Buglook: A Search Engine for Bug Reports

Georgi Chulkov

21 August 2008

Abstract

I present Buglook, s search engine designed to find bug reports within bug tracking systems. By limiting the search domain it supports more expressive queries than a general web page search engine can. It uses a simple but powerful query language to phrase queries. Buglook acquires its semistructured data from presentational interfaces meant for human beings rather than machines, and thus works on currently-existing widely deployed bug tracking systems. It uses latent semantic indexing combined with feature weights and filters to provide highly relevant search results with minimal false positives. Buglook has a reliable and flexible modular architecture and can scale to face many simultaneous users and highly dynamic bug repositories. Buglook's evaluation shows it to return better search results than a web search engine and an integrated bug tracker search engine, in a comparable amount of time.

Master Thesis
Supervisor: Prof. Jürgen Schöenwälder
Jacobs University Bremen

Contents

1 Introduction

Modern search engines make it very easy to find any web page that contains some given text in a few seconds. They continuously crawl and index the World Wide Web (WWW) and allow users to only worry about what to search for, not where. Despite the massive amount of web page content available worldwide, search engines typically take less than a second to return a list of results to their users [36]. The list is sorted by relevance in some way specific to the search engine [37].

To achieve such high performance, generic search engines restrict themselves to pure textual search. They view a web page as little more than a list of words, and make little effort to infer any structure or other semantic information from it [36]. There are two reasons for this. First, any processing beyond text indexing would be too expensive given the massive amounts of data involved. Second, search engines must deal with any web page on any website of any application domain. Generally, two pages from the set of web pages on the WWW have nothing in common beyond the use of HTML as their markup language.

If one restricts searching to a given set of websites within a well-known application domain, ignoring structural similarity between web pages is no longer an optimal (or the only) solution. A search engine that can derive structure from the similarity within such web pages could allow more expressive search queries, and could better define the relevance of web-pages in order to sort search results. Classic pure text search, on the other hand, is constrained to word similarity and proximity as its relevance criterion [37].

1.1 An Application Domain: Bug Tracking Systems

In my thesis I address the problem of searching within bug tracking system (BTS) websites also known as bug report systems, bug repositories, or bug trackers[1]. I present a search engine that specializes in finding bug reports inside such repositories.

A *bug report* is a description of a problem with some software or of some requested functionality to be added to the software. A typical bug report consists of two parts - a set of features that describe metadata such as when the report was opened, what its state is, and what component of some software it pertains to, and a free-form textual description of the problem itself. The latter usually also includes a list of comments by other users and/or software developers. Note that the term *bug* is also used to denote a bug report.

A *bug tracking system (BTS)* is a website used for submission and processing of bug reports. It consists of a database that contains all submitted reports in some fixed data structure, and underlying software that uses HTML templates to render a bug report into a full web page presentable to the user. The underlying software is also called a BTS; the distinction will be made explicit when the use of the term is not clear from its context[2]. Most BTSs do not allow direct access to their bug reports, but only to the bugs' presentational rendering in HTML. For a detailed survey of BTSs and what interfaces they allow, refer to section 3. The distinction between a BTS and other related

[1] I use these terms interchangeably throughout this document.

[2] The term *BTS site* refers to a specific installation that uses some BTS as its underlying software.

systems (trouble ticket systems, issue tracking systems, incident tracking systems, and defect tracking systems) is explained in [54].

My choice of application domain is motivated by the importance of search within bug repositories. In a classical support scenario, a user first reports a bug. A developer confirms it and then creates a fix, makes it available to users, and finally integrates it into the next version of the software. False alarms are usually handled by the developer by explaining why a reported bug is not in fact a problem with the software, but rather a mistake on the user's end. Consequently, bug reports found in bug trackers often contain the solution to the specific issues reported in them. When they do not, they typically at least confirm whether a given problem truly exists, whether the developer knows about it, and whether anyone is working on a fix. Over time bug trackers accumulate large amounts of such information, which can be very useful to other people that experience the same or similart problems.

Locating this information is often difficult with a typical search engine, largely because of the numerous irrelevant results that are likely to be returned for a given set of keywords. As an example, suppose a user is looking for bugs describing some malfunction, such that they are confirmed to be errors and are being worked on. If, beside the keywords describing the problem, a user also specifies keywords such as "confirmed" or "fixed", the user will likely get many bug reports that are confirmed and fixed, but do not pertain to their problem. If the user does not specify these additional keywords, they are likely to get many results that are not bug reports at all.

A search engine that is designed with bug reports in mind would be able to derive structure common to all reports, such as the state of the bug ("confirmed", "fixed", etc.),the date of reporting, the date of last activity, and many more properties that a general search engine has no concept of. This information can in turn be used within search queries to provide highly relevant results to a user, as well as to allow a user to filter results by certain criteria.

Given that bug repository systems implement their own search capabilities which are aware of the inherent properties of their content, why is a domain-specific search engine needed? The answer is twofold. First, a search engine can dive into many bug trackers simultaneously and thus save the user a lot of manual effort. Second, by having a view of the combined data of several trackers, a search engine can discover and utilize relationships and similarities between bug reports that exist in different bug trackers.

1.2 Motivating Example

Consider the following example that demonstrates the difficulty of using generic search engines in a software debugging task. Some time ago, the latest version of wine, the Windows compatibility layer for the Linux operating system, failed to compile. The error message indicated that another piece of software, valgrind, was causing the problem. A search on a popular search engine for the query "wine 0.9.51 does not compile because of valgrind" returned 110 results. Only a single result was somewhat related, describing a completely different (and older) issue related to valgrind and wine[3]. In particular, none of the 110 results included `http://bugs.gentoo.org/show_bug.cgi?id=202542`, a publically available bug report on the Gentoo Bugzilla bug tracker, which describes

[3]I was happy to see almost no results about the alcoholic beverage of the same name.

exactly that problem, and lists a temporary solution recommended until the problem is solved by the wine developers. Another popular search engine was tested to a similar effect. Disappointingly, even searching using Gentoo Bugzilla's own integrated search for the same query returned no results whatsoever. Clearly, the current state of search within bug trackers needs to be improved.

1.3 Searching in Bug Trackers vs. the Web

Web pages within a bug tracker differ significantly from general web pages on the web. A bug tracker is invariably backed by a database or some other data storage system, and uses a well-defined template to transform structured data into a HTML page. In contrast, a general web page can be either similarly generated from external data, manually written by a human being, or both. This distinction has several important consequences.

Using a single template guarantees the consistency of the HTML structure of all pages within a single bug tracker. The generated web pages differ only in their textual content or in the number of well-defined elements they have (e.g. the number of replies to a bug report), but their overall structure is identical. A given datum, such as the title of the bug report or the date of the last update, can be extracted in an identical way from every bug report. Structural similarity in turn makes it possible to parse the pages and extract their data with considerable accuracy, without resorting to complex natural-language processing tools.

On the other hand, there is no guarantee that a bug tracker will always generate the exact same page twice from the same input data. Subtle differences such as the bug tracker version, the current date, or the time it took to generate a page, imply that it is difficult to know if a page has been modified without downloading the page and extracting its modification time if it is contained in it. In addition, it is not possible to simply index "all pages", because their number is infinite, even though the source data they were generated from is finite.

1.4 Buglook

I present my approach to the problem of efficient and expressive search in bug tracking systems: a search engine that is able to collect data from many existing deployed bug trackers, search the collected data efficiently in response to expressive user queries, and return results with a high degree of relevance and a low percentage of false positives. More precisely, my contribution is a system that:

- Systematically obtains data from a large number of popular existing bug trackers, and transforms it into a unified data model for bug reports.
- Automatically monitors any bug trackers whose data has been collected in order to synchronize a local database to any changes to the monitored bug repositories.
- Periodically creates a term-document matrix of all collected bug reports.
- Calculates the similarity of bugs to a query using latent semantic indexing in order to return highly relevant results.

- Allows for expressive search queries consisting of a combination of terms, preferred features, and feature filters, all of which can have separate importance weights specified as part of a query.
- Presents an easy to use but powerful interface for users to interact with the system.

The search engine, called *Buglook*, has the the following additional properties:

- It completely abstracts away all differences between separate bug trackers. This applies not only to the significant presentational differences between sites, but also to the slightly different data models used by each bug tracking system. To achieve this, it maps each system's data model into a single unified data model which it uses in all subsequent operations.
- It is useful for searching in currently existing deployed popular bug trackers, rather than in specifically-defined theoretical ones that may never be deployed on a large scale.
- It strives to minimize the strain put on the sites it processes, beyond an initial systematic download. This is possible with the use of several techniques such as caching and update monitoring.
- It is designed to be efficient even against highly active bug repositories and many users querying the system simultaneously.
- It is divided into five independent pieces - data collection, storage, indexing, query answering, and user interface. The different pieces can run on separate physical machines. They can perform most of their functionality even when other pieces fail. For example, the query answer component and the user interface together can still answer queries when everything else is turned off or has failed.
- It presents an efficient user interface. As much processing as possible is done in advance of a search query. All non-interface components are able to run on a different physical machine from the interface components.
- It is highly configurable in terms of which BTSs are to be processed at what rate by what components, e.g. several data collectors can crawl several trackers but aggregate their data into the same storage system.

When I describe Buglook's existing functions, I also mention additional functionality that I did not have the time to implement within the timeframe of this thesis. In particular, I describe several approaches to turn Buglook into a significantly more scalable distributed system.

Finally, I also present a study of current popular bug trackers and their properties useful for bug search.

My thesis is organized as follows. Section 2 discusses previous research related to Buglook. A study of existing popular bug trackers and their properties is presented in section 3. Section 4 describes the overall architecture of the search engine, followed by sections 5, 6, 7, and 8, which go into the details of each subsystem. Section 9 is

a discussion of several ways to turn Buglook into a more scalable distributed system. Buglook's evaluation follows in section 10. Finally, section 11 contains ideas for future research, and I conclude the thesis in section 12.

2 Background and Related Research

2.1 Trouble Ticket Systems

The requirement for automated handling of trouble management systems via well-defined standardized interfaces is as old as trouble management systems themselves. Several documents [23, 33, 24], some of which almost as old as the World Wide Web, attempt to define precise formats for trouble tickets, as well as interfaces between service clients and service providers. In particular, [24] bears significant similarity to modern popular bug tracking systems. These documents however have important drawbacks: they are either too general and therefore lacking in detail and implementation [24], or too focused on telecommunications services and thus not applicable to software bug tracking systems.

Recognizing these problems, [26] draws upon ideas contained in the above documents and defines a trouble report format suitable for use in the IT domain. While in principle the Customer Service Management Trouble Report (CSMTR) format defined in [26] could be used by defining mappings between existing BTS data structures and the format, I have opted to use the *Unified Data Model (UDM)* presented in [54] instead. The UDM is a data structure that represents a bug report. It allows a bug report to be machine-readable easily, without the additional burden of presentational elements in it, and it abstracts away the differences between different bug tracking systems. The reason for this choice is that CSMTR still largely focuses on the interaction between a customer encountering a problem and a service provider dealing with it - a situation not necessarily encountered in the domain of software BTSs. A user searching a BTS for a solution to a software problem is not generally interested in who encountered the problem first or to whom it was reported; rather, the user is concerned with the problem itself and any solution contained within its report. The UDM of [54] recognizes this use case and mostly ignores such unnecessary information. In addition, the UDM is directly based on actual data structures found in popular BTSs. It is further described in section 5.1.

Mapping the bug report schema of every BTS to the UDM is not the only possible means of data integration. An alternative method used in the Piazza [52] system is to map differing data schemas to each other, rather than to a central mediated schema. The advantage of doing so over the UDM is that if the central schema has been designed as a best fit to a large number of dissimilar schemas, the conversion from any schema to the central one will be very lossy. The UDM, however, does not have that problem: it has been specifically designed to mediate several highly similar bug report schemas. Mapping to the UDM avoids some of the issues Piazza must face: the need to reformulate queries into one of many schemas, and the need to design a mapping between each possible pair of schemas.

Further studies [47, 28] focus on the efficiency and automation of ticket generation and maintenance, rather than on formalizing or otherwise improving the output interface

of trouble ticket systems. An overview of artificial intelligence techniques for automated handling of trouble reports (Rule Based Reasoning, Case Based Reasoning, fuzzy logic) is presented in [28]; however these are not applicable to existing BTSs because they require a formal trouble ticket language within a tightly-specified domain, or unspecified mappings from free-form textual descriptions to such a formal language.

Several studies explore the problem of trouble ticket similarity, in order to come up with tickets that might be helpful in resolving a new trouble, and even to automatically derive a solution by modifying or combining existing solutions. In [27], Case Based Reasoning (CBR) is used to automate the entire trouble resolution cycle - detection, reporting, deriving a solution from a preexisting knowledge base, reporting it to a user, and enhancing the knowledge base with the new solution. A similar system that integrates CBR in a standard trouble ticket system workflow is developed in [30].

While these studies offer useful insights in terms of correlating trouble tickets, they impose very strict requirements on their application domain and on their input. More precisely, they are targeted towards specific well-understood scenarios (e.g. network management) and can therefore make assumptions that do not hold in general, such as that there is a finite known set of concepts or that terms can effortlessly be related to each other because they were manually classified in categories. The domain of general bug reports is infinite: bugs describe software, and software can have any number of purposes and related concepts. Further, CBR requires a formal language for expressing problems and solutions, but Buglook cannot impose such a restriction to its input data (bug reports).

2.2 Information Retrieval

Information retrieval methods such as latent semantic indexing (LSI) [10] are very useful for correlating free-form textual documents that on the surface appear to be unrelated. LSI is a statistical method based on singular value decomposition that calculates the similarity of document vectors to other document vectors. It addresses the problem of *synonymy*, the ability to describe a single concept in more than one way, by discovering the similarity of documents via other common terms and assuming that different terms that often appear in similar documents are likely to relate to the same concept. It handles *polysemy*, the possibility of a single term to describe more than one concept, by statistically eliminating noise - terms that appear too often in otherwise unrelated documents have their similarity weights reduced. For a good mathematical description of the method, refer to [10].

Several enhancements to LSI are presented in [2]: relevance feedback, dynamic updates, and efficient storage. Relevance feedback is a technique that enhances the precision of LSI. Precision is one of two metrics that measure the performance of relevant document retrieval; the other metric is recall. Because LSI represents a query and a document in the same way as vectors, [2] claims that replacing the query with the vector sum of a number of very relevant documents and repeating the calculation improves the overall relevance of the returned document set.

The other enhancements to LSI relate to the actual singular value decomposition. Recomputing the document-term matrix on every update is very expensive. Folding-in and SVD-updating are methods to update an already existing matrix with new data as it arrives. The two methods have varying cost and correctness: the former is very cheap

but may lead to information loss if the new documents are significantly different from the existing ones in their use of terms, while the latter is more expensive but is as accurate as recomputing the entire matrix. Finally, [2] discusses methods to reduce the storage requirements of the possibly huge document-term matrix. For the purposes of Buglook these storage enhancements may not be necessary, due to both the computational power increases since [2] was published and the manageable size of the bug dataset. Section 3 suggests that the total number of publicly-available software bugs currently on the Internet does not exceed two million.

One alternative to LSI is Okapi [42]. Okapi is a probabilistic method of ranking document similarity that relies on statistical information about a database. It uses several variables such as the total number of documents, the average length of a document, the frequency of occurrence of some term in some documents or the entire collection, etc. It is less computationally demanding than LSI, and according to [51], performs significantly better on large heterogeneous document collections. Note that although Okapi's algorithm is known, no free implementation exists. The derivation of its various formulas has not been published; only the formulas themselves have.

LSI is useful to Buglook because bug reports contain much free-form textual data. In addition to this data, the search engine optionally uses features to assess bug similarity. A *feature* is a field-value pair with a typically predefined set of values that describes metadata for a bug report. Features denote important information about bugs such as whether the bug describes a problem or an enhancement request, whether the problem has been fixed, or what other bug reports the bug is dependent on. Features are stored in and retrieved from the BTS. The set of bug keywords and a textual summary are the only features whose values are not predefined[4].

Similarity of features is combined with semantic (LSI) similarity to obtain a final score. The method is similar to the ordered weighted averaging function [60] defined in [54], which itself is derived from a similar formula in [31]. The difference between Buglook and [54] is that the latter derives so-called feature vectors from bug reports and represents bug features as numbers in these vectors. Buglook does not use feature vectors, but compares the features of a query to the features in a bug directly. The approach of using both feature vectors and semantic vectors to asses bug similarity is first introduced in [56].

2.3 Bug Retrieval

Before a bug report can be processed in any way, it must be retrieved first. Traditional web crawling [36] is not suited to the websites of BTSs for several reasons:

- BTS web pages are generated dynamically, i.e. they do not exist until they are requested.
- There is generally no easily accessible index of all bug web pages in a given BTS, and there is no guarantee that every bug report is linked to in at least one other page that is itself accessible by following links.
- Most hyperlinks are self-referential, i.e. they point to the same page, or to a different page that contains the same structured information.

[4]The reader should be careful not to confuse a keyword set with a textual query.

- Hyperlinking most often does not indicate relationships between the page that contains the link and the one pointed to. For example, a page for some bug may contain a link "Next" that points to a completely unrelated bug, only because the two bugs were posted at nearly the same time, and the user requested a list sorted by creation date.

The last point renders classic ranking techniques such as PageRank [37] virtually useless in this case, because they rely on assumptions about linking relationships between bugs that do not generally hold within BTS sites.

A previous attempt to search bug reports specifically is described in [7]. Although this thesis has reused the Buglook name from [7], the two systems are entirely different. Rather than implementing search capabilities itself, [7] attempts to provide a convenient interface to use the integrated search capabilities of BTS sites. It forwards queries to a site and parses the returned search result pages. The system I describe in this thesis, on the other hand, implements search capabilities on a local database, rather than relying on external BTSs for searching. Doing so eliminates [7]'s scalability and performance problems, because query response time becomes independent of BTS response time and server bandwidth. Moreover, local search also addresses the issue of excessive load that the previous system puts on the BTSs it visits.

Despite the limitations of [7]'s Buglook, there is a small amount of overlap between the two approaches. The current system reuses the idea of parsers tuned to specific sets of BTSs[5]. The parsers transform a bug report contained in a site-specific HTML page into a common data structure. A detailed description of these parsers is presented in section 5.2. Aside from them, there is no overlap in approach or implementation between the Buglook of [7] and the Buglook presented in this thesis.

2.4 Distributed Operation

During Buglook's development I realized that to achieve high scalability, parts of Buglook need a distributed architecture where multiple machines can cooperate to arrive at a result quickly enough. A centralized architecture runs into limits related to bandwidth, computational power, and possibly storage space. This subsection describes prior research relevant to distributing various parts of the system. How various parts of Buglook could be distributed is the subject of section 9.

2.4.1 Distributed Search

The NetTrouble TTS for network management [47] uses a simple query forwarding mechanism, where each instance of NetTrouble acts as a node in a network. A node is only responsible for any trouble tickets submitted to that node, but it forwards queries for other tickets to other known nodes. This simple scheme unfortunately treats data as completely independent items, and does not attempt to relate them in any way. In addition, [47] omits the technical details of how a given node knows what other nodes to contact for information which it does not know locally.

More complex variations of distributing search queries have been proposed in [45] and [57]. Both studies recognize the problem of flooding search requests to a large

[5]The parsers are called *search modules* in [7].

number of peers in an unstructured P2P network. To address this problem, [45] adopts an interconnected hypercube approach to group peers so that related data is stored in close vicinity. On the other hand, [57] uses a feedback scheme to learn which peers tend to return useful results. Data in both studies is represented as cases from a case-based reasoning dataset, and therefore all cases have well-defined similarity metrics that can be used to improve query routing. While bug reports too have a well-defined similarity function (section 7.1), there is the additional difficulty that evaluating similarity via LSI requires access to the entire bug dataset.

Yet another idea is to base routing on the information within an ontology that categorizes data items into different topics. In [5], every document is a member of some category. Peers advertise the level of expertise (the amount of reliable data) they have for each topic, and this information is used to route queries close to some topic to the peers with the highest expertise on that topic, i.e the ones most likely to return relevent results. If one were to construct an ontology for categorizing bug reports, this approach could potentially become useful for Buglook as well.

The eSearch system [51] is a P2P IR system that has several useful insights to offer. Its core idea is to distribute documents according to the terms that they are relevant to, called *global indexing*, but to also store for each document the complete list of its related terms, a technique called *local indexing*. The former allows the system to only query a small set of nodes in order to find all documents relevant to a set of terms, and the latter allows nodes to rank documents without consulting other nodes. An important difference in similarity ranking between eSearch and Buglook is that eSearch uses the Vector Space Model (VSM) for relevance decisions, as opposed to LSI, which is an extension of VSM. While LSI is expected to assess similarity more accurately, it has the additional requirement over VSM that information about all documents must be known by all nodes. To compensate for the deficiencies of VSM as opposed to LSI (such as synonymy and polysemy), eSearch uses automatic query expansion [51]. In essence, it extracts the most important terms from the most relevant documents to an initial query, and adds them to the initial query to form a better final query. How automatic query expansion + VSM compares to LSI in terms of accuracy remains as an open research question. Last but not least, [51] proposes several enhancements to the Chord [49] distributed hash table protocol (on which eSearch is based) in order to achieve better load balancing among nodes.

Another P2P IR system, also introduced in [51], is pSearch. Unlike eSearch, pSearch uses Content-Addressable Network (CAN) [41] rather than Chord for routing, and LSI instead of VSM for document relevance evaluation. CAN partitions nodes in a Cartesian space of arbitrary dimension, assigning every datum a key that is a point in the Cartesian space. The fundamental idea of pSearch is to map an LSI-derived semantic vector of a document to a point in the CAN key space. Thus documents with high semantic similarity can be stored physically close to each other in the CAN overlay. In achieving this, pSearch addresses some of the same problems that a distributed Buglook would encounter.

First, if documents are grouped according to their semantics, they are not distributed equally. A large number of similar documents will be packed together, while few dissimilar documents will be spaced out. To deal with this issue, pSearch concentrates more nodes around densely-populated regions of the key space. A node that wants to join the

P2P overlay joins at a point in the CAN space that corresponds to the semantic vector of a randomly chosen document. Second, because of the high dimensionality of the LSI vectors (and thus the CAN key space), a successful search that includes all relevant documents is likely to have to cover a large amount of the search space[6]. pSearch addresses this by estimating the similarity of documents in neighboring nodes. Because relevant documents are clustered together, [51] claims, the search results of a query p will be found in the neighbors of the node N_p where the semantic vector of p routes to. The issue, then, is that in a high-dimensional space, N_p has too many neighbors. To select the right neighbors, N_p continuously retrieves random semantic vector samples from all of its neighbors as a background process. Given p, it first searches those neighbors whose samples most resemble the query.

The contribution most important to Buglook in [51] is an analysis of LSI when certain variants of the algorithm are considered. Eight different variants are tested. The results conclusively show that normalizing the initial input matrix before any rank-reduction, the rows of the term-matrix after SVD (see section 7.2), as well as the semantic vector of the query before similarity evaluation, substantially improves performance on large heterogeneous corpora. Further, [51] develops an algorithm, called eLSI, that is almost as accurate as LSI, but is significantly more efficient. It reduces the size of the input matrix by clustering similar documents together and replacing them with their centroids, effectively "summarizing" each cluster into a single vector. This reduces the number of columns substantially. A different procedure is used to reduce the number of rows (terms), leading to a much smaller matrix used as input to the SVD calculation.

2.4.2 Distributed SVD Calculation

Limiting scalability and a candidate for parallelization is the CPU-intensive factorization of the LSI matrix (see section 7.1). When bugs are added to the search engine's database, their semantic vectors must be included in the periodic computation of a new document-term matrix. To this end, [4] and [35] observe that in certain cases calculating and decomposing a new matrix from scratch is not necessary, but rather a previous state of the decomposed matrix can be updated with new data. The two studies explore algorithms to take advantage of the known previous matrix. Further, [4] and [35] develop parallelizable versions of the algorithm, for use in the domains of robotics and LSI respectively. The algorithms are shown to be suitable within a parallel supercomputing environment, but it is unclear if their communication overhead is low enough to work in a distributed network.

Two general parallelizable algorithms that do not assume a preexisting SVD calculation are presented in [3] and [15]. While both state that they are suitable for a distributed network environment, [15] further claims to be provably optimal with respect to both computational complexity and network message overhead.

The document-term matrix used for LSI is very sparse. There is a potential for savings of both storage space and CPU power by exploiting this property. A good comparative study of sparse matrix storage algorithms is given in [18]. Buglook's storage system uses a sparse matrix format to create and transfer a matrix to the indexing system, where the decomposition takes place.

[6]This problem is a particular case of what is known as the *curse of dimensionality* [59].

2.4.3 Distributed Storage

Any distributed SVD algorithm requires that all participating nodes have the same view of the input matrix. One way to achieve this synchronization is to store all input data in a distributed storage system. Although significant research has resulted in many such systems [22], only a few are potentially useful to Buglook, because of its particular requirements: high performance, file locking semantics, and shared access.

FARSITE [1] is a distributed filesystem with full filesystem semantics. It works by replicating files on many hosts in order to ensure reliability and read performance. It provides full locking semantics via a distributed directory service. Hosts that update files notify the directory service, and if the operation is allowed, data is then also pushed to other hosts that keep replicated copies of that file. If overload of the directory service is detected, certain portions of it are delegated to another directory service. FARSITE focuses on reliability, security, and consistency of data, but not disk space conservation or performance. Most importantly, no publicly-available implementation exists.

Kelips [21] is a simple gossip-based protocol for the distribution of files among hosts. In other words, all nodes have the same view of the entire set of stored files, but a single file may only be stored on some nodes, perhaps only a single one. It does not specify how files are actually stored, and it does not concern itself with file replication or security. As a result, it has a very simple design. For a network of n nodes, Kelips features $O(1)$ lookup-time at the expense of $O(\sqrt{n})$ routing table space overhead. The system is highly resistant to churn, but requires a constant high-bandwidth flow between nodes in order to maintain its routing tables. Reliability via file replication must be handled by the application. Kelips's design could be useful for Buglook in cases where distributed nodes share a fast network connection between them.

Ivy [32] is a distributed filesystem with NFSv3-like [6] semantics. Its main strength is that it allows modifications to existing files and concurrent writing without locks, by storing files as vectors of write-logs of different nodes. It does not require that nodes trust each other. Although the system provides mechanics for an application to resolve concurrent write conflicts, it cannot always do so autonomously. Ivy has good write performance, but reading can be very slow for files that are frequently modified by multiple hosts.

PAST [43] is a storage utility designed to run on top of the Pastry [44] distributed hash table. PAST is not a filesystem; files stored in it are immutable. PAST focuses on reducing the storage requirements of single nodes, reliability over slow links and unreliable peers, and security - design choices that are not critical to Buglook. Its performance is limited by the fact that lookup may need to go through several nodes before it reaches a node that can provide a given file. PAST assumes that a file is written once and read many times; the overhead of file insertion is relatively large.

OceanStore [25] is a global storage system that attempts to create a ubiquitous automatic reliable storage solution. OceanStore replicates files among many servers. Files are of two types - active files that are stored on a few reliable servers and contain provisions to be updated consistently, and archived files which are read-only files stored on potentially hundreds of servers simultaneously. OceanStore has two routing algorithms: a fast probabilistic one that is always tried first, and a slow deterministic one that is used if the former fails. Other features are complete security over untrusted nodes, the possibility to update files, access controls, and automatic learning of better routes

over time. The system also offers several APIs - from a full-featured software library to a Unix filesystem interface. A prototype of the system called Pond exists, but its performance has not been widely studied.

The Google File System (GFS) [16] is the distributed file system developed and used by Google. It maps files to fixed-sized chunks that are distributed under hundreds of trusted, but unreliable chunkservers for storage. A single master server manages the metadata for all files, such as mappings to chunks and directory structures. The system is optimized towards large multigigabyte files that are simultaneously appended to (but not overwritten) by several writers. It focuses on reliability and most of all performance, with little attention to security. Because it is used on a single high-speed network, the filesystem does not concern itself with routing. Although GFS is proprietary, a free filesystem called HDFS [53] based on the same principles as GFS is available.

CFS [9] is a block-level distributed storage system. It consists of a set of servers that distribute blocks of a small fixed size, with some replication, among each other. A client can store and retrieve blocks; a hash function determines where a certain block will be actually stored. Routing is implemented on top of Chord. It is the client's responsibility to define a filesystem that uses the block interface. CFS's reference implementation uses a tree concept, where a filesystem's metadata is stored in a linked list of blocks, and files are trees with root blocks pointing to actual data blocks much like inodes in traditional disk filesystems. During retrieval, once a root block has been received, data blocks can be requested in parallel, improving both load balancing and performance. CFS is optimized towards wide-area deployment over slower links.

Finally, Lustre [8] and GlusterFS [17] are distributed filesystems used in high performance computing applications. They focus on performance and reliability under the assumption that there is a very high speed connection between nodes. Both are widely used and thoroughly tested, and have free implementations.

3 Popular Bug Tracking Systems

One of the key design goals of Buglook is that it must operate on currently existing deployed BTS sites. For this reason, it is useful to know what the most popular BTS sites are, and what BTSs power them. This section presents a survey of several popular BTSs and BTS sites.

I explored the features of several BTSs, focusing on several properties that are relevant to this thesis. For each BTS I checked whether the BTS supported the functionality in question, and if so, which bug sites (i.e., specific installations of a BTS) supported that function. The results for BTSs and sites are shown in table 1 and table 4 respectively. An explanation of each column in these tables is given below.

For each BTS, I looked at how the system is licensed for use, how its collection of bugs can be accessed, whether it is possible to easily receive notification of updated data, whether the database schema used by the BTS is documented, whether the BTS keeps track of bug dependencies, and whether the BTS can be searched for bugs with a given property.

The license of a BTS affects its popularity. Generally, large free software projects tend to prefer a BTS licensed as free software itself. These projects also tend to be the ones that make their BTS sites public. For this reason, I excluded proprietary BTSs

BTS	License	Access	Updates	Schema	Deps	Search
Bugzilla	MPL	HTML, XML-RPC*	SMTP, RSS*	textual	optional	filter, keywords
Mantis	GPL	HTML, SOAP*	SMTP, RSS	graphical	yes	filter
Trac	BSD	HTML	SMTP, RSS*	graphical	no	filter, keywords
Debian BTS	GPL	HTML, SMTP	SMTP	unknown	optional	filter
phpBugTracker	GPL	HTML	SMTP	textual	yes	filter, keywords
Flyspray	LGPL	HTML	SMTP, RSS, XMPP	unknown	yes	filter, keywords

Table 1: Overview of bug tracking systems and some of their features (as of October 2007). Items marked with * are optional for each site. "Deps" denotes "Dependencies".

from the survey.

The most important function of a BTS is retrieving bug reports in their most current states. All systems must at a minimum have an HTML-based web interface, but convenient automated retrieval requires a formalized programmatic interface, typically based on XML-RPC or SOAP. While all BTSs support e-mail (via SMTP) as an update notification mechanism, e-mail is non-trivial to use for a web application. For instance, to receive notifications for all bug reports in a BTS by e-mail, an application would need to have its own e-mail address, register an account with that e-mail address in the BTS, and subscribe for every bug report of interest. Some systems support RSS or Atom feeds, which are significantly easier to use by a program.

In order to understand the structure of the information stored in a BTS, I investigated whether the underlying data model is documented. Some systems provide this information in a textual format while others only have informal raphical representations. Some systems do not provide a clear description of the data model underlying the BTS at all and it is necessary to reverse-engineer the data model by looking at specific bug reports.

Tracking any dependency relations between bug reports is useful, because it helps to correlate bugs. Generally, a bug is dependent on another bug if it cannot be resolved or acted upon, until the dependency is itself resolved or acted upon first.

Some systems allow full keyword search for their reports, while others only support searching via a set of predefined filters applied on the entire bug database.

Based on popularity and available documentation, I chose to focus on four BTSs: Bugzilla, Mantis, Trac and Debian BTS. With the exception of Debian BTS, which is only used for the Debian GNU/Linux distribution, the chosen BTSs all publish lists of known public sites that use them for bug tracking. Starting from these lists, I investigated all sites that were accessible and did not require authentication to browse their repositories (about 85 sites). Table 4 lists all sites with at least 1000 stored bugs.

For each site (as of October 2007), Table 4 shows which version of what BTS is used, how many bugs are stored there in total and how many have been added in one week, indicating the activity of the site. The table also specifies whether the site has been customized from its base BTS, whether it supports a programmatic XML-RPC interface or RSS feeds, and whether the site supports bug dependency relations. Based on table 4, I have estimated that the total number of publically available bugs is about 2 million.

The version of the underlying BTS largely impacts its set of available features. For example, Bugzilla only supports RSS feeds as of version 2.20, and XML-RPC as of version 3.0. Note that some sites hide this version number, possibly because this information may be sensitive with respect to unknown security exploits in the BTS source code.

The number of stored bugs and the rate of opening new bugs indicate the popularity and activity of a site. It is notable that the margin between the most popular Bugzilla sites and the most popular sites using other BTSs is very large. I believe the reason is that Bugzilla was the first widely-known open-source BTS when it was released in 1998. Mantis was only started in late 2000, and Trac is even newer.

Some sites customize their BTS in order to provide better integration of the bug tracker into the rest of their web site. While some sites only change the visual appearance of the BTS (marked as "light" customization in Table 4), others also modify the functionality of the BTS (marked as "heavy" customization). Customized sites pose a problem for automated bug retrieval: a system that is designed to derive structured data from presentational HTML (see section 5.2) will generally fail to handle a significant change of a site's appearance. In addition, customizing a BTS naturally makes upgrading the site to a newer version of the BTS much more difficult; therefore customized sites tend to lag behind in version number, and consequently lack features introduced in later versions of their underlying software.

Programmatic interfaces provided by protocols like XML-RPC or SOAP can be used by programs to directly query a bug tracker for structured data, without having to guess the value of any fields presented in human-readable form (HTML, e-mail). While this greatly simplifies interfacing to that bug tracker, no BTS currently makes such an interface a default option. It is an optional feature of the BTS at best, and not supported at all at worst. Only very few sites actually deploy and enable such programmatic interfaces, and clearly relying on their availability is not sufficient. RSS, on the other hand, is much more widely supported. RSS feeds allow programs to query a bug tracker for any updated bug reports, and while they are not as useful as XML-RPC interfaces, they still provide a better alternative to SMTP update notification.

4 Design and Architecture

This section describes the design decisions I made to build Buglook, whose capabilities are listed in section 1.4. These design decisions naturally lead to the architecture presented in section 4.2. In short, search is a complex operation that consists of several separate processes. The Buglook system is partitioned into subcomponents, each of which implements one of these processes and is described in its own subsequent section.

4.1 Design Goals

Buglook's primary task is to provide better search capabilities than classic textual search engines, both in terms of search result relevance and query expressiveness. Therefore I require a search technique that is more sophisticated than counting the number of literal text matches of a query within a set of documents. I have selected latent semantic indexing (LSI) [10] - an information retrieval method that is well-known for its accuracy and immunity to problems such as synonymy and polysemy. LSI's primary drawback is its large computational complexity. Perhaps because of its cost, LSI is not popular among search engines and is rarely used on a large scale with real-life (as opposed to experimental) datasets.

A good search engine needs to have high performance, and it needs to maintain its performance when dealing with large datasets and when queried by many users simultaneously. Scalability is Buglook's second key goal. Improving scalability is an iterative process - it consists of breaking up a system into as many modules as possible, recognizing which modules cause performance bottlenecks, and breaking up or redesigning these modules further.

Reliability is Buglook's third goal. Working towards features and scalability inevitably increases Buglook's overall complexity. To remain reliable, a complex system must be designed in a way that allows different parts to function when other parts fail.

These goals call for a modular architecture where modules are as independent as possible, perform completely distinct functions, and together comprise a search engine that uses LSI to give the best possible search results.

4.2 Four Daemons

Buglook consists of five components. Four of them act as services that are started once and perform their work indefinitely (or until they fail). They communicate between each other when necessary. These four services, called *daemons*, are:

- The *spider daemon* (section 5) collects bug reports from various BTSs and converts them into the Unified Data Model (section 5.1). Each successfully converted bug report is immediately sent to the storage daemon where it is saved persistently.
- The *storage daemon* (section 6) receives bug reports from the spider daemon and stores them persistently. When contacted by the math daemon, it uses its storage pool to build a document-term matrix of all bugs. This matrix, together with several other important pieces of data, is then sent to the math daemon.
- The *math daemon* (section 7) is the heart of the LSI method. It performs singular value decomposition (SVD) on a term-document matrix to obtain three matrices that can be used to answer queries.
- The *query daemon* (section 8) answers search queries. Once it receives the math daemon's output, it answers queries based on that dataset until it is contacted again and given updated matrices to work with.

The fifth component is a client library, called the *query client*, that submits queries to and receives results from the query daemon. Currently it is used within a web application, but it could also be used to build desktop search interfaces or be embedded into other programs.

All Buglook components are stateless by design. Apart from a very small number of configuration settings, they do not maintain any persistant state. This is a step towards Buglook's reliability goal - if a daemon terminates for any reason, it can be restarted and it will integrate into the otherwise running system automatically. To this end, all operations that depend on other daemons are carefully watched for failure, and attempted again if they fail or time out.

The overall Buglook architecture is depicted on figure 1. Note that any number of spider daemons can feed the same storage system, and the math daemon's output can

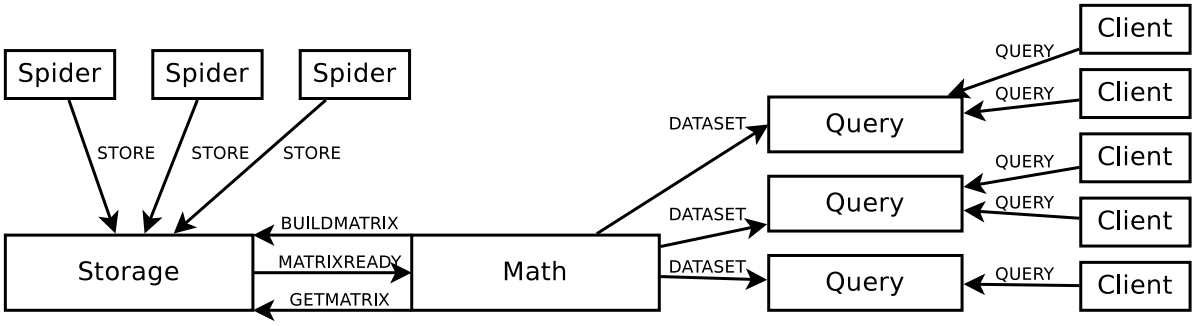

Figure 1: Buglook architecture: components and messages.

be used by any number of query daemons that can answer queries simultaneously as a load balancing mechanism.

4.3 Implementation Details

I wrote the majority of Buglook in the Python programming language [40], whose strengths are portability, fast development, flexibility for easy refactoring (thanks to a weak type system), and an impressively broad standard library. Only performance-critical sections are written in C, because it allows high performance with precise control over memory management and multithreading, at the cost of programming complexity.

All C portions of the codebase are enclosed into a single Python module called the *Buglook C eXtention (BCX)*. The storage, math, and query daemons make use of the BCX.

Buglook's web interface is based on Django [11], a Python web application framework. Django is relatively new and its application programming interface is not stable at this time, but is is very simple to use and intergrates well into existing web server setups.

Buglook additionally uses shove [48] for storage, Twisted Matrix [58] for networking and thread management, and SVDLIBC [50] for the matrix decomposition required by LSI.

4.4 Communication between Daemons

The four daemons communicate using a TCP-based network protocol, inspired by the HTTP [13] protocol. It uses request and response codes like HTTP does. For each message, one daemon assumes the role of a client while another acts as a server. The client opens a connection to the server and sends a request in the format:

```
operation parameter BUGLOOK/1.0
```

This line is followed by a blank line indicating the end of the request. `operation` indicates what the request is for, and `parameter`'s meaning depends on the request type. For some requests it can be blank; in that case there are two spaces between the operation identifier and the protocol version identifier. A blank line follows. The server answers with a response:

```
code message BUGLOOK/1.0
```

`code` is a status code that indicates the result of the operation - success of failure - while `message` is a corresponding human-readable description (or just `OK` for success). A blank line signals the end of the response. After sending it, the server closes the connection.

Both a request and/or a response can contain a data payload. In that case, the first line of the message is followed by the following line:

```
sha1sum length
```

`sha1sum` is a 20-byte hash value calculated over the data payload with the SHA1 [12] hash function, and `length` is the size of the data payload in bytes. These two fields together guarantee that the entire message was received without any data corruption. The blank line comes next, and last comes the data payload itself.

The data payload's precise content depends on the message, but it is generally constructed by taking a list of Python objects, serializing them with Python's pickle [38] algorithm, and compressing the resulting byte stream with zlib [14] compression.

Communication and multithreaded operation are both handled by the Twisted Matrix [58] library. It provides a convenient interface for scheduling simultaneous tasks and handles networking automatically as long the protocols themselves have been specified within callback functions.

Unfortunately, when I tested Buglook on non-trivially-sized datasets I found out that Twisted runs into problems when many connections are established over a short period of time, such as for instance when a spider daemon feeds bug reports to a storage daemon. Twisted also has several unrelated issues with task scheduling, which Buglooks works around in less than elegant ways. In retrospect, these problems and the low quality of its documentation make Twisted Matrix a poor choice.

I reimplemented the most performance-critical network client, the query client library, using sockets instead of Twisted Matrix, and the random connection timeouts disappeared. I did not have the time to reimplement all network connections with sockets, which means that bug retrieval performance is currently worse than it can be.

5 Data Collection: The Spider Daemon

Before it can search anything, Buglook must first acquire bug reports. A *spider daemon* is the part of Buglook which is responsible for communicating with a BTS. (Because many spiders can connect to the same storage daemon, multiple sites can be used simultaneously.) Initially, a spider obtains all bugs from its assigned BTS, and then monitors it for new and updated bugs. Every new or updated bug report is sent to the storage daemon as soon as it is available.

5.1 The Unified Data Model

All bug reports within Buglook are stored in the Unified Data Model (UDM) format. The *Unified Data Model* [54] is a data structure designed to represent bug reports from similar, but differring BTSs. It is a model that abstracts away the small differences

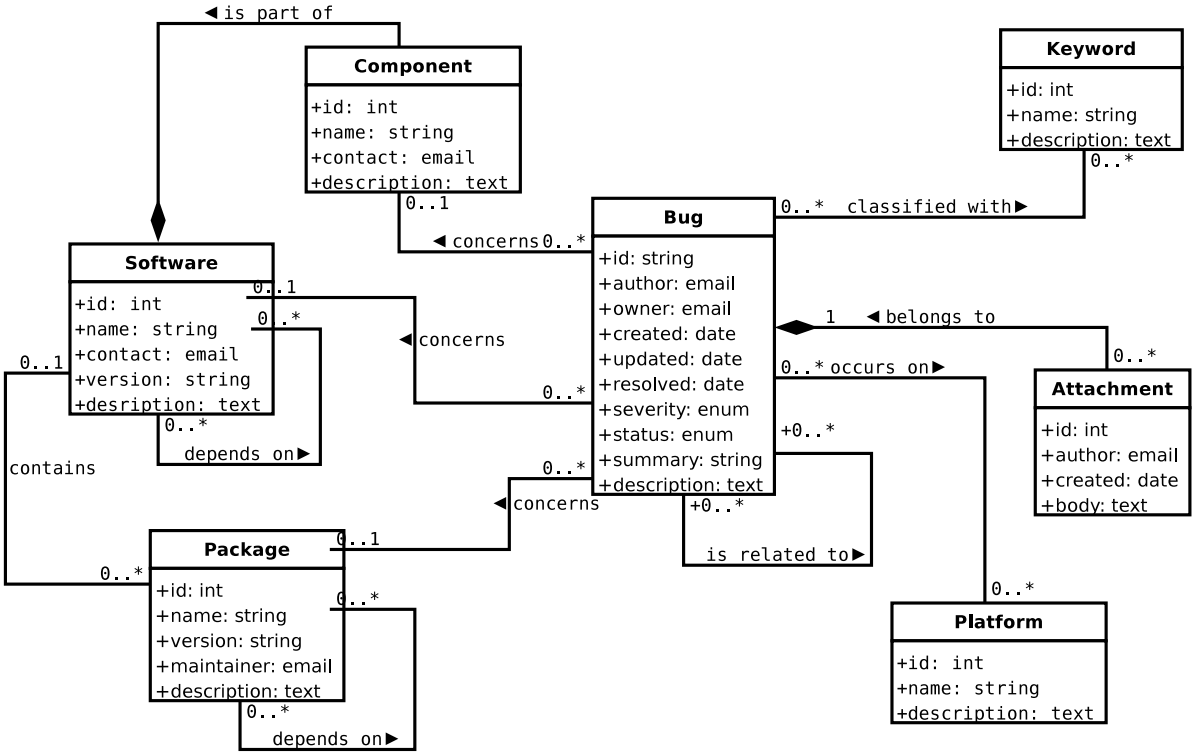

Figure 2: Unified Data Model UML diagram [54].

of bug representations present in popular bug trackers while preserving all information important to searching for bugs. Apart from the model itself, [54] defines mappings from the representation formats of several popular BTSs (Bugzilla, Debian BTS, Mantis, Trac) to the UDM.

The model as well as the mappings to the model are designed to minimize the data lost in conversion to UDM. Such data loss is inevitable when different BTSs have different expressiveness with respect to a given bug feature (e.g. some BTSs have more possible values for the "severity" attribute of a bug than others), or when some bug trackers have fields that others lack altogether.

A UML diagram of the model is shown on figure 2. Buglook uses the UDM as its core data structure to represent downloaded and parsed bug reports. An example bug in UDM is given in figure 3.

5.2 Parsing Bug Report Web Pages

While some BTS sites provide a machine-useable web service interface to their bug data, most do not. A web service interface (such as XML-RPC or SOAP) is only supported on some BTSs, and only in new versions. In all systems where such an interface is supported, it is an optional feature, and because optional features require additional effort from an administrator to be set up, they are rarely available. In addition, a web service interface often provides much less data than the human-readable web interface that is most commonly used.

Consequently, relying on the availability of a web service API is unrealistic. Instead, Buglook attempts to directly use the presentational HTML-based web interface which

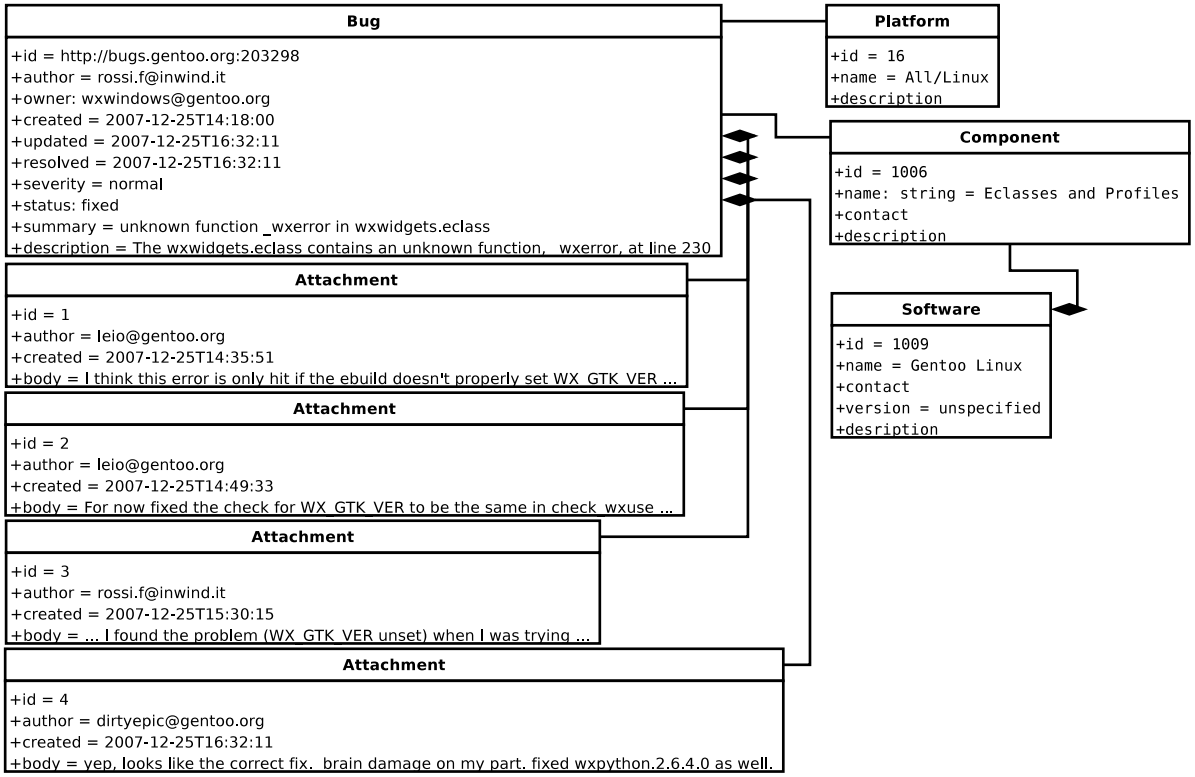

Figure 3: Example bug report in UDM.

human beings use in order to get as much access to information as ordinary users would.

The problem with presentational HTML pages is that the same structure can be presented in vastly different ways. As an example, consider an algorithm that must detect the end of a bug report comment and the beginning of the next one. In HTML, this boundary could be encoded as the closing of a `<div>` tag and the opening of another `<div>`, or as a new paragraph (`<p>`), or why not as a sequence of newlines (`<br>`)? Nothing prevents using the same elements in multiple contexts while rendering them differently (dictated by CSS tags). No consistency can be expected.

To tackle this problem, I note the following observations:

- Because bug report pages are generated from a template, all bug reports within a single BTS site have the same structure. An algorithm that can parse one bug report into UDM can parse all bugs in that site.
- BTS sites that use the same software have similar bug structure and often also similar appearance. The underlying BTS software determines the structure of the data it can work with, and only allows presentational customization of the displayed HTML pages.
- For each BTS there is a canonical appearance. Most sites do not find it necessary to customize their appearance, and use the one that the BTS provides by default.

Because of these similarities, I claim that a small set of parsers defined for each BTS's canonical appearance, together with specialized parsers for the most important

customized sites, can provide a very high degree of coverage of all BTS sites. Each parser has a set of sites that it can cover - its *sitetype*. Sitetypes are essentially equivalence classes of BTS sites with respect to parsing.

The spider daemon is designed to accommodate a set of parsers beside a common component independent of all of them. The common component is responsible for generic tasks such as downloading web pages, parsing HTML, etc. The parsers encapsulate all logic unique to a given sitetype, but nothing else. This arrangement allows more sitetypes to be supported with minimal duplication of effort.

At the time of this writing Buglook contains only a single parser for Bugzilla sites. Even so, this single parser correctly deals with the differences among several Bugzilla versions. Using tables 4 and 5, I have estimated that the current Buglook parser would be able to provide the system with at least 650000 bug reports from several large and many smaller sites.

After it converts a downloaded bug report to the UDM using a parser, the spider determines the set of terms seen in the bug and the count of each unique term. This data is stored along with the bug report via a `STORE` request to the storage daemon.

To determine the set of terms, Buglook first removes common words such as prepositions, pronouns, "there", "anyway", etc. The list of words to remove is taken from the SMART Information Retrieval System [46]; this list is very popular with information retrieval systems. Then, each word is stemmed using the Porter stemming algorithm [39] in order to eliminate ambiguity caused by different forms of the same word (e.g. "compiling" vs. "compiled"). Terms are counted in the summary (title), description, and all comments of the bug, with weights of 3, 2, and 1 respectively.

5.3 The Number of Bugs in a BTS

How does Buglook know how many bugs there are in a BTS? There is generally no way to ask the system itself. The spider could try to obtain each bug in turn during the initial systematic download, but the ID space is likely not continuous. It is entirely possible that a few IDs have no bugs attached to them (called "missing IDs" below), but that more bug reports exist with higher IDs ("present IDs"). Such holes in the ID space could be many in number, but are typically small. To find the bug report with the maximum numeric ID, Buglook uses algorithm 1. This algorithm attempts to find the maximum bug ID without probing too many IDs in order to do so.

Finding the maximum bug ID is the first thing a spider daemon does when it starts up. But the maximum ID is a moving target - it will increase as new bugs are added. A spider has to periodically run another algorithm (algorithm 2) to update its maximum. This second algorithm is to be executed periodically and so should probe as few IDs as possible. To compensate for any potential confusion caused by holes, algorithm 2 contains an element of randomness. As the spider runs the algorithm periodically, that randomness is likely to eventually pick an ID above any small hole.

When it finds a new maximum ID, Buglook downloads all bugs with IDs between the previous maximum and the new one.

Algorithm 1 Minimal Attempt Greatest Id Calculation (MAGIC): determine the maximum bug ID in a BTS.

1. Try exponentially increasing bug IDs until a missing one is found. Call the list consisting solely of the the missing ID X and the largest present ID m. (If missing ID is not a hole, the maximum ID is between it and m.)

2. Starting from the element of X and going down towards m, try the midpoint between m and the last element of X. Append each missing ID to X. Stop at the first present ID and call that m. Append m to X. (If there were no holes so far, m is the most likely maximum ID. But if there were holes, the next step will likely jump over them.)

3. While X has at least 2 elements, try the ID halfway between the last two elements of X. Delete the last element of X at every step and update m to the last (which is also the largest known) present ID.

4. If a missing ID is found before X is empty, make the missing ID the last element of X and break out of the loop. (At the end of this step, the maximum ID is most likely between m and the last element of X.)

5. Try every ID between m and the last element of X. Update m every time a present ID is found. When a missing ID is found, declare m the maximum. (In very rare cases, a large hole above the maximum will go undetected.)

6. Repeat steps $1 - 5$ starting from m until no larger maximum is found. (Doing so will jump over reasonably large holes just above m.)

5.4 BTS Update Synchronization

A systematic download of all bugs in a BTS puts a great strain on the BTS site and should only be done when absolutely necessary. Once a complete download of a site has been performed, the local copy must be updated to reflect any changes on the BTS. Bugs in the BTS may have been added, updated, or rarely removed. While all BTSs provide a way to uniquely identify a bug via its identifier, knowing when an update happened is a difficult problem.

In a typical BTS, interested users subscribe to a bug by adding their e-mail addresses to a list. When the bug in question is updated, all e-mail addresses on the list receive a message. Emulating this behavior does not work well for Buglook for several reasons:

- Buglook is interested in all bugs, but there is no way to know when a new bug is added to the BTS.
- Adding a service email (e.g. "buglook@example.org") to every bug report is very obtrusive and may disturb users or administrators of the BTS.
- Implementing an e-mail client only to be notified of updates is very complex. E-mail messages sent by bug trackers are typically clear-text (and not HTML) and would require a separate parser to extract the ID of the updated bug.

A different update system is necessary. One straightforward way to get updated information is to periodically poll known bug reports for changes. A backend process

Algorithm 2 Simple Probabilistic Expansion of Largest Limit (SPELL): determine an increased maximum ID starting from an old one.

1. Starting from an old maximum ID m, try adding an exponentially increasing number to m until a missing ID x is found. Update m to be the largest present ID found so far.

2. Try random IDs between m and x until a present ID is found. Call the latter m. Let x be the last missing ID probed.

3. Try every ID starting from x going down to m. The first present ID is the new maximum.

could download some bug report, reparse it, and if it detects that the bug has changed, update the local database with the new copy. To obtain newly entered (i.e. unknown) bugs, Buglook could use algorithm 2 from section 5.3 to determine the current total number of bugs in the BTS.

The problem with this approach is that it is practically the same as systematically redownloading the entire BTS. It wastes bandwidth and incurs an unnecessary load on the bug repository. Another problem is that specifying a polling rate is a difficult problem. Polling too often will redownload many bugs that have not been changed, and polling too rarely will fail to quickly track updates to highly active bugs.

The approach taken by Buglook is to define an individual polling rate per bug report. Given a default (preferably conservative) value for a time interval t_i for each bug i in a BTS, as well as minimum t_{min} and maximum t_{max} boundaries, t_i is initialized to the default value. Bugs that are in a closed state are initialized to $t_i = t_{max}$, because they are not expected to get updates. Buglook will poll every bug t_i seconds after it has been downloaded first. To reduce stress on the BTS when all intervals t_i are equal, a limit on the maximum number of bugs polled can be specified, or alternatively, the t_i values can be initialized to a random value within the range $t_{min} - t_{max}$.

If bug i has been updated since its last polling or initial download, t_i is divided by two. If the bug has not been updated, t_i is doubled. The rationale of this algorithm is that highly active bugs that should be tracked closely will likely receive a series of updates concentrated in a small time period. One update often triggers the next until a solution to the bug is found. Inactive bugs, on the other hand, are likely to remain unchanged for long periods if not forever.

After several iterations, the t_i values should reflect how active each bug in the BTS is, and will allow for a relatively non-uniform download times for bugs, in order to distribute stress evenly over a longer time period.

5.4.1 Improvement: RSS/Atom Feeds

Most, but not all BTSs also support a better notification system - RSS/Atom feeds [34, 20]. By visiting a certain URL, a user (or program) can obtain an XML document describing any new or updated bugs. While in theory parsing this document could be difficult, Buglook would not have to do so. It would merely need to obtain a list of new or updated bug identifiers; then it could download and parse any new bug reports directly from the BTS.

Feeds provide an update notification mechanism that has low overhead and that is

passive in nature, i.e. it requires the BTS to act by updating its feed. All potentially unnecessary bug redownloads would be eliminated, enabling Buglook to interact with a BTS in a friendly non-obtrusive manner.

Due to time limitations, I could not implement feed support into Buglook for my thesis, but it remains as a future possibility.

Note that feeds do not provide any means to detect if a bug report has been removed from the BTS. I believe that this is not a significant problem, because deleting a bug report is an extremely rare event. Instead, bugs are kept in a closed state for archival purposes.

A simple solution to this problem when using periodic polling would be to detect that a bug no longer exists when refreshing it. Then, a `DELETE` request to the storage could be defined.

6 Storage and Matrix Construction: The Storage Daemon

Every bug collected by the spider daemon is sent to Buglook's *storage daemon*. The storage daemon, as it name implies, is mainly concerned with storing the incoming stream of bugs persistently. It uses shove [48] as a storage backend. Shove is a Python library that provides a simple dictionary storage interface that can have various backends plugged into it. Currently, Buglook only uses shove's filesystem backend because it is fast enough, but should it become necessary, shove provides an impressive list of alternatives.

Each bug is stored by sending a `STORE` request, whose parameter is the ID (URL) of the bug, and its playload consists of the bug itself and its term counts.

6.1 Building a Term-Document Matrix

At any given point, a `BUILDMATRIX` request initiates the construction of a term-document matrix. The `BUILDMATRIX` request has the caller's hostname and port number as a parameter and no payload. These are used to contact the caller back when the matrix is ready.

The LSI algorithm [10] requires a $t \times d$ matrix A where t is the total number of unique terms in all bugs, and d is the total number of documents (bugs). The a_{ij} entry of A is the weighted frequency at which term i occurs in document j, normalized so that the Euclidean norm of each column equals 1. More precisely, each element of A is defined to be:

$$\begin{aligned} b_{ij} &= \ln(n_{ij} + 1) \cdot \ln \frac{d}{D_i} \\ a_{ij} &= \frac{b_{ij}}{\sqrt{\sum_{k=1}^{t} b_{kj}^2}} \end{aligned}$$

where n_{ij} is the number of times term i occurs in document j, and D_i is the number of documents that contain term i. The formula corresponds to the *ltc* term weighting scheme, used with normalization to unit vector length [51]. The resulting large matrix is sparse, which means that it can be stored and transferred over the network very efficiently.

6.2 Guaranteed Atomicity

Building A is an I/O-intensive process because it requires scanning through all collected bug reports. A fast computer with a large filesystem cache and a modern hard drive will build the matrix in only a few minutes even for very large datasets. However, if bug reports are added or updated while the matrix is being built, the matrix may become inconsistent. On the other hand, if no input bugs are allowed to come in, the operation of the spider daemon(s) will be disrupted.

To guarantee atomicity of the matrix build, the storage daemon caches incoming bug reports in memory while the matrix is being constructed. The cache is a separate RAM-backed instance of shove called S_1. When the matrix build is complete, the contents of S_1 is poured back into the main store S_0. Still, the problem is circular - what happens to new bugs during the transfer from S_1 to S_0? The answer is a RAM-backed S_2, which is alternated with S_1 while the other is busy synchronizing S_0. Each subsequent transfer takes less time and eventually synchronization terminates. At that point S_0 once again assumes the role of the only store.

When the matrix is ready, the storage daemon sends a `MATRIXREADY` request to whoever requested the matrix. The request has no parameter and no payload. Upon receiving it the the math daemon must explicitly ask for the matrix via a `GETMATRIX` request, which has no parameter and no payload, but its reply includes a payload. The justification for `GETMATRIX` is to save wasting a potentially large transfer should the opposing side be unable to process the incoming matrix for any reason.

The payload to the `GETMATRIX` reply is a tuple (A, I_t, I_d^{-1}, M), where:

- A is the serialized sparse term-document matrix A. Unlike most objects, matrices are never serialized using Python's pickle module, because it is too slow and space-inefficient for the purpose. Instead, I have written a custom serialization routine for both sparse and dense matrices.
- I_t is a term index - a dictionary which maps a term to the number of its row in A.
- I_d^{-1} is an inverse document index - a list of document IDs ordered by the column number in A.
- M is a copy of the features of all bugs, i.e. a list of all bug reports with their descriptions and comments removed. Its order corresponds to the order of I_d^{-1}. Because the bulk of each bug is removed, M is much smaller than the entire dataset. It is necessary for feature search (section 8.2).

7 Latent Semantic Indexing: The Math Daemon

7.1 LSI Benefits and Algorithm

Within the matrix A described in the previous section, each document is a column of t numbers. Geometrically, each document is a vector in a t-dimensional vector space. The similarity between two documents is the dot product between their vectors. While in principle the matrix A could be used to determine the similarity between two documents, the authors of [10] observe that this method preserves too much information including

any semantic noise present within the indexed documents: words with multiple meanings (polysemy) and concepts that may be described in more than one way (synonymy).

To remove such noise, LSI orthogonalizes terms by replacing A with a matrix A_k of a smaller rank k. The matrix A_k captures the most significant information present in A, due to the way it is constructed. To construct A_k, A is decomposed using singular value decomposition (SVD) into the matrix product $T\Sigma D$. T is a $t \times m$ orthogonal matrix of terms ($m = min(t, d)$ and $k < m$), Σ is a $m \times m$ diagonal matrix of singular values, and D is a $m \times d$ orthogonal matrix of documents. The three matrices can have their rows exchanged so as to sort Σ by the magnitude of its singular values.

The rank reduction consists of only keeping the k largest diagonal values in Σ. The rest are set to 0, whereby Σ becomes a $k \times k$ matrix. The columns in T and rows in D corresponding to the deleted values are removed as well. The result is the matrix product $A_k = T_k\Sigma_k D_k$, where T_k, Σ_k, and D_k are $t \times k$, $k \times k$, and $k \times d$ respectively. Note that A_k is still a $t \times d$ matrix. By removing small singular values, [10] claims that semantic noise is eliminated, while most of the signal is preserved. Geometrically, the vector space basis for document (and term) vectors is replaced with another basis of a smaller dimension k. Because of the lower dimensionality, vectors cannot be perfectly represented in the new basis and therefore are slightly altered, and so are their dot products. The new products better reflect semantic similarity between the documents.

Note that the choice of k is an open research question. Although [10] suggests a value between 50 and 350, [3] claims that significantly higher values in the order of 1000 to 10000 produce better results for very large datasets. Buglook currently makes k configurable, but uses 350 by default.

For the reasoning behind the rank reduction improvement, refer to [10]. For the mathematics of SVD, see [19].

7.2 SVD in Buglook

On startup, the *math daemon* sends a `BUILDMATRIX` request to the storage daemon. When it eventually receives a `MATRIXREADY` request, it sends `GETMATRIX` to receive the tuple (A, I_t, I_d^{-1}, M). The latter three are not used by the math daemon, but merely passed along to one or more query daemons later (section 8).

To perform singular value decomposition on A, Buglook currently uses SVDLIBC [50], an SVD library written in C. SVDLIBC has a simple interface and a proven codebase. Unfortunately, it also has some important limitations.

First, the library was written with 32-bit processors in mind, and all variables for matrix indexes, number of rows and columns, etc. are signed. This limits the maximum possible size of a matrix significantly. Throughout SVDLIBC, the C datatype `long` is used, which differs in length between 32-bit and 64-bit architectures. SVDLIBC's built-in serialization is lossy in that it reduces the precision of matrix values. Furthermore, different serialization formats incur different levels of precision loss. Buglook uses a custom serialization mechanism that, while lossless, is not portable across architectures.

Because Buglook's BCX (section 4.3) is written around SVDLIBC, it inherits the latter's lack of portability. For this reason, all daemons that use the BCX (storage, math, and query) must run on the same architecture. I have only tested them on the AMD64 architecture; running Buglook daemons on anything else is not guaranteed to

work (although it probably will work, as long as the architecture is the same for all three daemons).

More importantly, SVDLIBC makes no attempt to exploit any parallelism that may be available such as processors with multiple cores or multiprocessor machines. This is Buglook's most critical scalability limit at this time, because SVD is extremely expensive and its computational complexity grows exponentially [19]. Buglook would benefit greatly from a parallel SVD algorithm that can be run on multiple machines, multiple processor cores, or even on a modern graphics card.

When the SVD computation is complete, the math daemon reduces the rank of A to obtain A_k, or more precisely, the three matrices U, Σ, and V^T that would yield A_k under multiplication. Σ is diagonal and stored as a vector, while the other two are serialized. These three items, along with I_t, I_d^{-1}, and M, are sent to one or more query daemons as the payload of a `DATASET` request. The request has no parameter. Afterwards, the math daemon again contacts the storage daemon for a new matrix to work on.

7.2.1 Improvement: SVD Incremental Updates

The LSI algorithm has the property that the reduced-rank matrix A_k generally uses information from every vector. Adding another document or term requires a complete recalculation of the matrix, but performing SVD on each addition is not feasible. The classic way to work around this problem is to add new information in bulk updates. Given a starting set of vectors, the matrix A'_k begins to be computed. New documents (and potentially new terms) arrive continuously as more data is obtained, and are put into a queue. When the computation of A'_k is complete, A'_k becomes A_k. A new computation of a new A'_k begins with all old documents and all documents in the queue. Arriving documents are added to a new queue, the new A'_k becomes the new A_k when it is computed, and so on indefinitely. This is exactly how Buglook works.

However, this approach could be improved. The problem is that new documents are only useful after a potentially high latency period, because they must wait both until their queue is processed and while the next computation completes. As a consequence, it is possible that new documents may arrive faster than they can be incorporated into a new matrix, i.e. there is a possibility that each next queue will have a length larger than the one before it. This danger grows with time, because each new matrix adds more documents to the previous one.

Two alternatives to the continuous matrix recomputation are described in [2]. The first technique, folding-in a new vector, is computationally fast but inaccurate in that it is not equivalent to a full recomputation, but only approximates it.

Folding-in a document is the process of appending a $t \times 1$ document vector v_{new} into the rank-reduced matrix A_k. The new vector is first projected in the reduced k-dimensional basis of the document space. This projection is equal to the k-dimensional vector $v_k = T^T v_{new}$. Appending this vector as a new column to the matrix product $\Sigma_k D_k$ would require that this matrix product be calculated in the first place, which would defeat the efficiency goal of folding-in. Instead, the $1 \times k$ vector $v_{new}^T T_k \Sigma_k^{-1}$ is appended as a row to D_k^T to form $D_k'^T$. The new A'_k is then simply $T_k \Sigma_K D'_k$ [2]. The inaccuracy of folding-in results from a potentially lossy projection of v_{new} into v_k, and from the loss of orthogonality of D'_k.

The second technique, called SVD-updating, is more expensive than folding-in, but

results in an accurate index - the same index that would be obtained by rerunning the SVD algorithm all over again. In addition, while folding-in can only be used to add a single document, SVD-updating supports bulk addition of several documents. SVD-updating consists of three steps - updating terms, updating documents, and updating term weights. Each step is independent of the other two.

The first step adds r new terms that are present in some documents that need to be appended to A_k, but are not present in the documents that are already reflected in the matrix. Each term is represented by a normalized $1 \times d$ term vector. The $r \times d$ matrix of new terms is appended to A_k, resulting in a $(t+r) \times d$ matrix B. The updated matrix $B_k = T_B \Sigma_B D_B$ is calculated from B by reusing portions of the calculations already used to obtain A_k. The mathematical details can be found in [2]. An analogous procedure is used to add s documents (each represented by a $t \times 1$ document vector), by appending the $t \times s$ matrix of new documents to B_k to obtain a $t \times (d+s)$ matrix C, which is then used to derive the rank-reduced matrix C_k. The final step of SVD-updating updates the weights of a set of terms for each document, that is, it updates the frequencies of occurrence of a given term in a set of bug reports. I omit the mathematical details of the procedure; they can be found in [2].

Folding-in could solve the problem of high latency between the availability of a document and its presence in the matrix used for search queries. SVD-updating could solve the problem of documents continuously arriving too fast into the system. However, I remain doubtful that these techniques could remove the need for parallel SVD computation.

8 Finding Bug Reports: The Query Daemon

Buglook's *query daemon* is the component which answers search queries. To do so, it first needs to receive a dataset from the math daemon in a `DATASET` request, as described in the previous section. The payload of this request is the tuple $(U^T, \Sigma, V^T, I_t, I_d^{-1}, M)$, where the matrix product $U\Sigma V^T = A_k$ (U^T is $k \times t$, Σ is $k \times 1$, V^T is $k \times d$), I_t is a term index, I_d^{-1} is an inverse document index, and M is a list that contains the features of all bugs. Until a dataset is received, all queries will fail with an error message.

A program or web interface uses Buglook's *query client* to submit a query to the query daemon. The query is a single string submitted as the parameter of a `QUERY` request, which has no payload. The reply's payload consists of a list of search results, in which each entry is currently defined tobe a $(URL, summary)$ pair.

8.1 Query Language

Buglook has a simple query language designed to be very easy to use. In fact, a user may not even know there is a query language at all. A simple list of words such as "lost connection" is a valid query.

A query is a list of space-separated tokens. A token is either a term or a feature. Terms are stemmed before being searched for, so for example `compile` and `compiling` both denote the term "compil". A term is any string that is not recognized as a feature.

A feature is a string that begins with the character `$`, followed by the specification of some attribute. Example features are `$created>=2007-04` (created in April

2007 or later), `$owner~security` (the bug assignee includes the string "security"), and `$severity<=minor` (the bug is minor or a feature request). Inputting features causes bugs that have some of these features to appear higher in the result list. Bugs that do not have any of the specified features will still be in the list. It is also possible that otherwise very relevant bugs with no matching features will end up above irrelevant ones that have matching features.

To modify this behavior, one may designate a feature to be a filter by prepending `!` to it as in `!$status=fixed`. Instead of modifying the order of bugs, filters remove any non-conforming bugs from the list. A bug report must match all filters to appear in the search results.

The result list is ordered by the similarity of the query terms and features to each bug report. Within the similarity calculation, each term or feature (which is not a filter) has a weight. By default all weights are 1.0, but each weight can be modified. Prepending `+` to a term or feature doubles its weight; `-` halves it. If more precise control is required, a user may also prepend a number as in `#2.71`.

The following two examples illustrate the query language:

- `race condition sshd` searches for the terms "rac", "condit", and "sshd", each with a weight of 1.0.
- `++race -condition #3sshd +++$status=fixed !$author~g.chulkov` searches for the terms "rac", "condition", and "sshd" with weights 4.0, 0.5, and 3.0 respectively. Strong preference (weight 8.0) is given to fixed bugs, and the list will only include bugs filed by people with "g.chulkov" somewhere in their e-mail address.

A full specification of Buglook's query language appears in appendix A.

8.2 Search Algorithm

The input to the search algorithm for every query consists of (T, W^t, F, W^f, F^f). T is a list of terms, F is a list of features, W^t and W^f are lists of weights for the terms and features in T and F, and F^f is a list of features acting as filters.

The query daemon first consults I^t of its dataset to find the index of each term in T, therefore formally T consists of the indexes of the search terms, rather than of the terms themselves.

In theory, given A_k and a document vector (the query), one would take the dot product of the query with each column of A_k. In practice, $A_k = U\Sigma V^T$, and consequently taking the dot product is a three-step process:

1. Initialize a vector $\vec{v}$ of k elements such that $\vec{v}_i$ is:

$$\vec{v}_i = \sum_{j=1}^{\#T} U^T_{i,T_j} \frac{W^t_j}{\|W^t\|}$$

   where $\|W^t\|$ denotes the magnitude of W^t given by $\sqrt{\sum_{j=1}^{\#T} (W^t_j)^2}$.

2. Divide each element $\vec{v}_i$ of $\vec{v}$ by Σ_i.

3. Calculate a vector $\vec{s^t}$ of d elements has each element $\vec{s^t}_i$ is equal to:

$$\vec{s^t}_i = \sum_{j=1}^{k} \vec{v}_j V^T_{j,i}$$

The resulting vector $\vec{s^t}$ contains the term similarity of each document to the query.

To calculate feature similarity, Buglook uses the list of bug features M. It iterates over all bug reports (M) and checks every bug's features against F. Each matching feature has its weight in W^f added to the bug's similarity score. The initial similarity score is 0.0. Ultimately the similarity scores of all bug reports form the feature similarity vector $\vec{s^f}$.

$\vec{s^t}$ is combined with $\vec{s^f}$ by averaging each element to yield $\vec{s}$, the final similarity vector. From the list of bug reports the query daemon removes all those that do not have all features present in F^f. The resulting list of bugs is sorted by similarity and returned to the query client.

Iterating over M for every query at first seems to pose a performance problem. In reality this is not the case because M is small enough to remain in RAM. I measured no perceivable performance hit caused by the lookup on a small dataset (about 1500 reports), and even after a hundred-fold increase the lookup time is expected to remain below one second. Furthermore, this penalty is only incurred when features are specified in the search query. It is mitigated by the fact that any number of query daemons can answer queries simultaneously. (Note that a query daemon can currently answer a single query at a time due to Python's threading semantics, but because feature matching and similarity calculation both happen within the BCX, it is almost trivial to make search multithreaded.)

Should feature lookup performance become a concern on extremely large datasets, it is worth noting that M is static for each dataset. Consequently it could be beneficial to build indexes over various attributes of M such as B-tree indexes over dates or hash table indexes over e-mail addresses.

Note that the query daemon additionally needs M in order to give information about search results to the user. Without M, the daemon would only know the URL and bug number of each search result, none of which is particularly descriptive.

9 Buglook as a Distributed System

Buglook already allows connecting multiple spider daemons to a single storage daemon, as well as connecting a single math daemon to multiple query daemons. Consequently, the search engine has no practical limit to its data collection or query answering performance.

Unfortunately this is not enough. The major performance and scalability limitation of Buglook is the process of singular value decomposition, which happens in the math daemon and is required by LSI. The only countermeasure against this bottleneck is a distributed architecture, wherein Buglook can harness the power of several interconnected computers to accelerate its work. During my thesis's planning stage, I intended to experiment with various distributed setups, which I outline in this section. In retro-

spect, most of them turned out to be unnecessary, which goes to show that premature optimization can indeed be counter-productive.

9.1 Peer-to-Peer Query Propagation

One way to distribute Buglook is to propagate queries to several instances of the search engine. In this scenario, a Buglook instance receives a request from a user. It performs a search procedure to answer the query locally. In parallel, it forwards the same query to other instances that it knows about. These other instances, recognizing that the request comes from Buglook as opposed to a user, attempt to answer the query but do not forward it further. Upon receiving answers from several other instances, Buglook takes the most relevant bug reports from each list, including its own, and returns the resulting best list to the user.

This architecture would make sense if answering queries were slow. With Buglook's nearly-instantaneous search performance, however, this technique would likely introduce latency and thus decrease performance, rather than help.

9.2 Distributed Storage

A different approach is to distribute the data acquisition step. Simply put, whenever an instance obtains some set of bug reports, it shares them with other instances, which will then avoid downloading those particular bugs. Going further, storage itself could be distributed via one of the systems described in section 2.4.3.

During the development of Buglook, I expected a centralized storage system to be unable to scale to large datasets. Contrary to my expectations, a modern system with a large disk cache and fast disks builds the document-term matrix very quickly. Because it is possible to collect bugs while the matrix is being built, the short time this process requires is not a problem at all. The delay is greatly offset by the time it takes to decompose the matrix with SVD into a usable dataset. In other words, distributed storage would not lead to any noticeable benefits and would not be worth the additional complexity.

9.3 Distributed SVD

The best opportunity for parallelization is the SVD calculation required by LSI. The math daemon would have to be split in several parts. A *master* math daemon would receive an input matrix dataset from the storage system and would distribute it to other math daemons. Assuming the availability of a parallel SVD algorithm (section 2.4.2), each daemon would perform its role in the distributed computation and would send its results to the master. The master would combine the result and send the dataset to the query daemon(s). The setup is illustrated on figure 4.

A potential pitfall of the distributed SVD approach is the possible overhead of managing the numerous math daemons versus the gains that the particular distributed SVD algorithm is able to provide. Further, with decompositions of very large matrices taking days, it is very important that the system is able to tolerate the failure of some of its math daemons. These issues, while far from trivial, are worth exploring due to their potentially vast performance benefits.

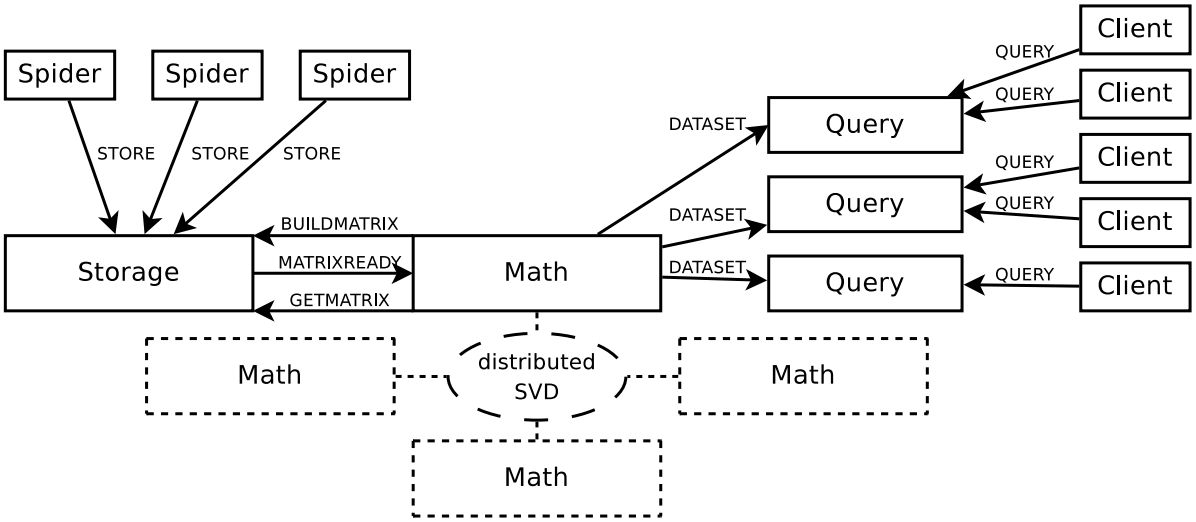

Figure 4: Buglook architecture with multiple math daemons.

Within the timeframe of my I was unable to experiment with distributed SVD. I must therefore leave this admittedly critical functionality as the subject of future work.

10 Evaluation

In this section I evaluate Buglook and how successfully it performs in terms of coverage, scalability, search quality, and search performance.

10.1 Data Acquisition Quality and Coverage

Data acquisition coverage refers to the number of bug reports that Buglook can reliably obtain and use to build its search index. Quality refers to the percentage of correctly-precessed bug reports within a single site.

A precise measurement of coverage would require to systematically download all bugs in all known public Bugzilla sites - an undertaking that could easily require weeks to complete. Instead, I downloaded small bug samples of most larger sites listed in tables 4 and 5, limiting myself to about a thousand bugs per site. Measuring quality is ever harder, because it requires that I manually make sure that a bug is acquired correctly. I did this with 10 randomly selected bugs per site. Consequently, my estimates for both coverage and quality are approximate at best.

Buglook's single Bugzilla parser appears capable of parsing at least 650000 bugs, because it covers many of the larger sites, as well as almost all small ones. Out of the estimated 2000000 total public bugs on the Internet, Buglook's coverage is at least 32.5%. It would be easy to provide custom parsers for the few heavily customized large Bugzilla sites, as well as for Mantis and Trac, to bring this figure dramatically up. I could not implement these extra parsers in time for my thesis.

No manually verified bug was parsed incorrectly, bringing my data acquisition quality estimate to an impressive 100%. I take this number with a grain of salt, however. Any bugs that might confuse the parser would have very unusual characteristics, and those

Search Engines	Matching Rate
Buglook-Bugzilla	0.424000
Buglook-Google	0.118507
Bugzilla-Google	0.136054

Table 2: Matching rates between search engines (all results).

bugs by definition would be very few and improbable to pick randomly.

Note that bug counts have grown by about 10% between October 2007 and August 2008. The increase applies both to the total number of bug reports on the web and on the number of bugs available to Buglook.

10.2 Scalability

Buglook's major weakness is the SVD calculation required by LSI, and I already discussed it in sections 7.2 and 9.3. Even on a fast 3 GHz Intel Core 2 Duo machine with 8 GB of memory, Buglook can only reasonably scale to around 10000 bugs. Unfortunately SVD's complexity is exponential, making a distributed SVD algorithm Buglook's only hope for large-scale deployment. The only alternative is replacing LSI with a less expensive similarity rating method, or with a modified more efficient LSI variant.

10.3 Search Quality

Most information retrieval systems are evaluated on controlled datasets with known similarity properties, allowing very precise measurements of *precision* and *recall* - metrics that measure the ratio of relevant results to total results and the ratio of relevant results to total relevant documents respectively [54]. Buglook, on the other hand, requires a very specific and largely unexplored dataset where the relevant results for a query are not previously known. Consequently, precision and recall are unfeasible to measure.

Instead, the best I can do is to use a metric introduced in [55] - *matching rate*. Given two search methods' result lists on a query, the matching rate is defined to be the number of search results found in both lists over the length of the shorter list. A high matching rate, [55] claims, suggests that the two search methods are good because they retrieved mostly the same documents.

Matching rate is most useful when the search methods in question are either very similar to each other, or are variants of the same method. In contrast, I compared three methods that are very different - Buglook's search with default weights and no features or filters, Bugzilla's integrated search with no filters, and Google's web search limited to the Bugzilla site in question. I expected a high matching rate between Bugzilla and Google and low matching rate between Buglook and either of the other two. I measured the matching rate between 12 queries. On three of them, Google failed to produce an acceptable number of results; as a result, I computed the matching rate between the other nine. The dataset was OpenSSH's `bugzilla.mindrot.org` Bugzilla site with 1487 bugs.

Table 2 shows that the results do not meet my initial expectations. First, Google's list is very different from both Buglook's and Bugzilla's lists. Second, there is much higher similarity between Buglook's and Bugzilla's lists. The first follows from Google's

Search Engines	Matching Rate
Buglook-Bugzilla	0.155555
Buglook-Google	0.061728
Bugzilla-Google	0.051851

Table 3: Matching rates between search engines (best 30 results).

inability to understand the concept of a bug report. Google does not know where to search in a page, or even which web page is a bug report and which is a list of bug reports. Although I gave Google a small advantage by filtering out results that were not bug reports, Google still returned consistently dissimilar results.

The high matching rate between Bugzilla and Buglook resulted from Buglook's search mechanism. Whereas Bugzilla and Google only allow results that have any of the search terms somewhere in the text, Buglook always returns the entire dataset, only changing the order to reflect relevance. Consequently, every bug returned by Bugzilla was also in Buglook's dataset. Because Bugzilla generally returned more results than Google, the similarity to Buglook ended up higher (false matching rate effect).

To remedy this error, I repeated the measurements but this time I only took the first 30 results and measured the similarity of those. This second test paints a somewhat different picture (table 3). All similarities are drastically small. Again, Google is very dissimilar to the other two. Buglook-Bugzilla still gives the largest similarity, even with the false matching rate effect reduced. This time, however, Buglook-Google does slightly better than Bugzilla-Google.

Matching rate measurements that small cannot be used to determine which methods are good, but they prove that the methods are very different from each other. A pessimistic interpretation could be that no method is good enough, and one would have to somehow combine all three to get good search results. Doing so would yield all good search results, but at the cost of many false positives.

A more realistic interpretation is that one of the methods is good, while the other two fail, but in different ways. Results consistently suggest that Google is one of the failing methods; intuitively, its lack of understanding of structure and its reliance only on literal keyword search could explain this. Google's PageRank algorithm is quite unsuitable to bug reports, for the reasons outlined in 2.3.

Both Buglook (via parsers) and Bugzilla (via direct access to the database) are aware of bug reports' structure. Which one is the good one among them? I claim that Buglook's LSI pays off: its ability to relate concepts and to take word variations (e.g. "compile" vs "compilation") let it retrieve bugs that Bugzilla's literal keyword search cannot. In a sense Bugzilla is the middle ground between Buglook's successful and Google's unsuccessful methods. It understands the structure of the dataset like Buglook does, but it is limited to literal keyword search like Google is.

For what it is worth, my subjective evaluation of search results confirms that Buglook's top results are the best among the three search engines. Of course, Buglook can return even better search results if the user chooses to exploit its advanced features such as term weights, features, or filters.

In summary, Buglook has better search quality and more features than either a generic web search engine or a search engine embedded in to a BTS site.

10.4 Search Performance

Search performance refers to the speed with which the search engine delivers results to a given query. Initially I expected that calculating the similarity of a query to every possible bug report would cause large reply latency. I was wrong - the complexity of search is linear[7]. On a 1500-bug dataset I measured an average latency of less than 10 ms with feature search enabled. This implies that 150000 bug reports can be searched in less than a second in the worst case, and likely faster than that. Performance is yet even better if features (or filters) are not specified in a query. Moreover, search can be parallelized to any number of query daemons. As a result, Buglook's search performance is on par with the fastest modern search engines.

In summary, Buglook has very reasonable coverage, excellent data acquisition quality, competitive search performance, and best-in-class search quality, offset by a disappointing scalability.

11 Future Work

In this section I outline directions for further research beyond what was possible in the timeframe of this thesis.

One possibility for improvement is to challenge the assumption that bug reports have no controlled vocabulary. Intuitively, bug reports for software consists mainly of terms from computer science and software engineering, as well as from vocabulary from the application domain of that software. If a taxonomy for both of these term sets were to exist then perhaps Buglook's similarity measurement could be improved further, such as by using taxonomy-based similarity metrics as in [29].

Bug reports differ from most general texts by the presence and high importance of machine output. Bugs often feature diagnostic messages, compiler errors, etc. These computer-generated passages carry a large amount of information that can be used to identify and relate bugs to each other and to queries. Buglook's current stemming algorithm works well with English words, but is completely unsuited to machine output. It may be worthwhile to explore algorithms that break up machine output into a list of terms, or make use of machine output in some other way.

Automatic relevance feedback as defined in [51] is another technique that looks promising. Automatic relevance feedback works by using the few most relevant documents to add even more terms to the original query. The new larger query has a smaller chance of missing important documents. In [51] the method is claimed to significantly improve the recall rate of term-based information retrieval methods such as LSI.

Finally, although I have opted to use LSI for Buglook, it is beneficial to explore alternative similarity ranking methods, such as Okapi [42] and eLSI [51], ideally by implementing all three and directly comparing their performance in terms of matching rate or other metrics.

[7] This is the case after a dataset has been prepared. Indexing itself has exponential complexity, but it happens ahead of any queries.

12 Conclusion

I have presented Buglook, a search engine for bug reports. It applies well-known information retrieval methods to the largely unexplored domain of bug reports and bug tracking systems. Buglook trades generality to gain expressiveness and better search result relevance.

Buglook takes a novel approach to data acquisition by relying on modular parsers for each type of bug tracking system is expects to search. In the absence of widely-deployed usable computer interfaces for interacting with BTSs, it achieves high efficiency in parsing bug report web pages while remaining workload-friendly to the sites it visits. It already covers many existing BTS sites, and many more could be supported with minimal effort.

Buglook is modular and extensible by design; all of its components work completely independently. This functional separation reduces the complexity of all components, improves reliability, and enables a flexible and extensible deployment architecture.

Buglook has a simple but powerful query language that is very easy to use, but allows very precise queries that return highly relevant search results with few false positives. It features very fast search of higher quality than either generic textual search engines or the search engines embedded into BTSs.

I believe that Buglook can be very useful. It provides a readily available testbed for further research on information retrieval methods as they apply to highly technical texts mixed with important passages of machine-generated output. More importantly, it can already help computer scientists, developers, and advanced users with the daily task of debugging software problems. To my knowledge, Buglook is the best way to search for bug reports on the Internet today.

Acknowledgments

I would like to thank Ha Manh Tran, Prof. Jürgen Schöenwälder, and Rodrigo Senra for their valuable advice and support. Without their insight and all the time they spent with me whenever I got stuck, this thesis would not make it. I also extend my gratitude to my family and friends, who put up with my busy schedule over several months. Thanks!

A Bugloook's Query Language

All Buglook queries are interpreted according to the rules outlined in this appendix. Below, the _ character denotes a space. Literals are enclosed in quotation marks ("); in particular, "" denotes the empty string.

```
QUERY    :-  TOKEN | TOKEN _ QUERY
TOKEN    :-  WEIADJ TERM | WEIADJ "$" FEATURE
WEIADJ   :-  "+" WEIADJ | "-" WEIADJ | "!" WEIADJ | "%" WEIGHT WEIADJ | ""
WEIGHT   :-  any integer or floating-point number with "." as the decimal
             point
TERM     :-  STRING
FEATURE  :-  FEATAUTH | FEATOWN | FEATCRE | FEATUPD | FEATSEV | FEATSTAT
FEATAUTH :-  "author" OPSTR STRING
FEATOWN  :-  "owner" OPSTR STRING
FEATCRE  :-  "created" OPCOMP DATE
FEATUPD  :-  "updated" OPCOMP DATE
FEATSEV  :-  "severity" OPCOMP SEVERITY
FEATSTAT :-  "status=" STATUS
OPSTR    :-  "=" | "~"
OPCOMP   :-  "=" | ">" | ">=" | "<" | "<="
DATE     :-  a date in the format YYYY-MM-DD or a prefix thereof
SEVERITY :-  "feature" | "minor" | "normal" | "major"
STATUS   :-  "open" | "fixed"
STRING   :-  any string that does not contain a space
```

Each token has a weight. The default weight of each token is 1.0. The weight adjustment (a string of `+`, `-`, or numeric weights) is used to modify the default weight and is parsed from left to right. A `+` doubles the weight, a `-` halves it, and `%`w sets the weight to w. Thus the adjustment `%5-` results in a weight of 2.5, while `++%3.14` is 3.14, because the `++`'s effect was "overwritten" by the `%3.14`. A `!` in the weight adjustment designates a feature to be a filter (whose weight is disregarded) and has no meaning for terms. Additional `!` modifiers have no effect. After the weight adjustment, a string that begins with `$` is considered a feature; all other strings are terms.

Features that operate on strings (`author`, `owner`) use the operator `=` to denote equality and `~` to denote inclusion, e.g. `author~chulkov` will match `g.chulkov@jacobs-university.de`. Features with dates (`created`, `updated`) use the canonical comparison operators. Note that it is possible to specify a date prefix. In that case, equality is checked only against the prefix; for example `created=2007` matches all dates in 2007. The other operators' meanings are the same as for string comparison, e.g. `updated>2008-04` matches all dates after April 2008.

For `severity` features, the ordering of values is defined to be:

```
major > normal > minor > feature
```

Each query is parsed into a 5-tuple (T, W^t, F, W^f, F^f), where T is a list of terms, W^t is a list of term weights (one for each term), F is a list of features, W^f is a list of feature weights (one for each feature), and F^f is a list of features acting as filters.

References

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[2] M. Berry, Z. Drmac, and E. Jessup. Matrices, vector spaces, and information retrieval. *Society for Industrial and Applied Mathematics*, 1999.

[3] M. Berry and D. Martin. Parallel SVD for scalable information retrieval. In *Proc. International Workshop on Parallel Matrix Algorithms and Applications*, 2000.

[4] T. Braun, A. Maciejewski, and H. Siegel. A parallel algorithm for singular value decomposition as applied to failure tolerant manipulators. In *Proc. 13th International Symposium of Parallel Processing*, 1999.

[5] J. Broekstra, M. Ehrig, P. Haase, F. van Harmelen, M. Menken, P. Mika, B. Schnizler, and R. Siebes. Bibster - a semantics-based bibliographic peer-to-peer system. In *Proc. 2nd Workshop on Semantics in Peer-to-Peer and Grid Computing (SEMPGRID '04)*, 2004.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification, 1995. RFC 1813.

[7] G. Chulkov. Buglook: A search engine for bug reports. Seminar Report, Jacobs University Bremen, 2007.

[8] Cluster File Systems, Inc. Lustre 1.6 operations manual, 2007.

[9] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[10] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 1990.

[11] Django - the web framework for perfectionists with deadlines. http://www.djangoproject.com/, retrieved on 2008-08-20.

[12] D. Eastlake and P. Jones. US secure hash algorithm 1 (SHA1), 2001. RFC 3174.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol - HTTP/1.1, 1999. RFC 2616.

[14] J-L. Gailly. ZLIB compressed data format specifiction version 3.3, 1996. RFC 1950.

[15] G. Gao and S. Thomas. An optimal parallel jacobi-like solution method for the singular value decomposition. In *Proc. International Conference of Parallel Processing*, 1988.

[16] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[17] GlusterFS user guide v1.3. http://www.gluster.org/docs/index.php/, retrieved on 2007-12-25.

[18] N. Goharian, A. Jain, and Q. Sun. Comparative analysis of sparse matrix algorithms for information retrieval. http://citeseer.ist.psu.edu/586549.html, retrieved on 2007-12-25.

[19] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics*, 1965.

[20] J. Gregorio and B. de Hora. The Atom publishing protocol, 2007. RFC 5023.

[21] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and scalable p2p dht through increased memory and background overhead. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[22] R. Hasan, Z. Anwar, W. Yurick, L. Brumbaugh, and R. Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *Proc. International Conference on Information Technology: Coding and Computing (ITCC '05)*, 2005.

[23] International Telecommunication Union - Telecommunication Standardization Sector. Trouble management function for ITU-T applications, 1999. ITU-T Recommendation X.790 - Corrigendum 1.

[24] D. Johnson. NOC internal integrated trouble ticket system functional specification wishlist, 1992. RFC 1297.

[25] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, 2000.

[26] M. Lager and M. Nerb. Defining a trouble report format for the seamless integration of problem management into customer service management. In *Proc. 6th Workshop of the OpenView Univerisity Association (OVUA '99)*, 1999.

[27] L. Lewis. A case-based reasoning approach to the management of faults in communications networks. In *Proc. 3rd International Symposium on Integrated Network Management with participation of the IEEE Communications Society CNOM and with support from the Institute for Educational Services (IFIP TC6/WG6.6*, 1993.

[28] L. Lewis and G. Dreo. Extending trouble ticket systems to fault diagnostics. *IEEE Network*, 1993.

[29] Y. Li, A. Bandar, and D. McLean. An approach for measuring semantic similarity between words using multiple information sources. *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[30] C. Melchiors and L. Tarouco. Fault management in computer networks using case-based reasoning: DUMBO system. In *Proc. 3rd International Conference on Case-Based Reasoning and Development (ICCBR '99)*, 1999.

[31] M. Montaner, B. Lopez, and J. Lluis de la Rosa. Improving case representation and case base maintenance in recommender agents. In *Proc. 6th European Conference on Advances in Case-Based Reasoning (ECCBR '02)*, 2002.

[32] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[33] Network Management Forum. Customer to service provider trouble administration information agreement, issue 1.0, 1997. NMF 601.

[34] M. Nottingham and R. Sayre. The Atom syndication format, 2005. RFC 4287.

[35] G. Oksa, M. Becka, and M. Vajtersic. Parallel SVD computation in updating problems of latent semantic indexing. In *Proc. ALGORITMY 2002 Conference on Scientific Computing*, 2002.

[36] L. Page and S. Brin. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th International Conference on World Wide Web*, 1998.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. *SIDL-WP-1999-0120*, 1999.

[38] pickle - Python object serialization. http://docs.python.org/lib/module-pickle.html, retrieved on 2008-08-20.

[39] M. F. Porter. An algorithm for suffix stripping. *Program*, 1980.

[40] Python programming language - official website. http://www.python.org/, retrieved on 2008-08-20.

[41] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, 2001.

[42] S. Robertson and S. Walker. Okapi/Keenbow at TREC-8. In *Proc. 8th Text Retrieval Conference (TREC-8)*, 2000.

[43] A. Rowston and P. Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles*, 2001.

[44] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[45] F. Ricci S. Berkovsky, T. Kuflik. P2P case retrieval with an unspecified ontology. *Artificial Intelligence Review Journal*, 2005.

[46] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[47] L. Santos, P. Costa, and P. Simoes. NetTrouble: A TTS for network management. In *Proc. SBT/IEEE International Telecommunications Symposium (ITS '98)*, 1998.

[48] Python package index: shove. http://pypi.python.org/pypi/shove/, retrieved on 2008-08-20.

[49] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 2003.

[50] SVDLIBC. http://tedlab.mit.edu/ dr/svdlibc, retrieved on 2008-08-20.

[51] C. Tang. *Data Sharing and Information Retrieval in Wide-Area Distributed Systems*. PhD thesis, University of Rochester, 2004.

[52] I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza peer data management project. *ACM SIGMOD Record*, 2003.

[53] The Apache Software Foundation. Welcome to Hadoop! http://lucene.apache.org/hadoop/, retrieved on 2007-12-25.

[54] H. Tran, G. Chulkov, and J. Schönwälder. Crawling bug trackers for semantic bug search. Jacobs University Bremen, 2007.

[55] H. Tran, G. Chulkov, and J. Schönwälder. Crawling bug trackers for semantic bug search. Jacobs University Bremen, 2008.

[56] H. Tran and J. Schönwälder. Fault representation in case-based reasoning. In *Proc. 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 2007.

[57] H. Tran and J. Schönwälder. Heuristic search using a feedback scheme in unstructured peer-to-peer networks. In *Proc. 5th International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P '07)*, 2007.

[58] Twisted. http://twistedmatrix.com/trac/, retrieved on 2008-08-20.

[59] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24rd International Conference on Very Large Data Bases (VLDB '98)*, 1998.

[60] R. Yager. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics*, 1988.

Site	System	Version	# Bugs	New Bugs
bugs.debian.org	Debian	N/A	349346	1036
bugzilla.redhat.com	Bugzilla	2.18-rh	177724	unknown
qa.mandriva.com	Bugzilla	3.0.2	33128	185
bugs.gentoo.org	Bugzilla	unknown	183365	538
bugzilla.novell.com	Bugzilla	3.0	61110	564
bugzilla.mozilla.org	Bugzilla	3.0.1+	173885	721
bugzilla.kernel.org	Bugzilla	2.22.2	9131	31
bugzilla.gnome.org	Bugzilla	unknown	364027	unknown
bugs.kde.org	Bugzilla	unknown	9655+	24+
issues.apache.org/bugzilla	Bugzilla	2.18.6	43554	unknown
www.openoffice.org/issues/query.cgi	Bugzilla	unknown	82396	249
bugs.eclipse.org/bugs	Bugzilla	unknown	204600	746
bugs.freedesktop.org	Bugzilla	2.22.1	12738	106
gcc.gnu.org/bugzilla	Bugzilla	2.20+	33800	unknown
qa.netbeans.org/issues	Bugzilla	unknown	116639+	unknown
bugzilla.mindrot.org	Bugzilla	3.0	1362	3
bugs.proftpd.org	Bugzilla	2.22.1-debian2	1968	4
www.reactos.org/bugzilla	Bugzilla	unknown	2722	24
bugzilla.samba.org	Bugzilla	unknown	5008	unknown
www.squid-cache.org/bugs	Bugzilla	3.0.1	2098	5
www.w3.org/Bugs/Public	Bugzilla	unknown	5157	48
bugzilla.wikipedia.org	Bugzilla	3.0	11607	92
bugs.webkit.org	Bugzilla	unknown	9425	unknown
bugs.winehq.com	Bugzilla	3.0	9943	108
bugs.wireshark.org/bugzilla	Bugzilla	unknown	1873	15
bugzilla.xensource.com/bugzilla	Bugzilla	2.20.1	1081	unknown
bugzilla.xfce.org	Bugzilla	unknown	3594	11
bugzilla.handhelds.org	Bugzilla	2.22.2	1846	0
bugtrack.alsa-project.org/alsa-bug	Mantis	1.0.6	3430	22
bugs.digium.com	Mantis	unknown	10765	63
bugs.centos.org	Mantis	unknown	1879	12
bugs.scribus.net	Mantis	1.0.7	6142	24
trac.edgewall.org	Trac	unknown	5948	unknown
bugs.icu-project.org/trac	Trac	0.10.4	5845	unknown
code.djangoproject.com	Trac	0.10.3	5610	unknown
dev.rubyonrails.org	Trac	0.10.5dev	11493+	unknown
trac.cakephp.org	Trac	0.10.4	3403	unknown
madwifi.org	Trac	0.10-mr-r130	1578	unknown
cvs.mythtv.org/trac	Trac	0.11dev-r6048	4078+	unknown
dev.laptop.org	Trac	0.11dev-r5992	4461	unknown
dev.plone.org/plone	Trac	0.10.4	7259	unknown
twistedmatrix.com/trac	Trac	unknown	2820	unknown
haiku-os.org/development	Trac	0.10.3dev	1596	unknown

Table 4: Overview of some popular bug tracking sites (as of October 2007), continued on the next page.

Site	Custom	XML-RPC	RSS	Deps	Login
bugs.debian.org	N/A	no	no	no	no
bugzilla.redhat.com	light	yes	yes	yes	no
qa.mandriva.com	none	yes	yes	yes	no
bugs.gentoo.org	none	no	yes	yes	no
bugzilla.novell.com	heavy	yes	yes	no	no
bugzilla.mozilla.org	none	yes	yes	yes	no
bugzilla.kernel.org	none	no	yes	yes	no
bugzilla.gnome.org	none	no	yes	no	no
bugs.kde.org	light	no	no	no	no
issues.apache.org/bugzilla	none	no	no	yes	no
www.openoffice.org/issues/query.cgi	heavy	no	no	yes	no
bugs.eclipse.org/bugs	heavy	yes	yes	yes	no
bugs.freedesktop.org	none	no	yes	yes	no
gcc.gnu.org/bugzilla	none	no	yes	yes	no
qa.netbeans.org/issues	heavy	no	no	yes	no
bugzilla.mindrot.org	none	no	yes	yes	no
bugs.proftpd.org	none	no	yes	yes	no
www.reactos.org/bugzilla	light	no	yes	yes	no
bugzilla.samba.org	light	no	no	yes	no
www.squid-cache.org/bugs	none	no	yes	yes	no
www.w3.org/Bugs/Public	none	no	yes	yes	no
bugzilla.wikipedia.org	light	no	yes	yes	no
bugs.webkit.org	none	no	yes	yes	no
bugs.winehq.com	light	yes	yes	yes	no
bugs.wireshark.org/bugzilla	none	no	yes	yes	no
bugzilla.xensource.com/bugzilla	none	no	yes	yes	no
bugzilla.xfce.org	none	no	yes	yes	no
bugzilla.handhelds.org	none	no	yes	yes	no
bugtrack.alsa-project.org/alsa-bug	none	no	no	yes	no
bugs.digium.com	none	no	yes	yes	no
bugs.centos.org	none	no	yes	yes	no
bugs.scribus.net	none	no	yes	yes	no
trac.edgewall.org	none	no	yes	no	no
bugs.icu-project.org/trac	none	no	yes	no	no
code.djangoproject.com	none	no	no	no	no
dev.rubyonrails.org	none	no	yes	no	no
trac.cakephp.org	none	no	yes	no	no
madwifi.org	none	no	yes	no	no
cvs.mythtv.org/trac	none	no	yes	no	no
dev.laptop.org	none	no	yes	no	no
dev.plone.org/plone	none	no	yes	no	no
twistedmatrix.com/trac	light	no	yes	no	no
haiku-os.org/development	none	no	no	no	no

Table 5: Overview of some popular bug tracking sites (as of October 2007), continued from the previous page. "Deps" denotes "Dependencies".