Title:
Optimization by Indices in ODRA

Authors:
Tomasz Marek Kowalski, Jacek Wiślicki, Kamil Kuliberda, Radosław Adamus, Kazimierz Subieta

Affiliations:
Polish-Japanese Institute of Information Technology, Warsaw, Poland
Computer Engineering Department, Technical University of Lodz, Lodz, Poland

Addresses:
Polish-Japanese Institute of Information Technology, ul. Koszykowa 86, 02-008 Warsaw, Poland
Computer Engineering Department, Technical University of Lodz, ul. Stefanowskiego 18/22, 90-924, Lodz, Poland

Emails:
jacenty@kis.p.lodz.pl, kamil@kis.p.lodz.pl, tkowals@kis.p.lodz.pl, radamus@kis.p.lodz.pl subieta@pjwstk.edu.pl

Abstract.

The paper presents the features and samples of use of the optimization by indices module implemented in the ODRA (Object Database for Rapid Applications development) prototype system. The ODRA indices implementation is based on Linear Hashing that can be used for standalone databases as well as for distributed environments in order to optimally utilize data grid computational resources. Implementation supports transparent optimization, automatic index updating and indices management facilities

.

Optimization by Indices in ODRA[*]

Tomasz Marek Kowalski, Jacek Wiślicki, Kamil Kuliberda,
Radosław Adamus, Kazimierz Subieta

Polish-Japanese Institute of Information Technology, Warsaw, Poland
Technical University of Lodz, Lodz, Poland
{ tkowals, jacenty, kamil, radamus}@kis.p.lodz.pl
subieta@pjwstk.edu.pl

Abstract. The paper presents the features and samples of use of the optimization by indices module implemented in the ODRA (Object Database for Rapid Applications development) prototype system. The ODRA indices implementation is based on Linear Hashing that can be used for standalone databases as well as for distributed environments in order to optimally utilize data grid computational resources. Implementation supports transparent optimization, automatic index updating and indices management facilities.

1. Introduction

Indices are auxiliary (redundant) database structures stored at a server. A database administrator manages a pool of indices generating a new one or removing them depending on the current need. As indices at the end of the book are used for quick page finding, a database index makes quick retrieving objects (or records) matching given criteria possible. Because indices have relatively small size (comparing to a whole database) the gain in performance is fully justified by some extra storage space. Due to single aspect search, which allows one for very efficient physical organization, the gain in performance can be even several orders of magnitude. The general idea of indices in object-oriented databases does not differ from indexing in relational databases [1, 2]. Many solutions can be adopted from traditional database systems and even theirs practicability can be significantly extended. There are also situations where indexing stops to be necessary for object-oriented databases e.g. in **join** operation due to introduction of reference objects and object identifiers.

ODRA is a prototype object-oriented database management system based on Stack Based Architecture (SBA) [8,9]. The main goal of the ODRA project is to develop new paradigms of database application development. The main goal of the ODRA development was to introduce a new, universal, declarative programming language, together with distributed, database-oriented and object-oriented execution environment. ODRA introduces its own query language SBQL (Stack Based Query Language) that is integrated with programming capabilities and abstractions, including database abstractions: updatable views, stored procedures and transactions.

One of the important features of prototype system ODRA is optimization by indices module, a part of ODRA optimization engine. Main features of the indices implementation includes: optimization module able to transparently choose appropriate index for a given query (if available), automatic update of indices in response to update of corresponding data and the administration module for indices managing.

The rest of the paper is organized as follows. Section 2 presents overall architecture of ODRA query optimization engine, section 3 discusses the features of indices in ODRA, section 4 describes ODRA index management facilities, section 5 exemplifies query optimization based on indices, section 6 shortly describes ODRA optimization framework, section 7 concludes.

2. Overall architecture of ODRA Query Optimization Engine

Figure 1 shows ODRA query optimization process in the context of query evaluation process.
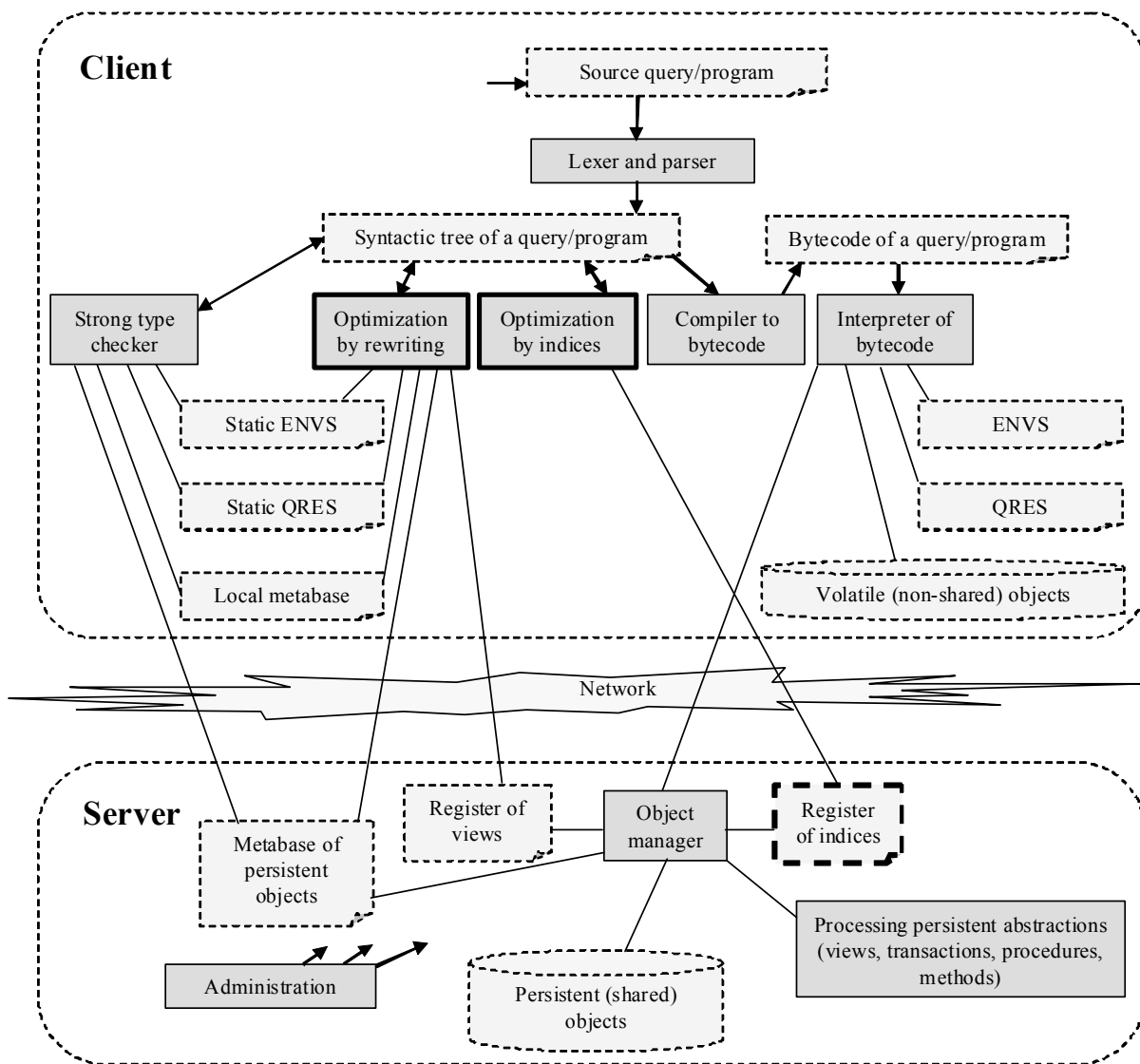
Figure 1 ODRA Architecture

The input for the optimization process is the abstract syntax tree (AST) of the input query. The optimization modules are divided into: optimization by rewriting and optimization by indices. The theoretical idea for these methods were developed and presented in many works e.g. [6, 7, 8, 9].

The rewrite optimization process modifies the query during compile-time with use of information stored in the metabase augmented with static query evaluation results. Currently ODRA supports several rewriting methods: changing the order of execution of algebraic operators; view rewrite (replacing view invocation with view body); removing dead sub-queries; factoring out independent sub-queries; shifting conditions as close as possible to the proper operator; methods based on the distributivity property of some query operators.

Optimization by indices searches for parts of the input query that can be transparently replaced with an index call. If such an index exists (added previously by the administrator) the query is rewritten to the form where the target part is replaced with an index invocation.

3. General Idea of ODRA Indices

In general, an index is a two-column table where the first column consists of unique key values and the other one holds non-key values, which in most cases are object references. Figure 2 shows example indices for a given object-oriented database store. Key values are used as an input for index search procedures. As the result, such a procedure returns suitable non-key values from the same table row. Keys are usually values of database objects specific attributes (*dense* indices) or represent ranges of these values (*range* indices).
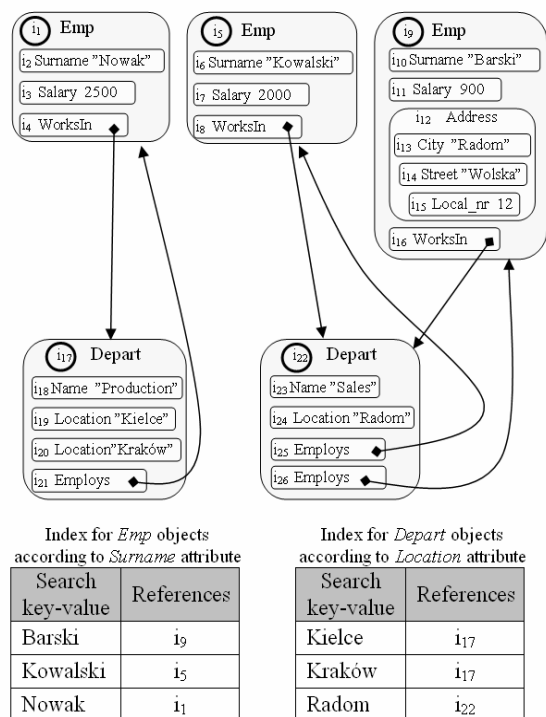
Figure 2 Example of dense indices for given object-oriented database store

Key values can be also result of expressions, build-in query language functions or user defined functions calculated from object attributes (function-based indices). The last approach enables the administrator to create an index matching exactly predicates within the frequently occurring queries, so their evaluation is faster and uses minimal amount of I/O operations.

The most popular data-structures used for index organization are indices with hash coding, indices based on B-tree (a balanced tree) and bitmap indices.

In a query optimization indices are used in the context of a **where** operator when the left operand is indexed by key values of the right operand selection predicates. Lets make an example using the database store structure presented on Figure 2. If the administrator will set an index named *idxEmpSalary* which returns references to *Emp* objects depending on their salaries then e.g. the following queries will produce the same result.

```
(Emp where Salary = 2000 and WorksIn.Depart.Name = "Sales").Surname;

(idxEmpSalary(2000) where WorksIn.Depart.Name = "Sales").Surname;
```

For big databases, replacing the **where** evaluation with a call of an index function call may cause performance gain even orders of magnitude. However to achieve this database should ensure index transparency and automatic index updating.

3.1 Physical Properties of Indices
The idea of indexing implies two important properties of indices:
- *index transparency* from a point of view of a database user. A programmer should not be aware of the indices existence, because they are used automatically during query evaluation. Therefore, the administrator of a database can freely generate new indices and remove them without a need to change the code of applications.
- *automatic index updating* which is the result of changes in a database. Indices, like all redundant structures, can lose cohesion if the database is updated. An automatic mechanism should improve, eliminate or generate a new index in case of database updates.

3.2 Index Classification
The most common classification of indices distinguishes primary and secondary ones or dense and range ones. From a query optimizer point of view a distinction of primary and secondary indices is less crucial because it does not lead to significant differences in optimizer algorithms, whereas a division into dense and range indices is essential:

- *a dense index* is *applied* when for each value in object's attributes a separate position in an index is created, e.g. for a *person* objects index, where any *name* occurring in a database can be a key-value,
- *a range index* means that index positions concern values within a given range, e.g. a range index for a *salary* attribute is a table where each index position describes a range of salaries: <0-500), <500-1000), <1000-1500)…, etc (Figure 3). Similarly range index positions for names can take following form: "names starting with a letter A", "names starting with a letter B", …, "names starting with a letter Z".

Range	Search key-value	References to Employees
<0, 500)	0	i_{15}
<500, 1000)	500	i_{72}, i_{43}
<1000, 1500)	1000	$i_{18}, i_{22}, i_{25}, i_{30}$
<1500, 2000)	1500	$i_{45}, i_{59}, i_{48}, i_{32}$
…	…	…

Figure 3 Example range index for Employees objects according to Salary attribute

3.3 Features of ODRA Indices

Currently the implementation supports indices based on Linear Hashing [3] structures which can be easily extended to its distributed version SDDS [4] in order to optimally utilize data grid computational resources. Nevertheless there is a wide range of different index structures that could be used in indexing in object-oriented databases similarly like in solutions occurring in relational ones [1, 2, 5, 10] : B-Trees (balanced trees), bitmap indices, etc.

An extended idea of an ODRA index works with multiple key indices. Additionally to key types mentioned earlier (*dense* and *range*) *enum* type was introduced to improve multiple key indexing (among other things).

ODRA supports local indexing which ensures index transparency by providing mechanism (optimization framework) to automatically utilize an index before query runtime evaluation and therefore to take the advantage of indices. ODRA CRUD (Create, Read, Update and Delete) is also equipped with triggers to ensure automatic index updating so existing indices are consistent with the databases state. All features which are necessary to provide correct work and usable indices are already implemented.

4. Index Management

All indices existing in a database are registered and managed by the ODRA *index manager*. The list of all indices and auxiliary information needed by the *index optimizer* are stored inside the special *admin* module. Each index is associated with a module where it was created and its name has to be unique. Therefore index manager checks whether a given index exists in the list of references to meta-base objects describing indices using the combination of a module's name and an index name: "*module_name.index_name*".

The administrator issues the 'add index' command in the administration module to create index in the database. The syntax of this command is the following:

```
add index <indexname> [ ( <type> [ | <type> ... ] ) ] on <creating_query>
```

where `<type>` is type indicator and can hold one of the following values:
- **dense** – default type indicator.
- **range** – indicating that the index should support optimizing range queries on a given key.
- **enum** – indicating that the index is set up on a key with countable limited set of values (usually with low cardinality); *enum* also supports optimization of range queries.

The number of type indicators depends of the number of keys used for indexing (specified by a creating query). Each of them applies to the following index key. Type indicators are optional and if they are unspecified then the index is dense on all keys. Index type indicators are described in the further section.

The syntax of command for removing an index from the register is simple:
```
remove index <indexname>
```

4

4.1 Example schema

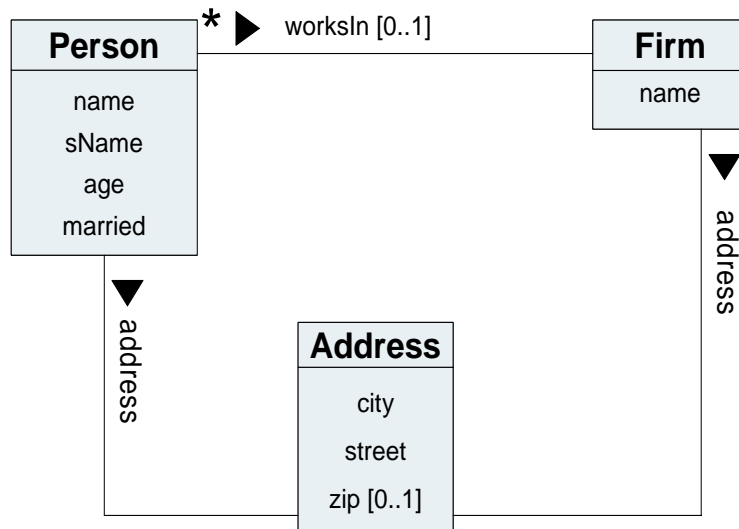The schema on **figure 4** will be used to exemplify indices usage.

figure 4 Example schema

4.2 Index Creating Query

Conceptually an index is a two-column table where the first column consists of unique key values and the other one holds non-key values which in case of ODRA are object references. For multiple keys indices a final key value (input for index search procedures) is treated as a combination of all individual key values.

To generate such a table ODRA first generates key values for each object. This is achieved using an SBQL query `<creating_query>`, which returns references to objects with associated key values. The administrator in order to create an index must provide the 'add index' command with such a query as a parameter.

   Syntax of `<creating_query>` is following:

```
<object_expression> join <key_expression>;
```

   where:
- `<object_expression>` - generates references to objects.
- `<key_expression>` - generates key values for given object

   Example query:

```
Firm join address.city;
```

Indexed objects are defined by `<object_expression>` which is bound in the lowest database section (database root). This expression should be built using only *dot* operators and name expressions i.e. simple path expressions. Additionally each name should directly address complex object (for the first name expression) or subobject (for the following name expressions) hence using reference objects threatens indexing the same object more then ones which is not currently supported.

Key value sub-expressions should be bound not lower then in *join* operator stack section. If the index is set on multiple keys then `<key_expression>` should consists of individual key expressions separated be comma, e.g.:

```
Person join ((2007 - age), address.city);
```

Keys are allowed to return values of following types: *integer*, *double*, *string, reference* (*dense* type indicator enforced) and *boolean* (*enum* type indicator enforced).

Important property of created index is the cardinality of keys. For each key it indicates the number of key values, which can be returned for given object. Usually the cardinality of all the keys is singular [1..1]. Then the number of elements in a creating query result is equal and adequate to the number of objects; thus the full index optimizer potential is enabled.

A different key cardinality causes that some objects are omitted in indexing (zero minimal cardinality) or more than one key value combination is generated for a single object (when the maximum cardinality is bigger then one then object references are not unique). These situations enforce more strict rules for query optimization utilizing index in order to preserve query semantics after optimization. Currently ODRA does not support indexing when keys maximum cardinality is above singular due to ambiguity in generating key values for given object.

4.3 Index Types

The syntax for creating index allows the administrator to specify general index key properties, i.e. concerning key values or the goal of optimization. These are achieved by introducing optional type indicators: *dense, range* and *enum.* The number of indicators is equal to the number of index keys defined by <creating_query> because each indicator is associated with a key.

The default type indicator for *integer, string, double or reference* values is *dense.* In case of *boolean* values, the *enum* type is always used. The *dense* indicator is always used for *reference* values (regardless of an indicator set by the administrator).

The *dense* indicator implies that the optimization of selecting queries which uses given key value a condition will be used only for selection predicates based on **'='** or *in* operators. Therefore the distribution of indexed objects in index (e.g. in hash table) can be more random. The order of key values has no significance for indexing.

```
add index idxEmpSal(dense) on Emp join salary;
```

The *range* indicator implies that optimized selection predicates will be based not only on **'='** or *in* operators but also on range operators: '>', '≥', '<' and '≤'. Within each index item object references are grouped according to key values ranges. In the current implementation ranges are dynamically split because each range is associated with individual bucket of linear hash map.

```
add index idxPerAge&WorkCity(range|dense)
    on Person join (age, worksIn.Firm.address.city);
```

The *enum* indicator was introduced in order to take an advantage of keys with countable limited set of distinct values. The performance of an index can be strongly deteriorated if key values have low cardinality e.g. person eye colour, marriage status (boolean value) or year of birth. Using the *enum* key type index internally stores all possible key values (or range for integer values) and uses this information to optimize the index structure.

The *enum* key type can deal with optimizing selection predicates exactly like in the case of the **range** indicator, i.e. for: '=', '**in**', '>', '≥', '<' and '≤' operators.

Another important property of *enum* keys occurring when index is set on multiple keys is that the optimizer can omit them if necessary during optimization of queries. If *enum* is set on all index keys and number of indexed objects is large then index call evaluation should prove great efficiency (each key value combination points to a separate object references array called bucket)

```
add index idxPerAge&Mar&City(enum|enum|enum)
    on Person join (age, married, address.city);
```

More example creating indices commands:

```
add index idxPerYear(range) on Person join (2007 - age);

add index idxPerCity(enum) on Person join address.cit
```

If optimization with indices is enabled in query evaluation (as described later) the only action for the administrator in order to take advantage of indexing is creation of proper indices (the rest of optimization is transparent for programmers). The next section describes ODRA's optimization rules which can be helpful in applying good indexing.

5. Query Optimization

In ODRA the use of indices is completely transparent for an application code. The programmer may be aware or not of existence of indices, but the code does not depend on it. The index optimizer automatically applies all possible indices during query compilation.

Besides this possibility, a user can also use indices explicitly. This feature is introduced for testing purposes in order to track mistakes in indexing engine, check semantic equivalence of introduced index optimizations and research into new possibilities in indexing.

In the following we briefly describe ODRA indices optimization engine module used for query optimization based on indices.

5.1 Explicit Use of an Index

Indices can be called in ODRA with syntax similar to procedure call:

```
<indexname> ( <key_param_1> [; <key_param_2> ...] )
```

The number of parameters is equal to the number of index keys. Each key parameter defines desirable value of a key. An index function call returns references to objects matching the specified criteria.

A key parameter expression can define a single value as a criterion. In such a case its evaluation should return *integer*, *double*, *string, reference* or *boolean* value result or reference to such a value. Below we present an examples calls for the sample index *idxPerAge&WorkCity* created in the previous section.

```
idxPerAge&WorkCity(27; "Łódź");
```

Alternatively a single value key parameter can be passed through a value of a binder named "$equal" (this is set by the index optimizer to increase readability):

```
idxPerYear(1980 groupas $equal);
```

To specify a range as a key value criterion parameter, an expression should return a structure consisting of four parameters:

```
(<lower_limit>, <upper_limit>, <lower_closed>, <upper_closed>)
```

where:
- `<lower_limit>` and `<upper_limit>` are key values specifying range,
- `<lower_closed>` is a boolean value indicating whether `<lower_limit>` belongs to a criterion range,
- `<upper_closed>` is a boolean value indicating whether `<upper_limit>` belongs to a criterion range.

Example indices calls:

```
idxPerAge&WorkCity((25, 30, true, true); "Warszawa");

idxPerAge&WorkCity(
    (0, (sum(Person.(2007 - age)) / count(Person)), true, false); "Łódź");
```

The last example returns references to persons working in Łódź whose age is below the average of all the persons from the database.

Like in case of single value key parameters, parameters specifying a range can be passed using the value of a binder named "$range" (this is set by the index optimizer):

```
idxPerYear((1978, 1982, true, true) groupas $range);
```

A key parameter can specify also collection of single key values as a criterion. This is done when a key parameter returns a bag of key values.

```
idxPerAge&WorkCity(25 union 30 union 35; "Boston");
```

Additionally, a binder named "$in" can be used to pass collection of key values (this is set by index optimizer):

```
idxPerAge&WorkCity((25 union 30 union 35) groupas $in;
                   "Boston" groupas $equal);
```

If a criterion parameter returns an empty bag then the index call returns the empty bag too.

5.2 Transparent Use of Indices - Index Optimization

The mechanism responsible for index transparency during query evaluation is called the *index optimizer*. Its function is to replace a part of a query with an index call in order to minimize amount of data processed.

This section describes general rules used in solving the problem of semantic equivalence of queries rewritten by the index optimizer and original input queries. Most of the following rules concern optimizing range queries. Index optimizer analysing the right operand of **where** non-algebraic operator takes into consideration all selection predicates joined with an conjunction **and** and disjunction **or** operators.

The basic index optimizer procedure works on selective queries where left side of the **where** operator is <object_expression> indexed by one or more indices. The algorithm analyses all selection predicates joined with the **and** operator and tries to find index that keys matches predicates. If more than one index is found optimizer selects one with best selectivity.

Firstly lets to consider how [0..1] key cardinality affects optimization. Using criteria with the discussed cardinalities may cause runtime errors because selection predicates based on '=', '>', '≥', '<' and '≤' operators force using single values as left and right operands. Unexpected number of operand values can cause runtime error. Using index call in optimization with these predicates would eliminate threat of error and therefore optimized query would not be semantically equal to original. As far in this cases optimization is allowed only if an **in** operator is used as a predicate because it does not constrain a cardinality of the right operand.

Example of unsafe predicate evaluation (may cause run-time error). Left side of selection predicate has cardinality [0..1] due to *worksIn* attribute:

```
Person where worksIn.Firm.address.city = "Szczecin";
```

Example of safe predicate evaluation as in operator is used:

```
Person where "Szczecin" in worksIn.Firm.address.city;
```

In discussed case the index optimizer supports optimization when predicates are defined '=', '>', '≥', '<' and '≤' operators only if proper exists predicate is used.

Example of safe predicate evaluation when '=' operator is used:
```
Person where exists(worksIn.Firm.address.city)
       where worksIn.Firm.address.city = "Szczecin";
```

A minimal cardinality of a key equal to zero indicates that the index may not contain references to all objects defined by an index <object_expression>. In case of multiple key index if such a key is omitted in selection predicates than it is possible that evaluation of the where operator may return references to objects that are not stored inside the index. Therefore the index optimizer would not apply optimization using such a index. To sum up, keys with minimal cardinality equal to zero are obligatory even if they are declared with *enum* type indicator.

Currently the maximum cardinality of keys greater then one is not supported by ODRA indices. However theoretically it would imply that an index call may return same object reference more than one time. To prevent such a problems in the future index optimizer uses the **unique** operator to remove redundant object references.

Index optimizer procedures described below are used mainly on keys with the cardinality [1..1].

If optimized query selection predicates specify only one limit of a range (lower or upper) then the second limit is generated automatically i.e. a possible smallest or biggest value for given key.

```
Original query:

Person where age > (sum(Person.(2007 - age)) / count(Person)) and
     "Warszawa" in worksIn.Firm.address.city;

Optimized query:

idxPerAge&WorkCity((sum(Person.(2007 - age)) / count(Person)), 2147483647,
     false, true), "Warszawa");
```

If there are more than one predicate or two opposite predicates describing the range on a given key then **min**, **max**, **union** and comparison expressions are used to obtain a correct key range parameter.

```
Original query:

((sum(Person.(2007 - age)) / count(Person)) as avgyear).(Person where  2007
     - age > avgyear and 1970 <= 2007 – age and 2007 - age < 1980);

Optimized query:

(sum(Person.(2007 - age)) / count(Person)) as avgyear).
     idxPerYear((max(avgyear union 1970), 1980, 1970 > avgyear, false));
```

The index optimizer in some cases uses **if then** expression to predict whether a given query returns no result and calling the index is unnecessary i.e. if selection predicates are in contradiction. This has to be checked e.g. when for a given key there exists more then a one selection predicate and at least one is based on '=' or **in** operator. If any of these selection predicates contradicts with a predicate based on '=' or **in** operator then such a query will return an empty bag as a result:

```
Original query:

((sum(Person.(2007 - age)) / count(Person)) as avgyear).
     (Person where 2007 - age >= avgyear and 1977 = 2007 – age);

Optimized query:

(sum(Person.(2007 - age)) / count(Person)) as avgyear).
          if (1977 >= avgyear) then idxPerYear(1977);
```

This procedure is used also when the key cardinality is different then [1..1], i.e. in case of two or more selection predicates based on **in** operator.
For multiple keys indices *enum* keys may be usually omitted in an index call. The index optimizer in order to omit key when no selection predicates were specified an sets both lower and upper bounds are to smallest and biggest key value:

```
Original query:

Person where true = married and address.city in "Wrocław";

Optimized query:

idxPerAge&Mar&City (
     (-2147483648 ,  2147483647 , true , true) groupas $range;
     (true) groupas $equal ; "Wrocław" groupas $equal );
```

To omit *boolean* key in an index call set key parameter criteria are used (**false union true**).

```
Original query:

Person where age > 30 and 33 >= age and address.city in "Wrocław";

Optimized query:

idxPerAge&Mar&City((30 ,  33 , false ,  true) groupas $range;
     (false union true) groupas $in ; "Wrocław" groupas $equal);
```

The index optimizer is also prepared to deal with queries where selection predicates are joined with **or** operators. As **or** weakens selection it also makes optimization more complex. Therefore if applying the index is possible

without considering predicates joined with an **'or'** operator then the optimizer may skip deeper analysis. In another case, in order to check all possibilities for indexing, the optimizer removes **or** operator and splits non-algebraic **where** operator expression on two partial selection expressions. Objects returned by both these expressions can be duplicated so it is necessary to leave only distinct object references what is achieved using a **unique** expression. Indexing will reduce the amount of data processed in a query only if it can be applied to both partial expressions. This procedure is recursive if there is more than one **or** operator. Let's consider the following example of optimization:

```
Person where age >= 25 and 40 > age and (address.city = "Szczecin" or
    "Szczecin" in worksIn.Firm.address.city);
```

query is split by the index optimizer into the following form:

```
unique(
    (Person where age >= 25 and 40 > age and address.city = "Szczecin")
     union (Person where age >= 25 and 40 > age
      and "Szczecin" in worksIn.Firm.address.city));
```

depending on a current cost model, the optimizer applies indices:

```
unique(
    (idxPerCity("Szczecin" groupas $equal) where age >= 25 and 40 > age)
    union
    (idxPerAge&WorkCity((25, 40, true, false) groupas $range; "Szczecin"));
```

6. Optimization Framework

ODRA administration module contains the optimization framework. The main goal of the optimization framework is to provide means for flexible applying a chain of different optimization methods in an execution of SBQL queries. Moreover it is also able to perform benchmarks and results comparison tests to estimate efficiency of different optimizations (and check different combinations of optimization methods during research). User can set required optimization sequence (including rewrite and indices methods) and run the queries in the normal or test mode. If the system works in test mode the user can receive information about a time of the static evaluation (typechecking), the optimization (using optimization chain) and the runtime execution. It is also possible to run the query in the compare mode where the times of optimized and un-optimized execution are compared. The optimization framework introduces also benchmark functionality to enable multiple executions of comparison tests on the same query and store the results in the form facilitating the further analysis.

7 . Conclusions

In the paper the rules concerning creating and using indices are briefly described. Still, the implementation of indexing for ODRA is not finished and may require a further research. The following features are considered to be implemented for index optimization are: other index structures (e.g. B-Tree, Sorted List) and optimization for other non-algebraic operator: quantifier **forany**. Additionally we consider extending the indexing onto distributed environment with use of SDDS and currently developed volatile indices technique.

References
[1] Burleson D.: *Turbocharge SQL with advanced Oracle9i indexing*, March 26, 2002, http://www.dba-oracle.com/art_9i_indexing.htm.

[2] Elmasri R. and Navathe S. B.: *Fundamentals of Database Systems 4th ed.* Pearson Education, Inc. 2004, ISBN: 83-7361-716-7

[3] Litwin W.: /Linear Hashing : a new tool for file and tables addressing/. Reprinted from VLDB-80 in READINGS IN DATABASES. 2-nd ed. Morgan Kaufmann Publishers, Inc., 1994. Stonebraker , M.(Ed.).

[4] Litwin W., Nejmat M. A., Schneider D. A.: /LH*: Scalable, Distributed Database System. 1996/. ACM Trans. Database Syst., 21(4):480-525.

[5] O'Neil P.E., Quasi D.: *Improved Query Performance with Variant Indexes.* Proceedings of SIGMOD, pp. 38-49, 1997

[6] Płodzień J.: *Optimization Methods In Object Query Languages*, PhD Thesis. IPIPAN, Warszawa 2000

[7] Płodzień J., Subieta K. (alias A.Kraken). *Object Query Optimization in the Stack-Based Approach*. Proc. of 3rd ADBIS Conf., Maribor, Slovenia, 1999, pp.303-316, Springer LNCS 1691

[8] SBA & SBQL Web pages: http://www.sbql.pl/

[9] K.Subieta. *Theory and Construction of Object-Oriented Query Languages* (in Polish), PJIIT - Publishing House, 2004, 522 pages

[10] Oracle9i *Data Warehousing Guide Release 2 (9.2).* Part Number A96520-01
http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96520/indexes.htm#97718