Annales UMCS Informatica AI IX, 1 (2009) 77–97
DOI: 10.2478/v10065-009-0006-z

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Query Optimization by Indexing in the ODRA OODBMS

Tomasz M. Kowalski[1], Michał Chromiak[2], Kamil Kuliberda[1], Jacek Wiślicki[1], Radosław Adamus[1], Kazimierz Subieta[3]

[1] *Technical University of Lodz, Stefanowskiego 18/22, 90-924 Lodz, Poland*
[2] *Institute of Computer Science, Maria Curie Sklodowska University, pl. M. Curie-Skłodowskiej 1, 20–031 Lublin, Poland*
[3] *Polish-Japanese Institute of Information Technology, Koszykowa 86, 02-008 Warsaw, Poland*

Abstract

We present features and samples of use of the index optimizer module which has been implemented and tested in the ODRA prototype system. The ODRA index implementation is based on linear hashing and works in a scope of a standalone database. The solution is adaptable to distributed environments in order to optimally utilize data grid computational resources. The implementation consists of transparent optimization, automatic index updating and management facilities.

1. Introduction

Indices are auxiliary (redundant) data structures stored at a server. A database administrator manages a pool of indices generating a new or removing an existing one depending on the current needs w.r.t. improving overall performance of applications. As indices at the end of a book are used for quick page finding, a database index makes quick retrieving objects (or records) matching given criteria possible. As indices have a relatively small size (comparing to the

whole database) the gain in performance is fully justified by some extra storage space. Due to single aspect searching, 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 the object–oriented databases does not differ from indexing in relational databases [**1**]. Many indexing methods can be adopted from relational database systems and even their applicability can be significantly extended. There are also situations where indexing methods from RDBMSs become outdated in object–oriented databases. In particular, join operations do not require extensive optimizations because in object databases the necessity for joins is much lower due to object identifiers and explicit pointer links.

ODRA (Object Database for Rapid Applications development) is a prototype object-oriented database management system based on the Stack Based Architecture (SBA) [**2, 3**]. The main goal of the ODRA project is to develop new paradigms of database application development and to introduce a new, universal declarative programming language, together with a 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.

An important feature of ODRA concerns the optimization engine responsible for increasing the performance of query execution. The essential component of the engine is the module that optimize queries by using indices. The main features of the indices implementation include: transparent choosing appropriate indices for a given query (if available), automatic update of indices in response to update of corresponding data and administrative management of indices.

The paper presents the above three aspects of indices implementation in ODRA. Section 2 contains a brief overview of selected OODBMSs index capabilities. Section 3 presents overall architecture of the ODRA query optimization engine. Section 4 discusses the features of indices in ODRA. Section 5 describes ODRA index management facilities. Section 6 exemplifies query optimization based on indices. Section 7 presents performance gain of proposed solution based on an example query. Section 8 concludes.

2. Query Optimizations with Indices in OODBMSs

In the case of the Versant ODBMS [**4**] a B–tree index can be used in an exact or range predicate processing. No index inheritance is present in the Versant database. An index can be created on an attribute of only one class. No class

inherit the index. To index subclass attributes, it is necessary to specifically set indices on each subclass. This results in the need for providing index consistency by a database administrator.

In the Objectstore DBMS [**5**] there are two types of improving query performance by indexing, i.e., with *indices* and with *superindices*. The first solution involves building indices on a collection of objects. A superindex is an index kind that is specially used for optimizing queries involving types that have many subtypes. By default, adding an index on a type results in recursive adding of indices to all its subtypes. Still for queries with a large and intricate hierarchy of subtypes the regular indexing can seriously deteriorate processing. Adding a superindex to a type with many subtypes differs from a default index in one essential feature: the superindex is only one. It eliminates a recursion; consequently, only one parent query operation occurs in contrast to multiple queries when using the regular index.

There is also a possibility to create a query that uses a *multistep* index, which is an index on a complex navigational path that accesses multiple public data members. It optimizes queries that use the same path. For example, if a query concerns all employees who works in the *Sales* department, an index on *WorksIn.Name* to *Emp* collection can be used. However, updating an index entry after data modification must be explicitly determined by the programmer. It is a serious drawback of an ObjectStore *multistep* index.

The ObjectStore ODBMS automatically optimizes a query applied to a collection. If an index is added to a collection, then the database first evaluates indexed fields and establishes a preliminary result set. Then, ObjectStore applies non-indexed fields and methods to the elements in the preliminary result set. In ObjectStore the optimization can be done manually by preparing a query or automatically otherwise. This means that a query is optimized to use exactly indices which are available on the collection being queried. The automatic optimization is convenient and effective. Moreover, it supports data independence, i.e., the database administrator is not constrained in establishing new or removing indices because application programs do not refer to them explicitly.

Let us consider the index usage in Objectivity/DB [**6**]. The main goal of an index is to optimize predicate scans and this is how it is implemented in Objectivity. The predicate used in the scan can be one of the following:

- A single optimized condition (=, ==, >, <, >=, <=, =∼ – string match) that tests the first key field of the index
- A conjunction (&&) of conditions in which the first conjunct is an optimized condition that tests the first key fields of the index (no disjunction – OR)

In AI - Informatica Objectivity/DB indexing an index for a class means indexing also objects references of all classes derived from the indexed one. The index structure maintains references to persistent objects of a particular class (so called *indexed class*) and its derived classes. An indexed class is specified during creation of an index. Objectivity/DB additionally supports concatenated index on several attributes (key fields). The order of key values of an index is very relevant regarding the proper activity of predicate.

While considering indexing in OODBMSs the way the GemStone database server handles the issue should also be noticed [**7, 8**]. GemStone indices address path–expressions. A variable name appearing in the beginning of a path is called *path prefix*. Then, a path contains a *sequence of links* and a path suffix; e.g. *Employee.worksIn.manager*. For each link (for an instance variable of an object) in the path suffix one index is available thus forming a sequence of index components. In GemStone identity indices directly support exact match lookups; whereas, equality indices and identity indices on *Boolean, characters* and *integers* directly support =, >, >=, <, <= and range lookups.

3. Query Optimization Engine Architecture

Fig. 1 shows the ODRA query optimization process in the context of a query evaluation process. The input for the optimization process is an abstract syntax tree (AST) of a query. The optimization modules are divided into optimization by rewriting and optimization by indices. The theoretical idea for these methods is presented in several documents, see e.g. [**9, 10, 2, 3**].

The rewriting optimization process modifies a query during compile–time with the 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 a view invocation by a 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 an 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.

4. The idea of ODRA indexing

In general, an index can be considered a two–column table, where the first column consists of unique key values and the other one holds non-key values,

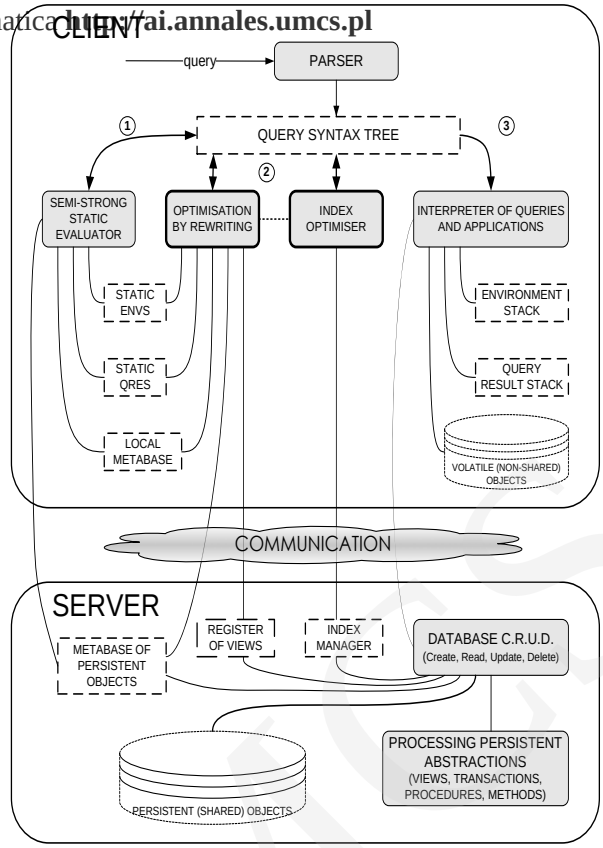


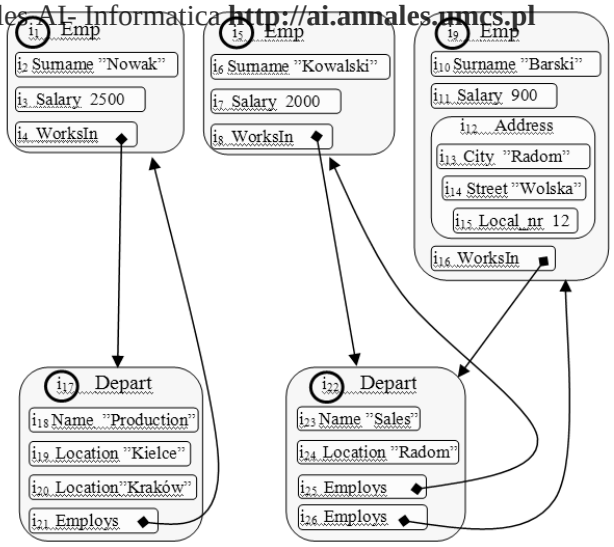
Fig. 1. ODRA optimization architecture

which in most cases are object references. Fig. 2 shows the example indices for a given object–oriented database store.

Key values are used as an input for index search procedures. As a 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).

Key values can be also calculated with the use of expressions that can contain build–in query language functions or user defined functions (*function-based indices* [**11**]). This approach enables the administrator to create an index matching exactly predicates within frequently occurring queries, so their evaluation is faster and uses the minimal amount of I/O operations.

In 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. Let us make an example using the database store structure presented in Fig. 2. If the administrator will set an index named



Index for *Emp* objects according to *Surname* attribute

Search key-value	References
Barski	i_9
Kowalski	i_5
Nowak	i_1

Index for *Depart* objects according to *Location* attribute

Search key-value	References
Kielce	i_{17}
Kraków	i_{17}
Radom	i_{22}

Fig. 2. Example of dense indices for a given object-oriented database store

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

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

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

4.1. Index Transparency.

In the common approach a programmer should not involve explicit operations on indices into an application program. To make indexing transparent from the point of view of a database application programmer, the database management system should ensure two important functionalities *index–based optimisation* and *automatic index updating*.

The first functionality means that indices are used automatically during query evaluation. Therefore, the administrator of a database can freely establish new indices and remove them without changing the codes of applications. The

[illegible] plies in *query optimisation* and particularly in the *index optimiser*.

The second functionality, i.e. an *automatic index updating*, is required due to possible changes in a database. Indices, like all redundant structures, can lose cohesion if a database is updated. An automatic mechanism should improve, eliminate or generate a new index in the case of database updates.

This paper focuses on the first functionality, i.e. index optimisation, which is the main topic of Section 6.

4.2. Index Classification.

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

• a *dense* index is applied when for each value in the object attributes a separate position in an index is created, e.g. for a person objects index, where any name occurring in the database can be a key–value,

• a *range* index means that index items concern values within a given range, e.g. a range index for a salary attribute is a table where each index item describes a range of salaries: $< 0 - 500)$, $< 500 - 1000)$, $< 1000 - 1500)$, ... etc (Table 1). Similarly, range index items for names can take the following form: "names starting with a letter A", "names starting with a letter B",..., "names starting with a letter Z".

Table 1. Example range index for Employees objects according to Salary attribute

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}
...	...	...

Indices can also be categorized according to physical data structures used for index organization. The most important data structures for implementing indices are the following:

- lexicographic,
- indices based on B–tree (a balanced tree),
- bitmap indices.

4.3. Features of ODRA indices.

Currently the implementation supports indices based on Linear Hashing [**12**] which can be easily extended to its distributed version SDDS [**13**] 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 to those in the solutions occurring in relational ones [**11, 1, 14, 15**]: B–Trees, bitmap indices, etc.

An extended idea of an ODRA index works with multiple key indices. Additionally to the key types mentioned earlier (*dense* and *range*) *enum* type was introduced to improve multiple key indexing (among other things). Moreover, thanks to properties of the SBQL language, i.e. orthogonality and compositionality, the implemented solution provides generic support for variety of index definitions including usage of complex expressions with polymorphic methods and aggregate operators.

ODRA supports local indexing which ensures index transparency by providing a mechanism (optimization framework) to automatically utilize an index before query runtime evaluation and therefore to take the advantage of indices. ODRA C.R.U.D. (Create, Read, Update and Delete) is also equipped with triggers to ensure automatic index updating so existing indices are consistent with the database state.

5. 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 a special admin module. Each index is associated with a module where it was created and its name has to be unique. Therefore, the 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 name and an index name: "module$_{\mathrm{n}}$ame.index$_{\mathrm{n}}$ame".

5.1. Example Schema.

The schema in Fig. 3 is introduced to exemplify the usage of indices.

The example schema illustrates personnel records of a company. It introduces several classes *PersonClass, StudentClass, EmpClass, EmpStudentClass* and two structure types *DeptType* and *AddressType*. Persistent instances of the

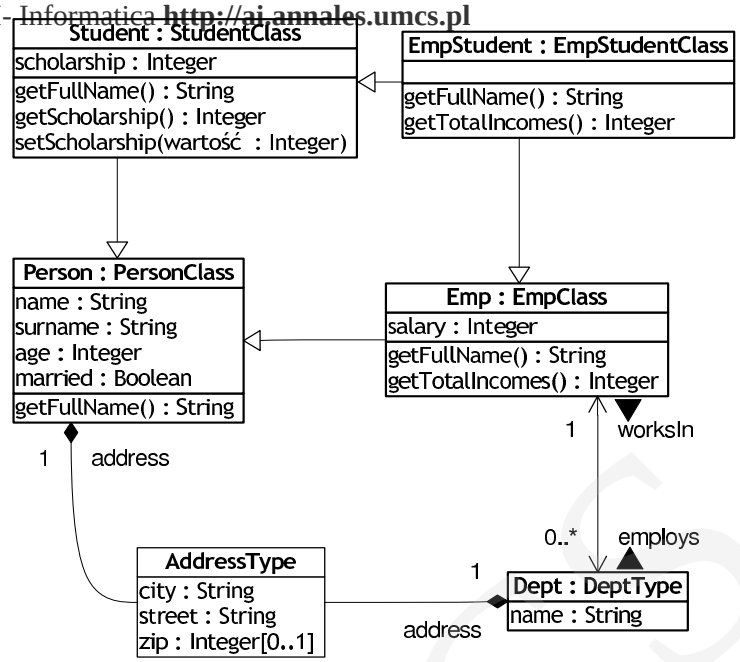


Fig. 3. Example object–oriented schema

classes mentioned above can be accessed using their instance names *Person, Student, Emp* and finally *EmpStudent*. The objects called *Dept* have *DeptType* structure with a primary attribute name and represent departments of the company. Each *Person* object stands for a person somehow connected with the company. Its attributes provide some basic information. Additionally, each *Dept* and *Person* object includes an *address* subobject which specifies data according to the *AddressType* structure. Instances of the *EmpClass* represent current employees of the company and extend *Person* object attributes with the salary attribute. *Emp* and *Dept* objects are associated with pointer objects named *worksIn* and *employs*. Another class, which extends the *PersonClass*, is the *StudentClass*. This class introduces the *scholarship* attribute. The last class presented in the schema is called *EmpStudentClass* and inherits from *EmpClass* and *StudentClass*. It is introduced to represent students who are simultaneously employees of the company. Using *Person* in an SBQL query results in returning all instances of the *PersonClass* class and its subclasses. Similarly, via *Emp* the programmer refers both to *EmpClass* and *EmpStudentClass* instances.

Beside attributes, classes comprise methods. Taking advantage of the polymorphism some methods are overridden in derived subclasses. E.g. *getTotalIncomes*() method of *EmpClass* returns the value of a *salary* attribute, but

for instance of the *Emp* class returns sum of *salary* and *scholarship* attributes.

5.2. 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 *dense* indicator implies that the optimization of queries which use the given key as a condition will be applied 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. The *dense* indicator is always used for reference values (regardless of an indicator set by the administrator). Moreover, it is the default type indicator for *integer, string, double* or reference key values.

```
add index idxEmpSalary(dense) on Emp(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 an index a hash function groups objects according to key value ranges. In the current implementation, ranges are dynamically split because each range is associated with an individual bucket of a linear hash map.

```
add index idxDeptSalary(range) on
   Dept(sum(employs.Emp.salary))
```

The *idxDeptSalary* index returns references to departments according to a value (or a value range) of a sum of department employees salaries. Its advantage is avoiding calculation of a complex selection predicate multiple times because it is already calculated during index creation. On the other hand, the maintenance of the *idxDeptSalary* index is very expensive and can cause serious deterioration during database updating.

The *enum* indicator is introduced in order to take the advantage of keys with a countable limited set of distinct values, i.e. keys with low values cardinality. 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 the 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 supports indexing selection predicates exactly as 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 the 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(age,
   married, address.city)
```

Other examples of creating indices commands are as follows:

```
add index idxPerZip(enum) on Person(address.zip)
```

The *enum* index which returns *Person* objects queried by a *zip* attribute of its subobject *address*. It is important to note that a *zip* attribute is optional and therefore this index stores only *Person* objects containing this attribute.

```
add index idxPerBirthYear(range) on Person(2009 - age)
```

The index returns *Person* objects according to the value of expression 2009 – *age*. It is assumed that this index is capable of processing range queries.

```
add index idxEmpTotalIncomes on Emp(getTotalIncomes())
```

The dense index uses the *Emp* class method *getTotalIncomes*() as a key for selecting *Emp* objects. This method is overridden for instances of the *EmpStudent* class.

The only action required from the administrator in order to take advantage of indexing is creation of proper indices since the rest of optimization is transparent for programmers. The next section describes the rules used by the Index Optimizer.

6. Query Optimization

In ODRA the use of indices is entirely 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 process.

Besides this possibility, we also use indices explicitly. This feature is introduced for testing purposes in order to 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.

6.1. Index Usage Syntax.

From the SBQL syntax point of view an index invocation is simply a procedure invocation:

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

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

A key parameter expression can define a single value as a criterion. In that case its evaluation should return *integer, double, string, reference* or *Boolean* value or reference to such a value. Below we present an example calls for the sample index *idxDeptName*:

```
idxDeptName("HR" groupas $equal)
```

A single value key parameter can be passed through a value of a binder named "$equal". Binders are used to increase readability and to make introducing new types of parameters for index calls easier.

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.

Examples of index calls:

The last example returns references to persons whose year of birth is below the average of all the persons from the database. Like in the case of single value key parameters, parameters specifying a range are passed using the value of a binder named "$range".

```
idxAgeBirthYear((1776, 1988, true, true) groupas $range)
```

```
idxPerBirthYear((1900, (sum(Person.(2009 - age)) /
   count(Person)), true, false) 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.

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

The binder named "$in" is used to pass a collection of key values.

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

6.2. Transparent Index–based 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. The index optimizer analyzing the right operand of a **where** non–algebraic operator takes into consideration all selection predicates joined with conjunction (**and**) or disjunction (**or**) operators.

6.2.1. *Optimization procedure.*

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 **and** operators and tries to find an index that keys matches the predicates. If more than one index is found, the optimizer selects one with the best selectivity.

6.2.2. *Semantic Equivalence Issue in Optimization Involving Optional Keys.*

Firstly, let us 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. An unexpected number of operand

values causes a run–time error. Using an index call in optimization with those predicates would eliminate threat of error and therefore optimized query would not be semantically equivalent to the original. In these cases the optimization is allowed only if **in** operator is used as a predicate because it does not constrain the cardinality of a right operand.

The example of an unsafe predicate evaluation that may cause a run–time error (left side of selection predicate has cardinality [0..1] due to *zip* attribute) is presented below:

```
Person where address.zip = 94107
```

To avoid the possibility of a run–time error the ”safe” **in** operator should be used:

```
Person where 94107 in address.zip
```

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

The example of safe predicate evaluation when ’=’ operator is used can be as follows:

```
Person where exists(address.zip) where address.zip = 94107
```

Only in the case of two previous examples of queries the Index Optimizer can apply the following query transformation:

```
idxPerZip(94107 groupas $equal)
```

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

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

6.2.4. *Aspects of Range Predicates Optimization.*

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 a given key. For example, the following query concerns the departments located in the Warsaw city whose employees together earn less than the best paid employee of the whole company.

Original query:

```
Dept where sum(employs.Emp.salary) < max(Emp.salary) and
   address.city = "Warszawa"
```

Optimized query:

```
idxDeptSalary((-2147483648, max(Emp.salary), true, false)
   groupas $range) where address.city = "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.(2009 - age)) / count(Person)) as avgyear).
  (Person where 2009 - age > avgyear and 1970 <= 2009 -
  age and 2009 - age < 1980)
```

Optimized query:

```
(sum(Person.(2009 - age)) / count(Person)) as avgyear).
  idxPerBirthYear((max(avgyear union 1970), 1980, 1970 >
  avgyear, false) groupas $range)
```

In some cases, the index optimizer can use **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 is to be checked e.g. when for a given key there exists more than 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 returns an empty bag:

Original query:

```
((sum(Person.(2009 - age)) / count(Person)) as avgyear).
  (Person where 2009 - age >= avgyear and 1977 = 2009 -
  age)
```

Optimized query:

```
(sum(Person.(2009 - age)) / count(Person)) as avgyear).if
  (1977 >= avgyear) then idxPerBirthYear(1977 groupas
  $equal)
```

This procedure is used also when the key cardinality is different from [1..1], i.e. in the case of two or more selection predicates based on **in** operator.

6.2.6. *Omitting Individual Index Keys.*

For multiple keys indices, *enum* keys may be usually omitted in an index call. The index optimizer, in order to omit a key when no selection predicates were specified, sets both lower and upper bounds to the smallest and largest key values: 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 the *Boolean* key in an index call, set key parameter criteria are used (false union true). Original query:

Optimized query:

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

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

6.2.7. *Predicates Disjunction and Considering Inheritance.*

The index optimizer is also prepared to deal with queries where selection predicates are joined with **or** operators. As disjunction weakens a selection, it also makes optimization more complex. Therefore if the application of an index is possible without considering predicates joined with **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. The objects returned by both these expressions can be duplicated so it is necessary to leave only distinct object references which is achieved using a **uniqueref** expression. Indexing reduces 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 us consider the following example of optimization:

```
Emp where age = 28 and married = true and (address.city =
   "Szczecin" or "Szczecin" in worksIn.Firm.address.city)
```

The query can be split by the index optimizer into the following form:

```
uniqueref((Emp where age = 28 and married = true and
   address.city = "Szczecin")
   union (Emp where age = 28 and married = true and
   "Szczecin" in worksIn.Firm.address.city))
```

and depending on a current cost model and existing indices, the optimizer can apply the transformation:

```
uniqueref((
   (Emp) idxPerAge&Mar&City(28 groupas $equal; true groupas
   $equal; "Szczecin" groupas $equal))
   union (idxEmpAge&WorkCity(28 groupas $equal; "Szczecin"
   groupas $equal) where married = true)
```

Selection predicates based on *age*, *married* and *address.city* expressions concern *EmpClass*'s superclass, i.e. *PersonClass*, and for that reason the administrator can equip the whole *Person* collection with the *idxPerAge&Mar&City* index. It can return instances that do not belong to *EmpClass* thus the optimizer has to introduce a facility removing non–*EmpClass* instances from the index invocation result. This can be done using an SBQL coerce operator. The syntax of the coerce operator was taken from the typical syntactic convention that is known from the languages such as C, C++, Java, etc. as cast. Consequently, the result of the *idxPerAge&Mar&City* index call is automatically cast to *Emp* collection because the original query concerns only employees.

In the presented approach to reusing an index in inheritance, indices considering the class which introduces the given key are more versatile as they can be used for optimising selection queries addressing subclasses collections.

7. Optimization Gain

Let us discuss the following test example. If an index call is located on the right side of a non–algebraic operator, e.g. a dot, then it is likely to be evaluated more then once during the query execution. This is shown using the following example with an *idxEmpTotalIncomes* index:

Query 1a. For 61 year old, married employees living in Łódź, working in Łódź or Wrocław retrieves a name concatenated with a surname and a number of employees with an equal amount of total incomes

reference	((*Emp* **where** *address*.*city* = "Łódź" **and** *worksIn*.*Dept*.*address*.*city* in ("Łódź" **union** "Wrocław") **and** *married* = **true** **and** *age* = 61) **as** *e*). (*e*.*name* + " " + *e*.*surname*, **count**(*Emp* **where** *getTotalIncomes*() = *e*.*getTotalIncomes*()))
index optimised	((*Emp* **where** *address*.*city* = "Łódź" **and** *worksIn*.*Dept*.*address*.*city* in ("Łódź" **union** "Wrocław") **and** *married* = **true** **and** *age* = 61) **as** *e*). (*e*.*name* + " " + *e*.*surname*, **count**(*idxEmpTotalIncomes*(*e*.*getTotalIncomes*()))

In Figs 4 and 5 the logarithmic scale is used also on the y–axis. The dependency between the optimization gain and the number of persons is close to linear and grows to 457 for 300000 objects.

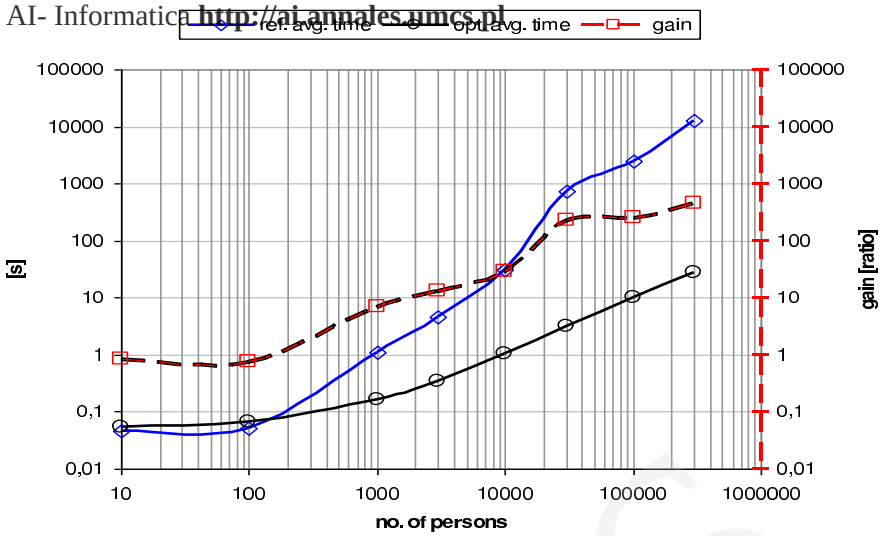


Fig. 4. Evaluation times and optimization gain for Query 1a

Additionally, introducing another index – *idxEmpAge&WorkCity* – in order to optimise evaluation of the first part of the query can significantly influence the performance:

Query 1b.

reference	((*Emp* **where** *address.city* = "Łódź" **and** *worksIn.Dept.address.city* in ("Łódź" **union** "Wrocław") **and** *married* = **true and** *age* = 61) **as** *e*). (*e.name* + " " + *e.surname*, **count**(*Emp* **where** *getTotalIncomes*() = *e.getTotalIncomes*()))
index optimised	((*Emp* **where** *address.city* = "Łódź" **and** *worksIn.Dept.address.city* in ("Łódź" **union** "Wrocław") **and** *married* = **true and** *age* = 61) **as** *e*). (*e.name* + " " + *e.surname*, **count**(*idxEmpTotalIncomes*(*e.getTotalIncomes*()))

For a database consisting of 300000 person objects two indices give the gain approximately 40 times greater. Despite such difference, the most important is an index repeatedly invoked, i.e. *idxEmpTotalIncomes*. Without this index the performance is not noticeably improved.

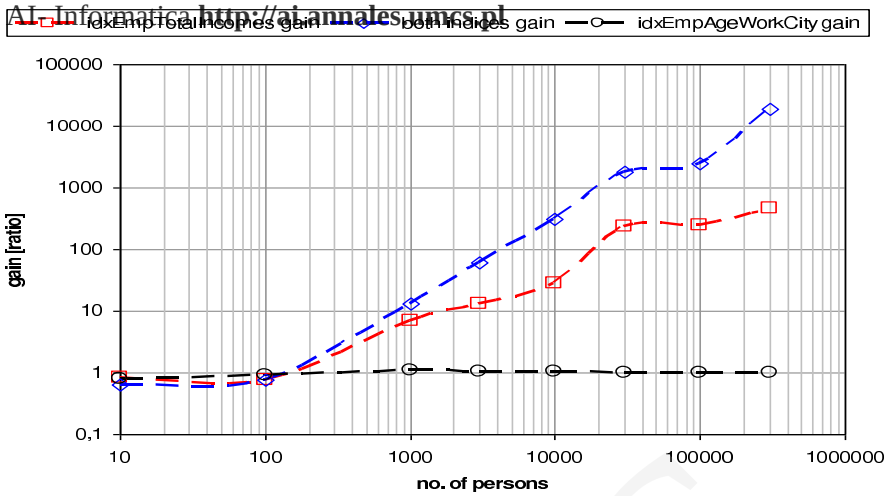


Fig. 5. Indices optimization gain for Query 1

8. Conclusion and Future Work

In the paper the rules concerning creating and taking advantage of indices in the ODRA prototype have been briefly described. In the presented approach the optimization is achieved through the described query transformation. The proposed implementation of indexing in ODRA enables creation and transparent automatic maintenance of indices facilitating processing of selection predicates based on arbitrary deterministic expressions consisting of path expressions, aggregate functions, class method invocations (taking into consideration inheritance and polymorphism). All functionalities necessary to provide the desired behaviour of indices are already implemented and functional. Still, the ODRA indexing is under development and requires further research. Future works include employing different index structures (e.g. B-Trees) and implementing new optimization methods taking advantage of indices (e.g. optimisation of rank queries). Additionally we consider extending indexing capabilities onto distributed environment using the SDDS method and currently developed volatile indexing technique.

References

[1] Elmasri R. and Navathe S. B., *Fundamentals of Database Systems 4th ed.*, Pearson Education, Inc. 2004, ISBN: 83–7361–716–7.
[2] SBA & SBQL Web pages: http://www.sbql.pl/
[3] Subieta K., *Theory and Construction of Object-Oriented Query Languages* (in Polish), PJIIT - Publishing House, 2004, 522.
[4] VERSANT Database Fundamentals Manual, (Release 7.0.1.0) July 2005.

[5] Java 2 Platform, Standard Edition 5.0 for all platforms, August 2008.
[6] Objectivity for Java Programmer's Guide, Release 9.3, October 13, 2006.
[7] GemStone Systems, Inc. www.gemstone.com
[8] Meier D., Stein J., *Indexing in an object-oriented DBMS*, Proceedings of the OODBS, IEEE Computer Society Press (1986) 171.
[9] Płodzień J., *Optimization Methods In Object Query Languages*, PhD Thesis. IPIPAN, Warszawa 2000.
[10] Płodzień J., Kraken A., *Object Query Optimization in the Stack-Based Approach*, Proc. of 3rd ADBIS Conf., Maribor, Slovenia, 1999, 303, Springer LNCS 1691.
[11] Burleson D., *Turbocharge SQL with advanced Oracle9i indexing*, March 26, 2002, http://www.dba–oracle.com/art_9i_indexing.htm
[12] 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.).
[13] Litwin W., Nejmat M. A., Schneider D. A., *LH*: Scalable, Distributed Database System*, ACM Trans. Database Syst. 21(4) (1996) 480.
[14] O'Neil P.E., Quasi D., *Improved Query Performance with Variant Indexes*, Proceedings of SIGMOD (1997) 38.
[15] Oracle9i Data Warehousing Guide Release 2 (9.2). Part Number A96520-01.