

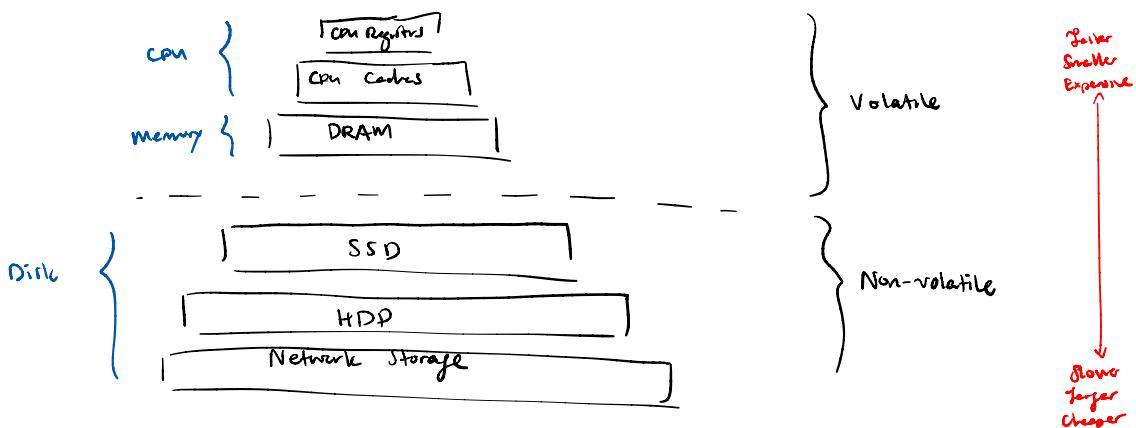
Query Planning
Operator Execution
Access Methods
Buffer Pool Manager
Dirk Manager

Lecture 1: Data Storage

Dirk-Based Architecture

DBMS assumes primary storage of database is non-volatile storage
 if lose power,
 data is wiped out

Storage Hierarchy



Access Times

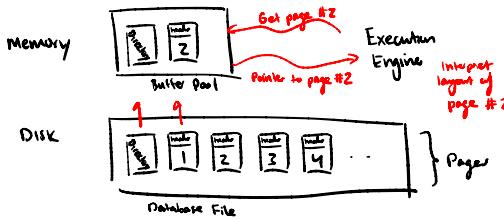
1 ns	L1 Cache Ref	← 1 sec
4 ns	L2 Cache Ref	← 4 sec
100 ns	DRAM	← 100 sec
16,000 ns	SSD	← 4.4 hrs
2,000,000 ns	HDD	← 3.3 weeks
~50,000,000 ns	Network Storage	← 1.3 years
1,000,000,000 ns	Tape Archives	← 31.3 years

Segmented vs Random Access

- Random access on non-volatile storage is almost always slower than sequential access
- DBMS want to maximize sequential access
 - Algorithms try to reduce # of writes to random pages so data is stored in contiguous blocks
 - Attaching multiple pages at same time is called an extent

Design Goals

- Allow DBMS manages databases that exceed the amount of memory available.
- Avoid large stalls / performance degradation
 - manage reading/writing to disk
 - maximize sequential access and random access.



Why not use the OS?

Proposed : use mmap (memory mapping) to store contents of a file into address space of a program.

- allow multiple threads to access mmap files to hide page fault stalls.

Problems :

1) Transaction Safety

- Os can flush dirty pages at any time

2) I/O Stalls

- DBMS doesn't know which pages are in memory.
- Os will stall on a page fault.

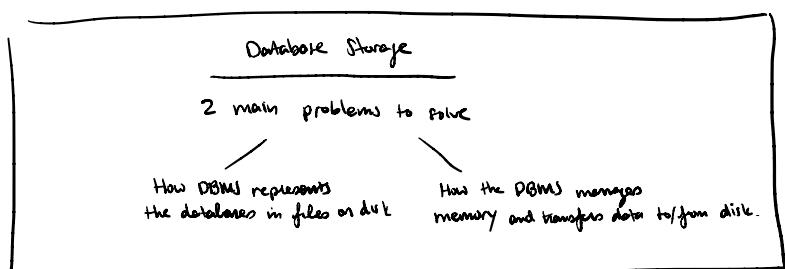
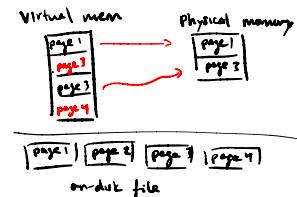
3) Error Handling.

- difficult to invalidate pages. (invalid memory address access)
- any access can cause SIGBUS that DBMS must handle.

4) Performance Issues

- Os data structure issues
 - TLB shutdowns
- when a processor restricts access to a page of shared memory,
every process must flush their
Translation Lookaside Buffer

=> The Os is not your friend



File Storage

DBMS stores a database as one or more files on disk in a proprietary format

- Os doesn't know anything about contents of these files.

The Storage manager is responsible for maintaining disk files

- some do their own scheduling for reads and writes to improve spatial/temporal locality of pages.
- organizes files as collection of **pages** : tracks < data read/write to page < available space.

Database Pages

- fixed size block of data } can contain tuples, metadata, indexes, log records
most systems do not mix page types
some systems require pages to be self-contained
- each page is given a unique identifier → DBM! uses an indirection layer to map page IDs to physical locations
- there are 3 notions of "pages" in a DBM!:
 1. Hardware page ~4 kB
 2. OS page ~4 kB
 3. Database page 512B-16kB

largest block of data
that the storage device
can guarantee fail-safe writes.

Page Storage Architecture

Different DBMS's manage pages in files on disk in different ways:

- 1) Heap file organization
- 2) Tree file organization
- 3) Sequential/sorted file organization
- 4) Hashing file organization

* At this level, don't need to know anything about what is inside of the pages *

insert a new tuple:

- 1) check page directory to find a page w/ a free slot
- 2) Retrieve page from disk (if not in memory)
- 3) Check slot array to find empty space in page that will fit

update existing tuple (using its record-id)

- 1) check page directory to find location of page
- 2) Retrieve page from disk (if not in memory)
- 3) Find offset in page using slot array
- 4) Overwrite existing data (if new data fits)

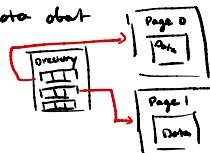
Heap file

- unindexed collection of pages w/ tuples stored in a random order.
- Create, get, write, delete Page
- Must support iterating over all pages.
- Easier to find pages if 1: single file.
- Need meta-data to keep track of what pages exist in multiple files and which ones have free space.

Page directory

- DBMS maintains special pages that track location of data pages in the database files
- need directory pages to be in sync w/ the data pages.

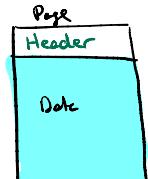
- the directory also records meta-data about available space:
 - 1) # of free slots per page
 - 2) list of free/empty pages



Page Header

Every page contains a header of metadata about its contents

- 1) Page size
- 2) Checksum
- 3) DBM! version
- 4) Transaction visibility
- 5) Compression information



- some systems (eg Oracle) require pages to be self-contained

Page Layout

For any page storage architecture, need to decide how to organize the data inside the page.

- 2 approaches: ① tuple-oriented ② log-oriented

Page
of tuples
tuple #1
tuple #2
tuple #3

Tuple Storage

- Strawman idea: keep track of # of tuples and simply append new tuple to end.
 - Q: What if we delete a tuple?
 - What if we have a nullable length attribute?

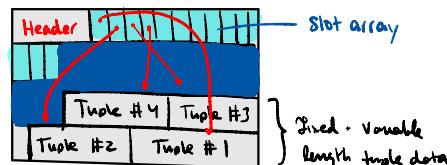
Record ID

- DBMS needs way to keep track of individual tuples: solution: assign a unique record identifier to each tuple.

Slotted Pages

- Most common layout scheme is slotted pages, where slot array maps "slots" to tuples' starting position offsets.
- page header keeps track of
 - 1) # of used slots
 - 2) offset of starting location of last slot used.

- solution:
1. (most common): page_id + offset/slot
 2. can also contain file location information

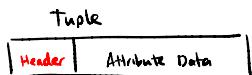


Tuple Layout

a tuple is essentially just a sequence of bytes \rightarrow It's the job of the DBMS to interpret those bytes into attribute types + values.

Tuple header

- Each tuple is prefixed w/ a header that contains metadata about it, e.g.
- We do NOT need to store metadata about schema.
- 1) Visibility info (concurrency control)
- 2) Bitmaps for Null values

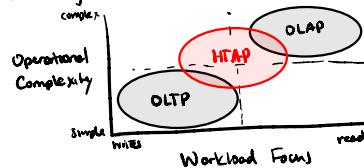


Data representation:

- 1) Integer / BigInt / SmallInt / TinyInt \rightarrow C/C++ representation
- 2) Float / Real v. Numeric/Decimal \rightarrow IEEE754 / Fixed-point decimal
- 3) Varchar / VarBinary / Text / Blob \rightarrow (Header w/ length) + (data bytes)
- 4) Time / Date / Timestamp \rightarrow don't worry about collating/sorting
 \rightarrow 32/64-bit integers of (micro)seconds since Unix Epoch

Database Workloads

- 1) On-line Transaction Processing (OLTP) : fast operations that only read/update a small amount of data at a time
think: processes data/line transactions
- 2) On-line Analytical Processing (OLAP) : complex queries that read a lot of data to compute aggregates
think: analyzes aggregate data
- 3) Hybrid Transaction + Analytical Processing : OLTP + OLAP on the same database



OLTP

- Simple Queries that read/update a small amount of data that is related to a single entity in the database

OLAP

- Complex Queries that read large portions of the database spanning multiple entities
- execute these workloads on data collected from OLTP application(s)

```
Select P.+, R.+
From pages AS P
Inner Join reviews AS R
On P.pageID = R.pageID
Where P.pageID = ?
```

```
UPDATE useracct
SET lastLogin = NOW(),
hostname = ?
Where userID = ?
```

```
Insert into reviews
values(?, ?, ?, ?, ?)
```

Data Storage Models

- DBMS can store tuples in different ways that are better for either OLTP or OLAP workloads
 - ↳ we've been assuming row storage (or row storage) so far



N-ary Storage Model (NSM)

- DBMS stores all attributes for a single tuple contiguously in a page.
- Ideal for insert-heavy workloads or OLTP workloads where queries tend to operate only on an individual entity.



- Advantages :
- ① Fast inserts, updates, & deletes
 - ② Good for queries that need entire tuple (e.g. select *)

- Disadvantages :
- ① Not good for scanning large portions of the table and/or subset of the attributes. (e.g. select count(single attribute))

Column Store :

Decomposition Storage Model (DSM)

- DBMS stores values of a single attribute for all tuples contiguously in a page
- Ideal for OLAP workloads where read-only queries perform large scans over a subset of table's attributes.



- Advantages :
- ① Reduces amount of wasted I/O because DBMS only reads data it needs
 - ② Better query processing & data compression

- Disadvantages :
- ① Slow for point queries, inserts, updates, & deletes because of tuple splitting/stitching

Observation

- I/O is main bottleneck if DBMS fetched data from disk during query execution
- DBMS can compress pages to increase utility of data moved per I/O operation
- Key trade-off is speed v. compression ratio
 - 1) Compressing the database reduces DRAM requirements
 - 2) It may decrease CPU costs during query execution.

lecture #2

Database Storage

Problem #1 : How DBMS represents the database in files on disk

Problem #2 : How DBMS manages memory and transfers data to/from disk

↓
today!

Storage Control:

- where to write pages on disk
- The goal is to keep pages that are often accessed together physically close on disk.

Temporal Control

- when to read pages into memory, and when to write them out to disk.
- The goal is to minimize the # of stalls from having to perform disk I/O

Buffer Pool Metadata

- The page table keeps track of pages that are currently in memory
- Also maintains additional meta-data per page:
 - 1) dirty flag
 - 2) pin/reclaim counter

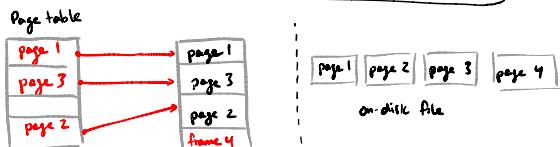
Buffer Replacement Policies

- when DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.
- Considerations:
 - 1) Correctness
 - 2) Accuracy
 - 3) Speed
 - 4) Meta-data overhead.

Naive Policies:

① Least Recently Used (LRU)

- Maintain a single timestamp for when each page was last accessed
- When the DBMS needs to evict a page, select the one w/ the oldest timestamp
 - keep the pages in sorted order to reduce seek time on eviction



Page table vs. Page Directory

mapping from page IDs to a copy of the page in buffer pool frames

- in-memory data structure that doesn't need to be stored on disk

mapping from page IDs to page locations in database files

- all changes must be recorded on disk to allow DBMS to find them on restart

② Clock

- Approximation of LRU that does not need a separate timestamp per page.

- each page has a reference bit
- when a page is accessed, set to 1.

- Organize the pages in a circular buffer with a "clock hand".

- Upon sweeping, check if a page's bit is set to 1.
 - If yes, set to 0. If no, evict.



Problems

- LRU + Clock replacement policies are susceptible to sequential flooding.

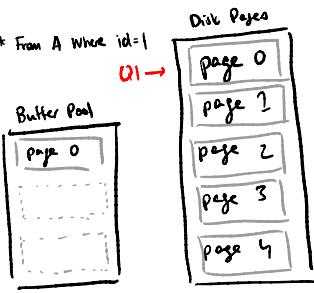
- A query performs a sequential scan that reads every page.
- This pollutes the buffer pool w/ pages that are read once, then never again.

- In some workloads, the most-recently used (MRU) page is the most un-needed page.

Sequential Scanning

Q1 Select * From A Where id=1

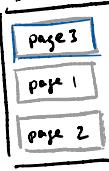
Q1 →



Q2 Select Avg(Val) from A

from A

Buffer Pool

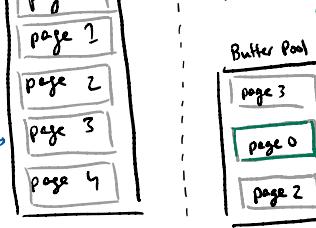


Q3 Select * From A where id=1

where id=1

Q3 →

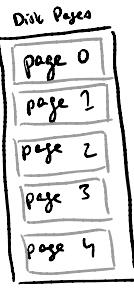
Buffer Pool



Q4 Select * From A

from A

Q4 →



Better Policies :

① JRU-K

- Track the history of last K references to each page as timestamp(s) and compute the interval between subsequent accesses.
- The DBMS then uses this history to estimate the next time that page is going to be accessed.

②

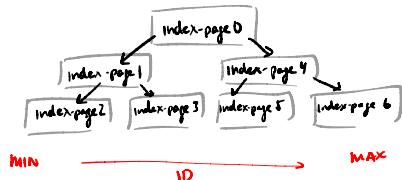
Localization

- DBMS chooses which pages to evict on a per-query basis. This minimizes the pollution of the pollution of the buffer pool from each query
 - keep track of the pages that a query has accessed
- EX: PostgreSQL maintains a small ring buffer that is private to the query.

③

Priority Hints

- DBMS knows about the content of each page during query execution.
- It can provide hints to the buffer pool about whether or not a page is important



Buffer Pool Optimizations

- 1) Dirty Page Eviction
- 2) Background Writing
- 3) Avoiding the OS
- 4) Multiple Buffer Pools
- 5) Pre-fetching
- 6) Scan-Sharing
- 7) Buffer Pool Bypass

Dirty Page Eviction

- how to handle dirty pages?
- Fast Part: If a page in the buffer pool is not dirty, the DBMS can simply 'drop' it.
- Slow Part: If a page is dirty, then the DBMS must write it back to disk to ensure its changes are persisted
- Need to consider trade-off between fast evictions & writing dirty pages that will not be read again

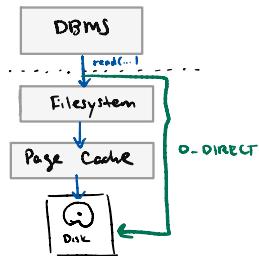
- Background Writing

- DBMS can periodically walk through the page table and write dirty pages to disk.
- When a dirty page is safely written, the DBMS can either evict page or insert the L. Need to be careful that the system doesn't write dirty pages before their dirty flag-log records are written.

- OS Page Cache

The OS
is
NOT your
friend!!

- Most disk operations go through the OS API. (☞)
Unless the DBMS tells it not to, the OS maintains its own filesystem cache (page/buffer cache)
- Most DBMS's use direct I/O (O_DIRECT) to bypass the OS's cache.
Redundant copies of pages
Different eviction policies
Loss of control over file I/O



- Multiple Buffer Pools

- DBMS doesn't always have a single buffer pool for the entire system.
 - Multiple buffer pool instances
 - Per-database buffer pool.
 - Per-page type buffer pool.
- Partitioning memory access across multiple pools helps reduce latch contention and improve locality.
- Approaches:

1) Object ID

- Embed an object identifier in record IDs and then maintain a mapping from objects to specific buffer pools.

(ObjectID, PageID, SlotNum)



2) Hashing

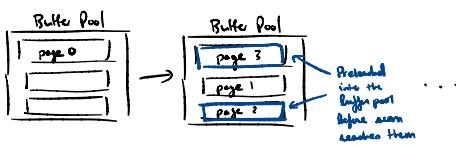
- Hash the page ID to select which buffer pool to access.

Hash(123) % n

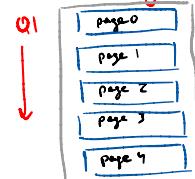
- Prefetching

- DBMS can also prefetch pages based on a query plan:
 - 1) Sequential Scans
 - 2) Index Scans

Sequential Scan

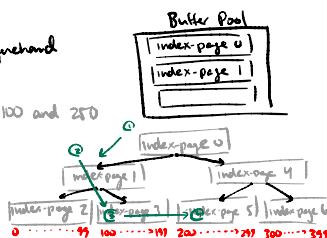


Disk Pages



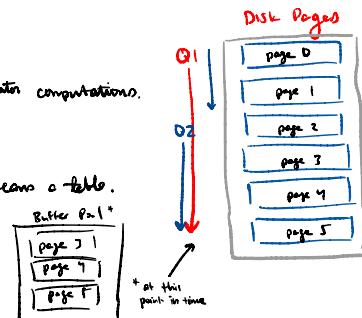
Index Scan

- need to know access patterns beforehand
- ex: Q2: Select * from A where val between 100 and 250



Scan Sharing

- Queries can reuse data retrieved from storage or operator computations.
 - called synchronized scans
 - different from result caching
- Allows multiple queries to attach to a single cursor that scans a table.
 - queries do not have to be the same
 - can share intermediate results
- "second query can attach to current scanning pointer instead of restarting scan"



Buffer Pool Bypass

- aka light scans
- segmented scan operator will not store fetched pages in the buffer pool to avoid overhead.
 - memory is local to scanning query
 - works well if operator needs to read a large sequence of pages that are contiguous on disk
 - can be used for temporary data (sorting, joins, etc)

Other Memory Pools

- DBMS needs memory for things other than just tables + indexes
- other memory pools are not always backed by disk. Implementation-dependent.
- Ex:
 - Sorting + Join Buffers
 - Query Caches
 - Maintenance Buffers
 - Log Buffers
 - Dictionary Caches

TL;DR: The DBMS can almost always manage memory better than the OS

- leverage the semantics of the query plan to make better decisions:
 - 1) Evictions
 - 2) Allocations
 - 3) Pre-fetching