CPT-S 415

Big Data

Yinghui Wu

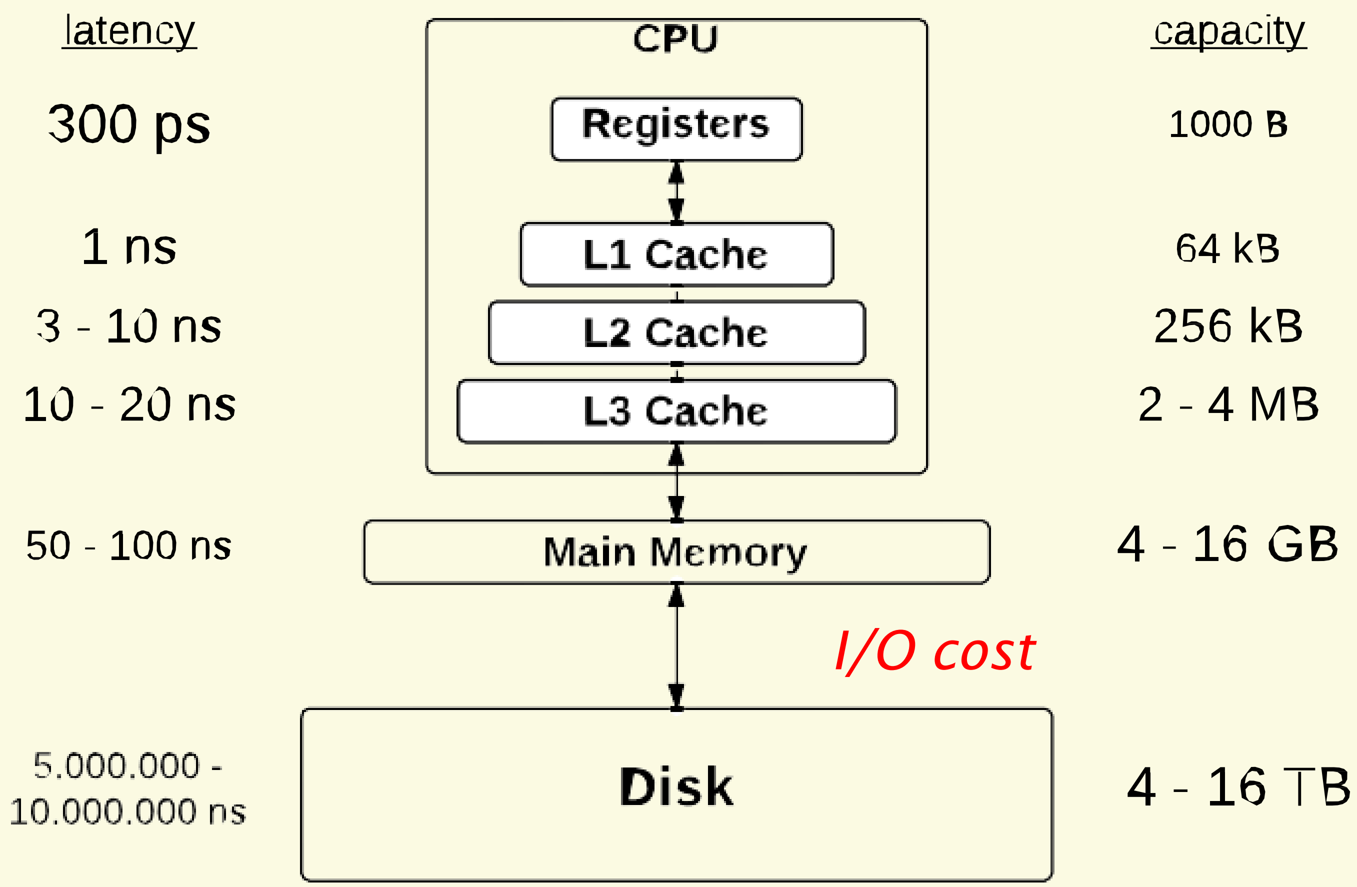
EME B45

CPT-S 415
Big Data

NewSQL databases

- ✓ In-memory DBMS

Recall Computer Architecture



Data taken from [Hennessy and Patterson, 2012]

Disk-oriented DBMS

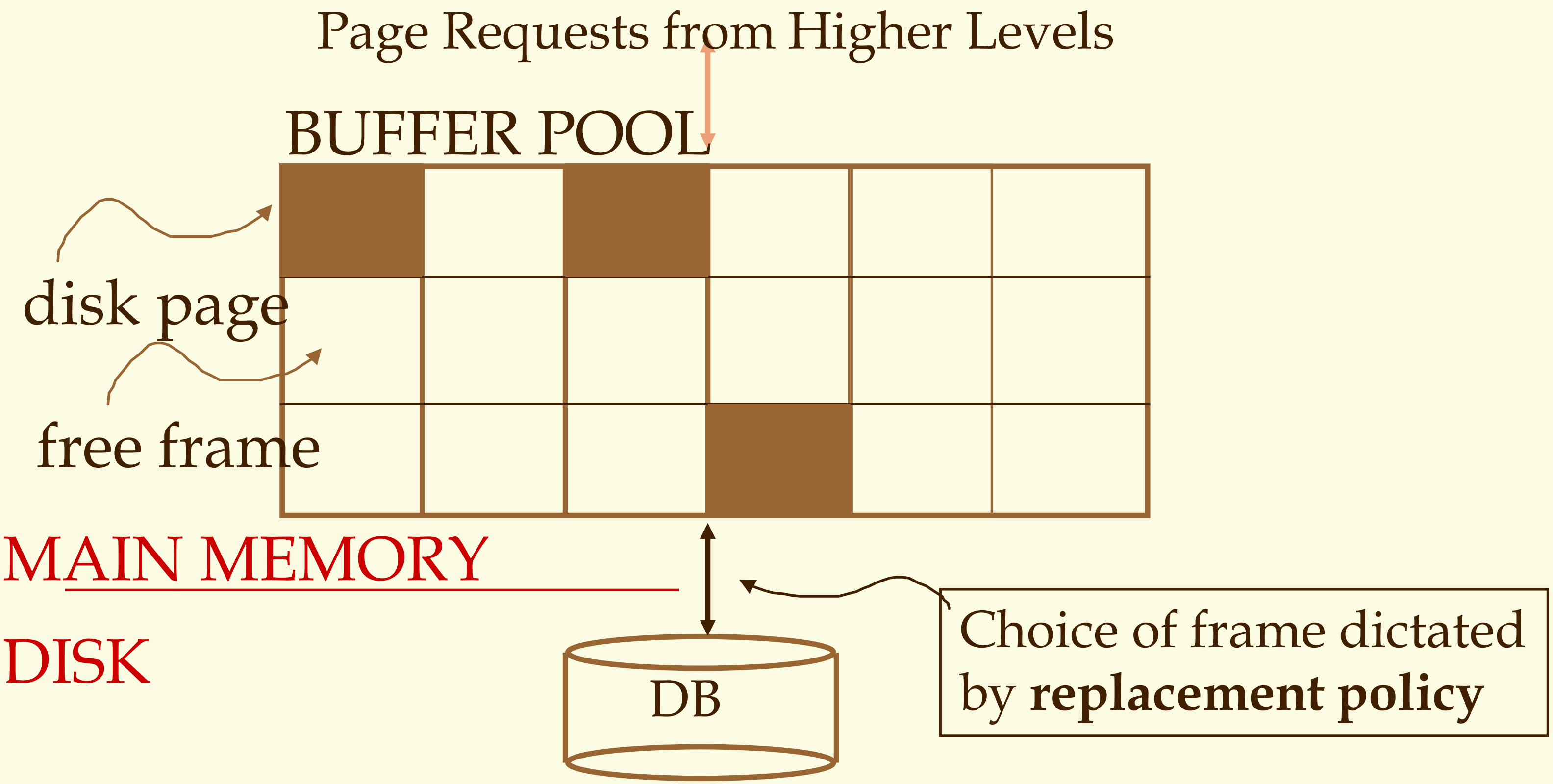
Disk- Oriented DBMS

The primary storage location of the database is on non-volatile storage (e.g., HDD, SSD).

→ The database is organized as a set of fixed-length blocks called slotted pages.

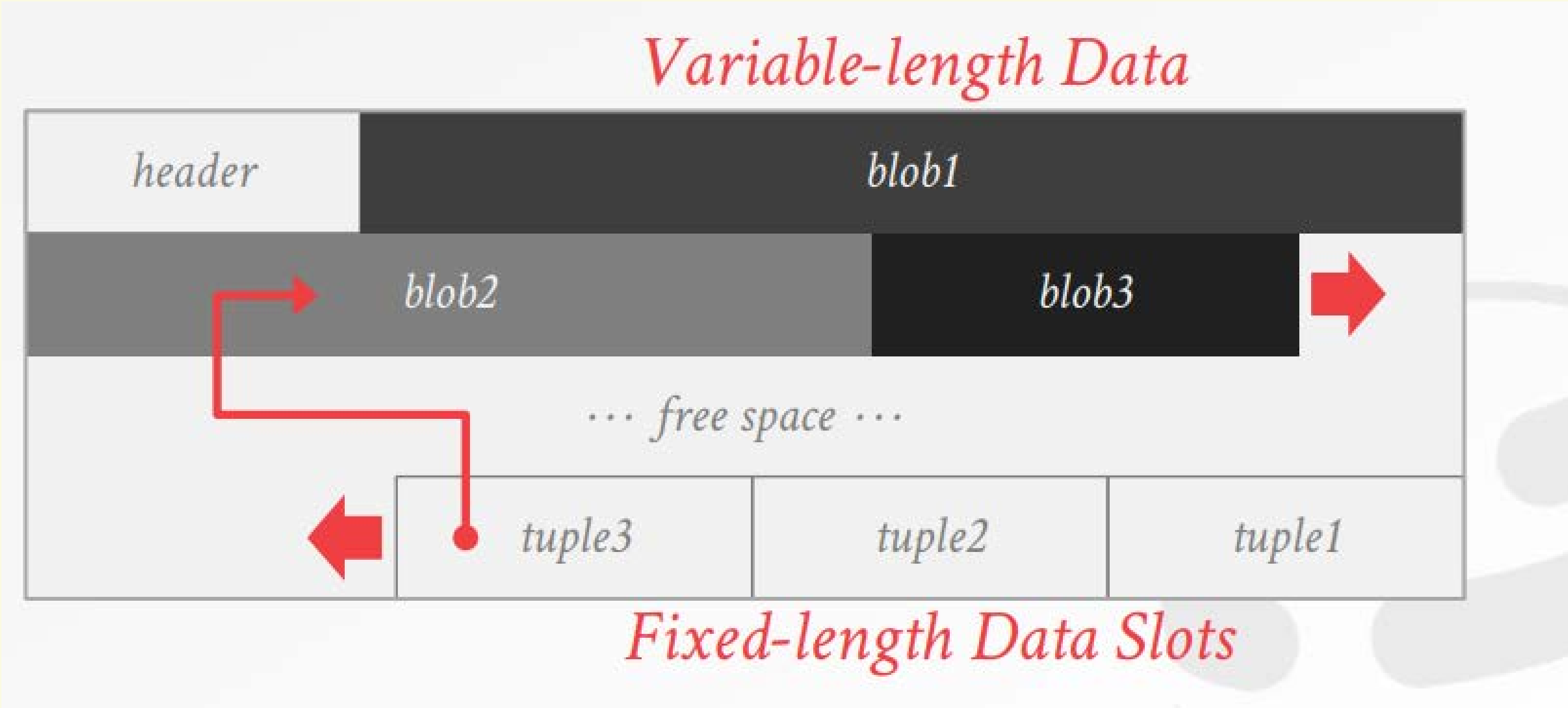
The system uses an in-memory buffer pool to cache blocks fetched from disk.
→ Its job is to manage the movement of those blocks back and forth between disk and memory.

Page request



- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageId> pairs is maintained.

A page

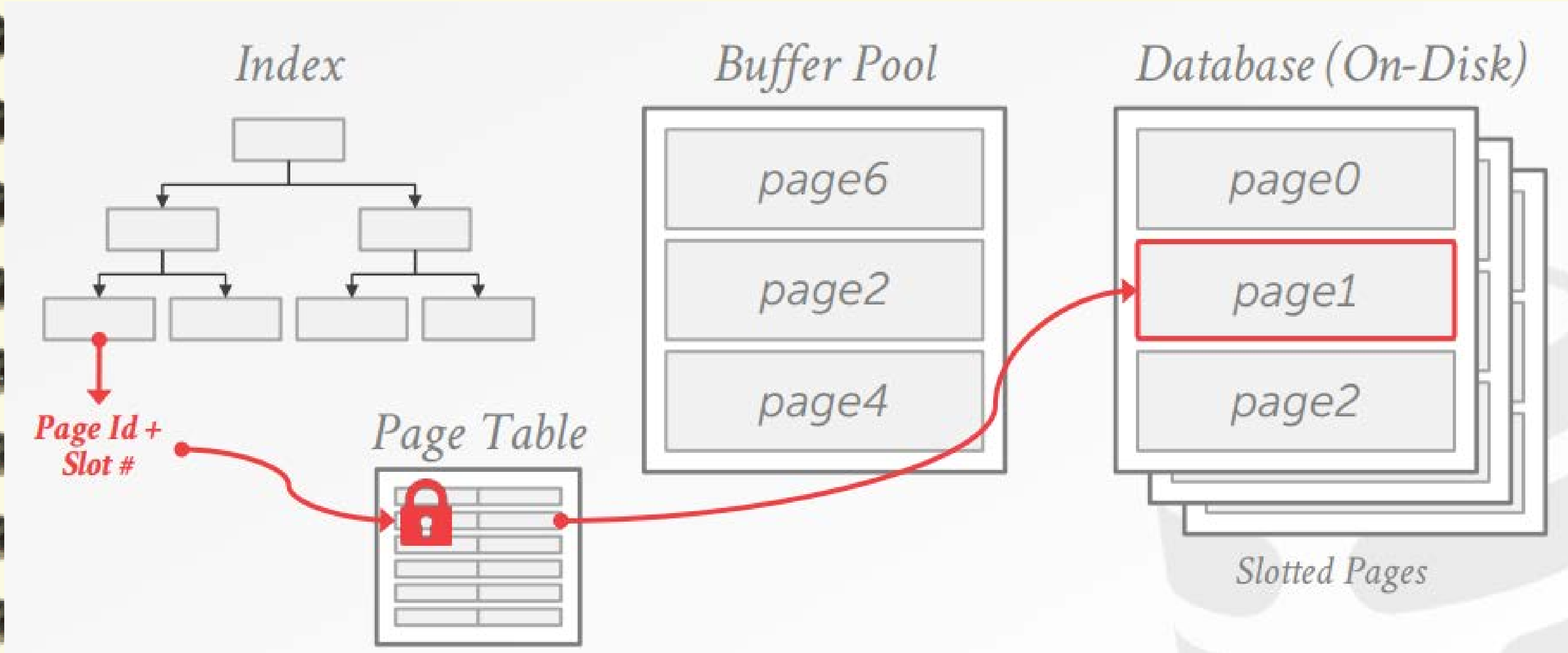


Variable-length Data
header
blob1
blob2
blob3
··· free space ···
tuple3
tuple2
tuple1
Fixed-length Data Slots

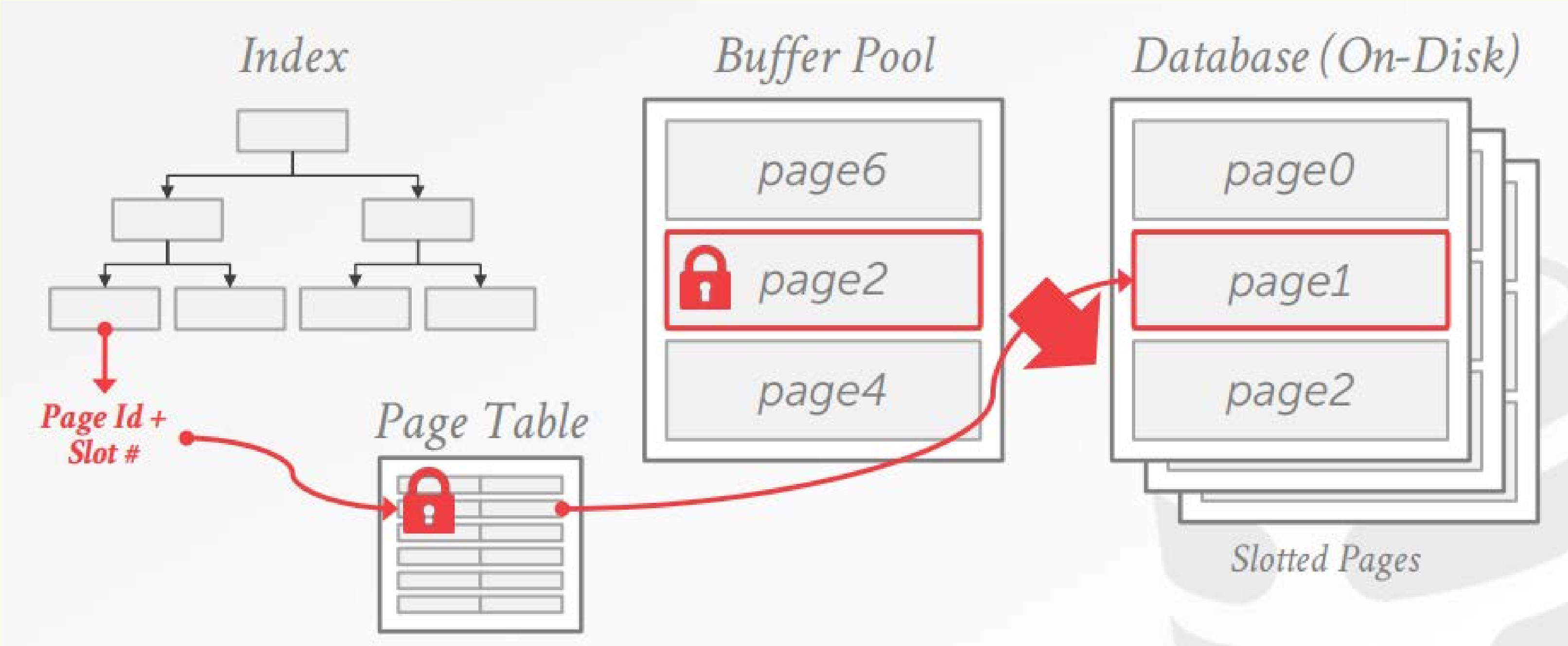
Buffer Pool Management

- When Query access a page:
  - DBMS checks if page is in memory
    - No – retrieve from disk and copy into frame of buffer pool
      - No free frames: find a page to evict
      - Dirty page evicted – write back to disk
    - Yes – translate on-disk addresses to in-memory addresses

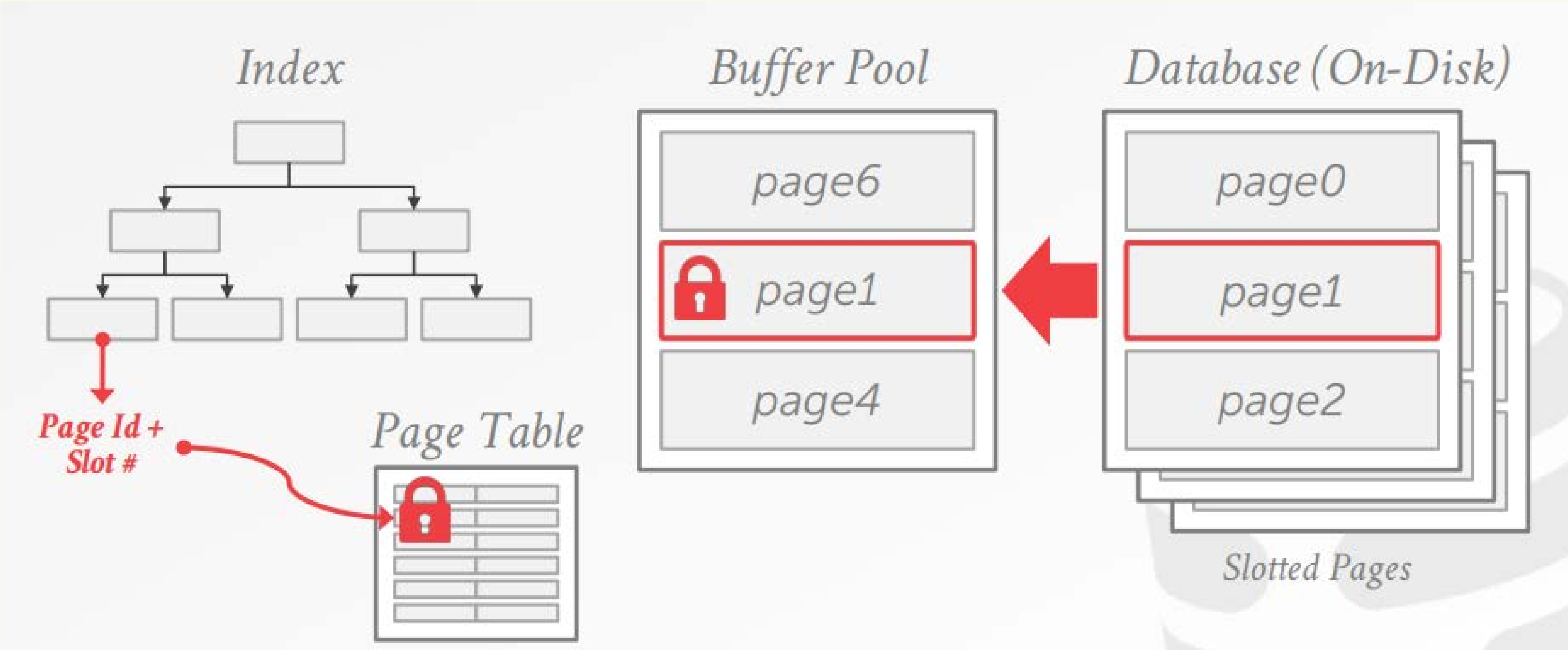
Example: page request



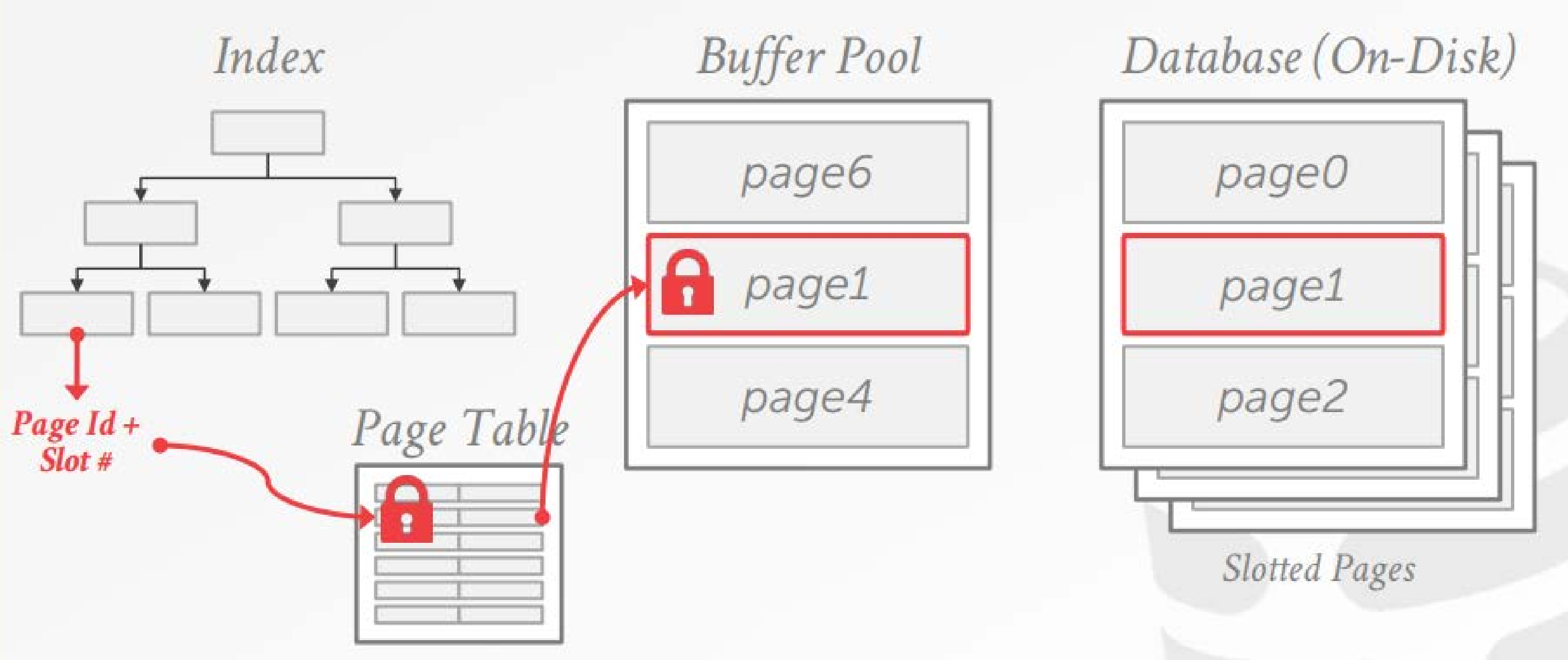
Example: page request



Example: page request



Example: page request



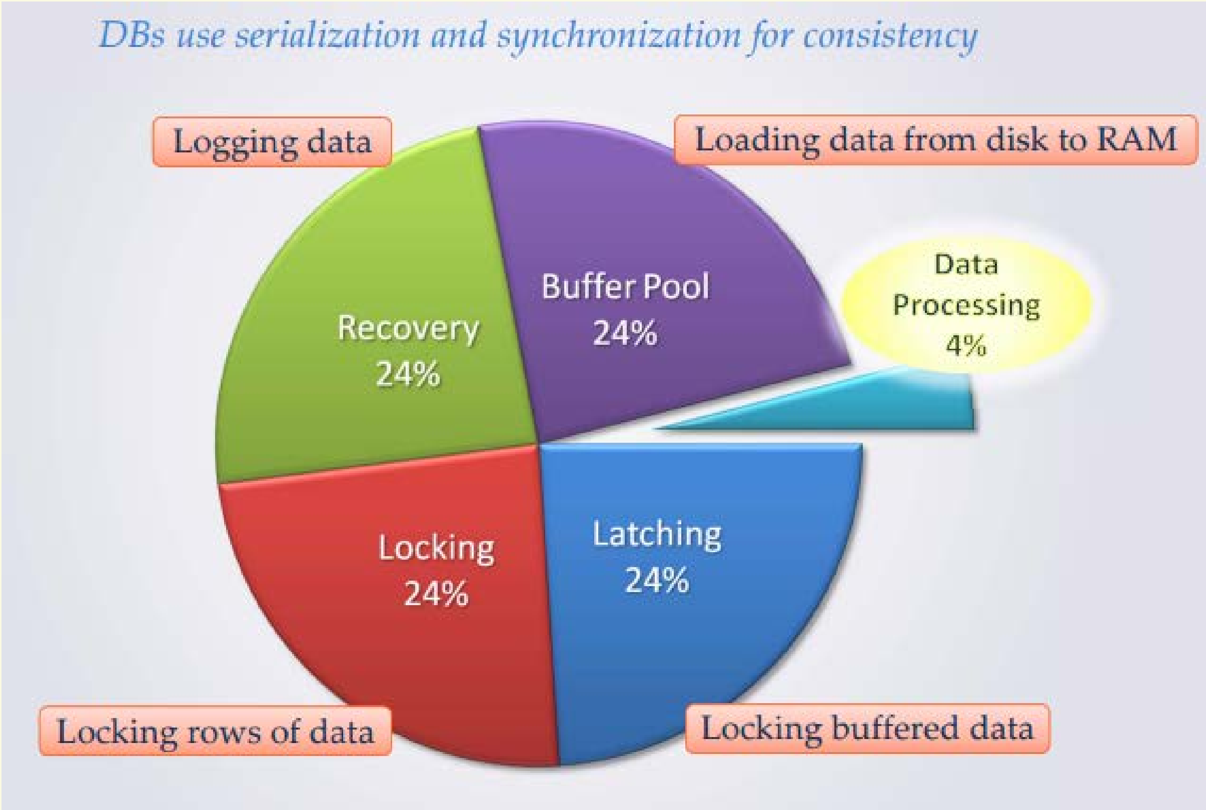
Overhead

- ✓ Buffer pool: Each tuple access needs to go through buffer pool management regardless of whether the data will always be in memory
  - – Translate tuple record id to memory location
  - – Pin pages to make sure they are not swapped to disk
- ✓ Concurrency control
  - – ACID guarantee: set locks and latches
- ✓ Logging & recovery
  - – “steal” + “no-force” buffer pool policy
  - – Log contains before and after images of modified record.

Overhead

- ✓ Buffer pool: Each tuple access needs to go through buffer pool management regardless of whether the data will always be in memory
  - Translate tuple record id to memory location
  - Pin pages to make sure they are not swapped to disk
- ✓ Concurrency control
  - ACID guarantee: set locks and latches
- ✓ Logging & recovery
  - “steal” + “no-force” buffer pool policy
  - Log contains before and after images of modified record.

Traditional DBMS overheads



"Removing those overheads and running the database in main memory would yield orders of magnitude improvements in database performance"

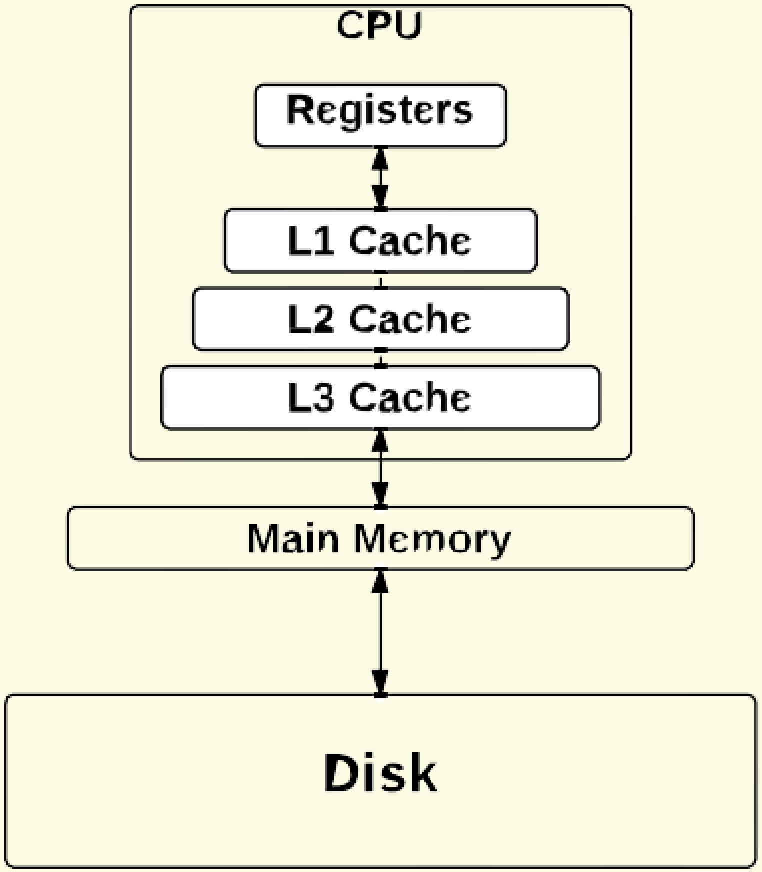
NewSQL design principles

- SQL + ACID + performance and scalability through modern innovative software architecture

- *Principle 1: minimizing or stay away from locking*
- *Principle 2: rely on main memory*
- *Principle 3: try to avoid latching*
- *Principle 4: cheaper solutions for HA*

In-memory databases

Recall Computer Architecture

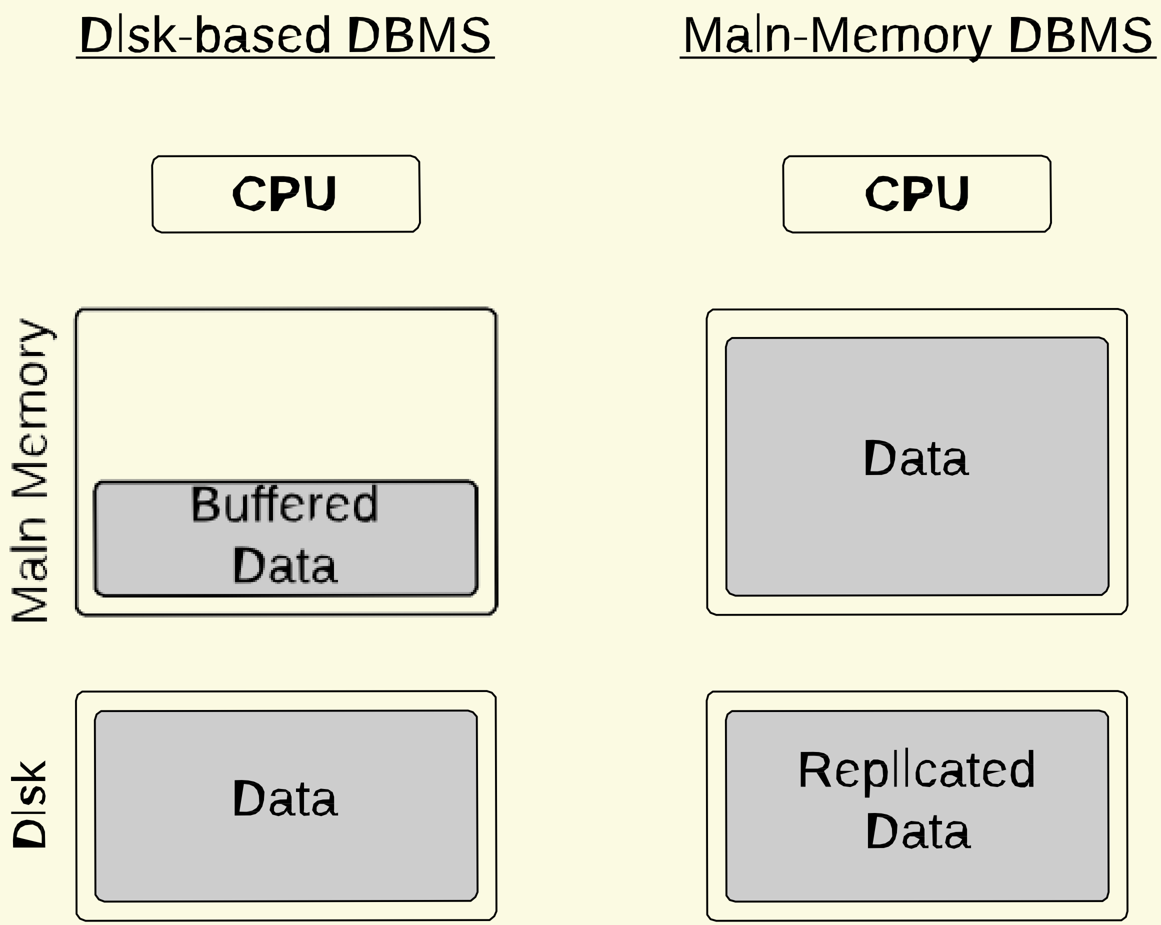


latency		capacity
300 ps	Registers	1000 B
1 ns	L1 Cache	64 kB
3 - 10 ns	L2 Cache	256 kB
10 - 20 ns	L3 Cache	2 - 4 MB
50 - 100 ns	Main Memory	4 - 16 GB
5.000.000 - 10.000.000 ns	Disk	4 - 16 TB

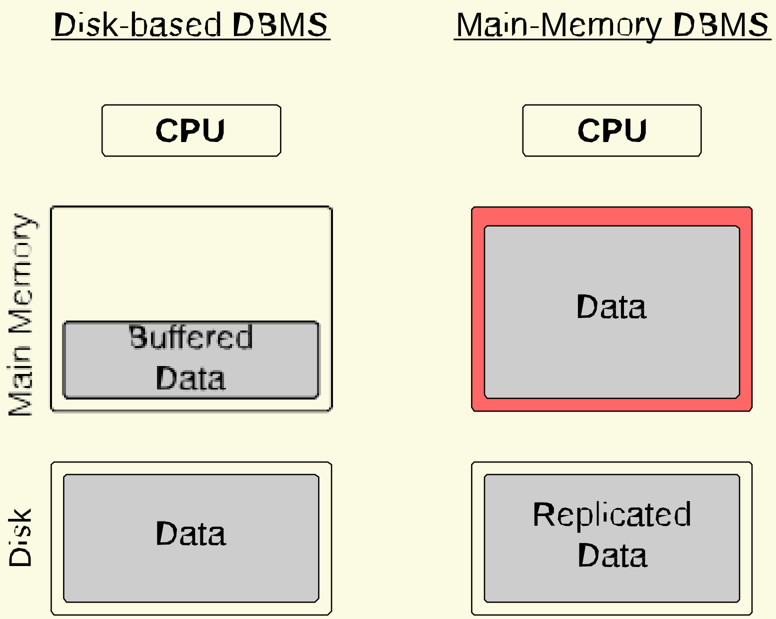
Cache cost – In-memory store

Data taken from [Hennessy and Patterson, 2012]

Disk-based vs. Main-Memory DBMS



Disk-based vs. Main-Memory DBMS (2)

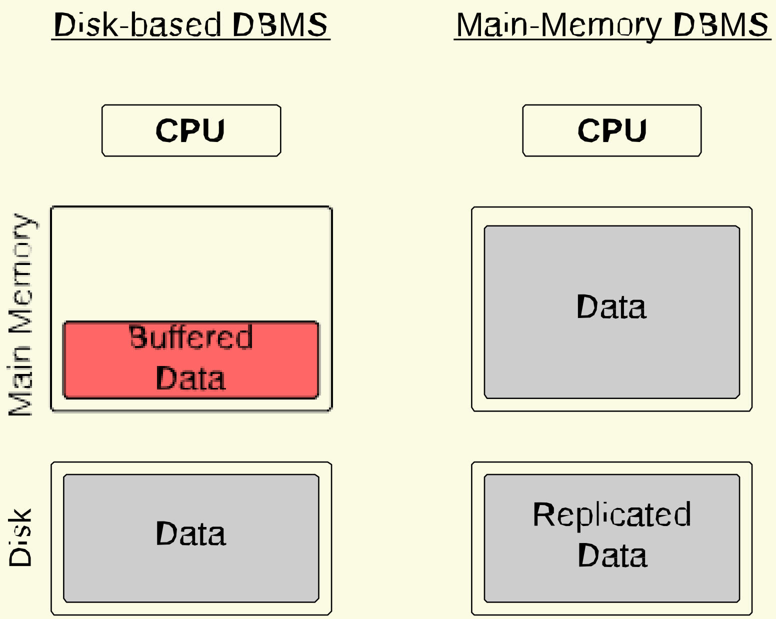


ATTENTION: Main-memory storage != No Durability

→ ACID properties have to be guaranteed

→ However, there are new ways of guaranteeing it, such as a second machine in hot standby

Disk-based vs. Main-Memory DBMS (3)

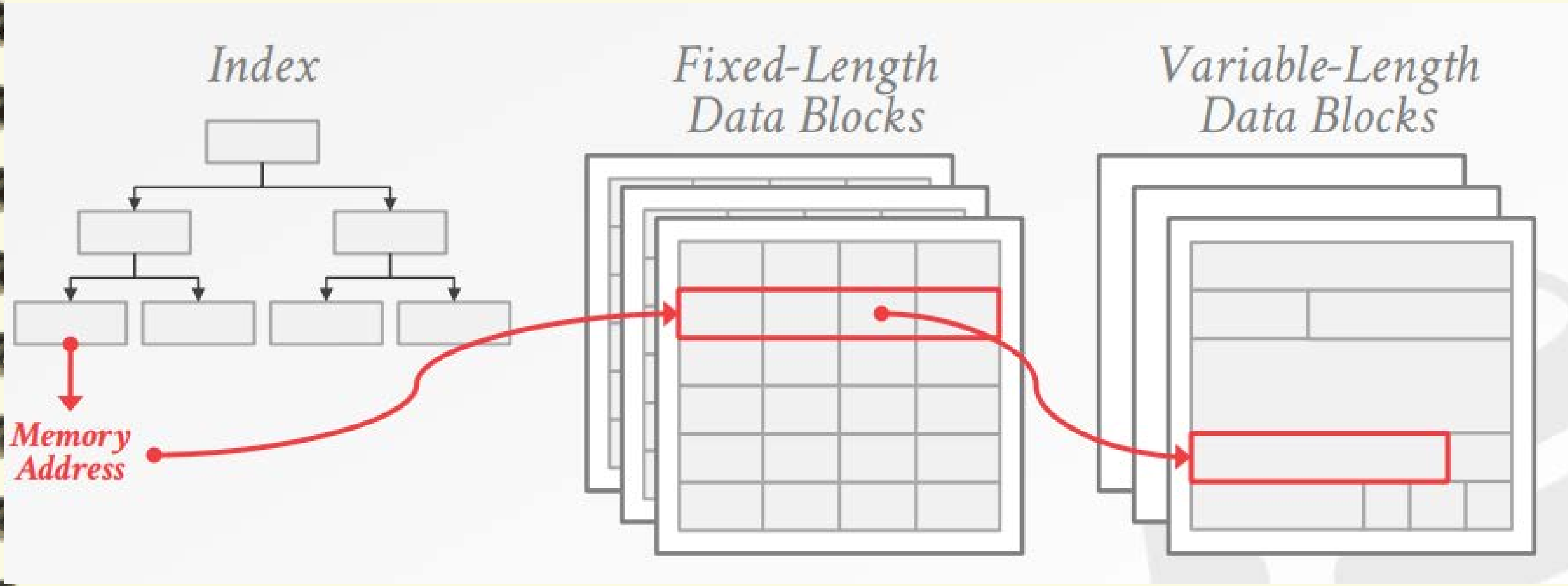


Having the database in main memory allows us to remove buffer manager and paging

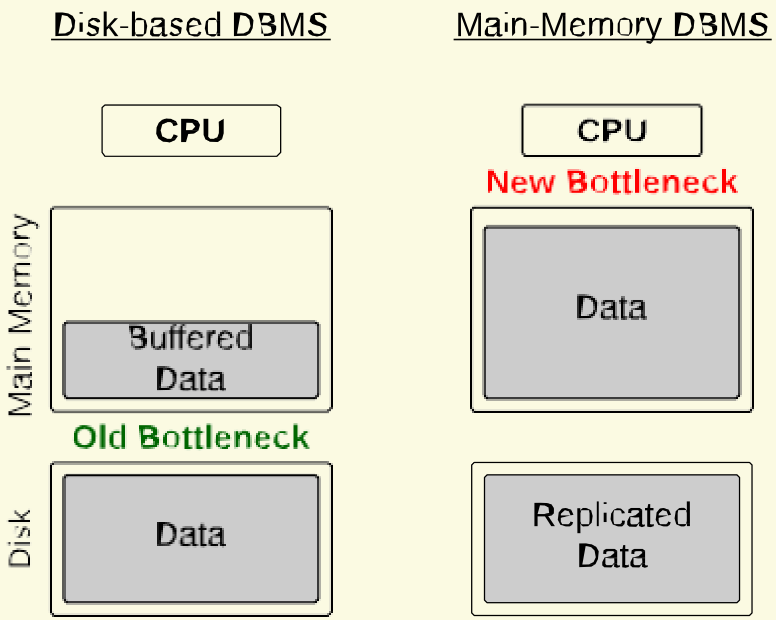
→ Remove level of indirection

→ Results in better performance

In-memory data organization



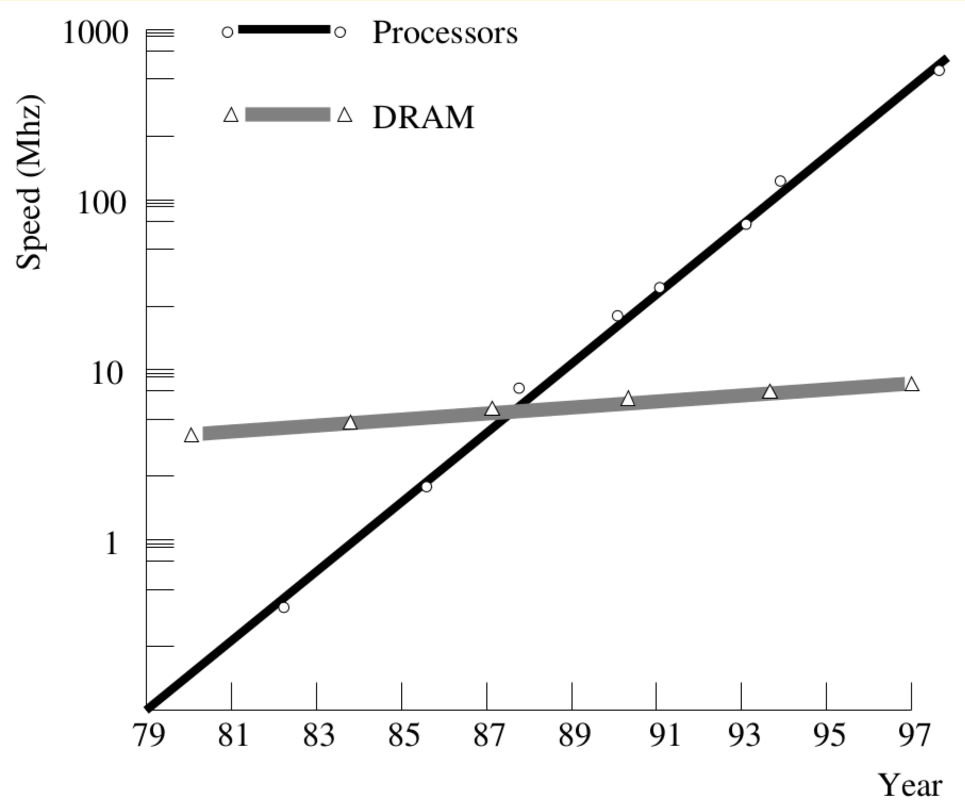
Disk-based vs. Main-Memory DBMS (4)



Disk bottleneck is removed as database is kept in main memory

→ Access to main memory becomes new bottleneck

The New Bottleneck: Memory Access



Accessing main-memory is much more expensive than accessing CPU registers.

→ Is main-memory the new disk?

Picture taken from [Manegold et al., 2000]

New bottleneck

- When I/O is no longer the bottleneck…
  - Locking/latching
  - Cache-line misses
  - Data movement

Rethink the Architecture of DBMSs

Even if the complete database fits in main memory, there are significant overheads of traditional DBMSs:

- Many function calls → stack manipulation overhead + instruction-cache misses
- Adverse memory access → data-cache misses

→ Be aware of the caches!

Cache awareness

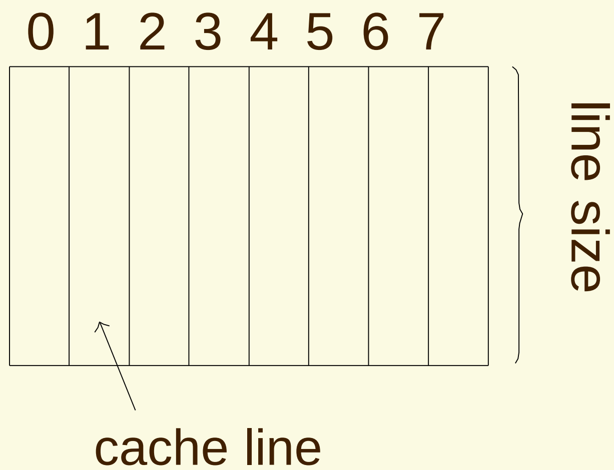
Principle of Locality

- ✓ Caches take advantage of the principle of locality.
  - The hot set of data often fits into caches.
  - 90 % execution time spent in 10 % of the code.
- ✓ Spatial Locality:
  - Related data is often spatially close.
  - Code often contains loops.
- ✓ Temporal Locality:
  - Programs tend to re-use data frequently.
  - Code may call a function repeatedly, even if it is not spatially close.

CPU Cache Internals

To guarantee speed, the overhead of caching must be kept reasonable.

- Organize cache in **cache lines**.
- Only load/evict **full cache lines**.
- Typical **cache line size**: 64 bytes.



The organization in cache lines is consistent with the principle of (spatial) locality.

Memory Access

On every memory access, the CPU checks if the respective cache line is already cached.

Cache Hit:

- Read data directly from the cache.
- No need to access lower-level memory.

Cache Miss:

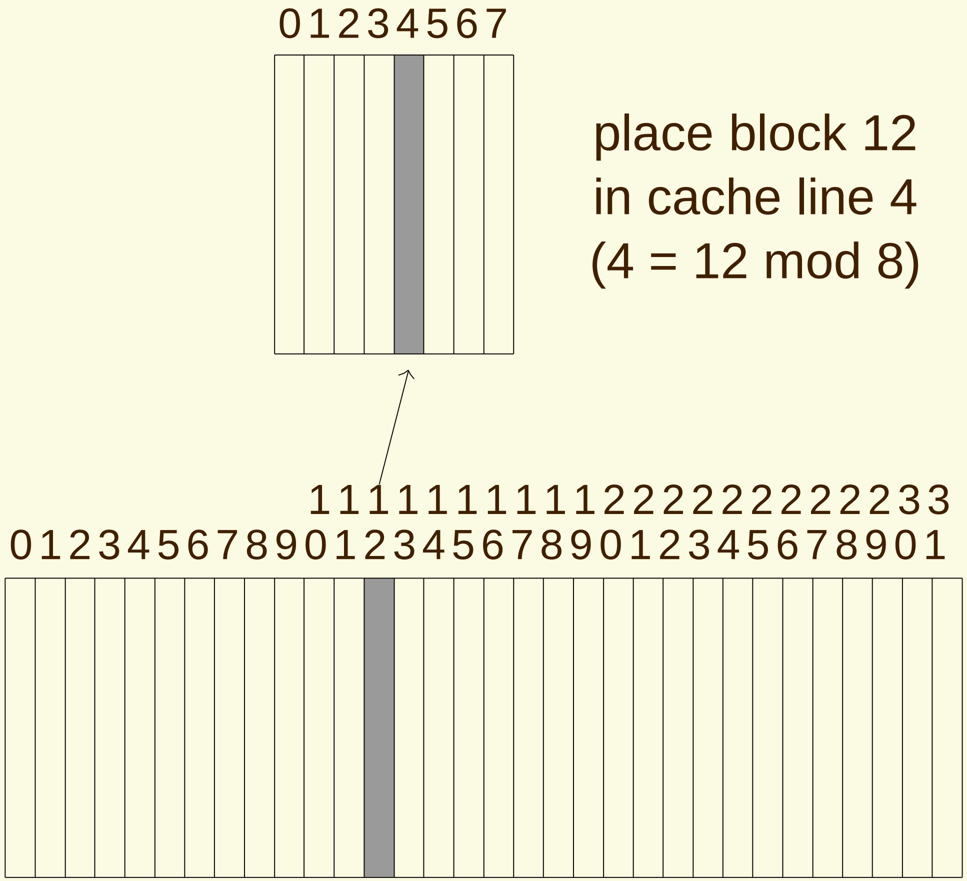
- Read full cache line from lower-level memory.
- Evict some cached block and replace it by the newly read cache line.
- CPU stalls until data becomes available.

Modern CPUs support out-of-order execution and several in-flight cache misses.

Block Placement: Direct-Mapped Cache

In a direct-mapped cache, a block has only one place it can appear in the cache.

- Much simpler to implement.
- Easier to make fast.
- Increases the chance of conflicts.



place block 12 in cache line 4 (4 = 12 mod 8)

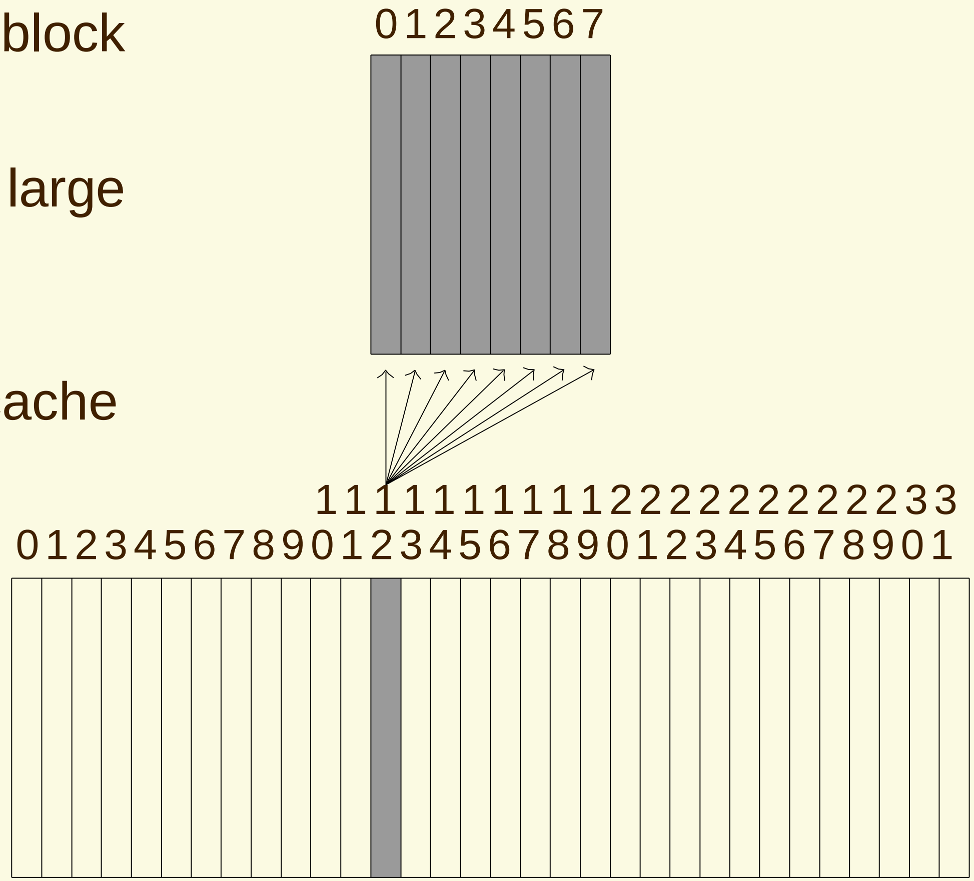
Block Placement: Fully Associative Cache

In a fully associative cache, a block can be loaded into any cache line

- Provide freedom to block replacement strategy.
- Does not scale to large caches

→ 4 MB cache,

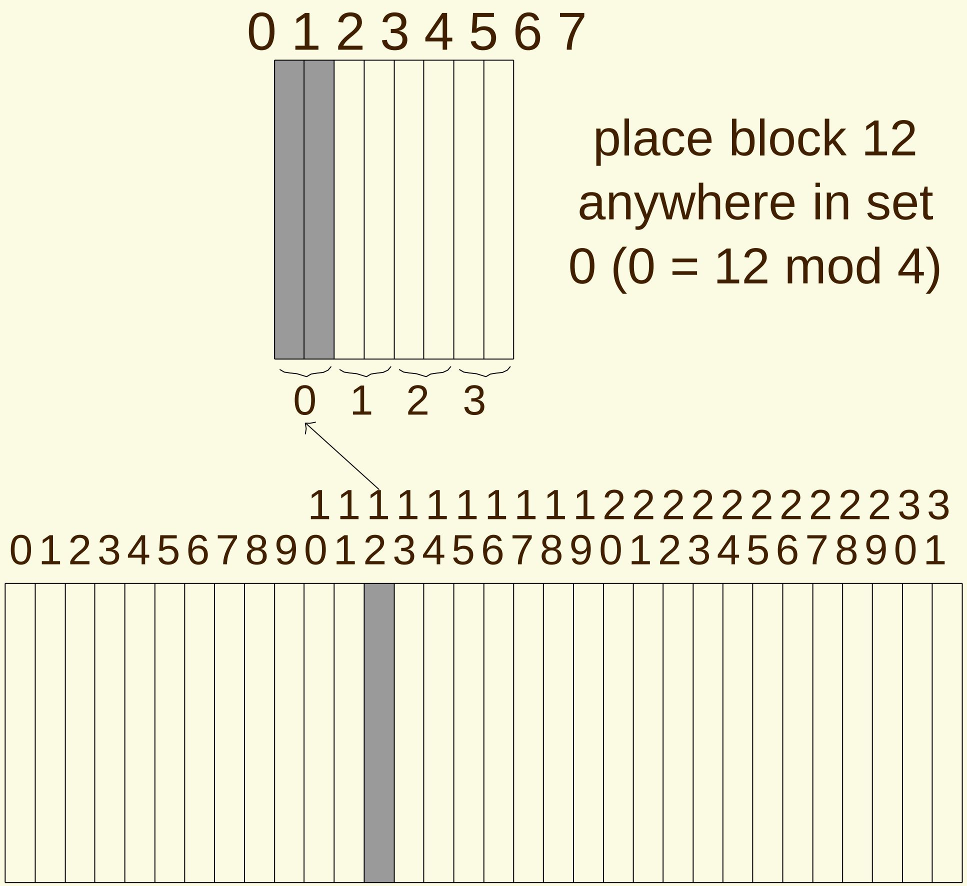
line size: 64 B: 65,536 cache lines.



Block Placement: Set-Associative Cache

A compromise are set-associative caches.

- Group cache lines into sets.
- Each memory block maps to one set.
- Block can be placed anywhere within a set.
- Most processor caches today are set-associative.



Block Replacement

When bringing in new cache lines, an existing entry has to be evicted:

Least Recently Used (LRU)

- Evict cache line whose last access is longest ago.

→ Least likely to be needed any time soon.

First In First Out (FIFO)

- Behaves often similar like LRU.
- But easier to implement.

Random

- Pick a random cache line to evict.
- Very simple to implement in hardware.

Replacement has to be decided in hardware and fast.

What Happens on a Write?

To implement memory writes, CPU makers have two options:
Write Through

• Data is directly written to lower-level memory (and to the cache).
→ Writes will stall the CPU.
→ Greatly simplifies data coherency.

Write Back

• Data is only written into the cache.

• A dirty flag marks modified cache lines (Remember the status field.)
→ May reduce traffic to lower-level memory.
→ Need to write on eviction of dirty cache lines.

Modern processors usually implement write back.

Putting it all Together

To compensate for slow memory, systems use caches.

- Typically multiple levels of caching (memory hierarchy).
- Caches are organized into cache lines.
- **Set associativity:** A memory block can only go into a small number of cache lines (most caches are set-associative).

In-memory DBMS will benefit from locality of data and code.

Processing models

Processing Models

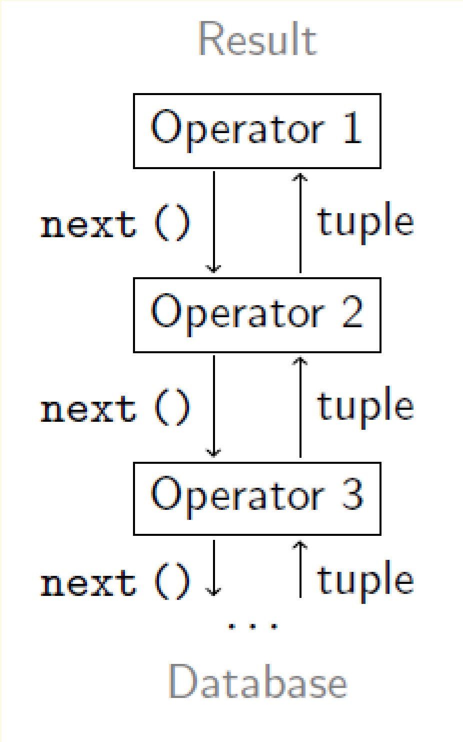
There are basically two alternative processing models that are used in modern DBMSs:

- Tuple-at-a-time volcano model [Graefe, 1990]
  - Operator requests next tuple, processes it, and passes it to the next operator
- Operator-at-a-time bulk processing [Manegold et al., 2009]
  - Operator consumes its input and materializes its output

Tuple-At-A-Time Processing

Most systems implement the Volcano iterator model:

- Operators request tuples from their input using next ().
- Data is processed tuple at a time.
- Each operator keeps its own state.

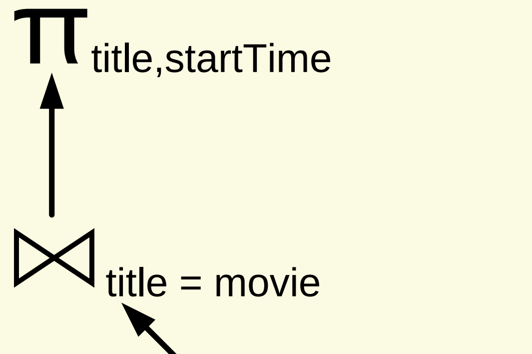


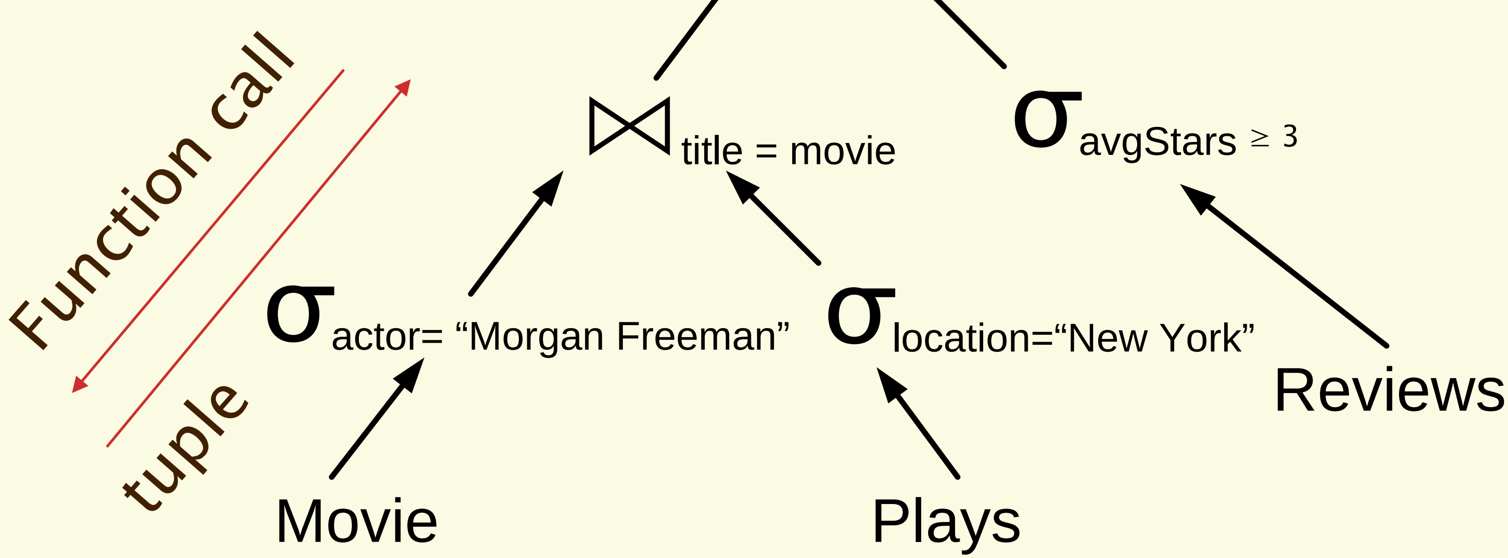
Example Logical Query Plan (revisit Lecture 7)

SELECT title, startTime

FROM Movie M, Plays P, Reviews R

WHERE M.title = P.movie AND M.title = R.movie AND P.location ="New York " AND M.actor = “Morgan Freeman” AND R.avgStars >= 3





Tuple-At-A-Time Processing - Consequences

- Pipeline-parallelism

→ Data processing can start although data does not fully reside in main memory

→ **Small intermediate results**

- All operators in a plan run **tightly interleaved**.

→ Their **combined** instruction footprint may be large.

→ **Instruction cache misses**.

- Operators constantly call each other's functionality.

→ Large **function call overhead**.

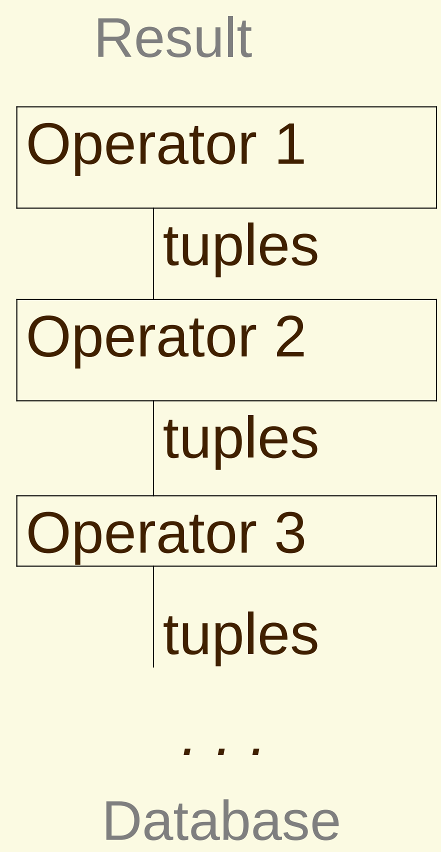
- The combined **state** may be too large to fit into caches.
  - *E.g.*, hash tables, cursors, partial aggregates.

→ **Data cache misses**.

Not a good option for in-memory DBMS (especially OLAP DBMS)

Operator-At-A-Time Processing

- Operators consume and produce **full tables**.
- Each (sub-)result is **fully materialized** (in memory).
- **No** pipelining (rather a sequence of statements).
- Each operator runs exactly once.

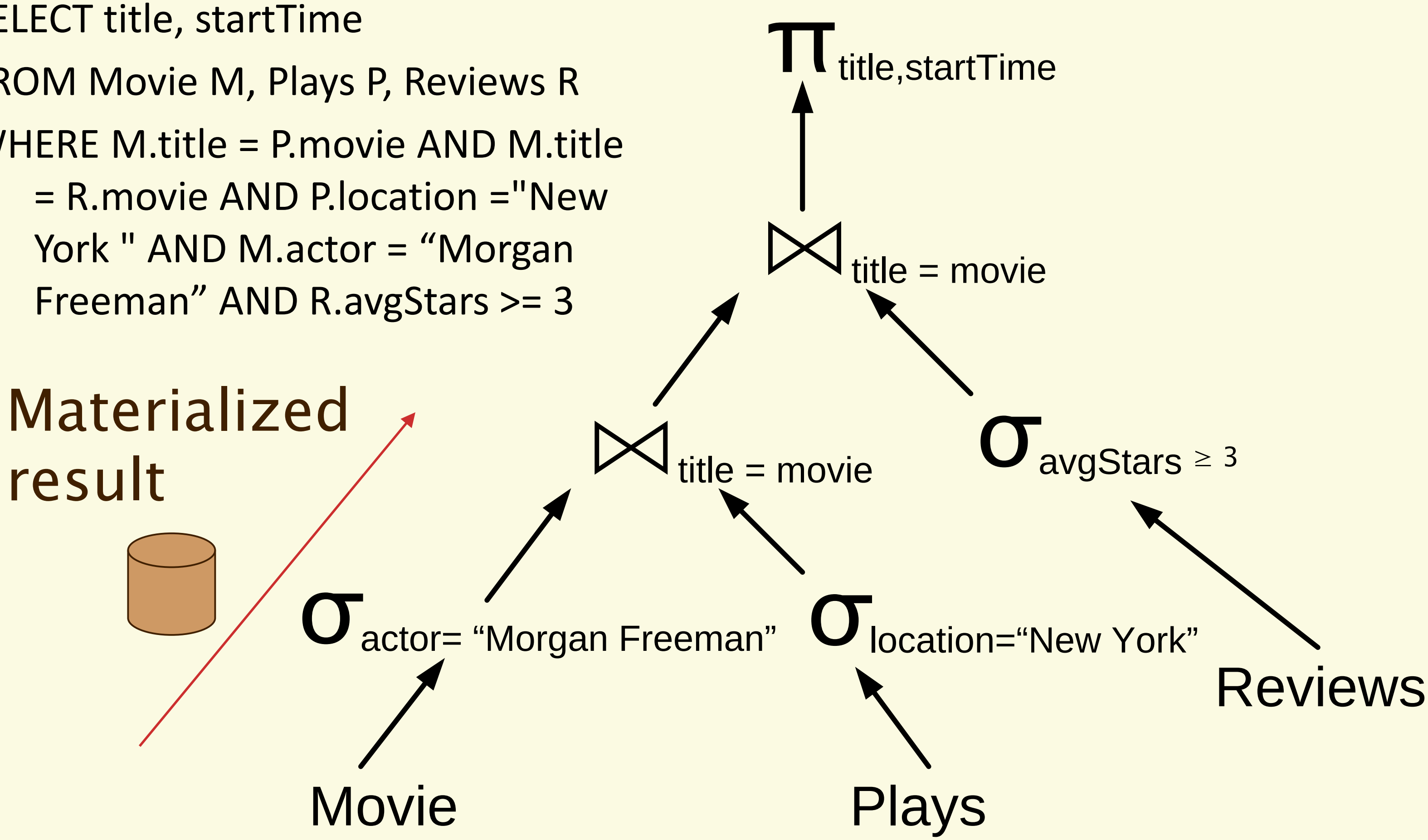


Example Logical Query Plan (revisit Lecture 7)

SELECT title, startTime

FROM Movie M, Plays P, Reviews R

WHERE M.title = P.movie AND M.title = R.movie AND P.location ="New York " AND M.actor = “Morgan Freeman” AND R.avgStars >= 3



Operator-At-A-Time Consequences

- Parallelism: **Inter-operator** and **intra-operator**
- Function call overhead is now replaced by **extremely tight loops** that
  - conveniently **fit into instruction caches**,
  - can be **optimized** effectively by modern compilers
- Function calls are now **out of the critical code path**.
- **No** per-tuple field extraction or type resolution.
  - **Operator specialization**, *e.g.*, for every possible type.
  - Implemented using **macro expansion**.
  - Possible due to column-based storage.

Implemented in H-store and VoltDB

- Fine for OLTP.
- What about OLAP?

Vectorized Execution Model

Idea:

•Use Volcano-style iteration,

but:

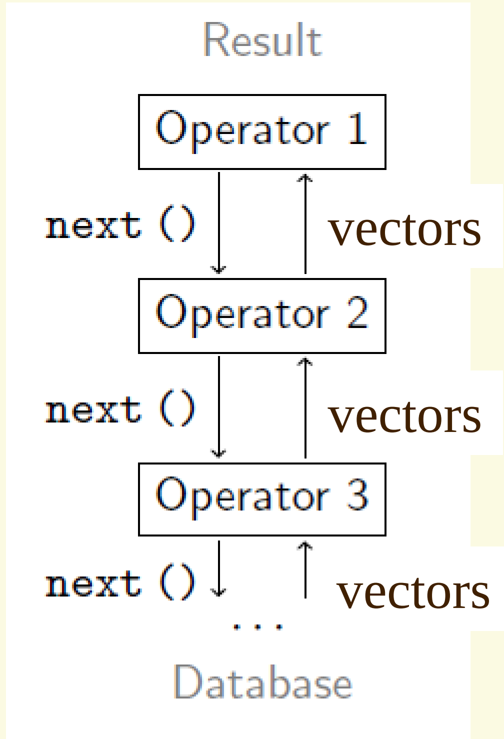
•for each next () call **return a large number of tuples**

→ a so called “vector”

Choose vector size

•**large enough** to compensate for iteration overhead (function calls, instruction cache misses, . . .), but

•**small enough** to not thrash data caches.

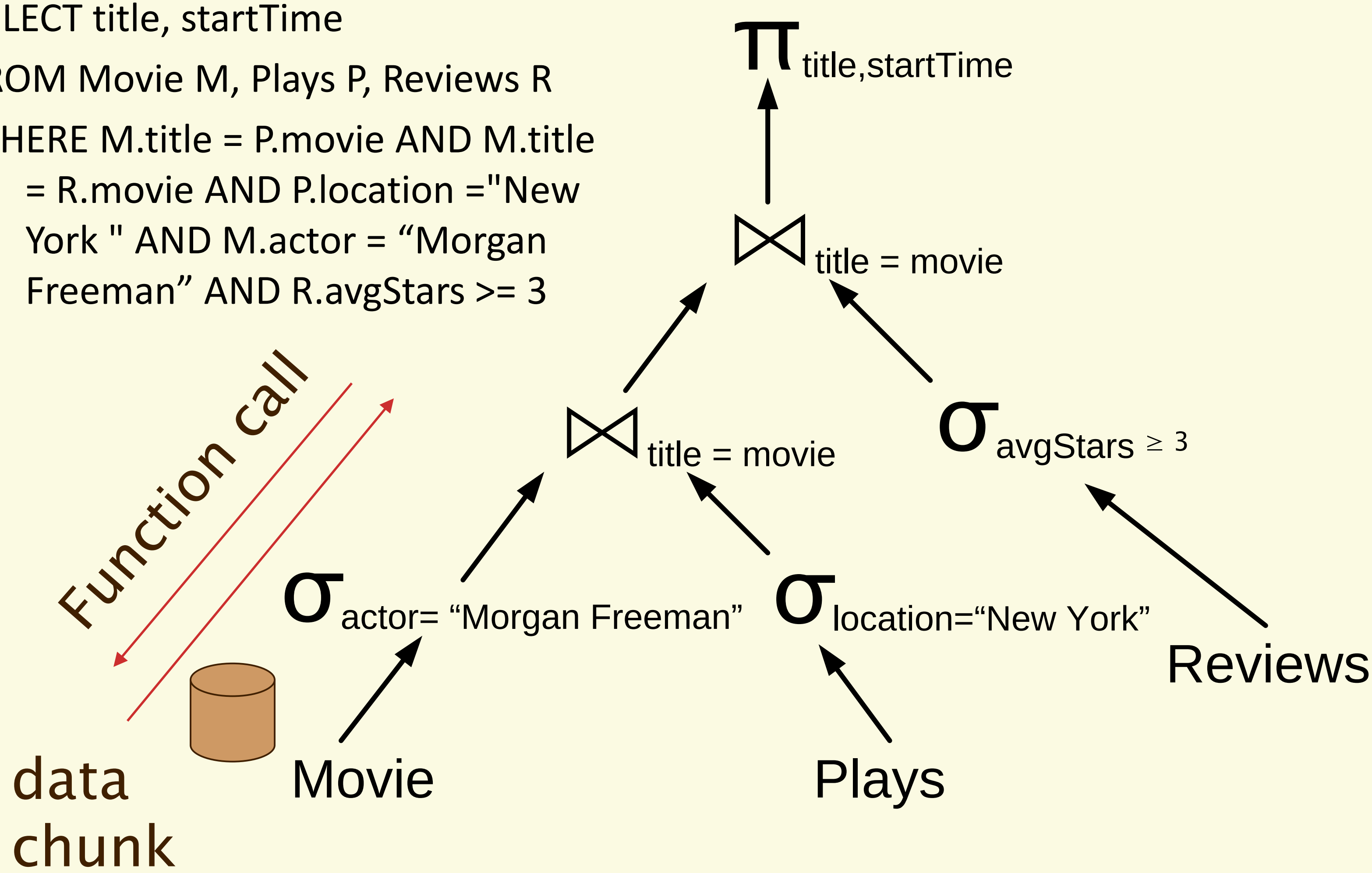


Example Logical Query Plan (revisit Lecture 7)

SELECT title, startTime

FROM Movie M, Plays P, Reviews R

WHERE M.title = P.movie AND M.title = R.movie AND P.location ="New York " AND M.actor = “Morgan Freeman” AND R.avgStars >= 3



What if larger-than-memory?

- ✓ Hybrid workload:
  - OLAP + OLTP
  - Small, frequently updated: “Hot Data” -- OLTP
    - News, social activities, posts, fresh data, fast data
    - Main Memory
  - Large, infrequent updated but support analytical queries: “Cold Data” – OLAP
    - SSD, Hard disk
  - A comparison with Disk-based systems. Hot vs. Cold.

A vision

- ✓ Non-volatile memory – storage level memory
  - Same read/write speed as DRAM
  - Persistent guarantee of SSD
- ✓ High-speed DRAM networks & Systems-on-a-Chip
  - Game changer for parallel/distributed algorithm design
- ✓ In-memory Data Analytics Systems
  - Big Data in your laptop!

Conclusion

- Overhead of Disk-based DBMS
  - Buffer pool
  - Concurrency control
  - Locking/latching

- In-memory DBMS
  - Data organization
  - Cache awareness
  - Query processing models

- What we haven't talked: Indexing? (T-trees: read: http://www.vldb.org/conf/1986/P294.PDF)