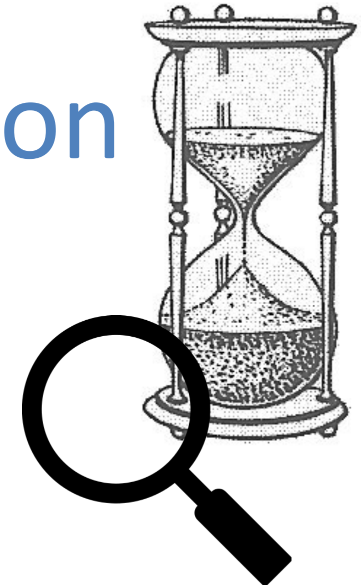
A Modular and Efficient Past State System for Berkeley DB

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

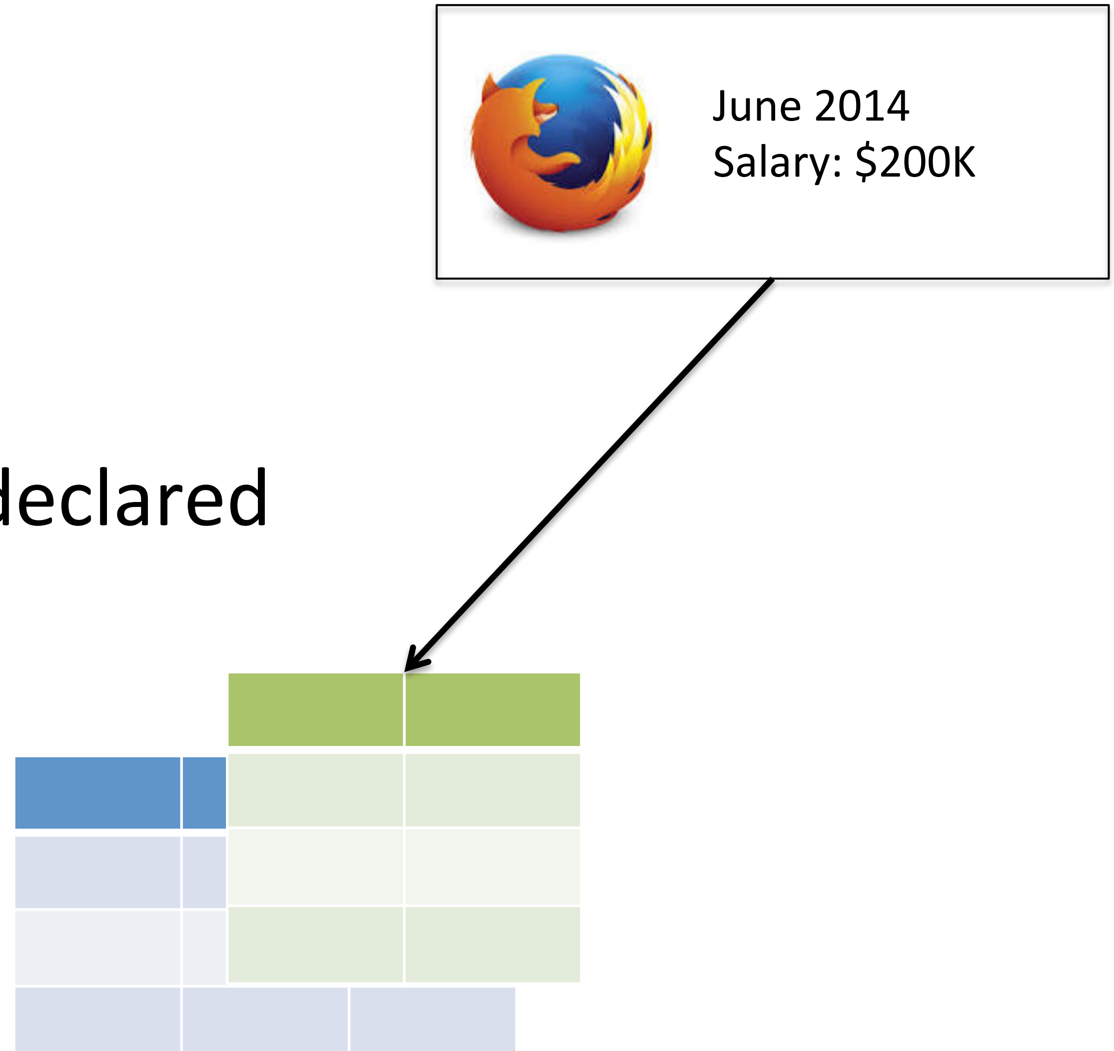
Programming Model

```
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'
```

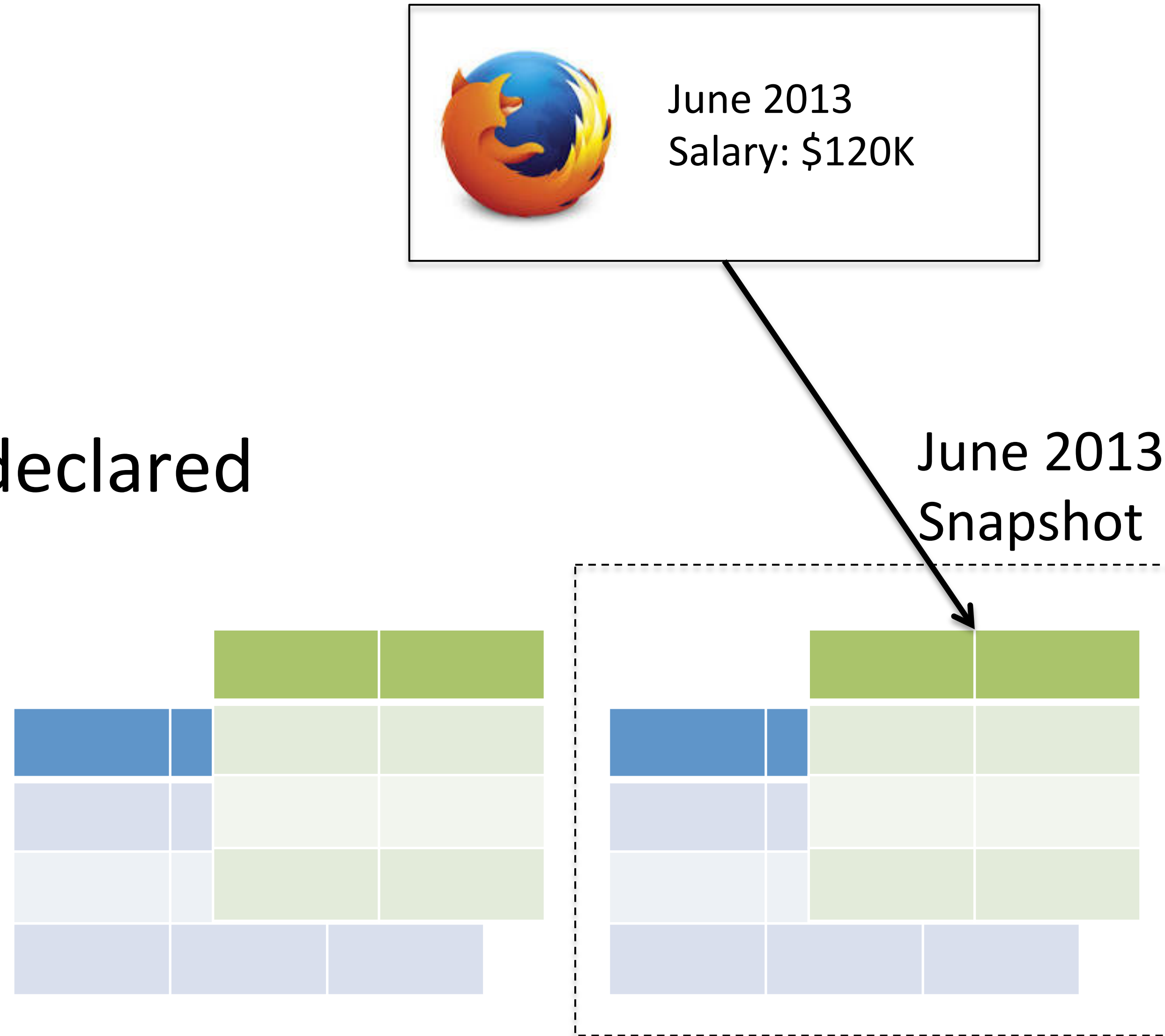
Snapshots are

- Consistent
- Global
- Named
- Application-declared

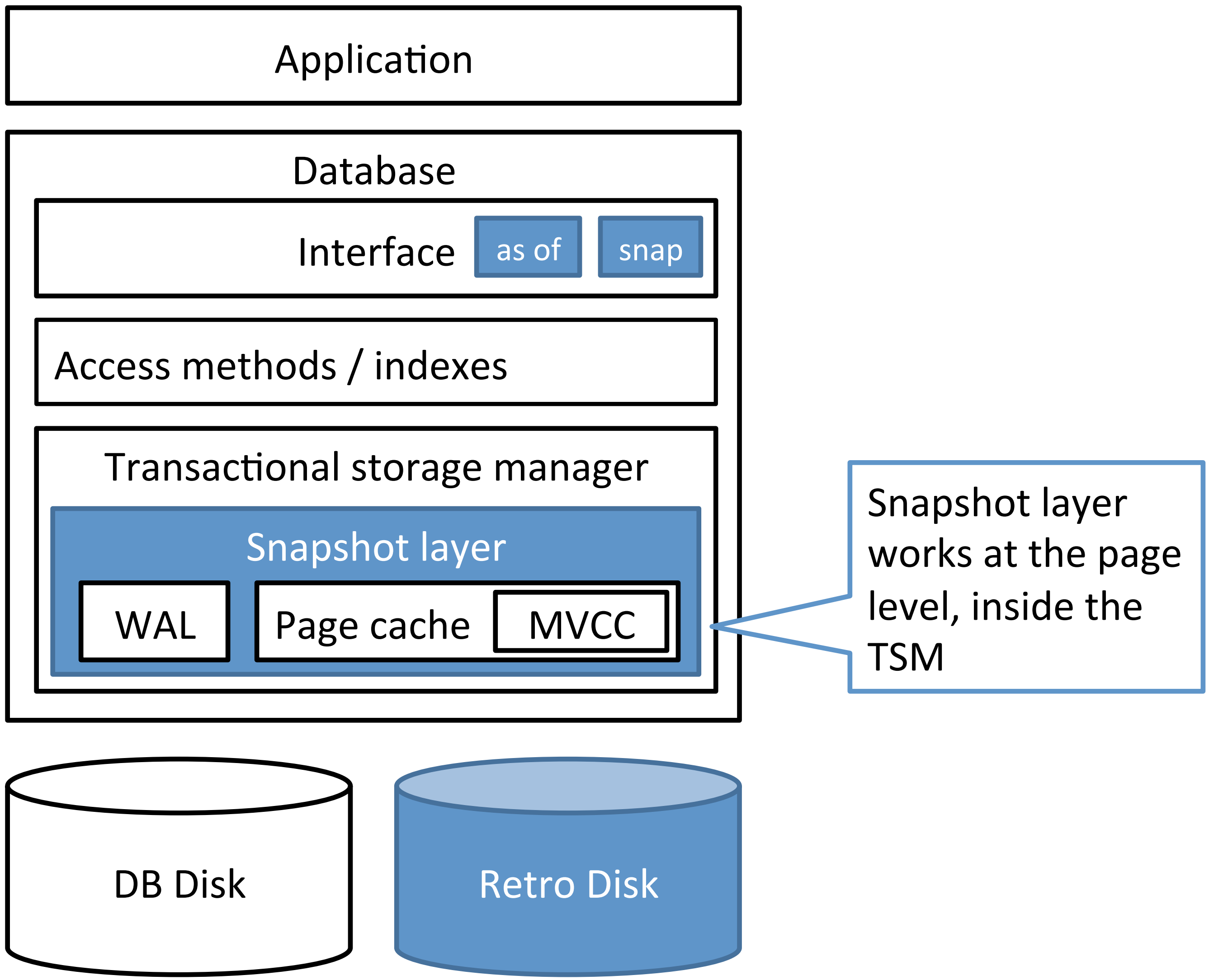


Snapshots are

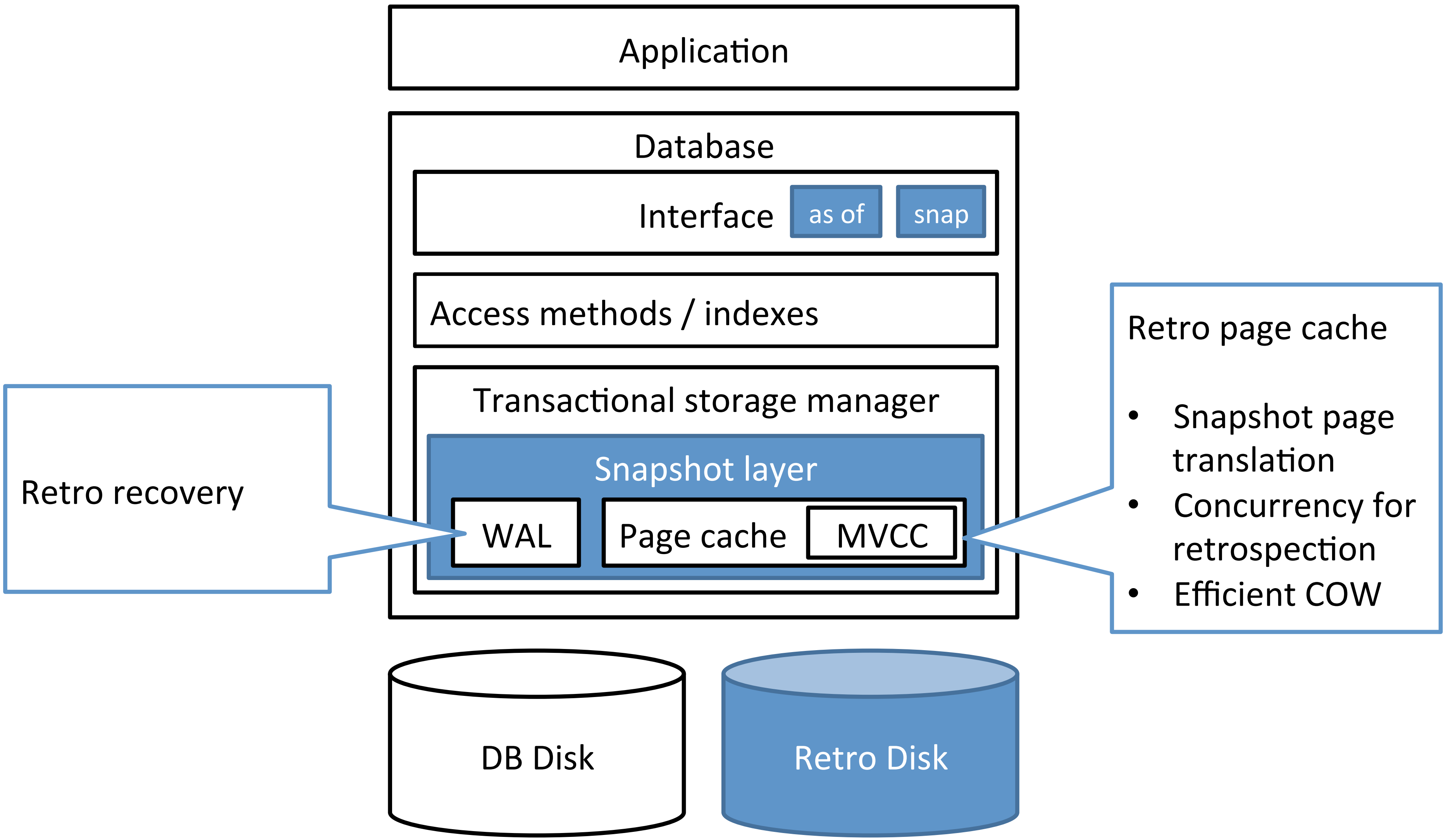
- Consistent
- Global
- Named
- Application-declared



Architecture

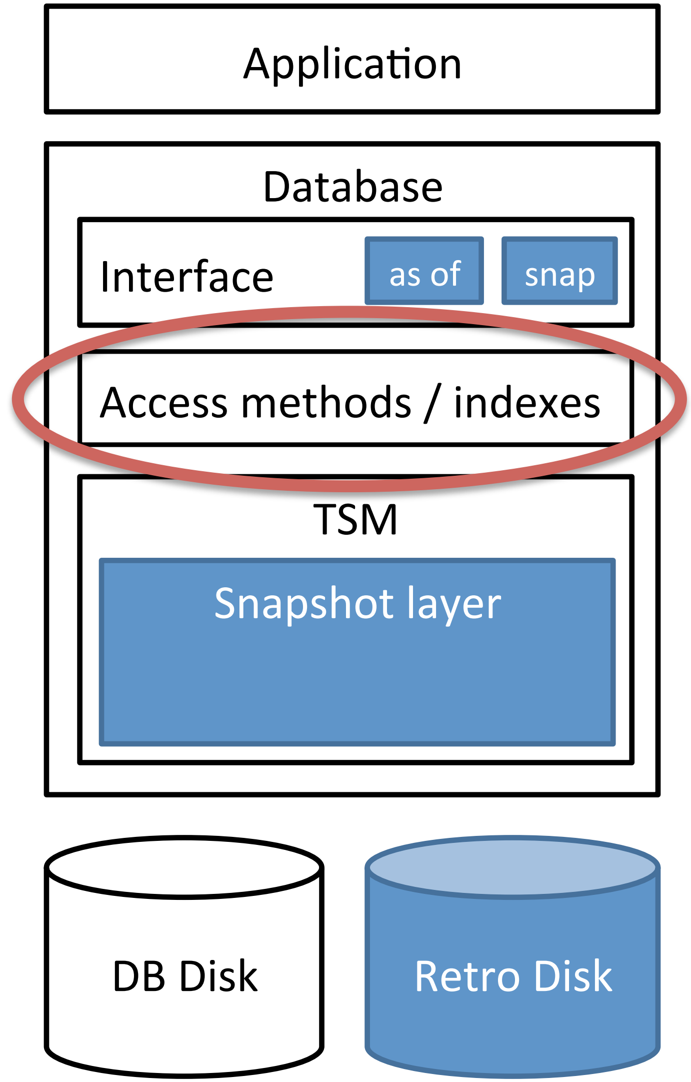


Protocol extensions



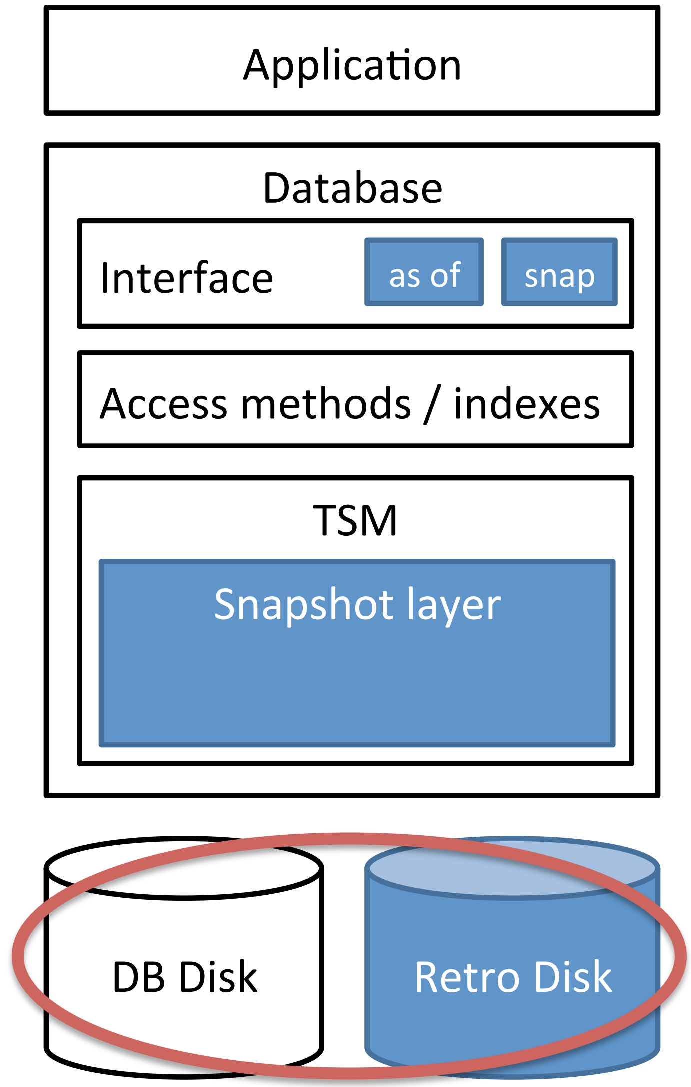
Why this design for BDB?

- Logical-level snapshots require significant modifications to the data store



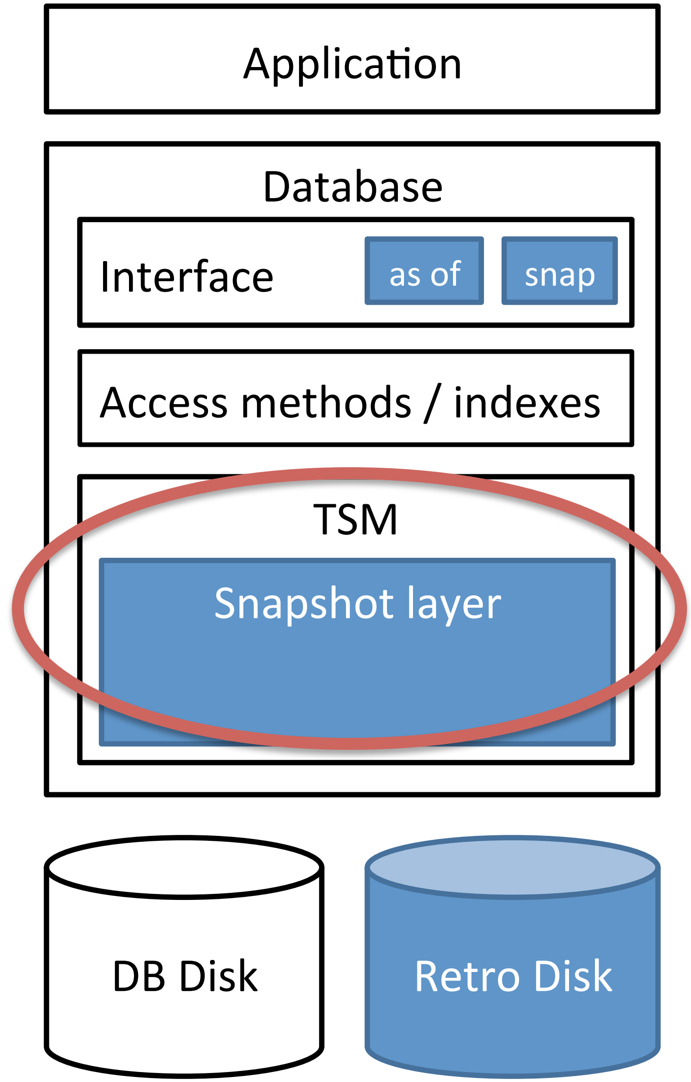
Why this design for BDB?

- 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?

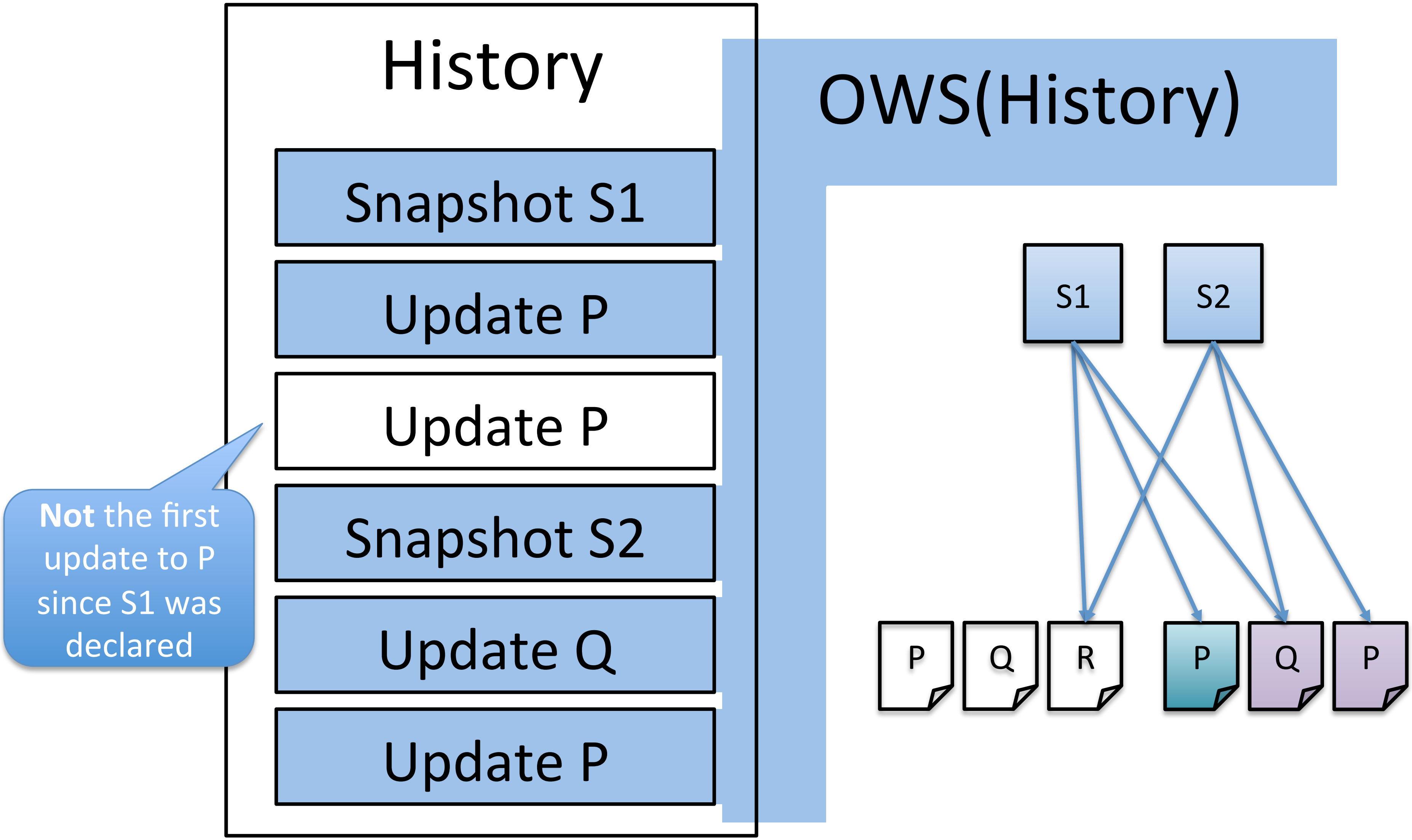
- Logical-level snapshots require significant modifications to the data store
- With low-level snapshots, it's expensive to get consistency
- Retro is not "too high" or "too low"
  - Simple integration and non-disruptive



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

OWS Example

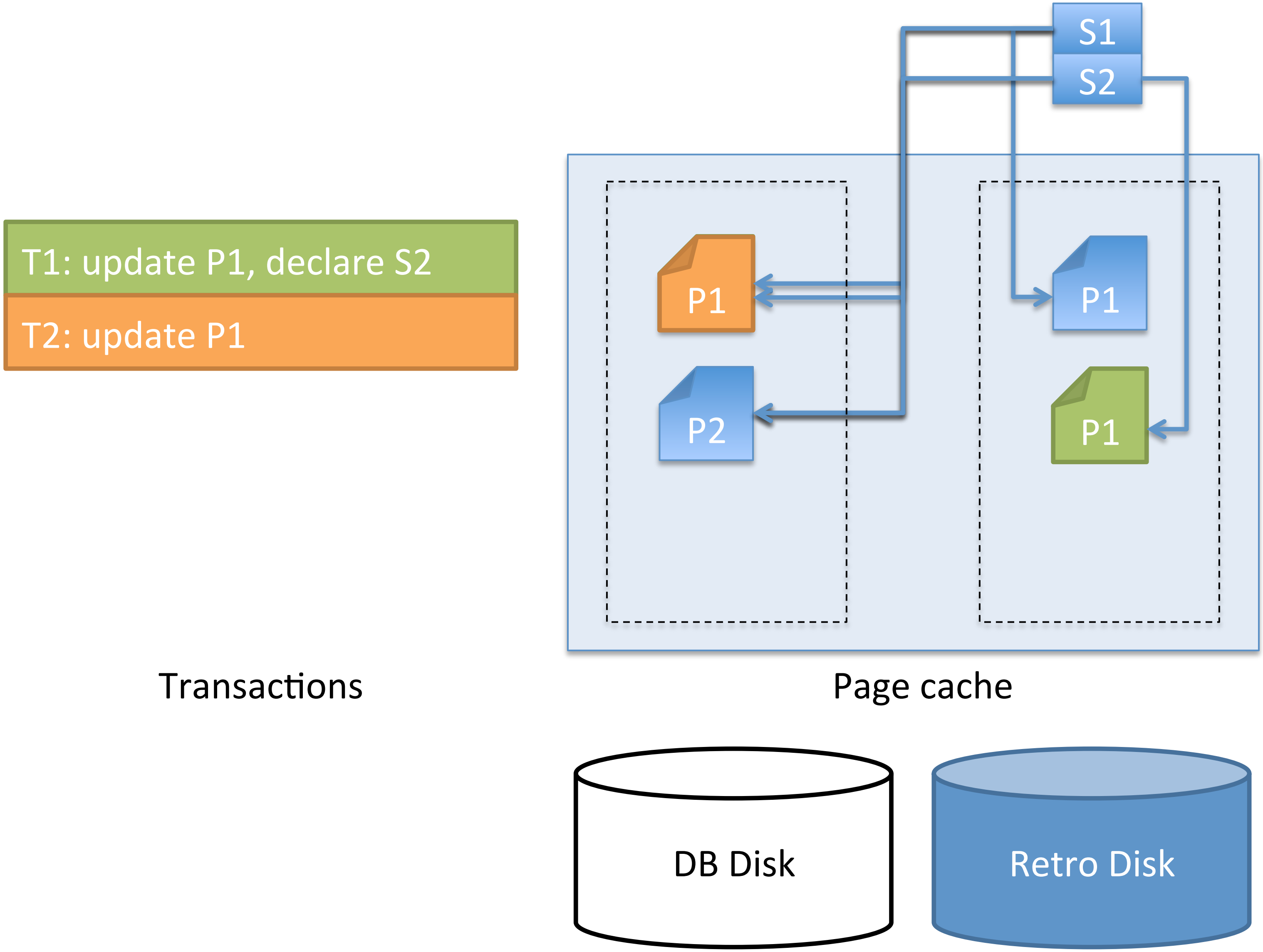


History
Snapshot S1
Update P
Update P
Snapshot S2
Update Q
Update P
OWS(History)
S1
S2
P
Q
R
P
Q
P
Not the first update to P since S1 was declared

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



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

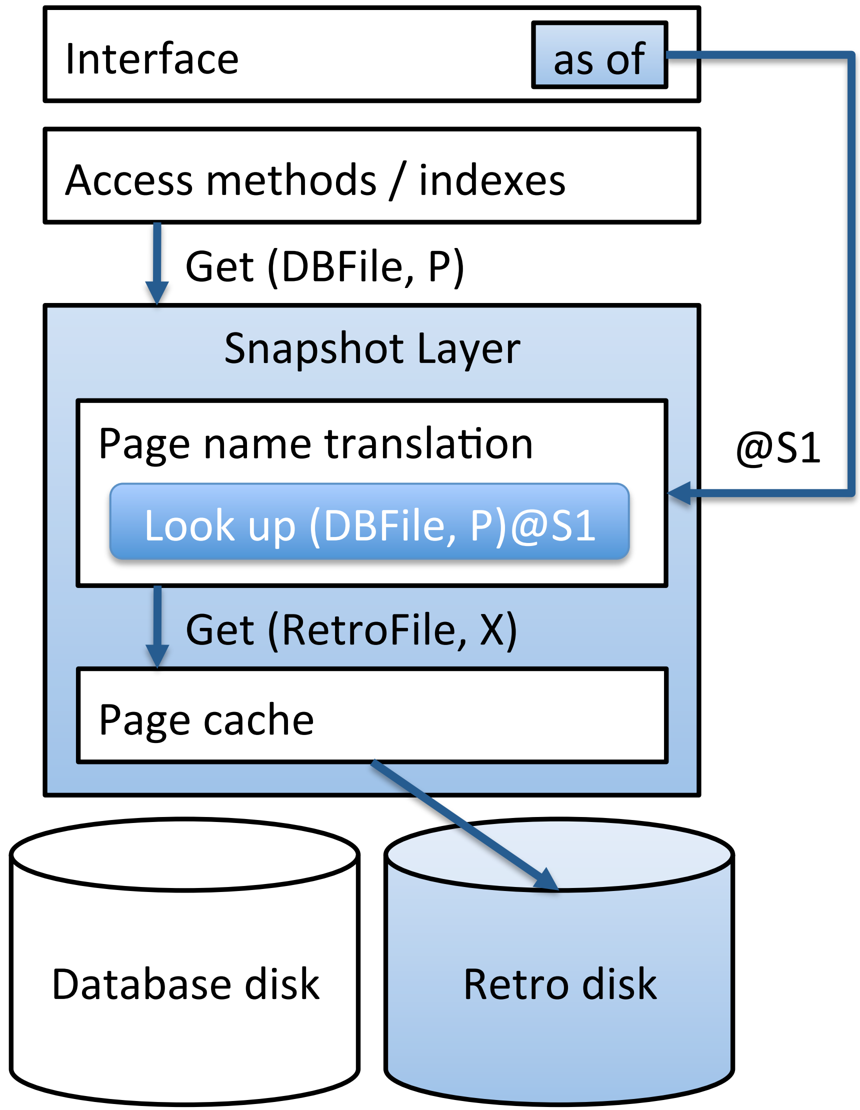
Protocol extension: Recovery

- 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

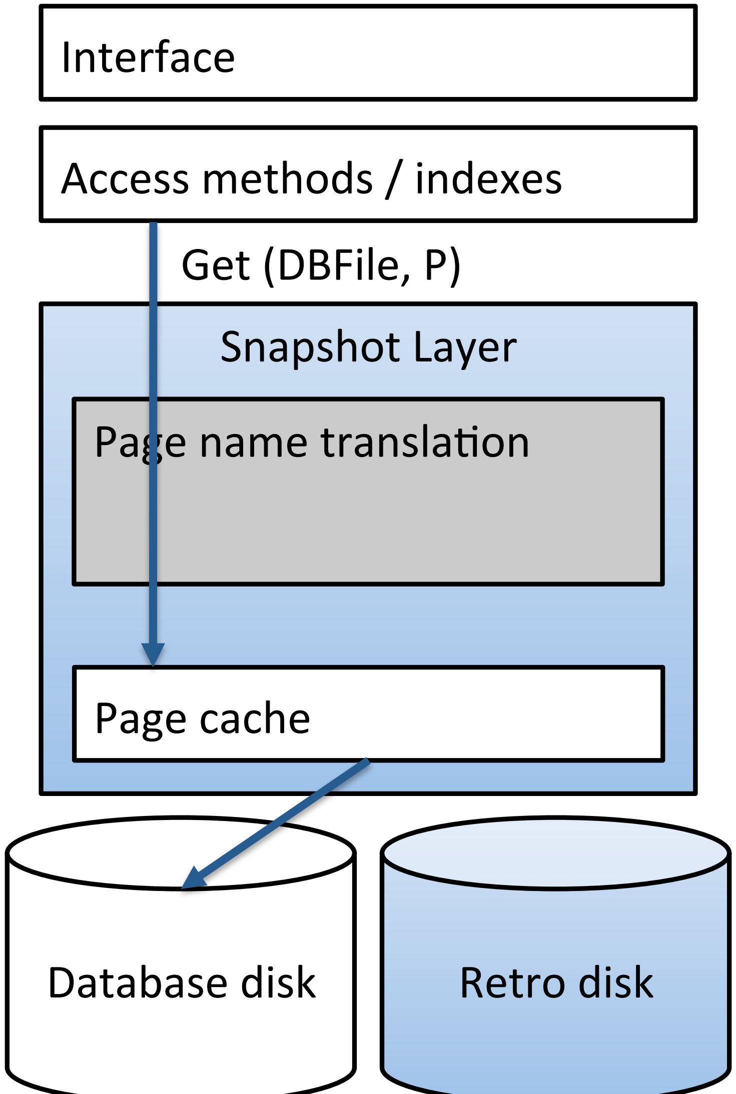
Protocol extension: MVCC

- 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*)



Current state queries



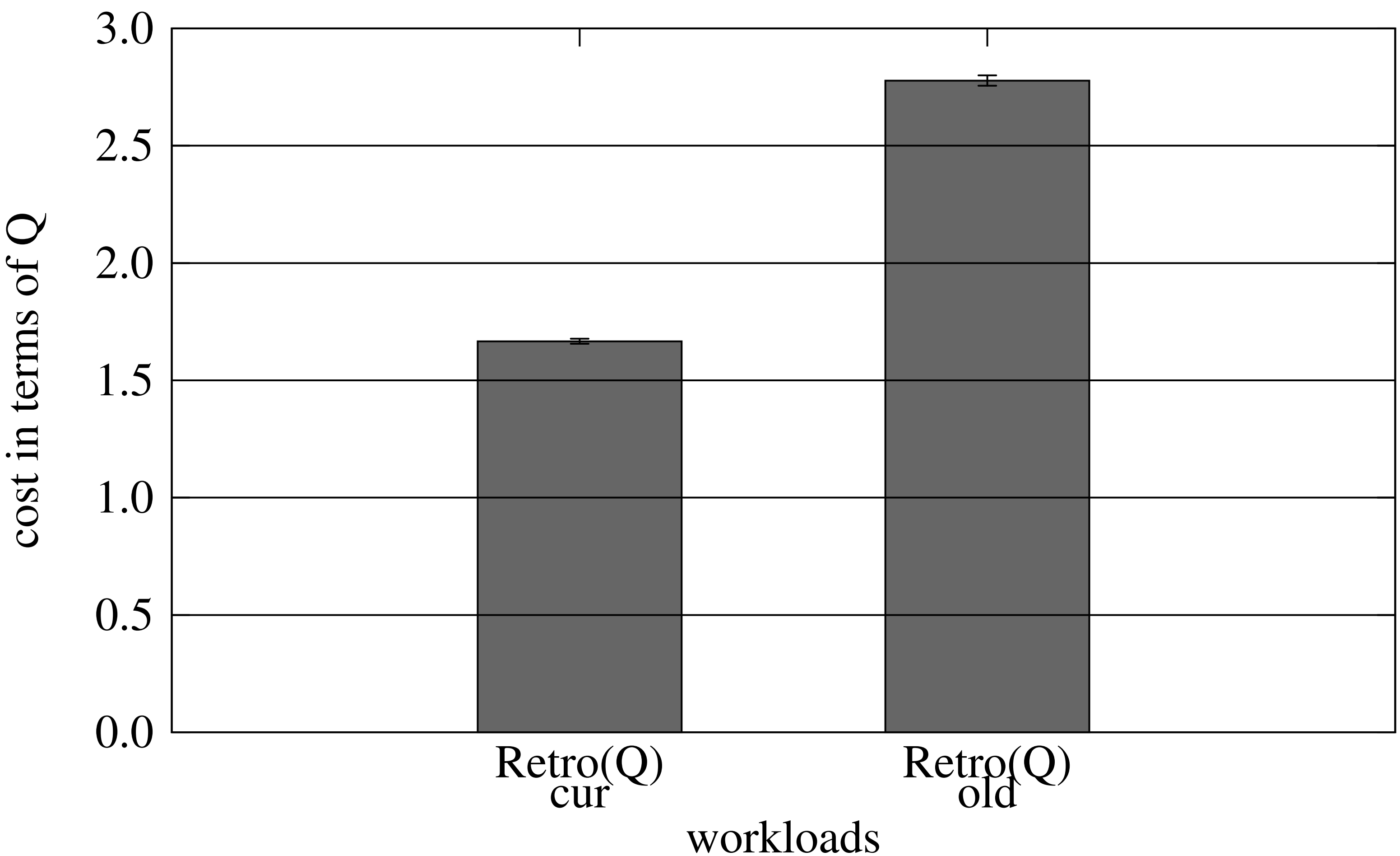
Gluing it together

- 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

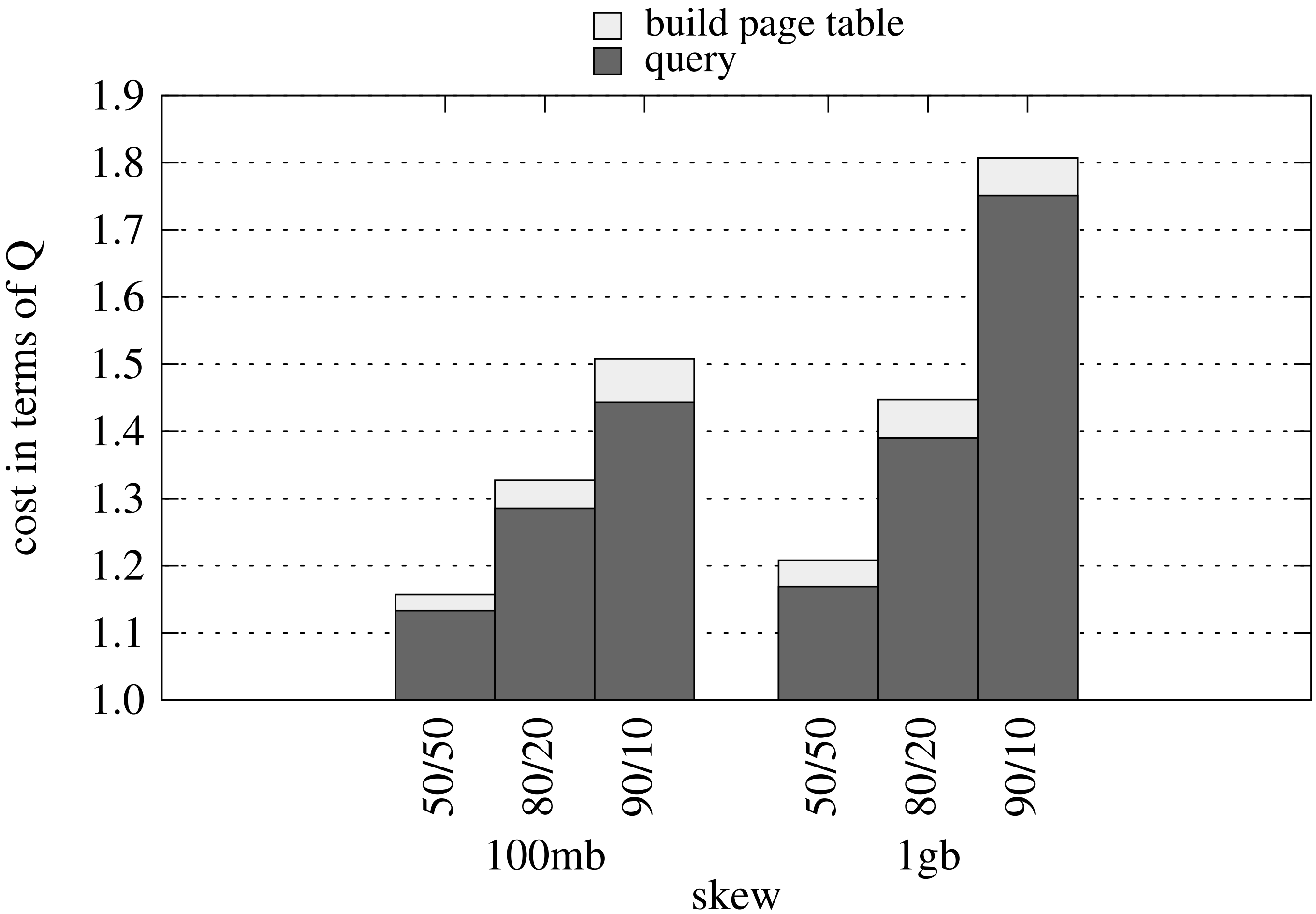
Experimental Results

- 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



Retrospection: I/O



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?