Understanding Layout Schema

Authors: Rajesh Bhalerao <rajesh_bhalerao@xyratex.com>,
Trupti Patil <trupti_patil@xyratex.com>
Version: Draft (0.1)

- Understanding Layout Schema
  - Overview
    - Traditional file layout
      - Optimization using extents
    - File layout for snapshots
    - File Layout in distributed name-space
    - Lustre case
    - pNFS and file layouts
  - Design Goals
  - Colibri Layout Types
  - Layout Schema Interfaces
    - Creating Layouts
    - Assigning Layout To A File
    - Updating or Modifying Layouts
    - Searching Layouts
      - Search by storage object id
      - Search by file id
    - Query / Lookup Layout
    - Deleting a Layout
    - List Layout types
  - Layout Schema
    - UML/ER Diagram
    - Layout Tables
      - File Id to Layout Id Mapping
      - Table “layouts”
      - Tabular representation of “layouts” table with example
      - Table “composite_layout_extent_map”
      - Tabular representation of “composite_layouts_extent_map” table with example
  - Assumptions
  - Queries
  - Notes from discussion held on 11 Oct 2011
      - Colibri File Striping using Component Objects
      - COB identifier / COB id / COB fid
      - Colibri File Layout
      - Layout Formula

Disjoint sets
Layout policy, File Creation, Resource Manager
Reference count on a layout
Use Case of a file layout with formula: Reading from a file F0
Composite layout
Layout attributes - Number of data and parity units in a parity group (N, K)
Use Cases

Overview

The layouts of the files are relevant for a disk-based file system. This is one of the most important services (or functionality) provided by the disk-based file-systems. They manage the underlying storage and optimally utilize the storage using user-opaque layout of the file data blocks as well as meta-data. The user accesses the files using the system calls and does not have to bother about the disk block management. The file layout refers to the arrangement of the disk blocks that belong to a file. The layouts are designed by taking into consideration the optimum storage utilization and efficient access to the data.

Traditional file layout

Most of the disk-based file-systems access any file using meta-data for that file, famously known on UNIX as inode. Along with other attributes of the file, the inode stores the file layout. In a classic file-system, this file layout is also called as block map. A typical EXT2 block map (a file layout) comprises of

1. 10 direct block
2. 1 indirect block
3. 1 doubly indirect block
4. 1 triple indirect block

The traditional block map in an inode would contain logical disk block numbers. A block map for direct blocks is simply an array of block numbers.

Optimization using extents

The traditional EXT2 file layout (block map) was evolved to store extents or block clusters for a file in EXT4. This provided more compact storage for the file layout and also provided better sequential access to the file data. This layout started storing <starting block number, len> instead of storing only a block number within the file layout.

File layout for snapshots

With advent of 24x7 availability of the applications and a requirement to back-up the on-line file-system created a need for creating the snapshots of the file system. A snapshot of a file-system creates a point in time copy of all the files in a file system. If a file is modified after the snapshot

is created, there is need to store only the incremental changes. Duplicating the entire block map for the file would waste the storage space. This created a need to access the new modified blocks and old unmodified blocks through the file. The classical block map was again modified to create a block map chain that would point to old block map as necessary. (Not an EXT4 feature.)

File Layout in distributed name-space

In many organizations, filesystems are deployed using NAS boxes. As the storage demands grow, more NAS boxes are added. This creates lot of administrative overhead. To provide storage scalability, a single unified namespace, and easier administration, many solutions such as clustered NAS, scale-out NAS, file switches and file virtualization products were developed.

Many of these products developed their own layouts. The files were accessed using a head server with bunch of NAS boxes behind it. The files are accessed using a top-level layout within a single distributed name space. This layout would then map to the underlying filesystem. The top-level layout now points to the files on different underlying file-systems rather than a block map. These layouts are stored either as data or attributes of a traditional inode. The underlying (stand-alone) filesystem uses the classical block-map layout.

Lustre case

Lustre has two levels of mapping. The first level of mapping is stored on MDS as a file EA. The layout is structured as {{ost1,fid1}, {ost2,fid2}, {ost3,fid3}, {ost4,fid4}, … {ost-n,fid-n}}, along with stripe size. So the logical offset in a file can be calculated into one of the object represented by {ost-index, fid}. With this info, lustre client communicates to that ost, accessing the object with that specified fid. Another mapping from object logical offset to disk block is done by a local disk file system, ldiskfs (the origin of ext4).

pNFS and file layouts

In a traditional NAS deployment, there is a NAS server and multiple NFS or CIFS clients. With the advent of SAN, even the client machines started having direct access to the storage. To boost the performance of overall system, pNFS like systems were developed. The pNFS clients started requesting file layouts so that they could perform direct I/O (this does not mean unbuffered I/O) to the files. This created different kinds of requirements for the file-layouts. The layouts that are to be transported need to be

1. Compact (not occupying a lot of space/memory)
2. Possibly they need to be in network-byte order (pre-encoded) for easy transport
3. A part (region of a file) of the layout could be delivered if necessary

Design Goals

Looking at the history of the file-layouts, it's very clear that the file layouts evolve over a period of time. A host of copy services (snapshot, back-up, archiving, data de-duplication, replication etc) can also influence the file layout. In Colibri the layout will be transported between two servers.

The file layout will need a compact description so that it will consume minimal resources (memory, network bandwidth). Colibri should support multiple layouts. It should allow accommodation of new (future) layouts. It should allow migration of file data from one layout to another. The design goals for the file layouts (and schema) are listed below:

1. Support multiple layouts within the file-system
2. Provide support to add new layouts
3. Provide support for composite/mixed layouts (combining different layout types together)
4. Provide migration of file data from one layout to another
5. Provide compact description of the layouts
6. Provide pre-encoded layouts (where necessary) for easy transport

Colibri Layout Types

HLD for layout schema lists the following layout types:

1. SNS
2. Local RAID
3. RAID levels
4. Future layouts such as de-duplication, encryption, compression

Although these layout types are well classified from schema perspective, we believe there will be two types of layouts that will be supported for the planned demo:

1. Parity de-clustering layout (c2_pdclust_layout) (Conceptually this is 'SNS layout with parity de-clustering as a property' and is referred as pdclust layout.)
2. Composite layout (mixed layout for representing layout under repair)

Layout Schema Interfaces

Creating Layouts

Colibri will support certain number of layout types (e.g. pdclust, raid, composite). Creating a layout of any of the supported types is an administrative task. (Note: Need to figure out what is the user interface to create a layout.)

There are various parameters that will have to be considered while initializing any file layout. Some of the parameters are listed below:

- Policy (e.g. prealloc blocks?)
- Layout Type
- Layout type specific data (This may contain sub-maps)
- Backing store

This interface will make following assumptions:

- While initializing a layout, parameters like the ones mentioned above will be considered by another encapsulating task viz. "Layout". The task "Layout Schema" will not play any

role into that.

- Backing store objects will be created before making this call. Appropriate heuristics will be used while creating these objects. For example, for a RAID-like layout the storage objects for the stripe units will not be located on the same disk [The term storage object used here is different from Colibri storage object. The storage object referred here will map to component object of Colibri]
- Sub-maps (aka sub-layout) will be created before creating top level layout.
- Layout creation operation will fill in type-specific data in the type tables (or the fields in a record that are specific to a layout)

Assigning Layout To A File

A file layout is one of the attributes of the file. A *layoutid* is assigned to a file when a file is created and it is then stored in the file attributes as one of the properties of the file. In Colibri, a file will be created using a file create (or open) FOP. There are various parameters that may need to be considered while assigning layoutid to a file. Some of the parameters are listed below:

- Parent dir inheritance attribute
- Policy (e.g. prealloc blocks?)

A few assumptions regarding file create operation:

- If a policy such as storage pre-allocation is used, the block (blocks of backing store) reservation will happen first.
- Creation of a file requires creation of its component objects and the creator (a client, usually) must assure that cobs can be created (i.e., that free identifiers exist).
- File-id to layout-id mapping is stored by the fileattr_basic table (also called as fab).

Updating or Modifying Layouts

The layouts will be modified under the following conditions:

- The system administrator changes the layout properties
- The underlying storage of the layout is affected

These events will lead to a composite layout until the file data migrates to the new layout completely. The layout schema should provide interface to update the existing layout. Modifying the layout may result in a new layout.

The modification to the layout opens up following design related queries:

- Should update to the layout change the *layoutid*?
- Should this function generate a layout change notification?
- Should the modified layout be marked as invalid till old layout is dropped by all the servers using it?

Searching Layouts

The layout provides mapping of logical file block to corresponding logical storage block(s). In

many circumstances identifying layout using *fileid* is useful. While in some other conditions the inverse (or reverse) mapping from storage to files is useful. Hence the search function should provide flexibility to obtain *layoutid* (key to the layout) using different parameters (or mechanisms).

The query interface of the layout will be used to obtain the details of the layout structure where as search interface will provide only the *layoutid*.

Search by storage object id

When there is a back-end storage failure, it has to be marked into the database (Colibri meta-data). The layouts affected by the storage device will also have to be updated. To provide this functionality, searching the layout by storage id (inverse mapping) is useful. This type of interface is also helpful for the recovery IO (Colibri middleware).

Search by file id

This mapping will often be used by the client performing the IO on the file. When an IO is being performed on a file, a layout will be obtained by using *fileid*.

Query / Lookup Layout

A layout is queried using a *layoutid*. This function will fetch all the details of the layout. This query can either be covering the entire file or a region of the file. For compact layouts such as layout formulae, this will not matter.

This interface will be used by the clients for performing IO against a file.

Query for a file region or partial layout is out of the scope of layout schema.

Deleting a Layout

The HLD talks about holding a reference to a layout and decrementing the reference to the layout when a file is deleted. Although this idea is quite appealing, it's implementation in the schema is not clear at this point in time. Until this design and use-case becomes clear, we will simply delete a layout when the file (using this layout) is deleted.

List Layout types

Layouts are influenced by layout types. A blockmap style layout will contain array (tree) of all the storage blocks. A formula based layout will work with parameters (variables) of the formula. There will be one or more tables storing the layout type information.

During the initialization of the layout module, it will be necessary to load all the known layout types from the database. This will in turn help to create a layout for the file of a desired layout type using the layout type operations.

Layout Schema

UML/ER Diagram

This is yet to be done. But the following tables should give some idea.

Layout Tables

File Id to Layout Id Mapping

File id to layout id mapping is stored by the basic file attributes table (FAB). Hence, there is no table in the layout schema to store this mapping. Following file id to layout id mapping is shown for the completeness of the example in this section and is assumed to be part of FAB.

File Id (c2_fid)	Layout Id (c2_layout_id)
fid1001	L1
fid1002	L1
fid1003	L1
fid1004	L2
fid1005	L2
fid1006	L3
fid1007	L4
fid1008	L5
...	...
fid....x	L6

Table “layouts”

Table Name	layouts
Key	layout_id (Type: c2_layout_id)
Record	- layout_type (pdclust, composite, coblist) - reference_count - byte_array

	- pointer to formula OR list_of_component_object_id if it is not a formula
Comments	*byte_array* is parsed and composed by the layout_type specific decoding and encoding methods. The For parity-declustered layout type with formula-type as "LINEAR", the *byte_array* contains the record with N (number of data units in parity group) (Type: uint32_t), K (number of parity units in parity groups) (Type: uint32_t) and unit size (Type: uint32_t). For parity-declustered layout type with formula-type as "LIST", the *byte_array* contains list of *cob ids*.

Tabular representation of "layouts" table with example

layout_id (c2_layout_id)	**layout_type** (c2_layout_type_id)	**reference_count** (c2_uint32)	**byte_array**
L1	pdclust	3	LINEAR, U, N, K ~~pointer_to_formula~~
L2	pdclust	2	LINEAR, U, N, K ~~pointer_to_formula~~
L3	pdclust	1	LINEAR, U, N, K ~~pointer_to_formula~~
L4	composite	1	
L5	composite	1	
L6	pdclust ~~coblist~~	1	LIST, cob1, cob2, cob3

Table "composite_layout_extent_map"

Conceptually, composite_layout_extent_map stores "extent-layout id" pairs for all the files using composite layouts.

This table is to be implemented using c2_emap table which is a framework to store extent map collection. c2_emap stores a collection of related extent maps. Individual maps within a collection are identified by an element of the *key* called as *prefix* (128 bit). In case of composite_layout_extent_map, *prefix* would incorporate *layout_id* for the composite layout.

Table Name	composite_layout_extent_map
Key	- composite_layout_id - last_offset_of_segment
Record	- start_offset_of_segment - layout_id
Comments	*layout_id* is a foreign key referring record, in the *layouts* table, of the corresponding layout.

Tabular representation of "composite_layouts_extent_map" table with example

segment_id	**prefix incorporating layout_id**	**segment_start_offset**	**segment_end_offset**	**layout_id**
L4:segment1	L4			L1
L4:segment2	L4			L2
L4:segment3	L4			L1
L5:segment1	L5			L1
L5:segment2	L5			L2
L5:segment3	L5			L3
L5:segment4	L5			L6

Assumptions

1. **Meta-data striping** (traditionally called *clustered meta-data*) is outside of T1 scope. This is by referring to SNS Overview Doc.
2. **Directory striping** is not yet considered in this document nor does it seem to be touched by the HLD. Will appreciate to receive inputs on that so that it can be considered into the

design.

Queries

1. **Supported Layout types:** What is the exact list of layout types that we need to consider during layout schema design?
2. **FileId to LayoutId mapping**:
    a. Will this mapping be stored as a part of the Fileattr_basic table (c2_cob_fabrec)?
    b. If yes, do we still want to store it in Layout Schema as well for any reason like performance?
3. **Layout updates:**
    a. When some properties/parameters of a layout are updated, the *layoutid* will be retained as is. Is that correct?
    b. Should layout modification function generate a layout change notification?
    c. Should the modified layout be marked as invalid till old layout is dropped by all the servers using it?
4. **Layout fields to indicate if active/degraded:** Do we need some fields in the layout type description tables to indicate if the specific layout is active, if it is in a degraded mode and anything else if any?
5. **Reference on a layout:** What do we mean by this and where and how is this to be used?
6. **UI for layout creation:** What is the user (admin) interface for creating a layout?
7. **Storage pool:** It is assumed that a storage pool is allocated per layout and thus can not be shared by multiple files (since there is one separate layout instance per file). Is that understanding correct?
8. **Composite layout type:**
    a. Can this be elaborated a bit as how do we want to store it?
    b. Is this applicable for pdclust layout type?
9. **Storing storage pool info:**
    a. Does layout schema need to store storage pool info or it will be stored by the pool module?
    b. If yes, what additional tables are required for that?
10. **Failure vector:**
    a. HLD of parity de-clustering algorithm mentions “Failure vector is an attribute associated with a layout and checked on every IO by the server“.
    b. Is the Storage Pool Table the right place to store failure vector?
11. **Parity de-clustering (pdclust) formula:**
    a. Referring to one possible formula by referring to the layout related video is SiFID = Ai + B + F * SomeLargeNumber.
    b. Would you please demonstrate how do different parameters of this formula (A, B, F) map to the input parameters N, K and anything else if any.
12. **Handling failure in case of pdclust layout:**
    a. Let us consider using a formula like SiFID = Ai + B + F * SomeLargeNumber, as is explained in the video regarding Layouts.

    b. After say first failure, the whole file will get a different layout now that F will have value of 1.
    c. Is this correct or do we have a composite layout - one layout upto some part of the file and another for the later part of the file?
13. **Layout caching:** As per the component description of T1.2IndTasks, layout caching is part of the Layout Schema task. Just wanted to confirm if this is valid as of now?
14. **Policy:** Can you please shed some light upon various policies those may play role into layout module like selecting a layout type for a file being created, storage preallocation, parent directory attributes etc.
15. **Local RAID:**
    a. Can we define what local RAID is?
    b. Do we need to consider that into the list of supported raid layouts?
16. **RAID and pdclust layout types relation:**
    a. It seems to be possible to derive pdclust parameters from raid parameters. In other words, pdclust could be said as a special case of RAID.
    b. Considering that, do we want the raid and pdclust layouts to be stored in the same table or different tables?

Notes from discussion held on 11 Oct 2011

Colibri File Striping using Component Objects

1. Colibri uses object based striping to store files.
2. For each file (as visible to an end user of file system), a number of "component objects" (COB) are created.
3. Typically a COB is created per device in the pool. (A pool in this context is a collection of servers running ioservices and storage devices attached to these servers.)
4. The file is "striped" across these component objects, similarly to how a traditional RAID device is striped across underlying physical devices.

COB identifier / COB id / COB fid

1. Fid in Colibri is 128-bit quantity, composed of 2 8-byte values.
2. To form a COB identifier, only a portion of file's fid can be used, because we need 8 bytes for parameter (that is file fid).
3. The whole pair (fid, COB index) is considered as COB fid and is also referred as COB identifier.

Colibri File Layout

A file layout is used by a client to do IO against a file.

The Colibri file layout stores component object ids either in a form of a list, or as a formula. The

most basic layout IO interface is a mapping of file's logical namespace into storage objects' namespace.

Mapping a logical file offset to physical block numbers involves two mappings as below:

1. Logical-offset-in-file to “cob-id, offset-in-cob” mapping
    a. Component object id is obtained by using the file's layout.
    b. Offset ??? (probably derived by using simple arithmetic)
2. Offset-in-cob to block-numbers
    a. This is stored/determined by the special type of storage object (not discussed any further in this document).

Layout Formula

1. Formula is a type of layout.
2. Using formula, we calculate cob ids each time we need them, instead of storing those cob ids.
3. Once user queries for cob ids for a file, we point to the formula using the layout DB and the formula does the job of calculating the cob ids.
4. Fid of i-th component object is calculated as a function of formula attributes (stored in the layout DB), formula parameters (supplied by the user as input) and i.
5. For example, a formula might have attribute X and parameter F. Then fid of i-th cob can be calculated as (X + F, i). When a file uses this formula, it substitutes its (file's) fid as F.
6. One layout formula (stored as a part of a layout), can produce different sets of cob fids, when different values of parameter are substituted into it. Multiple files can use the same layout formula (which is stored only once in the data-base) and yet have separate sets of cobs.
7. For example, a file with fid 100 say F1 has cobs (100, 0), ... (100, 83) and a file with fid 101 say F2 has cobs (101, 0), ... (101, 83). These files use the same layout formula to generate cob fids: (F, i), where F is an input parameter (file fid) and i is a cob "index". The whole pair (F, i) is used as a cob identifier.
8. Considering this same example above, formula is an economical way to store a single L0 record in the layout data-base. This is a record for a formula with a parameter F. Then file F1 stores L0.layoutid in its fab (basic file attribute) record and the file F2 stores the same.

Disjoint sets

1. Typically all files in a pool use the same values of N (number of data units in a parity group) and K (number of parity units in a parity group).
2. There will be multiple formulae for the same N and K, simply because with a single formula it's very difficult to arrange for disjointness of cob sets.
3. Takes the simplest example, we looked at above, when a cob fid is calculated as (F, i). This guarantees that different files (with different fids) get disjoint sets of cobs, but consecutive cob fids, e.g., (100, 7), (100, 8) are always scattered across different servers in the pool.
4. This means that a map that maps cob id to its location in the cluster (and we need such

map) will be extremely fragmented and large.

5. This is why there could be a "large_number" in a formula: allowing to map large continuous batches of cobs to the same server, while maintaining disjointness. (The formula being referred could be “SiFID = Ai + B + F * SomeLargeNumber”)
6. The "large_number" in formula is used to make cobid->serviceid map more compact. ??

Layout policy, File Creation, Resource Manager

1. Layout formulae are created administratively. The assignment policy should be more runtimeish.
2. Something like POSIX ACL assignment. E.g., a directory has a default layout and when a file is created in this directory it inherits this layout.
3. Plus an ioctl() to assign a layout.
4. "layouts can be assigned both by the server ("Lustre style") and by the client, when a client uses existing layout formula to create a file. In the latter case, the client can create inode and component objects concurrently."
5. Colibri has a so-called "resource manager" sub-system which allows various file system resources to be distributed across nodes.
6. Fids and layouts are resources. This means that a node (including a client) can "cache" a range of fids (not yet used for any file).
7. In a "traditional" file system, like Lustre, file creation happens as follows:
    a. A client sends CREATE request to the meta-data server.
    b. The server creates an "inode" and assigns attributes, like layout to the new file.
    c. Now, there are two possibilities: either meta-data server sends CREATE requests to the data-servers to create component objects, or it replies back to the client and the client sends these CREATEs.
8. File creation with Colibri Resource Manager:
    a. With our wonderful resource manager, the client can assign attributes, like fid and layout to the file _before_ it sends the CREATE fop to the md service.
    b. Therefore, it can send CREATE fops to ioservices (to create cobs), before it gets reply from the mdservice, because it already knows the layout.
    c. This removes an rpc latency from the critical meta-data path.
    d. And it adds an ability to batch operations on a client: create a million files locally, destroy 999999 of them and sends remaining CREATE to mds.
    e. Lustre and NFS would send 1000000+999999 rpcs instead of 1 in this case.

Reference count on a layout

1. Layouts need persistent reference counters too.
2. The layout must remain in the data-base while its layoutid is stored anywhere in the data-base.
3. But for a formula, the counter never drops to 0, because the formula is created and destroyed administratively.
4. And for a non-formulaic layout, the reference counter would be typically 1.
5. Refcount is incremented when a layout is associated with a new user.
    a. When a first file is created using this layout, this will be the first user.

   b. If multiple files use the same layout, the counter is incremented for each one.
   c. In other words, when any new file is created/associated with a particular layout, it's ref count is incremented.
6. Refcount is decremented when a layout is detached from its user.

Use Case of a file layout with formula: Reading from a file F0

1. Fetch F0 cob.
2. Fetch F0 fab.
3. Extract layoutid from fab.
4. Fetch corresponding layout from the layout table.
5. See that the layout is a formula, taking a single parameter F.
6. Substitute F0's fid into formula.
7. Get a set of cob fids to operate on.

Composite layout

1. T1.2IndTasks->components states that “Only need parity declustered layouts in db”. It looks like we need composite layout type as well (to demonstrate recovery).
2. It is not yet clear whether we can demonstrate SNS repair without composite layouts.
3. We want to store a composite layout like like list of "layout ids - extents" pairs.
4. But list is, generally, not scalable enough. Hence, a tree is needed.
5. Perhaps in a separate table.
6. TBD: Take a look at db/extmap.h.

Layout attributes - Number of data and parity units in a parity group (N, K)

1. Should each formula duplicate them?
2. They could be properties of something else that is specific per storage pool.
3. Considering them as layout attributes to start with and will explore more on if they could be properties of something that is specific per storage pool.
4. There is one important point about "something" that one has to keep in mind. Layouts can be invalidated by changes in hardware configuration, including failures. This is described (without much detail though) in "HLD of SNS repair".
5. TBD: Explore if N and K could be properties of something that is specific per storage pool.

Use Cases

[NZR:
Perhaps this belongs in the caching doc, but I think it might be useful to elaborate some use cases to clarify how layouts are generated, assigned, and modified. In addition to e.g. local file creation, file migration, and PDRAID reconstruction, I think some interesting layout cases to at least think about would be:
1. NBA

2. Multiple massive readers: single file read by large numbers of readers could turn into a bittorrent-like layout (clients can access multiple copies of data cached on multiple clients).
]