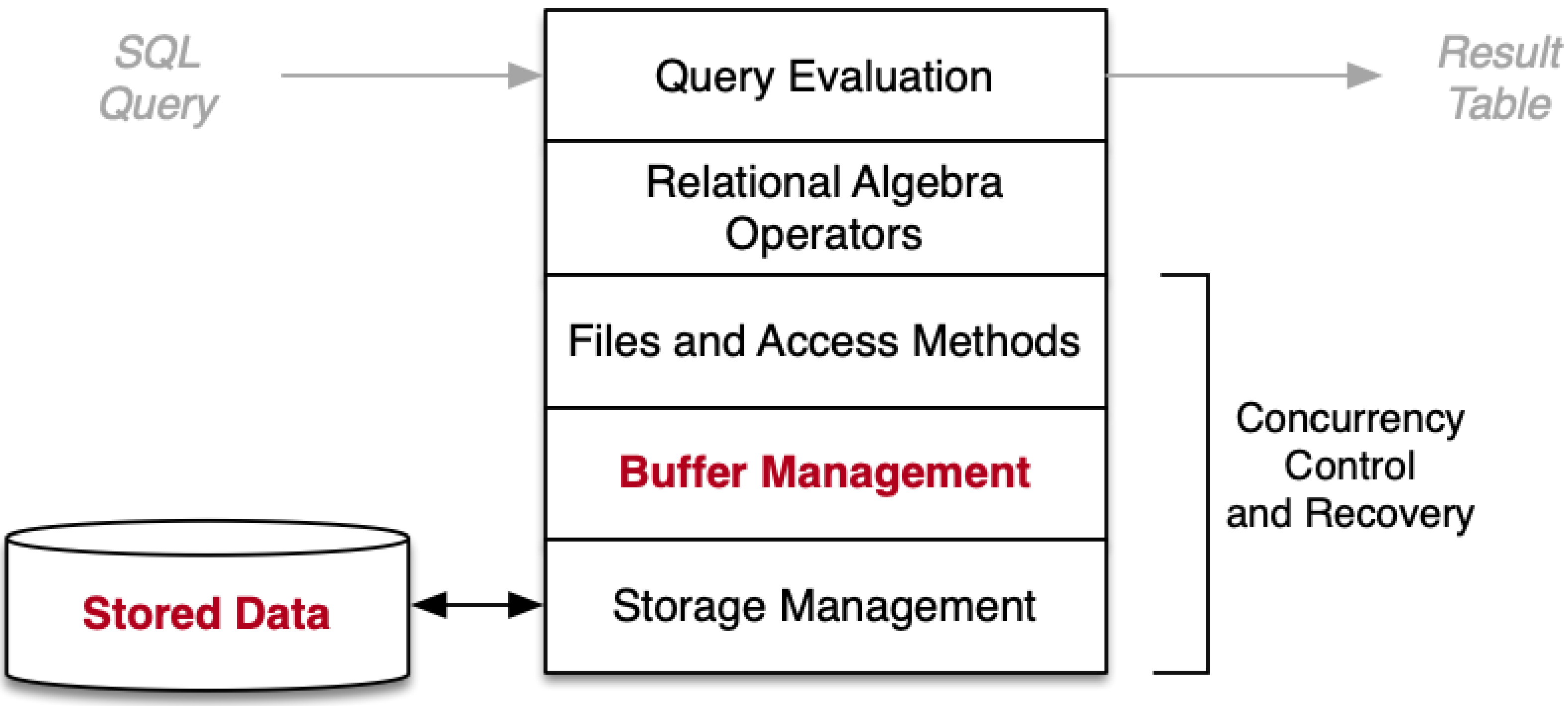
>>

Buffer Pool

❖ Buffer Pool


SQL Query
Query Evaluation
Result Table
Relational Algebra Operators
Files and Access Methods
Buffer Management
Storage Management
Concurrency Control and Recovery
Stored Data

❖ Buffer Pool (cont)

Aim of buffer pool:

- hold pages read from database files, for possible re-use

Used by:

- access methods which read/write data pages
- e.g. sequential scan, indexed retrieval, hashing

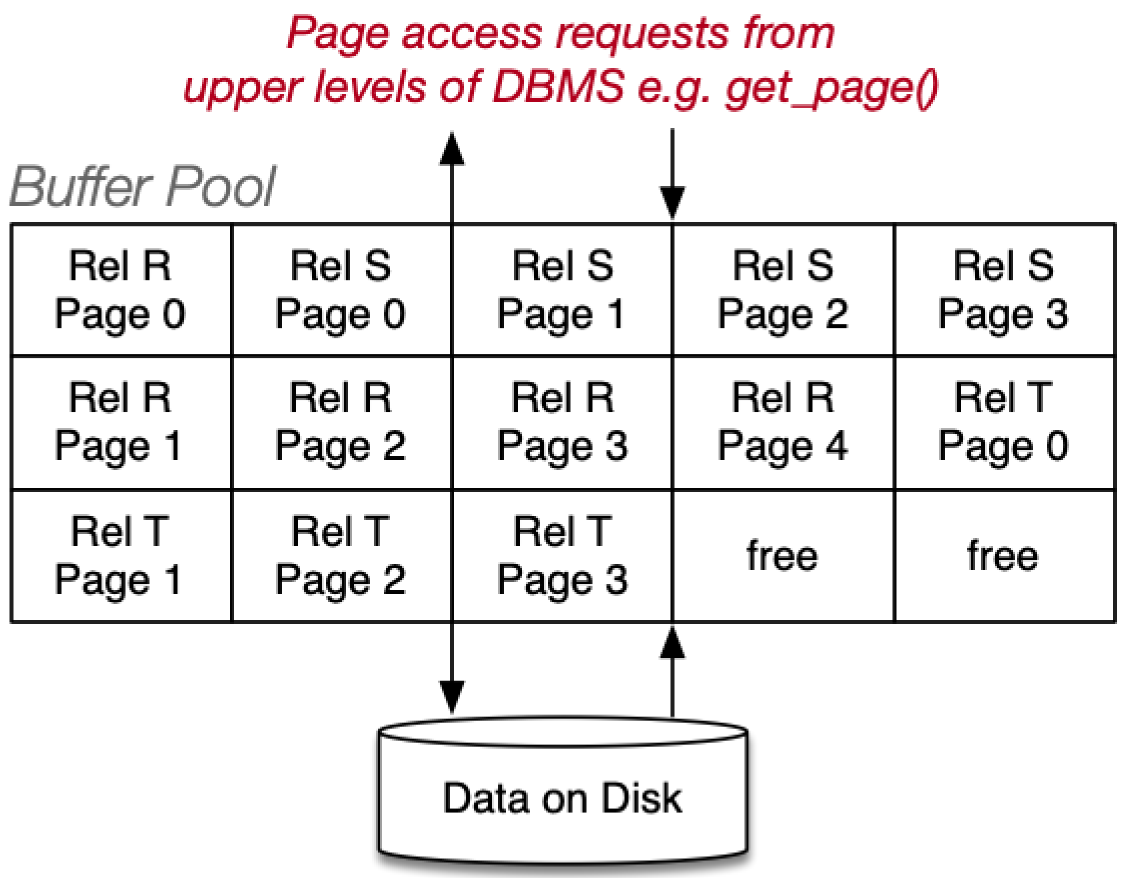
Uses:

- file manager functions to access data files

Note: we use the terms page and block interchangably

<< ^ >>

❖ Buffer Pool (cont)



<< ^ >>

❖ Buffer Pool (cont)

Buffer pool operations: (both take single **`PageID`** argument)

- **`request_page(pid)`**, **`release_page(pid)`**, ...

To some extent ...

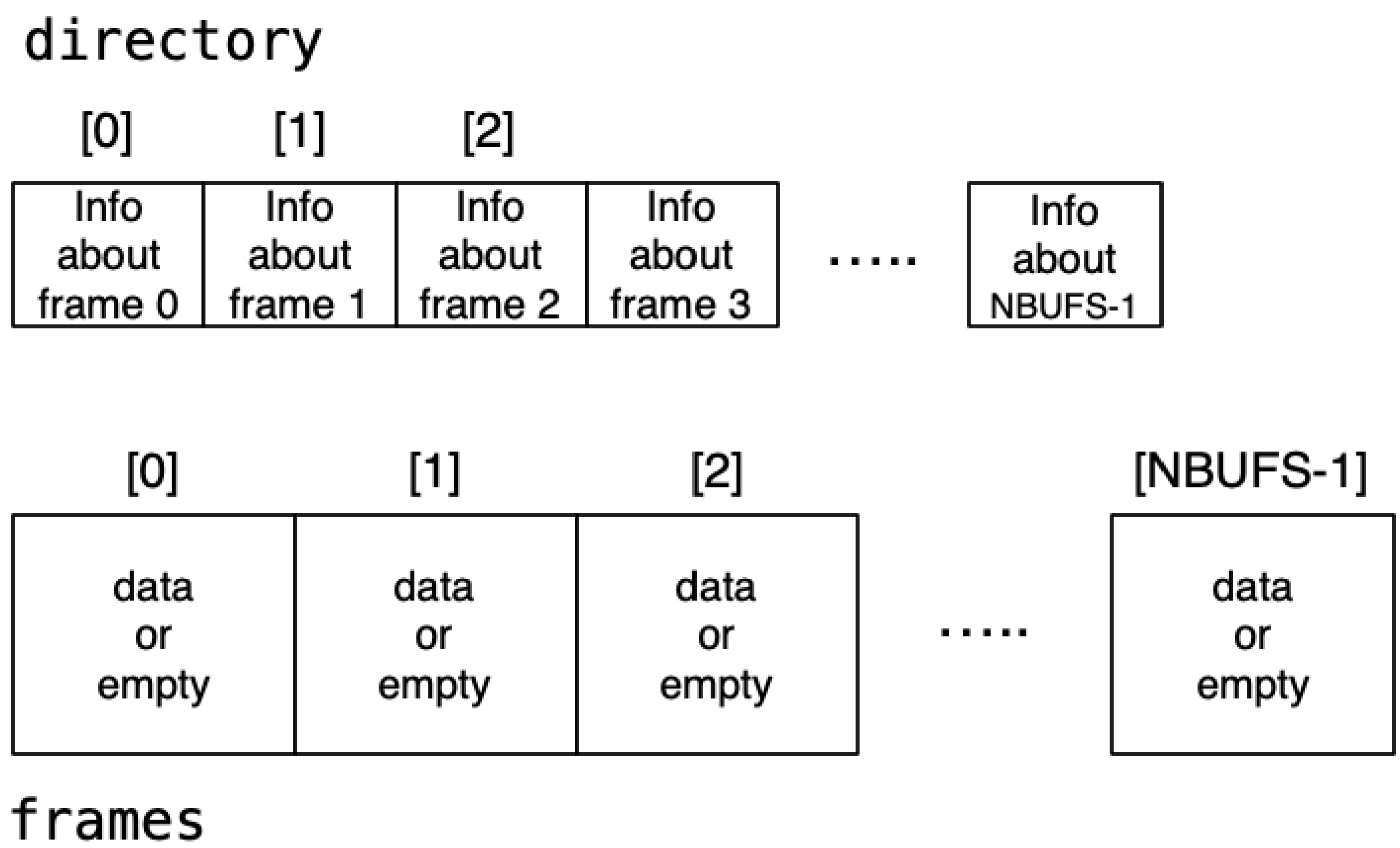
- **`request_page()`** replaces **`getBlock()`**
- **`release_page()`** replaces **`putBlock()`**

Buffer pool data structures:

- **`Page frames[NBUFS]`**
- **`FrameData directory[NBUFS]`**
- **`Page`** is **`byte[BUFSIZE]`**

<< ^ >>

❖ Buffer Pool (cont)



❖ Buffer Pool (cont)

For each frame, we need to know: (`FrameData`)

- which Page it contains, or whether empty/free
- whether it has been modified since loading (dirty bit)
- how many transactions are currently using it (pin count)
- time-stamp for most recent access (assists with replacement)

Pages are referenced by PageID ...

- PageID = BufferTag = (rnode, forkNum, blockNum)

❖ Buffer Pool (cont)

How scans are performed without Buffer Pool:

```
Buffer buf;
int N = numberOfBlocks(Rel);
for (i = 0; i < N; i++) {
   pageID = makePageID(db,Rel,i);
   getBlock(pageID, buf);
   for (j = 0; j < nTuples(buf); j++)
      process(buf, j)
}
```

Requires **N** page reads.

If we read it again, **N** page reads.

❖ Buffer Pool (cont)

How scans are performed with Buffer Pool:

```
Buffer buf;
int N = numberOfBlocks(Rel);
for (i = 0; i < N; i++) {
   pageID = makePageID(db,Rel,i);
   bufID = request_page(pageID);
   buf = frames[bufID]
   for (j = 0; j < nTuples(buf); j++)
      process(buf, j)
   release_page(pageID);
}
```

Requires **N** page reads on the first pass.

If we read it again, 0 ⩽ page reads ⩽ **N**

❖ Buffer Pool (cont)

Implementation of **`request_page()`**

```
int request_page(PageID pid)
{
   if (pid in Pool)
      bufID = index for pid in Pool
   else {
      if (no free frames in Pool)
         evict a page (free a frame)
      bufID = allocate free frame
      directory[bufID].page = pid
      directory[bufID].pin_count = 0
      directory[bufID].dirty_bit = 0
   }
   directory[bufID].pin_count++
   return bufID
}
```

<< ^ >>

❖ Buffer Pool (cont)

The **`release_page(pid)`** operation:

- Decrement pin count for specified page

Note: no effect on disk or buffer contents until replacement required

The **`mark_page(pid)`** operation:

- Set dirty bit on for specified page

Note: doesn't actually write to disk; indicates that page changed

The **`flush_page(pid)`** operation:

- Write the specified page to disk (using **`write_page`**)

Note: not generally used by higher levels of DBMS

❖ Buffer Pool (cont)

Evicting a page ...

- find frame(s) *preferably* satisfying
  - pin count = 0 (i.e. nobody using it)
  - dirty bit = 0 (not modified)
- if selected frame was modified, flush frame to disk
- flag directory entry as "frame empty"

If multiple frames can potentially be released

- need a policy to decide which is best choice

❖ Page Replacement Policies

Several schemes are commonly in use:

- Least Recently Used (LRU)
- Most Recently Used (MRU)
- First in First Out (FIFO)
- Random

LRU / MRU require knowledge of when pages were last accessed

- how to keep track of "last access" time?
- base on request/release ops or on *real* page usage?

<< ∧ >>

❖ Page Replacement Policies (cont)

Cost benefit from buffer pool (with n frames) is determined by:

- number of available frames (more ⇒ better)
- replacement strategy vs page access pattern

Example (a): sequential scan, LRU or MRU, $n \geqslant b$

First scan costs b reads; subsequent scans are "free".

Example (b): sequential scan, MRU, $n < b$

First scan costs b reads; subsequent scans cost $b - n$ reads.

Example (c): sequential scan, LRU, $n < b$

All scans cost b reads; known as sequential flooding.

<< ^ >>

❖ Effect of Buffer Management

Consider a query to find customers who are also employees:

```
select c.name
from   Customer c, Employee e
where  c.ssn = e.ssn;
```

This might be implemented inside the DBMS via nested loops:

```
for each tuple t1 in Customer {
    for each tuple t2 in Employee {
        if (t1.ssn == t2.ssn)
            append (t1.name) to result set
    }
}
```

❖ Effect of Buffer Management (cont)

In terms of page-level operations, the algorithm looks like:

```
Rel rC = openRelation("Customer");
Rel rE = openRelation("Employee");
for (int i = 0; i < nPages(rC); i++) {
    PageID pid1 = makePageID(db,rC,i);
    Page p1 = request_page(pid1);
    for (int j = 0; j < nPages(rE); j++) {
        PageID pid2 = makePageID(db,rE,j);
        Page p2 = request_page(pid2);
        // compare all pairs of tuples from p1,p2
        // construct solution set from matching pairs
        release_page(pid2);
    }
    release_page(pid1);
}
```

<< ^

❖ Effect of Buffer Management (cont)

Costs depend on relative size of tables, #buffers (n), replacement strategy

Requests: each rC page requested once, each rE page requested rC times

If nPages(rC)+nPages(rE) ⩽ n

- read each page exactly once, holding all pages in buffer pool

If nPages(rE) ⩽ n-1, and LRU replacement

- read each page exactly once, hold rE in pool while reading each rC

If n == 2   (worst case)

- read each page every time it's requested

Produced: 22 Feb 2021