The Bloomba Personal Content Database

Raymie Stata
Stata Labs, Inc., UC Santa Cruz
raymie@{statalabs.com, cs.ucsc.edu}

Patrick Hunt
Stata Labs, Inc.
phunt@statalabs.com

Thiruvalluvan M. G.
iSoftTech, Ltd.
thiru@isofttech.com

Abstract

We believe continued growth in the volume of personal content, together with a shift to a multi-device personal computing environment, will inevitably lead to the development of Personal Content Databases (PCDBs). These databases will make it easier for users to find, use, and replicate large, heterogeneous repositories of personal content.

In this paper, we describe the PCDB used to power Bloomba, a commercial personal information manager in broad use. We highlight areas where the special requirements of personal content and personal platforms have influenced the design and implementation of our PCDB. We also discuss what we have and have not been able to leverage from the database community and suggest a few lines of research that would be useful to builders of PCDBs.

1. Introduction

End users are facing two secular trends we believe will drive the development of a new type of "very large database:"

- **Proliferation of data.** Facing an explosion of email, office documents, IM transcripts, photos, and music, people need to manage an increasing number of digital items (in our view, what matters is the number, not the size, of items). Traditionally, hierarchical folders have been the primary means of managing these items. However, folders don't scale, and for increasing numbers of users, this problem is reaching crisis proportions.
- **Proliferation of devices.** Given multiple desktops (home and office), PDAs, smart phones, the Internet, and even in-dash car computers, the increasing volume of personal content is necessarily being distributed over multiple devices. Currently, movement of personal data among these devices is painful, if possible at all. Over time, this needs to become seamless if users are going to be able to fully utilize their digital content.

Today, users face a hodge podge of software and services for storing this data. Email, for example, is sometimes stored in specialized, local files (e.g., Outlook's .pst files), sometimes on servers, and sometimes replicated on both. Some office documents are stored in the local file system, but a surprisingly large number of them are stored as attachment in one's email repository. Photos are often stored in the file system, possibly indexed by specialized software running beside the file system, and also possibly replicated to a Web server. Contact information, like email, might be stored in a specialized, local file (again, a .pst file) and also synchronized out to a PDA and a phone. These various storage schemes do not interoperate, are all folder based, and are difficult to manage.

We believe this hodge podge of storage systems will be replaced by a single *Personal Content Database,* or PCDB. The PCDB will encompass all of the user's personal data: email, documents, photos, and even Web pages visited by the user. It will use associative retrieval, rather than folders, as the primary means of organizing. The PCDB will transparently move content among a user's multiple devices, and the PCDBs of multiple users will share content with each other based on policies set by the user. PCDBs will initially be small by VLDB standards – say, tens to small hundreds of gigabytes – but current trends suggest that they will grow to terabytes.

With Bloomba [1], we are trying to bring this vision to life. Bloomba is a search-based, desktop email client, with support for RSS, contact, and calendar

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

Table 1. Properties of Web vs. Personal Search

	Web Search	**Personal Search**
Corpus	Global & Infinite	Local & Finite
Activity	Discovery	Recovery
Computing Environment	Dedicated, Controlled	Borrowed, Hostile
Interface	Single task	Multi task
Dynamics	Batch	Interactive

management, built on a proprietary Personal Content Database. Bloomba is a commercial product, in wide use, primarily in business contexts. It replaces, rather than runs next to, other applications such as Eudora or Outlook. Its advantage is its fast, scalable search, and the productivity that results. As one CEO put it, "I estimate that Bloomba saves me about an hour per day, as a result of faster searching, quicker filing, better spam filtering and the automated organization of the smart groups." A review in Business Week put it more simply: "Bloomba is email that blows the others away."

Bloomba and its PCDB do not yet fulfill our full vision for PCDBs. Most significantly, it doesn't yet support replication. However, the positive response Bloomba has received so far suggests that PCDBs, even in limited forms, bring great value to users and will have an important role to play in the future of personal content management. In the meantime, we chose to focus first on email for a reason. Email is the largest, fastest-growing, and most dynamic collection of documents managed by most users. Also, it is becoming the primary gateway for bringing content into a personal environment, especially in a business setting. In tackling email, we've learned a lot about building PCDBs.

This paper provides an architectural overview of Bloomba's PCDB, with an eye to placing it in the context of the tradition of database work reported at VLDB and elsewhere. In the next section we describe some of the requirements and environmental constraints that shaped the PCDB. Section 3 describes the design of the PCDB. Section 4 provides a bit more design detail on a particularly interesting part of the PCDB, query execution. Section 5 discuses the concepts and technologies from the database community that we have and have not used; it also suggests areas of future research that would be of particular relevance to PCDBs. Finally, Section 6 offers some concluding remarks.

2. Requirements

Today, people are talking about personal search as if it were a simple extension of Web search. On the one hand, the success of Web search has elevated "search," as a User Interface metaphor, to a point where it is almost as widely understood as the venerable "folder" metaphor. This is a significant breakthrough. Even two years ago, the average business user had difficulty grasping Bloomba's search-based UI design. Thanks to the success of Web search, users today can quickly understand Bloomba and other applications that incorporate search as a major UI metaphor.

On the other hand, when we consider search as an application rather than a UI metaphor, Web and personal search are quite different. Further, personal search is hard, but for different reasons than Web. Table 1 summarizes some of these differences. When considering only the corpus, personal search seems much easier. The Web is vast and global; the desktop is local and finite. From a pure scale perspective, the Web is the harder problem. But personal search presents significant challenges in other ways.

Activity. First, it is easier to *discover* information than *recover* it. The simple query "Aaron Burr," for instance, will yield thousands of documents about him on the Web. For the most part, information on the Internet wants to be found; it is intentionally – even aggressively – optimized for search engines results.

But recovery of personal information requires higher precision. There is typically only one right answer, one message or document (or version of the document!) the user is looking for. Making matters worse, people typically adopt a steep discount function on our time. This means they won't invest the time to organize up front – nor should they, with the tsunami of digital information they face – so they invest it on the back end, with the expectation of a quick recovery process. Further, they know they once had the information. So the process of looking for things can quickly feel redundant, frustrating and interminably time-consuming.

Computing Environment. Web search engines are built from thousands to tens of thousands of dedicated machines. These machines are assigned specific tasks – some crawl, some index, some respond to queries. All the resources of a machine are dedicated to its one task.

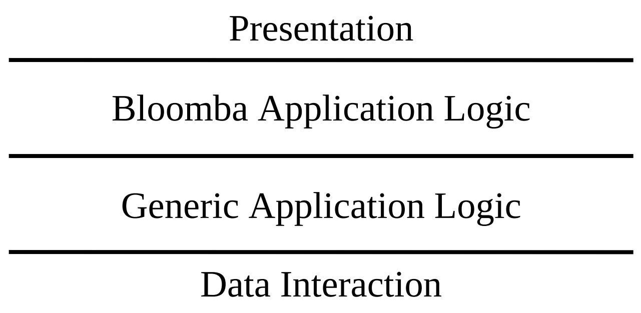


Figure 1. Architectural layering of Bloomba

On personal machines, resources such as computing cycles, RAM, and I/O transactions are expected to be dedicated primarily to the user's foreground activity. When this expectation is violated, users quickly get impatient. Thus, resources for indexing and disk-structure maintenance must be borrowed from this primary use.

In addition, Web search engines typically house their machines in dedicated host facilities with backup servers, restoration services, and redundant power supplies. Operating systems, memory configurations and hardware configurations are all finely tuned to be application-specific. The desktop is another world entirely. It's hostile. File scanners of various types can lock files for long periods of time, preventing even reads from occurring. Virus detectors and "garbage collectors" feel free to delete files they deem dangerous or redundant. And of course, there are users, who feel free to remove files and even entire directories they (mistakenly) deem to be unnecessary.

Interface. The interface to Web search engines supports a single task: executing queries. PCDBs are embedded in applications that support multiple tasks. In email, for example, finding messages is one of many tasks; users also want to view messages (and, at times, *avoid* reading messages), create them, and even relate them to their on-going projects. Search can support many of these tasks, but only if the UI is redesigned around the search paradigm (rather than being relegated to a "fast find" dialog box).

Dynamics. For the purposes of an individual query, content on the Web is static. Naturally, it changes over time, but the lifetime of a Web query is far shorter than the update cycle of the index.

Personal content, on the other hand, is dynamic, in two directions. First, new information is constantly being added. Emails come in and go out at a dizzying pace. New documents are created and sent and received as attachments. And all sort of content is being downloaded off the Web. Second, the information itself is dynamic over time. Emails change state as they are read, sent, and filed. Plus, capturing different versions of documents is essential to the flow of business. Business contracts, negotiations and agreements all have multiple versions; retrieving the correct version can have broad and deep financial implications.

In a PCDB, the lifetime of queries far exceeds these changes. As a simple example, when you look at the Inbox in a search-based email client, you are looking at the output of a query: as new messages enter the system, this output needs to be updated accordingly.

3. The Bloomba PCDB

As mentioned in the Introduction, Bloomba is a desktop, search-based, email, RSS, contact, and calendar manager, built on a proprietary Personal Content Database. In this section, we summarize the high-level design of Bloomba and its PCDB. We start by describing the layering of the Bloomba application, to provide the overall context in which the PCDB was designed. Next, we provide a brief functional description of the PCDB, summarizing its data model and the operations it supports. Finally, we describe the architecture of the PCDB itself, listing its components and explaining their functions.

3.1 Application Architecture

Even though Bloomba is a desktop system, it is structured like a modern N-tier server-based application. As illustrated in Figure 1, Bloomba is rigorously layered into four layers. Starting from the bottom, these layers have the following responsibilities:

- **Data Interaction.** The data interaction layer is responsible for data access and storage in our system. A central component of this layer is our PCDB, but it also includes other functionality, such as the protocol-specific part of message download, and the foreign-repository parsing part of import.

Rigorously separating data interaction from application logic has served us well. For example, the data access part of import – which we call a DatastoreReader – is responsible for reading foreign data stores and mapping its content in a universal model. Separate DatastoreReaders for Eudora, Mozilla, Outlook, and Outlook Express all map the foreign data to this universal model. Common application logic is used to map this universal model into Bloomba's model. This separation has made it easier to add new importers and also to tweak the details of how we map any data store into Bloomba.

- **Generic Application Logic.** The Generic Application Logic is responsible for that part of the application logic that would be common to any search-based PIM application. It includes document download and insertion, query execution, and the message-rules engine.
- **Bloomba Application Logic.** The Bloomba Application Logic is responsible for that part of the application logic that is specific to Bloomba, including folders, saved-searches, and smart groups. (As this implies, we do not see folders as being inherent to search-based email – a point illustrated by the design of Gmail [5].) In places, the generic application logic uses callback functions to call into the Bloomba Application Logic. For example, between downloading a message an inserting it into the PCDB, the Generic Application Logic calls back into the Bloomba Application Logic to ensure that the message is placed into the Inbox, which exists only at the level of the Bloomba Application Logic.
- **Presentation.** The Presentation Layer is responsible for direct interaction with the user. It's a thin layer which relies on the Bloomba Application Logic to implement the smarts of the program. This would allow us, for example, to build an alternative UI to Bloomba tailored to, say, smaller screens, such as those found on smart phones and PDAs. Also, the API between the Presentation Layer and the Bloomba Application Logic was designed to be friendly to high-latency environments, opening up the possibility of running the core part of Bloomba on one machine and its presentation another machine separated by a LAN or even a WAN.

3.2 PCDB Functionality

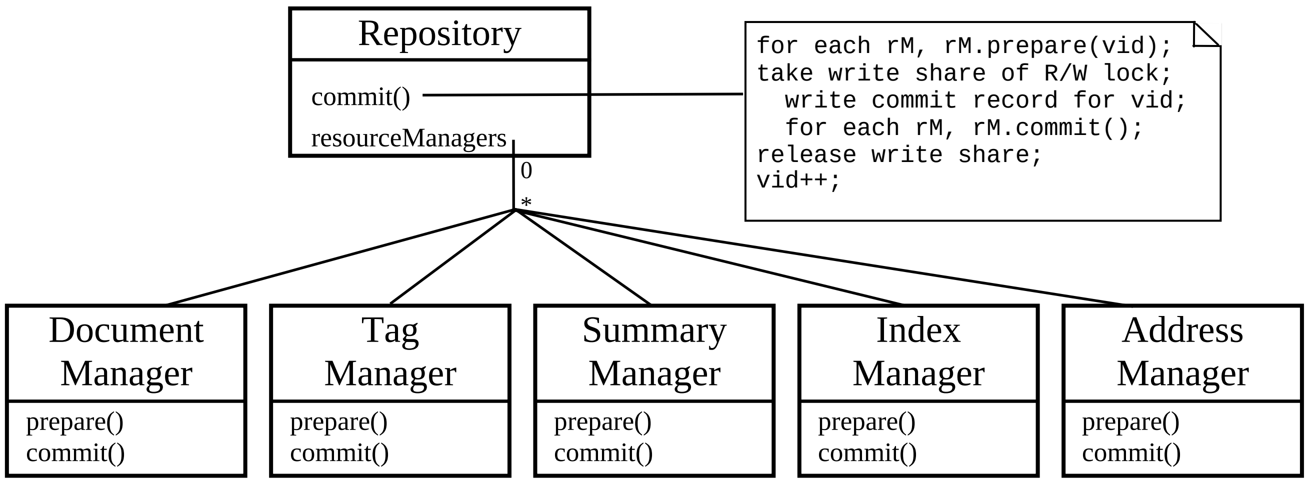
The PCDB is the central element of Bloomba's data-interaction layer. It is responsible for storing, searching, and returning documents.

The PCDB supports a simple, document-oriented data model more typical of an Information Retrieval system than a SQL database. In particular, the PCDB stores and retrieves objects we call *documents*. Documents themselves are immutable (although not immortal). However, documents are also associated with a mutable set of *tags*. Thus, when Bloomba receives an email message, it stores it in the PCDB as a single document. Bloomba then uses a tag to mark the message as "unread" and also uses a tag to indicate that the message is being stored in the Inbox. As the user manipulates the message, e.g., by reading it and/or moving it to another folder, the Bloomba Application Logic manipulates this tag set, not the actual message.

The PCDB's document abstraction is slightly richer than a plain sequence of characters. In particular, documents are recognized to have (immutable) fields, e.g., "From" and "Subject" are considered fields of an email message. Further, the PCDB also has a primitive notion of documents having a tree structure, i.e., a notion of compound documents containing other documents. We are considering moving to a more normalized model in which sub-documents are referred to by reference rather than by inclusion, allowing, among other things, space-savings when the same document appears as an attachment in multiple messages.

Documents in the PCDB have two different identifiers. *Document identifiers* are permanent, unique keys for individual documents. *Object identifiers* are non-unique identifiers that are shared by multiple documents that are meant to be versions of one another. For example, when a contact record in Bloomba is updated, a new document is created containing the updated version of the record. This new contact record has a unique document id, but it shares an object id with the previous versions of the record. To date, the Application Logic rarely uses object identifiers, a bit to our surprise. One idea we're considering is to eliminate object identifiers in favor of a "related-to" relationship which can track document precedence more generally than a straight versioning relationship.

In addition to storing documents themselves, the PCDB also stores *summary records* for the documents it is storing. Through a document's summary record, the application has fast access to useful summary information, such as the subject of an email, and also access to the tags associated with the document. Initially, the summary record was used solely for the purpose of quickly displaying a summary-list of large



```
for each rM, rM.prepare(vid);
take write share of R/W lock;
  write commit record for vid;
  for each rM, rM.commit();
release write share;
vid++;
```

Figure 2. Component Diagram for the PCDB

result sets. However, the summary record turns out to be the only way to retrieve the full tag-set associated with a document. Thus, the summary record has been useful for implementing many pieces of application logic and is even used by the query engine.

Against this data model, the PCDB exports a simple set of operations: creation of documents, retrieval of documents and summary records, modification of tag sets, and retrieval of indexes. In our design, the PCDB is not responsible for query execution but rather for delivering indexes against which the query engine runs. The query engine itself is considered part of the Generic Application Logic.

PCDB operations run within a light-weight transaction system designed largely around ensuring atomicity of updates. Threads in the Application Logic start transactions by calling "beginTransaction" and finish by calling "commit." Writes (e.g., document creation, tag updates) must occur inside a transaction, reads need not. Transactions do not cross thread boundaries. There's no way to abort a transaction other than by crashing the program. Transactions enforce the following weak variation of the ACID properties:

- **Atomicity and durability.** Transactions are, of course, atomic in the traditional sense (all or nothing), and their effects are persistent after a successful commit.
- **Consistency.** The PCDB ensures consistency across its own data (i.e., among the documents, the summary records, the tag collections, and various indexes). In addition, at commit time, the PCDB calls a callback function that allows the application logic to maintain application-level consistency constraints. As already suggested, this callback cannot abort the current transaction; instead, it ensures consistency by modifying the data about to be committed to ensure it satisfies certain (simple) application invariants.
- **Isolation.** We provide a weak form of isolation. The writes to transactional data made by one thread are not seen by other threads until commit time. However, transactions are not serialized. For instance, if a thread T reads transactional data and another thread subsequently commits changes to that same data before T commits, then T can see the new updates in subsequent reads.

3.3 PCDB Architecture

Figure 2 contains a component diagram for our PCDB. The PCDB is decomposed into a Repository object that coordinates the activities of five Resource Managers: the Document Manager, Tag Manager, Summary Manager, Index Manager, and Address Manager.

- **Repository.** The Repository serves two roles. First, it's a "Façade" object a la the Gang-of-Four [3]: a simplified interface to the complicated interactions among the Resource Managers. Thus, for example, a single "newDoc" method in the Repository calls methods on all of the Resource Managers. Second, the Repository is a Transaction Monitor in the classic sense [6], coordinating the transactions across the five Resource Managers.
- **Document Manager.** The Document Manager is responsible for persistent storage of documents. The Document Manager stores document data in a textual format similar to mbox files; this provides an extra-level of assurance to more technical users that they can be recovered. The Document Manager has a "dup detection" feature which prevents multiple copies of the (exact) same document from being inserted more than once.

- **Tag Manager.** The Tag Manager is responsible for persistent storage of tag data. It is also responsible for delivering an inverted form of that data to the query engine. (In fact, the data is actually stored in inverted form, i.e., as a map from tag ids to lists of doc ids.)
- **Summary Manager.** The Summary Manager is responsible for persistent storage of summary records. Because summary records contain mutable tag data as well as immutable document data, the Summary Manager must support record updates.
- **Index Manager.** The Index Manager is responsible for managing the full-text index of the document data. It is the manager most responsible for the signature feature of the Bloomba product: fast, scalable search. We choose to build our own index manager for reasons of cost, functionality, and performance (see Section 5).
- **Address Manager**. The Address Manager supports a PCDB operation not mentioned in the previous subsection: a guesser for address autocomplete. It does this by managing an index of email addresses. Address autocomplete is used during message composition: as the user types a name or address into the "to" or "cc" fields of the message, Bloomba suggests completions based on what has been typed so far. Rather than list completions alphabetically, which does not scale and is also error prone, Bloomba ranks possible completions according to factors such as recency and frequency of use and displays the top-five ranking addresses in rank order. Bloomba performs this ranking over all addresses in all messages rather than just those addresses in the user's address book. The index managed by the Address Manager ensures that this ranking occurs quickly even for a corpus of tens of thousands of addresses.

The Document and Tag managers together manage data we consider *precious data*, data provided by the user that we cannot replace. The other managers store non-precious data, data which can be reconstructed from the Document and Tag managers. In the implementations of the Document and Tag managers, we have striven to emphasize transparency, in an effort to improve reliability, over performance. However, the Tag manager is in the critical path of some performance-sensitive operations, which sometimes stresses our commitment to simplicity.

A desktop-friendly footprint and behavior was an important requirement influencing the design of all these components. This requirement has not influenced our design – the components described above should be familiar to any database implementer – but rather has influenced the design one level deeper. This point is illustrated by the transaction-coordination performed by the Repository. The Repository utilizes the venerable two-phase commit protocol well known to the database community (see pseudocode in Figure 2). Nothing new in the abstract, but we've engineered the details to fulfill our desktop requirements, for example:

- **Hiding latency and deferred work.** Of course, we have pushed almost all disk activity into the "prepare" methods to minimize contention on the reader/writer lock. We go further by deferring disk-intensive work onto (persistent) work queues and perform it in the background. For example, when documents are created, they are parsed, but their tokens are not immediately inserted into the index. Instead, a background thread is created to periodically insert the tokens of multiple documents into the index as a batch.

  Even this much is familiar to the database community, but we go further still. The deferred work performed by the background thread is performed only when the machine is idle (because, as mentioned in Section 2, we are "borrowing resources"). Thus, we've engineered the Resource Managers so that (a) background work can be deferred for long periods without adversely impacting PCDB performance and (b) background tasks can be aborted before completion, which we do when we detect that the user has started using the computer again.
- **Customizing semantics.** As mentioned earlier, our transaction semantics allow at most one document-insertion transaction to run at a time. This limitation allows us to easily stream those documents directly to disk without requiring substantial disk activity in the case of an abort. This reduces buffering requirements and disk activity, leading to a more desktop-friendly footprint.
- **Accommodating hostility.** Desktop file systems are hostile environments: between virus scanners, backup programs, garbage collectors, and rouge user behavior, files can be locked, modified and even deleted in unexpected ways. All of our Resource Managers have evolved to be highly robust to these possibilities. For all file operations other than reading and writing from already-opened files, we assume that failure is common rather than a rare exception. We've designed the Resource Managers to be robust under this assumption. An example of this design-principle in action is the commit record: we've both provided several layers of redundancy for the

commit record itself and have actually designed our restart sequence to work in the absence of a commit record.

- **Interactivity.** Users of desktop applications expect programs to be "responsive," which means, among other things, that the user can perform UI actions even when the system is busy with the synchronous parts of updating the database the user can see evidence of progress (e.g., a "spinning disk") for even relatively short (1/2 s) operations, can cancel long-running (>2 s) operations, and receive reasonable error messages when an operation fails (e.g., due to lack of disk space). These requirements have forced us to design a two-way communication channel that connects all layers of our system – Presentation, Application Logic, and Data Interaction. For example, as the data interaction layer is downloading a large message, it sends byte-level progress information back up to the presentation; or, when the user issues a cancellation, a signal must find its way down into the Resource Managers. Achieving interactivity has not only driven many of our detailed design decisions, it has also prevented us from using many off-the-shelf libraries (e.g., parsers) that are designed around a batch versus interactive model.

4. Query Runner

To provide a deeper look into how the requirements of the desktop environment have influenced our design, we present a deeper examination of one part of our system, *Query Runners.*

Query Runners are responsible for returning query results to the user interface. In addition to the usual constraints implied by our desktop environment (e.g., small footprint, few cycles, etc.), the design of our user-interface implied further requirements for our Query Runners:

- **Continuous.** Traditionally, queries are discrete: you start them, a result set is computed and returned, and the query terminates. In Bloomba, queries are used to populate aspects of the user-interface that need to be updated based on changes to the database. For example, when you select the "Inbox" for display in the message list, the message list is populated by a Query Runner executing the query "folder:Inbox". As the user moves messages out of the Inbox and/or new messages arrive in the background, the contents of this message list – and thus the results of this query – need to be updated automatically.
- **Counting.** Bloomba allows users to save an arbitrary number of searches. These "saved searches" are given names and are listed in a convenient location on the left-hand side of the UI (where folders are traditionally displayed). Bloomba displays a count of the messages that match each of these saved searches. These counts are tallied and provided to the UI via a special kind of Query Runner called a *Query Count Runner*.
- **Scalable concurrency.** The Bloomba user-interface runs a large number of these Query Runners in parallel: the main message list is populated by one; a large number are started for displaying message counts; and a few less-obvious ones are run for other purposes (e.g., providing fast access to one's calendar data from within the mail UI). Thus, Query Runners have to be light-weight and run in parallel.

In IR terminology, we have more of a filter engine than a query engine. That is, we're running a stream of new and changing documents against a relatively large set of fixed queries. As we discover that a new or changed document does or does not match one of these fixed queries, we need to update the result-set or count of that query. In the case of a changed document (or, more specifically, a document whose tag-set has changed), this implies knowing whether the old version of the document was or was not part of the old result set. We call this the *delta problem*.

One way to solve the delta problem is to memoize, that is, save in RAM the current result set for each query. However, the size of our result-sets can number in the thousands; multiplied by many outstanding queries, this approach does not scale. An alternative solution is to provide an "old version" and "new version" of the document to our query executive, which can then run the query twice and decide if a change is an "add," "remove," or "update." While this "pair approach" is more complicated than memoization, it scales better. Also, because documents themselves don't change but rather only their tag-sets change, the pair approach is easier to implement than it might first appear.

Given this background, let's look at the design of the Query Runner in a bit more detail. Figure 3 contains an object-diagram of the objects that cooperate to run a query. In this diagram, changes are made by the mutator thread on the right and are communicated to the query-runner thread on the left. The communication channel between these two threads is the "IndexWatcher" object. An IndexWatcher is really a producer-consumer queue of IndexPair objects (the mutator producing, the query-runner consuming).

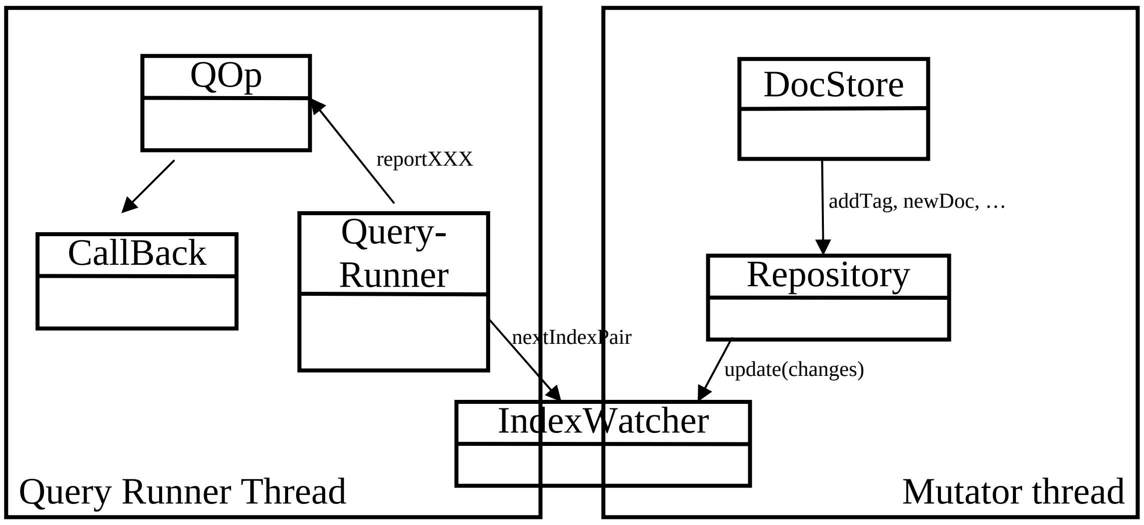


Figure 3. Object diagram for query execution

When the mutator thread commits a transaction, it places into this queue an IndexPair for the transaction. This IndexPair is a pair of Index objects, and "old state" Index and a "new state" Index. An Index object is a traditional inverted file: Given a term, the Index object returns a posting list of documents containing that term. These Index object contain posting lists for both the immutable documents and the mutable tag sets. The old and new Index objects share posting lists for the immutable terms, but have different lists for the mutable tags. The Indexes contained by an IndexPair contain postings for the same universe of documents. This universe is guaranteed to include all new and changed documents (it may include more).

The Query Runner sits in a loop asking the IndexWatcher to return the next pair in the queue. When the queue is empty, this loop blocks waiting for an update. When a new pair is returned, the query is compiled against both the old and new Indexes. From these two results sets, a delta is computed, and results are pushed up to the User Interface via the Callback object (in the lower-left of Figure 3). (As mentioned earlier, the PCDB implements just the IndexWatcher, IndexPair, and Index objects. Query compilation and execution are considered part of the application logic.)

5. The PCDB and the database community

We have been asked to comment on what we have used from the database community, what we have not used, and what we'd like to use (i.e., what database research might be of use to us).

When people from the database community learn of Bloomba they often ask, "Did you build it on a SQL database." The answer is no. When we started, we considered relational databases, XML databases, and also existing full-text indexers. In the end, we decided to build the PCDB from scratch.

As an Independent Software Vendor selling a product priced under $100, our ability to re-use existing software is severely limited by cost considerations. In particular, any commercial database would be cost-prohibitive. Still, there are a number of open-source SQL or XML databases and full-text indexing systems we could have embedded into Bloomba. When we looked at systems such as these, we found they were not suitable for personal content and/or for a desktop environment. For example:

- Early on we realized that the utility of personal data is strongly related to its freshness (i.e., people are much more likely to look for new email messages than old ones) (c.f., [2]). This observation is built into almost all of our Resource Managers: for example, the document and summary managers automatically "age" documents, and posting lists return new documents before old ones. The open-source systems we looked at did not have this same chronological bias.
- As mentioned in previous sections, for long-running operations, desktop programs need to provide feedback on progress; otherwise, users believe the system has hung. Our PCDB was built to provide such information, the open-source systems were not.
- Our PCDB uses a number of techniques to minimize its impact on other activities the user might be performing on the machine (such as editing its document). This includes minimizing the use of RAM by spilling to temporary files, and aborting intensive maintenance operations when

we detect user activities. Again, the open-source systems were not written with such sensitivities in mind.

- As mentioned earlier, desktops, and Windows desktops in particular, are quite hostile environments. We were concerned that these open-source systems would not be reliable in the face of virus scanners, garbage collectors, and so forth.

While we built the PCDB from scratch, we have used a many concepts and technologies developed in the database and information retrieval communities. Our bibles have been [6], [4], and [9]. Our challenge has been more to implement existing technologies consistent with the limits and expectations of desktop applications rather than to start from a blank slate.

Looking forward, one part of the database literature we are eager to dig into is the data mining literature. We believe there are many algorithms and techniques in that literature to be effectively leveraged on personal content. As one example of this, consider contact information. In addition to the user's own collection of contact cards, Bloomba has access to email addresses and associated "Display Names" in the headers of emails, plus a plethora of contact information in message bodies themselves. Further, databases of contacts are starting to emerge on the Web [7]. While a lot of data is available, it contains lots of duplicates, contradictions, and holes. Data fusion and cleaning techniques from the world of data mining might turn this raw data into more useful information.

There are a number of areas where we wished we had more support from the database community:

- **Data models.** We do not believe that the relational model is the best model for personal content. First, while personal content is definitely typed – emails, events, and contacts are all distinct types – these data in a PCDB doesn't want to be segregated by type. If personal content wants to be grouped at all, it's into heterogeneous, dynamic groups like "Personal" or "FallRelease." Also, personal content tends to be denormalized, e.g., a contact card contains multiple phone numbers. For these and other reasons, we don't believe the relational model is a good match to personal content.

  To those who disagree and believe that the relational model is appropriate for personal content, a proof point would be nice. For those who agree, this begs the evergreen issue of the database community: If not relational, then what? We leave this question as an exercise to the reader.

- **Incrementality.** Personal Content Databases receive a steady stream of incoming data that the user often expects to be indexed as soon as it arrives. One area where it felt like we were inventing more than we wanted to is in the area of incremental indexing. In this context, "indexing" is not just the full-text indexing but other indexes in our system, e.g., the Address Manager mentioned above, or the index on our lexicon (which we haven't discussed in this paper). Many promising indexing techniques we found in the literature assumed a batch context (or didn't address index-construction at all). Also, the desktop context introduces the requirements to defer intensive work until the machine is idle, and also to minimize the amount of RAM used; incremental indexing techniques in we did find failed to consider these additional requirements.

- **Availability.** It's obvious that a Personal Content Database needs to be highly reliable in that it does not lose data. The database literature is full of techniques for increasing reliability, techniques we've been able to leverage. However, a Personal Content Database also needs to be highly *available*. When a user gets on the phone with an important customer, for example, it is disastrous if personal content crucial to the call suddenly becomes unavailable.

  The standard database approach to availability is replication, typically across multiple machines, which is not applicable in a desktop context. The database literature also makes another, less obvious assumption regarding availability: it's either all or nothing. Our progress on the problem of perceived availability accelerated greatly when we realized that partial availability has high utility for users. For example, in the "important customer on the phone" scenario, let's say the index has become corrupted (e.g., because the user inadvertently deleted a file), requiring a 30-60 minute "rebuild." In this context, if you can let the user access the Inbox (i.e., receive messages into it, read message in it, and reply to message in it), then there's a good chance the user can have the phone call even if fast searching is not available.

  Thus, architectures and algorithms that support partial availability without the user of stand-by machines would be of great value.

- **Specialized indexes.** As we seek to add even more intelligence into Bloomba, we see our selves building more specialized indexes such as the Address Manager. This feels a bit unsatisfying. First, there's a layering problem: the functionality supported by these indexes is fairly specific to Bloomba and thus best fits in the Bloomba Application Logic, yet they also need to be at the

Data Interaction layer to participate in the transaction mechanism. Second, we're concerned that, if too many of these specialized indexes are added, commit latency could suffer – which has significant impact on the user's perception of system performance. Thus, another potential area of research is an extensible, scalable, high-performance system for specialized, applogic indexes.

- **Replication.** We believe that disconnected replication is a crucially important feature for personal content databases. However, in the database literature, replication is most often considered in the context of entire-database replication or strict-subset replication.

  We believe personal content databases need document-level replication as well. To illustrate, imagine a husband and wife with a large, family photo collection. The husband and wife each want their own databases, but they want those databases to share the family photos. In fact, they may want to share those photos (or a subset) with the extended family. As one person updates the metadata on the photos, the others probably want to see those updates. What's needed here is not sharing of entire databases, therefore, but flexible sharing of documents within those repositories.

6. Summary

We believe continued growth in the volume of personal content, together with a shift to multi-device personal computing environment, will inevitably lead to the development of Personal Content Databases (PCDBs). These databases will make it easier to find, use, and replicate a large, heterogeneous repository of personal content.

The database literature already contains many useful concepts and technologies for building PCDBs. However, the requirements of PCDBs imply engineering details that differ from today's typical database systems. These requirements include the need to ensure that the majority of the machine's resources are dedicated to the user's foreground task, the need for interactivity with the user, and the need for robustness within a hostile environment. PCDBs also introduce opportunities for new areas of database research, including research into new data models, incremental indexing, partial availability, specialized indexes, and extended models of replication.

Acknowledgements

Bloomba and its PCDB owe their existence to the entire teams at Stata Labs and Integrated SoftTech Solutions.

References

[1] Bloomba: http://www.statalabs.com

[2] Dumais, Cutrell, Cadiz, Jancke, Sarin, Robbins. Stuff I've Seen: A system for personal information retrieval and re-use. *Proceedings of SIGIR 2003*. Association for Computing Machinery, 2003.

[3] Gamma, Helm, Johnson, Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Garcia-Molina, Ullman, Widom. *Database System Implementation.* Prentice Hall, 1999.

[5] Gmail: http://gmail.google.com

[6] Gray, Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[7] Plaxo: http://www.plaxo/com

[9] Witten, Moffat, Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Ed. Morgan Kaufmann, 1999.