Transaction Recovery: Advanced Topics & Implementation

Caetano Sauer

caetano.sauer@salesforce.com

About me

- 2008-2012: Moved to Germany from Brazil, BSc and MSc degrees at TU Kaiserslautern
- 2012-2017: Obtained PhD at TU Kaiserslautern
  - Topic: Transaction Recovery (today's lecture!)
- 2017-today: Software Engineer in Hyper Team in Munich (Tableau/Salesforce)

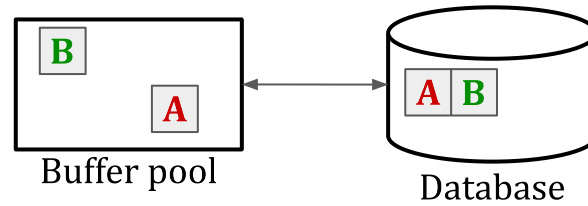
Agenda

- Recap of the basics
- Improving transaction throughput
- Faster recovery
- Availability while recovering
- Media recovery

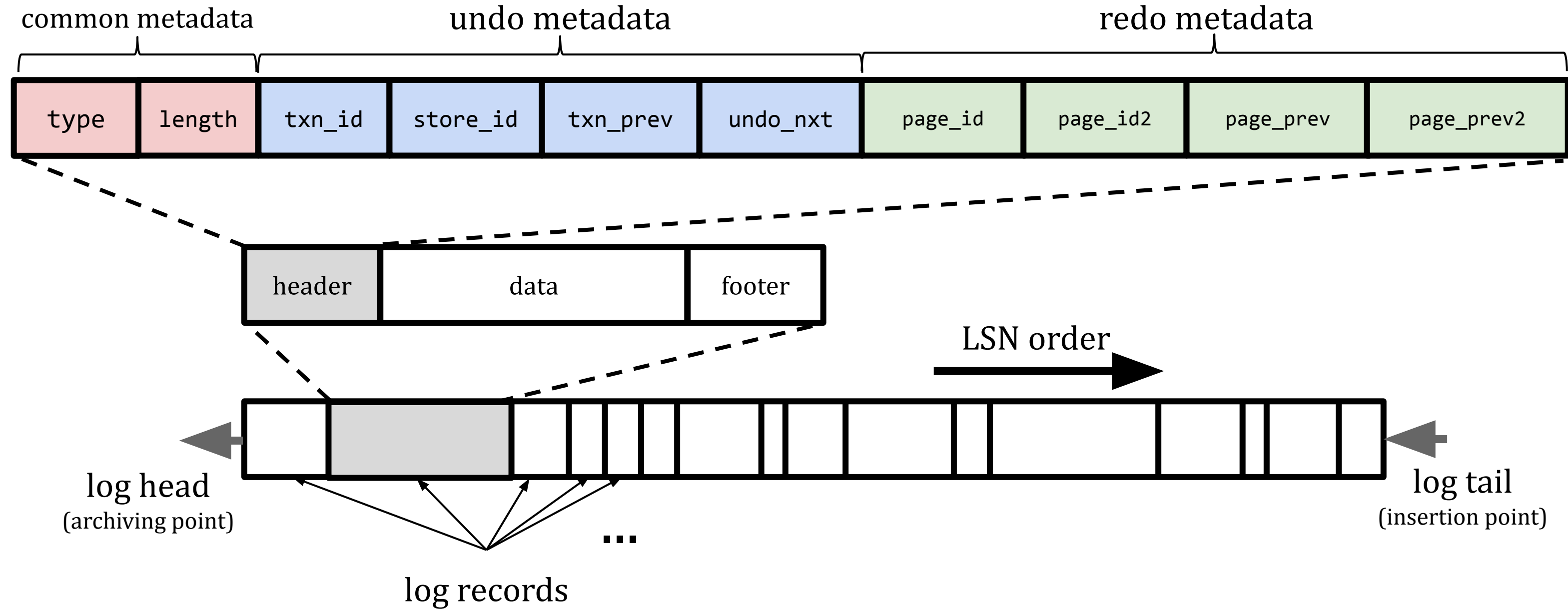
Agenda

- **Recap of the basics**
- Improving transaction throughput
- Faster recovery
- Availability while recovering
- Media recovery

Page-oriented storage



Write-ahead log



Normal operation

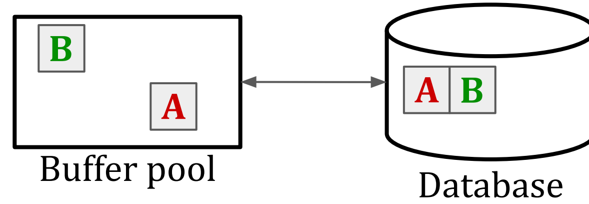
Volatile system state

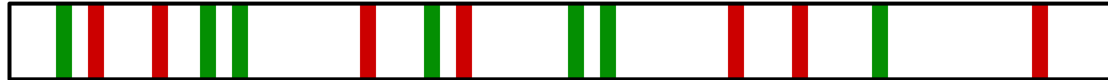
Dirty page table

PID	LSN
A	**x**
B	**y**

Active txn. table

TID	LSN
$\mathbf{T_1}$	**m**
$\mathbf{T_2}$	**n**



write-ahead log 

System failure

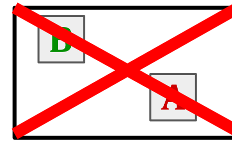
Volatile system state

Dirty page table

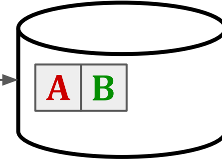
PID	LSN
A	x
B	y

Active txn table

TID	LSN
T_1	m
T_2	n

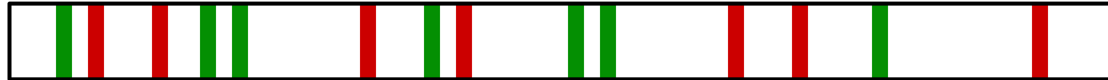


Buffer pool



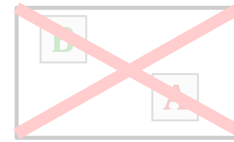
Database

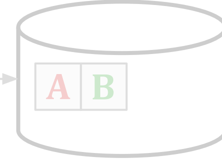
LOG



ARIES restart

Volatile system state





ARIES restart

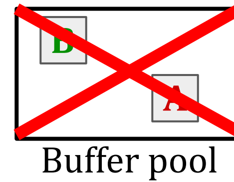
Volatile system state

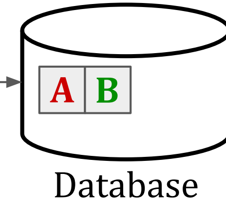
Dirty page table

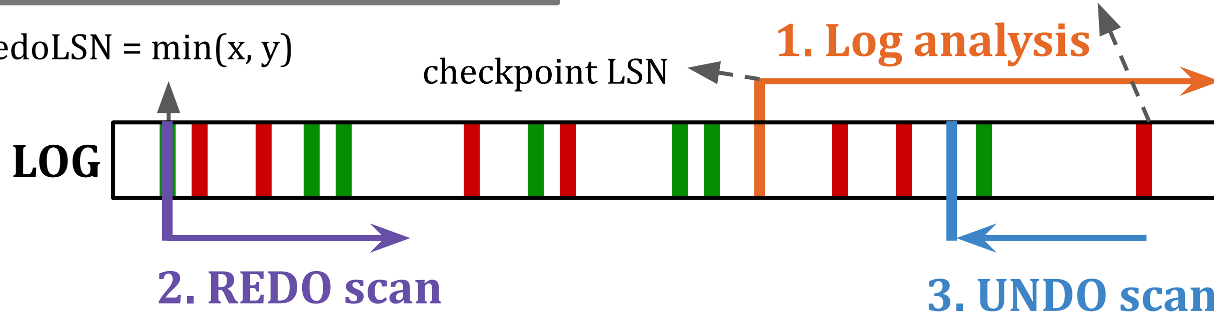
PID	LSN
A	**x**
B	**y**

Active txn. table

TID	LSN
$\mathbf{T_1}$	**m**
$\mathbf{T_2}$	**n**

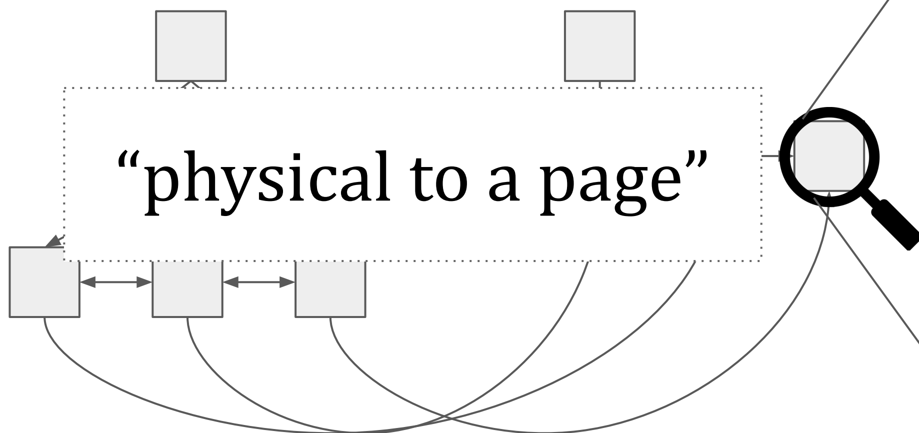




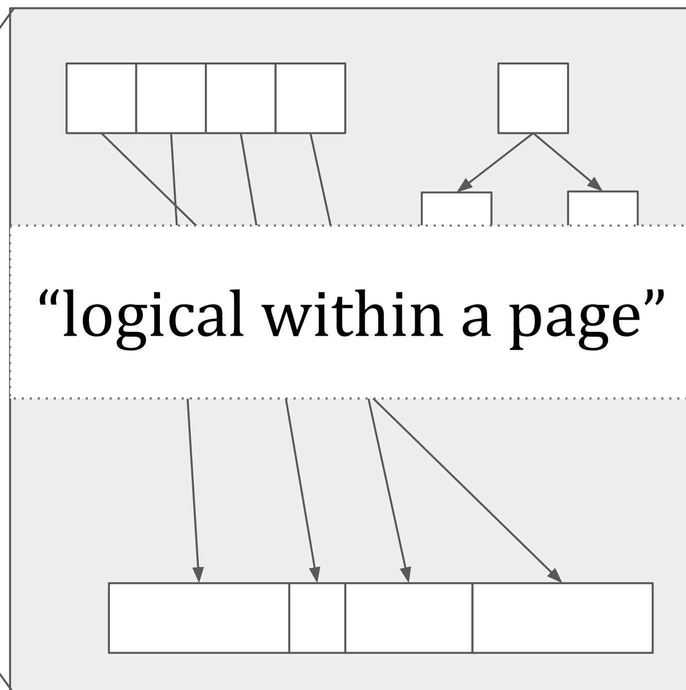


Physiological logging

index/table == linked data structure of nodes

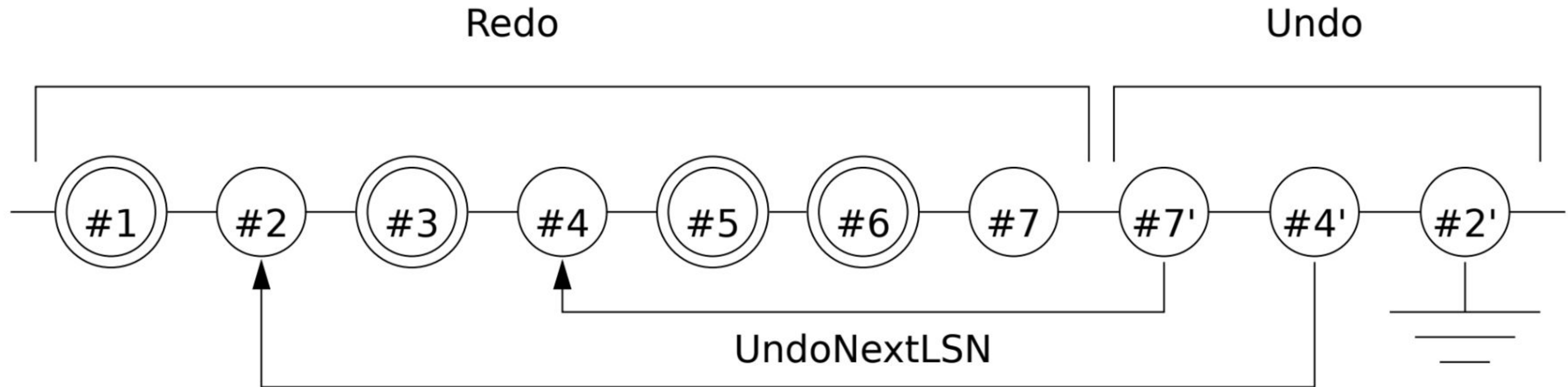


node == page with "mini data-structure"



"Being able to perform page-oriented redo allows the system to provide *recovery independence amongst objects*."

Logical UNDO with compensation



- CLRs (compensating log records) for undone changes
- #7' is a CLR for #7
- #4' is a CLR for #4

* Diagram from Thomas Neumann

Logical UNDO with compensation

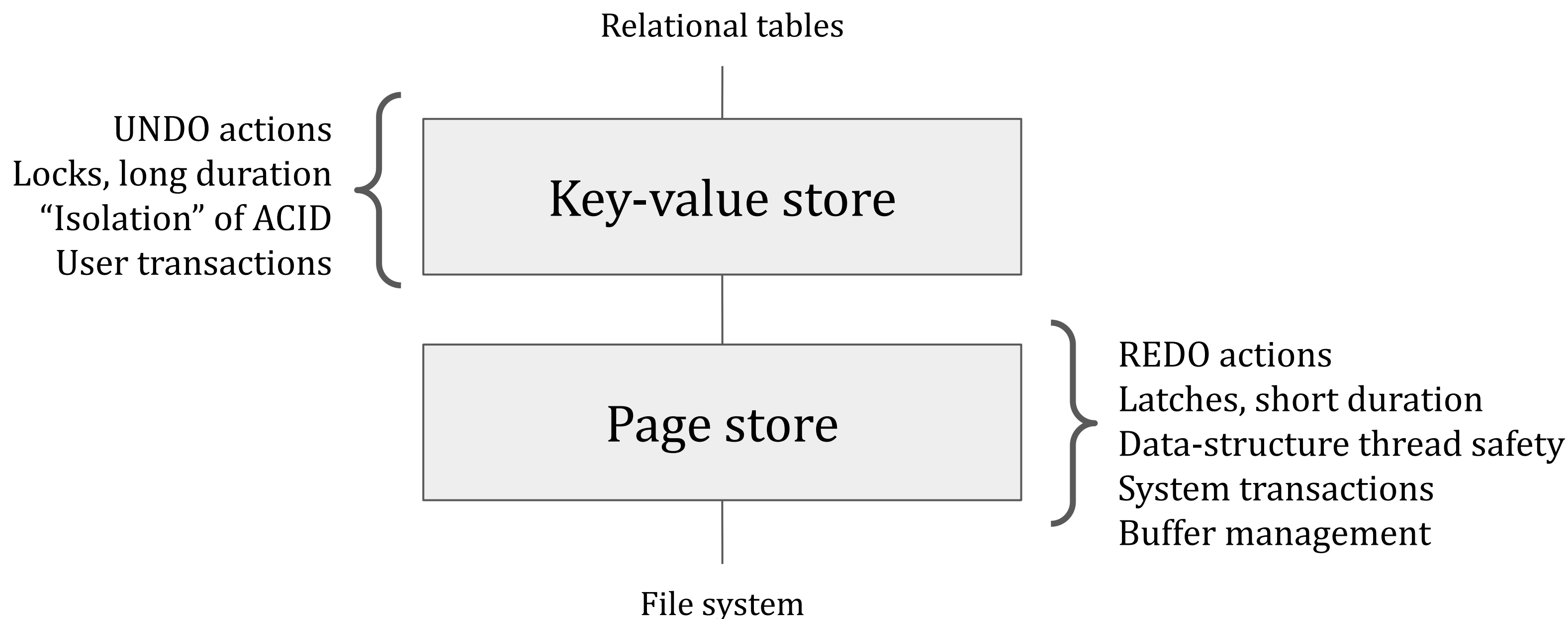
- Why logical?
  - **Record-level locking instead of page-level locking**
  - Data structure maintenance independent of transaction logic
    - Records might move to another page
  - Same logic for system restart (recovery operation) and transaction abort (normal operation)

- Why compensation log records?
  - Idempotent restart, i.e., make progress if system keeps failing
  - Convert UNDO actions into REDO actions
  - Sounds wasteful, but it’s key for a simple, high-performing architecture!

- **Repeating history** principle
  - REDO everything first, so the system state is exactly as it was before crash
  - Allows for a simpler architecture with better separation of concerns

If you remember one thing from today…

Separation of concerns in transaction processing!



ARIES Recap

- Three phases of recovery
- Key principle 1: Physiological logging
- Key principle 2: Logical UNDO with compensations
- Why logical logging is a bad idea

ARIES Recap

- Three phases of recovery
- Key principle 1: Physiological logging
- Key principle 2: Logical UNDO with compensations
- **Why logical logging is a bad idea**

Why logical logging is a bad idea (rant slide)

1. Recovery is **way** slower
    a. Same effort as normal operation
    b. Cannot run in parallel, otherwise might get a different serializable history
    c. All actions must be deterministic
    d. Checkpoints are very expensive, so not taken very often, so recovery even slower!
2. Normal operation is **not necessarily** faster
    a. Some form of in-memory logging (or MVCC) still required for transaction abort
    b. Overhead of writing those logs to disk is saved, but that's not on the critical path (more on this later)
3. Might compromise crucial features
    a. Indexing, space management, partial rollback, media recovery, …
4. It is bad economics
    a. SSDs are fast and cheap; use them!
    b. Check out Viktor Leis' **LeanStore** project

Agenda

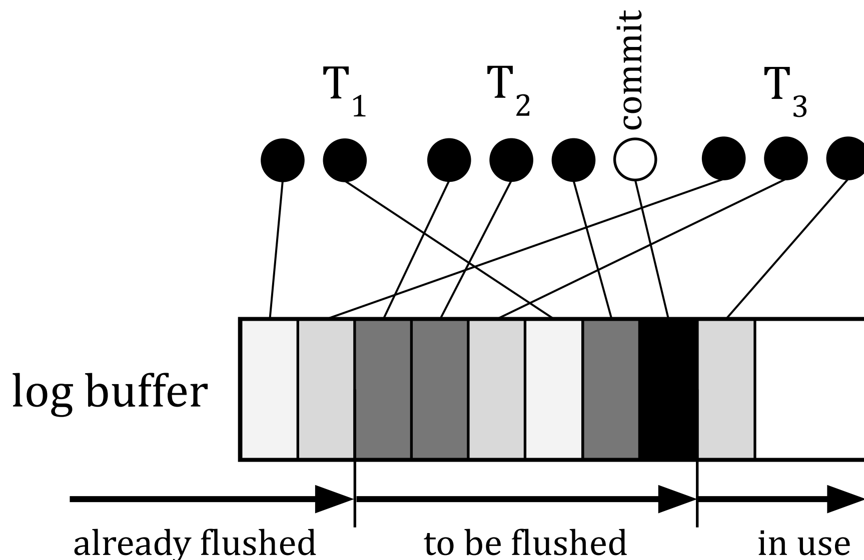
- Recap of the basics
- **Improving transaction throughput**
- Faster recovery
- Availability while recovering
- Media recovery

Key principles for faster normal operation

1. Remove logging from critical path
    a. Writing from main memory to SSD
        i. Group commit
        ii. Early lock release
    b. Writing from CPU to main memory
        i. Concurrent log buffers
        ii. Log partitioning
2. Log less data
    a. Log-record compression
    b. No-steal approaches (a.k.a., no-UNDO recovery)

Focus today

Log buffer



- Log buffer is an in-memory data structure; how to allow fast, concurrent operations on it?
- How to maintain correct transaction semantics in the presence of failures?
  - Durability of committed transactions
  - Handle partial writes
  - Avoid “holes” in the log

Sources of contention

R. Johnson et al.: Aether: A Scalable Approach to Logging, VLDB 2010

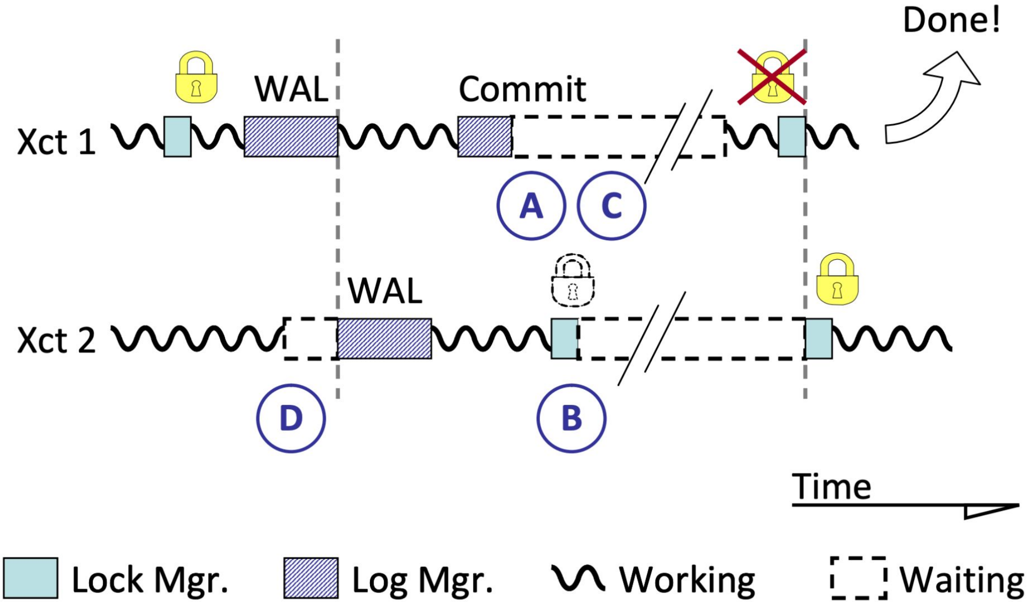
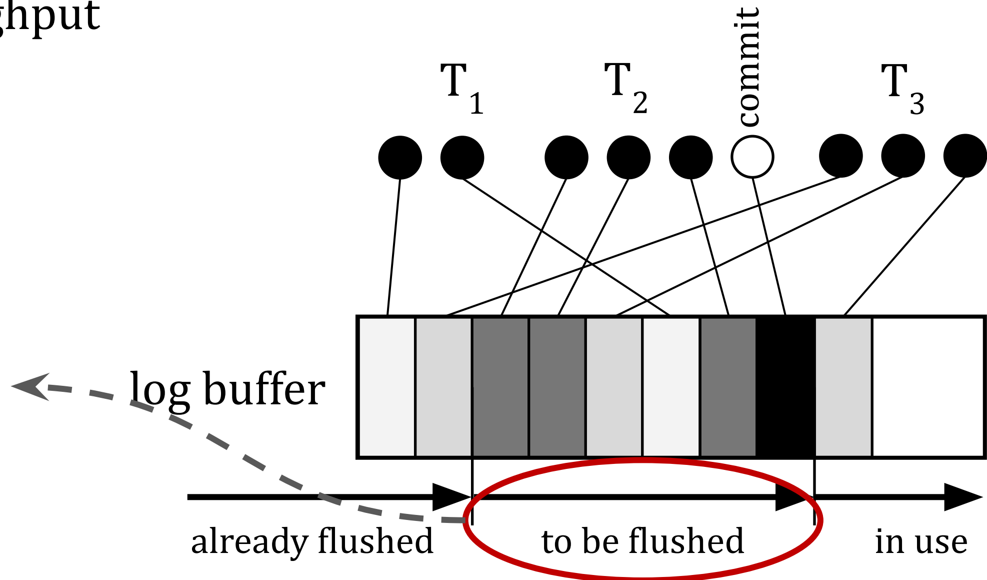


Figure 1. A timeline of two transactions illustrating four kinds of log-imposed delay: (A) I/O-related delays, (B) increased lock contention, (C) scheduler overload, and (D) log buffer contention.

Group Commit

- Key technique for scalability, already used in in-memory databases of the 80's
- Transactions don't commit as soon as they are ready, but rather accumulate in a buffer, so that multiple commits can happen with a single I/O operation
- Trades off latency for throughput



- Thread that was working on T2 can do some other work
- Dedicated log-flushing thread writes out log and notifies clients waiting for commit (asynchronously)

Early Lock Release

- What happens when a transaction commits:
    a. Append commit log record to the log buffer
    b. Wait for all logs until commit log record to be made durable (I/O latency)
    c. Release all locks (in two-phase locking)
- Problem: Other transactions will wait for step c, which involves an I/O operation
    a. Transaction throughput unavoidably decreases
- Solution: swap steps **b** and **c**!

Early Lock Release

- Safe, as long as **order constraints** are persisted
  - T1 writes a value x, appends a commit log record, and releases its lock
  - T2 now reads x and commits
  - Crash happens!
  - All log records of T1 must be present in the log before T2’s commit log record
- A centralized, sequential log easily avoids these problems
  - All log records before a commit log record must be persisted before acknowledging a commit
- Key distinction: pre-commit != commit acknowledgement

Aether

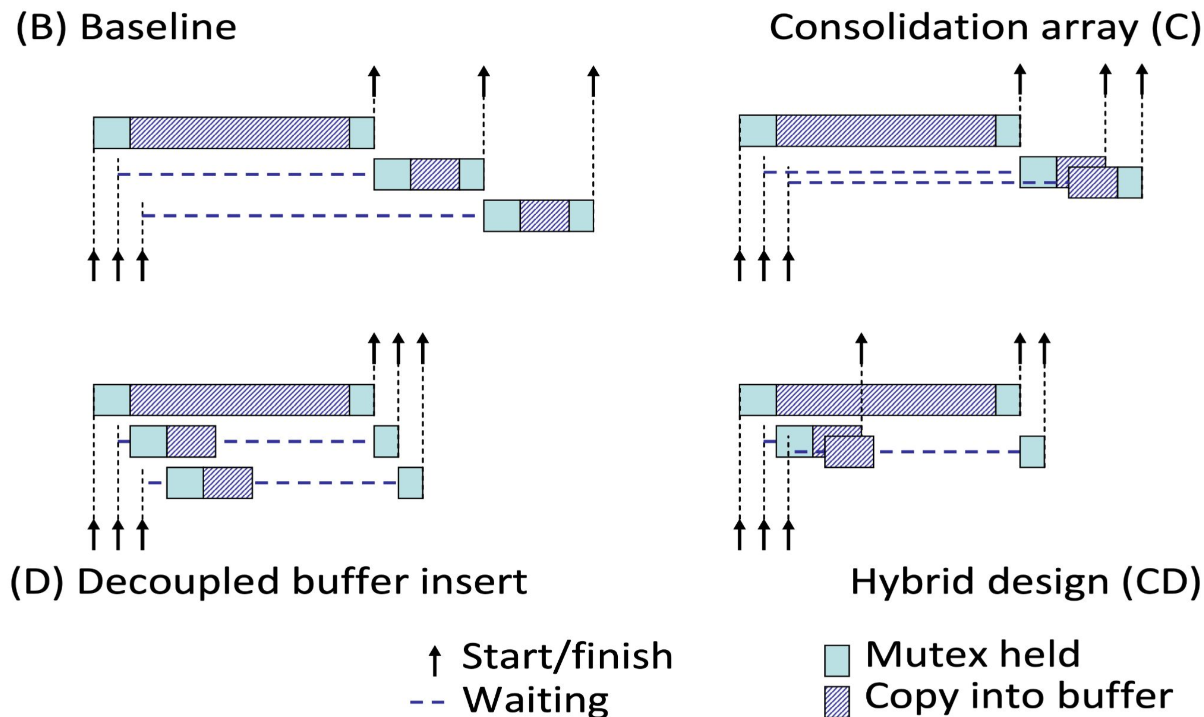
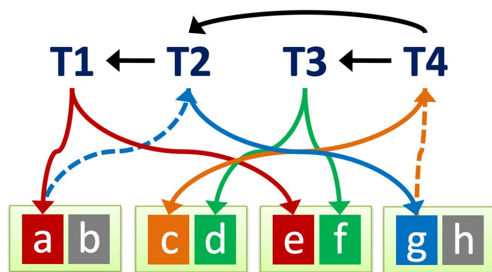
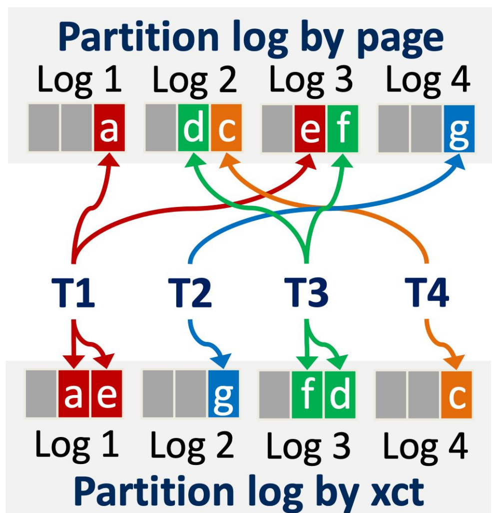


Figure 6. Illustrations of several log buffer designs. The baseline system can be optimized for shorter critical path (D), fewer threads attempting log inserts (C), or both (CD)

Log partitioning





- Partitioning by transaction performs better, due to locality and CPU affinity
- Challenge: keep track of dependencies and avoid holes in the log
  - GSN approach

Wang T., Johnson R.: Scalable Logging through Emerging Non-Volatile Memory, VLDB 2014

GSN = Global Sequence Number

- Assigns a GSN counter to each txn, each page, and each log
- Page GSN
    - Last modification done to this page, incremented with every modification together with the Txn GSN
- Txn GSN
    - Highest Page GSN seen by this txn so far
- Log GSN
    - Highest GSN inserted into this log so far

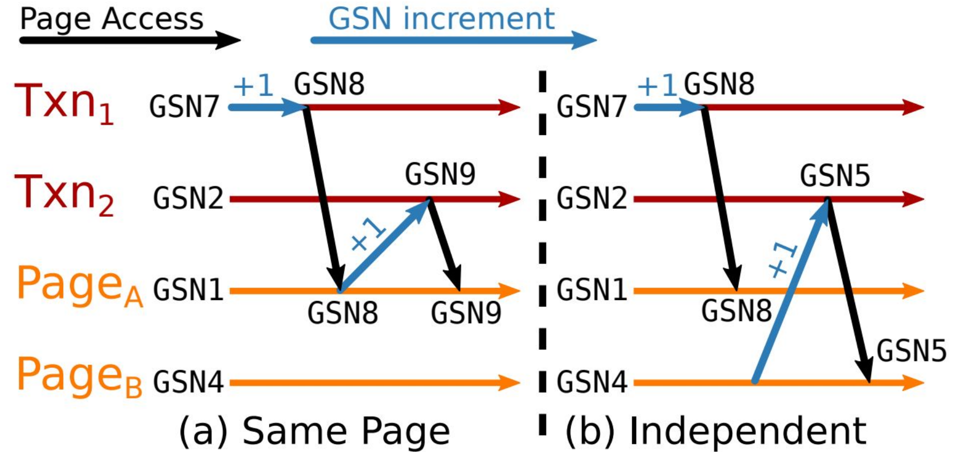


Figure 1: GSNs establish a partial order between log records sufficient for recovery. If two changes depend on each other, the second one will have a higher GSN (a). For independent changes, GSNs are unordered (b).

M. Haubenschild et al.: Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines, SIGMOD 2020

Group commit with GSN

- Wang & Johnson, VLDB 2014: Passive Group Commit
  - When txn commits, flush its own log and wait asynchronously on a queue until all other logs have been flushed up to the GSN of its commit log record.
- Haubenschild et al., SIGMOD 2020: Remote Flush Avoidance
  - Improves latency with a lightweight dependency tracking mechanism (better for NVM)
  - Txn only needs "remote flush" if it touched a page whose GSN was not durable at the time the txn started AND if any of those updates were logged on a different log

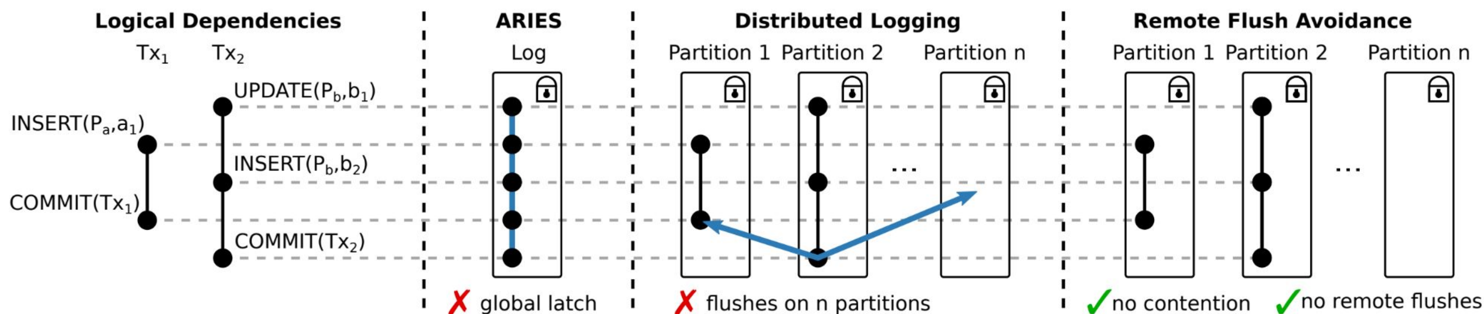
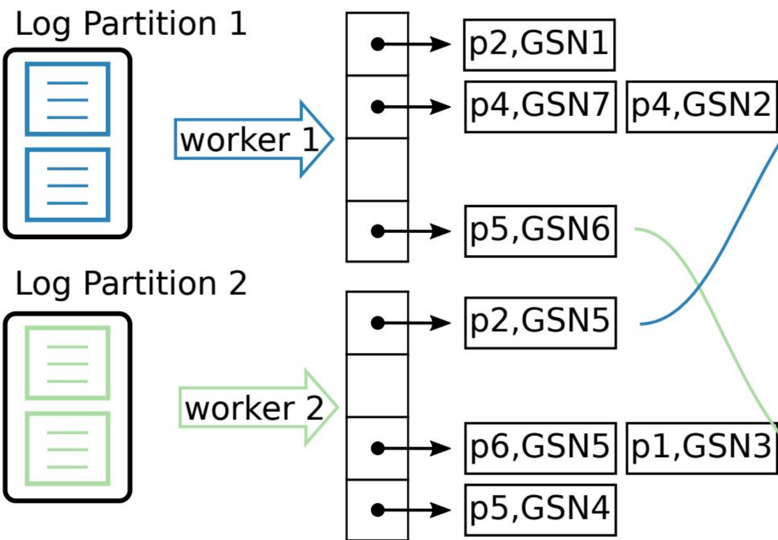
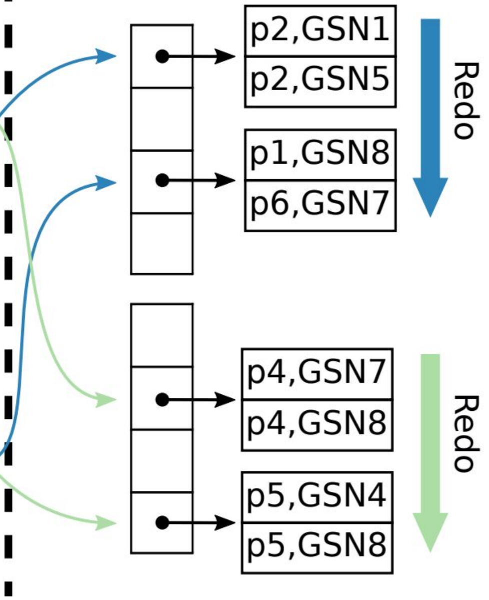


Figure 3: Two independent transactions and the synchronizing operations in different logging strategies.

Recovery with partitioned logs





M. Haubenschild et al.

Summary of techniques to improve throughput

- Group commit: reduce OS scheduling overhead; hide I/O latency
- Early lock release: remove log-induced contention on logical locks
- Consolidated buffer inserts (Aether): reduce contention on a single log buffer
- Log partitioning: remove contention with multiple log buffers

Agenda

- Recap of the basics
- Improving transaction throughput
- **Faster recovery**
- Availability while recovering
- Media recovery

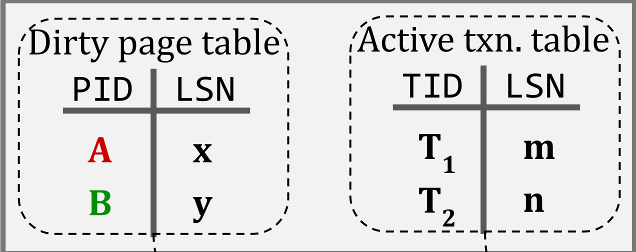
Key principles for faster recovery

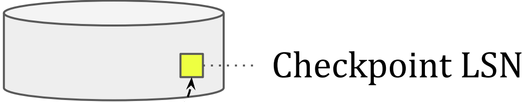
1. Fuzzy checkpoints
2. Fast page cleaning & provisioning
3. Parallel UNDO and REDO (more on this later, on availability section)

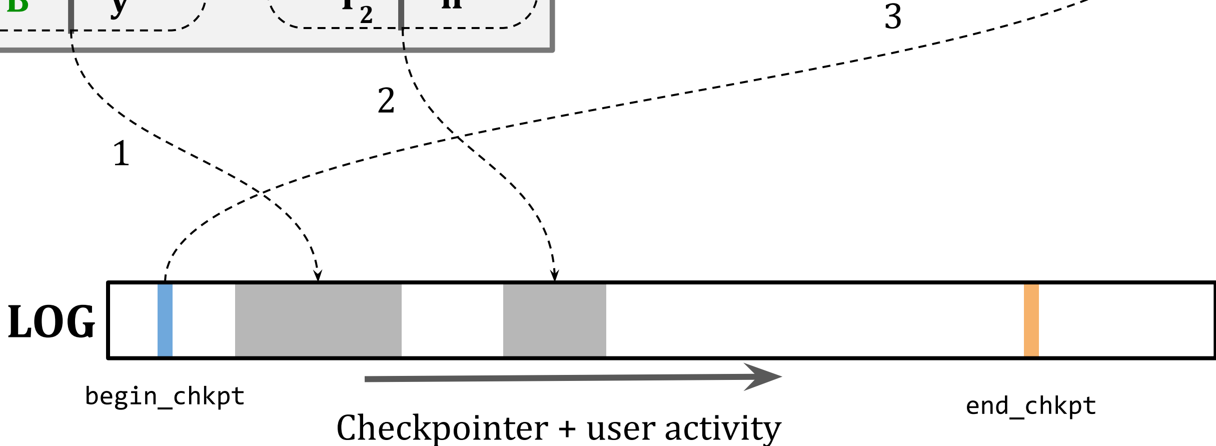
Fuzzy checkpoints

- "fuzzy" = does not capture global state at a frozen instant in time (i.e., inconsistent)
- Must happen continuously, in the background, without disrupting transactions
- System state != Database state
  - Checkpointer saves the state of volatile data structures: page table, txn. table, lock table, etc.
  - Page cleaner writes the contents of pages from buffer pool to database (more on this later)
- Effect on recovery phases
  - Checkpoints shorten log analysis
  - Page cleaner reduces REDO work
  - Nothing reduces UNDO work, because it depends on user activity

Fuzzy checkpoints





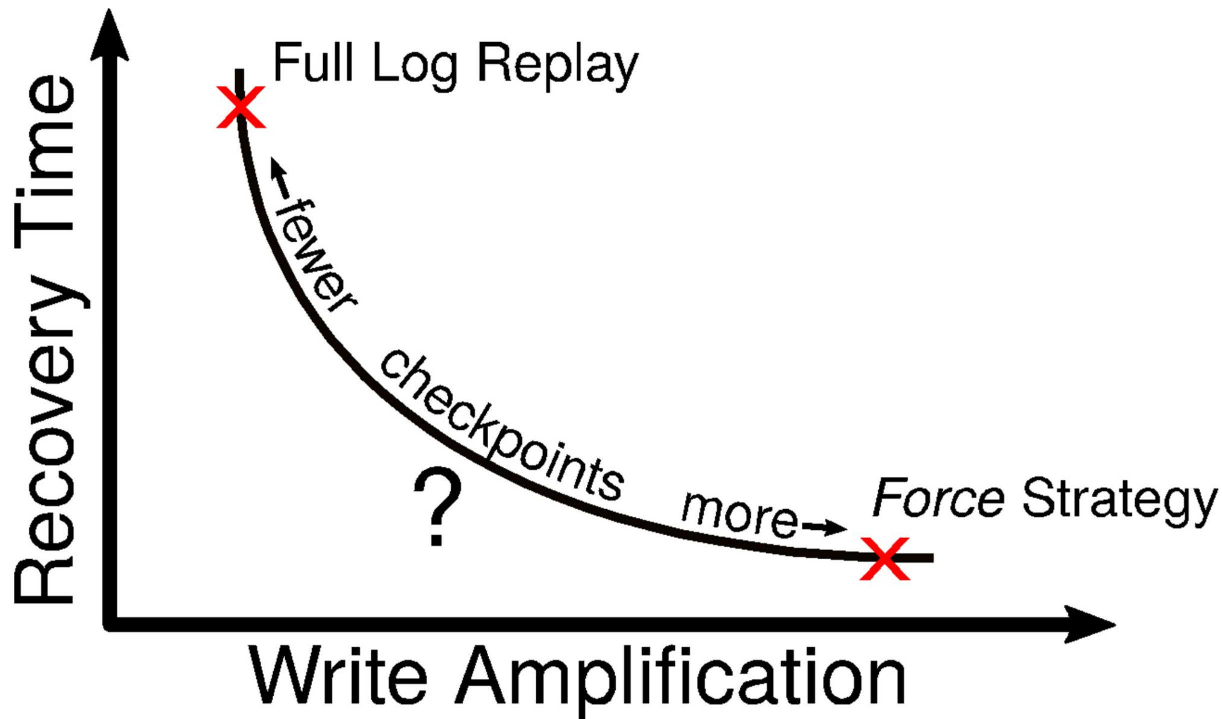


Fuzzy checkpoints

- Low disruption
  - No database contents are inspected, only auxiliary data structures
  - Data structures should support fuzzy, low-overhead scans (no global locks)
- How is it OK to have fuzzy state?
  - Tables will definitely change while checkpoint is happening
  - BUT: any **relevant** state change is also logged!
  - Thus, log analysis will read the tables from the checkpoint and update them as the log is scanned, reaching a consistent system state, as it was immediately before the crash
- What if system crashes while checkpoint is happening?
  - Checkpoint only considered complete if CheckpointLSN is updated, which happens atomically
  - Incomplete checkpoints are ignored
  - Alternative design: scan log backwards during log analysis phase and look for matching begin & end log records

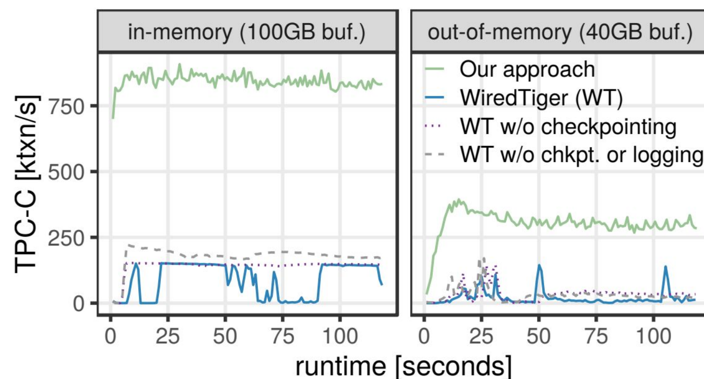
Page cleaning

Trade-off curve of page cleaning



Challenges of page cleaning

- Strike a balance between opposing goals:
  - Bound recovery time
  - Avoid wasting write bandwidth
- Make sure new pages can be allocated without delay (provisioning)
  - Essential for insertion-heavy workloads like TPC-C
- Execute as a continuous process without any I/O bursts or online disruptions



M. Haubenschild et al.

Page cleaning & provisioning

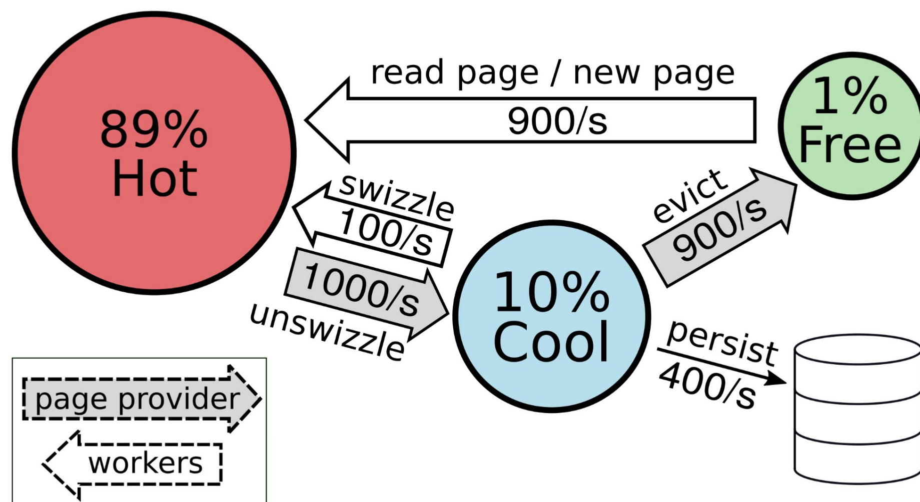
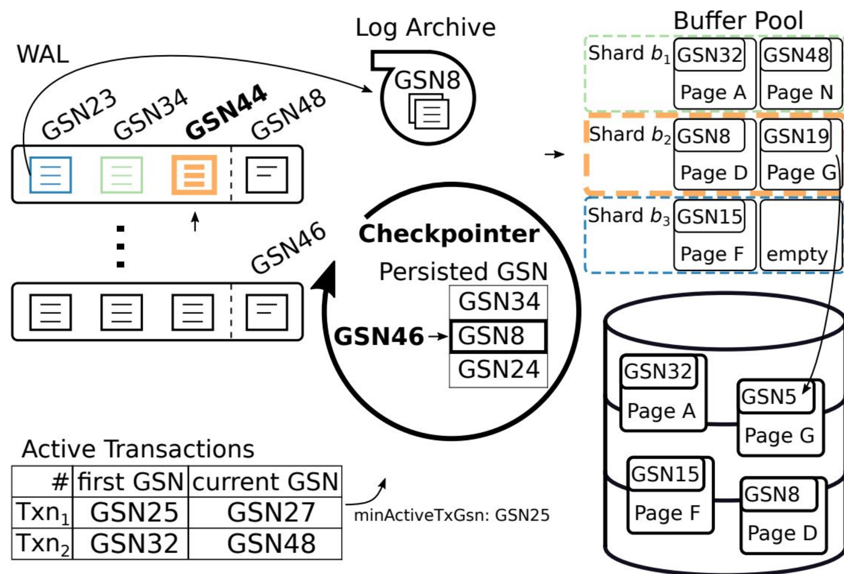


Figure 6: The division of LeanStore's buffer pool into hot (swizzled), cool (unswizzled) and free pages. Actions in the system cause pages to transition between the states. In steady state, worker threads request exactly as many free pages as the page provider supplies.

M. Haubenschild et al.

Continuous page cleaning

- Key idea: maintain a bound on the log size, and let log archiving process trigger cleaning of the buffer pool in small increments (shards)
- No bursts of activity or disruptions
- Redo recovery effort always bounded



#	first GSN	current GSN
Txn_1	GSN25	GSN27
Txn_2	GSN32	GSN48

Figure 5: Checkpointing example. The frequency of checkpoint increments is coupled to WAL volume.

M. Haubenschild et al.

Agenda

- Recap of the basics
- Improving transaction throughput
- Faster recovery
- **Availability while recovering**
- Media recovery

Key insight for higher availability

- Apply same recovery actions, just on a different schedule
  - Prioritize needs of application right after system failure
  - Reuse same concurrency protocols of normal operation (two-phase locking, buffer-pool latches) to guarantee correctness

ARIES optimization: concurrent UNDO

Volatile system state

Dirty page table

PID	LSN
A	**x**
B	**y**

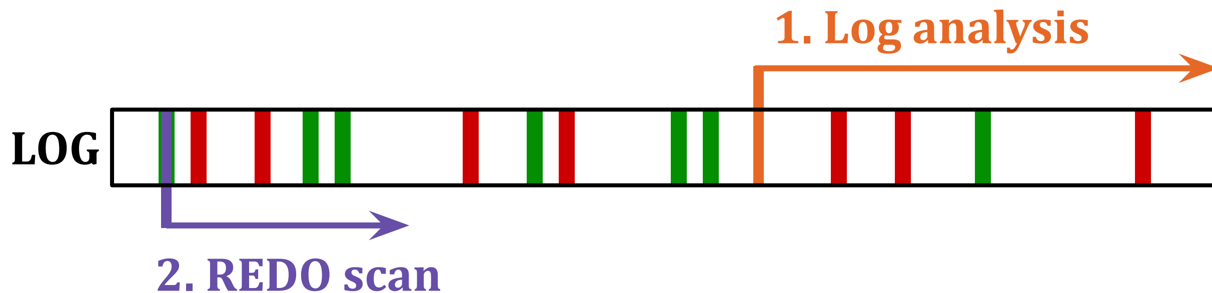
Active txn. table

TID	LSN
$\mathbf{T_1}$	**m**
$\mathbf{T_2}$	**n**

Lock table

Locks	TID
b,d	$\mathbf{T_1}$
f	$\mathbf{T_2}$

- Acquire and release locks during REDO
- Instead of UNDO log scan, abort loser transactions in parallel
- Follow *UndoNext* chain in the log or build stack of undo log records



works, but UNDO phase is usually negligible

Linked log records

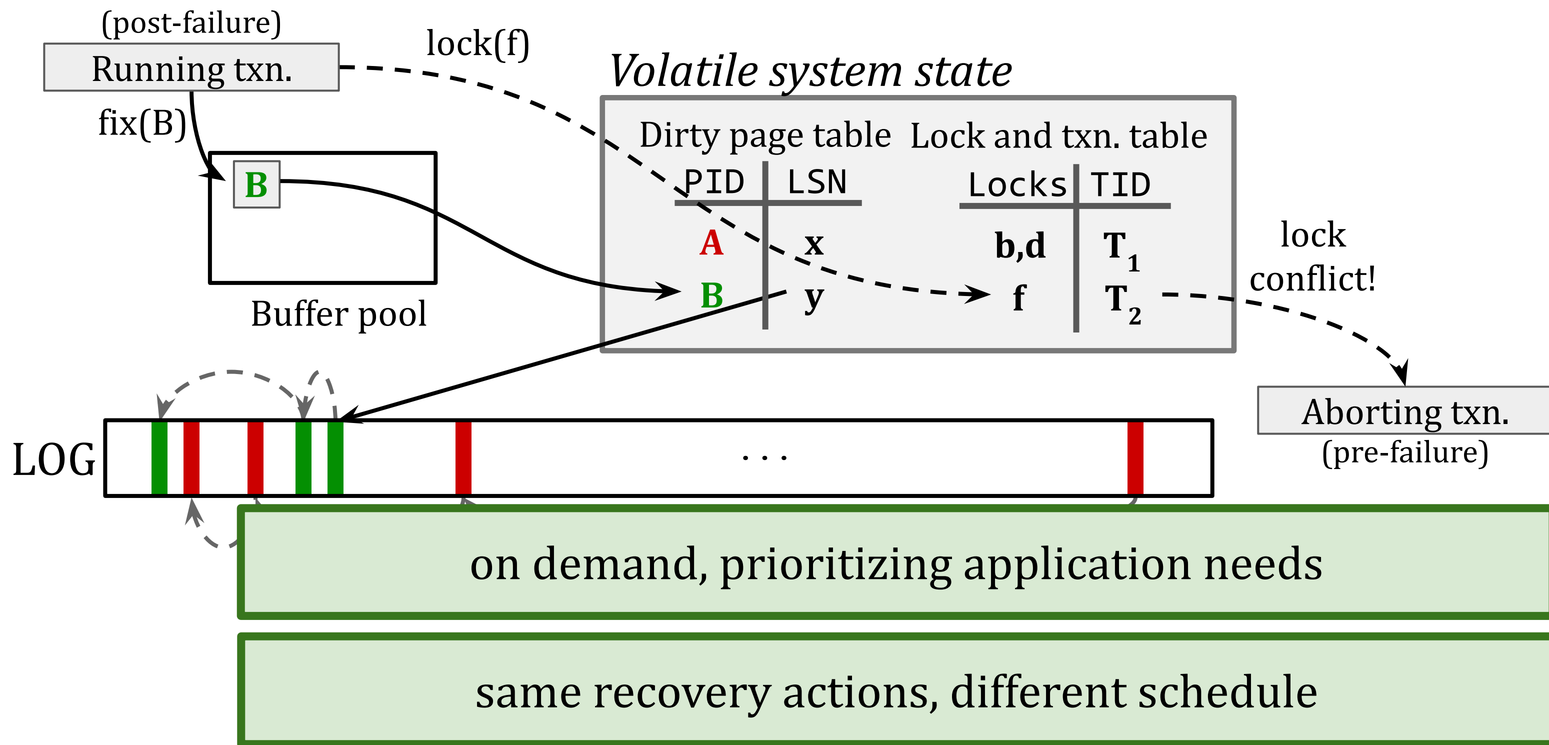
type	lsn	page id	page_prv	txn id	store id	txn prev	data
split	100	A, C	20, null	(sys)			“split page A into C”
ins	120	C	100	1	1	null	“insert record x into C”
upd	140	B	80	2	1	60	“update record y → y’ in B”
rebl	160	A, C	100, 120	(sys)			“rebalance records between A and C”
upd	180	C	160	2	1	140	“update record w → w’ in C”
ins	200	C	180	1	1	120	“insert record z into C”

A B C per-page log chains

per-transaction log chains 1 2

Instant restart

Graefe et al.: Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition. Synthesis Lectures on Data Management, 2016



on demand, prioritizing application needs

same recovery actions, different schedule

Agenda

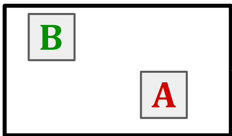
- Recap of the basics
- Improving transaction throughput
- Faster recovery
- Availability while recovering
- **Media recovery**

Media failure

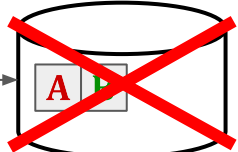
Volatile system state

Dirty page table	
PID	LSN
A	x
B	y

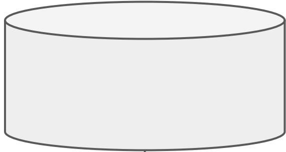
Active txn. table	
TID	LSN
T_1	m
T_2	n



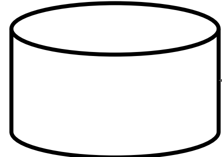
Buffer pool



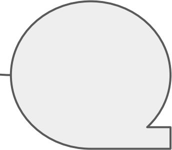
Database



Backup



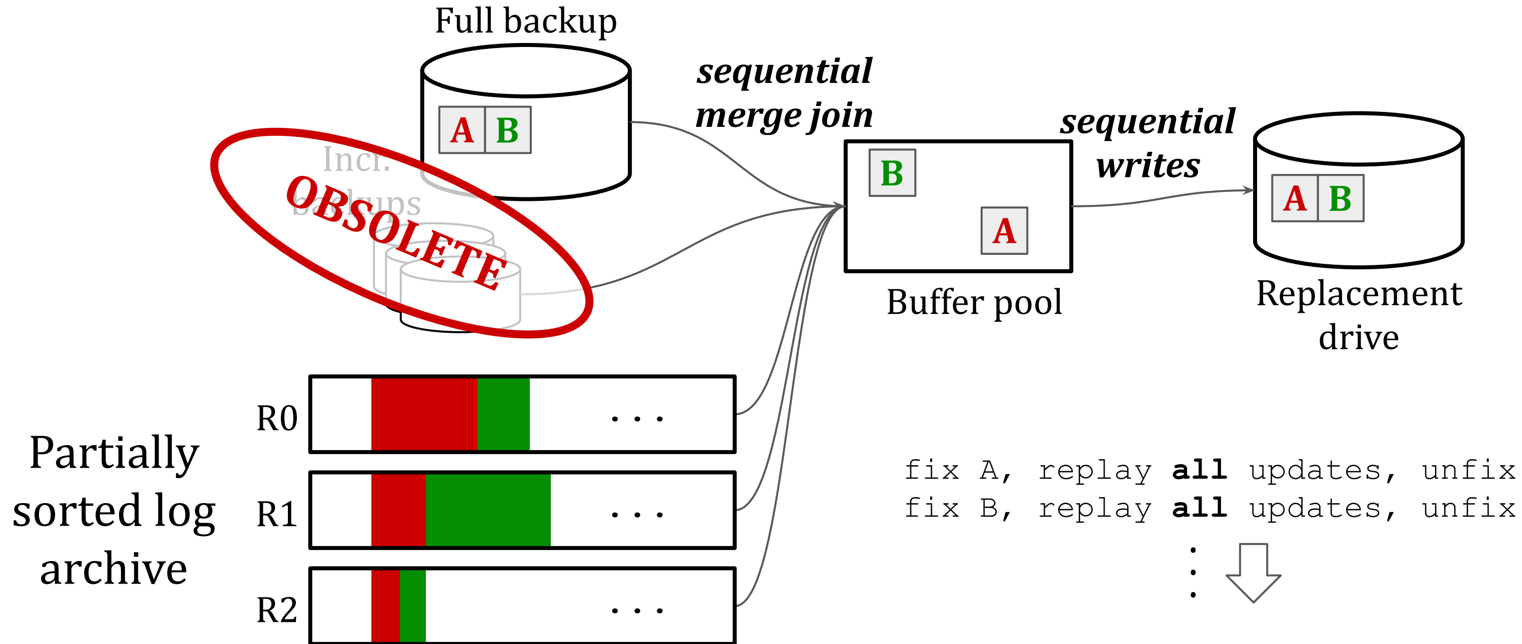
Replacement device



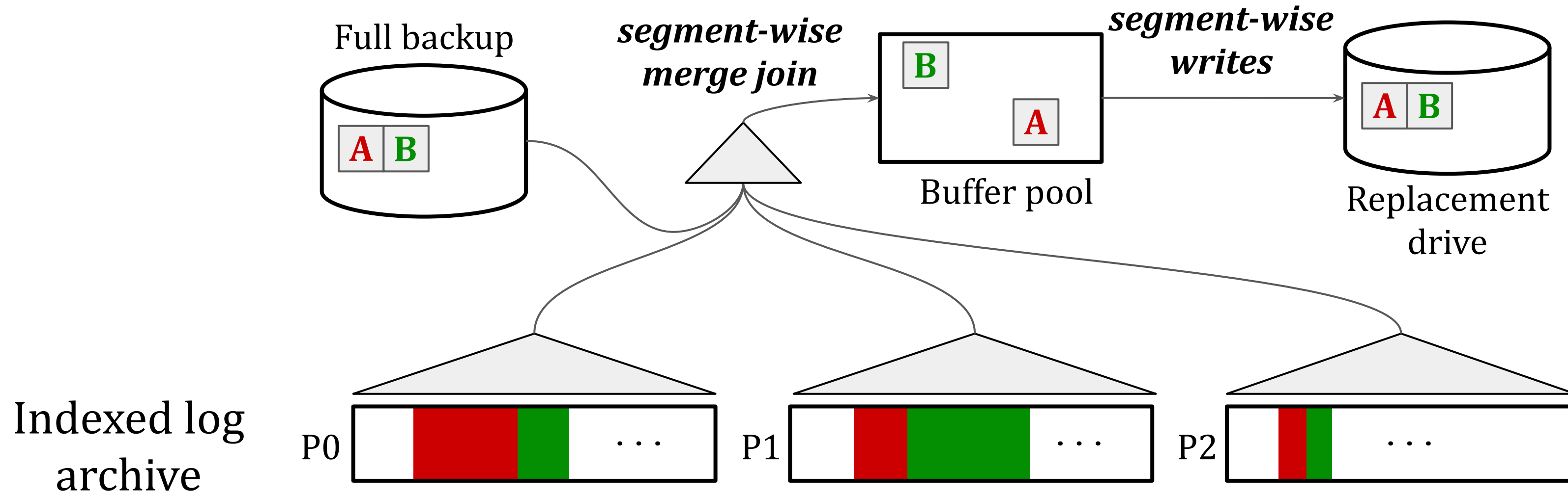
Log archive

LOG

Single-pass restore

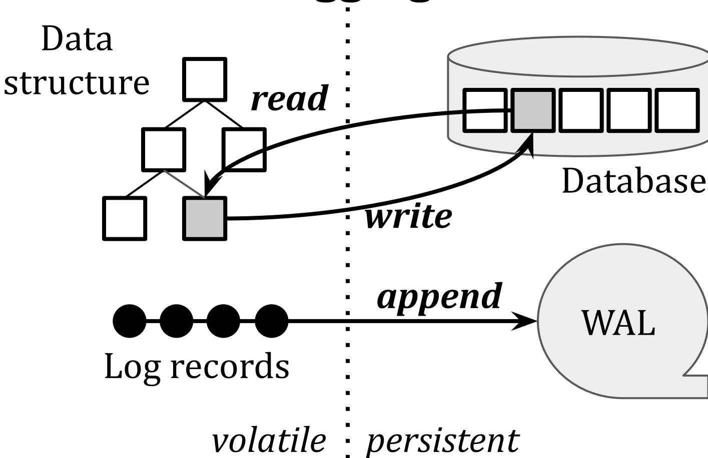


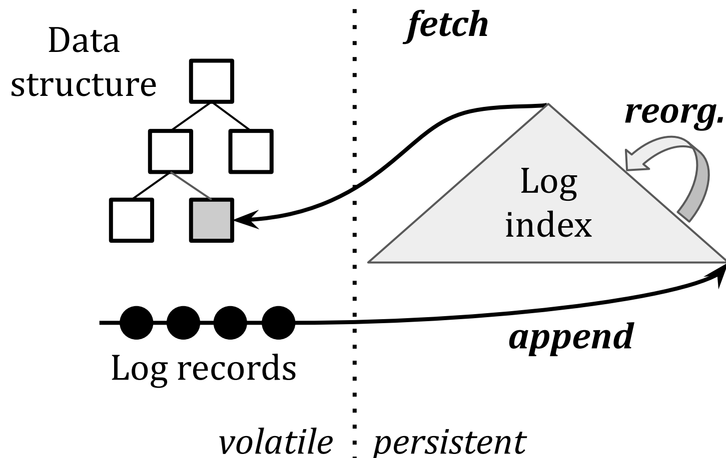
Instant restore



C. Sauer et al.: FineLine: Log-structured transactional storage and recovery, VLDB 2019

FineLine





Thank you!

caetano.sauer@salesforce.com