, i.e. all d_is must be either *vertical partitions* or *horizontal partitions* or *descriptions of attributes*.

A *vertical partition* V is a tuple $<v_1, \ldots, v_m>$ where V is a name of vertical partition and all for all $i = 1, \ldots, m$ v_i are either all (1) *horizontal partitions* or (2) all *descriptions of attributes*. Like in the previous case all v_is in a vertical partition must be of the same type, i.e. all v_is must be either *horizontal partitions* or *descriptions of attributes*.

A *horizontal partition* H is a tuple $<\phi, h_1, \ldots, h_m>$ where ϕ is a selection condition and for all $i = 1, \ldots, m$ all h_i are either (1) *vertical partitions* or (2) all *descriptions of attributes*. Like in the two previous cases all h_is in a horizontal partition must be of the same type, i.e. all h_is must be either *vertical partitions* or *descriptions of attributes*. A selection condition decides which instances of objects belong to a given partition.

A *description of attribute* is a quadruple $<a, t, m, i>$ where a is a name of attribute, t is a type of attribute, m is a multiplicity of attribute, and i is a potentially empty set of indexing tags. An index tag i is a pair $<x, k>$ where x is an identifier of an index an attribute belongs and k is a position of an attribute in an index key. A set of indexing tags determines the index keys an attribute belongs to and the respective positions within the keys.

An *association component* is a set of quadruples $<s, m, i, d>$ where s is an association name, m is a pair of multiplicities of association, i is an implementation indicator, and d is a data component name. Each quadruple describes the properties of a unidirectional association between a data component it belongs to and other data components in a database schema.

For example, a fragment of implementation schema, that describes a domain where students study at universities, enroll courses, and take projects is the following.

```
STUDENT(
 NAME( number integer(7)[1..1] {IDX1(1)},
       first-name string(30)[1..1],
       last-name string(30)[1..1] ),
 ADDRESS(
  SYDNEY(city='Sydney',
         street string(20)[1..1]{IDX2(1)},
         bldg integer(3)[1..1]  {IDX2(2)},
         flat integer(3)[0..1]  {IDX2(3)}),
  OTHER( true,
         city string(20)[1..1],
         street string(20)[1..1],
         bldg integer(3)[1..1],
         flat integer(3)[0..1] ) ),
 STUDIES[1..*][1..1] reference UNIVERSITY,
 ENROLS[1..*][1..*] index(number) COURSE,
 TAKES[1..1][1..1] nest PROJECT );
```

A conceptual schema, which has been used to create an implementation schema above is given at Figure 1. A fragment implementation schema given above describes implementation of a class `STUDENT` and unidirectional associations

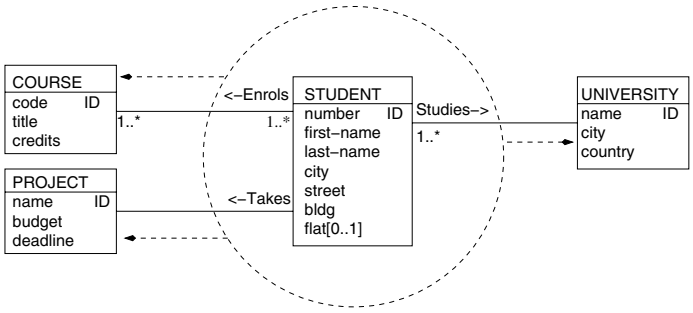

Fig. 1. A sample conceptual schema

`STUDIES`, `ENROLS`, and `TAKES` as it is marked with the dashed lines. Implementation of a data component consists of two vertical partitions `NAME` and `ADDRESS`. The second vertical partition consists of two horizontal partitions `SYDNEY` and `OTHER`. A horizontal partition `SYDNEY` includes the implementation of objects where a value of attribute `city` is equal to `'Sydney'` and the second horizontal partition includes all other objects. An index `IDX1` is created over the values of attribute `number` in all instances of data component `STUDENT`. The index key consists of one attribute `number`. Another index `IDX2` is created over the values of attributes `street`, `bldg`, and `flat` of the instances of horizontal partition `SYDNEY` within a vertical partition `ADDRESS`. The index key is a sequence of the attributes `(street,bldg, flat)`.

An association component consists of implementation of three unidirectional associations. A unidirectional association `STUDIES` is implemented through a reference to implementation of a class `UNIVERSITY`. A unidirectional association `ENROLS` is implemented through an index on implementation of a class `COURSE`. A unidirectional association `TAKES` is implemented through nesting of implementation of a class `PROJECT` within implementation of a class `STUDENT`.

An implementation indicator `reference` means that an association is implemented as a link-by-reference. An implementation indicator `index(number)` means that an association is implemented as an index on an attribute included in a data component of the other implementation. An implementation indicator `nest` means that a relationship is implemented as nested instance of one data component within an instance of another data component. A complete implementation schema for a design given in Figure 1 consists of four data components `STUDENT`, `UNIVERSITY`, `COURSE`, and `PROJECT` and six unidirectional association components `Studies`, `reverse(Studies)`, `Enrols`, `reverse(Enrols)`, `Takes`, `reverse(Takes)`, two for each association in a design.

A. Implementation of data components

The *instances of data components* are created when data are entered into a database. Then, each data component has a number of instances associated with it. An instance of data component is a structure of tuples assembled from attribute

values. The instances of data components are horizontally and vertically partitioned in a way determined by an implementation schema. All vertically partitioned instances of data components are linked such that it is possible to restore a complete instance. The same applies to groups of horizontally partitioned instances of data components. The links between the groups allow to restore a complete set of instances of data components. The association components are implemented in the ways determined by implementation indicators.

Implementation of instances of data components maps the instances of data components into the data blocks, which are the elementary components of persistent storage. Persistent storage is a sequence B of data blocks $<b_1, \ldots, b_n>$ of the same and fixed size, and accessible in either sequential model or in a random mode by providing a block number $i \in \{1, \ldots, n\}$. Let P be either vertical or horizontal partition that consist only of attribute descriptions $<a_1, \ldots, a_n>$. Then, an instance i_P of partition P is a tuple $<v_1, \ldots, v_n>$ where for all $i = 1, \ldots, n$ each v_i is a value of attribute a_i. Implementation of an instance i_P is a piece of persistent storage, that consists of the components $<l_1, \ldots, l_n, free, v_1, \ldots, v_n>$ where each l_i is a link to a value v_i of attribute a_i and $free$ is a chunk of persistent storage left free for future updates. It means that tuples of attribute values are implemented as fragments of persistent storage where a dictionary of links $l_1, \ldots, l_n$ keeps information about physical addresses of the values $v_1, \ldots, v_n$ and certain amount of persistent storage is left between a directory of links and attribute values for future extensions of attribute values.

Let I_P be a set of instances of either vertical or horizontal partition P. Implementation of a set of instances I_P is determined by a mapping imp of I_P into a contiguous sequence B of data blocks $<b_1, \ldots, b_n>$, $imp : I_P \rightarrow B$ such that for any two instances s_i and s_j if $i < j$ then $bnum(imp(s_i)) < bnum(imp(s_j))$ where $bnum(b_i)$ is a block number of b_i. To simplify a definition we assume an idealistic case where each instance $i_P \in I_P$ is implemented in exactly one data block. In practice it is not always true. For example, in a relational database system it is possible that a row spans over many data blocks. However, such situation is always considered as being harmful for performance and it is always recommended to increase block size such that each row fits in one block.

Let D be either vertical or horizontal partition that consists of other partitions $<d_1, \ldots, d_n>$. Then a set of instances of a partition D is a piece of persistent storage that consists of the components $< imp(d_1), \ldots, imp(d_n) >$, where each $imp(d_i)$ is an implementation of partition d_i. For example, consider a set of instances of data component D given in Figure 2. Each instance of data component is represented by a horizontal tile separated from adjacent instances with a dashed line. A set of instances is vertically partitioned into partitions $V1$, $V2$ and $V3$. It means that each implementation of instance of data component D is divided into 3 persistent storage chunks and linked with pointers. A vertical partition $V1$ is horizontally partitioned into partitions $H1$ and $H2$. A vertical

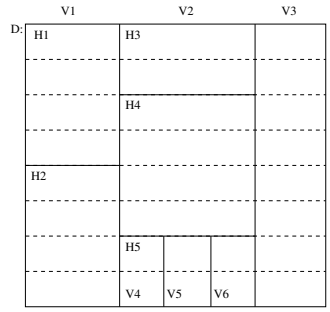

Fig. 2. A sample partitioning of a set of instances of data component

partition $V2$ is horizontally partitioned into $H3$, $H4$, and $H5$. Then, a horizontal partition $H5$ is vertically partitioned into $V4$, $V5$, and $V6$. The instances of data component D are implemented in a linear storage in the following way. Persistent storage is divided into three areas to implement the vertical partitions $V1$, $V2$ and $V3$. The first partition is divided into two subareas to implement the horizontal partitions $H1$ and $H2$. Persistent storage allocated for second vertical part is divided into three horizontal partitions. Persistent storage allocated for partition $H5$ is divided into three vertical partitions. A symbolic map of storage allocations is the following $[V1[H1, H2], V2[H3, H4, H5[V4, V5, V6]], V3]$ where any sequence of partition names included within square brackets represents a chunk of persistent memory implementing the respective partitions.

B. Implementation of association components

Implementation of association components allows database applications to navigate from one instance of data component to the other. Associations can be implemented in many different ways. Let d_i and d_i are the instances of data component that should be linked through an unidirectional association $<d_i, d_j>$. Then, in a general case a unidirectional association can be implemented in two different ways. First, if we access d_i then a location of d_j can be computed from the contents and/or location of d_i. In the simplest case, d_j is adjacent to d_i in the same data block and its location is computed from a location and length of d_i. In a relational database a value of foreign key in d_i references a value of primary key in d_j. The other way unidirectional association can be implemented as a persistent storage structure that allows for navigation from d_i to d_j. For example, in the simplest case d_i contains in its body a direct link to d_j.

Type of implementation of association component is determined by an *implementation indicator* that must be provided for specification of each unidirectional association. An implementation indicator *nest* means, that an association is implemented through nesting of implementation of schema instances. Then, all instances of the same schema are stored in a sequence of adjacent data blocks. An implementation indicator *reference* means, that an association is implemented through links (pointers) from elements of implementation of one schema instance to another. An implementation indicator

index(key) means, that an association is implemented through traversal of an index over a given *key*. In such a case the values of attributes on one side of association are used to form a *key*, which later on is used to traverse an index over the same attributes on the other side of association. An implementation indicator *value* means, that association is implemented as a pair of identical values on both sides of association like primary and foreign key in a relational model of data. Other implementation indicators determine other implementation techniques.

It is important to note, that implementation of association represent only unidirectional associations and, that in a general case two unidirectional implementations of the same association can be different. For example, and association `STUDENT TAKES SUBJECT` can be implemented through nesting as in the example above (`TAKES[1..1][1..1] nest PROJECT`) and through reference of in a reverse way (`TAKEN-BY[1..1][1..] reference STUDENT`). It is also important to know, that some combinations of implementation of associations are impossible. In a situation when the physical locations in persistent storage of instances of data components determine association with other instances it may not be possible to find these locations such that both unidirectional associations are implementable. For example, implementation of *many-to-many* association through nesting of both unidirectional association is not possible because in linear persistent storage it is impossible to order the records such that any two orders can be enforced, e.g. it is impossible to order the records in both ascending and descending order.

C. Logical design

An interesting problem is how an idea of variable implementation schemas can be applied to the existing database systems. A direct transformation of a conceptual schema $\mathcal{C}$ into an implementation schema $\mathcal{I}$ omits a stage of logical database design where a conceptual schema is transformed into a logical schema $\mathcal{L}$. On the other hand, the majority of existing database systems provide the users with a logical view of data and implementation of such system requires a transformation of a logical schema $\mathcal{L}$ into a physical schema $\mathcal{I}$. For example, relational database systems provide a logical view of a database as a collection of two dimensional tables with data linked through the values of primary and foreign keys implemented as the sequences of records stored in the sequences of data blocks. Application of an idea of variable implementation schema for an existing database system means that we need to find a transformation of a logical schema $\mathcal{L}$ into a more efficient implementation schema $\mathcal{I}'$ than the actual schema $\mathcal{I}$. It means that in all database applications, all operations on persistent storage organized accordingly to $\mathcal{I}$ must replaced with a set of operations on persistent storage organized accordingly to $\mathcal{I}'$. To achieve that we need to reverse engineer a logical schema $\mathcal{L}$ into a respective conceptual schema $\mathcal{C}$ and describe an existing implementation in a language XORA-SS as an implementation schema $\mathcal{I}$. Next, we find a transformation of formally described implementation schema $\mathcal{I}$ into a formally described implementation schema $\mathcal{I}'$ which is more efficient for a given set of applications. The transformation found will provide information on how the implementations of operations associated with $\mathcal{I}$ should be replaced with the implementations of equivalent operations associated with $\mathcal{I}'$. Finally, replacement of the implementations of operations used in the applications requires to relink the applications with the libraries implementing operation of a new schema $\mathcal{I}'$.

As a simple example consider a relational database that consists of two relational tables `emp(enum,fname,lname,dname)` and `dept(name,budget`. A primary key `name` in `dept` references a foreign key `dname` in `emp`. Assume, that a sample application joins the relational tables `emp` and `dept` in order to find all employees who belong to a given department. Reverse engineering of the relational schemas provides two classes `EMPLOYEE` and `DEPARTMENT` and an *one-to-many* association `DEPARTMENT Employs EMPLOYEE`. In the next step we find an implementation schema for a conceptual schema found in the previous step. An implementation schema must be consistent with the implementation of the relational tables `emp` and `dept` and implementation of association `Employs` as a *link-by-value*.

```
DEPARTMENT(name string(30)[1..1]{IDX1(1)},
           budget float[1..1],
 EMPLOYS[1..1][1..*]
                    value EMPLOYEE(dname));
EMPLOYEE(enum integer(7)[1..1] {IDX1(1)},
         fname string(20)[1..1],
         lname string(20)[1..1],
         dname string(30)[0..1] );
```

Implementation of traversal of association `EMPLOYS` in the schema above is done as hash based join of data components `DEPARTMENT` and `EMPLOYEE`. If we find that implementation of association `EMPLOYS` is more efficient for the application considered above through nesting of instances of data component `EMPLOYS` within the instances of data component `DEPARTMENT` then we change implementation of association `BELONGS` into

```
EMPLOYS[1..1][1..*] nest EMPLOYEE
```

Additionally, we change implementation of association `EMPLOYS` from hash based join into selection of nested values of data component `EMPLOYEE` for a given instance of data component `DEPARTMENT`. The new implementation is relinked with an application that finds all employees who belong to a given department. At the end, the contents of a database must be reloaded into persistent storage to be consistent with the new implementation schema.

A problem how to find the best implementation schemas for a given collection of applications and for a given frequency indicators associated with the applications is considered in the next section.

IV. Optimization of implementation schemas

Optimization of implementation schemas is performed through the adjustments of implementation techniques used for data components and association components in response to the recent and/or predicted database load. The adjustments are derived from the statistics collected from tracing of database applications and later on used to calculate the current costs of the processed applications. The implementation techniques are changed in order to minimize the future costs of the same applications.

A. Costs analysis

To estimate the costs a binary directed graph called as *implementation graph* is built from the specifications of implementation schemas. The nodes of an *implementation graph* represent data components of the schemas and the arcs represents association components. An application is modeled as a path that starts at a node, ends at another node, and passes through a number of edges. A set of all applications processed in a given period of time is used to find for each data component and for each unidirectional association the frequencies of traversals, updates, insertions, and deletes. The frequencies and information about implementation techniques of data components and associations are used to calculate the costs of processing. A cost of processing of a given data component d is described by a formula $cost(d) = f_u * cost(d_u, j) + f_i * cost(d_i, j) + f_d * cost(d_d, j) + f_t * cost(d_t, i)$ where $f_k * cost(d_k, j)$ is a cost of operation k performed with a frequency f_k on a data item d implemented using a technique j. The costs $cost(s)$ of processing of each unidirectional association s in an implementation graph are calculated in the same way. The total costs of processing imposed on each schema that consists of a data component d and n unidirectional associations $s_1, \ldots s_n$ is calculated as $cost(d) + cost(s_1) + \ldots cost(s_n)$ where $cost(s_i)$ is a total cost of traversal of unidirectional association s_i.

With the finite number of data components, associations, and implementation techniques it is possible to minimize the costs through considering all possible combinations of implementations of each data component d and unidirectional association s.

The maintenance of implementation graph during life time of a system allows for continuous monitoring of data processing costs. When the costs of processing of a data or association component significantly increase then a number of different implementations of a component are considered and for each implementation the total costs of processing are estimated using information about database load since the last modification of persistent structures. An implementation that provides the best estimation of the costs and better than the current implementation is selected for a transformation of the component and for the modifications of all applications processing the component. Modification of the applications is quite simple as it only requires re-linking of the applications with the libraries implementing the same data access routines in a way consistent with a new implementation. Transformation of implementation of a data or association component may be a very time consuming task because of large amounts of persistent storage to be processed and complexity of a new implementation. For example, a transformation of *value* technique of implementation of association into *reference* technique requires navigation through all instances of associations in order to establish forward references between the individual instances of data components. Such task is not performed immediately after a new implementation is decided but it is delayed to a moment when an expect load of a system is low over a period of time long enough for the transformation.

B. Transformation of user applications

Optimization of implementation schemas through the systematic adjustments of implementation techniques to the most frequent and time consuming applications requires automatic changes to implementation of the applications. Automatic changes of modifications require standardization of access methods across all different implementation schemas. Automatic reimplementation of applications needs every element of every component of implementations schema to have a separate access method for every possible implementation technique. Let $d_i \in D$, then $m_d(d_i, j)$ denotes implementation of access method for element d_i implemented with a technique j. Let $p_i \in P$, then $m_p(p_i, j)$ denotes implementation of access method for partition p_i implemented with a technique j. Similarly, for $s_i \in S, m_s(s_i, j)$ denotes implementation of access method that traverses a unidirectional association s_i implemented with a technique j. Then, if implementation technique for a data, partition, or association component is changed to speed up the applications then the applications which use access methods implemented with an old technique must be recompiled and relinked with access methods that use a new implementation technique. Polymorphic implementation of access methods does not need any syntactical changes to implementation of the applications.

It may happen that optimization may require the modifications to an existing partitioning component. For example a vertical partition can be changed or added. As partitioning is transparent to implementation of applications, i.e. a query optimizer picks a partition which is the most effective for a given application, entire application must be recompiled again a new set of partitions.

V. Discussion/Results

The idea of variable structures of persistent storage implementing a single conceptual schema has a number of advantages. First, it dramatically improves a process of database performance tuning. In the traditional approaches performance tuning of database applications was mainly based on the manual improvements to the algorithms used by the applications, on creation of additional persistent data structures, on tuning the parameters of database servers, SQL, and database transaction processing. Automatic transformation of persistent database structures allows for the adjustments of

data structures to the existing algorithms and for elimination of logical inconsistencies between the properties of algorithms and data structures and in a consequence it eliminates the improvements to algorithms and also the additional persistent database structures.

If a poor performance of a database application is caused by a lack of logical consistency between the persistent storage structures implementing a particular data model and a class of algorithms used by the application then a typical solution to a problem of was to apply a database system based on a different data model, i.e. a model which allows for better implementation of the algorithms used by the application. For example, an application that processes graph data structures is more consistent with object-oriented than with relational data model. In the approach proposed in this work an implementation schema can be adjusted to the algorithms without any changes to a conceptual schema.

Application of database systems based on many different logical data model creates problems with integration of information e.g. the famous *impedance mismatch problem* between a relational implementation model and object-oriented view at logical level. The idea of variable persistent database structures eliminates this problem as well. Automatic modification of persistent database structures eliminates a problem of integration of different database models. In fact the existing techniques of indexing and clustering in database systems are the special cases of variable persistent database structures. It makes it useful for an automated performance tuning through modifications of implemented data structures just before processing of an application. A typical example of such applications is the dynamic creation of indexes, vertical projections of relational tables, or even denormalization of relational tables in a period of time when database processing load is low

VI. Conclusions

This work investigates a problem of performance tuning of e-business applications through automatic transformation of persistent database structures in a response to the varying characteristics of database load. We consider a scenario where the new and modified e-business applications change the navigation patterns in a database system. Then, the database processing statistics collected from tracing of the applications indicate new processing costs and the changes that should be done to the persistent database structures in order to improve performance of the applications. The algorithm presented in this work finds the best persistent database structures for a given set of user applications and it shows how the applications must be changed in order to process the modified database structures.

We propose a sample language for specification of persistent database structure. The language allows for specification of multilevel horizontal and vertical partitioning of data objects and for specification of various ways how associations between the objects can be implemented. The language can be easily extended with more implementation techniques for persistent data structures and more access methods for implementation of database applications.

A significant advantage of the proposed approach is a direct transformation of conceptual database schemas into implementation schemas through mapping of classes of objects and association into implementation schemas. Such approach makes logical data models obsolete and it eliminates a gap between object-oriented programming languages and database models.

A number of interesting research problems remain to be solved. An interesting problem is an appropriate selection of the basic implementation techniques for persistent database structures and a system of abstract operation that combine such techniques into more sophisticated ones. Any system of persistent database structures must come with a respective system of elementary operation on these structures. Automatic implementation of systems of operation for complex persistent database structures is another interesting problem. Finally, implementation of a prototype database system with variable implementation schemas is a problem whose solution would confirm the validity and practicality of entire idea.

References

[1] J. R. Getta, "On database systems with variable implementation schemas," in *Proceedings of the 4th International Conference on Computational Intelligence and Software Engineering*, 2012.

[2] S. Lightstone, T. Teorey, and T. Nadeau, *Physical Database Design*. Morgan Kaufman, 2007.

[3] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. 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*, 2005, pp. 553–564.

[4] J. Siméon and P. Wadler, "The essence of XML," in *POPL*, 2003, pp. 1–13.

[5] J. Hrivnac, "Transparent persistence with Java Data Objects," *CoRR*, vol. cs.DB/0306013, 2003.

[6] S. Alagic, "A family of the ODMG object models," in *ADBIS*, 1999, pp. 14–30.

[7] J.-D. Cryans, A. April, and A. Abran, "Criteria to compare cloud computing with current database technology," in *IWSM/Metrikon/Mensura*, 2008, pp. 114–126.

[8] J. E. Richardson and M. J. Carey, "Persistence in the E language: Issues and implementation," *Software-Practice & Experience*, vol. 19, no. 12, pp. 1115–1150, 1989.

[9] J. W. Schmidt and F. Matthes, "The DBPL project: Advances in modular database programming," *Inf. Syst.*, vol. 19, no. 2, pp. 121–140, 1994.

[10] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri, "Implementation of the CORAL deductive database system," in *SIGMOD Conference*, 1993, pp. 167–176.

[11] J. Annevelink, R. Ahad, A. Carlson, D. H. Fishman, M. L. Heytens, and W. Kent, "Object SQL - a language for the design and implementation of object databases," in *Modern Database Systems*, 1995, pp. 42–68.

[12] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: a fast XQuery processor powered by a relational engine," in *SIGMOD Conference*, 2006, pp. 479–490.

[13] M. Liu and T. W. Ling, "A data model for semistructured data with partial and inconsistent information," in *Proceedings of the 7th International Conference on Extending Database Technology*, 2000.