Automatic Selection of Functional Indexes for Object Relational Mapping System

Aleksandra Boniewicz, Michał Gawarkiewicz and Piotr Wiśniewski

Faculty of Mathematics and Computer Science Nicolaus Copernicus University
Toruń, Poland

grusia, garfi, pikonrad@mat.uni.torun.pl

Abstract

Functional indexes are an important tool for database tuning, often not appreciated by programmers and system administrators. In practice, a programmer can only advise the deployment team such such indexes. Only a few very expensive systems have tools for the analysis of performance and automatic creation of functional indexes. The authors decided to provide the ability to define functional indexes at the stage of application development by supplying the appropriate API in object-relational mapping level, and implemented it as an extension for Hibernate. The interfaces proposed by the authors were developed with the functional indexes generator for selected DBMS: PostgreSQL, Oracle and IBM DB2. In addition, to facilitate the work of programmers, HQL query analysis algorithms detecting the usefulness of functional indexes in queries have been developed.

1. Introduction

When using business applications, one of the important part of tco is a widely understood database, both in terms of hardware and database management system. Thus, a very large attention is given to good design of the database schema and database tuning techniques. One of the important techniques for tuning the database are different types of indexes, which when appropriately chosen, could greatly speed up the application, especially if they relate to the frequently used queries.

Functional indexes are a class of indexes which is often ignored among designers and developers. The programmer has no tools for defining such indexes. Definition of the functional indexes is the responsibility of a database administrator. On the other hand the administrator using the available tools analyzing execution trace in most cases will not be informed of this possibility. These tools with very few exceptions, cannot detect the benefits of the use of functional indexes.

Functional indexes are a class of indexes which is often ignored among designers and developers. The programmer has no tools for defining such indexes. Definition of the functional indexes is the responsibility of a database administrator. On the other hand the administrator using the available tools analyzing execution trace in most cases will not be informed of this possibility. These tools with very few exceptions, cannot detect the benefits of the use of functional indexes.

When building database-driven applications the level of independence between the developed source code and the actually used database management system is an important problem. One of the possible solution is to use object-relational mappings as an intermediate layer. The essence of this idea starts with a single declaration of correspondence between objects and records in tables in abstract model, which is independent of the database system. Then, based on these declarations, databases queries in appropriate SQL dialect are automatically generated. This approach relieves the programmer from the knowledge of the nuances of SQL language's variants. As a result, an intensive increase in the usage of the object-relational mappings being observed, both in industry and research projects. Thus, such systems seem to be an interesting research field. One of the most popular object-relational mapping systems is Hibernate, which is an open source framework [1].
Developers of object-relational mapping systems are focused mainly on the basic issues of data persistence. Whereas relational database management systems are full of proposals processing. The use of them in projects based on ORMs ruins the fundamental architecture of project proposed by the object relational mappings. Some of these extensions would require the use of direct SQL code, such as recursive queries, while others would require additional work on the implementation, such as setting functional indexes.

In response to this problem, the authors of this paper decided to fill systematically the emerging gap, by extending the selected object-relational mapping systems with the features supplied by the leading relational database systems: Oracle, IBM DB2 and an open source system PostgreSQL. The first aspect analyzed and implemented by the authors are recursive queries. The results of the work on their integration with popular ORMs were presented at several conferences and published in [2-4, 9]. Another contribution to the Hibernate system is the support for analytical query optimization based on partial aggregation approach [5]. In this paper the authors focus on functional indexes, an important tool for database performance tuning.

2. Contributions

In this paper the authors present a developed mechanism that integrate the generation of the functional indexes in databases with object-relational mapping systems. The main result is the HQL query analysis algorithm that detects the expressions suitable for functional-indexing. Another aspect presented in this paper is the interface that defines functional indexes as annotations for persistent classes. The interface, and the index generation algorithm are developed for Oracle, IBM DB2 and PostgreSQL.

3. Motivating Examples

Let us consider some of the classic cases where it is worth to use the functional indexes technique. These examples are based on a simple database schema shown in the Figure 1.

**Example 1.** The table invoice contains columns total_value and total_tax. The total_tax column is dependent on total_value column but not directly, because products can have the various tax rates. Consider the following query:

```
SELECT *
FROM invoice
ORDER BY (total_value + total_tax) DESC
```
Above query with index on \((total\_value + total\_tax)\) expression runs from 10 times faster on 100,000 records set up to 40 times faster on 1,000,000 records set. Table 1 shows test results.

**Table 1. Query Invoices Sorted Descending by the Value with Tax**

<table>
<thead>
<tr>
<th>number of records</th>
<th>query without index</th>
<th>query with index</th>
<th>RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000 000 rec.</td>
<td>416 ms</td>
<td>11 ms</td>
<td>2.64 %</td>
</tr>
<tr>
<td>500 000 rec.</td>
<td>237 ms</td>
<td>10 ms</td>
<td>4.21 %</td>
</tr>
<tr>
<td>250 000 rec.</td>
<td>136 ms</td>
<td>9 ms</td>
<td>6.61 %</td>
</tr>
<tr>
<td>100 000 rec.</td>
<td>77 ms</td>
<td>18 ms</td>
<td>10.4 %</td>
</tr>
</tbody>
</table>

*Example 2*. The next problem is searching for large binary objects. Instead of comparing large binary objects the system can compare the values of a hash function on the objects, such as the MD5 function. If there are several objects with the same hash function value then the system compares those objects. An example of an SQL query that compares hash function values of objects:

```sql
SELECT id
FROM Product
WHERE MD5(picture) = MD5(????)
```

Index on the values of MD5 function on the column containing large objects significantly accelerates such searches as is presented in Table 2.

**Table 2. Checking if a Picture Exists in Database**

<table>
<thead>
<tr>
<th>number of records</th>
<th>Hibernate without index</th>
<th>Hibernate with index</th>
<th>RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 000 rec.</td>
<td>342 227 ms</td>
<td>38 ms</td>
<td>0.01 %</td>
</tr>
<tr>
<td>5 000 rec.</td>
<td>20 299 ms</td>
<td>19 ms</td>
<td>0.1 %</td>
</tr>
<tr>
<td>2 500 rec.</td>
<td>9 490 ms</td>
<td>18 ms</td>
<td>0.18 %</td>
</tr>
<tr>
<td>1 000 rec.</td>
<td>4 348 ms</td>
<td>15 ms</td>
<td>0.35 %</td>
</tr>
</tbody>
</table>

Those tests were taken on the desktop class computer. Their goal was not to exactly test performance gain experienced by the functional index, but only to test the order of magnitude of it.

The above examples show that the usage of functional indexes provides significant system acceleration. However, the business application designers do not have tools to create such indexes. Functional indexes are created on the application deployment stage, or at the stage of tuning. The developer can in the documentation only indicate necessity to create appropriate functional indexes. Any transfer of a solution to a different database requires to manually recreate indexes. This solution is an activity likely to cause errors, and consequently increases the cost of software maintenance.

Another important aspect is whether programmer will notice the usefulness of functional indexes to use in these examples. Only a few systems offer automatic detection of the need for functional indexes. Lack of common tools for automatic analysis of queries in this regard leads to unnecessary system overloading.
4. Analysis of the HQL Queries

One of the most popular systems of Object Relational Mappings for Java language is the Hibernate. This framework supports most of both commercial and open sourced major database management systems in the market. Retrieving objects from a database can be done in Hibernate in a few ways, but in the authors opinion the most interesting possibilities of retrieving objects gives the HQL language which has syntax similar to SQL. Example 3 shows sample query using HQL.

Example 3. Queries in HQL corresponding to queries from Examples 1 and 2 are as follows: for Example 1

From Invoice ORDER BY (totalValue + totalTax) DESC

for Example 2

SELECT id from Product WHERE MD5(picture) = MD5(:bin)

Notice, that the first of the above queries has no SELECT clause. Hibernate as a result returns objects or their references, hence the SELECT clause is not justified. In the WHERE clause we use the class field names. More about querying for objects the reader can find in [1].

There are relational database systems equipped with mechanisms to propose functional indexes [6-8]. In most cases implementation rests with the database administrator. In this paper the authors propose a different approach to the subject. The authors offer the programmer of the persistence layer the HQL query analysis tool early in the development phase of the project. Developer is notified if an analysis indicates the usability of the functional indexes. If they want to use the proposed functional indexes, annotations are generated automatically for persistent classes, which then will take care of creating appropriate indexes in the database when connected to it. Annotations are the subject of the next section.

In this section we will present the algorithm that analyzes the HQL language query tree. The present version of the implementation of this algorithm is limited only to the queries which do not contain any subqueries. The first step is to create a query tree using the mechanisms implemented in Hibernate. The tree for the first HQL query presented in the Example 3 is shown on the Figure 2.
Our algorithm will be presented using this query as an example. The second step is searching a node FROM in order to gather the table names with theirs aliases generated by Hibernate. This information is stored in a temporary list. In this example there is only one table called invoice with the alias called invoice0_.

The next step is looking for a subtree in which there expressions occur potentially suitable for indexing. These are the subtrees with roots WHERE, ORDER BY, GROUP BY. In this example there is only one interesting subtree - ORDER BY. For all those subtrees (in our example only one subtree) we are looking for a node of one of these types: METHOD_CALL, PLUS, MINUS, DIV, MOD, CONCAT. That is because almost all database functions tha occur in a query are placed in node of METHOD_CALL type. The exceptions are the operations +, - , /, %, ||. They are placed in nodes of types PLUS, MINUS, DIV, MOD, CONCAT. If we reached the right subtree (ie. The METHOD_CALL type) then we initiate the search for a node of the METHOD_NAME type. In our example there is no METHOD_CALL type but there is PLUS type, which may substitute the node type METHOD_NAME. The value for this node is the name of the applied database function, which in this example is ‘+’. If we consider Example 2, there will be a node METHOD_CALL and the node METHOD_NAME. The value of this node will be md5.

Also there is the variable walkMethodTable set to the NULL value. With the use of this variable we are able to mark, while performing depth-first recursive search, whether the analyzed function operates on columns of one or more database tables. If we come across many tables then an error is returned and index will not be proposed. Since functions can be nested, after finding the node of type METHOD_NAME we initiate the recursive depth-first search through the tree looking for the next functions. During the depth-first search the tree of the whole function is converted into the SQL form. After the recursion ends, if there was at least one column, we create temporary column, its table is walkMethodTable and the name is this SQL form. Such designed column is added to the list of other indexable columns. Nodes PLUS, MINUS, DIV, MOD, CONCAT have slightly different structures, but they are processed the same way. In the example for the given query will be offered an index on an expression (totalValue + totalTax).

If the developer approves the validity of the proposed index, the class invoice will automatically be decorated with appropriate annotations. Annotation's interface for defining of the functional indexes will be presented in the next section.

5. Interfaces Functional Indexes in Hibernate

In this section we will present an interface to define the functional indexes. This interface is used by the algorithm discussed in the previous section to set up proposed indexes.

Let us consider Example 1.

```
SELECT *
FROM invoice
ORDER BY (total_value + total_tax) DESC
```

A fast execution of this query requires an index on the (total_value + total_tax) expression. As it was mentioned earlier, Hibernate does not provide any mechanisms for defining such indexes, prompting the authors to develop the interface and generators of the functional indexes.

The interface is nothing more than @FunctionalIndex annotations supplied with a number of parameters to the persistent class. Example 4 shows the usage of it in the situation discussed above.
Example 4 Adding a functional index to Invoice class.

```java
@Entity
@Table(name = "invoice")
@FunctionalIndex(expr="(totalValue + totalTax)")
public class Invoice implements Serializable {
    ...
}
```

During synchronization of metadata with the database, particularly when creating a table in the database, the system after creating the table builds an index on it with an automatically generated command, which in this case for the PostgreSQL system would look like the following:

```sql
CREATE INDEX idx_invoice ON invoice ((total_value+total_tax));
```

An another possible parameter is `dbfunction="fname(p1, p2, ...)"` where `fname` is a function. The `dbfunction` parameter indicates a deterministic function in the database, which takes as parameters the indicated fields. Hibernate has no possibility to check whether the function is actually deterministic, and hence, if such an index makes sense. If one tries to create an index based on a non-deterministic function, the DBMS rejects the request with an error. The presented algorithm will detect this error and throw an exception indicating that the functional index was not created. The system will behave similarly when working on a database that does not support this type of index.

Use of the parameter `dbfunction` is shown in Example 2.

Example 5. In the case considered in Example 2 we need an index on the values of functions `md5(...)` on picture column of `product` table:

```java
@Entity
@Table(name = "product")
@FunctionalIndex(dbfunction="md5(picture)")
public class Product implements Serializable {
    ...
}
```

In the presented `FunctionalIndex` annotation exactly one of the parameters `expr`, `dbfunction` should appear.

In case when `dbfunction` parameter is used, it is possible to use additional parameters `returnType`, `bodyFunction`, `language` and `ifExists`. These parameters allow for creating own functions in the database. The function will be generated and stored in the database. The parameters `returnType` and `bodyFunction` should occur together and the parameters `language` and `ifExists` are optional extras for `returnType` and `bodyFunction` parameters.

Currently the parameters that create functions are only supported with the PostgreSQL database. The authors plan to extend their work to ORACLE and IBM DB2 database management systems.

6. Conclusions and Future Work

The proposed solution enables a programmer to define the functional indexes in the schema design process with an elegant and intuitive form. The interfaces presented in the paper respect to the architecture of a project developed with an object relational mapping
Moreover the presented algorithm proposes such indexes independent on database management system. As the consequence the advisor for PostgreSQL is received. Another research subject is to suggest multicoloumn indexes in an automatic way. Algorithm for this mechanism will use some special statistics gathered from workload of HQL queries without using a database query processor.

Next, the authors plan to continue the research on providing support for another features implemented in database management systems at the level of Object Relational Mapping. One of the research subjects is support for logical constraints that would allow for automatic generation of DBMS specific triggers.

References
