Grand Unified File Index (GUFI)

Background:

Many computing sites have multiple and multiple types of POSIX oriented file systems. At HPC sites, large parallel scratch file systems are used to allow supercomputers to access for checkpoint restart, input, analysis, etc. Often there is more than one parallel scratch file system for availability or other purposes and frequently technologies like the IBM's GPFS, Lustre, PVFS, and Panasas are used. In addition to parallel scratch file systems, there is often non parallel file systems to hold home and project directories which is often provided by NFS or other solutions. Additionally, HPC sites might have cool or cold storage file systems like campaign stores and or archives. While these cool/cold storage may not be visible as a normal POSIX mounted file system, the metadata about the files in these systems is often represented by POSIX tree based name and attribute spaces. A large site might have hundreds of billions to many trillion files in POSIX like tree name/attribute structures. Both users of these file systems and storage administrators need tools to find files or sets of files for a variety of purposes by looking at file attributes like path, name, size, date/time, and extended attributes of many kinds and across multiple file systems and types of file systems. The file system technologies used to provide these services like scratch, home/project, cool, and cold storage are optimized for their particular jobs but none are really optimized for metadata search.

There are certainly solutions for providing indexing capability over file system metadata that exist today like Nirvana [srb/nirvana ref], Robin Hood [ref], and Starfish Storage. Just as each file system type has been optimized for the service it provides, these file metadata indexes have also been optimized mostly to assist storage system administrators to manage the data in these various file storage systems. Most of the solutions in this indexing of files space is heavily skewed toward policy management.

Objectives:

The objectives of GUFI include:

- Search file metadata across trillions of files in minutes
- Search across many file systems and types of file systems metadata
- Metadata only searching including attributes and extended attributes
- Index works well for storage administrators
- Index works well for users
- Index honors POSIX security mechanisms including inherited tree based permissions (if you don't have execute on a directory you can't get into subdirectories under that directory, etc.)
- High thread and process/node parallelism exploitation
- Exploitation of flash for storage which implies highly threaded access and access sizes in the few kilobytes range (flash is highly optimized for 4K blocks for legacy reasons)
- Allows for full and incremental update from source file systems
- Custom parallel search capabilities

- Can use as a mounted file system where you get a virtual image of your file metadata based on query input
- Reasonable load/update times (hundreds of thousands/sec)
- Simple for a storage administrator to understand
- Small amount of code
- Easily extensible to add more queryable information/fields
- Could run on a laptop up to multi-node cluster
- Efficient for difficult POSIX name space/attribute operations like renames of directories from high in the tree, unlinks, etc.
- Leverage as much existing proven to be best of breed technologies like
    - Flash storage
    - Highly threaded/process parallelism
    - Familiar interfaces for query like sql
    - Open source/commercial file/database technologies
    - Unix tree traversal/security

Discarded approaches:

Indexing of file systems metadata is somewhat difficult given that file system metadata has two structure types. The name space (path names) metadata is a hierarchical tree while the attributes about those files is largely flat records. The two most prevalent methods for providing a file system metadata indexing service are flattened sharded and trees implemented as relational tables.

Some file indexing metadata indexing products flatten the tree into full paths with attributes for a file all in one record. These approaches provide extremely simple ways to shard the flat records much like Hive [ref]. Queries can be made to be arbitrarily fast because of the simplistic way to parallelize the workload. If a metadata indexing system is only to be used by storage system administrators and not users, this approach is appealing. Drawbacks to this approach include very difficult tree traversal to determine access rights for users, difficulty in making the output of a query look like a mounted virtual file system, and difficult POSIX tree name space operations like rename of a directory high in the tree could cause millions or even billions of metadata entry updates due to the flattened nature of the index.

Many examples of putting file system path trees into a relational table exist in products like Robin Hood []. This approach has a unique identifier for a file, like an inode or fid, and a parent inode/fid (the directory the entry resides in), in every record. The full path name is not in every recorded making high in the tree renames efficient, but tree traversal requires repetitive query from the top of the tree with each query looking for child records for each parent encountered (essentially a breadth first search with a query for every level encountered). The trade off of making rename from high in the tree is paid for by the queries being recursive. While this recursive breadth first walk is parallelizable, all the queries are going against the same table. It is possible to shard based on parent inode/fid but that complicates the rename from high in the table capability. The drawback of this approach is traversal is inefficient.

The GUFI Hybrid Approach:

In order to accommodate all the objectives for GUFI, a hybrid approach to indexing was chosen. The approach is a hybrid use of file system technology used for tree traversal/metadata management and database technology for flat attribute/extended attribute metadata. We leverage POSIX tree walking/directory traversal/permissions checking for directories only which has been optimized a quarter century to be very fast, have very aggressive caching behavior, and provide POSIX security inherently. The readdir-plus capabilities of most file systems is utilized to get extremely fast tree traversal. This approach also enables handling of renames of all kinds including from high in the tree trivially.

The metadata within a directory is flat and is provided by an open source very fast imbeddable database, sqlite3 which provides easy query sql syntax and flexibility for adding fields etc. Many open source relational databases get slow when records approach billions with complex table joins, but in GUFI a relational table is only the size of the entries in a single directory. Because typical directories have tens to hundreds of files in them, the metadata databases are tens of kilobytes making this a great match for flash technology. Using a breadth first search enables extremely wide thread parallelism and tree based decomposition allows for the trees to be broken up over multiple nodes in a cluster providing for scalability both with threads and explicit cross node parallelism. In our testing, a single table with no joins, which is how directory entries are stored, can easily handle hundreds of millions of entries.

Additionally, trees are very natural structures for compact representation of metadata that lives below a point in a tree. Simple directory and tree based indexes of statistical roll ups of the metadata in that directory or tree below can be created to limit where queries must run.

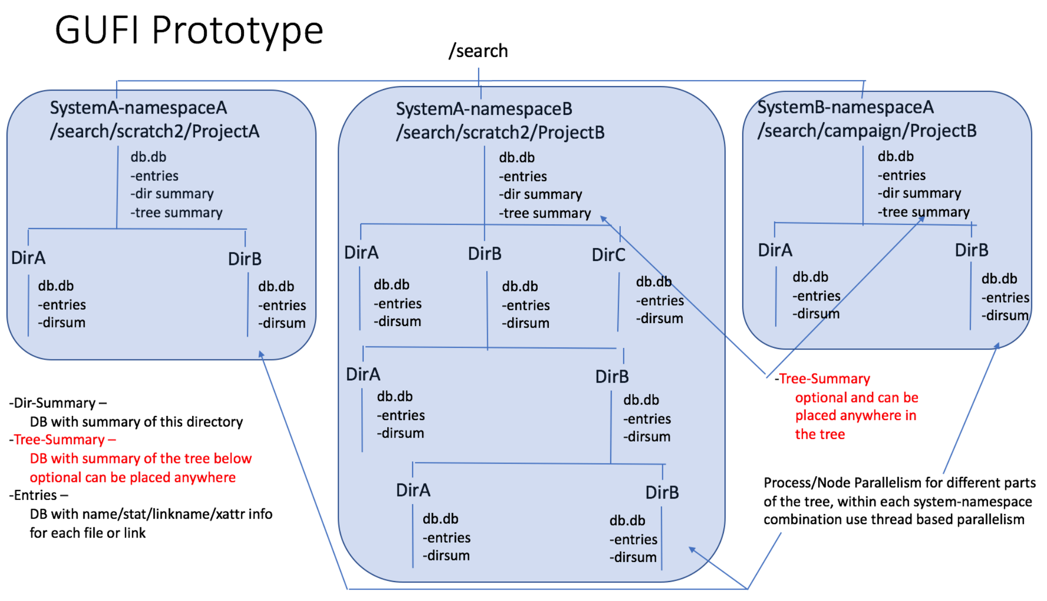


Figure A

Figure A above shows the namespace broken up into multiple trees for process-based parallelism cross node. Within each node breadth first search thread parallelism can be used. Each directory mimics a directory in the source file system where the metadata came from. The directory permissions also mimic the source file system and the database files within each directory are protected to mimic POSIX file permissions in the source file system, so if the directory has read permissions for a user, the user is allowed to enumerate the files in that directory and if the user has execute permissions in the directory and the tree above, the user can find out attributes about the entries.

There is a single database file in every directory that contains multiple sqlite tables.

There is an entries table/database file in each directory that represents the files and links that occur in that directory on the source file system, one record for each file or link. There is a key field in that table, namely the name of the file or link. This is not the path, as the path is constructed from the POSIX tree the database file lives in. The name is a key only because of the requirement to be able to make the output of a query look like a mounted file system which is provided via FUSE and in fuse, lookups are done by path which includes the name of the file, so to provide efficient lookups for the FUSE mounted query output function, name is the key.

Also in every directory is a summary table/database file. This table contains the statistical rollup information for the attributes of the files/links in that directory. The types of information kept in this table include, size of largest file, size of smallest file, number of files, number of links, dates of oldest file, dates of youngest file, files bigger than 1k/10k/100k/…, number of files with extended attributes, etc. This table can be consulted to determine if the entries table even needs to be looked at in a query. By user and by group rollup information and other rollup information about the directory can be kept in this table. Further the inode/fid and parent inode/fid are kept in this table in one record, so renames of directories is efficient.

Some directories can also have a treesummary table/database file. This table contains similar statistical rollup information to the summary table but it represents the rolled up information for all the files/links in all the directories in this directory and all subdirectories recursively. To create this treesummary table you only need to recursively query the summary tables for each directory in a walk down the tree and summarize the summary information, so creation of these tree summaries once the summary tables are created is extremely efficient and once created, these treesummary tables can be consulted to determine if a query needs to traverse into that part of the namespace. Like the summary table, user and group level summaries can be rolled up for the tree below in this table as well.

Loading:

Example programs have been created to load a GUFI style index from walking a POSIX file system from information extracted from several file/storage systems using different methods described below.

Loading a GUFI tree index directly from a source POSIX file system:
There is of course the need to load from a POSIX file system. This is accomplished by doing a breadth first walk of the source tree and using new thread work for each directory encountered. Master thread loops through assigning directories encountered that are retrieved from a queue. Worker threads create the target directory (recursively if necessary), read directory entries, stat to get attributes for entries, get extended attributes for entries in a loop and create the entries table containing file and link information for that directory. Summary information is calculated and stored in the directory summary table for that directory. As directories in the directory being worked on are encountered they are put on a queue for the master thread to assign to other threads. This makes for a very parallel way to extract information from the source file system and create the GUFI tree index. Since only one thread works on any one directory at a time, there is no need for locking for update of the summary or entries tables.

```
pn1614518:test ggrider$ ls -lR testdir  (walk the source tree)
total 8
-rw-r--r--@ 1 ggrider  staff    0 May 16  2017 a
-rw-r--r--@ 1 ggrider  staff    0 May 16  2017 b
drwxr-xr-x  6 ggrider  staff  204 Dec  9 15:08 c
lrwxr-xr-x@ 1 ggrider  staff    4 Jun 18 10:07 clink -> c/ca
-rw-r--r--@ 1 ggrider  staff    0 May 16  2017 d
-rw-r--r--@ 1 ggrider  staff    0 May 28  2017 dumbcom,ma
-rw-r--r--@ 1 ggrider  staff    0 May 28  2017 gary's dumb file

testdir/c:
total 0
-rw-r--r--@ 1 ggrider  staff    0 May 16  2017 ca
-rw-r--r--@ 1 ggrider  staff    0 May 16  2017 cb
drwxr-xr-x  3 ggrider  staff  102 Dec  9 15:08 cc
-rw-r--r--@ 1 ggrider  staff    0 May 16  2017 cd

testdir/c/cc:
total 32
-rwxr-xr-x@ 1 ggrider  staff  14252 May 22  2017 bfindex

pn1614518:test ggrider$ ls -lR testdirdup/testdir (walk the resulting GUFI tree)
total 40
drwxr-xr-x  4 ggrider  staff    136 Dec 11 10:13 c
-rwxr-xr-x  1 ggrider  staff  20480 Dec 11 10:13 db.db

testdirdup/testdir/c:
total 40
drwxr-xr-x  3 ggrider  staff    102 Dec 11 10:13 cc
-rwxr-xr-x  1 ggrider  staff  20480 Dec 11 10:13 db.db

testdirdup/testdir/c/cc:
total 40
-rwxr-xr-x  1 ggrider  staff  20480 Dec 11 10:13 db.db
```
Figure B

Figure B above shows the source file system and the resulting GUFI tree index that results. One goal of the GUFI design was efficient loading. The interpretation of that goal in this current load example would be to ensure that the time it takes to load the GUFI tree index is small compared to the extraction time from the source file system.

Performance results:
This was achieved but I don't really have good data as I was writing data from a readdir plus into an sqlite3 table, not making a gufi exactly. In that case the loading of the sqlite3 table was 10-20x faster than the readdirplus walk. Stat dwarfs readdirplus times so I suspect loading the gufi is more like 100X or more faster than walk+stat+getxattr from a source file system. It's clear this is easily done but I didn't do it exactly but for a paper one should probably run the experiment.

Loading a GUFI tree index directly from a completely flattened path, attributes list in a file with records from a directory grouped together in the file:
There is a need to load a GUFI tree index from flat file dumps of file systems. The method that was prototyped was using a file with the path and all attributes on one line per file/directory/link. The file was sorted so that all the entries for a directory fall immediately below the parent directory entry. The approach is almost identical to the above loading a GUFI index tree from a source POSIX file system except instead of walking the tree in breadth first, the master task reads the source input file looking for offsets and lengths of stanzas with the parent directory and direct children records and hands those chunks of the file off to worker threads to create the entries and directory summary tables.

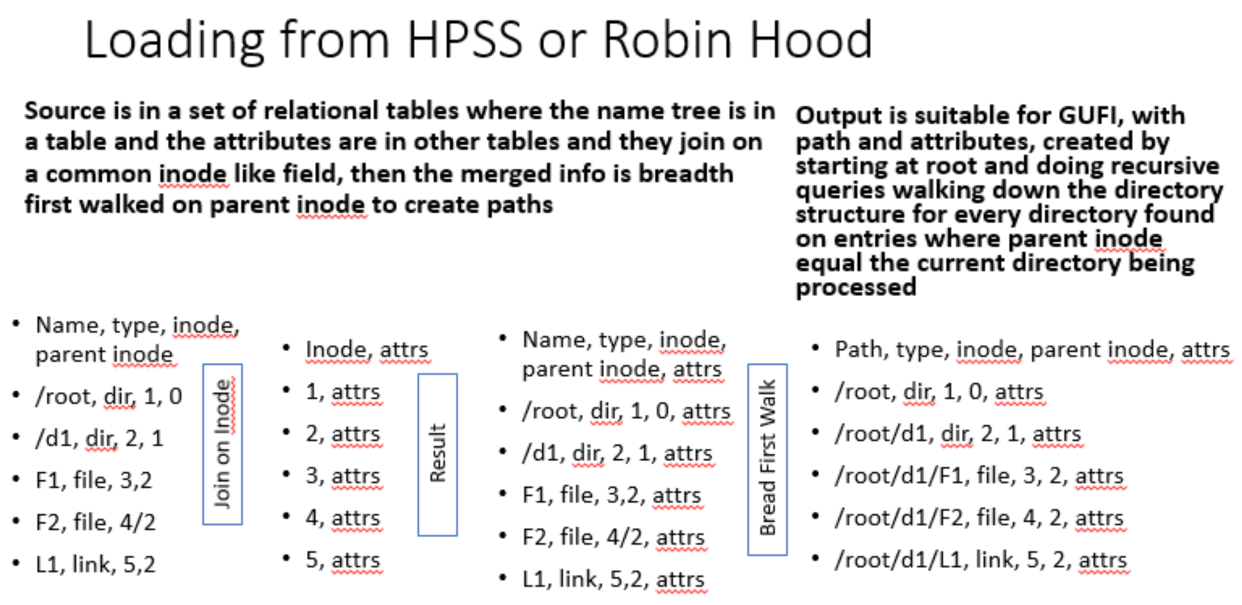
In this case it is not a given that loading a gufi index tree will outperform reading the input file. It is possible that it might due to the threading on the directory/table creation work, but that would depend on the performance of the storage being written to and read from.

Performance results:
I didn't keep the performance number exactly but it's clear it is faster than the above walking the source tree because you are reading serially and not doing readdir and stat and getxattr.

Loading a GUFI tree index from a source GPFS style policy run that does a readdir+ style tree walk producing full paths and inodes, an inode scan producing inodes and attributes, and a sort/merge on inode policy run:
This method of loading requires the merging of tree based information containing names which can be walked into paths and inode with flat inode information containing attributes. This method would do the above merge to create a file and use the above method of loading from a file in breadth first search order or some similar mechanism.

Performance results:
The goal again is for the gufi loading to be a small % of the total extract/load process.
I haven't done this exactly yet but here is what I have done
Don't know how long it takes for the readdirplus walk – guessing a few minutes
Then you have to sort on inode and on my mac sorting took 210sec for 200M records
Don't know the dump inode time which is sorted in inode order guessing a few minutes
Join of inode and readdir walk information took 34 minutes
Building a GUFI from a joined table took 18 minutes – but don't remember if that was terrible threaded. It's clear the gufi load is smaller, not sure by how much, 3-10X maybe.

Loading a GUFI index tree from a HPSS and Robin Hood systems where name tree is kept in a table with name, inode, and parent inode and attributes are kept in other tables by inode: This mechanism requires joining the name table with the attributes tables on the common inode like field and then doing a recursive set of queries that simulates breadth first tree traversal using the parent inode to find all entries in the name tree table that are in that directory and putting all directories encountered on a queue to be queried for entries. A very similar process flow to the loading from a POSIX source tree file system is used using breadth first search with master process and workers and a queue to hold directories encountered.



Loading from HPSS or Robin Hood
Source is in a set of relational tables where the name tree is in a table and the attributes are in other tables and they join on a common inode like field, then the merged info is breadth first walked on parent inode to create paths
Output is suitable for GUFI, with path and attributes, created by starting at root and doing recursive queries walking down the directory structure for every directory found on entries where parent inode equal the current directory being processed
Name, type, inode, parent inode
/root, dir, 1, 0
/d1, dir, 2, 1
F1, file, 3,2
F2, file, 4/2
L1, link, 5,2
Join on Inode
Inode, attrs
1, attrs
2, attrs
3, attrs
4, attrs
5, attrs
Result
Name, type, inode, parent inode, attrs
/root, dir, 1, 0, attrs
/d1, dir, 2, 1, attrs
F1, file, 3,2, attrs
F2, file, 4/2, attrs
L1, link, 5,2, attrs
Bread First Walk
Path, type, inode, parent inode, attrs
/root, dir, 1, 0, attrs
/root/d1, dir, 2, 1, attrs
/root/d1/F1, file, 3, 2, attrs
/root/d1/F2, file, 4, 2, attrs
/root/d1/L1, link, 5, 2, attrs

Figure 3

Figure 3 shows the concept of joining tables on the common inode like field and then the outcome of the breadth first query walk to produce paths from names, inodes, and parent inodes.

Peformance results:
I really haven't done a perf test. As usual, the goal is the cost of loading the GUFI index table is far less than the extraction cost.
Load of names table

```
# real 11m27.506s  to import 200M lines
load of entries table
# real 34m7.418s
recursive query on join of names and entries doing breadth first search single threaded
# real 151m35.371s
recursive query on join of names and entries doing breadth first search multiple threaded
6 threads real       25m27.165s
```

Creation of tree summary indexes
Recall that tree summary indexes exist in directories, though they are not required. These directories summarize statistics about all the directories below the directory in which the tree summary index resides. Creation of these indexes is very efficient because creation is accomplished by walking the tree below the directory in question and collecting and summarizing statistics from each of the directory summary tables. Looking at all entries below the directory in question is not necessary, since each directory is summarized into the summary table already.

An example of pulling data from a mysql robinhood and loading a GUFI tree ran at about 100M files/hour. We don't know how much of that time was pulling the information vs loading the gufi yet.

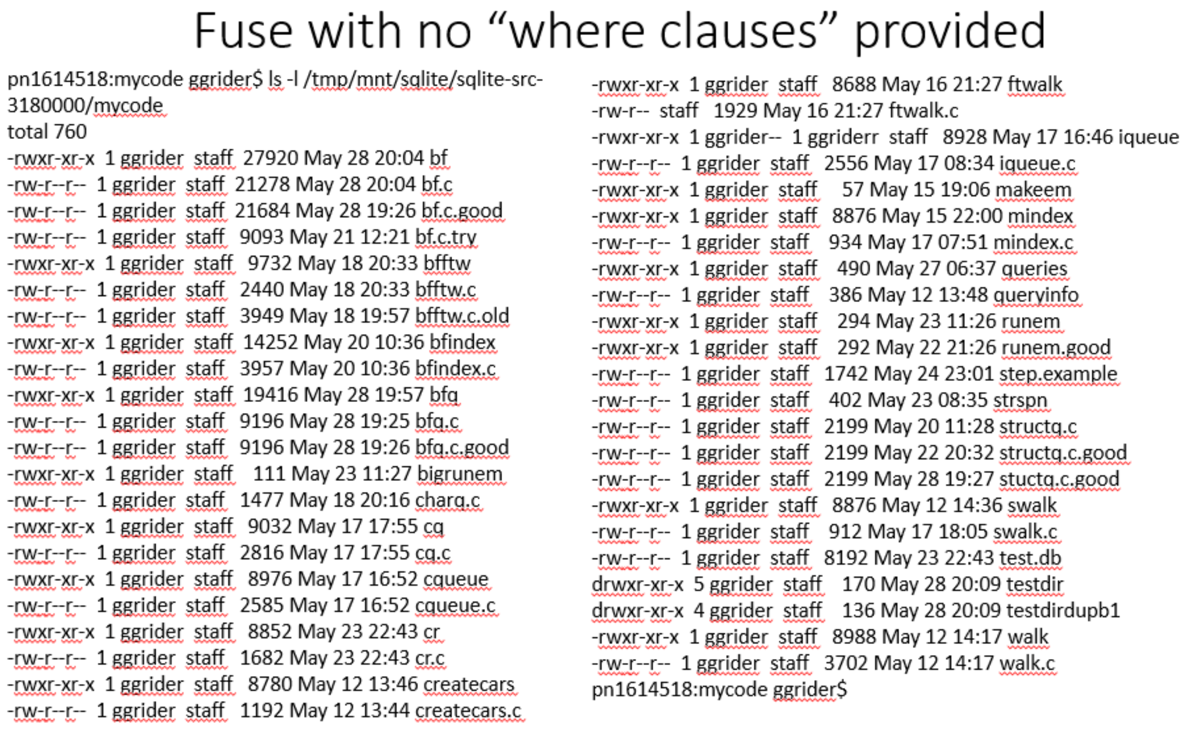
Query:
The query program essentially does a breadth first walk of the index tree and obeys POSIX permissions so users can only traverse into what they are allowed to see in the source file systems while root is allowed to traverse the entire tree. The breadth first walk creates very wide thread based parallelism, a thread per directory encountered, similar to the approach described above in the loading from a source POSIX tree. Each directory encountered is placed on a queue to be dispatched to another thread.

The query input parameters include various operational options like printing levels, max threads, input path, etc. Additionally three sql statements are allowed to be input, one for the treesummary tables encountered, one for the directory summary tables encountered, and one for the entries tables encountered.

Some examples
Query a 1 billion file and ~130,000 directory GUFI tree with 40 threads on a mac laptop.
10 of the directories have 10M files in them
For basically all scans of all 1 Billion entries takes about 3.5 minutes
For basically all scans of all 130,000 summary tables takes about 2 minutes.
Of course if tree summaries are used fairly high in the tree, this can eliminate massive parts of the tree to be examined, one would think 10X minimum but probably more like 100X
If users are running queries, they can only access the parts of the tree they can access in the source storage systems, so again one would expect a user to find something in their parts of the trees in 1/100th or less the time, so that would be like a small number of seconds (like 2-3).
These numbers are from a macbook pro laptop with an ssd drive, so a well outfitted server with many SSD devices or devices with higher IOP's one might expect sever times improvement.
Also, if your site has >> 1B files and or >> 200k directories to keep queries at the same speed you might want to do name space decomposition and have a GUFI over parts of your overall set of metadata trees which is pretty trivial to do.

The FUSE mounted query:

The concept behind this Fuse mounted query is to allow the user/admin to provide query input in the form of the three sql statements described above in the query section. When the fuse mount program is run, it uses these sql statements to provide the output of readdir, stat, getxattr, readlink, etc. So if you provide a set of select statements that says only show me files that are bigger than 1 gigabyte, the fuse will only traverse into directories or entire parts of the tree that the user has access to that have files that are larger than 1 gigabyte and then only show you files that are bigger than 1 gigabyte as an example. Normal metadata query only POSIX oriented commands may be run like ls, cd, find, etc.



Fuse with no "where clauses" provided

```
pn1614518:mycode ggrider$ ls -l /tmp/mnt/sqlite/sqlite-src-
3180000/mycode
total 760
-rwxr-xr-x  1 ggrider  staff  27920 May 28 20:04 bf
-rw-r--r--  1 ggrider  staff  21278 May 28 20:04 bf.c
-rw-r--r--  1 ggrider  staff  21684 May 28 19:26 bf.c.good
-rw-r--r--  1 ggrider  staff   9093 May 21 12:21 bf.c.try
-rwxr-xr-x  1 ggrider  staff   9732 May 18 20:33 bfftw
-rw-r--r--  1 ggrider  staff   2440 May 18 20:33 bfftw.c
-rw-r--r--  1 ggrider  staff   3949 May 18 19:57 bfftw.c.old
-rwxr-xr-x  1 ggrider  staff  14252 May 20 10:36 bfindex
-rw-r--r--  1 ggrider  staff   3957 May 20 10:36 bfindex.c
-rwxr-xr-x  1 ggrider  staff  19416 May 28 19:57 bfq
-rw-r--r--  1 ggrider  staff   9196 May 28 19:25 bfq.c
-rw-r--r--  1 ggrider  staff   9196 May 28 19:26 bfq.c.good
-rwxr-xr-x  1 ggrider  staff    111 May 23 11:27 bigrunem
-rw-r--r--  1 ggrider  staff   1477 May 18 20:16 charq.c
-rwxr-xr-x  1 ggrider  staff   9032 May 17 17:55 cq
-rw-r--r--  1 ggrider  staff   2816 May 17 17:55 cq.c
-rwxr-xr-x  1 ggrider  staff   8976 May 17 16:52 cqueue
-rw-r--r--  1 ggrider  staff   2585 May 17 16:52 cqueue.c
-rwxr-xr-x  1 ggrider  staff   8852 May 23 22:43 cr
-rw-r--r--  1 ggrider  staff   1682 May 23 22:43 cr.c
-rwxr-xr-x  1 ggrider  staff   8780 May 12 13:46 createcars
-rw-r--r--  1 ggrider  staff   1192 May 12 13:44 createcars.c
-rwxr-xr-x  1 ggrider  staff   8688 May 16 21:27 ftwalk
-rw-r--  staff   1929 May 16 21:27 ftwalk.c
-rwxr-xr-x  1 ggrider--  1 ggriderr  staff   8928 May 17 16:46 iqueue
-rw-r--r--  1 ggrider  staff   2556 May 17 08:34 iqueue.c
-rwxr-xr-x  1 ggrider  staff     57 May 15 19:06 makeem
-rwxr-xr-x  1 ggrider  staff   8876 May 15 22:00 mindex
-rw-r--r--  1 ggrider  staff    934 May 17 07:51 mindex.c
-rwxr-xr-x  1 ggrider  staff    490 May 27 06:37 queries
-rw-r--r--  1 ggrider  staff    386 May 12 13:48 queryinfo
-rwxr-xr-x  1 ggrider  staff    294 May 23 11:26 runem
-rwxr-xr-x  1 ggrider  staff    292 May 22 21:26 runem.good
-rw-r--r--  1 ggrider  staff   1742 May 24 23:01 step.example
-rw-r--r--  1 ggrider  staff    402 May 23 08:35 strspn
-rw-r--r--  1 ggrider  staff   2199 May 20 11:28 structq.c
-rw-r--r--  1 ggrider  staff   2199 May 22 20:32 structq.c.good
-rw-r--r--  1 ggrider  staff   2199 May 28 19:27 stuctq.c.good
-rwxr-xr-x  1 ggrider  staff   8876 May 12 14:36 swalk
-rw-r--r--  1 ggrider  staff    912 May 17 18:05 swalk.c
-rw-r--r--  1 ggrider  staff   8192 May 23 22:43 test.db
drwxr-xr-x  5 ggrider  staff    170 May 28 20:09 testdir
drwxr-xr-x  4 ggrider  staff    136 May 28 20:09 testdirdupb1
-rwxr-xr-x  1 ggrider  staff   8988 May 12 14:17 walk
-rw-r--r--  1 ggrider  staff   3702 May 12 14:17 walk.c
pn1614518:mycode ggrider$
```

Figure 4

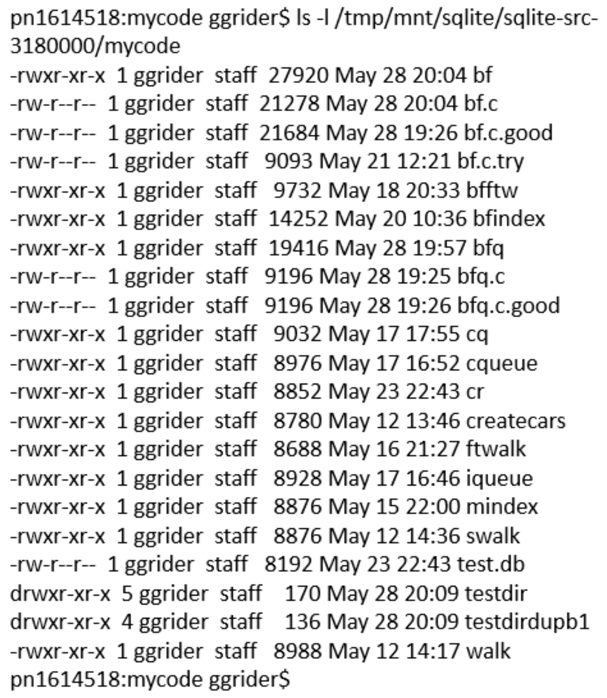
Figure 4 shows a FUSE using a GUFI index tree and input listing all files in a directory.

Below in Figure 5, is the same FUSE using the same GUFI index tree as input but with the following queries provided:

```
pn1614518:mycode ggrider$ echo "where size > 8000" > entries.query
pn1614518:mycode ggrider$ echo "where size > 1000" > summary.query
pn1614518:mycode ggrider$ cat entries.query
where size > 8000
pn1614518:mycode ggrider$ cat summary.query
where maxsize > 1000
```

Look in directories where there are files bigger than 1000
And

In those directories list the files that are bigger than 8000



```
pn1614518:mycode ggrider$ ls -l /tmp/mnt/sqlite/sqlite-src-
3180000/mycode
-rwxr-xr-x  1 ggrider  staff  27920 May 28 20:04 bf
-rw-r--r--  1 ggrider  staff  21278 May 28 20:04 bf.c
-rw-r--r--  1 ggrider  staff  21684 May 28 19:26 bf.c.good
-rw-r--r--  1 ggrider  staff   9093 May 21 12:21 bf.c.try
-rwxr-xr-x  1 ggrider  staff   9732 May 18 20:33 bfftw
-rwxr-xr-x  1 ggrider  staff  14252 May 20 10:36 bfindex
-rwxr-xr-x  1 ggrider  staff  19416 May 28 19:57 bfq
-rw-r--r--  1 ggrider  staff   9196 May 28 19:25 bfq.c
-rw-r--r--  1 ggrider  staff   9196 May 28 19:26 bfq.c.good
-rwxr-xr-x  1 ggrider  staff   9032 May 17 17:55 cq
-rwxr-xr-x  1 ggrider  staff   8976 May 17 16:52 cqueue
-rwxr-xr-x  1 ggrider  staff   8852 May 23 22:43 cr
-rwxr-xr-x  1 ggrider  staff   8780 May 12 13:46 createcars
-rwxr-xr-x  1 ggrider  staff   8688 May 16 21:27 ftwalk
-rwxr-xr-x  1 ggrider  staff   8928 May 17 16:46 iqueue
-rwxr-xr-x  1 ggrider  staff   8876 May 15 22:00 mindex
-rwxr-xr-x  1 ggrider  staff   8876 May 12 14:36 swalk
-rw-r--r--  1 ggrider  staff   8192 May 23 22:43 test.db
drwxr-xr-x  5 ggrider  staff    170 May 28 20:09 testdir
drwxr-xr-x  4 ggrider  staff    136 May 28 20:09 testdirdupb1
-rwxr-xr-x  1 ggrider  staff   8988 May 12 14:17 walk
pn1614518:mycode ggrider$
```

Figure 5

Current Schema:

```
char *esql = "DROP TABLE IF EXISTS entries;"
   "CREATE TABLE entries(name TEXT PRIMARY KEY, type TEXT, inode INT64, mode INT64, nlink
INT64, uid INT64, gid INT64, size INT64, blksize INT64, blocks INT64, atime INT64, mtime
INT64, ctime INT64, linkname TEXT, xattrs TEXT, crtime INT64, ossint1 INT64, ossint2 INT64,
ossint3 INT64, ossint4 INT64, osstext1 TEXT, osstext2 TEXT);";

char *ssql = "DROP TABLE IF EXISTS summary;"
   "CREATE TABLE summary(name TEXT PRIMARY KEY, type TEXT, inode INT64, mode INT64, nlink
INT64, uid INT64, gid INT64, size INT64, blksize INT64, blocks INT64, atime INT64, mtime
INT64, ctime INT64, linkname TEXT, xattrs TEXT, totfiles INT64, totlinks INT64, minuid
INT64, maxuid INT64, mingid INT64, maxgid INT64, minsize INT64, maxsize INT64, totltk
INT64, totmtk INT64, totltm INT64, totmtm INT64, totmtg INT64, totmtt INT64, totsize INT64,
minctime INT64, maxctime INT64, minmtime INT64, maxmtime INT64, minatime INT64, maxatime
INT64, minblocks INT64, maxblocks INT64, totxattr INT64,depth INT64, mincrtime INT64,
maxcrtime INT64, minossint1 INT64, maxossint1 INT64, totossint1 INT64, minossint2 INT64,
maxossint2 INT64, totossint2 INT64, minossint3 INT64, maxossint3 INT64, totossint3
INT64,minossint4 INT64, maxossint4 INT64, totossint4 INT64, rectype INT64, pinode INT64);";

char *tsql = "DROP TABLE IF EXISTS treesummary;"
   "CREATE TABLE treesummary(totsubdirs INT64, maxsubdirfiles INT64, maxsubdirlinks INT64,
maxsubdirsize INT64, totfiles INT64, totlinks INT64, minuid INT64, maxuid INT64, mingid
INT64, maxgid INT64, minsize INT64, maxsize INT64, totltk INT64, totmtk INT64, totltm
INT64, totmtm INT64, totmtg INT64, totmtt INT64, totsize INT64, minctime INT64, maxctime
INT64, minmtime INT64, maxmtime INT64, minatime INT64, maxatime INT64, minblocks INT64,
maxblocks INT64, totxattr INT64,depth INT64, mincrtime INT64, maxcrtime INT64, minossint1
INT64, maxossint1 INT64, totossint1 INT64, minossint2 INT64, maxossint2 INT64, totossint2
INT64, minossint3 INT64, maxossint3 INT64, totossint3 INT64, minossint4 INT64, maxossint4
INT64, totossint4 INT64,rectype INT64, uid INT64, gid INT64);";

char *vesql = "create view pentries as select entries.*, summary.inode as pinode from
entries, summary where rectype=0;";
```

```
char *vssqldir = "create view vsummarydir as select * from summary where rectype=0;";
char *vssqluser = "create view vsummaryuser as select * from summary where rectype=1;";
char *vssqlgroup = "create view vsummarygroup as select * from summary where rectype=2;";
char *vtssqldir = "create view vtsummarydir as select * from treesummary where rectype=0;";
char *vtssqluser = "create view vtsummaryuser as select * from treesummary where
rectype=1;";
char *vtssqlgroup = "create view vtsummarygroup as select * from treesummary where
rectype=2;";
```