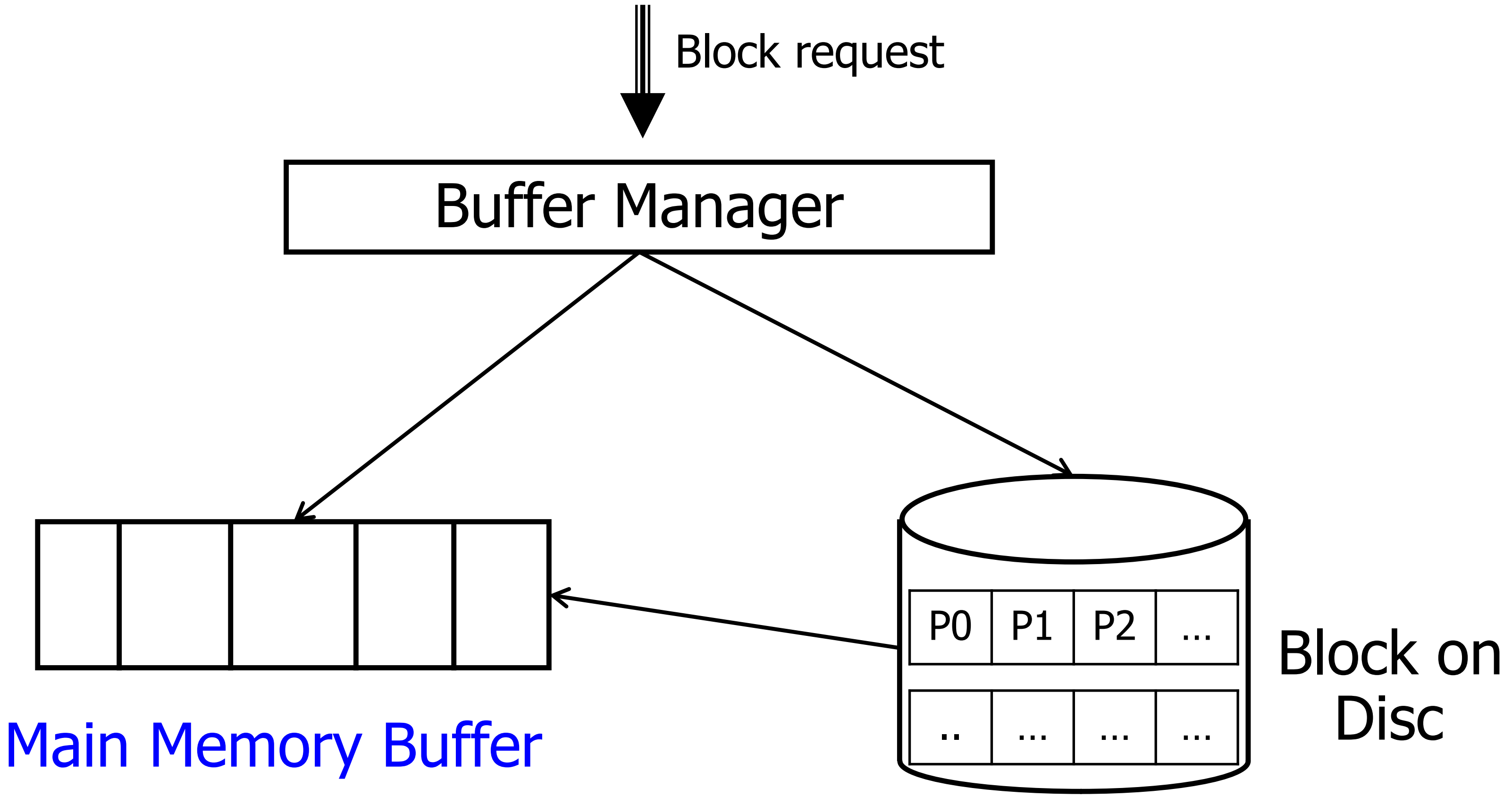
Datenbanksysteme II: Caching

Ulf Leser

Content of this Lecture

- Caching Overview
- Accessing tuples
- Cache replacement strategies
- Cache fetch policy: Prefetching

Caching = Buffer Management



IO Buffering

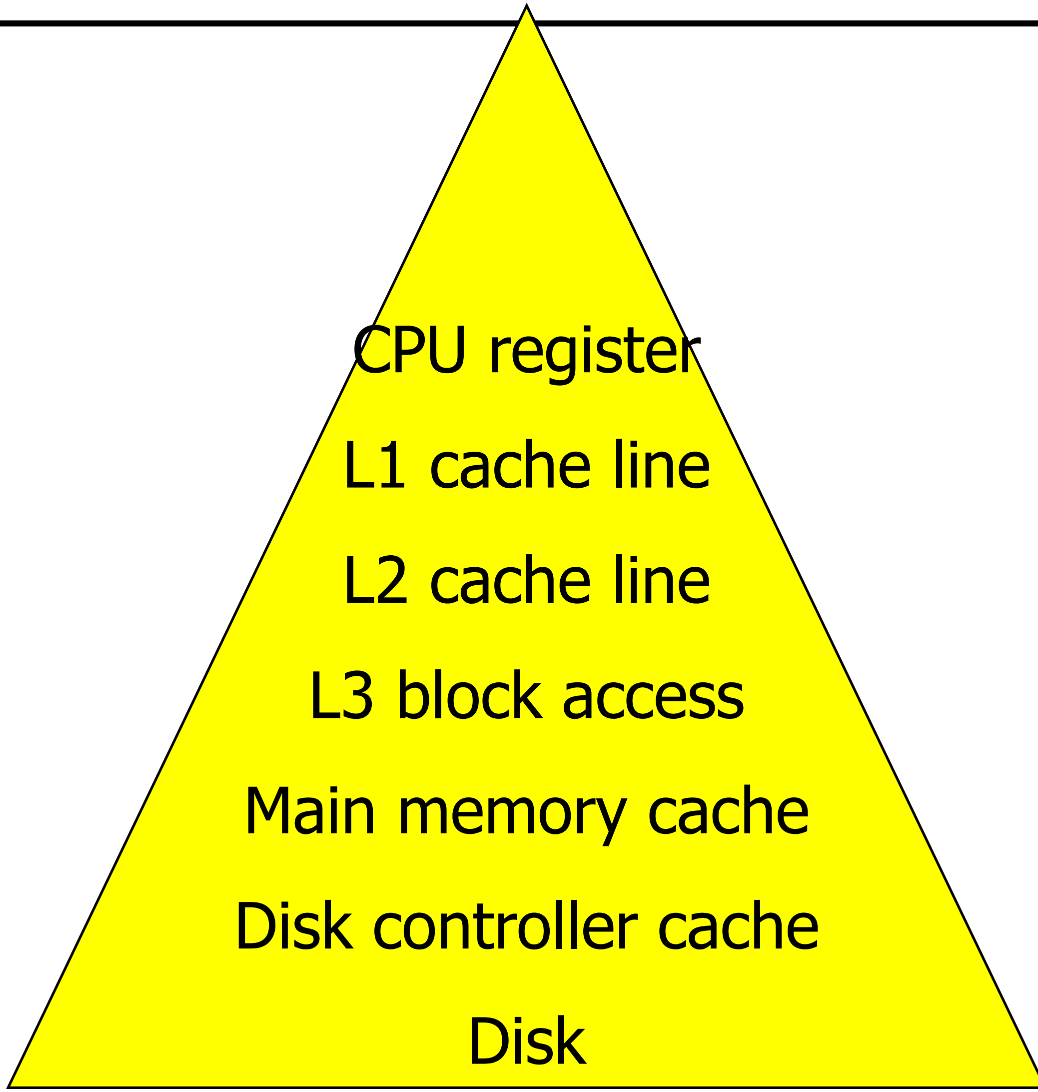
- RDBMS requests block Y from buffer manager
- Buffer manager checks if …
  - Y in cache: Grant access

    Address rewriting
  - Y not in cache
    - No free space in buffer?
      - Choose block Z in buffer
      - If Z has been changed – write Z to disc
      - Mark Z as free and proceed

      Cache replacement policy
    - Free space available?
      - Load Y into free space
      - Grant access

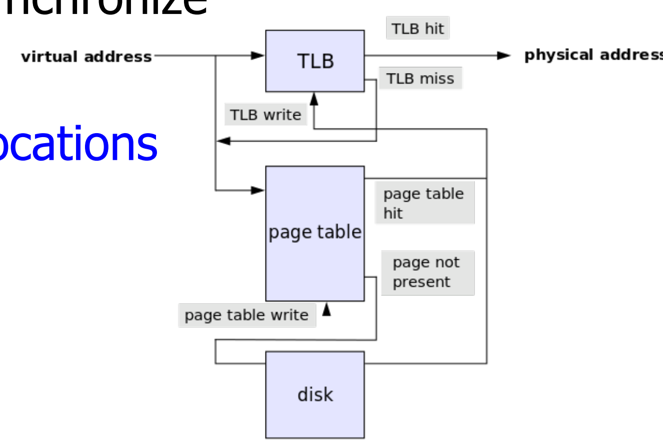
      Cache fetch policy

Same Problem across the Entire Storage Hierarchy



Finding a Block

- We need to check if block Y is in buffer
  - Y is physical block ID on disk yet a logical block ID in a virtual address space in memory
  - In memory, the blockID doesn't tell anything about the location
- Possibilities
  - Memory blocks store their own block ID
    - Find Y: Search all blocks
    - Slow, but no global data structures to synchronize
  - Mapping table
    - Manage a list of all blockIDs with their locations
    - Find Y: Scan this list
      - less memory access, can be sorted
    - Fast, requires synchronization
    - Performed in any OS all the time
      - OS page table, memory translation using translation lookaside buffer



Source: https://en.wikipedia.org/wiki/Page_table

Access with a TID

- Assume we access the 10^{th} attribute of a tuple TID
- Assume we use a mapping table – what should we return?
- By delegation: x:=getData(TID,10); y=getData(TID; 11)
  - getData() translates virtual TID into mem location and returns attribute value
  - Access requires one indirection
- By pointer: adr := getAdr(TID); x:=adr[10]; y=adr[11]
  - Adr now is a physical memory pointer that can be reused
  - Faster, but …

Access with a TID

- Assume we access the 10th attribute of a tuple TID
- Assume we use a mapping table – what should we return?
- By pointer: adr := getAdr(TID); x:=adr[10]; y=adr[11]
  - Adr now is a physical memory pointer that can be reused
  - Pinned tuples: Direct reference exists
    - From an index, from a transaction, from a running query, …
    - Tuple must not move, block must not be evicted
      - Cache manager must know, has less options
    - Requires special means to mark pinned tuples (ref counter)
    - If adr becomes invalid – core dump
  - Unpinned tuples: No references to location exist
    - Tuple may be moved
    - Block may be evicted

Content of this Lecture

- Caching Overview
- Accessing tuples
- Cache replacement strategies
- Cache fetch policy: Prefetching

Caching Strategies – Going Wrong

- Imagine a nested loop join
  - Outer relation A has 10 blocks, inner relation B has 6 blocks
- Buffer size 6 blocks
- Assume caching with FIFO (first in – first out)
  - Cache is filled with A1 and B1, B2, B3, B4, B5
  - Loading B6 replaces A1
  - For next inner loop, A1 must be loaded again, replacing B1
    - We need the next record in A1, which is not in memory any more
  - For loading A2, B2 is replaced, B1 replaces B3, …
- FIFO is a typical OS caching strategies
- DB needs to be able to control cache behavior

Caching Strategies – Better Strategy

- Imagine a nested loop join
  - Outer relation A has 10 blocks, inner relation B has 6 blocks
- Buffer size 6 blocks
- Proceed as follows
  - Cache is filled with A1 and B1, B2, B3, B4, B5
  - Build an inner-inner loop (blocked nested loop)
    - Keep A1 until finished with all its records
  - After B1,… loading B6 replaces, e.g., B1
  - For next outer loop, A2 replaces A1
  - Inner loop: B1, B2, B3, B4, B6 without replacement
  - Next: B6 replaces B2
  - …

Caching Aspects

- What to manage?
  - Records, blocks, chunks (sequences of blocks), tables
- How many blocks to load?
  - Optimal strategy ensures block is in buffer at time of request
  - "Block-at-a-time" versus "Read ahead" (prefetching)
- Which blocks to evict (replace)?
  - Cache replacement strategies
- Good cache management requires information flow from DB layers to buffer manager
  - Example: Scanning a relation (read ahead)
  - Example: Executing a "Nested Loop Join" (fix outer-loop blocks)

Granularity of Cached Units

- Records: Makes no sense at a blocked device like HDD
- Blocks (default): OS blocks or database blocks
- Chunks
  - Group blocks into larger "chunks"
  - IO on chunks can exploit sequentially access
  - Good for large operations (large table joins or sorts)
  - Bad for many local accesses (single records) across all tables
- Tables
  - Table are like ultra-large chunks
    - Whose size cannot be controlled by memory manager - bad
  - But: Fix heavily used (small) tables
  - E.g.: System catalog, Oracles CACHE parameter

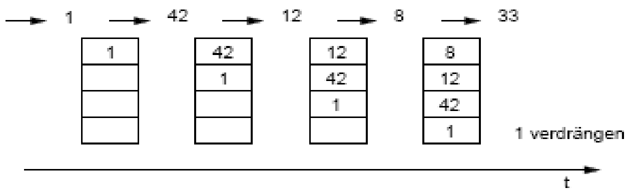
Cache Replacement: Based on What?

- What do we know of a block that is correlated to the probability of its future use?
  - Age
    - Time since block was loaded first
    - Or: Time since last access in memory
  - Living references (block is pinned)
  - Changed records (incurs block write)
  - Demand: Number of accesses over (recent) time
- Trade-offs
  - Young blocks have few refs, but are involved in current operations
  - Old blocks have many refs, but might already be out-of-fashion
  - Demand often is in bursts – use sliding window ("recent")
- Properties can be combined / weighted for decision

General Policies

- Last in first out (LIFO): Replace block that was loaded last
- First in first out (FIFO): Replace block that was loaded first
- Least frequently used (LFU): Replace block with smallest demand
- Least recently used (LRU): Replace block that was not access for the longest time
- Least reference density (LRD): Replace block with the worst ratio of age and demand
- Clock: Approximate age with less management
- Random: Chose block at random (nothing to manage)

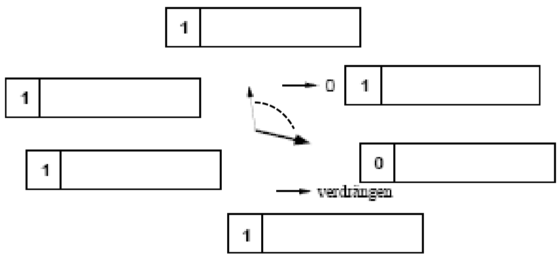
Implementing LRU with a LRU queue



- When block is requested
  - Critical operation: Search blockID in queue
  - Implemented with two lists
    - Queue sorted by least access
    - Hashmap: BlockIDs to queue positions (quasi-constant time)
- Access block
  - Search blockID in hashmap (almost O(1))
  - Block in cache
    - Follow pointer to queue
    - Delete entry in queue and reinsert on top of queue (O(log(n)))
  - Otherwise; load block: Add at top of queue (O(log(n)))
- Evict block
  - Find least block in queue; remove from queue and hashmap

CLOCK ("2nd chance" caching)

- Reserve a "used" bit B in each block
- Define cyclic order between blocks (e.g. linked list)
- Initialize a pointer to a randomly chosen block



- Block is used
  - Set B:=1
- Block needs to be replaced
  - Move cyclic pointer
  - If used=1, change to 0 and move pointer to next block
  - If used=0, replace this block and move pointer to next block
    - New block has B:=1
- Makes queue superfluous (hashmap still needed)

Content of this Lecture

- Caching Overview
- Accessing tuples
- Cache replacement strategies
- Cache fetch policy: Prefetching

Cache fetch policy: Pre-fetching or not

- Prefetching: Load blocks not yet needed but probably soon
- Examples
  - If block from relation is requested, also load next blocks
    - Possible full table scan?
  - If object is accessed, also load referenced objects
    - Not implemented in RDBMS, but in OODBMS / OR-mappers
- Disc pre-fetching – if sector is requested, read entire track
- Pre-fetching incurs replacement of multiple blocks
  - Evicts more blocks without knowing if this is for good
- Using sequential and asynchronous (non-blocking) IO, pre-fetching may save a lot of time

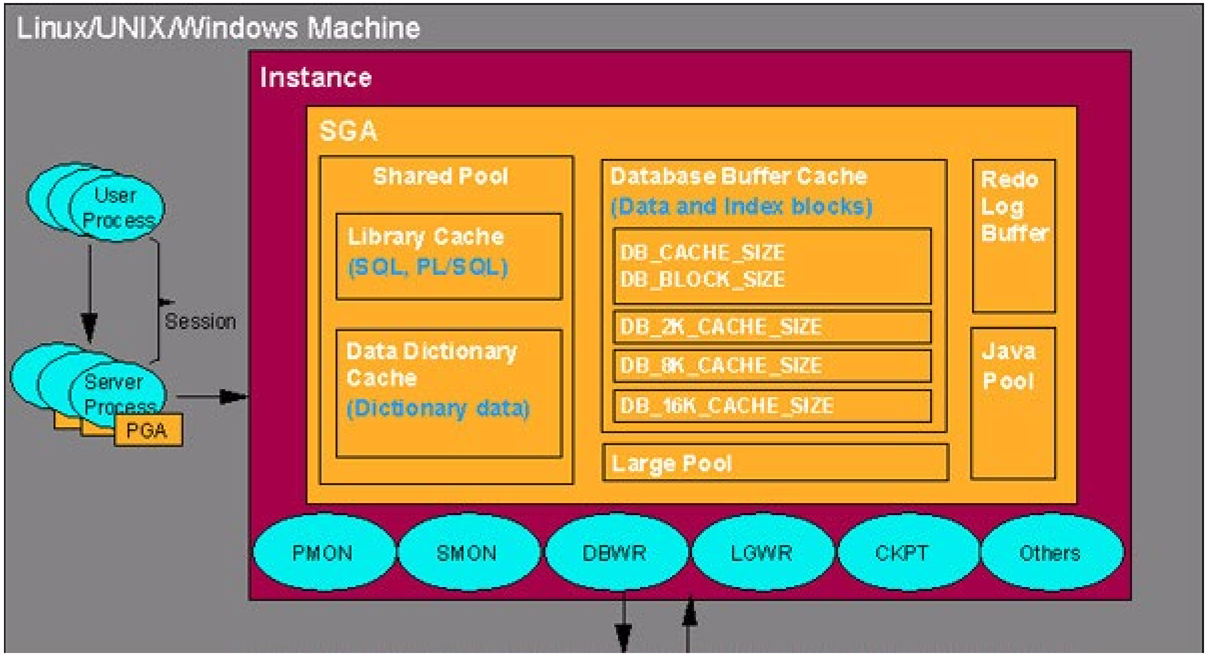
Other Cache Issues

- Be aware: Your data is not written immediately
  - With caching, data stays on volatile device much longer than without
  - Cache manager needs to check if writing before replacement is necessary at all (dirty flag)
  - Special care required – recovery strategies
- Cache consistency in distributed (shared-nothing) systems
  - If more than one process caches in different locations, data may become stale
  - Requires costly synchronization (the dead of distributed systems)
- Cache consistency in concurrent TX systems
  - If more than one TX changes data, multiple versions of a block may exist
  - Requires costly synchronization (see TX handling)

Semantic Caching: Caching Query Result

- Example
  - Q1: "Select name from person where age>45"
  - Q2: "Select * from person where age>18"
  - Q1 can be answered using result tuples from Q2
- Powerful but complicated technique
  - Can a query be answered using results of one or more other q's?
  - Query containment, "answering queries using views"
- Very complicated for write operations
  - Cached result blocks are not IO blocks
- Semantic caching not used by any real DB today
  - Note: Normal caching sometimes "mimics" semantic caching
  - If Q1 executed after Q2, blocks from Q2 are in cache
  - But: Computations need to be repeated (e.g. aggregation)

Many Tasks Compete for Main Memory



- SGA: System global area
  - Processes communicate through SGA
  - Requires locking of main memory structures – latches
- Library cache: buffers SQL prepared statements using LRU
- Java pool: area for java stored procedures
- Each process additionally gets its PGA (process global area)
- Each area is limited and can become a bottleneck

Take-Home

- Many cache strategies have been (and are still) developed
- General versus domain-specific
- Simple strategies are surprisingly good in many cases
  - LRU or even random
  - PostGreSql / MySql: LRU / Clock
  - Commercial databases: Unknown
    - With operator-depending fixing of blocks and special tricks for large operations