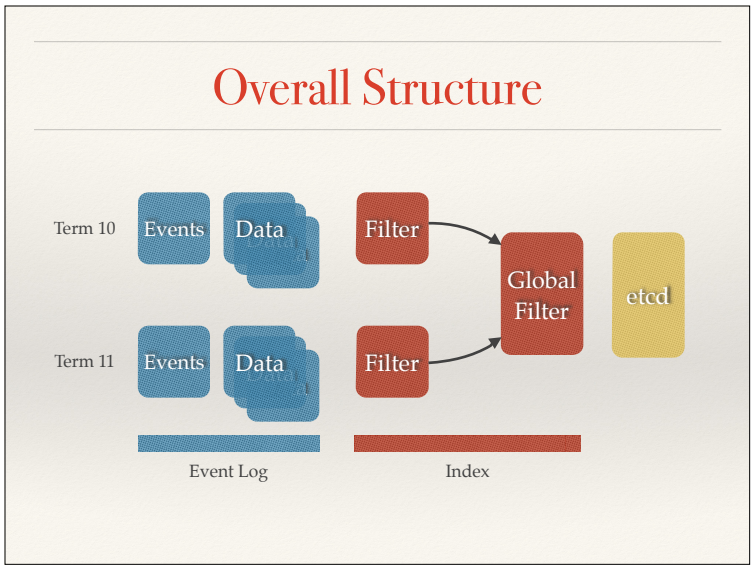
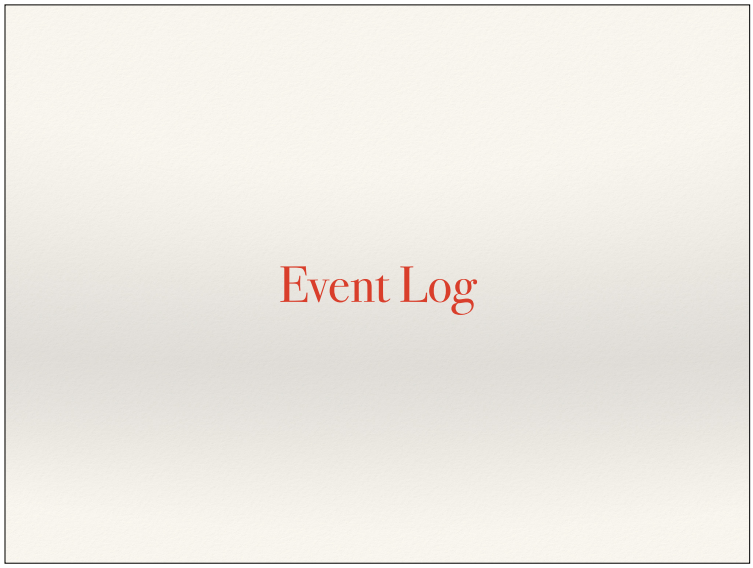
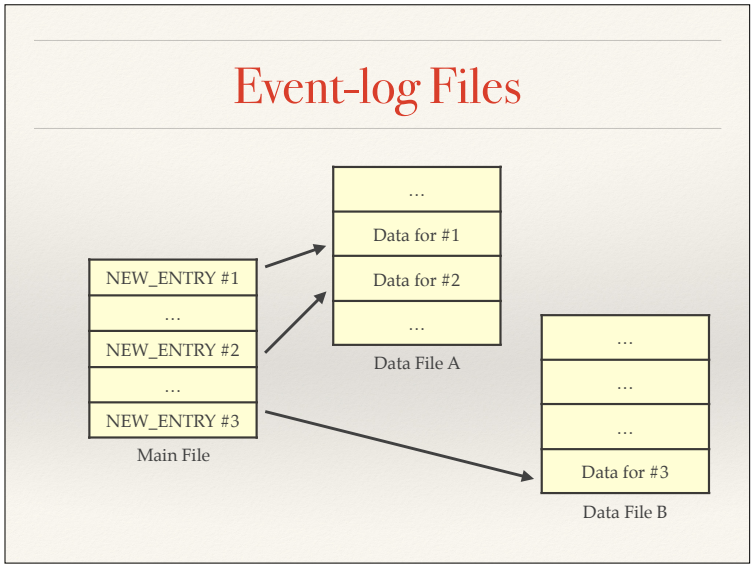


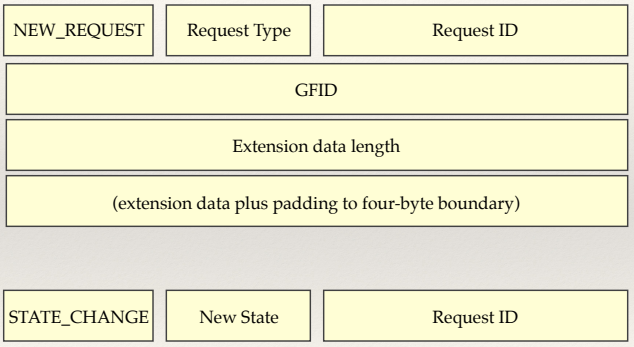
There are actually two parts to the journal - the event log and the index. The event log is divided into *terms* to support efficient space allocation and deallocation. Information about terms is stored in etcd. Information within terms is stored as a series of events corresponding to user I/O requests. The events themselves are stored in one file, while bulk data (e.g. for *writev*) is stored in one or more supplemental files.





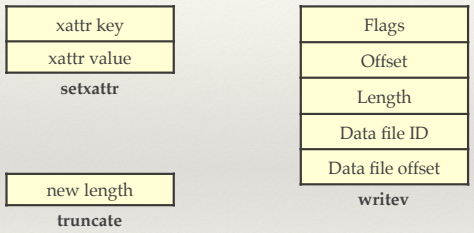
Each term's main event-log file contains short event records, which may contain pointers into one or more supplemental data files. Each such pointer consists of an ID identifying a particular data file, plus an offset into that file. Storing bulk data (especially that for *writev*) separately allows the main file to be scanned efficiently without having to read and then skip over megabytes of data that's not needed e.g. for reconciliation.

Event Format



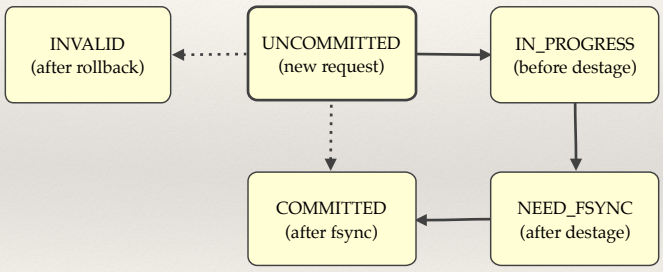
All entries in the event log start with a four-byte header, starting with an event type. The other contents of the header can vary depending on the event type. For the most common NEW_EVENT and STATE_CHANGE types, these contents include a request ID that is used to associate all related events with one another. In the special case of NEW_REQUEST, extra information describing the request follows the header, and is described on the next slide.

Extension Data

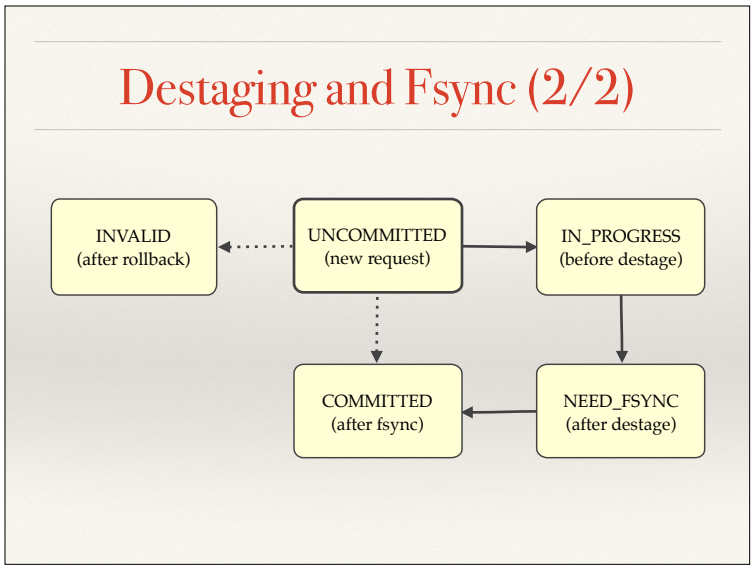


Extension data can take many forms, depending on the type of the user I/O request. These are just examples. Note that the *writev* example has two offsets - one within the actual file and one within the journal data file. The second offset, along with the data file ID, defines where the data landed within the journal before being destaged to the main file store.

Destaging and Fsync (1/2)

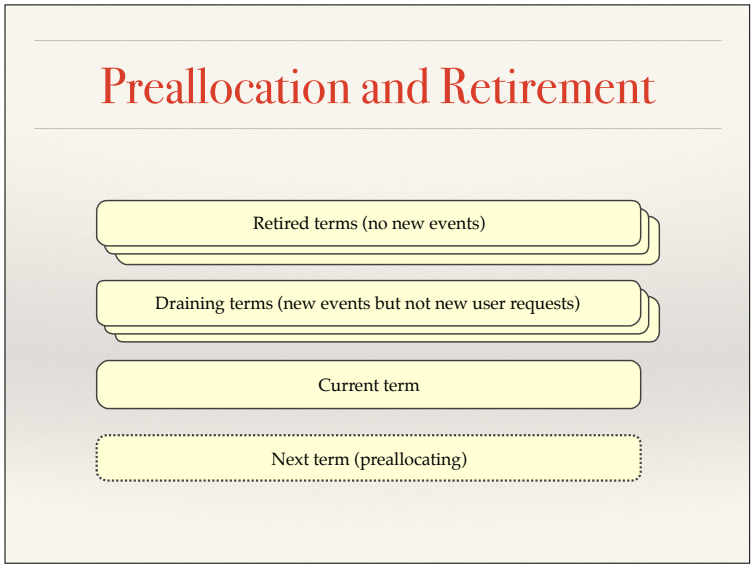


The life cycle of a request is shown above. As part of the higher-level protocol, the first NEW_REQUEST entry for a request puts it in UNCOMMITTED state. At our local discretion we destage, adding STATE_CHANGE events to IN_PROGRESS and NEED_FSYNC around the actual main-store operation. Finally, when a user's *fsync* request comes in (or at our local discretion) we issue a local *fsync* and add a final STATE_CHANGE(COMMITTED) event.
(continued)

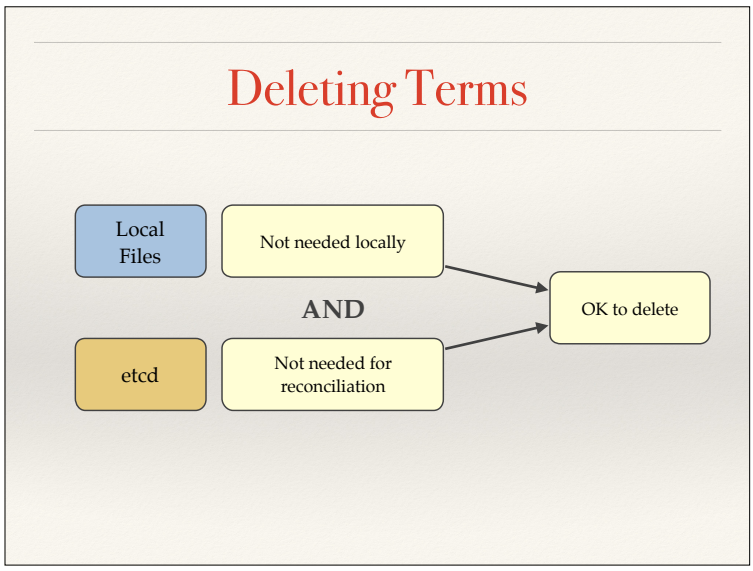


(continued)

There are two exceptions to the normal life cycle, both shown with dotted lines above. First, if a write is done with O_SYNC, the NEW_EVENT immediately puts the request into COMMITTED state with no further need to destage or *fsync*. Second, if a rollback message is received before the request gets to IN_PROGRESS, we quash the request by adding a STATE_CHANGE(INVALID) event.



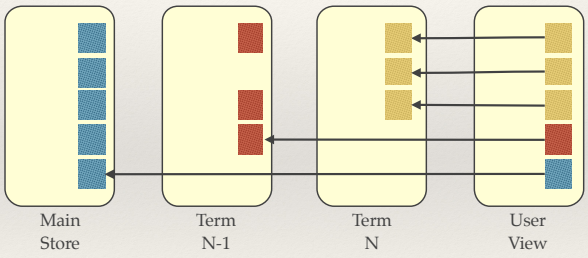
There will usually be multiple terms "in play" at once. The current term and next term are pretty self-explanatory. When a term stops being current, it stops collecting NEW_REQUEST events (those will go into the new current term) but might still collect other internal events related to destaging or reconciliation. When even those events have finished, the term is considered completed and is kept around only for reconciliation.



A term can only be deleted when two conditions are met - it's not needed locally (i.e. all data has been fully destaged and fsync'ed) and it's not needed for reconciliation. Therefore, we must check for eligibility whenever either of these conditions change.

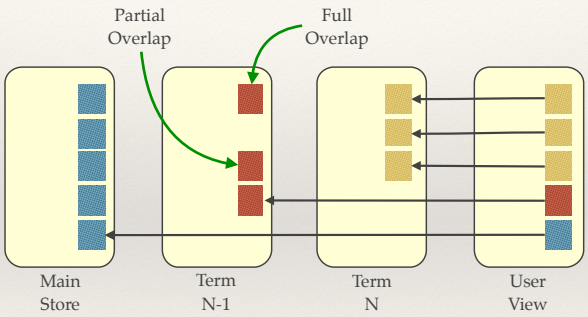
Index

Overlapping Operations



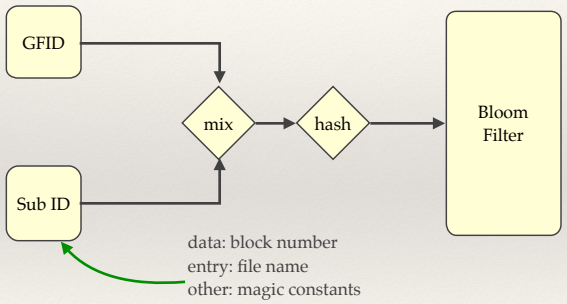
Because there might be an indefinite amount of time before writes are destaged from the journal into the main store, we need to account for in-journal data on reads. Each term therefore acts as on overlay on everything previous, "hiding" any older content from view. In the diagram above, only the blocks with arrows pointing to them represent data the user can still see. All other blocks are hidden.

Deleting Overlaps



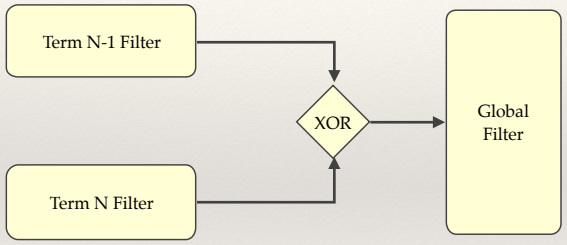
Because data which is "covered" by a write in a later term is no longer useful even for reconciliation, it shouldn't be included in filters (described later) either. Special OVERLAPPED events, similar to ROLLBACK events, are added to reflect such changes. Note that for a partial overlap this event might truncate or split the original event instead of quashing it entirely. These optimizations can all be done locally on each replica.

Term Filters



We maintain and store a Bloom filter for each term, to enable efficient determination of whether the term contains anything at all relevant to a subsequent read (or reconciliation). Each data block, directory entry, or metadata field is treated as a separate entity for filter purposes, even if a single request affects multiple entities. For each entity, we calculate a hash and use that to populate the term's Bloom filter.

Global Filter



We also maintain a global filter, to determine quickly whether any term still contains relevant data for a read or reconciliation. To avoid "filter pollution" the global filter is kept at a coarser granularity than per-term filters by using only GFIDs without futher refinement by sub-ID (e.g. block number). We do calculate and store GFID-only filters for each term, but only to enable (re)calculation of the the global filter.

Reconciliation

Tracking Reconciliation

	Replica A (leader)	Replica B	Replica C (dead)
Term 1234	9902	5319	9902
Term 1235	3122	3122	6157
Term 1236 (current)	0	0	0

To track partial reconciliations and know when the results eventually converge, we maintain a table in etcd of unique IDs for each reconciliation that has occurred. When all numbers for a term are equal and non-zero, we're fully reconciled. In the above table, the two colored cells show which replicas are still in need of reconciliation for which term. We might be able to complete reconciliation for B, so we try, but we can't do anything about C while it's down.

Complete Reconciliation

	Replica A (leader)	Replica B	Replica C (dead)
Term 1234	9902	9902	9902
Term 1235	3122	3122	6157
Term 1236 (current)	0	0	0

In the above table, we have reconciled between A and B for term 1234, and found no differences. We therefore conclude that the two reconciliation IDs represent equal states and update B's ID (from A's) to reflect that. Because C already had the now-shared reconciliation ID, term 1234 is now completely reconciled.

Incomplete Reconciliation

	Replica A (leader)	Replica B	Replica C (dead)
Term 1234	1740	1740	9902
Term 1235	3122	3122	6157
Term 1236 (current)	0	0	0

This time we attempted the same reconciliation as before, but found that there were differences between A and B. Therefore we generate a new reconciliation ID (1740) to represent equality among the participants, and also represent the new inequality between A and C (which still has the old ID 9902). C is now the one that's out of date relative to everyone else, so replication isn't complete yet, but it can be completed by reconciling with either A or B and finding no changes.