

Ranking In-Document, Off-Document and Persistent Vectors

Document Version: 1.0
Contact: ixe@ideare.com
Written by: spagnolo@ideare.com
Revised by: savona@ideare.com
Approved by: gulli@ideare.com

Summary

Premis.....	3
1. Ranking In-Document.....	3
2. Ranking Off-Document.....	3
3. IXE and Rank Off-Document	3
4. IXE: an IR system similar to a full OODBMS.....	4
5. A practical example: indexing and searching a web directory.....	5
5.1 Defining the schema.....	5
5.2 The indexing phase.....	6
5.3 Search phase.....	7
5.4 Storing into the Persistent Vector store.....	8
6. Logical Delete	9

Ranking In-Document, Off-Document and Persistent Vectors

Premis

This article presupposes a knowledge of the concepts contained in “*How to write a Search Application on IXE. Basic introduction: the video index case*”, “*How to write a Search Application on IXE. Advanced introduction: the web index case*” and “*Library Architecture*”.

1. Ranking In-Document

The IXE *rank* function allows us to order search results according to a document’s “relevance” with regard to keywords.

A document’s rank is computed at *search* time on the basis of information obtained from the indexed document. More particularly, rank is a function of the keywords’ frequency in the document, their location, therefore the possible *color* and respective *proximity*.

We can call this technique *Ranking In Document* as it is based solely on a document’s structure and its *internal* information.

2. Ranking Off-Document

The following pages will introduce a class of techniques known as *Ranking Off-Document*. This technique was developed to meet the needs of ranking on the basis of various parameters and not necessarily due to a document’s internal structure. For example, it may be interesting to rank a document on the basis of hits (*user popularity*), number of links to the document (*link popularity*), or other criteria of authoritativeness as well as information such as date, size and so on. Such information is not always obtainable from the document’s structure at search time but may, somehow, be stored before the search itself.

This class of techniques is know as *Off-Documentation* precisely because it gives weight to each document based on information external to it.

3. IXE and Rank Off-Document

To give weight to each document the easiest technique would be to store the off-document rank in a field of the document’s store (.bdb file) of the index to be built. However, this solution is not optimal in performance for two reasons:

- Management of the memory cache is not optimized to deal with rank. This causes problems of cache invalidation and consequent low percentage of cache hits.
- It is necessary for the search application to access each document's store for all documents satisfying the search query in order to obtain the off-document rank value. This value is used to sort results. This leads to a drop in performance caused by accessing the document store.

IXE supports advanced procedures for efficient management for off-document ranking.

It uses a suitable data structure, the *Persistent Vector Store*, which stores off-document rank information for each index entry (*DocID*).

We will describe a method that allows us to efficiently associate to a document information related to its authoritativeness. This information can be user-defined in the application, starting from various sources, before the search phase and can be combined, at search time, by user-defined rank functions.

4. IXE: an IR system similar to a full OODBMS

Ranking Off-Document has extended IXE's capabilities, beyond dealing with relational tables for objects to objects with nested objects. This makes IXE more akin to a *full OODBMS*.

The user can employ classes containing pointers to other objects while IXE will take care of storage and retrieval.

For example, it is possible to associate a structure containing rank values (a *PersistentVector*) to each document.

As specified in "How to write a Search Application on IXE. Basic introduction: the video index case", in IXE each document's format is described by an object derived from a metaclass definition. Within the metaclass it is possible to use a pointer to Persistent Vector type structures. At indexing time the Persistent Vector is saved, in a transparent fashion, in a `<table>.<name_of_the_reference>` file.

This results in the creation of an index consisting of three files (.bdb, .fti and .pst) and an additional file for every *PersistentVector* reference used in the class definition.

During a search, the persistent Vector is retrieved, mapped in the cache and can be used to compute the user-defined rank functions.

This will enable the insertion and/or modification of the rank value by simply retrieving the *DocID* value from the index, with reference to the key we are interested in, and using it to access the file containing the ranking.

The IXE library manages the memory cache so as to reduce access times to off-document rank files for the most frequently accessed documents.

The `SearchCollectionOf` class has been supplied with a `SearchCollectionOf::computeRank(res)` method that is invoked before the search results are ordered. Clearly the user can define a different `computeRank()` version by specializing on the method.

5. A practical example: indexing and searching a web directory

We will use an off-document rank example, an application for indexing and searching a web directory.

5.1 Defining the schema

First of all a *PersistentVector* must be defined in which to store the user-defined ranking. An object of this kind must be included in the *CatInfo* class definition. Below are the definitions of the *CatInfo.h* file.

```
struct RankWeight {
    RankWeight() : abstract_rank(0), avs_rank(0) { }
    nat8         abstract_rank; // Classified URL abstract rank
    nat8         avs_rank;     // Classified URL rank
};
```

In this case the rank, stored as Persistent Vectors, are simply a pair of *nat8* type values. We decide that internally *avs_rank* will be used to order and *abstract_rank* in the case of two *DocIDs* with the same *avs_rank* value.

```
class CatInfo : public DocInfo {
public:

    bool operator ==(DocInfo const &other) {
        return (size == other.size) && (time == other.time);
    }

    std::ostream& serialize(std::ostream& s) const;
    std::ostream& XML(int index, std::ostream& out) const;

    char const*    url;           // Classified URL
    Text<512>      title;         // Classified URL title
    Text<4096>     abstract;      // Classified URL abstract
    Text<2048>     keywords;      // Classified URL keywords
    Text<1024>     categoria;     // Classified URL category
    nat2          linguaggio;     // Classified URL language
    Text<512>     idc;           // Classified URL idc
    nat2          topsite;       // Classified URL topsite
    RankWeight*   rank;        // rank is a pointer to struct ==>
                                // stored on Persistent Vector Store

    META(CatInfo,
        (SUPERCLASS(DocInfo),
        VARFIELD(url,          512),
        KEY(title,             Field::fulltext),
        KEY(abstract,          Field::fulltext),
        KEY(keywords,          Field::fulltext),
        KEY(categoria,         Field::fulltext),
        FIELD(linguaggio),
        KEY(idc,                Field::fulltext),
        FIELD(topsite),
```

```

        FIELD(rank)          // One Persistent Vector
    )
);
};

```

Having included a reference like:

```
RankWeight* ranks;
```

In the *CatInfo* class definition creates a `<table>.<name_of_the_reference>` file in which the the sequence of Persistent Vectors is memorized. In this case the new index will consist of four files with the following extensions:

```
.bdb
.fti
.pst
.rank
```

The user is free to define, within his own application, different types of Persistent Vectors that will be saved, each in its own file and independently mapped in the memory.

5.2 The indexing phase

Now let's see what happens during the indexing phase. The *CatInfo* class is used in the *indexCat.cpp* file. A *Persistent Vector* is also defined:

```

CatInfo CatInfo;
...

RankWeight rw;    //For store ranks on Persistent Vector
...

field = ::strtok(0, "\\t\\0");
if (verbosity > 3)
{
    cerr << "\\t[" << field << "]\n";
}
rank = field ? atoi(field) : 0;
rw.abstract_rank = (rank > 0) ? rank : 0;

field = ::strtok(0, "\\t\\0");
if (verbosity > 3)
{
    cerr << "\\t[" << field << "]\n";
}
rank = field ? atoi(field) : 0;
rw.avs_rank = (rank > 0) ? rank : 0;
//Set ranks
CatInfo.rank = &rw;
...

table.insert(&CatInfo);

```

The *rw* object is filled with the appropriate data and assigned a *CatInfo.rank*. thus the call to method *Table::insert* saves the information in the *.rank* file.

5.3 Search phase

Let us now see how the off-document rank influences the search (with reference to the *searchCat.cpp* file).

First a *CatCollection* class must be defined.

```
class CatCollection : public SearchCollectionOf<CatInfo>
{
public:
    CatCollection(std::vector<std::string>& tableNames) :
        SearchCollectionOf<CatInfo>(tableNames) {
        //Init rankTables from mappedFile
        FOR_EACH (std::vector<Collection<CatInfo>*>, collections, collection)
            rankTables.push_back((RankWeight*)
                (*collection)->mappedFields[0].table->begin());
    }

    CatCollection(std::vector<Collection<CatInfo>*>& collections) :
        SearchCollectionOf<CatInfo>(collections) {
        //Init rankTables from mappedFile
        FOR_EACH (std::vector<Collection<CatInfo>*>, collections, collection)
            rankTables.push_back((RankWeight*)
                (*collection)->mappedFields[0].table->begin());
    }
}
```

ranktables definition is included in the definition for the class:

```
std::vector<RankWeight*> rankTables;
```

From the constructor code we can see how pointers to the *.rank* file are saved in the relevant *rankTables* table when the *CatCollection* is instanced.

The operation:

```
rankTables.push_back((RankWeight*)
    (*collection)->mappedFields[0].table->begin());
```

maps the *Persistent Vector* in the memory. Thus it is possible to access rank files by simply specifying the collection name and document's *DocID*.

The *SearchCollectioOf* class has been given a *SearchCollectioOf::computeRank(res)*. The user can specialize this method to suit his needs. The definition of the *CatCollection* class, for example, also includes the *computeRank* method defined by the user and is used transparently by IXE. In this way two *nat8* type values are used to build a single value where the integer part is shown by *avs_rank* and the decimals are shown by *abstract_rank*.

```

/*-----
 * Compute new rank for QueryResult
 *
 * The new rank is a double value with integral component equal to
avs_rank,
 * and fractional value equal to abstract_rank.
 *-----*/
bool      computeRank(QueryResult& res)
{
    RankWeight* rw= &rankTables[res.collection()][res.ID() - 1];
    char srank[32] = "";
    ::sprintf(srank, "%d.", rw->avs_rank);
    ::sprintf(&srank[::strlen(srank)], "%d", rw->abstract_rank);
    double new_rank = ::atof(srank);
    res.rank(new_rank);
    return new_rank > 0.0;
}

```

The *computeRank* method:

- Sets the *res.rank* value which is used by IXE for sorting.
- Returns a Boolean value that is used by IXE to either include this result too or exclude it for the logical delete operation.

At search time invoking *SearchCollectioOf::select* calls the *CatCollection::computeRank* method specialized by the user. Before the output of the results the application must read the rank values to correctly setup the *CatInfo* field.

```

catInfo->rank = &rankTables[rit->collection()][rit->ID() - 1];

```

The rank values can thus be used to draw up an output of the search results.

5.4 Storing into the Persistent Vector store

The .rank file produced by the indexing procedure is a sequence of data that can be viewed using a hexadecimal editor. For example:

```

>> od -x viaggi.rank | more

0000000 000a 0000 0000 0000 0030 0000 0000 0000
0000020 0014 0000 0000 0000 0027 0000 0000 0000
0000040 000a 0000 0000 0000 0019 0000 0000 0000
0000060 000a 0000 0000 0000 0020 0000 0000 0000
0000100 000a 0000 0000 0000 002e 0000 0000 0000
0000120 0014 0000 0000 0000 0024 0000 0000 0000
0000140 000a 0000 0000 0000 0030 0000 0000 0000
0000160 000a 0000 0000 0000 001a 0000 0000 0000

```

In this example each line represents the contents of a *RankWeight* structure. In particular, in the example we provided, the first 8 bytes are the *abstract_rank* while the second 8 bytes are the *avs_rank*.

The files' structure is closely tied to the schema's definition, or to be more precise, the ranks' definitions. Varying the definition:

```
struct RankWeight {
  RankWeight() : abstract_rank(0), avs_rank(0) { }

  nat8          abstract_rank; // Classified URL abstract rank
  nat8          avs_rank;      // Classified URL rank};
```

we modify the way in which data is saved in the *Persistent Vector Store*.

So it is clear the user can write an application to edit and modify the file according to his needs even after the indexing stage. The steps for doing this are:

- Determine the *docID* of the document to modify, using method `table::getDocID()`.
- `mmap` the `.rank` file, i.e.:

```
mappedFile mf("filename", ios::in | ios::out);
DocRank* docRanks = (DocRank*)mf.begin();

docRank[docID] = ...
```

- assign new values for the rank:

```
docRanks[docID] = DocRank(x, y, z, ...);
```

6. Logical Delete

Off-document ranking, described here, can also be used for *logical delete* of documents contained in the index. Thus it becomes possible to prevent documents being returned as search results, although they are not removed from the index physically.

If the user knows the *DocID* of the document he wishes to cancel, he can edit and modify the rank files so that the final rank value for that document, as calculated by the *ComputeRank* method is null. As a result, the search application will always ignore documents with a null value and will not include them in the results.