Grand Unified File Index (GUFI)

11/2025

Table of Contents

[License Info 4](#_Toc213941897)

[Background 5](#_Toc213941898)

[General 5](#_Toc213941899)

[GUFI Requirements 7](#_Toc213941900)

[Gufi Design Tradeoffs 8](#_Toc213941901)

[How Does GUFI Work 10](#_Toc213941902)

[General 10](#_Toc213941903)

[Extraction 11](#_Toc213941904)

[Query 12](#_Toc213941905)

[GUFI makes all these databases/sources look like one Database 13](#_Toc213941906)

[GUFI Basic Security Model 14](#_Toc213941907)

[GUFI Index Composability/Decomposability 15](#_Toc213941908)

[Structure Reference 16](#_Toc213941909)

[GUFI Development Location 17](#_Toc213941910)

[Contribution location 17](#_Toc213941911)

[GUFI Building 18](#_Toc213941912)

[System Tools 19](#_Toc213941913)

[Libraries 19](#_Toc213941914)

[Packages provided by GUFI 20](#_Toc213941915)

[Build 21](#_Toc213941916)

[Environment Variables 21](#_Toc213941917)

[CMake Flags 22](#_Toc213941918)

[Install 24](#_Toc213941919)

[make 24](#_Toc213941920)

[RPMs 24](#_Toc213941921)

[Database Schema 25](#_Toc213941922)

[Tables/Schema 25](#_Toc213941923)

[pentries 30](#_Toc213941924)

[vrsummary 30](#_Toc213941925)

[vrentries 30](#_Toc213941926)

[vrpentries 30](#_Toc213941927)

[treesummary 31](#_Toc213941928)

[vsummarydir 31](#_Toc213941929)

[vsummaryuser 31](#_Toc213941930)

[vsummarygroup 31](#_Toc213941931)

[vtsummarydir 31](#_Toc213941932)

[vtsummaryuser 31](#_Toc213941933)

[vtsummarygroup 31](#_Toc213941934)

[Xattrs views 31](#_Toc213941935)

[External Database views 32](#_Toc213941936)

[Indexing 33](#_Toc213941937)

[Directly Indexing a Filesystem 33](#_Toc213941938)

[Indirectly Indexing a Filesystem 35](#_Toc213941939)

[Extended Attributes 38](#_Toc213941940)

[External Databases 39](#_Toc213941941)

[Location for External Databases 41](#_Toc213941942)

[Post-Indexing 42](#_Toc213941943)

[Rollup 42](#_Toc213941944)

[Tree summaries 49](#_Toc213941945)

[Querying 51](#_Toc213941946)

[gufi query 51](#_Toc213941947)

[gufi sqlite3 59](#_Toc213941948)

[SQLite Functions 61](#_Toc213941949)

[gufi stat bin 64](#_Toc213941950)

[Parallel and Multi-site/Cloud Object Indexing (Work in Progress) 65](#_Toc213941951)

[Utility Tools 65](#_Toc213941952)

[Source Tree Reconstruction 65](#_Toc213941953)

[Deployment 68](#_Toc213941954)

[User and Index Setup 68](#_Toc213941955)

[RPMs 68](#_Toc213941956)

[Wrapper Scripts 68](#_Toc213941957)

[GUFI\_find 69](#_Toc213941958)

[GUFI\_ls 69](#_Toc213941959)

[GUFI\_stats 69](#_Toc213941960)

[GUFI\_du 69](#_Toc213941961)

[Runtime Configuration 69](#_Toc213941962)

[Colocating the Server and Client 70](#_Toc213941963)

[SSH 70](#_Toc213941964)

[Authentication 70](#_Toc213941965)

[gufi\_ jail 70](#_Toc213941966)

[Extensibility 70](#_Toc213941967)

[GUFI query deeper look 71](#_Toc213941968)

[Basic Query 72](#_Toc213941969)

[Advanced SQL queries 76](#_Toc213941970)

[Compound Queries 79](#_Toc213941971)

[Nearest Neighbor style (highly optimized) query for vector similarity/AI 85](#_Toc213941972)

[Example of using full text search 92](#_Toc213941973)

[Useful examples of queries for storage administrators 93](#_Toc213941974)

[GUFI virtual tables and connectivity into Python, R, Ruby, analytics, and AI ecosystems 95](#_Toc213941975)

[GUFI used in R 99](#_Toc213941976)

[GUFI used in Ruby 99](#_Toc213941977)

[GUFI distributed processing 100](#_Toc213941978)

[GUFI parallel processing 101](#_Toc213941979)

[run\_vt - Enabling any command local or remote to appear as a local sqlite table 102](#_Toc213941980)

[GUFI experimental commands (experimental) 103](#_Toc213941981)

[Source Storage System Specific Commands 103](#_Toc213941982)

[Bfmi (not recently maintained) 103](#_Toc213941983)

[Source Storage System Specific Commands/Scripts 104](#_Toc213941984)

[Bfresultfuse 104](#_Toc213941985)

[bffuse 107](#_Toc213941986)

[Special and Incremental loading/updating 109](#_Toc213941987)

[Background 109](#_Toc213941988)

[Incremental Background 109](#_Toc213941989)

[Simple full ordered change log replay (not implemented but parts exist) 110](#_Toc213941990)

[Partial change logs, unordered change logs, or no change logs 110](#_Toc213941991)

[The GUFI general incremental method 111](#_Toc213941992)

[114](#_Toc213941993)

[114](#_Toc213941994)

[117](#_Toc213941995)

[118](#_Toc213941996)

[119](#_Toc213941997)

[gufi\_incremental\_update utility 119](#_Toc213941998)

[Concepts for providing suspect lists for storage systems 120](#_Toc213941999)

[ZFS 120](#_Toc213942000)

[EXT2 120](#_Toc213942001)

[EXT3 120](#_Toc213942002)

[EXT2/3 121](#_Toc213942003)

[EXT4 121](#_Toc213942004)

[XFS 121](#_Toc213942005)

[Lustre 121](#_Toc213942006)

[IBM Scale 121](#_Toc213942007)

[Producing full GUFI index from Scale ILM features (experimental) 121](#_Toc213942008)

[TSM backup 122](#_Toc213942009)

[HPSStoGUFI 124](#_Toc213942010)

[ToDo 124](#_Toc213942011)

## License Info

* # This file is part of GUFI, which is part of MarFS, which is released
* # under the BSD license.
* #
* #
* # Copyright (c) 2017, Los Alamos National Security (LANS), LLC
* # All rights reserved.
* #
* # Redistribution and use in source and binary forms, with or without modification,
* # are permitted provided that the following conditions are met:
* #
* # 1. Redistributions of source code must retain the above copyright notice, this
* # list of conditions and the following disclaimer.
* #
* # 2. Redistributions in binary form must reproduce the above copyright notice,
* # this list of conditions and the following disclaimer in the documentation and/or
* # other materials provided with the distribution.
* #
* # 3. Neither the name of the copyright holder nor the names of its contributors
* # may be used to endorse or promote products derived from this software without
* # specific prior written permission.
* #
* # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
* # ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
* # WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
* # IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
* # INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
* # BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
* # DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
* # LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
* # OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
* # ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
* #
* #
* # From Los Alamos National Security, LLC:
* # LA-CC-15-039
* #
* # Copyright (c) 2017, Los Alamos National Security, LLC All rights reserved.
* # Copyright 2017. Los Alamos National Security, LLC. This software was produced
* # under U.S. Government contract DE-AC52-06NA25396 for Los Alamos National
* # Laboratory (LANL), which is operated by Los Alamos National Security, LLC for
* # the U.S. Department of Energy. The U.S. Government has rights to use,
* # reproduce, and distribute this software. NEITHER THE GOVERNMENT NOR LOS
* # ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR
* # ASSUMES ANY LIABILITY FOR THE USE OF THIS SOFTWARE. If software is
* # modified to produce derivative works, such modified software should be
* # clearly marked, so as not to confuse it with the version available from
* # LANL.
* #
* # THIS SOFTWARE IS PROVIDED BY LOS ALAMOS NATIONAL SECURITY, LLC AND CONTRIBUTORS
* # "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
* # THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* # ARE DISCLAIMED. IN NO EVENT SHALL LOS ALAMOS NATIONAL SECURITY, LLC OR
* # CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
* # EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
* # OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* # INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* # CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
* # IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
* # OF SUCH DAMAGE.

# Background

## General

We live in a time of massive data generation. Never have we been tasked with storing, sorting, extracting metadata, indexing, protecting, and organizing so much critical information at such a vast scale. This has created major challenges for large-scale data centers, where data is growing more rapidly than the capabilities of the storage-systems that house that data. Our data is useless, if we can’t find what we are looking efficiently. Queries can have a devastating impact on the performance of ongoing computations/operations, which must themselves have unimpeded storage-system access to avoid wasting computational resources. Furthermore, security requires that results of queries must be constrained to include only data that a given user or artificial intelligence entity is allowed to see, including even the filenames. AI has the potential to be an incredibly force multiplier for science/defense, but one of its main tenants is aggregation of lots of information, which in a need-to-know world makes for issues. Leveraging something like AI while containing its aggregation scope is a difficult problem in a highly shared but need-to-know controlled environment. It is also useful to consider the differences in types of queries done by users versus by systems/data management professionals for a variety of tasks and try to accommodate all areas in a comprehensive indexing capability.

Storage systems have need-to-know access technologies such as POSIX permissions used in basically all file systems, access control lists (ACL) used in file systems and object systems like in the cloud, Multi-Level-Security (MLS) systems which use labeling and compartmented/policy driven access control. The problem is, storage-systems are typically not good at metadata oriented queries over the file/object name space or over metadata extracted from the files/objects. In order to find what you are looking for and to leverage things like AI, one needs to be able to easily find and leverage extracted metadata. What’s worse, each storage-system, which emphasizes a different strength for the storage aspect of the solution, often different tools for metadata handling. There are a few tools in industry that help with metadata indexing and query but essentially none specialize in honoring user-oriented security, most are tools meant for system administrators to use and user access is something of an afterthought.

The Grand Unified File Index (GUFI) is a solution to finding and utilizing extracted metadata that honors access control of the source information and as users change their need-to-know the access is adjusted the same as the source information. GUFI is an index that holds storage-system metadata (i.e. filenames, access and creation dates, file attributes, extended attributes, etc.) and extracted metadata from those files like text and other attributes, pulled from huge file/data storage-systems using full and incremental index update GUFI allows for rapid searches that don’t have to impact the file-systems themselves, supporting both users and system/data managers. GUFI is a very fast software solution that offers speed and security, while minimizing impact on supercomputing and source storage-system resources.

GUFI can be applied to a variety of file, object, and archive systems (tape archives, and file-system, object systems, and others), making it a ubiquitous solution for indexing that supports arbitrary queries and unifying information from all the places where a stored information might reside. Further, GUFI can dynamically query other information from things like databases, key value stores, etc. acting as the user. GUFI is a one-stop shop for access-controlled query and consumption of metadata from all your holdings. GUFI allows the users, artificial intelligent systems/agents, and system administrators to focus on doing the actual work for which these queries are just prerequisites.

## GUFI Requirements

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

* Unified index over multiple heterogeneous storage systems including shared storage like home and project space, scratch for the very hot tier, warm tier like LANL’s campaign store/MarFS, Object Systems, and Archives.
* Enable dynamic access to other non-storage-system information
* Obey and support full POSIX tree attributes, permissions, hierarchy representation as well as access control lists and compartmented multi-level access control security etc.
* Stores/indexes metadata from derived from source folders, buckets, and files
* Shared index for users and admins, users can only see metadata derived from files/folders/buckets/trees they could read from the source files/folders/trees
* Parallel search capabilities that are very fast (minutes for billions of files/dirs) including parallelism both threads and cross node
* Parallel metadata extraction with both full and incremental updating of the index, make the updating of this index as painless as possible
* The index should be highly composable/decomposable to allow multi-site and multi-storage-system indexing. This includes sub setting and aggregating parts of the index, all while honoring user access controls.
* Index can live in a separate space or within the source file/storage systems themselves
* 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
* Provide a way to work in a pure POSIX environment requiring only a POSIX interface to storage systems for full and incremental metadata extraction in addition to other non-POSIX techniques.
* Exploit special interfaces from source file/storages systems provided by some storage systems for mass metadata manipulation/extraction like GPFS ILM, Lustre activity logs, extraction from Robinhood SQL, Extraction from HPSS DB2, etc.
* 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. Ensure connectivity with the Apache analytics ecosystem and emerging standards like Model Context Protocol (MCP) for AI consumption. Additionally, enable additional indexing and search types as they emerge, like full text search on extracted text and vector similarity over text or other extracted metadata, all while obeying user access control.
* The intent is to provide a periodically 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/multi-machine 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
  + open 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.

## Gufi Design Tradeoffs

The following items were considerations in the choices 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, this is what is done in most commercial file index products.
  + 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 (and other methods like access control lists) 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 parallization 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. 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 allows 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 decompose/compose/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.

## How Does GUFI Work

### General

Most file/storage-systems employ the concept of a “tree” of directories containing sub-directories, containing sub-sub-directories, etc. Even with storage systems that are not inherently tree based, users still implement folder like organizations. GUFI partitions storage-system metadata and metadata extracted from files/folders into many small databases, kept in a directory-tree having the same structure and access-controls (POSIX/ACL/MLS/etc.) as that of the original storage-system or in the source storage system itself. Each folder/bucket in the source storage-system directory-tree has a matching folder in the GUFI directory-tree, with the same access permissions. Due to the ability to do breadth first traversal of trees, this allows for fast parallel searches across the GUFI databases (one per folder). Furthermore, because the directory access-controls are replicated from the original storage-system, regular users can be permitted to conduct their own searches, because they will simply not have access to the GUFI DBs that live in folders they can’t access. This mimics POSIX security/access control exactly and with access control lists mimics cloud object access control as well.

On March 22, 2018, GUFI was released to the public via GitHub as open-source software.

The use of GUFI has two different phases: (1) extraction and (2) query. In other words, critical information is extracted for users to query (see Figure 1).

Figure (1) Metadata is extracted from a file-system into GUFI. (2) Users and system administrators can then run queries against GUFI.

### Extraction

In this step, GUFI scans an entire (full or incremental) source large scale source storage-system and generates a corresponding tree of directories containing compact databases (DB) in a separate tree or within the source tree. This “GUFI tree” is a replica in organization and security of the directories in the original storage-system (POSIX, ACL,MLS) but instead of files, each directory in the GUFI-tree contains a single database. The databases in each GUFI directory hold the metadata for files and soft links in the corresponding directory of the original storage-system as well as summary information for that directory and optionally summary information for the entire tree below that directory. Thus, it is easy for system administrators and users to understand the structure of GUFI.

The structure and function of the three types of sql record holding tables in each GUFI database are described below.

* The entries table houses extracted metadata for files and links within the corresponding directory. Database rows capture file name, size, inode (primary key), These are vital facts (attributes and extended attributes) about files without the heft of the actual files.
* The directory summary table houses extracted information about everything within the corresponding directory. It is a summary of that directory, including information about minimum file size, maximum file size, the number of files, and more.
* The (optional) tree summary table houses extracted summary of all summary information for all directories that live under the current directory. Like the

*directory summary* database, the *tree summary* database provides a summary, but rather than summarizing a directory, it summarizes everything in and below the directory in which the *tree summary* table is found. This is an optional table intended for further optimization of user queries, when necessary.

Each database has a number of useful views to make querying easier.

* The *pentries* view provides parent inode as a query-able variable to the entries table. The reason this exists is that parent inodes are not stored because that would make updating the index for moves of directories/files difficult, so parent inode is calculated/looked up so that parent inodes are never stored with child records.
* The *vsummarydir* view provides access to the entire directory summary and not a partial directory summary (say by user or group).
* 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)
* The *vtsummarygroup* view provides access to the directory summary for each group (if this summary has been populated (not by default but easily populatable via a query)
* The *vtsummarydir* view provides access to the entire tree directory summary and not a partial directory summary (say by user or group).
* The *vtsummaryuser* view provides access to the tree directory summary for each user (if this summary has been populated (not by default but easily populatable via a query)
* 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)
* The vrpentries is a rollup safe view of the pentries view (rollups to be covered later)
* The vrsummary is a rollup save view of the vsummarydir view (rollups to be covered later)

Again, the GUFI-tree replicates the same standardized POSIX access-control settings of the original storage-system tree. Thus, a well-tested security is provided “for free” by the storage-system in which the GUFI-tree is created. This safety is a key feature of GUFI. If the user does not have access to a folder in the original storage-system, they will not have access to the corresponding GUFI folder and databases in the GUFI system. Further, if the user doesn’t have folder access and read access for a file, the user does not have access to extracted metadata from that file. Queries that span the GUFI-tree in parallel will simply not enter directories that are off limits and won’t see extracted metadata for files the user can’t read.

There is a standard POSIX extraction mechanism one can use to extract full and incremental updates to a GUFI from source file systems. We area also developing exploitations of faster metadata extraction mechanisms that are storage system specific like using GPFS ILM features for full and incremental metadata extraction. We are also working to provide a similar capability for Lustre and HPSS.

### Query

Once constructed/updated, the GUFI tree can be queried in a very parallel and efficient way. Queries can be performed in parallel across the databases in the tree, during a parallel (breadth first) “walk” of the GUFI tree. Users/admins can query billions of files in a very efficient way.

Using SQL to query the GUFI databases/tables, the user has great control over the details and operation of their query. The user can configure the number of parallel processes and threads, and apply SQL that is specific to each of the three GUFI record holding tables or useful views described above. A suite of complex, custom user queries that are specialized to their individual needs becomes possible, such as building temporary intermediate tables, and composing multiple subqueries together.

In addition to query of the tables/views described above, many useful functions are also provided which are additions to normal SQLite3 functions (like date formatting etc.). (functions to be covered later)

GUFI queries the databases in parallel, via breath first walk threads and even via parallel processes pointed at different parts of the overall GUFI tree. These results can be accumulated or placed in an output database and further queried, depending on the requirements of the user. The ability to generate new monolithic databases supports subsequent simpler queries SQL for enabling grouping/ordering etc.

Sophisticated yet efficient custom queries can be easily constructed to exploit the output result database mode of operation, enabling SQL join operations between results tables from previous queries, though this is just an option, not a requirement. The result output database concept is a very powerful one.

Additionally, it is possible to provide a query and start a fuse daemon over all or some part of a GUFI tree and have the query run dynamically as metadata commands are run inside that fuse mounted file system and also a query can produce an output results database and a fuse daemon can be started that uses that output results database so that standard metadata commands can be run against the results. The commands supported on the fuse and fuse over results options have to be metadata only commands like ls, find, stat, xattr -l, etc.

### GUFI makes all these databases/sources look like one Database

Beyond providing an easy way to connect to, need-to-know enforcing, way to access metadata about vast heterogeneous data holdings, because of the way GUFI is deployed as large trees of smaller databases/indices, GUFI must take on the all-important job of making all those smaller databases/sources look like one database or even one database table. This approach is only way it is reasonable to use/connect to/reason about this index and it’s the only way we can leverage all the free work the Apache analytics ecosystem and to fit seamlessly into future facing interfaces like the Model Context Protocol now becoming popular in AI for agents.

## GUFI Basic Security Model

All the GUFI tools, indexer, query, etc. run as a user.

If a user wanted to run the indexer program(s), it will work, it will create a tree of only what you point it at and put the index where you tell it as long as you have access to both the top of the tree/subtree you point at and where you are putting the index tree.  That index will contain only directories (directory info (file names, stat information, etc.)) for the directories you could get into in the source tree, so you have be able to traverse into a directory at stat the directory content (a combo of read/execute bits on the directory and the directories above (for traverse into).

This works for a user for both what the user can directly see (user owns it or everyone can see it) but it also works for the group the user is in and for all the groups the user can see.  So users can have their own GUFI indices, and the query will run over that index tree.

For a group or set of groups, if that group(s) wanted to have a GUFI index, a user that is in that group(s) could run the indexing tool and then everyone could query that is in the group(s), if you put the index tree in a place they can see.  Root is not required unless you are making an index across a file system where root is the only one that has access to it all.

Remember GUFI index is just a file system, so if you put that index on a labeled file system and use it from an MLS setup, it will just obey that security model. Same goes for using ACLs, if the index you put your index on can use access control lists than you can protect the index information that way.  If you were going to index cloud object buckets, they use ACLs to give access to the bucket, so it wouldn’t be hard to whip up a way to index a bucket, put that into a directory in a GUFI tree on a file system that supports ACLs and you would just mimic the ACL of the cloud bucket and graft that into an index tree however you want.

Of course, it’s up to use to secure destination index tree however you want.  For the base GUFI info (directories, files, stat information, etc.) it’s just information that is protected by directory permissions – the file permissions don’t matter at all.  If you use extended attributes or external databases that contain metadata pulled from the contents of the file (like text extraction or something), that information in the source tree is protected by both traversal and the file permissions (you must have read on the file).  GUFI handles this situation for xattrs and external databases by doing permissions permutation database files.  It makes a database file for each permission permutation ( u+r (may be multiples g+r (may be multiples) p+r). It sets the permission on the db file equal to that permission permutation.  Then when you run a query, it tries to attach all those permission permutation db files (Sqlite lets you attach other db’s to an open instance). It will then concatenate the tables  from all the files into a single table so you think it’s just one table but it’s really many possibly – it doesn’t move any data until you read it).  Because the permissions permutation database files are set to a single permissions permutation, you may be able to see some of the information and may not see other parts, only if you could read the source file.  The attach succeeds if you can access it and if you don’t it fails.  So, we are protecting extracted information from the source file the same way the source file is protected (from a read point of view).

All of this will work for a user or root, but if you are a user it’s all caveated by – it only indexes and queries what you could or can see.  Root can see it all and a user can see what that user owns or any groups that user is in with appropriate traversal and file permissions etc.

You might wonder about merging info from SMB/windows shares. Since NFS4/PNFS support common security model with inheritance, it’s possible to index windows shares with GUFI a LINUX or likely with Cygwin/wsl/msys2 support, although this has not been tested. Since LINUX can mount windows shares, indexing from LINUX and providing web or the many other access methods GUFI has, works fine. GUFI does build and run on MacOS natively and can mount win shares and index them.

Since the GUFI query runs as the user, using the features of GUFI that allow the query to access data outside of GUFI index tables, such as intop(), strop(), blobop(), and virtual gufi tables (all these to be discussed later), all those things can run LINUX/MacOS/Cygwin commands from within the GUFI query. This enables the GUFI query to pull in information from non-GUFI sources as well as even modify files. In all cases, the user can do only what the user can do. For this reason, allowing users write access to GUFI indexes should be considered before doing so.

Since the GUFI query can get information via commands embedded in queries and virtual tables and since external databases could live in other trees than the index tree, there should be no good reason to all users to have write access to the GUFI index tree unless of course this is exactly what you want to enable.

## GUFI Index Composability/Decomposability

Because the index is tree in a file system, it just behaves like that.  Also, if you want to graft various indexes together, say you want to index your workstation and you want to index the shared files you have access too on  an NFS server or something.  you could make an index for your workstation and you could make an index for the NFS data you can see and you just graft those together however you want like make a /home/me/search dir and put your workstation index in /home/me/search/workstationindex and put the your NFS index in /home/me/search/nfsindex.  You can tar these things up as they are just trees and files and take them with you, give them to people, whatever you want.  Ultimately it will just obey the file system permissions of the index (which in most cases you want to match the source but there are times where that might not be true).

This is all enabled by the fact that we do not (other than tree summaries and rollups which can be trivially dropped and recreated) put anything into a single directory’s db tables that mentions the tree shape (parents or children).  Normally if you do a readdir of a directory you get all its files and its child directories.  Not in GUFI, the child directories (subdirs) in that directory are derived from readdir of the directory the index is in and NOT from looking up things in the db.db tables.  This makes gufi indexes composable and decomposable.  So in the GUFI fuse if you do an “ls” command, it translated into a readir and the readdir in the fuse first does a readdir in that index directory and gets all the children and hands you those as subdir names and then it does a select name from the entries table (which contains all the files and other artifacts like simlinks and fifos and ..).

# Structure Reference

As has been described above, the structure of the GUFI tree looks like the diagram in Figure 2 below.

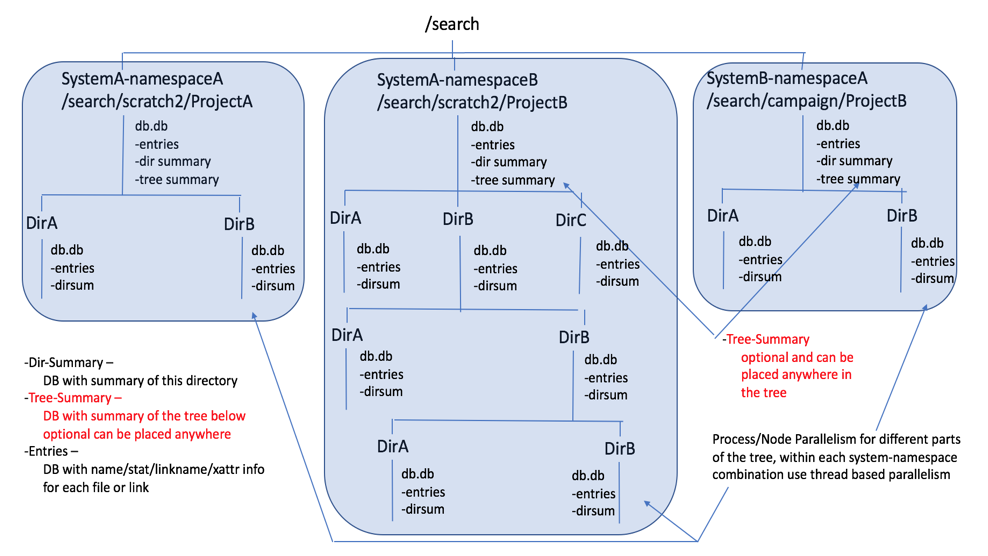


Figure 2 GUFI Overall Structure

The following diagram depicts how a source file/storage system metadata is extracted into a GUFI tree where file/link metadata is placed into a per directory GUFI database in Figure 3.

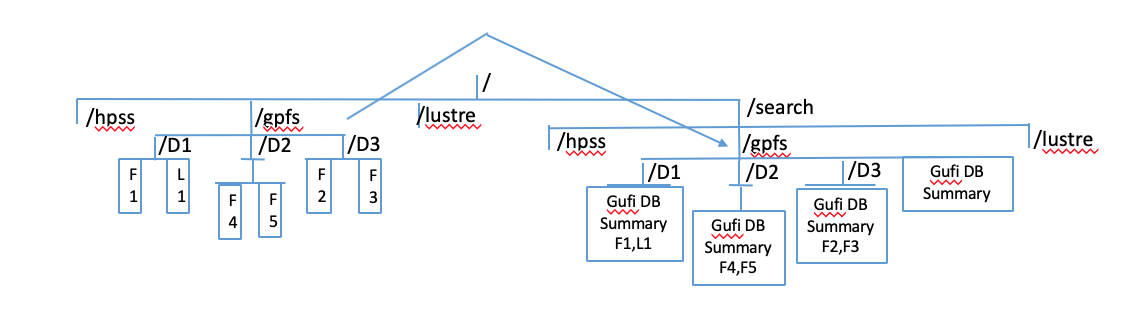


Figure 3 Mapping of source file system into GUFI tree

# GUFI Development Location

## Contribution location

Initial development was done at LANL via an internal git-hosting service. As of rev 0.1.0, we are moving GUFI to github. We invite anyone to feature-requests, etc, through github:

https://github.com/mar-file-system/GUFI

We will attempt to respond to requests, but we can't make promises about the level of resources that will be dedicated to GUFI maintenance.

All bug-reports, issues, requests, etc, should go through github.

For other conversation about GUFI (and MarFS), there is:

marfs-on-github@lanl.gov

See other components of MarFS at:

https://github.com/mar-file-system.

# GUFI Building

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.

## System Tools

|  |  |  |
| --- | --- | --- |
| Package/Executable | Required? | Comments |
| autotools | Yes | Some bundled packages use autotools |
| C Compiler | Yes | C11 support (actually only need C99  but AI packages require C11, so all of  GUFI now requires C11 whether or not  AI packages are built) |
| C++ Compiler | No | C++11 support - g++ 4.9.3, clang++-  3.9, or newer |
| CMake | Yes | Version 3.19 or higher |
| column | Yes |  |
| diff | Yes |  |
| find | Yes |  |
| getfattr |  | attr package  xattr -w on OSX |
| git | No | Only needed if configuring with  DEP AI=On |
| grep | Yes |  |
| ImageMagick | No | Only needed if building LATEX Documentation |
| LATEX | No | Only needed if building LATEX Documentation |
| Make | Yes |  |
| Python | Yes | Python 3+ |
| pip | No | Used for installing and client dependencies |
| patch | Yes |  |
| pkg-config | Yes |  |
| realpath | Yes |  |
| rpmbuild | No | Only needed if building RPMs |
| sed | Yes |  |
| setfattr | Yes | attr package  xattr -w on OSX |
| truncate | Yes | The -s option must be available |
| uname | No | Only needed if building RPMs |

## Libraries

|  |  |  |
| --- | --- | --- |
| Name | Required? | Comments |
| xattr | Yes |  |
| pcre | Yes | Version 2 compiled with -fPIC |
| fuse | No | Called osxfuse on OSX. Not updating to macfuse  for now because pkg-config can’t seem to find it. |
| db2 | No |  |
| gpfs | No |  |
| zlib | No |  |

## 

## Packages provided by GUFI

The source code for GUFI can be found at [https://github.com/mar-file-](https://github.com/mar-file-system/GUFI)system/ [GUFI](https://github.com/mar-file-system/GUFI).

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

|  |  |
| --- | --- |
| Name | Comments |
| GoogleTest | <https://github.com/google/googletest> |
| jemalloc | <https://github.com/jemalloc/jemalloc> |
| sqlite3 | <https://www.sqlite.org/index.html>  patched sqlite-autoconf-3430100.tar.gz  configured with the FTS5 extension enabled (https://www.sqlite.org/fts5.html) |
| sqlite3-pcre | [https://github.com/mar-file-system/sqlite3-pcre/](https://github.com/mar-file-system/sqlite3-pcre/tree/pcre2)  [tree/pcre2](https://github.com/mar-file-system/sqlite3-pcre/tree/pcre2) |
| sqlite3 lembed | https://github.com/asg017/sqlite-lembed |
| sqlite3 vec0 | https://github.com/asg017/sqlite-vec |

# Build

GUFI uses CMake to generate Makefiles. CMake 3.19 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.

The list below shows the environments where GUFI is known to have built,

run, and passed tests at some point. The environments labeled with a plus (+)

are currently being used on Github Actions. The environments labeled with

minus (-) are no longer used on GitHub Actions and might no longer work.

.

* Linux
* Alpine Linux Edge
* CentOS 7
* CentOS 8
* OpenSUSE 12.3
* Ubuntu 16.04 (Xenial)
* Ubuntu 18.04 (Bionic)
* Ubuntu 20.04 (Focal)
* Ubuntu 22.04 (Jammy)
* Ubuntu 24.04 (Noble)
* Rocky Linux 8
* Rocky Linux 9
* macOS
* 10.13 (High Sierra)
* 11 (Big Sur)
* 12 (Monterey)
* 13 (Ventura)
* 14 (Sonoma)
* 15 (Sequoia)
* Windows
* cygwin
  + GCC (not MinGW)
  + jemalloc turned off

## Environment Variables

|  |  |
| --- | --- |
| Setting | Description |
| CXX=false | Disable building of C++ code |

## CMake Flags

#### 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 |

#### 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 every time ${CMAKE\_BINARY\_DIR} is deleted. |
| DEP PATCH SQLITE3 OPEN=<On|Off> | Whether or not to patch SQLite3 open (default: Off) |
| DEP USE JEMALLOC=<On|Off> | Whether or not to build and link with jemalloc (default: On) |

#### Debug

CMAKE BUILD TYPE must be set to Debug for these to have effect.

|  |  |
| --- | --- |
| -D<VAR>=<VALUE> | Description |
| GPROF=<On|Off> | Compile with the -pg flag (default: Off) |
| GCOV=<On|Off> | Compile with the --coverage flag (default: Off) |

#### System Paths

Some files are installed into system paths that are not $*{*CMAKE INSTALL PREFIX*}*/bin

or $*{*CMAKE INSTALL PREFIX*}*/lib.

|  |  |
| --- | --- |
| -D<VAR>=<VALUE> | Description |
| CONFIG SERVER=<FILEPATH>  CONFIG\_CLIENT=<FILEPATH> | Path of configuration file used by scripts.  Two different paths are available in case the server and client  deployments place their configuration files at different  locations.  Both default to /etc/GUFI/config.  When CLIENT=Off, CONFIG\_SERVER is used as the location of the configuration file.  Note that many paths will conflict with the paths of other  packages if installing GUFI via package. If installing GUFI  with make install, point to a convenient location. |
| BASH COMPLETION=<On|Off> | Whether or not to install bash completion script to  /etc/bash completion.d. (default: On).  Useful when running make install without root. |

#### Features

|  |  |
| --- | --- |
| -D<VAR>=<VALUE> | Description |
| CLIENT=<On|Off> | Whether or not to install the gufi client when make install is called (default: Off) |
| QPTPOOL\_SWAP=<On|Off> | Whether or not to build QueuePerThreadPool with work swapping  (default: On) |

#### Testing

|  |  |
| --- | --- |
| -D<VAR>=<VALUE> | Description |
| ENABLE SUDO TESTS=<On|Off> | Run tests that require sudo (default: Off)  Can also force enabling of tests in case CMake Policy  CMP0109 comes up |
| TEST WORKING DIRECTORY=<PATH> | Directory to run tests in. Defaults to  ${CMAKE BINARY DIR}/test |

#### Docs

|  |  |
| --- | --- |
| -D<VAR>=<VALUE> | Description |
| LATEX BUILD=<On|Off> | Whether or not to build PDF documentation when LATEXis found |

# Install

## make

To install GUFI from the source build, run [sudo] make install from the build directory.

## RPMs

make package will be available for generating RPMs if rpmbuild was found at configuration time. By default, only the server RPM will be generated. In order to generate the client RPM (see Section [12](#_bookmark105)), the CMake flag -DCLIENT should be set to On.

# 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.

## Tables/Schema

The structure and function of the three types of sql record holding tables in each GUFI database are described below.

* The entries table houses extracted metadata for files and links within the corresponding directory.
  + "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,TEXT, crtime INT64, ossint1 INT64, ossint2 INT64, ossint3 INT64, ossint4 INT64, osstext1 TEXT, osstext2 TEXT);";
    - name TEXT character name of file or link
    - type TEXT character d for file or link
    - inode INT64 inode number from source system integer
    - mode INT64 unix mode bits integer
    - nlink INT64 unix number of links integer
    - uid INT64 unix uid integer
    - gid INT64 unix gid integer
    - size INT64 unix logical file size in bytes integer
    - blksize INT64 unix blksize for file integer
    - blocks INT64 unix number of blocks integer
    - atime INT64 unix access time epoch integer
    - mtime INT64 unix modificaton time epoch integer
    - ctime INT64 unix change time epoch integer
    - linkname TEXT unix string for link name character
    - xattr\_names TEXT concatenation of all extended attributes character
    - crtime INT64 create time epoch integer (some file systems provide this)
    - ossint1 INT64 storage system specific integer
    - ossint2 INT64 storage system specific integer
    - ossint3 INT64 storage system specific integer
    - ossint4 INT64 storage system specific integer
    - osstext1 TEXT storage system specific string character
    - osstext2 TEXT storage system specific string character
* The directory summary table houses extracted information about everything within the corresponding directory. It is a summary of that directory, including information about minimum file size, maximum file size, the number of files, and more.
  + "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, xattr\_names 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);";
    - * name TEXT character name of directory
      * type TEXT character d for directory
      * inode INT64 inode number from source system integer
      * mode INT64 unix mode bits integer
      * nlink INT64 unix number of links integer
      * uid INT64 unix uid integer
      * gid INT64 unix gid integer
      * size INT64 unix logical file size in bytes integer
      * blksize INT64 unix blksize for file integer
      * blocks INT64 unix number of blocks integer
      * atime INT64 unix access time epoch integer
      * mtime INT64 unix modificaton time epoch integer
      * ctime INT64 unix change time epoch integer
      * linkname TEXT unix string for link name character
      * xattr\_names TEXT concatenation of all extended attributes character
      * totfiles INT64 number files
      * totlinks INT64 summed links
      * minuid INT64 min uid
      * maxuid INT64 max uid
      * mingid INT64 min gid
      * maxgid INT64 max gid
      * minsize INT64 min size
      * maxsize INT64 max size
      * totltk INT64 number <= 1024
      * totmtk INT64 number > 1024
      * totltm INT64 number <= 1048576
      * totmtm INT64 number > 1048576
      * totmtg INT64 number <= 1073741824
      * totmtt INT64 number > 1073741824
      * totsize INT64 summed size
      * minctime INT64 min ctime
      * maxctime INT64 max ctime
      * minmtime INT64 min mtime
      * maxmtime INT64 max mtime
      * minatime INT64 min atime
      * maxatime INT64 max atime
      * minblocks INT64 min blocks
      * maxblocks INT64 max blocks
      * totxattr INT64 number of files/links with xattrs present
      * depth INT64 dept (number of slashes in path)
      * mincrtime INT64 min create time
      * maxcrtime INT64 max create time
      * minossint1 INT64 min ossint1
      * maxossint1 INT64 max ossint1
      * totossint1 INT64 summed ossint1
      * minossint2 INT64 min ossint2
      * maxossint2 INT64 max ossint2
      * totossint2 INT64 summed ossint2
      * minossint3 INT64 min ossint3
      * maxossint3 INT64 max ossint3
      * totossint3 INT64 summed ossint3
      * minossint4 INT64 min ossint4
      * maxossint4 INT64 max ossint4
      * totossint4 INT64 summed ossint4
      * rectype INT64 0 for total for entire dir 1 for totals by user 2 for totals by group
      * pinode INT64 parent directory inode
* The (optional) tree summary table houses extracted summary of all summary information for all directories that live under the current directory. Like the

*directory summary* database, the *tree summary* database provides a summary, but rather than summarizing a directory, it summarizes everything in and below the directory in which the *tree summary* table is found. This is an optional table intended for further optimization of user queries, when necessary.

* + "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);";
    - * totsubdirs INT64 number directories under this directory
      * maxsubdirfiles INT64 maximum number files in any subdir
      * maxsubdirlinks INT64 maximum number of links in any subdir
      * maxsubdirsize INT64 maximum summed size in any subdir
      * totfiles INT64 number files
      * totlinks INT64 summed links
      * minuid INT64 min uid
      * maxuid INT64 max uid
      * mingid INT64 min gid
      * maxgid INT64 max gid
      * minsize INT64 min size
      * maxsize INT64 max size
      * totltk INT64 number <= 1024
      * totmtk INT64 number > 1024
      * totltm INT64 number <= 1048576
      * totmtm INT64 number > 1048576
      * totmtg INT64 number <= 1073741824
      * totmtt INT64 number > 1073741824
      * totsize INT64 summed size
      * minctime INT64 min ctime
      * maxctime INT64 max ctime
      * minmtime INT64 min mtime
      * maxmtime INT64 max mtime
      * minatime INT64 min atime
      * maxatime INT64 max atime
      * minblocks INT64 min blocks
      * maxblocks INT64 max blocks
      * totxattr INT64 number of files/links with xattrs present
      * depth INT64 dept (number of slashes in path)
      * mincrtime INT64 min create time
      * maxcrtime INT64 max create time
      * minossint1 INT64 min ossint1
      * maxossint1 INT64 max ossint1
      * totossint1 INT64 summed ossint1
      * minossint2 INT64 min ossint2
      * maxossint2 INT64 max ossint2
      * totossint2 INT64 summed ossint2
      * minossint3 INT64 min ossint3
      * maxossint3 INT64 max ossint3
      * totossint3 INT64 summed ossint3
      * minossint4 INT64 min ossint4
      * maxossint4 INT64 max ossint4
      * totossint4 INT64 summed ossint4
      * rectype INT64 0 for total for entire tree 1 for totals by user 2 for totals by group
      * uid INT64 unix uid
      * gid INT64 unix gid

**Directory** Entries relationship

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. Note that the inode and pinode colums are strings and not integers as one might expect. As a general rule, do not query this table directly.

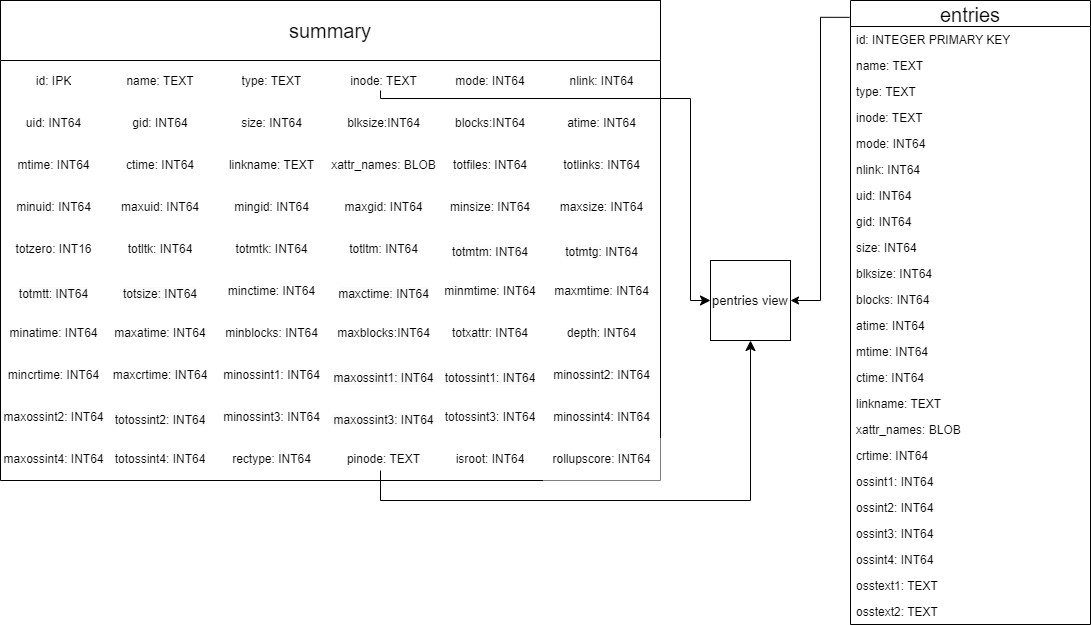


Figure 2: Database schema

### pentries

The original pentries view was defined as:

SELECT entries.\*, summary.inode AS pinode, summary.pinode AS ppinode 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.

Users should prefer query the pentries view over the entries table in order to obtain complete sets of data to query on, and should use pentries whenever paths are not required.

pentries rollup

(rollups will be covered later)

In order to simplify rolling up indexes (see Section [9.1](#_bookmark56)), 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.

### 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.

(rollups will be covered later)

### vrentries

The vrentries view does not exist because it would be a misuse of the schema: the entries table itself is never rolled up, so querying a vr version of it would result in incorrect output.

(rollups will be covered later)

### 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.

(rollups will be covered later)

### 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](#_bookmark67) for details.

### vsummarydir

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

### 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)).

### 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)).

### vtsummarydir

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

### 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)).

### 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)).

### Xattrs views

With the addition of extended attributes support, several more tables and views were added into db.db. From the user’s perspective, the xattrs view has been created and joined with entries, pentries, summary, vrpentries, and

vrsummary to create the convenience views xentries, xpentries, xsummary, vrxpentries, and vrxsummary. These views are always available whether or not extended attribute processing has been enabled. See Sections [8.3.2](#_bookmark49) and

[10.1.4](#_bookmark79) for details. Using vrxpentries and vrxsummary should be preferred over all previously mentioned views when paths are required.

### External Database views

(external databases will be covered in more detail below)

With the addition of external (user) database support, several more views were added into db.db. External database are variants of summary, pentries, xsummary, xpentries, vrsummary, vrxsummary, vrpentries, and vrxpentries

were added: esummary, epentries, esummary, epentries evrsummary, evrxsummary, evrpentries, evrxpentries. These views should be joined with the external databases that were attached. Using these views should be preferred over all previously mentioned views when paths are required.

# Indexing

The first step to using GUFI is indexing a source storage system.

An index created by GUFI retains the shape of the source storage system: 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](#_bookmark20) and [8.3.2](#_bookmark49). Additional database files may be created if extended attributes are extracted (see Section [8.3](#_bookmark47)).

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

## Directly Indexing a Filesystem

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 |
| -y <min level) | minimum level to traverse down |
| -z <max level) | maximum level to traverse down |
| -k *<*filename*>* | file containing directory names to skip |
| -M *<*bytes*>* | target memory footprint |
| -s <path> | file name prefix for swap files |
| -C *<*count*>* | Number of subdirectories allowed to be  enqueued for parallel processing. Any remainders will be processed in-situ |
| -e | compress work items |
| -q *<*basename*>* | Basename of file to keep track of during indexing |
| -D <start> <stop> | start and stop path names for partial indexing |

Table 1: gufi dir2index Flags and Arguments

#### Usage

gufi dir2index [flags] input dir... output dir

The index of input diri will be placed in output dir/$(basename input diri).

#### Plugins

Plugins may be created to insert extra data (such as filesystem specific information) as a source tree is indexed.

A plugin provides an instance of the plugin\_operations structure found in include/plugin.h. The plugin source is compiled and linked with any necessary external libraries into a shared library that is independent of GUFI (other than the header). Below is an example of compiling and using the lustre plugin that comes with GUFI:

# set based on the lustre source location on your machine:

$ export LUSTRE\_INCLUDE\_DIR=…

$ export LUSTRE\_LIBRARY\_DIR=…

$ cd <GUFI root>/contrib

$ gcc -g -O0 -c -fPIC -I../build/deps/sqlite3/include -I../include -I$LUSTRE\_INCLUDE\_DIR -I$LUSTRE\_INCLUDE\_DIR/uapi lustre\_plugin.c

$ gcc -shared -o liblustre\_plugin.so lustre\_plugin.o -L$LUSTRE\_LIBRARY\_DIR -llustreapi

Plugins can then be used in gufi\_dir2index using the -U flag:

$ LD\_LIBRARY\_PATH=$LUSTRE\_LIBRARY\_DIR ./src/gufi\_dir2index -U ../contrib/liblustre\_plugin.so -n1 /mnt/lustre /tmp/gufi\_index/

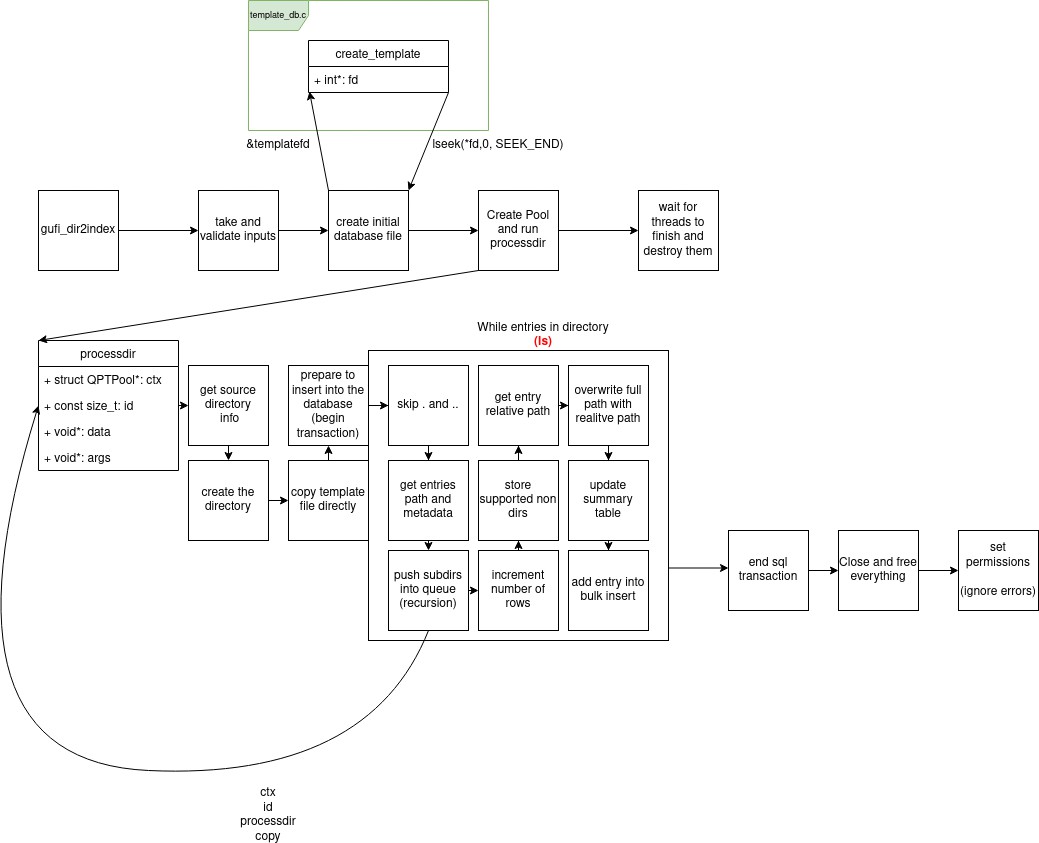


Figure 3: gufi dir2index workflow

## Indirectly Indexing a Filesystem

#### 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.

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 |
| -y <min level) | minimum level to traverse down |
| -z <max level> | maximum level to traverse down |
| -d *<*delim*>* | delimiter (one char) [use ’x’ for 0x1E] |
| -k *<*filename*>* | file containing directory names to skip |
| -M *<*bytes*>* | target memory footprint |
| -s <path> | file name prefix for swap files |
| -C *<*count*>* | Number of subdirectories allowed to be  enqueued for parallel processing. Any remainders will be processed in-situ |
| -e | compress work items |
| -q *<*basename*>* | Basename of file to keep track of during indexing |
| -D <start> <stop> | start and stop path names for partial indexing |

Table 2: gufi dir2trace Flags and Arguments

#### 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.

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.

#### Usage

gufi trace2index [flags] trace file... index root

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 |
| -s <path> | file name prefix for swap files |

Table 3: gufi trace2index Flags and Arguments

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

## 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.

#### 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.

#### 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 di- rectory 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 cur- rent 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 file- names 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.

#### 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.

## External Databases

In addition to extended attributes, users may place SQLite3 database files into the source file system to be indexed and attached into GUFI for querying. This effectively allows for users to associate and query arbitrary data with filesystem metadata.

The extended attribute code uses the more generic external database infrastructure to track the extended attributes that were not rolled in. If users wish to use their own external databases, they will have the same issues with rollups if they want to roll their information up for performance reasons if the external databases are extracted information from files and that information needs to be protected the same as the files the information was derived from. The same methods used as in the extended attribute roll in oriented permission permutation approach would apply. Using the GUFI external database feature helps with this ensuring of permissions and enabling rollups.

#### Usage

All external database files for a set of data should have the same basename. The files should be listed in a simple text file called “external.gufi” that is in the directory where a subset of the data will be joined with GUFI data. If the path is relative, it will be concatentated with the parent directory’s path to produce an absolute path.

-q may be passed in during indexing in order to verify that an entry listed in “external.gufi” is indeed a database file.

One more more additional database files, referred to as the “schema tem- plate”, or “template”, containing a copy of the same schema as each external database table should be maintained somewhere for usage when querying. The template file should generally have empty tables, but nothing prevents non- empty tables from being provided.

Note that “external.gufi” will also be included as an entry in the index (this may change in the future).

## Location for External Databases

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.

#### Permissions for the index location(s)

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.

# Post-Indexing

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

## 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.

#### 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.

How will we reduce the number of directories/databases given the security model leverages using the source data tree shape and permissions? The answer is permissions permutations based combining of many directories/databases into fewer directories/databases. Folder trees have a nice attribute that the permissions are very hierarchical and most subdirectories access is exactly the same as the parent directory they are in. The concept is to start walking the index tree from the bottom and traverse upwards, combining information upwards until the permissions change that don’t allow further upward traversal. We called the is process “rollup”. In testing across all our file systems at LANL including highly shared, home, project, scratch, archive with both young and extremely old storage systems and found that this rollup process reduces the number of directories/databases that must be traversed/opened for queries completely eliminated all but one of the dominate time using functions and improved the already good performance by orders of magnitude, making user queries mostly sub-second and extreme system admin queries to be very few minutes across trillions of file objects and billions of directories.

#### Rules

#### 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 con- flicting permission with, they are considered rolled up.

#### Whether or not a directory SHOULD roll up

In addition to the permission-based rules that determine whether or not a di- rectory 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 accu- mulate 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.

#### 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](#_bookmark58). 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.

1. 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.

#### Extended Attributes

Additional steps are needed to rollup xattrs. This is because in extended attributes, the attribute name is protected with the directory permission but the value of the attribute is protected with read access to the file the extended attribute is associated with. This means you may need more than one database in a directory with different permissions on each database matching the permission permutations needed to cover read access to the files within the directory to hold xattrs. This means you aren’t just rolling up the main GUFI database per directory, but you also need to roll up the ancillary xattr databases to hold the attribute values that are protected differently. We call the process that concatenates all the extended attribute databases into one database/table “roll in”. Because the GUFI query runs as the user, the user can only read extended attributes they are allowed to, so this roll in process can only roll in the attributes that the user has read access to the file the attribute is associated with.

* 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.

gufi rollup **executable**

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

gufi rollup index root

./gufi\_rollup

usage: ./gufi\_rollup [options] GUFI\_index ...

options:

-h help

-H show assigned input values (debugging)

-v version

-n <threads> number of threads

-L <count> Highest number of files/links in a directory allowed to be rolled up

-X Dry run

GUFI\_index GUFI index to roll up

Figure [4](#_bookmark65) 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](#_bookmark66).

#### Undoing a rollup

To remove rollup data from an index, run the gufi unrollup executable:

gufi unrollup index root

gufi\_unrollup

usage: ./gufi\_unrollup [options] GUFI\_index ...

options:

-h help

-H show assigned input values (debugging)

-v version

-n <threads> number of threads

GUFI\_index GUFI index to unroll up

27

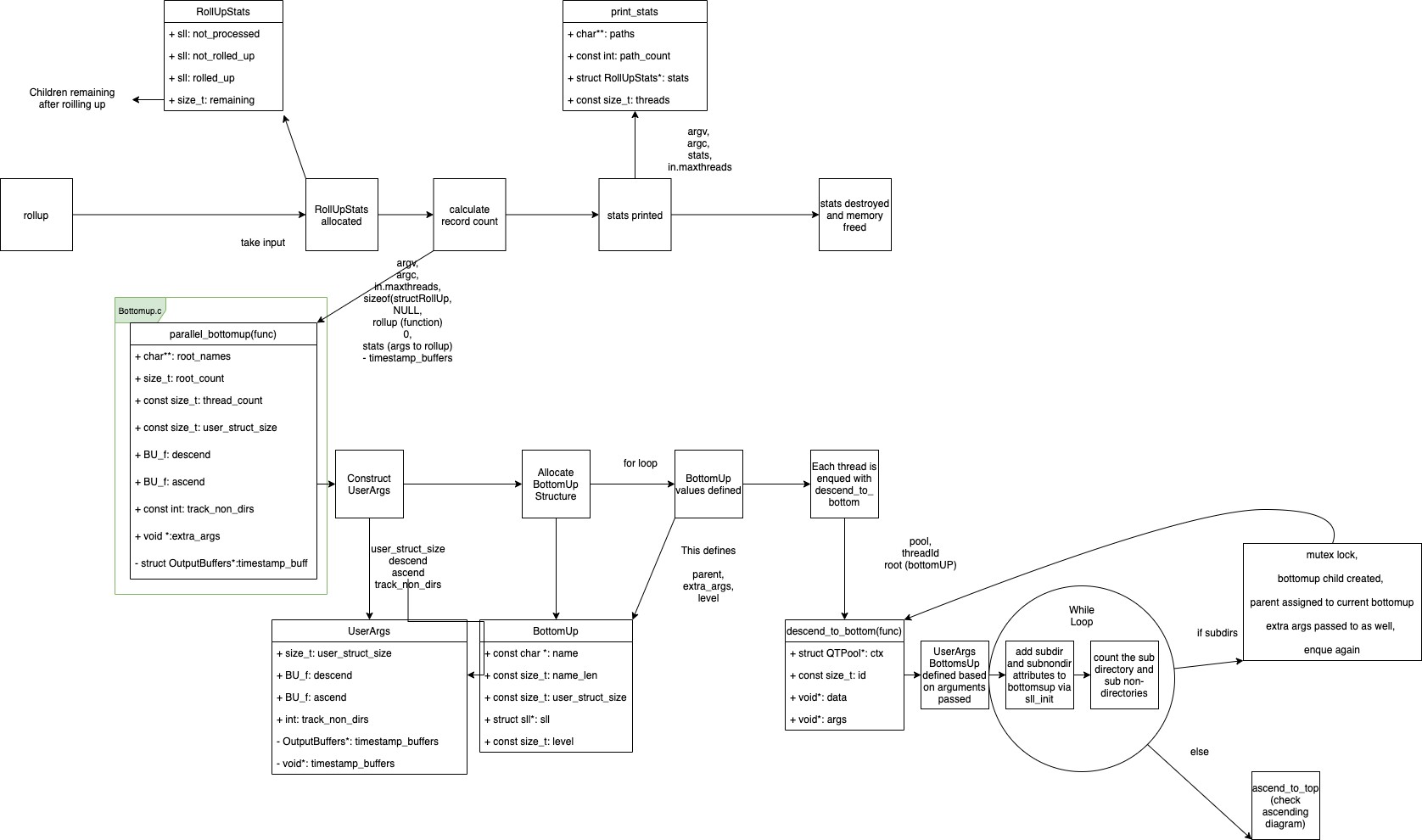


Figure 4: gufi rollup Workflow

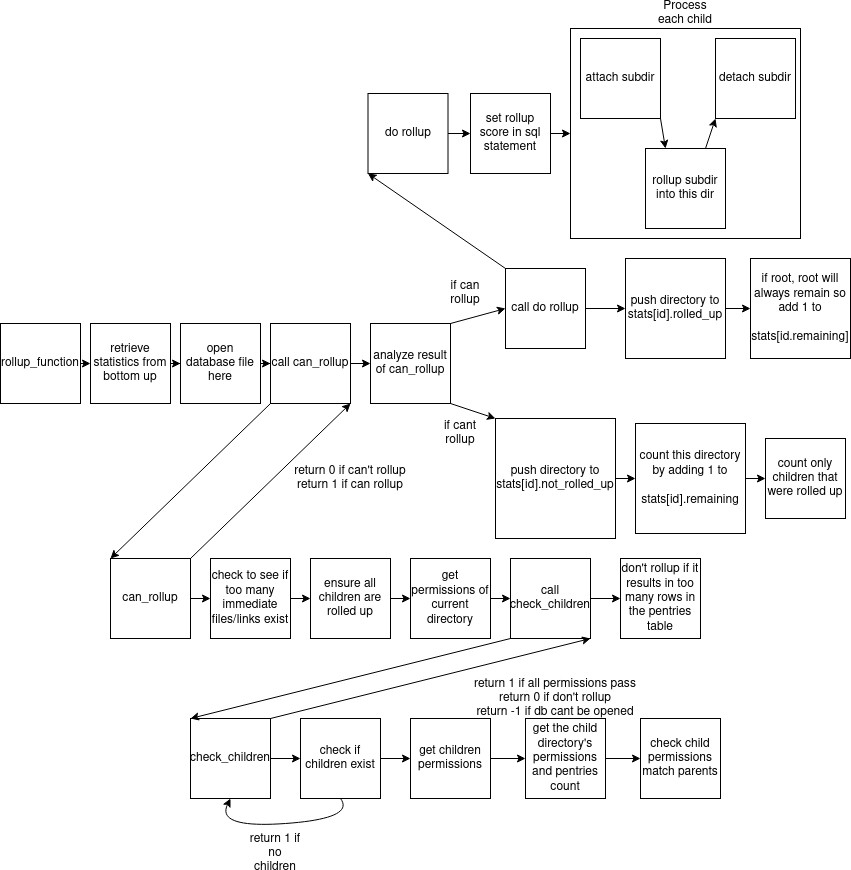


Figure 5: Rollup Function Diagram

## Tree summaries

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.

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 ap- proach will take a long time due to the repeated traversals across the same directories. Because of this, gufi treesummary generates the treesummary ta- ble 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

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 ta- bles 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

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 gener- ation 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.

gufi\_treesummary

usage: ./gufi\_treesummary [options] GUFI\_index

options:

-h help

-H show assigned input values (debugging)

-v version

-P print directories as they are encountered

-n <threads> number of threads

-d <delim> delimiter (one char) [use 'x' for 0x1E]

-X Dry run

GUFI\_index path to GUFI index

gufi\_treesummary\_all

usage: ./gufi\_treesummary\_all [options] GUFI\_index

options:

-h help

-H show assigned input values (debugging)

-v version

-n <threads> number of threads

GUFI\_index path to GUFI index

**It is highly recommended that the GUFI tree be mounted as read only for users as users could destroy the index for themselves if not careful.**

# Querying

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

### 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 should be given higher level utilities or specialized interfaces/web access, etc.

#### Flags

Flag Functionality

|  |  |
| --- | --- |
| -h | help |
| -H | show assigned input values (debugging) |
| -v | version |
| -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 |
| -u | Prefix row with 1 int column count and each column with 1 octet type and 1 size\_t length |
| -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 de-  fault. ex.: INSERT INTO *<*aggregate table name*>* SELECT \* FROM  *<*intermediate table name*>*) |
| -K *<*create aggregate*>* | SQL to create the final aggregation ta-  ble |
| -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 |
| -s <path> | File name prefix for swap files |
| -p <path> | Source path prefix for %s in SQL |
| -e | compress work items |
| -Q <basename>  <table>  <template>.<table>  <view> | External database file basename, per-attach table name, template + table name, and the resultant view |
| -D <start> <stop> | Start and stop path names for partial indexing |

Table 4: gufi query Flags and Arguments

#### Flag/Table Associations

A gufi query invocation should use at least one of the -T, -S, and -E flags to pass in SQL statements[1](#_bookmark76).

* -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](#_bookmark77) and thus associations between flags and tables are not enforced[3](#_bookmark78).

Prior to rolling up, the pentries view can be treated as an optional im- provement 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 con- tains the current directory’s information and subdirectories are not traversed. The summary table was updated directly, so querying it will return all directory information.

When querying, the caller should choose tables and views based on whether or not full paths will be queried. If no, the summary table and pentries, xpentries, or epentries views should be used. If yes, the vrpentries and vrsummary views and their variants should be used. The vrpentries and vrsummary 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 [10](#_bookmark94) for details). These views and function were set up to help simpliy the queries that users provide.

#### 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

1Not passing in any SQL statements results in gufi query simply walking the tree and filling up the inode and dentries caches.

2Anything 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.

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

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.

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

For the extremely proficient user, you can stack multiple queries on any of -T -S and -E separated by a semicolon.

#### 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 xattr 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 them- selves. They are entries, pentries, summary, vrpentries, and vrsummary enhanced with the xattr name and xattr value columns.

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

#### External Databases

Attaching external data to GUFI allows for arbitrary data to be associated with filesystem data. In order to remain flexible and not prescribe schema require- ments, using this feature is more complex than using other GUFI features.

Each external database file is presented as a view concatentating it to a table in the template database file. When a directory does not have a database file specified for attaching to a given view, the view is comprised of only the table from the template file.

Pass -I and -Q to gufi query. -Q has 4 positional arguments:

|  |  |
| --- | --- |
| -Q Argument | Explanation |
| basename | The basename of the database file in the current directory  to attach |
| table | The table in the database file to attach |
| template.table | The template file’s attach name (specified in -I) and table  name with a matching schema |
| view | The name of the view that will be made available for  -S and -E |

Finally, in -S and -E, the caller should join the view named in -Q with new GUFI views esummary, epentries, esummary, epentries, evrsummary, evrxsummary, evrpentries, and evrxpentries to pull data. The recommended columns to join on are name and type to reduce the likelihood of joining to the wrong rows.

Example Usage:

gufi query -I "ATTACH template.db AS template;"

-Q "external.db" "table" "template.table" "ext"

-E "SELECT \* FROM evrpentries

LEFT JOIN ext ON (evrpentries.name == ext.name) AND (evrpentries.type == ext.type);"

index root

For each view that is to be made available for querying, -I should be up- dated with more ATTACH SQL statements, another -Q should be passed in, and each query flag should be updated to use the new views.

Multiple External Databases Example:

gufi query -I "ATTACH template1.db AS template1;

ATTACH template2.db AS template2;"

-Q "externall.db" "table" "template1.table" "ext1"

-Q "external2.db" "table" "template2.table" "ext2"

-E "SELECT \* FROM evrpentries

LEFT JOIN ext1 ON (evrpentries.name == ext1.name) AND (evrpentries.type == ext1.type)

LEFT JOIN ext2 ON (evrpentries.name == ext2.name) AND (evrpentries.type == ext2.type);"

index root

#### 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.

#### 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.

#### 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 paral- lel, -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.

#### The -P %s replacement

The -p flag allows for you to pass in a string and have that string replace a “%s” that you include in your sql statements, allowing you to pass in like a source path for the source file system or other strings

#### Example Calls

Table 5: Parallel Results

|  |  |  |  |
| --- | --- | --- | --- |
| Output | gufi query | | /path/to/index |
| stdout | -S | "SELECT | \* FROM vrsummary;" |
| Per-thread  Files | -E  -o | "SELECT  results | \* FROM vrpentries;" |
| Per-thread  Database Files | -I  -S  -E  -O | "CREATE "INSERT "INSERT  results | TABLE results(name TEXT, size INT64);"  INTO results SELECT name, size FROM vrsummary;" INTO results SELECT name, size FROM vrpentries;" |

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 |

#### 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 practice, rolling up indexes of real filesystems reduces the number of directories (and thus the bottlenecks listed in Section [9.1.1](#_bookmark57)) 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.

#### Visualizing the Workflow

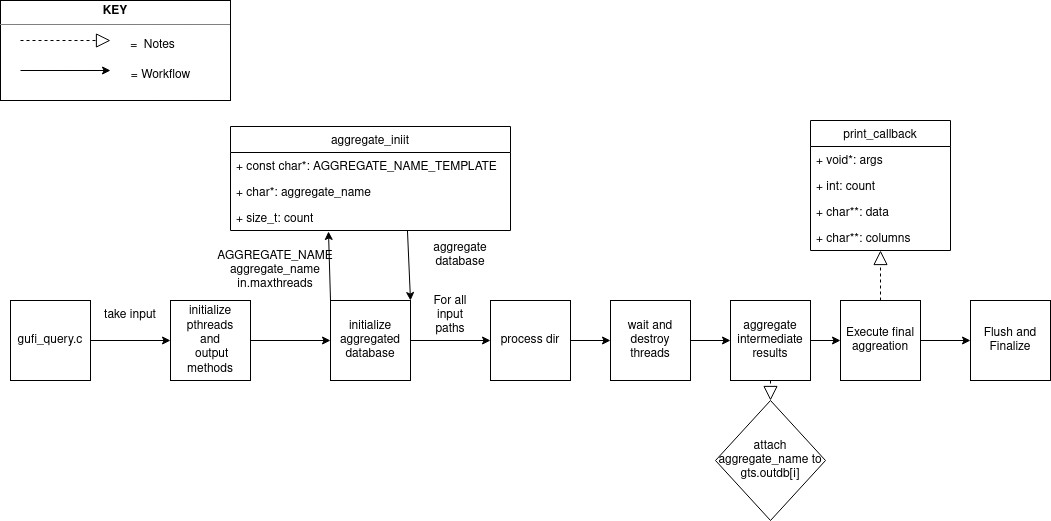


Figure 6: Workflow of gufi query

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.

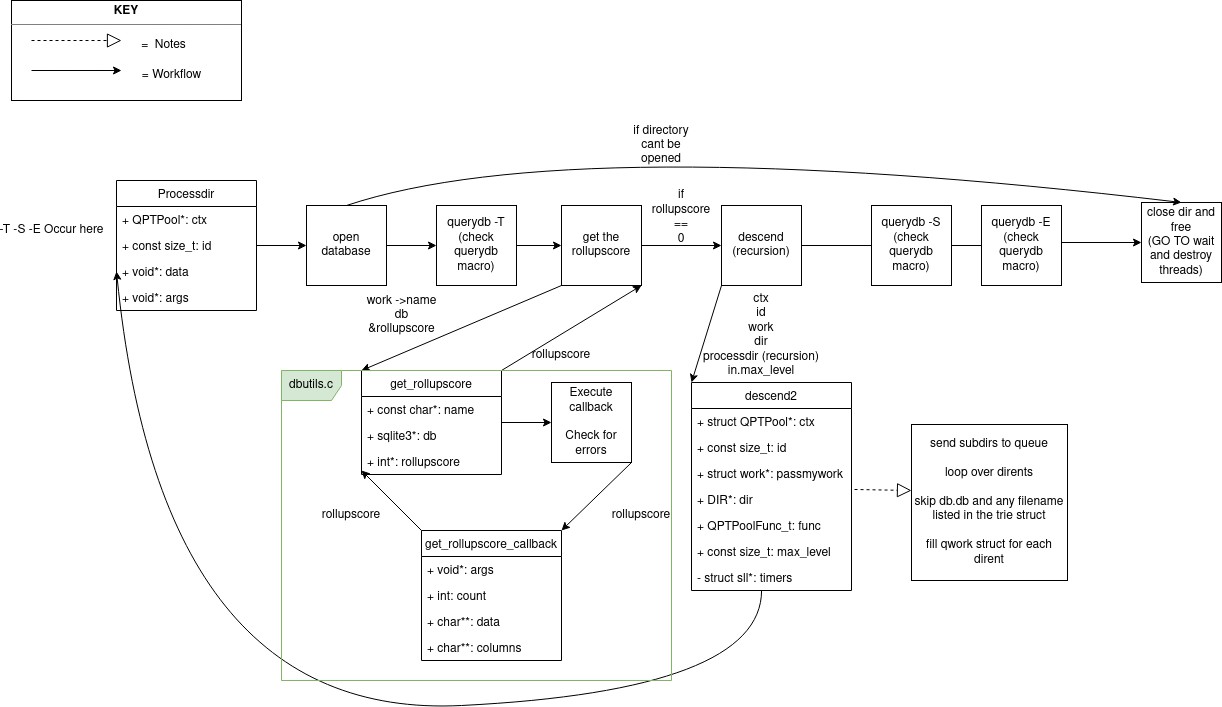


Figure 7: Workflow of processdir

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

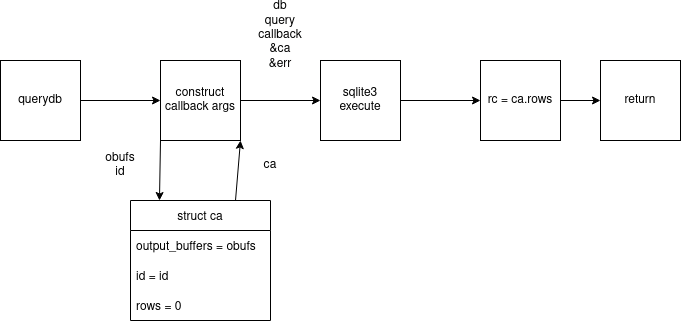


Figure 8: querydb macro workflow

### gufi sqlite3

gufi sqlite3 is a simple program that imitates most of the sqlite3 cli while providing GUFI specific functions. The main difference from the sqlite3 cli is that ”dot commands” are not supported.

#### Flags

Flag Functionality

|  |  |
| --- | --- |
| -h | help manual |
| -d *<*delim*>* | delimiter (one char) [use ’x’ for 0x1E] |

Table 7: gufi sqlite3 Flags and Arguments

#### Usage

gufi sqlite3 [options] [db [SQL]...]

If no SQL statements are passed in, SQL statements will be read in from stdin

## SQLite Functions

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

Table 8: 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](https://www.sqlite.org/lang_datefunc.html) with strftime(3)used for converting unix epoch time stamps which are what is stored in gufi date/time fields into readable text however you choose |
| 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 |
| intop(<OS command> | Runs a OS command as a child process that provides an integer value in standard out |
| strop(<OS command> | Runs a OS command as a child process that provides an string value in standard out |
| blobop(<OS command> | Runs a OS command as a child process that provides an set of text value in standard out |
| setstr(‘variablename’,sql which sets that variable) | Setstr allows you to set a gufi\_query variable from an SQL statement and then use that variable in a later sql statement. For example set a variable in -S and then use that variable in -E (-S “select setstr(‘mycnt’,count(\*)) from vrpentries;” -E “select size/{mycnt} from vrpentries;” |
| regexp(‘regex’,variable) | regexp(‘\w\*\.c$|\w\*. h$’,name) for example is anynumber of word characters followed by a dot followed by c at end of string or followed by h at end of string. This allows the full power of regex string matching for complex string search. |

Table 9: Several math functions have been added to complement the set of built-in math functions provided by SQLite. They return NULL when there are not enough values.

|  |  |
| --- | --- |
| Function | Purpose |
| stdevs(numeric column) | Sample standard deviation |
| stdevp(numeric column) | Population standard deviation |
| median(numeric column) | Median of values |

Table 10: 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(srollsubdirs, sroll) | Number of subdirectories under a directory. Only  available for use with vrsummary. When the index is not rolled up, the value is retrieved from C. When the index is rolled up, the value is retrieved from vrsummary. |

### 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.

**Flags**

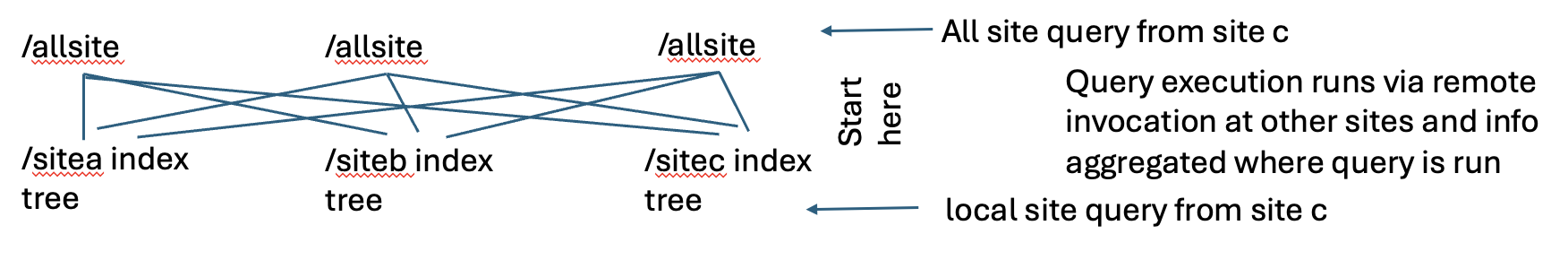
|  |  |
| --- | --- |
| Option | Description |
| -f <FORMAT> | use the specified FORMAT instead of the default.  A newline is outputted after each use of FORMAT. |
| -j | print the information in terse form; Same as  -t/--terse in stat(1). |

# Parallel and Multi-site/Cloud Object Indexing (Work in Progress)

Node parallel query is very possible given GUFI indexes are composable and therefore can be sharded as long as the security in the subtree sharded is maintained. Work is going on to have a top level query that can push the query work to lower level servers that have different subtrees of the GUFI index.

Multi-site indexing and query is possible. Multiple sites with the same uid/gid/identity scheme and security basis is simple, in that the index be built on any system at any site and each part can be grafted together into a single index given that GUFI indexes are composable.

Multiple sites that do not have the same uid/gid/identity scheme but have mapping that allows users to log into multiple sites just with different identities etc. Enabling this can utilize the Node parallel query where remote invocations that map users to sites can be used to run a single query across multiple sites and pass the portion of results back to where the query is invoked from. This is depicted this diagram below.



# Utility Tools

## Source Tree Reconstruction

Occasionally, it may be useful to reconstruct the original source tree struc- ture. Two utility tools have been created to do so: gufi trace2dir and gufi index2dir. As their names imply, they convert traces and index and convert them back into source directories, with files and symlinks. Note that currently, files are created and left with the size at 0. They are not set to their real sizes because users likely do not have the storage space available to them. Files with hole are not created with fallocate(2) in order to not use non-portable functions. This may change in the future.

Note that gufi index2trace is currently not available, but may be in the future.

gufi dir2index



index

dir

gufi dir2trace

gufi trace2index

gufi trace2dir

trace

gufi index2dir

Figure 9: Tools for Transforming Between Source Trees, Traces, and Indexes

* + 1. gufi trace2dir

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] |
| -x | pull xattrs from source file-sys into GUFI |

Table 11: gufi trace2dir Flags and Arguments

#### Flags

* + - 1. **Usage**

gufi trace2dir [options] trace file... output dir

gufi index2dir

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 |

Table 12: gufi index2dir Flags and Arguments

#### Flags

**Usage**

gufi index2dir [options] input dir output dir

# Deployment

GUFI functionality as described so far has been local (source filesystem and index on the same node). However, this is not the expected way to deploy GUFI at scale. Users should not log into the machine containing both the filesystems and indexes with an interactive shell. Instead, indexes are to be placed on a dedicated GUFI server (which may or may not be able to see the source filesystems) and users will call client scripts from client nodes to forward commands to the server via ssh.

There are multiple reasons why GUFI is being deployed this way. First, indexes should not be colocated with the source filesystems in order to not use the resources of the source filesystem nodes. Second, filesystems may not be visible to each other even if the indexes should be (the indexes can all be colocated with of one of the source filesystems, but then we are back at the first issue). Finally, the indexes are placed on a dedicated server instead of a frontend node because front-end nodes tend to be underpowered and are intended to be used as jumping-off points to more powerful nodes, but GUFI queries, which will be running in parallel and may be complex, will need a lot of resources.

## User and Index Setup

In order to prevent users from seeing data they should not see, first ensure that the uids and gids point to the same people across all filesystems being indexed. This should not be an issue for filesystems whose users have fixed uids and gids.

The /etc/passwd from each filesystem’s source system should be merged into the existing one on the server in order to set up the users. The login shells of every user should be set to /usr/sbin/nologin so that users cannot log in directly (see the SSH section for letting users in). The /etc/group from each filesystem’s source system should be similarly merged into the server’s /etc/group.

Individual indexes can then be moved onto the server, likely with gufi\_trace2index. Multiple indexes can be combined by simply placing them under a common location. For example, when combining the indexes projects and scratch, they can be both placed under search, and the tools can be pointed to search to search both indexes at once, or search/projects or search/scratch to search them individually. Note that search should (but does not have to) have an empty db.db (the tables exist but are all empty) file in it.

## RPMs

The build should be (re)configured with cmake -DCLIENT=On if it is not already enabled. make package will produce two RPMs instead of one. The server RPM will contain the actual GUFI implementation. The client RPM will contain renamed and modified gufi\_client wrapper scripts. These RPMs should be installed on their respective nodes.

### Wrapper Scripts

gufi\_query provides the core GUFI functionality. However, it is not expected to be used by users, as the syntax of gufi\_query is somewhat unwieldy, and users are not expected to understand how GUFI works. Wrapper scripts such as gufi\_find, gufi\_stat, gufi\_du, and gufi\_ls were created in order to provide users (and administrators) with predetermined queries that are used often and extract useful information (see the user guide for more details). The server has the actual implementation of the wrapper scripts while the client has multiple copies of gufi\_client renamed to the corresponding script’s name.

## GUFI\_find

GUFI version of find

A wrapper of gufi\_query that presents a near POSIX find command.

See command syntax for usage information.

### GUFI\_ls

GUFI version of ls

A wrapper of gufi\_query that presents a near POSIX ls command

See command syntax for usage information.

### GUFI\_stats

GUFI statistics

A wrapper of gufi\_query that calculates handy statistics over a tree.

See command syntax for usage information.

### GUFI\_du

GUFI disk usage

A wrapper of gufi\_query that calculates handy statistics over a tree.

See command syntax for usage information.

## Runtime Configuration

It is highly recommended that the GUFI tree be mounted as read only for users as users could destroy the index for themselves if not careful.

The wrapper scripts on both the server and client require a valid GUFI configuration file to function correctly. The path to the configuration file can be set at configure time with the CONFIG\_SERVER and CONFIG\_CLIENT CMake variables (both default to /etc/GUFI/config). The configuration file will be readable by all users but should only be writable by the administrators (664 permissions) in order to prevent non-admin users from changing how GUFI runs.

GUFI configuration files are simple text files containing lines of <key>=<value>. Duplicate configuration keys are overwritten by the bottom-most line. Empty lines and lines starting with # are ignored. The server and client require different configuration values. Example configuration files can be found in the config directory of the GUFI source, as well as in the build directory. The corresponding examples will also be installed with GUFI.

**Server**

|  |  |  |
| --- | --- | --- |
| Key | Value Type | Purpose |
| Threads | Positive Integer | Number of threads to use  when running gufi query. |
| Query | File Path | The absolute path of gufi query. |
| Sqlite3 | File Path | The absolute path of gufi\_sqlite3 |
| Stat | File Path | The absolute path of gufi stat bin. |
| IndexRoot | Directory Path | The absolute path of a GUFI tree  (or parent of multiple indexes). |
| OutputBuffer | Non-negative Integer | The size of each per-thread buffer  used to buffer writes to stdout. Setting this too low will affect performance. Setting this too high will use too much resources. |

**Client**

|  |  |  |
| --- | --- | --- |
| Key Value | Type | Purpose |
| Server | URI | The URI of the server (IP address, URL, etc.) |
| Port | Non-negative integer | Port number |

### Colocating the Server and Client

Installing both the server and client on the same node with either the RPMs or make install results in a bad setup where either the server wrapper scripts overwrite the client scripts or vice versa and should not be done.

The only time when it makes sense for both the server and client to be on the same node at the same time is during testing. When testing, the client wrapper scripts should be run out of the GUFI build directory. There, the client wrapper scripts will be called gufi\_client\_<name>, e.g., gufi\_client\_ls. Additionally, the server and client configurations can be merged into the same file if both the server and client read from the same configuration file path.

## SSH

### Authentication

Normal ssh, using passwords or user keys (~/.ssh/id\_rsa), should be disabled on the server. Instead, ssh should be done using host-based authentication (/etc/ssh/ssh\_host\_rsa\_key).

### gufi\_ jail

A restricted shell should be set up on the server to prevent arbitrary command execution.

gufi\_jail is a simple script that limits the commands users logging in with ssh are able to run to only a subset of GUFI commands. /etc/ssh/sshd\_config should be modified with the following lines:

Match Group gufi

ForceCommand /path/to/gufi\_jail

# Extensibility

As you can see from the schemas from these tables, these fields below are unused by gufi base and are reserved for site specific use in all the tables.

* + - * minossint1 INT64 min ossint1
      * maxossint1 INT64 max ossint1
      * totossint1 INT64 summed ossint1
      * minossint2 INT64 min ossint2
      * maxossint2 INT64 max ossint2
      * totossint2 INT64 summed ossint2
      * minossint3 INT64 min ossint3
      * maxossint3 INT64 max ossint3
      * totossint3 INT64 summed ossint3
      * minossint4 INT64 min ossint4
      * maxossint4 INT64 max ossint4
      * totossint4 INT64 summed ossint4
      * osstext1 TEXT storage system specific string character
      * osstext2 TEXT storage system specific string character

You can of course add entirely new tables in each database and populate and as long as you have inode or name in that new table, you can join with any entries and summary and pentries tables/views.

To populate these fields/tables you can always use gufi\_query as root with -w to allow writing.

For example gufi\_query -n 10 -E’create table tree.newtable …..’ myindextree

\*\*\* notice the tree.table name, this is required when you create new tables/views in GUFI

Then perhaps gufi\_query -n 10 -E’insert into newtable blah blah ..’ myindextree

Remember if you only want to traverse a few levels, gufu\_query can be told how far down to traverse. Also remember you can always use

Find -type d -maxdepth -level | xarg \_\_\_\_ gufi\_query. To run it at just one level in the tree too.

Also, it is possible to add entirely new sqlite databases at each directory. We have contemplated and likely will enable this via the attach feature of sqlite to have gufi\_query attach to your custom database and then you can join with gufi db tables if you have join fields etc. If you desire this feature feel free to let us know.

# GUFI query deeper look

This is a simple test tree

ggrider@pn2201328 bin % find test

test

test/l1.2

test/l1.2/l2.2

test/l1.2/l2.2/f1.xls

test/l1.2/l2.3

test/l1.2/l2.3/f1.xls

test/l1.2/f2.1.doc

test/l1.2/l2.1

test/l1.2/l2.1/f1.xls

test/l1.3

test/l1.3/l2.2

test/l1.3/l2.2/f1.bigger

test/l1.3/l2.3

test/l1.3/l2.3/f1.big

test/l1.3/f3.1.doc

test/l1.3/l2.1

test/l1.3/l2.1/f1.biggest

test/l1.1

test/l1.1/l2.2

test/l1.1/l2.2/f1.exe

test/l1.1/l2.2/f1.tar

test/l1.1/l2.3

test/l1.1/l2.3/f1.exe

test/l1.1/l2.3/f1.tar

test/l1.1/f2.1.doc

test/l1.1/l2.1

test/l1.1/l2.1/f1.exe

test/l1.1/l2.1/f1.tar

test/l1.1/f1.1.doc

make a gufi index

ggrider@pn2201328 bin % ./gufi\_dir2index test testi

ggrider@pn2201328 bin % testi

testi

testi/test

testi/test/l1.2

testi/test/l1.2/l2.2

testi/test/l1.2/l2.2/db.db

testi/test/l1.2/l2.3

testi/test/l1.2/l2.3/db.db

testi/test/l1.2/db.db

testi/test/l1.2/l2.1

testi/test/l1.2/l2.1/db.db

testi/test/l1.3

testi/test/l1.3/l2.2

testi/test/l1.3/l2.2/db.db

testi/test/l1.3/l2.3

testi/test/l1.3/l2.3/db.db

testi/test/l1.3/db.db

testi/test/l1.3/l2.1

testi/test/l1.3/l2.1/db.db

testi/test/l1.1

testi/test/l1.1/l2.2

testi/test/l1.1/l2.2/db.db

testi/test/l1.1/l2.3

testi/test/l1.1/l2.3/db.db

testi/test/l1.1/db.db

testi/test/l1.1/l2.1

testi/test/l1.1/l2.1/db.db

testi/test/db.db

See that there is a db.db in every directory, that is the gufi database per directory in the index tree.

## Basic Query

Please do not write queries against the tables directly, use vrsummary and vrpentries.

Please note that the use of datetime() is deprecated, please use strftime(format, timestamp)

The entries table is where the files/links are kept

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select name,type,size,uid,gid,atime from entries;' -d ' ' testi/test

f2.1.doc f 2 2078 20 1680818366

f3.1.doc f 2 2078 20 1680818374

f2.1.doc f 2 2078 20 1680818358

f1.1.doc f 2 2078 20 1680818345

f1.xls f 0 2078 20 1680817472

f1.xls f 0 2078 20 1680817476

f1.xls f 0 2078 20 1680817469

f1.bigger f 7 2078 20 1680817673

f1.big f 4 2078 20 1680817657

f1.biggest f 10 2078 20 1680817701

f1.exe f 0 2078 20 1680817412

f1.tar f 0 2078 20 1680817430

f1.exe f 0 2078 20 1680817415

f1.tar f 0 2078 20 1680817424

f1.exe f 0 2078 20 1680817390

f1.tar f 0 2078 20 1680817439

The same thing is available using vrpentries

Please use this instead if you write queries

The entries table is where the files/links are kept

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select name,type,size,uid,gid,atime from vrpentries;' -d ' ' testi/test

f2.1.doc f 2 2078 20 1680818366

f3.1.doc f 2 2078 20 1680818374

f2.1.doc f 2 2078 20 1680818358

f1.1.doc f 2 2078 20 1680818345

f1.xls f 0 2078 20 1680817472

f1.xls f 0 2078 20 1680817476

f1.xls f 0 2078 20 1680817469

f1.bigger f 7 2078 20 1680817673

f1.big f 4 2078 20 1680817657

f1.biggest f 10 2078 20 1680817701

f1.exe f 0 2078 20 1680817412

f1.tar f 0 2078 20 1680817430

f1.exe f 0 2078 20 1680817415

f1.tar f 0 2078 20 1680817424

f1.exe f 0 2078 20 1680817390

f1.tar f 0 2078 20 1680817439

This is not really that useful though, what is the directory structure for it, what are the user names and dates etc. (use the gufi functions)

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select path(),name,type,size,uidtouser(uid),gidtogroup(gid),datetime(atime,"unixepoch"),modetotxt(mode) from vrpentries;' -d ' ' testi/test

testi/test/l1.2 f2.1.doc f 2 ggrider staff 2023-04-06 21:59:26 -rw-r--r--

testi/test/l1.3 f3.1.doc f 2 ggrider staff 2023-04-06 21:59:34 -rw-r--r--

testi/test/l1.1 f2.1.doc f 2 ggrider staff 2023-04-06 21:59:18 -rw-r--r--

testi/test/l1.1 f1.1.doc f 2 ggrider staff 2023-04-06 21:59:05 -rw-r--r--

testi/test/l1.2/l2.2 f1.xls f 0 ggrider staff 2023-04-06 21:44:32 -rw-r--r--

testi/test/l1.2/l2.3 f1.xls f 0 ggrider staff 2023-04-06 21:44:36 -rw-r--r--

testi/test/l1.2/l2.1 f1.xls f 0 ggrider staff 2023-04-06 21:44:29 -rw-r--r--

testi/test/l1.3/l2.2 f1.bigger f 7 ggrider staff 2023-04-06 21:47:53 -rw-r--r--

testi/test/l1.3/l2.3 f1.big f 4 ggrider staff 2023-04-06 21:47:37 -rw-r--r--

testi/test/l1.3/l2.1 f1.biggest f 10 ggrider staff 2023-04-06 21:48:21 -rw-r--r--

testi/test/l1.1/l2.2 f1.exe f 0 ggrider staff 2023-04-06 21:43:32 -rw-r--r--

testi/test/l1.1/l2.2 f1.tar f 0 ggrider staff 2023-04-06 21:43:50 -rw-r--r--

testi/test/l1.1/l2.3 f1.exe f 0 ggrider staff 2023-04-06 21:43:35 -rw-r--r--

testi/test/l1.1/l2.3 f1.tar f 0 ggrider staff 2023-04-06 21:43:44 -rw-r--r--

testi/test/l1.1/l2.1 f1.exe f 0 ggrider staff 2023-04-06 21:43:10 -rw-r--r--

testi/test/l1.1/l2.1 f1.tar f 0 ggrider staff 2023-04-06 21:43:59 -rw-r--r--

Well, that’s nicer, path() and the other build in functions make this way more useful.

However we highly recommend using rpath(sname,sroll) instead of path() because it will work in rollup trees and well as non-rollup trees

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select rpath(sname,sroll),name,type,size,uidtouser(uid),gidtogroup(gid),datetime(atime,"unixepoch"),modetotxt(mode) from vrpentries;' -d ' ' testi/test

test/l1.2 f2.1.doc f 2 ggrider staff 2023-04-11 01:46:40 -rw-r--r--

test/l1.3 f3.1.doc f 2 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1 f1.1.doc f 2 ggrider staff 2023-04-11 01:46:40 -rw-r--r--

test/l1.1 f2.1.doc f 2 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.2/l2.2 f1.xls f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.2/l2.3 f1.xls f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.2/l2.1 f1.xls f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.3/l2.2 f1.bigger f 7 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.3/l2.3 f1.big f 4 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.3/l2.1 f1.biggest f 10 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1/l2.2 f1.exe f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1/l2.2 f1.tar f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1/l2.3 f1.exe f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1/l2.3 f1.tar f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1/l2.1 f1.exe f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1/l2.1 f1.tar f 0 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

Another neat feature is you can stack multiple sql statements inside -T, -S, -E clauses

See this:

ggrider@pn2201328 bin % ./gufi\_query -n1 -S'select "D-",rpath(sname,sroll),name from vrsummary;select "F-",rpath(sname,sroll),name from vrpentries;' -d " " testi/test

D- test test

D- test/l1.2 l1.2

F- test/l1.2 f2.1.doc

D- test/l1.3 l1.3

F- test/l1.3 f3.1.doc

D- test/l1.1 l1.1

F- test/l1.1 f1.1.doc

F- test/l1.1 f2.1.doc

D- test/l1.2/l2.2 l2.2

F- test/l1.2/l2.2 f1.xls

D- test/l1.2/l2.3 l2.3

F- test/l1.2/l2.3 f1.xls

D- test/l1.2/l2.1 l2.1

F- test/l1.2/l2.1 f1.xls

D- test/l1.3/l2.2 l2.2

F- test/l1.3/l2.2 f1.bigger

D- test/l1.3/l2.3 l2.3

F- test/l1.3/l2.3 f1.big

D- test/l1.3/l2.1 l2.1

F- test/l1.3/l2.1 f1.biggest

D- test/l1.1/l2.2 l2.2

F- test/l1.1/l2.2 f1.exe

F- test/l1.1/l2.2 f1.tar

D- test/l1.1/l2.3 l2.3

F- test/l1.1/l2.3 f1.exe

F- test/l1.1/l2.3 f1.tar

D- test/l1.1/l2.1 l2.1

F- test/l1.1/l2.1 f1.exe

F- test/l1.1/l2.1 f1.tar

Notice it runs the -S before running the -E query for each directory, this is technically what we call a compound query which is described in detail later.

Can we use the where clause?

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select rpath(sname,sroll),name,type,size,uidtouser(uid),gidtogroup(gid),datetime(atime,"unixepoch"),modetotxt(mode) from vrpentries where size>6;' -d ' ' testi/test

test/l1.3/l2.2 f1.bigger f 7 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.3/l2.1 f1.biggest f 10 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

notice only files bigger than 6 bytes

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select rpath(sname,sroll),name,type,size,uidtouser(uid),gidtogroup(gid),datetime(atime,"unixepoch"),modetotxt(mode) from vrpentries where name like "%big%";' -d ' ' testi/test

test/l1.3/l2.2 f1.bigger f 7 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.3/l2.3 f1.big f 4 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.3/l2.1 f1.biggest f 10 ggrider staff 2023-04-11 01:46:39 -rw-r--r—

Notice only files with big in the name

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select rpath(sname,sroll),name,type,size,uidtouser(uid),gidtogroup(gid),datetime(atime,"unixepoch"),modetotxt(mode) from vrpentries where dname like "%l1%";' -d ' ' testi/test

test/l1.2 f2.1.doc f 2 ggrider staff 2023-04-11 01:46:40 -rw-r--r--

test/l1.3 f3.1.doc f 2 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

test/l1.1 f1.1.doc f 2 ggrider staff 2023-04-11 01:46:40 -rw-r--r--

test/l1.1 f2.1.doc f 2 ggrider staff 2023-04-11 01:46:39 -rw-r--r--

Notice all the files that are in a directory with l1 in the name, this is very powerful and a feature of using vrpentries, you could ask for file info about files that are in directories with any attributes from the directory without a join (we did the join for you)

Of course queries against the summary table are similarly powerful.

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select rpath(sname,sroll),dname,totfiles,maxsize from vrsummary;' -d ' ' testi/test

test test 0 -9223372036854775808

test/l1.2 l1.2 1 2

test/l1.3 l1.3 1 2

test/l1.1 l1.1 2 2

test/l1.2/l2.2 l2.2 1 0

test/l1.2/l2.3 l2.3 1 0

test/l1.2/l2.1 l2.1 1 0

test/l1.3/l2.2 l2.2 1 7

test/l1.3/l2.3 l2.3 1 4

test/l1.3/l2.1 l2.1 1 10

test/l1.1/l2.2 l2.2 2 0

test/l1.1/l2.3 l2.3 2 0

test/l1.1/l2.1 l2.1 2 0

Notice the top directory didn’t have any files so no maxsize.

## Advanced SQL queries

So can we use the limit clause in sql?

Yes, BUT remember we are running a query for each dir, so the limit clause limits each query, not the total output

What about order by and group by? Same problem exists, it will be ordered or grouped by in every directory separately.

ggrider@pn2201328 bin % ./gufi\_query -n1 -E'select rpath(sname,sroll),name,size from vrpentries limit 1;' -d ' ' testi/test

test/l1.2 f2.1.doc 2

test/l1.3 f3.1.doc 2

test/l1.1 f1.1.doc 2

test/l1.2/l2.2 f1.xls 0

test/l1.2/l2.3 f1.xls 0

test/l1.2/l2.1 f1.xls 0

test/l1.3/l2.2 f1.bigger 7

test/l1.3/l2.3 f1.big 4

test/l1.3/l2.1 f1.biggest 10

test/l1.1/l2.2 f1.exe 0

test/l1.1/l2.3 f1.exe 0

test/l1.1/l2.1 f1.exe 0

Yes, a single file per directory (limit 1)

But we really want to limit the entire output, or sort the entire output or group by the total output, etc

Well also notice that up until now I have been using -n1. One thread to query, ouch for big queries. That is because if you use > 1 thread while the query will run, the output will get jumbled because many threads writing to stdout.

How do we deal with this?

With interim per thread output tables and aggregate table (in memory or in /tmp space temp db’s tables) you can solve this problem.

ggrider@pn2201328 bin % ./gufi\_query -n3 -I'create table out (oname TEXT);' -E'insert into out select rpath(sname,sroll) || "/" || name from vrpentries;' -K'CREATE TABLE aggregate (fname TEXT);' -J 'insert into aggregate (fname) select oname from out;' -G'select fname from aggregate;' -d ' ' testi/test

test/l1.3/f3.1.doc

test/l1.2/l2.3/f1.xls

test/l1.3/l2.2/f1.bigger

test/l1.1/l2.3/f1.exe

test/l1.1/l2.3/f1.tar

test/l1.1/f1.1.doc

test/l1.1/f2.1.doc

test/l1.2/l2.1/f1.xls

test/l1.3/l2.3/f1.big

test/l1.1/l2.1/f1.exe

test/l1.1/l2.1/f1.tar

test/l1.2/f2.1.doc

test/l1.2/l2.2/f1.xls

test/l1.3/l2.1/f1.biggest

test/l1.1/l2.2/f1.exe

test/l1.1/l2.2/f1.tar

./gufi\_query -n3 -I'create table out (oname TEXT);' -E'insert into out select rpath(sname,sroll) || "/" || name from vrpentries;' -K'CREATE TABLE aggregate (fname TEXT);' -J 'insert into aggregate (fname) select oname from out;' -G'select fname from aggregate;' -d ' ' testi/test

Notice -n3

-I creates an output table per thread for each thread to write results to, notice -E now is insert into select,

Notice the fancy sqlite || concat function

-K create aggregate table, -j insert from all the per thread out tables into the aggregate table, then -G query the aggregate table to see my output

-J does the query you want but puts the result into the output db

-G pulls the output db records and concats them into the aggregation table

Wow, this is pretty fancy.

So it enabled -n3 but what else could it do?

ggrider@pn2201328 bin % ./gufi\_query -n3 -I'create table out (oname TEXT);' -E'insert into out select rpath(sname,sroll) || "/" || name from vrpentries;' -K'CREATE TABLE aggregate (fname TEXT);' -J 'insert into aggregate (fname) select oname from out;' -G'select fname from aggregate order by fname asc;' -d ' ' testi/test

test/l1.1/f1.1.doc

test/l1.1/f2.1.doc

test/l1.1/l2.1/f1.exe

test/l1.1/l2.1/f1.tar

test/l1.1/l2.2/f1.exe

test/l1.1/l2.2/f1.tar

test/l1.1/l2.3/f1.exe

test/l1.1/l2.3/f1.tar

test/l1.2/f2.1.doc

test/l1.2/l2.1/f1.xls

test/l1.2/l2.2/f1.xls

test/l1.2/l2.3/f1.xls

test/l1.3/f3.1.doc

test/l1.3/l2.1/f1.biggest

test/l1.3/l2.2/f1.bigger

test/l1.3/l2.3/f1.big

This ordered by the entire path because we did this in the insert query. insert into out select rpath(sname,sroll) || "/" || name

We concatenated path and name into one field and then inserted that into the output databases and then aggregated into fname.

This is just a feature of SQL of course but powerful none-the-less.

How about something really interesting like this?

ggrider@pn2201328 bin % ./gufi\_query -n1 -I'create table out (oext TEXT,ocnt INT64);' -E'insert into out select ltrim(name,rtrim(name,replace(name,".",""))) as myext,count(inode) from entries group by myext;' -K'CREATE TABLE aggregate (fext TEXT,fcnt INT64);' -J 'insert into aggregate (fext,fcnt) select oext,ocnt from out;' -G'select fext,sum(fcnt) from aggregate group by fext;' -d ' ' testi/test

big 1

bigger 1

biggest 1

doc 4

exe 3

tar 3

xls 3

Ok, what is this?

First, we make an output table per thread out with oext an ocnt

Then we insert into out a query against entries that counts the number of files for each file extension (after the last dot). Yes ltrim(name,rtrim(name,replace(name,".",""))) is a fancy way to get everything after the last dot. It replaces all the dots in the name then it rtrims to get a string on the left side of the last dot then ltrim to get rid of that part – there may be more elegant things to do here (looks like there is an open src set of functions for this we don’t have maybe we should get <https://observablehq.com/@asg017/introducing-sqlite-path> )

So each thread has a table with all the extensions and counts that the thread encountered in its work.

Then we make an aggregate table with fext and fcnt

Then we load up the aggregate table from all the thread tables

Then we query the aggregate table but instead of count, we sum group by fext and presto, we have a list of the file extensions and the total number of files across the entire tree. We tried to let the threads do as much work as they could and just let the aggregate work do the last sum/group by.

Yes indeed, this is getting very powerful now, imagine how hard that would be with find and sort and merge.

We can also limit traversal depths and starting points as well.

**./gufi\_query -n1 -d'|' -E "select path()||'/'||name,size from vrpentries;" testidx**

\*\*\* start at the top

testidx/l1.2/f2.1.doc|2

…

testidx/l1.1/l2.1/f1.tar|0

**./gufi\_query -n1 -d'|' -E "select path()||'/'||name,size from vrpentries;" testidx/l1.1**

\*\*\* start lower

testidx/l1.1/f1.1.doc|2

…

testidx/l1.1/l2.1/f1.tar|0

**./gufi\_query -n1 -d'|' -z1 -E "select path()||'/'||name,size from vrpentries;" testidx**

\*\*\* maxdepth -z

testidx/l1.2/f2.1.doc|2

…

testidx/l1.1/f2.1.doc|2

**./gufi\_query -n1 -d'|' -z2 -y2 -E "select path()||'/'||name,size from vrpentries;" testidx**

\*\*\* maxdept –z mindepth -y

testidx/l1.2/l2.2/f1.xls|0

…

testidx/l1.1/l2.1/f1.tar|0

## Compound Queries

Compound queries is yet another powerful concept that can be used within the bfq program. Recall that you can control the SQL related work in bfq with the following options:

-T <SQL\_tsum> SQL for tree-summary table

-S <SQL\_sum> SQL for summary table

-E <SQL\_ent> SQL for entries table

-O <out\_DB> output DB one per thread

-I <SQL\_init> SQL init

-F <SQL\_fin> SQL cleanup

It is important to understand the order in which these operations occur to understand the power of compound queries.

The order is as follows:

* Output DB’s opened if called for one per thread
* Single SQL statement run (SQL\_init) once per thread
* Loop: Walk the GUFI tree and assign directories to threads, so a thread handles one full directory at a time
  + Thread works its way through a single directory
  + Multiple SQL statement run once for the directory SQL\_tsum
  + If overall “and” flag is off (an “or” condition) or if final SQL statement in SQL\_tsum produces < 1 records
    - Multiple SQL statement run once for the directory SQL\_sum
    - If overall “and” flag is off (an “or” condition) or if final SQL statement in SQL\_sum produces < 1 records
      * Records are put into the output
      * Multiple SQL statement run once for the directory SQL\_ent
      * If overall “and” flag is off (an “or” condition) or if final SQL statement in SQL\_ent produces < 1
        + Records are put into the output
      * endif
    - endif
  + endif
* endloop
* Single SQL statement run (SQL\_fin) once per thread

As you can see that if you want to run many sql statements you can run them with branches based on outcome. These sql statements can really be anything, they don’t even have to involved GUFI databases solely for that matter. If you want to use the “and” logic the last sql statement in a group must retrieve at least one record.

Within each group (SQL\_tsum, SQL\_sum, SQL\_ent) there can be multiple SQL statements separated by semicolons, so you can be as innovative as you would like.

As you can see, this flexibility to run many SQL statements one time per thread and many times per directory is a very powerful concept.

Now we should look at compound queries in our example data, this is using the -a flag in gufi\_query

You can do a query that uses

-S’select something about summary.’

And also

-E’select something about entries.”

If you use -a it means that it will run the -S query and then the -E query for every directory (this is a compound “or”

But if you don’t use the -a it is a compound “and” which means that it will only run the -E query if the -S query retrieves at least one row

ggrider@pn2201328 bin % ./gufi\_query -n1 -S'select "dir",rpath(sname,sroll),dname from vrsummary where dname like "%l2%";' -E'select name,size from vrpentries;' -d " " -a testi/test

f2.1.doc 2

f3.1.doc 2

f2.1.doc 2

f1.1.doc 2

dir testi/test/l1.2/l2.2 l2.2

f1.xls 0

dir testi/test/l1.2/l2.3 l2.3

f1.xls 0

dir testi/test/l1.2/l2.1 l2.1

f1.xls 0

dir testi/test/l1.3/l2.2 l2.2

f1.bigger 7

dir testi/test/l1.3/l2.3 l2.3

f1.big 4

dir testi/test/l1.3/l2.1 l2.1

f1.biggest 10

dir testi/test/l1.1/l2.2 l2.2

f1.exe 0

f1.tar 0

dir testi/test/l1.1/l2.3 l2.3

f1.exe 0

f1.tar 0

dir testi/test/l1.1/l2.1 l2.1

f1.exe 0

f1.tar 0

you see there are some directories missing (because they don’t match the like “%l2%”

but without the -a

ggrider@pn2201328 bin % ./gufi\_query -n1 -S'select "dir",rpath(sname,sroll),dname from vrsummary where dname like "%l2%";' -E'select name,size from vrpentries;' -d " " testi/test

dir testi/test/l1.2/l2.2 l2.2

f1.xls 0

dir testi/test/l1.2/l2.3 l2.3

f1.xls 0

dir testi/test/l1.2/l2.1 l2.1

f1.xls 0

dir testi/test/l1.3/l2.2 l2.2

f1.bigger 7

dir testi/test/l1.3/l2.3 l2.3

f1.big 4

dir testi/test/l1.3/l2.1 l2.1

f1.biggest 10

dir testi/test/l1.1/l2.2 l2.2

f1.exe 0

f1.tar 0

dir testi/test/l1.1/l2.3 l2.3

f1.exe 0

f1.tar 0

dir testi/test/l1.1/l2.1 l2.1

f1.exe 0

f1.tar 0

and here both directories and files are missing because the “and” cause some -E queries to not be run

You can also use the vrsummary table query to minimize the vrpentries query.

**If you query vrpentries and use directory info related where clauses, you are scanning all the files, if you want to first query the vrsummary table and only if you need to scan the vrpentries table, you can use –S and –E, there is an implied AND (if –S then do –E)**

**./gufi\_query -n1 -d'|' -S "select path(),totfiles from vrsummary where totfiles>1;" -E "select path()||'/'||name,size from vrsummary;" testidx**

\*\*\* notice –S and -E

testidx/l1.1|2

testidx/l1.1/l1.1|224

testidx/l1.1/l2.2|2

testidx/l1.1/l2.2/l2.2|128

testidx/l1.1/l2.3|2

testidx/l1.1/l2.3/l2.3|128

testidx/l1.1/l2.1|2

testidx/l1.1/l2.1/l2.1|128

**./gufi\_query -n1 -d'|' -E "select path()||'/'||name,size from vrsummary where totsize>1;" testidx**

\*\*\* remember vrpentries has vrsummary info in it

testidx/l1.2/l1.2|192

testidx/l1.3/l1.3|192

testidx/l1.1/l1.1|224

testidx/l1.3/l2.2/l2.2|96

testidx/l1.3/l2.3/l2.3|96

testidx/l1.3/l2.1/l2.1|96

We also don’t want to forget how powerful the treesummary table is.

**Powerful aggregate information in tree summary, summarizes everything below, at every level, which trees have this in them**

**DU command is now free (milliseconds)**

**./gufi\_query -n1 -d'|' -z1 -T "select path(),totsubdirs,totsize,maxsize from treesummary;" testidx**

\*\*\* notice –z is maxdepth and –T is used for treesummary queries

testidx|12|29|10

testidx/l1.2|3|2|2

testidx/l1.3|3|23|10

testidx/l1.1|3|4|2

**./gufi\_query -n1 -d'|' -z1 -T "select path(),totsubdirs,totsize,maxsize from treesummary where totxattr>0;" testidx**

\*\*\* we just queried which subtrees have any xattrs

testidx|12|29|10

testidx/l1.1|3|4|2

**./gufi\_query -n1 -d'|' -z1 -y1 -T "select 'trees that have xattrs '||path() from treesummary where totxattr>0;" -E "select path()||'/'||name from vrpentries where xattr\_names not null;" testidx**

\*\*\* -z and –y limit us to looking at level 1 tree summaries

\*\*\* -T runs first and –E is only run on subtrees that have totxattr>0 which saves walking down any subtrees that don’t have xattrs on any file. (this is a huge savings)

\*\*\* and we even then list the files that have the xattrs

trees that have xattrs testidx/l1.1

testidx/l1.1/f1.1.doc

testidx/l1.1/f2.1.doc

We also don’t want to forget querying on extended attributes.

**./gufi\_query -n1 -d'|' -x -E "select path()||'/'||name,xattr\_name,xattr\_value from xpentries;" testidx**

\*\*\* notice –x to turn on xattr for query and xpentries which joins the xattr values

testidx/l1.2/f2.1.doc||

testidx/l1.3/f3.1.doc||

testidx/l1.1/f1.1.doc||

testidx/l1.1/f2.1.doc|user.garyxattr|garyxattrval

…

testidx/l1.1/l2.1/f1.tar||

**./gufi\_query -n1 -d'|' -x -E "select path()||'/'||name,xattr\_name,xattr\_value from xpentries where xattr\_name like '%gary%’;” testidx**

\*\*\* and of course you can search the xattr name and value

testidx/l1.1/f2.1.doc|user.garyxattr|garyxattrval

Group by is also a very useful SQL concept.

**./gufi\_query -n1 -d'|' -I "create table outtable(opath text, oname text, osize int64);" -K "create temp table aggtable(apath text, aname text, asize int64);" -E "insert into outtable select path(),name,size from vrpentries;" -J "insert into aggtable select \* from outtable;" -G "select count(),asize from aggtable group by asize;" testidx**

\*\*\* group by is powerful, notice group by count of files and size (so 9 files are zero bytes)

9|0

4|2

1|4

1|7

1|10

The powerful ability to run OS commands is important as well.

**./gufi\_query -n1 -d'|' -x -E "select path()||'/'||name,intop('cat testsrc'||substr(path(),instr(path(),'/'))||'/'||name||' | wc -c') from vrpentries where size>0;"**

\*\*\* running cat file | wc –c (word count) return int

testidx

testidx/l1.2/f2.1.doc|2

testidx/l1.3/f3.1.doc|2

testidx/l1.1/f1.1.doc|2

testidx/l1.1/f2.1.doc|2

testidx/l1.3/l2.2/f1.bigger|7

testidx/l1.3/l2.3/f1.big|4

testidx/l1.3/l2.1/f1.biggest|10

**./gufi\_query -n1 -d'|' -x -E "select strop('wc -c testsrc'||substr(path(),instr(path(),'/'))||'/'||name) from vrpentries where size>0;" testidx**

\*\*\* return string (this is any command the user can run

       2 testsrc/l1.2/f2.1.doc

       2 testsrc/l1.3/f3.1.doc

       2 testsrc/l1.1/f1.1.doc

       2 testsrc/l1.1/f2.1.doc

       7 testsrc/l1.3/l2.2/f1.bigger

       4 testsrc/l1.3/l2.3/f1.big

      10 testsrc/l1.3/l2.1/f1.biggest

/gufi\_query -n1 -d'|' -E "select blobop('ls -ld '||path()) from entries;" testidx

drwxr-xr-x  6 ggrider  staff  192 Mar  6 20:06 testidx/l1.2

drwxr-xr-x  6 ggrider  staff  192 Mar  6 20:06 testidx/l1.3

drwxr-xr-x  6 ggrider  staff  192 Mar  6 20:06 testidx/l1.1

drwxr-xr-x  6 ggrider  staff  192 Mar  6 20:06 testidx/l1.1

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.2/l2.2

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.2/l2.3

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.2/l2.1

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.3/l2.2

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.3/l2.3

drwxr-xr-x  4 ggrider  staff  128 Mar 11 12:38 testidx/l1.3/l2.1

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.1/l2.2

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.1/l2.2

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.1/l2.3

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.1/l2.3

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.1/l2.1

drwxr-xr-x  3 ggrider  staff  96 Mar  6 20:06 testidx/l1.1/l2.1

## Nearest Neighbor style (highly optimized) query for vector similarity/AI

An example of using text extracted from files to have a full text search and a AI vector nearest neighbor search.

What does this little tree look like (it’s a some powerpoint slide that have had some text extracted)

find testv -type f

testv/lanl-sparse-access/db.db

testv/quadrics/db.db

testv/seagate/abof/ebof-abof compstg/db.db

testv/seagate/abof/db.db

testv/seagate/db.db

testv/NGP-Mem-Accel/Fugaku-next/db.db

testv/NGP-Mem-Accel/db.db

testv/NGP-Mem-Accel/sparse papers/db.db

testv/Inman-tech/db.db

testv/nvidia/db.db

testv/db.db

The db.db has these current tables

all the normal gufi tables/views (entries family, summary family, treesummary family, rollup family, etc.

We could add metadata with separate external db’s but since these are all my files on my mac I just put the tables into db.db

Extra metadata is in a virtual table fts5 (full text search), that table has several fields

There is a record per page of the file.

winode == the inode for the file the text was extracted from

numw = number of words

wordf = the full text field with all the words

CREATE VIRTUAL TABLE words using fts5 (WINODE,TYPE,NWM,NUMW,WORDF)

/\* words(WINODE,TYPE,NWM,NUMW,WORDF) \*/;

I decided to strip out the first part of the text and put it in a new table in a text field (I could have done this directly from the fts table I think, but I wanted it to be easy to think about)

This is where is drop and make a little table with tinode and tblob where the tinode==the inode for the file the text came from and the tblob is just a test field

#drop text table for one per file to the gufi db.db files

find testv -type f -exec ./gufi\_sqlite3 {} 'drop table gtext;' \;

#add a text table for one per file to the gufi db.db files

find testv -type f -exec ./gufi\_sqlite3 {} 'create table gtext (tinode int64, tblob text);' \;

Now I populate that table by querying the fts virtual table and pull all the text from all the pages and concats it into a single field so one row per file. I could have done a text per page if I wanted

# populate the text table

./gufi\_query -n1 -d'|' -w -S "insert into gtext (tinode, tblob) select winode, group\_concat(wordf) from words group by winode;" testv

Now I need a vector virtual table so I can extract the vectors from the text on a per file/inode basis. I put in vinode==inode for the file the vectors are pulled from the text from. the model I will use is 384 dimensions.

#add a vec table to the gufi db.db files

find testv -type f -exec ./gufi\_sqlite3 {} 'create virtual table gvec using vec0(vinode int64,fp384 float[384]);' \;

Now I download a gguf model from hugging face and give it a shorter name

#download the gguf model from huggingface

curl -L -o all-MiniLM-L6-v2.e4ce9877.q8\_0.gguf https://huggingface.co/asg017/sqlite-lembed-model-examples/resolve/main/all-MiniLM-L6-v2/all-MiniLM-L6-v2.e4ce9877.q8\_0.gguf

mv all-MiniLM-L6-v2.e4ce9877.q8\_0.gguf minilm384.gguf

Lets see if sqlite can use that model by just running gufi\_query and just have it make a vector out of a textd blurb.

#test it with just a string

./gufi\_query -n1 -d'|' -w -S "INSERT INTO temp.lembed\_models(name, model)

select 'minilm384', lembed\_model\_from\_file('minilm384.gguf');select lembed('minilm384','The United States Postal Service is an independent agency) from summary;" testv

I need to load vectors from the text. You have to load the model first that is the first sql statement and then I just inserted the lembed processed vectors into the vec table. The -w is needed to open the db in write mode. Notice I needed to truncate the tblob as this model is really a sentence oriented model.

./gufi\_query -n1 -d'|' -w

-S "INSERT INTO temp.lembed\_models(name, model) select 'minilm384', lembed\_model\_from\_file ('minilm384.gguf');

insert into gvec (vinode, fp384) select tinode, lembed('minilm384',substr(tblob,1,256) from gtext;" testv

Lets see if it really loaded up any records into the vec table

./gufi\_query -n1 -d'|' -w -S "select count(\*) from gvec;" testv

2

…

5

And lets see if it has the inodes into the vinodes

./gufi\_query -n1 -d'|' -w -S "select vinode from gvec;" testv

64322544

…

3820591

Now lets get real and do a super fancy nearest neighbor. It does aggregation, trims the aggregate per thread, sets a variable that can limit the queries as the nearest neighbor distance gets better and better as the traversal runs.

./gufi\_query -n1 -d'|' -a -I "create table vtvec(vtinode int64,vtpath text, vtname text, vtsize int64,vtdist real,vttext text);" -K "create table vavec(vainode int64,vapath text, vaname text, vasaze int64,vadist real,vatext text);" -S "INSERT INTO temp.lembed\_models(name, model) select 'minilm384', lembed\_model\_from\_file('minilm384.gguf');select 'GUFIGREP',setstr('mxd',case when count(vtinode)>=3 then max(vtdist) else 1000000 end) from vtvec;" -E "with myvecq as (select vinode,distance as vdistance from gvec where fp384 match lembed('minilm384','deltafs') order by vdistance asc limit 3) insert into vtvec select inode,path(),name,size,vdistance,replace(tblob,CHAR(10),' ') from myvecq inner join vrpentries on inode=cast(vinode as int64) inner join gtext on inode=tinode where vdistance<={mxd} order by vdistance asc limit 3; delete from vtvec where vtinode not in (select vtinode from vtvec order by vtdist asc limit 3);" -J "insert into vavec select \* from vtvec order by vtdist asc limit 3;" -G "select \* from vavec order by vadist asc limit 3;" testv | grep -v GUFIGREP

29264497|testv/NGP-Mem-Accel/sparse papers|Load-balanced\_Gather-scatter\_Patterns\_for\_Sparse\_D.pdf|1084788|1.0793194770813|90% … mmm 0.82

3820577|testv/seagate/abof/ebof-abof compstg|flash-ebof-offloads-excerpts-v1.pptx|710564|1.08760523796082|1/0 Time (sec) … 00 +- @ DeltaFS IMD “a VPIC Baseline

3820576|testv/seagate/abof/ebof-abof compstg|flash-ebof-offloads-excerpts.pptx|566342|1.08760523796082|1/0 Time (sec) ...@ DeltaFS IMD “a VPIC Baseline

We wanted to have a super low memory needed, super efficient nearest neighbor query capability and this is about as super efficient as one can imagine.

Wow, lets explain that

./gufi\_query -n1 -d'|'

-a \*\*\*\*we need to use -a to force -S and -E to both run in that order

-I "create table vtvec(vtinode int64,vtpath text, vtname text, vtsize int64,vtdist real,vttext text);" \*\*\*\*this is a per thread aggregate

-K "create table vavec(vainode int64,vapath text, vaname text, vasaze int64,vadist real,vatext text);" \*\*\* this is an all threads total aggregate it runs once at the start

-S "INSERT INTO temp.lembed\_models(name, model) select 'minilm384', lembed\_model\_from\_file('minilm384.gguf'); \*\*\* you need to load the model

select 'GUFIGREP',setstr('mxd',case when count(vtinode)>=3 then max(vtdist) else 1000000 end) from vtvec;" \*\*\* this uses setstr to record the max distance per thread. The GUFIGREP is just a cheap trick to be able to remove output from this select – there is probably a better way. Mxd is the variable that gets loaded each time and it’s the max distance of the best matches. It sets mxd to high until there are at least 3 records (we are querying on best 3) then it sets it to the max distance of the best 3)

-E "with myvecq as (select vinode,distance as vdistance from gvec where fp384 match lembed('minilm384','deltafs') order by vdistance asc limit 3) insert into vtvec select inode,path(),name,size,vdistance,replace(tblob,CHAR(10),' ') from myvecq inner join vrpentries on inode=cast(vinode as int64) inner join gtext on inode=tinode where vdistance<={mxd} order by vdistance asc limit 3;

\*\*\* this queries the vec table, joins with the vrpentries to pick up path,name,size, and joins with the gtext table (the raw text) all in inode.

\*\*\* it orders by distance and keeps the best 3 and it limits the records from the vec table using the mxd variable set above.

delete from vtvec where vtinode not in (select vtinode from vtvec order by vtdist asc limit 3);"

\*\*\* this trims the per thread aggregate table to the best 3 as we travers for each db.

-J "insert into vavec select \* from vtvec order by vtdist asc limit 3;"

\*\*\* this runs last and every thread takes its per thread and loads it into the global (all threads) aggregate table.

-G "select \* from vavec order by vadist asc limit 3;"

\*\*\* this is the final query on the all threads global aggregate

testv | grep -v GUFIGREP

\*\*\* this is the index tree top and of course we just grep put the GUFIGREP (it’s a kludge but we can fix it pretty trivially if we need to)

Now lets hook this up to a chatbot

Lets use llamafile

Lets use gufi looking at your holdings as the rag db

This script runs a nearest neighbor gufi search combined with a text like query (could/should have been an fts query for more powerful punch). We are also not using chunking just look at the first 256 bytes of text in the files for the vector nearest neighbor because the model doesn’t do long text)

#!/opt/homebrew/bin/python3

import subprocess

import os

import time

import sys

import string

argc=len(sys.argv)

#print('number of args %d' % (argc))

if argc<4:

print('run-vecnn-conv.py "vecphrase" "likephrase" "englishquestion"')

sys.exit()

# do the fancy nearest neighbor query and fill in the nearest neighbor text for RAG

myq=('./gufi\_query -n1 -d\'|\' -a -I "create table vtvec(vtinode int64,vtpath text, vtname text, vtsize int64,vtdist real,vttext text);" -K "create table vavec(vainode int64,vapath text, vaname text, vasaze int64,vadist real,vatext text);" -S "INSERT INTO temp.lembed\_models(name, model) select \'minilm384\', lembed\_model\_from\_file(\'minilm384.gguf\');select \'GUFIGREP\',setstr(\'mxd\',case when count(vtinode)>=3 then max(vtdist) else 1000000 end) from vtvec;" -E "with myvecq as (select vinode,distance as vdistance from gvec where fp384 match lembed(\'minilm384\',\'%s\') order by vdistance asc limit 3) insert into vtvec select inode,path(),name,size,vdistance,replace(tblob,CHAR(10),\' \') from myvecq inner join vrpentries on inode=cast(vinode as int64) inner join gtext on inode=tinode where vdistance<={mxd} and tblob like \'%s\' order by vdistance asc limit 3; delete from vtvec where vtinode not in (select vtinode from vtvec order by vtdist asc limit 3);" -J "insert into vavec select \* from vtvec order by vtdist asc limit 3;" -G "select quote(vapath||\'/\'||vaname),quote(vatext),quote(vadist) from vavec order by vadist asc limit 3;" testcr | grep -v GUFIGREP' % (sys.argv[1],sys.argv[2]))

#print (myq)

top\_contexts = []

proc = subprocess.Popen(myq,stdout=subprocess.PIPE,shell=True,text=True)

for line in proc.stdout:

#print(line)

ct=line.split('|')[1]

#print(ct)

top\_contexts.append(ct)

#this is the nearest neighbor text for RAG

context = " ".join(top\_contexts)

#use Context Information: in the conversation with the question using context top nearest neighbor query to get RAG and pass in the question

mll=('./llamafile-0.9.3 --silent-prompt --nologo -m /opt/homebrew/models/distilabelorca-tinyllama-1.1b.Q8\_0.gguf --cli -c 2048 --temp 0.0 -p "Based on the given material, generate an appropriate response to the user\'s question using Context Information: %s, Query:%s?"' % (context,sys.argv[3]))

ml=('./llamafile-0.9.3 --nologo -m /opt/homebrew/models/distilabelorca-tinyllama-1.1b.Q8\_0.gguf --cli -c 2048 --temp 0.0 -p "%s?"' % (sys.argv[3]))

#print (mll)

# now hopefully show off that without RAG its not too well informed but with RAG its well informed

print ("NO CONTEXT")

os.system(ml)

print ("\n")

print ("WITH GUFI RAG CONTEXT")

os.system(mll)

What was the result using slingshot as the vector nearest neighbor and like %slingshot% to match text (could be fts would be better) and what is slingshot as the question we are going to ask the bot.

This is what was passed to the bot

Based on the given material, generate an appropriate response to the user's question using Context Information: '10.5 Olympus Cabs (400kva), 4 CDUs 2,684 Dual Socket Nodes Intel SPR CPUs (no HBM) 256 GiB DDR/Node 639 TiB DDR Total 4 VAN Intel SPR CPUs (no HBM) 512 GiB DDR/Node, 1.92 TB SSD 51.6% Global BW 3 River Racks Cray Shasta Slingshot Fabric 640 GB/sec ,Cray Shasta 2 Olympus Cab (400kva), 1 CDU 2 River Racks Gateway (8) / DVS (8) Slingshot Fabric Lustre Storage ClusterStor E1000 ,Cray Shasta 9 River Racks 24 Olympus Cabs (400kva), 8 CDUs Gateway (5) / DVS (8) ee Slingshot Fabric ClusterStor E1000 56% Global BW 36 FSUs : 9.2 PB Usable 1.29 TB/s BW ,instruction [count \_ Percentage , Query:what is slingshot?]

This is running the script

./run-vecnn-conv.py "slingshot" "%slingshot%" "what is slingshot" 2>/dev/null

NO CONTEXT

what is slingshot?

WITH GUFI RAG CONTEXT

] Slingshot is a high-performance computing (HPC) fabric that provides a scalable, high-bandwidth, and low-latency interconnect for HPC systems. It is based on the Cray Shasta architecture and is optimized for high-performance computing applications. Slingshot provides a high-bandwidth, low-latency interconnect for HPC systems. It is based on the Cray Shasta architecture and is optimized for high-performance computing applications.

## Example of using full text search

You can use the virtual table type fts5 and make a new table or new external database with an fts5 virtual table in it that you store text you want to use full text search on. In this example the full text search table is called words and the gufi table is of course entries (or really should be vrpentries). The two tables are joined on inode in entries = tinode in words. You can see using the MATCH verb in the fts table query. You have all the power of GUFI tables/joins/external interfaces plus full text search mixed in.

A screenshot of a computer

AI-generated content may be incorrect.

## Useful examples of queries for storage administrators

**./gufi\_query -n1 -d'|' -z0 -T "select path(),totltk, totmtk-totmtm-totmtg-totmtt, totmtm-totmtg-totmtt,totmtg-totmtt, totmtt from treesummary;" testidx**

\*\*\* make a simple histogram 0-1k, 1k-1m, 1m-1g, 1g-1t, >1t. - all this is already in tree summary

testidx|16|0|0|0|0

**./gufi\_query -n1 -d'|' -z1 -T "select path(),totltk, totmtk-totmtm-totmtg-totmtt, totmtm-totmtg-totmtt,totmtg-totmtt, totmtt from treesummary;" testidx**

\*\*\* same as above but down a level

testidx|16|0|0|0|0

testidx/l1.2|4|0|0|0|0

testidx/l1.3|4|0|0|0|0

testidx/l1.1|8|0|0|0|0

**./gufi\_query -n1 -d'|' -I "create table outtable(opath text,omaxsize int64);" -K "create table agtable(apath text,amaxsize int64);" -S "insert into outtable select path(),totsize from summary;" -J "insert into agtable select \* from outtable;" -G "select apath,amaxsize from agtable order by amaxsize desc limit 4;" testidx**

\*\*\* find the top directory with largest file, can use same efficiency techniques as the nearest neighbor examples

testidx/l1.3/l2.1|10

testidx/l1.3/l2.2|7

testidx/l1.1|4

testidx/l1.3/l2.3|4

**./gufi\_query -n1 -d'|' -I "create table outtable(opath text,oname text,osize int64);" -K "create table agtable(apath text,aname text,asize int64);" -S "insert into outtable select path(),name,size from vrpentries;" -J "insert into agtable select \* from outtable;" -G "select apath||'/'||aname,asize from agtable order by asize desc limit 4;" testidx**

\*\*\* find the largest file, can use same efficiency techniques as the nearest neighbor examples

testidx/l1.3/l2.1/f1.biggest|10

testidx/l1.3/l2.2/f1.bigger|7

testidx/l1.3/l2.3/f1.big|4

testidx/l1.2/f2.1.doc|2

**./gufi\_query -n1 -d'|' -I "create table outtable(opath text,oname text,osize int64);" -K "create table agtable(apath text,aname text,asize int64,o12 int64,o34 int64,o57 int64,o810 int64);" -S "insert into outtable select path(),name,size from vrpentries;" -J "insert into agtable select opath,oname,osize,case when osize between 0 and 2 then 1 else 0 end, case when osize between 3 and 4 then 1 else 0 end, case when osize between 5 and 7 then 1 else 0 end, case when osize between 8 and 10 then 1 else 0 end from outtable;" -G "select sum(o12),sum(o34),sum(o57),sum(o810) from agtable;" testidx**

\*\*\* make a histogram of file sizes, can use same efficiency techniques as in nearest neighbor examples

13|1|1|1

**./gufi\_query -n1 -d'|' -I "create table outtable(opath text,oname text,omtime int64);" -K "create table agtable(apath text,aname text,amtime int64);" -S "insert into outtable select path(),name,mtime from vrpentries;" -J "insert into agtable select \* from outtable;" -G "select apath||'/'||aname,(unixepoch()-amtime)/60/60/24 from agtable order by unixepoch()-amtime desc limit 4;" testidx**

\*\*\* calculating age of file mtime from today in days – notice unixepoch() – all gufi times are unix epochs

testidx/l1.1/l2.1/f1.exe|733

testidx/l1.1/l2.2/f1.exe|733

testidx/l1.1/l2.3/f1.exe|733

testidx/l1.1/l2.3/f1.tar|733

**./gufi\_query -n1 -d'|' -I "create table outtable(opath text,oname text,osize, int64omtime int64);" -K "create table agtable(apath text,aname text,asize int64,amtime int64);" -S "insert into outtable select path(),name,size,mtime from vrpentries;" -J "insert into agtable select \* from outtable;" -G "select apath||'/'||aname,asize,(unixepoch()-amtime)/60/60/24 from agtable order by asize desc,unixepoch()-amtime desc limit 4;" testidx**

\*\*\* find the biggest and oldest files

testidx/l1.3/l2.1/f1.biggest|10|733

testidx/l1.3/l2.2/f1.bigger|7|733

testidx/l1.3/l2.3/f1.big|4|733

testidx/l1.1/f1.1.doc|2|733

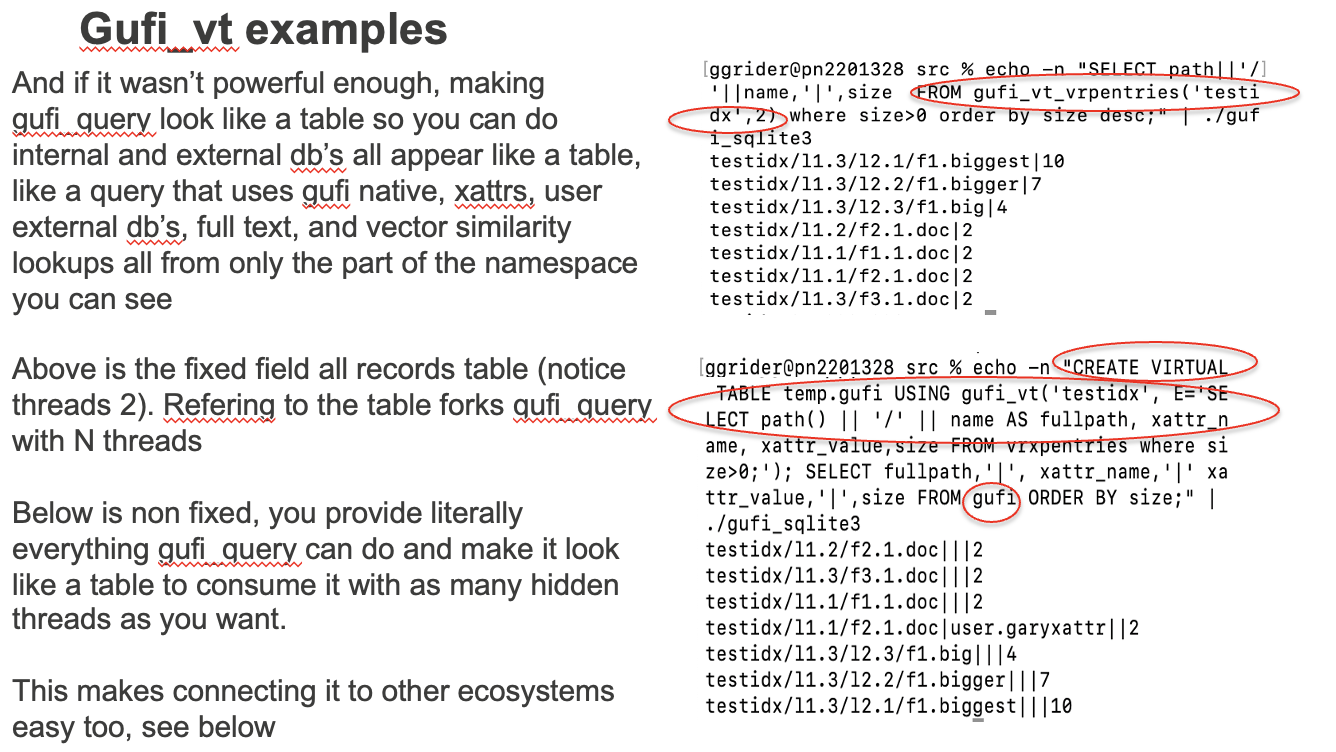
# GUFI virtual tables and connectivity into Python, R, Ruby, analytics, and AI ecosystems

To provide simple connectivity to Python, R, the Apache Analytics ecosystem and the emerging AI Model Context Protocol ecosystem, simple to complex GUFI queries needed to appear as a single sqlite table, which already has connectivity for these languages/ecosystems.

Virtual tables in sqlite give you the ability to make things look like sqlite tables including both sqlite and non sqlite information.

* **The provided Gufi\_vt family virtual table makes gufi\_query look like an sqlite table**
* **You can also use gufi\_sqlite3 (an interactive sqlite3 interface that has all the gufi extensions) to provide a virtual table from GUFI)**
* **Gufi\_vt\_vrpentries (automatically generated fixed all rows all cols+path)**
* **Gufi\_vt\_vrsummary (automatically generated fixed all rows all cols+path)**
* **Gufi\_vt\_treesummary (automatically generated fixed all rows all cols+path)**
* **Gufi\_vt is a way for you to generate a completely custom virtual table query using all the power of gufi\_query including external databases, running OS commands, etc.**
* **Because the results from the gufi\_query in gufi\_vt\* tables returns back to your query you can get aggregation (sort/group..) inside your query.**

The below example shows a fixed and custom table created using gufi\_sqlite3



Because these GUFI virtual tables can appear as an sqlite3 table, its simple to connect it to Python/R and other tools. This below is one example of making a GUFI query into an sqlite3 table and connect it to Sqlalchemy which is a standard package you can use from Python and other languages to talk to various SQL databases.



Some examples of how to pull gufi\_vt for a gufi virtual table or query generated virtual table

This one uses the power of gufu\_vt to run a gufi query and create a virtual table from that query dynamically

#!/usr/bin/env python3

import sqlalchemy

EXT = 'src/gufi\_vt'

INDEX = 'index'

# enable extension loading and load gufi\_vt

# https://stackoverflow.com/a/48869774/341683

def load\_extension(conn, \_):

conn.enable\_load\_extension(True)

conn.load\_extension(EXT)

conn.enable\_load\_extension(False)

def main():

# open an in-memory SQLite db through SQLAlchemy

engine = sqlalchemy.create\_engine('sqlite://', echo=False)

# call load\_extension when a connection is made with this engine

sqlalchemy.event.listen(engine, 'connect', load\_extension)

conn = engine.connect()

conn.execute(sqlalchemy.text('''

CREATE VIRTUAL TABLE temp.gufi\_results

USING gufi\_vt(

index='{0}',

I='CREATE TABLE intermediate(name TEXT, size INT64, filecount INT64);',

S='INSERT INTO intermediate SELECT rpath(sname, sroll), totsize, totfiles FROM vrsummary;',

K='CREATE TABLE aggregate(name TEXT, avgsize INT64);',

J='INSERT INTO aggregate SELECT name, size / filecount FROM intermediate',

G='SELECT name, avgsize FROM aggregate'

);'''.format(INDEX)))

results = conn.execute(sqlalchemy.text('SELECT \* FROM gufi\_results;'))

for row in results:

for col in row:

print(col, end=' ')

print()

if \_\_name\_\_ == '\_\_main\_\_':

main()

This one just used the built in gufi\_vt\_penties virtual table

#!/usr/bin/env python3

import sqlalchemy

EXT = 'src/gufi\_vt'

INDEX = 'index'

# enable extension loading and load gufi\_vt

# https://stackoverflow.com/a/48869774/341683

def load\_extension(conn, \_):

conn.enable\_load\_extension(True)

conn.load\_extension(EXT)

conn.enable\_load\_extension(False)

def main(query):

# open an in-memory SQLite db through SQLAlchemy

engine = sqlalchemy.create\_engine('sqlite://', echo=False)

# call load\_extension when a connection is made with this engine

sqlalchemy.event.listen(engine, 'connect', load\_extension)

conn = engine.connect()

results = conn.execute(sqlalchemy.text(query))

for row in results:

for col in row:

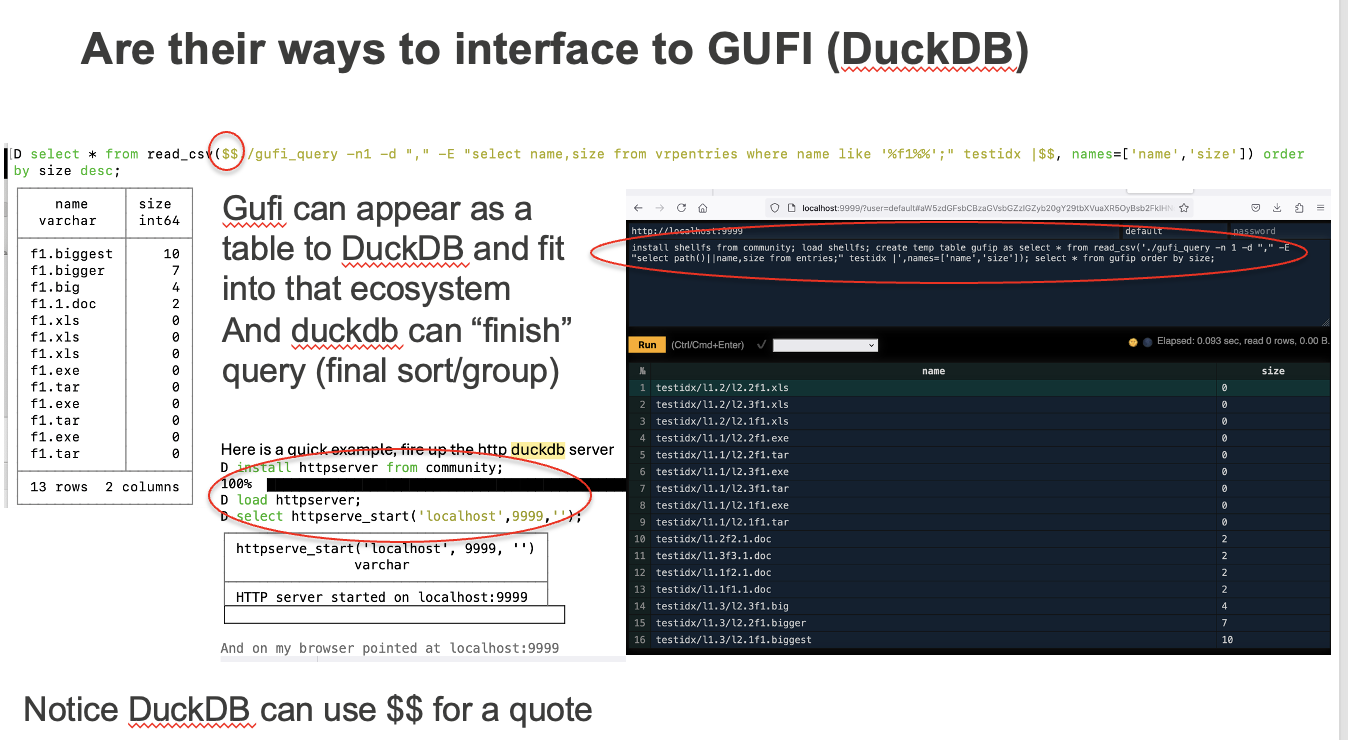
print(col)

if \_\_name\_\_ == '\_\_main\_\_':

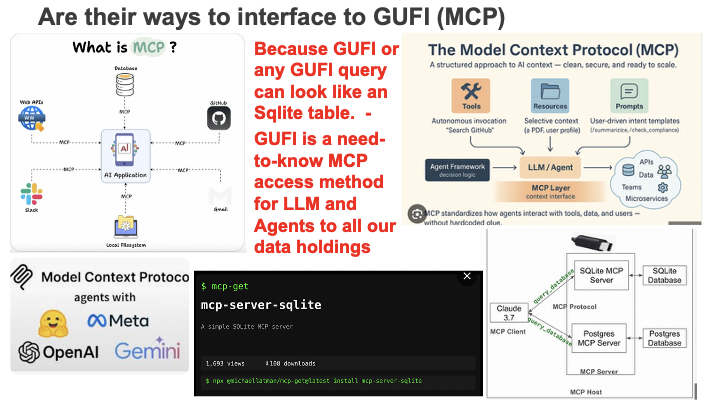
main('SELECT path || \'/\' || name FROM gufi\_vt\_pentries(\'{0}\');'.format(INDEX))

**An easy way to fit into the Apache Analytics ecosystem is to cooperate with DuckDB**

Another way to fit into the Apache Analytics ecosystem to interface to lots of open source software is to enable DuckDB, the most popular analytics columnar database in open source, to be able to see gufi queries as DuckDB a DuckDB table. Below is a demo of that solution.



Given this ability to connect GUFI by a virtual table making GUFI look like an SQLite3 table, Model Context Protocol for AI agents should also just work just like the above two examples.



## GUFI used in R

#!/usr/bin/env Rscript

#

# need to disable median function in addqueryfuncs (src/addqueryfuncs.c)

library(RSQLite)

conn <- dbConnect(RSQLite::SQLite(), ":memory:", loadable.extensions = TRUE)

dbExecute(conn, "SELECT load\_extension('src/gufi\_vt');")

dbGetQuery(conn, "SELECT name, size FROM gufi\_vt\_pentries('index');")

dbDisconnect(conn)

## GUFI used in Ruby

#!/usr/bin/env ruby

**require** 'sqlite3'

db = SQLite3::Database.new(':memory:')

db.enable\_load\_extension(**true**)

db.load\_extension("src/gufi\_vt")

db.execute("SELECT name, size FROM gufi\_vt\_pentries('index');") **do** | row |

row.each { | col |

print(col, " ")

}

print("\n")

**end**

db.close()

## GUFI distributed processing

In order for GUFI to be used in a multi-user setting, processes that run queries via gufi\_query or via GUFI virtual tables must run as processes with the user’s UID set to match how the index directory tree was created. There are times where the user cant be “logged in” to where the GUFI index tree is mounted. You can of course distributed the index tree via NFS or other similar mechanism, while mounting it read only would be recommended so users don’t corrupt it accidentally, but its possible to do. Performance may suffer on queries though in this case because you are doing what could be somewhat of an IOPS workload across a network. It can be far more performant/efficient to ship the query to where the GUFI index tree is locally mounted and sometimes for security reasons you cant allow users to be “logged in” to where the query is initiated from. An example of this is implementing web and restful services. Those services should typically be run not as root so that they can setuid() to the user when acting on the users behalf. Monolithic databases don’t have this problem because they run as network services and you authenticate to those services via network operations using passwords or credentials etc. There is obviously a need for GUFI to have a similar capability, be able to be accessed remotely.

This function is provided through the GUFI virtual table function. Virtual tables allows. You to turn an query into virtual table so you can access the data from that query as just an sqlite table. This enables the python and other ecosystems to just treat GUFI as just an sqlite table and you can use ORM/Data Frames functions just as you would if GUFI was just a single sqlite database. With the remote\_cmd and remote\_arg keywords in the gufi\_vt and gufi\_vt\_\* virtual tables. This allows you to run GUFI queries remotely. You could use something like ssh, or pick your favorite remote execution method, to run the query in a different place than you consume the data. This is extremely powerful, in that you can run a python program that uses ORM/Data Frames using your favorite python packages anywhere as long as where you are logged in can have a credential to do remote execution to another machine.

Further, because of the power of sqlite virtual tables, if you are using a framework like Apache Drill or similar that enables data joining/fusion with many data sources, because GUFI queries can look like sqlite tables (local or remote), that means GUFI information (with all its external database power/commands become fields/…) can be joined/fused with other forms of data in a completely native way for those frameworks.

See below, this executes a query remotely using gufi\_vt.

    echo ".load gufi\_vt"

    echo "CREATE VIRTUAL TABLE gufi USING gufi\_vt("prefix", E="SELECT rpath(sname, sroll, name) AS fullpath FROM vrpentries;", verbose=0, remote\_cmd="ssh", remote\_arg="remote");SELECT fullpath FROM gufi ORDER BY fullpath ASC;"

) | sqlite3

Each remote\_arg is one argv that should be passed to ssh (or whatever program is used): if you wanted to pass "-p 1234" to ssh, you would have to pass them separately with remote\_arg="-p" and remote\_arg="1234" so that weirdness in the user strings are not handled correctly.

In gufi\_vt\_\*, argument count is fixed, so all arguments passed to ssh had to be in one string ("remote -p 1234") which might be parsed incorrectly.  See below:

The above sections that describe how GUFI can fit into Python, R, Ruby, etc. show how easy it is to run GUFI queries in a familiar language/framework, and with this remote capability, that means you can run those queries from anywhere.

## GUFI parallel processing

While gufi is parallel within a single machine via extreme use of threads, there is also a need to enable parallelism across machines. As in the above example where the GUFU virtual tables can run a remote process to distributed GUFI for various reasons, if you merely replace ssh with something like pdsh and set your directory structure up to be the same at a high level across processing nodes, you can achieve parallelism.

A hypothetical example is running across 2 sites/gufi tree nodes.

At site/machine1 you have /search/site1/home and /search/site1/project. At another site you have /search/site2/home and /search/site2/archive.

Your query to produce a virtual table of that query output from both sites could be something like this:

    echo ".load gufi\_vt"

    echo "CREATE VIRTUAL TABLE gufi USING gufi\_vt("/search", E="SELECT rpath(sname, sroll, name) AS fullpath FROM vrpentries;", verbose=0, remote\_cmd="pdsh", remote\_arg="-N"), remote\_arg="-w"), remote\_arg="gary@site1gufi.site1.org,garygrider@site2gufi.site2.org");SELECT fullpath FROM gufi ORDER BY fullpath ASC;"

) | sqlite3

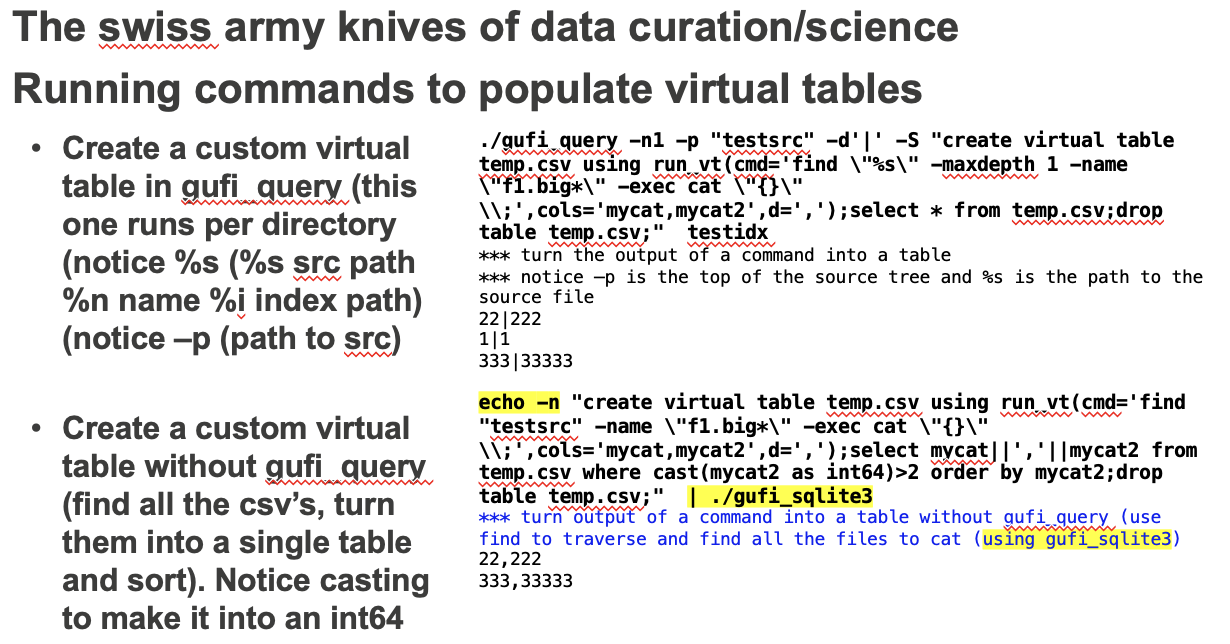
In theory you now have run a parallel gufi\_query on two different machines, even in two different sites/uid silos/etc. using your local identity/credential. Each query could be many threads (that is controllable with gufi\_vt parameters). The -N suppresses pdsh from prefixing all lines of output from which host they came from and also makes it so buffering to put all lines from a single host together. Remember, pdsh runs over ssh but there are options for the underlying distribution mechanism including even MPI if you were building a highly parallel gufi index cluster tree. Imagine how powerful of a tool this, any external database joins and all the fanciness of GUFI run in parallel across machines in parallel per machine with threads and producing a virtual table that your python program can treat as just an sqlite table and use your favorite ORM/Data Frame or data fusion packages on. Your R program could pull in data on the fly from all over the place. Your MCP program could launch a 1000 way nearest neighbor vector search.

This is an important and extreme powerful concept.

## run\_vt - Enabling any command local or remote to appear as a local sqlite table

There is a special GUFI virtual table called run\_vt that allows the output of any command, local or remote) to be mapped into an sqlite table. This of course doesn’t have to be a GUFI command it could be anything that produces. Similar to the above, this means that Python and other ecosystems including data frames and data fusion technologies can use the output of any command to become a table that can be used natively.

This is an example use:



# GUFI experimental commands (experimental)

## Source Storage System Specific Commands

Bfmi (not recently maintained)

Query Robinhood mysql db and list the tree and/or create output gufi tree

(this function has not been built or run in a while)

Usage: bfmi [options] robin\_in

options:

-h help

-H show assigned input values (debugging)

-p print file-names

-n <threads> number of threads

-d <delim> delimiter (one char) [use 'x' for 0x1E]

-x pull xattrs from source file-sys into GUFI

-P print directories as they are encountered

-b build GUFI index tree

-t <to\_dir> dir GUFI-tree should be built

-o <out\_fname> output file (one-per-thread, with thread-id

suffix)

robin\_in file containing custom RobinHood parameters

example contents:

/path - top dir-path (RH doesnt have a name

for the root)

0x200004284:0x11:0x0 - fid of the root

20004284110 - inode of root

16877 - mode of root

1500000000 - atime=mtime=ctime of

root

localhost - host of mysql

msqluser - user of mysql

mypassword - password for user of

mysql

mysqldb - name of db of mysql

future options:

-U create by user summary record

-G create by group summary record

Flow:

open Robinhood input file that has how to communicate with mysql and info about root directory

root directory is put on a queue

output file(s) are opened one per thread

mysql connections are made, on for each thread

threads are started

loop assigning work (directories) from queue to threads

each thread processes a directory by querying all records with parentid=id

for that directory

if directory put it on the queue and duplicate the directory if making a gufi

if link or file print it to screen or out file

and build an entries table with entries and keep a sum for the directory

close directory

write directory summary table

end

close output files if needed

close mysql connections

you can end up with an output file per thread

### Source Storage System Specific Commands/Scripts

(Not maintained recently)

runbfmi (not recently maintained - run bfmi to read from a robinhood mysql db and list and/or create a gufi tree - requires an input file on how to talk to mysql

runtsm (not a script, just a set of commands one can use) – commands used to build an incremental GUFI update using TSM backups to determine which files/links have changed to provide a suspect list to the incremental GUFI update process.

## Bfresultfuse

Fuse file system from GUFI query output

Recall we are working with this source tree

testdir:

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 a

-rwxr-xr-x@ 1 ggrider staff 79655 Nov 22 19:38 b

drwxr-xr-x@ 6 ggrider staff 192 Nov 30 06:53 c

-rwxr-xr-x@ 1 ggrider staff 4 Mar 20 2018 clink

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 d

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 dumbcom,ma

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 gary's dumb file

testdir/c:

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 ca

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cb

drwxr-xr-x@ 3 ggrider staff 96 Nov 30 06:53 cc

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cd

testdir/c/cc:

-rwxr-xr-x@ 1 ggrider staff 14252 Mar 20 2018 bfindex

And we are working with this GUFI tree

testdirdup/testdir:

drwxr-xr-x 4 ggrider staff 128 Dec 10 12:31 c

-rw-r--r-- 1 ggrider staff 20480 Dec 10 12:59 db.db

testdirdup/testdir/c:

drwxr-xr-x 3 ggrider staff 96 Dec 10 12:15 cc

-rw-r--r-- 1 ggrider staff 20480 Dec 10 12:31 db.db

testdirdup/testdir/c/cc:

-rw-r--r-- 1 ggrider staff 20480 Dec 10 12:15 db.db

To use the bfresultfuse daemon you first must run a query that produced a set of output tables that have the proper fields to be used by a fuse daemon.

This query was run to produce output databases:

rm outfpdb\*

../bfq -Pp -n2 -O outfpdb -E "insert into qout select fpath(),name,type,inode,nlink,size,mode,uid,gid,blksize,blocks,mtime,atime,ctime,linkname,xattrs from entries where size < 50000;" -S "insert into qout select fpath(),name,type,inode,nlink,size,mode,uid,gid,blksize,blocks,mtime,atime,ctime,linkname,xattrs from summary;" -I "create table qout (fullpath text, name text, type text, inode int(64),nlink int(64),size int(64), mode int(64),uid int(64),gid int(64), blksize int(64), blocks int(64), mtime int(64), atime int(64), ctime int(64), linkname text, xattrs text);" -a testdirdup/testdir

This query collects all required fields from both the summary record for the directory records and from the entries table which collects all the file and link information. As you can see in this case all directories are asked for and all files < 50000 bytes. The output from the queries is inserted into output databases called outfpdb.\* in a table called qout. Notice the -a flag which tells bfq to list directory records and file records (an “or” condition) which is required as we need the directory records for the fuse deaemon to work properly.

ls -l outfpdb\*

-rw-r--r-- 1 ggrider staff 8192 Dec 11 11:02 outfpdb.0

-rw-r--r-- 1 ggrider staff 8192 Dec 11 11:02 outfpdb.1

As you can see, it created two output databases, one per thread.

Now you can start the bfresult fuse daemon and point it at these databases. First we ensure the daemon isn’t already running, then we make a mountpoint and then start the bfresultfuse daemon

umount mnt

rm -rf mnt

mkdir mnt

../bfresultfuse -d -s mnt qout outfpdb 2 2>/dev/null &

[1] 63692

Try some commands against the mountpoint.

df

Filesystem 512-blocks Used Available Capacity iused ifree %iused Mounted on

/dev/disk1s1 1953800440 752122976 1193874904 39% 2191579 9223372036852584228 0% /

devfs 444 444 0 100% 768 0 100% /dev

/dev/disk1s4 1953800440 6291496 1193874904 1% 3 9223372036854775804 0% /private/var/vm

map -hosts 0 0 0 100% 0 0 100% /net

map auto\_home 0 0 0 100% 0 0 100% /home

bfresultfuse@osxfuse0 28 28 0 100% 9 0 100% /Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/mnt

ls mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/

a c clink d dumbcom,ma

ls -lR mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/

total 10

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 a

drwxr-xr-x@ 6 ggrider staff 192 Nov 30 06:53 c

-rwxr-xr-x@ 1 ggrider staff 4 Mar 20 2018 clink

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 d

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 dumbcom,ma

mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir//c:

total 7

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 ca

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cb

drwxr-xr-x@ 3 ggrider staff 96 Nov 30 06:53 cc

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cd

mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir//c/cc:

total 30

-rwxr-xr-x@ 1 ggrider staff 14252 Mar 20 2018 bfindex

stat mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/

872415311 10 drwxr-xr-x 9 ggrider staff 1 288 "Dec 11 11:02:52 2018" "Nov 30 06:53:24 2018" "Nov 30 06:53:24 2018" "Dec 31 17:00:00 1969" 4096 1 0 mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/

stat mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/a

872415311 12 -rwxr-xr-x 1 ggrider staff 1 0 "Nov 30 06:53:24 2018" "Mar 20 17:38:47 2018" "Nov 30 06:53:24 2018" "Dec 31 17:00:00 1969" 4096 0 0 mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/a

stat mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/c/shouldntwork

stat: mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/c/shouldntwork: stat: No such file or directory

xattr -l mnt/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup2/testdir/c/cb

xattrs: com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69

Now we unmount the fuse daemon

umount mnt

[1]+ Done ../bfresultfuse -d -s mnt qout outfpdb 2 2> /dev/null

As you can see, you are running ls/stat/xattr against the mnt mounted fuse file system which is the query results, and in case you looked (or didn’t) it did exclude the file larger than 50000 as instructed by the bfq command that generated the results you were examining, which was in the testdir directory

rwxr-xr-x@ 1 ggrider staff 79655 Nov 22 19:38 b

## bffuse

See the GUFI index tree as a mounted file system, dynamic queries.

Recall we are working with this source tree

testdir:

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 a

-rwxr-xr-x@ 1 ggrider staff 79655 Nov 22 19:38 b

drwxr-xr-x@ 6 ggrider staff 192 Nov 30 06:53 c

-rwxr-xr-x@ 1 ggrider staff 4 Mar 20 2018 clink

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 d

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 dumbcom,ma

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 gary's dumb file

testdir/c:

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 ca

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cb

drwxr-xr-x@ 3 ggrider staff 96 Nov 30 06:53 cc

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cd

testdir/c/cc:

-rwxr-xr-x@ 1 ggrider staff 14252 Mar 20 2018 bfindex

And we are working with this GUFI tree

testdirdup/testdir:

drwxr-xr-x 4 ggrider staff 128 Dec 10 12:31 c

-rw-r--r-- 1 ggrider staff 20480 Dec 10 12:59 db.db

testdirdup/testdir/c:

drwxr-xr-x 3 ggrider staff 96 Dec 10 12:15 cc

-rw-r--r-- 1 ggrider staff 20480 Dec 10 12:31 db.db

testdirdup/testdir/c/cc:

-rw-r--r-- 1 ggrider staff 20480 Dec 10 12:15 db.db

To use the bffuse daemon you need to point that fuse daemon at either the top or subdirectory in the GUFI tree

First lets make sure the fuse is not running.

umount mnt

rm -rf mnt

mkdir mnt

Now run the fuse daemon pointed at the GUFI tree, notice we provide the where clause size < 5000 which will only show files in this fuse mountpoint < 5000 bytes in size

../bffuse -d -s mnt testdirdup2/testdir "size < 5000" 2>/dev/null &

[1] 64230

Try some commands against the mount point.

ls -lR mnt

total 48

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 a

drwxr-xr-x@ 6 ggrider staff 192 Nov 30 06:53 c

-rwxr-xr-x@ 1 ggrider staff 4 Mar 20 2018 clink

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 d

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 dumbcom,ma

mnt/c:

total 32

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 ca

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cb

drwxr-xr-x@ 3 ggrider staff 96 Nov 30 06:53 cc

-rwxr-xr-x@ 1 ggrider staff 0 Mar 20 2018 cd

mnt/c/cc:

stat mnt/a

872415313 4 -rwxr-xr-x 1 ggrider staff 1 0 "Nov 30 06:53:24 2018" "Mar 20 17:38:47 2018" "Nov 30 06:53:24 2018" "Dec 31 17:00:00 1969" 4096 0 0 mnt/a

xattr -l mnt/c/ca

xattrs: com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69

Unmount the fuse daemon.

umount mnt

[1]+ Done ../bffuse -d -s mnt testdirdup2/testdir "size < 5000" 2> /dev/null

As you can see there are 2 missing files from the original source, the two that are not < 5000 in size which are these:

In directory testdir

-rwxr-xr-x@ 1 ggrider staff 79655 Nov 22 19:38 b

In directory testdir/c/cc

-rwxr-xr-x@ 1 ggrider staff 14252 Mar 20 2018 bfindex

# Special and Incremental loading/updating

## Background

Since a GUFI index tree is a file system tree that mimics the shape and permissions of the source storage systems, be they POSIX or POSIX like file systems, NFS/SMB file systems, Object systems that use buckets, or other, access control is controlled by some combination of permissions to traverse to the location of the information and access to the information once traversed to. This is done via POSIX, access control lists, MLS/compartmenting, etc. The base concept in GUFI is to store databases that represent the files/objects/other kinds of information in small databases on a per folder basis where a folder is a directory or a folder or a bucket or some organizing concept that has common traversal permissions over all the contents of the folder. Each item in the folder may have further access control like POSIX read permissions or access control lists, etc. Building a GUFI index tree from a source storage system(s) involves reproducing the folders/buckets as directory trees in a POSIX file system which holds the index and setting the traversal permissions the same as the source storage system being indexed.

Full/initial indexing a source storage system can be done using the gufi\_dir2index which walks the source storage system using breadth first parallelism and reproduces the source storage systems folder/bucket structure and access control and places a database in each index tree folder that represents the contents of that folder in the source storage system. gufi\_dir2trace and gufi\_trace2index can be used to extract a trace file(s) from the source storage system and then move the trace to the desired place and create the GUFI index tree from the trace(s). Since GUFI indexes are composable/decomposable, this can be done any way the user would like, typically using subtrees.

There are also custom GUFI full indexing or subtree composable full indexing for certain source storage systems that have custom/faster ways to extract the source storage system metadata. Spectrum Scale has its ILM and there has been work to provide a GUFI index loader from that mechanism. Lustre has several ways to approach full extraction of metadata and there is work going on to exploit this. There is a custom GUFI indexer for HPSS that uses DB2 queries available to HPSS licensed sites. Also, there is work that has been done on extracting metadata from Robinhood into a GUFI index tree. These custom Indexers are in various states of maturity.

## Incremental Background

As was mentioned above, there are custom ways to extract metadata from some source storage systems there is likewise custom ways to track/find/log changes to file systems, the Spectrum Scale ILM or the Lustre activity log for example. There has been work on both incremental extractors of recent changes to the source storage systems and using that information to incrementally update the GUFI index tree. Because GUFI indexes are trees and the source storage systems are trees or at least can be thought of as trees or bushes or grass, incrementally updating a GUFI index tree is in essence determining changes to the source tree and various levels of efficiency regarding ways to use that information to update the GUFI index tree.

A common approach to producing interim index artifacts to be used to update the GUFI index tree has emerged.

### Simple full ordered change log replay (not implemented but parts exist)

Probably the simplest way to incrementally update a GUFI tree is to find or be given the directories that have had a change, new file, changed file, deleted file, new parent directory (moved), new child directory, deleted child director, and deleted directory. If a source storage system could provide an ordered log of all these events with full paths it would be trivial to produce an incremental update to the GUFI index using a simple replay. That would work something like this:

* Drop rollups (utility exists)
* Drop tree summaries (utility exists)
* Remove summary information affected by directory content changes (counts/sums/etc.). (no utility exists because usually its easier to just recreate everything in folder from the source instead of incrementally updating entries records and resuming/counting fields in the summary record.
* Replay the ordered log. (no utility exists because no source file system provides such a log at least thus far)
* Recreate the summary information. (utility does not exist (see Remove summary above)
* Recreate the rollups and tree summaries (utility exists)

Of course, the above method could be parallelized using subtree decomposition but it would depend on the log as to if changes can be contained within a decomposed subtree, as file directories/moves/hard links can affect the entire file system.

It seems like it will be rare to find a large parallel file system that can easily supply a fully ordered log, so it hasn’t been a priority to enable this incremental update method.

### Partial change logs, unordered change logs, or no change logs

Since it will be rare to find a large parallel file system that implements a full in order log, since that activity could slow the parallel file system down, we have concentrated on providing incremental when we get logs, not necessarily of exactly what operations have occurred and not necessarily in any particulare order.

Different sources are capable of different logs at different costs/performance and/or different sources have different tools or costs associated with trying to find changes. Since this is the most common situation as no current source storage system produces a full ordered full path change log, we felt that finding a common approach/utilities to work in this most common situation that was desirable. Within this common approach, there are really two methodologies:

* Find the lowest directory of a subtree that has had changes made under it and just recreate the subtree indexes by recursively deleting below that point in the subtree and recreate the subtree from the source storage system and since GUFI is decomposable/composable, this is clearly possible.
* Find the directories that have had a change of any kind and use current GUFI index tree and the newly changed source tree to determine both structural changes that need to be made to the GUFI index tree as well as directories that need to be rescanned to replace the index within those directories. It is possible to reuse many of the directories, so rescanning that stat()s everything is not necessary for some parts of the tree/subtree, even in a subtree that has had changes. The above method just recreates the GUFI index subtrees without reuse of anything below the lowest point where changes have been made, and this method tries to reuse any directories that haven’t changed.

In both cases special care needs to be paid to hard links as changes to the underlying inode can imply changes to multiple subtrees. In theory, this shouldn’t matter if you are looking for changes in the source storage system because the change to an inode that appears in more than one subtree will indicate a change has happened. The caveat to that is, if a change is logged to an inode that happens to be hard linked to files in more than one subtree, if the log doesn’t tell you both subtrees changed then one subtree part of your index could be stale. For this reason, looking at link count for all changes to files may be required depending on the source storage system if the logging does not account for this situation.

In all cases you need to drop treesummaries and unroll all rollups before you start incrementally updating yesterday’s GUFI index tree with today’s new source storage system changes. There are utilities for doing that provided with GUFI.

For now we are going to ignore hard links and we will mention toward the end of this incremental update section how hard links might be handled in the case when those changes are provided in a log that may not log all the path/filenames connected to the inode that changed.

## The GUFI general incremental method

This method is the second method mentioned above that doesn’t require having an in order log. Further it doesn’t recreate any directories that aren’t required to be recreated, so it doesn’t pick the highest point in the subtree where changes have happened. This method taked in suspected changes in directories or files from either external sources like Lustre log and GPFS ILM scans. It doesnt even need the path of the change, just an inode is enough.

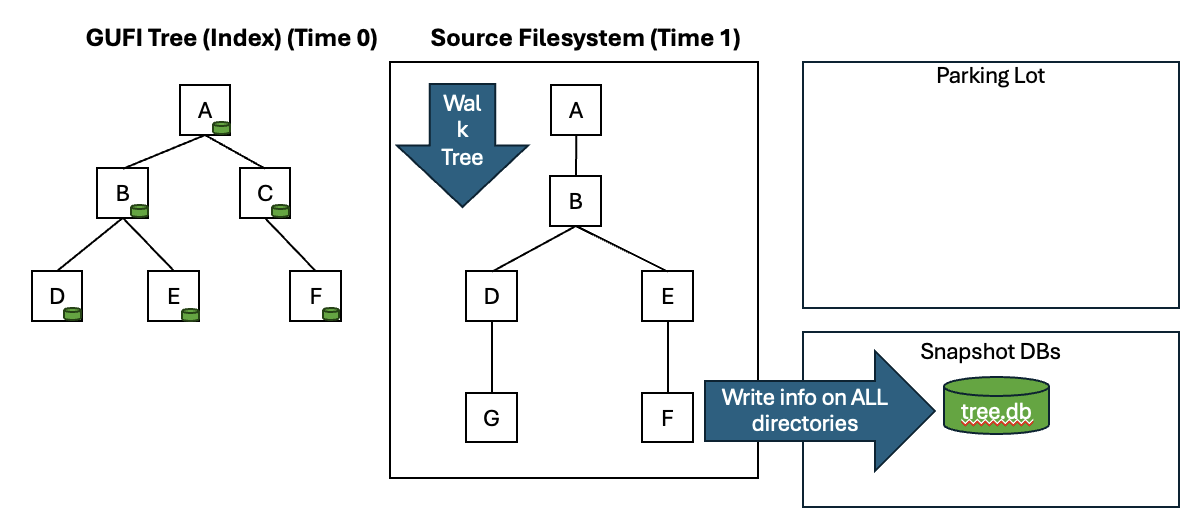
This method works like the discussion below.

An incremental scan of the todays source tree is performed. If external information is provided about what directories and/or files have had changes merely by providing a list of inodes, the incremental scan does not have to stat() everything, but if this information is not provided, then stat() of everything is required.

The output of this scan is two things:

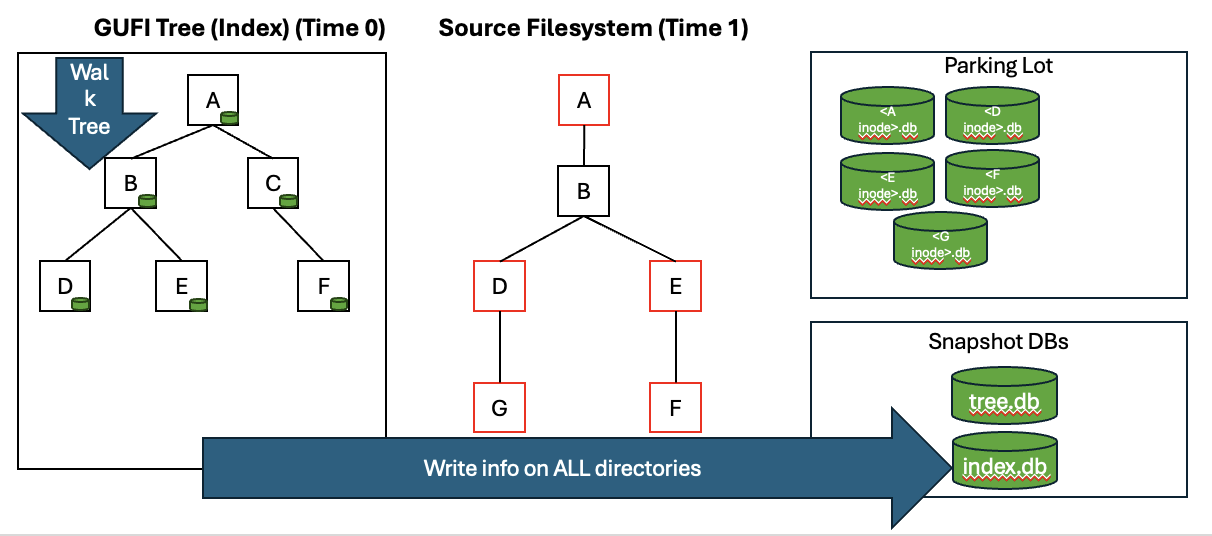
1. A parent child path/inode map for the entire todays source tree
2. A gufi database from the stat() of the entries in the directories that are found with change indicated via a date compare or are directories are indicated via the external source of inodes that have change indicated. We call those suspects.

Next, a query is done of yesterday’s GUFI index tree to build a parent/child path map from yesterday. Depicted below



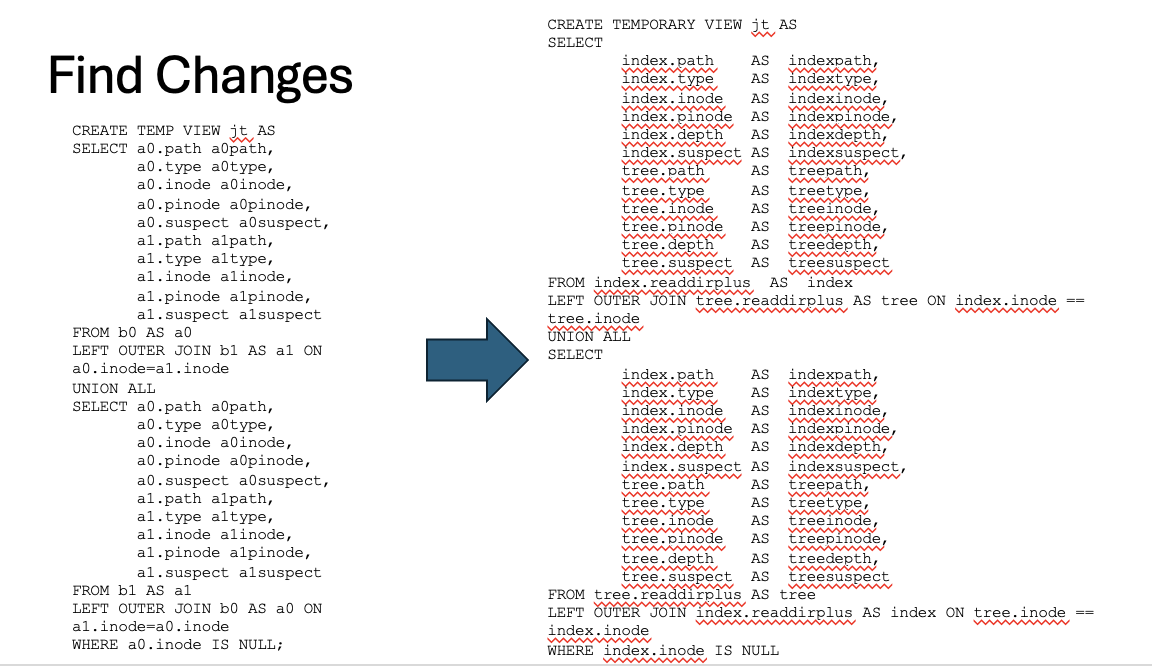
Diagram

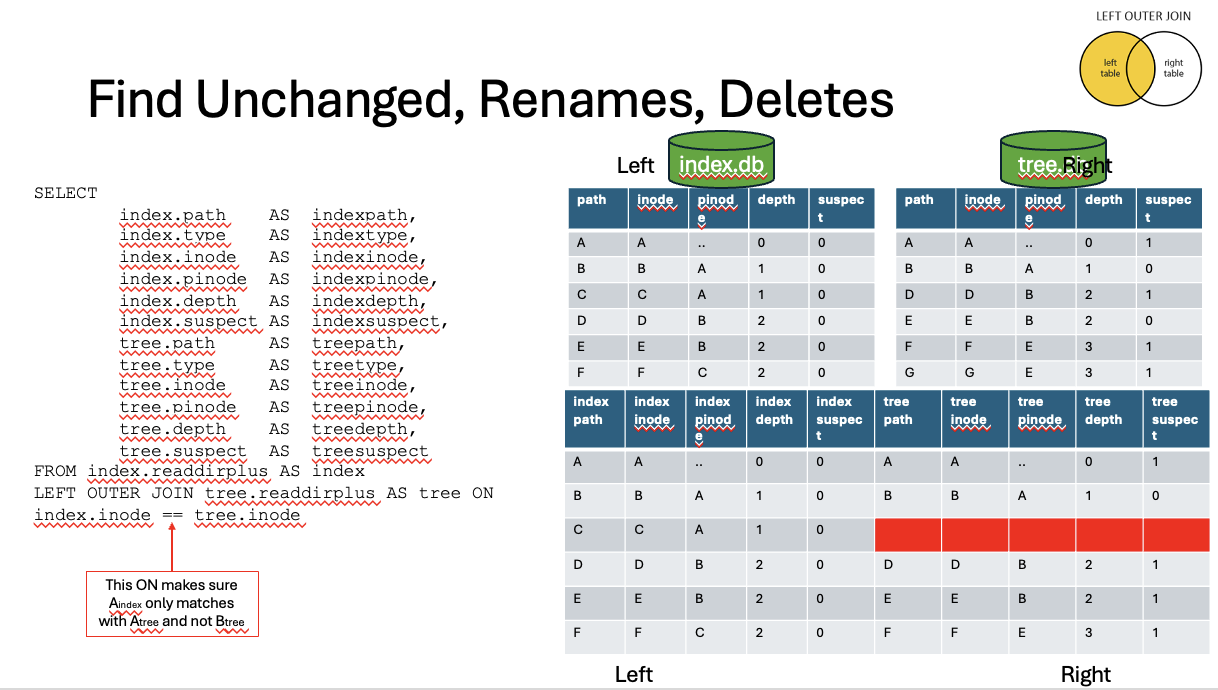
AI-generated content may be incorrect.



Table

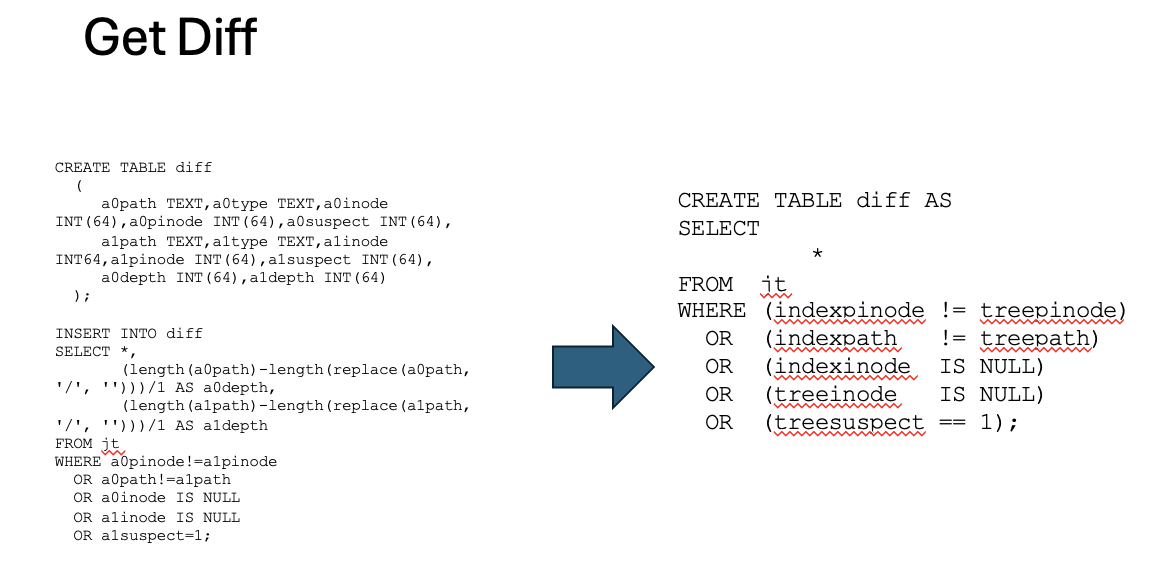
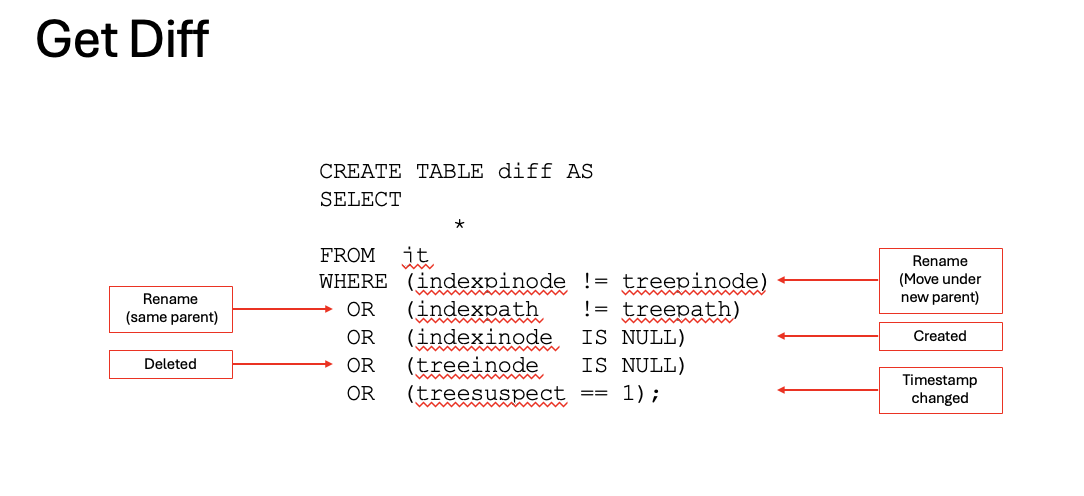
AI-generated content may be incorrect.Next the tree.db which is the today’s parent/child/path/suspect dir database is compared with the yesterday’s parent/child/path map.

To find changes, we use use left outer joins between these two tables with union all. This allows us to quickly find new directories (which appear in the right but not in the left), deleted directories which appear in the left but not in the right). We can also use the suspect marking (from external input or from walking and stat()ing everything to determine which directories changed in some way. This is depecited below:



## 

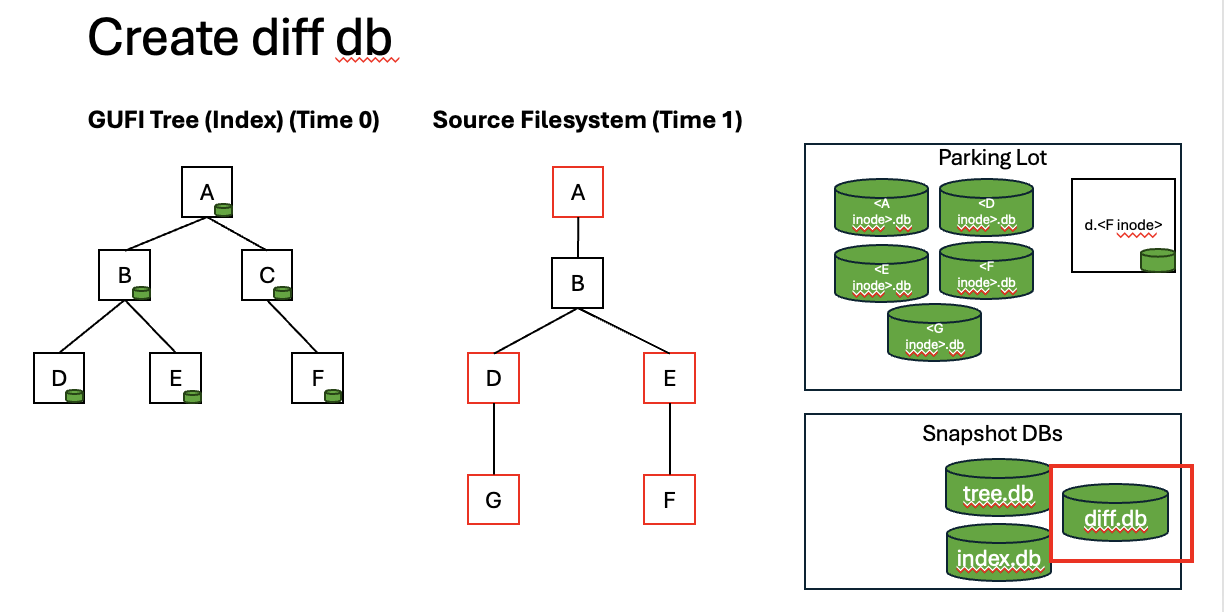
## 

We then build a Diff database with all the changes and what kind they are.

Diagram

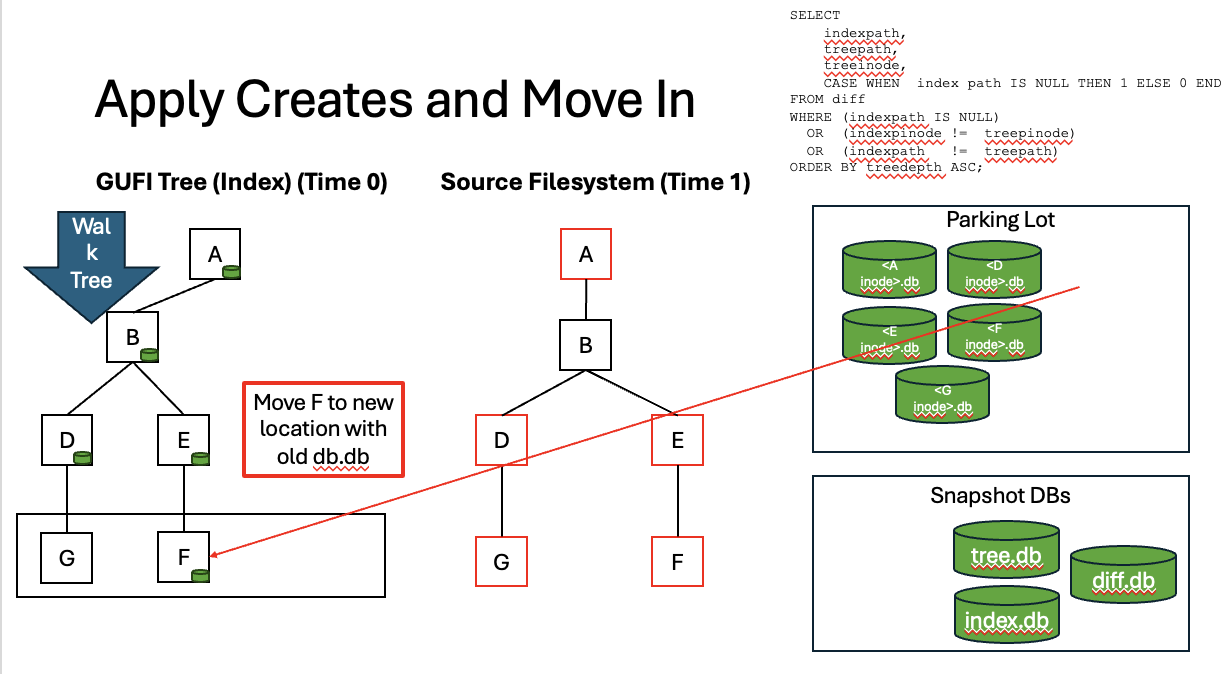
AI-generated content may be incorrect.

Diagram

AI-generated content may be incorrect.Next we remove deleted directories and move out directories that aren’t in the right places anymore or directories that will be replaced

Next we move in new directories and directories that have changed.

## 



Next we apply updated GUFI directory databases

Diagram

AI-generated content may be incorrect.

## 

Diagram

AI-generated content may be incorrect.

## 

## gufi\_incremental\_update utility

gufi\_incremental\_update is the application that performs the above description of the gufi incremental process.

gufi\_incremental\_update

-h help

-H show assigned input values (debugging)

-v version

-n -threads number of threads

-d -delim delimiter for output file

-suspect-file path to input suspect file (inode type (d,f,l) derived from change log/ILM

-suspect-method 0,1,2,3

- no suspects

-1 suspect file could contain directory,file,or link inodes

-2 suspect file could contain file or link inodes, directories have mtime

and ctime compared with -suspect-time

-3 all directories, files, and links have their mtime and ctime

checked with -suspect-time

-suspect-time time in seconds since epoch for suspect comparison

(ignored if gufi index tree is stored in source tree)

-x -xattrs process xattrs when creating gufi directory db’s

-e compress in memory data

GUFI\_tree GUFI index tree path

tree Source data path

snapshotdb database file containing records of all directories

parking\_lot directory to place update db.dbs and moved directories

Flow: this process is described above

More information:

-suspect-file is where you provide an input file with inode and type for files, links, or directories that have had a change of some kind

If you also use -suspect-time, then each input suspect will be checked with stat() to see if it has really changed ctime/mtime > suspect-time

-suspect-time is the time for suspect time comparison

-suspect\_mode 3 stats dirs, files, and links, compares with -suspect-time, does not use -suspect-file

-suspect mode 1 used with -suspect-file will compare inodes in the suspect-file with inodes of directories, files, and links. This is for when the file/link/directory changes are all provided in the suspect-file. ( if -suspect-time is provided each suspect provided will be checked ctime or mtime > suspect-time)

-suspect mode 2 used with -suspect-file will compare inodes in the suspect-file with inodes files and links only, directory ctime and mtime will be compared with -suspect-time. This is for when only file/link info is provided in the suspect-file and directory changes must be derived from the source file system. ( if -suspect-time is provided each suspect provided will be checked ctime or mtime > suspect-time)

-suspect mode 0 will ignore finding changes and just create a parent/child path map database.

For Spectrum Scale and Lustre, Suspect mode 1 is the most likely usable. The Lustre log would have dir, link, and file change indicators. The Scale ILM inode scan would have inodes from dirs., links, and files that have changes.

For storage systems that have no way to provide changes other than walking the entire source file system, use -suspect mode 3 with a -suspect-time set to the last time this utility was run.

## Concepts for providing suspect lists for storage systems

The purpose of the suspect list is to take advantage of source storage system capabilities to quickly find and/or log changes in some storage system dependent way, since there really aren’t standards for providing differences.

ZFS provides the zfs snapshot and zfs diff commands which will list changes and type of change.

EXT2appears to have no special capabilities for change discovery

EXT3 can log inodes and superblock changes

EXT2/3 the e2tools e2ls can list file system attributes by going directly to the base device

EXT4 apparently can log inodes and superblock changes

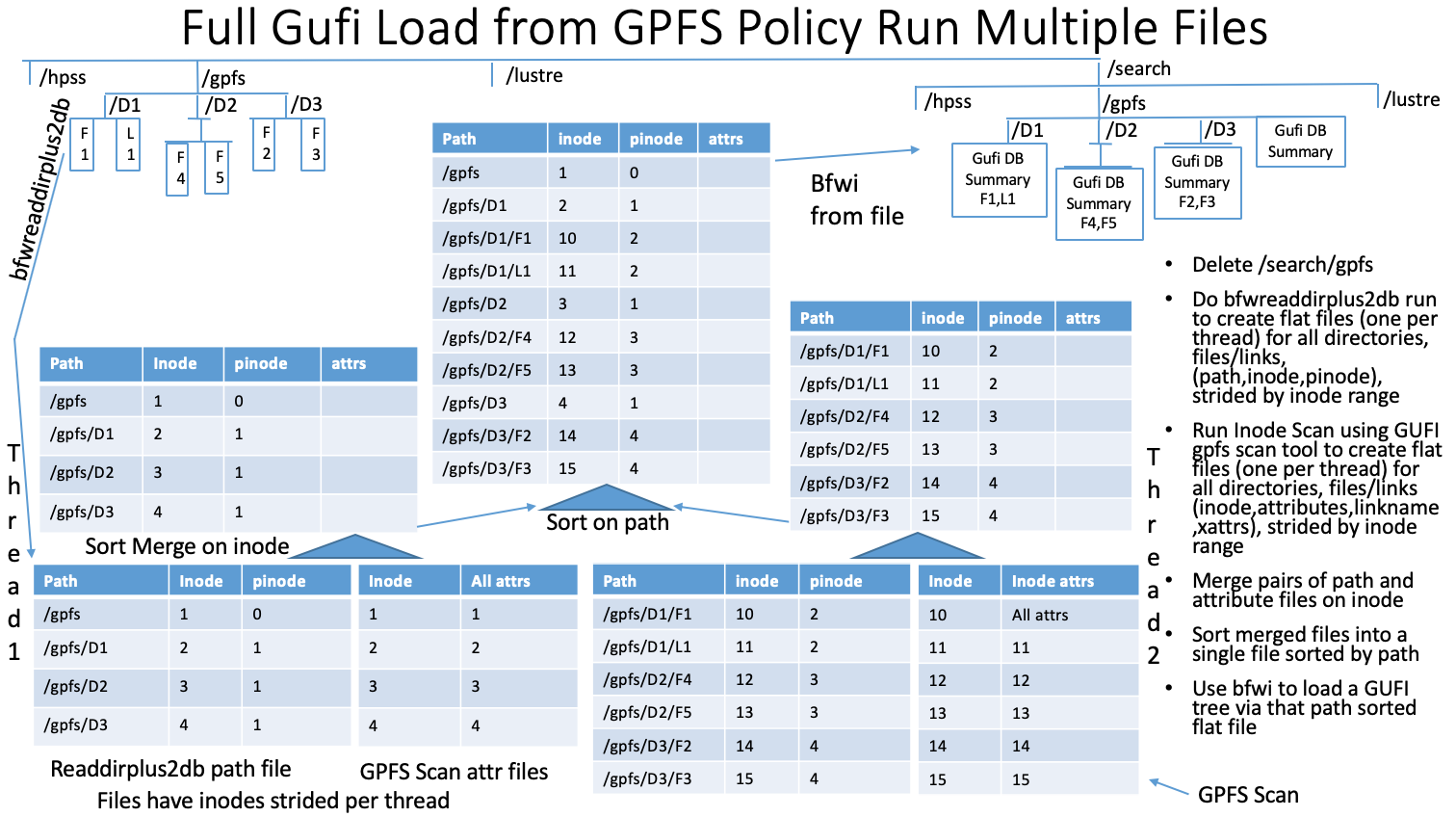
XFS can be made to use inotify tools, so inotify tools could provide some suspect list info

Lustre provides LFS find which lists file system attributes without going through collecting information from the OST file servers which won’t get you an accurate file size but it might get you an accurate mtime which you can compare to a last indexed date to make a list of files/directories that have had a change of some kind. Size would be gotten by the reindex a directory process. If the mds is ZFS based you could try to use snap and diff on the mdt and/or OST’s which might require mapping fid’s and aggregating info from the OST’s unless all you are looking for is mtime on the mds which might be able to make a list of suspects. Ldiskfs is an EXT file system and you could use E2tools on the mds and/or ost’s but again you might need to map fid’s unless all you are looking for is mtime changes on the mds. Of course Lustre also has a change log. It might be possible to convert that log into something that could be collected via the bfwreaddirplus2db walk to compare with the lustre log entries. Lustre also supports dmapi which could be used to catch file system updates similar to inotify tools.

IBM Scale has the ILM features which give you a shadow serialized inode table so you can scan the serial inode table very fast and even seek into the inode space and scan in parallel. This could make a list of suspects because the inode scan provides mtime/ctime, etc.. Normally to match the inode listing with the path/name you need to do a readdirplus walk of the full file system then do a sort merge of the readdirplus path/inode list and the inode/stat info list. Since the bfwreaddirplus2db process does a walk, it could look for a suspect list by inode, eliminating the need to do a sort merge. Scale also supports dmapi which could catch file system updates similar to inotify tools. Scale also supports varying degrees of audit logging capabilities.

## Producing full GUFI index from Scale ILM features (experimental)

To produce a full indexing of Scale, one can just do a normal gufi\_dir2index but you also could use a combination of ILM and GUFI tools described here:



The process first creates a set of inode/attribute files using the GUFI GPFS scan program, one file per thread strided by inode. Next bfwreaddirplus2db is used to walk the source file system tree with no stating, to produce a path/inode/pinode file per thread strided by inode with same stride as used by the GPFS scan. The pairs of matching strided files (inode/attribute and path/inode/pinode) are merged (a pair per thread), and then the resultant merged files are sorted into a single file in the required bfwi -u flat file order which requires the directory record and then children records follow that directory record immediately. In order to make sorting of flat files for bfwi -u flat file load you must sort on path (full pathname for directories and only path without name for files and links) as a first field sort and type as a second field sort. This is assisted by the fact that bfwreaddirplus2db has type in its output and also puts in a sort field that has path in it suitable for easy sort on these two fields. The resultant file is loaded into a GUFI by using bfwi with the option to take input from a file.

Incremental update of a GUFI tree using the GPFS inode scan capability simply uses the GUFI GPFS Inode scan program with the option to build a suspect list (inode type) one per thread with the suspect date provided from the last GUFI update time.

TSM backup (experimental)

As was described above regarding incremental updates of GUFI trees can be accomplished by providing a suspect input file into the incremental process. If you have an out of band way to find out what has changed in a file system, using that to provide this suspect input is a way to make the walking of the source storage system less disruptive.

In the case of if the source storage system is backed up by TSM, it is possible to use a query of TSM to tell you what files/links have been changed (backed up) since the last GUFI incremental and thus since that query provides inode for the changed files/links it is possible to format the output of that query into the standard suspect file format (inode type(f or l).

This process also requires that you query TSM to see if a backup has run since the last GUFI incremental to determine if its worth running a GUFU incremental, as if there hasntn been a backup, you would not be able to produce a valid suspect list making the GUFI incremental less useful as you would not catch all suspect directories properly.

Of course, since you ran a GUFI full or incremental some time ago, to determine if you want to do another GUFI incremental, you need to get time of last GUFI incremental into a variable we will call “lastgi”.

You can use this command to get the time last backup for the file system in question into a variable we will call “lastbu”.

dsmc q filesp /var | grep '/var' | awk '{ system("./tsmtime2epoch " $2 " " $3) }'

Notice the output of the dsmc command is piped through grep and awk and then fed into a program tsmtime2epoch which converts the date time provided by the dsmc command into a seconds since epoch for comparison with the “lastgi”.

if lastbu>lastgi then doing a GUFI incremental is fine, otherwise dont bother.

To start a new GUFI incremental don’t forget to put new GUFI time into new GUFI incremental file for use next time.

To query tsm to get a suspect list you will want to limit the files/links you put into the suspect list to only those backed up since the last GUFI incremental, you need to get the last GUFI incremental time to use in the dsmc query.

run this command to convert last GUFI incremental time and put the output into variable we will call “fdft”.

./tsmepoch2time lastgi

Tsmepoch2time which will give you a string with -fromdate and -fromtime for a dsmc backup command

Now you can run the dsmc query backup command looking for files backed up since last GUFI incremental and output a file called filesuspects

dsmc q backup -filesonly -detail fdft /var | grep 'Inode#' | awk -F 'Inode#' ' { print $2 } ' | awk ' { print $2 " f" } ' > filesuspects

Notice the variable “fdft” in this command which contains the -fromdate -fromtime info.

Now you have a suspect file to feed into bfwreaddirplus2db to finish up the incremental process called filesuspects

**Assisting suspect list generation**

If you are a file system implementor, having cheap ways to track changes is nice to enable GUFI and other indexing/backup mechanisms. With GUFI, it is not necessary to log every file change, just mark directories as “dirty” in the directory or in a log. It is not strictly necessary for the log to be in absolute order which enables parallelism in logging, and it is not necessary to log multiple changes made that impacts a directory or its contents. Of course, if you decide to deploy GUFI databases inside your file system directories themselves, then directory add/modify/move/delete becomes not necessary, but of course that is a tradeoff in adding traffic to metadata requests/scans from GUFI users doing queries contending with normal file/storage system activity and it complicates rollups which are part of how GUFI is so fast.

HPSStoGUFI – Loading a GUFI tree/index from an HPSS DB2 database. (in use at LANL)

# ToDo