Using TDB in Greenstone to Support Scalable Digital Libraries

John Thompson[†], David Bainbridge[†], and Hussein Suleman[‡]

[†]Department of Computer Science,
University of Waikato,
Hamilton, New Zealand
jmt12@students.waikato.ac.nz, davidb@cs.waikato.ac.nz

[‡]Department of Computer Science,
University of Cape Town,
Cape Town, South Africa
davidb@cs.waikato.ac.nz

Abstract. This paper reports on performance improvements in the open source Greenstone digital library software that resulted from a more detailed understanding of the demands made of its database component, when building large collections. The work was undertaken as part of a larger drive to support parallel processing during the ingest of Very Large Digital Library (VLDL) collections using this software. In terms of a database requirement, Greenstone is set apart from many other digital library solutions, by default using only a flat-file database component to operate, as opposed to a full-blown relational database. However, despite the simplicity of this type of database, our review of the literature revealed that little is known about how this type of database performs in a digital library context when a high volume of data is processed. Through the work presented here we make some inroads that address this imbalance; we also show how utilizing a transaction-based flat-file database (that supports parallel reader/writer access) dramatically improves performance not only in parallel processing, but in the current non-parallel process as well.

Keywords: Greenstone, Trivial/Transaction Database (TDB), Parallel Processing, VLDLs

1 Introduction

The creation and practical use of Very Large Digital Libraries (VLDLs) has raised many new issues, particularly related to size—aspects such as disk space, number of documents, and number of unique terms—sustainability, and interoperability (e.g., Mangi et al.[4]). But technology has also advanced and, with multi-processor/hyper-threading now the norm in consumer computers, explicitly developing techniques for digital library software that exploits parallel processing capabilities looks promising as a solution to some of these scale concerns.

For example, Stanfill [5] has proposed a number of ways to leverage the power of parallel processing that is applicable to the information retrieval component of a digital library system, and that can be applied to indices in the order of terabytes.

It is perhaps unsurprising then that the initial ingest of collections of this size is also time-consuming. In our experience real-world VLDLs may take in excess of several days of processing to create. The HathiTrust Digital Library, for example, found that a carefully optimized import of their seven million documents took ten days.[1] In previous published test results [1], we showed that importing time in the Greenstone Digital Library software is roughly linear with the number of documents.

This paper looks at the efforts of refactoring Greenstone's ingest process to utilize parallel processing as a solution to the issue of growing processing times and, as part of this work, the replacement of Greenstone's default database (GDBM[2]) with one that supports parallel access (TDB[3]). The structure of the paper is as follows. We start by detailing the initial work on parallel processing in Greenstone and explain how this led to utilizing TDB as a replacement database. We then present testing results that show the advantages of using TDB for large scale collection building, not only when applied to parallel processing, but in serial ingests too. We conclude with a discussion of the implications of these findings and future work.

2 Method

As mentioned above, exploration in the underlying database utilized in Greenstone was initiated as part of a larger theme of work to augment the software with parallel processing capability. While Greenstone already had several features that could be quickly leveraged to segment the import process into batches (each of which could be processed independently), it needed some refactoring and additional support to create and manage multiple processes. This support was provided by the Open MPI library [2], a mature open-source Message Passing Interface (MPI) project built with a focus on high performance and supporting a wide variety of operating systems and hardware configurations (for instance multi-core and clustered).

While the majority of the Greenstone importing process was found to work well when applied to multiple processes, the interaction between Greenstone and the default database backend (GDBM) it uses to store processing information did not, as the database system only allows a single process to write to it. This limitation had not seemed an issue with normal serial processing, but now required the addition of locking around database writes in order to prevent

[1] `http://www.hathitrust.org/blogs/large-scale-search/forty-days-and-forty-nights-re-indexing-7-million-books-part-1`

[2] `http://www.gnu.org/software/gdbm/`

[3] Short for Trivial Database, developed by the Samba project. For more details see `http://tdb.samba.org/`

errors in the database and abnormal program terminations. This locking was implemented by way of calls to $flock()$—which in early tests did not seem to add much overhead to the interactions with GDBM.

Initial test results, however, did not exhibit the performance gains expected, with parallel processing on eight CPUs barely halving the ingest time. We profiled[4] the importing code and tracked the issue down to output IO and in particular the opening and closing of connections to the GDBM database (often occuring several times per document). In order to minimize the amount of time any particular writer has an exclusive lock over the database, the connection to the GDBM was only opened momentarily for each write and then immediately closed. Structuring the GDBM connections this way was also necessary as, even in serial Greenstone, there may be several different, independent, parts of the import process that read and/or write to the database during ingest. This finding also highlights the impact that delineated critical sections[5] of code (effectively requiring serial execution) can have on a parallel process.

After reviewing other database alternatives, the decision was made to replace the GDBM database with one that better supported parallel processes. Trivial Database (TDB) was developed for use in the Samba project[6] as a means to share data between multiple processes. TDB was specifically designed to allow multiple writers at once. It does so by dividing the content of the database into a number of 'buckets' and then performing two different flavors of locking (readonly and write locks) as well as deferred updates to ensure consistency within a transaction.[7] Crucially, as TDB supports multiple writers across one or more threads, we were able to open a single connection (per thread) to the database right at the start of collection ingest and then have it persist through to ingest completion.

At this stage we were able to rerun our parallel tests with TDB as the database, and we were pleased to see a significant increase in performance (See Figure 1 for an overview). The parallel tests also showed an even more significant increase in the performance of the *serial* Greenstone ingest (denoted on the figure as having zero threads). Following up on this unexpected result we ran another series of tests to measure the performance of TDB compared with all of the other database types currently supported by Greenstone—namely: GDBM, SQLite and the new GDBM with locking. We analyse these results, and the original parallel tests in more detail in the next section.

Most of the tests presented here were carried out on an Intel Core i7 with eight cores (hyper-threaded from four physical cores), 4GB of RAM and a single 2TB harddrive (7200RPM, 138Mb/s transfer rate). When testing parallel processing performance we used a collection of 10,000 documents ingested over a variety of Greenstone configurations, number of worker threads and batch sizes, while the later serial tests instead used a number of collections ranging from

[4] `http://code.google.com/p/perl-devel-nytprof/`
[5] `http://en.wikipedia.org/wiki/Critical_section`
[6] `http://www.samba.org/`
[7] `http://en.wikipedia.org/wiki/Database_transaction`

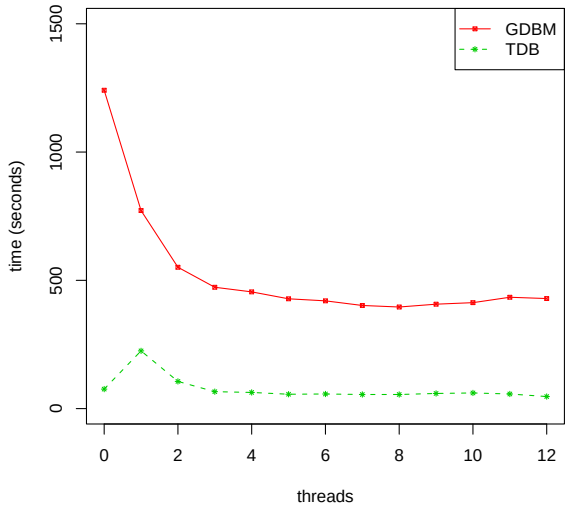

Fig. 1. Comparison of import times between GDBM with TDB over varying numbers of process threads.

100 to 50,000 documents in size. In all cases the plain text documents contained computer generated lorem ipsum text.[8] The results for each test captured the real, user and system time of importing, averaged over multiple executions. The output materials from the tests were compared where possible and, aside from acceptable differences in timestamps, they were consistent.

3 Results

We start by showing an example result from the testing carried out to determine the potential benefits of parallel processing in Greenstone. Figure 1 illustrates the importing time (real time in seconds) of 10,000 documents, comparing GDBM with simple file locking versus TDB and while increasing the number of processing threads. When the number of threads is zero then ingest was undertaken in a serial manner. The graph shows that the GDBM process benefits from increasing the number of threads until around 8–9 threads. On the surface this agrees with current literature suggesting the optimal thread count for compute-bound load occurs at P or $P + 1$, where P is the number of processors on the computer [3]. But parallel processing had, at best, only halved processing time, where the predicted speed increases on compute-bound loads should be closer to linear (accepting some cost of parallel processing overhead) with the number of

[8] `http://www.lipsum.com/`

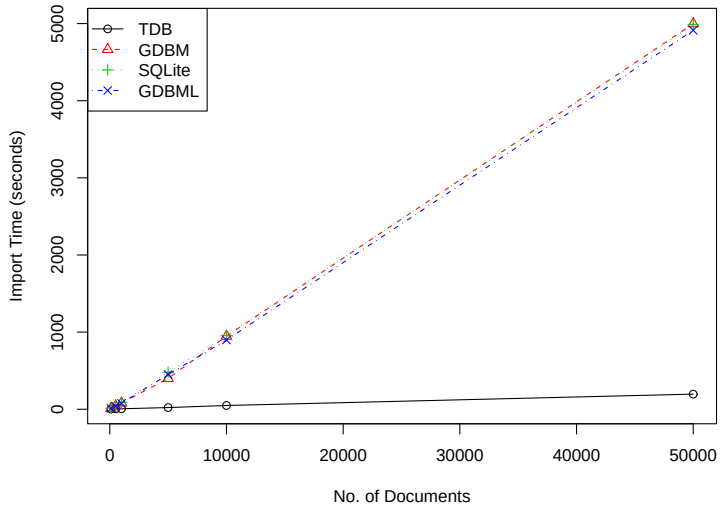

Fig. 2. Ingest times (real time) of various databases as number of documents increase.

threads (up to the optimal thread count). This implied that the ingest process was not compute-bound, and subsequent profiling proved this, showing IO was the limiting factor.

In exploring ways to minimize the bottleneck occurring around IO, TDB was tested as an alternative database back-end. As can be seen from the graph the improvement was immediate. The worst performance of TDB where threads equals one—which means you get all the overhead of parallel processing but none of the benefits—was still twice as fast as the best performance of GDBM with locking. However, what surprised us was the gain in performance in the serial case, where TDB performance was almost $\times 10$ better than the standard Greenstone import process. The graph shows that little benefit was actually gained by applying more threads to a TDB import, with parallel processing only shaving a few percent off the processing time. This warranted further study.

Figure 2 shows a comparison of serial ingest times (real time in seconds) of several of the back-end databases available to Greenstone against the new TDB database. GDBM and SQLite are standard in Greenstone, while GDBML represents the version of GDBM with simple file locking, developed for parallel processing. All three show similar performance and exhibit linear growth. While TDB also shows linear growth, its performance as the number of documents increases is a magnitude better than the alternatives (approaching 1/30th of the processing time of plain GDBM).

Typically the times for multiple runs of any particular test using GDBM, SQLite and GDBML databases were all within a few seconds of each other.

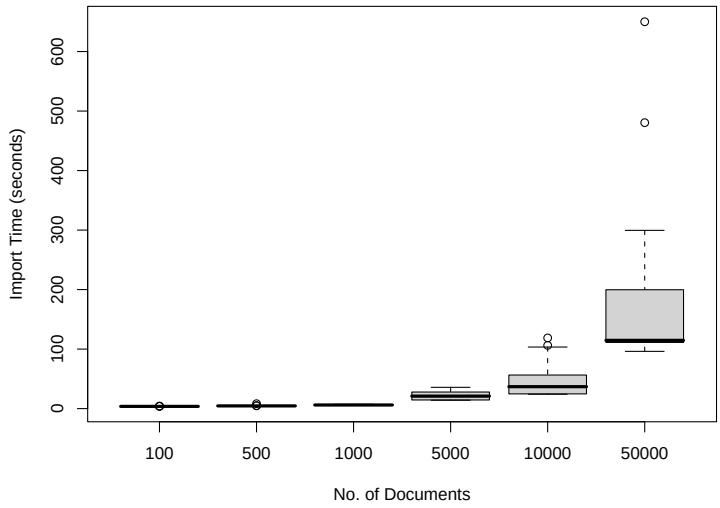

Fig. 3. Detail of the Ingest times (real time) of TDB as number of documents increase.

TDB, however, exhibited a highly varying range of process times—with some runs taking 5–6 times longer than others. This detail was smoothed away in Figure 2 but Figure 3 shows a more detailed breakdown of the TDB runs. The magnitude and frequency of these abnormally long test runs appear proportional to the number of documents processed, as shown by the escalating outliers in the box and whisker chart.

To shed further light on the greater variation in TDB processing time we compared the real vs. system times of fifteen of the test runs for each number of documents. These results are shown in Figure 4, with the total height of each stacked bar representing real time and the lighter shaded component at the bottom representing system time. The majority of abnormal runs showed the extra time occurring in system time—indicative of blocking on locks or other system level factors. When the ingest process was broken down, step-by-step, and tested it was found that certain IO intensive operating commands, such as `rm` (used to remove a previously imported collection), were just as likely to have abnormal running times.

Subsequent testing on several computer exhibiting a range of operating systems and file systems (shown in Table 1), seems to point to an intermittent system level disk journaling/caching activity that is exacerbated by use of the EXT3 file system, rather than being caused by Greenstone, TDB or the combination thereof.

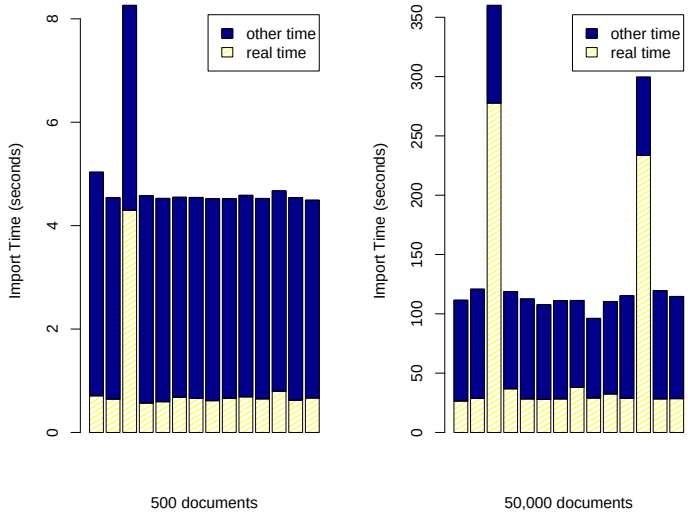

Fig. 4. Breakdown of real time vs system time for two TDB test collections.

Table 1. Real time (in seconds) statistics of 100 test runs ingesting then deleting 5,000 documents using TDB and comparing several operating and file systems

OS	FS	GS Ingest			`rm command`		
		min	max	avg	min	max	avg
CentOS	EXT3	9.050	15.140	9.447	0.200	25.840	3.789
Ubuntu	EXT3	8.170	11.440	8.717	0.140	18.860	2.622
CentOS	EXT4	9.070	9.790	9.305	0.200	0.730	0.226
Ubuntu	EXT4	8.540	9.190	8.701	0.160	0.610	0.213

4 Conclusions and Future Work

The results clearly show significant benefits in using TDB versus GDBM when ingesting, with import times being an order of magnitude faster than traditional importing. That is not to say that GDBM, in itself, offers lower performance—indeed we are currently investigating how to restructure the Greenstone/GDBM connection to ensure that database connections need not be short-lived (and thus could persist for the import process) to see if we might reach similar performance with GDBM. It simply highlights that using a database that allows multiple readers/writers better suits the current architecture and algorithms employed by Greenstone. TDB also provides a strong platform for continuing development of parallel processing; an evolution that has as a requirement the ability to access the database used by Greenstone from multiple, parallel, processes.

The results also demonstrate that there is still some benefit in parallelising the import process in Greenstone and that the development of parallelisation support leads to optimizations that benefit the general architecture as well. Given the increasing proliferation of high-core CPUs (with 12-cores available at the time of writing), multi-processor-capable systems and commodity publicly-available high performance computing in the form of pay-per-use virtual utility computing, it is essential that software like Greenstone and its peer systems plan for parallelisation as their architectures evolve.

While attempting to address an issue identified during parallel processing development on the import phase of Greenstone, we have serendipitously discovered that changing to a database that supports multiple readers and writers can also provide significant benefits for non-parallel Greenstone ingests. By utilizing one such database, TDB, we were able to record processing times a magnitude times faster than traditional Greenstone imports. TDB also offers the opportunity for parallel processing during the import process. Improvements such as these are needed to mitigate the problem of import times linear to numbers of documents, and ensure that the Greenstone Digital Library software is capable of creating very large scale digital libraries in reasonable time.

References

1. Bainbridge, D., Witten, I.H., Boddie, S., Thompson, J.: Stress-testing general purpose digital library software. In: Proceedings of the 13th European conference on Research and advanced technology for digital libraries. pp. 203–214. ECDL'09, Springer-Verlag, Berlin, Heidelberg (2009), http://portal.acm.org/citation.cfm?id=1812799.1812828
2. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting. pp. 97–104. Budapest, Hungary (September 2004)
3. Goetz, B.: Java theory and practice: Thread pools and work queues. Tech. Rep. j-jtp0730, IBM, New York, United States (2002), http://www.ibm.com/developerworks/library/j-jtp0730/index.html
4. Manghi, P., Pagano, P., Ioannidis, Y.: Second Workshop on Very Large Digital Libraries: in conjunction with the European Conference on Digital Libraries. SIGMOD Rec. 38, 46–48 (June 2010), http://doi.acm.org/10.1145/1815948.1815959
5. Stanfill, C.: Partitioned posting files: a parallel inverted file structure for information retrieval. In: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 413–428. SIGIR '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/96749.98247