A Modular and Efficient Past State System for Berkeley DB

Ross Shaull
NuoDB

Liuba Shrira
Brandeis University

Barbara Liskov
MIT/CSAIL
Snapshots and Retrospection

• Past states of data can provide insights
  – trend analysis
  – anomaly and intrusion detection

• Auditing may require past-state retention

• Saving **consistent** past states (*snapshots*) is challenging and not available in all data stores
What is Retro

• Snapshot system for Berkeley DB implemented in a novel way

• The idea
  – Low-overhead (non-disruptive)
  – Simple programming model
  – Straightforward integration

• Approach
  – Layered design
  – Extend BDB protocols to create Retro protocols
begin;
insert into accounts values(...);
update accounts
set balance=0 where name=‘Tom’;
commit with snapshot(S);

select as of S * from accounts
where name = ‘Tom’
Snapshots are

- Consistent
- Global
- Named
- Application-declared

June 2014
Salary: $200K
Snapshots are

- Consistent
- Global
- Named
- Application-declared
Architecture

- Application
- Database
  - Interface
    - as of
    - snap
  - Access methods / indexes
  - Transactional storage manager
    - Snapshot layer
      - WAL
      - Page cache
      - MVCC
      - Snapshot layer works at the page level, inside the TSM
- DB Disk
- Retro Disk
Protocol extensions

Application

Database

Interface as of snap

Access methods / indexes

Transactional storage manager

Snapshot layer

WAL Page cache MVCC

Retro recovery

Retro page cache
- Snapshot page translation
- Concurrency for retrospection
- Efficient COW

DB Disk Retro Disk
Why this design for BDB?

- Logical-level snapshots require significant modifications to the data store
Why this design for BDB?

- Logical-level snapshots require significant modifications to the data store
- With low-level snapshots, it’s expensive to get consistency
Why this design for BDB?

- Logical-level snapshots require significant modifications to the data store
- With low-level snapshots, it’s expensive to get consistency
- Retro is not “too high” or “too low”
  - Simple integration and non-disruptive
Overwrite sequence (OWS)

- OWS(H) is a tagging of history H
  - which page pre-states to save
  - the snapshot pages a retrospective query accesses
  - which pre-states and snapshot declarations to recover
Not the first update to P since S1 was declared.
SavedAfter

- **Durable** table that tracks latest snapshot a page was saved after
  - Tracks latest “first update after” tag from OWS(H)

- **Used when**
  - Performing retrospection
  - Saving snapshot pages (normal operation & recovery)

- **Can be costly because it is shared data structure**
  - SavedAfter Cache accelerates SavedAfter by scribbling tag on page header in page cache
Snapshot pages and Page Sharing

T1: update P1, declare S2
T2: update P1

Transactions

Page cache

DB Disk

Retro Disk
Protocol extensions: Recovery

• Like database, snapshots are written asynchronously (non-disruptiveness)
• Retro saves pre-states during BDB recovery
  – Snapshot declarations are also logged
• Identify needed pre-states using SavedAfter during recovery
Protocol extension: Recovery

• Runtime invariants
  – Snapshots are made durable first: WAS-invariant

• Recovery-time extension
  – Recover snapshot metadata first
  – Idempotent: Start, SavedAfter tell if pre-state was saved already
Protocol extension: MVCC

• Concurrent access to current state and snapshots
• Efficient copying of snapshots
• Retrospection runs using MVCC and page requests are redirected to snapshot pages that have migrated to pagelog
Retrospection (querying as of)

- Interface
- Access methods / indexes
  - Get (DBFile, P)
- Snapshot Layer
  - Page name translation
    - Look up (DBFile, P)@S1
  - Get (RetroFile, X)
  - Page cache
- Database disk
- Retro disk
Current state queries

- Interface
- Access methods / indexes
- Get (DBFile, P)
- Snapshot Layer
  - Page name translation
  - Page cache
- Database disk
- Retro disk
Gluing it together

- Implemented as a set of callbacks
  - About 250 lines of modifications to BDB source
  - Call into about 5000 lines of snapshot layer code
- Retro is thread-safe
- Care taken to follow OWS(H) order in face of concurrency
Experimental Results

- Database and snapshot data are written to one disk, logs to the other.
- Database size is 1 gb.
- Snapshot store on Retro disk can be >100 gb.
- **Non-disruptiveness**
  - Random update workload with and without Retro.
  - With Retro, declare snapshot after every transaction.
  - Enforcing invariants for snapshot durability imposes about 4% overhead on throughput.
Retrospection: Overhead

![Graph showing the cost in terms of Q for Retro(Q) old and Retro(Q) cur workloads.](image)
Retrospection: I/O

The graph illustrates the cost in terms of Q skew for different data sizes and skew ratios. The x-axis represents skew ratios (50/50, 80/20, 90/10), and the y-axis represents the cost in terms of Q. The graph shows the cost for 100mb and 1gb data sets, with the cost for build page table and query tasks differentiated by colors.
Conclusions

• Simple, novel design for adding retrospection
  – Yet supports powerful programming model
• Non-disruptive, long-lived snapshots
  – Key to useful snapshot system
• Layered approach
  – Flexible and relatively low-level, generalizes
• Extended standard transactional algorithms
Thank you

• Questions?