An Efficient Approach to Similarity-Based Retrieval on Top of Relational Databases

Jürgen Schumacher[1] and Ralph Bergmann[2]

[1] tec:inno GmbH
Sauerwiesen 2, D-67661 Kaiserslautern, Germany
schumacher@tecinno.com
[2] Center for Learning Systems and Applications (LSA)
University of Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
bergmann@informatik.uni-kl.de

Abstract This paper presents an approach to realize a case retrieval engine on top of a relational database. In a nutshell the core idea is to approximate a similarity-based retrieval with SQL-queries. The approach avoids duplicating the case data or compiling index structures and is therefore ideal for huge case bases which are often subject to changes. The presented approach is fully implemented as part of the commercial CBR toolbox orenge and the experimental evaluation demonstrates impressive retrieval behaviour.

1 Introduction

Case-Based Reasoning (CBR) has become a very important technique for realizing intelligent product recommendation agents for Electronic-Commerce applications [3,8,9]. The core of such applications is a product database that describes the specific features of each available product. When applying CBR, this product database is treated as a case base, i.e., each product record in the database is interpreted as a case in the case base. During the retrieval phase of the CBR cycle, product cases are retrieved based on the similarity to the query that describes the product requirements that are important for the customer. The retrieved products can then be presented to the customer or they may be further customized, e.g., by applying adaptation techniques from CBR [6].

One core problem when developing such E-Commerce applications is to enable the case-based recommendation agent to get *efficient*, *up-to-date*, and *consistent* access to the product database. The typical application scenario is as follows:

- The products are stored in an existing (relational) database that is independent from the recommendation agent. This database is maintained by the supplier of the products and is also used for many other services related to the products.

– The number of products in the database is very huge.
– There are rapid and continuous changes in the product database, in particular when products are individual items (e.g., used cars) that become unavailable when they are sold.
– The representation of products is quite simple, typically a list of attribute values.

To link the product database with the case-based retrieval component, different architectural variants are possible [4].

Retrieval Inside the Database. From a purely technical point of view, the ideal approach would be to integrate the similarity-based retrieval function into the database itself. This would enable efficient data access and would ensure an up-to-date and consistent view on the data. In such an ideal scenario, the database would be able to automatically process a similarity-based query formulated in a standardized query format enabling flexible representation of similarity measures. In the recent research literature on databases some very interesting results can be found on indexing structures for similarity-based retrieval [2,1,7]. Unfortunately, these techniques have not yet been introduced into commercially available standard databases. There also exists no standardized language for a similarity-based query formulation that is widely accepted by the database community. Due to the large number of different databases from different vendors that could possibly hold product data, it is currently not feasible to realize the case-based retrieval inside the database in the near future.

Retrieval on Top of the Database. Since from a pragmatic point of view the integration of the retrieval function into the database is not feasible at the moment, the second architectural variant must be applied as a medium-term solution. This variant means to realize the similarity-based retrieval on top of the database. The retrieval engine becomes a separate module that interfaces with the database to get access to the product data. Here again, different variants are possible.

– *Bulk-loading all products into the retrieval engine:* This approach replicates the product database inside the retrieval engine. The shortcomings of this approach are that the required storage space is doubled and that it is necessary to update the duplicated case base every time when the product database is updated. Hence, major consistency problems arise.
– *Constructing and storing an index structure for similarity-based retrieval in the retrieval engine:* Here, the problem of duplicating the product data is avoided, but the consistency problem remains.
– *Approximating similarity-based retrieval with SQL-queries:* This third option is a MAC/FAC-like approach. Like proposed in [5], for every query a sequence of SQL queries is constructed to fetch cases from the database which are then used for a detailed similarity assessment. This approach avoids the

consistency problem and always relies on the up-to-date data. The challenge is to develop an approach that maintains efficiency.

In the remainder of this paper, we analyze in detail the problem of approximating similarity-based retrieval by SQL-queries and we presents a new retrieval algorithm which generalizes the results presented by [5]. The presented approach is implemented in Java as a retrieval service for tec:inno's orenge-framework and will be in use at the time of the workshop. We will give some experimental results of performance measurements in section 4.

2 Foundations

We assume that the cases and queries are represented as lists of attribute-value pairs $(A_1 = a_1, \ldots, A_n = a_n)$. The allowed range or *type* T_i of attribute A_i is defined either by a subset of (natural or real) numbers or by a finite set of symbolic values. The value of an attribute in a case must be one element of the type or the special value undefined: $a_i \in T_i \cup \{\text{undefined}\}$.

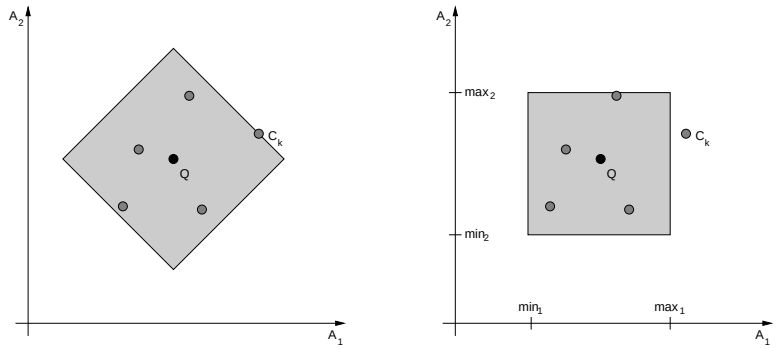
Such cases can be easily represented by one table in a relational database: Each attribute corresponds to one column of the table, one row of the table completely describes a single case. An attribute is undefined if the corresponding value in the database table is null. The relation between the case structure and the database tables is called a *mapping*. In the following we will identify the attributes with the table columns and denote the columns also by A_i.

For such cases it is relatively easy to compute a similarity between query and case: For each attribute we can define a local similarity measure which describes the similarity between different attribute values. For numerical attributes this is a monotonous function on the distance of the values,[1] for a symbolic attribute it is defined by a table containing the similarity values. We assume further that the local similarity is 1 if the query attribute is undefined, which expresses a "don't care"-semantic, and 0 if only the case attribute is undefined, which expresses a penalty for the case which lacks a feature the user asked for. The global similarity between two objects can be calculated by accumulating the local similarities of each attribute. In the following we assume that a weighted average of the local similarities is computed, but the considerations also apply to other similar approaches.

Besides a query object, a query can contain additional information for the retrieval: first it can contain *user weights* for each attribute to express which attributes are more or less important for the customer, second the customer can specify *filters* which define hard constraints for the attribute values of the case. Finally, it contains the number of cases to retrieve, let this number be k.

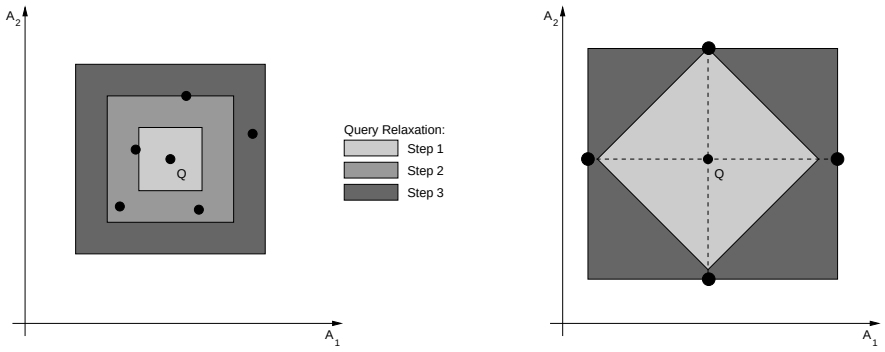
First we consider a query without weights and filters, also we assume all attribute types to be numeric or at least totally ordered. Then we can view our case base as an n-dimensional euclidean space in which each object is represented by a point, if all attributes are defined. The k most similar cases then lie within

[1] The greater the distance, the smaller is the similarity of two values



(a) The Similarity Rhombus (b) The SQL Rectangle

Figure1. Volumes covered by queries



(a) Retrieval Rings (b) Included Rhombus

Figure2. Retrieval with Rings of SQL-Rectangles

a kind of "hyper-rhombus" centered on the query point Q, where the size of the rhombus is determined by the least similar of the retrieved cases (see figure 1(a)). If we view in the same way a SQL-query of the form

$$\begin{aligned} &\text{SELECT } a_1, \ldots, a_n \text{ FROM CaseTable} \\ &\text{WHERE } (a_{i_1} \geq min_{i_1} \text{ AND } a_{i_1} \leq max_{i_1}) \ \ldots \\ &\quad \text{AND } (a_{i_m} \geq min_{i_m} \text{ AND } a_{i_m} \leq max_{i_m}) \end{aligned} \tag{1}$$

where $a_{i_1}, \ldots, a_{i_m}$ are those attributes in the query which are not undefined, we get a "hyper-rectangle" (see figure 1(b)). The goal is now to use the rectangular SQL-queries to retrieve the cases inside the similarity-rhombus from the database.

Our solution is to construct a series of rectangular "rings" around the query point as shown in figure 2(a). In step 0 the retrieval starts with the query point

```
ALGORITHM DBRetrieve
  INPUT :  Object query, Integer k
  OUTPUT : List of Object result[1...k]

  BEGIN
    step = 0
    WHILE (NotFinished(query, result)) DO
      sqlQuery = RelaxQuery(query, step, k)
      dbResult = ExecuteSQLQuery(sqlQuery)
      newCases = SortCases(query, dbResult)
      merge(result, newCases)
      step = step + 1
  END
```

Figure3. Basic Retrieval Algorithm

itself and the cases from the next ring are retrieved as long as more cases are needed to determine the retrieval result.[2] This process is called *query relaxation*. The cases read from the database are ordered according to their similarity to the query, and finally the k most similar cases are returned. So our technique can be described by the algorithm shown in figure 3.

The function `NotFinished(query, result)` has to implement a suitable criterion to decide if the retrieval can be finished. One criterion could be that the retrieval should always yield a complete result list, i.e., there are no other cases in the case base which are more similar to the query than those in the result list. The similarity rhombus already covered by the SQL-queries is defined by the most similar object lying exactly on the borders of the rectangle (see figure 2(b)). If the least similar case in the result list is more similar to the query than this object then we know that we will not find more similar cases by relaxing the query any further. However, in practice this criterion can easily mean that a huge part of the case base has to be searched, because the object in the corners of the SQL rectangle are less similar to the query the more attributes the query has. Also, for many applications completeness of the search is not that important. Therefore, an alternative criterion is to stop the retrieval if k results have been found. The tests have shown, that this is sufficient to yield a good result if we make sure that the SQL-queries cover significantly more than k cases.

The speed of the relaxation is crucial to the performance of the retrieval: If we relax the query too fast, we risk to retrieve too many cases from the database which have to be imported in the search engine and evaluated. This would cost both much main memory and the ordering of the cases would take long. If the relaxation is too slow, we would have to do a lot of SQL-queries. While this may be optimal for memory use and evaluation, it would be slowed

[2] Note that the algorithm can also be used as an "any-time"-algorithm, where the user can examine a intermediate result and trigger another step, until the result is sufficient.

down by the overhead involved with each database query. Often the database runs on another machine than the search engine, and so this would create a lot of network traffic. Therefore we want to retrieve the necessary cases with only a few database queries.

For the following considerations we view the space of cases as a n-dimensional hyper-cube of volume 1, i.e., all edges of the cube have the length 1. The distance between two values x and y of the same type is defined as $d(x, y) = 1 - sim_i(x, y)$ where sim_i is the local similarity measure for this type. Further we assume that the cases are distributed equally in this space. The goal of the following consideration is to find parameters for the first step of the query relaxation so that in this step already enough cases are retrieved from the database to build an acceptable query result. For this we introduce the parameter N_{max}, which is the maximum number of cases we want to read from the database in a single step. It clearly has to be a function of k because we want the k most similar cases to be included in this number of cases, therefore we write it as a function $N_{max}(k)$.

To generate the bounds of the database query we now have to determine which *volume* the query should cover in order to have a result of $N_{max}(k)$ cases. For a SQL-query this volume can be computed as:

$$V = \prod_{j=1}^{m} (1 - sim_{i_j}(min_{i_j}, max_{i_j})) \tag{2}$$

If the query object is $Q = (q_1, \ldots, q_n)$ and we select the bounds of the SQL-query so that the speed of the relaxation is equal for all attributes, we get:

$$\begin{aligned} \forall_{j=1}^{m} : sim_g = sim_{i_j}(q_{i_j}, min_{i_j}) = sim_{i_j}(q_{i_j}, max_{i_j}) \\ \Rightarrow V = (2 \cdot (1 - sim_g))^m \end{aligned} \tag{3}$$

We call sim_g the *minimal global similarity* because all cases covered by the rectangle are at least this similar to the query. Because we have assumed the cases to be distributed equally we can describe the volume of one case as $V_C = 1/N$ where N is the number of cases in the case base, which leads to:

$$V = V_C \cdot N_{max}(k) \Rightarrow sim_g = 1 - \frac{1}{2} \left(\frac{N_{max}(k)}{N} \right)^{\frac{1}{m}} \tag{4}$$

In this way the minimum global similarity for the first relaxation step $sim_g^1 = sim_g$ depends only on the number of defined attributes in the query, the requested size of the result set and the number of cases in the database.

To construct the SQL-query, the local similarity measures have to be *invertible*, i.e., is must be possible to determine the following set:

$$SIM_j(q_j, sim_g) = \{x_j \in T_j \mid sim_j(q_j, x_j) \geq sim_g\} \tag{5}$$

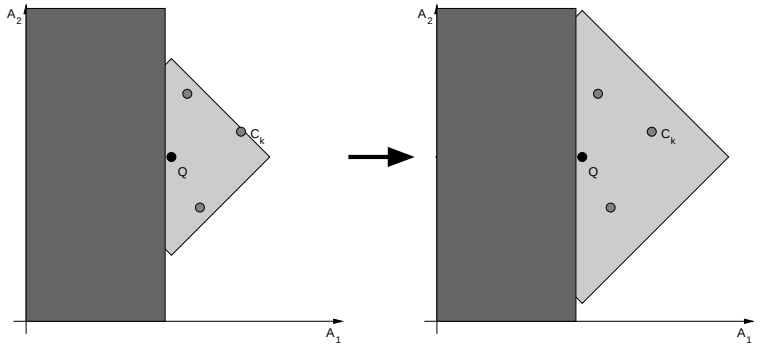


Figure4. Effect of a Filter

This set contains all attribute values for which the local similarity to the query is greater than the given minimum global similarity. If the attribute type is totally ordered and the similarity measure is monotonous with respect to the distance of the values, we can also express this set as an interval $SIM_j(q_j, sim_g) = [min_j, max_j]$. Both forms of the set can be used easily to construct a SQL-query; the WHERE-Clause of the query in step s we denote with C_s.

For further steps s we can compute sim_g^s the same way by using a linear growth of the query volume, but this way sim_g^s would change very slowly for higher s, which could mean that no new cases are covered by the query. To prevent this and to ensure that eventually all cases of the case base can be reached, we set $sim_g^s = 1 - s \cdot (1 - sim_g)$.

Finally, we have to modify the SQL queries so that in each step not the complete rectangle but only the current ring is read from the database. This can be done by excluding the rectangle of the step before in the query:

SELECT attributes FROM table WHERE C_s AND NOT (C_{s-1})

3 Advanced Features

Earlier we stated that the similarity of a case to the query is computed as the weighted average of the local similarities of the attributes. Additionally a query can define *user weights* that modify the predefined weights of the attributes, and *filters* that exclude cases with certain attribute values. This all significantly changes the form of the similarity rhombus and therefore has to be taken into account when constructing the database queries.

A filter just cuts off a part of the solution space like shown in figure 4. While in general the effect on the rhombus can be quite complex, we have found that in practical applications often a very special filter type is used: the "more"- or "less"-filter. It specifies that for all resultant cases a certain attribute should

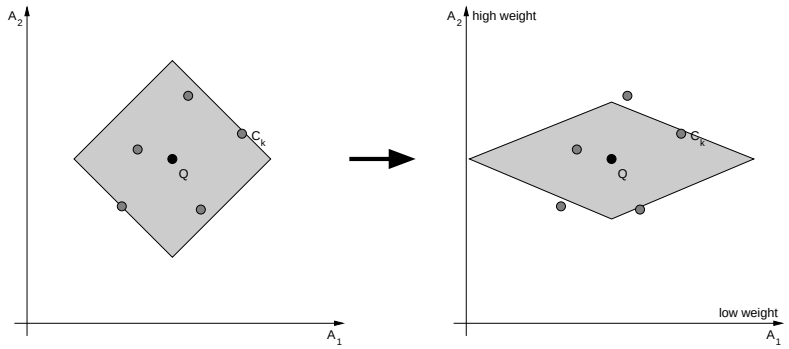


Figure5. Effect of Weights

have a value greater or smaller than the one defined in the query. This kind of filter is easy to consider: It cuts off exactly one half of the similarity rhombus. We can easily exclude the filtered cases in the SQL-query so that we do not have to read cases which cannot be valid solutions. But this will have the consequence that we get only about half of the number of cases we wanted to be read. To compensate for this, we have to double the query volume for every attribute which is constrained by a filter.

To consider weights, we denote the complete weight of attribute A_j with w_j, where the default weight is 1. Different weights for different attributes change the form of the similarity rhombus like shown in figure 5: An attribute with a higher weight has to be matched more closely than one with a lower weight. Therefore, we have to change the form of the SQL rectangle in a similar way. The goal is that the similarity of a case in the corner of the rectangle is the minimum global similarity sim_g. This can be done by computing for each attribute A_j an own *minimum local similarity* $sim_{l,j}$, which leads to the following equation:

$$sim_l, j = 1 + \frac{\sum_j w_j \cdot (sim_l - 1)}{w_j \cdot m} \tag{6}$$

A last problem concerns *underspecified queries*. These are queries which have many exact matches in the database. With the naive approach "load all cases from the database selected by the query" this leads to huge amounts of data to be read from the database. To avoid this, we implemented the import from the database in a way that only $N_{max}(k)$ cases are imported from the database result set. Should it be necessary to import more cases later, the same result set can be used to read the remaining cases of this query without having to perform a new database query. Of course in general this makes the result almost certainly incomplete, but as it occurs in most cases only with exact matches, this is not a serious problem.

4 Evaluation

In this section we want to present some results of experiments which we conducted to examine the performance of our retrieval approach.

The software used for the tests implements all features described in section 3. The first version stops after at least k cases have been loaded from the database, so it is possible that the retrieval results of this engine are incomplete. A second version of the retrieval engine therefore does further relaxation steps until the completeness of the retrieval result can be ensured. In the following, we will call this version the "complete DB-retrieval". We compared these two engines to the "linear retrieval" which holds all cases in the main memory without a connection to the database and performs the retrieval by computing the similarity of all cases and selecting the k best.

For the function $N_{max}(k)$ used during the query relaxation in the DB retrieval, we chose the following definition:

$$N_{max} = \max\left(N_{min}, N_{min} + N_{incr} \cdot \log_2 \frac{k}{N_{R,min}}\right) \tag{7}$$

It is motivated by the idea that for small k we want to read relatively more cases from the database in order to ensure that the best k cases are actually found, while for bigger k the completeness of the result is not as important as the performance. We got good results with the settings $N_{R,min} = 10$, $N_{min} = 100$, and $N_{incr} = 50$.

The case base used for the tests contains 8622 cases having 3 symbolic and 4 numeric attributes each. The retrieval engines are implemented in Java and compiled to native code using the IBM HPJ compiler. They work as TCP/IP-servers to which queries can be sent over the network. The queries as well as the results are written in XML. All timings include the sending of queries and receiving of results. As the data base management system we used a Microsoft SQL Server 7.0.

In the first experiment we compared the retrieval times of the different engines for queries with different numbers of defined attribute values. In all tests $k = 10$ cases were retrieved. We started with two attributes and extended the query in each step by another attribute. Table 1 shows the results of this experiment. Besides the times needed for the retrieval the table also gives the number of SQL-queries executed by the DB-retrieval engines. Finally we determined the "α-error" of the first version of the DB-retrieval, which is the number of most similar cases *not* found. This gives a measure of the incompleteness of the DB-retrieval.

We see that the linear retrieval takes up to 40 times as long as the DB-retrieval in this experiment. Maybe more surprising is that in all tests the first version of the DB-retrieval-engine produced a complete result, i.e., the result list contained the 10 most similar cases or at least cases which are equally similar to the ones returned by the linear retrieval. Further, we see that in the tests

No. Attr.	Linear Retr. (ms)	DB-Retr. (ms)	Steps	α-Error	Complete DB-Retr. (ms)	Steps
2	6701	175.9 ± 49.1	2	0	150.0 ± 33.8	2
3	7059	187.3 ± 46.1	2	0	190.0 ± 51.5	2
4	7405	205.9 ± 39.3	2	0	191.0 ± 36.5	2
5	7656	233.9 ± 45.1	2	0	401.0 ± 27.4	3
6	8067	288.3 ± 49.2	3	0	450.0 ± 44.0	4
7	8290	306.1 ± 42.0	2	0	301.0 ± 62.4	3

Table1. Retrieval times with respect to number of attributes defined in the query.

with 5, 6, and 7 defined attributes the complete DB-retrieval needed to do one more step to ensure the completeness of the result though no further results were found in this extra step.

From this experiment we find that our retrieval technique has a very good performance. Further we can conclude that, at least for small k, the possible incompleteness of the DB-retrieval-approach is not critical: Usually the most similar cases are found.

We wanted to verify this conclusion with a further test in which we used the same query but changed the requested result set size k. The query in this experiment has 6 defined attribute values, and k is increased from 10 in three steps to 30, 50, and 70. The results are shown in table 2.

k	Linear Retr. (ms)	DB-Retr. (ms)	Steps	α-Error	Complete DB-Retr. (ms)	Steps
10	8106	168.9 ± 41.1	2	0	150.0 ± 46.2	2
30	8155	261.0 ± 71.1	2	0	290.0 ± 44.4	3
50	8458	281.4 ± 51.5	3	4	380.0 ± 33.9	4
70	8719	368.9 ± 55.5	4	22	20740 ± 808	5

Table2. Retrieval times with respect to the result set size.

We see that for small result set sizes like 10 or 30 we get again complete retrieval results from the first version of the DB-retrieval engine. Also in both cases the performance is similar to the results of the first experiment. But for $k = 50$ we see that there is a incompleteness in the DB-retriever's result: 4 cases of the 50 most similar ones are not found. The complete DB-retriever avoids this by doing one more retrieval step. In the final test with $k = 70$ the incompleteness is even worse, 22 of the most similar cases are not found. Again the complete retriever tries to prevent this with an additional relaxation step, but in this cases this leads to loading all cases into the main memory, and therefore the performance of the complete version of the DB-retrieval is very bad.

As a conclusion from this experiment we see that with increasing result set sizes the DB-retrieval can easily produce incomplete results. We also see that while it is possible to prevent this by doing additional relaxation steps this can have the effect of a very bad performance, so usually one will be content with the incomplete result which is retrieved in very short time, because in many domains the performance will be more critical than the completeness. In favour of the "incomplete" version of the DB-retrieval we can give the fact, that in the tests that produced incomplete results, the similarity of the found cases was only slightly less than that of the cases in the complete result. So while the found cases were not the perfect ones, the still were very good in terms of similarity.

Also, the DB-retrieval is quite efficient in terms of memory consumption: While the linear retriever used about 75 MB of memory, the DB-retrieval needed less than 30 MB.

5 Discussion and Future Work

Finally we want to give a short discussions of the benefits and shortcomings of our approach. We recognize the following advantages:

- *Efficient Retrieval:* The evaluation indicates that a good retrieval performance can be expected even on very huge case bases.
- *No Consistency Problem:* Each query is answered by using all cases in the database without having to rebuild any index structures or loading cases into the retrieval engine. Cases are available immediately after they are stored in the database. This makes this technique optimal for applications with case bases that change very often.
- *Independence of the Database System:* Any standard SQL database implementation can be used for this technique. No database-specific programming has to be done inside or outside the database.
- *Use of Existing Databases:* If the tables are in the required form they can be used directly for retrieval. Otherwise one can apply a database view to provide the data in a suitable form, but usually nothing has to be changed in the database schema.

On the other hand, there are some limitations that have to be worked on. First, the retrieval technique presented so far is limited to attribute-value representations. Extensions towards representing structured cases are necessary. As a first step in this direction a feature is under development and evaluation using n-to-m-relations in the database for representing attributes holding sets of values.

Second, the current model is also based on the idealistic assumption of a uniform distribution of the cases on the representation space. If this assumption is violated the efficiency of the retrieval will decrease because either too many cases will be retrieved from the database or too many SQL queries are created. We expect that this problem can be avoided by incorporating an introspective learning approach that estimates the density of the case base in different regions.

Finally, more evaluation needs to be done in order to learn about the behaviour of the retrieval for different sizes of case bases or different parameters for the functions used in the retrieval. By the time of the workshop we also will be able to report lessons learned from a commercial application in which this retrieval engine is used.

Acknowledgements

The authors wish to thank the staff of tec:inno for valuable comments during the development of the database retrieval engine.

Funding for this research has been partially provided by the Commission of the European Communities (ESPRIT contract EP 27.068, the WEBSELL project - Intelligent Sales Assistants for the World Wide Web). The partners of WEBSELL are tec:inno (prime contractor, Germany), IMS (Ireland), IWT Magazin Verlags GmbH (Germany), Adwired AG (Switzerland), Trinity College Dublin (Ireland) and the University of Kaiserslautern (Germany).

References

1. S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional space. In *Proc. 14th Int. Conf. on Data Engineering (ICDE'98)*, pages 209–218, 1998.
2. S. Berchtold, D. A. Keim, and H.-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proc. 22nd Int. Conf. on Very Large Data Bases*, pages 28–39, 1996.
3. R. Bergmann, S. Breen, M. Göker, M. Manago, and S. Wess. *Developing Industrial Case-Based Reasoning Applications: The INRECA Methodology.* Lecture Notes in Artificial Intelligence, State-of-the-Art-Survey, LNAI 1612. Springer Verlag, 1999.
4. M. Donner and T. Roth-Berghofer. Architectures for integration CBR-systems with databases for e-commerce. In *7th German Workshop on Case Based Reasoning (GWCBR'99)*, 1999.
5. H. Schimazu, H. Kitano, and A. Shibata. Retrieving cases from relational databases: Another strike toward coporate-wide case-based systems. In *Proceedings of the 13th International Joint Conference in Artificial Intelligence (IJCAI'93)*, 1993.
6. S. Schmitt and R. Bergmann. Applying case-based reasoning technology for product selection and customization in electronic commerce environments. In *12th Bled Electronic Commerce Conference.*, 1999.
7. T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 154–165, 1998.
8. W. Wilke. *Knowledge Management for Intelligent Sales Support in Electronic Commerce.* DISKI 213. Infix Verlag, 1999.
9. W. Wilke, M. Lenz, and S. Wess. Intelligent sales support with CBR. In M. Lenz, B. Bartsch-Spörl, H.-D. Burkhard, and S. Wess, editors, *Case-Based Reasoning Technology: From Foundations to Applications*, pages 91–113. Springer Verlag, 1998.