A Framework for Recommending OLAP Queries

Arnaud Giacometti, Patrick Marcel, Elsa Negre
Laboratoire d'Informatique
Université François Rabelais de Tours - France
{arnaud.giacometti,patrick.marcel,elsa.negre}@univ-tours.fr

ABSTRACT

An OLAP analysis session can be defined as an interactive session during which a user launches queries to navigate within a cube. Very often choosing which part of the cube to navigate further, and thus designing the forthcoming query, is a difficult task. In this paper, we propose to use what the OLAP users did during their former exploration of the cube as a basis for recommending OLAP queries to the user. We present a generic framework that allows to recommend OLAP queries based on the OLAP server query log. This framework is generic in the sense that changing its parameters changes the way the recommendations are computed. We show how to use this framework for recommending simple MDX queries and we provide some experimental results to validate our approach.

Categories and Subject Descriptors

H.2.7 [**Database Administration**]: Data warehouse and repository; H.3.3 [**Information Search and Retrieval**]: Query formulation

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

OLAP analysis, MDX queries, recommendation

1. INTRODUCTION

Traditional OLAP users interactively navigate a cube by launching a sequence of queries over a datawarehouse, what we call an analysis session (or session for short) in the following. This process is often tedious since the user may have no idea of what the forthcoming query should be [16]. The problem we address in this paper is thus the following: How to help the user to design her forthcoming query?

As an answer, we propose to exploit what the other users did in their former navigation on the cube, and to use this information as a basis for recommending what the forthcoming query could be. To this end, we present a generic framework for recommending OLAP queries, that uses both the log of the server, i.e., the set of former sessions on the cube, and the sequence of queries of the current session.

The framework relies on the following process:

1. Partitioning the log to group queries that are similar, in order to cope with the sparsity of the log.
2. Generating candidate recommendations by first finding which sessions of the log match the current session and then predicting what the forthcoming query can be.
3. Ranking the candidate queries so as to present to the user the most relevant queries first.

This framework is generic in the sense that it does not impose a specific way of partitioning the log, generating candidate queries or ranking the candidates. Instead these actions are left as parameters of the framework, that can be instantiated in various ways in order to change the way the recommendations are computed. The rationale behind the proposal of a generic framework is that recommedations highly depend on the users and the data on which recommendations are computed. For instance, [2] reports the need for flexible, user-adaptive recommender systems.

Our contribution includes the presentation of various instantiations of the proposed framework to recommend MDX queries [11]. For one of the instantiations, we present the results of several experiments that we have conducted to assess the efficiency of recommendation generation and the quality of the recommendations.

The paper is organized as follows: Section 2 presents related work. Section 3 motivates our approach with a simple example. Section 4 presents the formal definitions. Section 5 introduces our generic framework and Section 6 illustrates its use to generate basic recommendations. Section 7 presents our experimental results. We discuss future work in Section 8.

2. RELATED WORK

To the best of our knowledge, this is the first work dealing with the problem of recommending OLAP (especially MDX) queries. The idea of using what the other users did to generate recommendations is very popular in Information Retrieval [2], and Web Usage Mining [17]. For example, [3] uses a k-means algorithm to cluster queries submitted to

DOLAP'08, October 30, 2008, Napa Valley, California, USA.

a search engine, generate candidate recommendations and rank the candidates. Our contribution is to adapt these existing techniques to OLAP.

In the area of OLAP, the work of [14, 15] shares with our work the goal of predicting the forthcoming OLAP query. Nevertheless, there exists many differences between their works and ours: First the main concern of [14, 15] is to prefetch data, not to recommend a query. Second, [14, 15] does not deal with MDX queries, and the way queries are grouped into classes only relies on the schema of the query (i.e., dimensions and levels) whereas the distance that we use takes the members into account. Finally, a Markov Model is used to predict the forthcoming query, whereas we chose not to use a probabilistic model.

3. MOTIVATING EXAMPLE

In this section we illustrate with a simple example the generic idea under our framework. Consider an OLAP server used by several users. Each user can open a session on the server to navigate the cube by launching a sequence of queries expressed in the MDX language. The server logs these sessions, i.e., the sequences of queries launched during each session. For instance, suppose the log is composed of the following three sessions: $s_1 = \langle q_1, q_2, q_3, q_4\rangle, s_2 = \langle q_5, q_6, q_7\rangle, s_3 = \langle q_8, q_9, q_{10}\rangle$ where the q_is are OLAP queries.

Suppose now a new session, called the *current session*, is opened by a user. Say for instance this session is: $\langle q_{11}, q_{12}\rangle$. He could be interested in how the other users explored the cube and use this information to design its forthcoming query.

Now, given the number of users and the number of sessions, the log can be very large, albeit not as large as the cube itself. In addition, the various users may have different interests, thus their queries may navigate very different parts of the cube. It means that it may be unlikely to find a given query more than once in the log. Thus the log can be both large and sparse. In order to cope with largeness and sparsity, the queries in the log can be grouped into classes, so as to *partition* the log. A class of queries being a set of queries that are closed enough to one another. And then, in the sessions of the log, queries are replaced with the class they belong to, what we call a *generalized session*. In our running example, suppose the generalized sessions are: $g_1 = \langle c_1, c_2, c_3, c_4\rangle, g_2 = \langle c_2, c_3, c_5\rangle, g_3 = \langle c_4, c_3, c_5\rangle$ since for example q_2, q_5 belong to class c_2, q_3, q_6, q_9 belong to class c_3 and so on.

Of course, in the current session itself the queries can be replaced with the class they belong to. In our example, suppose the *generalized current session* is $g = \langle c_2, c_3\rangle$, since it is found that q_{11} belongs to c_2 and q_{12} belongs to c_3.

Now, it can be found that the current session is *matching* with sessions s_1 and s_2 of the log (since in our case g is a subsequence of g_1, and g_2). The user might be interested in knowing the queries that follow the matched sequence in those sessions. Looking at the generalized sessions g_1 and g_2, we can *predict* that such queries belong to classes $c_4 = \{q_4, q_8\}$ and $c_5 = \{q_7, q_{10}\}$, called the *candidate classes*. As queries and not classes should be proposed to the user, a single query must be extracted from each candidate class. It can be for instance the query that *represents* the class the most. In our example, suppose these queries are q_4 and q_7. They are called the *candidate queries*. Finally, the user may be interested in only one query, not many. Or at-least the candidate queries should be presented in a given order. Thus the candidate queries must be ordered, for instance by *ranking* them based on their proximity with the last query of the current session. In our example, suppose the ranking is $\langle q_7, q_4\rangle$, and thus the first query that is recommended to the user is q_7.

The generic framework we propose in Section 5 allows to go beyond this simple example. Indeed computing the log partitioning, session matching, prediction of candidate classes, class representative and ranking are left as parameters. Various instantiations of the framework can be imagined as it is illustrated in Section 6.

4. BASIC DEFINITIONS

In this section we give the basic definitions underlying our framework. Let R be a relation instance of schema $sch(R) = \{A_1, \ldots, A_N\}$. We denote by $adom(A_i)$ the active domain of every attribute $A_i \in sch(R)$.

Cubes and Dimensions.

An N-dimensional cube C is a tuple $C = \langle D_1, \ldots, D_N, F\rangle$ where:

- For $i \in [1, N]$, D_i is a dimension table of schema $sch(D_i) = \{L_i^0, \ldots, L_i^{d_i}\}$. For every dimension $i \in [1, N]$, each attribute L_i^j describes a level of a hierarchy, j being the depth of this level. L_i^0 is the lowest level which equals the primary key of D_i.

- F is a fact table of schema $sch(F) = \{L_1^0, \ldots, L_N^0, m\}$ where m is a measure attribute.

In the following, note that the name of a dimension $D_i, i \in [1, N]$ is also used to denote an attribute of active domain $adom(D_i) = \bigcup_{j=0}^{d_i} adom(L_i^j)$. For every $i \in [1, N]$, $adom(D_i)$ is the set of all members of dimension D_i.

Also note that, in this paper, the cube definition is made without loss of generality w.r.t. the problem we consider because we do not use the way queries are evaluated but simply how queries are expressed.

Cell Reference.

Given an N-dimensional cube C, a cell reference (reference for short) is an N-tuple $\langle r_1, \ldots, r_N\rangle$ where $r_i \in adom(D_i)$ for all $i \in [1, N]$.

Given a cube C, we denote by $ref(C)$ the set of all references of C.

Distance between references.

Given a cube C, a distance between cell references in $ref(C)$ is a function from $ref(C) \times ref(C)$ to the set of real numbers.

Query.

In this paper we consider simple MDX queries, viewed as set of references, as defined in [5].

Let $C = \langle D_1, \ldots, D_N, F\rangle$ be an N-dimensional cube and $R_i \subseteq adom(D_i)$ be a set of members of dimension D_i for all $i \in [1, N]$. A query over an N-dimensional cube C is the set of references $R_1 \times \ldots \times R_N$.

Given a cube C, we denote by $query(C)$ the set of possible queries over C.

Distance between queries.
Given a cube C, a distance between queries in $query(C)$ is a function from $query(C) \times query(C)$ to the set of real numbers.

User Session.
Given a cube C, a user session $s = \langle q_1, \ldots, q_p \rangle$ over C is a finite sequence of queries of $query(C)$. We denote by $query(s)$ the set of queries of a session s, by $session(C)$ the set of all sessions over a cube C and by $s[i]$ the i^{th} query of the session s.

Database Log.
Given a cube C, a database log (log for short) is a finite set of sessions. We denote by $query(\mathscr{L})$ the set of queries of a log $\mathscr{L}$.

Class of queries.
Given a cube C, a class of queries is a set $Q \subseteq query(C)$.

Class representative.
Given a cube C, a class representative is a function from $2^{query(C)}$ to $query(C)$.

Query set partitioning.
Given a cube C and a distance between queries, a query set partitioning is a function p from $2^{query(C)}$ to $2^{2^{query(C)}}$ such that, for all $Q \subset query(C)$, p computes a partition of Q under the form of a set P of pairwise disjoint classes of queries.

Query classifier.
Given a cube C a query classifier cl is a function from $query(C) \times 2^{2^{query(C)}}$ to $2^{query(C)}$ such that if $q \in query(C)$ is a query, $P \subseteq 2^{query(C)}$ is a set of classes then $cl(q, P) \in P$. We say that $cl(q, P)$ is the class of q.

Generalized session.
Given a session s and a set of classes of queries, the generalized session of s is the sequence of classes of each query of s in turn.

Formally, given a cube C, a set of classes of queries P, a query classifier cl and $s = \langle q_1, \ldots, q_p \rangle$ a session over C, the generalized session gs of s is the sequence $\langle c_1, \ldots, c_p \rangle$ where:

- $c_i = cl(q_i, P)$ is the class of q_i for all $i \in [1, p]$.
- $\forall i, c_i \in P$.
- $\forall i, q_i \in query(C)$.

We denote by by $gs[i]$ the i^{th} class of the generalized session gs.

Query ranking.
Given a cube C and a set of queries $S \in 2^{query(C)}$, a query ranking $rank$ is a function from $2^{query(C)}$ to a tuple of queries, such that $rank(S)$ orders the queries of S.

5. THE GENERIC FRAMEWORK

In this section we detail the generic framework for recommending OLAP queries. The framework uses both the sequence of queries of the current session, and the query log of an OLAP server, i.e., the sequences of queries formerly launched on the cube. It consists of the three following steps, as illustrated in Figure 1 which takes back the example presented in Section 3:

1. The first step consists in using a query set partitioning to partition the query log in order to compute all the generalized sessions of the log.

2. The second step consists in using the generalized current session and the set of generalized sessions of the log to predict candidate recommendations.

3. The last step consists in ranking the candidate recommendations.

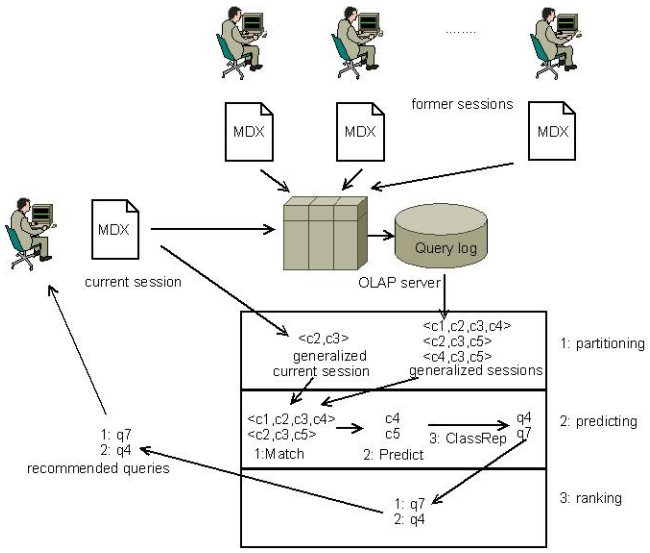

Figure 1: Overview of the generic framework

Each of these steps is parameterized with one of more functions. By changing these parameters, the way recommendations are computed changes (this is illustrated in Section 6). We now present these steps into more details.

5.1 Partitioning the log and computing the generalized sessions

This step consists in partitioning the log. To this end, this step uses a query set partitioning. This partitioning uses a distance between queries to determine the partitions. It outputs a set of sets of queries, that we call a set of classes of queries in the following.

Once the log is partitioned into a set of classes of queries, the set of generalized sessions is computed. The principle is straightforward: For each session of the log, replace in the session each query with the class it belongs to. The generalized current session can be computed as well, but in a different manner since the queries of this session are unlikely to belong to the classes computed by the partitioning. Thus a query classifier is needed to find for each query of the current session its class.

An example of partitioning is introduced Section 6.1.

5.2 Computing candidate recommendations

The previous step has computed the set of generalized sessions. Now, using this set and the sequence of queries of the current session, a set of candidate recommendations is computed by Function 1. The principle of the algorithm is the following. In addition to the set of generalized sessions and the current session, the algorithm uses three functions, Match, Predict, ClassRep. Match is used to find a set of generalized sessions matching a given generalized session (Part 2 : predicting, Left : Match of Figure 1). Predict is used to compute, for a set of generalized sessions, a set of candidate classes (Part 2 : predicting, Middle : Predict of Figure 1). ClassRep is used to obtain the query that represents a class (Part 2 : predicting, Right : ClassRep of Figure 1). The algorithm uses these functions in the following way. First, the generalized current session is obtained (Line 1 of Function 1). Then, Match is used to search among the set of generalized sessions which ones are matching the generalized current session (Line 2 of Function 1). This function outputs a set of pairs indicating which generalized sessions match the generalized current session and the position of the matching. From those pairs, Predict extracts a set of classes that will be the basis for the recommendation (Line 3 of Function 1). And then for each class extracted, ClassRep is used to obtain the query representing the class (Line 5-7 of Function 1). This set of queries is returned as an answer (Line 10 of Function 1). Note that it can be empty.

Examples of functions Match, Predict and ClassRep are introduced Section 6.2.

Function 1 GenerateCandidateQueries (S_c, $\mathscr{S}_{GS}$, Match, Predict, ClassRep)

Require:
S_c: The current session
$\mathscr{S}_{GS}$: The set of generalized session, as output by the previous step
Match: A function generating candidate classes
Predict: A function predicting recommendation classes from a set of classes
ClassRep: A class representative

Ensure: a set of candidate queries

```
1: Let GS be the generalized session of S_c
2: M ← Match(GS,S_GS)
3: C ← Predict(M)
4: Q ← ∅
5: if C ≠ ∅ then
6:    for each c ∈ C do
7:       Q ← Q ∪ ClassRep(c)
8:    end for
9: end if
10: return Q
```

5.3 Ranking the candidate recommendations

In the previous step, a set of recommendations is computed. The purpose of this next step is to select the most suitable one w.r.t. a satisfaction criterion expressed by the user. To this end, a query ranking is needed, that orders the candidate recommendations.

Examples of satisfaction criteria are introduced Section 6.3.

6. INSTANTIATIONS OF THE FRAMEWORK

In this section, we introduce different instantiations of our framework to illustrate the applicability of our generic algorithms.

6.1 Computing generalized sessions

6.1.1 Distance between queries

In this paper, we consider only simple MDX queries as defined in Section 4, i.e., sets of references. To compute the distance between such sets, the Hausdorff distance [7, 10] can be used because it allows essentially to compare two sets. This distance relies on the computation of distance between the elements of the sets, references in our case. To compute a distance between references the classical Hamming distance [6] can be used because of its simplicity, of its ease to be understood and computed. This distance d_h is defined by $d_h(\langle a_1, ..., a_N\rangle, \langle b_1, ..., b_N\rangle) = \sum_{i=1}^{N} compare(a_i, b_i)$ where $compare(a_i, b_i) = 1$ if $a_i = b_i$, 0 otherwise.

The Hausdorff Distance d_{H} between two sets q_1 and q_2 is defined by: $d_{\mathrm{H}}(q_1, q_2) = \max\{ \max_{r_1 \in q_1} \min_{r_2 \in q_2} d(r_1, r_2), \max_{r_2 \in q_2} \min_{r_1 \in q_1} d(r_1, r_2) \}$

In our case, q_1 and q_2 are queries, i.e., sets of references, and the distance d used is the Hamming distance d_h.

For example, consider a 3-dimensional cube and two queries q_1 and q_2 over this cube, such that $q_1 = \{\langle a, 1, x\rangle, \langle a, 1, y\rangle\}$ and $q_2 = \{\langle a, 1, y\rangle, \langle b, 1, y\rangle\}$. It is left to the reader to verify that the Hausdorff distance between q_1 and q_2 equals 1.

6.1.2 Query partitioning and query classifier

Partitioning the set of queries can be done by using a simple clustering algorithm like K-medoids [8]. In that case, the query classifier can associate the query with the class for which the class representative is the closest to the query.

6.2 Computing Candidate recommendations

6.2.1 Match

A first example of the Match function simply consists in looking in the set of generalized sessions for the class of the current query (i.e., the last query of the current session).

A second more sophisticated example of the Match function that we consider in this paper is borrowed from the area of Approximate String Matching [12]. Approximate String Matching is the problem of finding in a text where a given string occurs allowing a limited number of errors of matching. In our case, generalized sessions can be viewed as sequences of elements just as strings are considered as sequences of characters. Then, our problem of finding in a set of generalized sessions where the current generalized session, or a slightly modification of it occurs can be translated to the problem of approximate string matching, for which classical algorithms can be used. More precisely, these algorithms take as inputs a text t, a string p, a set of operations Op and a threshold th and return a set of pairs $\langle p, c\rangle$ where p is a position and c is a number. The output indicates the positions in the text where the string matches the text and for each position, the number of transformations using operations in Op that are necessary for transforming the string to obtain the matching. For instance if $approximateStringMatching$ is such a function then $approximateStringMatching($"$hello\ word$","$world$", $\{removeOneLetter\}, 1)$ would output $\langle 7, 1\rangle$ since removing one letter of "world" would cause the function to detect a matching at position 7 in "hello word". In our context, the

text is one of the generalized sessions, the string is the generalized current session, the set of operations is a set of operations allowed to transform a generalized session.

Function 2 presents a Match function based on approximate string matching, for a fixed set of operations and a fixed threshold (Line 1-2). It iterates until at-least one matching is found or the cost of transforming the generalized current session exceeds the threshold (Line 5). At each iteration step, the set of generalized sessions is scanned (Line 6) and the approximateStringMatching algorithm is used to determine if by using *cost* operations of Op on the generalized current session, matchings can be found (Line 7). If matchings can be found, then the output of the algorithm will be the set of pairs $\langle g, i\rangle$ where g is the generalized session where a matching is found and i is the position of the matching (Line 10). Otherwise, one more transformation on the generalized current session is permitted (Line 12).

Function 2 Match (GS, $\mathscr{S}_{GS}$)

Require:
GS: The generalized current session,
$\mathscr{S}_{GS}$: The set of generalized sessions.
Ensure: A set of pairs $\langle g, i\rangle$ where g is a generalized session and i is an integer

```
1: Let Op be a set of operations on generalized session
2: Let t be an integer threshold
3: cost ← 0
4: Cand ← ∅
5: while Cand = ∅ and cost ≤ t do
6:    for each g ∈ S_GS do
7:       S=approximateStringMatching(GS,g,Op,cost)
8:    end for
9:    if S ≠ ∅ then
10:      Cand ← {⟨g, i⟩|⟨i, c⟩ ∈ S}
11:   else
12:      cost ← cost + 1
13:   end if
14: end while
15: return Cand
```

Obviously, this Match function can be instantiated in many ways, depending on what threshold or operations are used. For instance, the set of operations can be reduced to one operation, namely the removal of the first element of the string. In our case, it simply means that we try to match the generalized current session with every generalized session and if no matches are found then we remove the first class of the generalized current session and we iterate. As an illustration, consider the generalized sessions given in Section 3. Suppose now that the current generalized session is $g = \langle c_1, c_4, c_3\rangle$. The call to $approximateStringMatching(g_3, g, \{removeFirstElement\}, 1)$ outputs the pair $\langle 1, 1\rangle$.

Now as another example of the Match function, the set of operations can be extended. In the area of approximate string matching, many operations have been proposed (see [12] for an overview). A given set of operations corresponds to a given distance between strings, in such way that the distance between two strings x and y is the minimal number of operations used to transform x into y. For instance, the Levenshtein (or edit) distance allows not only deletions but also insertions or substitutions, and can be used in our context, as well as many other classical distances.

Another example of the Match function would consider not only the set of generalized sessions computed from the log, but also all the generalized sessions that can be generated by crossing the generalized sessions of the log. For instance, if we consider the generalized sessions $g_1 = \langle c_1, c_2, c_3, c_4\rangle$ and $g_2 = \langle c_2, c_3, c_5\rangle$ of the example given in Section 3, then it could make sense to add the generalized session $g_4 = \langle c_1, c_2, c_3, c_5\rangle$ to the set of generalized sessions, by considering that as c_5 follows the sequence $\langle c_2, c_3\rangle$ in g_2 then it could follow the sequence $\langle c_1, c_2, c_3\rangle$ that is found in g_1. In this case, if the generalized current session is $\langle c_1, c_2, c_3\rangle$ then the Match function output two candidates c_4 and c_5 instead of one, since a matching is found for g_1 and for g_4.

6.2.2 Predict

The Match function outputs a set of pairs $\langle g, i\rangle$ where g is a generalized session and i is the position in the session where a match is found. A very simple Predict function is to output, for every such pairs, the successors of $g[i]$. More sophisticated Predict functions can be defined where not only successors but also predecessors are considered.

6.2.3 Class representative

There are many ways of computing a class representative. For instance, it can be the intersection or union of the queries of the class. For the simple MDX queries we consider, note that this union can be computed by the following function (where N is the number of dimension of the cube)[1].

Function 3 Union (C)

Require: C: A class of MDX queries
Ensure: An MDX query being the class representative of C

```
1: for j ∈ [1..N] do
2:    R_j ← ∅
3: end for
4: for each q_i ∈ C do
5:    for j ∈ [1..N] do
6:       R_j ← R_j ∪ π_j(q_i)
7:    end for
8: end for
9: return R_1 × R_2 × ... × R_N
```

The class representative can also be the most representative query of the class in the sense of the query partitioning adopted in the first step of the framework. For instance, assuming that the K-medoids is used as the query partitioning, the representative can be the medoid of the class.

6.3 Ranking candidate recommendations

Again there are many ways of ranking the candidates, from very basic to sophisticated ones. We list here a just few:

- Ranking the candidates according to how close to the last query of the current session they are.
- Ranking the candidates according to their number of occurrences.
- Ranking the candidates according to their number of references not already seen by the user in the current session.

[1]Indeed the union of two MDX queries as defined in Section 4 is not simply the union of sets of references, since an MDX query must be a set of references expressed as a cartesian product.

- Ranking the candidates according to a user profile. For example, [5] proposed a way of ordering MDX queries based on a user profile.

6.4 Default recommendation

As previously noted, the set of candidate recommendations can be empty. In that case, it could be useful to still be able to provide the user with a default recommendation. Various default recommendations can be proposed to the user. For instance, borrowing an idea from [9] we can propose as a default recommendation the representative of the authority class or the hub class, i.e., the class that has the highest number of successors (resp. predecessors). For instance, a hub can be found by simply computing for all classes c of the set of generalized sessions its number of occurrences minus the number of times it appears as the last class of the sequence. The hub is the class for which this number is the maximum.

7. EXPERIMENTATIONS

In this section, we present the results of the experiments we have conducted to assess the capabilities of our framework. We used synthetic data produced with our own data generator. Both our prototype for recommending queries and our generator are developed in Java using JRE 1.6.0_02. All tests are conducted with a Pentium 4 - 630 (3Ghz) with 2GB of RAM using Windows XP.

7.1 The instanciation of the framework

We have tested a particular instanciation of the generic framework, using some of the parameters presented in Section 6. More precisely, we use the Hausdorff Distance as the distance between queries. The partitioning is done by using the K-medoid algorithm as implemented by the Java Machine Learning library [1]. The query classifier associates a query with the class for which the medoid is the closest to the query, and the class representative is the medoid of the class. The Match function uses the approximate string matching approach with the set of operations reduced to the removal of the first element of the string. Thus this function tries to find if some generalized sessions match suffixes of the generalized current session. The Predict function simply returns $g[i]$ for a given generalized session g and position i. The candidate recommendations are ranked according to how close to the last query of the current session they are. Finally, the default recommendation computes the medoid of the hub class.

7.2 Generating cube and sessions

We use our generator to generate a cube and a set of sessions over this cube, using [13] and [15] as an inspiration to obtain realistic sizes for the cube and the sessions.

7.2.1 Generating the cube

To generate a cube, our generator uses the following parameters: A number of dimensions (T), a maximum number of levels per dimension (U) and a maximum number of values per dimension (V). For our experiments, we use a fixed values for T, U and V. The resulted cube has 6 dimensions, a maximum of 4 levels per dimension and a maximum of 100 values per dimension. If we consider the cube as a set of references, these parameters construct a cube of 1 000 000 000 000 references. Note that only the dimension tables are generated and not the cube itself (i.e., the cartesian product of the dimension tables), meaning that the generated data hold in main memory.

7.2.2 Generating the sessions

To generate a log, our generator uses the following parameters: A number of sessions in the log (X), a maximum number of queries per session (Y), and a maximum number of references per query (Z). In our test, only Z is fixed, and we use $Z = 100$ since it is reasonable to consider that users will not produce a cross table larger than 10×10 as the answer to an MDX query. Again, only the set of members in each dimension are generated as a query, not the set of references. In a session, the queries are generated in the following way: A first query is generated by random. Then each subsequent query in turn is generated by choosing randomly to modify or not the set of members of the dimensions of the previous query, in order to simulate the behavior of a user designing related queries.

7.3 Results

7.3.1 Performance analysis

Our first experiment assesses the efficiency of the instantiation of the framework to generate the recommendations. The performance is presented in Figure 2 according to various log sizes. These log sizes are obtained by playing with parameters X (number of sessions) and Y (maximum number of queries per session). X ranges from 20 to 200 and Y ranges from 10 to 150. We thus obtain logs of size varying between 100 and 10000 queries. Note that the current session is also generated with the session generator and thus with Y ranging from 1 to 100.

Note that what is measured is the time taken by the steps that are performed on-line, namely the computation of the generalized current session, the matching, the prediction, the class representative and the ranking of the candidates. The clustering and the calculation of the set of generalized sessions are done off-line and thus their execution times are not taken into account.

For this instantiation of the framework, Figure 2 shows that the time taken to generate the recommendations increases with the log size but remains highly acceptable.

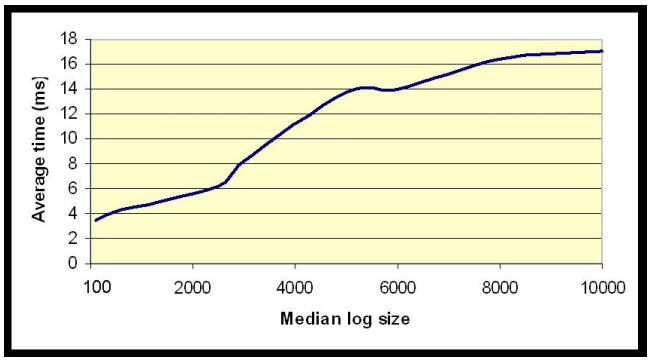

Figure 2: Performance analysis

7.3.2 Analyzing the precision of the recommendation

This experiment is a classical precision test of the process. Precision is a well known indicator used in information

retrieval [4] that measures the proportion of relevant document retrieved. In our case we slightly adapt this indicator to evaluate if a current session that matches perfectly one part of one of the sessions s of the log, say up to the position i, will give rise to recommending the query $s[i+1]$.

To assess this precision, we choose as the current session one of the session of the log from which we remove the last query. Then we observe whether this expected last query is closed to the recommendation. As ideally this last query would be the recommendation, we simply define the precision as the frequency of recommending this last query.

The factor that will affect the precision is the quality of the clustering. This quality is traditionally measured by computing the intra-cluster distance and inter-cluster distance. The intra-cluster distance is the average for all clusters of the average distance of each element of the cluster to its medoid, and the inter-cluster distance is the minimum distance of any two medoids of the clustering. The quality is the ratio of this two distances since a good clustering should minimize the intra-cluster distance and maximize the inter-cluster similarity.

Figure 3 shows that precision increases with the cluster quality, as it is expected. More interestingly it suggests that a cluster quality above 0.7 gives a sufficiently good precision. Note that we have a good precision altogether because we keep all the queries of the classes instead of keeping only medoid and because the recommended queries are ranked according to their distance to the last query of the current session and our generator generates queries that are closed to each other.

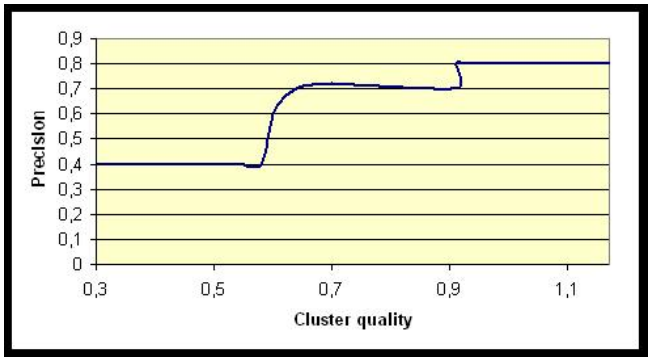

Figure 3: Precision of the recommandation

7.3.3 Measuring the quality of the recommendation

The aim of this final experiment is to assess the quality of the recommendation. Obviously as the log we used are synthetic data, the quality we measure is an objective indicator and not the relevance of the recommendation from the user point of view. This indicator mixes two measures: How distant are the queries of the current session to the classes that the query classifier associates with them, and the size of the suffix of the generalized current session for which atleast one matching is found in the set of generalized sessions. This indicator is a value ranging from 0 to 1, defined by the following formula:

$$\left(\frac{Suffix\_Quality}{Nb\_Dim} \times \alpha\right) + \left(\frac{Suffix\_Size}{S_c\_Size} \times (1-\alpha)\right)$$

where:

- $Suffix$ is the suffix of the generalized current session (it is the remaining generalized current session corresponding to a generalized session in logs), and $Suffix\_Size$ is the size of $Suffix$.
  Note that if there is no recommendation, there is no suffix thus we consider that $Suffix\_Size = 0$.

- $Suffix\_Quality$ is the average of the distances between each query of the current session and the medoid of the class which is associated with this query by the query classifier.

- Nb_Dim is the number of dimensions of the cube. Actually this number also corresponds to the maximum distance between queries. Indeed, the Hausdorff distance we use only relies on maxima and minima of the distance between references. As we use the Hamming distance as the distance between references, the Hausdorff distance cannot exceed the arity of a reference, that is the number of dimensions. Therefore we use Nb_Dim to normalize the $Suffix\_Quality$.

- S_c_Size is the size of the current session, used to normalize the size of the suffix of the generalized current session.

- α is a weight ranging from 0 to 1.

For this experiment the log are generated with the same protocol as in the performance analysis. Moreover, only clustering of quality greater 0.7 are considered. The weight α is fixed to 0.25 to give a higher importance to the size of $Suffix$.

Figure 4 shows how the quality is influenced by the median number of queries in each cluster. We notice that quality increases periodically according to the median number of queries in each cluster. This period increases when the number of cluster falls. So it suggests that the number of clusters should be adapted to the log size in order to obtain objectively good recommendations. However, we note that an average between 20 and 30 queries by cluster allows to obtain recommendations of good quality for clusters of size higher than 100.

8. CONCLUSION

In this paper, we propose a generic framework for recommending OLAP queries. Our framework is generic in the sense that it can be instantiated to change the way recommendations are computed. We give some examples of how it can be instantiated. For one instantiation where MDX queries are recommended, we present the results of some experiments we have conducted that show that recommendations can be computed efficiently and in which cases precise and objectively good recommendation can be expected.

Our future work include:

- The conduction of experiments on real data sets in order to better assess the quality of the recommended queries, as well as the assessment of various instantiations of our framework in order to determine to what context they are better adapted. To this end, we are looking for real users feedbacks.

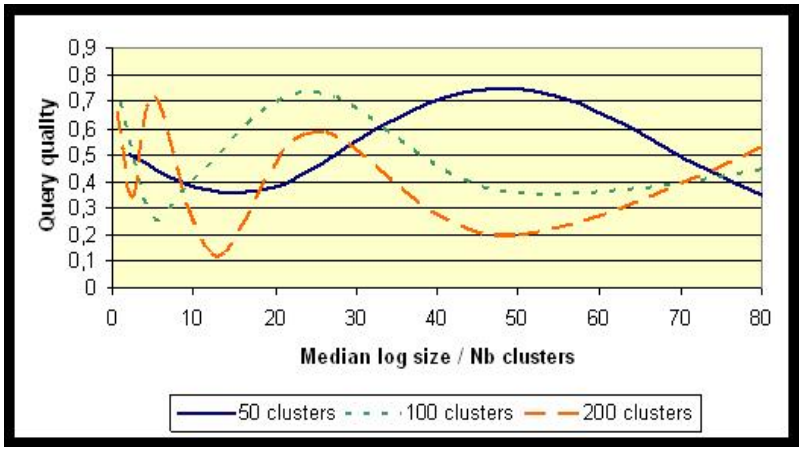

Figure 4: Quality of the recommendation

- The investigation of other instantiations of our framework. For example, we would like to investigate how the distance between queries can take into account the fact that dimensions are hierarchically structured. As another example, we would like to experiment other Match, Predict and ClassRep functions. More precisely, the Match function which is at the heart of the candidate generation, could be instantiated with the classical distances used in Approximate String Matching.
- The incorporation into our framework of the OLAP operations (like pivot, roll-up, slice) used during the session, in order to better take into account the way the users design their sessions.
- The extension of our definition of query in order to capture a larger part of the MDX language.

9. REFERENCES

[1] T. Abeel. Java machine learning library. Available at http://sourceforge.net/projects/java-ml, 2008.

[2] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. Knowl. Data Eng.*, 17(6):734–749, 2005.

[3] R. A. Baeza-Yates, C. A. Hurtado, and M. Mendoza. Query recommendation using query logs in search engines. In *EDBT Workshops*, pages 588–596, 2004.

[4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] L. Bellatreche, A. Giacometti, P. Marcel, H. Mouloudi, and D. Laurent. A personalization framework for olap queries. In *DOLAP*, pages 9–18, 2005.

[6] R. W. Hamming. Error detecting and error correcting codes. *Syst. Tech. J.*, 29:147–160, 1950.

[7] F. Hausdorff. *Grundzüge der Mengenlehre*. von Veit, 1914.

[8] L. Kaufmann and P. J. Rousseeuw. Clustering by means of medoids. In Y. Dodge, editor, *Statistical Data Analysis based on the L1 Norm*, pages 405–416. Elsevier/North Holland, Amsterdam, 1987.

[9] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[10] G. Matheron. *Random Sets and Integral Geometry*. John Wiley and sons, 1975.

[11] Microsoft Corporation. Multidimensional expressions (MDX) reference. Available at http://msdn.microsoft.com/en-us/library/ms145506.aspx, 2008.

[12] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[13] OLAP Council. APB-1 OLAP benchmark - release II. Available at http://www.olapcouncil.org/research/bmarkly.htm, 1998.

[14] C. Sapia. On modeling and predicting query behavior in olap systems. In *DMDW*, pages 2.1–2.10, 1999.

[15] C. Sapia. Promise: Predicting query behavior to enable predictive caching strategies for olap systems. In *DaWaK*, pages 224–233, 2000.

[16] S. Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, pages 307–316, 2000.

[17] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.