Duplicate Removal in Information Dissemination

Tak W. Yan   Hector Garcia-Molina
Department of Computer Science
Stanford University
Stanford, CA 94305
{tyan, hector}@cs.stanford.edu

Abstract

Our experience with the SIFT [YGM95] information dissemination system (in use by over 7,000 users daily) has identified an important and generic dissemination problem: duplicate information. In this paper we explain why duplicates arise, we quantify the problem, and we discuss why it impairs information dissemination. We then propose a Duplicate Removal Module (DRM) for an information dissemination system. The removal of duplicates operates on a per user, per document basis – each document read by a user generates a request, or a *duplicate restraint*. In wide-area environments, the number of restraints handled is very large. We consider the implementation of a DRM, examining alternative algorithms and data structures that may be used. We present a performance evaluation of the alternatives and answer important design questions such as: Which implementation is the best? With "best" scheme, how expensive will duplicate removal be? How much memory is required? How fast can restraints be processed?

1 Introduction

Global information systems are becoming commonplace. Conventional library systems, such as LOCIS at the Library of Congress, allow users to remotely search online catalogs for bibliographical information. Electronic bulletin boards such as USENET News (Netnews) [Cro82] are ever more popular, with millions of users and megabytes of daily traffic. Rapidly gaining momentum are some wide area information systems, such as World-Wide Web [BLCGP92], that allow users to search and browse remote file systems, document repositories, and even multimedia databases.

Some systems adopt the *information dissemination* (a.k.a. selective dissemination of information [Sal68], information filtering [LT92], alert, routing) model. A user subscribes to an information dissemination system with a *profile* that describes his interests. A profile is typically made up of a number of keyword queries. The system collects new documents from underlying sources, matches them against user profiles, and routes relevant information to users. Complementary to traditional search mechanisms, information dissemination helps users cope with information overload. At Stanford we have set up two experimental systems, SIFT-Netnews and SIFT-CSTR (SIFT [YGM95] stands for Stanford Information Filtering Tool). The former disseminates Netnews articles to over 7,000 users world-wide, and the latter delivers computer science technical report records to hundreds of researchers.[1] The source code is available for anonymous ftp (URL ftp://db.stanford.edu/pub/sift/sift.1-2.netnews.tar.Z); several SIFT servers at other sites are also operational or being ported.

In such global information systems, one of the many challenging problems is the proliferation of redundant information. Duplicate documents arise for many reasons. The foremost is that digital documents can be reproduced with extreme ease and at almost no cost. For example, in Netnews, a user may cross-post a news article in many newsgroups. He may repost it a few days later, again to multiple newsgroups. Further, there is no loss in the quality of the copies, and thus they can be replicated again. For example, a user who has made a copy of the article may repost it in yet some other newsgroups or channels (e.g., mailing lists).

By a duplicate, we mean not just an exact copy as described above, but in general a document closely similar in content to some other document and not giving users any extra information. For example, in traditional library catalog system, duplicate bibliographical records refering to the same technical reports are very common. The reasons leading to the existence of such duplicate records are mainly human input errors or inconsistencies, such as different practice in record creation, translation differences, and typographical errors. The traditional library community has recognized the problem of duplicate detection, especially in systems that provide union catalogs merging multiple bibliographical databases [HR79,WM79,Goy87,BH91,Rid92,ORO93].

In modern information systems, documents are not limited to short, structured bibliographical records, but rather full-text documents existing in different media types, with complex inter-document relationships among them. This rich content gives rise to more sources of duplication. One major source is the different media formats in which a document may exist. For instance, a technical report may be written in LaTeX, and converted into DVI and postscript. It may also be converted into plain text or HTML. The hardcopy may be scanned in as images and then converted to text via optical character recognition (OCR). All these may be made available on-line. Another source of duplication is versioning. A document may undergo a number of versions in its life-span. For example, a technical report may have a short and a full version. A user, after reading the short version, may find the full version a duplicate.

The existence of redundant information impairs the usefulness of information systems. For instance, when searching a bibliographical database, the existence of duplicate records is undesirable: users find duplicate information items intermixed in retrieval results hard to distinguish. In information dissemination, it is important that *new* information is delivered. If a document is delivered to a user, its exact/close copies will certainly reach the user over and over again. It is imperative that duplicate documents are detected and not delivered. In an experiment carried out to investi-

[1]The reader is encouraged to try out the systems. For WWW access, please connect to URL http://sift.stanford.edu. For email access, send an email message with the word "help" in the body to netnews@sift.stanford.edu or elib@sift.stanford.edu.

gate the degree of duplication in the Netnews articles sent out by SIFT (detailed in Section 2.4), we found that on average some 18% of articles received by a user overlap 80% or more in content (e.g., number of sentences) with some articles seen previously. We believe this is a major drawback diminishing the value of the system. Indeed, SIFT users have complained about this problem.[2]

To provide duplicate removal, we realize that individual users may have different requirements for duplicate elimination. A user, depending on what documents he has read previously, may consider a document a duplicate while another user does not. He may also want to specify how close a document must be to a seen one for it to be a duplicate. For example, a user may find an article not very interesting and want to remove any duplicate more than 70% similar in content. On the other hand, if a user receives an interesting document, he may want to remove only identical copies. Thus, the removal of duplicates operates on a per user, per document basis – each document read by a user generates a request for duplicate removal. We call such a request a *duplicate restraint*. The scale of global information systems makes the processing of duplicate restraints challenging. For example, in information dissemination, both the number of users and the number of incoming documents are large, so the number of restraints is very large. In SIFT, some 80,000 documents are matched against profiles of over 7,000 users every day; if we keep restraints valid for say ten days, we estimate that more than a million restraints need to be checked for every incoming document.

In this paper we study the design of a *Duplicate Removal Module (DRM)* in an information dissemination system. In Section 2, we first give a summary of related work. We then state our model of an information dissemination system and other assumptions. We present a taxonomy of duplicates in digital documents. We describe the results of a study to quantify the duplicate problem in SIFT and to illustrate the taxonomy. Partly based on the taxonomy we present the desired functionality of a DRM in Section 3. Next we consider the implementation of a DRM. There are many alternative algorithms and data structures that may be used; we describe them in in Section 4.

One critical concern with copy detection and restraint management is its cost: not only do we have to detect the duplicates with earlier documents, but we have to keep a detailed "history" of what user saw what document when. Will all this processing be too expenisve? Will the "history" take up unreasonable amounts of space, especially given the rates at which document generation and subscribers are growing? What DRM scheme is best and makes the costs manageable? To answer these questions, in Section 5 we present a performance evaluation and discuss the trends that our results yield. Finally, we conclude in Section 6.

[2]Subscribers to the DBWORLD mailing list should also be quite familiar with the duplicate problem.

2 Background

2.1 Related Work

As mentioned above, past work has been done in identifying duplicate records in bibliographical systems [HR79,WM79,Goy87,BH91,Rid92,ORO93]. However, the definition of duplicates is system-wide; as far as we know, per user, per document duplicate removal has not been studied before. Reference [BDGM95] proposes techniques to test for duplicates by comparing textual content; in this work we assume the availability of such techniques to aid duplicate removal. Previous research in information dissemination (see, e.g., [LT92]) has not addressed the duplicate information problem.

2.2 Model of an Information Dissemination System

Our model of an information dissemination system assumes a client-server architecture. A client, usually an individual user, subscribes to a server with an interest profile. The profile is typically made up of a number of keyword queries. Each profile has a unique identifier, called *profileid.* Since there is a one-to-one relationship between a user and a profile, we sometimes use the word "profile" in place of the word "user"; e.g., we say we "send a document to a profile" when we actually send the document to the user who has submitted the profile.

The server collects documents from information sources. Each document has an identifier, or *docid.* The server matches the content of each document against the profiles (i.e., the queries). Each profile has a number of associated restraints that specify what documents are considered duplicates for the profile.

2.3 Taxonomy of Duplicates

Little work has been done to classify duplicates existing in digital documents. It is important to identify what may be considered duplicates by a user and consequently what kinds of duplicates should be handled by a DRM. Thus, in this section we describe a taxonomy of duplicates. We remark that this taxonomy is not just applicable to dissemination but to search scenarios as well.

As mentioned above, digital document duplicates are not just identical copies that have the exact same byte sequence, but rather documents that the recipient thinks do not give him new information. We may classify duplicates as *intentional duplicates* or *extensional duplicates.*

2.3.1 Intentional Duplicates

Two documents are *intentional duplicates* if their creators intended them to be duplicates, even though they may differ substantially in content. Below we outline several subclasses of intentional duplicates.

- **Replication** Replication means directly replicating a document. For instance, posting the same message to multiple newsgroups is an act of replication.

- **Indirection** Indirection means that a document is actually just a reference to another document that is of interest. This is an important point: two references are duplicates if they refer to the same document, even though the references themselves differ in content a lot. For instance, bibliographic records written in totally different formats (e.g., MARC and RFC-1357 [Coh92]) are duplicates if they refer to the same document. For most users, two references to the same document should be considered duplicates, but for librarians, it may be necessary to distinguish them.

- **Versions** The same document may have different versions. For example, a technical report may be the extended abstract of another. Although the content of the short version is different from that of the full version, a user may find the short version sufficient for his information need and regards the full one a duplicate.

- **Multiple Formats** Similarly, a digital document may exist in different media types, such as plain text, HTML, LaTeX, DVI, postscript, scanned-image, OCR-ed text, or certain PC-application format. They may be classified as distinct documents by some users, and duplicates by some others.

- **Nesting** A document can be nested within another. For example, a technical report may be an anthology of a number of reports. A user, having read the anthology, is probably not interested in reading the individual reports. In Netnews, a user may include an article when composing a response to it.

Table 1 shows some sample document types and the classes of duplicates that may exist in these types. A 'yes' entry means duplicates of a class may exist in the corresponding document type. For example, technical reports (refer to the third column) may have identical replicas, versions, multiple formats, and nesting relationship. Note that, however, indirection duplicates are not possible with technical reports.

Dimension	Bibliographical Records	Technical Reports	Netnews
Replication	yes	yes	yes
Indirection	yes	no	no
Version	yes	yes	no
Multiple format	yes	yes	no
Nesting	yes	yes	yes

Table 1: Classes of Duplicates

2.3.2 Extensional Duplicates

We may also define duplication solely in terms of the textual content of the documents. Two documents are *identical duplicates* if they have the exact same words occurring in the exact same sequence. They are *partial duplicates* if portions of their texts contain the exact same words occurring in the exact same sequence. We may consider text units of different granularities (such as words or sentences) and compute the percentage of overlap of text units in a pair of documents.

We call duplicates defined by overlapping content *extensional duplicates*. Note that a pair of extensional duplicates should in all reasonable cases be be a pair of intentional duplicates, but the converse is not true (i.e., two documents with very different content may still be intentional duplicates).

2.4 Duplicates in Netnews

To quantify how serious the duplicate problem in SIFT is, we carried out the following study. We first randomly selected a sample of 60 SIFT users. We kept track of what documents they received over a period of 10 days (from Jan 10 to Jan 19, 1995). Using these data, we ran two experiments.

Experiment 1

We used the documents collected on January 19 as our basis (1,343 distinct documents were matched to the 60 profiles; a total of 1,486 matchings). We compared them against the documents received on previous days. As an example, let us look at the comparison against Jan 10 documents. Consider a matching on Jan 19, say profile P receiving document D. For each document E that matched P on Jan 10, we computed $O(E, D)$, defined as the percentage of sentences in D that are also in E. (The Netnews message header is removed before comparison.) We recorded the *daily highest overlap*:

$$M_{\text{Jan 10}}(P, D) = \max_{E \text{ matches } P \text{ on Jan 10}} O(E, D).$$

This number tells us how much of D's content was already seen by P on Jan 10. For the comparison among documents on Jan 19, we assumed the user receives documents in alphabetical order, and similarly computed the highest overlap against previous documents.

For each matching (P receiving D), we then determined its *overall highest overlap* across all ten days,

$$\max_{i=10,\ldots,19} M_{\text{Jan } i}(P, D).$$

We counted the number of matchings with overall highest overlaps in different ranges ((0, 0.1), [0.1, 0.2), etc.). The counts are shown in the last column of Table 2. For instance, the number 243 in the box (Overall, 1.0) says that there are 243 Jan 19 matchings in which the user receives a document entirely included in a document he received within the previous 10 days. This represents a fraction of $243/1,486$ or 16.4% of all matchings.

We also want to know how Netnews duplication decreases with time. We thus counted, for each day i, the number of daily highest overlaps ($M_{\text{Jan } i}(P, D)$ for all P–D matchings on Jan 19) in different ranges. The results are shown in Table 2. As an example, the number 43 in box (Jan 10, $(0, 0.1)$) says that there are 43 matchings in which the document overlaps between 0 to 10% with some document the same user received on Jan 10. Note that the numbers from Jan 10 to Jan 19 on a row need not add up to the "Overall" number.

Overlap	Jan 10	Jan 11	Jan 12	Jan 13	Jan 14	Jan 15	Jan 16	Jan 17	Jan 18	Jan 19	Overall
$(0, 0.1)$	43	53	55	85	58	36	78	89	58	48	102
$[0.1, 0.2)$	26	30	45	37	31	28	21	49	68	39	84
$[0.2, 0.3)$	15	21	25	17	24	12	20	36	47	36	83
$[0.3, 0.4)$	11	8	10	13	10	10	10	19	30	20	59
$[0.4, 0.5)$	2	5	6	8	5	4	4	7	34	8	52
$[0.5, 0.6)$	9	10	15	18	7	9	18	31	38	24	73
$[0.6, 0.7)$	2	1	1	5	2	3	2	5	19	12	31
$[0.7, 0.8)$	1	0	1	1	1	1	0	1	8	13	20
$[0.8, 0.9)$	0	0	0	0	0	2	0	0	5	6	10
$[0.9, 1.0)$	0	0	0	0	0	0	0	0	9	13	20
1.0	0	1	12	1	2	8	0	0	20	224	243

Table 2: Results from Experiment 1 of Netnews Duplication Study (Total # Matchings is 1,486)

The table gives us a lot of information about duplication in Netnews. First, the bottom row contains matchings whose documents are wholy included in some other seen document. We manually checked the sample documents and found that these are predominantly cross-postings or reposts of the same message; i.e., they are duplicates of the replication type. Secondly, there are quite many overlaps in the range from 0.3 to less than 1.0; from the Overall column, approximately 18% of all matchings fall into this range. Manual inspection indicates that these are articles in conversational threads, which include previous articles entirely or partially; i.e., they are nesting duplicates.

The rest of the overlaps comes from two sources. The first is noise from the copy detection algorithm we used. That is, sometimes the same sentence occurs in two unrelated documents by chance. This kind of overlaps is mainly found in the low overlap range (< 0.1). The other source of overlaps comes from the "signatures" in Netnews messages. Many users posting articles like to add a signature at the end of message. Thus articles written by the same user overlap at least in the signature portion.

Looking at the number of "real" duplicates (with at least 30% overlap) across the ten days, we can see duplicates become quite infrequent after about ten days.

Experiment 2

For each incoming SIFT document, we want to find out how many earlier documents overlap with it more than 10%, how many overlap more than 20% and so on. This is useful for our performance evaluation later. We thus took each document from Jan 19 in turn, and counted how many documents in the previous 10 days overlapped with it at different percentages. We then averaged the counts over all Jan 19 documents. The results are shown in Table 3. For example, for a document D from Jan 19, on average there were 0.41 documents within the previous ten days that contained 100% of D. Also note that there were on average 2.03 past documents overlapping D at least 30% ("real" duplicates).

Overlap	$= 1.0$	≥ 0.9	≥ 0.8	≥ 0.7	≥ 0.6	≥ 0.5	≥ 0.4	≥ 0.3	≥ 0.2	≥ 0.1	> 0.0
Avg. # Duplicates	0.41	0.45	0.47	0.50	0.64	1.48	1.70	2.03	3.19	4.61	10.71

Table 3: Results from Experiment 2 of Netnews Duplication Study

2.5 Copy Detection Blackbox

Given a pair of documents, we need some test to determine if they are duplicates. This test is not the focus of this paper. Previous work has been done in this area (e.g., [Rid92] for intentional duplicates, [BDGM95] for extensional duplicates). Rather, we assume the availability of such a mechanism, in the form of a *Copy Detection Blackbox (CDB)*. A CDB registers a collection of documents. Given a test document D, the blackbox returns the registered documents that it judges to be duplicates of D.

For example, given a document D, the output from an intentional CDB (Figure 1(A)) may be a set $\{\langle E, \text{version}\rangle, \langle F, \text{format}\rangle\}$, meaning that E is a version of D, and F is a duplicate of D in a different format. Similarly, for an extensional CDB (Figure 1(B)), the output may be a set $\{\langle E, 0.8\rangle, \langle F, 0.9\rangle\}$, meaning that D is 80% contained in E, and 90% in F.

3 Functionality of a Duplicate Removal Module (DRM)

As mentioned in Section 1, whenever a user receives a document, a duplicate restraint may be generated. A restraint is a tuple

$$\langle \text{profileid } P, \text{docid } E, \text{definition } L, \text{expiration } T\rangle.$$

It specifies that document E matched profile P, and because of this, future copies of E (by definition of L) should not be sent to P. After date T, the restraint is discarded. Below we elaborate on the L and T parameters.

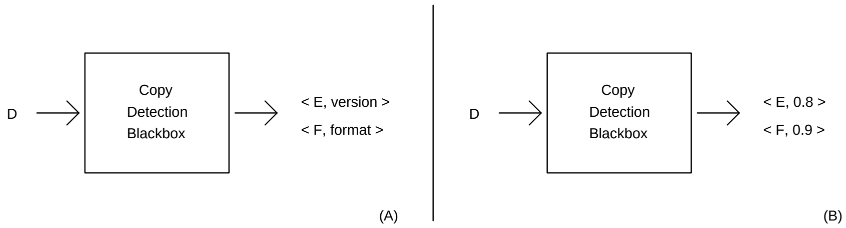

Figure 1: Copy Detection Blackbox (A) for Intentional Duplicates (B) for Extensional Duplicates

3.1 Setting User Definition of Duplicates

For a seen document, the user can define what should be considered its duplicates by specifying the definition L. Depending on what kind of CDB is available, L can be used to define either intentional or extensional duplicates. For brevity, we focus on the extensional duplicate case from this point on. We remark that the our DRM implementation schemes and evaluation results presented later on are still applicable to the intentional duplicate case.

The user should be able to say that if a document D exceeds a certain threshold of overlap against some seen document E (the degree of overlap as judged by the CDB), then it should be considered a duplicate. That is, $L(E, D) = (O(E, D) > t)$, for some $0 < t < 1$, and some overlap measure O used by the CDB. We call the threshold t the *duplication threshold*. The duplication threshold can be set differently for different documents. For example, in a Netnews dissemination system, a user may set a 100% threshold for for an important article (e.g., about a fix to a software bug), so that only duplicates that are *entirely* included in the original article are removed. On the other hand, for an article that the user does not particularly like (e.g., a recurring Call for Participation announcement for an uninteresting conference), he may set the threshold to 80%.

For good performance, the DRM may impose a system-wide *minimum duplication threshold*. As an example, in Netnews, it may not make sense to remove any document overlapping less than 30% with any seen one. So a minimum duplication threshold of 30% may be set.

3.2 Setting Time Window

A DRM cannot maintain a duplicate restraint indefinitely, otherwise the number of restraints would be unbounded. Thus there should be a time window for which a restraint is valid, and this time window should correspond to the *susceptible period* of the particular type of document.

Susceptible period is the period during which duplicates may arise and try to enter the system. For instance, for bibliographical records this may be a long period, as records may be created for different versions of document. For Netnews, this period is short: a user may post the same message a week later, but it is unlikely that he posts it a year later. (Even if he does, he probably has a reason and the document should not be considered as a duplicate.) For instance, from the data in Table 2, we may estimate the susceptible period of Netnews articles at 10 days.

To bound its work load, a DRM should thus impose a system-wide maximum time window for which restraints are valid. On the other hand, it is desirable for the user to specify their own time windows for different restraints, within the system-wide maximum time window. For documents that keep recurring, the user may specify a long period. For others, the user may specify a shorter period.

The registered body of documents in the copy detection blackbox should correspond to the time window of the valid restraints. One simple way to guarantee this is to make sure that the registered documents include all documents received by the server within the system-wide time window.

3.3 Default Restraint vs. Individual Restraint

A user can set the duplication threshold and the time window for each document received. However, it is bothersome to do this for all documents. So it is desirable to have a default duplicate restraint (each user may have his own default restraint). When a match is made, the default is automatically generated. The user may subsequently modify the restraint.

4 Implementation of a Duplication Removal Module (DRM)

In this section we discuss the design of a DRM in an information dissemination system.

4.1 Client vs. Server Processing

We first ask the question: where should duplicates be removed? One option is to perform elimination at the client end. In this case, the server is just responsible for matching incoming documents against user profiles and routing relevant documents to the users. The user restraints are maintained at the client end. As the client receives documents, it looks up what documents overlap the current one and to what extent. If it turns out the user has already received a duplicate of the current document, it is not presented to the user.

Another option is to perform duplicate elimination at the server end. The server maintains a database of duplicate restraints for all users. When a document arrives, the set of profiles that match it by content and did not receive a duplicate of the document previously are identified. The document is then delivered to the users of such profiles.

One complication may arise in the processing. There is a lapse of time between the generation of a default restraint (right after the document is sent out) and the corresponding update is received. Say a document D is sent to a user and a default restraint is generated, but he does not read D and send in his restraint update until one day later. During this period, the server receives some document E that the user would consider a duplicate of D. However, as his restraint update is not received yet (only his default restraint is in place), the server will not be able to remove E. We note that this is a problem only when the individual restraint is substantially different from the default. At the same time, the default should be "conservative" (i.e., with a high duplication threshold) so that the user will not miss documents. This pending period is also a problem in client processing. To get around the problem, the removal of duplicates needs to occur right before a document is displayed to the user, as restraints may have just been updated. This may affect the response time and be a nuisance.

Yet, the main problem with duplicate elimination at the client end is that special client software is needed. We believe the user should not need more than electronic-mail capability to utilize an information dissemination system. We attribute the success of the Stanford SIFT servers to the little effort required to use the system. No special software other than his own favorite mail reader is needed. To add duplicate detection capability at the client side, either current mail readers would have to be modified, or a new mail reader would have to be developed, distributed to the users, and accepted by them; both goals are difficult to achieve in practice.

Another benefit of server duplicate elimination is the reduction in message traffic. In environments where copies are common, this can lead to significant savings in network traffic. As observed in [YGM95], a major performance bottleneck in information dissemination is the sending of updates to the users. Even a 10% reduction in the number of messages sent out would improve performance significantly. Thus, it is advantageous to remove redundant information at the server end.

For the reasons mentioned here, in this paper we focus on duplicate elimination at the server end.

4.2 Document Flow in a Server

Suppose a document, D, arrives at a server. Conceptually, the server has to determine the set of profiles $\mathcal{P}$ that match D by content. It also has to determine the set of profiles $\mathcal{Q}$ that have seen a duplicate of D previously. Then it will send D to the users in the set $\mathcal{P} - \mathcal{Q}$.

There are of course many different ways to achieve the above conceptual description. For example, we may process one profile at a time, first determining if D is a duplicate for that profile, and if not, determining if it matches the profile by content. However, for efficiency, during content matching, we should match the document against all profiles (using an index of profiles [YGM94a, YGM94b]) at the same time, rather than one profile at a time. The one profile at a time scheme, dubbed the brute force method in [YGM94a,YGM94b], is found to perform orders of magnitude worse than when a profile index is used. Given this, only the following processing sequence (which is not far from the conceptual picture above) makes sense. (1) For an incoming document D, we first perform filtering based on its content against all profiles. This generates a set of matching profiles $\mathcal{P}$. (2) Next we identify a set of documents $\mathcal{C}$ that overlap with the current document and the percentages of overlap. (3) Restraints that have profileids and docids in the cartesian product

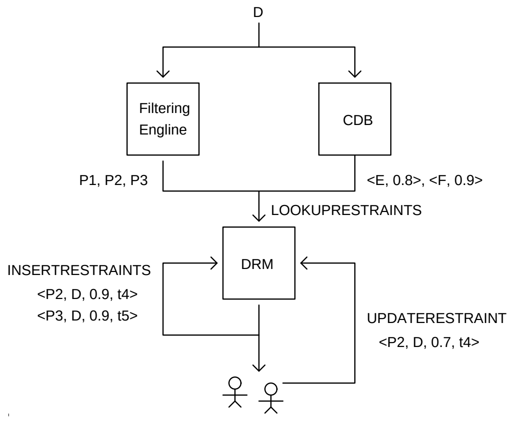

Figure 2: Document Flow in a Server

of $\mathcal{P}$ and $\mathcal{C}$ are checked to generate a set of profiles $\mathcal{Q}'$ that should not receive the document.

For example, in Figure 2, D is an incoming document. We first perform content-based filtering and identify a set of matching profiles, $\mathcal{P} = \{P1, P2, P3\}$. Next, past documents that overlap with D are identified by accessing the CDB. Suppose E is a document overlapping 80% with D, and F overlapping 90%. The output from the CDB is $\{\langle E, 0.8\rangle, \langle F, 0.9\rangle\}$ (hence $\mathcal{C} = \{E, F\}$). Next user restraints with profileids in $\mathcal{P}$ and docids in $\mathcal{C}$ are retrieved from the database of restraints. With $\{P1, P2, P3\}$ and $\{E, F\}$, suppose we retrieve the following restraints: $\langle P1, E, 0.9, t_1\rangle$, $\langle P1, F, 0.6, t_2\rangle$, $\langle P2, E, 0.9, t_3\rangle$ (assuming that times t_1, t_2, t_3 make these restraints valid). The individual duplication thresholds are then checked. Here D is a duplicate for $P1$, as it overlaps more than 60% with F. It is not a duplicate to $P2$ or $P3$. (Thus $\mathcal{Q}' = \{P1\}$.) The users for profiles $P2$ and $P3$ receive the document.

At this point, a set of default restraints are inserted into the database; say $\langle P2, D,\ 0.9,\ t_4\rangle$, $\langle P3, D,\ 0.9,\ t_5\rangle$. Later, suppose the user for $P2$ reads the document and submits a different restraint. The default restraint is then modified.

4.3 Duplicate Removal Module

The DRM maintains a database of restraints. It performs the following operations on the restraint database. Operation InsertRestraints adds a set of restraints to the database. It is used after a set of profiles are found to match a document, the default restraints are formed and are added to the database at the same time. (We remark that the set of restraints should have the same docid.) Operation UpdateRestraint is invoked to update a restraint. It is used when a user submits an actual elimination restraint and the old default restraint has to be modified. Operation LookUpRestraints is used to retrieve restraints with some specified profileids and docids. We may also need the operation Purge, which is invoked to delete all expired restraints from the database.

The restraint database is at the core of the processing of the DRM. Below we look at alternative implementations. We have to be able to efficiently identify restraints that are relevant for a new document. At the same time we have to balance the costs of insertions, updates, and purging of restraints. We look at several specialized data structures that cater to the workload characteristics of a DRM. They will be evaluated in Section 5.

4.3.1 Main Memory Buffers

We assume that the restraints are kept on disk, with indexing or hashing structures constructed to support efficient lookups. However, for performance, we assume that we keep two separate main memory buffers to batch the InsertRestraints and UpdateRestraint operations, called the I-buffer and the U-buffer respectively.

An InsertRestraints operation simply inserts the new restraints into the I-buffer. When the buffer is full, the restraints are written to disk, and the indexing or hashing structures updated. Details vary for each scheme, and will be covered in the subsections below.

Similarly, an UpdateRestraint operation does not immediately modify the restraint on disk, but just adds the updated restraint to the U-buffer. (We assume that when updating, the complete modified restraint is available; i.e., no partial updates.) In case the restraint to be modified is still in the I-buffer and not written to disk yet, UpdateRestraint modifies the main memory restraint directly.

Now a LookUpRestraint operation has to check the buffers also. It has to first look up the restraints on disk. Next the restraints in the U-buffer are checked; if any of the restraints retrieved from disk has been modified, it is replaced with the buffered version. Finally, the new restraints in the I-buffer are also looked up.

Next we look at several ways to organize the restraints on disk, using different indexing or hashing structures.

4.3.2 Indexing on Profileid

The first option is to build an index on profileids. We keep restraints with the same profileid sequentially in one list. A mapping takes a profileid to the disk location of its list of restraints. As the number of profiles is fixed and relatively small, we assume the mapping fits in main memory. Figure 3(A) illustrates this structure with an example.

These lists are constantly being updated, as new restraints are added for each profile. Thus, they should be updatable incrementally, without the need to read and then write an entire list. To efficiently support this, we leave some disk space at the end of each list. This way, new restraints can simply be added to the allocated space at the end of the list.

An InsertRestraints operation adds the restraints to the I-buffer. When the buffer is full,

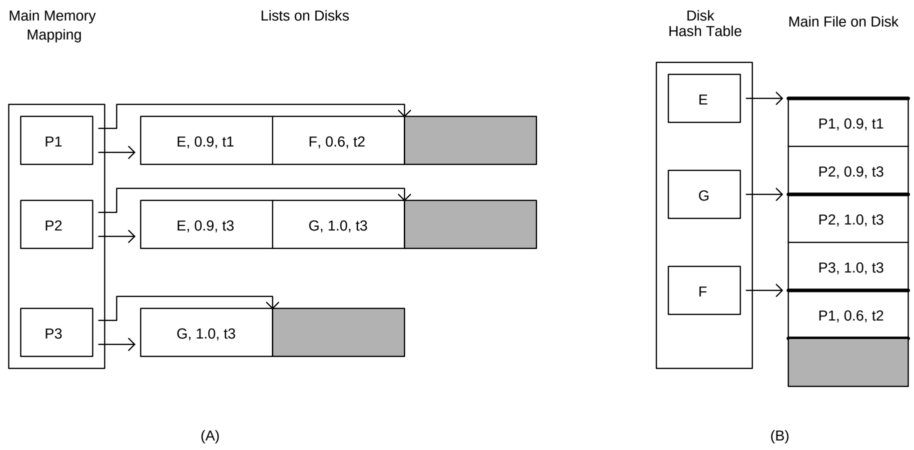

Figure 3: Indexing on (A) Profileid (B) Docid

we add all new restraints to disk. New restraints for each distinct profile are appended to the end of its list. (We may thus assume that the restraints in a list are sorted by insertion times; this is useful in later analysis.) We assume that we keep in main memory a pointer to the location of the free space for each list. If the allocated extra space is not sufficient to hold the new restraints, the entire list is retrieved. It is compacted, with expired restraints removed. New restraints are then appended at the end. If that is still not sufficient, then the list is written to a new disk location with extra disk blocks added to the end. An UpdateRestraint operation just adds the update restraint to the U-buffer. When full, we retrieve the list for each distinct profileid, scan for the docids, and modify the restraints found.

For a LookUpRestraints operation, we first look up the restraints on disk. We retrieve for each profileid its list and scan it to find the restraints with the specified docids. Next the main memory buffers are checked, as described in Section 4.3.1.

A Purge operation is a no-op in this scheme, as purging is performed during inserting.

4.3.3 Indexing on Docid

In this scheme, we try to minimize storage and keep all restraints contiguously in a file (see Figure 3(B) for an example). Restraints with the same docid are stored together. An index is built on the docid, which maps a docid to the disk location of its restraints. As the number of docids that the restraints reference is large, we assume the index resides on disk. We assume the index is implemented as a hash-file. [3]

An InsertRestraints invocation adds the restraints to the I-buffer. When the buffer is full, we append the restraints to the end of the file. Entries for the docids are inserted into the index

[3] We have also evaluated the option of using a B+tree as the index, but this option is not very attractive. The number of docids is in general very large and thus the B+tree is very large, making both storage and processing costs expensive.

file. For a UPDATERESTRAINT operation, the updates are batched. When the buffer is full, we access the index file, retrieve the restraints for each distinct docid, modify them as needed and write them back to disk.

For this index structure, it is necessary to have a purge operation to get rid of expired restraints. This PURGE operation goes through the whole file, compacts it by removing expired restraints, and writes it back out. The index is updated to reflect the changes in the locations of the restraints. We assume that this operation is performed periodically.

For a LOOKUPRESTRAINTS operation, each specified docid is looked up against the index in turn to retrieve its associated restraints, and those with the specified profiles are returned. The main memory buffers are then checked.

4.3.4 Partitioned Hashing

In partitioned hashing, a main file stores all the restraints. The file is divided into a number of buckets; a bucket is a number (w) of consecutive disk blocks. These buckets are arranged into a $g_p \times g_d$ grid. Each bucket is located by its coordinates in the grid. A hash function H_p hashes a profileid to a number x, $0 < x < g_p$, and a hash function H_d hashes a docid to a number y, $0 < y < g_d$. A restraint with profileid p and docid d is placed in the bucket $(H_p(p), H_d(d))$. We thus have a family of hashing schemes, configurable by the parameters w, g_p, and g_d. Note that for the degenerate cases ($g_p = 1$ or $g_d = 1$), we are simply considering hashing on the profileid or the docid.

An INSERTRESTRAINTS operation adds new restraints to the I-buffer. When it is full, we insert each restraint in turn. We locate the appropriate buckets and read them into main memory. We insert the restraints. If there is no empty slot in a bucket, we remove all expired restraints (if any) in it, and insert the restraints. If there is still no slot, we insert the restraints into the overflow area. Finally, when the whole batch is processed, we write the buckets back to disk.

Similarly, an UPDATERESTRAINT operation adds the updated restraint in the U-buffer. When full, we hash on the profileids and docids to locate the appropriate buckets and read them in. We scan for each restraint, and if it is found, we update the restraint. After all restraints are processed, we write the buckets back to disk.

A LOOKUPRESTRAINTS operation is processed as follows. We process each docid and profileid pair in turn. We hash on the ids to find the bucket. Within the bucket we search for the specified restraint and return it if found. The main memory buffers are then checked.

5 Performance Evaluation

In this section, we present an analytical performance evaluation of the different implementation schemes of the restraint database. We answer several important design questions, such as: Which scheme performs the best, storage-wise and running time-wise, under different scenarios? How costly is duplicate removal? How do we tune the purging period to obtain best performance? What is the impact of some important parameters, such as the incoming traffic and the average restraint period? Since our goal is to identify a good duplicate removal scheme for our operational SIFT server, in our analysis we use parameter values derived from SIFT.

5.1 Performance Model

We assume there are n_p profiles. The average number of documents received daily is n_d. Given a document D, let n_{dup} be the average number of past documents within the maximum expiration time window that overlap D above the minimum duplication threshold (i.e., the expected size of set $\mathcal{C}$ in Section 4.2). A random document has a probability ϕ of matching in content with a random profile. A random document has a probability $\kappa\phi$ of matching in content with a random profile and not being a duplicate to the user of the profile (as determined by his valid restraints).

Parameter	Base Value	Description
n_p	6,200	total # profiles
n_d	80,000	avg. # documents received daily
n_{dup}	2.03	avg. # documents above minimum duplicate threshold
ϕ	0.00036	Pr(a random document matches a random profile)
κ	0.8	fraction of matched documents not removed as duplicates
δ	0.04	fraction of restraints updated
t_{window}	10	avg. restraint period (in days)
t_{purge}	5	how often purging is done (in days)
z_i	1,000	# restraints in I-buffer
z_u	1,000	# restraints in U-buffer
u_{disk}	4	# bytes to store a pointer to a disk location
u_{main}	4	# bytes to store a (main memory) pointer
u_p	4	# bytes to store a profile identifier
u_d	32	# bytes to store a document identifier
u_t	2	# bytes to store an expiration date
u_r	1	# bytes to store a threshold
u_b	1,024	# bytes in a disk block
s	15×10^{-3}	disk seek time + latency (in sec.)
v	$u_b \times 10^{-6}$	disk tranfer rate (in sec. per block)
R	10	main memory storage cost / disk storage cost

Table 4: Model Parameters

To model the updating of restraints, we assume that a restraint is updated only once, after the user reads the corresponding document. Further, we assume that the updates for a user's restraints arrive within one day after the initial insertions, and a user's updates for one day arrive together in a batch. The scenario we are depicting is that the user reads the documents he receives in one session daily, and during the session, he sends back restraint updates. This is of course only one of many possible scenarios. We assume that the average fraction of restraints that are modified is δ.

We assume that the average restraint is valid for t_{window} days, and t_{purge} is the parameter that controls how often a purge is done to a list/bucket/file; i.e., a list/bucket/file is purged every t_{purge} days. We assume that the I-buffer holds z_i restraints and the U-buffer holds z_u restraints.

The size of a profileid is u_p bytes, that of a docid is u_d bytes, and the number of bytes to hold a duplication threshold and an expiration date are u_r and u_t respectively. A main memory pointer takes u_{main} bytes and a disk pointer take u_{disk} bytes. We assume that the disk block size is u_b bytes, the disk seek plus latency time is s sec., and the disk transfer rate is v sec. per block.

Table 4 summarizes these parameters, together with others introduced later. The base values of some parameters are derived from the SIFT data collected over the period Jan 10 – 19, 1995, during the study described in Section 2.4. The average number of profiles n_p was 6,200 and the daily average number of incoming documents n_d was 80,000. On average 177,000 matchings were made per day, so $\phi = 177,000/(6,200 \times 80,000) = 0.00036$. We assume the CDB returns duplicates with more than 30% overlap ("real" duplicates), and from Table 3 we obtain the value of 2.03 for n_{dup}. We set a t_{window} of 10 days. Finally, we assume that all 100% duplicates (0.16 of all matchings from the Overall column in Table 2), plus some partial duplicates are removed, and estimate κ at 0.8. These base values just form a starting point in our performance evaluation. We vary the parameter values over wide ranges later on.

5.2 Performance Metrics

In our performance analysis, we focus on two performance metrics. The first concerns the total storage requirements for the different schemes. In all schemes, mainly disk space is consumed, and so we use the total number of disk blocks required as the metric. However, in some schemes, substantial main memory is also required. In order to have one single storage requirement metric for easy comparisons, we introduce a scaling factor, R, which is the ratio between the costs of equal amount of main memory space and disk space. The storage requirement metric is thus the total disk storage required, plus the scaled-up main memory required. One final point is that in some schemes, the storage required varies from day to day, as the restraints are accumulated and purged. Our space metric computes the *maximum* storage required.

The second metric compares the total execution times required of the different schemes. Since we believe the processing time is dominated by disk accesses, we focus on I/O time. One intuitive metric we may use is the number of disk block accesses. However, this does not model sequential vs. random I/Os. Thus, we compare instead the expected total I/O *time* required for each scheme in one day. For the PURGE operation, which is invoked periodically by some schemes, we compute its daily contribution by dividing the total purge time by the invocation period.

5.3 Evaluation Results

We derive analytical solutions to the storage and I/O requirements of the various schemes. Due to space limitations, here we omit the details of the analysis, which can be found in the appendix. In this subsection we present the performance evaluation results.

We first show the costs of the schemes under the base case setting. We then look at the impact on performance of the scale of the information dissemination system and the extent of the duplication problem. Next we study how certain implementation parameters can be tuned for good performance. Finally we dicuss how costly is the DRM, compared against the CDB and content matching.

For the base case evaluation, we first look at the results for partitioned hashing. Recall that partitioned hashing is really a family of schemes, configurable by w, g_p, and g_d (the bucket size and the grid dimensions). The parameters are not independent; the product of the three should be greater than the number of blocks required to store all restraints (see appendix). Through the analysis and some experiments (omitted due to space limitations), we have determined that one w–g_p–g_d combination works best: $w = 1, g_p = 1$, and a large enough g_d to adequately hold the restraints. This is just hashing on docid. We will thus focus on this hashing scheme in the evaluation below. Just to understand if this conclusion changes with alternative parameter settings, we also consider a partitioned hashing scheme, with $w = 1$, $g_p = n_p$, and g_d set to a value large enough to hold all data.

Table 5 shows the evaluation results using the base case values. First of all, apparent from these results is that duplicate removal is *not* very costly; the storage of restraints takes 67 to 112 MB, and processing them takes a total of a few hours daily for a single dedicated disk. Secondly, we note that the total disk storage is the least for indexing on docid (67 MB), about 40% less than the storage required than the other schemes, while hashing on docid requires the least I/O costs, taking a total of 4,116 sec. (1.14 hours) of I/O time.

Scheme	Total Space Requirement (blocks)	Total I/O time (sec.)
Indexing on Profileid	100,411	9,709
Indexing on Docid	67,093	6,574
Partitioned Hashing	111,600	8,369
Hashing on Docid	111,600	4,116

Table 5: Base Case Performance Evaluation Results

5.3.1 Scale of the Information Dissemination System

Next we study how the scale of the information dissemination system impacts the costs of duplicate removal. We look at two parameters: n_p, the number of profiles in the system; and n_d, the number of arriving documents per day.

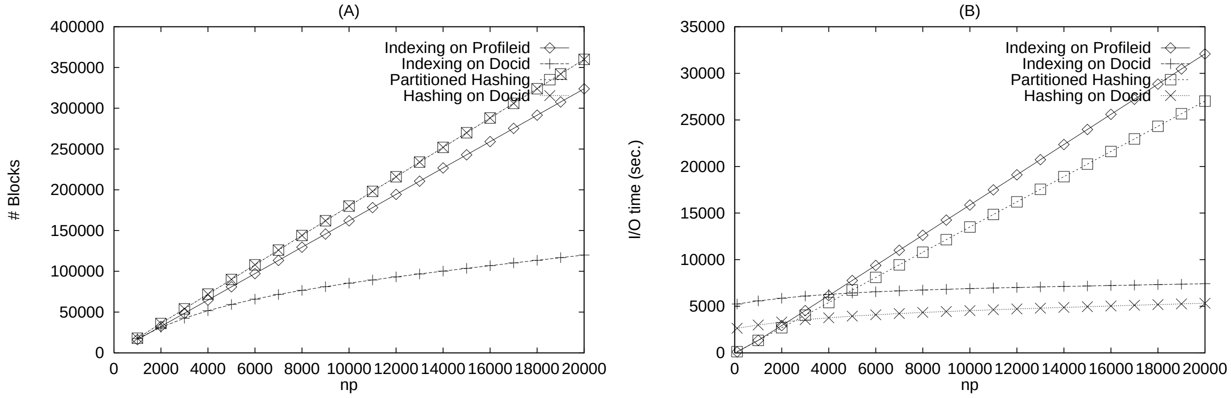

Figure 4: Storage and I/O Costs vs. n_p

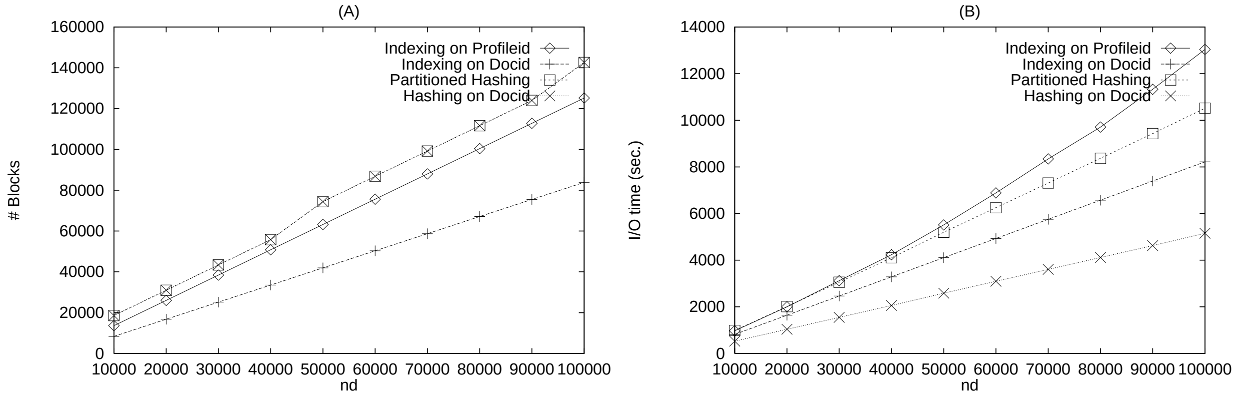

Figure 5: Storage and I/O Costs vs. n_d

As the number of profiles grows (refer to Figure 4(A)), the storage costs of all schemes expectedly increase, for most parts linearly. Indexing on docid requires the least storage space, taking about 120 MB for 20,000 users; its rate of growth is lowest, approximately 6 KB/user. The other schemes take about 350 MB for 20,000 users, and grow at a rate of 17.5 KB/user approximately. For I/O costs (Figure 4(B)), we note that at small number of users, indexing on profileid is the best scheme. This is because the number of lists accessed by each operation is small. As n_p increases, however, the I/O costs of indexing on profileid and partitioned hashing quickly increase. Except for small values of n_p, hashing on docid is the best, growing at a rate of 0.12 sec./user. Relating these back to SIFT, the user population has been growing at more than 500 a month. This translates to a storage increase of 8.75 MB and an I/O time increase of 60 sec. per month for the indexing on docid scheme.

Based on a January 1993 Netnews readership report [Rei93] and our Jan 1995 study, we estimate the number of Netnews articles per day is increasing at a rate of more than 2000 articles per month. Figure 5 shows how this increase affects performances. As expected, the disk space required

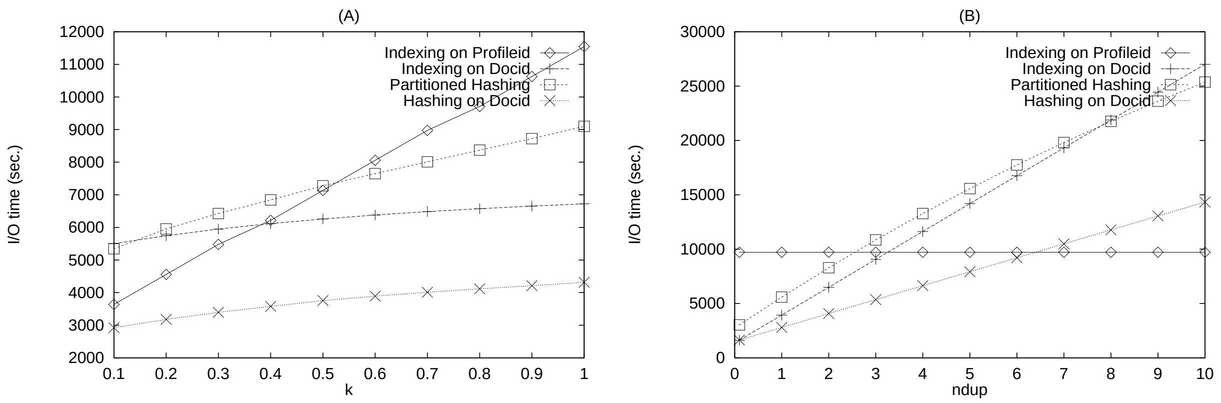

Figure 6: I/O Costs vs. κ and n_{dup}

increases with the number of documents, as the number of restraints increases. Indexing on profileid always requires the least storage, taking about 0.8 KB/document. Similarly, the I/O costs increase when volumes of information. Hashing on docid performs the best, requiring approximately 0.05 sec. of I/O time per document.

5.3.2 Extent of the Duplication Problem

Next we look at how the extent of the duplicate problem impacts the costs of duplicate removal. Relevant parameters include κ, the fraction of matchings not eliminated by duplicate removal, n_{dup}, the average number of past documents judged to be duplicates by the copy detection black box when given a document, and t_{window}, the average time window of restraints.

The value of κ may change as the occurrences of duplicates become less or more frequent, or as users vary their duplication thresholds. A κ value close to 1 means duplicates are rare, or users consider most documents non-duplicates (setting high duplication thresholds). A small value (close to 0) means duplicates are more common, or users tend to set low duplication thresholds. As κ increases, the number of documents actually sent to users increases, leading to an increase in the number of restraints. Thus, in general the storage and I/O costs increase with κ. In Figure 6(A), we show the results for the I/O cost comparison. We can see that hashing on docid is always the best. We remark that in Netnews, κ is likely to be at the high end.

The value of n_{dup} may change as the minimum duplication threshold allowed by the DRM changes. Refering back to Table 3, when only duplication thresholds greater than 30% is allowed, n_{dup} is 2.03. However, if we lower the allowed limit, say to 0%, then n_{dup} would be 10.71. Parameter n_{dup} in turn controls the work required of a LookUpRestraints operation. For small n_{dup}, hashing on docid is the best (Figure 6(B)), as we only have to access a small number of blocks. Indexing on docid is also good, for the same reason. Notice that the performance of indexing on profileid is independent of n_{dup}. Now when n_{dup} is large, the costs of the methods looking up docids

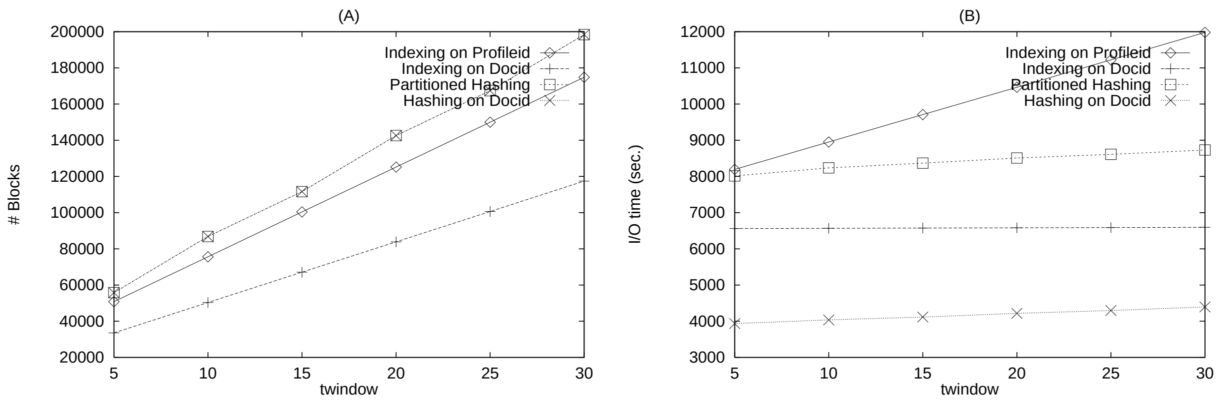

Figure 7: Storage and I/O Costs vs. t_{window}

are higher, making them less attractive than indexing on profileid.

Finally, we look at the impact of the average time window. If the window is long, more restraints are kept, and thus both storage and I/O costs increase (Figure 7). The increase in I/O costs is especially marked for indexing on profileid. In this scheme, the list associated with each list become longer, and thus retrieving a list takes longer time. On the other hand, for indexing on docid, we access restraints for a particular docid directly, and the number of blocks accessed for a docid does not increase with the time window. Under all values of t_{window} studied, hashing on docid still performs the best.

5.3.3 Implementation Parameters

In the implementation of the DRM, we can control several parameters: the purge period, and the two buffer sizes. First, our results (not shown) indicate that the storage costs of all schemes increase with longer purge periods, as expected. However, there are no substantial savings in the I/O costs, except perhaps for hashing schemes (Figure 8(A)). For indexing on profileid, the I/O costs even increase with longer purge periods. The reason is that, although the costs of purging is reduced, the costs for the other operations increase as the lists become longer. The net result is that I/O costs for indexing on profileid increase with longer purge periods. Note that, however, having a slightly longer purge period than one day is quite beneficial for hashing schemes.

Next, we look at the impact of the sizes of the main memory buffers. In Figure 8(B), we start with no buffer, and then consider an I-buffer of different sizes, up to 50,000 restraints (2 MB). We truncate the graph to show the interesting portion better. We note that the buffer only has significant effect on the indexing on profileid scheme. As we accumulate the inserted restraints, more and more restraints refer to the same profileids. Thus, we require fewer I/Os than when we have to insert restraints separately for each document sent out. For the I/O cost of indexing on docid, there is a big initial drop, but then the line stays flat. The initial drop is due to the

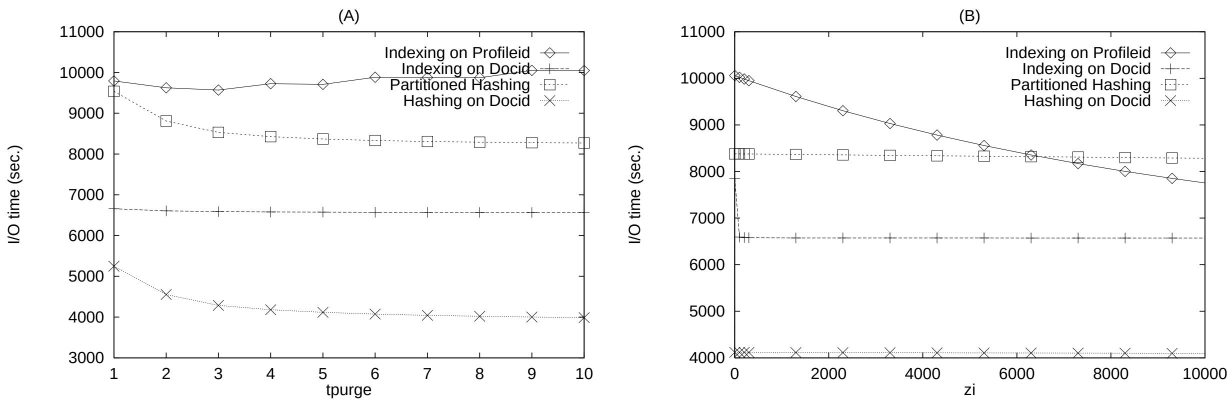

Figure 8: I/O Costs vs. t_{purge} and z_i

reduction in seeks required to append the new restraints. Yet, having a small buffer already reduces the contribution of the seek times to an insignifcant level, compared to other I/O time components, which are not affected by having the buffer. The effect on the hashing schemes is small, as the number of distinct docids is not reduced much by batching. Similar observations can be made for the U-buffer.

5.3.4 The Costs of Duplicate Removal

Although here we have not studied in detail the costs incurred by the Copy Detection Blackbox (CDB), we can provide some high-level comparison to put the DRM costs in perspective. In addition, we also compare the DRM costs with those of content matching (i.e. the main SIFT filtering function).

Let us assume that the CDB computes the percentage of overlapping sentences. We assume it uses hashing for its processing [BDGM95]. For each incoming document D, the CDB hashes the sentences to their hash values, reads the corresponding buckets, and then inserts D's docid into the buckets. Thus two I/Os are required for each sentence (assuming no bucket overflow). As [BDGM95] shows, randomization techniques can be used to substantially reduce the number of sentences hashed.

Using the documents collected on Jan 19, we estimate the number of sentences per document to be 50. With randomization, we need to hash on say one-fifth of the sentences, thus taking 20 I/Os per document. In a day there is a total of 80,000 new documents, but we only access the CDB when a document matches some profile. We may compute the number of matched documents as $n_d(1-(1-\phi)^{n_p})$ (derivation omitted), or 71,000. Thus in a day, it takes a total I/O time of $71,000 \times 20 \times (15 \times 10^{-3} + 1024 \times 10^{-6})$, or 22,800 sec. This is about 5 times more I/O effort than for restraint management (DRM).

For storage, we assume documents are registered for a period of 10 days. Each document is hashed to 10 values on average, so its docid (32 bytes) appears in 10 buckets. Thus, the number of blocks required is $\lceil 10 \times 10 \times 32 \times 71,000/1024 \rceil$, or 221,875 blocks. This is in the same order of magnitude as the storage costs of the DRM.

Finally, for content matching, from our previous work [YGM94a] we may estimate the storage cost as 0.125 block/profile, and the I/O cost as 56×10^{-6} sec./document/profile. Using the base case values, we estimate that a total of 775 blocks of disk space is required, much less than the DRM plus CDB storage cost. For the I/O time, a total of 28,000 sec. is needed, comparable to the sum of the DRM and CDB times.

6 Conclusions

We conclude that the combined DRM and CDB storage cost will be significantly higher than that of content filtering, and the duplicate removal I/O effort will roughly double the amount of work done by a dissemination server. While this is certainly non-trivial, we believe that the benefit of duplicate-free information is definitely worth the extra cost. Furthermore, considering today's hardware costs, the CDB and DRM costs are not that expensive.

Although the CDB and DRM costs are in the same order of magnitude, the CDB is inherently more expensive. We have shown that with an intelligent design, the DRM costs can be made tolerable, in spite of the large amounts of data that restraints represent.

In general indexing on the docids of restraints is always the best scheme for storage, and performs fairly well in terms of I/O costs. Hashing on docid is the best for I/O costs except when the number of users is small, or when the average number of duplicates is large. We also note that partitioned hashing do no better than simply hashing on docid.

We conclude that there is no need to have a long restraint purge period. For most schemes, a purge period of one day is sufficiently; for hashing, it may be worthwhile to have a purge period of two days to reduce the I/O costs at the costs of higher space requirement. We note that batching of restraint insertions and updates is in general not very useful. The only cases where savings are significant are: (1) batching insertions and updates for indexing on profileid; and (2) batching insertions for indexing on docid.

Finally, based on these evaluation results, we have identified the desirable setting of the DRM that we are going to implement in SIFT. We will set a default time window of 10 days and a minimum duplicate threshold of 0.3. We will use hashing on docid, with no main memory buffers and a purge period of two days. We believe that our results could similarly be of use to others implementing dissemination services.

References

[BDGM95] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proc. ACM SIGMOD Conference*, 1995.

[BH91] J. Bunge and J. Handley. Sampling to estimate the number of duplicates in a database. *Computational Statistics & Data Analysis*, 11(1):65–74, 1991.

[BLCGP92] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 1(2):52–8, 1992.

[Coh92] D. Cohen. *A Format for E-mailing Bibliographical Records (RFC-1357).* Network Information Center, SRI International, Menlo Park, California, 1992.

[Cro82] D. Crocker. *Standard for the Format of ARPA Internet Text Messages (RFC-822).* Network Information Center, SRI International, Menlo Park, California, 1982.

[Goy87] P. Goyal. Duplicate record identification in bibliographic databases. *Information Systems*, 12(3):239–42, 1987.

[HR79] T. Hickey and D. Rypka. Automatic detection of duplicate monographic records. *J. Libr. Automn*, 12(2):126–42, 1979.

[LT92] S. Loeb and D. Terry. Editors. Special Section on Information Filtering. *Communications of the ACM*, 35(12):26–81, 1992.

[ORO93] E. O'Neill, S. Rogers, and W. Oskins. Characteristics of duplicate records in OCLC's online union catalog. *Library Resources & Technical Services*, 37(1):59–71, 1993.

[Rei93] B. Reid. USENET Readership summary report for January 1993. *USENET Newsgroup news.lists*, Feb 8 1993.

[Rid92] M. Ridley. An expert system for quality control and duplicate detection in bibliographic databases. *Program*, 26(1):1–18, 1992.

[Sal68] G. Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, New York, 1968.

[WM79] M. Williams and K. MacLaury. Automatic merging of monographic databases – identification of duplicate records in multiple files: the iucs scheme. *J. Libr. Automn*, 12(2):156–68, 1979.

[YGM94a] T.W. Yan and H. Garcia-Molina. Index structures for information filtering under the vector space model. In *Proc. International Conference on Data Engineering*, pages 337–47, 1994.

[YGM94b] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Transactions on Database Systems*, 19(2):332–64, 1994.

[YGM95] T. Yan and H. Garcia-Molina. Sift – a tool for wide-area information dissemination. In *Proc. 1995 USENIX Technical Conference*, pages 177–86, 1995.

Appendix A: Performance Analysis (for Refereeing)

In the appendix we present the performance analysis of the different DRM implementation schemes in detail. Some expressions derived from the model parameters and used in the following analysis are summarized in Table 6 for easy reference.

In computing the storage costs, we ignore the I- and U-buffer costs, as they are the same for all schemes, and are insignificant compared to the storage costs of other components.

Symbol	Description
n_{dist}	# distinct documents sent out to users
θ	Pr(a random document is sent to a random profile)
$z_{i,p}$	# distinct profileids in a full I-buffer
$z_{i,d}$	# distinct docids in a full I-buffer
$z_{u,p}$	# distinct profileids in a full U-buffer
$z_{u,d}$	# distinct docids in a full U-buffer
g_p	a profileid hashes to a number between 0 and g_p in partitioned hashing
g_d	a docid hashes to a number between 0 and g_d in partitioned hashing
w	# blocks in a hash bucket

Table 6: Derived Expressions Used in Analysis

Buffering

Before analyzing each indexing/hashing scheme, let us study the behavior of the I- and U-buffers, common to all schemes.

In a full I-buffer, there are z_i restraints. Each restraint corresponds to the delivery of a document to a profile. We first answer the question: given a document D whose docid is in the buffer, what is the probability that D is sent to a profile P? Note that this is not simply $\theta = \kappa\phi$, the probability that a random document is sent to a random profile, since we know that D has been delivered to some profile (as it is in the buffer). Let us denote the desired probability by α. We can derive α as

$$\begin{aligned}
\alpha &= \Pr(\text{a document } D\text{, whose docid is in the I-buffer, is sent to a profile } P) \\
&= \Pr(\text{a document } D \text{ is sent to a profile } P \mid D\text{'s docid is in the I-buffer}) \\
&= \frac{\Pr(\text{a document } D \text{ is sent to a profile } P \text{ and } D\text{'s docid is in the I-buffer})}{\Pr(D\text{'s docid is in the I-buffer})} \\
&= \Pr(\text{a document } D \text{ is sent to a profile } P) / \Pr(D \text{ is sent to some profile}) \\
&= \theta/(1-(1-\theta)^{n_p})
\end{aligned}$$

With this, we derive that the expected number of restraints per docid in the I-buffer is $n_p\alpha$, and consequently, the number of distinct docids is $z_{i,d} = z_i/(n_p\alpha)$.

For the number of distinct profiles in the buffer, we note that $1-\alpha$ is the probability that a profile does not receive a document whose docid is in the I-buffer, $(1-\alpha)^{z_{i,d}}$ is the probability that it does not receive any document whose docid is in the I-buffer, and $1-(1-\alpha)^{z_{i,d}}$ is the probability that it receives some document whose docid is in the I-buffer. Finally, $n_p(1-(1-\alpha)^{z_{i,d}})$ is the number of distinct profileids in the batch. Each of these profileids corresponds to $z_i/z_{i,p}$ restraints.

We now consider the U-buffer. Recall that restraints are only updated within one day after insertions, and updates from a profile appear in the same batch. We define β as the probability that a docid D is referenced by a U-buffer restraint referencing profileid P. Similarly as above, we derive β as $\theta\delta/(1-(1-\theta\delta)^{n_d})$. Thus, the expected number of restraints referencing a certain profileid P is $n_d\beta$. The number of distinct profileids in the U-buffer is $z_{u,p} = z_u/(n_d\beta)$ and the number of distinct docids is $z_{u,d} = n_d(1-(1-\beta)^{z_{u,p}})$.

Indexing on Profileid

For this index structure, we need substantial main memory to store the mapping. The mapping is implemented as a open-chaining hash table. Each profileid has one entry in the hash table. We assume we keep an array equal in size to the number of profiles, i.e., n_p slots. Each slot has a main memory pointer to a chain of entries with profileids that hash to that slot. Each entry on a chain is made up of a profileid, a disk location of the list of the profile, a disk location to the end of the list, and also a main memory pointer to the next entry on the chain. The amount of main memory required is thus $n_p(2u_{main} + 2u_{disk} + u_p)$ bytes.

Next we look at the disk storage. For each profile we store a list of its restraints in contiguous blocks. The size of a restraint record in each list is equal to the sum of the sizes of a docid, a duplication threshold, and an expiration date, $u_d + u_r + u_t$. A profile receives $n_d\theta$ documents a day. As we assume that we keep restraints for $t_{window} + t_{purge}$ days (so that we only need to purge a list every t_{purge} days), the number of blocks required for each list is $\lceil \frac{(t_{window}+t_{purge})n_d\theta(u_d+u_r+u_t)}{u_b} \rceil$. The total disk space required is the product of the number of profiles (n_p) and the last expression. We add to this the main memory required, multiplied by R, which is the scaling factor for comparing disk space cost and main memory cost (i.e., we assume that main memory costs R times as much as disk storage).

$$M_{\mathrm{IP}} = n_p \lceil \frac{(t_{window} + t_{purge})n_d\theta(u_d + u_r + u_t)}{b} \rceil + R \times \frac{n_p(2u_{main} + 2u_{disk} + u_p)}{u_b}$$

We now proceed to the time requirement of this index structure. First, when the I-buffer is full, we append the restraints to the appropriate lists. Recall that the number of distinct profiles in this batch is $z_{i,p}$ and each such profile has $z_i/z_{i,p}$ restraints. We just need to read the last block of each list, add the new restraints, and write them out. For each list, this takes time $s + v + s + v\lceil \frac{z_i/z_{i,p}(u_d+u_r+u_t)}{b} \rceil$. For one day there are $n_p n_d\theta/z_i$ batches. Thus the total time required is

$$T_{\text{IP}}^{\text{I}} = n_p n_d \theta \frac{z_{i,p}}{z_i}(2s + v(1 + \lceil \frac{z_i/z_{i,p}(u_d + u_r + u_t)}{u_b} \rceil).$$

Next, when the U-buffer is full, we process the batch by updating the affected restraints. Recall that the number of distinct profiles in the buffer is $z_{u,p}$ and each such profile has $z/z_{u,p}$ updated restraints in the batch. We first retrieve the list for each distinct profileid. On average there are $(t_{window} + t_{purge}/2)n_d\theta$ restraints in a list, taking time $s + v\lceil \frac{(t_{window}+t_{purge}/2)n_d\theta(u_d+u_r+u_t)}{u_b} \rceil$ to retrieve. We then write back the modified blocks. Recall that we assume that the restraints in the list are sorted by insertion times, and that the updated restraints in a batch should be chronologically adjacent to one another. The number of affected blocks is thus $\lceil \frac{z_u/z_{u,p}(u_d+u_r+u_t)}{u_b} \rceil$. For one day there are $n_p n_d \theta\delta/z_u$ batches, and thus the total time required is

$$T_{\text{IP}}^{\text{U}} = n_p n_d \theta\delta \frac{z_{u,p}}{z_u}(s + v\lceil \frac{(t_{window} + t_{purge}/2)n_d\theta(u_d + u_r + u_t)}{u_b} \rceil + s + v\lceil \frac{z_u/z_{u,p}(u_d + u_r + u_t)}{u_b} \rceil).$$

Next, for an implicit purge of a list, we need to retrieve it ($\lceil \frac{(t_{window}+t_{purge})n_d\theta(u_d+u_r+u_t)}{u_b} \rceil$ blocks), remove expired restraints, and write a new one out ($\lceil \frac{t_{window}n_d\theta(u_d+u_r+u_t)}{u_b} \rceil$ blocks). This takes time $s + v\lceil \frac{(t_{window}+t_{purge})n_d\theta(u_d+u_r+u_t)}{u_b} \rceil + s + v\lceil \frac{t_{window}n_d\theta(u_d+u_r+u_t)}{u_b} \rceil$ sec. On average, in one day there are n_p/t_{purge} lists purged, taking time (in sec.)

$$T_{\text{IP}}^{\text{P}} = \frac{n_p}{t_{purge}}(s + v\lceil \frac{(t_{window} + t_{purge})n_d\theta(u_d + u_r + u_t)}{b} \rceil + s + v\lceil \frac{t_{window}n_d\theta(u_d + u_r + u_t)}{u_b} \rceil).$$

Finally, for a LookUpRestraints operation, for each profile in the matching set ($n_p\phi$ in all), we retrieve its restraints. This takes time $n_p\phi(s + v\lceil \frac{(t_{window}+t_{purge}/2)n_d\theta(u_d+u_r+u_t)}{u_b} \rceil)$ sec. In one day there are n_d invocations, and so the total I/O time is

$$T_{\text{IP}}^{\text{L}} = n_d n_p \phi(s + v\lceil \frac{(t_{window} + t_{purge}/2)n_d\theta(u_d + u_r + u_t)}{u_b} \rceil).$$

Summing these up, the total I/O time required for one day is

$$T_{\text{IP}} = T_{\text{IP}}^{\text{I}} + T_{\text{IP}}^{\text{U}} + T_{\text{IP}}^{\text{P}} + T_{\text{IP}}^{\text{L}}.$$

Indexing on Docid

In this scheme, disk storage is required for the main file of restraints as well as the index file. To derive an expression for the storage cost, an important number is the average number of distinct documents sent out in a day, denoted by n_{dist}. We may derive n_{dist} as follows. The probability that a document is not sent to a profile is $1 - \theta$. The probability that it is not sent to any of the n_p profiles is $(1-\theta)^{n_p}$. Thus the probability that it is sent to any profile is $1-(1-\theta)^{n_p}$. Consequently, $n_{dist} = n_d(1 - (1 - \theta)^{n_p})$.

Now in the index file, the number of entries is equal to the number of distinct documents sent out in $t_{window} + t_{purge}$ days, or $(t_{window} + t_{purge})n_{dist}$. Each entry has size $u_d + u_{disk}$. Since a disk block can hold $\lfloor u_b/(u_d + u_{disk})\rfloor$ entries, we need $\lceil (t_{window} + t_{purge})n_{dist}/\lfloor u_b/(u_d + u_{disk})\rfloor\rceil$ blocks.

We next look at the size of the main file that stores the restraints. The restraints are stored contiguously in the file. The size of a restraint record is equal to the sum of the sizes of a profileid, a duplication threshold, and an expiration date, $u_p+u_r+u_t$. In one day, the total number of restraints is $n_p n_d \theta$. The disk space required for one day is $n_p n_d \theta(u_p + u_r + u_t)$ bytes; for $t_{window} + t_{purge}$ days this is $\lceil (t_{window} + t_{purge})n_p n_d \theta(u_p + u_r + u_t)/u_b\rceil$ blocks. Summing up the above expressions, the total disk space requirement is then (in blocks):

$$M_{\mathrm{ID}} = \lceil \frac{(t_{window} + t_{purge})n_p n_d \theta(u_p + u_r + u_t)}{u_b} \rceil + \lceil \frac{(t_{window} + t_{purge})u_{dist}}{\lfloor u_b/(u_d + u_{disk})\rfloor} \rceil.$$

We now determine the average total I/O time required daily for operations on the file. First, for inserting restraints, when the I-buffer is full, we simply write out the new restraints at the end of the main file and then update the index. The appending of the z_i new restraints takes time (in sec.) $s + v\lceil z_i(u_p + u_r + u_t)/u_b\rceil$. There are $z_{i,d}$ distinct documents in the batch. To add the docid entries to the index takes time $z_{i,d}(s + v)$. There are $n_p n_d \theta/z_i$ batches daily, so the total I/O time required is (in sec.):

$$T^{\mathrm{I}}_{\mathrm{ID}} = \frac{n_p n_d \theta}{z_i}(z_{i,d}(s + v) + s + v\lceil \frac{z_i(u_p + u_r + u_t)}{u_b} \rceil).$$

Next we consider the UPDATERESTRAINT batches. Recall that the number of distinct documents in the batch is $z_{u,d}$. For each of these, we access the index (1 block), scans the restraints for the docid ($\lceil \frac{n_p\theta(u_p+u_r+u_t)}{u_b}\rceil$ blocks), modify the specified restraints, and write them back out. There are $n_p n_d \theta\delta/z_u$ batches per day, and so the total I/O time required is:

$$T^{\mathrm{U}}_{\mathrm{ID}} = n_p n_d \theta\delta \frac{z_{u,d}}{z_u}((s + v) + 2s + 2v\lceil \frac{n_p\theta(u_p + u_r + u_t)}{u_b} \rceil).$$

For a PURGE operation, we read in all restraints, remove expired ones, and write out a new main file. The old main file has $\lceil (t_{window} + t_{purge})n_p n_d \theta(u_p + u_r + u_t)/u_b\rceil$ blocks, and the new one has $\lceil t_{window} n_p n_d \theta(u_p + u_r + u_t)/u_b\rceil$ blocks, so purging the file takes $2s + v(\lceil (t_{window} + t_{purge})n_p n_d \theta$ $(u_p + u_r + u_t)/u_b\rceil + \lceil t_{window} n_p n_d \theta(u_p + u_r + u_t)/u_b\rceil)$ sec. To update the index, for almost every docid in the old file, we need to either delete or update an entry. We thus simply read in the entire hash file once, and write it out with the locations updated. The I/O time is then $2s+2v\lceil (t_{window} + t_{purge})n_{dist}/\lfloor (u_b/u_d + u_{disk})\rfloor\rceil$. The total I/O time, averaged over t_{purge} days, is then

$$T^{\mathrm{P}}_{\mathrm{ID}} = \frac{1}{t_{purge}}(4s + 2v\lceil \frac{(t_{window} + t_{purge})n_{dist}}{\lfloor u_b/(u_d + u_{disk})\rfloor} \rceil +$$

$$v(\lceil\frac{(t_{window}+t_{purge})n_p n_d\theta(u_p+u_r+u_t)}{u_b}\rceil+\lceil\frac{t_{window}n_p n_d\theta(u_p+u_r+u_t)}{u_b}\rceil)).$$

Finally, we look at the time requirement of a LookUpRestraints operation. The set of duplicates for an incoming document has average size n_{dup}. For each of them, we access the index and then read the restraints for that document. Thus for n_{dup} documents, we spend a total I/O time (in sec.) of $n_{dup}(s+v+s+v\lceil\frac{n_p\theta(u_p+u_r+u_t)}{u_b}\rceil)$. We multiply the last expression by the average daily number of incoming documents, n_d, to obtain the total I/O time required for the LookUpRestraints operations per day.

$$T^{\mathrm{L}}_{\mathrm{ID}}=n_d n_{dup}((s+v)+s+v\lceil\frac{n_p\theta(u_p+u_r+u_t)}{u_b}\rceil).$$

Summing up the total I/O time required in one day, we obtain

$$T_{\mathrm{ID}}=T^{\mathrm{I}}_{\mathrm{ID}}+T^{\mathrm{U}}_{\mathrm{ID}}+T^{\mathrm{P}}_{\mathrm{ID}}+T^{\mathrm{L}}_{\mathrm{ID}}.$$

Partitioned Hashing

In partitioned hashing schemes, the size of a restraint record is $u_p+u_d+u_r+u_t$ bytes. One block can fit $\lfloor u_b/(u_p+u_d+u_r+u_t)\rfloor$ restraints. The total number of restraints kept in the file is $(t_{window}+t_{purge})n_p n_d\theta$. We thus need at least $(t_{window}+t_{purge})n_p n_d\theta/\lfloor u_b/(u_p+u_d+u_r+u_t)\rfloor$ blocks. Suppose one bucket has w blocks and we arrange the buckets into a $g_p\times g_d$ grid. To ensure enough space is available to hold all restraints, we must satisfy the constraint $g_p g_d w>(t_{window}+t_{purge})n_p n_d\theta/\lfloor u_b/(u_p+u_d+u_r+u_t)\rfloor$. For ease of computation, we enforce $g_p g_d w\geq N$, N being the least multiple of n_p greater than the right-hand-side in the first inequality.

We may thus consider a family of hashing schemes by varying g_p, g_d, and w. In our evaluation, we vary w from 1 on, g_p from 1 to n_p, and set g_d to be $\lceil\frac{N}{g_p w}\rceil$. We stop when $\frac{N}{g_p w}$ is less than 1. However, the cost formulas for the different configurations are the same, as derived below.

First of all, the space required of all hashing schemes is simply (in blocks)

$$M_{\mathrm{H}}=g_p g_d w.$$

Next, we consider the I/O costs. We need to determine the number of I/O accesses per I-buffer or U-buffer batch. Let us first derive an expression useful for this purpose. Suppose we have some elements arranged into a grid of r rows and c columns. Consider the following experiment X: we randomly pick a column, and randomly pick j elements within this column (repetitions allowed). We answer the question: how many distinct elements are picked after i iterations of X?

For an element, the probability that its column is not picked in an experiment X is $1-1/c$. The probability that its column is picked, but it is not picked is $1/c(1-1/r)^j$. So in one experiment, the probability that it is not picked is $\gamma = 1 - 1/c + 1/c(1 - 1/r)^j$. As X is repeated i times, the probability that it is never picked is γ^i. Finally, the expected number of elements picked is $G(c, r, i, j) = cr(1 - \gamma^i)$.

Now we consider processing the restraints in the I-buffer. The restraints are grouped by docids; the number of docids is $z_{i,d}$. The number of restraints per docid is $z_i/z_{i,d}$. When accessing the buckets, for each distinct docid, we may think of ourselves as picking a random column, and then picking $z_i/z_{i,d}$ random elements from that column. Consequently, the number of bucket accessed is $G(g_d, g_p, z_{i,d}, z_i/z_{i,d})$.

As the number of batches is $n_p n_d \theta / z_i$, the total time required is

$$T_{\mathrm{H}}^{\mathrm{I}} = \frac{n_p n_d \theta}{z_i} G(g_d, g_p, z_{i,d}, z_i/z_{i,d})(s + vw).$$

For the U-buffer, a batch has $z_{u,p}$ distinct profileids, each having $z_u/z_{u,p}$ associated restraints. We may similarly derive the number of bucket accesses per batch as $G(g_p, g_d, z_{u,p}, z_u/z_{u,p})$. There are $n_p n_d \theta \delta / z_u$ batches a day, and so the total I/O time required is

$$T_{\mathrm{H}}^{\mathrm{U}} = \frac{n_p n_d \theta \delta}{z_u} G(g_p, g_d, z_{u,p}, z_u/z_{u,p})(s + vw).$$

Next, a bucket is purged every t_{purge} days. As this is piggybacked with a InsertRestraints operation, we do not need to read in the bucket again. We simply purge the bucket and write it out.

$$T_{\mathrm{H}}^{\mathrm{P}} = \frac{g_p g_d}{t_{purge}}(s + vw).$$

Finally, for a LookUpRestraints operation, for one incoming document, the operation is called once with n_{dup} docids and $n_p \phi$ profileids. The number of buckets accessed is $B(n_p \theta, n_{dup})$, where $B(x, y) = g_p \times g_d \times (1 - (1 - 1/g_p)^x)(1 - (1 - 1/g_d)^y)$. The time required for all incoming documents in one day is thus

$$T_{\mathrm{H}}^{\mathrm{L}} = n_d B(n_p \phi, n_{dup})(s + vw).$$

Summing up the total I/O time required in one day, we have

$$T_{\mathrm{H}} = T_{\mathrm{H}}^{\mathrm{I}} + T_{\mathrm{H}}^{\mathrm{U}} + T_{\mathrm{H}}^{\mathrm{P}} + T_{\mathrm{H}}^{\mathrm{L}}.$$