Using COLLABORATIVE FILTERING to Weave an Information TAPESTRY

David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry

Tapestry is an experimental mail system developed at the Xerox Palo Alto Research Center. The motivation for Tapestry comes from the increasing use of electronic mail, which is resulting in users being inundated by a huge stream of incoming documents [2, 7, 12]. One way to handle large volumes of mail is to provide mailing lists, enabling users to subscribe only to those lists of interest to them. However, as illustrated in Figure 1, the set of documents of interest to a particular user rarely map neatly to existing lists. A better solution is for a user to specify a *filter* that scans all lists, selecting interesting documents no matter what list they are in. Several mail systems support filtering based on a document's contents [3, 5, 6, 8]. A basic tenet of the Tapestry work is that more effective filtering can be done by involving humans in the filtering process.

In addition to content-based filtering, the Tapestry system was designed and built to support *collaborative filtering.* Collaborative filtering simply means that people collaborate to help one another perform filtering by recording their reactions to documents they read. Such reactions may be that a document was particularly interesting (or particularly uninteresting). These reactions, more generally called *annotations,* can be accessed by others' filters. One application of annotations is in support of moderated newsgroups.

META-1017
IPR2025-00430

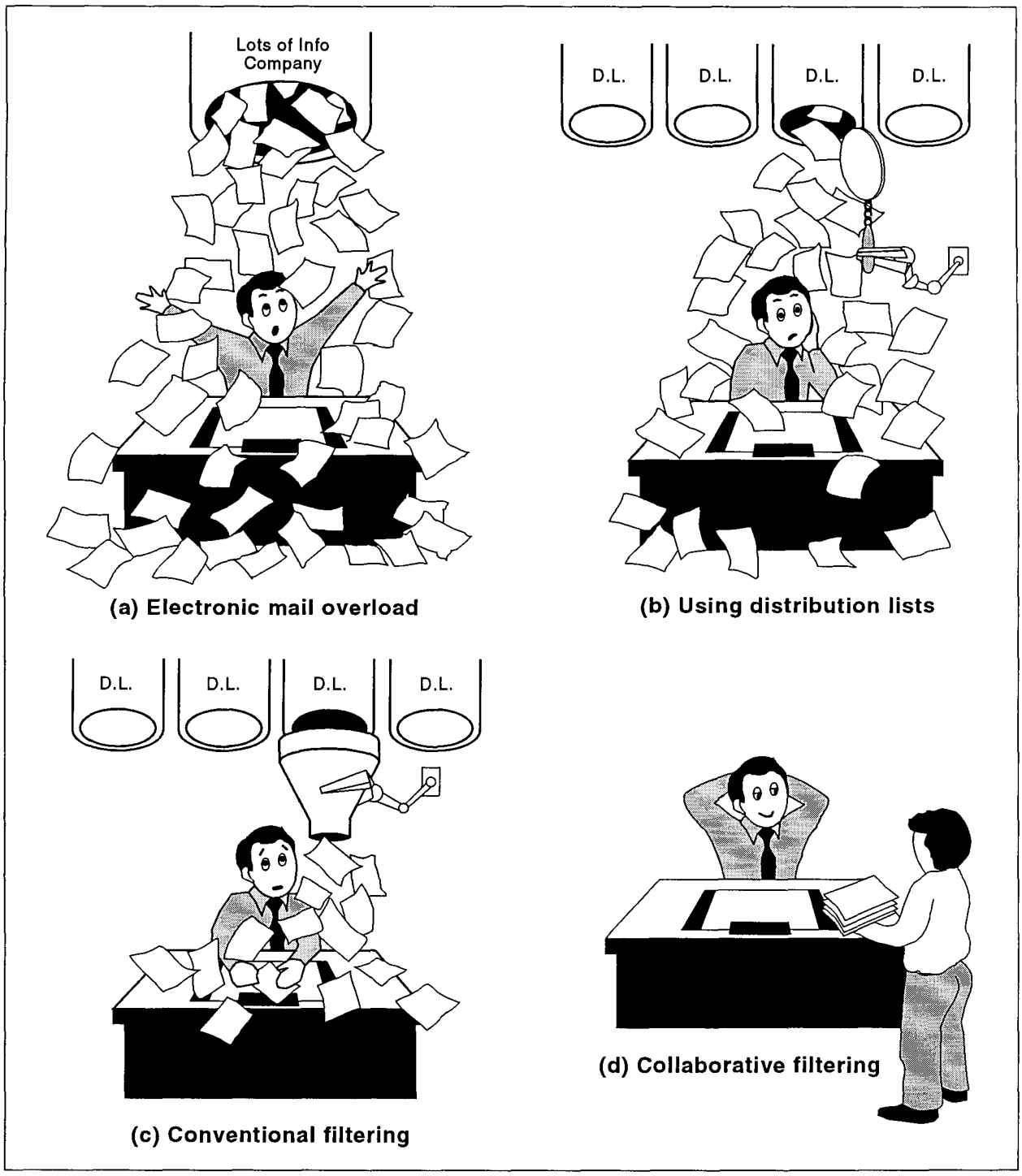


Figure 1. (a) electronic mail overload; (b) using distribution lists; (c) conventional filtering; (d) collaborative filtering

Currently, moderated groups have a single moderator, who selects a subset of messages to be posted to the moderated group. With annotations, a group can have many moderators. To see the newsgroup as it would be moderated by (say) Smith, simply filter for those articles that Smith endorsed with an annotation.

Implicit feedback from users (e.g., some user sent a reply to a document) can also be utilized in the filtering process. For example, suppose you would like to receive "interesting" documents from the NetNews newsgroup comp.unix-wizards in the mail, but you don't know how to write a search expression that characterizes them, and you don't have time to read them all yourself. However, you know that Smith, Jones and O'Brien read all of comp.unix-wizards newsgroup material, and reply to the more interesting documents. Tapestry allows you to filter on "documents replied to by Smith, Jones, or O'Brien."

Collaborative filtering is novel because it involves the relationship be-

tween two or more documents, namely a message and its reply, or a document and its annotations. Unlike current filtering systems, Tapestry filters cannot be computed by simply examining a document when it arrives, but rather require (potentially) repeatedly issuing queries over the entire database of previously received documents. This is because sometime after a document arrives, a human (say Smith) may read that document and decide it is interesting. At the time he replies to it (or annotates it), you want your filter to trigger and send you the original document.

Tapestry is more than a mail system, because it is designed to handle any incoming stream of electronic documents. Electronic mail is only one example of such a stream: others are newswire stories and NetNews articles [10]. Moreover, Tapestry is not only a mechanism of filtering mail, it is also a repository of mail sent in the past. Tapestry unifies *ad hoc* queries over this repository with the filtering of incoming data.

A typical scenario of Tapestry system usage is as follows. A user decides on 'mail filtering' as an area of interest. To find documents on this topic, the user issues an *ad hoc* query, perhaps by searching for the keyword "filtering." This returns too many documents. The user eventually discovers that searching, either for documents containing both 'information' and 'filtering,' or for documents containing "filtering" that received at least three endorsements, works much better. Having tested this, this search is installed as a query filter, and from now on, all new documents satisfying this filter will be delivered to the user's mailbox.

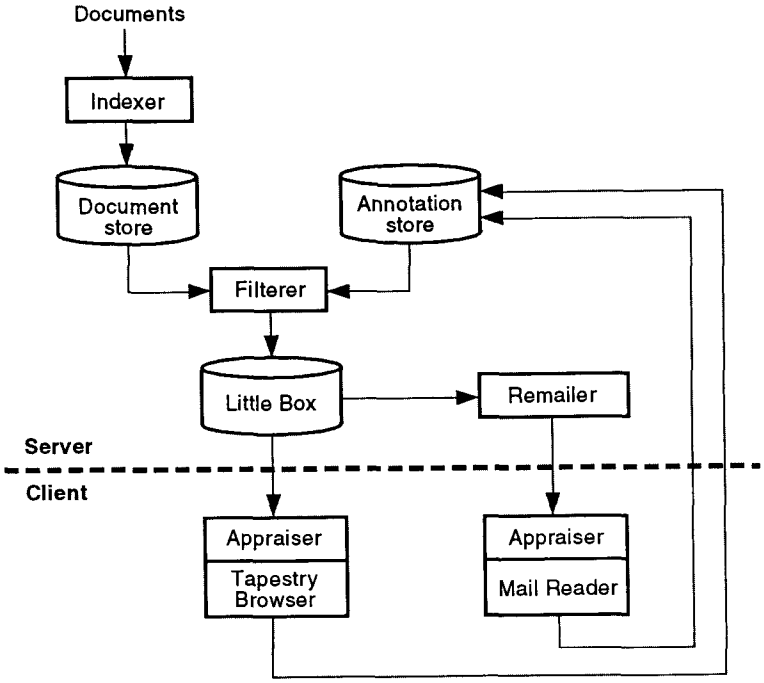
Architecture

Figure 2 shows the flow of documents through the major architectural components of Tapestry. These components are:

- *Indexer.* Reads documents from external sources such as electronic mail, NetNews, or newswires and adds them to the document store. The indexer is responsible for parsing documents into a set of indexed fields that can be referenced in queries.
- *Document store.* Provides long-term storage for all Tapestry documents. It also maintains indexes on the stored documents so that queries over the document database can be efficiently executed. The document store is append-only.
- *Annotation store.* Provides storage of annotations associated with documents. The annotation store is also append-only.
- *Filterer.* Repeatedly runs a batch of user-provided queries over the set of documents. Those documents matching a query are placed in the little box of the query's owner.
- *Little box.* Queues up documents of interest to a particular user. Each user has a little box, where documents are deposited by the filterer and removed by a user's document reader.
- *Remailer.* Periodically sends the contents of a user's little box to the user via electronic mail. This is intended for users who wish to access Tapestry with their current mail reader.
- *Appraiser.* Applies personalized classification to a user's documents (i.e., to those documents in the user's little box). This function can automatically prioritize and categorize documents.
- *Reader/Browser.* Provides the user interface for accessing Tapestry services. This includes facilities for such tasks as adding/deleting/editing filters, retrieving new documents, displaying documents, organizing documents into folders, supplying annotations, and running *ad hoc* queries.

Tapestry uses a client/server model. Two styles of interaction with the server are envisioned. The preferred mode of interaction is via a reader/browser which provides users with easy access to the full range of filtering and annotation functions. Users that do not want to, or are not able to, use the Tapestry browser can access Tapestry services from a conventional mail reader by having a remailer daemon send documents that match a user's filters to the user via electronic mail. Users can also send mail documents to the Tapestry server to invoke any of its operations, such as adding filters, adding annotations, and even running *ad hoc* que-

Figure 2. The flow of documents through Tapestry



ries. The Tapestry architecture is flexible about the location of the client/server split. Figure 2 illustrates one possible division.

Most of the Tapestry architecture follows naturally from the goal of providing collaborative filtering. For example, to support filters involving relationships between documents, there must be a document store. In order for users to communicate their ratings of documents, annotations are provided. The following subsections provide a more detailed rationale for some of the architectural components.

Document and Annotation Stores

Ideally, the Tapestry store will save documents forever. With the decreasing price of disk storage, this is becoming increasingly practical. As will be explained in the section "Filter Queries," documents as seen by the filter query language must be *immutable*. This means that once a document arrives in the document store, it is never modified. Thus Tapestry documents can be conveniently stored in newer technologies such as write-once, read-many (WORM) disks.

Annotations are stored separately from documents, with links connecting each annotation to its associated document. It might seem more natural to combine documents and annotations into a single store, with the annotations to a document appended as additional fields. There are several reasons why this was not done. First, since annotations for a document arrive after the document itself, appending annotations as additional fields would violate the immutability requirement. Second, some annotations are themselves complex objects, and those annotations are more simply stored as separate records with pointers back to the document they annotate. The issue of complex annotations also arises in the Tapestry query language (see the subsection entitled "Annotations").

Appraisers

Tapestry users want more than a binary sieve that can only accept or reject a document. For example, a user might want to assign priorities to messages, giving messages that announce meetings a higher priority than messages that announce promotions. And it would be handy to be able to change priorities. For example, the arrival of a message containing an update about a meeting (perhaps announcing a new meeting room) might cause the previous announcement to be given a lower priority, but probably not deleted, since it may contain details not repeated in the updating message.

To support classification of documents, Tapestry provides *appraiser* functions. Fitting appraisers into the overall architecture is not completely straightforward. At first it would seem simplest to run each user's appraiser on the server as documents arrive. However, this has a potentially serious drawback. Filtering on incoming documents is a very computationally intensive task. Imagine a Tapestry system with hundreds of users, each with dozens of filter queries, running on a document stream of tens of documents per minute. Running appraisers directly on the incoming document stream would put them on the critical performance path. To avoid this, the Tapestry architecture performs filtering in two steps. The first level of filtering is performed by filter queries, which are binary: they either accept or reject a document. The accepted documents for a user are then placed into that user's *little box*. The second level of filtering is done by appraiser functions that run only over the contents of the little box. Unlike the "big box" (the global Tapestry database), the little box will have few enough messages to allow them to be copied to the workstation. This allows the user's mail-reading program or browser to provide more complex appraiser functions than could be supported in the server.

Browsers

The Tapestry architecture supports browsers that combine the functions of a mail reader and a traditional document browser. Corresponding to the role of mail reader, such a browser should supply 'new mail' functionality. The server supports this by delivering the results of filter queries (new mail) to the little box, leaving it up to the client to remove the results. Browsers periodically run the appraiser over the documents in the little box, record their document identifiers, and then delete them from the little box. *Ad hoc* queries are another way to get documents into the browser. *Ad hoc* queries are made to the server in the same query language as filter queries and may return documents that were not previously in the browser.

In traditional mail systems, each mail reader obtains and stores its own copy of each message. Thus messages sent to a large mailing list are stored many times. Since Tapestry provides an immutable document store, Tapestry browsers need only keep a document identifier (i.e., pointer). When a user deletes a message from the browser, the document still exists and can be recovered using an *ad hoc* query.

Users of a browser would like to be able to issue queries that involve both document fields and *private fields*. Private fields store information such as whether a document has been read yet, and which folders it is in. A browser can store private fields along with other document fields, making them easily available for *ad hoc* queries. However, since documents must appear immutable to filter queries, and private fields are mutable, private fields can only be referenced by *ad hoc* queries, not filter queries.

Tapestry Query Language (TQL)

A key part of Tapestry is filtering documents, with the filters specified as queries. Hence, choosing the language in with filter queries are written was one of the important design decisions. One obvious choice was to use SQL[1], the widely used standard query language for relational databases. Adopting it as the Tapestry query language would have had the additional advantage of simplifying the implementation, because Tapestry is implemented on top of a commercial database which supports SQL.

We rejected using SQL as our query language for two reasons. First, there is a serious mismatch between the relational model and the Tapestry model of documents. The set of fields in a document is extensible, whereas SQL schemas have a

fixed set of fields. Also, SQL does not directly support sets, whereas many document fields are set-valued. Examples are the 'To:' field of mail messages, and the 'Newsgroups:' field of netnews articles. Second, we wanted to make it easy for users to type in *ad hoc* filter queries, and we thought the amount of boiler plate in SQL made that difficult.[1]

Thus Tapestry has its own language known as TQL (for *T*apestry *Q*uery *L*anguage). The next two subsections describe TQL informally by the use of examples. Even though TQL is easier to use than SQL, we expect most users will not use TQL directly, but instead will issue queries from a browser using predefined (but possibly parameterized) queries.

Basic Examples

A TQL query is a Boolean expression. It selects those documents that satisfy the expression. The set of allowable TQL expressions are similar to statements in first order predicate calculus. They combine "atomic formulas" with Boolean operators, and they can have free variables quantified by EXISTS or FORALL. Unlike predicate calculus however, TQL supports sets.

The simplest Tapestry queries are atomic formulas, which involve relational operators like = and < as well as the wildcard matching operators LIKE. An example is:

```
m.subject =
  'Next Tapestry Meeting'
```

which selects exactly those documents (or *m*essages) *m* whose subject field *m*.subject is "Next Tapestry Meeting."

TQL queries reference the fields of documents using m.*field,* where *field* is the name of a document field. Each field has a type. Some common fields and their types are listed in Table 1. Most correspond to fields of mail messages and newsgroup articles. One exception is 'words', which is the set of all words occurring in the body of the document.

More complex TQL queries are built up by combining atomic formulas with Boolean operators as in the following query:

```
(m.sender = 'Smith' OR
   m.date < 'April 15, 1991') AND
   m.subject LIKE '%Tapestry%'.
```

This query selects messages that were either from 'Smith' or else sent before April 15, and whose subject field included the word 'Tapestry'. As in SQL, the % symbol is a wildcard symbol that matches any number of characters.

The major difference between TQL and predicate calculus is TQL's support for sets. A simple example of a Tapestry query using set-valued fields is the atomic formula:

Table 1. Common fields and their types

to	set of strings
date	date
sender	string
cc	set of strings
subject	string
newsgroups	set of strings
in-reply-to	set of documents
words	set of strings

```
m.to = {'Joe', 'Tom'}
```

which matches documents whose m.to fields include 'Joe' and 'Tom' (and possibly others). Sets can involve operations other than =, such as the query:

```
m.to = {'Joe', LIKE '%Bill%'}
```

which asks for an m.to field containing at least 'Joe' and a name containing 'Bill'.

Quantified variables are needed for collaborative queries. An example is:

```
EXISTS (m1: m1.sender = 'Joe'
  AND m1.in-reply-to = {m})
```

which selects all documents *m* that Joe has replied to.

Finally, a user's filter queries can reference the queries of another user. For example, the TQL query:

```
m IN Terry.Baseball
  AND m.words = {'Dodgers'}
```

returns all the messages selected by Terry's 'Baseball' query that contain the word 'Dodgers'.

Annotations

The design of TQL presented so far follows rather naturally once the decision is made to have the query language match the form of electronic documents such as mail messages and NetNews articles. It is not so straightforward to decide how to handle annotations. As explained in the previous subsection entitled "Document and Annotation Stores," annotations are not stored as fields of the document they annotate. However, this does not preclude TQL treating them as additional document fields, and indeed this is the most natural representation for annotations such as priority. A notation such as 'm.a.priority' could be used to access the priority of a document, the 'a' serving to map out a separate name space for annotations. Similarly, the folders to which a document belongs could be a set valued field, 'm.a.folders'.

Things do not work smoothly for the more complex annotations used to support collaborative filtering. Consider trying to implement voting using additional document fields. If **vote** is to be an annotation field, then 'm.a.vote' would have to be a set of votes, each of which has a structure of its own, such as who the voter was, and the value of his vote. So a query such as "messages voted for by weiser" would be expressed as something like 'the set m.a.vote must have a member v with v.owner = weiser', and this would require extending the set notation of the previous section.

The way this query is written in TQL is:

```
a.type = 'vote'
  AND a.owner = 'weiser'
  AND a.msg = m
```

By introducing an annotation object, which always has a field **msg** that links it to a document, the kind of queries that support collaborative filters become simpler. We mentioned earlier that collaborative queries use EXISTS. The preceding query has an implicit EXISTS, and can also be written as:

```
EXISTS (a: a.type = 'vote'
  AND a.owner = 'weiser'
  AND a.msg = m)
```

The cost of introducing separate

[1]This is not meant as a criticism of SQL. Tapestry filter queries are much more specialized than general SQL queries, which is why they can be written with less boiler plate.

annotation objects is that simple queries such as "documents of priority 10" become slightly more complex:

a.type = 'priority' AND a.value = 10 AND a.msg = m

Since one of the major design goals of Tapestry was to support collaborative filtering, we felt the design with separate annotation objects was preferable.

Filter Queries

The heart of the Tapestry server is the Filterer, which executes users' filter queries. A straightforward method of implementing a filter query is to periodically execute it, say once every hour. This approach has the problem of returning all the old messages that matched the query the last time it ran, so something must be done to suppress these messages. Moreover, there is another more serious problem, namely that periodic execution can exhibit unpredictable behavior.

Consider the query: "select documents to which nobody has sent a reply." When a document is added to the database, it matches the query. However, once a reply document arrives, the document being replied to no longer matches the query. If a particular document were to arrive in the database at 8:15 and a reply to it arrived at 8:45, then the document would not be returned by a system that simply ran the filter query every hour on the hour (see Figure 3(a)), but would be returned by a system that ran it every hour on the half hour (b), since the document would match at 8:30. This raises the general question: "What are reasonable semantics for a filter query that executes repeatedly?" In other words: What guarantees can be provided to users about the set of documents returned by a filter query?

Users should not need to understand the implementation of the system in order to know what results to expect as the result of a filter query. The semantics should be independent of how the system operates internally and when it chooses to perform various operations, such as executing queries. Two users with the same filter query should see the same result data. This implies that the semantics of filter queries should be time-independent.

Continuous Semantics

Tapestry gives filter queries *continuous semantics,* which is defined as follows:

The results of a filter query is the set of data that would be returned if the query were executed at every instant in time.

That is, the system guarantees to show the user any document that would be selected by the query at any time. The system may implement this behavior in any number of ways, such as collecting results and presenting them to the user periodically, but the actual set of results eventually seen by the user is well defined and time-independent.

Rewriting the preceding in symbols, let $Q(t)$ denote the set of documents that would be returned by the execution of query Q over the database that existed at time t. That is, $Q(t)$ is the result of running Q at time t. When a query Q is executed with continuous semantics, it returns not $Q(t)$, but rather:

$$\bigcup_{s \le t} Q(s)$$

Filter queries are qualitatively different from one-time queries. Consider the user who wants to see all the documents that do not receive replies. The obvious formulation, "select documents to which nobody has sent a reply," when executed as a filter query, would return every document to the user, since every document has no replies when it arrives. This is undoubtedly not what the user intended. The problem does not lie with continuous semantics, but rather with the user's imprecise specification of his filter query. Finding the documents that *never* receive a reply would require waiting forever, but in practice a short wait will return a good approximation, since most messages are replied to quickly. Thus a more precise query would be something like: "select documents that are more than two weeks old and to which nobody has sent a reply." This illustrates the point that some queries only make sense when executed on a one-time basis, and are not suitable as filter queries that are repeatedly executed.

Implementation

How can continuous semantics be realized in a practical system? Certainly, running a query at every instant in time is not possible, and if it were possible, would not be practical. This remainder of this section gives an overview of techniques for providing continuous semantics in an effective and efficient manner. An earlier paper gives full details of how this is done [13].

The key to providing efficient continuous semantics is the following observation: Given a query whose result set is nondecreasing over time, the simple technique of periodically executing the query and returning the new results yields continuous semantics. Such a query is said to be *monotone.* The frequency with which a monotone query is executed simply affects the size of each batch of results, not the collective set of results.

Tapestry implements filter queries with continuous semantics in two stages. First, a query is rewritten as a monotone query that returns at least all documents currently matching the original query or else matched it at some time in the past. If the rewritten query is Q, and Tapestry has previously evaluated Q at time τ, then at time t Tapestry can implement continuous semantics by returning $Q(t) - Q(\tau)$ to the user, where '–' stands for set difference.

In general, the sets $Q(t)$ and $Q(\tau)$ are almost the same, and contain mostly documents that have already been returned to the user. Computing $Q(t) - Q(\tau)$ is very inefficient, since $Q(t)$ and $Q(\tau)$ both return large sets, but then most of these documents 'cancel' when $Q(t) - Q(\tau)$ is computed. So Tapestry has a second stage, in which the monotone query Q is rewritten as an *incremental query,* $Q^I(\tau, t)$, that can quickly compute an approximation to $Q(t) - Q(\tau)$

To summarize the discussion so far, when a filter query is submitted to Tapestry, it is first rewritten to a monotone query Q, and then Q is further rewritten to an incremental Q^I. This incremental query is what is used by the Tapestry filterer. The filterer repeatedly runs the incremental query, queues up the selected documents for delivery to users, records the time at which each query

was run, waits some period of time, and then repeats this process using the recorded times as parameters to the incremental queries. This algorithm is shown in Figure 4.

We can now explain why Tapestry does not allow documents to be deleted (that is, uses an append-only document store). Because the filterer runs at discrete times, if documents could be removed, different users could receive different results from the same filter, depending on when the filter ran relative to document deletion. This would be a violation of continuous semantics.

Examples

A couple of example should give the flavor of the query transformations. Consider the query "show messages sent by Joe," which can be expressed in TQL as:

```
m.sender = 'Joe'
```

This query is already monotone since the set of messages sent by Joe is strictly nondecreasing over time. Therefore, the query simply needs to be converted into an incremental form. Recall that the incremental query $Q^I(\tau, t)$ should return messages that began matching the original between times τ and t. For the preceding example, the incremental query considers all messages that arrived in this time range:

```
m.sender = 'Joe' AND
  (τ < m.ts AND m.ts ≤ t)
```

The "ts" field is a timestamp added by Tapestry when the message arrives in the document store.

As a more complicated example, consider the query "show bug reports that are more than 2 weeks old and have not been answered." In TQL, this can be written as:

```
m.to = 'BugReports' AND
  m.ts + [2 weeks] < now() AND
NOT EXISTS (mreply:
  mreply.in_reply_to = {m})
```

This query is not monotone since it may select a message after the message becomes two weeks old and stop selecting the message when a reply arrives. Tapestry converts it into the following monotone query:

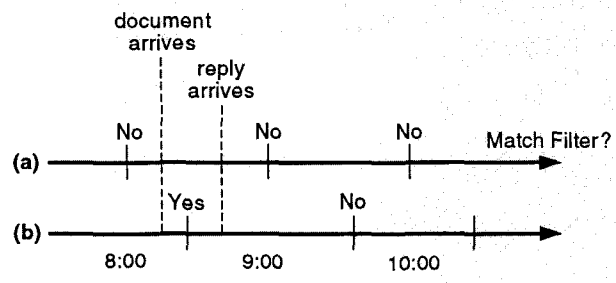
```
m.to = 'BugReports' AND
  m.ts + [2 weeks] < now() AND
NOT EXISTS (mreply:
  mreply.in_reply_to = {m}
  AND mreply.ts < m.ts + [2 weeks])
```

This monotone query has a slightly different meaning than the original query, but one that is consistent with continuous semantics. Specifically, it says "show bug reports that are not answered within 2 weeks."

The incremental version of this query considers all messages that became two weeks old in the time between τ and t:

```
m.to = 'BugReports' AND
  m.ts + [2 weeks] < now() AND
  (τ < m.ts + [2 weeks] AND
  m.ts + [2 weeks] ≤ t) AND
NOT EXISTS (mreply:
  mreply.in_reply_to = {m} AND
  mreply.ts < m.ts + [2 weeks])
```

Because t and now() are the same time in practice, this query can be simplified a bit by removing the "AND m.ts + [2 weeks] < now()" clause.




```
Set τ = −∞
FOREVER DO
  set t := current time
  Execute query Q^I(τ, t)
  Return result to user
  set τ := t
  Sleep for some period of time
ENDLOOP
```

Figure 3. Nondeterministic behavior of filters

Figure 4. Filter query execution

The Current System

A system that embodies the architecture presented in the previous section, "Architecture," is currently in use by a small number of researchers. The following subsections describe the implementation of various components of the current Tapestry system.

Database Manager

Tapestry stores documents, annotations, and filter queries in a commercial relational database management system provided by Sybase [11]. Information about messages is stored in a set of relational tables. A single table does not suffice since this information does not fit cleanly into the relational model. In particular, there is no single collection of attributes that apply to all messages, and some of the attributes, such as the set of recipients or newsgroups for a message, are set-valued. Information that is common to all messages, and is not set-valued, is stored in a table that has one entry per message. Other information that varies from message to message is stored in an auxiliary table. Each message may occupy one or more rows in this table. Similarly, set-valued attributes are stored in a special table in which each value of a set occupies a single row. Annotations, which, like messages, have an extensible set of attributes, are stored in several tables as well. As stated earlier, one of the principal motivations behind the design of TQL was to hide this complex database schema from Tapestry users.

Indexer

The indexing program is responsible for understanding a given document format, extracting attributes from the document, and storing these in the database. Logically, a separate indexing program exists for each type of document that is added to the Tapestry system. For example, the format of NetNews articles and mail messages is very different than that of articles appearing in the *New York Times*. Fortunately, the indexer is the

only part of the Tapestry system that is sensitive to the format of a document. New sources of documents can be added simply by writing new indexing programs.

For NetNews, the indexer takes all the header fields in the message and translates them into tapestry message fields. In addition, the words in the body of the message are added to a set-valued Tapestry field named 'words'. Words on a stop list of common English words are not added, and each word is stemmed to eliminate inflected forms (e.g., 'ran' is indexed as 'run'). No proximity of frequency information is kept for words in the body.

As of this writing, we are indexing a subset of NetNews (the 'comp' subtree), keeping the last 100MB of data around at any given time. This is about 12 days worth of data, or 43,000 messages. Our Sybase tables and indexes occupy an additional 300MB of storage.

TQL-to-SQL translator

Before a TQL query can be executed over the Tapestry database, it must be converted to SQL, the query language used by the Sybase database manager. The Tapestry system compiles (or translates) each TQL query into an equivalent SQL query. For *ad hoc* queries, this translation is done directly on the query provided by the Tapestry user. For filter queries, the TQL statement is first converted into its bounding monotone query and incrementalized, as described in the preceding section, "Filter Queries," and then translated into SQL. The SQL query for a filter is then maintained in the Sybase database as a stored procedure. A stored procedure is more efficient than an *ad hoc* query since the query optimization overhead is amortized over the many executions of the query.

Because information about messages and annotations is distributed throughout several tables in the Tapestry database, the SQL equivalent of most TQL queries involves one or more database join operations between one or more tables. Therefore, the SQL queries can be quite complicated. Figure 5 shows a sample TQL query along with the resulting SQL query. Studies have shown that a good query optimizer, provided with suitable database indexes, can produce query plans that allow these complex queries to run efficiently. In particular, the execution cost of an incremental query produced by our translater is proportional to the number of messages added to the database since the query last ran and is not dependent on the overall size of the database. See our paper on continuous queries for more details [13].

Remailer

Messages that are selected by a user's filter queries are queued up for delivery to that user. These queues, which constitute the users' 'little boxes', are also stored in the Tapestry database. Eventually, we plan to build Tapestry clients that access these queues directly, including a Tapestry browser. Meanwhile, we have built a remailing agent that periodically retrieves all of the messages that have been selected for a user and send each message to that user via electronic mail. Each message is modified to include an extra header field that indicates which filter(s) selected the messages. This is used as input to the appraiser, permits a user to understand why the message was selected, and provides a valuable feedback for debugging or refining a filter query.

Mail Readers

Having the Tapestry server send selected messages to users electronically eliminated the need to build special clients. An important advantage is that users can continue to use their favorite mail readers to manage both their private mail and Tapestry documents selected by their filter queries. While we do not believe this to be the ideal means of interacting with the Tapestry service, it has allowed us to quickly make use of the filtering capabilities.

Some Tapestry clients use the Andrew Messages reader developed at Carnegie Mellon University [9]. Like most modern mail readers, it provides a nice user interface for reading messages and moving them into mail folders. Moreover, it supports the "FLAMES" language, which allows users to write a simple form of 'appraisers' that automatically move messages matching a given predicate or rule into a given folder. In particular, users can write FLAMES rules to identify and process messages that were sent by the Tapestry service and selected by a certain filter query.

To experiment with a different type of appraiser function, we added prioritizing queries to the Cedar-based mail reader developed at Xerox PARC called Walnut [4]. Users can supply a set of queries that can be applied to all incoming messages. As with the FLAMES rule, these queries can look for the special header field indicating that a message is from the Tapestry service. Each query assigns a numerical priority to messages that match the query. If a message

Figure 5. Example of translation from TQL to SQL

```
EXISTS(ml:((τ < m.ts AND m.ts ≤ Now()) OR
           (τ < ml.ts AND ml.ts ≤ Now))) AND
           ml.sender = "Joe" AND
           ml.in_reply_to = {m})                    TQL
-------------------------------------------------------
SELECT m.id FROM msgs m WHERE                       SQL
  EXISTS(SELECT * FROM msgs ml WHERE
    ((@tau < m.ts AND m.ts <= getdate()) OR
    (@tau < ml.ts OR ml.ts <= getdate()) AND
    (ml.sender = "joe") AND
    EXISTS(
      SELECT * FROM reply_to tl, msgs tml
      WHERE tl.id = ml.id AND tl.reply_ref = 1
      AND
        tl.msg_id = tml.msg_id AND
        tml.id = m.id))
```

matches several queries, then it is assigned the maximum of the priorities. Walnut will display messages within a folder in various orders including priority order. This allows users to quickly see the high-priority messages (and ignore the low-priority ones). To date, our experience with prioritizing queries has been quite positive. They have convinced us of the value of having appraisers that further classify and organize messages selected by filter queries.

Name Canonicalizer

It is very common for queries to involve the names of mail senders and receivers. There are two problems with these names. First, a given person usually has multiple electronic names. Second, if a name has any chance of being unique, it must be highly qualified, and that works against our goal of making it easy to type an *ad hoc* query. This subsection presents our design (not yet implemented) for dealing with naming.

The second problem is the easiest to solve. In the "official" TQL query language, names are fully qualified. However, users will normally enter queries via a browser. Thus, the browser can offer an **expand** command, which takes a shorthand and expands it to be fully qualified. This not only saves typing, but also serves to verify that the name was expanded as expected.

The first problem is more difficult, because there is not a 1:1 mapping between names and people. Suppose we simplify the problem by assuming that each person referenced in a query can be uniquely named with an Internet name of the form name@site, where name and site each are of the form part1.part2 . . . There is still a problem because both names and sites can have many aliases, and so the mapping is many:1. In other words, although a person can be specified unambiguously, it is difficult to find all documents involving a given person, because of all the aliases.

Our solution involves creating a canonical form for each name, which is a fully qualified Internet name, along with a program that converts names to canonical form. For the 'From' field of mail originating within PARC, the canonicalizer can do a perfect job. For other names, it must use heuristics.

Once such a canonicalizer exists, it can be used when executing a query such as

m.sender = 'weiser'

It would be too expensive to perform the three steps of examining the **Sender** field of each document, canonicalizing it, and then comparing that with canonical form of 'weiser' each time an incremental query was executed.

Instead, the raw names in documents are processed as they arrive in Tapestry. Although the names could simply be replaced with their canonical forms, that is not done because the canonicalizer is imperfect, and we want to make it easy to update its translations when an error is discovered.[2]

Our solution is that as documents arrive in Tapestry, each raw name in the document that has not been seen before is run through the canonicalizer, and added to a table that contains [raw name, canonical name] pairs.

Then the query

m.sender = 'Weiser:PARC:Xerox'

is converted to

m.sender = names.canonname AND
  names.rawname =
  'Weiser:PARC:Xerox'

The advantage of having the level of indirection is that we can easily compensate for incorrect heuristics in the canonicalizer by changing entries in the **names** table.

Summary and Future Work

Tapestry is an experimental system designed to receive, filter, file and browse electronic documents that arrive in a continuous stream. Because this class of documents includes email, Tapestry is intended to be used as a replacement for current email systems.

The novelty of Tapestry lies in its support for collaborative filtering. Users are encouraged to annotate documents, and these annotations can then be used for filtering. We envision two types of readers for various classes of documents. Eager readers will read all the documents in the class in order to get immediate access. More casual readers will wait for the eager readers to annotate, and read documents based on their reviews. Experience with NetNews suggests that there will not be a lack of readers willing to be 'eager' annotators.

When a Tapestry user installs a filter that uses annotations, documents matching that filter are returned as soon as the document receives the specified annotations. Thus Tapestry filters can be thought of as running continuously. The primary technical innovation in Tapestry is an efficient algorithm for implementing filter queries that have predictable semantics.

Future works falls into two categories. First, we need to accumulate more user experience with Tapestry so we can better analyze how well the design actually works in practice. Second, the Tapestry design presented in this article is missing a few important pieces. One of these pieces is security: the integration of private mail with public information such as NetNews is unlikely to be widely accepted without a strong security scheme. Another missing piece is the browser. We have not yet done a detailed design of a browser. The integration of different information streams provided by Tapestry may enable some interesting new browser techniques.

Acknowledgments

We would like to thank Pavel Curtis, Doug Cutting, and Maria Okasaki for carefully reading a draft of this article.

References

1. ANSI Database language SQL. (Apr. 1991), DIS 9075:199x(E).
2. Denning, P.J. Electronic junk. *Commun. ACM 25*, 3 (Mar. 1982), 163–165.
3. Gifford, D.K., Baldwin, R.W., Berlin, S.T. and Lucassen, J.M. An architecture for large scale information systems. In *Proceedings Tenth Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985), pp. 161–170.
4. Kent, J., Terry, D. and Orr, W.S.

[2]This has the unfortunate side effect of destroying append-only semantics, but there does not seem to be any way around this problem.

Browsing electronic mail: Experiences interfacing a mail system to a DBMS. In *Proceedings Fourteenth International Conference on Very Large Databases* (VLDB), (Los Angeles, Calif., Aug. 1988), pp. 112–123.

5. Lutz, E., Kleist-Retzow, H.V. and Hoerning, K. MAFIA—An active mail-filter agent for an intelligent document processing support. *Multi-User Interfaces and Applications,* S. Gibbs and A.A. Verrijn-Stuart, Eds., North Holland, 1990, pp. 16–32.
6. Malone, T.W., Grant, K.R., Turbak, F.A., Brobst, S.A. and Cohen, M.D. Intelligent information sharing systems. *Commun. ACM 30,* 5 (May 1987), 390–402.
7. Palme, J. You have 134 unread mail! Do you want to read them now? In *Proceedings IFIP WG 6.5 Working Conference on Computer-based document Services* (Nottingham, England May 1984), pp. 175–184.
8. Pollock, S. A rule-based message filtering system. *ACM Trans. Off. Inf. Syst. 6,* 3 (July 1988), 232–254.
9. Rosenberg, J., Everhart, C.F. and Borenstein, N.S. An overview of the Andrew Message System. In *Proceedings SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology* (Stowe, Vt., Aug. 1987), pp. 99–108.
10. Smith, B. The Unix Connection. *Byte 14,* 5 (May 1989), 245–251.
11. Sybase. Transact-SQL user's guide. Sybase, Inc., Oct. 1989.
12. Terry, D.B. 7 steps to a better mail system. *Message Handling Systems and Application Layer Communication Protocols,* P. Schicker and E. Stefferud, Eds., North Holland, 1991, pp. 23–33.
13. Terry, D.B., Goldberg, D., Nichols, D. and Oki, B. Continuous Queries Over Append-Only Databases, In *Proceedings ACM-SIGMODS Symposium on the Management of Data,* (San Diego, June 1992), pp. 321–330.

CR Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation, retrieval models*; H.4.3 [**Information Storage and Retrieval**]: Communications Applications—*electronic mail*

General Terms: Design, Documentation

Additional Key Words and Phrases: Information filtering, Tapestry

About the Authors:

DAVID GOLDBERG is a member of the research staff at Xerox Palo Alto Research Center. Current research interests include full-text databases, floating-point and user interfaces.

DAVID NICHOLS is a member of the research staff at Xerox Palo Alto Research Center. Current research interests include distributed systems and information retrieval.

Authors' Present Address: Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304; email: {goldberg, nichols}@parc.xerox.com

BRIAN M. OKI is a senior member of the technical staff at Teknekron Systems, Inc. Current research interests include distributed systems for real-time applications, information storage and retrieval systems, fault-tolerance, databases, and program language methodology. **Author's Present Address:** Teknekron Software Systems, Inc. 530 Lytton Avenue, Suite 301, Palo Alto, CA 94301; email: boki@tss.com

DOUGLAS TERRY is a member of the research staff at Xerox Palo Alto Research Center. Current research interests include distributed computing, ubiquitous information systems, and database management. **Author's Present Address:** Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304; email: terry.pa@xerox.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is give that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/92/1200-061 $1.50

CARE plants the most wonderful seeds on earth.

Seeds of self-sufficiency that help starving people become healthy, productive people. And we do it village by village by village. Please help us turn cries for help into the laughter of hope.