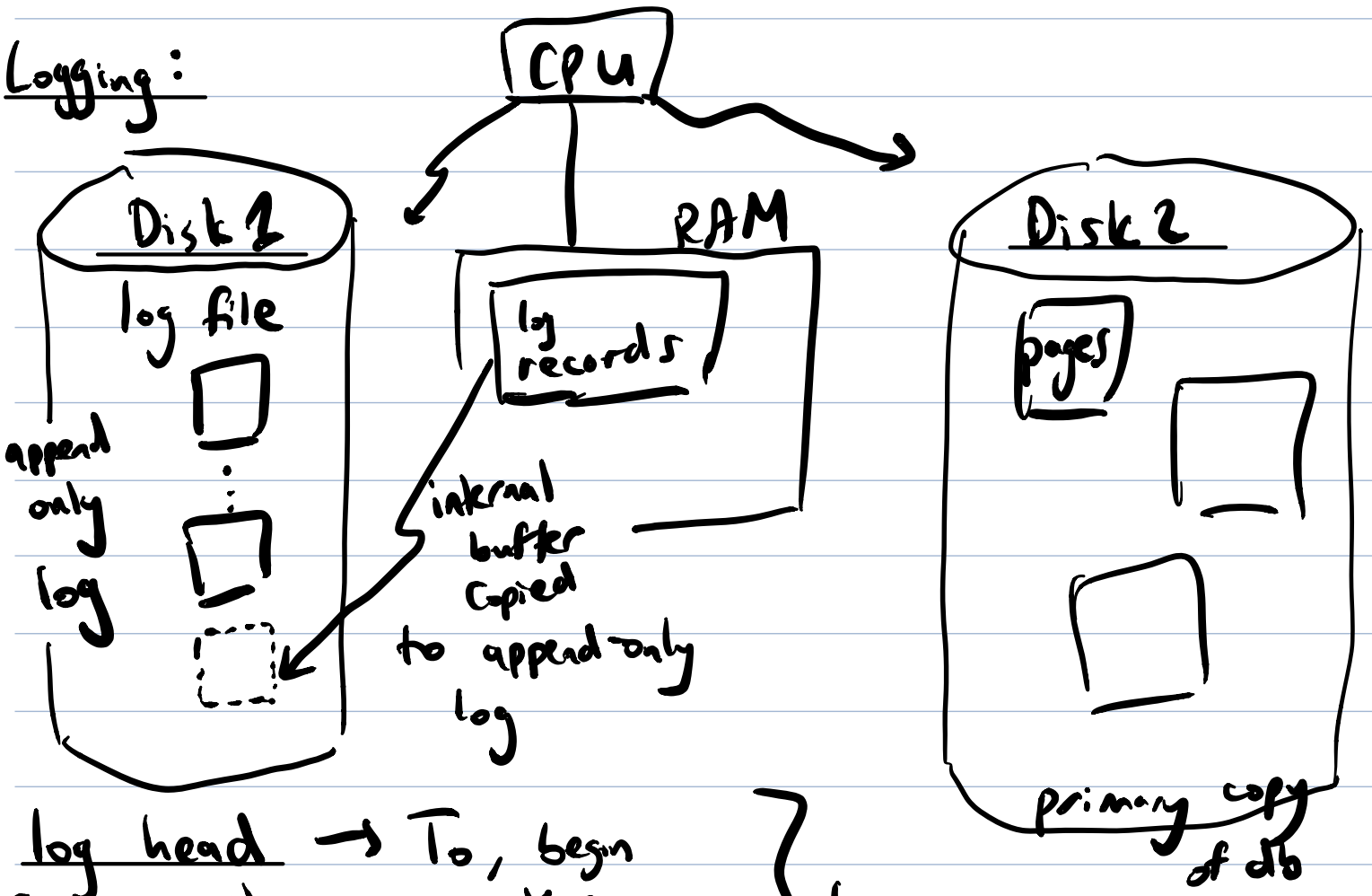
Shadowing: not updating primary copy of database and transaction ⇒ assign root of changes to new version

Logging:



log head → (oldest part)
log tail → (newest part)

T_0, begin
T_0, X, 99, 100
⋮
T_3, commit

} log

recovery algorithm reconciles durable primary copy w/ the log

Need to follow write-ahead logging (WAL):

1) undo rule: log record for update is written to log disk before corresponding

data (page) is written to the main disk (guarantees atomicity)

2) <u>redo rule:</u> all log records for a tx are written to log disk before commit (guarantees durability)

recover from lost RAM:
1) undo everything in WAL (T)
2) redo everything (↓)

may not need to undo everything:
- undo aborted tx
- abort/active tx
- redo committed tx

how do we prevent WAL from growing unbounded?

A: Check pointing
- periodically enter mode: intention to write a checkpoint (not normal processing)
  - → no new transactions
  - → wait for active tx to finish
  - → wait for dirty pages to be written back to primary disk
  - → THEN write a checkpoint log record and flush to log disk

→ now we know at that checkpoint record, db is in a good state and we don't need to look further up

Next big optimization: Checkpointing is a transaction, that records info allowing dbms to infer how far further back to look

Summary:
- ACID guarantees correctness on concurrent db access and data storage
- C/I based on serializability
  → (schedulers, transaction manager)
- D/A based on recovery manager
  ⇒ synchronous writing is too inefficient
  ⇒ replaced by synchronous writes to a LOG and WAL

tuning: adjust physical/conceptual schemas to adjust to changed workloads or requirements on performance characteristics
→ everything changes!
  ⇒ never outside of "maintenance modes"
  → change naturally happens to underlying workload

Good design and tuning requires understanding the database workload.

Definition

A workload description contains:

- most important {queries, updates} and their frequency
- desired performance goal for each query or update

db tuning goal: make a set of applications execute as fast as possible

- optimize for response time?
- optimize for overall throughput? (eg. #tx / second)

as DBA, how can we affect performance?

- make queries faster (data structs, clustering,
- make updates faster (locality) replication)
- minimize congestion due to concurrency

We talked about standard physical design:

- every pk: primary index

- zero or more secondary indices

Storage options: unsorted (heap) file, sorted file, hashed file, dynamic hashing

add indices: B+ trees, R trees, hash tables, ISAM, VSAM

IBM products (IDS) B+ trees w/o splitting/merging

eg. multiple range querying

Multidimensional indexing

eg. Efficiently find all cities in long/lat query latitude

ISAM, VSAM are static indexing approaches:
- these make sense for read-heavy workloads
- like arrays
- column-store is a static indexing scheme with an array at its core
- OLAP workloads

key takeaway: B-trees aren't the end of the story

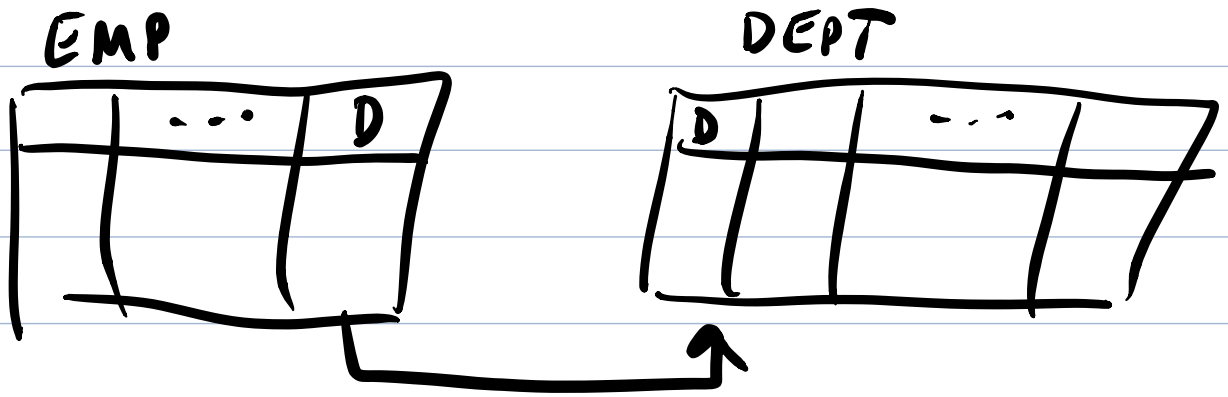
select * from emp where LN = 'Smith'
- if no indices for LN => all blocks of heap file must be scanned

create index LNIndex on emp (LN) [CLUSTER]
①

drop index LNIndex
②

①: - ↓ execution time for lastname selections
- ↑ execution time for insertions
- ↑ or ↓ exec. time for updates or deletions of tuples from emp
- ↑ space to represent emp

Co-clustering:



- interleave tuples from two relations in the same file
→ useful for storing hierarchical data (1 : N relationships)
- effects on performance
1) speed up joins, particularly FK joins
2) sequential scans of either relations becomes

slower

join indexes: allow replacing joins by index lookups

materialized views: allow replace queries by index lookups

multi-attribute indices: complex search/join cond's

Problem 1: optimizer needs to know if/where to use such indices/views

Problem 2: balance cost of rematerialization and savings for queries.