Generation of a Reference Data Set for Query Personalization

Verónika Peralta

Laboratoire PRiSM, Université de Versailles
45, avenue des Etats-Unis
78035 Versailles Cedex
FRANCE

veronika.peralta@prism.uvsq.fr

Technical Report [1]

October 2007

Abstract. This report describes the procedure followed for building a reference data set for measuring the pertinence of personalization algorithms. The reference data set consists of a set of queries, a set of user profiles and the query results that are pertinent for the user. In this way, the results obtained with a personalization algorithm can be compared to reference results in order to evaluate pertinence of the algorithm.

1. Introduction

Data personalization is one of the main solutions to improve relevance of data in information retrieval and database systems. Before being executed, user queries are reformulated on the basis of user profile preferences. This allows targeting user's center of interest and thus delivering pertinent results and reducing result size.

In order to measure the pertinence of results and then measuring the behaviour of personalisation algorithms, we need to compare delivered results with those effectively preferred by the user. In other words, we need a reference data set that contains several queries and the sets of query results that are pertinent for the user. In this way, the results obtained with a personalization algorithm can be compared to reference results in order to evaluate the precision and recall of the algorithm. Such a data set also allows the comparison among personalization algorithms, regarding pertinence, performance, result size or other measures.

In this report we describe the construction of a reference dataset for query personalization, which is the first step for building a personalization benchmark. There exist several benchmarks among which we can cite the TPC benchmarks for database server performances [6] or the TREC benchmarks for information retrieval systems [5]. However, as far as we know, there is no benchmark providing a validation framework to query personalization algorithms. A benchmark for query personalization should also manage different users and their preferences. Specifically, they should provide a large database, a set of user profiles and user queries as well as the reference results associated to each profile and query, i.e. they should provide collections of triplets {(profile, query) $\rightarrow$ results}.

Our dataset is derived from two public databases, i.e. MovieLens [1] and IMDb [2]. Both databases deal with data about movies. The IMDb database contains rich information about films, actors, directors, the places where they are produced, their budgets, their categories and the average rank given by the users who had evaluated them. IMDb describes more than 850.000 movies at the moment we have extracted its data (October 2006). The MovieLens database contains very few information about films but provides a huge amount of evaluations given by users who have seen these films. MovieLens provides a dataset composed of more than 1 million evaluations

[1] This research was partially supported by the French Ministry of Research and New Technologies under the ACI program devoted to Data Masses (ACI-MD), project #MD-33.

given by 6.040 users on 3.883 films. The two databases are complementary as they almost target the same movies (actually the set of films referred in MovieLens is a subset of those referred in IMDb). The main advantage of using these databases is that they provide a large volume of data, which is freely available at Internet. In addition, cinema data is very easy to understand, use and analyze. However, the join between the two databases is not easy to perform as there is not a universal identifier for the contained movies. The only common data is the titles of the movies but, unfortunately, they suffer from many problems such as abbreviation, translations into different languages and, generally, lack of writing standardization. Locally, each database has also many dirty data which needs to be cleaned and homogenized. Finally, the two databases, in particular IMDb, are semi-structured databases; their loading into a relational database necessitates several transformations. Consequently, using the two databases needs a substantial effort which we have done [4] as this kind of benchmark is not only useful for our evaluation but can benefit to a wide database community working on query personalization.

This report describes the procedure followed for building the reference data set from IMDb and MovieLens data. Section 2 presents and overview of the approach and a motivating example. Section 3 presents the design of the reference data set; specifically, it describes the procedures for generating user profiles based on MovieLens users' ratings, for generating a set of queries over IMDb data, and for obtaining the reference results for each pair (query, profile). Auxiliary routines and storage issues are also discussed. Section 4 describes the construction of the reference data set, discussing parameterization and execution of the generation procedures and presenting results and statistics. Finally, Section 5 concludes and presents future steps in order to complete the benchmark for query personalization.

2. Overview of the approach

In order to motivate the need of a reference data set, consider the user query of Figure 1, asking for *action movies posterior to 2000*. There is a large set of movies that satisfy query criteria. However, if we consider that user prefers French movies played by Jean Reno, the result size reduces considerably by delivering more pertinent results. Personalization algorithms take into account the user profile and reformulate the user query by including additional filtering criteria.

However, the results delivered by the personalization algorithm may exclude pertinent results and include non-pertinent results. Knowing the set of movies that the user effectively prefers, i.e. the set of results that are really pertinent, we can compare it with the results proposed by the personalization algorithm and quantitatively evaluate the behavior of the algorithm.

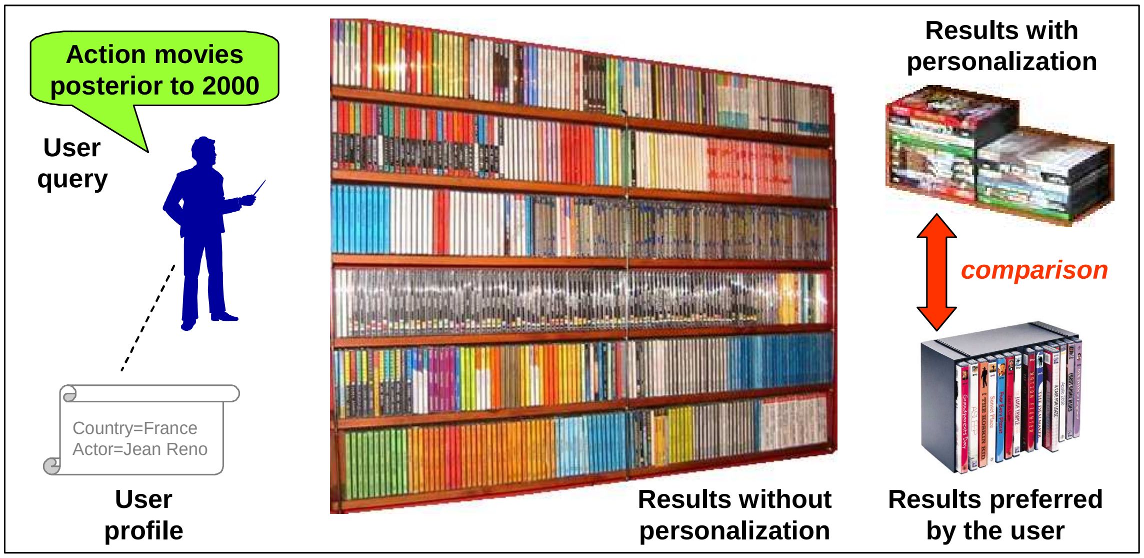

Action movies posterior to 2000
User query
Country=France
Actor=Jean Reno
User profile
Results without personalization
Results with personalization
comparison
Results preferred by the user

Figure 1 – Example of query personalization and comparison of results

In the general case, we consider a certain database instance, a user query Q_i, a user profile P_j and a personalization algorithm A_k. Figure 2 illustrates the query results obtained with and without personalization and their comparison with reference results. Specifically:

– The whole rectangle (grey zone) represents the space of solutions to Q_i, i.e. each point represents a tuple in the query domain;

– R_i (outer yellow oval) represents the set of results of Q_i obtained using the database query interface, without considering user profile;

– R_{ij} (inner red oval) represents the set of results of Q_i that are considered as pertinent by the user having profile P_j. Note that R_{ij} is a subset of R_i.

– R'_{ijk} (inner green oval) represents the set of results of Q_{ijk}, which corresponds to query Q_i enriched with profile P_j by algorithm A_k. As personalization algorithms add restriction predicates to queries, R'_{ijk} is also a subset of R_i.

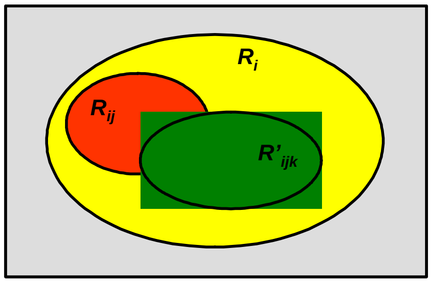
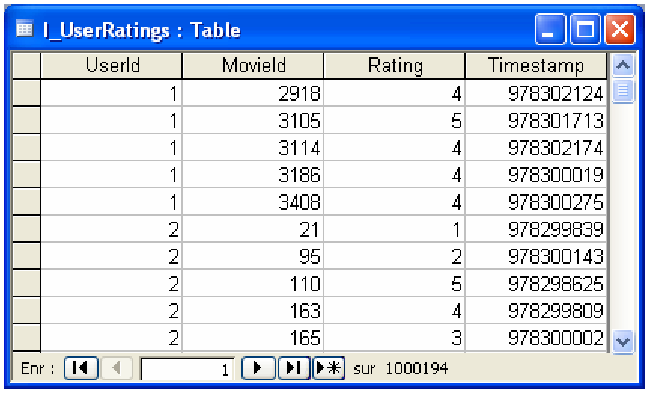


Figure 2 – Comparison of personalized results with reference results

The perfect personalization algorithm should returns all pertinent results and exclude all non-pertinent results. In other words, R_{ij} and R'_{ijk} should coincide. In practice, some pertinent results are discarded ($R_{ij} – R'_{ijk}$) and some non-pertinent results are returned ($R'_{ijk} – R_{ij}$). Precision and recall measures indicate such deviations from the reference result.

In order to provide a test platform for personalization algorithms, we need to build: a set of user profiles, a set of queries and the reference results for each pair (query, profile).

In order to generate user profiles, we use movie ratings provided by MovieLens. In fact, instead of asking certain users to manually build their profiles and classify query results according to their pertinence, we reuse movie ratings already expressed by real users. Specifically, each tuple of the I_UserRatings table of the integrated database (illustrated in Figure 3) corresponds to a user evaluation, registering the user identifier, the movie identifier, the rating (in a 1-5 star scale) and a timestamp (unused). See [4] for details on the extraction of MovieLens data and the feeding of the I_UserRatings table.


I_UserRatings : Table

UserId	MovieId	Rating	Timestamp
1	2918	4	978302124
1	3105	5	978301713
1	3114	4	978302174
1	3186	4	978300019
1	3408	4	978300275
2	21	1	978299839
2	95	2	978300143
2	110	5	978298625
2	163	4	978299809
2	165	3	978300002

Enr : 1 sur 1000194

Figure 3 – Part of the instance of the I_UserRatings table

User profiles are generated by joining the I_UserRatings table with other tables of the integrated database (which describe movie features) and extracting the common features of the movies with higher evaluation. These common features constitute the user profile. Tables describing movie features were extracted from IMDb database. See [4] for details on the extraction of IMDb data and their integration with MovieLens data.

Actually, we partition the I_UserRatings table in two subsets:

- *Training set*, from which user profile is generated, and
- *Test set*, which is proposed as source database for performing queries and measuring personalization results.

This partitioning assures that the obtained measures are not biased. Furthermore, considering different partitioning strategies we can obtain more precise measures.

Figure 4 illustrates the partitioning of the set of movies evaluated by user j in training and test sets, highlighting the set of preferred movies. User profile is extracted from the preferred portion of the training set.

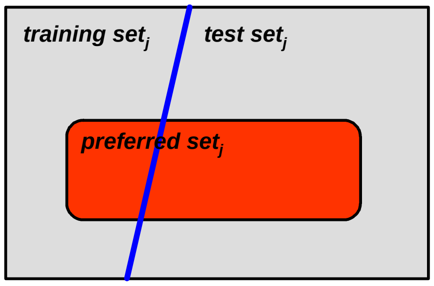


Figure 4 – Extraction of user profiles from the preferred training set

In order to define user queries, we consider as space of solutions, some of the movies that the user has already evaluated (those of the test set), and we generate queries asking for movies that satisfy different criteria (predicates on movie features obtained from IMDb). To this end, we join the I_UserRatings table with some tables describing movie features and we add some filtering conditions on such features. Both, joining tables and filtering conditions are randomly generated.

As we know the rating of each movie, we can easily build the set of reference results, i.e. those movies in the query result that have a good rating. In other words, the grey zone of Figure 2 consists of the tuples of the training set of a given user, the yellow zone consists of the tuples that satisfy a certain query and the red zone consists of the subset of tuples that have a good rating.

Next section describes the mechanisms for generating user profiles, user queries and reference results.

3. Design of the reference database

The reference database consists in a set of user profiles, a set of user queries and the corresponding “good” results for each couple (profile, query). In this section we present the design of the reference database, i.e. we describe the procedures for generating user profiles and user queries, the procedures for calculating reference results and the database structures for storing user profiles, user queries and reference results. This section only describes the design of such procedures; the setting of the appropriate parameters and the obtained results are presented in next section.

3.1. Generation of user profiles

User profiles are sets of predicates that state user preferences on movie features. Profile predicates have the form *feature=value*, where *value* ranges in the domain of the movie *feature*. For example, a certain user may prefer *movies spoken in French* or *action movies*; which is expressed by the predicates: *Language = French*, *Genre = Action*.

In order to extract a user profile from a set of user evaluations, we look for common features of the evaluated movies, for example, if most of the movies the user has assigned a great rating are filmed in France, we deduce that the user prefers *movies filmed in France*, and we propose the predicate *LocationCountry=France*.

In this section we describe the generation of a large set of predicates. Query personalization algorithms may choose among the generated predicates and build different user profiles. To this end, we associate a *weight* to each extracted predicate, which represents the percentage of the evaluated films that satisfy the predicate. Weights allow conforming more or less restrictive user profiles by choosing the predicates with higher weights or accepting predicates with lower weights.

Weighted predicates have the form <*table.attribute operator value (weight)*> where: *table* and *attribute* refer to an attribute of a table of the integrated schema (referencing a movie feature), *value* is an element of the attribute domain, *operator* $\in$ {=,<,≤,>,≥} and *weight* represents the percentage of the evaluated films that satisfy the predicate.

Some examples of weighted predicates are:

- I_MovieLanguages.language = English (80)
- I_Countries.continent = Europe (25)
- I_MovieGenres.genre = Comedy (40)
- I_MovieYears.year ≥ 2000 (90)
- I_MovieBusiness.budgetusd ≥ 10.000.000 (60)

These examples can be interpreted as *among the films the user has evaluated, 80% are spoken in English, 25% have been filmed in Europe, 40% are comedies, 90% are posterior to year 2000 and 60% have reported more than 10 million dollars.*

The generation of predicates consists of three main steps:

1) Partitioning user evaluations in order to determine training, test and preferred sets
2) Extracting predicates for the evaluations on the preferred training set
3) Computing weights for the extracted predicates (eliminating predicates with low weights)

The following sub-sections describe each step:

3.1.1. Partitioning of user evaluations

In order to partition user evaluations, we define a set of conditions (training, test and preferred conditions) that allow delimiting the training, test and preferred sets. We test different partitioning strategies, i.e. different ways of partitioning user evaluations.

Training conditions have the form *attribute < value*, where *attribute* is a numeric attribute of the I_UserRatings table and value is a value of the attribute domain. In order to define conditions, we added five attributes (named C1, C2, C3, C4 and C5) to the I_UserRatings table, all of them taking random values between 0 and 9. Test conditions are the negations of training conditions, i.e. they have the form *attribute ≥ value*. Preferred conditions have the form *I_UserRatings.rating ≥ value*, where *value* is a number between 1 and 5.

Training, test and preferred sets are defined as views on the I_UserRatings table according to these conditions:

- TrainingSet:
```
SELECT * FROM I_UserRatings WHERE TrainingCondition;
```
- TestSet:
```
SELECT * FROM I_UserRatings WHERE TestCondition;
```
- PreferredSet:
```
SELECT * FROM I_UserRatings WHERE PreferredCondition;
```

Preferred training and preferred test sets are defined as conjunction of the corresponding conditions:

- PreferredTrainingSet:
```
SELECT * FROM I_UserRatings WHERE TrainingCondition AND PreferredCondition;
```
- PreferredTestSet:
```
SELECT * FROM I_UserRatings WHERE TestCondition AND PreferredCondition;
```

Additional parameters are used in the generation of predicates. They allow generating meaningful predicates, i.e. discarding predicates having a small weight (MinWeight threshold) and predicates appearing in too few evaluations (MinEval threshold).

The Ref_strategies table encloses all these parameters (see Table 1). Each tuple of this table corresponds to a different strategy. A strategy id allows identifying strategies.

3.1.2. Extraction of predicates

We generate different types of predicates, each one corresponding to a movie feature (e.g. language, country, actor or genre). In order to generate predicates for a given feature, we join user evaluations with the table storing such feature and we count the movies that correspond to each feature value. For example, if we consider the *language* feature, we count how many movies correspond to English, Spanish, etc. We parameterized the features to look for in the Ref_PredicateTypes table (see Table 1).

We used two algorithms for generating predicates, which consider equality and inequality of values respectively. The former computes the number of films 'having a certain feature value' and the latter computes the number of films 'having more than a certain feature value'. The approach can be extended with other predicate-extracting methods, for example, clustering algorithms. Both algorithms are implemented as PL-SQL procedures; they are sketched as follows:

- *Ref_ExtractEqualityPred*: This procedure computes the feature values describing evaluated movies and counts the number of evaluations corresponding to each feature value. This is done for each user and each strategy, storing the computed predicates in a temporal table (Ref_Aux2, described later in Table 1).
- *Ref_ExtractInequalityPred*: This procedure computes the feature values describing evaluated movies and counts the number of evaluations corresponding to greater values than each feature value. This is done for each user and each strategy, also storing the computed predicates in the temporal table (Ref_Aux2).

Both algorithms discard predicates having a small support, i.e. being present in few tuples. The MinEval parameter of the Ref_Strategies table set such threshold.

An additional procedure is used for discarding predicates corresponding to Null or dummy values (e.g. AttributeValue='_unknown'):

- *Ref_DiscardDummyPred*: This procedure eliminates (from the Ref_Aux2 table) the predicates corresponding to Null or dummy values.

The implementation of these procedures is described in Annex 1.

3.1.3. Computation of predicate weights

In order to compute predicate weights, we divide the number of user evaluations corresponding to each predicate (e.g. the number of evaluated movies having *language='English'*) by the total of user evaluations. The former value is registered in the Ref_Aux2 table, the latter value has to be computed. We propose 2 PL-SQL procedures for computing the total of user evaluations and calculating weights respectively:

- *Ref_CountEvaluations*: This procedure computes the number of evaluations in the preferred training set for each user and each strategy. It stocks the computed results in a temporal table (Ref_Aux1, described later in Table 1).
- *Ref_ComputePredWeights*: This procedure computes predicate weights. Basically, it divides the number of evaluations corresponding to each predicate (those stored in Ref_Aux2) by the total of evaluations of the corresponding user and strategy (those stored in Ref_Aux1). The predicates having low weight (according to the MinWeight parameter of the Ref_Strategies table) are discarded. The generated predicates are stored in the Ref_Predicates table, which is described in next sub-section.

The PL-SQL code of these procedures is listed in Annex 1.

3.1.4. Storage issues

Table 1 describes the tables used in the generation of predicates.

Generation parameters are stored in the Ref_Strategies and Ref_PredicateTypes tables. They store partitioning strategies and predicate types respectively. A view (Ref_GenerationParameters) computes the Cartesian product of strategies and predicate types, i.e. it indicates the combinations of parameters for generating all predicate types for all strategies.

```
CREATE OR REPLACE VIEW Ref_GenerationParameters AS
SELECT  S.strategyid, S.minweight, S.minrating, S.mineval,
   S.trainingcondition, P.execid, P.lookupview, P.attributename,
   P.function, P.param1, P.param2, P.param3
FROM Ref_Strategies S, Ref_PredicateTypes P;
```

The temporal tables Ref_Aux1 and Ref_Aux2 store the number of evaluations per user and strategy and the generated predicates per user and strategy respectively. Both tables are used for computing predicate weights and eliminating the predicates having low weights. The remaining predicates (and their weights) are stored in the Ref_Predicates table.

Table	Attributes	Constraints
Ref_Strategies *Description of partitioning strategies*	– StrategyId: Numeric(2) – TrainingCondition: String(50) – TestCondition: String(50) – PreferredCondition: String(10) – MinRating: Numeric(1); the threshold of the preferred condition – MinWeight: Numeric(5,3) – MinEval: Numeric(5)	Primary key: StrategyId Not null: StrategyId, MinWeight, MinRating, MinEval, TrainingCondition, TestCondition, PreferredCondition
Ref_PredicateTypes *Description of predicate types*	– PTypeId: Numeric(3) – LookupView: String(40); an auxiliary view that relates movies with predicate feature – TableName: String(40); the table that stores the feature – AttributeName: String(30); the attribute that stores the feature value – Function: String(20); the algorithm to be used for generating predicates of this type – Param1: String(40); extra parameter for such function – Param2: String(40); extra parameter for such function – Param3: String(40); extra parameter for such function	Primary key: PTypeId Not null: PTypeId, LookupView, TableName, AttributeName, Function
Ref_Aux1 *Temporal table used for storing the number evaluations of each user for each strategy (in the preferred training set)*	– StrategyId: Numeric(2) – UserId: Numeric(4) – MovieCount: Numeric (5); the number of evaluated movies	Primary key: StrategyId, UserId Not null: StrategyId, UserId, MovieCount
Ref_Aux2 *Temporal table used for storing predicates for each user and each strategy (in the preferred training set)*	– StrategyId: Numeric(2) – PTypeId: Numeric(3) – UserId: Numeric(4) – AttributeValue: String(256) – Operator: String(5) – MovieCount: Numeric (5); the number of evaluated movies satisfying the predicate	Primary key: StrategyId, PTypeId, UserId, AttributeValue, Operator Not null: StrategyId, PTypeId, UserId, AttributeValue, Operator, MovieCount Index: StrategyId, UserId
Ref_Predicates *Generated predicates per user and strategy*	– StrategyId: Numeric(2) – PTypeId: Numeric(3) – UserId: Numeric(4) – TableName: String(40) – AttributeName: String(30) – AttributeValue: String(256) – Operator: String(5) – Weight: Numeric	Primary key: StrategyId, PTypeId, UserId, AttributeValue, Operator Not null: StrategyId, PTypeId, UserId, TableName, AttributeName, AttributeValue, Operator, Weight Index: StrategyId, UserId

Table 1 – Tables used for the generation of profile predicates

3.2. Generation of queries

In order to generate a large set of queries, we proceed as follows:

- We depart from the query: SELECT I_UserRatings.movieid FROM I_UserRatings
- We randomly generate a set of predicates and we add them in the WHERE clause
- We add all tables referenced in predicates to the FROM clause
- We complete the WHERE clause with the necessary join predicates for relating all tables of the FROM clause. We eventually add new tables to the FROM clause if they are necessaries for joining other tables.

As an example, consider the predicates of Figure 5a. We add them to the WHERE clause of the query (Figure 5b, in green and italics), and we add predicate tables (I_MovieLanguages, I_Countries and I_MovieGenres) to the FROM clause (Figure 5b, in bleu and bold). Figure 5d shows a portion of the database schema containing predicate tables. Note that I_MovieLanguages and I_MovieGenres join with I_UserRatings by movieid, but they have no common attributes for joining with I_Countries. The last join is carried out using the I_MovieCountries table. The added tables and predicates are shown in Figure 5c (in red and italics).

(a)

I_MovieLanguages.language = English
I_Countries.continent = Europe
I_MovieGenres.genre = Comedy

(b)

SELECT I_UserRatings.movieid
FROM I_UserRatings, **I_MovieLanguages, I_Countries, I_MovieGenres**
WHERE I_MovieLanguages.language = 'English'
AND I_Countries.continent = 'Europe'
AND I_MovieGenres.genre = 'Comedy'

(c)

SELECT I_UserRatings.movieid
FROM I_UserRatings, I_MovieLanguages, I_Countries, I_MovieGenres, *I_MovieCountries*
WHERE I_MovieLanguages.language = 'English'
AND I_Countries.continent = 'Europe'
AND I_MovieGenres.genre = 'Comedy'
AND I_MovieGenres.movieid = I_Movies.movieid
AND I_MovieLanguages.movieid = I_Movies.movieid
AND I_MovieCountries.movieid = I_Movies.movieid
AND I_Countries.country = I_MovieCountries.country

(d)

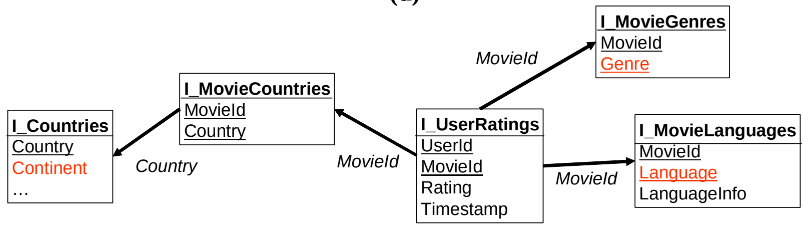


Figure 5 – Example of query generation

Starting from a set of predicates, the construction of SQL queries is quite straight-forward. The point is how to generate the predicates. In order to carry out such generation, we follow the same approach used for generating profile predicates, i.e. we extract common attribute values describing the evaluated movies of each user.

However, the generation process has some important differences: (i) we consider all user evaluations, not only those having high ratings, (ii) we also extract predicates having low weights, and (iii) we randomly choose a small number of predicates. These differences allow generating queries that considerably differentiate from user profiles. Concretely, while user profiles contain all high-weight predicates, queries contain few randomly chosen predicates, which rarely represent user preferences. In addition, movie features that are irrelevant for users may be chosen for query predicates. These three differences also allow the generation of typical queries but returning result sets of varied sizes. Specifically, the selection of a small number of predicates (from 1 to 5) avoids generating monster queries that returns no data. However, the randomness of the selection allows obtaining result sets of different sizes, ranging from almost empty sets when queries have several restrictive predicates (of low weight) to almost all data when queries have few non-restrictive predicates (of high weight).

Sub-section 3.2.1 describes the generation of query predicates and Sub-section 3.2.2 describes the construction of SQL queries.

3.2.1. Extraction of query predicates

The extraction of query predicates follows the same procedures explained for extracting profile predicates (see Sub-section 3.1). Actually, we define a special strategy (with StrategyId=0) that defines the preferred training set as containing all user evaluations. Then, we generate one query per user.

Among the generated predicates, we randomly select a small number of predicates (between 1 and 5). This is carried out by 4 algorithms, implemented as PL-SQL procedures:

- *Ref_SetQueryPredNumber*: This procedure randomly set a desired number of predicates (between 1 and 5) for each query. It stocks the computed results in a temporal table (Ref_AuxQ1).
- *Ref_SetPredRandomWeight*: This procedure computes candidate predicates for each query and assigns a random weight to each predicate. Candidate predicates are taken from the Ref_Predicates table, selecting the strategy 0. The random weights will be used later for selecting a small number of predicates per query (according to the desired number of predicates stored in the Ref_AuxQ1 table). The procedure stocks the generated predicates in a temporal table (Ref_AuxQ2).
- *Ref_SelectQueryPred*: This procedure selects, among the candidate predicates of each query (stored in the Ref_AuxQ2 table), the ones having higher random weight. The number of predicates to select is taken from the Ref_AuxQ1 table. A sub-procedure (Ref_SelectQueryPred_aux) is used for carrying out the selection for each query. The selected predicates are stored in a temporal table (Ref_AuxQ3).
- *Ref_DeleteQueryConflictivePred*: This procedure eliminates the conflictive predicates of the Ref_AuxQ3 table, i.e. when several predicates reference a same attribute (e.g. *language='English'* and *language='Spanish'*), the one having a higher random weight is kept. To this end, the procedure first computes the maximum random weight for each attribute (which are stored in the Ref_AuxQ4 temporal table) and then, proceeds to the selection. The obtained predicates are stored in the Ref_QueryPredicates table.

The PL-SQL code of these procedures is listed in Annex 2. The table that stores the generated predicates as well as the temporal tables used in the generation are described in Table 2.

Table	**Attributes**	**Constraints**
Ref_AuxQ1 *Temporal table used for storing the number of predicates to select per query*	– QueryId: Numeric(4) – PredNumber: Numeric(2); the number of predicates to select	Primary key: QueryId Not null: QueryId, PredNumber
Ref_AuxQ2 *Temporal table used for storing candidate predicates per query (with random weights)*	– PTypeId : Numeric (3) – QueryId: Numeric (4) – TableName: String(40) – AttributeName: String(30) – AttributeValue: String(256) – Operator: String(5) – Weight: Numeric – Rnd : Numeric; random weight used for random selection	Primary key: PTypeId, QueryId, AttributeValue, Operator Not null: PTypeId, QueryId, TableName, AttributeName, AttributeValue, Operator, Weight, Rnd Index: QueryId
Ref_AuxQ3 *Temporal table used for storing randomly selected predicates per query*	– PTypeId : Numeric (3) – QueryId: Numeric (4) – TableName: String(40) – AttributeName: String(30) – AttributeValue: String(256) – Operator: String(5) – Weight: Numeric – Rnd : Numeric; random weight used for random selection	Primary key: PTypeId, QueryId, AttributeValue, Operator Not null: PTypeId, QueryId, TableName, AttributeName, AttributeValue, Operator, Weight, Rnd Index: QueryId

Ref_AuxQ4 *Temporal table used for storing the maximum random weight for conflictive predicates (those referencing a same attribute)*	– PTypeId : Numeric (3) – QueryId: Numeric (4) – Rnd : Numeric; random weight	Primary key: PTypeId, QueryId Not null: PTypeId, QueryId, Rnd
Ref_QueryPredicates *Generated predicates per query*	– PTypeId : Numeric (3) – QueryId: Numeric (4) – TableName: String(40) – AttributeName: String(30) – AttributeValue: String(256) – Operator: String(5)	Primary key: PTypeId, QueryId Not null: PTypeId, QueryId, TableName, AttributeName, AttributeValue, Operator

Table 2 – Tables used for the generation of query predicates

3.2.2. Construction of SQL queries

As previously explained, we depart from a query selecting movie ids (SELECT I_UserRatings.movieid FROM I_UserRatings), we add query predicates to the WHERE clause and we add predicate tables to the FROM clause. In addition, we add extra conditions to the WHERE join in order to join predicate tables to the I_UserRatings table, possibly adding transitive tables to the FROM clause.

Each type of predicate determines the join conditions and extra tables necessaries for the join, which are derived from the database schema. They are stored in 2 tables (Ref_PTypeJoinConditions and Ref_PTypeJoinTables), which are described in Table 3.

The procedures for generating SQL queries from those tables are implemented in Java. The obtained queries are stored in the Ref_Queries table, also described in Table 3.

Table	Attributes	Constraints
Ref_PTypeJoinConditions *Generated predicates per query*	– PTypeId : Numeric (3) – LookupView: String(40) – AttributeName: String(30) – JoinCondition: String(40)	Primary key: PTypeId, JoinCondition Not null: PTypeId, TableName, AttributeName, JoinCondition
Ref_PTypeJoinTables *Generated predicates per query*	– PTypeId : Numeric (3) – LookupView: String(40) – AttributeName: String(30) – JoinTable: String(40)	Primary key: PTypeId, JoinTable Not null: PTypeId, TableName, AttributeName, JoinTable
Ref_Queries *Reference queries*	– QueryId: Numeric (4) – QueryText: String(2000) – RelationsNumber: Numeric(2); the number of tables in the where clause	Primary key: QueryId Not null: QueryId, QueryText

Table 3 – Tables used in the generation of SQL queries

3.3. Computation of Reference Results

Given a query Q, a user U and a strategy S, the query should be executed on the user *test set* corresponding to the strategy. The test set is computed as a view on user evaluations, as follows:

– TestSet: `SELECT * FROM I_UserRatings WHERE TestCondition AND userid=U`

Replacing the I_UserRatings table by this view, corresponds to add the test condition and the condition on user id to query expression. For example, the query

```
SELECT I_UserRatings.movieid
FROM I_UserRatings, I_MovieCountries
WHERE I_UserRatings.movieid = I_MovieCountries.movieid
AND   I_MovieCountries.country = ‘France’
```

is unfolded to:

```
SELECT I_UserRatings.movieid
FROM I_UserRatings, I_MovieCountries
WHERE I_UserRatings.movieid = I_MovieCountries.movieid
AND   I_MovieCountries.country = ‘France’
AND   TestCondition AND userid=U;
```

The reference results are computed in the same way, also adding the preferred condition to the query.

Next section describes the execution of the procedures and the analysis of the obtained results.

4. Construction of the Reference Database

In this section we describe the results and statistics obtained from the execution of the previously described procedures, i.e. those for generating user profiles and user queries. We firstly describe the setting of strategies and the selection of relevant movie features. Then, we present the number of extracted predicates, analyzed by several factors and the number of generated queries, also analyzed by several factors. Finally, we present statistics on result sizes for pairs <query, profile>.

4.1. Setting of strategy parameters

As previously argued, we aim at generating different partitioning strategies in order to obtain unbiased experimental results.

In order to set appropriate partitioning sizes we tested different parameters. Firstly, five random attributes were added to the I_UserRatings table (namely, C1, C2, C3, C4 and C5), each one ramdomly filled with an integer between 0 and 9. Therefore, training conditions were expressed in the form Ci < N, $1 \leq i \leq 5$, $0 \leq N \leq 9$. We defined two training sizes, with 50% of tuples (i.e. Ci < 5) and 30% of tuples (i.e. Ci < 3) respectively. Secondly, we defined three preferred set sizes, with rating ≥ 3, rating ≥ 4 and rating ≥ 5 respectively. This leads to 6 combinations of parameters.

Table 4 shows the average number of ratings in the preferred training set for each type of strategy and Table 5 shows the average number of ratings in the preferred training set per user and type of strategy.

Training size	All ratings	Rating ≥ 3	Rating ≥ 4	Rating ≥ 5
100 %	1.000.194	836.464	575.272	226.307
50 %	500.217	418.294	287.788	113.189
30 %	299.825	250.627	172.594	67.804

Table 4 – Average number of ratings in the preferred training set per type of strategy

Training size	All ratings	Rating ≥ 3	Rating ≥ 4	Rating ≥ 5
100 %	166	138	95	37
50 %	83	69	48	19
30 %	50	42	29	11

Table 5 –Average number of ratings in the preferred training set per user and type of strategy

Note that the number of ratings in the preferred training set is too small when considering rating ≥ 5. So, we did not consider such setting. We kept a total of 21 strategies, which are shown in Table 6; strategy 0 is used later for query generation; the remaining strategies are packed by 5, for $1 \leq i \leq 5$.

Strategy id	Training condition	Test condition	Preferred condition
0	Ci < 10		Rating ≥ 0
1-5	Ci < 5	Ci ≥ 5	Rating ≥ 3
6-10	Ci < 5	Ci ≥ 5	Rating ≥ 4
11-15	Ci < 3	Ci ≥ 3	Rating ≥ 3
16-20	Ci < 3	Ci ≥ 3	Rating ≥ 4

Table 6 – Parameters of strategies

Further parameters are the minimum weight (MinWeight) and the minimum number of evaluations (MinEval) were set to 10 and 5 respectively for all strategies. We preferred generating a great number of predicates even having low weights because they can be filtered thereafter.

4.2. Set of predicate types

We considered a set of 35 attributes containing relevant movie features, which are listed in Table 7. Some of them were implemented (those needing an equality or inequality comparison function), the remaining ones (which are shadowed) were kept as future work.

NºAtt	Table	Attribute	Comparison function
2	I_MovieGenres	genre	equality
3	I_MovieCountries	country	equality
4	IV_MovieCountries	continent	equality
5	I_MovieYears	year	equality
6	IV_MovieYears	decade	equality
7	I_MovieYears	year	clustering
8	I_MovieRatings	rating	equality
9	I_MovieRatings	rating	inequality
10	I_MovieRatings	votes	inequality
11	I_MovieKeywords	keyword	equality
12	I_MovieLanguages	language	equality
13	I_MovieProductionCompanies	companyname	equality
14	IV_MovieProductionCompanies	country	equality
15	IV_MovieProductionCompanies	continent	equality
16	I_MovieProductionCompanies	companyname	reconciliation
17	I_MovieColors	color	equality
18	I_MovieSounds	soundmix	equality
19	I_MovieSounds	soundmix	reconciliation
20	I_MovieBusiness	budgetusd	inequality
21	I_MovieBusiness	revenueusd	inequality
22	I_MovieLocations	zone	equality
23	IV_MovieLocations	country	equality
24	IV_MovieLocations	continent	equality
25	I_MovieRunningTimes	country	equality
26	IV_MovieRunningTimes	continent	equality
27	I_MovieRunningTimes	durationinterval	equality
28	I_MovieRunningTimes	duration	clustering
29	I_MovieDirectors	director	equality
30	I_MovieWriters	writer	equality
31	I_MovieProducers	producer	equality
32	I_MovieCostumeDesigners	costumedesigner	equality
33	I_MovieProductionDesigners	productiondesigner	equality
34	I_MovieActresses	actress	equality
35	I_MovieActors	actor	equality
36	I_MovieLinks	linktype	equality

Table 7 – Candidate attributes describing movie features

4.3. Obtained profile predicates

Having defined strategy parameters and types of predicates, we proceeded to execute the profile generation procedures described in Sub-section 3.1. We obtained 8.779.207 predicates for all users and all strategies. The following figures analyze the number of obtained predicates per strategy, weight, user and attribute.

Figure 6 shows the number of predicates extracted for each strategy. As expected, the number of predicates decreases for more restrictive strategies.

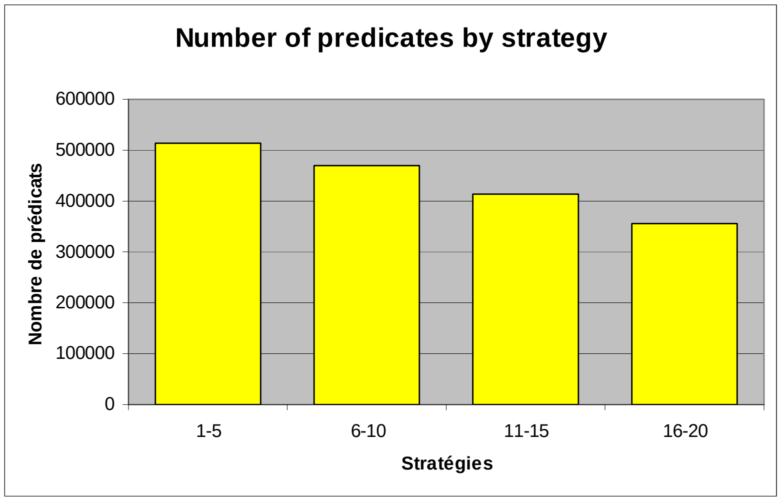


Figure 6 – Average number of predicates per type of strategy

Figure 7 also considers predicate weights; it shows the number of predicates having a weight greater or equal to a given value. Note that the distribution is similar for all types of strategies.

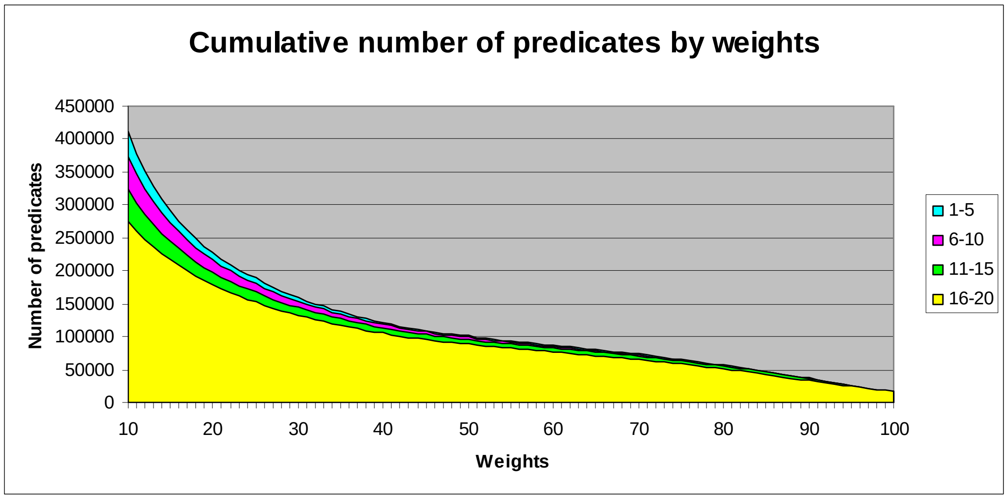


Figure 7 – Cumulative number of predicates by weights

Figures 8, 9, 10 and 11 incorporate a new dimension: the number of users having a certain profile size. By profile size we mean the number of predicates generated for the user. Figure 8 shows that the number of predicates generated for each user varies largely, from 1 to more than 160. As special cases, we note that there is nearly a hundred users having a small number of predicates (close to 20) and nearly a hundred users having close to 80 predicates. This distribution is very different when only considering weights greater or equal to 30 (Figure 9) or 50 (Figure 10). In the former, most users have profile sizes between 20 and 30 predicates, and in the latter, most users have profile sizes between 15 and 20 predicates. Figure 11 illustrates the case of predicate weights greater or equal to 90. In this case, all users have a few number of predicates. Note that in all figures, the distributions are quite independent of the type of strategy.

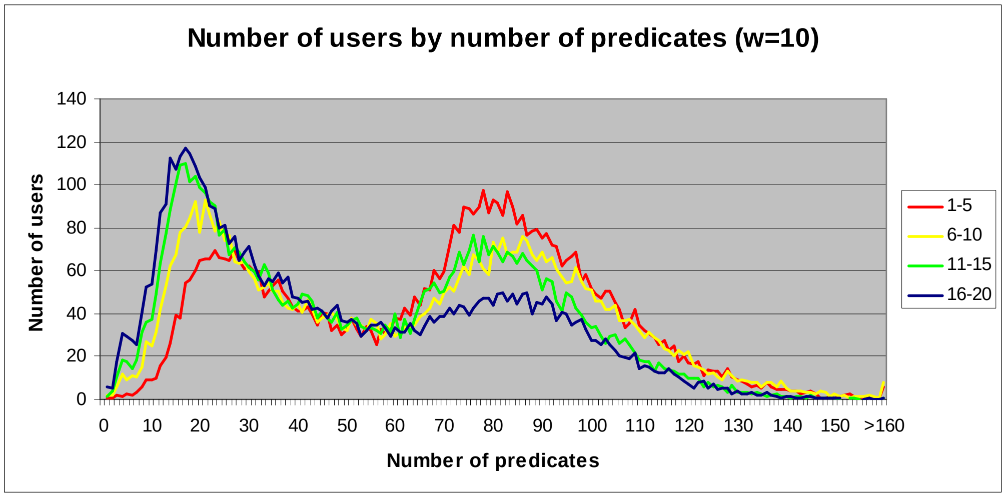


Figure 8 – Number of users having a certain profile size (with predicate weight ≥10)

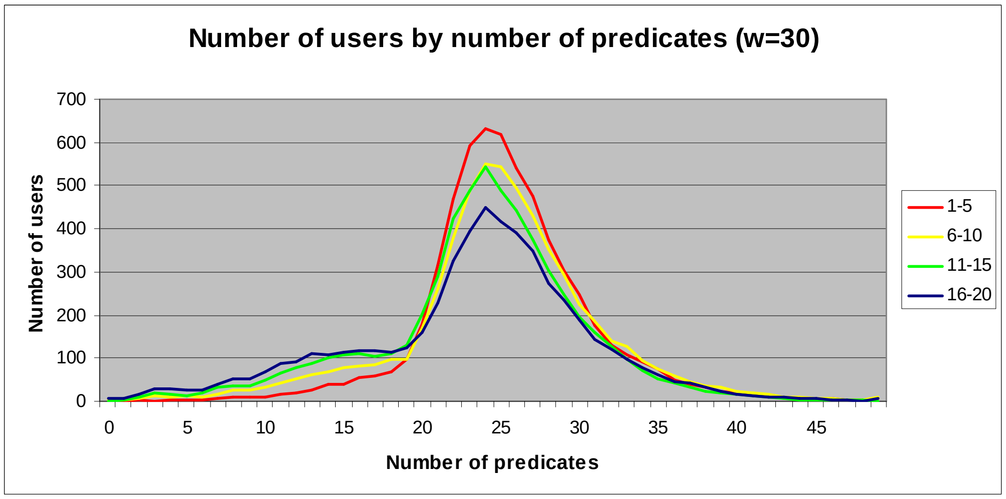


Figure 9 – Number of users having a certain profile size (with predicate weight ≥30)

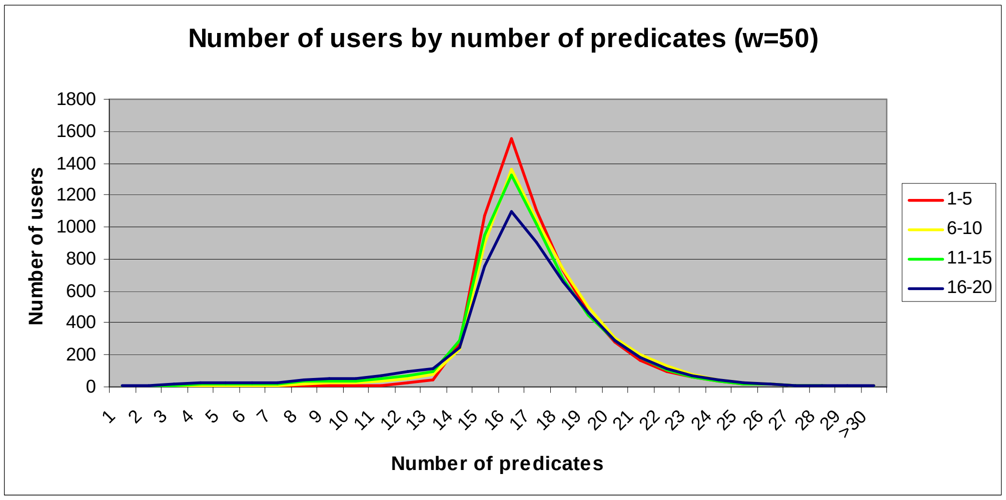


Figure 10 – Number of users having a certain profile size (with predicate weight ≥50)

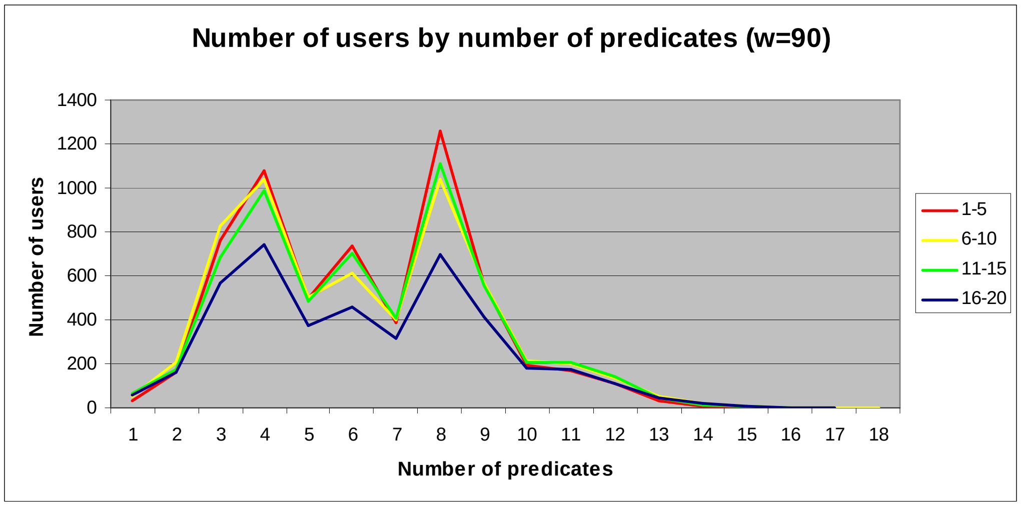


Figure 11 – Number of users having a certain profile size (with predicate weight ≥90)

Finally, Figure 12 shows the number of predicates per type of predicate, highlighting the most generated predicates concerns keywords (type 11), IMDb global rating (type 9), genre (type 1) and link type (type 36). Conversely, there are too few predicates concerning running year (type 8), and casting (types 29 to 35).

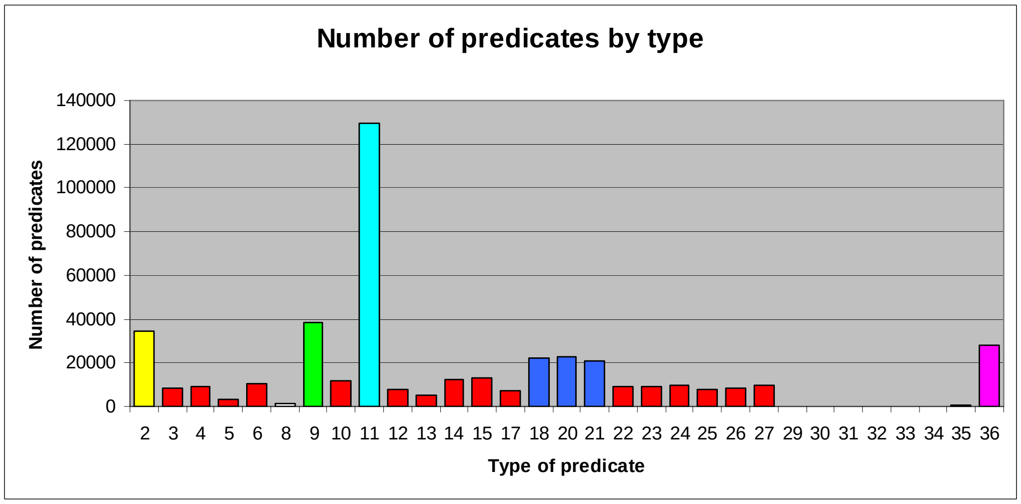


Figure 12 – Number of predicates per type (for all strategies)

4.4. Obtained queries

After generating user profiles, we proceeded to execute the query generation procedures. We firstly generated query predicates, using the same procedures than for profile generation, but fixing the strategy 0 (special strategy that extracts predicates from the whole set of ratings). Therefore, we obtained 6040 queries, one per user. We obtained an initial set of 622.061 predicates for all queries, from which we randomly selected 18.142. We kept 15.996 predicates after eliminating contradictory ones. The following figures show the number of predicates of each query at each generation stage.

Figure 13 shows the number of queries having a certain number of generated predicates. Note that most queries have between 80 and 130 predicates but there are some queries with an enormous number of predicates. In all cases, the random selection of a small number of predicates assures that the query does not represent the user profile.

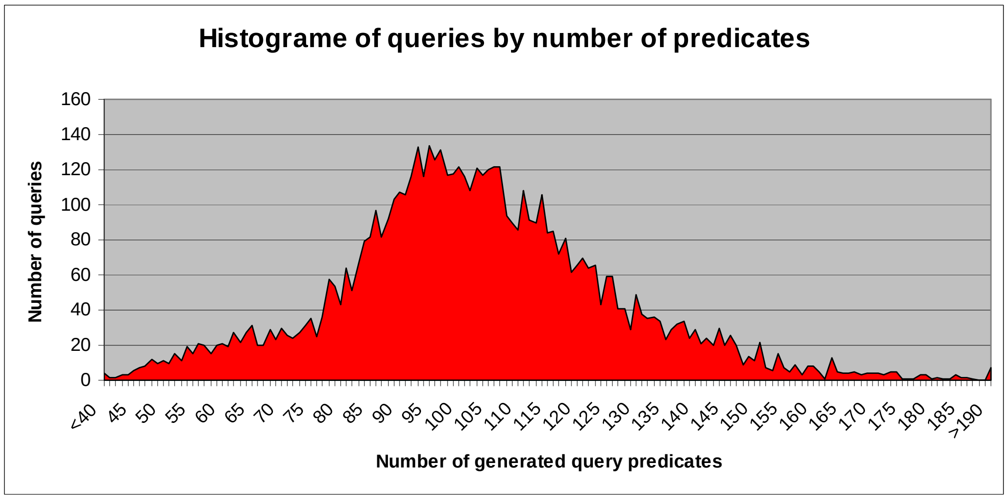


Figure 13 – Number of queries having a certain number of predicates (initial generation)

Figure 14 also shows the number of queries having a certain number of predicates, but after random selection and after elimination of contradictory predicates. The former is quite uniformly distributed but in the latter, there are less queries having 4 and 5 predicates (because they contained more contradictory predicates).

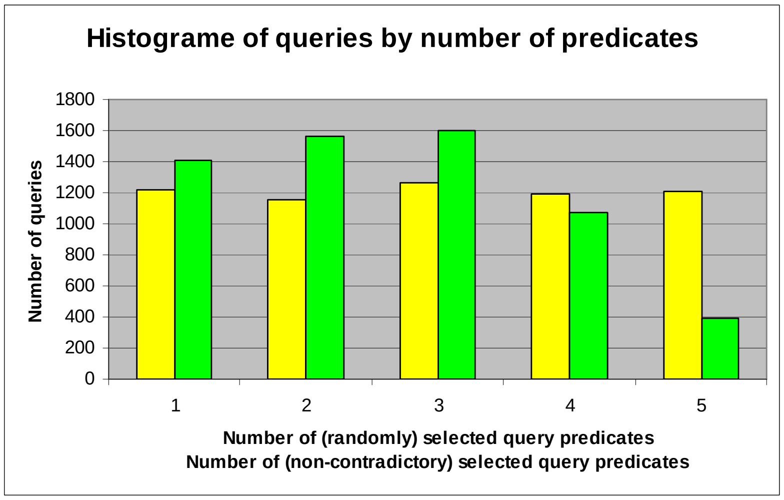


Figure 14 – Number of queries having a certain number of predicates (🟨 after random selection and 🟩 after elimination of contradictory predicates)

After fixing query predicates, the algorithm presented in Sub-section 3.2.2 was executed in order to generate SQL queries. Finally, we added a special query, with query id = 0, that has no predicates.

4.5. Execution of queries

After generating queries, we executed them over the several test sets (for all users and all strategies) and we measured the size of the obtained results. The following figures illustrate this fact for one particular strategy.

Figure 15 shows the number of queries having at least a certain result size. By result size we mean the number of tuples returned by the query. We took two measures: the average of result sizes for all users, and the maximum result size for a user (the user for whom the query returns the most of results). Note that most queries returns less than 20 tuples in average but they return more results for some users.

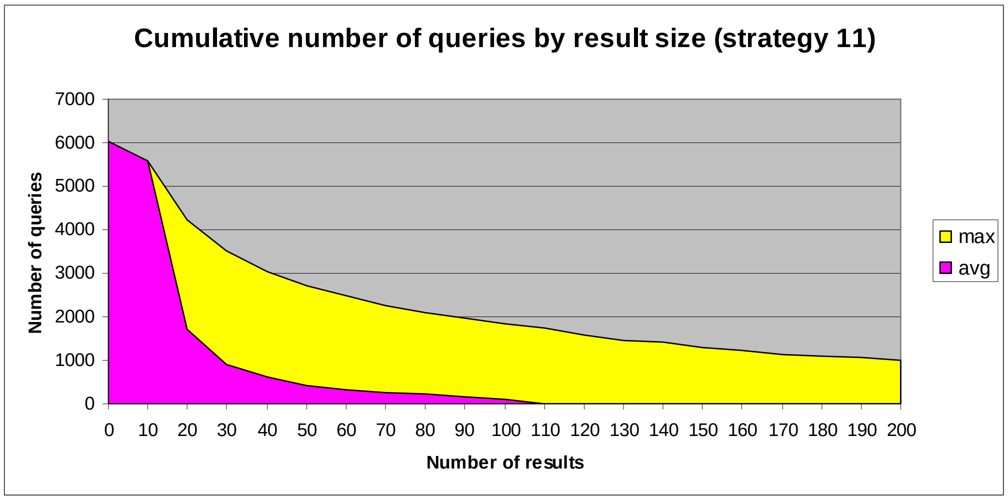


Figure 15 – Number of queries having a certain result size or greater

Figure 16 shows the number of users for whom we obtained at least a certain result size. We also took two measures: the average of result sizes for all queries, and the maximum result size for a query (the query that returns the larger result for the user). Note that most users receive less than 20 tuples in average but they receive more results for some queries.

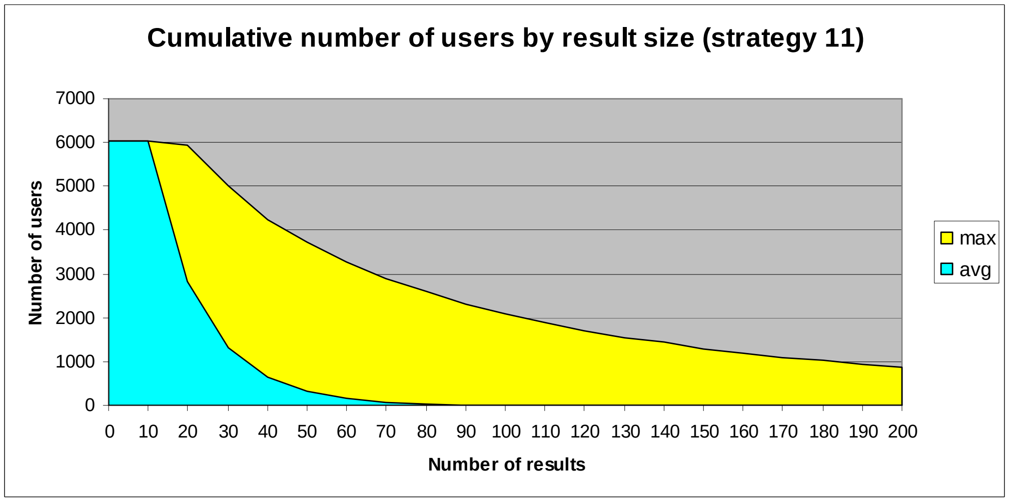


Figure 16 – Number of users for whom we obtained a certain result size or greater

Result sizes are larger for strategies 1 to 10 and smaller for strategies 16 to 20, but distributions of previous figures are conserved.

5. Conclusion

In this report we described the procedure followed for generating a reference data set for query personalization. Specifically, we described the generation of user profiles, the generation of queries and the computation of reference results. We also executed those procedures and showed statistics on the obtained profile predicates, query predicates and result sizes.

This reference data set is large enough to support the definition of several personalization benchmarks, either by limiting the number of predicates in each user profiles or by selecting subsets of users or queries. A first benchmark built over this reference data set is described in [3].

As future work, we aim at considering communities of users having similar profiles and defining community profiles. The queries of the data set may be executed on the test set of a community (the union of the test sets of all users in the community) obtaining queries with larger result sizes. Additional tests may compare the results obtained with the user profile with the results obtained with the community result.

6. References

[1] GroupLens Research: “movielens: helping you to find the right movies”. Web site, ULR: http://movielens.umn.edu, last accessed on July 9th, 2006.

[2] Intenet Movie Database, Inc.: “The Intenet Movie Database”, Web site, URL: http://www.imdb.com/, last accessed on July 9th, 2007.

[3] Kostadinov D. : “Personnalisation de l’information : une approche de gestion de profils et de reformulation de requêtes”. PhD thesis, Université de Versailles Saint-Quentin en Yvelines, December 2007.

[4] Peralta, V.: “Extraction and Integration of MovieLens and IMDb Data”. Technical Report, Laboratoire PRiSM, Université de Versailles, Versailles, France, July 2007.

[5] Text REtrival Conference (TREC). URL: http://trec.nist.gov/, last accessed on September 2007.

[6] Transaction Processing Performance Council. URL: http://www.tpc.org/, last accessed on September 2007.

7. Annexes

7.1. Annex 1 – Procedures for generating profile predicates

Ref_ProfileGeneration

```
CREATE OR REPLACE PROCEDURE Ref_ProfileGeneration

AS
BEGIN

   DELETE from Ref_Aux1;
   COMMIT;

   DELETE from Ref_Aux2;
   COMMIT;

   DELETE from Ref_Predicates;
   COMMIT;

   Ref_ExtractEqualityPred;
   Ref_ExtractInequalityPred;
   Ref_DiscardDummyPred;
   Ref_CountEvaluations;
   Ref_ComputePredWeights;

   COMMIT;

END Ref_ProfileGeneration;
/
```

Ref_ExtractEqualityPred

```
CREATE OR REPLACE PROCEDURE Ref_ExtractEqualityPred

AS
  stmt VARCHAR2(5000);

  parameters Ref_GenerationParameters%rowtype;

  CURSOR cursor_parameters is
    SELECT   *
    FROM Ref_GenerationParameters
    WHERE function = 'Equality';

BEGIN

  FOR parameters IN cursor_parameters LOOP

    stmt:= 'INSERT INTO Ref_Aux2 (
      SELECT       ' || parameters.strategyid || ',' ||
        parameters.ptypeid ||',
        U.userid,
        T.' || parameters.attributename || ',
        =,
        count(distinct U.movieid)
      FROM   I_USERRATINGS U, ' || parameters.lookupview || ' T
      WHERE  T.movieid = U.movieid
      AND    U.rating >= ' || parameters.minrating || '
      AND    U.' || parameters.trainingcondition || '
      GROUP BY U.userid, T.' || parameters.attributename || '
      HAVING count(distinct U.movieid) >= ' || parameters.mineval || '
    )';

    EXECUTE IMMEDIATE stmt;

    COMMIT;

  END LOOP;

END Ref_ExtractEqualityPred;
/
```

Ref_ExtractInequalityPred

```
CREATE OR REPLACE PROCEDURE Ref_ExtractInequalityPred

AS
  stmt VARCHAR2(5000);

  parameters Ref_GenerationParameters%rowtype;

  CURSOR cursor_parameters is
    SELECT   *
    FROM  Ref_GenerationParameters
    WHERE function = 'Inequality';

BEGIN

  FOR parameters IN cursor_parameters LOOP

    stmt:= 'INSERT INTO Ref_Aux2 (
      SELECT       ' || parameters.strategyid || ',' ||
        parameters.ptypeid ||',
        U.userid,
        P.' || parameters.attributename || ',
        >=,
        count(distinct U.movieid)
      FROM   I_USERRATINGS U, ' || parameters.lookupview || ' T, ' ||
        parameters.param1 || ' P
      WHERE  U.movieid = T.movieid
      AND    T.' || parameters.attributename || ' >=
        P.' || parameters.attributename || '
      AND    U.rating >= ' || parameters.minrating || '
      AND    U.' || parameters.trainingcondition || '
      GROUP BY U.userid, P.' || parameters.attributename || '
      HAVING count(distinct U.movieid) >= ' || parameters.mineval || '
    )';

    EXECUTE IMMEDIATE stmt;

    COMMIT;

  END LOOP;

END Ref_ExtractInequalityPred;
/
```

Ref_DiscardDummyPred

```
CREATE OR REPLACE PROCEDURE Ref_DiscardDummyPred

AS
BEGIN

  DELETE FROM Ref_Aux2
  WHERE attributevalue = 'NULL';

  DELETE FROM Ref_Aux2
  WHERE attributevalue = '_unknown';

  DELETE FROM Ref_Aux2
  WHERE attributevalue = '_multiple';

  COMMIT;

END Ref_DiscardDummyPred;
/
```

Ref_CountEvaluations

```
CREATE OR REPLACE PROCEDURE Ref_CountEvaluations

AS
  stmt varchar2(5000);

  parameters Ref_Strategies%rowtype;

  CURSOR cursor_parameters is
    SELECT *
    FROM Ref_Strategies;

BEGIN

  FOR parameters IN cursor_parameters LOOP

    stmt:= 'INSERT INTO Ref_Aux1 (
      SELECT  ' || parameters.strategyid || ',
        U.userid,
        count(*)
      FROM   I_UserRatings U
      WHERE  U.rating >= ' || parameters.minrating || '
      AND    U.' || parameters.trainingcondition || '
      GROUP BY U.userid
    )';

    EXECUTE IMMEDIATE stmt;

    COMMIT;

  END LOOP;

END Ref_CountEvaluations;
/
```

Ref_ComputePredWeights

```
CREATE OR REPLACE PROCEDURE Ref_ComputePredWeights

AS
BEGIN

  INSERT INTO Ref_Predicates (
  SELECT  X1.strategyid,
    X2.ptypeid,
    X1.userid,
    A.tablename,
    A.attributename,
        to_char(X2.attributevalue),
    X2.operator,
    (X2.moviecount/X1.moviecount)*100
  FROM Ref_Aux2 X2, Ref_Aux1 X1, Ref_PredicateTypes A, Ref_Strategies S
  WHERE   X2.strategyid = X1.strategyid
  AND     X2.userid = X1.userid
  AND     X2.ptypeid = A.ptypeid
  AND     X1.strategyid = S.strategyid
  AND  (X2.moviecount/X1.moviecount)*100 >= S.minweight
  );

  COMMIT;

END Ref_ComputePredWeights;
/
```

7.2. Annex 2 – Procedures for generating query predicates

Ref_QueryGeneration

```
CREATE OR REPLACE PROCEDURE Ref_QueryGeneration

AS
BEGIN

   DELETE from Ref_AuxQ1;
   COMMIT;

   DELETE from Ref_AuxQ2;
   COMMIT;

   DELETE from Ref_AuxQ3;
   COMMIT;

   DELETE from Ref_AuxQ4;
   COMMIT;

   DELETE from Ref_QueryPredicates;
   COMMIT;

   Ref_SetQueryPredNumber;
   Ref_SetPredRandomWeight;
   Ref_SelectQueryPred;
   Ref_DeleteQueryConflictivePred;

   COMMIT;

END Ref_QueryGeneration;
/
```

Ref_SetQueryPredNumber

```
CREATE OR REPLACE PROCEDURE Ref_SetQueryPredNumber

AS
BEGIN

   random.rndinit();

   INSERT INTO Ref_AuxQ1 (
   SELECT U.userid, random.rndint(5)+1
   FROM I_UserRatings U
   GROUP BY U.userid
   );

   COMMIT;

END Ref_SetQueryPredNumber;
/
```

Ref_SetPredRandomWeight

```
CREATE OR REPLACE PROCEDURE Ref_SetPredRandomWeight

AS
BEGIN

  random.rndinit();

  INSERT INTO Ref_AuxQ2 (
  SELECT  X.ptypeid,
    X.userid,
    X.tablename,
    X.attributename,
         X.attributevalue,
         X.operator,
    X.weight,
    random.rndflt()
  FROM Ref_Predicates X
  WHERE X.strategyid=0
  );

  COMMIT;

END Ref_SetPredRandomWeight;
/
```

Ref_SelectQueryPred

```
CREATE OR REPLACE PROCEDURE Ref_SelectQueryPred

AS

  preds Test_RuleNumber%rowtype;

  CURSOR cursor_preds is
    SELECT   *
    FROM Ref_AuxQ1;

BEGIN

  FOR preds IN cursor_preds LOOP

    Ref_SelectQueryPred_aux (preds.queryid, preds.prednumber);

  END LOOP;

END Ref_SelectQueryPred;
/
```

Ref_SelectQueryPred_aux

```
CREATE OR REPLACE PROCEDURE Ref_SelectQueryPred_aux(
   query in NUMBER,
   predcount in NUMBER)

AS

   i NUMBER;

   rules Ref_AuxQ2%rowtype;

   CURSOR cursor_rules is
      SELECT   *
      FROM Ref_AuxQ2 T
      WHERE T.queryid = query
      ORDER BY T.rnd DESC;

BEGIN

   i:=0;

   OPEN cursor_rules;

   WHILE i < predcount LOOP

      FETCH cursor_rules into
         rules.ptypeid,
         rules.queryid,
         rules.tablename,
         rules.attributename,
                rules.attributevalue,
         rules.operator,
         rules.weight,
         rules.rnd;
      EXIT WHEN cursor_rules%notfound;

      INSERT INTO Ref_AuxQ3 VALUES (
         rules.ptypeid,
         rules.queryid,
         rules.tablename,
         rules.attributename,
                rules.attributevalue,
         rules.operator,
         rules.weight,
         rules.rnd
      );

      i:= i+1;

   END LOOP;
   CLOSE cursor_rules;

   COMMIT;

END Ref_SelectQueryPred_aux;
/
```

Ref_DeleteQueryConflictivePred

```
CREATE OR REPLACE PROCEDURE Ref_DeleteQueryConflictivePred

AS

BEGIN

   INSERT INTO Ref_AuxQ4 (
   SELECT  ptypeid, queryid, max(rnd)
   FROM Ref_AuxQ3
   GROUP BY ptypeid, queryid
   );

   INSERT INTO Ref_QueryPredicates (
   SELECT  X.ptypeid, X.queryid, X.tablename, X.attributename,
           X.attributevalue, X.operator
   FROM Ref_AuxQ3 X, Ref_AuxQ4 Y
   WHERE X.ptypeid = Y.ptypeid
   AND     X.queryid = Y.queryid
   AND     X.rnd = Y.rnd
   );

   COMMIT;

END Ref_DeleteQueryConflictivePred;
/
```