File Systems – Journaling

Dongkun Shin (dongkun@skku.edu)
Embedded Software Laboratory
Sungkyunkwan University
http://nyx.skku.ac.kr/

EXT3 Journaling (Write-Ahead Logging)

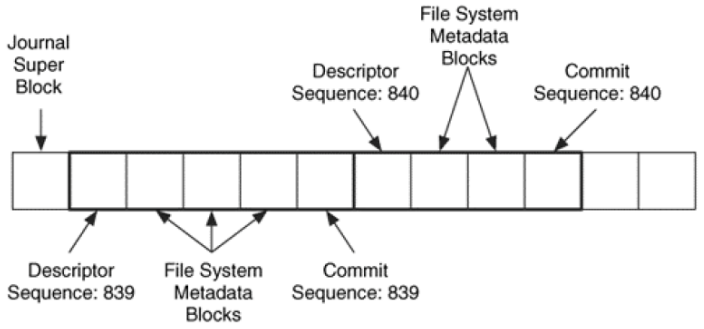
- Journaling Filesystems
  - Updates to filesystem blocks might be kept in dynamic memory for long period of time before being flushed to disk
  - A dramatic event such as a power-down failure or a system crash might thus leave the filesystem in an inconsistent state
  - To overcome this problem, each traditional Unix filesystem is checked before being mounted ➔ too long time
  - avoid running time-consuming consistency checks on the whole filesystem
  - Instead, look in a special disk area that contains the most recent disk write operations named journal

EXT3 Journaling

- The idea behind Ext3 journaling
  - First, a copy of the blocks to be written is stored in the journal
  - When the data is committed to the journal, the blocks are written in the filesystem
- When system failure occurred before a commit to the journal
  - Either the copies of the blocks relative to the high-level change are missing from the journal or they are incomplete;
  - e2fsck ignores journals. ➔ undo
- When system failure occurred after a commit to the journal
  - The copies of the blocks are valid, and e2fsck writes journals into the filesystem. ➔ redo

EXT3 Journal Layout

- Journal superblock
  - The first block in the journal
  - contains the first logging data address and its sequence number.
- Updates are done in transactions, and each transaction has a sequence number.
- Journal descriptor block
  - Each transaction starts with a descriptor block
  - contains the transaction sequence number and a list of what blocks are being updated.
- Following the descriptor block are the updated blocks.
- When the updates have been written to disk, a commit block is written with the same sequence number.



EXT3 Journaling Modes

- Data
  - All filesystem data and metadata changes are logged into the journal.
  - requires many additional disk accesses, safest and slowest
- Ordered
  - Only changes to filesystem metadata are logged into the journal.
  - Data blocks are written to disk before making changes to associated filesystem metadata
  - Default Ext3 journaling mode.
- Writeback
  - Only changes to filesystem metadata are logged; this is the method found on the other journaling filesystems and is the fastest mode.
- %mount –t ext3 –o data=writeback /dev/sda2 /jdisk

EXT3 Journaling Modes

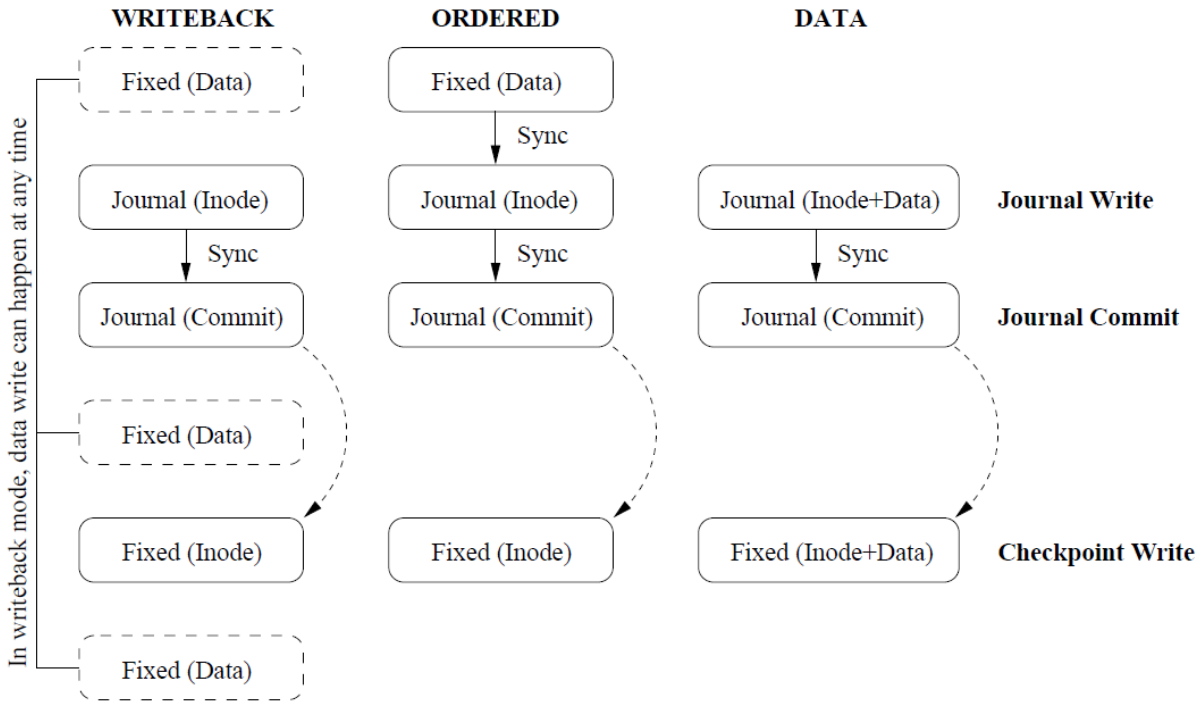


Figure 1: Ext3 On-Disk Layout.

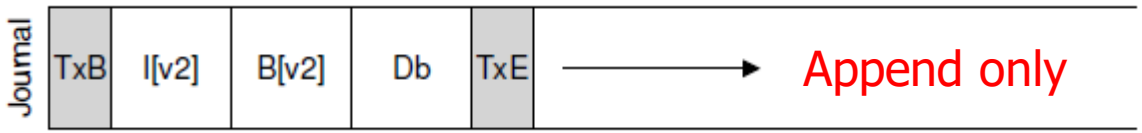
Data Journaling

1. Journal write

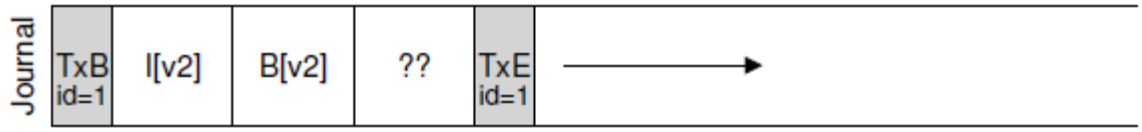
- Write the transaction to the log
  - transaction-begin block (TxB): transaction identifier (TID), information about the pending update (e.g., the final addresses of I[v2], B[v2], Db)
  - all pending data and metadata updates: the exact contents of the blocks themselves (physical logging, cf. logical logging)
  - transaction-end block (TxE): marker of the end of this transaction, and also contain the TID
- Wait for these writes to complete.

2. Checkpoint

- Write the pending metadata/data updates to their final locations.
- writes I[v2], B[v2], and Db to their disk locations

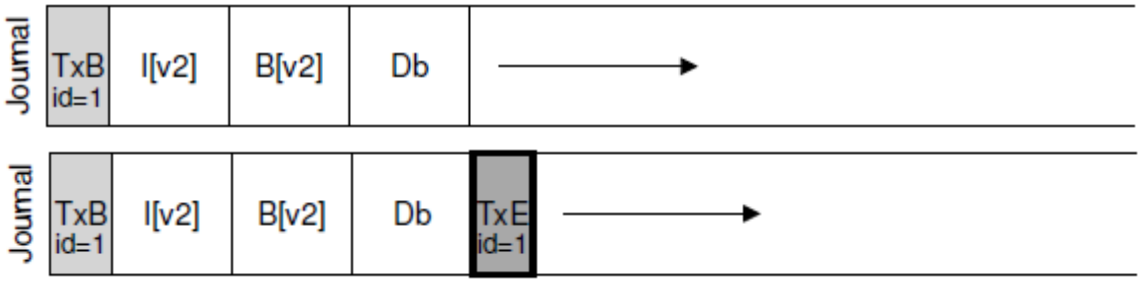


- **Crash** during the writes to the journal.
  - Disk internally may perform scheduling and complete small pieces of the big write in any order



Data Journaling

- Two steps of transactional write
  - writes all blocks except the TxE block to the journal
  - When those writes complete, the file system issues the write of the TxE block

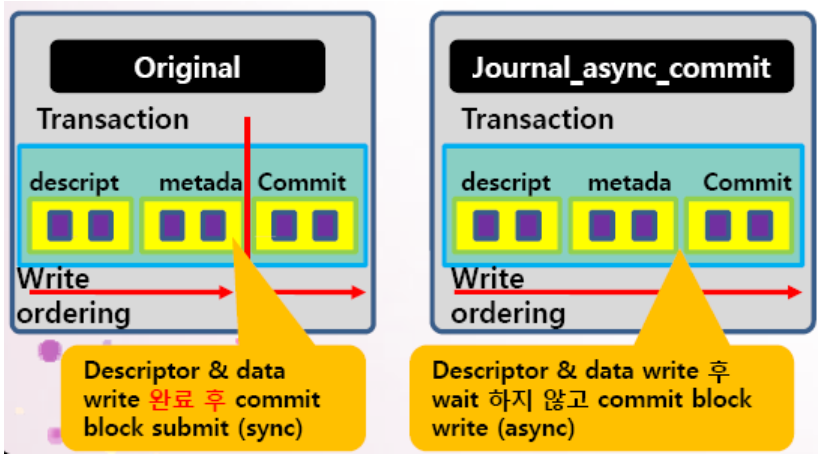


Due to write buffer in disk, we need write barriers

- Three phases
  - 1. Journal write: Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete (write barrier, flush)
  - 2. Journal commit: Write the transaction commit block (containing TxE) to the log; wait for write to complete (flush or atomic 512B); transaction is said to be committed.
  - 3. Checkpoint: Write the contents of the update (metadata and data) to their final on-disk locations.

journal_async_commit

- Asynchronous
  - JC write does not wait the journal data write
- Improve performance
- Use journal_checksum

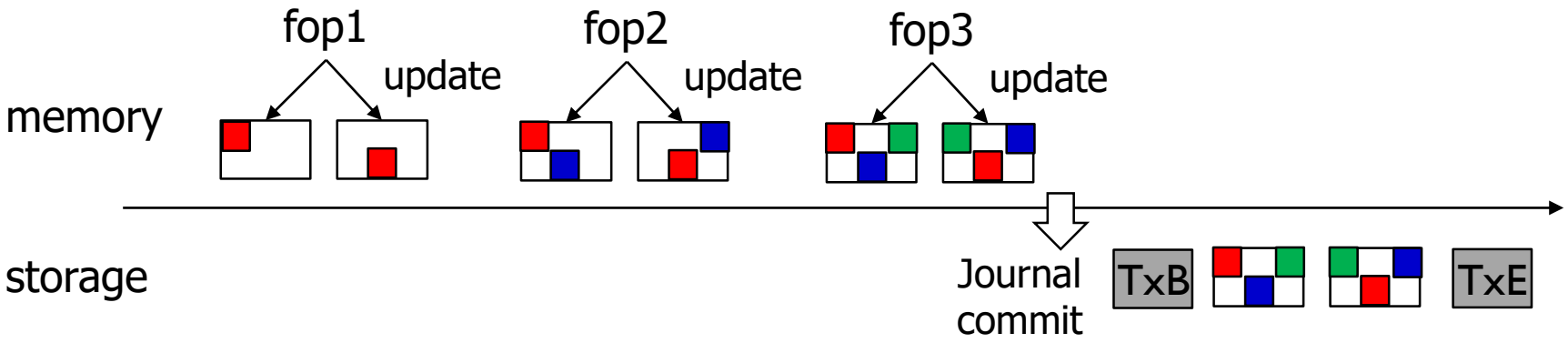


Recovery

- If the crash happens before the transaction is written safely to the log (i.e., before the journal commit completes)
  - The pending update is simply skipped.
- If the crash happens after the transaction has committed to the log, but before the checkpoint is complete
  - The file system can recover the update.
  - Redo logging: when the system boots, the recorded transactions are replayed (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations.
- Crash may happen at any point during checkpointing
  - some of the updates to the final locations of the blocks have completed.
  - These updates are simply performed again during recovery.

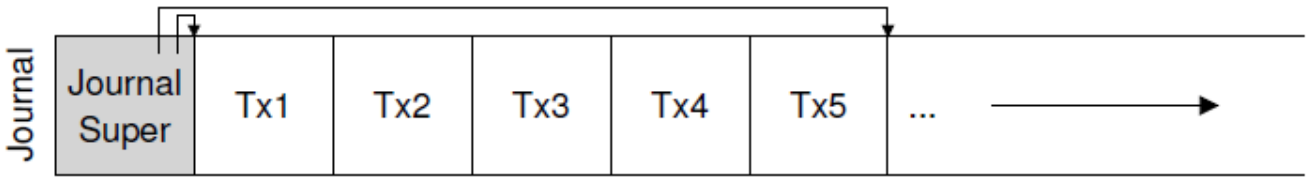
Compound transaction

- Rather than commit each update to disk one at a time; all updates can be buffered into a global transaction (e.g., Linux ext3)
- File system just marks the in-memory data structures as dirty, and adds them to the list of blocks that form the current transaction.
- When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates.
- Can avoid excessive write traffic to disk



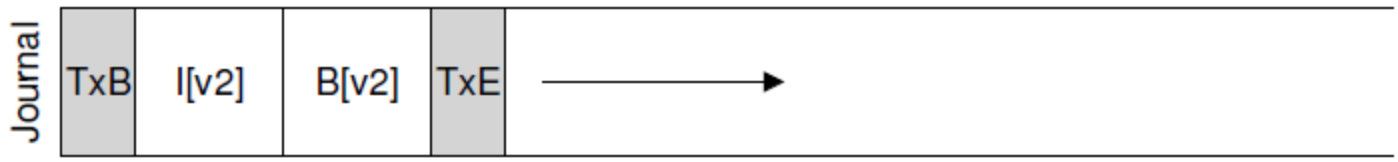
Making The Log Finite

- The write-ahead log is of a finite size. If we keep adding transactions to it, it will soon fill.
  - The larger the log, the longer recovery will take
  - When the log is full, no further transactions can be committed to the disk.
- Journaling file systems treat the log as a circular data structure, re-using it over and over ➔ circular log
- Once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused.
- Mark the oldest and newest non-checkpointed transactions in the log in a journal superblock



Metadata Journaling

- In data journaling, for each write to disk, we are also writing to the journal first, thus doubling write traffic
  - especially painful during sequential write workloads
  - there is a costly seek between writes to the journal and writes to the main file system
- Ordered journaling (metadata journaling)
  - user data is not written to the journal
  - The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write
  - reduces the I/O load of journaling
  - When should we write data blocks to disk?
  - If we write Db to disk after the transaction (containing I[v2] and B[v2]) completes, the file system is consistent but I[v2] may end up pointing to garbage data.



Metadata Journaling

- Write the pointed-to object before the object that points to it
  - write data blocks (of regular files) to the disk first, before related metadata is written to journal
  - can guarantee that a pointer will never point to garbage

- Ordered journaling (Linux ext3/4, Windows NTFS, SGI' XFS)

  1. Data write: Write data to final location; wait for completion
  (the wait is optional; Step 1 must complete before Step 3).

  2. Journal metadata write: Write the begin block and metadata to the log; wait for writes to complete.

  3. Journal commit: Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now committed.

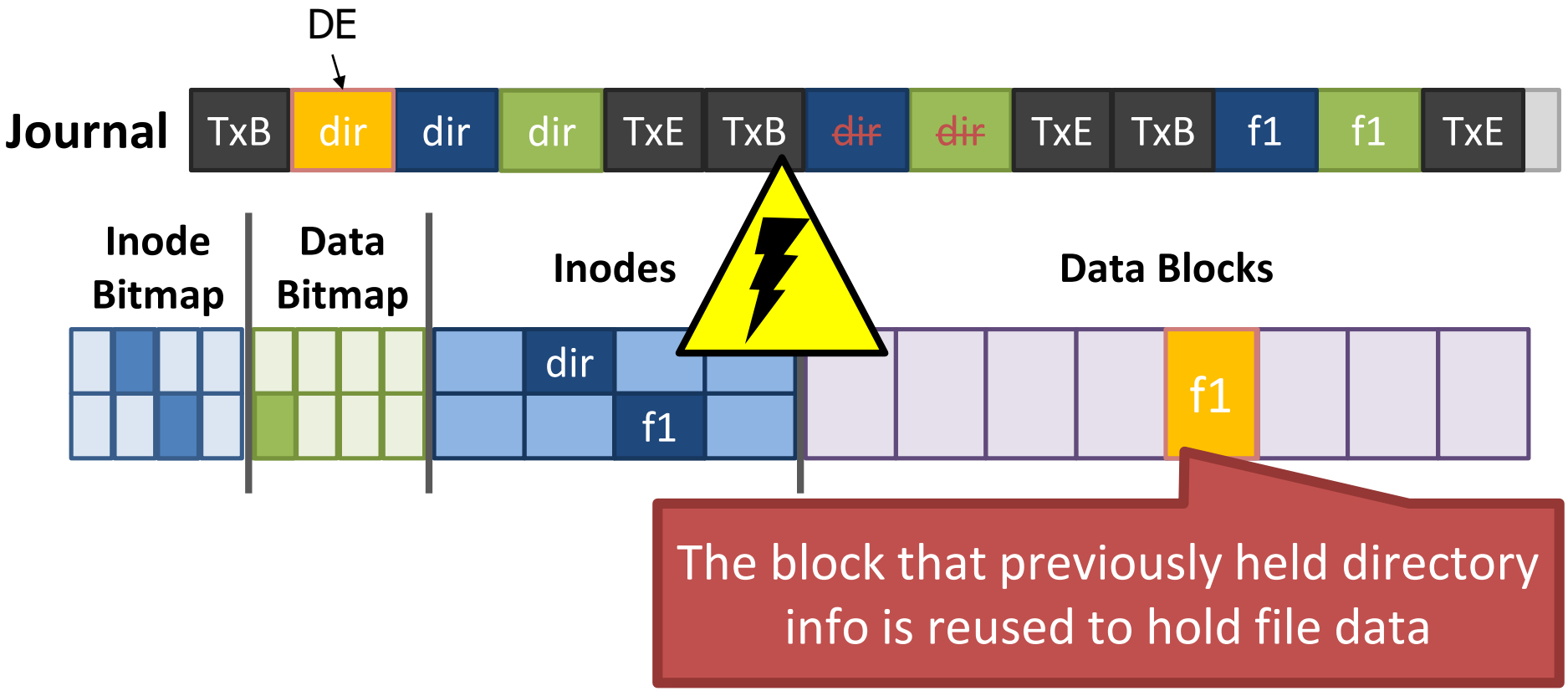
  4. Checkpoint metadata: Write the contents of the metadata update to their final locations within the file system.

  5. Free: Later, mark the transaction free in journal superblock.

- Ext3 supports also the unordered journaling (writeback journaling)

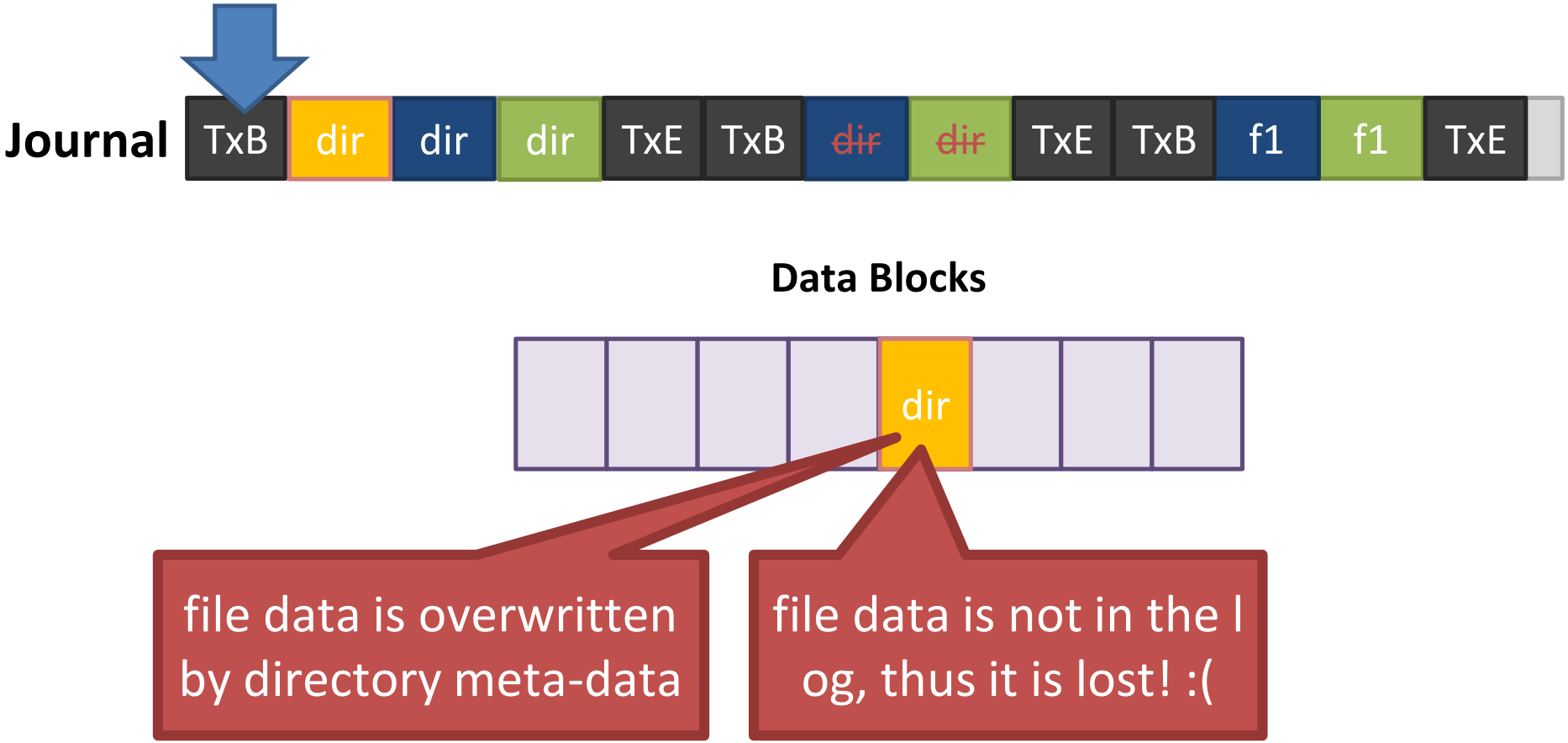
Delete and Block Reuse in metadata journaling

1. Create a directory: inode and data are written
2. Delete the directory: inode is removed
3. Create a file: inode and data are written



The Trouble With Delete

- What happens when the log is replayed?



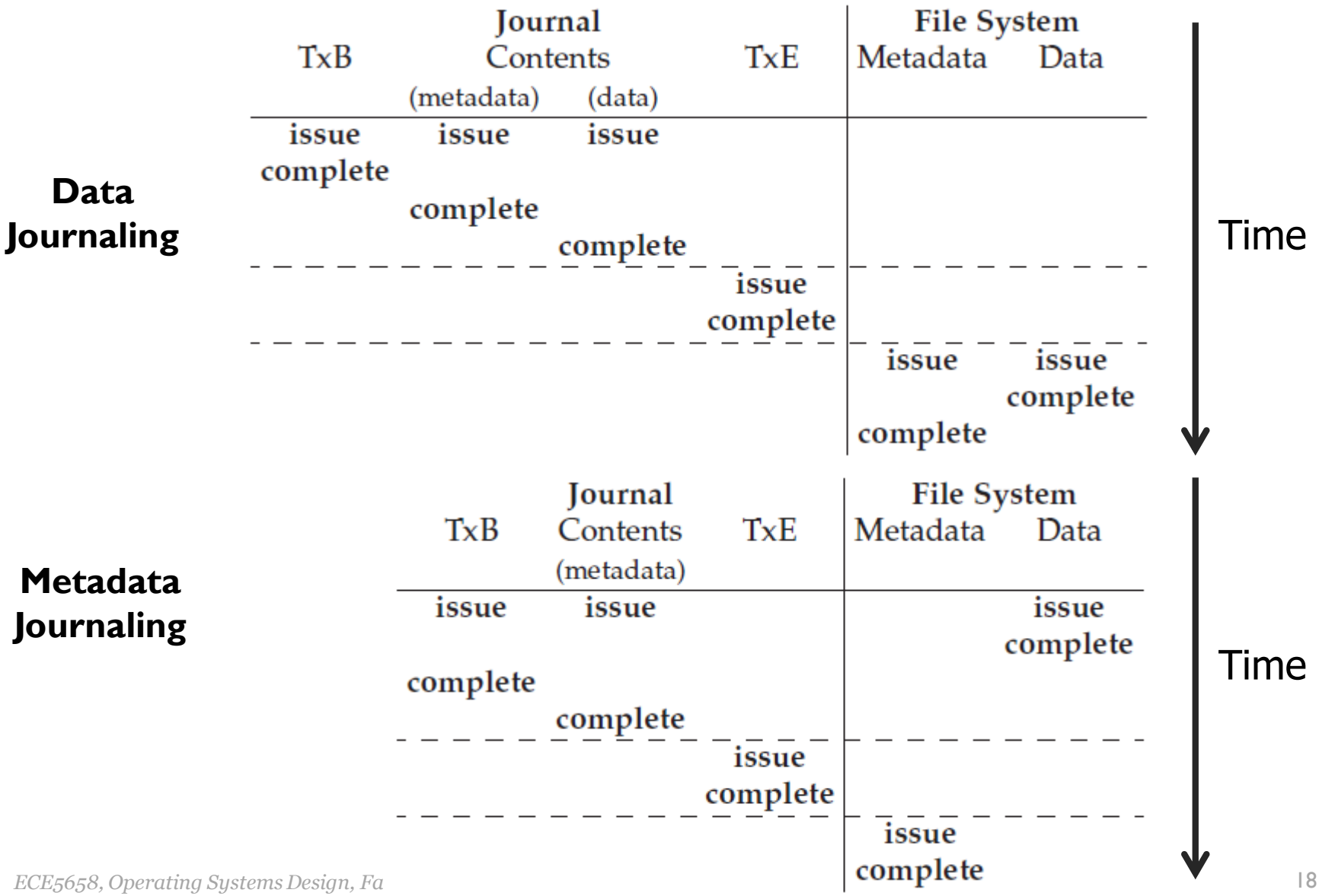
Handling Delete

- Strategy 1: don't reuse blocks until the delete is checkpointed and freed
- Strategy 2: add a revoke record to the log
  - ext3 used revoke records

Journal

If the log is replayed, ign ore DE block 100

Timeline


Data Journaling
Journal
TxB
Contents
(metadata)
(data)
TxE
File System
Metadata
Data
issue
complete
issue
complete
issue
complete
issue
complete
issue
complete
issue
complete
Time
Metadata Journaling
Journal
TxB
Contents
(metadata)
TxE
File System
Metadata
Data
issue
complete
issue
complete
issue
complete
issue
complete
issue
complete
Time

Performance of Journaling

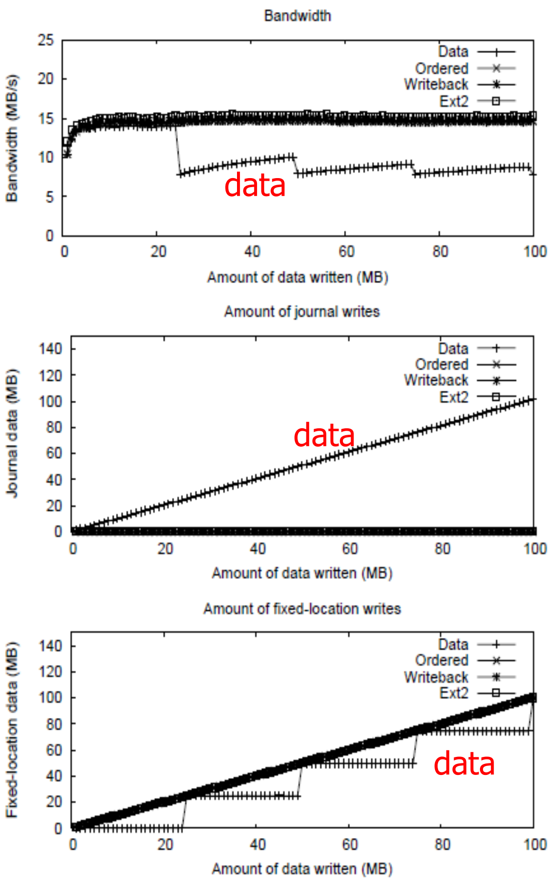


Figure 3: **Basic Behavior for Sequential Workloads in ext3.** *Within each graph, we evaluate ext2 and the three ext3 jour- na … x- … then performs an fsync. Each graph examines a different met- ric: the top graph shows the achieved bandwidth; the middle graph uses SBA to report the amount of journal traffic; the bot- tom graph uses SBA to report the amount of fixed-location traffic. The journal size is set to 50 MB.*

Sequential write then fsync

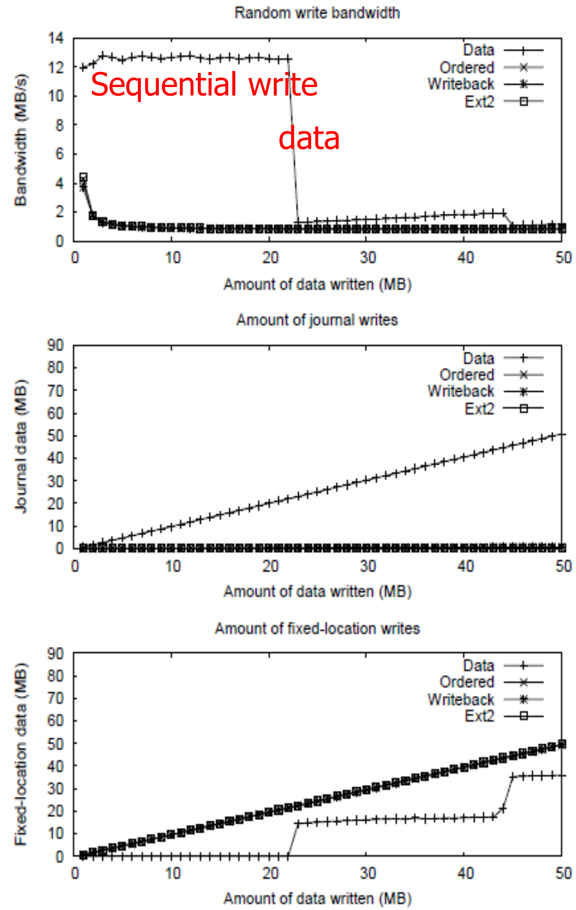


Figure 4: **Basic Behavior for Random Workloads in ext3.** *This figu … issues 4 KB writes to … fsync once for every … idth, middle graph sh … graph reports the fixed-location traffic. The journal size is set to 50 MB.*

Random 4K write
+ fsync after 256 writes

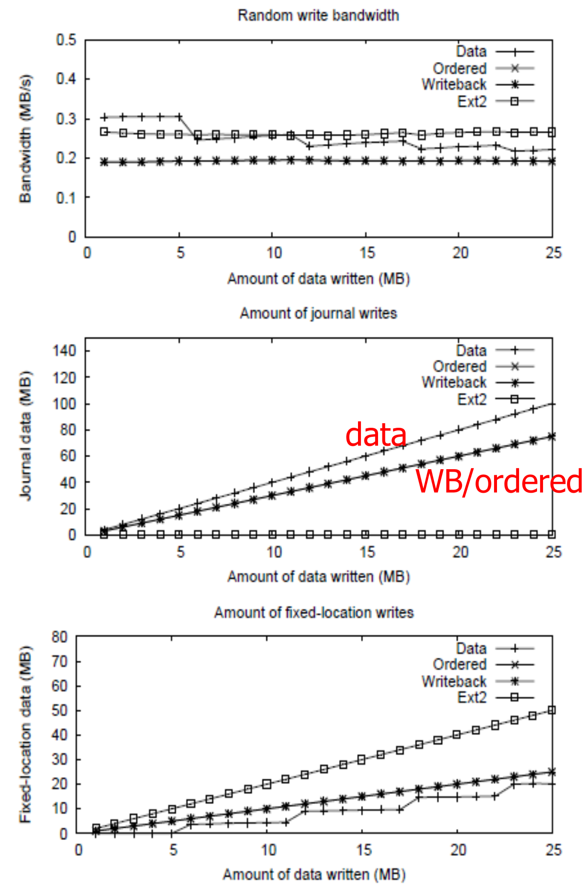


Figure 5: **Basic Behavior for Random Workloads in ext3.** *This figure i … oad issues 4 KB random writ … te. Bandwidth is shown in the … d-location writes are reported … g SBA. The jour- nal size is set to 50 MB.*

Random 4K write
+ fsync per writes

V. Prabhakaran, "Analysis and evolution of journaling file systems," USENIX ATC'05

Performance of Journaling

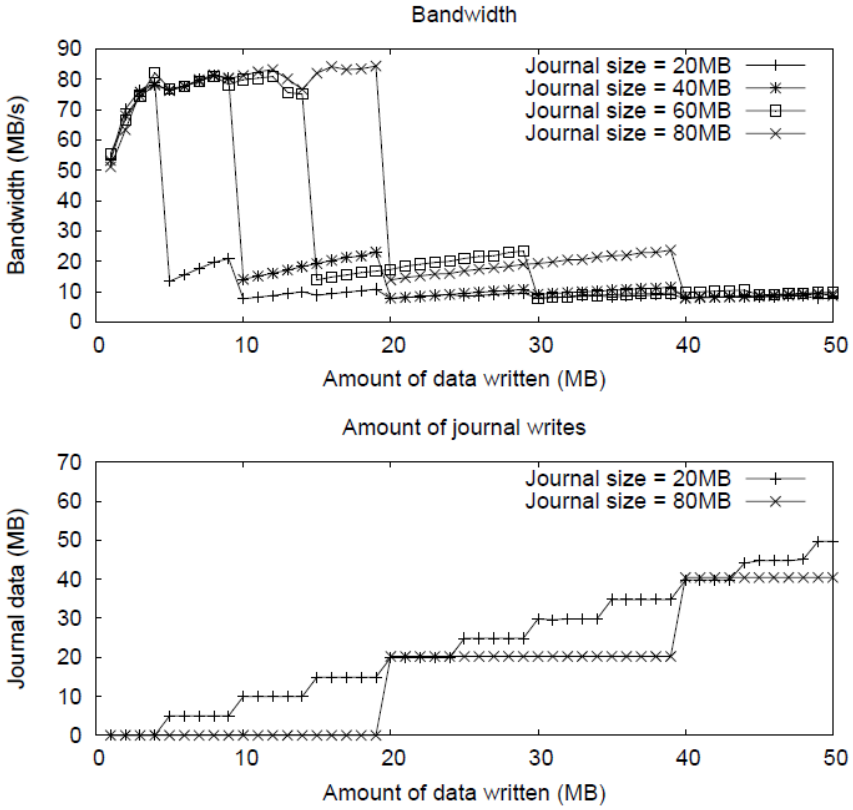
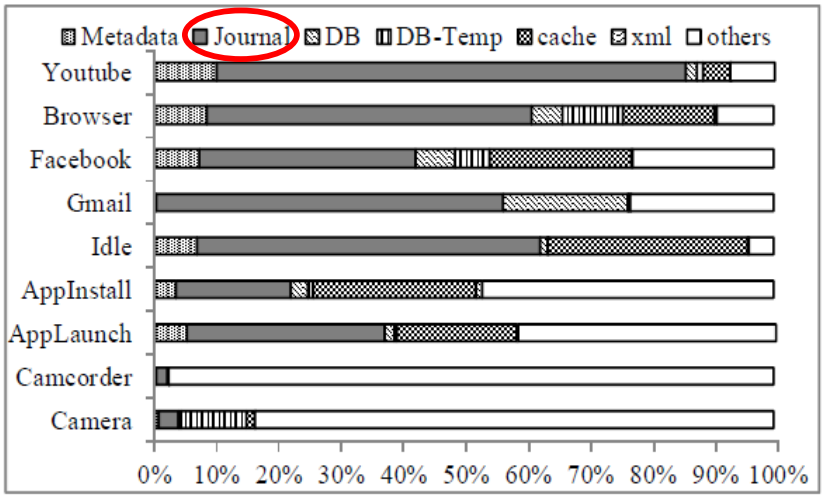
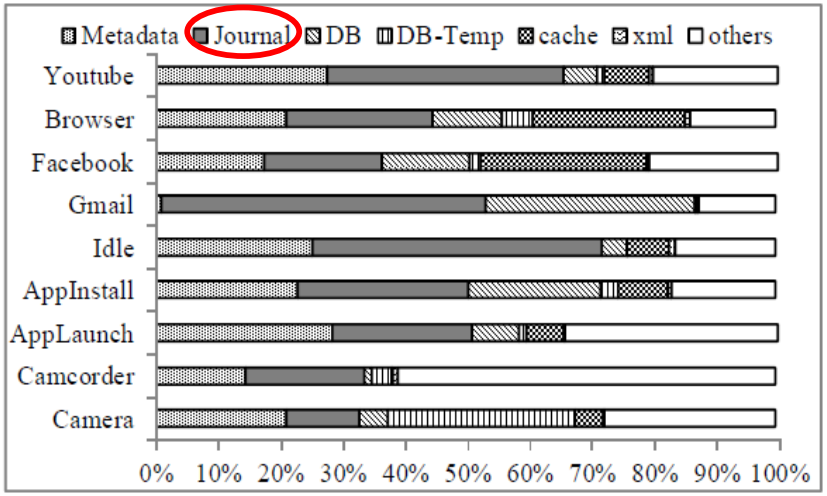


Figure 7: **Impact of Journal Size on Commit Policy in ext3.** *The topmost figure plots the bandwidth of data journaling mode under different-sized file writes. Four lines are plotted representing four different journal sizes. The second graph shows the amount of log traffic generated for each of the experiments (for clarity, only two of the four journal sizes are shown).*

Android Workload



(a) distributions of written sectors

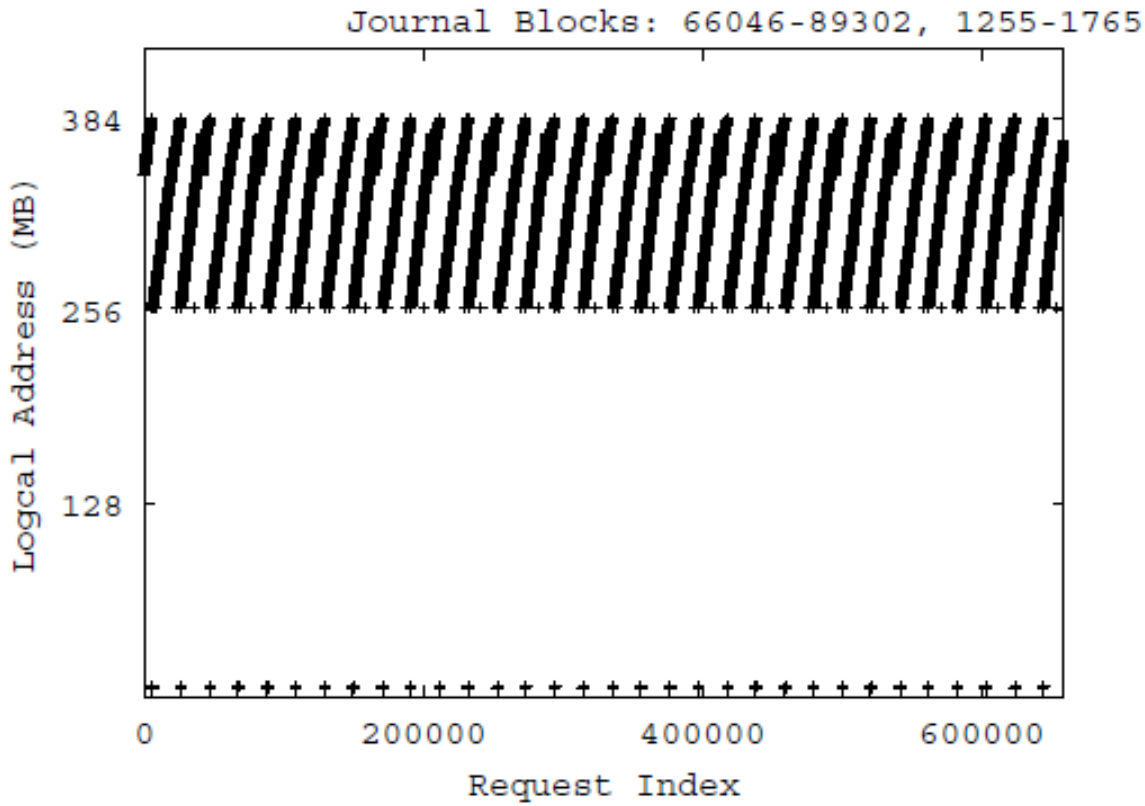


(b) distributions of write requests

	Average size of write requests (blocks)							Average number of overwrites per block						
	meta	jrnl	DB	temp	cache	xml	others	meta	jrnl	DB	temp	cache	xml	others
Youtube	1.4	7.4	1.2	3.3	2.2	1.1	1.4	18.0	1	12.0	1.0	1.0	1	2.0
Browser	1.0	5.8	1.2	5.2	1.6	0.9	1.8	39.0	1	248.0	3.0	5.0	1	2.0
Facebook	1.2	5.5	1.3	9.6	2.5	1.1	3.3	9.8	1	8.6	1.8	2.2	1	2.2
Gmail	0.8	2.8	1.6	22.0	2.9	1.0	4.7	13.0	2.1	21.3	1.0	2.3	10	1.3
Idle 10Hrs	1.3	5.6	1.3	5.8	22.5	1.1	1.3	40.2	1.2	13.0	1.3	2.0	1	1.8
AppInstall	1.3	5.8	1.1	2.5	28.6	11.5	23.8	22.0	1	25.0	4.0	2.0	1	1.0
AppLaunch	1.1	8.3	1.2	2.7	19.4	1.6	7.2	44.0	1	7.8	2.2	2.0	1	2.8
Camcorder	1.1	5.9	1.5	2.5	1.0	1.5	95.8	2.9	1	5.0	1.7	1.0	1	1.1
Camera	1.1	7.8	2.6	10.1	8.2	1.0	86.3	3.9	1	1.8	2.2	1.5	1	1.2

Android Workload

- EXT4 journal write pattern (App. install workload)
  - Overwritten by round robin fashion.



JBD2 on-disk journal data structure

Journal Descriptor Block

```
typedef struct journal_header_s
{
        __be32      h_magic;
        __be32      h_blocktype;
        __be32      h_sequence;
} journal_header_t;
```

```
typedef struct journal_block_tag_s
{
        __be32      t_blocknr;
        __be16      t_checksum;
        __be16      t_flags;
        __be32      t_blocknr_high;
} journal_block_tag_t;
```

block number at home location

Journal Commit Block

```
struct commit_header {
        __be32          h_magic;
        __be32          h_blocktype; //JBD2_COMMIT_BLOCK
        __be32          h_sequence;
        unsigned char   h_chksum_type;
        unsigned char   h_chksum_size;
        unsigned char   h_padding[2];
        __be32          h_chksum[JBD2_CHECKSUM_BYTES];
        __be64          h_commit_sec;
        __be32          h_commit_nsec;
};
```

JBD2 Transaction data structure

- atomic handle
  - Include all metadata blocks updated by a single file operation
  - Need to be persistent atomically
- transaction
  - For efficiency, several atomic handles are grouped into a single transaction (compound transaction)
  - written to the journal after a fixed amount of time elapses or there is no free space left on the journal to fit it.
- jinode
  - Tracking the target inode

JBD2 Journal structure

```
struct journal_s
{
        …
        struct buffer_head      *j_sb_buffer;
        journal_superblock_t    *j_superblock;
        transaction_t           *j_running_transaction;
        transaction_t           *j_committing_transaction;
        transaction_t           *j_checkpoint_transactions;
        struct buffer_head      *j_chkpt_bhs[JBD2_NR_BATCH];
        unsigned long           j_head;
        unsigned long           j_tail;
        unsigned long           j_free;  /* # of free blocks in the journal */
        struct block_device     *j_dev;
        struct inode            *j_inode;
        unsigned long           j_commit_interval;
        ...
}
```

JBD2 Transaction structure

```
struct transaction_s
{
        journal_t           *t_journal;       /* Pointer to the journal for this tx */
        tid_t               t_tid;            /* Sequence number for this tx */
        enum {  T_RUNNING, T_LOCKED, T_FLUSH, T_COMMIT,
                T_COMMIT_DFLUSH, T_COMMIT_JFLUSH,T_COMMIT_CALLBACK,
                T_FINISHED
        }                   t_state;          /* Tx's current state */
        unsigned long       t_log_start;      /* this tx's commit start location */
        int                 t_nr_buffers;     /* # of buffers on the t_buffers list */
```

JBD2 Transaction structure

```
 /* buffers reserved but not yet modified by this tx */
struct journal_head     *t_reserved_list;
 /* metadata buffers owned by this tx */
struct journal_head     *t_buffers;
 /* forget buffers (we can un-checkpoint once this tx commits) */
struct journal_head     *t_forget;
 /* buffers still to be flushed before this tx can be checkpointed */
struct journal_head     *t_checkpoint_list;
 /* buffers submitted for IO while checkpointing */
struct journal_head     *t_checkpoint_io_list;
 /* metadata buffers being shadowed by log IO */
struct journal_head     *t_shadow_list;
 /* inodes whose data we've modified in data=ordered mode. */
struct list_head        t_inode_list;
```

JBD2 Transaction structure

```
spinlock_t      t_handle_lock;          /* Protects info related to handles */
unsigned long   t_max_wait;  /* Longest time some handle had to wait for running Tx */
unsigned long   t_start, t_requested;  /* When Tx started, commit was requested */
struct transaction_chp_stats_s t_chp_stats;   /* Checkpointing stats  */
atomic_t        t_updates;              /* # of outstanding updates running on this Tx */
atomic_t        t_outstanding_credits; /* # of buffers reserved but not yet modified */
transaction_t *t_cpnext, *t_cpprev;  /* Forward/backward links of Txs awaiting CP */
unsigned long       t_expires;          /*  When will the Tx expire in jiffies?  */
ktime_t         t_start_time;           /*  When this Tx started, in nanoseconds */
atomic_t        t_handle_count;         /*  How many handles used this Tx?  */
unsigned int t_synchronous_commit:1;    /*  Some process is waiting for it to finish. */
int             t_need_data_flush;          /*  Disk flush needs to be sent */
struct list_head    t_private_list;         / * To store fs-specific data  structures */
} transaction_t;
```

JBD2 Transaction State

- T_RUNNING
  - New handles can be added
- T_LOCKED
  - Waiting for commit
- T_FLUSH
  - Flush data blocks (ordered mode)
- T_COMMIT
  - Flush JD block and metadata blocks

EXT4 JBD2 APIs

- ext4_journal_start() ➔ jbd2_journal_start()
  - Initialize journal handle for file operation
  - start_this_handle()
    - make sure that there is enough journal space for the handle to begin
- ext4_journal_get_write_access() ➔ jbd2_journal_get_write_access()
  - Add buffer head to journal handle
    - transaction->t_reserved_list

EXT4 JBD2 APIs

- ext4_handle_dirty_metadata() ➔ jbd2_journal_dirty_metadata()
  - Called when the registered buffer head is modified
  - Set jbddirty
    - Will not be flushed by pdflush
  - Remove bh from transaction->t_reserved_list
  - Add bh to transaction->t_buffers
- ext4_jbd2_file_inode() ➔ jbd2_journal_file_inode()
  - To manage user data blocks which must be flushed before journal blocks in the ordered mode
  - Add inode to transaction-> t_inode_list

EXT4 JBD2 APIs

- ext4_journal_stop() ➔ jbd2_journal_stop()
  - Close journal handle at the end of file operation
- jbd2_journal_flush()
  - called at any time to commit and checkpoint transactions

EXT4 JBD2 APIs