DB Goals

- Concurrency Control:
  - Individual users see consistent states
    - Even though ops for many users may be interleaved
- Recovery:
  - Database is fault-tolerant
    - State not corrupted from software, system, media failure
- Why?
  - Write apps w/out explicit concern for either
    - All about programmer productivity, safety, etc.

Transactions

- Multiple users manipulating data safely
- ACID properties of a transaction
  - Atomicity: transform is all or nothing
  - Consistency: make only correct changes
    - Often expressed as declarative integrity constraints
      - “Salary of grad student cannot exceed min. wage/2”
  - Isolation: partial changes hidden from others
    - Pretend that yours is the only Tx running
  - Durability: committed changes survive subsequent failures
- AID provided by DBMS, C by programmer
  - DB is consistent iff contents result only from successful transactions;
  - Rest of C requires run-time triggers, etc. - ignore for now.

Std. example - Durability

- **Transfer()**
  - A_bal = Read(A)
  - A_bal -= 50
  - Write(A, A_bal)
  - B_bal = Read(B)
  - B_bal += 50
  - Write(B, B_bal)

  Those are underlying ops actually performed. In SQL, could be expressed as:
  - update accounts set balance=balance-50 where user='A'
  - update accounts set balance=balance+50 where user='B'
  - But same challenges apply...

If Crash: $50 disappears!
Isolation

- ReportSum()
  - $A_{\text{bal}} = \text{Read}(A)$
  - $B_{\text{bal}} = \text{Read}(B)$
  - printf("Riches:  %d\n", $A_{\text{bal}} + B_{\text{bal}}$);  
    prints $250$

- But consider interleaving:
  - Transfer()
    - $A_{\text{bal}} = \text{Read}(A)$
    - $A_{\text{bal}} -= 50$
    - Write($A$, $A_{\text{bal}}$)
    - $B_{\text{bal}} = \text{Read}(B)$
    - $B_{\text{bal}} += 50$
    - Write($B$, $B_{\text{bal}}$)
  - ReportSum()
    - $A_{\text{bal}} = \text{Read}(A)$
    - $B_{\text{bal}} = \text{Read}(B)$
    - printf("Riches:  %d\n", $A_{\text{bal}} + B_{\text{bal}}$);
    prints $300$

Why diff from filesystems?

- Both isolation and fault tolerance
  - Most filesystems don’t provide isolation
    - App must explicitly lock/etc.
  - Many don’t provide guaranteed recovery
    - Some - ext2 - don’t even guarantee that the FS itself will be usable after a crash. (!)

- Powerful combination
  - But also very complex and w/high overhead
  - Reasonable - FS is more general, etc.
  - And DBs let you pick...

Context: SQL, etc

- Single statements that affect multiple data objects
  - UPDATE grades set grade=grade+1;
  - Can be quite hairy -- conditionals, format conversion, etc.

- BEGIN TRANSACTION
  - insert into grades VALUES ("dave", "F")

- COMMIT TRANSACTION
  - Likely used by procedural language driving the transaction
    - But same machinery needed to make individual statements atomic!

- For perf: Can often select isolation level / read-only tx
  - Can also just LOCK TABLE (to avoid row-level locks and transaction overhead)

Failures

- Transaction failure
  - Code aborts, based on input/database inconsistency
  - [programmer escapes complexity]
  - Mechanical aborts caused by concurrency control solutions to isolation
  - Frequent events, “instant” recovery needed

- System failure (fail-stop)
  - DBMS bug, OS fault, HW failure: wipe out volatile memory but durable memory (disk) survives
  - Infrequent events, “minutes” to recover

- Media failure (fail-stop)
  - IO code bugs, disk HW failures: loss of disk info
  - Rare events, “hours” to recover from checkpoints & audit logs

- Note not talking about corruption (}
Recoverable System Model

- UNDO: rollback aborted transaction
  - Transaction (transaction failure) or global (system failure)
  - Employs short term log
- REDO: repeat complete transaction on old DB data
  - Partial (system failure) or global (media failure)
  - Employs long term history log (tape)

Tools for protecting internal consistency

- static mappings
  - if they don’t change, they don’t cause problems
  - most people don’t think of this one most of the time…
- “atomicity” of writes
  - ala the tri-state post-write guarantee of per-sector ECC
  - Atomic unit often called a “page” by DB folk
- update ordering
  - simply ensuring that one update propagates before another
- real atomicity
  - ensuring that a set of updates all occur or none do

“Atomicity” of writes as a tool

- Unwritten guarantee provided by per-sector ECC
  - because the ECC check will fail if only partially written
- Same trick can be used by FS or applications
- Good for grouping inter-related updates

Update ordering as a tool

- Just what it sounds like…
- We’ve seen this a lot so far
  - Softupdates
  - B-link tree updates
- Good for single-direction dependencies
  - just do one before the other
- Problem: doesn’t work for bidirectional dependencies
  - which, unfortunately, is most of them
- Solution: some can be converted to single-direction
  - because some directions are more important than others ;)
  - clean-up must be done after system failures
Views of the Database

- Current DB: on disk + memory buffers
  - Some transactions are in flight with data and metadata changes in memory buffers that might not occur
- Materialized DB: crash restart before applying log processing
  - Some logically completed changes may not be visible on disk because some memory buffers were lost
  - Recovery: Go from materialized DB -> Current DB
- Physical DB: on disk
  - All disk blocks including out-of-date blocks, incomplete data structures and free space

Sequencing Views

- Changing non-volatile memory
  - “Modifications” of current DB may cause “writes” to physical DB that are not part of materialized DB until pointer structs are updated, i.e. “propagated” (like LFS updates, which learned from DB theory)
  - Some DBs overwrite prior copies so write=propagate, but this makes changes in materialized DB non-atomic (harder to recover)
  - Implementing atomic disk changes via non-overwrite propagation is based on writing a collection of new versions of buffer pages into free space then writing one block:
    - a root data structure pointer, a current maximum timestamp, a pointer to the new page table, etc

Stealing & Forcing

- Recall: Buffer manager decides to write memory pages out to disk
- If uncommitted Tx modifications can overwrite most recent committed item on non-vol storage: STEAL (otherwise NO-STEAL)
  - STEAL is ugly - must UNDO
  - NO-STEAL could require too much mem or swapping
- If buffer ensures all updates by Tx are reflected on non-volatile storage b4 commit, FORCE (otherwise NO-FORCE)

Temporary Log Files

- Redundant info for coping with failure: “write ahead logging”
  - On-disk temporary (write-ahead) log file contains all that is needed to transform materialized DB to current DB
  - Memory pressure can push uncommitted dirty data to database; in “overwrite” DBs this requires UNDO log records (STEAL) written before propagation; in non-overwrite DBs such writes are “forgotten” when memory is lost
  - Commit logically forces propagation (FORCE), but efficiency concerns cause DBs to avoid synchronous IO, instead writing REDO log records before transaction commit
  - Point: WAL allows STEAL/NO-FORCE buffer management (asynchrony!)
- Log types
  - Physical vs Logical: capture data values or operations giving values
  - State or transition: capture full values or differences from last values
  - Page or record: capture full page values or only the records changing
Physiological Logging

• In practice, many systems:
  – Log records refer to single page
  – May reflect logical operations on page
• e.g.,
  – Insert would specify new value of tuple
  – Would not specify free-space manip. or reorganization on page as a result of insert
  – REDO logic would have to do that
• Tuple insert that touched multiple pages would require 1 log record for each page updated. Avoids consistency problem but reducing somewhat the log size from physical logging.

Checkpoint

• Checkpoints limit REDO processing
  – REDO goes to beginning of log, but that can be really slow
  – FORCE propagation = no REDO (UNDOs on pre-written pages)
  – Transaction consistent checkpoint: Quiesce all transactions, propagate all dirty data, write log entry (long unavailability)
    • Allows partial REDO to start here and global UNDO recovery processing to stop here
  – Action consistent checkpoint: Quiesce transaction-caused actions (calls into DBMS), propagate all dirty data, write log entry
    • Allows partial REDO recovery processing to start here, but UNDO may go back further to find BEGIN for oldest incomplete transaction
  – Fuzzy checkpoints are those that “propagate” committed writes only to log (pointer to log address of REDO record), to reduce REDO processing on restart
    • Notice unpropagated in successive checkpoints & propagate

Crash Recovery

Figure 8. A crash recovery scenario.

Comparisons

| Table 4. Evaluation of Logging and Recovery Techniques Based on the Introduced Taxonomy |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| propagation strategy           | ATOMIC          | ATOMIC          | ATOMIC          | ATOMIC          | ATOMIC          |
| further replacement            | STEAL           | -STEAL          | STEAL           | -STEAL          | STEAL           |
| EOT processing                 | FORCE           | +FORCE          | FORCE           | +FORCE          | FORCE           |
| checkpoint type                 | TOC             | TCC             | ACC             | Fuzzy           | TOC             |
| materialized EOT index after system failure | DC              | DC              | DC              | DC              | DC              |
| cost of transaction UNDO       | +               | +               | +               | +               | +               |
| cost of partial REDO at restart | +               | +               | +               | +               | +               |
| cost of global UNDO at restart | +               | +               | +               | +               | +               |
| overhead during normal processing | +            | +               | +               | +               | +               |
| frequency of checkpoints       | +               | +               | +               | +               | +               |
| checkpoint cost                | ++              | ++              | ++              | ++              | ++              |

Notes: Abbreviations: DC, device consistent (checker); AC, action consistent; TC, Transaction consistent. Evaluation symbols: --, very low; -, low; ++, high; ++, very high.
Nesting, Savepoints, Chaining

- Various tools for programmer convenience
- Nesting: embedded transactions (code reuse)
  - outer transaction durable; abort cascades to top
  - allows isolation for concurrent nesting
- Savepoints: tryagain within transaction
  - rollback to savepoint; not durable until commit
  - Allows partial transaction rollback
- Chaining: series of related transactions
  - language trick for back2back transactions
  - “intermediate” commit makes durable changes

Eval

- Taxonomy/survey paper
  - no evaluation other than explaining tradeoffs in principle
  - Extensive reference to implementations, but all quite old now

Locking

- Common: two-phase locking
  - Once a transaction has released a lock, it may not obtain any additional locks
    - Growing phase, shrinking phase
    - Ensures serializability
- 2PL implemented by DBMS lock manager
  - Grants/blocks locks
  - Deals with deadlock
    - Avoidance? Detection?
      - Avoid: pre-declare locks, abort instead of block
      - Detect: timeouts (how long??), waits-for graph cycle (abort/rollback)

Selective Isolation

- Serializability may be too expensive
  - Consider data analysis prog that aggregates over 1,000,000s of tuples
    - Think Sawzall examples - approx. top-10 list
  - One or two inconsistent views may not matter
  - Relaxed consistency is a huge deal in many pragmatic systems
    - see, e.g., TACT paper - tunable availability and consistency trade-offs for distributed sys.
    - Example: Airline reservations may tolerate += 1 available seat using existing overbooking mechanisms
– Read uncommitted
  • No isolation! (fast & ugly)

– Read committed
  • A re-read of data may see data modified since start (but those mods. only done by committed Tx)
    – Allows DB to write committed Tx

– Repeatable read
  • On re-execute query, different result set may be returned (though values the same)
    – Allows deletes, etc. to be written