A Spotlight on Spotlight

Rini Kaushik
rini@cs.wisc.edu
CS736 Project, Fall 2007
University of Wisconsin, Madison

ABSTRACT

In this paper, we have made an attempt to take a look under the hood of Spotlight and understand some of the following questions amongst others:

- How well-integrated is Spotlight with the file system?
- How reliable is the interface between Spotlight and the file system?
- Is live update a myth or a reality under heavy file system activity?

Based on our experiments, it seems that Spotlight is not really tightly integrated with the file system. In reality, it is "bolted" on to the file system. The interface between the file system and Spotlight is not reliable and live updates are not really "live" in case of heavy file system activity.

1 INTRODUCTION

Data is exploding at a very fast pace and capacities of storage devices are increasing. Hence, substantial amount of information is available even on personal computer systems. As a result, there is an imminent need to provide superior search mechanisms to the personal computer system user for searching through the vast amount of information present. Current file systems are inadequate in providing a superior search experience as:
They provide a hierarchical name space only
There is no way to provide custom metadata to the file system which could aid the user in search later on
Most of the search technologies present today such as Beagle [2], Tracker [3], and Lucene [4] etc. are not tightly integrated with the file system.

Spotlight is Apple's search technology for extracting, storing, indexing, and querying metadata and content of the file system [1]. It provides an integrated system-wide service for searching and indexing. It promises to be tightly-integrated with the file system and hence, we decided to look under Spotlight's hood to understand the level of integration with the file system and throw spotlight on Apple's Spotlight.

1.1 Fsevents Infrastructure

Fsevents is an in-kernel notification system for informing user-space subscribers of file system changes. Spotlight receives file system change notifications by subscribing to the fsevents infrastructure. Spotlight relies on this mechanism to keep its information current—it updates a volume's metadata store and content index if file system objects are added, deleted, or modified. Spotlight is the primary subscriber of the fsevents interface. Table 1 lists the various fsevents generated by the fsevents infrastructure.

Event Type	Description
FSE_CREATE_FILE	A file was created
FSE_DELETE	A file was deleted
FSE_STAT_CHANGED	A file's attributes were changed
FSE_RENAME	A file was renamed
FSE_CONTENT_MODIFIED	A file's contents were modified
FSE_CREATE_DIR	A directory was created
FSE_CHOWN	A file's ownership was changed

Table 1. Table of the fsevents generated by the fsevent infrastructure

The kernel exports the mechanism to user space through a pseudo-device (/dev/fsevents). A user-space program interested in learning about file system changes can subscribe to the mechanism by accessing this device. Specifically, a watcher opens /dev/fsevents and clones the resultant descriptor using a special ioctl operation (FSEVENTS_CLONE). A read call on the cloned descriptor blocks until the kernel has file system changes to report. When such a read call returns successfully, the data read contains one or more events, each encapsulated in a kfs_event structure.

The elements of a per-watcher event queue are not the events themselves but pointers to kfs_event structures, which are reference-counted structures that contain the actual event data [6]. In other words, all watchers share a single event buffer in the kernel. There is a global array of event buffers fs_event_buf in the kernel which is limited in size to 2048 elements.

Various functions in the VFS layer call add_fsevent() call to add fsevents to the global array of events fs_event_buf. Table 2 lists the translation between VFS functions and the resulting fsevent type. Figure 1 shows the link between the event pointers in the per-watcher fs_event_watcher structure and the actual events in the in-kernel global fs_event_buf array.

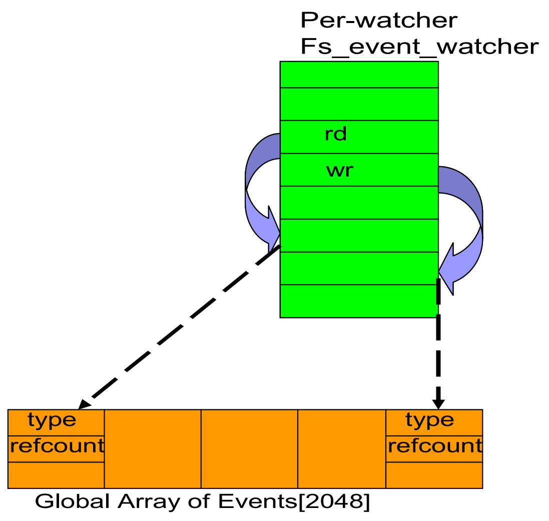

Figure 1. Shows link between per-watcher event queue and the global array of events

VFS Function	Fsevent Type
Symlink(), mknod(), link()	FSE_CREATE_FILE
Vnode_setattr()	FSE_CHOWN, FSE_STAT_CHANGED
Unlink(), rmdir()	FSE_DELETE
Rename()	FSE_RENAME
Vn_close()	FSE_CONTENT_MODIFIED

Table 2. The Table shows the correspondence between the VFS layer call and the resulting fsevent type

1.2 Spotlight Subsystem

The Spotlight server (mds) is the primary daemon in the Spotlight subsystem. It is responsible for receiving change notifications through the fsevents interface, managing the metadata store, and serving Spotlight queries.

Spotlight uses a set of specialized plug-in bundles called metadata importers for extracting metadata from different types of documents, with each importer handling one or more specific document types. Each importer understands the format of a specific document type and also helps in converting the document into a textual form which can later be used by Search Kit in creating a content index of the document. The mdimport program acts as a harness for running these importers. It can also be used to explicitly import metadata from a set of files. An importer returns metadata for a file as a set of key-value pairs, which Spotlight adds to the volume's metadata store. Spotlight provides apis for writing custom metadata importers. A list of metadata importers present by default on the system can be found by executing "mdimport –L". Table 3 lists a subset of file types and the corresponding importer present on the system by default.

File type	Importer
Image	Image.mdimporter
pdf	PDF.mdimporter
Audio	Audio.mdimporter
RTF	RichText.mdimporter
Office files	Microsoft Office.mdimporter

Table 3. The Table lists the file types and the corresponding importers

Mdsync process is used for rescanning the volume and rebuilding the index either after the system is rebooted or after re-indexing is manually initiated.

Spotlight provides several ways for end users and programmers to query files and folders based on several types of metadata: importer-harvested metadata, conventional file system metadata, and file content (in the case of files whose content has been indexed by Spotlight). The Mac OS X user interface integrates Spotlight querying in the menu bar and the Finder. For example, a Spotlight search can be initiated by clicking on the Spotlight icon in the menu bar and typing a search string. Command line tool mdfind can also be used to query the files. In addition, apis are provided by Spotlight to programmatically query the system [9].

A separate index is maintained per-volume. On a volume with Spotlight indexing enabled, the /.Spotlight-V100 directory contains the volume's content index (ContentIndex.db), metadata store (store.db), shadow metadata store (.store.db), and change notification log file (.journalHistoryLog).

The content index is built atop Apple's Search Kit technology [7], which provides a framework for searching and indexing text in multiple languages. The metadata store uses a specially designed database in which each file, along with its metadata attributes, is represented as an MDItem object, which is a Core Foundation–compliant object that encapsulates the metadata.

1.3 Spotlight's Architecture

As a file is created, modified or deleted, the VFS layer puts an fsevent in the global array. Spotlight's mds server which is continuously reading from the fsevent device, reads the fsevent in. It checks the file type of the file for which the fsevent was generated and based on the type of the file, it invokes the corresponding importer via mdimport. Mdimport reads the metadata and the content of the file from the file system and creates <key, value> pairs which it sends back to the mds server. Mds server stores the same in the metadata index (Store.db) and in the content index (ContentIndex.db). Subsequently, a user can submit a query for a metadata or content based query via mdfind, finder or programmatically. These queries will be serviced by the mds server. Mds server will read the index stores and return the results back to the user. Figure 2, puts the Spotlight Architecture together.

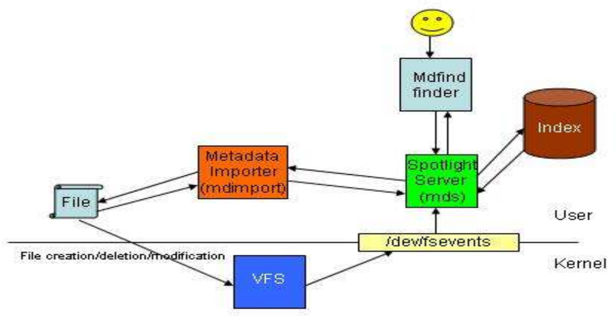

Figure 2. Spotlight architecture

2 SETUP AND METHODOLOGY

This section describes the systems used for the experiments and the experimental methodology.

2.1 SYSTEM

The experiments were done on a MacBook with 2.16 GHz Intel Core 2 Duo and 1 GB 667 MHz DDR2 SDRAM. The operating system version was Mac OS 10.4.10 and the kernel version was Darwin 8.10.2. The hard disk was a Serial-ATA device with a capacity of 111.79 GB and a speed of 1.5 Gigabit. The experiments were done on a 10 GB volume SpotlightTest which was formatted with Journaled HFS+ file system.

2.2 METHODOLOGY

I used a combination of micro benchmarks, system tools and created an fsevent subscriber to evaluate Spotlight.

While conducting the experiments, I used various tools to better understand the system activity. I used fs_usage to report the file system related activity occurring on the system. The file system activity was pertaining to the activity generated by the Spotlight processes such as mds, mdimport, mdsync and mdnsserver. The fs_usage output contains the timestamp, call name, file descriptor, byte count, pathname, offset, the elapsed time spent in the system call and the process name. I also used ktrace which does kernel tracing on a per-process basis to trace the system calls, namei translations and the IO calls made by the Spotlight processes.

To characterize the cpu utilization during indexing, I used top and iostat for characterizing the disk transfers and disk throughput.

To get an accurate number of dropped fsevents, I checked the /var/log/system.log for the number of messages "fs_events: add_event: event queue is full! Dropping events". The add_fsevent() call in vfs_fsevents.c places this error message whenever it is unable to find an empty slot to place a fsevent in the global array.

I used the following command line tools provided by Spotlight:

mdutil is used to manage the Spotlight metadata store for a given volume. In particular, it can enable or disable Spotlight indexing on a volume, including volumes corresponding to disk images and external disks.

mdimport can be used to explicitly trigger importing of file hierarchies into the metadata store. It is also useful for displaying information about the Spotlight system.

- The -A option lists all metadata attributes, along with their localized names and descriptions, known to Spotlight.
- The -X option prints the metadata schema for the built-in UTI types.
- The -L option displays a list of installed metadata importers.

mdcheckschema is used to validate the given schema file—typically one belonging to a metadata importer.

mdfind searches the metadata store given a query string, which can be either a plain string or a raw query expression. Moreover, mdfind can be instructed through its -onlyin option to limit the search to a given directory. If the -live option is specified, mdfind continues running in live-update mode, printing the updated number of files that match the query.

mdls retrieves and displays all metadata attributes for the given file.

To restrict indexing only on the SpotlightTest volume, I used mdutil tool to switch indexing off on the rest of the volumes in the system.

To minimize the impact of fragmentation, I deleted the index using mdutil tool and all the folders under SpotlightTest volume between experiments and also between iterations of the same experiment.

I developed a variant of hfsdebug to check the extent of fragmentation present in the content and metadata index.

I used mdfind tool to query the index.

The main areas which I wanted to analyze in Spotlight were:

- Reliability of the fsevent interface
- Liveness of Index Update
- Performance profile of indexing
- Storage characterization of the index
- System calls characterization of indexing
- Performance impact of Spotlight
- Indexing optimizations such as incremental indexing or single-instancing
- Performance of queries

3 RESULTS

3.1 Reliability of fsevents

In this series of experiment, I reproduced situations in which file system events were dropped by the fsevents infrastructure. Thereafter, I analyzed the effect of the dropped events on the consistency of the index.

I wrote a fsevent watcher (SlowWatcher) and subscribed the same to the fsevent infrastructure, thereby creating two watchers in the system – Spotlight and SlowWatcher. I also wrote a multi-threaded java program in which each thread operated on a separate directory and overwrote all the 18603 files contained in the directory with a unique string per-thread and per-iteration.
I was able to reproduce dropped events in two ways:

- Increasing the slowness of the SlowWatcher
- Increasing the file system activity on the system by increasing the number of threads in the java program

Figure 4 characterizes the effect of the slowness of the watcher on the dropped events. A watcher which is slow in reading the events from the global fsevents queue results in the kernel dropping the fsevents as the kernel is unable to find a free slot to put the new fsevent pertaining to a file system change. Since, the limit of the global buffer is just 2048, the buffer over-run happens rapidly. Figure 4 characterizes the effect of substantial file system activity on fsevent drop. With substantial file system activity, the fsevents get generated at much faster pace than what can be consumed by the subscribers and hence, some fsevents get dropped.

To analyze the effect of dropped events on the consistency of the index, after every iteration, I queried for the unique string which was embedded in the text of the file that iteration. I used mdfind to trigger the query. In the query results, all the files whose change notifications were missed were missing in the query results. Thus, Spotlight seems to be silently ignoring the dropped events. The alternative, which is re-scanning the entire volume to figure out the changes in the file system, is also bad. The time to re-scan would increase substantially if the volume is heavily populated.

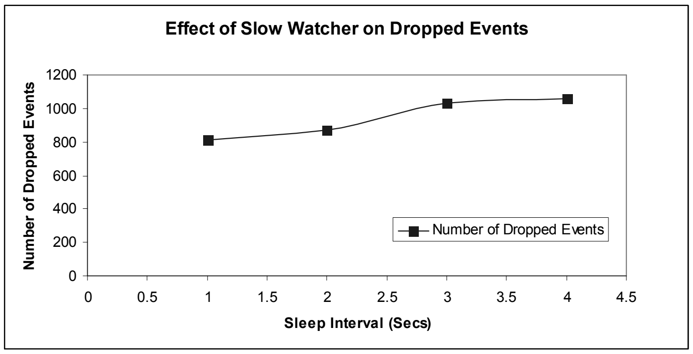

Figure 3. The graph characterizes increase in number of dropped events generated as a result of increasing the slowness of the fsevent watcher. The slowness was created by introducing a wait between ioctls to read fsevents from the /dev/fsevent device by the SlowWatcher

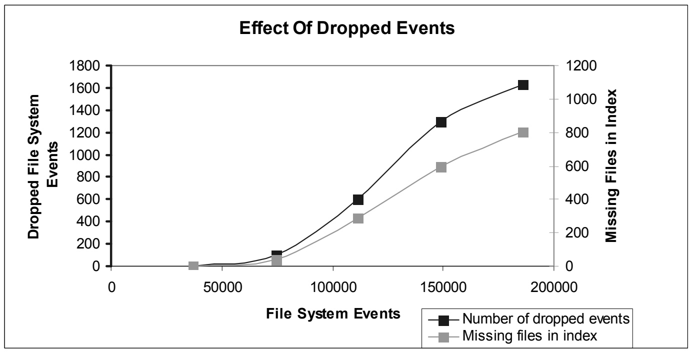

Figure 4. The graph characterizes increase in number of dropped events as file system activity is increased on the system. Primary y-axis characterizes number of dropped events as file system activity is increased and Secondary y-axis characterizes inconsistent files in the index. X-axis characterizes the actual file system events generated as a result of increased file system activity (two per file in the experiment)

3.2 Liveness of Index Updation

Spotlight claims to support live update of index upon file content or attribute change. In this experiment, I wanted to see if the live update still holds true in case of substantial file system activity.

I used two flavors of increasing file system activity – sequential and parallel. The java test program used java runtime to invoke "tar –xvf" on linux-2.6.13.1.tar under different parent directories. A linux-2.6.13.1 folder contains 18603 files and is 236MB in size. In the parallel flavor, configurable number of parallel threads did one untar each. Figure 5 characterizes the increase in the time to index as the size of the data to be indexed increases in parallel. In the sequential flavor, configurable number of untars happened in sequence in a single thread. Figure 6 characterizes the increase in the time to index as the size of the data to be indexed increases sequentially. The time to index was the cumulative time to index all the resultant linux folders. The end time of the indexing was considered to be the time when the last fsync of the ContentIndex.db and Store.db occured.

Based on the experiments, the live update claim doesn't really hold true in case of substantial file system activities. The index update can range from seconds to even hours depending on the file system activity. The indexing is an expensive and time-consuming process. Also, mdimport runs at a low priority and any system activity gets precedence over mdimport. This delays indexing further. Also, the claim that queries can be updated live also doesn't hold true under substantial file system activity as the queries are not answered while indexing is in progress.

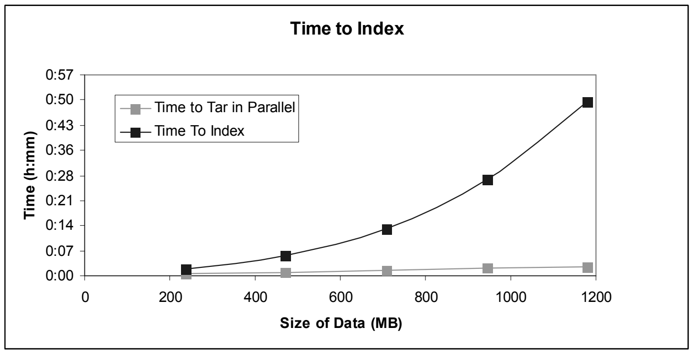

Figure 5. The graph characterizes the increase in the time to index as size of data to be indexed is increased in parallel. For each data size point the graph shows the time taken by the tars to finish and the time to index.

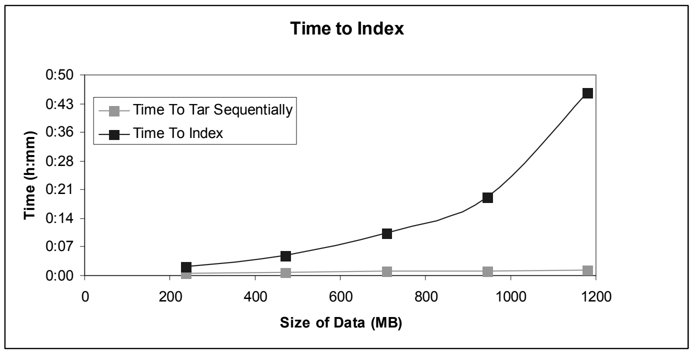

Figure 6. The graph characterizes the increase in the time to index as size of data to be indexed is increased in sequence. For each data size point the graph shows the time taken by the tars to finish and the time to index.

3.3 Performance Profile of Indexing

In this set of experiment, I untarred a 2.6.13.1.tar which resulted in a 236MB folder containing 18603 files. The graphs below contain a snapshot of the ensuing CPU Utilization and the Disk transfers and throughput while indexing is in progress on the files in this folder.

As shown in Figure 7 and Figure 8, the indexing is a CPU and disk intensive process if the number of the files to be indexed is substantial. It seems that importers use unbuffered I/O to bypasses the buffer cache; this way, the buffer cache will not be polluted because of the one-time reads generated by the importer. As a result, number of disk transfers is substantial. Mds and mdimport utilize close to 100% cpu majority of the time during indexing.

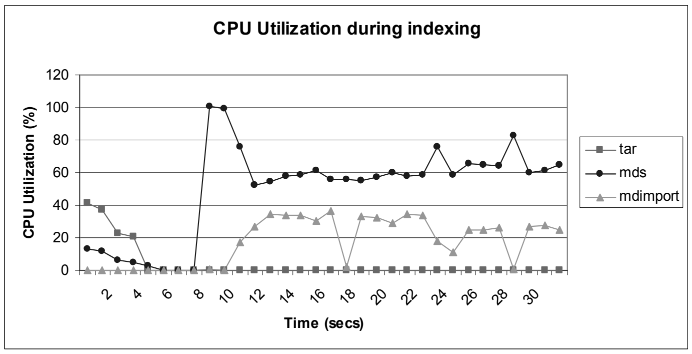

Figure 7. The graph shows a snapshot of CPU Utilization during indexing of a 236MB folder containing 18603 files.

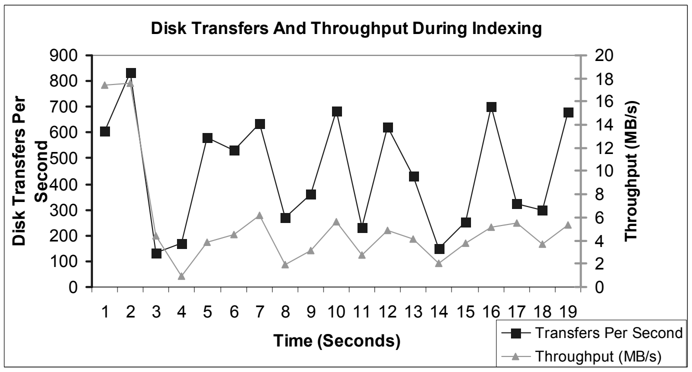

Figure 8. The graph shows a snapshot of disk transfers and throughput during indexing of a 236MB folder containing 18603 files.

3.4 Storage characterization of index

In these set of experiments, I have tried to analyze the storage characterization of the index.

To analyze the space requirement increase of the metadata and the content index, I increased the index able data on the system by increasing number of folders on the system each containing 18603 text files and 236MB in size. Figure 9 characterizes the increase in the size of the content index (ContentIndex.db) and metadata index (Store.db) as a result of the data increase. As can be seen from the figure, the content index is 17% on an average of the data size.

To analyze the reduction in the space requirement of the index as a result of deletion of the file data which was previously indexed, I deleted the folders created above in sequence and plotted the resulting decrease in the content and metadata index. As shown in Figure 10, the content index doesn't decrease in size as a result of the deletion even after days. Only the metadata index reduces in size. The reduction is proportional to the increase in the previous graph.

Figures 11 and 12, characterize the fragmentation introduced in the content and metadata index as a result of the file data increase in the first experiment in this section. Figure 11 characterizes the increase in the number of the extents in the indexes and Figure 12 characterizes the increase in the number of the blocks in the indexes. As seen in the graphs, the indexes get quite fragmented fast. In fact, the indexes are the most fragmented files on the system.

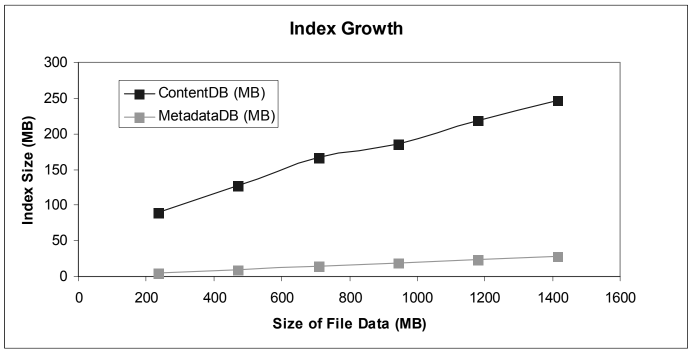

Figure 9. The graph characterizes the increase in the index size as the data is increased in the system. A new folder 236MB in size and containing 16803 files is added at each iteration and y-axis plots the corresponding increase in the content index (ContentIndex.db) and metadata store (Store.db)

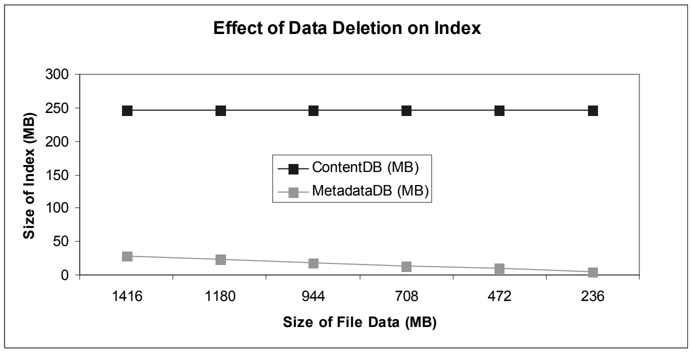

Figure 10. The graph characterizes the decrease in the index size is data is decreased in the system. A folder 236MB in size and containing 18603 files is deleted at each iteration. The corresponding decrease in the metadata index (Store.db) is plotted on the y-axis. There is not decrease in the content index (ContentIndex.db)

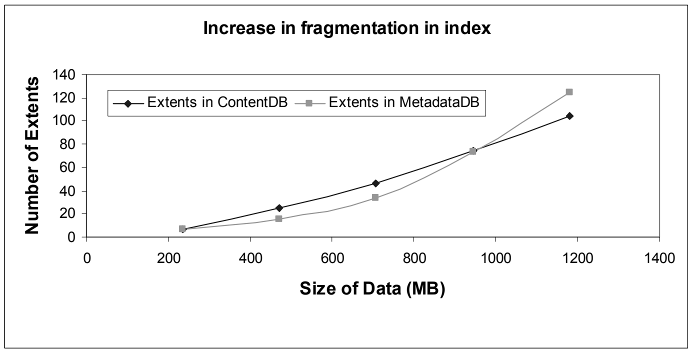

Figure 11. The graph characterizes the increase in the extents in the indexes as data is increased sequentially in the system. A folder 236MB in size and containing 18603 files is added at each iteration. The corresponding increase in the extents in the metadata store (Store.db) and content index (ContentIndex.db) is plotted on the y-axis.

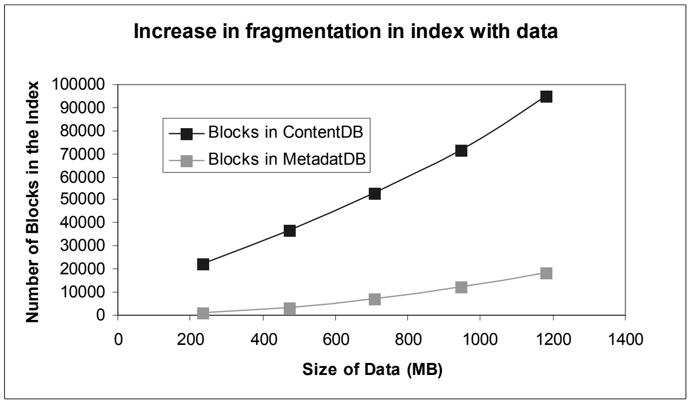

Figure 12. The graph characterizes the increase in the blocks in the indexes as data is increased sequentially in the system. A folder 236MB in size and containing 18603 files is added at each iteration. The corresponding increase in the blocks in the metadata store (Store.db) and content index (ContentIndex.db) is plotted on the y-axis.

3.5 System call characterization of indexing

This section characterizes the system calls that happen per Spotlight process during the indexing of one file and also the breakdown of the number of system calls that happen as a result of indexing a 236MB folder containing 18603 text files.

Table 3 shows the timeline of system calls generated while indexing a single file foo.txt.

Java	mds	mdimport
Open foo		
Write		
Close		
	read() fsevent	
	getfileattr()	
		lstat()
		getfileattr()
		open()
		fstat()
		read()
		close()
	pwrite() .storedb	
	pwrite() ContentIndex.db	

Table 3. Breakdown of system calls generated per process during the indexing of a file foo.txt

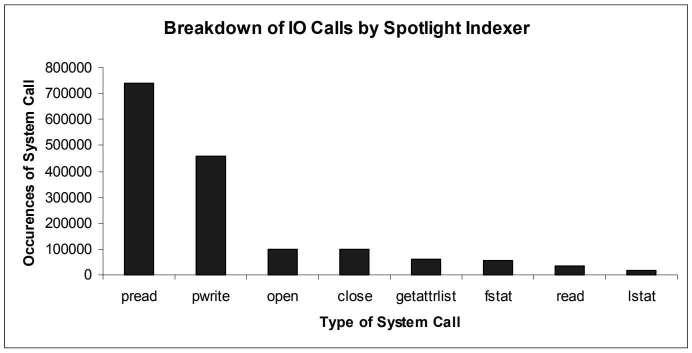

Figure 13. This graph shows a breakdown of the system calls generated as a result of indexing a 236MB folder containing 18603 files

As observed in the experiments, there are several file attribute retrieval calls made per file during indexing. Since, the fsevent does return the attributes of a file, the indexer should be able to just use the same instead of invoking multiple calls to get the attributes of the file.

3.6 Performance impact of Spotlight

In this set of experiment, I wanted to analyze the performance impact of Spotlight on the system. I used a macro-benchmark IOZone for this purpose. I changed the code of IOZone so that it uses text files (which are indexeable) with real content as opposed to containing just strings of "aaa". I ran IOZone first with indexing turned on the volume and then with indexing turned off on the system. Figure 14 characterizes the throughput of write operations with/without indexing.

As seen in Figure 14, the performance impact of Spotlight indexing during file system activity is minimal. The Spotlight server, mds just reads the resulting fsevents generated as a result of the file system changes while the file system changes are happening. Once, the file system changes are done, mds invokes mdimport on the files for whom change notifications were received.

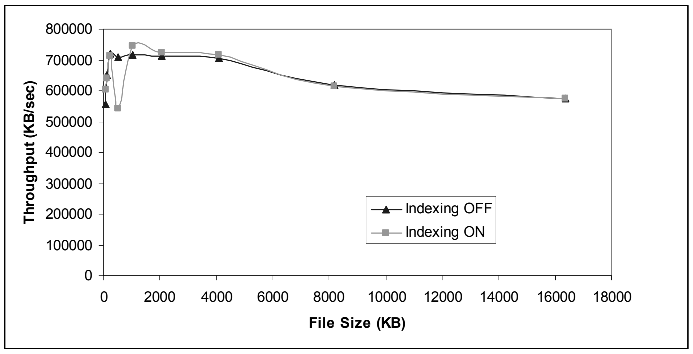

Figure 14. This graph shows the disk throughput of file writes on the y-axis with indexing turned OFF vs. indexing turned ON.

3.7 Indexing Optimizations

In this set of experiments, I wanted to check if any optimizations such as incremental indexing exist in the system. I appended 34 Bytes of data at the end of 1 MB, 2MB, 3MB, 4MB and 5MB files. In order to minimize buffer caching of these files, I first created these files and then did IO on several files each more than 1 GB in size. Subsequently, I appended 34 Bytes to the files. I measured the time to index each file once the data is appended to the file, which is plotted in Figure 15.

As seen in Figure 15, the time to index the files increases with the size of the file. This indicates that there is no incremental indexing present in the system.

Ideally, I wanted to do this experiment with filesizes in GB instead of MB. However, I realized that Spotlight doesn't seem to be indexing any data at an offset greater than 10MB.

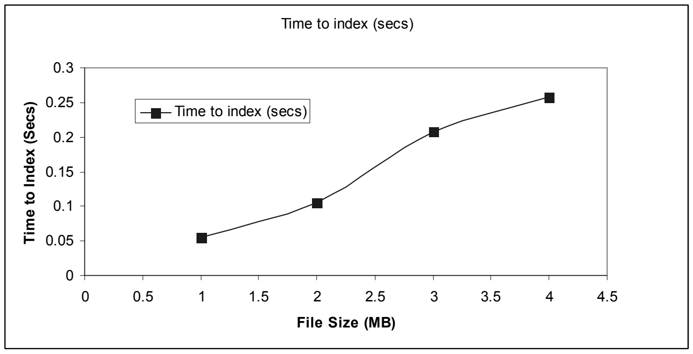

Figure 15. The graph characterizes the time to index files as new data is appended to the end of pre-indexed files.

4 SUMMARY

Reliability of the fsevent interface

The fsevent interface is not reliable. A slow watcher or substantial file system can result in dropped fsevents. Spotlight silently ignores the dropped events. As a result, the index gets inconsistent with the actual content of the files for which the fsevents were dropped. There is a small hard limit on the global event buffer which further aggravates the problem. The alternative to trigger re-scan of entire volume is also not a good solution as it can be a performance hog in case of heavily populated volume. There is a need to have a better infrastructure for sharing the file system events with the user-space than fsevents.

Liveness of Index Update

In event of substantial file system activity, the index updation is not really live. Indexing is an expensive and time-consuming process in itself if the number of files to be indexed is large. Also, the mdimport process runs at a low priority. Thereby, any activity which results in high file system changes, will get a priority over indexing. Thus, the actual time to update the index = time for the file system activity to finish + time to index the files. Any queries triggered while indexing is happening on the system are not answered.

Performance of Indexing

Indexing is a performance intensive operation characterized by high CPU utilization and disk transfers.

Storage Characterization of Index

The content index increases quite substantially as data is increased on the system. On average, it is 17% of the actual data that was indexed. Content index is not deleted upon folder deletion leading to further waste. The metadata index has lot of duplicate information about the file system as it stores the standard metadata of the files in addition. The indexes tend to get fragmented very fast as well.

System call characterization of indexing

There are several calls to get the attributes of the files which seem superfluous. Since, fsevents does embed the attributes of the file in the event itself, the same can be used as an optimization. Also, mdimport uses no_cache option to reduce it's footprint on the buffer cache. On the flip side, such an approach prevents it from reading data of the file from the cache and leads it to the expensive approach of reading from the disk.

Performance of Spotlight

Since mdimport runs at a low priority, any system activity takes precedence over mdimport. However, mds does read the events from the fsevents infrastructure and log them into .journalhistorylog. Thus, performace impact of Spotlight is minimal.

Indexing optimizations

No indexing optimizations such as incremental indexing are present in Spotlight. Appending small amount of data to an already indexed file, results in the file getting re-indexed all over again. This would lead to quite a bit of wasted indexing effort if the files are huge. A one byte append at the end will lead to the huge file getting read from the disk and getting reindexed.

5 FUTURE WORK

We would like to characterize the performance of queries in the following way:

- Time to query as number of files in which the word is contained is increased.
- Time to query as index gets fragmented
- Time to update query results upon file system change

In addition, it would be good to repeat the earlier experiments with larger file sizes and also different content types such as pdf, image files etc.

REFERENCES

[1] http://developer.apple.com/macosx/spotlight.html

[2] http://beagle-project.org/Main_Page

[3] http://www.gnome.org/projects/tracker/

[4] http://lucene.apache.org

[5] http://research.microsoft.com/adapt/phlat/

[6]http://www.opensource.apple.com/darwinsource/10.4.10.x86/

[7] http://developer.apple.com/documentation/UserExperience/Reference/SearchKit/

[8] http://developer.apple.com/documentation/Carbon/Conceptual/MetadataIntro/index.html

[9] http://developer.apple.com/documentation/Carbon/Conceptual/SpotlightQuery/index.html#//apple_ref/doc/uid/TP40001841

[10] http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_ProgGuide/TechnologyOverview/chapter_3_section_1.html#//apple_ref/doc/uid/TP40005289-CH3-SW1

[11]http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/fs_usage.1.html

[12]http://developer.apple.com/documentation/Darwin/Reference/ManPages/man8/iostat.8.html

[13]http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/top.1.html