GUFI Developer's Guide

GUFI Developers

September 11, 2023

Contents

1 License **4**

2 Introduction **6**

3 Requirements **7**

4 Design Considerations **9**

5 Dependencies **11**
- 5.1 System Tools . . . 11
- 5.2 Libraries . . . 11
- 5.3 Packages provided by GUFI . . . 12

6 Build **13**
- 6.1 Environment Variables . . . 13
- 6.2 CMake Flags . . . 13
  - 6.2.1 General . . . 13
  - 6.2.2 Dependencies . . . 14
  - 6.2.3 Debug . . . 14
  - 6.2.4 Client . . . 14
  - 6.2.5 Testing . . . 14
  - 6.2.6 Docs . . . 15

7 Database Schema **16**
- 7.1 `entries` . . . 16
- 7.2 Directory `summary` . . . 16
- 7.3 `pentries` . . . 17
  - 7.3.1 `pentries_rollup` . . . 17
- 7.4 `vrsummary` . . . 17
- 7.5 `vrpentries` . . . 17
- 7.6 `treesummary` . . . 17
- 7.7 `vsummarydir` . . . 18
- 7.8 `vsummaryuser` . . . 18

7.9 `vsummarygroup` . . . 18
7.10 `vtsummarydir` . . . 18
7.11 `vtsummaryuser` . . . 18
7.12 `vtsummarygroup` . . . 18
7.13 `xattrs` . . . 18

8 Indexing **19**
8.1 Directly Indexing a Filesystem . . . 19
8.1.1 `gufi_dir2index` . . . 19
8.2 Indirectly Indexing a Filesystem . . . 20
8.2.1 Trace Files . . . 20
8.2.2 `gufi_dir2trace` . . . 21
8.2.3 `gufi_trace2index` . . . 21
8.3 Extended Attributes . . . 22
8.3.1 Roll In . . . 22
8.3.2 Schema . . . 23
8.3.3 Usage . . . 23
8.4 Location . . . 23
8.4.1 Permissions . . . 23

9 Post-Indexing **24**
9.1 Rollup . . . 24
9.1.1 Bottlenecks . . . 24
9.1.2 Rules . . . 24
9.1.3 Steps . . . 25
9.1.4 Extended Attributes . . . 25
9.1.5 `gufi_rollup` executable . . . 26
9.1.6 Undoing a rollup . . . 26
9.2 Generate `treesummary` Tables . . . 29
9.2.1 `gufi_treesummary` . . . 29
9.2.2 `gufi_treesummary_all` . . . 29
9.2.3 `gufi_rollup` . . . 30

10 Querying **31**
10.1 `gufi_query` . . . 31
10.1.1 Flags . . . 32
10.1.2 Flag/Table Associations . . . 33
10.1.3 Short Circuiting with Compound Queries . . . 33
10.1.4 Extended Attributes . . . 34
10.1.5 Aggregation . . . 34
10.1.6 Per-Thread Output Files . . . 34
10.1.7 Per-Thread Output Database Files . . . 34
10.1.8 Example Calls . . . 35
10.1.9 Behavior When Traversing Rolled Up Indexes . . . 35
10.1.10 Visualizing the Workflow . . . 36
10.2 `querydbs` . . . 37

10.3 SQLite Functions 39
10.4 `gufi_stat_bin` 41
10.4.1 Flags 41

11 Implementation Details 42
11.1 `QueuePerThreadPool` 42
11.1.1 Adding Work 42
11.1.2 Processing Queued Work 42
11.1.3 Usage 43
11.2 `BottomUp` 44
11.3 Printing to `stdout` 44
11.4 Optimizations 45
11.4.1 Reduced Branching 45
11.4.2 Allocations 45
11.4.3 Not Calling `lstat(2)` During Tree Walk 45
11.4.4 Enqueuing Work Before Processing 45
11.4.5 String Manipulations 46
11.4.6 Combining Strings with `memcpy(3)` 46
11.4.7 In-Situ Processing 46
11.4.8 Smaller Enqueued Work Items 46
11.4.9 Compression with zlib 46
11.4.10 Database Templates 47
11.4.11 SQLite 47
11.4.12 Caching Queries (Not Merged) 48

1 License

2 Introduction

Over the years, the amount of data we store and use has grown exponentially to the point that petabytes of storage is not uncommon. What used to be a simple task of accessing and sorting through information has been compounded into an arduous task with the size and scale of super-computing data centers. Being able to query data effectively, while also taking into account user permissions becomes paramount into accomplishing daily tasks. This is what the Grand Unified File Index (GUFI) tool aims to accomplish.

This process of efficiently accessing data is accomplished by recreating the tree structure via indexing. Each directory contains an SQL database file that stores the metadata of the files as well as summary information for that directory and optionally summary information for the entire tree below that directory.

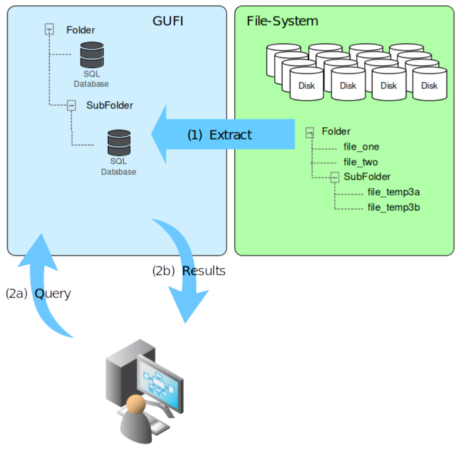


Figure 1: Layout and user interaction with GUFI

3 Requirements

The following requirements were all considered in the design choices made in producing this capability:

- Unified index over multiple heterogeneous storage systems including home, project, scratch, campaign, and archive
- Obey and support full POSIX tree attributes, permissions, hierarchy representation, etc.
- Metadata-only supporting attributes and extended attributes
- Shared index for users and admins
- Fast parallel search capabilities
- Parallel metadata extraction for creating new indexes
- Parallel metadata extraction for incremental updates of existing indexes and ease of using this data to update the index
- Index can live in a separate space or within the source file/storage systems themselves where possible
- Can appear as mounted file system where you get a virtual image of your file metadata based on query input and also a pre-run query can also appear as a mounted file system
- Full/Incremental update from sources with reasonable update time/annoyance
- Provide a way to work in a pure POSIX environment requiring only a POSIX interface to storage systems for full and incremental metadata extraction
- Exploit special interfaces from source file/storages systems provided by some storage systems for mass metadata manipulation/extraction like GPFS ILM
- Be open source software and leverage other open source software and/or open interfaces
- Very transparent and simple so one can easily understand/enhance/administrate with simple to understand formats. Avoid black box anything.
- Extensible capabilities, especially in the query area, for example enable outputs from query to be consumable by humans or other programs and ability to connect in external data sources into query results.

- Ability to store both base POSIX information and potentially source storage system unique metadata per entry.
- The intent is to provide a nearly consistent index (doesn't have to be a perfect snapshot or continuously keeping up with source file/storage systems.
- The intent is not to produce a policy management system for users or admins but of course it could provide the index usable for policy management function.
- Keep the code base small by leveraging existing technology as much as possible both hardware and software including:
  - Flash storage: Assume the index would be small enough to fit in flash storage or even perhaps memory with a few hundred bytes of index per entry. Assume metadata mount per directory will typically be a few kilobytes so parallel access ends up looking like multi-kilobyte random reads which is well suited to flash devices which can provide millions of read IOPS.
  - Both process and thread parallelism: where possible enable both types of parallelism for speed and efficiency
  - A standard and powerful basis for search: enable the exploitation of the power and stability of an existing basis for search even if the interface to the user doesn't export that interface (like SQL or other)
  - Commercial database technology: no need to invent our own underlying indexing/database technology given the abundance of solutions available
  - Commercial file system technology: leverage commercial file system technology and its strengths including extremely fast traversal and access control which is probably of the most optimized code in the world
  - Ppen source software: leverage open interfaces/software where possible agnostic to leveraged parts where possible: if depending on external function where possible enable use of more than one provider of that external function
  - Very transparent and simple so one can easily understand/enhance/administrate.

4 Design Considerations

The following considerations were made to produce a GUFI capability:

- Why not just flatten the entire metadata entry space and shard it and index some fields which enables extremely simple scaling for queries?
  - Events like rename (mv /top/b2 /top/b1) high in the source tree causes potentially billions of records to be updated/replaced.
  - We desire a single parallel index capability used by users and admins which requires POSIX sharing security to be enforced which is complicated by the inheritance that directory read/execute has on the tree below which is one of the powerful concepts of POSIX. This security capability is hard to implement in a flat space due to the tree inheritance feature.
  - A single user can see only very little of the overall metadata space, simple flat sharding requires looking at a lot of records that a tree/graph based approach would eliminate.
  - Sharding is however important to enable parallelism, just simple flat sharding appears to be problematic.
- Leverage things that work very well, ways to reduce the number of records needed to be looked at/updated, etc.
  - Just buy product if it exists and if possible without lock in etc. that does much of what we want. We were unsuccessful in finding the product to buy.
  - The POSIX tree walk (directories only (readdir+)) mechanism - optimized to the extreme, speed, enables breadth parallel fan out, and enforces shared security, enables renames at very low cost
  - Breadth first search parallelizes extremely well especially for threading mechanisms and especially wide namespaces enable rapid parallelization which is common in supercomputing sites like ours
  - SQL is extremely powerful and very stable but monolithic commercial SQL database systems are often expensive and require special knowledge to run. However, user space/embedded SQL databases work very well as long as individual database files are not enormous (<TB) and the application doesn't require a lot of joins. SQLite3 is used heavily in the smart phone business so it is becoming quite ubiquitous. Embedded databases work in POSIX file systems appearing as just files so can obey POSIX security/access control trivially. SQL is an amazingly powerful way to express queries, more powerful than actually needed for this application. SQLite3 is open source, has enormous support, is very fast for this use case, is extensible and allows for connecting to other data sources and outputting about any type of output.

- Flash devices can sustain extremely high IOPs where IOPs are in small numbers of kilobytes or larger.
- Trees are a natural structure for rolling up representative data, so you get indexing function almost for free.
- If you consider just the directories (which provide the shape of the tree) in most large deployed file/storage systems, there is a natural collection of metadata entries (in a single directory) of 20-1000s of entries (files/links) giving a nice way to shard (by directory) which enables parallelism while still honoring POSIX security. In many POSIX file systems, the more files in a single directory the worse the performance, but with an embedded file system file with few-to-no joins, the more entries in that flat file the better as that represents a serial read.
- Embedding database files into a POSIX tree allowed for replicating the source file/storage system metadata entirely to enable an isolated performance domain or placing the index into the source storage/file system itself. This approach also enables lots of choices for the underlying file system to store the index in and provides trivial ways to backup/copy/replicate the index. This approach is trivial to understand and is completely transparent. It also enables the ability for query output to appear as a POSIX file system pretty trivially as well.

5 Dependencies

A number of dependencies that should be installed before attempting to build GUFI. There are also many optional dependencies that can enable optional features.

5.1 System Tools

Package/Executable	Required?	Comments
autotools	Yes	Some bundled packages use autotools
C Compiler	Yes	C99 support
C++ Compiler	No	C++11 support - g++ 4.9.3, clang++-3.9, or newer
CMake	Yes	Version 3.1 or higher
Make	Yes	
Python	Yes	Python 2.7 or Python 3
pip	No	Used for installing and client dependencies
git	No	1.8.5 or higher if using the performance history framework
LaTeX	No	Only needed if building LaTeX Documentation
pkg-config	Yes	
realpath	Yes	
rpmbuild	No	Only needed if building RPMs
setfattr	Yes	attr package `xattr -w` on OSX
truncate	Yes	The -s option must be available
uname	No	Only needed if building RPMs

5.2 Libraries

Name	Required?	Comments
xattr	Yes	
pcre	Yes	Version 1
fuse	No	Called osxfuse on OSX
db2	No	
gpfs	No	
matplotlib	No	Python library used for plotting numbers collected by the performance history framework
zlib	No	

5.3 Packages provided by GUFI

Several packages are provided/downloaded, built, and installed automatically:

Name	Comments
GoogleTest	`https://github.com/google/googletest`
jemalloc	`https://github.com/jemalloc/jemalloc`
paramiko	`https://github.com/paramiko/paramiko` patched ae3d0febef17a8ece5268bbf6c210a30573ce800 Use pip to download python-bcrypt, python-cryptography, and python-pynacl
sqlite3	`https://www.sqlite.org/index.html` patched sqlite-autoconf-3270200.tar.gz
sqlite3-pcre	`https://github.com/mar-file-system/sqlite3-pcre`

6 Build

GUFI uses CMake to generate Makefiles. CMake 3.1 or higher is required. From the root directory of the GUFI source, run:

```
mkdir build
cd build
cmake .. [options]
make
```

The recommended options for deployment are `-DCMAKE_BUILD_TYPE=Release -DCLIENT=On`.

GUFI is known to build on Ubuntu 16.04 (Xenial), Ubuntu 18.04 (Bionic), Ubuntu 20.04 (Focal), Ubuntu 22.04 (Jammy), CentOS 7, CentOS 8, RockyLinux 8, OSX 10.13 (High Sierra), OSX 11 (Big Sur), OSX 12 (Monterey), and OSX 13 (Ventura). Building on Windows with cygwin, GCC (not MinGW), and with jemalloc turned off also works. OpenSUSE 12.3 was supported in the past, and may still work.

6.1 Environment Variables

Setting	Description
`CXX=false`	Disable building of C++ code

6.2 CMake Flags

6.2.1 General

`-D<VAR>=<VALUE>`	Description
`CMAKE_INSTALL_PREFIX=<PATH>`	Install to a custom directory when running `make install`
`CMAKE_BUILD_TYPE=Debug`	Build with warnings and debugging symbols turned on

6.2.2 Dependencies

`-D<VAR>=<VALUE>`	Description
`DEP_DOWNLOAD_PREFIX=<PATH>`	Location of downloaded dependencies. If the expected files are found, they will not be downloaded. The default path points to the bundled dependencies.
`DEP_BUILD_DIR_PREFIX=<PATH>`	Location to build dependencies. Defaults to ${CMAKE_BINARY_DIR}/builds
`DEP_INSTALL_PREFIX=<PATH>`	Location to install the dependencies. Defaults to ${CMAKE_BINARY_DIR}/deps. If the dependencies are not installed in ${CMAKE_BINARY_DIR}, they will not need to be redownloaded, rebuilt, or reinstalled everytime ${CMAKE_BINARY_DIR} is deleted.
`DEP_PATCH_SQLITE3_OPEN=<On|Off>`	Whether or not to patch SQLite3 open
`DEP_USE_JEMALLOC=<On|Off>`	Whether or not to build and link with jemalloc

6.2.3 Debug

`CMAKE_BUILD_TYPE` must be set to `Debug` for these to have effect.

`-D<VAR>=<VALUE>`	Description
`PRINT_CUMULATIVE_TIMES=<On|Off>`	Print cumulative statistics at the end of some executables.
`PRINT_PER_THREAD_STATS=<On|Off>`	Print `gufi_query` event timestamps.
`PRINT_QPTPOOL_QUEUE_SIZE=<On|Off>`	Print size of work queues every time a work queue receives a new item or pops items off.
`GPROF=<On|Off>`	Compile with the `-pg` flag.

6.2.4 Client

`-D<VAR>=<VALUE>`	Description
`CLIENT=<On|Off>`	Whether or not to install paramiko and gufi_client when `make install` is called

6.2.5 Testing

`-D<VAR>=<VALUE>`	Description
`TEST_WORKING_DIRECTORY=<PATH>`	Directory to run tests in. Defaults to `${CMAKE_BINARY_DIR}/test`

Do not turn these off unless you know what you are doing:

-D<VAR>=<VALUE>	Description
`RUN_ADDQUERYFUNCS=<On|Off>`	Compile `gufi_query` without calls to `addqueryfuncs`.
`RUN_OPENDB=<On|Off>`	Compile `gufi_query` without calls to `opendb`.
`RUN_SQL_EXEC=<On|Off>`	Compile `gufi_query` without calls to `sqlite3_exec`.

6.2.6 Docs

-D<VAR>=<VALUE>	Description
`LATEX_BUILD=<On|Off>`	Whether or not to build PDF documentation when LaTeXis found

7 Database Schema

GUFI stores extracted metadata in SQLite3 database files. Each directory contains a database file named db.db that contains a set of tables and views designed to facilitate efficient querying of metadata.

7.1 `entries`

The `entries` table contains file and link metadata extracted from the source filesystem using `stat(1)`, `readlink(2)`, and `llistxattr(2)`. There are also a few OS specific columns (`oss*`) that are unused in base GUFI.

7.2 Directory `summary`

The directory `summary` table contains the metadata of the current directory. Additionally, it contains columns that summarizes the `entries` table located in the same database file, such as minimum and maximum file sizes, uids, and gids. These summary columns can be used to determine whether or not the `entries` table needs to be queried at all.

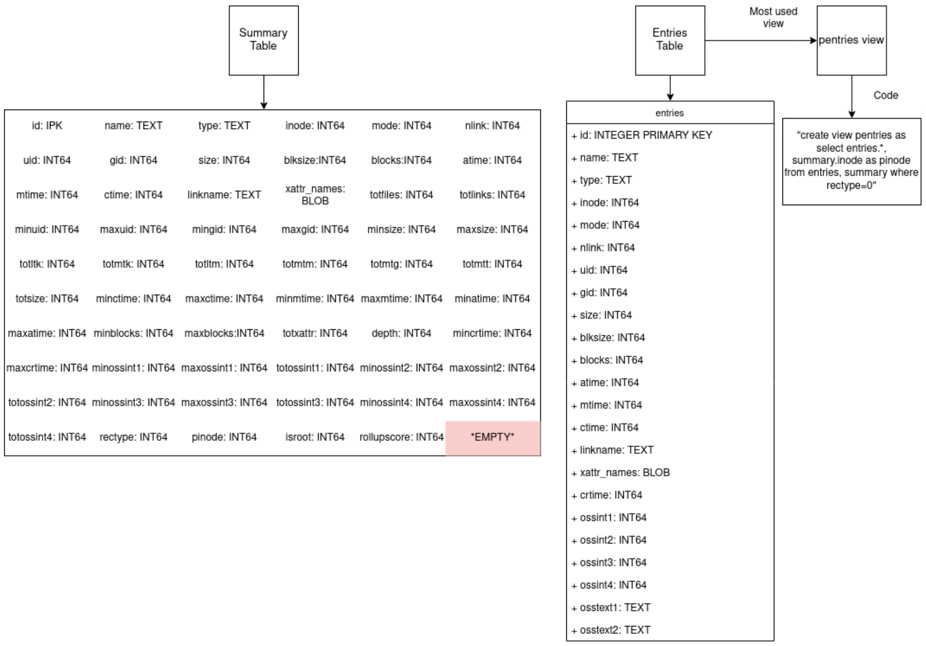


Figure 2: Database schema

7.3 `pentries`

The original `pentries` view was defined as:

```
SELECT entries.*, summary.inode FROM entries, summary;
```

This was done in order to not store the parent inode of each entry, which would be the same for every entry.

In general, users should query the `pentries` view over the `entries` table in order to obtain complete sets of data to query on.

7.3.1 `pentries_rollup`

In order to simplify rolling up indexes (see Section 9.1), the `pentries` view was modified to also union with the `pentries_rollup` table, which contains all child `pentries` views copied into the current directory's database file.

7.4 `vrsummary`

The `vrsummary` view is the `summary` table with a handful of columns repeated as aliases. This was done so that the `rpath` SQL function can be called to generate the path of a given directory with the same view and query whether or not the index has been rolled up, and thus should be preferred over querying the `summary` table. Using the `summary` table directly with a rolled up index is possible, but will complicate queries.

7.5 `vrpentries`

The `vrpentries` view is the `pentries` view with a few `summary` table columns aliased. This allows for the `rpath` SQL function can be called to generate the path of a given directory with the same view and query whether or not the index has been rolled up, and thus should be preferred over querying `entries` and `pentries`. `rpath` with the `vrpentries` view is called with the same arguments that are used with `vrsummary`. Using the `pentries` view directly with a rolled up index is possible, but will complicate queries.

7.6 `treesummary`

Similar to the directory `summary` table, GUFI also provides functionality to generate `treesummary` tables. Instead of summarizing only the data found in the entries table of the current directory, the `treesummary` table summarizes the contents of the entire subtree, allowing for queries to completely skip processing entire subtrees. See Section 9.2 for details.

7.7 `vsummarydir`

The `vsummarydir` view provides access to the entire directory summary and not a partial directory summary (say by user or group).

7.8 `vsummaryuser`

The `vsummaryuser` view provides access to the directory summary for each user (if this summary has been populated (not by default but easily populatable via a query)).

7.9 `vsummarygroup`

The `vsummarygroup` view provides access to the directory summary for each group (if this summary row has been created (not by default but easily created via a query)).

7.10 `vtsummarydir`

The vtsummarydir view provides access to the entire tree directory summary and not a partial directory summary (say by user or group).

7.11 `vtsummaryuser`

The `vtsummaryuser` view provides access to the tree directory summary for each user (if this summary row has been created (not by default but easily created via a query)).

7.12 `vtsummarygroup`

The `vtsummarygroup` view provides access to the tree directory summary for each group (if this summary has been populated (not by default but easily populatable via a query)).

7.13 `xattrs`

With the addition of extended attributes support, several more tables and views were added into db.db. Additional database files with different schemas were also added. See Section 8.3.2 for details.

8 Indexing

The first step to using GUFI is indexing a source filesystem.

An index created by GUFI retains the shape of the source filesystem: directories that exist in the source filesystem also exist in the index. If an administrator created the index, the directories will also have the same access permissions, uid, and gid.

Indexes will not contain any of the files that the source filesystem contained. Instead, all metadata extracted from a single directory of the source filesystem will be placed into a single database file, called db.db, in the cooresponding directory in the index. Each database file will be created with a fixed schema that includes the tables listed in Sections 7 and 8.3.2. Additional database files may be created if extended attributes are extracted (see Section 8.3).

Index processing (creation) occurs on a per-directory basis, and thus is highly parallelizable.

8.1 Directly Indexing a Filesystem

8.1.1 `gufi_dir2index`

`gufi_dir2index` is used to directly create an index based off of the contents of a provided directory.

Flag	Functionality
-h	help manual
-H	Show assigned input values
-n <num_threads>	define number of threads to use
-x	pull xattrs from source file-sys into GUFI
-z <max_level>	maximum level to go down to
-k <filename>	file containing directory names to skip
-M <bytes>	target memory footprint
-C <count>	Number of subdirectories allowed to be enqueued for parallel processing. Any remainders will be processed in-situ
-e	compress work items

Table 1: `gufi_dir2index` Flags and Arguments

8.1.1.1 Usage

```
gufi_dir2index [flags] input_dir... output_dir
```

The index of `input_dir`$_i$ will be placed in `output_dir/$(basename input_dir`$_i$`)`.

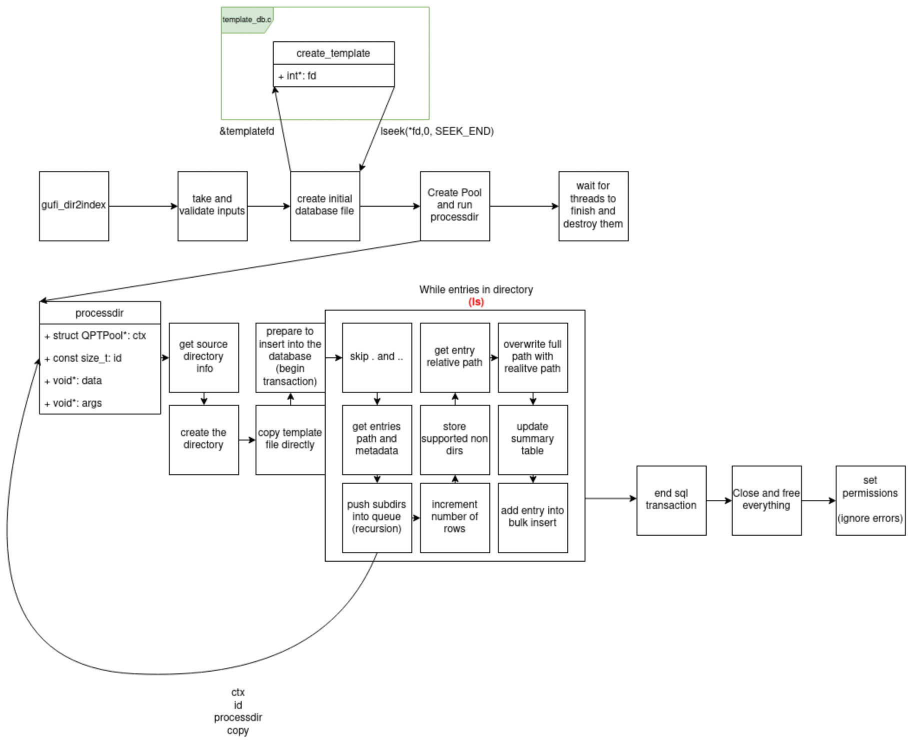

template_db.c
create_template
+ int*: fd
&templatefd
lseek(*fd,0, SEEK_END)
gufi_dir2index
take and validate inputs
create initial database file
Create Pool and run processdir
wait for threads to finish and destroy them
processdir
+ struct QPTPool*: ctx
+ const size_t: id
+ void*: data
+ void*: args
get source directory info
create the directory
prepare to insert into the database (begin transaction)
copy template file directly
While entries in directory
(ls)
skip . and ..
get entry relative path
overwrite full path with realitve path
get entries path and metadata
store supported non dirs
update summary table
push subdirs into queue (recursion)
increment number of rows
add entry into bulk insert
end sql transaction
Close and free everything
set permissions
(ignore errors)
ctx
id
processdir
copy

Figure 3: `gufi_dir2index` workflow

8.2 Indirectly Indexing a Filesystem

8.2.1 Trace Files

Traces files, or flat files, are text files containing `lstat(2)` information pulled from source filesystems separated by delimiters. Trace files are generated by `gufi_dir2trace` and are processed by `gufi_trace2index`.

The default delimiter is the ASCII Record Separator character `\x1E`, but can be changed to any 8-bit character. Characters that can appear in filesystems should not be used as the delimiter in order to not confuse `gufi_trace2index`.

The metadata pulled from each directory is represented by a "stanza" in a trace file. A stanza starts with a line of directory metadata followed by zero or more lines of regular file and symbolic link metadata. Note that the summary of each directory is not stored in trace files and are instead generated when the index is built.

Due to the parallelism of indexing, stanzas can appear in any order within a trace file and can be placed in any of the per thread trace files that are generated. Traces files can be concatenated together if doing so is necessary or more convenient.

8.2.2 `gufi_dir2trace`

`gufi_dir2trace` generates trace files to allow for indexes to be easily transfered to different locations rather than requiring entire trees to be copied around.

Flag	Functionality
-h	help manual
-H	Show assigned input values
-n <num_threads>	define number of threads to use
-x	pull xattrs from source file-sys into GUFI
-d <delim>	delimiter (one char) [use 'x' for 0x1E]
-k <filename>	file containing directory names to skip
-M <bytes>	target memory footprint
-C <count>	Number of subdirectories allowed to be enqueued for parallel processing. Any remainders will be processed in-situ
-e	compress work items

Table 2: `gufi_dir2trace` Flags and Arguments

8.2.2.1 Usage

```
gufi_dir2trace [flags] input_dir... output_prefix
```

Trace files with the name `output_prefix.${i}`, where i ∈ [0, number of threads), will be created.

8.2.3 `gufi_trace2index`

`gufi_trace2index` is used to convert trace files into indexes. It is essentially the same as `gufi_dir2index` except it obtains data from trace files instead of the filesystem being indexed.

Per thread trace files may be passed into `gufi_trace2index` directly. Note that passing in too many trace files at once might result in running out of file descriptors. Alternatively, the per thread trace files may be concatenated in any order into a smaller number of larger files for processing.

Extended attributes will be processed if they are found in the traces. There is no need to tell `gufi_trace2index` to process them with `-x`.

8.2.3.1 Usage

```
gufi_trace2index [flags] trace_file... index_root
```

Each source filesystem found in the trace files will be converted to an index placed underneath `output_dir`.

Flag	Functionality
-h	help manual
-H	Show assigned input values
-n <num_threads>	define number of threads to use
-d <delim>	delimiter (one char) [use 'x' for 0x1E]
-M <bytes>	target memory footprint

Table 3: `gufi_dir2trace` Flags and Arguments

8.3 Extended Attributes

GUFI supports the indexing and querying of extended attributes (xattrs).

Reading standard filesystem permissions of files only requires read (and execute) access to the directory. Extended attribute names are visible this way. However, because xattr values are user defined data, their permissions are checked at the file level, requiring changes to how GUFI stores data. For more information on xattrs, see `xattr(7)`, `llistxattr(2)`, and `lgetxattr(2)`.

Directories containing files from multiple users might have xattrs that are not readable by all who can view the `lstat(2)` data of the directory. As GUFI was originally designed to only use directory-level permission checks, a number of modifications were made to process xattrs without violating their permissions.

8.3.1 Roll In

Extended attributes that are readable by all who have access to the directory are stored in the `xattrs_pwd` table in the main database. These xattrs are referred to as "rolled in".

The rules that determine whether or not an xattr pair can roll in are as follows:

- File is 0+R
- File is UG+R doesnt matter on other, with file and parent same usr and grp and parent has only UG+R with no other read
- File is U+R doesnt matter on grp and other, with file and parent same usr and parent dir has only U+R, no grp and other read
- Directory has write for every read: `drw*rw*rw*` or `drw*rw*___` or `drw*______` - if you can write the dir you can chmod the files to see the xattrs

Extended attributes that cannot be read by all who can read the directory are stored in external per-uid and per-gid databases set with `uid:nobody` and `nobody:gid` owners respectively. This makes it so that non-admin users cannot access the xattrs stored in external databases that they do not have permissions to access.

8.3.2 Schema

The main database and all external databases contain the following tables and views with the 3 columns `inode`, `name`, and `value`:

- The `xattrs_pwd` table contains all extended attributes of the current directory that were placed into this database file.
- The `xattrs_rollup` table contains all extended attributes that were placed into the children directories that were subsequently rolled up into the current directory.
- The `xattrs_avail` view is the union of all extended attributes in `xattrs_pwd` and `xattrs_rollup` in this database file.

Additionally, each main database file has the following tables and views in order to keep track of which files were created by GUFI for the purposes of storing xattrs:

- The `xattrs_files_pwd` table contains a listing of external database filenames that contain xattrs that were not rolled in.
- The `xattrs_files_rollup` table contains a listing of external database filenames that contain xattrs that were not rolled in, but were brought in by rolling up.
- The `xattrs_files` view combines the listings of database filenames found in `xattrs_files_pwd` and `xattrs_files_rollup`.

8.3.3 Usage

Extended attributes are not pulled from the filesystem by default. In order to pull them, pass `-x` to `gufi_dir2index` or `gufi_dir2trace`.

Note that only xattr pairs in the user namespace (`user.*`) are extracted.

8.4 Location

GUFI is expected to be used to query the indexes of many filesystems all at once. However, the source filesystems are not expected to be accessible from each other. In order for the indexes from many disconnected systems to be queried at once, they should all be built under a common directory on a single machine.

8.4.1 Permissions

If a GUFI tree is located on a different machine than the source tree, the users and groups are probably not available on the machine with the index on it. `/etc/passwd` should be populated with the entries from the source machine and modified so that they do not have a home directory, and are not allowed to log in (`/sbin/nologin`). Similarly, `/etc/group` should be updated with the source machine's groups and modified to remove any unnecessary information.

9 Post-Indexing

After indexing, additional operations can be performed on the index in order to increase query performance.

9.1 Rollup

Rolling up is a major optimization that can reduce the number of directories that need to be opened when querying by significant amounts while still obtaining the same results as an unmodified index.

9.1.1 Bottlenecks

During querying, every single directory runs a fixed set of operations:

- `opendir(3)`
- `readdir(3)` (Looped)
- `sqlite3_open_v2`
- `sqlite3_exec_v2`
- `sqlite3_close(3)`
- `closedir(3)`

Each of these operations have costs associated with them. Some are fixed and some depend on the shape and contents of the index. Reducing the number of directories processed allows for fixed costs to be amortized.

9.1.2 Rules

9.1.2.1 Whether or not a directory *CAN* roll up

In order for a child directory to be allowed to roll up into its parent directory, it must follow two rules. First, all of its children must be rolled up. Second, all of its children must have permissions that satisfy any one of the following conditions with the parent.

- World readable and executable (o+rx)
- Matching user, group, and others permissions, with the same user and group
- Matching user and group permissions, readable and executable (ug+rx) with the same user and group, and not world readable and excutable (o-rx)
- Matching user permissions, readable and executable (u+rx) with the same user and not group or world readable and executable (go-rx)

Note that because leaf directories have no children with which to have conflicting permission with, they are considered rolled up.

9.1.2.2 Whether or not a directory *SHOULD* roll up

In addition to the permission-based rules that determine whether or not a directory can roll up, we also have a small check to say whether or not a directory should roll up.

As data is rolled upwards in the index, databases towards the top will accumulate more and more data from its subtrees. If directories are allowed to roll up without limit, some databases will become significantly larger than others, causing large amounts of tail latency.

The `gufi_rollup` executable has the `-L` flag that limits the number of entries that may be found within a single directory.

9.1.3 Steps

The processing of rolling up involves a number of steps that update the tables of the parent directory.

First, the target directory is checked to ensure it can and should roll up its children into itself using the rules listed in 9.1.2. If rollup will proceed, the parent's `summary` table is updated with a rollup score of 1.

Then, the contents each child is copied into the parent:

1. The child's `pentries` view is copied into the parent's `pentries_rollup` table. The parent's `pentries` view is updated automatically.
2. The child's `summary` table is copied into the parent's `summary` table with the names of each child directory prefixed with the parent's name.

Rolling up can be viewed as flattening the contents of the index while taking into consideration the permissions of each directory.

Rolling up will cause the size of the index to grow significantly due to the amount of data being replicated. One obvious optimization would be to only roll up to the top-most level where a directory can roll up to (one extra copy of the subtree instead of repeated copies). This however, will only allow for queries to take advantage of rollups if they start above the top-most rollup directory. Starting a query below the top-most rollup level would result in the original subtree's query time whereas the implemented method of rolling up allows for queries starting at any point in the index to take advantage of rollups.

The rollup operation can take some time, and so indexes are not rolled up automatically. The `gufi_rollup` executable must be called manually.

9.1.4 Extended Attributes

Additional steps are needed to rollup xattrs:

- The child's `xattrs_avail` view (without external database data) is copied into the parent's `xattrs_rollup` table.
- The child's `xattrs_files` view (without external databases data) is copied into the parent's `xattrs_files_rollup` table.

- The child's external database files are copied into the parent. If the parent already has an external database file with the same uid or gid, the contents of the external database are copied into the parent's external database's `xattrs_rollup` table instead.

9.1.5 `gufi_rollup` executable

In order to apply rollup to an index, run the `gufi_rollup` executable:

```
gufi_rollup index_root
```

Figure 4 shows the overall structure of the `gufi_rollup` executable. `gufi_rollup` recursively descends the index and performs the rollup operation on a directory once all of the directory's children have been processed as shown in Figure 5.

9.1.6 Undoing a rollup

To remove rollup data from an index, run the `gufi_unrollup` executable:

```
gufi_unrollup index_root
```

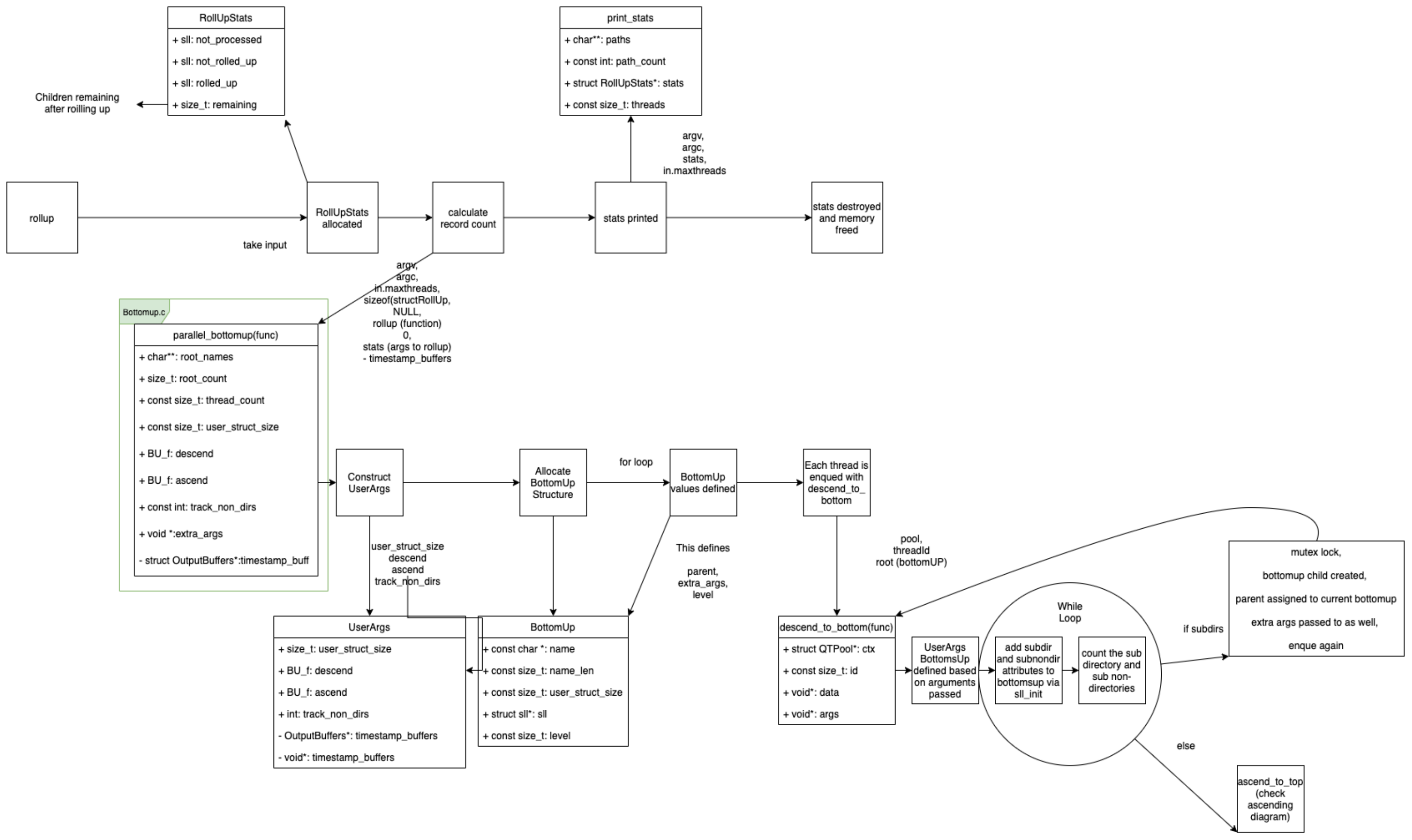
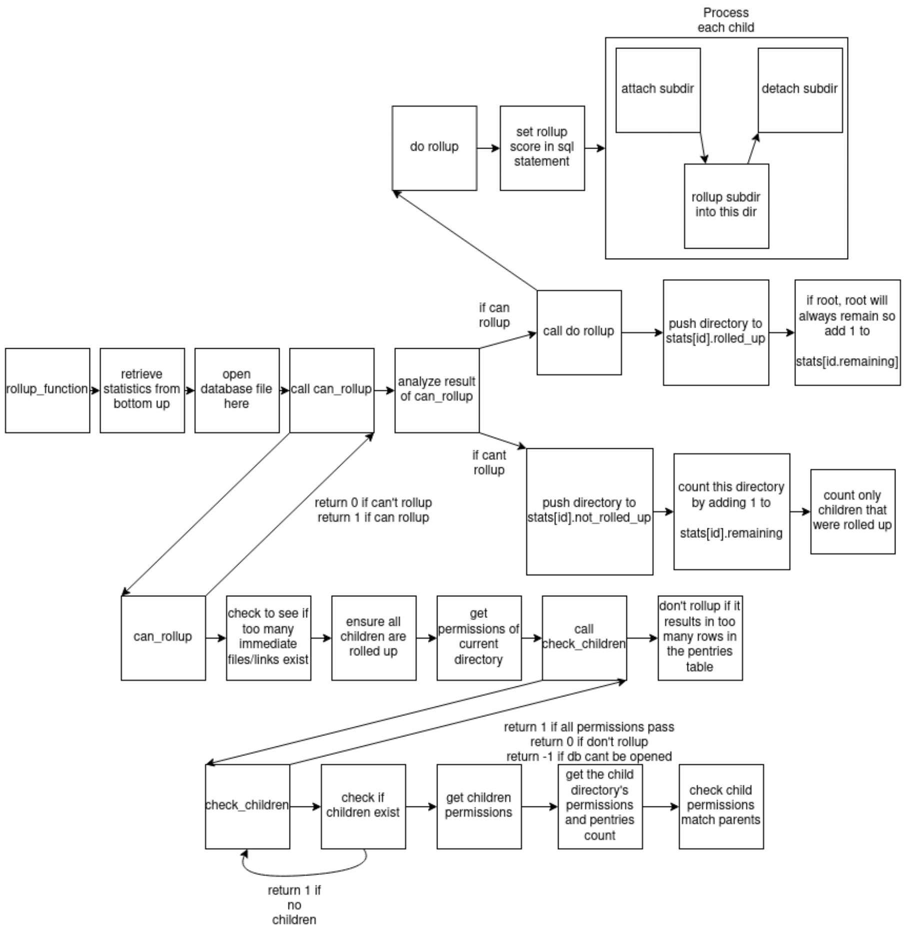

rollup
take input
RollUpStats allocated
RollUpStats
+ sll: not_processed
+ sll: not_rolled_up
+ sll: rolled_up
+ size_t: remaining
Children remaining after roilling up
calculate record count
argv, argc, in.maxthreads, sizeof(structRollUp, NULL, rollup (function) 0, stats (args to rollup) - timestamp_buffers
stats printed
argv, argc, stats, in.maxthreads
print_stats
+ char**: paths
+ const int: path_count
+ struct RollUpStats*: stats
+ const size_t: threads
stats destroyed and memory freed
Bottomup.c
parallel_bottomup(func)
+ char**: root_names
+ size_t: root_count
+ const size_t: thread_count
+ const size_t: user_struct_size
+ BU_f: descend
+ BU_f: ascend
+ const int: track_non_dirs
+ void *:extra_args
- struct OutputBuffers*:timestamp_buff
Construct UserArgs
user_struct_size descend ascend track_non_dirs
UserArgs
+ size_t: user_struct_size
+ BU_f: descend
+ BU_f: ascend
+ int: track_non_dirs
- OutputBuffers*: timestamp_buffers
- void*: timestamp_buffers
Allocate BottomUp Structure
BottomUp
+ const char *: name
+ const size_t: name_len
+ const size_t: user_struct_size
+ struct sll*: sll
+ const size_t: level
for loop
BottomUp values defined
This defines parent, extra_args, level
Each thread is enqued with descend_to_ bottom
pool, threadId root (bottomUP)
descend_to_bottom(func)
+ struct QTPool*: ctx
+ const size_t: id
+ void*: data
+ void*: args
UserArgs BottomsUp defined based on arguments passed
While Loop
add subdir and subnondir attributes to bottomsup via sll_init
count the sub directory and sub non-directories
if subdirs
mutex lock,
bottomup child created,
parent assigned to current bottomup
extra args passed to as well,
enque again
else
ascend_to_top (check ascending diagram)

Figure 4: gufi_rollup Workflow



Process each child
attach subdir
detach subdir
rollup subdir into this dir
do rollup
set rollup score in sql statement
if can rollup
call do rollup
push directory to stats[id].rolled_up
if root, root will always remain so add 1 to stats[id.remaining]
rollup_function
retrieve statistics from bottom up
open database file here
call can_rollup
analyze result of can_rollup
if cant rollup
push directory to stats[id].not_rolled_up
count this directory by adding 1 to stats[id].remaining
count only children that were rolled up
return 0 if can't rollup
return 1 if can rollup
can_rollup
check to see if too many immediate files/links exist
ensure all children are rolled up
get permissions of current directory
call check_children
don't rollup if it results in too many rows in the pentries table
return 1 if all permissions pass
return 0 if don't rollup
return -1 if db cant be opened
check_children
check if children exist
get children permissions
get the child directory's permissions and pentries count
check child permissions match parents
return 1 if no children

Figure 5: Rollup Function Diagram

9.2 Generate treesummary Tables

The `treesummary` table is an optional table that is placed into db.db. It contains a summary of the entire subtree starting at current directory using minimums, maximums, and totals of numerical values.

When a query is provided to `gufi_query -T`, the `treesummary` table is queried first. Because `treesummary` tables do not necessarily exist, `gufi_query` first checks for the existence of the `treesummary` table in the directory being processed before performing the `-T` query. If the `-T` query returns no results, the entire subtree will be skipped.

9.2.1 gufi_treesummary

Starting from a directory provided in the command line, `gufi_treesummary` recursively traverses to the bottom of the tree, collecting data from the `summary` table of each child directory database. If a `treesummary` table is discovered in a subdirectory, descent down the tree is stopped as the `treesummary` table contains all of the information about that directory as well as all subdirectories underneath it. Once all of the data has been collected, it is summarized and placed into the `treesummary` table of the starting directory.

Generating `treesummary` tables for all directories using this top-down approach will take a long time due to the repeated traversals across the same directories. Because of this, `gufi_treesummary` generates the `treesummary` table for the provided directory only.

If generating `treesummary` tables using `gufi_treesummary`, the tables should be generated at optimal points within the index. For example, if the index is on a home directory, it may be useful to generate `treesummary` tables at each user's home directory.

Example Call:

```
gufi_treesummary index_root
```

9.2.2 gufi_treesummary_all

`gufi_treesummary_all` generates `treesummary` tables for all directories. This is done by walking to the bottom of the tree and generating `treesummary` tables while walking back up, which only occurs after all subdirectories have had their `treesummary` tables generated. Leaf directories, by definition, do not have subdirectories, and further traversal down is unnecessary and impossible. Their `treesummary` tables are thus duplicates of their `summary` tables, providing the base case for the walk back up the tree. Directories above the leaves can then use the `treesummary` data found in their immediate subdirectories, which are 1) guaranteed to exist and 2) guaranteed to summarize the entire subdirectory's subtree, to generate their own `treesummary` tables.

Example Call:

```
gufi_treesummary_all index_root
```

9.2.3 `gufi_rollup`

Just as with `gufi_treesummary_all`, rolling up a tree involves walking to the bottom of the tree and working upwards. This allows for `treesummary` generation to be performed automatically during the roll up operation, resulting in `treesummary` tables being generated for all directories whether or not they were rolled up.

10 Querying

Once a GUFI tree with one or more indexes has been created, it should be queried using GUFI tools and SQL statements.

10.1 `gufi_query`

`gufi_query` is the main tool used for accessing indexes.

`gufi_query` processes each directory in parallel, passing user provided SQL statements to the current directory's database. Because of this, callers will need to know about GUFI's database and table schemas. We expect only "advanced" users such as administrators and developers to use `gufi_query` directly. Users who do not know how GUFI functions may call `gufi_query`, but might not be able to use it properly.

10.1.1 Flags

Flag	Functionality
-h	help
-H	show assigned input values (debugging)
-E <SQL ent>	SQL for entries table
-S <SQL sum>	SQL for summary table
-T <SQL tsum>	SQL for tree-summary table
-a	AND/OR (SQL query combination)
-n <threads>	number of threads (default: 1)
-j	print the information in terse form
-o <out_fname>	output file (one-per thread, with thread-id suffix)
-d <delim>	one char delimiter (default: `\x1E`)
-O <out_DB>	output DB
-I <SQL_init>	SQL init
-F <SQL_fin>	SQL cleanup
-y <min-level>	minimum level to descend to
-z <max-level>	maximum level to descend to
-J <SQL_interm>	SQL for intermediate results (no default. ex.: `INSERT INTO <aggregate table name> SELECT * FROM <intermediate table name>`)
-K <create aggregate>	SQL to create the final aggregation table
-G <SQL_aggregate>	SQL for aggregated results (no default. ex.: `SELECT * FROM <aggregate table name>`)
-m	Keep mtime and atime same on the database files
-B <buffer size>	size of each thread's output buffer in bytes
-w	open the database files in read-write mode instead of read only mode
-x	enable xattr processing
-k	file containing directory names to skip
-M <bytes>	target memory footprint
-e	compress work items

Table 4: `gufi_query` Flags and Arguments

10.1.2 Flag/Table Associations

A `gufi_query` call should use at least one of the `-T`, `-S`, and `-E` flags to pass in SQL statements[1].

- `-T` should be used to query the `treesummary` table.
- `-S` should be used to query the `summary` table and its variants.
- `-E` should be used to query the `entries` table and its variants.

Note that user provided SQL statements are passed directly into SQLite[2] and thus associations between flags and tables are not enforced[3].

Prior to rolling up, the `pentries` view can be treated as an optional improvement to the `entries` table. After rolling up, the `pentries` view will have been updated extensively and will contain both the original `entries` data as well as the rolled up data. Querying the `entries` table of a rolled up index will return a subset of the total results that exist in the index since it only contains the current directory's information and subdirectories are not traversed. The `summary` table was updated directly, so querying it will return all directory information.

However, instead of querying the `summary` table and switching between querying `entries` and `pentries`, the `vrpentries` and `vrsummary` views should always be used. These views contain data from the `summary` table that allow for the path of each row relative to the starting path to be generated using the `rpath` SQL function, whether or not the index has been rolled up, with consistent input arguments (see Table 8 for details). These views and function were set up to help simpliy the queries that users provide.

10.1.3 Short Circuiting with Compound Queries

At each directory, the set of user provided SQL statements run on the db.db file in the following order: if `-T` was provided, it will run if the optional `treesummary` table exists in the database. If `-T` was not run or returned at least one row of results, `-S` will run if it was provided. If `-S` was not provided or returned at least one row of results, `-E` will run if it was provided.

Conversely, if `-T` runs and does not return any results, processing is stopped before running the `-S` query. If `-S` does not return any results, the `-E` query is not run. This allows for threads to short circuit the processing of a directory if it is determined early on that no rows would be obtained from a later query. For example, if a query is searching for files larger than 1MB, but the `-S` query found that the range of file sizes is 1KB to 5KB, there is no need to run the `-E` query to get rows, since no rows will match in any case.

[1]Not passing in any SQL statements results in `gufi_query` simply walking the tree and filling up the inode and dentries caches.

[2]Anything that can be done with SQL can also be done on the databases in an index. To prevent accidental modifications, databases default to opening in read-only mode.

[3]Querying tables/views with the wrong flags may result in unexpected output.

To turn off short circuiting and always run all queries for each directory, pass the `-a` flag to `gufi_query`.

10.1.4 Extended Attributes

When querying for xattrs, pass `-x` to `gufi_query` to build the `xattrs` view for querying. This view is a SQL union of rolled in xattrs and any external databases that successfully attaches. Attaching the database files checks the permissions of the xattrs. `UNION` removes duplicate entries that showed up in multiple external databases, leaving a unique set of xattrs accessible by the caller.

The `xentries`, `xpentries`, `xsummary`, `vrxpentries`, and `vrxsummary` views are convenience views generated so that users do not have to perform joins themselves. They are `entries`, `pentries`, `summary`, `vrpentries`, and `vrsummary` enhanced with the `xattr_name` and `xattr_value` columns. Like the `xattrs` view, these views are also dropped at the end of directory processing.

Note that entries with multiple extended attributes will return multiple times in this view.

10.1.5 Aggregation

There are cases where independent per-thread results are not desirable, such as when sorting or summing, where the results from querying the index must be aggregated for final processing.

In order to handle these situations, the `-I` flag should be used to create per-thread intermediate tables that are written to by `-T`, `-S`, and `-E`. The intermediate table results will then be aggregated using `-J` into the final table created by `-K`. The rows stored in the final table are processed one last time as a whole, rather than as results from independent threads, using `-G`.

10.1.6 Per-Thread Output Files

The `-o` flag causes results to be outputted to text files. When outputting in parallel, per-thread output files are created with the thread id appended to the filename provided with the flag. When aggregating, the aggregate results are written to the filename specified by `-o` with no filename modifications.

10.1.7 Per-Thread Output Database Files

The `-O` flag allows for results to be written to SQLite database files instead of text files. The resulting filenames follow the same creation rules as `-o`. However, the queries passed into `gufi_query` need modifications. When writing in parallel, `-I` is needed to create the table for each per-thread output database. `-T`, `-S`, and `-E` should be modified to write to the per-thread tables in the same way as writing to the intermediate tables when aggregating. When writing aggregate results to a database, `-G` is not needed as `-J` already wrote the results into the aggregate table. However, `-G` may still be provided to get results during the `gufi_query` in addition to queries on the results database file later on.

Output database files may be passed to the `querydbs` executable for further processing.

10.1.8 Example Calls

Table 5: Parallel Results

Output	`gufi_query /path/to/index`
`stdout`	`-S "SELECT * FROM vrsummary;"`

Per-thread Files	`-E "SELECT * FROM vrpentries;"`
	`-o results`

Per-thread Database Files	`-I "CREATE TABLE results(name TEXT, size INT64);"`
	`-S "INSERT INTO results SELECT name, size FROM vrsummary;"`
	`-E "INSERT INTO results SELECT name, size FROM vrpentries;"`
	`-O results`

Table 6: Aggregate Results

Output	`gufi_query /path/to/index`

`stdout`	`-I "CREATE TABLE intermediate(name TEXT, size INT64);"`
	`-S "INSERT INTO intermediate SELECT name, size FROM vrsummary;"`
	`-K "CREATE TABLE aggregate(name TEXT, size INT64);"`
	`-J "INSERT INSERT aggregate SELECT * FROM intermediate;"`
	`-G "SELECT * FROM aggregate ORDER BY size DESC;"`

Single File	`-I "CREATE TABLE intermediate(name TEXT, size INT64);"`
	`-E "INSERT INTO intermediate SELECT name, size FROM vrpentries;"`
	`-K "CREATE TABLE aggregate(name TEXT, size INT64);"`
	`-J "INSERT INSERT aggregate SELECT * FROM intermediate;"`
	`-G "SELECT * FROM aggregate ORDER BY size DESC;"`
	`-o results`

Single Database File	`-I "CREATE TABLE intermediate(name TEXT, size INT64);"`
	`-S "INSERT INTO intermediate SELECT name, size FROM vrsummary;"`
	`-E "INSERT INTO intermediate SELECT name, size FROM vrpentries;"`
	`-K "CREATE TABLE aggregate(name TEXT, size INT64);"`
	`-J "INSERT INSERT aggregate SELECT * FROM intermediate;"`
	`-O results`

10.1.9 Behavior When Traversing Rolled Up Indexes

When `gufi_query` detects that a directory being processed has been rolled up, the thread processing that directory does not enqueue work to descend further down into the tree, as all of the data underneath the current directory is available in the current directory. While this may seem to only reduce tree traversal by "some" amount, in practise, rolling up indexes of real filesystems reduces the number of directories (and thus the bottlenecks listed in Section 9.1.1) that need to be processed by over 99%, significantly reducing index query time.

Queries are not expected to have to change too often when switching between unmodified indexes and rolled up indexes. If they do, they should not have to change by much.

10.1.10 Visualizing the Workflow

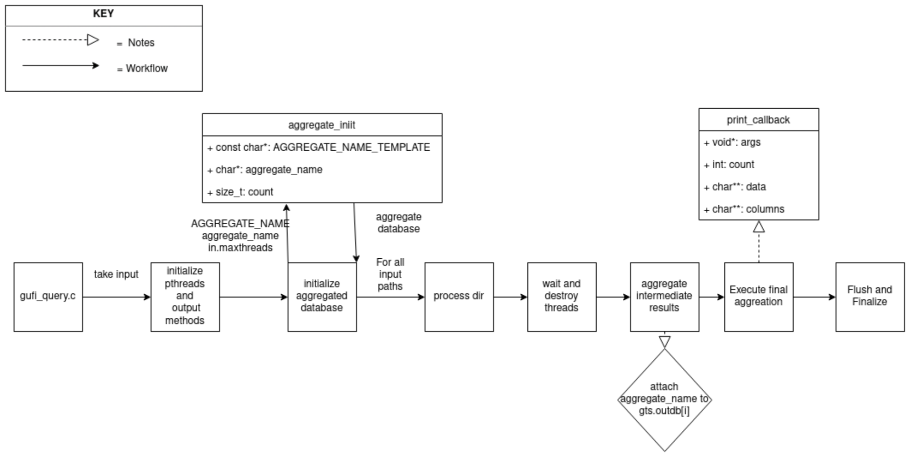


Figure 6: Workflow of `gufi_query`

10.1.10.1 `processdir` The core of `gufi_query` is the `processdir` function. This is where the -T, -S, and -E flags are processed. Multiple instances of this function are run in parallel via the thread pool in order to quickly traverse and process an index.

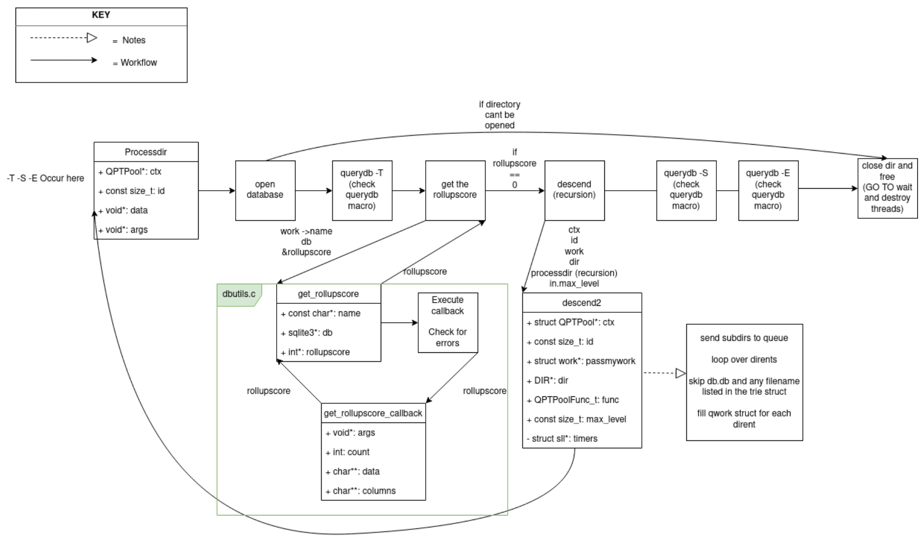


Figure 7: Workflow of `processdir`

10.1.10.2 `querydb` The `querydb` macro is used to execute SQL statements and handle errors.

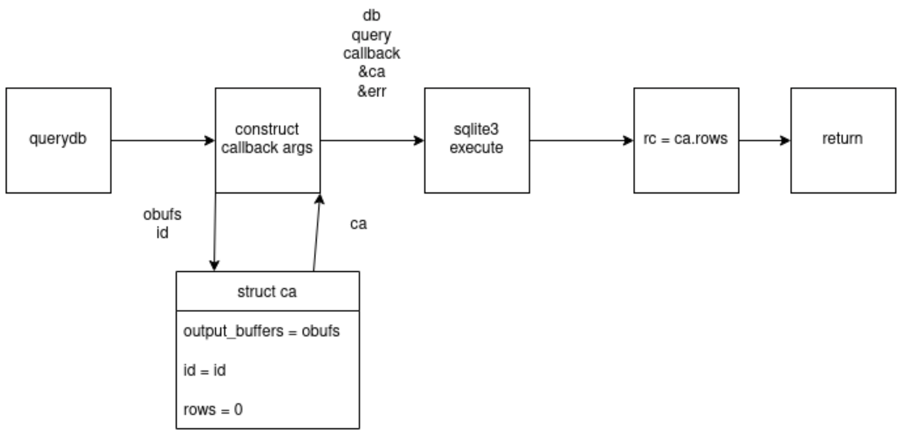


Figure 8: `querydb` macro workflow

10.2 querydbs

`querydbs` attaches all database files it is given into a single view for querying. All tables in the provided database files should have the same name and schema. For tables called `table`, the view combining all tables will be named `vtable`.

This was intended to allow for easy post-processing of `gufi_query -O` output database files.

`querydbs` does not have a `gufi_` prefix because it will work for any set of input databases with the same schema. For example, `querydbs` can also be used directly on individual database files from an index.

Note that there is a hard limit of 125 databases that can be queried by querydbs that is set by SQLite3.

10.3 SQLite Functions

Several convenience functions are added into each database instance opened for querying.

Table 7: SQLite functions that are available in `gufi_query` and `querydbs`.

Function	Purpose
`uidtouser(uid)`	Converts a UID to a user name
`gidtogroup(gid)`	Converts a GID to a group name
`modetotxt(mode)`	Converts numerical permission bits to text
`strftime(format, timestamp)`	Replaces SQLite's custom `strftime` with `strftime(3)`
`blocksize(bytes, unit)`	Converts a size to a the number of blocks of size `unit` needed to store the first argument. `unit` is the combination of at least one integer and/or a prefix and a suffix: Prefix: `K`, `M`, `G`, `T`, `P`, `E` Suffix (multiplier): *no suffix* (1024), `B` (1000), `iB` (1024) Return value is a string. Note that this function was implemented to replicate `ls` output and is meant for use with `gufi_ls` only, so use with caution. Examples:

Call	Output
blocksize(1024, '1000')	2
blocksize(1024, '1024')	1
blocksize(1024, 'K')	1K
blocksize(1024, '1K')	1
blocksize(1024, 'KB')	2KB
blocksize(1024, '1KB')	2
blocksize(1024, 'KiB')	1KiB
blocksize(1024, '1KiB')	1

`human_readable_size(bytes)`	Converts a size to a human readable string

Table 8: SQLite functions that require the context of an index and thus are only available in `gufi_query`.

Function	Purpose
`path()`	Current directory relative to path passed into executable
`epath()`	Current directory basename
`fpath()`	Full path of current directory

`rpath` (sname, sroll)	Current directory relative to path passed into executable taking into account the rolled up name in summary table and rollup score. Should only be used with the `sname` and `sroll` columns of the `vrpentries`, `vrsummary`, `vrxpentries`, and `vrxsummary` views.
	
	Usage:
	`SELECT rpath(sname, sroll)`
	`FROM vrsummary/vrxsummary;`
	
	`SELECT rpath(sname, sroll) || "/" || name`
	`FROM vrpentries/vrxpentries;`

`level()`	Depth of the current directory from the starting directory
`starting_point()`	Path of the starting directory
`subdirs_walked()`	Number of subdirectories that were enqueued while processing the current directory. This function will return 0 if the current directory was rolled up. Only available in `-S` and `-E`. Not recommended for use with `-E`.

10.4 `gufi_stat_bin`

`gufi_stat` is a script that can be used to extract individual entries from an index like `stat(1)`. However, it is not a wrapper for `gufi_query`. Instead, it is a wrapper for `gufi_stat_bin`, a compiled executable. `gufi_stat_bin` does not use the configuration file and thus needs to be provided the search path.

10.4.1 Flags

Option	Description
`-f <FORMAT>`	use the specified FORMAT instead of the default. A newline is outputted after each use of FORMAT.
`-j`	print the information in terse form; Same as `-t`/`--terse` in `stat(1)`.

11 Implementation Details

In addition to glue code that combines the various well known technologies used to implement GUFI, there exists in the codebase several pieces of code, design decisions, and optimizations that have made GUFI performant.

11.1 `QueuePerThreadPool`

Thread pools are generally written with a single work thread (with a single lock) from which to threads pull from. This does not work past a handful of threads. GUFI can run on hundreds of threads, and thus required a more performant thread pool. The solution to this was `QueuePerThreadPool`.

`QueuePerThreadPool` is named so because originally, there was one work queue (and lock) maintained for each thread in the thread pool. Now, there are multiple work queues maintained for each thread.

11.1.1 Adding Work

By default, work is added to threads in a round robin fashion in order to distribute work evenly and to attempt to prevent, or at least reduce the amount of, contention experienced by any one work queue.

This function can be changed during initialization.

11.1.2 Processing Queued Work

Because work items are enqueued in parallel while work items are processed, popping off work items one at a time results in one lock per removal that might experience contention while the queue is being modified. In `QueuePerThreadPool`, when a thread pops off work for processing, **all** work items in the queue are removed, resulting in the removal of multiple work items with only a single lock that might experience contention. All work is processed before the thread returns to the work queue to find more work.

There exists a second work queue, called the deferred work queue, that is pushed to if the thread pool is initialized with a non-zero `queue_limit`. Work items are placed in the deferred work queue when the normal work queue has more than `queue_limit` items enqueued. The deferred work queue is only processed if the work queue is empty when the worker thread goes to look for more work. The work queue may still be pushed to if it was recently moved for processing (since the work queue now has as size of 0). This changes the order that work is processed, allowing for work to drain, reducing memory pressure, while still continuously processing work when there is work present.

If a thread discovers that it does not have work items in either queue but the thread pool still has outstanding work, it will search for more work in other threads. If work items are found, the thread will steal some of them, causing the stolen work items to experience less latency between enqueuing and processing. Note that when work items are stolen, the work queue is searched first, and if nothing is found, the deferred queue is searched next, instead of the next thread's work queue.

11.1.3 Usage

1. Create a thread pool:

   `QPTPool_t *pool = QPTPool_init(nthreads, args);`

   `nthreads` sets the number of threads in this thread pool.

   The `args` argument will be accessible by all threads that are run.

2. Setting Properties:

   `QPTPool_init` is intentionally kept simple and uses default values for some features. These values may be modified using `QPTPool_init_with_props` or `QPTPool_set_*` functions before `QPTPool_start` is called.

   By default, `QueuePerThreadPool` will push new work items in a round robin fashion. This can be changed to a custom function with `QPTPool_set_next`. This function is set at the context level instead of at `QPTPool_enqueue` in order to not require a branch to figure out whether or not the provided function pointer is valid.

   `QPTPool_set_queue_limit` causes work items to be pushed into the deferred work queue as described above.

   `QPTPool_set_steal` sets the numerator and denominator of the multiplier used when work items are being stolen from other threads. For the first queue where `queue.size * numerator / denominator` results in at least 1 work item, that many work items will be taken from the front of the queue.

3. Getting Properties:

   Properties may be extracted from the context using the `QPTPool_get_*` functions.

4. Start the thread pool:

   `QPTPool_start(pool);`

5. Add work:

   `QPTPool_enqueue(pool, id, function, work);`

   The function passed into `QPTPool_enqueue` must match the signature found in `QueuePerThreadPool.h`. The `work` argument will only be accessible to the thread processing this work.

   The thread that will receive the new work item is not `id`. Rather, `id` is treated as the source thread id and `threads[id]->next_queue` will be where the new work item is enqueued.

6. Wait for all work to be completed (threads are joined):

   `QPTPool_wait(pool);`

   This function exists to allow for the collection of statistics before the context is destroyed.

7. Destroy the pool context:

   `QPTPool_destroy(pool);`

11.2 `BottomUp`

`BottomUp` performs parallel downward directory traversal followed by parallel upward directory traversal only after every single child of a directory has been traversed. Users pass in functions to run when descending and ascending and have access to `lstat(2)` data collected during the tree traversal.

Note that `BottomUp` cannot be used for normal GUFI operations. GUFI expects each directory to be independently processed - directories will always start processing before their children, but might not complete processing until after their children have completed (the thread might have been interrupted). `BottomUp` does not enqueue children until the directory has been processed by the user provided descend function. Ascent only occurs after all children have been processed, allowing for the directory to see any changes that occurred under it when the user provided ascend runs.

`BottomUp` is used primarily for three executables: `gufi_rollup`, `gufi_treesummary_all`, and `parallel_rmr`. They all require child directories to be processed and modified before processing a directory.

11.3 Printing to `stdout`

Printing in parallel to a single target such as `stdout` without locking results in a jumble of text. Additionally, printing incurs a base cost in addition to the amount of data being printed. The OutputBuffers code, in conjunction with the `print_parallel` function, allows for data to be placed into individual buffers of known, customizable sizes before being written to `stdout` in a single call when a buffer is full.

The `print_parallel` function works as follows:

1. The size of the line to be printed, including the column delimiters and newline, is calculated.

2. If the line can fit into the provided buffer, write it into the buffer.

3. If the line cannot fit into the provided buffer, lock the print mutex, flush the buffered data, and then print the line. This maintains the ordering of the data outputted by the current thread.

11.4 Optimizations

In order for GUFI to be performant, many optimizations were used and implemented.

11.4.1 Reduced Branching

In order to reduce the number of failed branch predictions experienced by GUFI, branching was removed where possible. The main way this was done was by intentionally skipping `NULL` pointer checks that are repeated or always expected to be valid.

11.4.2 Allocations

Dynamic allocations are more costly to make than static allocations. To reduce the amount of dynamic allocations, C-string are usually declared as fixed size arrays instead of pointers.

During descent, each work item is initially allocated on the stack since there is no way to tell beforehand whether or not the path pointed to by a directory entry is a directory. If the path is a directory to be enqueued, the work item is copied into dynamically allocated memory first. If not, the work item allocated on the stack is used.

Additionally, allocations are not performed by the standard malloc. Instead, `jemalloc(3)` is used to override `malloc(3)`. See jemalloc's website for details.

11.4.3 Not Calling `lstat(2)` During Tree Walk

`struct dirent`s are returned when reading a directory with `readdir(3)`. glibc's implementation of `struct dirent` provides extra fields not required by POSIX.1. GUFI takes advantage of the nonstandard `d_type` field to not call `lstat(2)` when determining whether or not the entry is a directory. This also prevents memory allocations of work items that end up not being enqueued for processing.

11.4.4 Enqueuing Work Before Processing

In order to reduce the amount of time the thread pool spends waiting for work, work is enqueued before doing the main processing for a directory. When walking a tree, such as with `gufi_dir2index`, `gufi_dir2trace`, and `gufi_query`, subdirectories are enqueued before processing source filesystem information and running queries. `gufi_trace2index` reads each trace file using a single thread. However, each stanza is not processed as it is encountered. Instead, each stanza's file offset is enqueued in the thread pool, allowing for the directories of the index to be generated in parallel.

11.4.5 String Manipulations

Where possible, strings are not copied, and are instead referenced. Additionally, calls to `strlen(3)` are avoided in order to not walk memory one byte at a time. Some string lengths, such as those for input arguments are obtained with `strlen(3)`. Afterwards, string manipulations are done by using or modifying lengths to offset into or cut short strings.

11.4.6 Combining Strings with `memcpy(3)`

One method of combining C-strings is by concantenating them with `snprintf(3)` with format strings containing only `%s` format specifiers. Instead of parsing the format string, the `SNFORMAT_S` function was created to do `memcpy(3)`s on the arguments, skipping figuring out whether or not inputs are strings and how long they are by finding `NULL` terminators. Instead, lengths are obtained as by-products of previous string manipulations and the values are reused.

11.4.7 In-Situ Processing

Occassionally, GUFI might encounter extremely large directories. This results in many long lived dynamic allocations being created during descent, which can overwhelm memory. Users can set a subdirectory limit so that if too many subdirectories are encountered within a single directory, subdirectories past the user provided count will be processed recursively using a work item allocated on the thread's stack instead of being dynamically allocated and enqueued for processing. This reduces memory pressure by limiting the amount of work items that extremely large directories would otherwise spawn. The subdirectories being processed recursively may themselves enqueue dynamically allocated subdirectory work or recurse further down with subdirectory work allocated on the stack.

11.4.8 Smaller Enqueued Work Items

The main data structure that is enqueued is `struct work`. This struct was approximately 14KiB in size prior to 2227d00. After moving the parts of this structure that were not necssary for directory tree traversal to `struct entry_data`, `struct work` was reduced to slightly over 8KiB.

This was used to fix Issue 121.

11.4.9 Compression with zlib

When zlib is detected during CMake configuration, `struct work` can be compressed to further reduce the size of each work item that is sitting in memory waiting to be processed. The compressed buffer, originally allocated with `sizeof(struct work))` bytes, is then reallocated to the compressed size. The bulk of `struct work` is made up of text strings followed by `NULL` characters, both of which are highly compressible, meaning that compressed work items can be expected to be much smaller than uncompressed work items.

Note that `struct work` is its own compressed buffer. Whether or not the work item is compressed and the compressed length are now the first two fields of `struct work`. When a work item is compressed, a pointer pointing to it will have less space allocated to it than `sizeof(struct work))`.

This was used to fix Issue 121.

11.4.10 Database Templates

Every directory in an index contains at least one database file, called db.db, containing the `lstat(2)` data from the source filesystem. When creating indexes, a database file is created with the same schema as db.db and is left unfilled. When each directory in the index is processed, the database file created earlier is copied into the directory as a bytestream instead of having SQlite open new database files for each directory. This avoids the multitudes of checks done by SQLite when setting up databases and tables. The same is done for external xattr database files.

11.4.11 SQLite

As SQLite is a major component in GUFI, attempts were made to optimize its usage. Some optimizations were made at compile time. See the SQLite Compile-time Options page for details.

11.4.11.1 Locking In order to prevent multiple threads from corrupting data, SQLite implements locking. In GUFI, each database is only ever accessed by one thread:

- When indexing, only one thread writes to each directory's database.
- When querying, the per-thread results are written to per-thread databases. After the tree walk, the per-thread databases are merged serially into a final database.

Locking despite never modifying databases in parallel is not useful, and was removed by setting `-DSQLITE_THREADSAFE=0` in the compile flags.

11.4.11.2 VFS In addition to not locking SQLite in-memory operations, locking at the filesystem level was also disabled. Instead of opening SQLite database files with the default VFS, GUFI uses the `unix-none` VFS, which causes all file locking to become no-ops. See The SQLite OS Interface or "VFS" for details.

11.4.11.3 Memory Tracking Memory tracking was disabled with `-DSQLITE_DEFAULT_MEMSTATUS=0`.

11.4.11.4 Temporary Files Temporary files can be stored to disk or in memory. GUFI forces all temporary files to be stored in memory with `-DSQLITE_TEMP_STORE=3`.

11.4.12 Caching Queries (Not Merged)

When queries are performed on indexes, they are processed from scratch by each thread for each directory. An obvious optimization would be to reduce the amount of string parsing and query planning by compiling each query once (or a small number of times such as once per thread) at the beginning of a run and saving the compiled queries for repeated use during the index traversal.

An attempt at caching queries was made with Pull Request #95. Unfortunately, caching queries at best seemed to perform on par with the latest GUFI and at worst, slightly slower than the latest GUFI. This was true for both simple queries and complex queries with `JOIN`s.