**Advanced Topics in Operating Systems, CS262a**

**Prof. Eric A. Brewer (with help from Rusty Sears)**

# Segment-oriented Recovery

ARIES works great but is 20+ years old and has some problems:

- viewed as very complex
- no available implementations with source code
- part of a monolithic DBMS: can use reuse transactions for other systems?
- LSN in the page breaks up large objects (Fig 1), prevents efficient I/O
- DBMS seems hard to scale for cloud computing (except by partitioning)

Original goals (Stasis):

- build an open-source transaction system with full ARIES-style steal/no-force transactions
- try to make the DBMS more "layered" in the OS style
- try to support a wider array of transactional systems: version control, file systems, bioinformatics or science databases, graph problems, search engines, persistent objects, ...
- competitive performance

These goals were mostly met, although the open-source version needs more users and we have only tried some of the unusual applications.

Original version was basically straight ARIES; development led to many insights and the new version based on segments.

**Segment-oriented recovery (SOR)**

A *segment* is just a range of bytes

- may span multiple pages
- may have many per page
- analogous to segments and pages in computer architecture (but arch uses segments for protection boundaries, we use them for recovery boundaries; in both, pages are fixed size and the unit of movement to disk)

Key idea: change two of the core ARIES design points:

- Recover segments not pages
- No LSNs on pages
  - ARIES: LSN per page enable *exactly once* redo semantics
  - SOR: estimate the LSN conservatively (older) => *at least once* semantics [but must now limit the actions we can take in redo]

**The Big Four Positives of SOR:**

**1) Enable DMA and/or zero-copy I/O**

- No LSNs per page mean that large objects are contiguous on disk
- No "segmentation and reassembly" to move objects to/from the disk
- No need to involve the CPU in object I/O (very expensive); use DMA instead

**2) Fix persistent objects**

- Problem: consider page with multiple objects, each of which has an in memory representation (e.g. a C++ or Java object)
  - Suppose we update object A and generate a log entry with LSN=87
  - Next we update object B (on the same page), generate its log entry with LSN=94, and write it back to the disk page, updating the page LSN
  - This pattern breaks recovery: the new page LSN (94) implies that the page reflects redo log entry 87, but it does not.
  - ARIES "solution": disk pages (in memory) must be updated on every log write; this is "write through caching" -- all updates are written through to the buffer manager page
- SOR solution:
  - there is no page LSN and thus no error
  - Buffer manager is written on cache eviction -- "write back caching". This implies much less copying/ marshalling for hot objects.
  - This is like "no force" between the app cache and the buffer manager (!)

**3) Enable high/low priority transactions (log reordering on the fly)**

- With pages, we assign the LSN atomically with the page write [draw fig 2]
  - not possible to reorder log entries at all
  - in theory, two independent transactions could have their log entries reordered based on priority or because they are going to different log disks (or log servers)
- SOR: do not need to know LSN at the time of the page update (just need to make sure it is assigned before you commit, so that it is ordered before later transactions; this is trivial to ensure -- just use normal locking and follow WAL protocol)
  - High priority updates: move log entry to the front of the log or to high priority "fast" log disk
  - Low priority updates go to end of log as usual

**4) Decouple the components; enable cloud computing versions**

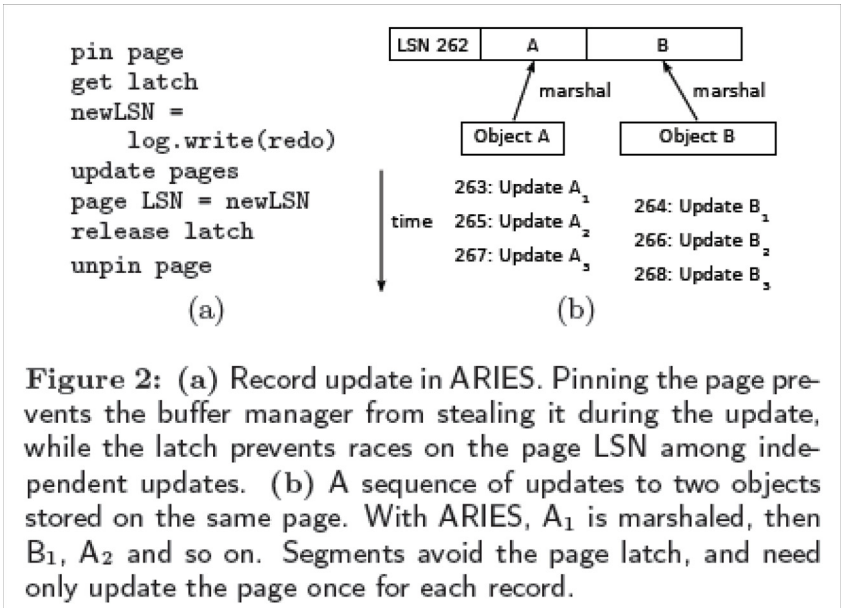


**Figure 2:** (a) Record update in ARIES. Pinning the page prevents the buffer manager from stealing it during the update, while the latch prevents races on the page LSN among independent updates. (b) A sequence of updates to two objects stored on the same page. With ARIES, $A_1$ is marshaled, then $B_1$, $A_2$ and so on. Segments avoid the page latch, and need only update the page once for each record.

- background: "pin" pages to keep them from being stolen (short term), "latch" pages to avoid race conditions within a page
- subtle problem with pages: must latch the page across the call to log manager (in order to get an LSN atomically)
- SOR has no page LSN and in fact no shared state at all for pages => no latch needed
- SOR decouple three different things:
  - App <-> Buffer manager: this is the write-back caching described above: only need to interact on eviction, not on each update
  - Buffer manager <-> log manager: no holding a latch across the log manager call; log manager call can now by asynchronous and batched together
  - Segments can be moved via zero-copy I/O directly, with no meta data (e.g. page LSN) and no CPU involvement. Simplifies archiving and reading large objects (e.g. photos).
- Hope: someone (ideally in CS262) will build a distributed transaction service using SOR
  - Apps, Buffer Manager, Log Manager, Stable storage could all be different clusters
  - Performance: (fig 11): 1-3 orders of magnitude difference for distributed transactions

**Physiological Redo (review):**

- Redos are applied exactly once (using the page LSN)
- Combination of physical and logical logging
  - physical: write pre- or post-images (or both)
  - logical: write the logical operation ("insert")
- Physiological:

- redos are physical
- normal undos are like redos and set a new LSN (does not revert to the old LSN -- wouldn't work given multiple objects per page!)
- to enable more concurrency, do not undo structural changes of a B-Tree (or other index); instead of a physical undo, issue a new logical undo that is the inverse operation. Enables concurrency because we can hold short locks for the structural change rather than long locks (until the end of the transaction)

o Slotted pages: add an array of offsets to each page (slots), then store records with a slot number and use the array to look up the current offset for that record. This allows changing the page layout without any log entries.

**SOR Redo:**

- o Redos may applied more than once; we go back farther in time than strictly necessary
- o Redos must be physical "blind writes" -- write content that do not depend on the previous contents
- o Undos can still be logical for concurrency
- o Slotted page layout changes require redo logging

**Core SOR redo phase:**

- o periodically write estimated LSNs to log (after you write back a page)
- o start from disk version of the segment (or from snapshot or whole segment write)
- o replay all redos since estimated LSN (worst case the beginning of the truncated log), even though some might have been applied already
- o for all bytes of the segment: either it was correct on disk and not changed or it was written during recovery in order by time (and thus correctly reflects the last log entry to touch that byte)

**Hybrid Recovery**: Can mix SOR and traditional ARIES

- o Some pages have LSNs, some don't
- o Can't easily look at a page and tell! (all the bytes are used for segments)
- o Log page type *changes* and zero out the page
  - recovery may actually corrupt a page temporarily until it gets the type correct, at which point it rolls forward correctly from the all zero page.
- o Example of when to use:
  - B-Trees: internal nodes on pages, leaves are segments
  - Tables of strings: short strings good for pages especially if they change size; long strings are good for segments due to contiguous layout

**Extra stuff:**

Why are there LSNs on pages?

- o so that we can atomically write the timestamp (LSN) with the page
- o problem: page writes aren't actually atomic anymore
- o solution: make them atomic so that ARIES still works

- write some bits into all sectors of a page (8K page = 16 512B sectors); compare those bits with a value store somewhere else. No match => recover the page (but may match and be wrong). Assumes sectors are written atomically, which is reasonable.
- write a checksum for each page (including the LSN) and check it when reading back the page

o Both solutions impact I/O performance some

o SOR approach:

- checksum each segment (or group of segments if they are small)
- on checksum failure, can replay last redo, which can fix a torn page (and then confirm using the checksum); if that doesn't work go back to old version and roll forward
- blind writes can fix corrupted pages since they do not depend on its contents