Travel in the virtualized past: cheap fares and first class seats

Liuba Shrira

(with Ross Shaull (Brandeis) and Catharine van Ingen (Microsoft))

Long-lived snapshots

So, disk is cheap

a storage system can take snapshots of past states and retain for a long time

Past state analysis is increasingly important…

CRM: Casino upgrades coupons for hi-spenders **(the morning after)**

ICU monitoring system: past response to drugs, interesting snapshots: abnormalities, specialists' visits.. **(one lifetime)**

Wikipedia citation in a legal ruling: what was judge Posner thinking? **(many lifetimes)**

How to support interesting past state analysis over long time?

Premise: **BITE**

what you need is

a storage system capability for

back-in-time execution (BITE):

run read-only applications against

snapshots of past states **in addition** to

current state

to answer in **real-time**

new and **old** questions

What can you do with BITE?

Analyze past: to "predict future"

Reflect:

Organize past: "selective memory"
rank with BITE, keep interesting stuff for longer

Verify past: "audited memory"
(spotless mind..)
validate constraints with BITE,
undo/fix "bad" transactions + dependents

BITE Snapshots: Semantics

Consistent snapshots: invariants hold for old code
(consistency differs in different systems)

BITE of general code:
(ad-hoc new code vs canned queries)

Application chooses the snapshot: meaningful to app
(vs "some time in the past" in SI, or every 30sec)
at high "resolution" (vs backup)

BITE Snapshots: Implementation

Where is your long-lived past ?

physically -
today: **too close**
(Postgress, Temporal DB, CVFS) **disruptive** in long term
or too far
(warehouse: Netezza) **no real-time analysis**

and logically, in the software stack -
too high
(e.g. logical record level) - **complex**
or low
(e.g. VSS, below cache) – **disruptive for consistent snapshots**

We want:

"Right" look:

snapshots, look like current state

(not the other way around – like temporal DB)

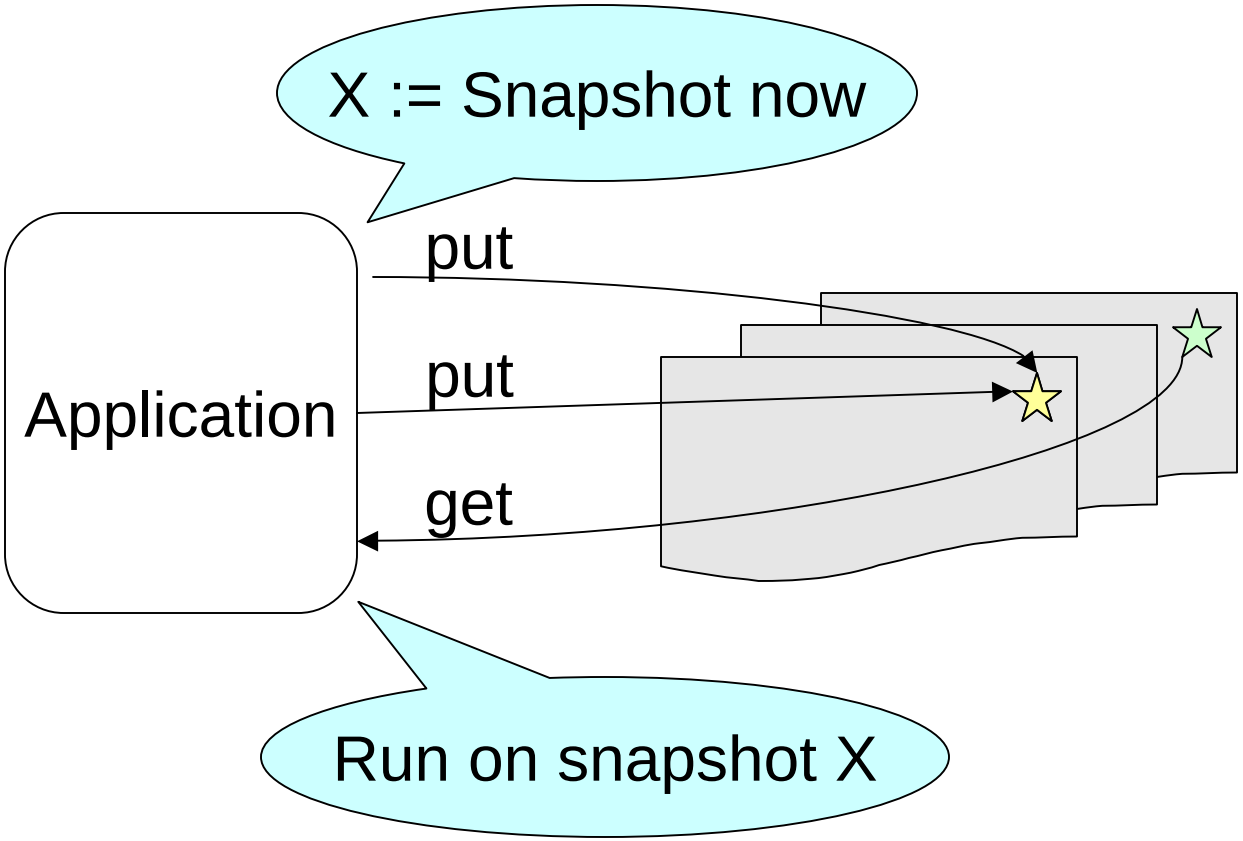
"Right" distance:

run BITE programs in real-time in-house non-disruptive to the storage system (short or long term)

The “right look” -

past virtualized as current state

Our Snapshot System



Applications "time travel" in a past virtualized to look like present.

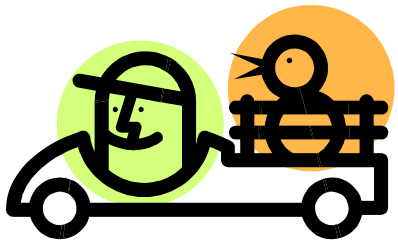
Back-In-Time Execution (BITE)

The "right distance" -

a snapshot box inside every storage system
runs code over snapshots in real-time

in-house (not warehouse)

"..a chicken in every pot"..



A snapshot: Interface

Current state DB storage: pages + page table

A Snapshot: virtualizes Db storage

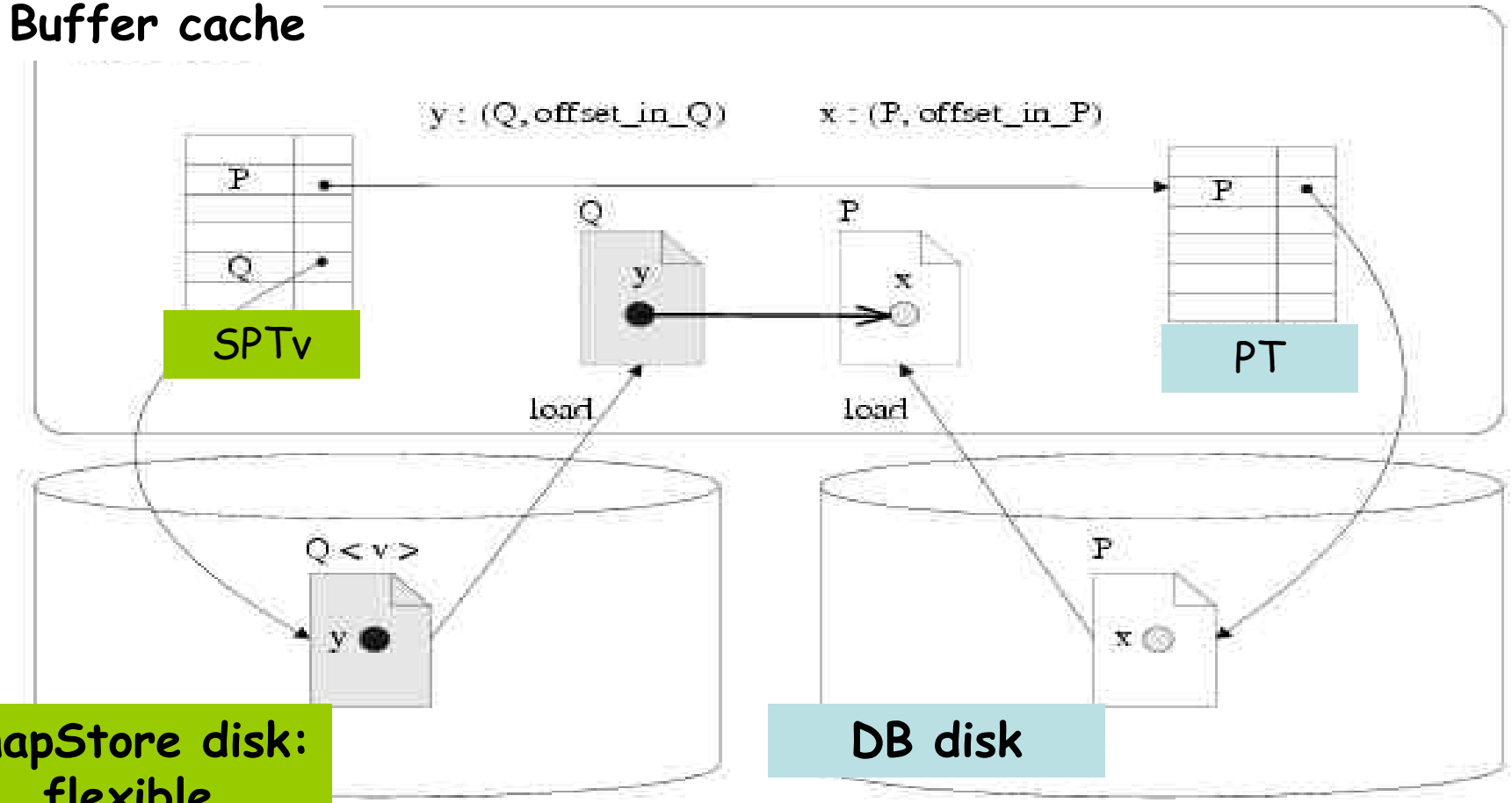
snapshot pages + snapshot page table

So **BITE is transparent**:

for snapshot v mount Snapshot Page Table(v)

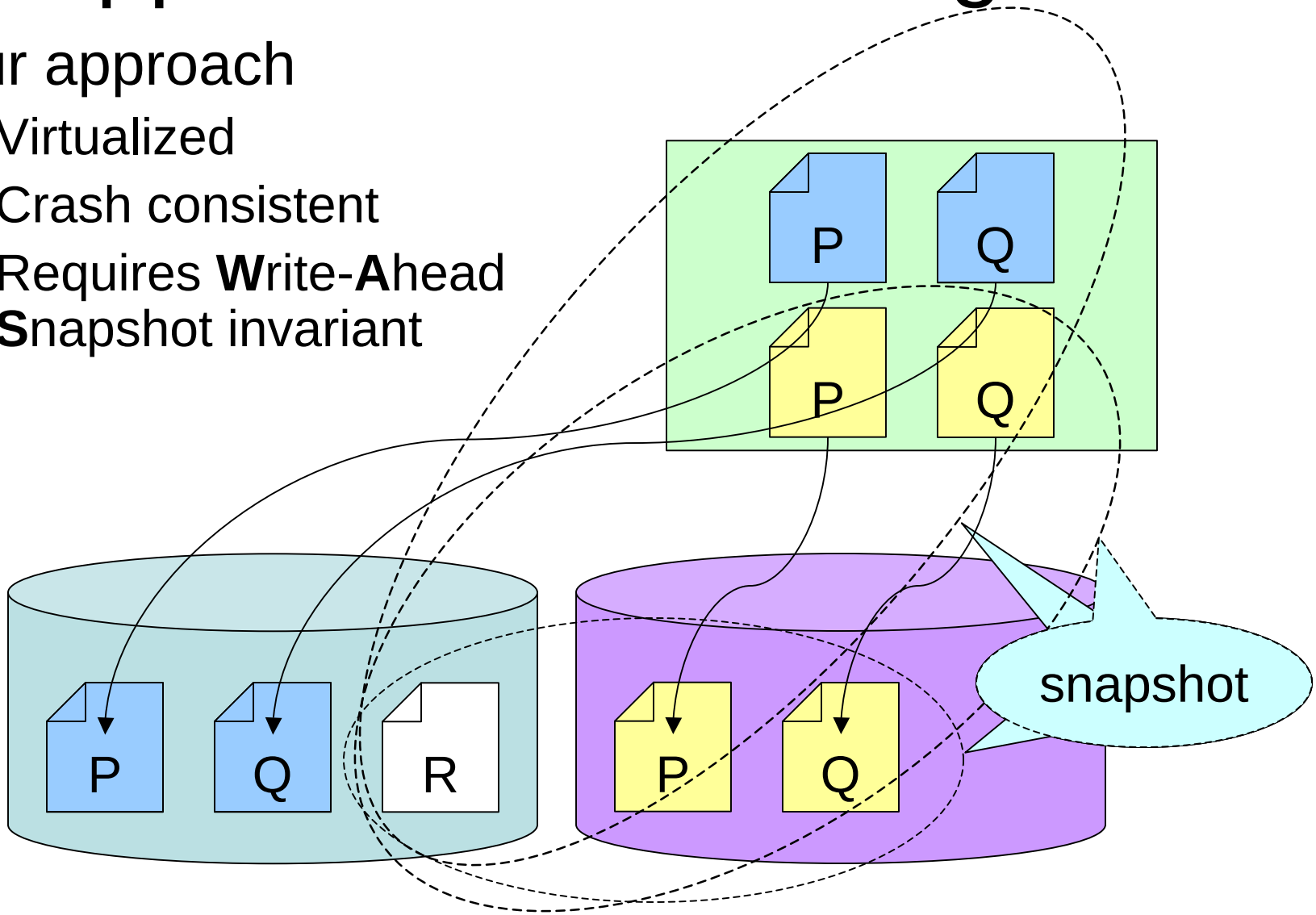
BITE(v): code accesses snapshot V pages

(1) page Q (modified after v) (2) P (unmodified)



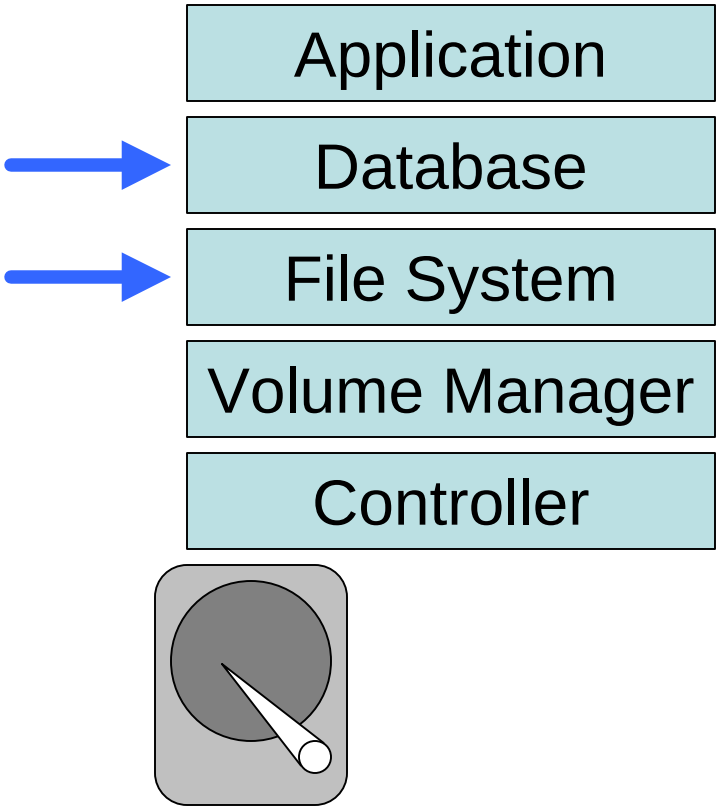
Our Approach: Cache Integration

- Our approach
  - Virtualized
  - Crash consistent
  - Requires **W**rite-**A**head **S**napshot invariant



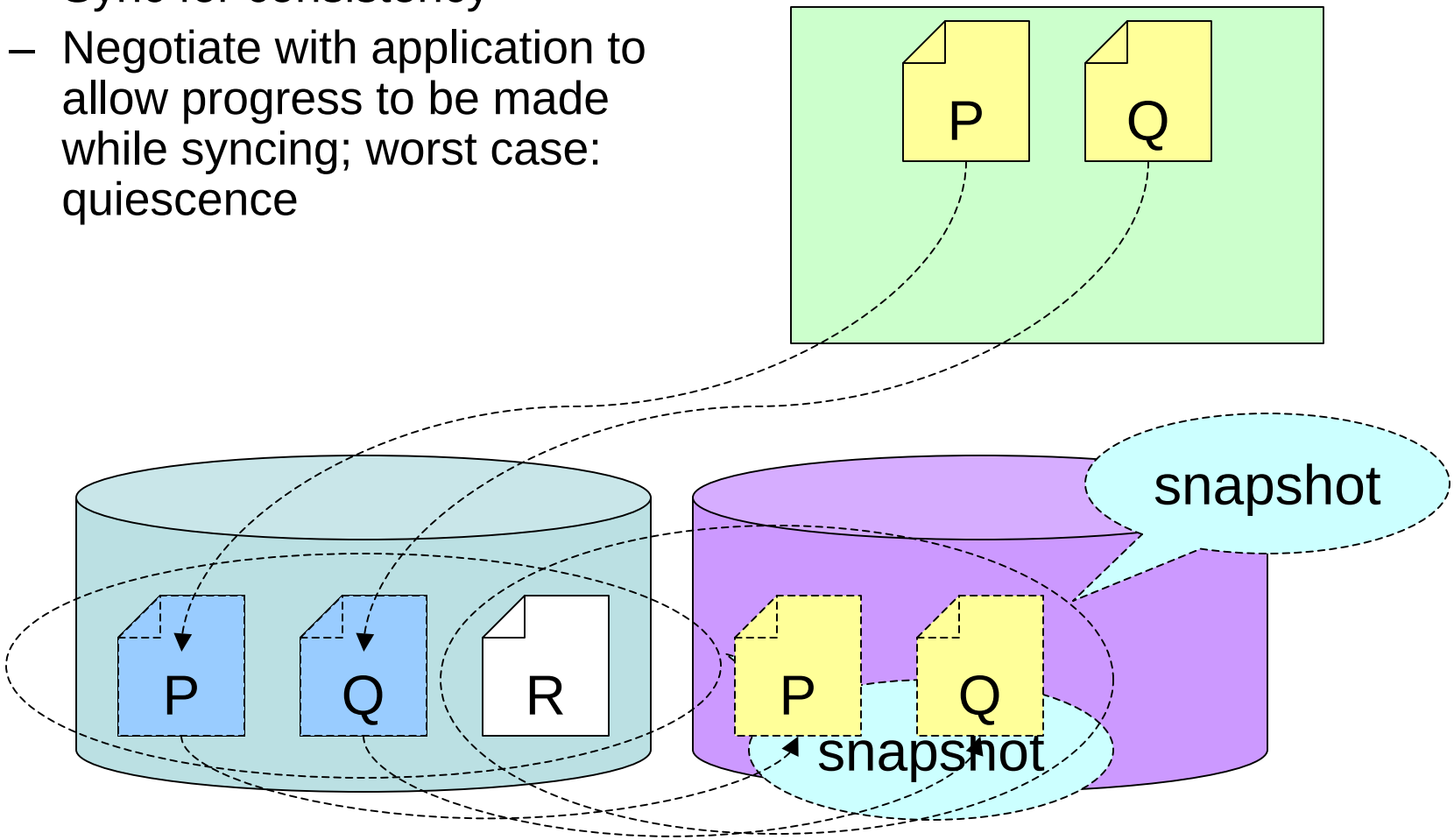
Best Level for BITE?

- High level
  - Database, file system
  - Leverage recovery
  - Delay writes



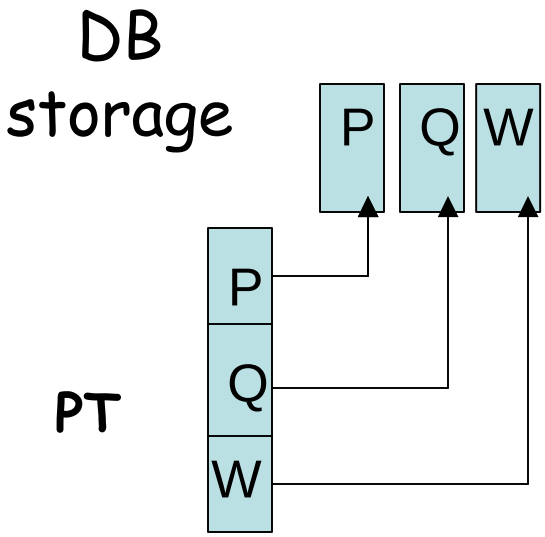
Without cache: disk COW

- On disk
  - Sync for consistency
  - Negotiate with application to allow progress to be made while syncing; worst case: quiescence



More details

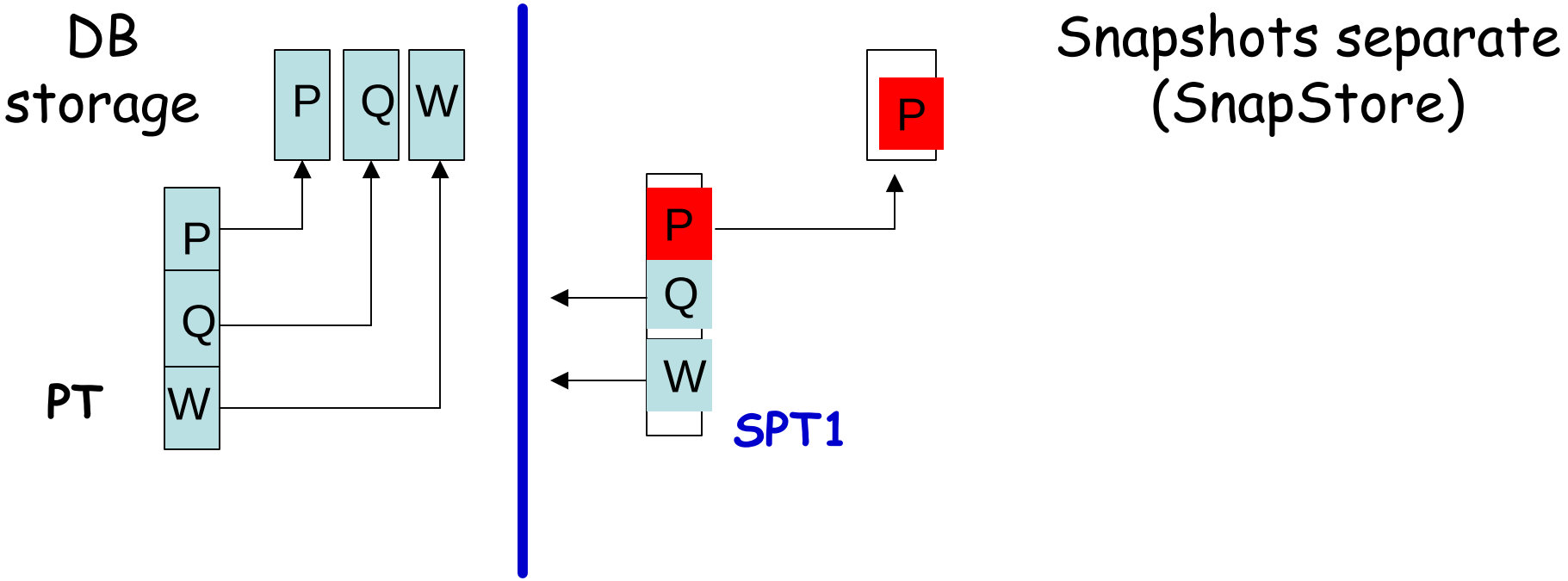
Split COW



step 1: app declares a snapshot v1

step 2: app modifies page P

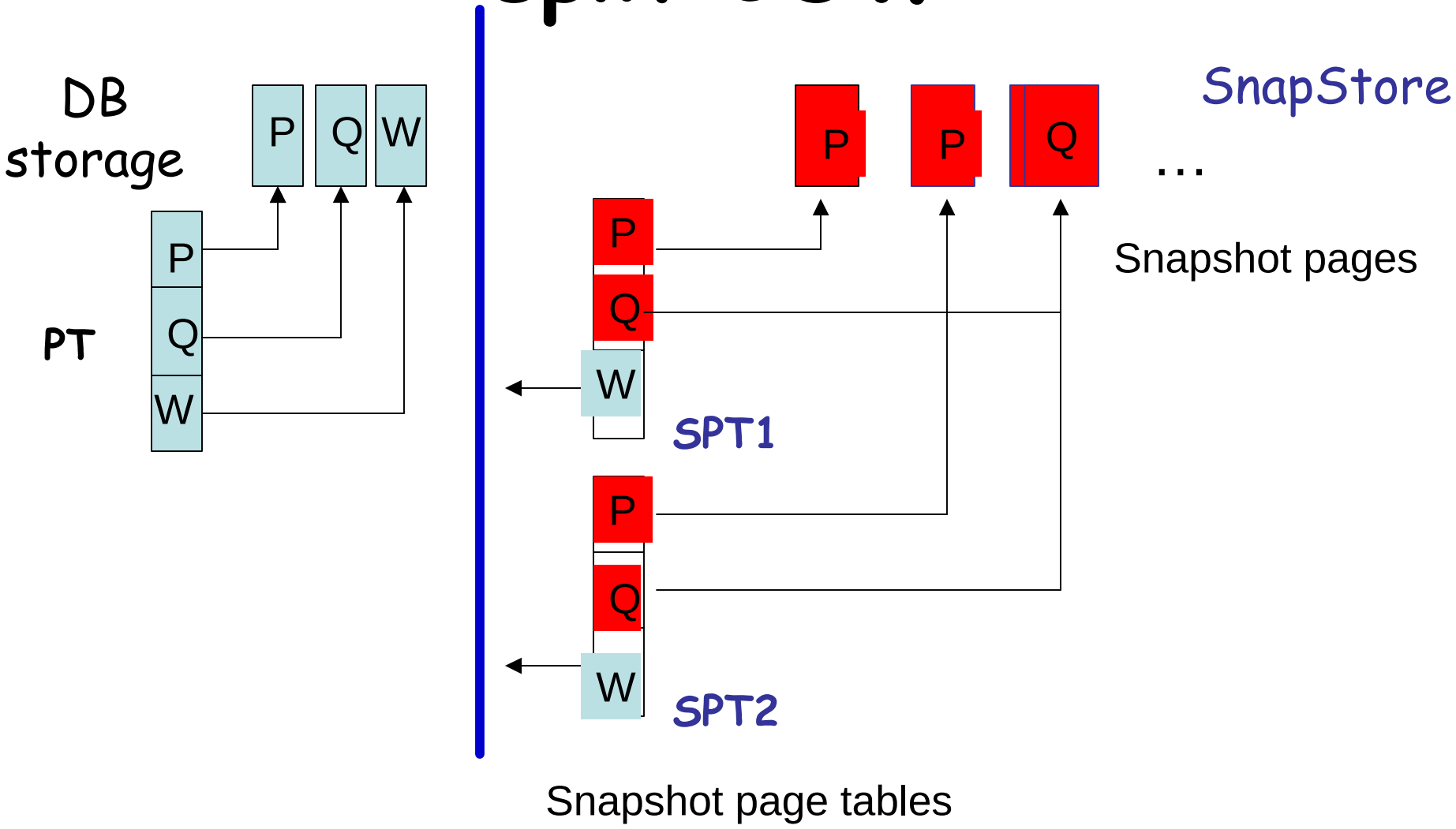
split COW



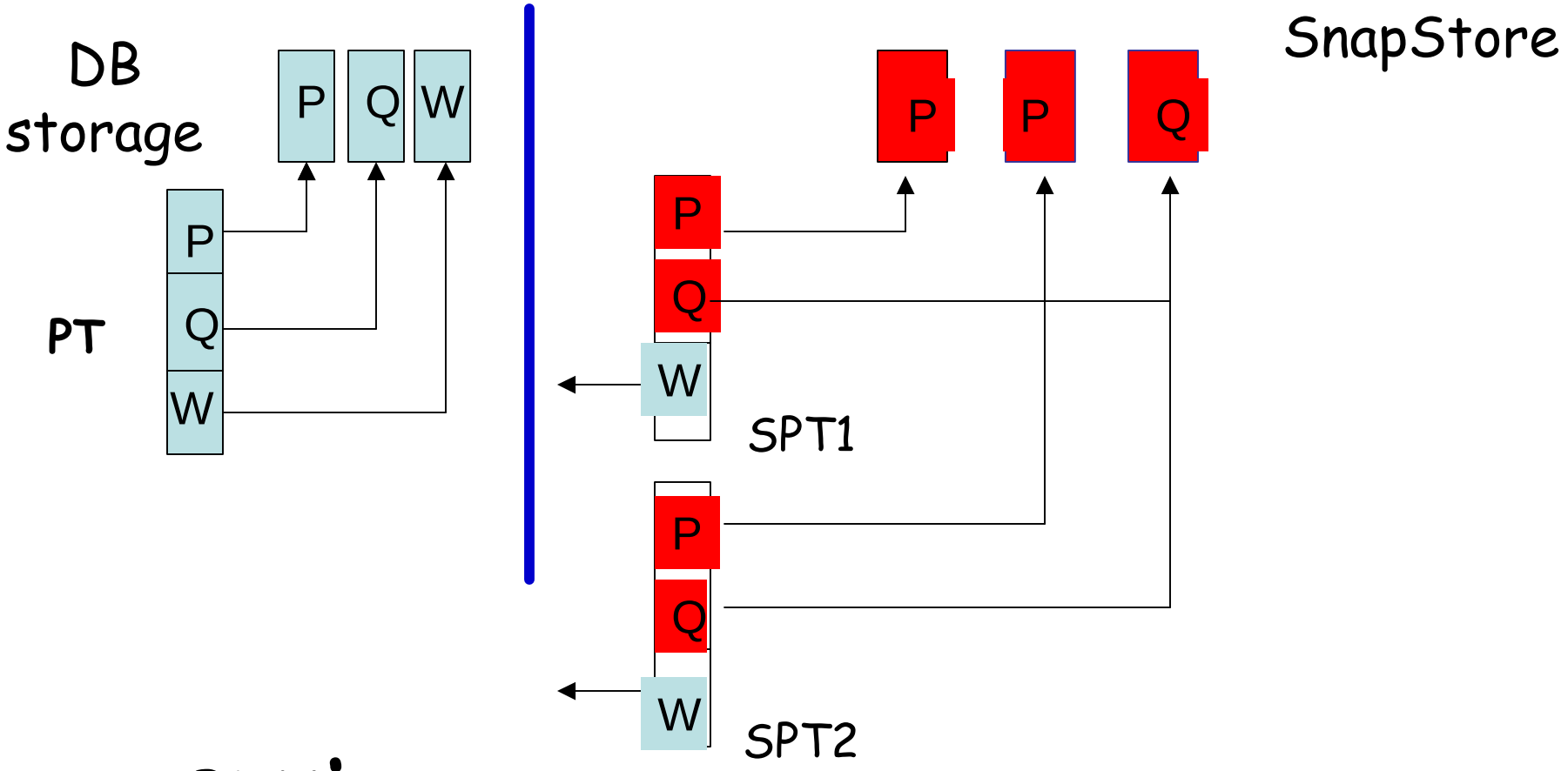
First P update after v1 retains before-image of P

cont… app declares snapshot v2
app commits updates to P, Q

split COW


DB storage
P
Q
W
PT
P
Q
W
SnapStore
P
P
Q
...
Snapshot pages
P
Q
W
SPT1
P
Q
W
SPT2
Snapshot page tables

split COW



BUY:

update in-place

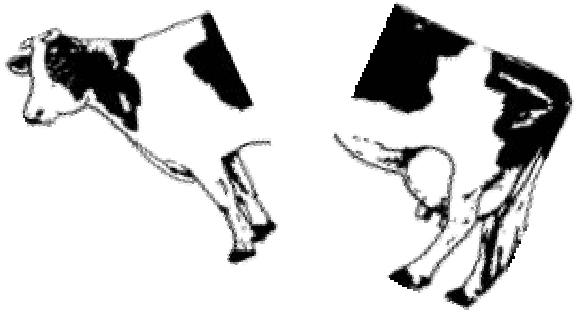
Pay extra write

but no declustering

(cheap to change snapshot rep:

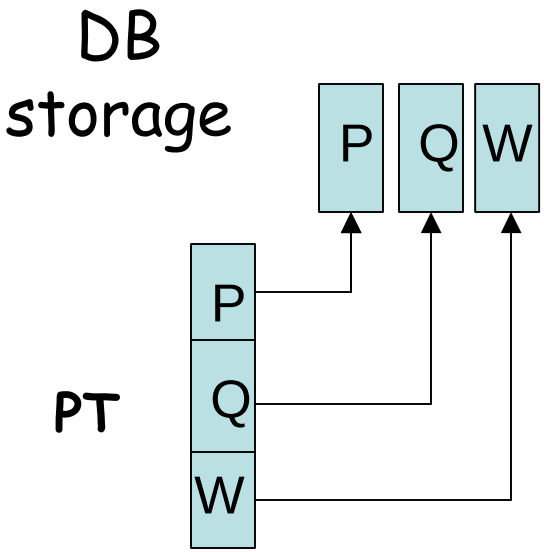
how (diff, stripe, crypto)

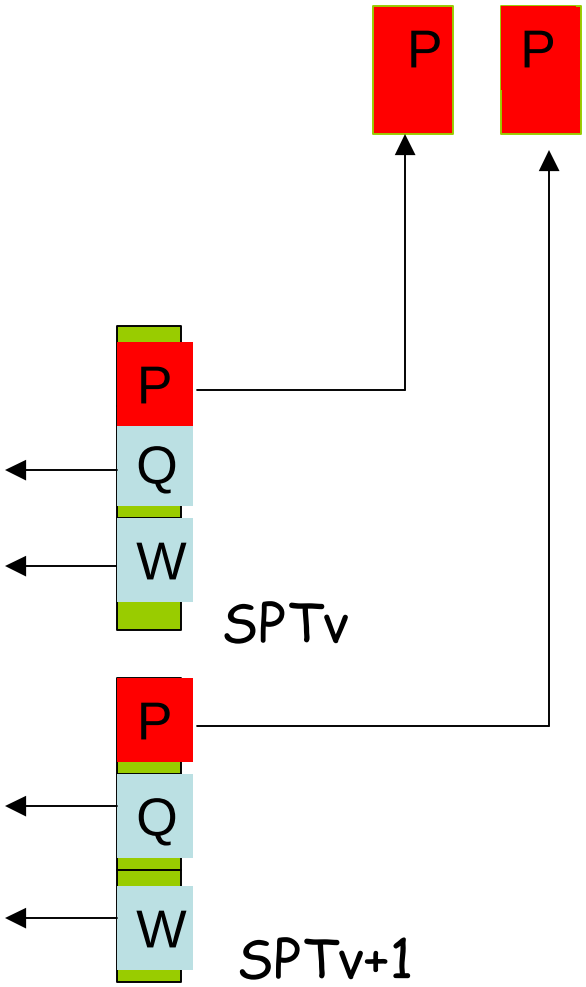
and where you write)



Problem: finding snapshot pages

Indexing split COW snapshots





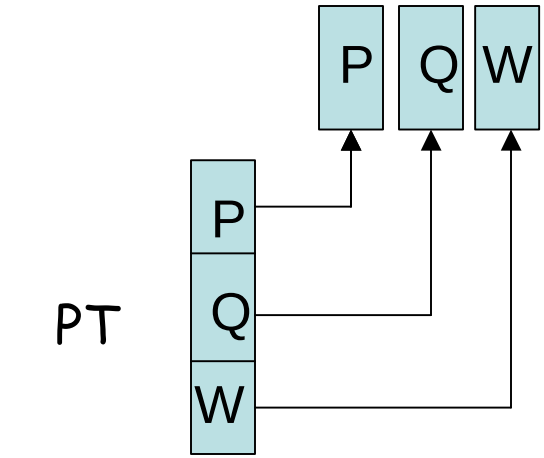
...

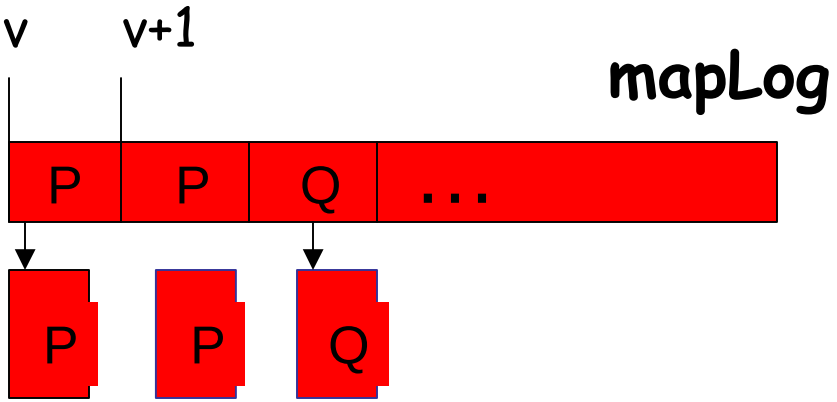
Problem:

Code needs to find snapshot (v) pages For BITE (v)

But updating snapshot page tables can be costly

Instead: write snapshot page mappings in a log





Lookup:
scan the mapLog

MapLog: a new indexing method for split COW snapshots

Key notion: **FEM** – first encountered mapping

Notice where the mappings for v start in a log

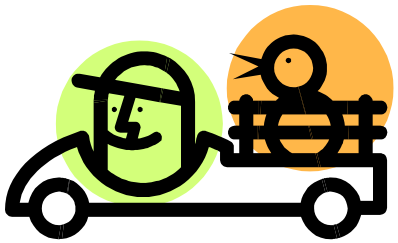
Write mappings in correct order (decoupled from page order)

Mapping Order Invariant:

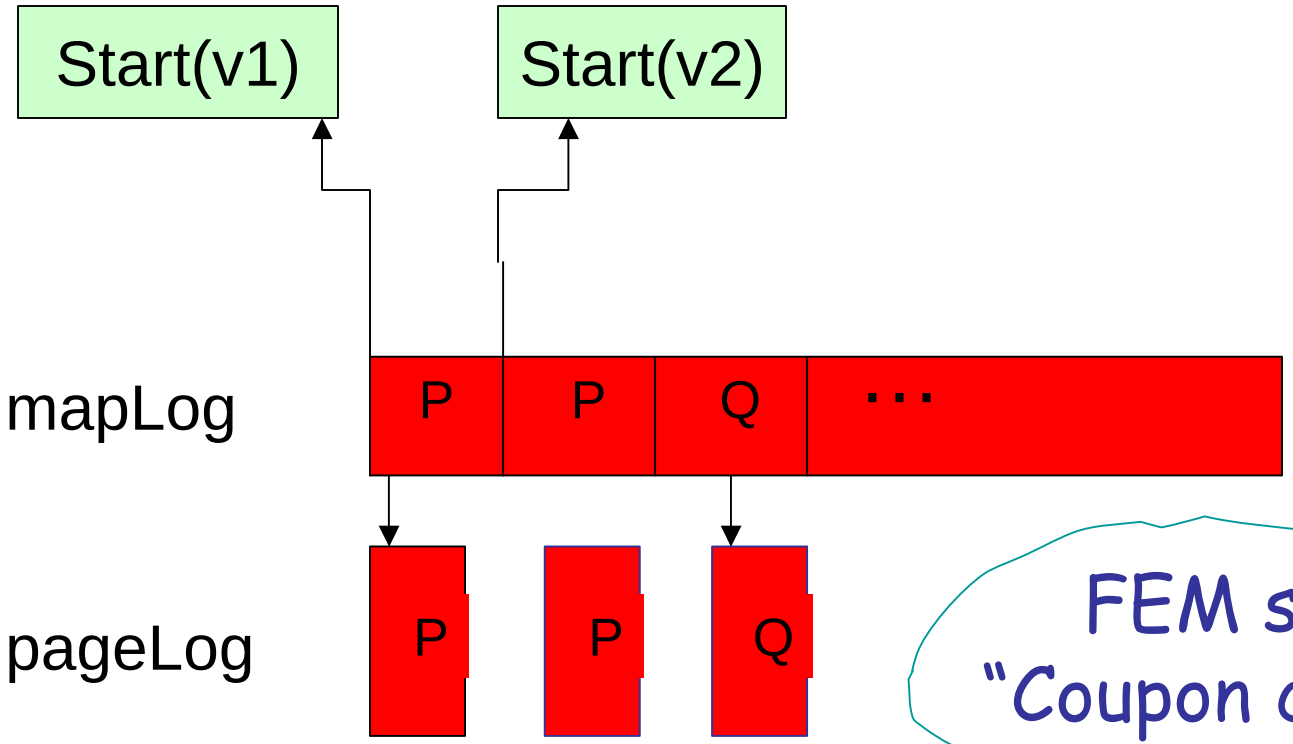
mappings retained for snapshot v,
are written before
mappings retained for snapshot v+1

Lookup

scan mapLog from start v collecting **FEMs**



MapLog algorithm:

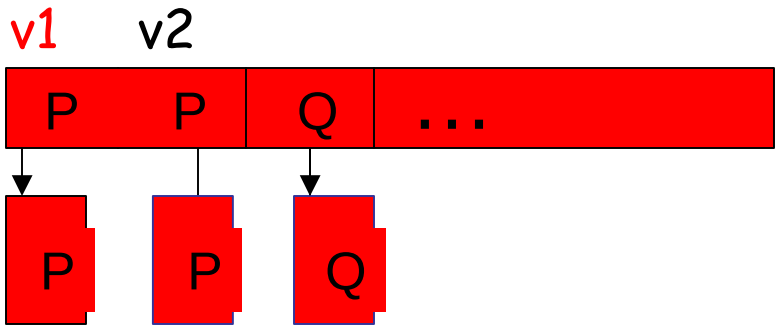


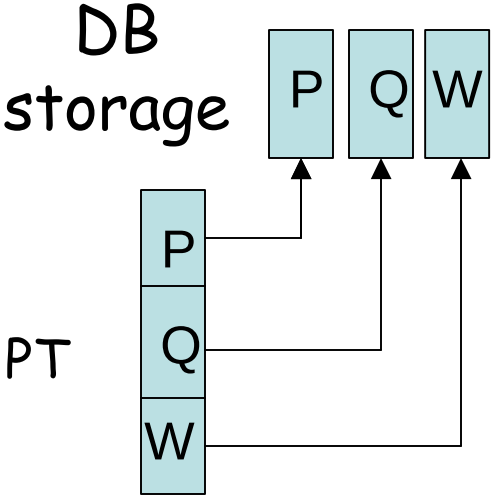
FEM search =
"Coupon collection"

To lookup page P for snapshot v:

scan mapLog from Start(v1) to FEM(P)

Skewed updates



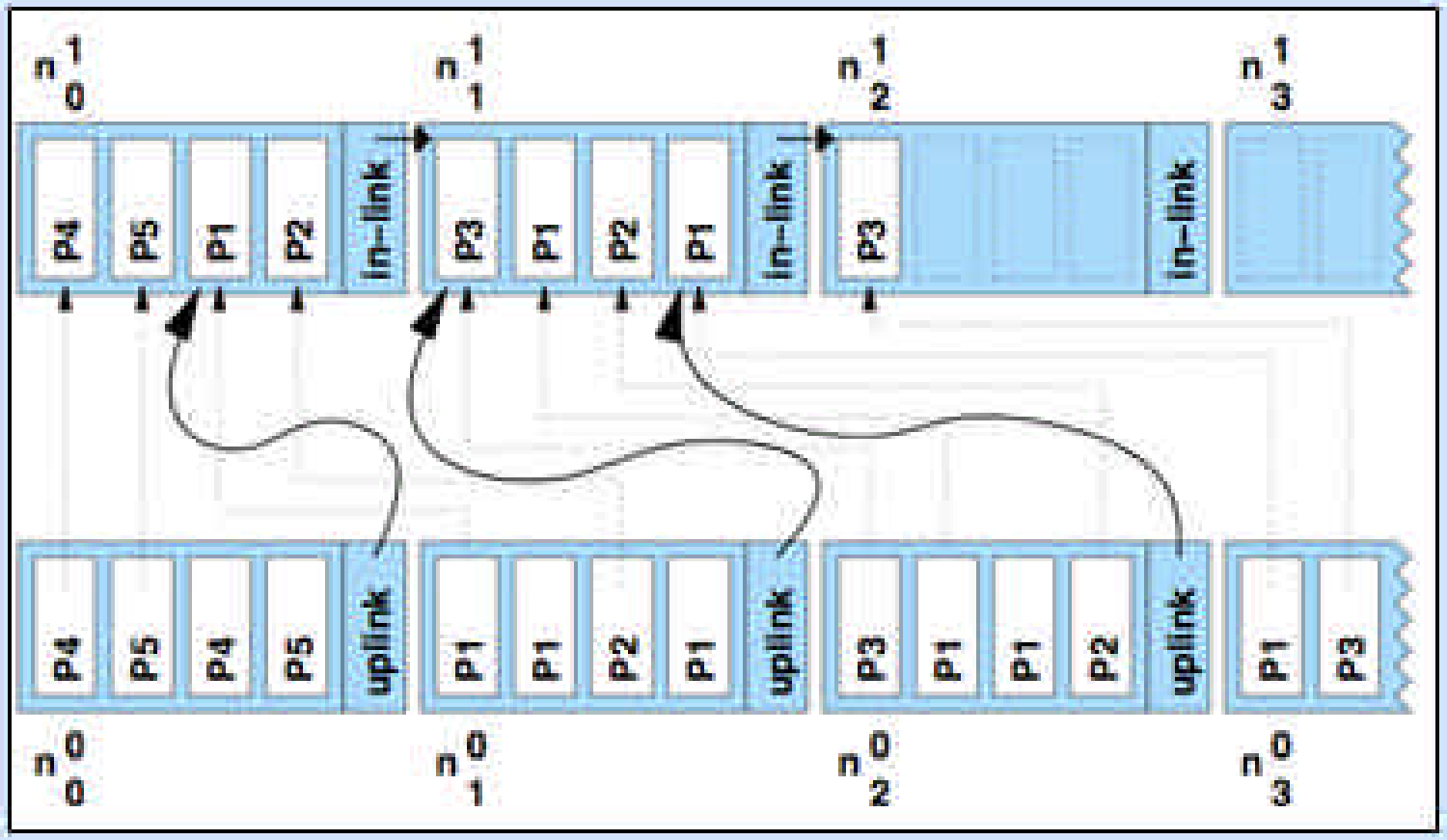


Background mapping writes - cheap

But **foreground** scan to find
a "cold" page is **slow** -
"hot" mappings in the way

Yet, many mappings are "hot" and
many pages in a snapshot are "cold"

Skippy Scan



2-Skippy: high lane drops duplicates

K-Skippy: optimal index for COW snapshots

disk i/o optimal write and lookup operations

Allows to run code in real-time over
multi-year snapshots, as efficient
as short-lived snapshots
even in skewed workloads

As fast as fastest "as of" temporal access methods (TSB,..)
but cheap writes (important for snapshot GC)

Status + paper trail

SNAP, non-disruptive split snapshot system
runs in experimental Thor-2 object storage system (icde05),

Thresher, snapshot storage manager: no copy GC
runs in SNAP (usenix06)

SKIPPY, read-write optimized long-lived index method for COW
runs in SNAP, BDB (icde08)

SNAP/embed, split snapshots – in progress
runs in commercial BDB

Performance results look good

a 5000 feet view:

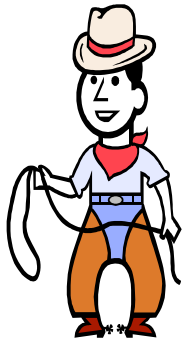
Non-disruptive snapshots mean:

Snapshots should keep up with DB performance
without blocking application access to DB

Cost of WAS-Invariant

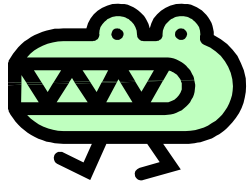
- Prototype implemented in Berkeley DB 4.5.20
- 1.8 GB database; snap-1
- Writing due to WAS (& Skippy) can be hidden
  - Uniform: about 1 to 1 (cache: 9994 dirty pages)
  - Highly skewed (99/1): 35 to 1
  - Trickle to avoid slowing down checkpoint
    - Maintains WAS invariant because trickle before chkpt
- Not end of story for BDB
  - CPU costs: cache COW + Skippy
  - We are analyzing how these costs can be minimized

High-Order Bit:



Long-lived, split snapshots of past states

that run **code** in **real-time**



virtualized in the buffer cache

are **cheaper** than you may think!

Take home

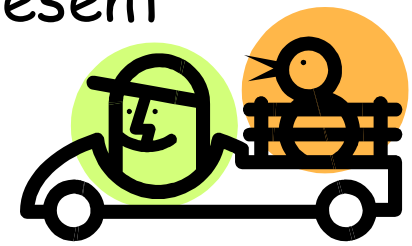
New snapshot approach

new semantics: application specified, persistent, discriminated

new architecture: split COW : Skippy, cache-COW, GC

virtualizes the past to look like the present

in the buffer manager



in my pot:

transactional storage system

SNAP, now Berkeley DB,

Your pot? BITE over our collective memories?

Eager Discriminated Snapshot GC

Free!

the cost:

creating (duplicating) for each rank separate Mapper

is minimal

BITE latency

Traversal T1:

Current DB	Page-based snapshots	Diff-based (d=4) snapshots
17.53s	27.06s	42.11s

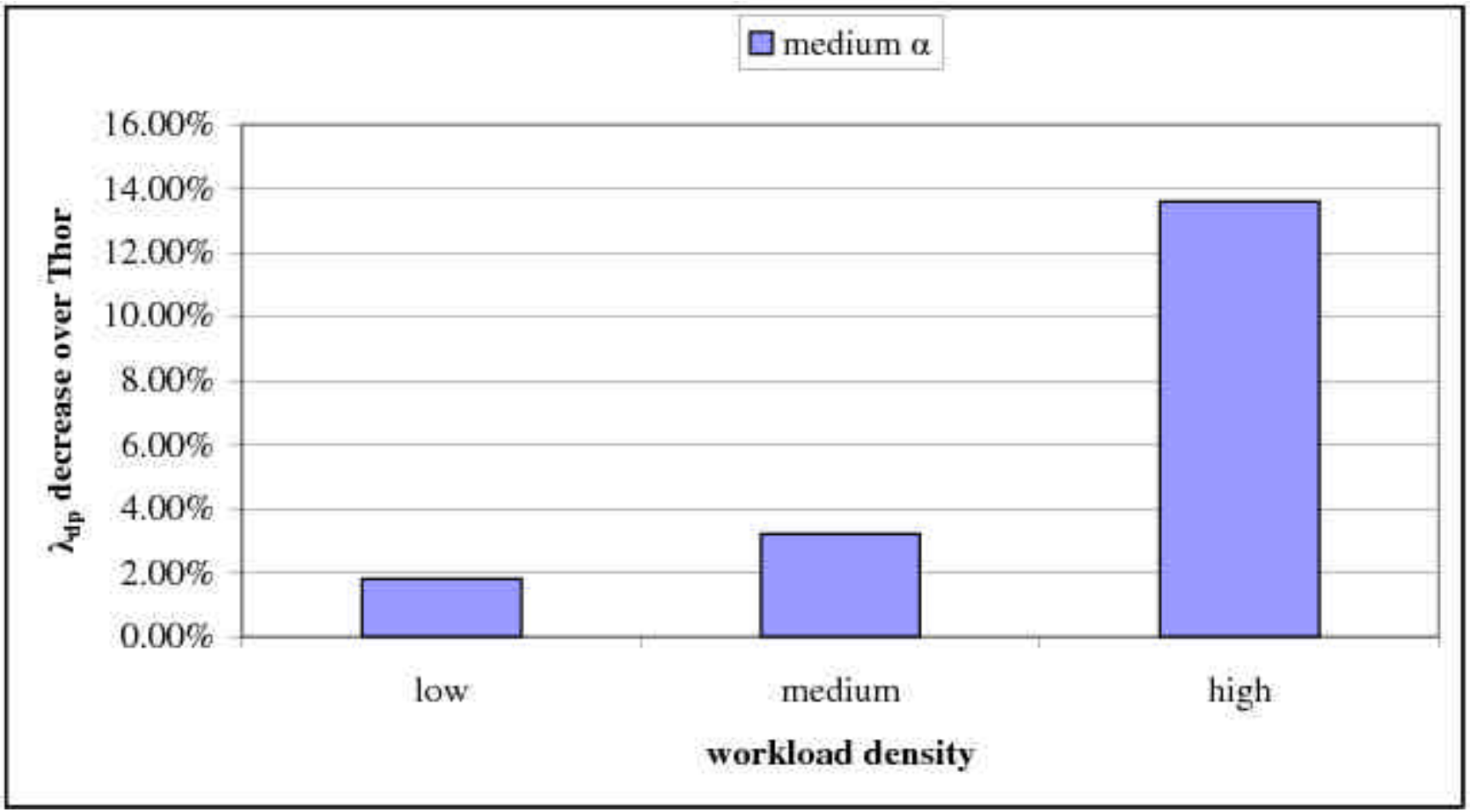
slow, but dual representation accelerates to page-based

Non-disruptiveness for dual = the hardest working rep

how much drop in

rate-of-drain / rate-of-pour ?

Non-disruptiveness: single user drop relative to Thor



Non-disruptiveness, heavy load multi-user: "DB works harder"