Laid Locking HLD

Qian YingJin

06/01/24

Contents

1 Requirements

1.1 Avoid locking across multiple OSTs

Locking across more than one OST, which results from the write across the stripe boundaries, may cause unexpected problem such as eviction of client. For example, assume a client needs to queue K locks on K OSTs for a write. Supposed that last OST goes down and the client got to time-out for waiting last lock grant. At the same time the other OSTs place callback on locks already granted to the client. The lock will time out and hence the client is evicted by first K - 1 OSTs and time out on last one.

So we must design new locking model especially for mirror LAID and LAID5 to avoid locking accross more than one OST.

1.2 Locking for parity

The file system will lock extents but the parity block is subject to a separate lock policy. One approach is to lock the stripes including the parity block. Another is to use an

optimistic or service based locking scheme for the parity block. When all clients write to one file, it must be assumed that there will be a significant number of writes are partial writes from one client, leading to contention for the parity block. The cluster will go as slow as the slowest elements, so the efficiency of handling shared parity is critical.

For the former: locking including the parity block with stripe-group-size granularity, it elimates the conflict for parity update and simplifies the processes in degraed case as client has already acquried locks covered the whole stripe group for later preread the necessary stripe units to reconstruct the blocks store on failed OSTs, but it reduces the concurrency.

For the latter, we have proposed several schemes such as server based locking for parity block and updating parity on OSTs which is discribed in the LAID5_HLD.lyx. But they can not resolve the degraded IO well because client still need to acquire the PR locks covered the stripe row to reconstruct the block, which make the locking process very complex in the degraded case. And for small write it must update the parity via method read-modify-write. In addition, it makes the design of laid5 very complex.

So we prefer to fromer scheme, but we need to design a mechanism to avoid the locking accross more than one OST.

1.3 Recoverable lock

Another requirement for Laid locking is to achieve the goal that If the client crashes, the locks for redundant file are recoverable. The locking protocols can be extentded to ensure consistency pessimistically in the following way: If the client crashes holding a write lock, then the lock server on OST detects the client failed and marks the target range suspect and recomputes the parity on recovery to ensure it is consistent before granting the lock to another client. This requires maintaining the lock state at the OSTs on stable storage so that on recovery the lock server can determine the suspect objects and associated ranges. But this pessimistic scheme makes lock operations expensive since the state of each lock operation must be recorded on persistent storage. So we need design a optimistic mechanism to reduce the overhead of lock operation.

2 Functional specification

2.1 Definitions

- ***Master locking*** - Choose an objects in the redundant set of a file stripping over as a master object. All the extent lock operations for IO just place on the lock resource: master object.
- ***Server-based locking*** - Lock operations just places on the lock server. locks don't enqueue and cache in the local lock namespace.

2.2 Master locking

In the original design for locking of mirror LAID and LAID5, it acquires extent lock from all lock servers the set of "related" redundant objects (For LAID1/LAID01, it is all mirrors; For LAID5, it is all objects the file stripping over) resides on. Now we introduce a new locking strategy: Master locking. By this way it can not only reduce the lock operations but also avoid locking accross many OSTs. But it involves the remastering process when the lock server master object resides on occurs failure.

If the read, write or locking operation fails and involves a time out, the related object needs to marked as "failed" on MDS, and MDS should coordinate the remastering process for all client open this file. At the same time, MDS should update the LOV descriptor on all clients and mark the corresponding OSC as inactive, otherwise all client would incur the timeout.

2.3 Versioning write

The generic idea of versioning write is that: Each object on OST keeps the new version of blocks for the write protected by the extent lock from client , and cancels the old blocks, commits the new versions when the whole stripe groups have updated. By this protocol, system can suffer client and stroage-node crash failures in all case of LAID, and avoids the excess work of two-phase commits. Althogh verioning write is mainly used for recovery and beyond the scope of locking, but our impelmentation is combined with lock strategy. So here we mainly focus on how to commit the new version and discard the old blocks via lock strategy and just give out the rough design of disk format and manage of various version from different client and inconsistent recovery about it.

2.4 Time-Based Locking

The lease is defined as the time that a lock is presumed to be valid. A lease defines a contract between a client and a server in which the server promises to respect the clients' locks for a specified period. In our LAID design lease-based lock can be just used for write locking. The lock server grants a extnet lock to client with a lease. As the extent lock is granted with stripe-group-size granulairty in case of LAID5, OST can identify the write as new version of file data during valid period of lease; At the time the lease nearly expires, client flushes all cache data in the lock extent, then notifies the OSTs the entire stripe groups has commited and replace the old blocks with the new version. After that renew the lease if necessary. That is to say, use lease to time-out client locks. We use this time-out mechanism to implement the recoveryable lock.

To implement time-based locking the following functionality still needs be done:

- client takes extent lock with a lease from lock server.
- If a lease is not renewed in time, the locks are broken.
- When a lease expires, the server can be assured that the client no longer acts on the locked data, and can safely redistribute locks to other clients after inconsistent recovery for the suspect extent of the lock.

3 Use cases

3.1 Open file

- When client opens the file, MDS returns the index of master object to client.
- All clients take lock operations on the same master object.

3.2 Write file

- For LAID5, client extands the extent lock with stripe-group-size granularity.
- Acquire the extent lock from the lock server master object resides on.
- After that, wirte the data to cache and release the extent lock.

3.3 Remaster

- When detect that the OST master object resides on occurs failure, client will start the remaster process.
- Client sends a RPC to MDS to request changeing the master object.
- MDS chooses another object in the redundant set as the new master object.
- MDS notifies all clients to replace the master object with the new one by lock callback.

3.4 Truncate file

- Client performs a truncate operation on a file.
- Extend the start of the extent lock PW[attr->ia_size, OBD_OBJECT_EOF] with stripe-group-size granularity (start = attr->ia_size & (stripe_size - 1).
- Client gets the exten lock from master object.
- Client contracts to OSTs to punch the objcts.
- Client cancels the extent lock.

3.5 Glimpse size (LAID5 with master locking)

- Client requests a extent lock PR[0, OBD_OBJECT_EOF] to protect the size from the server master object resides on with a GLIMPSE flag.
- The lock server (OST) will reply to the client when the glimpse callback have been finished. The lvb contained size information or lvb vector is retruned to the client in the completion. (Details see subsection 4.3).

- Client sents lock request NL[0, OBD_OBJECT_EOF] for other objects to get the object size informtion if necessary.
- Client merges the object size to get the file size information.
- Client cancels the extent lock from the server master object resides on.

3.6 Extent lock blocking callback

- The lock callback just places on the master object. But in the callback function, we still need to flush the cache in the same extent of other objects in the redundant set.
- After flushing, send messages to all object in the redundant set to commit the write in the lock extent.

3.7 Client crash

- After the client crashes, the extent lock grant to this client will break when the lease expires.
- when detect the failure of client by the lease expiring, OST starts inconsistent recovery according to the extent lock.
- Other clients want to acquire a conflicting extent lock already grant to the non-responding client (maybe crash, maybe disconnect with the OST), the requested client must wait until the lease expires.

4 Logic specification

To meet new functionality Laid locking should work out the following fields.

4.1 Master Locking

4.1.1 Algorithm of master locking

Here “Master locking” is short for that just acquire the extent lock form the master object in the redundant set (For mirror LAID, it is one of the mirrors; For LAID5, the extent should be extanded with stripe-group-size granularity) . Although it can reduce the concurrency of file IO, but it also reduces the count of lock operations and the chance of locking failure and simplify the design of LAID5. The algorithm of master locking is described as follow: (Here we don’t consider the complex failures such as network failure, network partition. For example, MDS can talk to OST but the client can not.)

1. When client opens a file for r/w, MDS will return the index of master object with timestamp (it usually the index of first object and the OST it resides on must be active) of the redundant sets, and stores this information in file inode.

2. All lock acquirements just place on the master objects. That is to say, for mirror LAID, client just acquires locks from one of the mirrors; for LAID5, client just acquires locks with stripe-group-size granularity but we expect the lock has protected the entire stripe groups and lock the same extent in the other objects in the redundant set.

3. When read, write, lock operation fails and involves a timeout, the client notifies MDS by a RPC. MDS marks the related object as “failed”.

4. If the object is one of master objects of file (For LAID01, a file may have several master objects), MDS still needs to coordinate the remastering process. MDS first chooses another object in the redundant set as master object the OST which resides on is active, and then notify the clients opened this file to change the master object and this can be done by the mechanism similar with lock callback. The callback message should contain the following information: failed object id and new master object id and the timestamp of remastering.

5. When the client receives the remastering message, first replace the master object and mark the failed object as unwritable, then flush all cache in the lock extent but skipping the failed object, and cancel the extent lock acquired for the failed master object in the local lock namespace.

6. After MDS receives all related client replies, reply success to the clients initiate and want the remastering.

7. If timeout occurs to one of the client, MDS will evict the client.

8. After that, all operation is done as normal.

4.1.2 Lock callback for master locking

The lock callback just places on the master object. In case of LAID5, because we think the master lock covers the same extent in the other objects in the redundant set, so client should also flush the caches belongs to the other objects in the same extent in the blocking ast.

4.2 Size management

4.2.1 Recently seen size

In the original lock strategy, we can obtain the recently seen size of the objects in the peggyback lvb of lock opertion, But for master locking it just acquires locks from one of redundant objects, so we may need a special process especailly in case of LAID5:

- When request a extent lock, client sends the lock request to lock server master object resides on if can not find a match lock in the local lock namespace.

- After acquired the extent lock from lock server, client must also request lock NL to the OSTs other objects reside on. Notice: this kind lock is a little similar with glimpse lock, it executed as an intent associdated with the lock and resource to get the size information.
- After get the recently seen size information, client stores them in *lois* of various objects.

4.2.2 Lock value bock vector

Here we introduce lock value block vector. The lock resource for master object in LDLM on OST contains the lock value block vector, in which maybe also store lvb of other objects in the redundant set returned by clients. It is mainly used by glimpse callback.

4.2.3 Glimpse size

The client get the file size by doing a size glimpse callback. The process of size probe by glimpse under master locking is as following:

- Client send a extent lock request PR[0, -1] with glimpse flag to the lock server master object resides on.
- If the lock resource for master object has no other incompatible locks:
- 1) Ther server will grant the lock to client just with the size informaion of master object.
- 2) Similar with recently seen size, client sends NL lock request (intent lock) to the other objects to get the size information.
- 3) Client gets file size by mergeing the objects' size .
- 4) Client cancels the extent lock PR[0, -1].
- If the lock resource on lock server has incompatible locks:
- 1) The lock server first scans all locks of the resource and choose the highest of the PW locks which are larger than the size in the LVB and performs a glimpse callback.
- 2) In case of LAID5, Glimpse callback does not cause the lock to be relinquished but instead the client granted the lock returns the KMSs of all objects to the lock server on OST.
- 3) The lock server packs these size information of all objects into the LVB vector and reply to client with the flag HAS_LVB_VECTOR.
- 4) When client receives the reply with flag HAS_LVB_VECTOR, unpacks all objects' size from the LVB vector, and then compute the file size.
- 5) After that, client cancels the extent lock.

4.3 Lease-Based Locking

In distributed systems, a node will occasionally become unresponsive. Other machines cannot determine whether a non-responding machine has crashed, is stalled or merely in a network partition. Lease-based lock can easily solved these failure. But here we focus on how to utilize the new lock strategy to achieve consistency and simplify the recovery.

4.3.1 Lease phases

In the lease-based locking, we still use master locking for the redundant file. The lease is subdivided into three phase: Lease valid (T1), flush period (T2), renewal period (T3). T1 contains two part: maximum amount of time writting the data to cache should take, plus the time file pages in the lock extent could cache on client. T2 is the amount of time flushing cache to OST should take. T3 is the amount of time renewal lease should take.

4.3.2 Versoining write

Every lock request for write passes with a version number. The version number is: UUID + timestamp + range, where the UUID is UUID of client starts the write operation, timestamp could be *inode->i_mtime* or the time client start the lock operation, the range is the extent of the lock. All pages during this write are tagged with the version number. The pages send to OST with version number, too. Before client notifies the object the block in the version can be committed, all writes to the object are regarded as the new version. New version values in the object data can store like ext3COW (reference the paper - “ext3COW: The Design, Implementation and Analysis of Metadata for a Time-Shifting File system”). Objects on OST can manage the new version blocks by two hierarchy: client_UUIT / timestamp, and we can save this information in the EA of the object.

4.3.3 Algorithm of recoverable locking strategy

The algorithm of recoverable locking strategy is discribed as follow:

1. Client A sends extent lock request with the version number V(version).
2. Lock server on OST grants a extent lock L with a proper term lease to client A.
3. After acquired, the lock L cached on local lock namespace of client is tagged with the version number. When client finds a compatible extent lock on local namespace and if the left lease is not enough to write the data to cache, the client renews the lease immediately.
4. Client A begins to write the data into cache. Every page covered by the extent lock is taged with the version it belongs to: {UUID, Timestamp, Range}. At the end of T1, DLM on client begins to flush the cache in the extent of lock L.

5. During the period of T1 and T2 before send commit message , all data write to object on OST are stored as new version blocks.

6. After finish flushing, the client notifies all objects in the redundant set to commit the blocks with the version of the lock.

7. When OST receives commit message to the object with version V(A, T, R) (where the A is short for client A, T is short for timestamp, R is short for range), all new version block of the object, generated by the client A and with timestamp smaller that T, can be committed and the old blocks can be canceled.

8. Client can also peggy a flag in the commit message to indicate whether it want to renew the lock or cancel it to the lock server master object resides on.

9. If client want to renew the lock, the lock server will give a new term lease to the client, and extentd expire time of the lock on OST's lock namespace and so does client after receive the repay from OST.

10. If lock server don't receive any renewal message when the lease expires, it will regard the client as failure, revoke the extent lock and involve a recovery process according to the lock extent.

4.3.4 Lease term

Long lease terms are significantly more efficient both for the client and server on files that are accessed repeatedly and have relatively little write-sharing. But it also adds the amount of suspect extent after failure on recovery; While short lease terms can minimize the delay resulting from client and server failures but it could affect the performance. So the DLM should return with each lock an idea of how much data should be cached, how long the cache time in the lease should be. The DLM can also scale the amount of time for the lease based on count of clients granted the lock and object count of the redundant set.

4.3.5 Handling for fsync operation

After .fsync operaion, all cache should be committed to objects on OSTs. Successful fsync operation should commit all new version blocks generated by this client. So after sync the caches, we should also send a commit message to all objects to cancel the old blocks and commit new version from this client.

4.4 Handle write across the boundaries for LAID0

For LAID0, we still face the problem that locking acorss more than one OST when write across the stripe boundaries. Here just give out the proposed design.

For a big write across the stripe boundaries, we subdivided the write extent E into small extents in the various objects: {e(1), e(2), e(3),...} and store this information in the *loi*.

4.4.1 Direct IO

For direct IO, we can acquire and cancel the extent in function *lov_brw_async*, that is to say process as the following sequence: lock the extent e(i), write data to object o(i), unlock the extent e(i). And this lock could even be server based lock.

4.4.2 Writeback cache

For writeback cache, we acquire the extent lock e(i) when write the first page belonged to the object o(i); Unlock the extent lock when write the last page in the extent of the object. (Is there any problem that acquire the extent lock during the page locked??).

Another way to avoid locking across more than one OST is that: We change the *ll_file_write/read*, in which we don't use *generic_file_write/read ;* Instead we divide the user data buf into stripe_count pieces according to the metadata information: stripe_count, stripe_pattern, stripe_size. And then the lock acquirement, writing data to cache and lock release just places and acts on their own assocaited object just like the scheme of Direct_IO above. The pseudo code is shown as follow:

```
static ssize_t ll_file_write(struct *file, const char *buf,
                     size_t count, loff_t *ppos)
{
    ...
    struct lov_stripe_md *lsm;
    struct ldlm_extent sub_extent;
    char *sub_buf;
    ...
    for (i = 0; i < lsm->lsm_stripe_count; i++) {
       sub_buf = NULL;
       sub_buf = get_stripe_buf(buf, count, ppos, lsm, &sub_extent);
       if (sub_buf == NULL)
             continue;

       ll_tree_lock(...sub_buf, sub_extent...);
       ll_file_stripe_write(...);
       ll_tree_unlock(...);
   }
...
}
```

For mmap wirte, it usually doesn't exist this problem.

5 State management

5.1 Scalability & performance

5.1.1 Redundant set

To simplify the recovery for LAID, all OSTs in the cluster would be better to divide into many sets such as mirror sets, LAID5 sets with 3 objects and LAID5 sets with 4 objects, etc. So that we can reduce the count of OSTs involve the recovery for LAID, make the distributed log for LAID recovery collected more easily.

5.1.2 Performance impact

- "Divide write" for LAID0: It has little impact on the performance because in this scheme the extra work is divide the use buf to sub set and it doesn't add the workload and latency of lock and write operations.
- Version write: Version write will impact the performance. Because the extent lock are best nerver refreshed with leases, but just sit there for the little environment with little write sharing and frequently repeated write; Another is that every new version number must store in the EA of the object when first write to the object in this version and delete when the new version is committed.

5.1.3 Lazy parity flushing under master locking

Under master locking, we can delay flush parity even until the renewal or callback of the extent lock. By this way, it can greatly reduce the latency of updating parity and improve the performance.

5.2 Recovery changes

5.2.1 failure handle for remastering

- When client wants to initiate the remastering but find MDS is unrespondable, it should discard all cache and report error immediately. (This client may be also evicted by MDS)
- When MDS coordinates the remastering process but one of the client opened the file occurs failure, the MDS should evict the client and notify OSTs the file stripping over to evict the client.

5.2.2 Inconsistent recovery

When the client crashes, a recovery process will trigger by the expiring of the lease. The OST master object resides on will notify all objects in the redundant set to roll-back to a consistent state by discarding the new version blocks in the lock extent; Or reconstruct suspect rangs in the uncommitted new versions srored in EA.

In the case of single OST's failure which will result in inconsistent write, we can notify the master object write a log record contains the suspect extent of inconsistent write when send commit message to all objects in the redundant set, or log the state of all lock extents after failure into stable storage; Another way is that allocate spare object as a log object when create the redundant file and use it to log the inconsitent write extent or write all data of the failed object to the spare object.

5.3 Disk format changes

We use a disk-oriented copy-on-write scheme to support our file versioning, which is very similar with ext3COW. The copies of data blocks for the new version exist only on disk not in memory. Any write to the file with a new version number from client creates a new physical version. The first step is to duplicate the inode which initially share all data block in common with the old one. The first time that a logical block in the new version is updated, allocates a new physical disk block to hold the data and subsequent updates to the same data with same version number are also written to the new block, preserving old block for the old version. When commit the write, we just need to commit the index blocks (direct blocks and indirect blocks) of new version and discard the old one.

Version number should save both in the EA and in inode of the object. Every write rpc to the object should be checked whether the write extent is intersect with the new version but from yet another client. As write extent from different clients should be disjoint, so when occured this case, it should involve the recovery or optimistic lazy commit.

6 Alternatives

6.1 Server based lock for parity

6.1.1 Definition

Server based lock is a kind of lock that the lock just sets in the namespace of lock server, doesn't cache in local namesapace of client (we can not execute lock match on client for this kind locks) and it is difficult from the callback lock which usually also enqueue the lock in the local namespace. Lustre uses callback extent lock to cache the data on client. But we can not do that for server-based extent lock.

As there is only one lock and one unlock message per high-level operation (update parity), the protocol is trivially two-phase and therefore serializable. And the lock server on OST will queues a client's lock request if there is an conflicting outstanding lock on the requested range. Once all the conflicting locks have been released, a response is returned to the client.

6.1.2 Server-based extent lock for parity control

The alternative strategy for laid locking is that use server based lock for parity which is a separated lock strategy from callback extent lock used by data. Server based lock for

parity control is for short durations when caches are flushed - for these locks the focus is on rapid acquiring and releasing when multiple writes involve the same file stripe group.

All the process under this lock strategy for LAID5 could be same as master locking except the process of updating and syncing the parity which is described as the following:

1. Acquire the server-based extent lock covered the syncing parity.
2. Preread the old parity data.
3. Compute the new parity and write to object.
4. Release the server-based extent lock.

6.1.3 Shortcomings of server-based lock for parity

Server based lock for parity has several shortcomings:

1. Old parity can not cache on client .
2. The parity could be only updated via method read-modify-write which would badly hurt the perfromance according to current LAID5 design.
3. Make the design of LAID5 complex.
4. Make the IO in degraded case more complex.

As we use server-based lock for parity, which means that parity can not be cached on client. Every time update the parity via method read-modify-write we must preread the parity blocks from OST, it would result in bad performance for small reprated write with high locality and random write.

The following is the reason why this lock strategy can not support reconstruct-write for writeback cache:

- Supposed that a stripe row on client A contains three strip units {D1, D2, P3} where P3 is the parity stripe unit. D1, D2 are marked as SU_DIRTY, which indicates it is a full write.
- At some time, client B wants to do a partial write to D1. At the blocking ast of lock callback, client A would just flush the cache D1 and also the old data cache of D1 should be discarded. Here we can not update the parity via reconstruct-write as the D2 may be diritied again in the future but we have no way to know that for mmap write, so we delay the parity update until the flushing of last pages in this stripe row (Details see the LAID5_HLD.lyx).
- After that, client A may begin to flush D2. But as the old cache of D1 is invalid, client A can only update the parity via method read-modify-write —It leads that the before write of D1 have not updated the parity because we are intented to update parity via method reconstruct-write.

6.1.4 Banded extent lock

To solve the problem above for full write, we propose a new lock strategy banded extent lock: When flush the caches in the extent of the callbacking extent lock, we also flush the caches in the same extent from other objects in the redundant set; At the same time update and sync the parity, which is very similar with the processing of extent lock with stripe-group-size granulairty (master locking); And then discard the old cache of data units. After that the blocking ast is counted as finished.

6.2 Update the parity on OST

6.2.1 Algorithm of updating the pairty on OST

In the LAID5_HLD.lyx, we also propose yet another design: update the parity on OST. It is based on the following formula: A + B + C = A + C + B, where the "+" is short for XOR operation. That is to say executing XOR operation out of order can get the same result. In this strategy, we needn't take any lock for parity control. The algorithm is described as follow:

1. Client just acquires callback extent lock for the data, not acquiring any lock for the parity.
2. Client also uses banded extent lock to handle the lock callback and flush the cache when occured lock conflict.
3. When client updates the parity, If all data units in the stripe row are in cache we can update via method reconstruct-write, and mark the parity page as PARITY_OVERWRITE; If it is a partial write, we calculate the P(update) = D(new) + D(old) and mark the parity page as PARITY_UPDATE.
4. On OST if the received parity page is marked as PARITY_OVERWRITE, just write to the disk; If it is marked as PARITY_UPDATE, we should first read the old pairty block P(old), and calculate the new parity P(new) = P(old) + P(update), and then write to disk.
5. For read in the degraded mode, the client still needs to acquire or extend extent locks from other objects with stripe-group-size alignment to preread the old data and reconstruct the unacessable blocks.

6.2.2 Advantage of updating parity on OST

- It doesn't need complex remastering process.
- Have no effect on the size management and maybe recovery.

6.2.3 Shortcoming of updating parity on OST

- Parity updating for partial write must finish on OST. It adds the OSTs' workload.
- To avoid confliction, OST must update and write the parity from various clients in a serializeable way.
- Processing of IO in degrade is very complex. E.x. we must take PR lock to cover the stipe row to read the old data and reconstruct the failed blocks in the read context when occur OST failure suddenly.
- When occured inconsistent write (casued by network problem), It is necessary to notify other clients opened the file by lock callback on MDS.

6.3 Comparison

We will give out a comparison of various lock strategies by a sample in the following. Assume that the stripping information of a file is that: stripe count 3, stripe_size 1M; *E* stands for the extent of a write request; Master object is obj1. ELCK is short for extent lock; MLCK is short for master locking; SLCK is short for server-based lock; NLCK is short for non-lock for parity.

```
LAID0      |                  LAID5       | stripe pattern
S1    S2    S3   |            S1    S2    P1    | stripe size    1M
S4    S5    S6   |            S3    P2    S4    | stripe size    1M
S7    S8    S9   |            P3    S5    S6    | stripe size    1M
...   ...   ...  |            ...   ...   ...   |
obj1  obj2  obj3 |            obj1  obj2  obj3  | stripe count   3
OST1  OST2  OST3 |            OST1  OST2  OST3  |
```

Pattern	Lock stragegy	Write extent	ELCK on obj1	ELCK on obj2	ELCK on obj3	
LAID0	Original ELCK	[1M, 4M-1]	[1M, 2M-1]	[0, 1M-1]	[1, 1M -1]	
LAID1	Master Locking	[1M, 4M - 1]	[1M, 4M - 1]	NULL	NULL	Just take l
LAID5	Mater Locking	[1M, 4M -1]	[0, 2M-1]	NULL	NULL	Just take l
LAID5	Master Locking	[1M, 3.5M-1]	[0, 2M-1]	NULL	NULL	Exte
LAID5	Service-based lock	[1M, 4M-1]	[1M, 2M-1]	[0, 1M-1]	[1M, 2M-1]	Need SLCK for
LAID5	Update on OSTs	[1M, 4M-1]	[1M, 2M-1]	[0, 1M-1]	[1M, 2M-1]	Non loc

7 Focus for inspections

- Does it have any serious bad effect on the current recovery scheme that taking extent lock with stripe-group-size granularity just from master object?
- Is there any opptimistic strategy for laid locking, such as time ordering? And any suggestions?