BRIDGES BETWEEN SILOS:
A Microsoft Research Project

Gina Venolia
User Interface Architect
Microsoft Research
Software Improvement Group

January 2005

SUMMARY

Enterprise data is locked away in silos. As a result people spend too much time looking for information – or they spend too little and make decisions based on incomplete information. The simplest remedy is to pull all the silos into a common full-text search index, but doing so realizes only part of the opportunity. The rest lies in automatically finding and representing the relationships among the objects in the index. Relationships take many forms, ranging from schematized references to allusions in text. All these relationships can be recorded in the index, thus combining structured, semi-structured, and unstructured relationships together in a normalized representation, forming *bridges between data silos*. The result is a graph, which makes it possible to make queries like "find all *x*'s related to *y*." For example the graph could be used to find all the people or discussions related to a particular method. The graph is stored in a SQL-based search index that also has a rich notion of time and history. The index is exposed to the user in three ways: a search portal, an implicit query sidebar, and object-based search commands in client applications. The index improves search in several ways, allowing for richer filtering, scoring, and search results presentation. These ideas can turn siloed data into working knowledge, making the enterprise work more efficiently.

DATA SILOS

Enterprise data is locked away in silos, making it much less useful than it could be. Take the information around software development process as an example: The code and its history are in the source code control system; bugs, work items, and test cases are in the bug database; specs are in Microsoft Word documents in on file servers, communication is in individuals' email stores, and so on. Because of siloing the full value of the data is not realized, and so knowledge-based processes are not as efficient as it could be. The same is true of any knowledge-based endeavor.

Siloing makes accessing data harder in several ways. Searching across multiple silos means invoking each one's client, using its unique user interface, and receiving the results in separate buckets. Some silos don't support full-text search at all! Browsing suffers from the same problems as searching, made worse because each silo has its own organizational structure. Some silos have no way to link to a particular item, and so hinder the free flow of information. And because each silo has its own API, it's difficult to develop analysis tools that draw data from multiple silos. Siloing impedes the discovery, synthesis, and flow of knowledge. As a result people spend too much time looking for information – or they spend too little and make decisions based on incomplete information.

Email is the most recalcitrant silo. Email archives are locked down per user, rather than being shared repositories. And yet email is crucially important: It is where discussions happen and often the only place that important decisions are recorded. It's been said that *email is where knowledge goes to die*[1]. Any attempt to break down the barriers between silos has to address the problems of email.

There have been a few approaches developed to address the problems of siloed data. A classic is to try to build the one silo that will meet everyone's needs. With *federated search* a user's query is spawned to each silo, and the results are then integrated. Enterprise search systems such as Google Search Appliance and Microsoft SharePoint Portal Server Search pull information from a variety of sources into a single full-text search engine. These approaches realize only part of the opportunity. The rest lies in automatically finding and representing the relationships among the objects in the index, as I'll describe in the remainder of this paper. These relationships form *bridges between data silos*.

BRIDGES BETWEEN SILOS

The relationships that each data source contributes take a variety of forms:

[1] Attributed to Bill French in his essay, *Transforming Information into Knowledge at the Portal*.

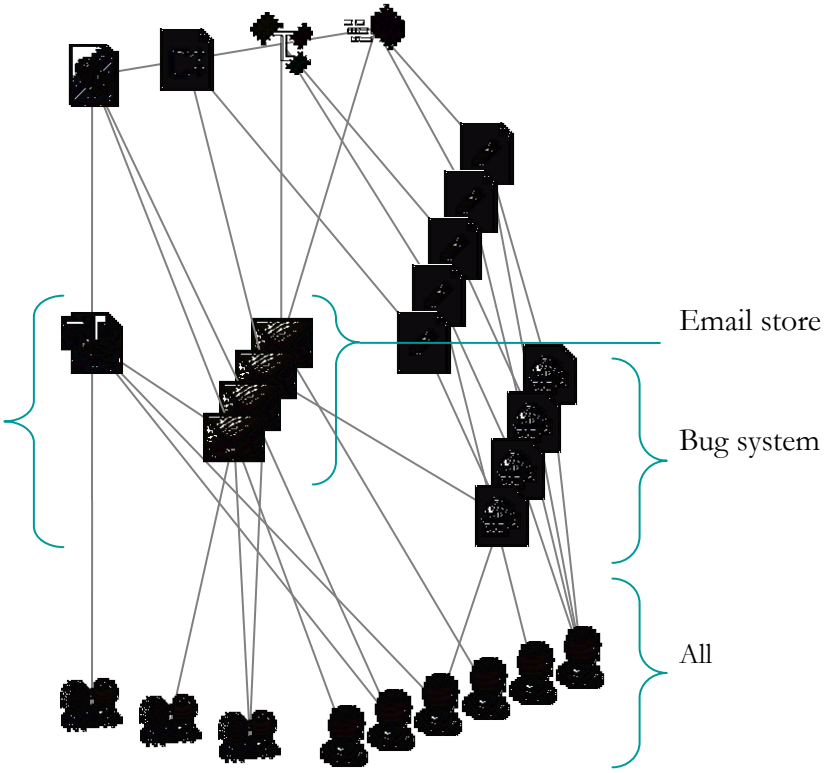

Figure 1: Some of the items and relationships in an index over four data sources. The source code control system contributes change-lists, source files, classes, methods, DLL's, people, and relationships among them. The email store contributes emails, people, and relationships. The bug database contributes bugs, people, and relationships. File servers contribute sites, documents, people, and relationships. Relationship type and direction are not depicted.

- *Explicit references:* For example, a check-in to a source code control system explicitly declares the files it alters.
- *Implicit-but-exact references:* For example with a little processing on the checked-in files it is possible to determine which classes and methods were altered by a change-list.
- *Ambiguous relationships:* Some relationships are implicit and fuzzy. For example, the correspondence between check-ins and bug actions could be detected by looking for the pattern of a particular person making a change-list and resolving a bug at about the same time.
- *Textual references:* Relationships may be buried in human-readable text. Some of these are explicit, e.g. a change-list comment might contain a URL or path to a file server. These can be detected using simple regular expressions.
- *Textual allusions:* Human-readable text may also contain less-explicit relationships. For example a change-list comment might mention "maryw", "Robert", "bug 345", etc. Some of these allusions could be detected using a combination of dictionary-based approaches, regular expressions, and natural language processing techniques[2]. Ambiguous references could be resolved using the index itself. For example there may be multiple bug databases that include

[2] *Named entity detection* and *factoid detection* are two relevant techniques.

a bug numbered 345. Searching the index for the database most strongly associated with the person who made the mention can help to narrow down the field. Time-based heuristics could further refine the set.

All these relationships are recorded in the index, thus combining structured, semi-structured, and unstructured relationships together in a normalized representation. Though some of the computations are costly, they need happen only once. Ambiguity is handled by recording a confidence rating with each relationship.

The result is a graph, as shown by the example in Figure 1. With this graph it becomes possible to make queries like "find all *x*'s related to *y*." For example the graph could be used to find all the people related to a particular method, even though it might take multiple "hops" in the graph to find them: Some people are related because they edited the method, others might have mentioned the method by name in email, and still others might have edited bugs that either related to the change-lists or mentioned the method in the bug comments. Because there are multiple, redundant paths it's possible to measure the strength of the association between the method and each related person.

USER INTERFACE

The graph that represents the bridged silos is only as interesting as the user interface that exposes it. There are three distinct manifestations: a search portal, an implicit query sidebar, and object-based search commands in client applications[3]. Like all search portals, it is a web site that allows the user to create a query, execute it, and see the results. Like some more modern search portals it allows the user to save important queries and view a stream of matching items as they appear in the index[4].

The implicit query sidebar proactively shows the items related to the one currently being viewed in the active application. It watches the active applications (using plug-ins or other means) to determine the object in focus, executes a "find all *x*'s related to *y*" query in the background each time it changes, and displays the results in the sidebar. Presenting the results in a meaningful way

[3] There may be another user interface on the index: a browser over the graph. At this point I don't have scenarios to drive the motivation and design of a browser.

[4] Some blog and news search engines provide an RSS feed of search results. For example, these are links to the feeds for the search "microsoft google" on Yahoo! News and Feedster. MSN Search is experimenting with this feature. I am not convinced that an RSS is sufficient to deliver search subscription results, however, as it ignores the fact that the value of a result changes over time and does not provide means to divide-and-conquer the search results.

is a nontrivial UI challenge[5]. A button in the sidebar launches the search portal with the same query, allowing the user to interactively explore the results and refine the query.

Finally a simple command is added to applications to invoke the search portal with a query on the object in focus. This is especially useful for those who don't want to spend resources on the implicit query sidebar.

SCENARIOS

Let's consider some scenarios where bridges between silos can make a big difference.

> *A developer debugs his way into unfamiliar code. He can understand what the code is doing but not why it's doing it. He needs more information than is present in the source file to figure it out. He needs to find the relevant material to read or the right people to talk to.*

Today the developer could try to find the relevant material by searching the intranet or his email store for relevant keywords, but he probably won't because it's difficult to find the right terms to search on. He might access some team-specific database showing source code ownership, hoping that it's current and relevant. More likely, he asks someone who he thinks will get him closer to finding the right person to talk with, an increasingly unworkable solution given the globalization of software development.

With the proposed system he may have turned on the implicit query sidebar, in which case he glances at it to see the documents, emails, bugs, change-lists, people, etc. that are most relevant to the current method. If he needs to dig deeper he clicks on a button that invokes the complete search results UI in the search portal. If he doesn't have the sidebar visible then he kicks off the search with a command in the context menu.

> *A project manager is has taken a dependency on a DLL being delivered by another team, but she is somewhat wary about their ability to deliver as promised. Her first line of defense is to initiate communication with the appropriate people. She also wants to keep an eye on the relevant electronic communication but she's very busy so won't have time to scrutinize every email, document, change-list, and bug.*

Today the project manager subscribes to the relevant discussion lists, hopes that the important conversations don't happen off-list, and hopes that they catch her attention in her inbox. She also searches for relevant terms on the intranet and then subscribes by email to the search results. The

[5] The challenge lies in explaining the relationship between the object in focus and each result. Tony Tang explored these issues during his summer internship with me in a project called Stuff I've Seen for Visual Studio, or SIS4VS.

net cast by these two approaches is both too wide, potentially getting too many irrelevant results, and too narrow, missing discussions that happen in other discussion lists, bug comments, etc.

With the proposed system she searches for the name of the DLL, searches for items related to it, and then adds the search to her subscriptions. Later she returns to the search portal to review the most-relevant new items as her time allows. The user interface for viewing search subscription results allows her divide-and-conquer the items to focus on specific aspects or to group them into contexts for faster perusal.

> *A dev lead in Hyderabad needs to get a deeper understanding of the reasoning behind an architectural decision that was made before the project was transferred from the US. He has found a spec that talks about the final decision but does not explain the thought processes behind it. He needs to find the relevant materials or the right people to talk with.*

Today the dev lead browses the team's file servers, searches through the public folder archives of the relevant discussion lists, and searches the intranet. He tries to make sense of the information he's found but it's difficult because there's no way to look at it on a timeline. He finds three copies of a spec and has no way to determine their order. His detective-work is further complicated because he can't tell the organizational structure or discussion list memberships as they existed at the time of the decision he's researching.

With the proposed system he uses the spec as an entrypoint for search. He views the results chronologically and reads the story of the problem and its solution as it unfolds. He sees the organization of the key players as it evolves over the decision-making process.

—

An index that builds bridges between data silos is useful in many other scenarios: understanding the context for a bug during a triage meeting, when a developer is handed a work item in unfamiliar code, etc. The themes that emerge from these scenarios are:

- Finding objects related to the one being viewed
- Reliance on high-quality scoring to make the most relevant objects salient
- Search subscriptions
- Time and history as important organizing principles

BUILDING THE INDEX

This section describes the system in greater detail. The core is a search index, which would likely be implemented as a SQL Server database exposed through a web service API. It would likely use SQL Full-Text Search to provide the basic searching capability. The index contains *items* and

links representing the objects and relationships in the silos. An index is provisioned by *crawlers*, which keep the index in sync with its data sources.

An item in the index represents a user-meaningful object pulled from a data sources by a crawler. The same item may be submitted by multiple crawlers, forming the footings of a bridge between those silos. Each item contains properties to store its name and the text submitted for it to the full-text search index. Each item also contains properties that records when the object first became available and when it went off-line, allowing the index to represent time and history. A type system allows the crawler to specify additional properties to be stored with the items. If a crawler detects the deletion of an object it may either delete the item from the index or, better yet, set its end date property.

The index is a network-based, shared resource. It might be deployed at a team level or even intranet-wide. Each item is governed by its own access restrictions, thus public, group, and private information intermingle in the index.

A link in the index represents a directed relationship between a pair of items. Like items, links have a type system and start- and end-dates. A link is accessible to a given user if both its endpoint items are accessible. Links have a confidence rating that allows a crawler to record the ambiguity of a reference[6].

A crawler talks to a data source using its native API and creates the appropriate items and links for objects and relationships that have been created, modified, and deleted in the data source.

Crawlers scan any human-readable text for textual references and allusions, creating an item and a link for each one detected. File contents, e.g. Word documents, PowerPoint presentations, and C++ files, may contain both structured and textual references. The structured references are handled by opening the file and scanning its document object model. Textual references and allusions within a file may be found from the document object model directly, or by running it through MSSearch's *IFilter* interface and then scanning the resulting plain text.

For example the source code control system crawler queries the database for the recent change-lists. For each change-list it creates a *ChangeList* item and submits the comments to the full-text search index. It also scans the change-list comment for any textual references and allusions, creating items and *Mentions* links to match. It creates a *DomainAccount* item for the person who submitted the change-list and connects it with the *ChangeList* using an *Author* link. For each file revision it creates a *Revision* item, a *File* item, a collection of *Folder* items, *Contains* links

[6] There is a third component to the data model, *attributes*. An attribute adds a collection of properties to an item, providing an additional mechanism for extending the data model. It is intended for use by crawlers that contribute additional information about existing items, e.g. test coverage. Like items and links, attributes have a type system and start- and end-dates.

connecting them, and another *Contains* link connecting the *ChangeList* and the *Revision*. It fetches the file itself and creates the appropriate subclass of *Contents* item by the file extension and then does any file-specific processing. For example, for C++ and C# files it parses the code and then walks the code document object model, creating *Class* items, *Method* items, and *Contains* links as needed. It also scans any human-readable text in the file (comments, strings, etc.) for textual references and allusions, creating items and *Mentions* links to match.

As I mentioned earlier, crawling email deserves special attention. There are two complementary approaches. In the first a crawler indexes email sent to a special alias. To archive a discussion list, one simply adds the alias as a member[7]. In the second approach a crawler runs on the user's machine to archive the user's email store. A message archived this way is access-restricted to people it is addressed to. These two approaches are complementary and together result in the greatest amount of email being accessible to the largest appropriate audience[8].

SEARCH

The primary means for accessing data in the index is search, which is exposed as part of the web service API on the index. Search is implemented by turning the query specification into a SQL query, executing it, and then returning the resulting table. There are three main clients for search: the search portal, the implicit query sidebar, and crawlers that need to resolve ambiguous references.

At an abstract level, searching this index is like any search system: the index takes a query specification and returns a set of results. The query specification includes filters that restrict the search results and parameters that control the search algorithm. The differences between this and other search engines follow from the relationships and history represented in the index.

The graph of relationships, represented in the index as links between items, provides for richer filtering, scoring, and search results presentation. In addition to the usual filtering provided by full-text search and metadata search, it is possible to filter on patterns in the graph, e.g. "find all the folders that contain C++ files". Link-based static scoring may be more robust than its web cousin because the links come from a richer set of relationships than just hyperlinks. The graph connectivity to a set of anchor items can be used as another scoring term. One way the search portal or implicit query sidebar can use this is to personalize the search results by scoring based

[7] There may actually be two aliases: one that restricts access to the items to only list members and one that makes the items public. It is possible to manually archive an email by forwarding it to either alias.

[8] There is a fundamental tension between retaining email as part of the institutional memory of an organization and the desire to purge it to reduce legal exposure. The proposed system can help somewhat by reducing the individual's motivation for keeping a private email archive.

on connectivity to the user. Finally, the search results can be augmented with items that don't match the query's filter but are strongly related to it in the graph. This feature is pivotal to implicit query, which is a simple search for a few specific items, augmented with related items.

History, represented in the index as start- and end-dates for items and links, also provides for richer scoring and search results presentation. It becomes possible to measure the importance of an item over time by watching the temporal pattern of links being made to it. The value of the item at the present time (or some time in the past) can be used as another factor in search results scoring. With a strong notion of history it becomes possible to present search results in chronological order, helping the user to understand the items in context.

RESEARCH QUESTIONS

There are many questions about this proposal that can't be answered without building it and studying a deployment. There are several questions about the infrastructure of the system:

- For the system to be truly extensible the schemas for each crawler must be independent. *Can each crawler's schema be designed in isolation from the others?*
- Portions of the graph will be exposed to the user directly. It is important that the items and links be meaningful to the user. *Is there a minimum of "connective tissue" in the crawlers' schemas?*
- Textual allusions are a crucial source of relationships in the graph. *Is it possible to detect almost all of the textual allusions present in the source material? Will few spurious matches be detected? Is it possible in practice to resolve ambiguous allusions? Is a notion of link confidence sufficient for representing the remaining ambiguity? Can each detector be designed in isolation from the others?*
- The notion that items lose their importance with time – the concept of "yesterday's news" – has intuitive appeal. However the "staying power" of items varies greatly. *Is it possible to accurately model and predict the importance of an item over time based on the evidence in the index?*
- The graph may prove useful for scoring search results and finding related items. *Is it possible to establish meaningful link weights for computing connectivity in the graph? Is there an algorithm that effectively computes graph connectivity in a meaningful way? Can that algorithm be implemented efficiently in SQL?*

The user interface presents several opportunities for innovation, and with those several challenges:

- The richness of the index provides an opportunity for displaying time and relationships in search results. *Is it possible to simultaneously depict search score and temporal sequence in the results display? Can the results display be improved by rendering links among the items?*
- The implicit query sidebar must be able to effectively explain why items are in the results. *Is it possible to depict the relationship between the focused object and the items in the results?*
- The UI for iterative query refinement is easy when the user is simply editing search keywords – a text box and "go" button suffice. For search that includes structured terms it's not so

simple. *Can a simple, easy-to-use interface allow for combined keyword and faceted metadata search?*

- I think that search subscriptions will become an increasingly important part of the user's search toolbox. *Is it possible to integrate subscriptions into the search UI? Can the UI for search subscriptions help the user to stay on top of the incoming results without overwhelming them?*

The goal of the system is to turn siloed data into working knowledge. However achieving that goal is not a given:

- The system can achieve its goal only if it is used! *If we build it, will they come? What are the forces that encourage and inhibit individual adoption?*
- Just because it's used doesn't mean people are better at achieving their individual goals. *Are people who use the system more effective at their jobs because of it? How can this be measured? What are the distraction costs?*
- The system might be responsible for increased contact between people because it makes the right people easier to find. On the other hand it may decrease interpersonal contact because users can "skip the middleman" and get the data directly. *Does the system foster or hinder human-to-human communication?*
- The system could be seen as increasing surveillance and decreasing privacy. *Does the system engender a sense of loss-of-privacy? What can be done to ameliorate that perception or fact?*