Relevant Filtering in a Distributed Content-based Publish/Subscribe System

8.1. Introduction

Sources of information are multiplying on the Web for several years, especially due to the success of news portals and social networks which produce information in real time. These flows of information can be kept and processed, often in *RSS* [RSS 03] and *Atom* [Ato 07] formats. But it turns out that nowadays the amount of data which has to be analyzed daily is so large [HME 11] that a user may miss information of interest. Thus, a given user can be lost with such an amount of sources and frequency of updates [TRA 14]. Pub/Sub systems (Redis [CAR 13], Scribe [ROW 01], Siena [CAR 01], Echo [EIS 00]) have been designed to face the problem of aggregating and delivering information of interest (bookmarks and topics) to end users.

For these reasons, we advocate a *content-based Publish/Subscribe* paradigm for Web 2.0 syndication in which information consumers are decoupled (in both *space* and *time*) from feeds (produced flows of items) and they can express their interest with *keyword-based subscriptions* which are computed using content-based filtering. However, even if information is filtered through the matching process, users remain flooded by notifications [HME 11]. Some propositions enhance the filtering process by removing redundant information (*i.e.*, *Novelty* [ZHA 02, CLA 08]) and/or by taking into account information diversification of the delivered items (*i.e.*, *Diversity* [DRO 09b, DRO 09a, PRI 08]) which is generally presented as a *top-k* issue. However, the Pub/Sub context discards traditional *top-k* approaches due to real-time notifications and the time constraint which cannot remove a notified item from the past.

Chapter written by Cédric DU MOUZA and Nicolas TRAVERS.

Very few works have been proposed to take into consideration both relevance, novelty and diversity in a *real-time* Pub/Sub context. Our Pub/Sub system `FiND` [HME 15] addresses this with a two-step process: matching and filtering. For matching keyword-based subscriptions, we assume the existence of an index [HME 12]. The second step is the core of this chapter and aims at computing items' content already delivered to a user to filter new incoming items. The difficulty for keeping a real-time Pub/Sub system is to evaluate novelty and diversity on-the-fly for every incoming item.

This chapter focuses on the way to enhance relevance of filtering and to integrate such a process in two different implementations: a centric-based version and a distributed version in a NoSQL environment. Our contributions in this paper are:

– definitions for novelty and diversity in this particular context, along with a proposal for a weighting score (*Term Discrimination Values* - *TDV*) adapted to the characteristics of items and subscriptions;

– an efficient filtering algorithm for *real-time* Pub/Sub systems based on novelty and diversity which exploits redundancy between subscriptions' history. Two optimized implementations are proposed in centralized and distributed contexts;

– a validation which highlights the complementarity of novelty and diversity both in centralized and NoSQL environments;

– enhancement of *TDV* [WIL 85] computation by proposing incremental versions in a distributed environment.

The chapter is organized as follows. An overview on novelty and diversity process in pub/sub techniques is detailed in Section 8.2. Then we present the data model (Section 8.3) on which relevance in notifications is based and developed in Section8.4 by introducing novelty and diversity. Two distinct implementations are detailed in Section 8.5 and compared in Section 8.7. Section 8.6 deals with TDV scores update. We conclude in Section 8.8.

8.2. Related Works: Novelty and diversity filtering

In Pub/Sub systems, users are often faced with an issue on notifications which turn to be flooded by information. To enhance the quality of filtering, two properties on information were proposed: *Novelty* and *Diversity*. In filtering by novelty, the objective is to discard an item whose information has already been notified (truncate or a similar content). *Diversity* captures a complementary kind of redundancy since it measures whether the information contained in a given item is globally present in the set of recently notified items or not. We present here how those two properties are used in litterature, first on the Web and then in a Pub/Sub context.

When searching a document on the Web, we assume static and already known documents. The objective is to present to users the k most relevant and diverse documents matching a query. To achieve this, some models are based on probabilities for diversity [ANG 11] or on graphs for computing distance [DRO 12a]. Some of them propose to modify diversity measures by focusing on non common attributes between items based on user-defined filters [YU 09], by defining a trade-off between similarity and diversity [SMY 01], by integrating entities and sentiment in a *Greedy* Max-Min algorithm [ABB 13], by defining time-based distances with a gaussian similarity [KEI 12], or by comparing an item with the compression of all previous texts like the NCD distance [CAR 10]. Globally, these techniques allow to compute large texts in a static Top-k evaluation and cannot adapt to our context since we consider small items, which changes relevance of previous methods. Moreover real-time delivery of information is an important constraint that cannot be ignored.

Some approaches focus on continuous filtering like in the Pub/Sub context, combined with Top-k techniques. They may be based on fixed size windows in order to garantee the amount of items to keep in the system like [DRO 12b] which uses a dynamic index to quickly find if an item is diverse or not on a frequently updated snapshot of items, [GAB 04] which focuses only on novelty with extracted entities from items, [MIN 11] which presents an incremental approach for time-based diversification or [PAN 12] which extract topics from items for a simple coverage distance. However fixed-size windows hardly manage different notification rates for subscriptions. In fact, low rates will keep very old items to filter out incoming items and high rates will remove recent items which should remove duplicates.

The closest approach from our solution [DRO 09b] (detailed in Section 8.5.1) uses Top-k windows to compute diversity on real-time delivery. It is based on the interchange algorithm which notifies an item if exchanging it from the previous Top-k levels up the diversity. However this solution can deliver items from previous windows if considered as non-diverse, or remove items from the past for future filtering steps. Our experiments illustrate that this approach tends to locally diversify information, but not over time. Moreover, keeping all items will lead to scale up issues.

8.3. A Publish/Subscribe data model

Our pub/sub data model relies on the fact that published items are mainly text oriented. Then, subscriptions are *long lasting* (continuous) queries under the form of keyword-based subscriptions. Whenever a news item is published, it gets evaluated against the set of subscriptions submitted to the system and for every matching subscription the corresponding subscriber is notified.

8.3.1. *Data model*

The set of stored subscriptions is denoted by $\mathcal{S}$ and their total number by $|\mathcal{S}|$. Each subscription $s \in \mathcal{S}$ includes a set of distinct terms from a vocabulary $\mathcal{V}_S = \{t_1, ..., t_n\}$. $\mathcal{I} = [I_1, ...I_m]$ denotes the feed of incoming items. Items $I \in \mathcal{I}$ are also formed by a set of terms ($I \subseteq \mathcal{V}_I$, with $\mathcal{V}_I$ the vocabulary of items). In this context, a match occurs if and only if all of the terms of a subscription s are also present in a news item I (i.e., broad match semantics).

Table 8.12: Example of keyword based subscriptions

Subscription	s_1	s_2	s_3	s_4
Terms	$t_1 \wedge t_2 \wedge t_4$	$t_1 \wedge t_3$	$t_1 \wedge t_2 \wedge t_5$	$t_2 \wedge t_4$

Consider the set of subscriptions $\mathcal{S}$ illustrated in Table 8.12. Matching item $I = \{t_2, t_4, t_5\}$ against $\mathcal{S}$ will result in the set of matched subscriptions $\mathcal{M} = \{s_4\}$ since t_2 and t_4 of s_4 are contained in I. The matching process has been fully developped in [HME 16] by proposing tree-based structures to evaluate subscriptions. This paper goes one step further by enhancing notifications by integrating a relevance feature.

Processing textual content requires to take into account term weights for items and subscriptions. However, traditional techniques in Information Retrieval [BAE 99] cannot be adapted to our context. In the following sections, we will develop the TDV weighting approach adapted to the pub/sub context, which is more convenient for the filtering step.

8.3.2. *Weighting terms in textual data flows*

The partial matching process and filtering quality addressed by our system must take into account a weighting for terms in order to compute scores. In view of the foregoing this weighting must be computed at low cost as well as relevant. Several term weighting models are proposed in the literature like the *Term Frequency* (TF - frequency of a term in a document) combined with *Inverse Documents Frequency* (IDF - inverse frequency of a term in all documents) [BAE 99], the *Term Discrimination Value* (TDV - see below) [SAL 75] or the Term Precision (TP - number of relevant vs non-relevant terms) [BOO 74].

In the context of this work, we rely on the *TDV* weighting function which is more adapted to the quality of vocabularies in Web Syndications Systems. In fact, an item is a short set of terms where term frequencies cannot be used, so the *tf/idf* standard function is unsuitable (experimentally prooved in Section 8.5.1). Moreover, the *TDV* weighting function measures how a term helps to distinguish a set of documents (*i.e.*

the term influence on the global entropy). So basically for our subscriptions set, neither a very frequent term (present in many subscriptions, so this term is not a selective filter for subscriptions), nor a very uncommon term (present in very few subscriptions, so assuming this is not a typo, it will probably never lead to a notification) have an important TDV value. Finally, the simplicity of computation is all the more important, since we only have to compute a sum of weights for each subscription (Section 8.5.1.2).

Definition 7 (TDV)**.** *The discrimination value for a term t_k is the difference between the vector-space density of the occurrence matrix with t_k and the density of the matrix without t_k. So assuming a similarity distance $sim(I_1, I_2)$ between items, like for instance the cosine of the Euclidian distance, we compute the density as the average pairwise similarity between distinct items:*

$$\Delta(\mathcal{I}) = \frac{1}{|\mathcal{I}| \times (|\mathcal{I}| - 1)} \sum_{i=1}^{|\mathcal{I}|} \sum_{j=1 \wedge j \neq i}^{|\mathcal{I}|} sim(I_i, I_j)$$

Finally the TDV value for a term t_k is:

$$tdv(\mathcal{I}, t_k) = \Delta(\mathcal{I} - \{t_k\}) - \Delta(\mathcal{I})$$

For the sake of simplicity, we denote the TDV value for the term t_k $tdv(t_k)$ instead of $tdv(\mathcal{I}, t_k)$. Based on this function, we can weight the different terms of items and subscriptions. Each term weight w_i is the TDV value $tdv(i)$ normalized by the sum of TDV values of the query terms.

Of course, computing and updating the TDV is a real time consuming process but it can be done in parallel with the filtering process. For clarity purpose, we focus now on the way to integrate TDV in novelty and diversity computation and then detail in Section 8.6 the way to optimize TDV updates incrementally in a NoSQL environment.

8.4. Publish/Subscribe Relevance

In *top-k* approaches, notified items are computed on the whole set of items, leading to delays for items delivery. In our approach, we consider the set of notified items for a given subscription, so-called subscription *history*, and we use this history to filter out in real time the incoming item just after the matching process. This section presents our approach and the definitions adopted, and the instantiation is presented in Section 8.5.

8.4.1. *Items and Histories*

In our context, we define an item as a set of terms. Each term is associated to a term weight denoted by w_i which is used to compute distances and similarities. To

compute novelty and diversity, a Pub/Sub system must keep already notified items, also called *subscription history* H. Each one is a time-ordered set of items linked to a subscription. Each time an item is notified for a subscription, it is added to its history.

8.4.2. *Novelty*

The objective when filtering by novelty is to discard an item that does not contain new information with respect to items in the subscription history, *i.e.*, an item I with a truncate or a similar content of a previous item I'. Since, in our context, history is time dependent, the measure of novelty *new(I, I')* should be asymmetric [ZHA 02] to test how new an incoming item is w.r.t. an existing one and not conversely. Finally, we define the novelty of an incoming item I with respect to an existing history H by comparing I with all items in H, one by one.

Definition 8 (Novelty item-history)**.** *Given a history of items H, an item I, and a novelty item-item measure new (like the one proposed by Definition 10), I is said new with respect to H iff:*

$$\forall I' \in H,\ new(I, I') \geq \alpha$$

We assume that the novelty threshold α is a parameter fixed for the user for his subscription according to the defined or required output rate of items. Section 8.7 will show impact on the filtering rate and quality, histories, and performances in our system.

8.4.3. *Diversity*

Diversity captures a complementary kind of redundancy (towards novelty) since it measures whether the information contained in a given item is globally present in the set of notified items or not (segmented information). The objective is to detect whether an incoming item conveys new information regarding the subscription's history of notified items. Users objective is to receive only the interesting items with different information; the items are filtered by their content. The degree of diversity of an item for a user w.r.t. its subscription history is measured as how much it can increase the average pairwise distance *dist(I,I')* between the history's items [DRO 09a]. Observe that to keep $D(H)$ and $D(H')$ (with I) comparable, we must remove from H an old item I_o before adding I. To satisfy diversity criteria I must be on average more distant from all items in H than at least one of the items in H. We decided to choose I_o as the oldest item in H assuming that I_o is more likely to be the most distant item since its information is older and deprecated. Focusing on only one item of the history allows us to avoid the quadratic complexity and scaling up the system.

Definition 9 (Diversity of items)**.** *Assume a history of items H where $D(H)$ is the average pairwise distance between its items. An item I improves the diversity of H (I_o the oldest item of H) if and only if:*

$$D(H \cup \{I\} - \{I_o\}) > D(H) \qquad D(H) = \frac{1}{|H| * (|H| - 1)} \sum_{I \in H} \sum_{(I' \in H \wedge I' \neq I)} dist(I, I')$$

Observe that the two average distances must be comparable, so the number of items in histories must be identical. Otherwise, if we compare H to $H \cup I$, the new item I must be far more distant from items in H to make it more diverse than in our proposition. It justifies our choice to interchange I with I_o, the more likely distant item since the difference of time makes information naturally more distant.

8.4.4. *Filtering process overview*

To resume our time dependant filtering process, an incoming item I which matches a subscription must verify novelty and diversity over the subscription history H. First, the novelty of I is checked with H and if at least one similarity is below the threshold α it is discarded for H. Second, the diversity of H is compared to the diversity of $H \cup I - I_o$. If I increases the average distance, then it is notified and added to H.

A subscription is said to be satisfied by an item only if either matching and filtering processes are validated. For the matching process, it meens that all subscriptions' terms are contained into the item. According to the filtering process, the item passes through both novelty and diversity.

8.4.5. *Relevance choices*

8.4.5.1. *Novelty*

Novelty checks if information from I has already been delivered. For example, if I' contains I and appends additional information, then I is not new compared to I', but I' is new compared to I. Therefore symmetric measures like the *Jaccard* measure are unsuitable. Consequently we adopt the following measure, inspired from *Newsjunkie* [GAB 04], for the novelty of an item compared to another one:

Definition 10 (Novelty item-item)**.** *Let α be a threshold of novelty, $\alpha \in [0, 1]$, and I and I' two items. I is said to be new compared to I' if and only if:*

$$new(I, I') = \frac{\sum_{t \in (I \setminus I \cap I')} tdv(t)}{\sum_{t \in I} tdv(t)} \geq \alpha$$

This measure computes the weighted coverage of terms from I without taking into account terms present in I' w.r.t sum of weight for terms only present in I. Note that we chose the tdv value as terms weight according to our discussion above.

8.4.5.2. *Similarity in Diversity*

To measure diversity we need to compute the distance between items. Several distance measures are proposed in literature to compute diversity on a set of documents. Most frequently used are *Cosine* [ZHA 02], *Euclidean* [DRO 12a, PRI 08] and *Jaccard* [DRO 12b] but we can also quote *Pearson* derived from *Cosine*, *Dice* derived from *Jaccard* or *Levenstein*. For short items, *Euclidean* is known to produce more relevant results [BAV 10]. Thus we consider in our system a diversity function based on an *Euclidean* distance weighted by TDV values.

8.4.5.3. *Discussion for the choice of a distance function*

To measure diversity we need to compute the distance between items. Several distance measures are proposed in the literature to compute diversity on a set of documents. The most frequently used are *Cosine* [ZHA 02], *Euclidean* [DRO 12a, PRI 08] and *Jaccard* [DRO 12b] but we can also quote *Pearson* derived from *Cosine*, *Dice* derived from *Jaccard* or *Levenstein*.

The *Jaccard* measure computes the distance as the ratio between the intersection and the union of the terms of two documents (here items). However it does not take into account the importance of items terms. Conversely the *Cosine* measure allows to compute the distance between two vectors of term weights: the similarity of two documents corresponds to the correlation between the vectors (angle between the vectors). An important property of the *Cosine* similarity is its independence of document size. However since our items are short as observed in [BAV 10], Cosine and Jaccard are not appropriate. So we focus on *Euclidean* distance. Indeed for the *Euclidean* distance, items are more distant if they have important terms not in common. As for the *Cosine* metric, in order to make our score independent of the items size, we normalized the *Euclidean* distance, as shown in the following definition:

Definition 11 (Distance between 2 items)**.**

$$dist(I, I') = \sqrt{\sum_{t \in (I \cup I' \setminus I \cap I')} tdv^2(t)}$$

With the *Euclidean* distance, the contribution of a term which is shared by two items is null since the difference of their tdv values is 0. So we compute only the terms not in common with the tdv^2 value.

8.5. Real-time Integration of Novelty and Diversity

We must recall that in the Pub/Sub context, queries (subscriptions) are stored to be evaluated on-the-fly. Therefore all the optimization is done in the way to process millions of queries on incoming items. One of the main technique is to factorize treatments since there is a high probability that queries share same operations.

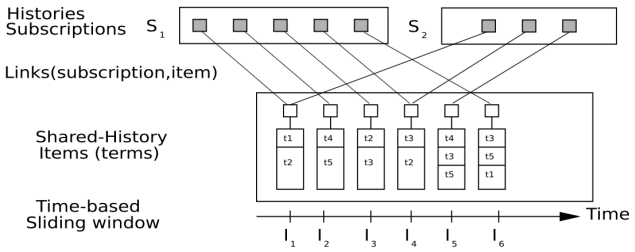

Figure 8.45: Example of a sliding window

8.5.1. *Centralized implementation*

In this section, we present our solution to quickly filter out items based on novelty and diversity criteria. It also allows us to efficiently store and to manage items histories for all subscriptions.

8.5.1.1. *Shared history*

Since an item can belong to several histories, we need to avoid keeping all items. A simple solution consists in storing the last N notified items [ZHA 02] for each subscription and in factorizing histories by storing each item only once. However the publication rate strongly differs from one source to another and this approach will impact the filtering quality. In fact, important items can be removed too quickly (active sources) or a highly filtering items could never disappear (rarely notified sources). We conclude that relevance of filtering will be impacted by those item-based histories.

To optimize memory consumption, we adopt a *shared history* which is basically a time-based sliding window $\mathcal{W}$ which contains all items notified at least once during the last period. Subscriptions histories are stored as ordered sets of pointers to related items in $\mathcal{W}$. Figure 8.45 presents an example of a sliding window $\mathcal{W}$ and two subscriptions S_1 and S_2 with their corresponding histories and pointers to the shared history.

8.5.1.2. *Shared history Filtering algorithm*

Filtering by novelty and diversity with a large number of subscriptions which share common items raises a real optimization challenge. Indeed, a naïve algorithm which checks for an incoming item first novelty, then diversity, with the histories of all the subscriptions it matches, has the following cost (Definitions 8 and 9):

$$C_{filter}(I,S) = \sum_{s \in S} \sum_{I' \in H_s} new(I,I') + \sum_{s \in \rho(S)} D(H_s \cup \{I_n\} - \{I_o\})$$

where S corresponds to the set of subscriptions matched by the incoming item I and $\rho(S)$ represents the ratio of S for which I satisfied the novelty threshold. Assume that

term weights are computed and considered as constant that the average history size is N_H items (number of computations per history) and the average item size is S_I (time for each computation is based on item size). Since the cost depends on the number of computation per history and the time for each computation is based on item size, the average complexity for this algorithm is:

$$C_{filter}(I,S) = O(|S|.N_H.S_I) + O(|\rho(S)|.(N_H.S_I)^2)$$

Experiments in Figure 8.50 show that the novelty has a filtering rate proportional to the chosen threshold α. This results in $|S| \sim |\rho(S)|$ and in a global quadratic complexity:

$$C_{filter}(I,S) = O(|S|.(N_H.S_I)^2)$$

To achieve web scaling we propose to optimize the filtering algorithm by factorizing computations for different subscriptions. As explained previously, due to the quadratic complexity of the diversity computation, the novelty must be first evaluated. However, browsing H several times for processing similarities can be costly, especially as novelty does not filter enough (see Section 8.7.3.1). In order to avoid the scanning of history twice, both filters are applied in one course. Algorithm 1 presents the processing with shared histories and optimized computations for novelty and diversity.

The algorithm processes each item I' in H. Since I must be compared to I' each time it appears in a history, we compute $new(I,I')$ and $dist(I,I')$ only once to benefit from (I,I') co-occurrences. Thus we check if this value has already been computed for another subscription. If not, we compute and register it (line 4), otherwise we just retrieve the stored value (line 6). If I is not new, the algorithm stops (line 8-9). Remember that diversity requires to compute the average pairwise distance between items of H and due to this quadratic complexity we evaluate novelty first. Then, we cumulate the distance (I,I') with others from H only if it is not the oldest item I_o (line 10-11). Secondly, as explained above, the diversity computation can be simplified to the comparison between the sum of distances from I_o and I (line 14). In that case, sums of distances are updated (line 15-16), I is added to the history and notified (line 18-19).

To resume, our algorithm integrates two main optimizations. The first one exploits the high probability of computing several times the similarity and distance for each pair of items (I,I'). Further computations of pairs are constant and not dependant of item size. These values are stored during the filtering process of I and deleted when there is no more subscription to check. The gain depends on the co-occurrence ratio $\sigma \in [0,1]$ of items in subscription histories, defined by the number of co-occurrences of items pairs checked during the filtering step on the total number of pairs required:

$$\sigma = 1 - \frac{\#cooccurrences}{\#pairs}$$

The second optimization deals with the computation of the density which changes for each notified item. To avoid the quadratic complexity of computing the sum of

Algorithm 1 Novelty and diversity filtering on a history

Require: An item I, a history H and $\alpha \in [0, 1]$ novelty threshold
1: sum$_H \leftarrow 0$; sum$_I \leftarrow 0$; I$_o \leftarrow$ H[0]; //oldest item
2: **for all** I' $\in$ H **do**
3: **if** I.getInfo(I') = null **then**
4: N $\leftarrow$ novelty(I,I'); d $\leftarrow$ dist(I,I'); I.putInfo(I',N,d);
5: **else**
6: N $\leftarrow$ I.getInfo(I').N; d $\leftarrow$ I.getInfo(I').d;
7: **if** N $< \alpha$ **then**
8: return;
9: **else if** I' != I$_o$ **then**
10: I.sum $\leftarrow$ I.sum + d;
11: **if** I.sum > I$_o$.Sum **then**
12: **for all** I' $\in$ H **do**
13: I'.sum $\leftarrow$ I'.sum+I.getInfo(I').d
14: H $\leftarrow$ H $\cup$ I;
15: Notify I;

pairwise distances in H, we propose to store computed sums of distances $I.sum$ for each item of the history H with all items received later. Then the density of H is the sum of $I.sum$. Since oldest items are removed with their stored values, no other update has to be done for remaining items. Furthermore the distance computations also benefits from the "σ" co-occurrence gain. Formally $I.sum$ is a stored value equal to:

$$I.sum = \sum_{(I' \in H \wedge I'.\tau > I.\tau)} dist(I, I')$$

Since diversity is the comparison between $D(H')$ and $D(H)$ with $H' = H \cup \{I_n\} - \{I_o\}$, it results that checking whether an incoming item increases the diversity or not may be simplified as follows:

$$\frac{2 \times \sum_{I' \in H'} I'.sum}{|H'| \times (|H'| - 1)} > \frac{2 \times \sum_{I' \in H} I'.sum}{|H| \times (|H| - 1)}$$

$$|H| = |H'| \Rightarrow \sum_{I_k \in H'} I_k.sum > \sum_{I_k \in H} I_k.sum$$

$$H' = H \cup \{I_n\} - \{I_o\} \Rightarrow \sum_{I_k \in H} I_k.sum + I.Sum - I_o.Sum > \sum_{I_k \in H} I_k.sum$$

So the diversity test consists in checking if:

$$I.sum > I_o.sum$$

To conclude, the complexity of our algorithm benefits from the co-occurrence between items for novelty and diversity, which results in the following linear complexity.

Lemma 3 (Shared-history complexity)**.** *Algorithm 1 has a linear complexity w.r.t. the number of subscriptions matched by one item, the average history and item size.*

Proof. Assume σ denotes the average co-occurrence ratio between items, $|S|$ the number of subscriptions matched by the incoming item, N_H the average history size and S_I the average item size. Then with the shared-history management, the filtering cost given by Algorithm 1 is:

$$C_{filter}(I,S) = C_{nov}(I,S) + C_{div}(I,\rho(S)) = O(|S|.\sigma.N_H.S_I) + O(\alpha|S|.\sigma.N_H.S_I)$$

$$= O(|S|.N_H.S_I)$$

■

The complexity of computing $D(H)$ and $D(H \cup I)$ is about O($|H|$), while the complexity of computing the average pairwise distance between items of H is about O($|H|^2$). This optimization in computing the density reduces processing time.

8.5.2. *Distributed filtering*

To complete our work on Pub/Sub systems, we try to compare our centralized implementation with a distributed version. To achieve this, we propose integration in the `MongoDB` NoSQL database in which we must adapt our data model and the way to distribute the computation of Novelty and Diversity. `MongoDB` seems to be more adapted to this context since we need: 1) some flexibility with a document-oriented model, 2) control where to place data in the cluster, 3) guarantee the consistency for proper filtering scores. Moreover, the implementation complexity of the similarity functions discards some other document-oriented NoSQL databases (Cassandra, DynamoDB, CouchBase).

First, we need to define a physical data model that takes into account the correlation between items and subscriptions in order to compute novelty and diversity. There are two possibilities: subscription-based (histories of items) and item-based (list of notified subscriptions).

8.5.2.1. *Subscription-based Model*

The subscription-based model nests for each subscription all the history of items in a single document, as shown in Listing 8.19. The advantage is the accessibility to all items and pre-computed sum of distances, and simplifies the design of the distributed computation with the Map/Reduce functions since it corresponds to Algorithms 1 without the shared history. For an incoming item, 1) all subscriptions are queried (*Map function*), 2) novelty and diversity are evaluated and then 3) every notification leads to an update (append and remove) on the corresponding history and sum of distances (*Reduce function*). The interesting point of this algorithm is the distribution of the computation according to the number of subscriptions.

```
{"subID":1024, "terms":[{"tID":103, "tdv":0.073}, {"tID":1710, "tdv":0.090}],
 "history": [
  {"itemID":8, "sumDist":0.11,"time":21345, "terms":[{"tID":23,"tdv":0.005},"..." ]},
  {"itemID":12,"sumDist":0,   "time":45678, "terms":[{"tID":12,"tdv":0.068},"..." ]}
]}
{"subID":1260, "terms":[{"tID":101, "tdv":0.071}, {"tID":1514, "tdv":0.086}],
 "history": [
  {"itemID":5, "sumDist":0.20,"time":12345, "terms":[{"tID":15,"tdv":0.025},"..." ]},
  {"itemID":9, "sumDist":0.12,"time":34567, "terms":[{"tID":12,"tdv":0.068},"..." ]}
  {"itemID":12,"sumDist":0,   "time":45678, "terms":[{"tID":12,"tdv":0.068},"..." ]}
]}
```

Listing 8.19: Subscription-based model

However, items are highly redundant in the repository since one item is stored in every linked subscription history. It has a strong impact on the repository size and efficiency. It is worth noticing that updating documents, have a huge impact on efficiency in MongoDB. In fact, document growth requires to reallocate blocks of data in the repository, which is a very costly process in a database (traditionally tackled with a PCTFREE). Our experiments will enlighten this point of view.

8.5.2.2. *Item-based Model*

In the item-based model, each item is stored only once in the repository with all linked subscriptions which are nested (Figure 8.20). The advantage of this method is to maximise the evaluation of novelty and the distance for diversity since it is done only once for each pair of items.

Since updating stored items is an issue in NoSQL databases, we need to store an item once, and remove it when getting out of the time window. Consequently, we need to compute the sum of pairwise distances between items in a same history. Contrary to "*I.sum*" in the centralized system (Algorithm 1), the sum of distances as designed in Algorithm 1 must be set in preceding order (to avoid updates). Thus, each distance with preceding items is stored in the current item according to the corresponding subscription. For this, "distItems" stores for each itemID (the key) the distance with the local item (value), this will help to recompute diversity of I_o vs I.

Listing 8.20 shows four items chronologically stored with nested subscriptions. We can see that subscription 1024 (resp. 1260) has notified items 8 and 12 (resp. 5, 9 and 12). Each item distance is stored with preceding stored items (*e.g.,* in item 12, subscription 1260 has two distances with items 5 and 9). This method is more complex depending on the way to distribute the notification process. Algorithms 2 and 3 illustrate the Map/Reduce implementation of the distributed process. The Map function compares an incoming item with each stored one and emits for each corresponding subscription (in the history) information useful in the Reduce function. The latter reconstructs the spread history and check the novelty.

Algorithm 2 Map Function of the item-based model

Require: A stored item I_s, an incoming item I_n
1: **if** I_s.time $<$ window_time **then** `return`;
2: N $\leftarrow$ `novelty`(I_s.terms, I_n.terms) ; D $\leftarrow$ `dist`(I_s.terms, I_n.terms)
3: **for all** nested subscriptions s **do**
4: **if** `match` (I_n.terms, s.terms) **then**
5: **if** N $< \alpha$ **then** `emit`($s.subID$, {"new": false})
6: **else** `emit`($s.subID$, {"new":true, "ID":I_s.itemID, "dist":D, "distItems":s.distItems, "time":d.time}) **end if**

Algorithm 3 Reduce Function of the item-based model

Require: List of grouped items l (for a subscription)
1: sum $\leftarrow$ 0; sum$_o$ $\leftarrow$ 0; oldest $\leftarrow$ l[0]
2: **for all** status s in l **do**
3: **if** s.new == false **then** return null **end if**
4: **if** oldest.time $>$ s.time **then** oldest $\leftarrow$ s **end if**
5: **for all** status s in $l - oldest$ **do**
6: sum $\leftarrow$ sum + s.dist ; sum$_o$ $\leftarrow$ sum$_o$ + s.distItems[s.ID]
7: **if** sum $>$ sum$_o$ **then** return l **end if**

```
{"itemID":5, "time":12345, "terms":[{"tID":12, "tdv":0.068}, "..." ],
  "subscriptions":[
    {"subID":1260, "terms":["..." ], "distItems":{}},
    {"subID":1411, "terms":["..." ], "distItems":{"3":0.310}}
]}
{"itemID":8, "time":23456, "terms":[{"tID":23, "tdv":0.005}, "..." ],
"subscriptions":[
    {"subID":1024, "terms":["..." ], "distItems":{}},
]}
{"itemID":9, "time":34567, "terms":[{"tID":15, "tdv":0.025}, "..." ],
  "subscriptions":[
    {"subID":1260, "terms":["..." ], "distItems":{"5":0.553}},
    {"subID":1411, "terms":["..." ], "distItems":{"5":0.610}}
]}
{"itemID":12, "time":45678, "terms":[{"tID":12, "tdv":0.068}, "..." ],
  "subscriptions":[
    {"subID":1024, "terms":["..." ], "distItems":{"8":0.441}},
    {"subID":1260, "terms":["..." ], "distItems":{"9":0.703, "5":0.503}},
    {"subID":1536, "terms":["..." ], "distItems":{}}
]}
```

Listing 8.20: Item-based model

The Map function (Algorithm 2) first checks if the stored item I_s remains in the time window (correct timestamp, line 1). Then it computes novelty and distance between I_s and the incoming item I_n (line 2). Then, for each nested subscription s, it checks the matching process (embedded terms, line 4). If the novelty N is lower than the threshold (line 5), it emits a *false* status to the Reduce function (the grouping key is the subscription ID) in order to avoid a notification; I_n can be new for other I_s

and given to the reduce function anyway. Finally, the distance D and sum of distances $distItems$ are emitted (line 6) to compute novelty in the Reduce function.

The Reduce function (Algorithm 3) requires to recompose subscriptions (grouping key "subID"). If a group occurs on a subscription ID, it means that at least one match occurred in the Map function step. Then we must check if at least one novelty is false (line 3) and get the oldest item (line 4). Then compute the sums of distances for I and I_o (line 7). If the sum of distance levels up the diversity (line 9), the list of distances is returned to notify the item and store the item in the repository.

8.5.2.3. *Distributing Strategies*

To make a choice between those two strategies we must take into account the distribution process. It relies on a horizontal scaling of the database where data are distributed across multiple shards (servers). The goal is to optimize distribution of computations and avoid network communications between the map and reduce functions (called *shuffle*).

The subscription-based strategy queries the subscriptions to check novelty and diversity in the *Map*. If an item is notified it must be added to histories, and all other items must be updated (sumDist). The updated version is computed during the *Reduce* but locally if we distribute documents according to the subscription ID.

For the item-based strategy, the *Map* phase is applied on items and benefits from co-occurrence of items in subscription histories. So novelty and diversity distances are computed once. The *Reduce* phase aggregates histories for each matching subscription. The result provides the new item to insert. It generates more communications between shards when aggregating in the *Shuffle* phase. However, we can enhance this by distributing documents according to the first subscription ID, which is sorted on the number of correlated items (more likely to group more items).

Naturally, the item-based strategy brings better performances especially when highly distributed. Avoiding document updates and factorizing the novelty and diversity in the *Map* phase decreases drastically the computation but the *Shuffle* phase consumes the bandwidth for large numbers of subscriptions (number of aggregates). The experiments will confirm our conclusions (see Section 8.7.5).

8.6. TDV updates

As seen in section 8.3.2, TDV computation is a very time consuming process. We need to extract similarities between all pairs of items in a collection N times: one without all the terms, N times by removing a term to compute the density of this term. Consequently this process cannot be done at a real time and must be evaluated in parallel with the filtering process. Initially, it took two days to compute 10M items. Even

if a TDV does not evolve much over time; it relies on the evolution of the presence of a term compared to all of them, on all items. The computation step needs to start from the beginning each time we need to provide new TDV values.

Our works to enhance the computation of TDV updates is twofold: i) adapt TDV computation techniques in an incremental process, ii) adapt our algorithm in a distributed context in order to scale up.

8.6.1. *TDV computation techniques*

TDV computation is a heavy process which requires to evaluate the density of a collection of items. It computes similarities between all pairs of items. Basically to provide a TDV value for a given term, densities must be computed twice, first with all the terms and second without the given term. Thus the TDV value must be computed for every term of the collection. We study in this section how to provide better solutions to compute TDV. In literature three main approaches are proposed. For conciseness purpose, we only present the main ideas and comparisons between those techniques instead of giving precise algorithms which can be found in [LAL 15].

8.6.1.1. *Naive Approach*

[WIL 85] explains the natural way to compute the TDV of a term t_i. We need to extract two densities, the first one sums the similarities of all pairs of documents in the collection, while the second one computes this density after removing t_i from all documents. The TDV value is the difference between two densities, with and without the given term. The TDV value $\Delta(\mathcal{I})$ represents the relationship between documents $\mathcal{I}$.

The complexity of the algorithm is highly dependent on the number of terms N and the number of documents M: $2M^2N^2 + 2M^2N$.

8.6.1.2. *Centroid Method*

Then, [WIL 85] proposes a simplified method by creating a centroid of the documents in the collection. This center of gravity is denoted by $\mathcal{G} = (G_1, ..., G_k, ...G_N)$ where G_k with w_{ik} (the weight of term k in the i^n document) is defined by:

$$G_k = \frac{\sum_{i=1}^{M} w_{ik}}{M}$$

For each term, this centroid provides the average occurrence. Then the TDV of a single term is computed by the similarity between each document with the centroid instead of each document of the collection. The complexity is then: $2MN^2 + 3MN$. We can notice that we save the combination between each pair of documents, but the TDV value remains dependent on similarity computations between documents and the centroid.

	Static Approach	**Incremental Approach**
`Naive method`	$2M^2N^2 + 2M^2N$	$2MN^2 + 2MN$
`Centroid method`	$2MN^2 + 3MN$	$2MN^2 + 2MN + N$
`Clustering method`	$MN^2 + 3MN$	$2MN^2 + MN + 2N$

Table 8.13: Complexities of the different methods

8.6.1.3. *Cluster Concept Covering Method (C^{3}M)*

Another approach [CAN 90] proposes to compute TDV with a clustering method. C^{3}M algorithms places two documents in the same cluster if they are likely to answer the same query. The algorithm chooses a number of clusters by creating the probabilities to select the correlation between a term and an item.

The probabilities are computed by three main components α, β and δ. Consider a matrix of term weights w_{ij} in all documents, with i is a term, and j is a document. We compute α_i as the sum of weights for each term and β_j the weights for each document. Then, we produce a matrix of documents term correlation $N \times N$. Each cell is the probability $delta_{ik}$ that gives how much the couple (i, k) are similar. This similarity is given by coupling weights and normalizing with sum of weights α and β:

$$\alpha_i = \frac{1}{\sum_{i=1}^{N} w_i} \qquad \beta_j = \frac{1}{\sum_{j=1}^{M} w_j} \qquad \delta_{ik} = \alpha_i \times \sum_{l=1}^{N} w_{il} \times w_{kl} \times \beta_l$$

An interesting property is extracted from the diagonal of this matrix. In fact, $\delta_k k$ is the coupling similarity between a document and itself, it therefore shows how much a document k is dissimilar from others.

Thus, the number of clusters $\mathcal{C}$ is the sum of the diagonal probabilities δ_{kk}. The relational is: more clusters there are, more dissimilar documents are. The TDV is then the difference between the number of clusters $\mathcal{C}$ with term t_i and the number of clusters $\mathcal{C}_i$ without term t_i. The main idea is that the number of clusters is inversely proportional to the density of a collection. So the number of clusters and TDV are given by the following formulas:

$$\mathcal{C} = \sum_{k=1}^{M} \delta_{kk} \qquad tdv(i) = \mathcal{C} - \mathcal{C}_i$$

The production of TDV values is then extremely simplified and thus the complexity is reduced to: MN^2+3MN. Even if we earn comparisons between terms of documents, we must notice that this clustering method provides approximate results. In fact, this solution focuses only on documents in a same cluster, not all the documents.

8.6.1.4. *Approaches comparison*

The different complexities are resumed in the first column of Table 8.13 where N is the number of terms and M the number of items. We can notice that every technique is characterized by the correlation between documents and similarities between them (MN^2), even if the optimized techniques try to reduce this step. Specific gains are found in the following step by combining the matrix or resume of the content of documents which lead to the reduction of the second step.

8.6.2. *Incremental Approach*

Those techniques are dedicated to compute a whole static collection of documents. But to follow its evolution in our highly dynamic context, we need to study an incremental way to compute TDV. To achieve this we extract the minimal information to compute for a new incoming item (or set of items). In order to keep a constant size for the collection we remove the oldest item for each add-on.

In the preceding techniques, each new item I_{new} leads to:

– The `naive method` requires to compute similarities of the incoming item with the whole the collection and deduce it with the old computed density:

$$\Delta_{new} = \Delta_{old} + \sum_{i=1}^{M} sim(d_i, I_{new})$$

with a complexity of $2MN^2 + 2MN$, the first scan of the collection M is saved.

– The `centroid method` must update every changing terms of I_{new} in centroid c_{new}. Then we compute the similarity of each document with c_{new}:

$$\Delta_{new} = \sum_{i=1}^{M} sim(d_i, c_{new})$$

with a complexity of $2MN^2 + 2MN + N$, a small gain is obtained since the centroid must be updated and recomputed.

– The `clustering method` requires to recompute the number of clusters on each dimension, thus it must update probabilities. Those updates require an update of each probability in the matrix, focused on the incoming item, and then recompute the number of clusters. This leads to a complexity of $2MN^2 + MN + 2N$ where updates cost more than computing the whole collection from the beginning.

8.6.2.1. *Comparison*

All the complexities from the four techniques in static and incremental approaches are resumed in Table 8.13. It synthesizes in the first column the complexity of the TDV computation for all terms of a vocabulary and secondly its computation in an

incremental way. We must notice that even if each dimension is huge, the number of items M remains the most critical one (can be more than 100 times more).

The naive method complexity is brought by the density computation which must be done N+1 times whereas the centroid one is done with a unique vector. According to the clustering method time is spent to compute the matrix once, and then three computations are done on this matrix. By taking into account the fact that the vocabulary evolves far less than the number of items, we can conclude than the Naive method is not realistic. However two others remain extremely time consuming.

8.6.3. *TDV in a distributed environment*

The last step of TDV updates enhancement is the integration of those algorithms in a NoSQL database. Since the calculation of TDVs is very computational, we need to distribute it in a distributed environment in order to scale it up. To achieve this, we need to model each needed structure in the incremental approaches. First, data structures must be defined, and then develop algorithms in Map/Reduce algorithms.

Documents in the collection are very simple JSON documents in which term identifier tID and initial weights w are given:

```
{"docID":1, "terms":[{"tID":0,"w":0.33},{"tID":5,"w":0.33},
                     {"tID":12,"w":0.33}]}
```

Adapting those algorithms requires to compute several steps of parameters, matrix, and constants. Moreover in a distributed environment, we need to take care of number and types of updates (time consuming in NoSQL), how much distributed is the computation, and how to merge results before the reduce function. Thus we will compare the three approaches in incremental modes in a distributed environment.

8.6.3.1. *Naive method*

To compute the Naive method in a distributed environment, we need to compute all the similarities between all the items. This approach is not realistic, in fact NoSQL databases do not provide "join" operations which are necessary to compute pairs, and making M queries on a collection is not a proper solution. Thus, a two-step algorithm is necessary to achieve it and illustrated in Map/Reduce language in Figure 8.46.

The first step focuses on the term dimension (N) which is lower than the items one, like in [ELS 08] with similarity computations with Map/Reduce. It executes one query which creates for each term a list of (doc,weight). The second step computes the result of the previous step by summing the term-pair similarities, for each term. The result of the previous step is the opposite of TDV, a simple query gives the TDV

First step

```
map(){
  for(t in terms)
    emit(t.tID, {"docId":docId,
      "w":t.w});
}
```

```
reduce (key, values){
  return {"tID":key, "docs":values};
}
```

Second step

```
map(){
  for(d1 in docs)
    for(d2 in docs)
      if(d1.docID < d2.docID)
        emit(tID, d1.w*d2*w);
}
```

```
reduce (key, values){
  sum = 0;
  for(v in values)
    sum += v;
  return sum;
}
```

Figure 8.46: Map/Reduce steps for the Naive Method

value of each term of the collection. The similarity is here simplified to the sum of common weighted terms between documents.

We can see that huge lists of documents and weight are produced and aggregated for each term. Once the result of this step is computed and stored on several servers, the second step is computed on it. It provides for each pair of documents the sum of term similarities, and then computes the density (with and without the term). The obtained TDV value is computed in a simpler way by computing only the sum of term pairs between documents. Therefore, TDVs will be flattened in comparison to the standard values. However, the computation of these values is far more efficient in this context.

The incremental approach adds or removes term weights to every term of the stored collection obtained in the first step. Then, it is resumed to: i) remove the first step, ii) add term weights from the new item and remove the oldest one (two update queries on all terms documents), iii) compute the second step.

8.6.3.2. *Centroid Method*

To compute the Centroid method in a distributed environment, we need to compute the centroid on all the stored items. To achieve this, a two-step Map/Reduce process is defined (Figure 8.47).

The first step computes the centroid by summing the terms weight from all the items, creating a $centroidVector$ in the reduce function where each key "tID" is the sum of weights in the collection for this term. Once $centroidVector$ is obtained, the second step computes the similarities between all documents with this centroid. Each map function produces a similarity with the centroid for a given term, after removing the term from the document (terms - t). The reduce function sums the similarities of a key "tID" to produce the density of the collection for this term. The result is the density Δ (`null` key) and every terms' (`tID`) density Δ_i. Recall that $tdv(i) = \Delta_i - \Delta$.

First step

```
map(){
  for(t in terms)
    emit(t.tID, t.w);
}
```

```
reduce (key, values){
  sum = 0;
  for(v in values) sum += v;
  return sum;
}
```

Second step

```
map(){
  centroid = [centroidVector];
  emit(null, sim(terms, centroid));
  for(t in terms)
    emit(t.tID, sim(terms-t, centroid));
}
```

```
reduce (key, values){
  sum = 0
  for(v in values) sum += v;
  return sum;
}
```

Figure 8.47: Map/Reduce steps for the Centroid Method

First step

```
map(){
  var alpha = 0;
  for(t in terms){
    alpha += t.w;
    emit({"t":"beta","v":t.tID}, t.w});
  }
  emit({"t":"alpha","v":docID}, alpha});
}
```

```
reduce (key, values){
  sum = 0;
  for(v in values)
    sum += v;
  return sum;
}
```

Second step

```
map(){
  sum = [];
  beta = [betaVector];
  for(t in terms){
    sum[0] += t.w ** 2 / beta[t.tID];
    sum[t.tID] += t.w ** 2 / beta[t.tID];
  }
  for(key => value in sum)
    if(key != 0)emit(key,sum[0]-sum[key]);
    else        emit(null,sum[0]);
}
```

```
reduce (key, values){
  sum = 0;
  alpha = [alphaVector];
  for(v in values)
    sum += v;
  return sum / alpha[key];
}
```

Figure 8.48: Map/Reduce steps for the Clustering Method

The incremental approach modifies the previous steps by: i) removing the first step, ii) adding the incoming item and removing the oldest one, iii) updating the centroid directly in main memory (no space consumption), iv) Compute the second step.

8.6.3.3. *Clustering method*

The distributed clustering method must compute the three components to define the number of clusters: α (document vector), β (term vector) and δ (dissimilarity matrix's diagonal). A two steps algorithm is then necessary as shown in Figure 8.48.

The first step focuses on α and β computation. In the map function, we must distinguish types of computations by creating a specific key: type (α, β) and value

(termID and docID). The reduce functions then aggregates the values and produce all the information as output. Both vectors are recomposed locally.

The second step is optimized to compute either δ and every δ_i (without terms) to produce the number of clusters $\mathcal{C}$ and $\mathcal{C}_i$. The map function produces an array of sums of term weights combined to β both for all the terms. The reduce function makes for each termID the sum of all δ_i, combined with α to give the final number of clusters. Final TDV are computed locally by making the difference between $\mathcal{C}$ (key 'null') and every $\mathcal{C}_i$. The drawback of this approach is to send very huge vectors (α and β) in the query to all the servers which grows up the network communication cost.

The incremental adaptation of this distributed algorithm requires to update α, β and δ. To achieve this, we: i) remove the first step, ii) update α and β locally and add the new item in the collection, iii) process the second step.

8.7. Experiments

In this section, we study the behavior of TDV computation, the filtering system in both centralized and NoSQL environments. We will also show the impact of several parameters (*i.e.*, novelty threshold, diversity and size of the sliding window) with a real dataset of items. Finally, thanks to a user validation, we study the quality of our system with different settings and a periodic filtering based on a *top-k* approach.

8.7.1. *Implementation and description of datasets*

For our experiments, we used a subset from a real dataset of items acquired over an 8-month campaign from March to October 2010 [TRA 14]. Subscriptions were generated by using the ALIAS sampling method [WAL 77]. It produced 10M of subscriptions which follow the distribution of terms occurrences on the Web, and Web query size reported in [BEI 04], based on the vocabulary of 1.5M of distinct terms extracted from items. It is characterized among others by a maximal size equal to 12 terms and on average 2.2 terms. We implemented the filtering system with the standard Java v1.6.0_20. All experiments were run on a 3.60 GHz quad-core processor with 16 GB in JVM memory.

8.7.2. *TDV updates*

In order to evaluates TDV updates with static and incremental approach in a NoSQL environment, we stored our collection of items in a MongoDB database. To simulate the incremental insert of items with measure the average time on the last 10,000 items. We made our experiments on a cluster of 16 servers and on a subset of

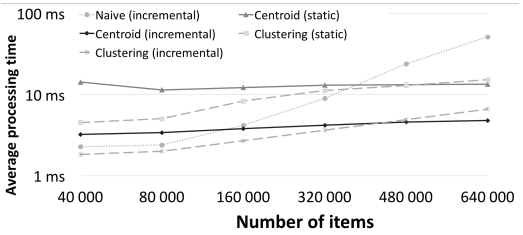

Figure 8.49: TDV computation costs in a NoSQL environment. Static vs Incremental

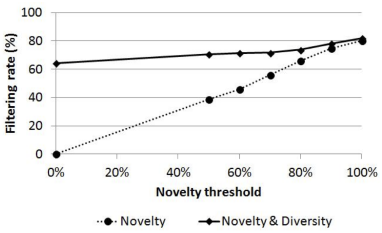

Figure 8.50: Filtering rate by varying novelty threshold

our dataset (640k items). To optimize the *reduce* steps of our treatment, we needed to choose which information to *shard* (organize data on several servers). By looking at reduced keys that are used on the second steps, it groups on the tID. However, each item has several terms and we chose the most popular term of each item (after a local sort) to be the most selective one. Thus, items are organized according to their most popular term in the collection.

Figure 8.49 plots the performances (average time to process an item in *milliseconds* in log scale) of each approach by varying the amount of treated items. We must notice that the Naive static algorithm is not shown since it took more than 800s to process each item. We can see that in a static mode, the Clustering method obtains better results than the Centroid one, but equivalent around 640k items. In fact, the Centroid method is less dependent on the number of items to process since every item only process the centroid, while the clustering method has to provide the β vector which size is the number of items. For the incremental implementations, the naive method grows up too fast while the clustering method requires to process β vectors. Finally, the incremental centroid method in a NoSQL environment remains constant for each item.

8.7.3. *Filtering rate*

We study in this section the impact of the novelty threshold and diversity on the filtering rate, as well as of the number of subscriptions and the window size. The results presented in this section correspond to the average filtering rate (number of notified items over the number of items that match the subscription) of the subscriptions satisfied at least once during the last day of the studied week.

8.7.3.1. *Impact of the novelty threshold*

Figure 8.50 shows the novelty's filtering rate when varying novelty threshold for a window size of 24 hours (dashed lines). We observe that the filtering rate linearly increases with the novelty threshold. We notice that 38% of items are filtered when the

Table 8.14: Filtering rate by varying window size

window Size	Filtering Rate
12 H	61.06%
24 H	70.71%
48 H	75.93%

Table 8.15: Number of subscription & notifications w.r.t. subscriptions size

$\|s\|$	# of subscriptions	Average # of statisfied items	Filtering rate
1	2 030 375	505.31	86.79%
2	1 804 265	21.94	57.71%
3	293 666	4.98	42.88%
>3	28 776	2.45	35.52%

novelty threshold is set to 50%, *i.e.*, when half information is not redundant. We recall that item's novelty is based on its weighted coverage (Definition 10). On average, only 20% of items that satisfy a subscription, do not contain redundant information: 80% of the items are filtered out when the novelty threshold is equal to 100%.

8.7.3.2. *Impact of diversity*

Figure 8.50 also illustrates that filtering by diversity reduces the number of items to notify (solid line). Diversity acts as a strong filter since the filtering rate when considering only diversity (*i.e.*, novelty threshold of 0%) is equal to 64.34%. Figure 8.50 proves that novelty and diversity are complementary filters. Observe that the filtering rate slightly increases with the value of the novelty threshold if diversity is also considered (64.34% for a novelty threshold of 0% up to 82% for a threshold of 100%). However the benefit for having both filters is double since filtering by novelty further allows to decrease the number of items to consider for the costly diversity computation.

For the following experiments, we set the novelty threshold to 50% (best quality from Table 8.16) and take into account the diversity for filtering process.

8.7.3.3. *Impact of window size*

Table 8.14 shows the filtering rate for different sliding window sizes. The size of the window impacts the filtering rate: with larger window size, items stay longer in histories and are used to filter new items. Although larger sliding windows have an impact on histories? length (see next section), items notified in large sliding windows stay more time in histories but information remain diverse enough to generate new notifications. For following experiments the sliding window size is set to 24 hours.

8.7.3.4. *Impact of subscriptions size*

Table 8.15 presents for each size of subscriptions its distribution which follows the one from Web queries [BEI 04]. Most of subscriptions are short (size lower than 4). We can also note that the number of notified items by subscription decreases drastically with subscription size: while short subscriptions are often matched (>500 items/day), large subscriptions are rarely notified (< 5 items/day).

According to our results, we can say that the filtering rate is highly dependent on diversity (present or not) as well as the novelty threshold. But subscriptions size also have a significant impact.

8.7.3.5. *Histories size*

We capture the variation of history size over time. We get the average size every six hours over the studied week with three different sliding window sizes. Figure 8.51 shows this variation for a novelty threshold equal to 50%, the values presented are the average size of the history of subscriptions satisfied at least once in the first six hours of the week (3.35 million subscriptions). It should be noted that the size of the history at time τ is equal to the number of items in the window of the considered size p (items published after $\tau - p$). During the initialization phase histories become larger with large sliding windows. The peak of each sliding window corresponds to the accumulation of items that ends at window-size period (12/24/48 H). The accumulation is due to the fact that empty histories do not play their filtering role. Indeed since density keeps growing during this initialization step, there is almost no filtering by diversity and most items are notified. After this initialization period, the items which were greedily added to histories at the beginning go out the sliding window which leads to a drastic decrease of the histories size during one window-size period. First items disappear which contributes to the gap between the peak and the deep exactly one period after. h The same effect occurs with less magnitude for the 12 and 48H sliding window-size, in fact a small sliding window empties quickly and must restart the density computation, while a large sliding window empties slowly and filters a lot. The 24H sliding window-size has a more stable behavior since fills up and empties with an appropriate rate. The history now allows to filter items by novelty and diversity and its size stabilizes. We also measured this variation with different novelty thresholds and confirmed these conclusions. The initialization phase corresponds to the diversification of histories.

Another conclusion from Figure 8.51 is that history size is dependant of the window size. For 12 and 24h, the number of items in the history is globally equal to the window size (10/20), while the 48H sliding window-size is half more with 70. Even if old items contribute to diversifying the information, the filtering rate (Table 8.14) growth is not proportional to the window size. So as the history size which needs a greater number of items to filter diversity.

8.7.4. *Performances evaluation in the centralized environment*

As presented in Section 8.5.1.2, we present here three different implementations of our system with a NAÏVE approach without optimization, a CO-OCCURRENCE approach with the exploitation of the co-occurrences ratio σ of items, and the DIVERSITY approach which pre-computes and stores densities in every history.

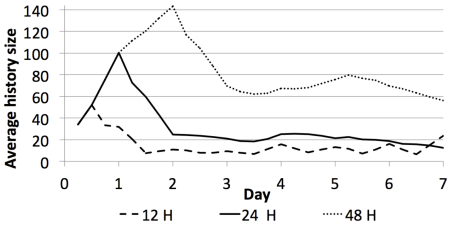

Figure 8.51: Variation of histories' size over time

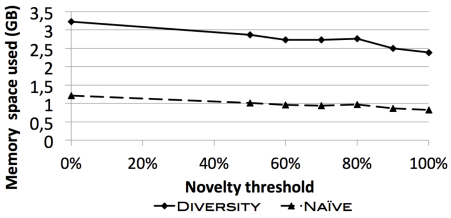

Figure 8.52: Memory space vs novelty threshold

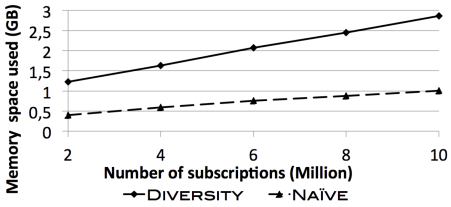

Figure 8.53: Memory space vs number of subscriptions

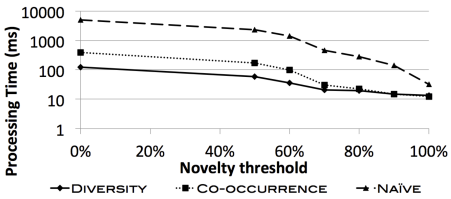

Figure 8.54: Processing time when varying novelty threshold

8.7.4.1. *Memory requirements*

Since the CO-OCCURRENCE approach stores extra-values only during the filtering process, the amount of space used by this implementation is equal to the NAÏVE implementation. Consequently we present comparison only between the NAÏVE and the DIVERSITY implementations.

Figure 8.52 shows the memory space used by the sliding window and subscription histories for various novelty thresholds. When the filtering rate is increasing, fewer items are stored in the sliding window, thus it reduces memory consumption for both optimized and normal implementations. The DIVERSITY implementation requires more memory space since sums of distance scores are pre-computed and stored for each history. Observed that it requires consequently a memory space proportional to the size of histories, so inversely proportional to the filtering rate noticed in Figure 8.50. For instance, for a rate of 50% we require 2.866MB of memory, while for a rate of 100% (+16%) we require only 2.387MB (-16.68%).

Figure 8.53 illustrates the variation of the memory consumption by varying the number of subscriptions. For this experiment, the filtering rate and the average sliding window size are fixed. We observe that the memory space increases linearly w.r.t. the number of subscriptions indexed in both implementations, since each history store

information linked to the sliding window. However since the DIVERSITY implementation requires more space to store extra-information compared to the NAÏVE version, but the ratio remains constant at 2.4. The NAÏVE implementation uses 399 MB (resp. 1009 MB) while the DIVERSITY optimization uses 1227 MB (resp. 2866 MB) for 2M (resp. 10M) of subscriptions.

8.7.4.2. *Processing time*

We now study the gain obtained by the optimizations of our system. Figure 8.54 shows that the average time (in log scale) decreases with the novelty threshold and therefore history size. The NAÏVE implementation requires much more computing time especially for low novelty thresholds. The rationale lies in its CO-OCCURRENCE optimization which reduces the number of similarities and distances computations. The NAÏVE implementation is on average 5 times more costly than the optimized ones, except for high thresholds where histories are short and few similarities/distances are computed. Moreover, the difference between CO-OCCURRENCE and DIVERSITY results decreases with the size of histories which depends on the novelty threshold: the gain is 68% for a novelty threshold of 0%, and 13% for a novelty threshold of 80%, due to the complexity of $O(1)$ (find $I_o.sum$) for the diversity computation.

Since the processing time mainly relies on history size, it is also dependent on the sliding window size. Especially for the NAÏVE and CO-OCCURRENCE implementations where the growth of computation time is more important as shown in Figure 8.55. In fact, computation of diversity is dependent on sliding window size. On the other hand, the processing time for the DIVERSITY implementation exhibits a moderate increase, except for large windows size (48h) where histories are larger, which means more distance computation and updates of sums. In fact, DIVERSITY stores sums of distances between items, while CO-OCCURRENCE implementation requires to recompute them. Since larger windows filter more (Table 8.14), the distance computations for each history, except for the DIVERSITY implementation which adds those values the first time an item is notified whatever the number of updates for histories is. On the other hand, the NAÏVE implementation computes distances each time, even if histories are updated. This requires 21 to 31 times more times than optimized solutions.

As we can see in Figure 8.56, processing time increases linearly with the number of subscriptions for both optimizations, while the NAÏVE implementation increases very fast since no co-occurrences between subscriptions are used. According to CO-OCCURRENCE and DIVERSITY implementations, it was expected to be sub-linear since similarity and distance computations between items are stored during the process to avoid its re-computation. So, with the growth of the number of subscriptions, the probability to have a same couple of items in different subscriptions grows. However, the gain for CO-OCCURRENCE is far more interesting (-93%) than DIVERSITY (-63%) since similarity and distance functions are very costly (compared to sums for diversities). Nevertheless, the DIVERSITY implementation needs 2.7 less time on average than the CO-OCCURRENCE one.

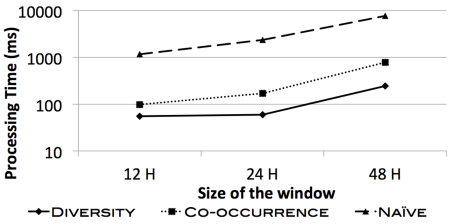

Figure 8.55: Processing time for different sizes of sliding windows

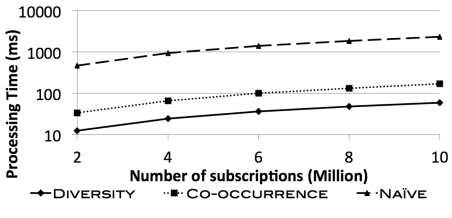

Figure 8.56: Processing time by varying the number of subscriptions

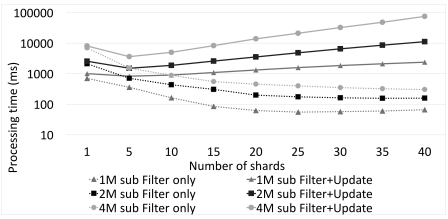

Figure 8.57: Processing time for Subscription-based model by varying the number of shards

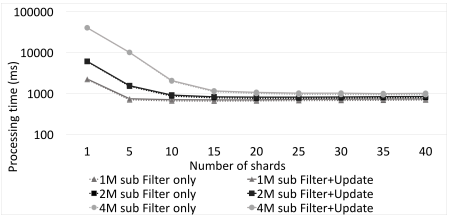

Figure 8.58: Processing time for Item-based model by varying the number of shards

8.7.5. *Performances Evaluation in a Distributed Environment*

We now study the effect of distributing our filtering process as proposed in section 8.5.2. Subscription-based and item-based models are placed on a collection with 1M, 2M and 4M of subscriptions with 10 days of history already processed. We then get the average processing time of items with 100k items. We plot the processing time by varying the distribution of the number of shards (servers).

Figure 8.57 plots processing time of the subscription-based model for the matching process (dotted lines) and the total time with subscription updates (lines). The filtering process is efficient since it takes around 300 ms per item. However, we can notice that the update time is getting more and more important with the number of shards. In fact, the number of updates is spread all over the shards, since those updates are appended in blocks of data, new reallocation of blocks must be done and distribution of data. This time grows up dramatically, which shows that this solution is not scalable.

Figure 8.58 shows the processing time of the item-based model for both matching and total times. We can see that update time is very low (between 50 and 70 ms), in fact, only one item is stored. This time grows up since the number of subscriptions to

be nested increases. According to the filtering time, it is longer than the subscription-based model (around 3 times more), but the total time stabilizes after 20 shards for 4M subscriptions. This is due to the fact that the distribution of the process is maximized in connection with the filtering step (map) and the *shuffle* step which requires network communications.

Despite good response times for the item-based model, it remains 50 times higher than the centralized version. The factorization effect and the shared history management have a huge impact on processing time which cannot be done in a distributed context since data cannot be properly interconnected. However, we can see that in Figure 8.58 the processing time reaches an asymptote (1000 ms) for any number of subscriptions. Consequently, with more subscriptions, the difference of time between the distributed and centralized versions will decrease, and the centralized system will not fit in main memory with a huge number of subscriptions (see Figure 8.53).

By extrapolating our results, the centralized version (Figure 8.56) should reach the threshold of 1000 ms of processing time with more 400M subscriptions which would require more than 40 GB of main memory. In a distributed context, it should require about 120 shards to scale up (Figure 8.58). The point of intersection of the curves can be considered as the threshold of Big Data (Volume) in this context.

8.7.6. *Quality of Filtering*

In this section we study the quality of our filtering step with users' behavior. To compute the relevance of our system, we compare chosen items by users and those obtained by our system. To validate our choice, we compare the quality of filtering when changing the weighting score, the novelty similarity and its threshold. We also compare our real time filtering with a *top-k* algorithm [DRO 09b].

To achieve this we have extracted 10 subscriptions on which we gathered matched items. Then we asked users to filter manually items according to novelty and diversity of information. Users had to read texts and to decide if an item is new or if its information is globally contained in previous items. In order to preserve our context of real time filtering, items were displayed in sequence in order to filter them in chronological order and histories were shown to users. 60 users performed 106 validations on our subscriptions. Those users come from academics and PhD students in computer science. Since filtering out by novelty is more trivial than by diversity, we kept items in the result set only if they were chosen by more than 60% of the users (75% for novelty), giving more weight to diversity.

The *top-k* algorithm [DRO 09b] determines the k most distant items from a set of items satisfying subscriptions to achieve diversity. A result set is initialized with the two most distant items among the items satisfying the subscription and extended

	Diversity only	*top-k*	**coverage 25%**	**coverage 50%**	**coverage 75%**	**TF-IDF**	**TF-IDF 50%**	**Jaccard 50%**
Precision	0.782	0.711	0.930	0.939	0.944	0.764	0.884	0.916
recall	0.698	0.634	0.652	0.652	0.610	0.618	0.545	0.652
F-Measure	0.726	0.660	0.732	0.736	0.710	0.626	0.646	0.729

Table 8.16: Filtering relevance with various techniques, thresholds and metrics

with the next most diversifying items. Each subscription has its own value k which is equal to the history size generated by our approach. Having a same size will allow us to make results sets comparable for quality measurement. Moreover, this algorithm cannot take into account the novelty since it is an asymmetric measure based on time. We must recall that our window-based approach relies on the time assumption which means that none of the notified items can be removed from the result set, while the *top-k* algorithm could put an old item in the new snapshot.

Table 8.16 shows the average precision, recall and F-Measure for all the subscriptions compared to the user result set. Different settings on our system have been made to find the most relevant measure for our filtering step with: diversity without novelty, the *top-K* approach result set, different thresholds for the weighted coverage with diversity, changing TDV by the standard TF-IDF with and without novelty, and finally novelty computed by the *Jaccard* distance. We compare especially our novelty measure by weighted coverage (Definition 10) with the standard *Jaccard* similarity for different thresholds, but also the relevance of our terms weight TDV versus TF-IDF, either in diversity or novelty. We also study the behavior of the *top-k* algorithm.

We can see that a combination of diversity and novelty produces better results than diversity alone, especially for the precision of the result. However, result set recall decreases when using the novelty which can be too selective and not diverse enough. As expected, TF-IDF weights cannot have a good impact on measures since items are short, so the TF is low and only IDF is taken into account. With a low precision (0.884) and recall (0.545) it gives the lowest F-Measure of our tests. According to novelty, the effect of the asymmetric measure and lack of weights for terms makes the *Jaccard* measure less relevant for the precision of the result set. Finally, the *top-k* technique is not as relevant as our solution since using an interchange algorithm to choose most diverse items do not rely on a real-time assumption as for user validation. The relevance of our technique with a real time filtering system, using a TDV-weighted coverage measure for novelty with a threshold of 50% gives a good accuracy.

8.8. Conclusion

In this paper, we present a Pub/Sub system, which filters by novelty and diversity on the fly. The filtering is based on items already notified to a user. We choose a

sliding window based on time to manage the subscriptions history. Our main contributions are (a) the proposition of the TDV to weight terms, combined with (b) a weighted coverage measure for novelty which is asymmetric and adapted to small items, (c) designing an optimized system which factorizes similarities and distances, and reduces diversity computation costs (d) a distributed implementation of our filtering process, (e) a distributed and incremental implementation of TDV updates computation, and (f) a quality measurement of our propositions with a user validation based on real time filtering with novelty and diversity.

From our experimental study, we show that novelty and diversity are complementary filters. Moreover we observe that the filtering rate depends on novelty threshold and on window size, and diversity has less effect for large window size. The performances of our system are also studied and we obtain an average gain of 97% in processing time with our optimization for factorizing co-occurrences and computing the density of history. The distributed implementation of the filtering process is efficient with an item-based modeling but with very high number of subscriptions (about 400M). We compare the quality of our system with different settings and a top-k and show that real-time delivery is a strong constraint which our system guarantees with a TDV-weighted coverage combined with diversity.

For further work, we aim to tune the quality of the diversity measure since cosinus and Euclidean do not focus on the same kind of filtering. Another track is to integrate correlations between users in order to integrate collaborative filtering in the way to filter items by interests thanks to other users.

8.9. Bibliography

[ABB 13] ABBAR S., AMER-YAHIA S., INDYK P., MAHABADI S., "Real-time recommendation of diverse related articles", *WWW*, p. 1–12, 2013.

[ANG 11] ANGEL A., KOUDAS N., "Efficient diversity-aware search", *SIGMOD*, ACM, p. 781–792, 2011.

[Ato 07] ATOM, "Atom: The Atom Publishing Protocol", 2007, J. Gregorio, ed. and Google and B. de Hora, ed. and NewBay Software.

[BAE 99] BAEZA-YATES R. A., RIBEIRO-NETO B. A., *Modern Information Retrieval*, ACM Press / Addison-Wesley, 1999.

[BAV 10] BAVI V., BEIRNE T., BONE N., MOHR J., NEAL B., "Comparison of Document Similarity Metrics", 2010, Computer Science Department, Western Washington University Information Retrieval.

[BEI 04] BEITZEL S. M., JENSEN E. C., CHOWDHURY A., GROSSMAN D. A., FRIEDER O., "Hourly analysis of a very large topically categorized web query log.", *SIGIR*, p. 321–328, 2004.

[BOO 74] BOOKSTEIN A., SWANSON D., "Probabilistic Models for Automatic Indexing", *Jour. American Society for Information Science*, vol. 25, num. 5, p. 312-318, 1974.

[CAN 90] CAN F., OZKARAHAN E. A., "Concepts and Effectiveness of the Cover-Coefficient-Based Clustering Methodology for Text Databases", *TDS*, vol. 15, num. 4, p. 483–517, 1990.

[CAR 01] CARZANIGA A., ROSENBLUM D. S., WOLF A. L., "Design and evaluation of a wide-area event notification service", *ACM TOCS*, vol. 19, num. 3, p. 332-383, 2001.

[CAR 10] CARMEL D., ROITMAN H., YOM-TOV E., "On the Relationship Between Novelty and Popularity of User-generated Content", *CIKM*, p. 1509-1512, 2010.

[CAR 13] CARLSON J. L., *Redis in Action*, Manning Publications Co., Greenwich, CT, USA, 2013.

[CLA 08] CLARKE C. L., KOLLA M., CORMACK G. V., VECHTOMOVA O., ASHKAN A., BÜTTCHER S., MACKINNON I., "Novelty and diversity in information retrieval evaluation", *SIGIR*, ACM, p. 659–666, 2008.

[DRO 09a] DROSOU M., PITOURA E., "Diversity over Continuous Data", *IEEE Data Eng. Bull.*, vol. 32, num. 4, p. 49-56, 2009.

[DRO 09b] DROSOU M., STEFANIDIS K., PITOURA E., "Preference-aware publish/subscribe delivery with diversity", *DEBS*, ACM, p. 1–12, 2009.

[DRO 12a] DROSOU M., PITOURA E., "DisC diversity: result diversification based on dissimilarity and coverage", *VLDB*, vol. 6, num. 1, p. 13-24, 2012.

[DRO 12b] DROSOU M., PITOURA E., "Dynamic diversification of continuous data", *EDBT*, ACM, p. 216–227, 2012.

[EIS 00] EISENHAUER G., BUSTAMANTE F. E., SCHWAN K., "Event services for high performance computing", *HPDC*, p. 113-120, 2000.

[ELS 08] ELSAYED T., LIN J., OARD D., "Pairwise Document Similarity in Large Collections with MapReduce", *ACL*, p. 265–268, 2008.

[GAB 04] GABRILOVICH E., DUMAIS S., HORVITZ E., "Newsjunkie: Providing Personalized Newsfeeds via Analysis of Information Novelty", *WWW*, p. 482-490, 2004.

[HME 11] HMEDEH Z., VOUZOUKIDOU N., TRAVERS N., CHRISTOPHIDES V., DU MOUZA C., SCHOLL M., "Characterizing Web Syndication Behavior and Content", *WISE*, p. 29–42, 2011.

[HME 12] HMEDEH Z., KOURDOUNAKIS H., CHRISTOPHIDES V., DU MOUZA C., SCHOLL M., TRAVERS N., "Subscription Indexes for Web Syndication Systems", *EDBT*, ACM, 2012.

[HME 15] HMEDEH Z., DU MOUZA C., TRAVERS N., "A Real-time Filtering by Novelty and Diversity for Publish/Subscribe Systems", *SSDBM*, p. 1-4, 2015.

[HME 16] HMEDEH Z., KOURDOUNAKIS H., CHRISTOPHIDES V., DU MOUZA C., SCHOLL M., TRAVERS N., "Content-Based Publish/Subscribe System for Web Syndication", *JCST*, vol. 31, num. 2, p. 357-378, 2016.

[KEI 12] KEIKHA M., CRESTANI F., CROFT W. B., "Diversity in Blog Feed Retrieval", *CIKM*, p. 525-534, 2012.

[LAL 15] LALAOUI O., Efficient TDV computation in no SQL environment, Master's thesis, Vertigo team - CEDRIC Laboratory, 2015.

[MIN 11] MINACK E., SIBERSKI W., NEJDL W., "Incremental diversification for very large sets: a streaming-based approach", *SIGIR*, ACM, p. 585–594, 2011.

[PAN 12] PANIGRAHI D., DAS SARMA A., AGGARWAL G., TOMKINS A., "Online selection of diverse results", *WSDM*, ACM, p. 263–272, 2012.

[PRI 08] PRIPUŽIĆ K., ŽARKO I. P., ABERER K., "Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w", *DEBS*, ACM, p. 127–138, 2008.

[ROW 01] ROWSTRON A. I. T., KERMARREC A.-M., CASTRO M., DRUSCHEL P., "SCRIBE: The Design of a Large-Scale Event Notification Infrastructure", *NGC*, p. 30-43, 2001.

[RSS 03] RSS, "RSS 2.0: Really Simple Syndication", 2003, Berkman Center for Internet and Society at Harvard Law School. http://www.rssboard.org/rss-specification.

[SAL 75] SALTON G., WONG A., YANG C. S., "A Vector Space Model for Automatic Indexing", *ACM J.*, vol. 18, num. 11, p. 613-620, 1975.

[SMY 01] SMYTH B., MCCLAVE P., "Similarity vs. Diversity", *CBR*, p. 347-361, 2001.

[TRA 14] TRAVERS N., HMEDEH Z., VOUZOUKIDOU N., DU MOUZA C., CHRISTOPHIDES V., SCHOLL M., "RSS feeds behavior analysis, structure and vocabulary", *IJWIS*, vol. 10, num. 3, p. 291–320, 2014, Top 3 of IJWIS journal papers for 2014.

[WAL 77] WALKER A., "An efficient method for generating discrete random variables with general distributions", *TOMS*, vol. 3, p. 253-256, 1977.

[WIL 85] WILLETT P., "An algorithm for the calculation of exact term discrimination values", *Inf. Process. Manage.*, vol. 21, num. 3, p. 225-232, 1985.

[YU 09] YU C., LAKSHMANAN L., AMER-YAHIA S., "It takes variety to make a world: diversification in recommender systems", *EDBT*, ACM, p. 368–378, 2009.

[ZHA 02] ZHANG Y., CALLAN J., MINKA T., "Novelty and redundancy detection in adaptive filtering", *SIGIR*, ACM, p. 81–88, 2002.