High Level Design Specification for Index Api Module

Danilov Nikita

2006.01.20–2006.01.23

Contents

1 Introduction

This document describes the design of Index Api Module (*iam*) that is to be used by other Lustre components to persistently and transactionally store keyed data items. iam is to be developed during Colibri-4 development cycle (2006.01–2006.03).

2 Distribution

People interested in

- inspecting iam design
- improving or debugging iam code
- using iam interfaces

should read this document. Understanding of ext3 htree code is assumed.

This document contains confidential information, and is a property of ClusterFS. If you received it without appropriate authorization, please shoot yourself to death before continuing.

3 Background on indexing in Lustre (what's going on?)

3.1 The need for indexing

In many places Lustre persistently stores and later retrieves data items associated with keys. Most obvious example is an OST object index (`OBJECT/` directory), where every inode managed by the OST is indexed by some identifier (`group/id`), other instances include FID location directory of the upcoming fids implementation, that maps fids to store cookies, iopen()-namespace in MDD, and, ultimately, implementation of directory namespaces in MDD, where each directory maps name to inode number.

In all these cases Lustre needs some sort of efficient persistent *map* that translates keys (inode numbers, fids, file names) into values (inodes, store cookies, inode numbers). Multiple techniques are used to implement these maps:

- in most cases, indices are implemented on the top of directory service provided by underlying ldiskfs, that is, ext3 htree is used as an index,
- mds uses iopen patch to index inodes.

iam should provide single interface that higher layers can use to do the indexing in all above cases.

3.2 ext3 htree

htree code implements B+-tree with the following limitations:

- tree cannot be taller than 3 levels;

- key size is fixed at 32 bits;
- key values (called *hashes* in the htree) are not stored on the leaf level, as they can be recomputed from records;
- tree operations are synchronized by single per-tree mutex;
- nodes are not compacted on record removal;
- htree is implemented on top of flat file, leading to the „two-level-indexing“: downward node pointers, stored in the index nodes, are logical offsets within flat file, that are translated into physical block numbers through standard direct-indirect-double-indirect pointers stores in the inode;
- records in the leaf level nodes are not sorted by key. This simplifies insertion (new record can always be inserted at the first empty slot of sufficient size), but complicates node split, and, especially, iteration over the tree (htree maintains additional in-memory only rb-tree to handle this).

4 Requirements (what do we want?)

- implementation: it is assumed that initial implementation of index api is based on existing ext3 htree code, but api should be designed to be generic enough to allow different implementations.
- scalability:
  - number of records (tree height)
  - key width
  - record width
  - hash collision (non-unique keys) handling
  - better concurrency control (finer grained locking)
- compatibility:
  - existing ODB layout can be upgraded
  - fsck support
- compacting: optional for release 1 probably.
- assess other tree types.
- iterator api: we need an iterator that can efficiently resume a stopped iteration even if new entries were added between stop and restart. Knowing the "largest fid" in the tree easily is another attractive features. Alex has already worked with a sorted hash and htree (replacing ext3's native hash).
- transactions: replay, reply reconstruction of osd_create commands needs to play nicely with the index, preferably transaction free by ordering.

5 Functional Specification (what to do?)

osd api is expended to include new indexing interface that provides users with an abstraction of persistent transactionally updated container where records indexed by keys can be stored and later retrieved. Few preliminary definitions are in order:

key a bit-string that identifies a record. iam doesn't care about the meaning of the key. The only operations performed on keys are tests: `less-than`, `greater-than`, `equal-to`. All keys in the given tree are of the same size.

record a data item stored in a container. Each record in the container is identified by a key, but not necessary uniquely. If multiple records have the same key, we talk about „non-unique keys“ (this is called „hash collision“ in ext3 htree).

container a persistent transactional storage for records, identified by keys.

cursor a data structure representing a location within a container. Cursor can be moved within container, and operations on records located under the cursor can be performed.

5.1 simple container interface

This interface is suitable for majority of iam users. It provides basic container functionality, while having certain limitations:

- it doesn't export internal iam locking to the user (which is an advantage);
- its applicability to the case of non-unique keys is limited.

Interface methods are:

- create new empty container, or open existing one specifying some parameters, at least:
  - key size (32 bits, 64 bits, sizeof (fid), etc.);
  - compatibility mode (container should be binary compatible with ext3 htree);
- insert new record into container, record and its key are supplied by the caller;
- lookup record by the key, and return it if found;
- delete record with given key;
- replace record with given key with new given record;

5.2 cursor interface

This interface provides fuller access to container. It's not specified whether simple interface is implemented on the top of cursor one (albeit this is reasonable to assume). Interface methods are:

- create a cursor, located at the first record with given key;
- move cursor one record right;
- return record under the cursor;
- insert new record at the cursor;
- replace record under the cursor;
- delete record under the cursor;

Cursor interface is for users that want more freedom in manipulating container contents, in particular, in the cases where non-unique keys are present.

Both interfaces are transactional, that is, operations that mutate the tree are performed atomically in the context of some explicitly specified transaction.

6 Use Cases (how to use?)

6.1 fids location database

MDD-side fid code has to map a fid to the location (MDS id) and storage cookie (inode number plus generation number). To this end iam index is created in which fid is used as a key, and a pair of location and a storage cookie is used as a record. When new object is create a record is inserted into that index. This record is looked up to resolve fid into location and storage cookie. Keys and records are of fixed size in this example.

6.2 osd object index

Similarly to MDS-fids, OST code has to resolve group/id pair into inode/generation pair. Again, an index is created where group/id is used as a key and inode/generation as a value. In this case we are bound by compatibility requirement as existing code uses `OBJECT/` directory hierarchy to implement such mapping. Switching to non-legacy index implementation would require changes to fsck, because `OBJECT/` is the only directory that references OST objects. If such references go away, unmodified fsck would scavenge all objects into `lost+found`.

6.3 iopen

MDS uses iopen kernel patch to retrieve inode from the disk, given inode number. While we cannot actually maintain an index with inodes as records (because ext3 expects inodes to be at fixed predetermined locations), we can emulate iam interface in this case, that is, to create an implementation of iam methods that uses inode number as a key, and ext3 inode tables as underlying container. This would unify client code, and would ease porting of MDS to the user-level.

6.4 mdd namespace

MDD uses ldiskfs directories to implement lustre directory hierarchy. New code will access these directories through iam interface.

7 Logical Specification (how to do?)

7.1 Layering

OBD interface exported by OSD will be expanded to include iam interface as described in the functional specification. This is implemented by adding new sub-structure to the `struct ost_obd` with separate operation vector. This (compared with directly adding iam methods to the obd_type->typ_ops) allows

- OSD to use iam internally (see use cases), and
- to have multiple implementations of iam within the same OSD code.

First implementation of iam is based on existing ext3 htree code. New functions are added to ext3/ldiskfs, and exported directly (i.e., through `EXPORT_SYMBOL()` rather than through inode/file operations).

7.2 Changes to ext3

We need code to support:

- taller trees (should be simple);
- different key sizes;
- leaf-level nodes with sorted records;
- per-node locking during tree lookup and tree updates;
- we also need a support for legacy mode binary compatible with existing htrees. To achieve this, internal interfaces are introduced within iam implementation that allows to share common code between legacy and new modes. E.g., there will be a method ->record_key_get() that in new node returns key stored in the node, and for legacy mode recomputes key (i.e., hash) for the record on the leaf level.

8 State Specification (how it works?)

Complete formal state specification of tree is beyond scope of this document.

8.1 Locking

Depending on the time available we can follow one of the possible routes:

- keep „global“ per-container mutex;
- switch to global read-write semaphore;

- switch to per-node read-write semaphore: downward tree traversals use coupled read-locks to descend, mutating tree operation use the notion of a „safe-node“. (For write-able cursor safe-node is always a root of the tree.)
- Explore advanced algorithms (e.g., from [1]).

9 Alternatives (how else?)

- implement iam outside of ldiskfs on the top of flat file functionality provided by the latter;
- implement iam directly on the top of block-device layer;
- discard cursor interface and non-unique keys. Force keys uniqueness by prepending „generation“ to them (this route was followed by reiserfs v3);
- implement iam as extensible hash table, rather than balanced tree.

10 Focus for inspection and possible problems

- missed requirements;
- some ext3 htree specifics that would complicate implementation on iam on top of it;
- interaction of jbd transactions;
- interaction with buffer cache (especially, meta-data aliasing).

11 References

[1] http://www.acm.org/sigmod/vldb/journal/VLDBJ2/P361.pdf