Georgia Tech

Lecture 7: Buffer Management

CREATING THE NEXT®

Administrivia

- **EvaDB Assignments**
  - EvaDB Assignment 1: checkpoint on Sep 26, final submission on Oct 12
  - EvaDB Assignment 2: checkpoint on Oct 31, final submission on Nov 21
- 5-min presentations by students with the top-10 projects in class

Collaboration Guidelines

- Student collaboration:
  - Explain your code to your class-mate to see if they know why it doesn't work.
  - Help your class-mate debug if they've run into a wall.

Today's Agenda

- Recap
- BuzzDB
- Buffer Pool Manager
- Buffer Pool Optimizations
- Buffer Replacement Policies

Georgia Tech

Recap

Data Representation

- INTEGER/BIGINT/SMALLINT/TINYINT
  - C/C++ Representation
- FLOAT/REAL vs. NUMERIC/DECIMAL
  - IEEE-754 Standard / Fixed-point Decimals
- VARCHAR/VARBINARY/TEXT/BLOB
  - Header with length, followed by data bytes.
- TIME/DATE/TIMESTAMP
  - 32/64-bit integer of (micro)seconds since Unix epoch

Workload Characterization

- On-Line Transaction Processing (OLTP)
  - Fast operations that only read/update a small amount of data each time.
  - OLTP Data Silos
- On-Line Analytical Processing (OLAP)
  - Complex queries that read a lot of data to compute aggregates.
  - OLAP Data Warehouse
- Hybrid Transaction + Analytical Processing
  - OLTP + OLAP together on the same database instance

BuzzDB

BuzzDB

- BuzzDB – version 8
- BuzzDB – version 9
- BuzzDB – version 10
- BuzzDB – version 11
- BuzzDB – version 12

C++: Serializing and Deserializing

```
std::string filename = "page.dat";

// Serialize to disk
db.page.write(filename);

// Deserialize from disk
Page page2;
page2.read(filename);
```

Georgia Tech

C++: Serializing a Database Page

```
// Write this page to a file.
void write(const std::string& filename) const {
  std::ofstream out(filename);
  // First write the number of tuples.
  size_t numTuples = tuples.size();
  out.write(reinterpret_cast<const char*>(&numTuples), sizeof(numTuples));

  // Then write each tuple.
  for (const auto& tuple : tuples) {
      // Write the number of fields in the tuple.
      size_t numFields = tuple->fields.size();
      out.write(reinterpret_cast<const char*>(&numFields), sizeof(numFields));

      // Then write each field.
      ...
  }

  out.close();
```

C++: Serializing a Database Page

```
// Write this page to a file.
void write(const std::string& filename) const {
    // Then write each field.
    for (const auto& field : tuple->fields) {
        // Write the type of the field.
        out.write(reinterpret_cast<const char*>(&field->type), sizeof(field->type));
        // Write the length of the field.
        out.write(reinterpret_cast<const char*>(&field->data_length), sizeof(field->data_leng
        // Then write the field data.
        out.write(field->data.get(), field->data_length);
    }
  }

  out.close();
}
```

C++: Deserializing a Database Page

```
// Read this page from a file.
void read(const std::string& filename) {
    std::ifstream in(filename);

    // First read the number of tuples.
    size_t numTuples;
    in.read(reinterpret_cast<char*>(&numTuples), sizeof(numTuples));

    std::cout << "Num Tuples: " << numTuples << "\n";

    // Then read each tuple.
    for (size_t i = 0; i < numTuples; ++i) {
        auto tuple = std::make_unique<Tuple>();
        ..
    }
}
```

C++: Deserializing a Database Page

```
// Read this page from a file.
void read(const std::string& filename) {

    // Read the number of fields in the tuple.
    size_t numFields;
    in.read(reinterpret_cast<char*>(&numFields), sizeof(numFields));

    // Then read each field.
    for (size_t j = 0; j < numFields; ++j) {
        // Read the type of the field.
        FieldType type;
        in.read(reinterpret_cast<char*>(&type), sizeof(type));
        // Read the length of the field.
        size_t data_length;
        in.read(reinterpret_cast<char*>(&data_length), sizeof(data_length));
        // Then read the field data.
        std::unique_ptr<char[]> data(new char[data_length]);
```

C++: Deserializing a Tuple

```
// Read this page from a file.
void read(const std::string& filename) {

  // Read the number of fields in the tuple.
  size_t numFields;
  in.read(reinterpret_cast<char*>(&numFields), sizeof(numFields));

  // Then read each field.
  for (size_t j = 0; j < numFields; ++j) {
    // Read the type of the field.
    FieldType type;
    in.read(reinterpret_cast<char*>(&type), sizeof(type));
    // Read the length of the field.
    size_t data_length;
    in.read(reinterpret_cast<char*>(&data_length), sizeof(data_length));
    // Then read the field data.
    std::unique_ptr<char[]> data(new char[data_length]);
```

C++: Deserializing a Tuple

```
// Read this page from a file.
void read(const std::string& filename) {

  // Add the field to the tuple.
  switch(type){
      case INT:
      {
          int val = *reinterpret_cast<int*>(data.get());
          auto field = std::make_unique<Field>(val);
          tuple->addField(std::move(field));
          break;
      }
      case STRING:
      {
          char* val = reinterpret_cast<char*>(data.get());
          auto field = std::make_unique<Field>(std::string(val, data_length));
          tuple->addField(std::move(field));
          break;
```

C++: Persiting Changes to Disk

```
// Serialize to disk
db.page.write(filename);

// Deserialize from disk
auto loadedPage = Page::deserialize(filename);

// PROBLEM: Deletion only in memory, not on disk
loadedPage->deleteTuple(0);

// Deserialize again from disk -- page unchanged
auto loadedPage2 = Page::deserialize(filename);
```

C++: Persiting Changes to Disk

```
// Serialize to disk
db.page.write(filename);

// Deserialize from disk
auto loadedPage = SlottedPage::deserialize(filename);

loadedPage->print();

std::cout << "Deleting slots 0 and 7 \n";
loadedPage->deleteTuple(0);
loadedPage->deleteTuple(7);

loadedPage->write(filename);

// Deserialize again from disk -- page is updated this time
auto loadedPage2 = SlottedPage::deserialize(filename);

loadedPage2->print();
```

C++: Extending Database File Automatically

```
bool status = try_to_insert(key, value);

// Try again after extending the database file
if(status == false){
    extendDatabaseFile();
    ..
}
```

C++: Extending Database File Automatically

```
void extendDatabaseFile() {
    //std::cout << "Extending database file \n";

    // Create a buffer with PAGE_SIZE bytes
    auto empty_slotted_page = std::make_unique<SlottedPage>();

    // Write the buffer to the file, extending it
    file.seekp(0, std::ios::end);
    file.write(empty_slotted_page->page_data.get(), PAGE_SIZE);
    file.flush();

    // Update number of pages
    num_pages += 1;

    // Load page into memory
    auto page_itr = num_pages - 1;
    auto loadedPage = SlottedPage::deserialize(file, page_itr);
```

Database Storage

- Problem 1: How the DBMS represents the database in files on disk.
- Problem 2: How the DBMS manages its memory and moves data back-and-forth from disk.

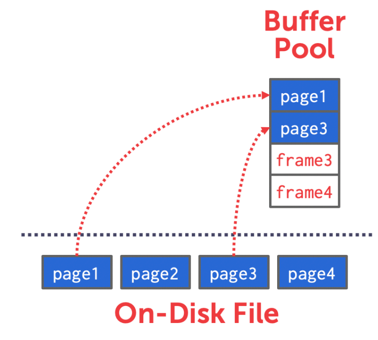
Buffer Pool Manager

Database Storage

- **Spatial** Control:
  - Where to write pages on disk.
  - The goal is to keep pages that are used together often as physically close together as possible on disk.
- **Temporal** Control:
  - When to read pages into memory, and when to write them to disk.
  - The goal is minimize the number of stalls from having to read data from disk.

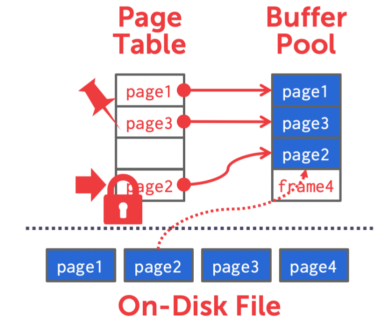
Buffer Pool Organization

- Memory region organized as an array of fixed-size pages.
- An array entry is called a **frame**.
- When the DBMS requests a page, an exact copy of the data on disk is placed into one of these frames.



Buffer Pool Meta-Data

- The **page table** keeps track of pages that are currently in memory.
- Also maintains additional meta-data per page:
  - **Dirty Flag**
  - **Pin/Reference Counter**



Locks vs. Latches

- **Locks:**
  - Protects the database's **logical contents** from other transactions.
  - Held for **transaction** duration.
  - Need to be able to rollback changes.
- **Latches:**
  - Protects the critical parts of the DBMS's internal data structure from other threads.
  - Held for **operation** duration.
  - Do not need to be able to rollback changes.
  - C++: std::mutex

Page Table vs. Page Directory

- The **page directory** is the mapping from page ids to page locations in the database files.
  - All changes must be recorded on disk to allow the DBMS to find on restart.
- The **page table** is the mapping from page ids to a copy of the page in buffer pool frames.
  - This is an in-memory data structure that does **not** need to be stored on disk.

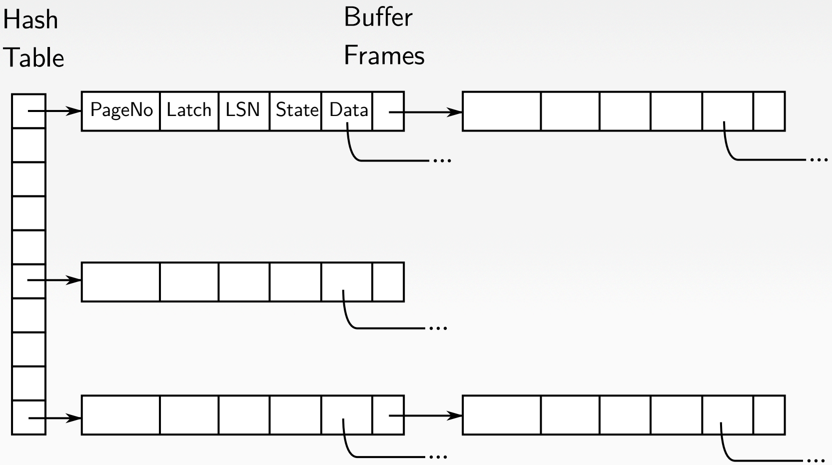
Buffer Manager Interface

Basic interface:

1. FIX (uint64_t page_id, bool is_shared)
2. UNFIX (uint64_t page_id, bool is_dirty)

Pages can only be accessed (or modified) when they are **fixed** in the buffer pool.

Buffer Manager Implementation



The buffer manager itself is protected by one or more **latches**.

Buffer Frame

Maintains the state of a certain page within the buffer pool.

pageNo	the page number
latch	a read/writer latch to protect the page (note: must **not** block access to unrelated pages!)
LSN	LSN of the last change to the page, for recovery (buffer manager must force the log record containing the changes to disk before writi
state	clean/dirty/newly created etc.
data	the actual data contained on the page

(will usually contain extra information for buffer replacement)

Usually kept in a hash table.

Buffer Pool Optimizations

Buffer Pool Optimizations

- Multiple Buffer Pools
- Pre-Fetching
- Scan Sharing
- Buffer Pool Bypass
- Background Writing
- Other Pools

Multiple Buffer Pools

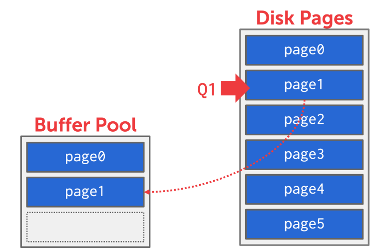
- The DBMS does not always have a single buffer pool for the entire system.
  - Multiple buffer pool instances
  - Per-database buffer pool
  - Per-page type buffer pool
- Helps reduce **latch contention** and improve **locality**. Why?

Multiple Buffer Pools

- Approach 1: Object Id
  - Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.
  - Example: <object_id, page_id, slot_number>
  - ObjectId $\longrightarrow$ Buffer Pool Number
- Approach 2: Hashing
  - Hash the page id to select whichbuffer pool to access.
  - Example: HASH(page_id) % (Number of Buffer Pools)

Pre-Fetching: Sequential Scans

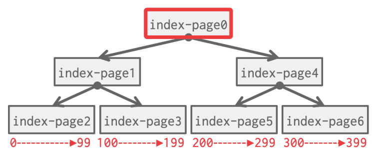
- The DBMS can prefetch pages based on a query plan.
  - Sequential Scans



Pre-Fetching: Index Scans

- The DBMS can prefetch pages based on a query plan.
  - Index Scans

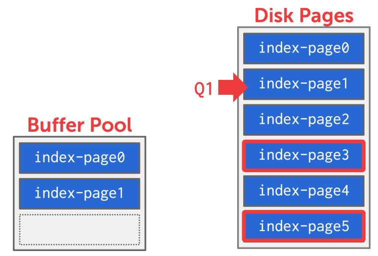
```
SELECT *
FROM A
WHERE val BETWEEN 100 AND 250;
```



Pre-Fetching: Index Scans

- The DBMS can prefetch pages based on a query plan.
  - Index Scans

```
SELECT *
FROM A
WHERE val BETWEEN 100 AND 250;
```



Scan Sharing

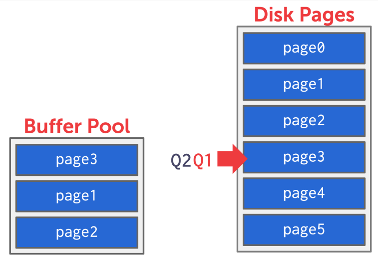
- Queries can **reuse data** retrieved from storage or operator computations.
  - This is different from **result caching**.
- Allow multiple queries to attach to a single cursor that scans a table.
  - Queries do not have to be exactly the same.
  - Can also share intermediate results.

Scan Sharing

- If a query starts a scan and if there one already doing this, then the DBMS will attach to the second query's cursor.
  - The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure.
- Fully supported in IBM DB2 and MSSQL.
- Oracle only supports cursor sharing for identical queries.

Scan Sharing

Q1: SELECT SUM(val) FROM A;
Q2: SELECT AVG(val) FROM A;
Q3: SELECT AVG(val) FROM A LIMIT 100;



Buffer Pool Bypass

- The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.
  - Memory is local to running query.
  - Works well if operator needs to read a large sequence of pages that are contiguous on disk. What is it called?
  - Can also be used for temporary data (sorting, joins).
- Called **light scans** in Informix.

OS Page Cache

- Most disk operations go through the OS API.
- Unless you tell it not to, the OS maintains its own filesystem cache.
- Most DBMSs use direct I/O (O_DIRECT) to bypass the OS's cache.
  - Redundant copies of pages.
  - Different eviction policies.

Background Writing

- The 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 the page or just unset the dirty flag.
- Need to be careful that we don't write dirty pages before their **log records** have been written to disk.

Other Memory Pools

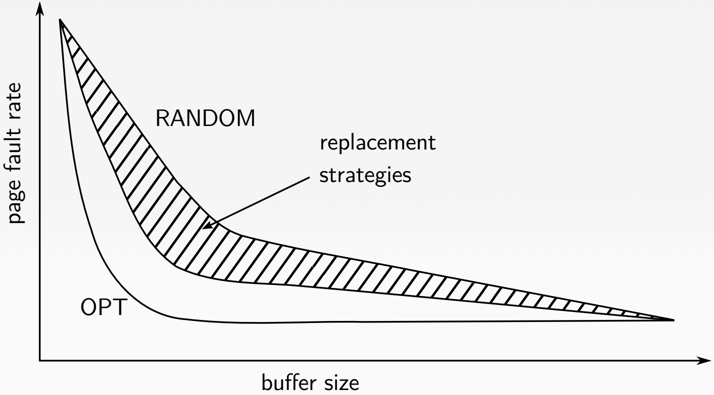
- The DBMS needs memory for things other than just tuples and indexes.
- These other memory pools may not always backed by disk. Depends on implementation.
    - Sorting + Join Buffers
    - Query Caches
    - Maintenance Buffers
    - Log Buffers
    - Dictionary Caches

Buffer Replacement Policies

Buffer Replacement

- When the 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.
- Goals:
  - Correctness
  - Accuracy
  - Speed
  - Meta-data overhead
- Page State:
  - clean pages can be simply discarded
  - dirty pages have to be written back first

Buffer Replacement Policies



Buffer Replacement Policy - FIFO

First In - First Out (FIFO)

- Simple replacement strategy
- Buffer frames are kept in a linked list (queue)
- Pages inserted at the end, removed from the head
- Keeps the pages that were most recently added to the buffer pool

Does **not** retain frequently-used pages

Buffer Replacement Policy - LFU

Least Frequently Used (LFU)

- Remember the number of accesses per page
- Infrequently used pages are removed first
- Maintain a priority queue of pages

Sounds plausible, but too expensive in practice.

Buffer Replacement Policy - LRU

Least-Recently Used (LRU)

- Maintain a timestamp of when each page was last accessed.
- When the DBMS needs to evict a page, select the one with the **oldest access timestamp**.
  - Keep the pages in sorted order to reduce the search time on eviction.
  - Buffer frames are kept in a double-linked list
  - Remove from the head
  - When a frame is unfixed, move it to the end of the list
  - “Hot” pages are retained in the buffer

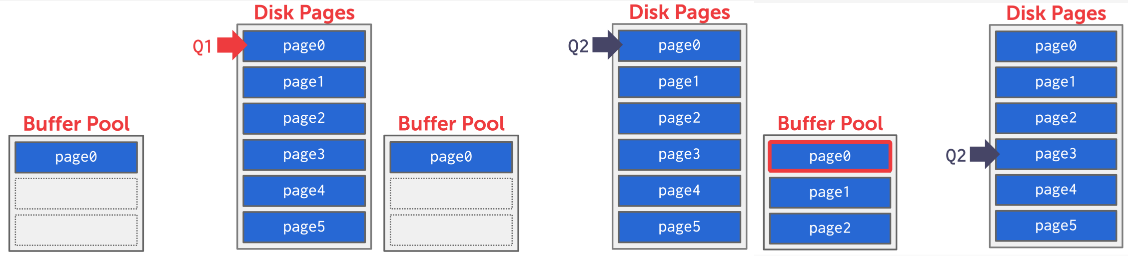
A very popular policy.

Problems

- LRU is susceptible to **sequential flooding**.
  - A query performs a sequential scan that reads every page.
  - This **pollutes** the buffer pool with pages that are read once and then never again.
- The most recently used page is actually the most unneeded page.

```
Q1: SELECT * FROM A WHERE id = 1;
Q2: SELECT AVG(val) FROM A; -- Sequential Scan
```

Sequential Flooding



Better Policies - LRU-K

- Track the history of last K references to each page as timestamps 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.
- Degenerates to classic LRU when K = 1
- **Scan resistant** policy

Better Policies - 2Q

Maintain two queues (FIFO and LRU)

- Some pages are accessed only once (*e.g.*, sequential scan)
- Some pages are hot and accessed frequently
- Maintain separate lists for those pages
- **Scan resistant** policy

1. Maintain all pages in FIFO queue
2. When a page that is currently in FIFO is referenced again, upgrade it to the LRU queue
3. Prefer evicting pages from FIFO queue

Hot pages are in LRU, read-once pages in FIFO.

Better Policies - Priority Hints

- The DBMS knows what the context of each page during query execution.
- It can provide hints to the buffer pool on whether a page is important or not.
- 2Q tries to recognize read-once pages
- But the DBMS knows this already!
- It could therefore give **hints** when unfixing
- Example: **will-need** or **will-not-need** hint will determine which queue the page is added to

Conclusion

- The DBMS can manage that sweet, sweet memory better than the OS.
- Leverage the semantics about the query plan to make better decisions:
  - Evictions
  - Allocations
  - Pre-fetching

Next Class

- Buffer Management Implementation