Grand Unified File Index (GUFI)

Table of Contents

[License Info 3](#_Toc439563)

[Background 5](#_Toc439564)

[General 5](#_Toc439565)

[GUFI Requirements 6](#_Toc439566)

[Gufi design tradeoffs 7](#_Toc439567)

[How does GUFI work 9](#_Toc439568)

[General 9](#_Toc439569)

[Extraction 10](#_Toc439570)

[Query 11](#_Toc439571)

[Structure Reference 12](#_Toc439572)

[General 12](#_Toc439573)

[Tables/Schema 13](#_Toc439574)

[SQL Supplemental Functions 18](#_Toc439575)

[SQL functions beyond SQLite3 standard functions 18](#_Toc439576)

[GUFI Development Location 18](#_Toc439577)

[Contribution location 18](#_Toc439578)

[GUFI Building 19](#_Toc439579)

[Requirements 19](#_Toc439580)

[Directory Structure 19](#_Toc439581)

[Build 19](#_Toc439582)

[Test run 20](#_Toc439583)

[GUFI Commands Reference 20](#_Toc439584)

[Commands 20](#_Toc439585)

[bfwreaddirplus2db 20](#_Toc439586)

[bfwi 22](#_Toc439587)

[bfti 23](#_Toc439588)

[bfq 24](#_Toc439589)

[dfw 26](#_Toc439590)

[bfresultfuse 27](#_Toc439591)

[bffuse 28](#_Toc439592)

[querydb 29](#_Toc439593)

[querydbn 29](#_Toc439594)

[GUFI\_find (under construction) 30](#_Toc439595)

[Source Storage System Specific Commands 30](#_Toc439596)

[GPFS scan tool (under construction) 30](#_Toc439597)

[Bfmi (not recently maintained) 30](#_Toc439598)

[Tests/Tools 31](#_Toc439599)

[Commands/Scripts 31](#_Toc439600)

[tsmtime2epoch 31](#_Toc439601)

[tsmepoch2time 32](#_Toc439602)

[make\_testdirs 32](#_Toc439603)

[runbfwi 32](#_Toc439604)

[runbfti 32](#_Toc439605)

[runbfq 32](#_Toc439606)

[runquerydb 32](#_Toc439607)

[runquerydbn 32](#_Toc439608)

[runbfwreaddirplus2db 32](#_Toc439609)

[runbfqforfuse 32](#_Toc439610)

[runbffuse 32](#_Toc439611)

[rundfw 32](#_Toc439612)

[runlistschemadb 32](#_Toc439613)

[runlisttablesdb 32](#_Toc439614)

[rungroupfilespacehogusesummary 33](#_Toc439615)

[runuserfilespacehogusesummary 33](#_Toc439616)

[rungroupfilespacehog 33](#_Toc439617)

[runuserfilespacehog 33](#_Toc439618)

[runoldbigfiles 33](#_Toc439619)

[listschemadb 33](#_Toc439620)

[listtablesdb 33](#_Toc439621)

[userfilespacehog 33](#_Toc439622)

[groupfilespacehog 33](#_Toc439623)

[generategidsummary 33](#_Toc439624)

[generateuidsummary 33](#_Toc439625)

[oldbigfiles 33](#_Toc439626)

[groupfilespacehogusesummary 33](#_Toc439627)

[userfilespacehogusesummary 33](#_Toc439628)

[deluidgidsummaryrecs 33](#_Toc439629)

[gitest 33](#_Toc439630)

[dfw2gufitest 34](#_Toc439631)

[bfwiflat2gufitest 34](#_Toc439632)

[gengidsummaryavoidentriesscan 34](#_Toc439633)

[genuidsummaryavoidentriesscan 34](#_Toc439634)

[gufitest.p 34](#_Toc439635)

[Correctness tests: 35](#_Toc439636)

[cincr3deld 35](#_Toc439637)

[cincr3movd 35](#_Toc439638)

[cincr3addd 35](#_Toc439639)

[cincr3adddelmovdf 35](#_Toc439640)

[cincr2modf 36](#_Toc439641)

[cflatloaddfw 36](#_Toc439642)

[cflatloadfinddfw 36](#_Toc439643)

[cflatloadbfwi 36](#_Toc439644)

[cbfqbftiquerydb 36](#_Toc439645)

[cbfwiinsrc 36](#_Toc439646)

[cbfwrp2dbfullinsrc 36](#_Toc439647)

[cbfwrdp2dbincrinsrc 36](#_Toc439648)

[Source Storage System Specific Commands/Scripts 36](#_Toc439649)

[runbfmi (not recently maintained 36](#_Toc439650)

[runtsm 36](#_Toc439651)

[User Guide 37](#_Toc439652)

[Setup 37](#_Toc439653)

[bfq, querydbn, and querydb 37](#_Toc439654)

[bfq 37](#_Toc439655)

[querydbn 42](#_Toc439656)

[Compound Queries 43](#_Toc439657)

[querydb 44](#_Toc439658)

[bfresultfuse 44](#_Toc439659)

[bffuse 47](#_Toc439660)

[GUFI\_find (under construction) 49](#_Toc439661)

[Administration Guide 49](#_Toc439662)

[The Tree Summary Concept 49](#_Toc439663)

[Other Administrative Tools 50](#_Toc439664)

[Full GUFI Load from a source file system tree 51](#_Toc439665)

[Full GUFI Load from a flat file 53](#_Toc439666)

[Full GUFI load into source tree 55](#_Toc439667)

[Incremental GUFI Load 56](#_Toc439668)

[Incremental GUFI update of a GUFI tree directly within the source tree 59](#_Toc439669)

[GPFS Special Full and Incremental Load (needs updated from Scott) 60](#_Toc439670)

[TSM or other Backup System Special Incremental Load Assist 61](#_Toc439671)

[Lustre Special Full and Incremental Load (under construction) 62](#_Toc439672)

[HPSS Special Full and Incremental Load (under construction) 62](#_Toc439673)

# 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.
* #
* # -----
* # NOTE:
* # -----
* #
* # GUFI uses the C-Thread-Pool library. The original version, written by
* # Johan Hanssen Seferidis, is found at
* # https://github.com/Pithikos/C-Thread-Pool/blob/master/LICENSE, and is
* # released under the MIT License. LANS, LLC added functionality to the
* # original work. The original work, plus LANS, LLC added functionality is
* # found at https://github.com/jti-lanl/C-Thread-Pool, also under the MIT
* # License. The MIT License can be found at
* # https://opensource.org/licenses/MIT.
* #
* #
* # 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 before have we been tasked with storing, sorting, and organizing so much critical information at such a vast scale. This has created major challenges for supercomputing-scale data centers, where data is growing more rapidly than the capabilities of the file-systems that house that data. Our data is useless, if we can’t wrangle and query it efficiently, but queries can also have a devastating impact on the performance of ongoing computations/operations, which must themselves have unimpeded storage/file-system access to avoid wasting computational resources. Furthermore, security requires that results of user-queries must be constrained to include only data that a given user is allowed to see, including even the filenames. It is also useful to consider the differences in types of queries done by users versus by systems/data management professionals and try to accommodate both areas in a comprehensive indexing capability.

The Grand Unified File Index (GUFI) is an index that holds file-system metadata (i.e. filenames, access and creation dates, file attributes, extended attributes, etc.) pulled from huge file/data storage-systems using full and incremental index update, allowing 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 resources.

GUFI can be applied to a variety of file and archive systems (tape archives, Parallel File-systems (PFS), and others), making it a ubiquitous solution for indexing that supports arbitrary queries and unifying information from all the places where a file might reside. It’s a one-stop shop for file-system metadata.

Now, sitting in the driver seat, a GUFI user can rapidly search metadata from billions of files without straining the performance of the file-system itself. This allows the users and system administrators in a big data center 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 home, project, scratch, campaign, and archive
* Obey and support full POSIX tree attributes, permissions, hierarchy representation, etc.:
* Metadata only supporting attribute and extended attributes
* Shared index for users and admins
* Parallel search capabilities that are very fast (minutes for billions of files/dirs)
* Parallel metadata extraction with both full and incremental updating of the index, make the updating of this index as painless as possible
* Index can live in a separate space or within the source file/storage systems themselves where possible
* Can appear as mounted file system where you get a virtual image of your file metadata based on query input and also a pre-run query can also appear as a mounted file system
* Full/Incremental update from sources with reasonable update time/annoyance
* Provide a way to work in a pure POSIX environment requiring only a POSIX interface to storage systems for full and incremental metadata extraction
* Exploit special interfaces from source file/storages systems provided by some storage systems for mass metadata manipulation/extraction like GPFS ILM
* Be open source software and leverage other open source software and/or open interfaces
* Very transparent and simple so one can easily understand/enhance/administrate with simple to understand formats. Avoid black box anything.
* Extensible capabilities, especially in the query area, for example enable outputs from query to be consumable by humans or other programs and ability to connect in external data sources into query results. Ability to store both base POSIX information and potentially source storage system unique metadata per entry.
* The intent is to provide a nearly consistent index (doesn’t have to be a perfect snapshot or continuously keeping up with source file/storage systems.
* The intent is not to produce a policy management system for users or admins but of course it could provide the index usable for policy management function.
* Keep the code base small by leveraging existing technology as much as possible both hardware and software including:
  + flash storage: Assume the index would be small enough to fit in flash storage or even perhaps memory with a few hundred bytes of index per entry. Assume metadata mount per directory will typically be a few kilobytes so parallel access ends up looking like multi-kilobyte random reads which is well suited to flash devices which can provide millions of read IOPS.
  + both process and thread parallelism: where possible enable both types of parallelism for speed and efficiency
  + A standard and powerful basis for search: enable the exploitation of the power and stability of an existing basis for search even if the interface to the user doesn’t export that interface (like SQL or other)
  + commercial database technology: no need to invent our own underlying indexing/database technology given the abundance of solutions available
  + commercial file system technology: leverage commercial file system technology and its strengths including extremely fast traversal and access control which is probably of the most optimized code in the world
  + 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?
  + Events like rename (mv /top/b2 /top/b1) high in the source tree causes potentially billions of records to be updated/replaced
  + We desire a single parallel index capability used by users and admins which requires POSIX sharing security to be enforced which is complicated by the inheritance that directory read/execute has on the tree below which is one of the powerful concepts of POSIX. This security capability is hard to implement in a flat space due to the tree inheritance feature.
  + A single user can see only very little of the overall metadata space, simple flat sharding requires looking at a lot of records that a tree/graph based approach would eliminate.
  + Sharding is however important to enable parallelism, just simple flat sharding appears to be problematic.
* Leverage things that work very well, ways to reduce the number of records needed to be looked at/updated, etc.
  + Just buy product if it exists and if possible without lock in etc. that does much of what we want. We were unsuccessful in finding the product to buy.
  + The POSIX tree walk (directories only (readdir+) mechanism – optimized to the extreme, speed, enables breadth parallel fan out, and enforces shared security, enables renames at very low cost
  + Breadth first search parallelizes extremely well especially for threading mechanisms and especially wide namespaces enable rapid 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. SQL is an amazingly powerful way to express queries, more powerful than actually needed for this application. SQLite3 is open source, has enormous support, is very fast for this use case, is extensible and allows for connecting to other data sources and outputting about any type of output.
  + Flash devices can sustain extremely high IOPs where IOPs are in small numbers of kilobytes or larger.
  + Trees are a natural structure for rolling up representative data, so you get indexing function almost for free.
  + If you consider just the directories (which provide the shape of the tree) in most large deployed file/storage systems, there is a natural collection of metadata entries (in a single directory) of 20-1000s of entries (files/links) giving a nice way to shard (by directory) which enables parallelism while still honoring POSIX security. In many POSIX file systems, the more files in a single directory the worse the performance, but with an embedded file system file with few to no joins, the more entries in that flat file the better as that represents a serial read.
  + Embedding database files into a POSIX tree allowed for replicating the source file/storage system metadata entirely to enable an isolated performance domain or placing the index into the source storage/file system itself. This approach also enables lots of choices for the underlying file system to store the index in and provides trivial ways to backup/copy/replicate the index. This approach is trivial to understand and is completely transparent. It also enables the ability for query output to appear as a POSIX file system pretty trivially as well.

## How does GUFI work

### General

Most file-systems employ the concept of a “tree” of directories containing sub-directories, containing sub-sub-directories, etc. GUFI partitions file-system metadata into many small databases, kept in a directory-tree having the same structure and access-controls as that of the original file-system or in the source file system itself. Each folder in the original file-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 directory). Furthermore, because the directory access-controls are replicated from the original file-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 mimic’s POSIX security/access control exactly.

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

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

### Extraction

In this step, GUFI scans an entire (full or incremental) source large scale file-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 file-system, 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 in the corresponding directory of the original file-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.

Additionally 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)

Again, the GUFI-tree replicates the same standardized POSIX access-control settings of the original file-system tree. Thus, a well-tested security is provided “for free” by the POSIX file-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 file-system, they will not have access to the corresponding GUFI folder and databases in the GUFI system. Queries that span the GUFI-tree in parallel will simply not enter directories that are off limits.

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, the following functions are also provided which are additions to normal SQLite3 functions (like date formatting etc.)

* The Uidtouser() function converts the Unix numeric userid to a user name
* The gidtogoup() function converts the Unix numeric groupid to a group name
* The modetotxt() function converts the Unix numeric mode to the human readlable drwxrwxrwx string
* Because paths are not stored in the databases because of the same difficulty of making updates like moves to the index difficult the following path related functions are available
  + The path() function provides the relative path to that entry (not including the entry name which is in the name variable)
  + The fpath() function provides the full path to that entry (not including the entry name which is in the name variable)
  + The epath() function provides the immediate directory name that the entry is in (not path just name)

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 ad hoc results 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. These commands have to be metadata only like ls, find, stat, xattr -l, etc.

# Structure Reference

## General

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

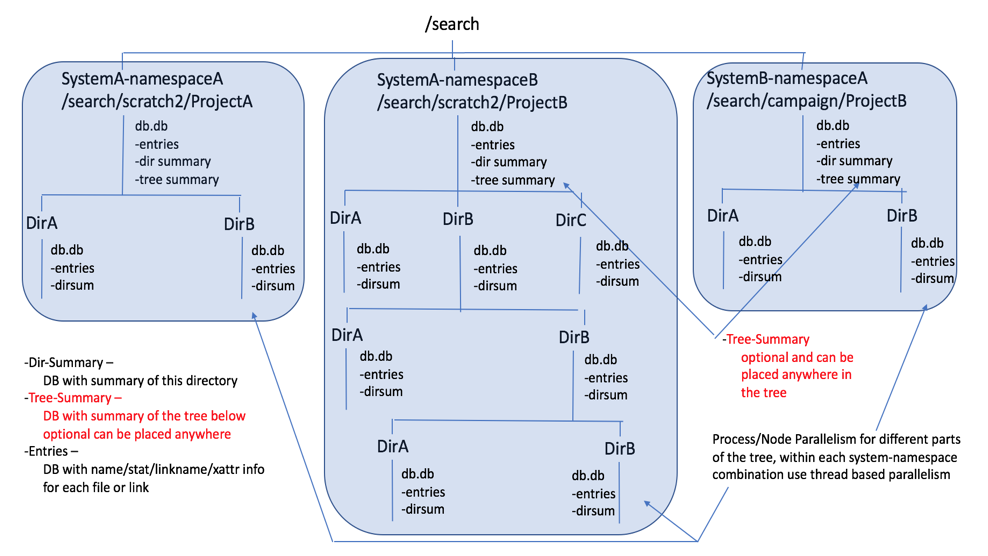


Figure 2 GUFI Overall Structure

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.

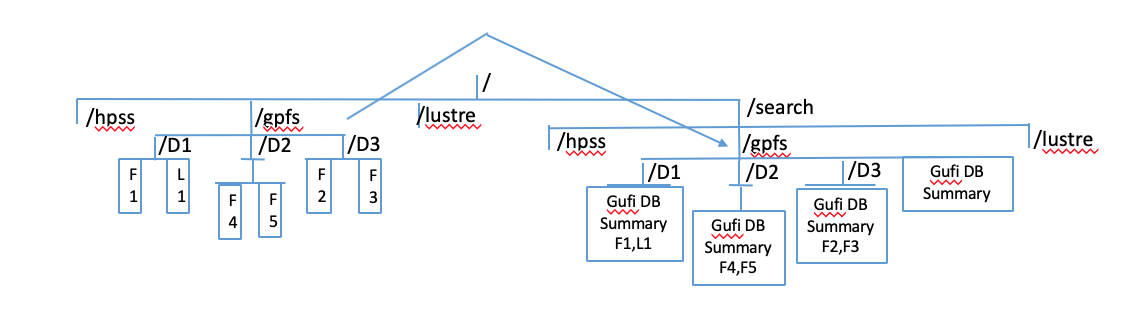


Figure 3 Mapping of source file system into GUFI tree

## 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, xattrs 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
    - xattrs 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, xattrs TEXT, totfiles INT64, totlinks INT64, minuid INT64, maxuid INT64, mingid INT64, maxgid INT64, minsize INT64, maxsize INT64, totltk INT64, totmtk INT64, totltm INT64, totmtm INT64, totmtg INT64, totmtt INT64, totsize INT64, minctime INT64, maxctime INT64, minmtime INT64, maxmtime INT64, minatime INT64, maxatime INT64, minblocks INT64, maxblocks INT64, totxattr INT64,depth INT64, mincrtime INT64, maxcrtime INT64, minossint1 INT64, maxossint1 INT64, totossint1 INT64, minossint2 INT64, maxossint2 INT64, totossint2 INT64, minossint3 INT64, maxossint3 INT64, totossint3 INT64,minossint4 INT64, maxossint4 INT64, totossint4 INT64, rectype INT64, pinode INT64);";
    - * 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
      * xattrs 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

Additionally 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 updating the index for moves of directories/files difficult, so parent inode is looked up in the summary so that parent inodes are never stored with child records.
  + "create view pentries as select entries.\*, summary.inode as pinode from entries, summary where rectype=0;";
* The vsummarydir view provides access to the entire directory summary and not a partial directory summary (say by user or group).
  + "create view vsummarydir as select \* from summary where rectype=0;";
* 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)
  + "create view vsummaryuser as select \* from summary where rectype=1;";
* 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)
  + "create view vsummarygroup as select \* from summary where rectype=2;";
* The vtsummarydir view provides access to the entire tree directory summary and not a partial directory summary (say by user or group).
  + "create view vtsummarydir as select \* from treesummary where rectype=0;";
* 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)
  + "create view vtsummaryuser as select \* from treesummary where rectype=1;";
* 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)
  + "create view vtsummarygroup as select \* from treesummary where rectype=2;";

If you are going to use bfresultfuse, the fuse daemon that can be mounted over a query result the output database table must have the following fields for both directories and files.

* "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);
  + All fields are just the same as the entries table except fullpath is an additional field and can be generated using the fpath() function.

# SQL Supplemental Functions

## SQL functions beyond SQLite3 standard functions

The following functions are added to the SQLite3 funcionality by GUFI:

* The uidtouser() function converts the Unix numeric userid to a user name
* The gidtogoup() function converts the Unix numeric groupid to a group name
* The modetotxt() function converts the Unix numeric mode to the human readlable drwxrwxrwx string
* Because paths are not stored in the databases because of the same difficulty of making updates like moves to the index difficult the following path related functions are available
  + The path() function provides the relative path to that entry (not including the entry name which is in the name variable)
  + The fpath() function provides the full path to that entry (not including the entry name which is in the name variable)
  + The epath() function provides the immediate directory name that the entry is in (not path just name)

Also, a handy function for converting unix epoch timestamps to date/time in SQLite3 is

datetime(date variable like mtime or other,'unixepoch').

# 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

## Requirements

sqlite3 for all

prce (a package that enables regular expression searching in sqlite3) (v 8.42 currently used)

mysql for bfm\* (pulling GUFI info from a Robinhood database)

db2 for bfd\* (pulling GUFI info from an HPSS DB2 database)

fuse for the two fuse applications bffuse and bfresultsfuse

The build seems to be working on OSX (10.12.6), and Linux (CentOS 7).

## Directory Structure

/ .c .h makefile, README, NOTES.txt, TBD for only the currently supported things

/misc all old stuff/unsupported stuff

/test a test directory with all the current functional tests and a /test/testdir input tree etc.

/test/old all old tests

/scripts handy scripts for doing various things

/C-Thread-Pool thread pool package

## Build

# Only needed once (C-Thread-Pool is a git submodule)

git submodule init

git submodule update

# suggested:

make clean

# build (into local dir)

make -> libgufi.a, bfwi, bfti, bfq, querydb, querydbn, make\_testdirs, bfwreaddirplus2db

# if desired, do any/all of these:

make mysql -> bfmi

make dfw -> dfw

make tools -> querydb, querydbn, make\_testdirs

# for debugging:

make clean

make <some\_target> DEBUG=1

# to reset the 'test' directory /test/testdir (e.g. after testing):

make clean\_test

## Test run

# example\_run builds the GUFI software, then generates a dummy

# directory-tree in test/in, extracts the corresponding GUFI-tree, and

# runs a simple query.

./example\_run

# GUFI Commands Reference

## Commands

The following are detailed descriptions of the individual command tools.

bfwreaddirplus2db

Breadth first readdirplus walk of input tree to list the tree, or create an output db and/or output files of encountered directories and or files. This program has two primary uses: to find suspect directories that have changed in some way that need to be used to incrementally update a GUFI tree from source file system changes and to create a full dump of all directories and or files/links either in walk order (directory then all files in that dir, etc.) or striding inodes into multiple files to merge against attribute list files that are also inode strided.

Bfwreaddirplus2db can also be used to do a full gufi if you are writing the gufi tree into the source tree. Bfwreaddirplus2db can replace all directory databases that are suspect (based on the various suspect modes (stat dir, stat dir and file, provided an input file with dirs and or files that are suspect (inodes). You can also tell bfwreaddirplus2db that all directories are suspect using the -Y flag and the -b flag tells it to create a gufi in the source tree. Of course you can use it to incrementally update (replace suspect directory databases) as well using the suspect mode methods.

Usage: bfwreaddirplus2db [options] input\_dir

options:

-h help

-H show assigned input values (debugging)

-o <outfile> output file one per thread writes path, inode,

pinode, type, sortfield to file (sortfield enables sorting for bfwi load from flat file)

-r insert files and links into db

-b build GUFI index tree in the src tree

-b is in conflict with -t in bfwreaddirplus2db

-R insert dirs into db snapshot the dir structure

-n <threads> number of threads

-d <delimiter> delimiter for output file

-O <outdb> write a set of output dbs one per thread with

directories and/or files and mark

Suspect directories output path, inode,

pinode, type and suspect flag

-Y default all directories as suspect

-Z default all files links as suspect

-W <insuspectfile> path to input suspect file, if suspects are

generated via another program like a GPFS

Inode Scan

-A <suspectmethod> suspect method (0 no suspects, 1 suspect

File for dirs, files/links, 2 suspect

stat dirs and file for files/links, 3

suspect stat dirs, files/links)

-c <suspecttime> time in seconds since epoch for suspect

Comparison, (ignored if gufi tree in src

tree)

-g <stridesize> stride size for striping inodes into separate

output db's and or files

-t <to\_dir> if this is provided suspect directories will

have gufi dbs created into this directory

named by the suspect dirs inode, this flag is

in conflict with -b flag in bfwreaddirplus2db

-x process xattrs if you are creating gufi dbs

of suspect directories

Flow:

if using input suspect file (inode and type), read that in and put into memory trie for lookups

input directory is put on a queue

threads are started

loop assigning work (directories) from queue to threads

each thread lists the directory readdirplus

if using suspect file, see if this inode matches a suspect file or dir or link

if using suspect mode involving stat then use that to help determine of a directory is suspect

if we are striping the inodes into different outdb's and/or files, we have to lock on the output db/file to place records into proper outdbs/files

if directory put it on the queue write to outdb/file

if link or file write to outdb/file if writing files

put dirs and/or files into output db (path, inode, pinode, type, suspect) into output dir

if to\_dir provided suspect directories will have gufi dbs created into this directory named by the suspect dirs inode

if to\_dir is directly in the source tree then place the gufi dbs in the source tree and on incremental just use the ctime of the gufi db to determine new changes to the directory to know if the db needs to be refreshed

if writing output file write files, links, and directories to output files per thread (can use striping of inodes across files)

close directory

end

close outdb/outfile if needed

bfwi

Breadth first walk of input tree to list the tree, or create a GUFI index-tree

Usage: bfwi [options] input\_dir

options:

-h help

-H show assigned input values (debugging)

-p print files as they are encountered

-n <threads> number of threads

-d <delim> delimiter (one char) if you are using an input

file

-x pull xattrs from source file-sys into GUFI

-P print directories as they are encountered

-b build GUFI index tree

-t output directory for GUFI index tree

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

thread-id suffix)

-D dont descend in the tree (make a single gufi db at

one level of the tree

-u input mode is from a file so input is a file not a

dir

future options:

-U create by user summary per directory

-G create by group summary per directory

If -P -p and -x are used in conjunction with -o to create an output file per thread

the format of the files created make it so that you can run bfwi again against one or all of them concatenated) with the -u option

Flow:

input directory is put on a queue either from a readdir/stat or from an input file

if an input file is provided, it must be of the proper form with directories in stanzas with directory then all files/links then another directory. The directories don’t have to be in any particular order.

output file(s) are opened one per thread

threads are started

loop assigning work (directories) from queue to threads by reading directories and walking or reading input file

each thread lists the directory readdir/stat and xattr if called for

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 or input file

write directory summary table

end

close output files if needed

if desiring an output file you can end up with an output file per thread

Location of GUFI-tree:

bfwi will re-create the dir-path of <input\_dir> underneath <to\_dir>.

For example, if <input\_dir> is /a/b/c and <to\_dir> is /q/r/s, the GUFI-tree will

be created at /q/r/s/a/b/c.

This would be the path to provide to other commands that take a

GUFI-tree as input. We have debated whether the GUFI-tree should just be build at /q/r/s/c. That may come in a future release. We're opento discussion.

If you are putting your GUFI-tree index right into the source tree, you must follow the same syntax where input\_dir is the top of the src tree and -t to dir is the directory above the source tree. This keeps the same syntax for where you want your gufi tree built.

Bfwi will recognize that the to dir and input dir are the same real paths and if so will save some work on creating output directories, setting the mode of the output directories, and will ignore the database file (DBNAME defined in bf.h default db.db) when building the index. This way indexes built in the source tree and in another tree will create the exact same index.

bfti

Walks breadth first below the input directory path and summarizes

all directories below it into a **tree summary table** record by reading all the directory summaries in the tree below

Usage: bfti [options] GUFI\_tree

options:

-h help

-H show assigned input values (debugging)

-P print directories as they are encountered

-n <threads> number of threads

-s generate tree-summary table (in top-level DB)

GUFI\_tree path to GUFI tree-dir

future options:

-U create by user summary record

-G create by group summary record

Flow:

input directory is put on a queue

threads are started

loop assigning work (directories) from queue to threads

each thread reads the directory and the summary table for each the dir

if directory put it on the queue

accumulate each directory summary into a global summary for all directories below the input directory

close directory

end

open/create and write tree summary record into treesummary table that summarizes all the directories below it

NOTE: The input <GUFI\_tree> should've already been created via 'bfwi'.

See the note under bfwi, regarding location of created GUFI-trees.

bfq

Breadth first walk of a GUFI tree. We optionally perform queries

against the tree-summary, the directory-summary, and/or the directory contents tables, per directory encountered (obeys POSIX permissions for access to directory info). You supply your own SQL statements for tree, summary, and entries, and select AND/OR logic between tree/directory/entry queries. The traversal can write its output to

stdout, one file per thread, or one database output per thread. SQL init allows you to run an SQL statement per thread associated to the output db before starting the walk, and an SQL statement per thread on the output db after the walk.

Usage: bfq [options] GUFI\_tree

options:

-h help

-H show assigned input values (debugging)

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

-S <SQL\_sum> SQL for summary table

-E <SQL\_ent> SQL for entries table

-P print directories as they are encountered

-a AND/OR (SQL query combination)

-p print file-names

-n <threads> number of threads

-o <out\_fname> output file (one-per-thread, with thread-id suffix), implies -e 1

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

-O <out\_DB> output DB, implies -e 1

-I <SQL\_init> SQL init

-F <SQL\_fin> SQL cleanup

-y <min level> minimum level to go down

-z <max level> maximum level to go down

-v <count> number of intermediate databases to use (preferrably prime)

-w <count> number of intermediate databases to skip when selecting (should be primitive root of intermediate database count, or 1)

-G <SQL\_aggregate> SQL for aggregated results (deaults to "SELECT \* FROM entries")

-J <SQL\_interm> SQL for intermediate results (deaults to "SELECT \* FROM entries")

-e <0 or 1> 0 for aggregate, 1 for print without aggregating (implied by -o and -O)

GUFI\_tree find GUFI index-tree here

Default behavior is to use interim/aggregate db’s, you have to turn that off with -e 1 if you don’t want interim/aggregation to occur.

Flow:

input directory is put on a queue

output file(s) are opened one per thread if needed

output dbs are opened if needed one per thread

Interim result dbs are opened if needed as specified

if init SQL provided run once per thread

threads are started

loop assigning work (directories) from queue to threads

each thread lists the directory and queries the gufi tables for that directory

if directory put it on the queue

if treesql input run query on treesummary table

and/or applied on whether to continue

if dirsql input run query on summary table - if printdir - print, if output to db do that

and/or applied on whether to continue

if entsql input run query on entries table - if print - print, if output to db to that

close directory

end

close output files if needed

if fin SQL provided run that per thread

close outputdb

close interim db’s

attach interims to produce a aggregate db from interim result dbs

run query on aggregate result db

you can end up with an output file per thread and/or a dbfile per thread

Compound SQL statements:

When providing SQL statements to bfq and querydb you can put more than one SQL statement in the same string using semicolons at the end of each statement, however the only SQL statement that will have output displayed if you have chosen to display output is the last SQL statement in the string. This enables complex things like attaching an input database to join with on each query (issued at ever level/directory found), or other highly powerful but complex things.

dfw

Depth first walk of tree using readdir, or readdir/stat, or

readdir/xattr or readdir/stat/xattr based on passed-in flags, just writes output to stdout

dfw - dfw depth first file system tree walk printing:

path, type, stat info, linkname

optionally if asked for xattrs

syntax dfw -s -x -h -d -l directorytowalk (or file to print info on)

-s stat everything - otherwise this is just a readdirplus walk

-x get xattrs

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

-D dont traverse/descend the tree, just stat this directory or file

-l print in loader format (only works if -x and -s are provided

-h help

if you run with no options, this is just a readdirplus walk and print

if you run with -s it stats everything and prints

if you run with -x it pulls xattr info and prints that

If you us -s -x -d '|' -l in conjunction (using any delimiter you wish)

dfw will print out a record suitable for sorting and then loading using bfwi -u option which loads/builds a gufi tree from a flat file

the file format for bfwi -u is director then all the files and links for that directory, then another directory etc.

the record format is pathname,normal attrs, linkname, xattrs and must be delimited by something unique from the data

the output of dfw in this use mode puts 2 sort fields on the end of the records (which can be left in the file as bfwi -u will ignore them)

the sort records are path (the directory full pathname for a directory or just the directory path for a file/link (field 17), and type (field 18) )

To get the proper order you need to sort the file using the following method or similar

sort -t '|' -k17,17 -k18,18 -o sortedoutputfile outputfromdfw

this sortedoutputfile can be used in bfwi -u as the input file to create a gufi tree from

dfw does not have sqlite or cthreads calls so it should be easier to build this on a computer, run to get the output file and move the output file

to a machine for bfwi to be used to load into a gufi tree

Normally you would pass dfw the name of a directory and it would walk the tree, but since the input is a pathname, it could a pathname be to a file

If its a pathname to a file, dfw will just print info about that file. This enables you to use something like

find path -exec dfw -s -x -d '|' -l -D {} \; which will print information about files that are found using the find command and make a suitable file for loading

into gufi using bfwi -u or other uses. Notice the use of -D which tells dfw to not descend into the tree, just stat this pathname and print,

because find will be traversing or whatever you ask it to do.

Flow:

if input is a file/link then stat/pull xattrs and print,

if input is a directory then single threaded depth first walk, using readdir+, or stat depending on input parms and optionally get xattrs, write output to stdout

bfresultfuse

Fuse daemon to read the output of a properly formed bfq query with output dbs and present it as a fuse mount point. It is designed to be run as a user in conjunction with an output database from a bfq query that the user has access to.

Usage: bfresultfuse -s mountpoint tablename\_from\_bfq\_query inputdb number\_of\_inputdbs

options:

-s tells fuse to run single threaded –

this has only been tested in single

thread mode

mountpoint directory on which you want the fuse

mounted

tablename\_from\_bfq\_query this fuse daemon takes the output of a

bfq query which produced output db's

(one per thread) as input to the fuse

the query must be formed properly to

have the right fields/order in the

output dbs

see test/runbfqforfuse as how to run

bfq to produce output dbs suitable for input into this fuse daemon

this is the output table name in the

output dbs from the bfq run

inputdb base name for output dbs of bfq query

used as input to this fuse

(basename.threadnum)

number\_of\_inputdbs this is how many threads you ran on the

bfq to produce the output dbs to tell

the fuse how many dbs there are to

union together

Flow:

This is a fuse daemon that honors

getattr, readdir, readlink, access, getxattr, listxattr

meaning you can only run commands that use the implicated posix calls against this mount point like

ls, stat, find, etc. you can not run file ops like open/read/write/close etc. so no cat or the like

it issues queries into the inputdbs to respond to these fuse requests. The fuse daemon is designed to work on the output of a bfq query like the following

../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 testdirdup2/testdir

As you can see -O created an ouitput db per thread -I is used to create the output table(s) one per thread with the appropriate fields for the fuse deamons expectations. The query must contain -S and -E to get both directory and file/link records and those queries use the insert into output table select method.

bffuse

Fuse daemon to sit atop the gufi tree and present the gufi tree as a mounted fuse file system on the mountpoint provided. It will honor a partial sql statement you provide as input. The partial sql is just the where part and only supports where clause on files/links, so like “size > 40000”. You must look at the schema for the entries table in the gufi db to see what fields you have to provide sql where clause

Usage: bffuse mountpoint top\_dir\_of\_gufitree

options:

-s tells fuse to run single threaded –

this has only been tested in single

thread mode

mountpoint directory on which you want the fuse

mounted

top\_dir\_of\_gufitree point this where in the gufi tree you

want to have this fuse file system

to present as a fuse file system

it can be the very top of the gufi

tree but it can be lower in the

gufitree as well

"sql where clause" the sql where clause on the

files/links (on the entries table)

It is designed to be used in single user mode you launch the fuse and point it at the place in the gufi tree you want to, and provide the query criteria for what types of files/links you want to look at

all subsequent readdir/stat operations honor that query criteria

when you use standard file system commands on the fuse mount like find/ls/stat etc.

Flow:

This is a fuse daemon that honors

getattr, readdir, readlink, access, getxattr, listxattr

meaning you can only run commands that use the implicated posix calls against this mount point like

ls, stat, find, etc. you can not run file ops like open/read/write/close etc. so no cat or the like

it uses the gufi tree for directory tree placement lookups and issues queries into the gufi database per directory level to respond to these fuse requests

it is designed to be run as a user, if you run it as root then it should be a private mount point for root

### querydb

run an SQL query against a table in a single level directory GUFI db - just point it at the directory containing the db

Usage: querydb [options] [-s] DB\_path SQL

options:

-h help

-H show assigned input values (debugging)

-N print column-names (header) for DB results

-V print column-values (rows) for DB results

DB\_path path to dir containinng db.db.\*

SQL arbitrary SQL on DB

Compound SQL statements:

When providing SQL statements to bfq and querydb you can put more than one SQL statement in the same string using semicolons at the end of each statement, however the only SQL statement that will have output displayed if you have chosen to display output is the last SQL statement in the string. This enables complex things like attaching an input database to join with on each query (issued at ever level/directory found), or other highly powerful but complex things.

### querydbn

run an SQL query against output databases (one per thread) from bfq, it can be one per thread, so provide prefix of the dbname, the program will create a union of the tables from each of the db files and it will get a view name by adding a v to the beginning of your table name

Usage: querydbn [options] [-s] DB\_path DB\_count SQL tabname

options:

-h help

-H show assigned input values (debugging)

-N print column-names (header) for DB results

-V print column-values (rows) for DB results

-p print file-names

-s dir-summary (currently-unused internal functionality)

DB\_path path to dir containinng db.db.\*

DB\_count number of DBs (should match thread-count used in 'bfq')

SQL arbitrary SQL on each DB (unified into single

view)

table\_name name of view table = 'v<table\_name>'

Compound SQL statements:

When providing SQL statements to bfq and querydb you can put more than one SQL statement in the same string using semicolons at the end of each statement, however the only SQL statement that will have output displayed if you have chosen to display output is the last SQL statement in the string. This enables complex things like attaching an input database to join with on each query (issued at ever level/directory found), or other highly powerful but complex things.

### GUFI\_find (under construction)

Jason to provide similar man page like text for this utility

## Source Storage System Specific Commands

### GPFS scan tool (under construction)

Scott to provide similar man page like text for this utility

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

## Tests/Tools

### Commands/Scripts

#### tsmtime2epoch

Usage: tsmtime2epoch datestring timestring

datestring – mm/dd/yyyy as it would come from dsmc q filesp /filesystem

timestring- hh:mm:ss as it would come from dsmc q filesp /filesystem

output – printable seconds since epoch

#### tsmepoch2time

Usage: tsmepoch2time seconds-since-epoch

seconds-since-epoch - printable seconds since epoch

output – “-fromdate=mm/yy/yyyy -fromtime hh:mm”

For use in a dsmc q backup command

#### make\_testdirs

Usage: make\_testdirs [ options ] dirname

options:

-d <n\_dirs> number of subdirs

-f <n\_files> number of files per subdir

-h help

The following functional tests all work together from the test directory, create a gufi index from input testdir into testdirdup and run queries/etc.

runbfwi - run bfwi and create gufi tree in testdirdup/testdir from input testdir

runbfti - run bfti and create a tree index at the top of testdirdup/testdir

runbfq - run various queries on testdirdup/testdir gufi tree including output files and output dbs and files

runquerydb - do various queries on one of the gufi dbs in testdirdup/testdir

runquerydbn - do various queries on outdb.\* output dbs using the union capability of querydbn

runbfwreaddirplus2db - run tests that walk trees using readdirplus and write path,inode,pinode etc. to output dbs

runbfqforfuse - run bfwi to create gufi then run bfq to create an output db and then run bfresultfuse on that output db

runbffuse - run bfwi to create gufi then run bfq to show you have a gufi then run bffuse over that gufi

rundfw - do various walks on the testdir tree

runlistschemadb - list the schema in a gufi db

runlisttablesdb - list the tables in a gufi db

rungroupfilespacehogusesummary - generate gid summary records via summary and entries

runuserfilespacehogusesummary - generiate uid summary records via

summary and entries

rungroupfilespacehog - do parallel query to create output dbs and list group file space hogs

runuserfilespacehog - do parallel query to create output dbs and list user file space hog

runoldbigfiles - do parallel query to craete output dbs and list old/big files

The following are scripts in the scripts directory

listschemadb - display schema of a gufi db

listtablesdb - display tables in a gufi db

userfilespacehog - do parallel query to create output dbs and list user file space hog

groupfilespacehog - do parallel query to create output dbs and list group file space hogs

generategidsummary - generate gid summary records via entries scan, write one record in provided directory database summary table for each group (type = 2)

generateuidsummary - generate uid summary records via entries scan, write one record in provided directory database summary table for each uid (type = 1)

oldbigfiles - do parallel query to create output dbs and list old/big files

groupfilespacehogusesummary - generate gid summary records via summary and entries

userfilespacehogusesummary - generate uid summary records via summary and entries

deluidgidsummaryrecs - delete uid and gid summary records in summary tables.

gitest- run an incremental gufi update test using both suspect mode 2 and 3 and test it against full gufi loads

dfw2gufitest - a python script that makes a source tree, makes a full gufi tree using bfwi using the source tree, then runs dfw on the source tree to make a flat output file that is then sorted and bfwi is used to create another gufi tree using the flat file input and then uses bfq on both gufi trees to compare to see if the flat file load process gets the same answer as the source tree loaded gufi tree. The same thing is run again using find -exec dfw ... to produce the flat file for the new gufi tree and same comparison with original gufi is done.

bfwiflat2gufitest - a python script that makes a source tree, makes a full gufi tree using bfwi using the source tree, then runs bfwi on the source tree to make flat output files that that are concatenated (one per thread) and bfwi is used to create another gufi tree using the flat file input and then uses bfq on both gufi trees to compare to see if the flat file load process gets the same answer as the source tree loaded gufi tree

gengidsummaryavoidentriesscan - generate gid summary records in summary tables using shortcut if all files in directory are in same group

genuidsummaryavoidentriesscan - generate uid summary records in summary tables using shortcut if all files in directory are in same user

gufitest.p

Gufitest allows you to run all the following tests just by providing the argument for that test. It also allows you to run all correctness testing and all functional testing. It also lets you run all tests.

The tests are correctness or functional. The functional tests are largely stand alone bash scripts in test or scripts directory.

Gufitest.py creates a temporary directory /tmp/gufitest and puts a source tree in that space and runs the test from that directory so to not dirty up the source trees. For the correctness tests a value like number of rows from a db query or a bfq output or other is compared completely to known values or comparing multiple methods. These tests will be described below the argument test list. The functional tests are run in the temp directory and output from the test is stored in /tmp/gufifuncout. On functional tests the number of lines of output is compared to a known value.

This is the list of arguments you can provide to gufitest.py

argument required

all - all correctness and functional tests

allc - all correctness tests

allf - all functional tests

cincr3modf - correctness incremental mode 3 mod file

cincr3deld - correctness incremental mode 3 del dir segment

cincr3movd - correctness incremental mode 3 mov dir segment

cincr3addd - correctness incremental mode 3 add dir segment

cincr3adddelmovdf - correctness incremental mode 3 add del mov dir segment and add file

cincr2modf - correctness incremental mode 2 mod file (use file suspect list file)

cflatloaddfw - correctness flat load using bfwi -u from dfw

cflatloadfinddfw - correctness flat load using bfwi -u from find dfw

cflatloadbfwi - correctness flat load using bfwi -u from bfwi

cbfqbftiquerydb - correctness bfq and bfti and querydnb

cbfwiinsrc - correctness bfwi stored in src tree vs elsewhere

cbfwrp2dbfullinsrc – correctness bfwreaddirplus2db full gufi create in source tree compared to bfwi full gufi

cbfwrdp2dbincrinsrc - correctness bfwreaddirplus2db incremental gufi update in source tree compared to bfwi full gufi

fbfq - functional bfq

fbfti - functional bfti

fbfwi - functional bfwi

fdfw - functional dfw

fquerydb - functional querydb

freaddirplus2db - functional readdirplus2db

fquerydbn - functional querydbn

fgroupfilespacehog - functional groupfilespacehog

fgroupfilespacehogusesummary - functional use summary

fuserfilespacehog - functional userfilespacehog

fuserfilespacehogusesummary - functional use summary

fuidgidsummary - functional uidgidsummary

foldbigfiles - functional oldbigfiles

flistschemadb - functional listschemadb

flisttablesdb - functional listtablesdb

fgenuidgidsummaryavoidentriesscan - functional genuidgidsummaryavoidentriesscan

#### Correctness tests:

cincr3modf - correctness incremental mode 3 mod file

This test does incremental mode 3 after modifying a file, it tests incrementally updated gufi versus a new full gufi using bfq outputs.

cincr3deld - correctness incremental mode 3 del dir segment

This test does incremental mode 3 after deleting a directory segment, it tests incrementally updated gufi versus a new full gufi using bfq outputs.

cincr3movd - correctness incremental mode 3 mov dir segment

This test does incremental mode 3 after moving a directory segment, it tests incrementally updated gufi versus a new full gufi using bfq outputs.

cincr3addd - correctness incremental mode 3 add dir segment

This test does incremental mode 3 after adding a directory segment, it tests incrementally updated gufi versus a new full gufi using bfq outputs.

cincr3adddelmovdf - correctness incremental mode 3 add del mov dir segment and add file

This test does incremental mode 3 after adding, deleting and moving directory segments and files, it tests incrementally updated gufi versus a new full gufi using bfq outputs.

cincr2modf - correctness incremental mode 2 mod file (use

file suspect list file)

This test does incremental mode 2 after modifying a file, it tests incrementally updated gufi versus a new full gufi using bfq outputs.

cflatloaddfw - correctness flat load using bfwi -u from dfw

This test compares a load from flat using bfwi -u from output of dfw

cflatloadfinddfw - correctness flat load using bfwi -u from find

dfw

This test compares a load from flat using bfwi -u from output of find -exec dfw {}

cflatloadbfwi - correctness flat load using bfwi -u from bfwi

This test compares a load from flat using bfwi -u from output of bfwi

cbfqbftiquerydb - correctness bfq and bfti and querydnb

This test runs many queries using both bfq and in some cases bfti and bfq (using tree summary tables) and compares answers to known values. Some tests create output databases and uses querydbn to query resultant databases

cbfwiinsrc – correctness bfwi created gufi in the src tree compared with bfwi created gufi elsewhere compared with bfq from both gufi trees using bfq output.

cbfwrp2dbfullinsrc – correctness bfwreaddirplus2db full gufi db create in source tree compared to bfwi create of full gufi db not in source tree using bfq output.

cbfwrdp2dbincrinsrc – correctness test bfwreaddirplus2db for an incremental update of a gufi db in place in the source tree, replacing all suspect directory databases compared to bfwi full using bfq output.

### 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.

# User Guide

## Setup

The following examples are queries run against GUFI tree testdirdup/testdir which was created from a test directory testdir

This is ls -lR information about the 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

This command was run to produce the testdirdup/testdir GUFI tree

../bfwi -n 2 -xP testdirdup testdir

The following testdirdup/testdir GUFI tree with ls -lR

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

As you can see the GUFI tree contains a GUFI database at each directory called db.db.

## bfq, querydbn, and querydb

The primary low level user tools are bfq, querydbn, bffuse, bfresultfues, and querydb. Other tools that utilize these underlying tools like GUFI\_find and other scripts are possible to create.

### bfq

Query to standard out select all variables from entries tables and also use the relative path (path()) function, default delimiter is |

../bfq -Pp -n 1 x-E "select path(),\* from entries;" testdirdup/testdir

testdirdup/testdir|clink|f|8602501824|33261|1|2078|20|4|4096|8|1543878081|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir|a|f|8602501826|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir|gary''s dumb file|f|8602501828|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir|dumbcom,ma|f|8602501830|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir|d|f|8602501834|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir|b|f|8602501836|33261|1|2078|20|79655|4096|160|1543590368|1542940736|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir/c|ca|f|8602501838|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir/c|cd|f|8602501840|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir/c|cb|f|8602501844|33261|1|2078|20|0|4096|0|1543586004|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

testdirdup/testdir/c/cc|bfindex|f|8602501846|33261|1|2078|20|14252|4096|32|1543878081|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

As you can see the relative path is provided and all the variables are printed including xattrs

Query to standard out of selected fields and show date conversion (limit to 10 entries per directory). This time also select the summary record via the vsummarydir view for the directory which show up as type=d. You select from summary using -S and entries using -E.

../bfq -Pp -n 1 -S "select path(), name,type,size,datetime(mtime,'unixepoch') from vsummarydir limit 10;" -E "select path(),name, type, size, datetime(mtime,'unixepoch') from entries limit 10;" testdirdup/testdir

testdirdup|testdir|d|288|2018-11-30 13:53:24|

testdirdup/testdir|clink|f|4|2018-03-20 23:38:47|

testdirdup/testdir|a|f|0|2018-03-20 23:38:47|

testdirdup/testdir|gary''s dumb file|f|0|2018-03-20 23:38:47|

testdirdup/testdir|dumbcom,ma|f|0|2018-03-20 23:38:47|

testdirdup/testdir|d|f|0|2018-03-20 23:38:47|

testdirdup/testdir|b|f|79655|2018-11-23 02:38:56|

testdirdup/testdir|c|d|192|2018-11-30 13:53:24|

testdirdup/testdir/c|ca|f|0|2018-03-20 23:38:47|

testdirdup/testdir/c|cd|f|0|2018-03-20 23:38:47|

testdirdup/testdir/c|cb|f|0|2018-03-20 23:38:47|

testdirdup/testdir/c|cc|d|96|2018-11-30 13:53:24|

testdirdup/testdir/c/cc|bfindex|f|14252|2018-03-20 23:38:47|

As you can see relative path is used again, the directories show up as type=d records and the time stamp is converted to human readable form

Query to standard out but this time use full path function (fpath()) and only files listed and only files > 1000 bytes

../bfq -Pp -n 1 -E "select fpath(),\* from entries where size > 1000;" testdirdup/testdir

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup/testdir|b|f|8602501836|33261|1|2078|20|79655|4096|160|1543590368|1542940736|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup/testdir/c/cc|bfindex|f|8602501846|33261|1|2078|20|14252|4096|32|1543878081|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

As you can see only two files matched that query of files with size > 1000

Query to standard out but this time select only from the directory summary view where total bytes in all the files in that directory are > 20000 bytes

../bfq -Pp -n 1 -S "select fpath(),totsize from vsummarydir where totsize > 20000;" testdirdup/testdir

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup|79659|

As you can see only one directory satisfies that criteria

Query to standard out and select directories that have totsize > 20000 “and” files with size > 1000. The “and” is important and is the default. This query did not bother to look at files in other directories except the one that had a totsize > 20000.

../bfq -Pp -n 1 -S "select fpath(),type,totsize from vsummarydir where totsize > 20000;" -E "select fpath(), type, size from entries where size > 1000;" testdirdup/testdir

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup|d|79659|

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup/testdir|f|79655|

As you can see we got the directory again and the file in that directory.

Query to standard out and do same query as above where directory totsize > 20000 “or” files with size > 1000. This doesn’t exclude looking at files in directories with totsize > 20000

../bfq -Pp -n 1 -S "select fpath(),type,totsize from vsummarydir where totsize > 20000;" -E "select fpath(), type, size from entries where size > 1000;" -a testdirdup/testdir

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup|d|79659|

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup/testdir|f|79655|

/Users/ggrider/sqlite/sqlite-src-3180000/gufi-113018/test/testdirdup/testdir/c/cc|f|14252|

You can see that we got the one directory and two files this time

This “and/or” -a flag is very powerful as it can eliminate looking in directories for all the file/link entry information if that is what you want it to do.

Not shown is a similar function that if you or your GUFI administrator has placed tree summary records anywhere in the tree, when these are encountered, if you are using the -T “select variable1,etc. from vtsummarydir where totsize > 20000” “and” -S or -E, the and in this case will not traverse lower in the tree if the -T criteria is met, so you can eliminate walking huge portions of the GUFI tree if you are using tree summary records in levels of your directory tree. The syntax is the same as above and you can always use -T, -S, and -E with “and” so that all criteria must be met for all three queries in order for records to be chosen. This is an enormously powerful feature.

Queries can be used to produce output files as well. This query select all fields from the entries tables and writes them to output files (one per thread) and in this query there are two threads.

../bfq -Pp -n 2 -o outq -E "select path(),type,size from entries;" testdirdup/testdir

ls -l outq\*

-rw-r--r-- 1 ggrider staff 78 Dec 10 20:19 outq.0

-rw-r--r-- 1 ggrider staff 181 Dec 10 20:19 outq.1

cat outq.0

testdirdup/testdir/c|f|0|

testdirdup/testdir/c|f|0|

testdirdup/testdir/c|f|0|

cat outq.1

testdirdup/testdir|f|4|

testdirdup/testdir|f|0|

testdirdup/testdir|f|0|

testdirdup/testdir|f|0|

testdirdup/testdir|f|0|

testdirdup/testdir|f|79655|

testdirdup/testdir/c/cc|f|14252|

As you can see you get your output into one file per thread. Combining these results is trivial such as cat outq.\* > myoutput

Another very powerful capability is to produce an output database per thread. This allows you to do queries and produce subsets of information that you can then run further queries on the result set. This is particularly useful when you want to use order by or group by clauses across all results.

Query to output databases requires the use of the -I parameter to create the output table and you must use the insert into SQL statements to populate the output database.

This query looks at all files and counts the number of files and links and sums the bytes in all the files and links per directory. It uses the -I for creation of the output databases and -O for the path base name to the output databases

rm outdb\*

../bfq -n 2 -O outdb -I "create table sument (uid int64, type text, totfiles int64, totsize int64);" -E "insert into sument select uid, type, count(\*), sum(size) from entries group by type;" testdirdup/testdir

ls -l outdb\*

-rw-r--r-- 1 ggrider staff 8192 Dec 10 20:29 outdb.0

-rw-r--r-- 1 ggrider staff 8192 Dec 10 20:29 outdb.1

As you can see it created two outdb databases. The following sqlite3 utility confirms there is a table sument in both and one database has 2 records, one for each directory processed by that thread with total number and total size of files in each directory. The other database processed one directory

sqlite3 outdb.0

SQLite version 3.24.0 2018-06-04 14:10:15

Enter ".help" for usage hints.

sqlite> .tables

sument

sqlite> select \* from sument;

2078|f|6|79659

2078|f|1|14252

sqlite> .quit

sqlite3 outdb.1

SQLite version 3.24.0 2018-06-04 14:10:15

Enter ".help" for usage hints.

sqlite> select \* from sument;

2078|f|3|0

sqlite> .quit

### querydbn

To deal with the fact that you have multiple output databases, the querydbn utility is available that glues the output databases together and union’s the tables together to allow you to query the output databases as if they were one database. This is very powerful as you can then do any query you want on the result set. Also you can see there are 4 variables, uid, type, totfiles, totsize as the bfq query instructed to be produced.

In this invocation of querydbn, we tell it to list headings and records, attach to the two databases and select \* from vsument. The union of all the databases/tables is a view named v”table” so in this case sument tables are unioned into vsument.

../querydbn -NV outdb 2 "select \* from vsument;" sument

processing query name outdb numb dbs 2

ATTACH 'outdb.0' as outdb0

ATTACH 'outdb.1' as outdb1

sqlu: create temp view vsument as select \* from outdb0.sument union all select \* from outdb1.sument;

after union running select \* from vsument;

uid|type|totfiles|totsize|

2078|f|6|79659|

2078|f|1|14252|

2078|f|3|0|

query returned 3 records

You can see it put the two databases/tables together and ran the query over the entire result set.

Of ourse you might want to do group by if you were producing a summary per user or something. In this case we just use order by to show you can sort the result set on totfiles.

../querydbn -NV outdb 2 "select \* from vsument order by totfiles;" sument

processing query name outdb numb dbs 2

ATTACH 'outdb.0' as outdb0

ATTACH 'outdb.1' as outdb1

sqlu: create temp view vsument as select \* from outdb0.sument union all select \* from outdb1.sument;

after union running select \* from vsument order by totfiles;

uid|type|totfiles|totsize|

2078|f|1|14252|

2078|f|3|0|

2078|f|6|79659|

query returned 3 records

As instructed, it sorted by totfiles. Yes this is a contrived example but this is an extremely powerful concept where you want to group/order results or just further query/reduce etc.

### 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 retieve 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.

An example of how one might want to use this powerful feature is to connect an input database to join with on queries within each directory. To do that you might in the -T string put in an attach database command and then create temporary view command. Then the -S or -E strings you could query that view.

Another example is that you could generate new tables within the GUFI infrastructure or elsewhere on a per directory basis or overall that you use insert into using attach and insert into commands.

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.

### querydb

Another useful utility is querydb which allows you to run a query against just one directory’s GUFI db

../querydb -NV testdirdup/testdir/c/cc "select \* from entries"

processing query name testdirdup/testdir/c/cc

name|type|inode|mode|nlink|uid|gid|size|blksize|blocks|atime|mtime|ctime|linkname|xattrs|crtime|ossint1|ossint2|ossint3|ossint4|osstext1|osstext2|

bfindex|f|8602501846|33261|1|2078|20|14252|4096|32|1543878081|1521589127|1543586004||com.apple.quarantine0081;5bbb7939;Firefox;DEFCFB61-7DB3-4D22-A53B-D815CE6F9C69|0|0|0|0|0|||

query returned 1 records

As you can see we just did a query against the testdirdup/testdir/c/cc/db.db entries table.

## bfresultfuse

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

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

## GUFI\_find (under construction)

This is where we put what you are trying to do and how to run GUFI\_find – Jason can fill this in?

# Administration Guide

## The Tree Summary Concept

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. See the Compound Query section for details.

bfti is the command used to place a tree summary record into a GUFI database. It is recommended that the GUFI administrator pick some level within the GUFI tree to keep tree summary records, perhaps at the top of the tree for each user to summarize that user or project. If updates occur to the GUFI tree in the directories below where the summary record has been written, you must delete that record and reproduce it using bfti.

Bfti is typically very fast as it just walks the tree and processes the directory summary record for every subdirectory and then writes a new tree summary record.

This is how bfti is run on our example GUFI tree using 2 threads and printing the directories as they are processed and totals

../bfti -n 2 -Ps testdirdup/testdir

testdirdup/testdir d 8602818121 1537503405 16877 4 2078 20 128 4096 0 1544500208 1544471971 1544471971 0

testdirdup/testdir/c d 8602818130 8602818121 16877 4 2078 20 128 4096 0 1544563611 1544470277 1544470277 0

testdirdup/testdir/c/cc d 8602818140 8602818130 16877 3 2078 20 96 4096 0 1544500214 1544469311 1544469311 0

totals:

totfiles 10 totlinks 0

totsize 187822

minuid 0 maxuid 2078 mingid 0 maxgid 20

minsize 0 maxsize 79655

totltk 8 totmtk 2 totltm 10 totmtm 0 totmtg 0 totmtt 0

minctime 0 maxctime 1543586004

minmtime 0 maxmtime 1542940736

minatime 0 maxatime 1543878081

minblocks 0 maxblocks 160

totxattr 10

mincrtime 0 maxcrtime 0

minossint1 0 maxossint1 0 totossint1 0

minossint2 0 maxossint2 0 totossint2 0

minossint3 0 maxossint3 0 totossint3 0

minossint4 0 maxossint4 0 totossint4 0

totsubdirs 3 maxsubdirfiles 6 maxsubdirlinks 0 maxsubdirsize 79659

../querydb -NV testdirdup/testdir "select \* from treesummary;"

processing query name testdirdup/testdir

totsubdirs|maxsubdirfiles|maxsubdirlinks|maxsubdirsize|totfiles|totlinks|minuid|maxuid|mingid|maxgid|minsize|maxsize|totltk|totmtk|totltm|totmtm|totmtg|totmtt|totsize|minctime|maxctime|minmtime|maxmtime|minatime|maxatime|minblocks|maxblocks|totxattr|depth|mincrtime|maxcrtime|minossint1|maxossint1|totossint1|minossint2|maxossint2|totossint2|minossint3|maxossint3|totossint3|minossint4|maxossint4|totossint4|rectype|uid|gid|

3|6|0|79659|10|0|0|2078|0|20|0|79655|8|2|10|0|0|0|187822|0|1543586004|0|1542940736|0|1543878081|0|160|10|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|

query returned 1 records

As you can see the entire tree is summarized with all these statistics for all subdirs and a record is placed into the tree summary database in testdirdup/testdir/db.db

## Other Administrative Tools

In general, most of the GUFI commands and basically all the scripts and tests can be useful in administering a GUFI tree/set of GUFI trees.

* bfwreaddirplus2db and bfwi are of course primarily used for loading/incrementally updating GUFI trees.
* bfq while primarily a query tool can be helpful for query for administrators as well as given bfq queries can produce output or even update GUFI databases itself (see compound queries above for inspiration on this topic), bfq can be a very powerful administration tool
* bfti is a valuable tool for creating tree summaries which are described above in the bfti command section, in the bfq command section, and below in the usage of tree summaries section.
* dfw is potentially a useful tool to just have a good source code implementation of a depth first walker
* querydb is useful for looking at individual GUFI databases within one directory
* querydbn is very useful for combining result databases into one queryable database/table
* make\_testdirs is useful for making a test directory to play with
* The test and scripts directories are very helpful in learning how to make GUFI do useful things
* gitest is particularly useful when designing an incremental update of GUFI from a storage system

The Gufi tree is just a POSIX tree so can be backed up, tar’d, etc. trivially. As such, it also can be be exported via NFS or other, etc.

Location of GUFI-tree:

bfwi will re-create the dir-path of <input\_dir> underneath <to\_dir>.

For example, if <input\_dir> is /a/b/c and <to\_dir> is /q/r/s, the GUFI-tree will

be created at /q/r/s/a/b/c.

This would be the path to provide to other commands that take a

GUFI-tree as input. We have debated whether the GUFI-tree should just be build at /q/r/s/c. That may come in a future release. We're opento discussion.

If you are putting your GUFI-tree index right into the source tree, you must follow the same syntax where input\_dir is the top of the src tree and -t to dir is the directory above the source tree. This keeps the same syntax for where you want your gufi tree built.

Bfwi will recognize that the to dir and input dir are the same real paths and if so will save some work on creating output directories, setting the mode of the output directories, and will ignore the database file (DBNAME defined in bf.h default db.db) when building the index. This way indexes built in the source tree and in another tree will create the exact same 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.**

## Full GUFI Load from a source file system tree

This section describes how one would run a full gufi load from a scratch. As you can see below in figure 3, extracting the metadata

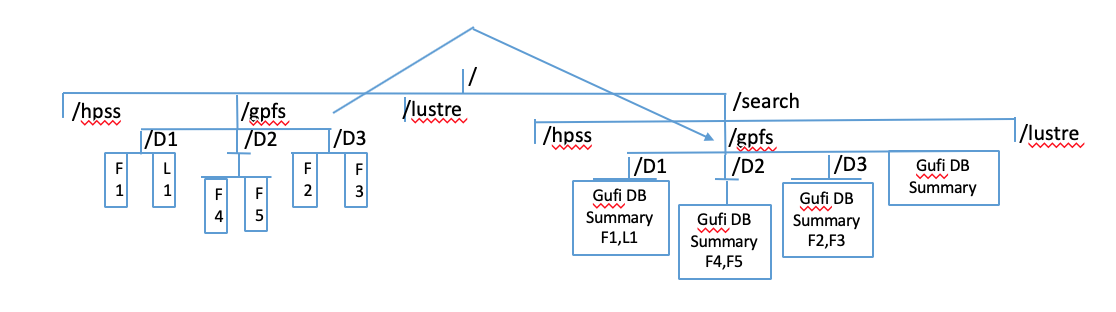


Figure 3 Multi-threaded Full Load of GUFI from POSIX file system using pure POSIX access methods

from the source POSIX tree into the GUFI tree, the file/link entries and directory summary information is put in GUFI databases per directory. To accomplish a full GUFI load you would:

* Remove /search/gpfs tree
* Run bfwi with input tree /gpfs and output tree as /search specifying if you want xattrs processes or not and number of thread for breath first threading walk
* Process will be parallel via breadth first walk/gufi index create threads

You also can use bfwi to place the GUFI index directly into the source file system. The name of the database files is a #define in bf.h if you don’t like db.db as the name.

This is a single process multi-threaded example of a full

This is ls -lR information about the 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

This command was run to produce the testdirdup/testdir GUFI tree

../bfwi -n 2 -xP testdirdup testdir

The following testdirdup/testdir GUFI tree with ls -lR

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

As you can see the GUFI tree contains a GUFI database at each directory called db.db.

The following diagram in figure 4 depicts a multi-process and multi-thread method of doing a full GUFI load from a source storage system using POSIX access methods.

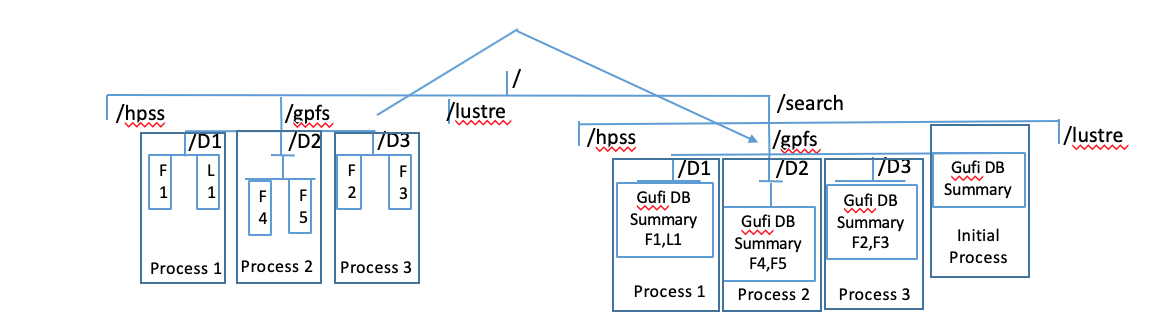


Figure 4 Multi-process and Multi-threaded Full Load of GUFI from POSIX file system using pure POSIX access methods

* Assumes /search is on a distributed file system or distribute in some way if the different processes are run on different nodes, otherwise its just multiple processes on a single node where each process is itself threaded.
* Delete /search/gpfs
* Run bfwi with input tree /gpfs and output tree as /search specify do not descend to create Gufi DB at that level. This produces a GUFI database for the top level.
* For each of /d1 /d2 /d3
  + Run bfwi with input tree /gpfs/d[1,2,3] and output tree as /search specifying if you want xattrs processes or not and number of thread for breath first threading walk
* Process will be parallel via processes/nodes and within each process via breadth first walk/gufi index create threads

This multi-processed and multi-threaded approach should scale as widely as your name space is.

## Full GUFI Load from a flat file

The bfwi program can not only build a full GUFI tree from walking a source file system tree, but it also has two other functions. It can write an output file per thread with path, type, attributes, linkname, and xattrs. The third function bfwi has is to build a GUFI tree from a flat input file. The input file format to use bfwi with the -u option which is the build GUFI from a flat file mode is as follows:

* Overall, the format has a directory record followed by all the file/link records for that directory, then another directory record followed by its children, etc.
* The record format is as follows:
  + Pathname, type,inode, mode, nlink, uid, gid, size, blksize, blocks, atime, mtime, ctime, linkname, and xattrs.
  + Each field is separated by a delimiter of your choice including a delimiter at the end of the record followed by a newline

There are many ways of producing these input files including writing your own flat file producing code.

There is a way to use flat file mode using bfwreaddirplus2db and the GPFS inode scan program that is described below in the full load for GPFS section below. The flat file produced is fed into bfwi -u to create the GUFI. Bfwreaddirplus2db outputs a file that has path, inode, type, and a sort field to assist in sorting for bfwi -u load from flat file.

Another way to produce a flat load file is to use bfq and do a query with the above format and then that file can be taken to load a different GUFI with bfwi -u.

bfwi itself can produce flat files, one per thread by walking a source file system tree. bfwi -n 2 -d '|' -x -Pp -o loadfile topofsrctree is an example that will create output into afiles loadfile.0 and loadfile.1 (one per thread) and it will write pathname, attrs, linkname, and xattrs (signified by the -x). The -Pp will print both directory and file/link records and it puts the records in the proper order directory followed by file/link if you use the above syntax using the | as a delimiter. The output files can be fed back into bfwi (either two different instances of bfwi or you can concatenate the files together using cat loadfile.0 loadfile.1 > loadfile and use one instance of bfwi by using bfwi syntax like this: bfwi -n 2 -d '|' -x -b -t newgufitop -u loadfile. This will produce a GUFI tree at newgufitop directory. As you can see you use the same delimiter to create the file and to consume it. The test python script bfwiflat2gufitest is a working example of this load from flat file using bfwi output.

dfw also can produce a file that can be used by bfwi -u to build a GUFI from a flat file. This is a single threaded depth first walker that does not require Sqlite to be installed on the computer where it runs, so it might be useful for building a flat file on one machine and sending the flat file to another to be converted into a GUFI. The syntax for dfw to produce a file for this is dfw -s -x -d '|' -l d0 > load.out. It uses the delimiter | in this case and produces a file load.out. Since this is a depth first walk, it does not put the records in the proper order for bfwi -u to load them, they must be sorted. For this reason dfw adds sort fields on the end of each record to making sorting easy and bfwi -u loading from the file doesn’t care if there are extra fields at the end of the record. In this case the added fields are fields number 17 (path) and 18 (type). You can use x sort -t '|' -k17,17 -k18,18 -o load.srt load.out to sort the file properly and create load.srt as the file used to load the GUFI tree using bfwi -u. The syntax for loading the GUFI from flat file is same as above, bfwi -n 2 -d '|' -x -b -t %s -u load.srt . The python script dfw2gufitest is a working example of this load method.

Yet another way to create a load file is the same as the dfw example above except you can use the find command to do the walking.

The find command can run dfw in don’t descend mode to print out a record for everything it finds. The syntax for the find command could be something like this find d0 -exec dfw -s -x -d '|' -l -D {} \; >> load.out. Just as in the above example, the file is not sorted properly so you use the same sort technique as above. Remember that the find command must create a directory record for every directory so you must run a find command that will do that.

The python script dfw2gufitest is a working example of this load method.

The same multi-process/multi-threaded approach should work fine, so you can have multiple flat files that load into the same GUFI and as long as the entire tree (every directory) is present in the files and the format is proper and no duplicate records exist, multiple bfwi -u loads can happen in parallel into the same GUFI. This is the same as above using GUFI to load from a source tree in parallel by doing a name space decomposition. You could do that same thing with bfwi or dfw or other methods for making multiple flat files to load from. This multi-processed and multi-threaded approach should scale as widely as your name space is.

## Full GUFI load into source tree

Full loading of a GUFI tree directly into the source tree is a special and simpler case. As always one can use multi-process/multi-threaded approach using name space decomposition. There are at least two ways of accomplishing a full load of a GUFI tree into the source tree.

The first is to use bfwi to build the gufi. You must use the -b and -t to\_dir flags and the -t flag has to be the directory above the source tree and of course the input\_dir is the top of the source tree. If the real paths of the top of the source tree and the to\_dir/input\_dir are the same then GUFI will do an efficient load of a GUFI tree into a source tree. It should e possible to even use the flat file feature with the correct pathing within the load file although that might be rare. This can be seen in the correctness test:

cbfwiinsrc – correctness bfwi created gufi in the src tree compared with bfwi created gufi elsewhere compared with bfq from both gufi trees.

The second way to do a full load of a GUFI directly into the source tree is to use bfwreaddirplus2db with the -Y flag telling it to make all directories suspect and the -b flag telling it to create a GUFI db and it will be in the source tree as that is the only function bfwreaddirplus has for creating GUFI trees. Another way would to not use the -Y flag but provide a last run time stamp of zero which would mark every directory as suspect but that would be less efficient. This can be seen in the following correctness test:

cbfwrp2dbfullinsrc – correctness bfwreaddirplus2db full gufi db create in source tree compared to bfwi create of full gufi db not in source tree

## **Incremental GUFI** Load

The incremental load mechanism can depend on the source storage system being loaded from. This section describes the modes of incremental that can be done with mostly GUFI provided tools and POSIX methods. In Figure 5 below the basis for the incremental update of a GUFI Tree Index is described via Tree Snap shots of directory only information and suspect directory information. As you can see this mechanism can update a GUFI Index without having to completely recreate it. Tools like bfwreaddirplus2db and other mechanisms for producing source storage system changes can be used to produce this incremental update capability.

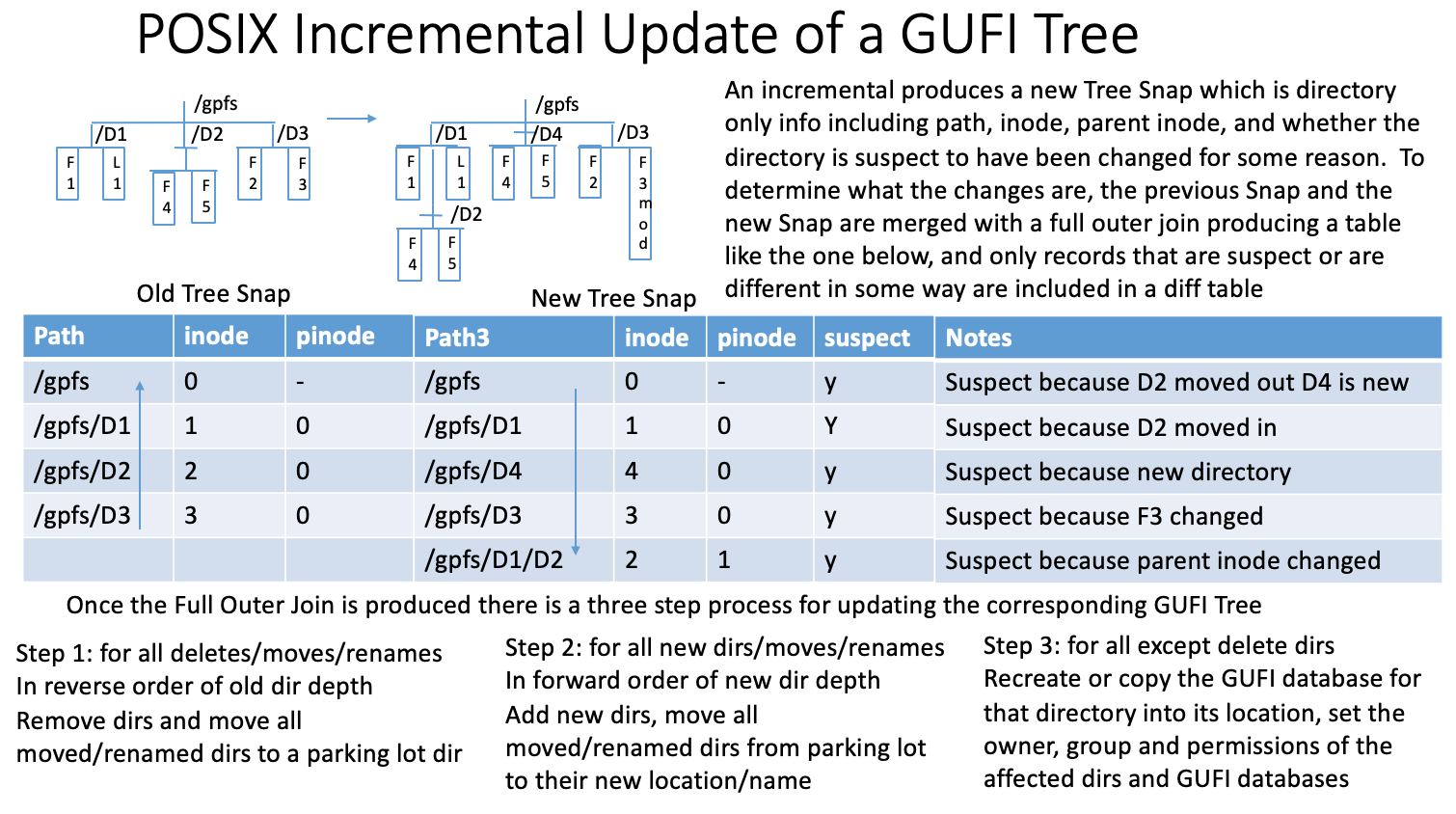


Figure 5 POSIX Incremental Update of a GUFI Tree

The primary tool for creating Tree Snaps of directory information with suspect directory information is bfwreaddirplus2db.

bfwreaddirplus2db -R -n 2 -A 3 -O incrsnapdb -c timelastrun -x -t stagingarea topofchangedsrctree

-R inserts directory info into the output database incrsnapdb

-O is the name of the output snap database (one per thread)

-c provided the time the last full/incremental was run for comparison

-x indicates that xattrs are desired

-t is a staging area where gufi databases for suspect directories can be stored (using their inode as the name of the db file)

-A 3 is a suspect mode – 3 causes suspects to be determined by bfwreaddirplus2b to stat all directories and files to determine what is suspect

Bfwreaddirplus2db is pointed at the top of the src tree after changes have occurred.

In this case bfwreaddirplus2db will walk in parallel the tree, stat all directories and files/links and produce a directory only info tree snap database (one per thread) and it will contain all the directories, inode, parent inode, and suspect if anything about the directory or its contents have changed. Also, bfwreaddirplus2db will produce a GUFI database file for each directory that is suspect and place that database file in the -t staging area directory with the directories inode as name of the database file.

To produce the “diff” database table from yesterdays tree snap database and todays incremental tree snap database, the following is done (this uses python) for convenience and this all exists in gitest in the test subdirectory.

#this example is 2 threads for yesterday and two threads for today snaps, so you have to attach yesterday and today databases

fd = open("incrdiff", "a")

fd.write("attach \'fullinitsnapdb.0\' as full0;\n")

fd.write("attach \'fullinitsnapdb.1\' as full1;\n")

fd.write("attach \'incrsnapdb.0\' as incr0;\n")

fd.write("attach \'incrsnapdb.1\' as incr1;\n")

# create temp yesterday and today views incrs via union for all thread databases

fd.write("create temp view b0 as select \* from full0.readdirplus union all select \* from full1.readdirplus;\n")

fd.write("create temp view b1 as select \* from incr0.readdirplus union all select \* from incr1.readdirplus;\n")

#create temp view thats a full outer join the sqlite3 way

fd.write("create temp view jt as select a0.path a0path,a0.type a0type,a0.inode a0inode,a0.pinode a0pinode,a0.suspect a0suspect,a1.path a1path,a1.type a1type,a1.inode a1inode,a1.pinode a1pinode,a1.suspect a1suspect from b0 as a0 left outer join b1 as a1 on a0.inode=a1.inode union all select a0.path a0path ,a0.type a0type,a0.inode a0inode,a0.pinode a0pinode,a0.suspect a0suspect,a1.path a1path,a1.type a1type,a1.inode a1inode,a1.pinode a1pinode,a1.suspect a1suspect from b1 as a1 left outer join b0 as a0 on a1.inode=a0.inode where a0.inode is null;\n")

#this produced a full outer join as described in the Figure 5 above

# create the output diff table which is only records of suspect or directories that changed in some way

fd.write("create table diff (a0path text,a0type text,a0inode int(64),a0pinode int(64),a0suspect int(64),a1path text,a1type text,a1inode int64,a1pinode int(64),a1suspect int(64),a0depth int(64) ,a1depth int(64));\n")

# load the diff table from a query of all differences and build depths

fd.write("insert into diff select \*,(length(a0path)-length(replace(a0path ,'/','')))/1 as a0depth,(length(a1path)-length(replace(a1path ,'/','')))/1 as a1depth from jt where a0pinode!=a1pinode or a0path!=a1path or a0inode is null or a1inode is null or a1suspect=1;\n")

fd.close()

#build the diff db

os.system('sqlite3 diffdb \'.read incrdiff\'')

# So we now have a diffdb sqlite3 database with a diff table that is a full outer join of yesterdays and todays snap db for only the records that indicate changes have occurred.

The next part of the processing goes through the three steps indicated in Figure 5. Again, this all exists in gitest in the test directory in the gincr() routine which does the entire incremental process as described in this section. Each step requires a query from the diff table ordered properly for the step and you just process each type of action within each step as described.

Step 1:

for all deletes/moves/renames

In reverse order of old dir depth

Remove dirs and move all moved/renamed dirs to a parking lot dir

Query to produce these records ordered properly

proc = subprocess.Popen('sqlite3 diffdb \'select a0path,a1path,a1inode,case when a1path is null then 1 else 0 end from diff where a1path is null or a0pinode!=a1pinode or a0path!=a1path order by a0depth desc;\'',stdout=subprocess.PIPE,shell=True)

Step 2:

for all new dirs/moves/renames

In forward order of new dir depth

Add new dirs, move all moved/renamed dirs from parking lot to their new location/name

Query to produce these records ordered properly

proc = subprocess.Popen('sqlite3 diffdb \'select a0path,a1path,a1inode,case when a0path is null then 1 else 0 end from diff where a0path is null or a0pinode!=a1pinode or a0path!=a1path order by a1depth asc;\'',stdout=subprocess.PIPE,shell=True)

Step 3:

for all except delete dirs

Recreate or copy the GUFI database for that directory into its location, set the owner, group and permissions of the affected dirs and GUFI databases

Query to produce these records

proc = subprocess.Popen('sqlite3 diffdb \'select a0path, a1path, case when a0path=a1path then 0 else 1 end, a1inode, a1pinode, case when a0pinode=a1pinode then 0 else 1 end from diff where a1path is not null order by a1depth asc;\'',stdout=subprocess.PIPE,shell=True)

**Other suspect methods besides method3**

Bfwreaddirplus2db can operate in other suspect modes.

Suspect mode 3 described above stats directories, files, and links to produce the outputs desired with suspects treated.

Suspect mode 2 is also useful in that if you have an out of band way to determine files/links that have changed, you can use -A 2 -W insuspectfile options in bfwreaddirplus2db. Suspect mode 2 you have to provide a file that has “inode type” on each line where type is f or l for file or link. Inode is the inode for the file. Some systems like GPFS and even TSM backed up file systems have the ability to produce lists of inodes that have changed in in an out of band way, so you can use utilities to produce the insuspectfile and bfwreaddirplus2db will use this method for suspect files/links instead of stating every file. Every directory will be stated but not ever file.

Suspect mode 1 is similar to suspect mode 2 except that no stating is done at all. In this case the options -A 1 -W insuspectfile are used and insuspectfile must be of the same format as suspect mode 2 but it would types would include d, f, and l where d is for directories. This method is not recommended for active file systems because of time differences between running a storage system centric utility to make the list of suspects and running bfwreaddirplus2db can cause directories to disappear causing problems.

Suspect mode 0 is to don’t bother with suspects. This will produce a directory only snap of the tree with path, inode, pinode, and suspect where suspect is always zero.

Another important feature of bfwreaddirplus2db is the -g option which will stride inodes into different output snap database files. This is useful if you are trying to merge inode/attribute/xattr information with path on inode for full loads etc.

## **Incremental GUFI** update of a GUFI tree directly within the source tree

The bfwreaddirplus2db can be used with the -b flag to tell it to create a GUFI tree which will be in the source tree because that is the only way bfwreaddirplus2db can create a GUFI tree, with no -T flag, and with any of the modes of determining suspect directories. With -b and proper suspect mode flags and suspect input files if needed as shown above, it can just replace suspect directory databases in place. Since there is no need to deal with moves/unlinks/renames of directories, this process is pretty simple, just run bfwreaddirplus2db with what ever suspect mode scheme you choose and it can just replace the suspect directory database

Bfwreaddirplus2db if using -b to put the gufi db directly in the source tree, the -c suspecttime flag is not used. The existence of a gufi db in a directory and the ctime on existing gufi db files are used to determine if something has changed in the directory or if its new and the gufi db is created or refreshed as needed making incremental updates trivial.

An example of this incremental in src tree gufi update can be seen in the correctness test cbfwrdp2dbincrinsrc.

## GPFS Special Full and Incremental Load (needs updated from Scott)

Of course, the POSIX full and incremental methods will work for GPFS including splitting up the name space and using process parallelism in addition to thread parallelism.

If you desire to use the GPFS ILM inode scan feature of GPFS to assist with the full GUFI load process, the below figure 6 describes how that can be done.

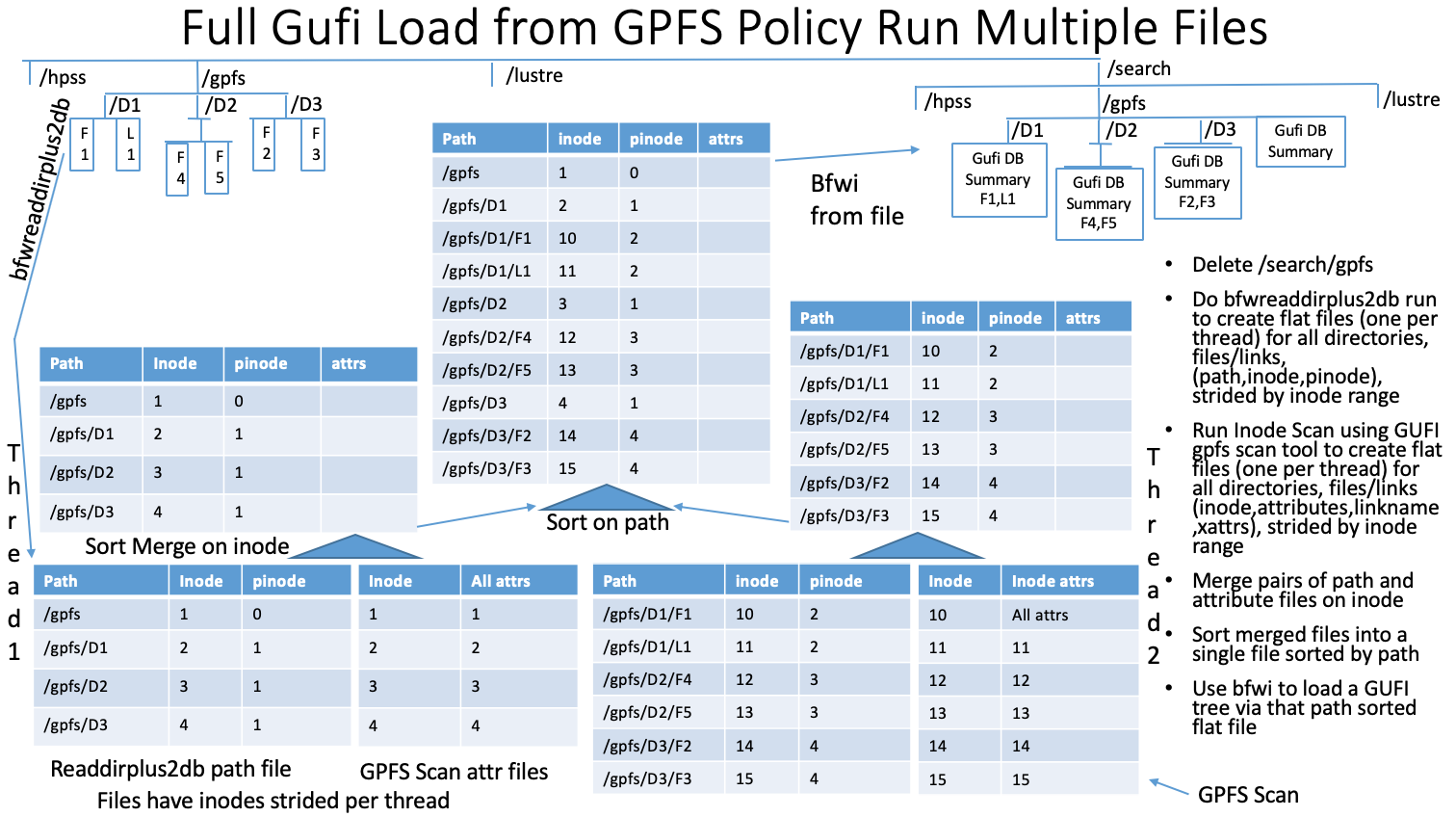


Figure 6 GPFS use of Inode scan for Full load to GUFI Tree

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 or other Backup System Special Incremental Load Assist

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 calle “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

## Lustre Special Full and Incremental Load (under construction)

This is where we put lustre full and incremental load

## HPSS Special Full and Incremental Load (under construction)

This is where we put HPSS full and incremental load

------------ this is just copied in to help with the HPSS secton ------------

Old HPSS to GUFI document can be stolen from once this function is working:

HPSStoGUFI – Loading a GUFI tree/index from an HPSS DB2 database

Background

Recall that GUFI places an sqlite database file per directory in which file and link information is placed in a GUFI directory tree (name, uid, gid, mode) that mimic’s source file systems (name, uid, gid, mode) thus getting all POSIX permissions security and fast tree walking from the underlying GUFI directory tree. The database file in each directory has several tables, the main three are the tree summary which if present summarizes the tree of directories/files/links below, the directory summary that summarizes the files/links in that directory, and the entries table which is where the details for each file/link in that directory are kept. Since the tree summary and the directory summary tables are just summarizations of the entries table we concentrate this discussion on creating/updating the GUFI tree itself and the entries table within each directory.

This is the table structure of the entries table which exists at each directory in the GUFI tree which mimic’s the source storage system.

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

Where:

Name – file/link/directory name (not path just name)

Type – director, link, or file

Inode – a unique identifier for this name space entry – related to source unique name space identifier

Mode – POSIX mode bits which describe type of entry and permissions

Nlink – number of links pointing to this name space entry

Uid and gid – uid and gid of the source entry

Size – size in bytes

Blksize – blksize for this file in the source storage system

Blocks – blocks for this file in the source storage system

Atime – last access time in the source storage system

Mtime – last file contents modification time in source storage system

Ctime – last metadata change time in the source storage system

Linkname – path to where this points to if if this is a link

Xattrs – extended attributes the source storage system allows you to see

Crtime – create time – not posix but many storage systems have this

Ossint1 – source storage system specific int

Ossint2 – source storage system specific int

Ossint3 – source storage system specific int

Ossint4 - source storage system specific int

Osstext1 - source storage system specific text string

Osstext2 - source storage system specific text string

The HPSS storage system’s metadata resides in a db2 database and has many tables. Most of the tables are not relevant to a GUFI tree index but all the tables are listed for completeness, and the tables that are important to the HPSStoGUFI discussion are listed at the bottom with the fields of interest

CREATE TABLE @schema.ACCTLOG, CREATE TABLE @schema, CREATE TABLE @schema.ACCTSUM, CREATE TABLE @schema.BFCOSCHANGE, CREATE TABLE @schema.BFDISKALLOCREC, CREATE TABLE @schema.BFMIGRREC, CREATE TABLE @schema.BFPURGEREC, CREATE TABLE @schema, CREATE TABLE @schema.BFTAPESEG, CREATE TABLE @schema.BFFILEHASH, CREATE TABLE @schema.DISKSEGUNLINK, CREATE TABLE @schema, CREATE TABLE @schema, CREATE TABLE @schema.NSACL, CREATE TABLE @schema.NSFILESETATTR, CREATE TABLE @schema.SSPVDISK, CREATE TABLE @schema.SSPVHISTORY, CREATE TABLE @schema.SSPVTAPE, CREATE TABLE @schema.STORAGEMAPDISK, CREATE TABLE @schema.STORAGEMAPTAPE, CREATE TABLE @schema.STORAGESEGAUX, CREATE TABLE @schema.STORAGESEGDISK, CREATE TABLE @schema.STORAGESEGDISKEXTENTS, CREATE TABLE @schema.STORAGESEGTAPE, CREATE TABLE @schema.STORAGESEGTAPEABSADDR, CREATE TABLE @schema.TAPESEGUNLINK, CREATE TABLE @schema.USERATTRS, CREATE TABLE @schema.VVDISK, CREATE TABLE @schema.VVTAPE, CREATE TABLE @schema.VVTAPEVERFCHKPT

HPSStoGUFI related tables/fields

BITFILE contains size of file, important timestamps to determine if migrations or other potential changes the file have been made, class of service, number of copies, and file family

CREATE TABLE @schema.BITFILE (

BFID CHAR (19) FOR BIT DATA NOT NULL,

BFATTR\_DATA\_LEN BIGINT,

BFATTR\_CREATE\_TIME INTEGER,

BFATTR\_MODIFY\_TIME INTEGER,

BFATTR\_WRITE\_TIME INTEGER,

BFATTR\_READ\_TIME INTEGER,

BFATTR\_COS\_ID INTEGER,

BFATTR\_STORAGE\_SEG\_MULT INTEGER,

FAMILY\_ID INTEGER,

)

IN BITFILE INDEX IN BITFILEI

NSOBJECT contains name, type, linkcount, inode which links to nstext table, parentInode,uid,gid, permissions for mode, create, modification, and last access times, and bitfile\_id which links to bitfile and bfdiskseg tables.

CREATE TABLE @schema.NSOBJECT (

TYPE CHAR (1) FOR BIT DATA,

LINK\_COUNT BIGINT,

PARENT\_ID BIGINT NOT NULL,

OBJECT\_ID BIGINT NOT NULL,

UID INTEGER,

GID INTEGER,

USER\_PERMS CHAR (1) FOR BIT DATA,

GROUP\_PERMS CHAR (1) FOR BIT DATA,

OTHER\_PERMS CHAR (1) FOR BIT DATA,

TIME\_CREATED INTEGER,

TIME\_LAST\_READ INTEGER,

TIME\_MODIFIED INTEGER,

NAME VARCHAR(255) NOT NULL,

BITFILE\_ID CHAR (19) FOR BIT DATA,

)

IN NSOBJ INDEX IN NSOBJI

NSTEXT contains two types of records we care about type=x’02’ (xattr) and type=x’03’ (linkname) which are linkname pointed to if this is a link and xattrs if present

CREATE TABLE @schema.NSTEXT (

TYPE CHAR (1) FOR BIT DATA NOT NULL,

OBJECT\_ID BIGINT NOT NULL,

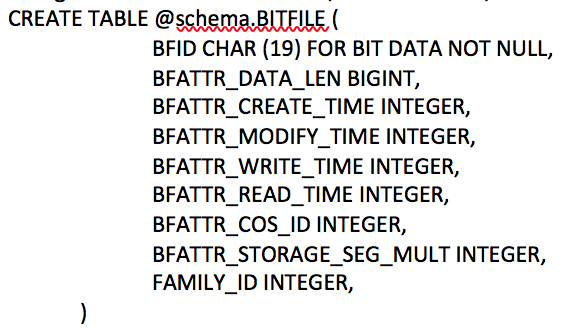
TEXT VARCHAR(1023)

)

IN NSTEXT INDEX IN NSTEXTI

BFDISKSEG tells us if the file is on disk presently in HPSS through the presence of disk segment records for the bitfile

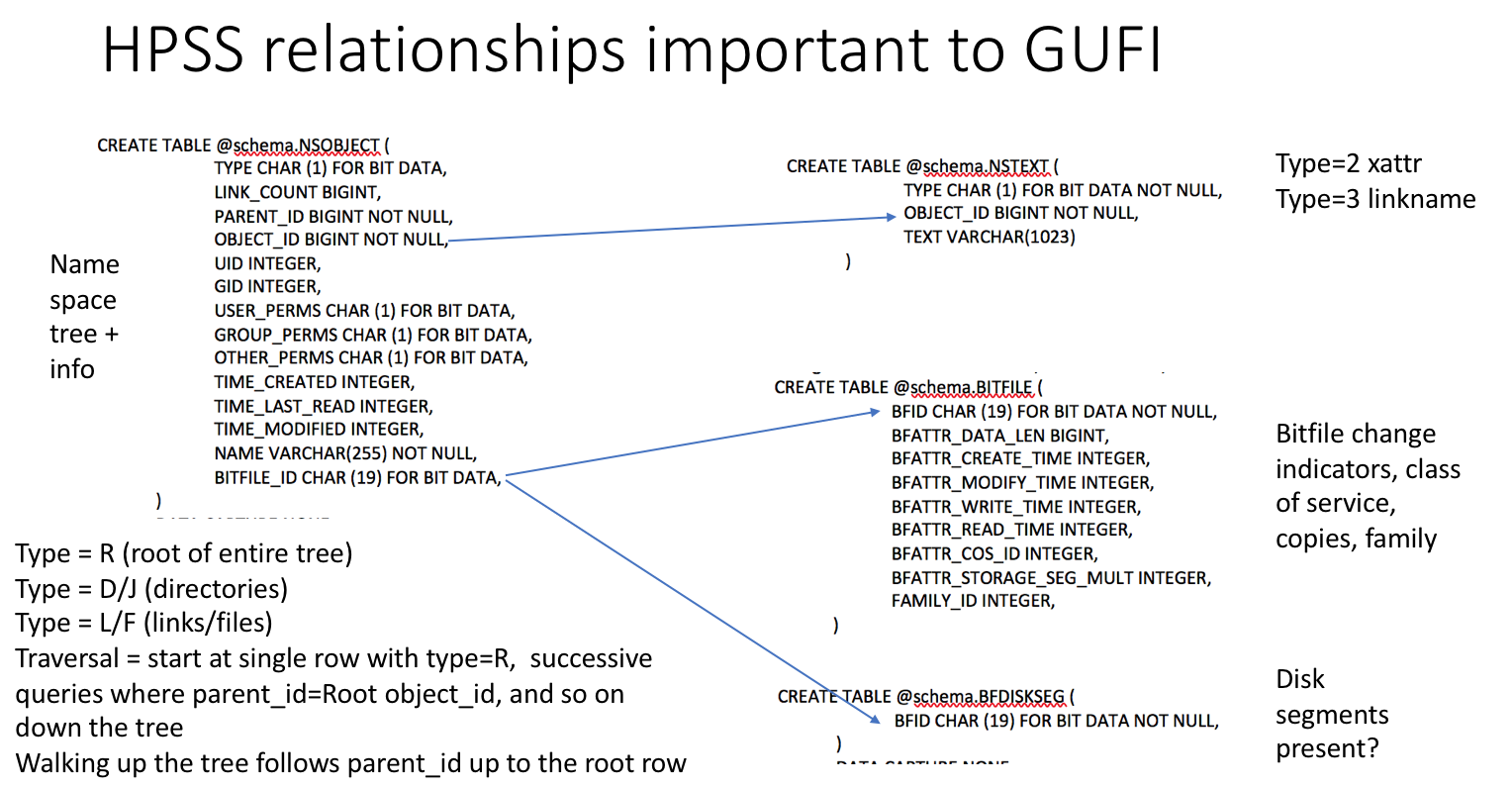
CREATE TABLE @schema.BFDISKSEG (

BFID CHAR (19) FOR BIT DATA NOT NULL, 

)

IN BFDKSG INDEX IN BFDKSGI

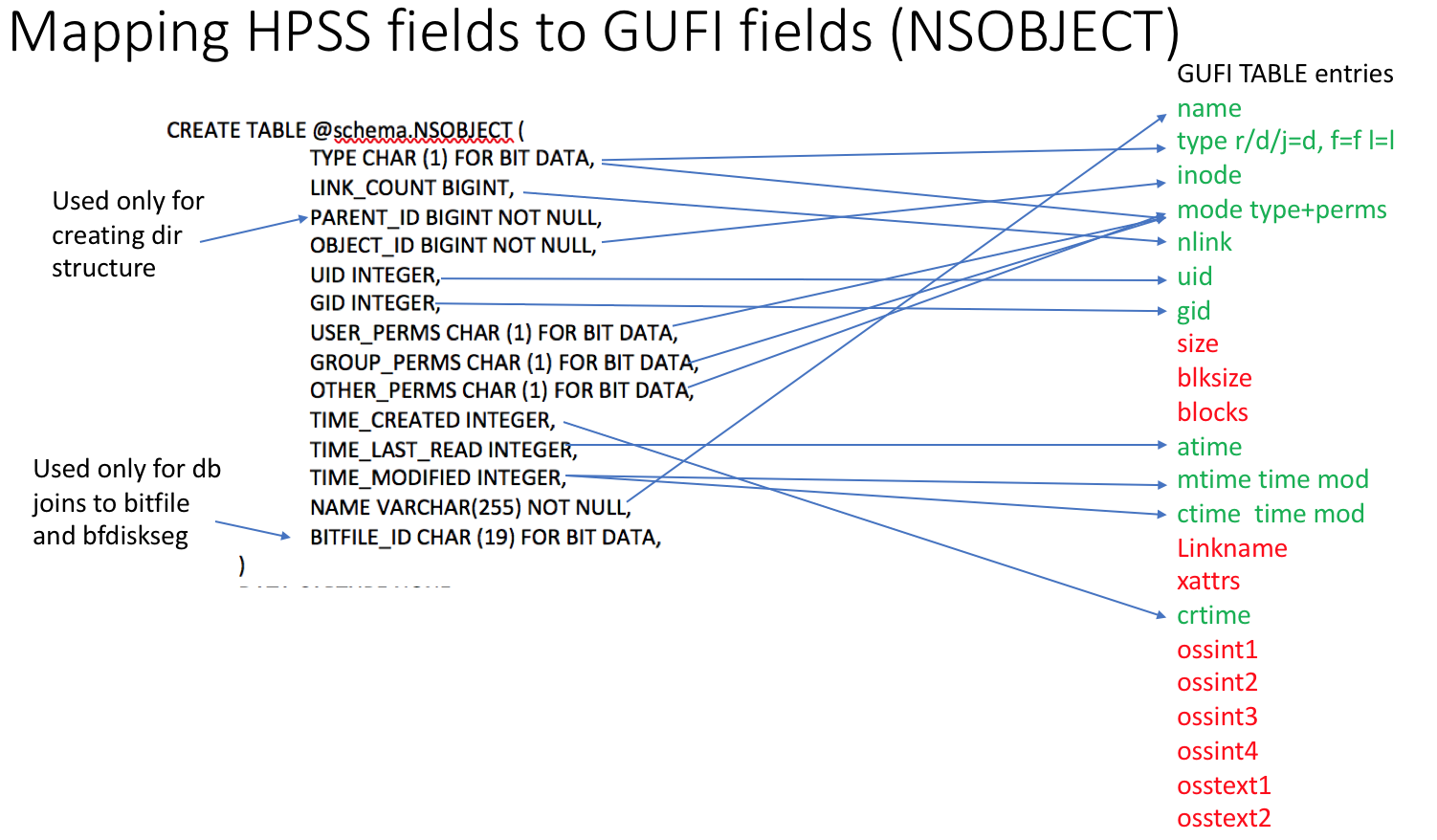
This diagram shows the relationships in HPSS that are important to the collection of the GUFI information.

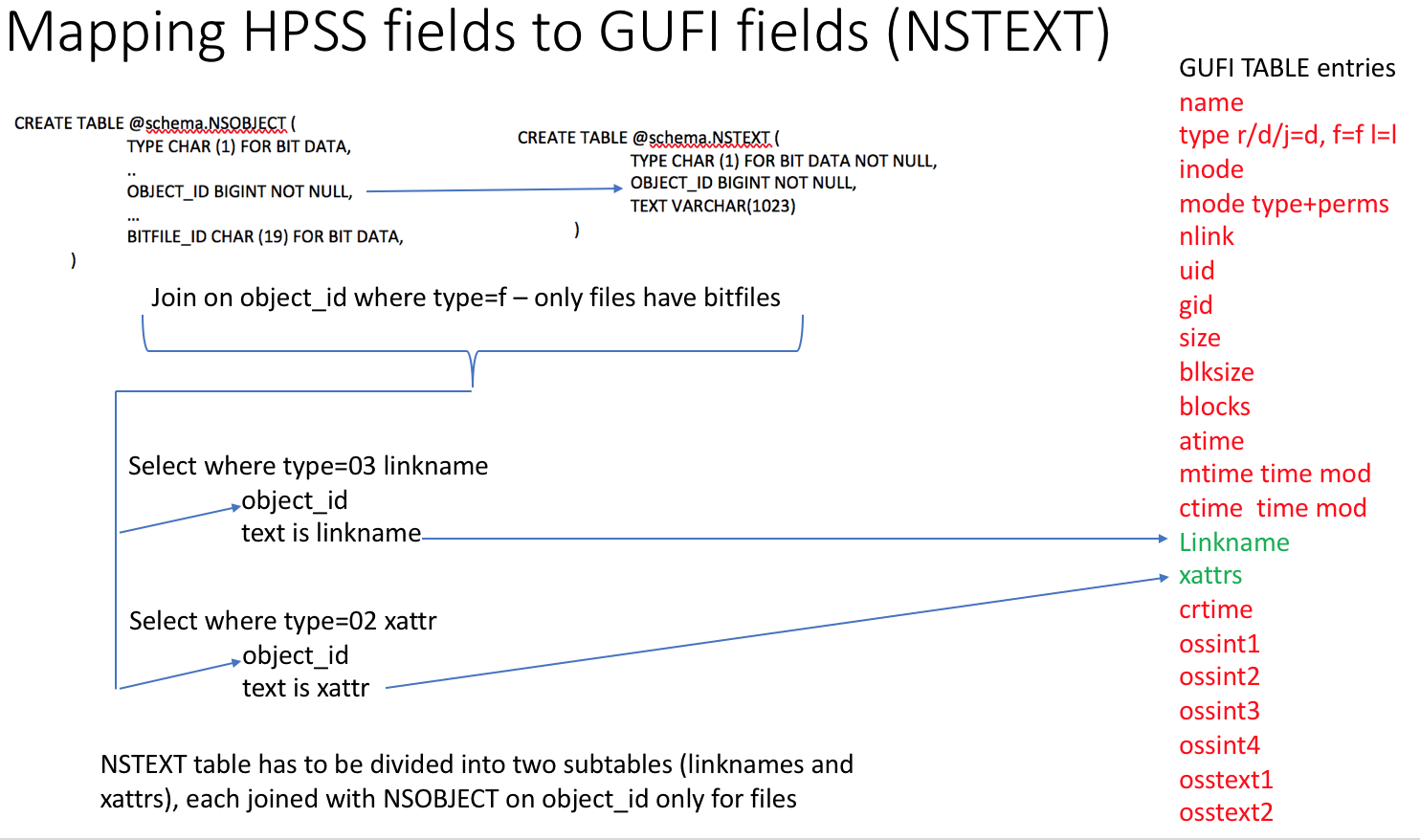


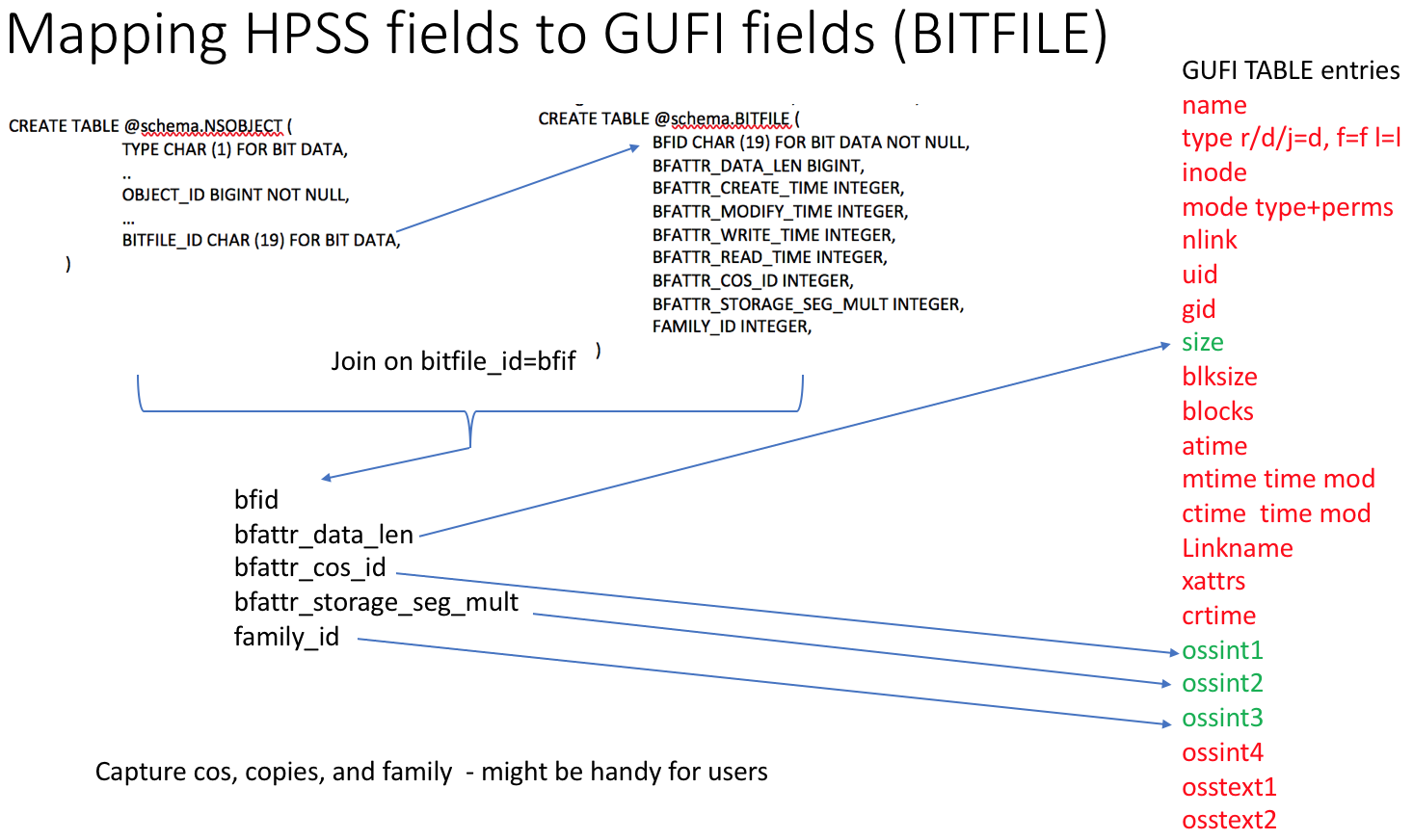
As you can see it takes many joins to connect the information needed to produce the GUFI index.

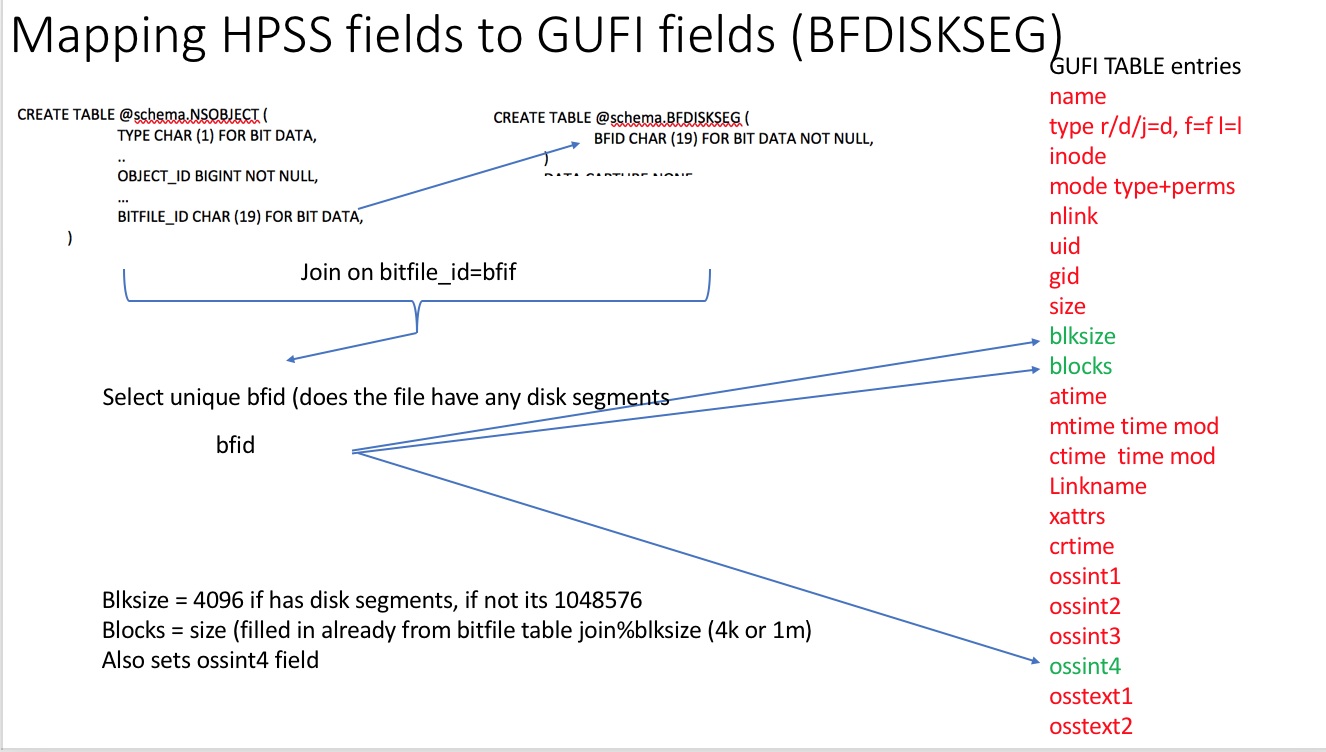
* NSOBJECT joined with (select only linknames from NSTEXT) on object\_id)
* NSOBJECT joined with (select only xattrs from NSTEXT on object\_id)
* NSOBJECT joined with BITFILE on bitfile\_id=bfid (to get size, cos, copies, family)
* NSOBJECT joined with (select distinct bfid from BFDISKSEG on bitfile\_id=bfid) to get on disk indicator
* So data comes from what is essentially 5 tables
  + NSOBJECT
  + NSTEXT linknames
  + NSTEXT xattrs
  + BITFILE
  + BFDISKSEG distinct bfid’s

Mapping of the HPSS fields into GUFI fields from the five tables occurs in the next set of diagrams:









HPSStoGUFI Full

The strategy for the full load from scratch of a GUFI from HPSS DB2 is to do joins/sorts to get one table that has all the GUFI entries table fields into one row, then walk that tree in parallel and when directories are encountered, put them on a queue (breadth first) and create the GUFI tree on the fly.

Currently the full HPSStoGUFI is implemented as three programs

Dumpbf.sqc – dumps the bitfile and bfdiskseg tables to one table one row per bitfile with only fields we care about. This creates an sqlite table called bf that has all the columns from the db2 tables bitfile and bfdiskseg that we need for GUFI

Dumbns.sqc – dumps the nsobject and nstext tables into one row per object with bitfile id used to join to the Dumpbf.sqc output, this creates an sqlite table called ns that contains all the columns from nsobject and nstext that we need for GUFI.G

prepnsbf – is a script that joins ns and bf on bitfile\_id and makes useful indexes

Bfhi.c – runs an sql query on nsbf and makes a gufi tree using thread per directory queries against the join.

Sqc is embedded sql in c programs so binding to db2 is needed

HPSStoGUFI Incremental

The strategy for the incremental update of a GUFI from HPSS DB2 is to use the fact that GUFI updates can be just directories that have had changes. The trick is figuring out what changed from last time the incremental was run and keep around as little information as possible for comparison. Tricky parts are related to figuring out how to handle directory moves and deletes of entire directories.

The following logic can be run every period to determine changes from the last period:

# this declares a temprorary table called session.todayo (this should be a persistent table)

# it contains all the information needed to capture the name space modifications

# and potential directories that should be re-indexed and for what reason

# in this example 1473873667 is the time of last incremental run

declare global temporary table session.todayo (pinode bigint, inode bigint, name varchar(255),type char(1), bfi smallint, nsi smallint, nsdi smallint) on commit preserve rows

# this part inserts records into session.todayo – for various

# reasons

insert into session.todayo (pinode,inode,name,type,bfi,nsi,nsdi)

# SBF is for each file

# that is indicated by a change that hasoccurred resulting from

# some kind of update to the bitfile (migration or other event)

# make a list of all these bitfile\_id’s

with SBF (SBFID) as (select BFID from HPSS.BITFILE where max(BFATTR\_MODIFY\_TIME,BFATTR\_WRITE\_TIME,BFATTR\_READ\_TIME,BFATTR\_CREATE\_TIME)>=1473873667),

# SBFN takes the list above (which is all the files (bitfiles) that

# had a bitfile related change and joins it with the NSOBJECT table on

# bitfile\_id which converts the list above into a list of distinct

# parent\_id’s (or directories) that are indicated having something

# that has changed in them due to bitfile changes

SBFN (SBFIDP) as (select distinct PARENT\_ID from SBF left inner join HPSS.NSOBJECT on SBFID=BITFILE\_ID),

# SNS makes a list of all the parent\_ids (directories) that have

# had a change due to a modified date in the NSOBJECT table

# ctime, mtime, atime like changes to the namespace entries

SNS (SNSID) as (select distinct parent\_id from HPSS.NSOBJECT where max(time\_created,time\_last\_read,time\_modified)>=1473873667 and (type=x'81' or type=x'82')),

# NSDIR is the tree info pinode, inode, name, typectime, rtime, mtime from

# NSOBJECT for all directorylike things

# so this is all the directories in the system

NSDIR (NSDIRPINODE, NSDIRINODE, NSDIRNAME, NSDIRTYPE,NSCTIME,NSRTIME,NSMTIME) as (select PARENT\_ID, OBJECT\_ID, NAME,type,time\_created,time\_last\_read,time\_modified from HPSS.NSOBJECT where type=x'84' or type=x'85' or type=x'86')

# this joins the above list of directories

# indicated by bitfile updates

# list of directories indicated by name space

# and list of all directories

# and creates entries that are inserted into session.todayo

# that have the tree (inode, pinode, name, type) and flags for

# if indicated by bitfile, or name space change

# this is kept to compare with the next incremental run

# it is the size of all the directories in the system, but not paths

# just name, inode, pinode, type, and some flags for update indicator

# you need the entire directory space so you can traverse todays

# snapshot of the namespace (you don’t need files/links, just dirs)

select NSDIRPINODE, NSDIRINODE, NSDIRNAME, NSDIRTYPE, case when SBFIDP is not null then 1 else 0 end, case when SNSID is not null then 1 else 0 end, case when max(NSCTIME,NSRTIME,NSMTIME)>=1473873667 then 1 else 0 end from NSDIR left outer join SBFN on NSDIRINODE=SBFIDP left outer join SNS on NSDIRINODE=SNSID

# this is just a select count of all the records in session.todayo

# which is just the number of directories

select count(inode) from session.todayo

# SSDIR just makes a list of all the directories that are indicated

# from the session.todayo table

with SSDIR (SSDIRPINODE,SSDIRINODE,SSDIRNAME,SSDIRBF,SSDIRNS,SSDIRNSD) as (select pinode, inode,name, bfi, nsi, nsdi from session.today where MAX(bfi,nsi,nsdi)>0)

# from the list above of indicated directories

# this pulls all the needed GUFI fields from NSOBJECT

# it pulls linknames from NSTEXT via a subquery)

# it pulls xattrs from NSTEXT via a subquery and a function

# hpss.cleanx() which is a c program user defined function

# that cleans out unwanted substrings in the xattr field

# this could be replaced with a regex if that part of DB2 was

# installed. It is not a trusted udf() so it is a bit slow

# but that doesn’t matter much as this is only the records that

# are from directories that are indicated

# it pulls needed fields from BITFILE via subquery

# it pulls On disk info from BFDISKSEG –

# using distinct bitfile\_id

# this query pulls all the info you will need to do directory

# refreshes for the indicated directories

Select NSO.TYPE,NSO.LINK\_COUNT,NSO.PARENT\_ID,NSO.OBJECT\_ID,NSO.UID,NSO.GID,NSO.USER\_PERMS,NSO.GROUP\_PERMS,NSO.OTHER\_PERMS,NSO.TIME\_CREATED,NSO.TIME\_LAST\_READ,NSO.TIME\_MODIFIED,NSO.NAME,(select text from hpss.nstext as NSTL where type=x'03' and NSO.OBJECT\_ID=NSTL.OBJECT\_ID),(select hpss.cleanx(text) from hpss.nstext as NSTX where type=x'02' and NSO.OBJECT\_ID=NSTX.OBJECT\_ID),'+;:-',NULL,SSDIRBF,SSDIRNS,SSDIRNSD from SSDIR left outer join HPSS.NSOBJECT as NSO on SSDIRINODE=NSO.OBJECT\_ID union select NSO.TYPE,NSO.LINK\_COUNT,NSO.PARENT\_ID,NSO.OBJECT\_ID,NSO.UID,NSO.GID,NSO.USER\_PERMS,NSO.GROUP\_PERMS,NSO.OTHER\_PERMS,NSO.TIME\_CREATED,NSO.TIME\_LAST\_READ,NSO.TIME\_MODIFIED,NSO.NAME,(select text from hpss.nstext as NSTL where type=x'03' and NSO.OBJECT\_ID=NSTL.OBJECT\_ID),(select hpss.cleanx(text) from hpss.nstext as NSTX where type=x'02' and NSO.OBJECT\_ID=NSTX.OBJECT\_ID),(SELECT concat(BFATTR\_DATA\_LEN,concat('+',concat(BFATTR\_COS\_ID,concat(';',concat(BFATTR\_STORAGE\_SEG\_MULT,concat(':',concat(FAMILY\_ID,'-'))))))) from HPSS.BITFILE where BFID=NSO.BITFILE\_ID),case when (SELECT distinct BFID from HPSS.BFDISKSEG where NSO.BITFILE\_ID=BFID) is not null then 1 else 0 end,SSDIRBF,SSDIRNS,SSDIRNSD from SSDIR left outer join HPSS.NSOBJECT as NSO on SSDIRINODE=NSO.PARENT\_ID where NSO.TYPE=x'81' or NSO.TYPE=x'82'

The above process creates three tables for each period

* DirectorySnap - A table with all the directories in the storage system from that day (inode, pinode, name, type (root or directory)
* DirectoriesSuspect - A table with all the directories that are indicated as having changes/new
* FilesLinksSuspect - A table with all the files/links in all the directories that are indicated

Once you have all the info above, the procedure for determining directories that have been deleted since the last period is

select \* from session.yesterday except select \* from session.todayo

we will call this the DeletedDirectorSnap

So you have DirectorySnap, DirectoriesSuspect, FilesLinksSuspect, and Deleted DirectorySnap

For each period

Given this information, the below is how you incrementally update the GUFI table with the incremental information. (worked this out with Jeff (feb 12 on the phone – we think it will work)

Pseudocode:

Create copy of session.yesterday.DirectorySnap called DirectoryInterim

Old idea:

Loop until all Suspect directories in session.today.DirectoriesSuspect and all directories in session.today.DeletedDirectorysnap have been satisified, by updating those tables with a complete flag column

Query session.today.DirectoriesSuspect using session.today.DirectorySnap to get effected path (path for a new directory, destination path for a move or rename, or path for an update) new directory is determined by looking at yesteterday.session.DirectorySnap to see if the directory is new, destination path for move is determined by changing parent inode, name change would be indicated by the directories name changed. Only retrieve the records that have not already been satisfied by previous iterations. Sort that list by destination effected path.

Take above list and try to process in order, if you cant process then you skip to the next record. Processing means you make the change to the GUFI tree (move, rename, update) and wipe out the sqlite table with summary and entries tables and remake those based on a query of session.today.FilesLinksSuspect where directory Inode=parent inode in the FilesLinksSuspect table. If you can make the change then you also update the DirectoryInterim table updating the directory record appropriately (if its name changed, if its parent inode changed). You also mark the session.today.DirectorySuspect with success by indicating this record has been satisfied. If for whatever reason you cant update the GUFI, then you skip to the next record in the is list

Query session.today.DeletedDirectorySnap using DirectoryInterim to get effected path (using the interim shows you where the directories to be deleted are right now in this process), you only pull records that haven’t already been satisfied from the last iterations. You sort that list descending on the number of slashes in the effected path.

You take that list and process in order, trying to delete the directory in GUFI. If you can delete it you also mark the session.today.DeletedDirectorySnap indicating this record is satisfied, you aslo delete the record out of DirectoryInterim so your current state is updated. You may not be able to accomplish the delete however and if so you just skip to the next record. Reason for not being able to delete the directory is that there are directories below that directory that till exist.

End loop

In the end, the DirectoryInterim (which was made by copying the session.yesterday.DirectorySnap and updating it with the necessary mods) should be the same as the session.today.DirectorySnap

We think this will converge and eventually get all the records processed. The reason for the iteration is due to operations high in the tree being dependent on operations low in the tree, like cant move something into the top of the tree because some set of operations have to happen lower in the tree first, like a delete of a directory branch to clear out the place where the move destination is, etc.

New idea:

1. Process all suspect directories that are renames/moves (new name/new parent\_id) by moving these directories to a “holding” directory in descending destination tree order using the session.today.DirectorySnap, session.yesterday.DirectorySnap, and session.today.DirectorySuspect tables, you may need to rename those directories you are moving to eliminate name collisions in the holding directory – but you have the destination name/path so renaming is not an issue – in fact it could be a help by using the inode as prt of the temporary name or similar. Update the gufi by moving the dirs and update the Interim table.
2. Process all deletes from the todays delete table (which came form an except operation between yesterdays directorysnap and todays directorysnap. Pull paths from interim table for those inodes, sort them in ascending path order (paths from interim table) and delete the dirs in gufi and delete them from interim
3. Process all the suspect directories that are new or changed in anyway (moving from holding directory to where they go using target name and creating new) in gufi and update the interim table

At the end you might want to compare the interim directorysnap and the todays directorysnap to ensure you match as a check.

Utilities

Some utilities to make this all easier have been written:

Cleanx() is a c routine that is a db2 user defined function, it is to clean up the xattrs and remove substrings we don’t want in the GUFI index. Db2 has a regex feature that could be used for this but it was not installed. It might be better to rewrite this in regex with the db2 feature installed. The udf runs slowly because its not a trusted udf, so it runs as a separate process somehow. We could try to make it a trusted udf as an alternative. Source is cleanx.c

The make script for making cleanx into an executable and defining the udf is in makecleanx

Walkup() is an sql function that takes an object\_id and walks up to get the full path of the object you are interested in. the source is in db2hpsspath

The incremental update HPSStoGUFI SQL is called hpsstodb2incr.db2

Currently the full HPSStoGUFI is implemented as three programs

dumpbf.sqc – dumps the bitfile and bfdiskseg tables to one table one row per bitfile with only fields we care about. This creates an sqlite table called bf that has all the columns from the db2 tables bitfile and bfdiskseg that we need for GUFI

dumbns.sqc – dumps the nsobject and nstext tables into one row per object with bitfile id used to join to the Dumpbf.sqc output, this creates an sqlite table called ns that contains all the columns from nsobject and nstext that we need for GUFI.G

prepnsbf – is a script that joins ns and bf on bitfile\_id and makes useful indexes

bfhi.c – runs an sql query on nsbf and makes a gufi tree using thread per directory queries against the join.

sqc is embedded sql in c programs so binding to db2 is needed

Special make scripts for dumpbf and dumbns are

makeitb and makeitn