First Year Report

Russ Ross <rgr22@cl.cam.ac.uk>
University of Cambridge Computer Laboratory

31 January 2004

1 Introduction

Over the past year I have explored several different areas of research. My explorations have been motivated by my interests in systems research, functional programming languages and advanced type systems, and also by my experience while working in an industrial setting prior to arriving at Cambridge.

From the latter I brought a healthy dose of frustration at the current state of database systems, which led me to propose a main memory database system tightly coupled with an advanced functional language such as OCaml or Haskell. Such a database would offer a data structure view of the content (as opposed to the relational view of RDBMSs) with higher order functionals and perhaps list comprehensions as the primary query methods. These constructs would still allow some query optimizations by the system, but they would also give users a greater deal of control to consider application-level logic at the query level, possibly even overriding standard ACID transaction semantics when appropriate. This project coupled my interest in systems research with my interest in functional languages, and while I did background reading and some design sketches, I did not pursue it to the stage of a full proposal.

My prior job experience[1] gave me extensive background in translation tools and particularly in translation memory, a tool used by translators to reduce redundant translation work and promote consistency by providing hints based on previous translation work done by the same translator or others on the same project. At the time, commercial realities prevented me from pursuing some of the more interesting research topics that could have aided my work in this field (the main-memory database system being one of them). One of the most important problems for translation memory is fuzzy string matching (also called approximate string matching), and I have taken some time to consider a new approach to this classic problem, which I believe offers some advantages over existing approaches for both natural language processing and computational biology, the two main application areas that use fuzzy matching. I give an overview of that approach here.

I also did some minor work on the Xen[1] hypervisor, part of the XenoServers[6] project, but I decided to quit working on Xen in favor of other aspects of the XenoServers project; this is part of what eventually led me to my thesis proposal in the area of distributed filesystems[14]. More directly related to my thesis proposal is a userspace NFS version 3 server that I wrote in Objective CAML (OCaml)[2] to allow copy-on-write semantics at the file level. I describe that project in more detail in this document.

OCaml is an advanced functional language in the ML family with a high performance native compiler and good interoperability with C and other languages as well as the Unix API, making it a better candidate for systems work than most functional languages. I spent some time learning it and used it for the NFS server implementation that I describe later in this report. As part of that learning process, I implemented a solution to the programming contest for the International Lisp Conference 2003[7] in OCaml. Afterward, I implemented my solution in Common Lisp in order to enter the contest and won the prizes[8] for “Fastest Entry” and “Most Elegant Entry.” After learning OCaml I learned Standard ML in order to supervise undergraduates in the Part IA ML course, where I also led course-wide review sessions.

I selected two of the more interesting topics to describe in more detail, the first for its novelty and the second for its relevance to my ongoing research in the area of distributed filesystems[14]. The first is fuzzy string matching and the second is my NFS server.

[1] I worked for Idiom Technologies, Inc. for three years. See `http://www.idiominc.com/`

[2] OCaml information is available at http://www.ocaml.org/

2 Fuzzy string matching

Approximate string matching is an important problem in information retrieval and in computational biology[10]. In many applications defining a similarity metric provides greater flexibility than relying on more strict exact-match semantics. I investigated a new approach to this problem and discuss it here in the context of the most common existing methods.

Two metrics dominate approximate match scoring schemes. The Hamming distance measures the number of characters that differ when comparing two strings character-by-character without considering inserts or deletes. The Levenstein distance is based on an edit transcript that minimizes the number of insert, delete, and modify operations necessary to transform one string into another. Both schemes can be visualized by following one string from beginning to end and asking what needs to happen at each position to produce the other string. The Hamming distance allows only *keep* and *substitute* operations in its edit transcript, while the Levenstein distance allows *keep*, *substitute*, *insert*, and *delete* operations. The transcripts are much like the output of the Unix `diff` utility, and a numerical score can be computed by assigning a penalty score to each operation and comparing that to the total lengths of the two strings.

2.1 Dynamic programming algorithms

The Levenstein distance can be computed using the classic string-to-string correction algorithm[16]. This algorithm finds the minimal edit distance to transform one string into another, where the expense of each step is based on the type of transformation required (delete, insert, substitute, or keep). It uses dynamic programming to find the cheapest path through the lattice of all possible edit transcripts and runs in $O(nm)$ time where n and m are the lengths of the two strings. Variations of the algorithm put restrictions on the number of local insert and delete operations allowed (effectively restricting how far a character can move and still be considered the same character) to lower the runtime cost.

The Levenstein distance provides a simple scoring metric that often produces intuitive results, but at times it can be too narrow in its scope. Consider these examples:

1. `aaaaabbbbb` → `aaaaaaaaaa`
2. `ababababab` → `aaaaaaaaaa`
3. `aaaaabbbbb` → `bbbbbaaaaa`

Using Levenstein distance to compare these transformations yields the same result in each case. 5 characters have to be deleted, 5 inserted, and 5 remain the same. The contiguity of the operations in (1) is not considered, and the transposition that happens in (3) is similarly ignored. In natural language processing and computational biology (two of the main application domains for fuzzy matching) both properties can be significant. Consider these two strings of words:

> before locking the door on your way out make sure you have your keys with you
>
> make sure you have your keys with you before locking the door on your way out

Using simple scoring functions based on the Levenstein distance will yield a match score of around 1/2 or 1/3 (similar to example 3 above), though a human observer would typically assign a higher score when asked to assess the similarity of these two strings.

2.2 q-gram algorithms

The Hamming distance is more restrictive in the transformations it considers, but it lends itself to faster search algorithms based on q-grams. In a typical algorithm, a string is described by the set of every sequence of q (typically 3-6) adjacent letters/words. As an example, the string "simple⋆test" might be described as the following set of 3-grams (also called shingles):

> { sim, imp, mpl, ple, le⋆, e⋆t, ⋆te, tes, est }

Two strings are compared by testing how many q-grams they have in common. Many variations of the algorithm have been researched depending on the relative sizes of the two strings and whether one or more has a pre-computed index[11]. Some minimum number of matching shingles can be computed to guarantee that no matches with a given Hamming distance will be overlooked. Variations have been explored to allow the more flexible Levenstein distance metric[4, 3, 9] but they only work on restricted versions of the problem with limitations on the inserts and deletes allowed.

q-gram approaches can use indexing and other optimizations to allow for quick searches, unlike the dynamic programming algorithm and its variants which act directly on the entire strings being compared. They are primarily a search tool, however, and do not directly yield a score, so a second comparison that measures the Hamming distance or the Levenstein distance is normally required when a similarity score is desirable.

2.3 Set Resemblance

A new approach that I investigated for approximate string matching is based on defining some mapping of strings to sets and then computing a score based on the set resemblance[2]. The set resemblance of two sets α and β is defined as $\frac{|\alpha|\cap|\beta|}{|\alpha|\cup|\beta|}$ giving a number in the range $[0, 1]$.

Using set resemblance as the base metric leaves a great deal of flexibility for defining the set members. One possibility that I propose uses weighted sets. We create our set as the union of all q-gram sets such that $q = 2^n$ for every whole number n. Each member is given weight $\frac{1}{2^{q-1}}$. For the string "cambridge" we get the set members:

weight 1: c, a, m, b, r, i, d, g, e

weight .5: ca, am, mb, br, ri, id, dg, ge

weight .25: camb, ambr, mbri, brid, ridg, idge

weight .125: cambridg, ambridge

for a total weighted size of 14.75. Of this size, 9 (61%) comes from letters in the string, and 5.75 (39%) comes from the ordering and grouping of the letters. As string size increases, more weight is gradually given to the ordering of longer and longer sequences, approaching 50% in the limit. Around 50% of the score comes from having the right letters, and the remaining 50% from having them in the right order. For problems with small alphabets or very long strings, it may be useful to drop all elements of weight 1 or even those of weight .5. In long DNA sequences where the alphabet size is 4, the presence of short sequences is often uninformative as all short sequences will occur frequently.

For some problem domains, redefining the strings proves useful. For many natural language applications, using a word (or the stem of the word) instead of a letter gives better results. Combinations of words and letter sequences (appropriately weighted) may also yield useful results for some problems.

This approach promises to unify the fast search capabilities of q-grams with the scoring abilities of the dynamic programming approaches, while handling cases such as transposition of substrings and contiguous vs. discontiguous matching regions more gracefully than either. In the example given earlier of transposed English text, the set resemblance using words (not letters) is 0.855. There is no clear "correct" score for this example, but 0.855 is a closer match to intuition than a score based on Levenstein distance or Hamming distance. The scores for the three numbered examples in that same section are 0.291, 0.177, and 0.822 respectively.

Defining the problem in terms of sets gives many opportunities for search optimizations, and even for approximations across a network when the exact sets aren't available[5]. Suffix trees and other optimizations that have been applied to q-gram searches[11] can be adapted to searches based on set resemblance with the search result giving an actual resemblance score instead of an approximation that must be refined later. The approach is also simple and accounts for inserts, deletes, transpositions, and substitutions in an easy to understand way.

The lineage of most approximate string matching research traces back to the Wagner and Fischer paper[16], but this approach seem to have more in common with a later paper[15] that gives both better performance and more compact edit scripts, i.e., shorter diffs, by allowing block moves.

3 Userspace Copy-on-write NFS server

I developed a userspace NFS3[12] server that allows users to dynamically stack multiple real directories to appear as a single directory with file level copy-on-write, features that facilitate the creation of customized views of shared filesystems. The idea is reminiscent of the filesystem in Plan 9[13], but everything is carried out in userspace through namespace transformations.

The motivating application for this server is an environment where multiple domains run on a single machine under the Xen hypervisor[1]. Multiple virtual domains running on a single machine will typically share a lot of file data (such as the OS and many application files), so a copy-on-write filesystem offers a convenient way of sharing a pre-configured system, while allowing each participant to retain a private view of the filesystem (all changes are made to copies of the original files) and easily identify the changes that have been made.

3.1 Basic usage and semantics

The server is primarily concerned with translating between two namespaces according to rules defined by the users of the filesystem. We call the original namespace α and the namespace exported to the client β. By default, the same hierarchy maintained by α is exported (with the root of β starting at the mount point) with the same access permissions. At any point in the namespace tree, the β namespace can be remapped to one or more points in the α space with read-only or read-write permissions.

Any directory can include a file with the special name `.~mount`, which contains a list of α directory names that will be stacked at the current point in the β namespace. Each directory can be included read-only or with read-write permissions. In the case where multiple α directories are mapped to a single β directory (the current α directory is normally implicitly included), the ordering of the directories is strictly maintained. When a client requests a file read operation, the list of α directories is searched in order and the first occurrence of the requested file is returned. In this way, an instance of a file that precedes another in the list of directories masks the second occurrence. When the request is for a write operation, the same process is followed if the first instance of the file found is in a read-write mounted directory. If it is not, the system scans the directories in order for the first directory that is writable according to the `.~mount` file rules. If this location will mask the old file, then the file is copied into its new masking position and the write request is serviced. Otherwise, the write operation fails with a permission error.

Stacking rules are inherited by subdirectories, so a user can create a directory tree sparsely populated with file updates to a source code tree, for example, and then stack the source tree behind it through a single `.~mount` file directive, effectively patching the source tree in the β view.

Recursive stacking of mount rules is also permitted (with detection of cycles), allowing one user's copy-on-write view of a base directory to form the base for another user. Consider this example:

- User `rgr22` has a directory called `/mnt/cow/rgr22/foo` which contains some source code.
- User `nlm24` creates a `.~mount` file in `/mnt/cow/nlm24/bar` populated with the single line *read "/mnt/cow/rgr22/foo"*, indicating that `foo` is to be stacked with `bar` with a lower priority, i.e., files in `bar` will mask those in `foo`.
- `rgr22` runs "`make`" in `foo` producing a series of object files; these object files are visible to `nlm24` in `bar`
- `nlm24` makes a change to a source file in `bar` and runs `make`; as updates are made to files they are copied automatically to `bar` and `foo` remains unchanged. The updated files in `bar` mask those in `foo` so to `nlm24` it appears that the files have been updated directly
- User `rgr22` creates a new directory called `/mnt/cow/rgr22/goo` with a `.~mount` file containing the line *read "/mnt/cow/nlm24/bar"*; `goo` includes the unchanged files in `foo` and the updated files in `bar` (with the latter masking the former in the case of conflicts). Note that in the absence of a `.~mount` file in `bar` this same effect could be created by inserting the line *read "/mnt/cow/rgr22/foo"* before the existing line in the `.~mount` file in `goo`.

- `rgr22` runs "`make clean`" in `goo` yielding a directory with no object files but with the source files as changed by `nlm24` in `bar`.

As the "`make clean`" operation in `goo` demonstrates, the server can hide files in response to a remove request that cannot be serviced directly due to access restrictions. A file is hidden if, while scanning the ordered list of stacked directories, the system finds a file with the same name but a prefix of `.~~` before finding a matching file, so a file `foo.o` in the `foo` directory will not be visible in `goo` if either `bar` or `goo` contains a file called `.~~foo.o`. This is essential not only to support direct file removals, but also to support common update patterns. Many text editors do not overwrite a file directly, but instead create a new file with the update contents, remove the original file, and rename the newly created file. If any of these steps fails, the operation is reported to the user as a failure.

Metadata operations present some additional challenges to maintain familiar semantics. A request to remove a file may be serviced by actually attempting to remove it (and possibly failing according to normal file permissions) or it may be serviced by the creation of a `.~~` mask file, possibly masking a file that the user wouldn't normally have permission to remove. Renaming a file may require copying the file and masking out the old file or it may work as a simple rename in the α name space. Metadata updates such as changing the time stamp of a file often require copying the whole file to satisfy access constraints.

Renaming a directory (when permissions do not allow the operation to proceed directly) is more complicated since copying all contained files is both impractical and incorrect: changes to files in the original α directory would not be reflected in the copy. In addition, the user's β view of the directory may include several other α directories (possibly found through recursive `.~mount` file traversals). The solution (not yet implemented) is to create a new `.~mount` file in the new directory with the appropriate rules to include both the original α directory and any other included directories that were visible in the original β directory. In the most complex case, a writable α directory may be renamed directly, but a new `.~mount` file will still be required to capture other stacked directories visible in the old β directory, and a `.~~` mask file must also be created to hide the old view of those same stacked directories under their old β name.

Removing a directory is also slightly more complicated than normal. If the directory is empty (or contains only `.~~` prefixed files) it can be removed directly. If a directory was removed directly or if the β-visible directory is not writable, a `.~~` mask file may need to be created to mask the remaining directories that are visible with that β name. A few anomalies will still remain: "`rmdir directory`" on a non-empty directory may succeed in some cases (when the directory is read-only but a `.~~` file can be created to mask it) but not in other cases (when the directory is writable so the normal semantics apply and the directory must first be emptied). These exceptions are currently only partially implemented.

3.2 Implementation details

One of the objectives of this system was to make it as transparent as possible. If a writable α directory is layered on a read-only α directory, all changes made in the β directory are readily visible in the writable α directory. No additional metadata is kept anywhere. The full stacking arrangement can be inferred from the directory structures and the contents of `.~mount` files.

Two directory structures that are stacked together may not have the same tree structures. The server internally tracks absent writable directories that should mirror read-only directories, labeling them "shadow directories." When a write request is issued to a file in a read-only α directory, multiple levels of shadow directories may be created as real directories to make the copy-on-write operation succeed. This makes it convenient to mount a read-only directory tree into a private branch of the β hierarchy and make changes to it. By removing the mount, a sparse directory structure containing only modified files is left behind which can be easily moved or re-applied as a patch through another mount operation.

The server is implemented in OCaml which has proved to be well suited to the task. Strong typing and type inference have been particularly helpful in implementing the details of the NFS3 protocol; the type verifier catches most errors in populating structures and result codes at compile time. The bulk of the code is devoted to operations on trees to translate between α and β namespaces, a task for which a functional language like OCaml is ideal.

Several implementation tasks remain, particularly around namespace cache management, so the server is not yet suitable for general use, but the implementation is complete enough to use for testing purposes. I intend to complete the implementation for use in the early roll-out of the XenoServers project, while my future research into distributed filesystems will provide a possible direction for later stages of the project.

4 Conclusion

I have explored several varied topics during my first year. Some were the result of my time in the industry while others were more closely related to larger projects going on in the computer laboratory. I've found it exciting and rewarding and I look forward to the rest of my time here as I continue with my PhD research.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugabauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *ACM SOSP*, October 2003.

[2] A. Z. Broder. On the resemblance and containment of documents. *Proceedings of Compression and Complexity of SEQUENCES*, 1997.

[3] S. Burkhardt and J. Karkkainen. One-gapped q-Gram Filters for Levenstein Distance. *Proc. 13th Symposium on Combinatorial Pattern Matching (CPM '02)*, Spring 2002.

[4] S. Burkhardt and J. Karkkainen. Better Filtering with Gapped q-Grams, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, number 2089*, 2003.

[5] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast Approximate Reconciliation of Set Differences. BU Computer Science Tech Report 2002-019, 2002.

[6] K. Fraser, S. Hand, T. Harris, I. Leslie, and I. Pratt. The Xenoserver Computing Infrastructure. Tech Rep. UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, January 2003.

[7] Nick Levine. ILC 2003 Programming Contest. http://www.ravenbrook.com/doc/2003/05/28/contest/, 2003.

[8] Nick Levine and Nicholas Barnes. ILC 2003 Programming Contest Judges' Report. http://www.ravenbrook.com/doc/2003/05/28/contest/judges-report.txt, 2003.

[9] R. Muth and U. Manber. Approximate multiple string search. *Proc. CPM'96*, pages 75–86, 1996.

[10] E. W. Myers. An overview of sequence comparison algorithms in molecular biology. Technical Report TR 91-29, Department of Computer Science, University of Arizona, Dec. 1991.

[11] G. Navarro. Indexing methods for approximate string matching. Technical report, University of Chile, 2000.

[12] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. NFS Version 3 Design and Implementation. *Proceedings of the Summer USENIX Conference*, pages 137–152, June 1994.

[13] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *Proceedings of the 5th ACM SIGOPS European Workshop*, pages 72–76, Mont Saint-Michel, 1992.

[14] Russ Ross. Thesis Proposal, January 2004.

[15] W. F. Tichy, The string-to-string correction problem with block moves. *ACM Trans. Computer Systems, 2, (4)*, pages 309–321, 1984.

[16] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM 21*, pages 168–173, 1974.