

Ranking In-Document, Off-Document e 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

Indice

1.	Ranking In-Document	3
2.	Ranking Off-Document	3
3.	IXE e Rank Off-Document	4
4.	IXE: un sistema di IR simile ad un full OODBMS	4
5.	Un esempio d'uso: indicizzazione e search di una web directory	5
5.1	Definizione dello schema	5
5.2	Fase di indicizzazione	6
5.3	Fase di search	7
5.4	Memorizzazione nel Persistent Vector Store	9
6.	Delete logica	10

Ranking In-Document, Off-Document e Persistent Vectors

Premessa

La trattazione di questo argomento presuppone la conoscenza di concetti di base per i quali si rimanda ai documenti: “*Come scrivere un’applicazione di Search con IXE. Introduzione: il caso di un indice per il video*”, “*Come scrivere un’applicazione di Search con IXE. Introduzione avanzata: il caso di un indice per il web*” e “*Library Architecture*”.

1. Ranking In-Document

La funzione di *rank* disponibile in IXE consente di ordinare i risultati della ricerca in modo da rispettare la “rilevanza” dei documenti, in relazione ai termini ricercati.

Il rank di un documento viene calcolato a tempo di *search* sulla base di informazioni ricavate dal contenuto del documento indicizzato. In particolare, il rank è funzione del numero di occorrenze delle keywords ricercate all’interno del documento, della loro posizione, ovvero dell’eventuale *colore* e della rispettiva *proximity*.

Questa tecnica può essere definita *Ranking In Document* perché si basa esclusivamente sulla struttura del documento e sulle informazioni *interne* al documento stesso.

2. Ranking Off-Document

Nelle pagine seguenti verrà presentata una classe di tecniche nota come *Ranking Off-Document*. Questo approccio nasce dall’esigenza di calcolare il rank in base a parametri diversi e non necessariamente legati alla struttura interna del documento. Ad esempio, può essere interessante calcolare il rank di un documento in base al numero di accessi allo stesso (*user popularity*), oppure al numero di link che puntano al documento esaminato (*link popularity*), oppure ad altri criteri di autorevolezza o altre informazioni quali la data, la dimensione e così via. Informazioni di questo tipo non sono necessariamente dedotte dalla struttura del documento a tempo di *search*, ma possono essere, in qualche modo, memorizzate prima della ricerca vera e propria.

La classe di tecniche viene definita *Off-Document* appunto perché attribuisce un peso a ciascun documento sulla base di informazioni *esterne* al documento stesso.

3. IXE e Rank Off-Document

Per associare un peso a ciascun documento, la soluzione più semplice è quella di memorizzare il rank off-document in un campo opportuno del document store (ovvero nel file .bdb) dell'indice da costruire. Tale soluzione, tuttavia, risulta essere sub-ottimale in termini di prestazioni per due motivi principali:

- La politica di gestione della memoria cache non è ottimizzata per il trattamento dei rank. Ciò causa problemi di invalidazione della cache e, di conseguenza, basse percentuali di cache hits.
- È necessario che l'applicazione di search acceda al document store per ogni documento che soddisfi la ricerca, al fine di recuperare il valore del rank off-document. Tale valore viene poi usato per l'ordinamento dei risultati. Ciò comporta un degrado di performance derivante dall'accesso al document store.

IXE supporta meccanismi avanzati per la gestione efficiente del rank off-document.

L'implementazione utilizza un'opportuna struttura dati, il *Persistent Vector Store* che, per ogni elemento dell'indice, ovvero per ogni *DocID*, memorizza le informazioni relative al rank off-document.

Nel seguito descriveremo, dunque, una metodologia che ci consentirà di associare in modo efficiente, a ciascun documento, informazioni relative alla sua autorevolezza. Tali informazioni potranno essere definite a livello applicativo dall'utente attingendo a varie fonti, prima della fase di ricerca vera e propria, e potranno essere combinate a tempo di search per mezzo di funzioni di rank definite dall'utente.

4. IXE: un sistema di IR simile ad un full OODBMS

L'introduzione del Ranking-Off Document ha esteso IXE in modo da consentire il trattamento non solo di tabelle relazionali di oggetti, ma anche di oggetti con oggetti annidati. In questo modo IXE è più vicino ad un *full OODBMS*.

L'utente può usare classi contenenti puntatori ad altri oggetti ed IXE si occuperà di memorizzarli e recuperarli.

Per esempio, è possibile associare ad ogni documento una struttura contenente i valori di rank, cioè un *PersistentVector*.

Come specificato in "*Come scrivere un'applicazione di Search con IXE. Introduzione: il caso di un indice per il video*", in IXE il formato di ogni documento è descritto da un oggetto derivato da una definizione di metaclassa. All'interno della metaclassa è possibile usare un puntatore a strutture di tipo *Persistent Vector*. Al momento dell'indicizzazione il *Persistent Vector* viene salvato, in modo trasparente, in un file `<table>.<name_of_the_reference>`.

Il risultato di tutto questo è che verrà prodotto un indice costituito dai tre files che conosciamo (.bdb, .fti e .pst) ed in più un file per ogni riferimento ad un *Persistent Vector* usato nella definizione della classe.

In fase di search, il *Persistent Vector* viene recuperato, mappato in cache e può essere usato per calcolare le funzioni di rank definite dall'utente.

L'uso di questa tecnica consentirà di inserire e/o modificare il valore dei rank semplicemente ricavando dall'indice il valore del *DocID*, relativo alla chiave che ci interessa, ed usandolo per accedere al file contenente i rank.

La libreria IXE si occuperà di gestire un meccanismo di cache in memoria che riduca i tempi di accesso al file dei rank off-document, per quei documenti più frequentemente acceduti.

La classe *SearchCollectionOf* è stata dotata di un metodo *SearchCollectionOf::computeRank(res)* che viene invocato prima che i risultati della ricerca vengano ordinati. Ovviamente, specializzando il metodo l'utente può definire una diversa versione di *computeRank()*.

5. Un esempio d'uso: indicizzazione e search di una web directory

Useremo come esempio di rank off-document, un'applicazione di indicizzazione e search di una web directory.

5.1 Definizione dello schema

Prima di tutto occorre definire un *PersistentVector* in cui memorizzare i rank definiti dall'utente. Un oggetto di questo tipo deve essere incluso nella definizione della classe *CatInfo*. Di seguito riportiamo le definizioni contenute nel file *CatInfo.h*.

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

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

In questo caso i rank, memorizzati come Persistent Vectors, sono semplicemente una coppia di valori di tipo *nat8*. Decidiamo che, internamente, *avs_rank* verrà usato per l'ordinamento e *abstract_rank* verrà usato nel caso di due elementi con lo stesso valore di *avs_rank*.

```
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
      )
);
};

```

Il fatto di aver incluso nella definizione della classe *CatInfo* un riferimento del tipo:

```
RankWeight* ranks;
```

produrrà la creazione di un file `<table>.<name_of_the_reference>` in cui verranno memorizzate le sequenze di *PersistentVector*. Nel caso in questione, il nuovo indice sarà composto da quattro files con le estensioni:

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

L'utente è libero di definire, all'interno della propria applicazione, differenti tipi di *Persistent Vectors* che verranno salvati, ciascuno su un proprio file e mappati in memoria in modo indipendente.

5.2 Fase di indicizzazione

Vediamo ora cosa succede durante la fase di indicizzazione. Nel file *indexCat.cpp* viene usata la classe *CatInfo* e viene definito anche un *Persistent Vector*:

```

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);

```

L'oggetto *rw* viene riempito con i dati opportuni ed assegnato a *CatInfo.rank*. In questo modo, la chiamata al metodo *Table::insert* salva i dati nel file *.rank*.

5.3 Fase di search

Esaminiamo adesso come l'introduzione del rank off-document influenzi la search (si faccia riferimento al file *searchCat.cpp*).

Prima di tutto occorre definire una classe *CatCollection*.

```

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());
    }
}

```

La definizione di *rankTables*, la tabella di rank, è inclusa nella definizione della classe:

```

std::vector<RankWeight*>    rankTables;

```

Dal codice del costruttore vediamo come, al momento dell'istanziamento della classe *CatCollection*, vengano salvati i puntatori al file *.rank* nell'apposita tabella *rankTables*. L'operazione:

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

mappa in memoria il *Persistent Vector*. In questo modo, si potrà accedere al file dei rank semplicemente specificando come coordinate il nome della collezione ed il *DocID* del documento.

La classe *SearchCollectionOf* è stata dotata di un metodo *SearchCollectionOf::computeRank(res)*. Tale metodo può essere specializzato dall'utente al fine di renderlo adatto ad esigenze particolari.

La definizione della classe *CatCollection*, ad esempio, include anche il metodo *computeRank* definito dall'utente ed usato da IXE in maniera trasparente. In questo caso vengono usati due valori di tipo *nat8* per costruire un valore unico di cui *avs_rank* rappresenta la parte intera ed *abstract_rank* la parte decimale.

```
/*-----
 * 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;
}
```

Il metodo *computeRank*:

- setta il valore di *res.rank* che viene usato internamente da IXE per effettuare l'ordinamento
- restituisce un valore booleano che viene usato da IXE per includere anche questo risultato oppure escluderlo per l'operazione di delete logica.

A tempo di search, l'invocazione del metodo *SearchCollectionOf::select*, provocherà la chiamata del metodo *CatCollection::computeRank*, specializzato dall'utente.

Infine, prima di stampare i risultati, l'applicazione di search deve leggere i valori dei rank per settare opportunamente il campo della *CatInfo*.

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

I valori di rank possono essere così usati per corredare i risultati della ricerca stampati come output.

5.4 Memorizzazione nel Persistent Vector Store

Il file `.rank`, prodotto dall'indicizzazione, è semplicemente una sequenza di dati che può essere facilmente visualizzata con un editor esadecimale. Ad esempio:

```
>> 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 questo esempio, ogni linea è il contenuto di una struttura *RankWeight*. In particolare, per il caso che abbiamo descritto, i primi 8 bytes rappresentano *abstract_rank*, i secondi 8 bytes sono invece *avs_rank*.

La struttura del file è, dunque, strettamente legata alla definizione dello schema o, più precisamente, alla definizione dei ranks. Variando la definizione:

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

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

si modifica il modo in cui i dati vengono salvati nel *Persistent Vector Store*.

Partendo da questa considerazione, è chiaro che per l'utente è possibile scrivere un'applicazione per editare il file e modificarlo a seconda delle necessità, anche in una fase successiva all'indicizzazione. I passi da seguire sono i seguenti:

- Determinare il docID del documento da modificare, usando il metodo `table::getDocID()`.

- Fare un mmap del file `.rank`, cioè:

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

docRank[docID] = ...
```

- Assegnare il nuovo valore di rank:

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

6. Delete logica

L'implementazione delle tecniche di ranking off-document, descritta in queste pagine, può essere sfruttata per ottenere anche la *cancellazione logica* di documenti presenti nell'indice. In altri termini, è possibile fare in modo che alcuni documenti, pur non essendo rimossi *fisicamente* dall'indice, non vengano mai restituiti come risultati di search.

Conoscendo il *docID* del documento che vuole cancellare, l'utente può intervenire editando e modificando il file che contiene i rank, per fare in modo che il valore finale del rank per quel documento, calcolato dal metodo *computeRank*, sia nullo. Come conseguenza, l'applicazione di search ignorerà sempre il documento con valore di rank nullo e non lo inserirà tra i risultati.