Big Data

Big Data

Beyond Relational Data
noSQL databases

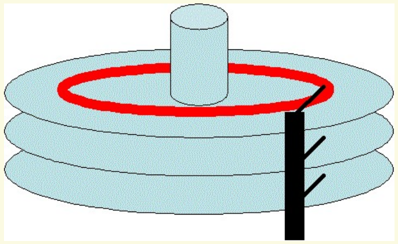
- ✓ Column store
- ✓ In-memory DBMS

Column store

Row Store and Column Store

- Most of the queries does not process all the attributes of a particular relation.

- For example the query
  - Select c.name and c.address
  - From CUSTOMES as c
  - Where c.region=Mumbai;

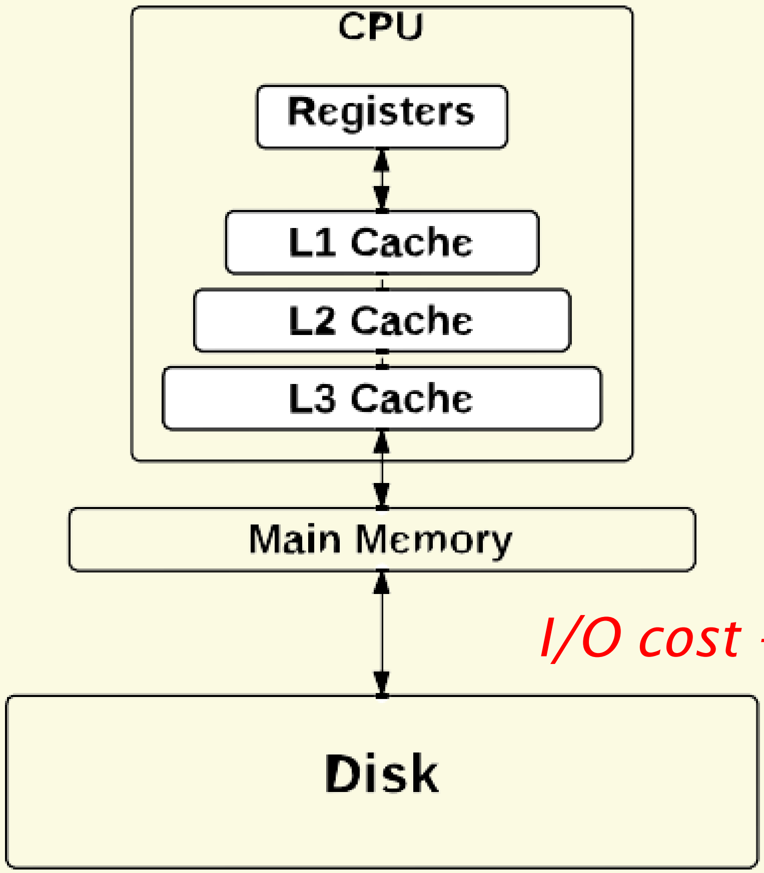


- Only process three attributes of the relation CUSTOMER. But the customer relation can have more than three attributes.

- Column-stores are more I/O efficient for read-only queries as they read, only those attributes which are accessed by a query.

YW1

Recall Computer Architecture



latency	capacity
300 ps	1000 B
1 ns	64 kB
3 - 10 ns	256 kB
10 - 20 ns	2 - 4 MB
50 - 100 ns	4 - 16 GB
5.000.000 - 10.000.000 ns	4 - 16 TB

Cache cost – In-memory store

I/O cost – column store

Data taken from [Hennessy and Patterson, 2012]

Slide 5

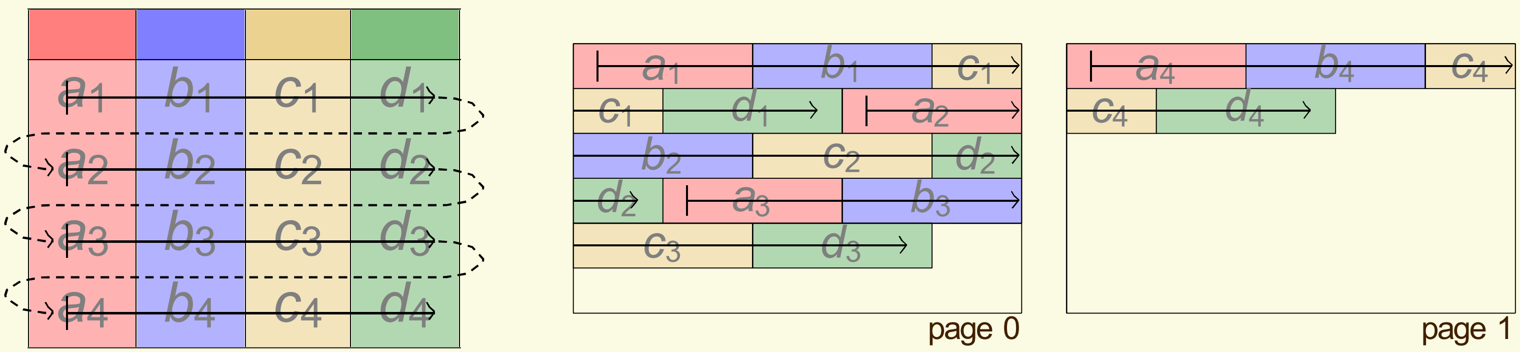
YW1 Yinghui Wu, 9/15/2016

Row Store and Column Store

- ✓ In row store data are stored in the disk tuple by tuple.
- ✓ Where in column store data are stored in the disk column by column

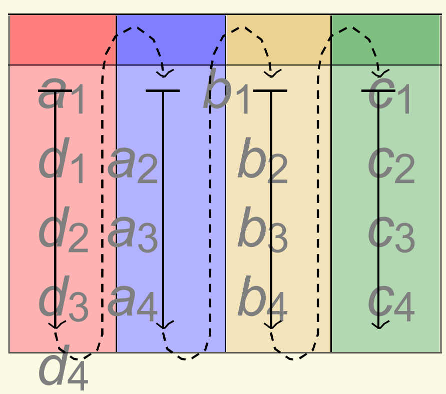
Row-stores

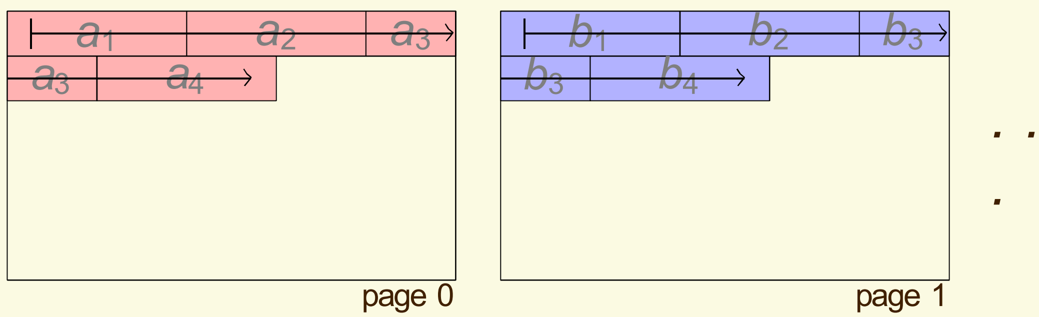
In a row-store, a.k.a. row-wise storage or n-ary storage model, NSM:
all rows of a table are stored sequentially on a database page.



Column-stores

a.k.a. column-wise storage or decomposition storage model, DSM:





The effect on query processing

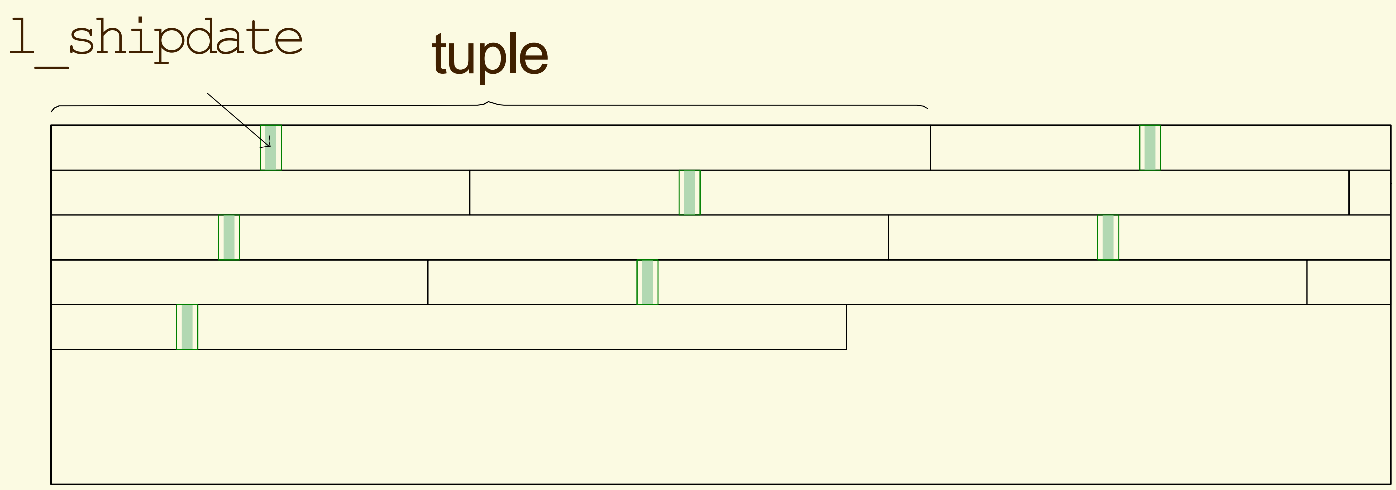
Consider, e.g., a selection query:

```
SELECT COUNT(*)
FROM lineitem
WHERE l_shipdate = "2016-01-25"
```

This query typically involves a full table scan.

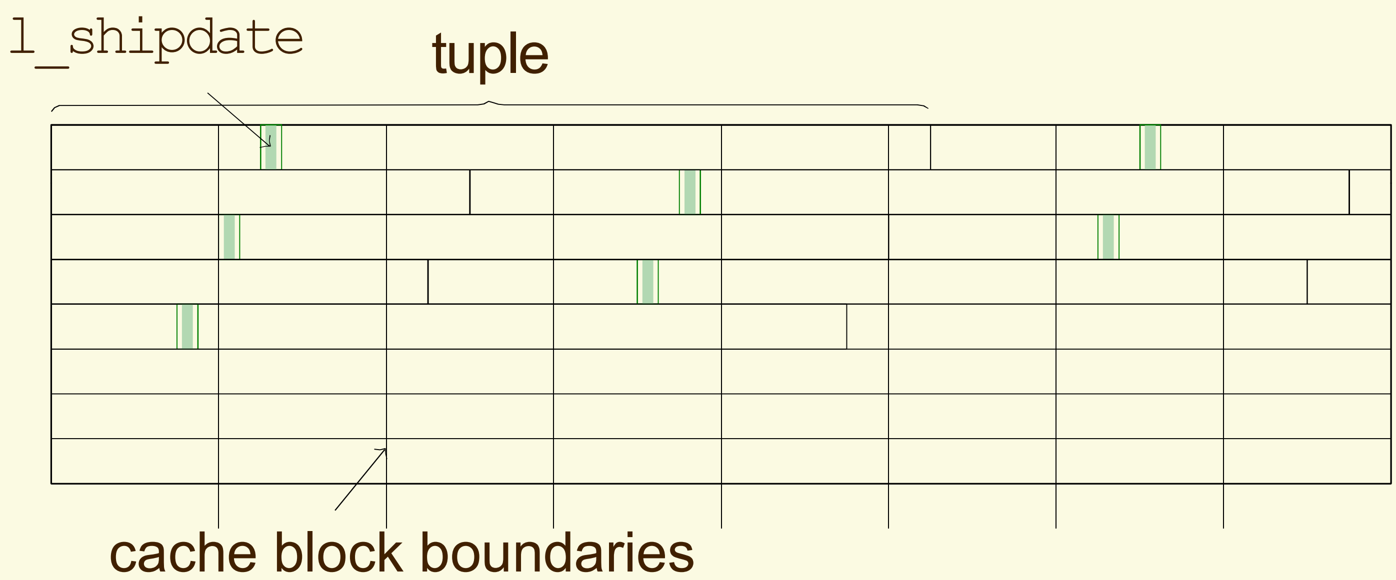
A full table scan in a row-store

In a row-store, all rows of a table are stored sequentially on a database page.



A full table scan in a row-store

In a row-store, all rows of a table are stored sequentially on a database page.

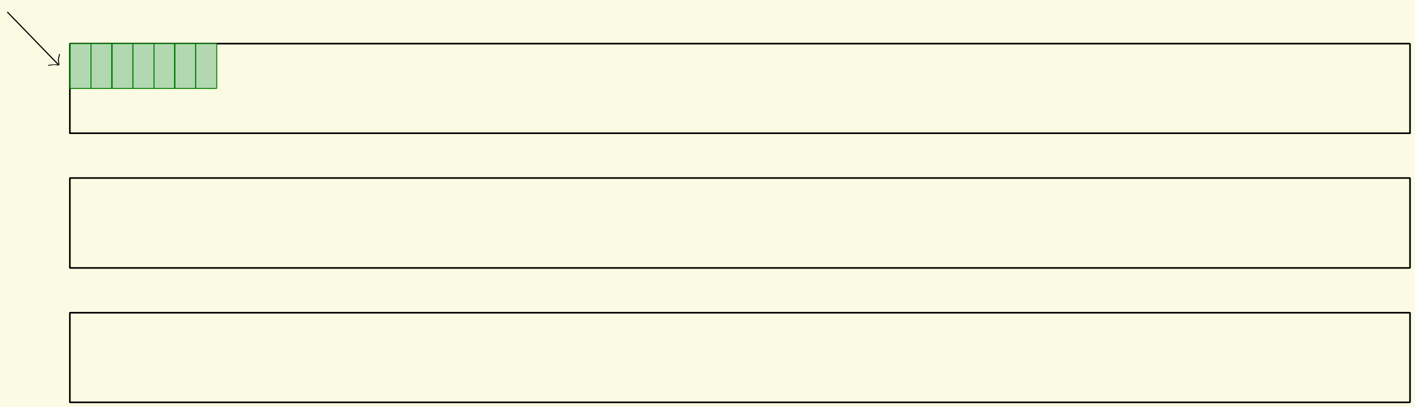


With every access to a l_shipdate field, we load a large amount of irrelevant information into the cache.

A ”full table scan” on a column-store

In a column-store, all values of one column are stored sequentially on a database page.

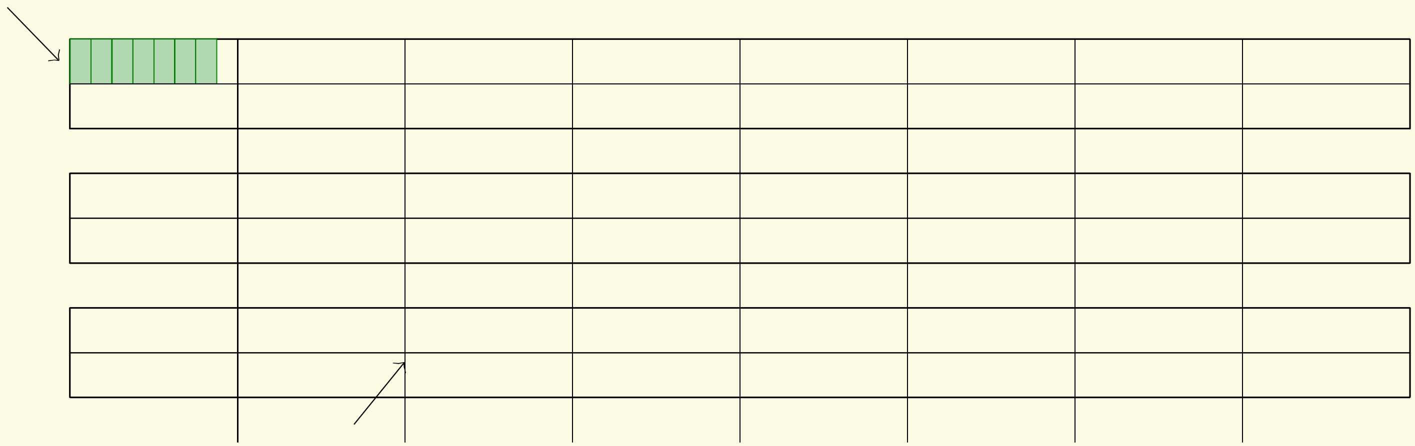
`l_shipdate(s)`



A "full table scan" on a column-store

In a column-store, all values of one column are stored sequentially on a database page.

`l_shipdate(s)`



cache block boundaries

All data loaded into caches by a "`l_shipdate` scan" is now actually **relevant** for the query.

Column-store advantages

- ✓ Less data has to be fetched from memory.
- ✓ Amortize cost for fetch over more tuples.
- ✓ If we're really lucky, the full (`l_shipdate`) data might now even fit into caches.
- The same arguments hold also for in-memory based systems (we will see soon).
- Additional benefit: Data compression might work better.

Why Column Stores?

- ✓ Can be significantly faster than row stores for some applications
  - Fetch only required columns for a query
  - Better cache effects
  - Better compression (similar attribute values within a column)
- ✓ But can be slower for other applications
  - OLTP with many row inserts, ..
- ✓ Long war between the column store and row store camps :-)

Row Store and Column Store

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

- ✓ So column stores are suitable for read-mostly, ~~read-intensive,~~ large data repositories

Column store noSQL system

Column Stores - Data Model

- ✓ Standard relational logical data model
  - EMP(name, age, salary, dept)
  - DEPT(dname, floor)
- ✓ Table – collection of projections
- ✓ Projection – set of columns
- ✓ Horizontally partitioned into segments with segment identifier

Column Stores - Data Model

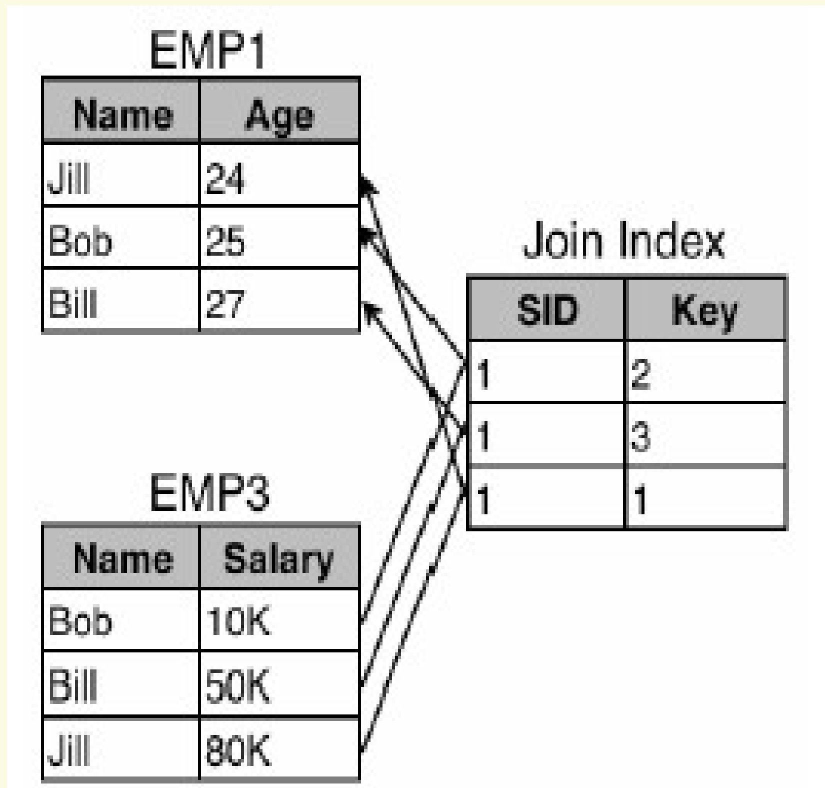
- ✓ To answer queries, projections are joined using Storage keys and join indexes
- ✓ Storage Keys:
  - – Within a segment, every data value of every column is associated with a unique Skey
  - – Values from different columns with matching Skey belong to the same logical row

Column Stores – Data Model

- ✓ Join Indexes
  - T1 and T2 are projections on T
  - M segments in T1 and N segments in T2
  - Join Index from T1 to T2 is a table of the form:
    - (s: Segment ID in T2, k: Storage key in Segment s)
    - Each row in join index matches corresponding row in T1
  - Join indexes are built such that T could be efficiently reconstructed from T1 and T2

Column Stores – Data Model

- ✓ Construct EMP(name, age, salary) from EMP1 and EMP3 using join index on EMP3



Compression

- ✓ Trades I/O for CPU
  - Increased column-store opportunities:
  - Higher data value locality in column stores
  - Data compression techniques such as run length encoding far more useful
- ✓ Schemes
  - Null Suppression
  - Dictionary encoding
  - Run Length encoding
  - Bit-Vector encoding
  - Heavyweight schemes

Query Execution - Operators

- **Select:** Same as relational algebra, but produces a bit string
- **Project:** Same as relational algebra
- **Join:** Joins projections according to predicates

- **Aggregation:** SQL like aggregates
- **Sort:** Sort all columns of a projection
- **Decompress:** Converts compressed column to uncompressed representation
- **Mask**(Bitstring B, Projection Cs) => emit only those values whose corresponding bits are 1
- **Concat:** Combines one or more projections sorted in the same order into a single projection
- **Permute:** Permutes a projection according to the ordering defined by a join index
- **Bitstring operators:** Band – Bitwise AND, Bor – Bitwise OR, Bnot – complement

Row Store Vs Column Store

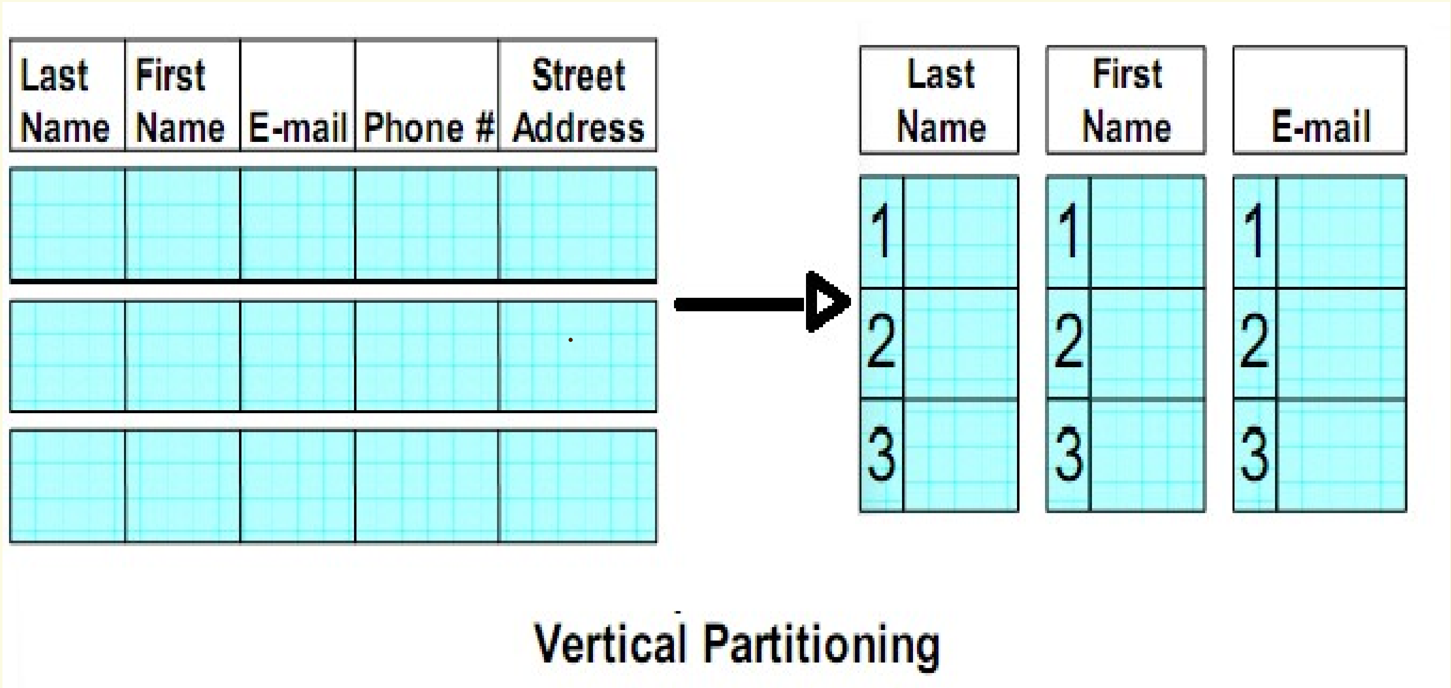
- ✓ the difference in storage layout leads to that one can obtain the performance benefits of a column-store using a row-store by making some changes to the physical structure of the row store.
- ✓ This changes can be
  - Vertically partitioning
  - Using index-only plans
  - Using materialized views

Vertical Partitioning

- ✓ Process:
  - – Full Vertical partitioning of each relation
    - Each column =1 Physical table
    - This can be achieved by adding integer position column to every table
    - Adding integer position is better than adding primary key
  - – Join on Position for multi column fetch

- ✓ Problems:
  - – "Position" - Space and disk bandwidth
  - – Header for every tuple – further space wastage
    - e.g. 24 byte overhead in PostgreSQL

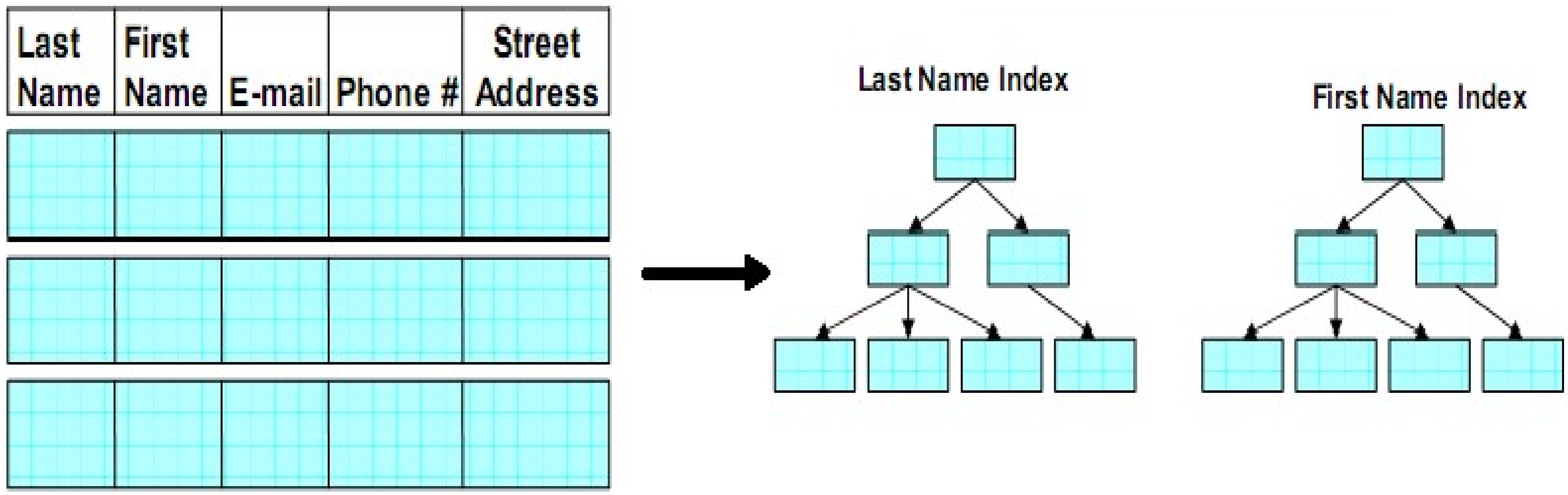
Vertical Partitioning: Example



Index-only plans

- Process:
  - Add B+Tree index for every Table.column
  - Plans never access the actual tuples on disk
  - Headers are not stored, so per tuple overhead is less
- Problem:
  - Separate indices may require full index scan, which is slower
  - Eg: SELECT AVG(salary)
    FROM emp
    WHERE age > 40
  - Composite index with (age, salary) key helps.

Index-only plans: Example



Index Every Column

Materialized Views

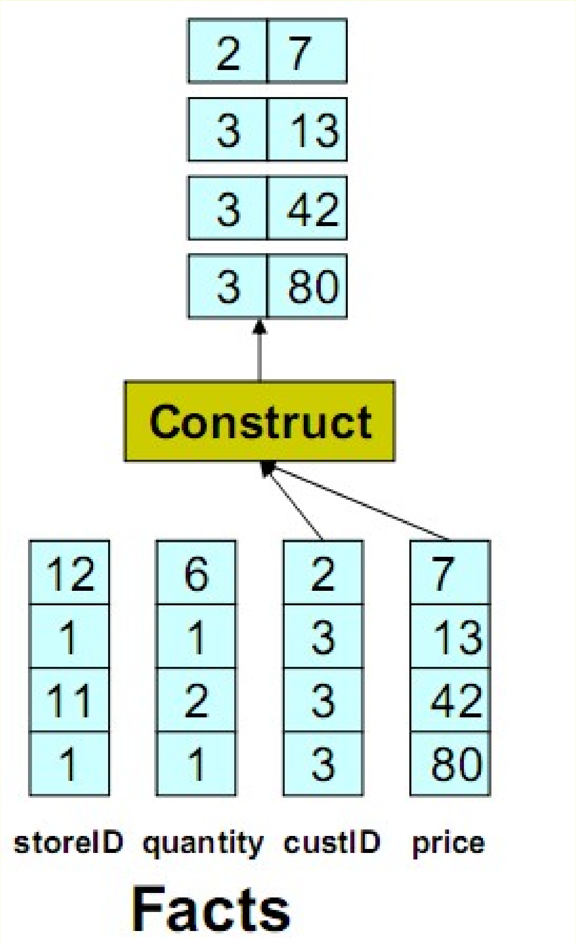
- ✓ Process:
  - – Create ‘optimal' set of MVs for given query workload
  - – Objective:
    - Provide just the required data
    - Avoid overheads
    - Performs better
- ✓ Expected to perform better than other two approach
- ✓ Problems:
  - – Practical only in limited situation
  - – Require knowledge of query workloads in advance

Materialized Views: Example

- ✓ Select F.custID

  from Facts as F

  where F.price>20

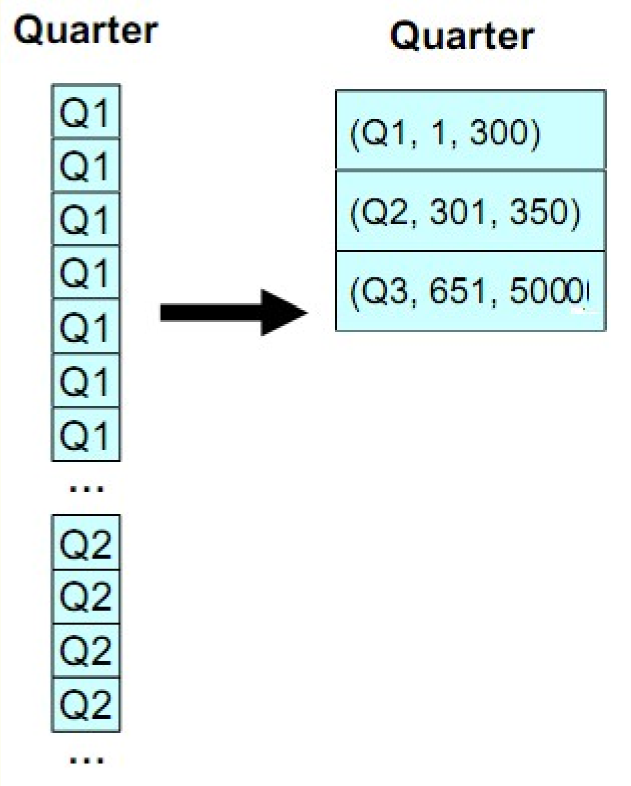


Optimizing Column oriented Execution

- ✓ Different optimization for column oriented database
  - Compression
  - Late Materialization
  - Block Iteration

Compression

- If data is sorted on one column that column will be super-compressible in row store
- eg. Run length encoding



Compression

- ✓ Low information entropy (high data value locality) leads to High compression ratio
- ✓ Advantage
  - Disk Space is saved
  - Less I/O
  - CPU cost decrease if we can perform operation without decompressing
- ✓ Light weight compression schemes do better

Late Materialization

- ✓ Most query results entity-at-a-time not column-at-a-time
- ✓ So at some point of time multiple column must be combined
- ✓ One simple approach is to join the columns relevant for a particular query

  But further performance can be improve using late-materialization

- Idea: Delay Tuple Construction
- Might avoid constructing it altogether
- Intermediate position lists might need to be constructed
- Eg: SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10
  - Output of each predicate is a bit string
  - Perform Bitwise AND
  - Use final position list to extract R.a

Advantages: Unnecessary construction of tuple is avoided
Direct operation on compressed data
Cache performance is improved

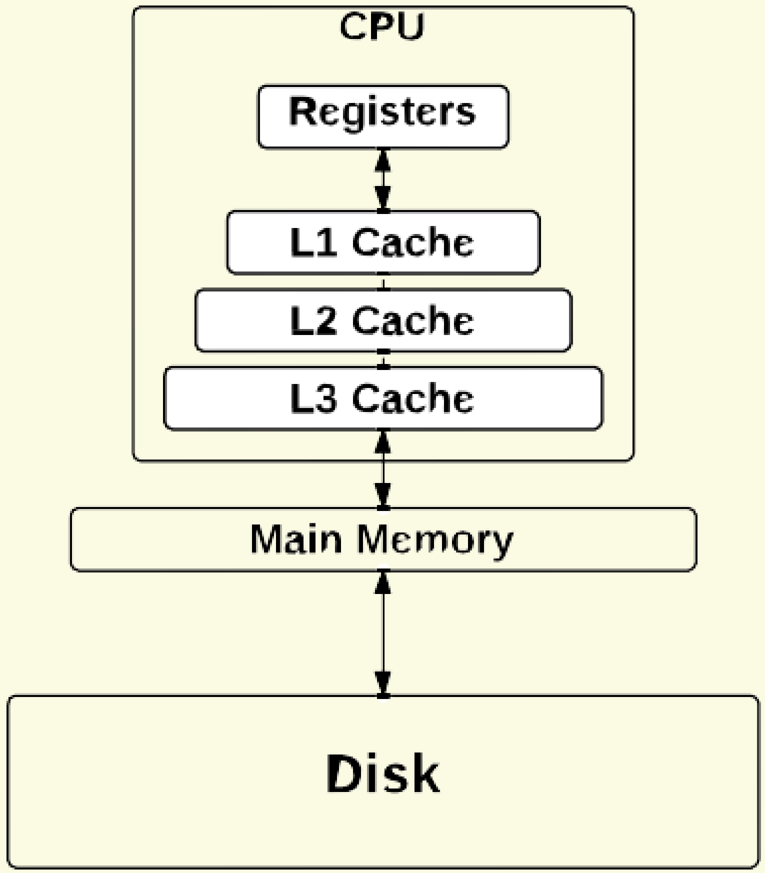
Block Iteration

- ✓ Operators operate on blocks of tuples at once
- ✓ Iterate over blocks rather than tuples
- ✓ Like batch processing
- ✓ If column is fixed width, it can be operated as an array
- ✓ Minimizes per-tuple overhead
- ✓ Exploits potential for parallelism
- ✓ Can be applied even in Row stores – IBM DB2 implements it

In-memory databases

YW1

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

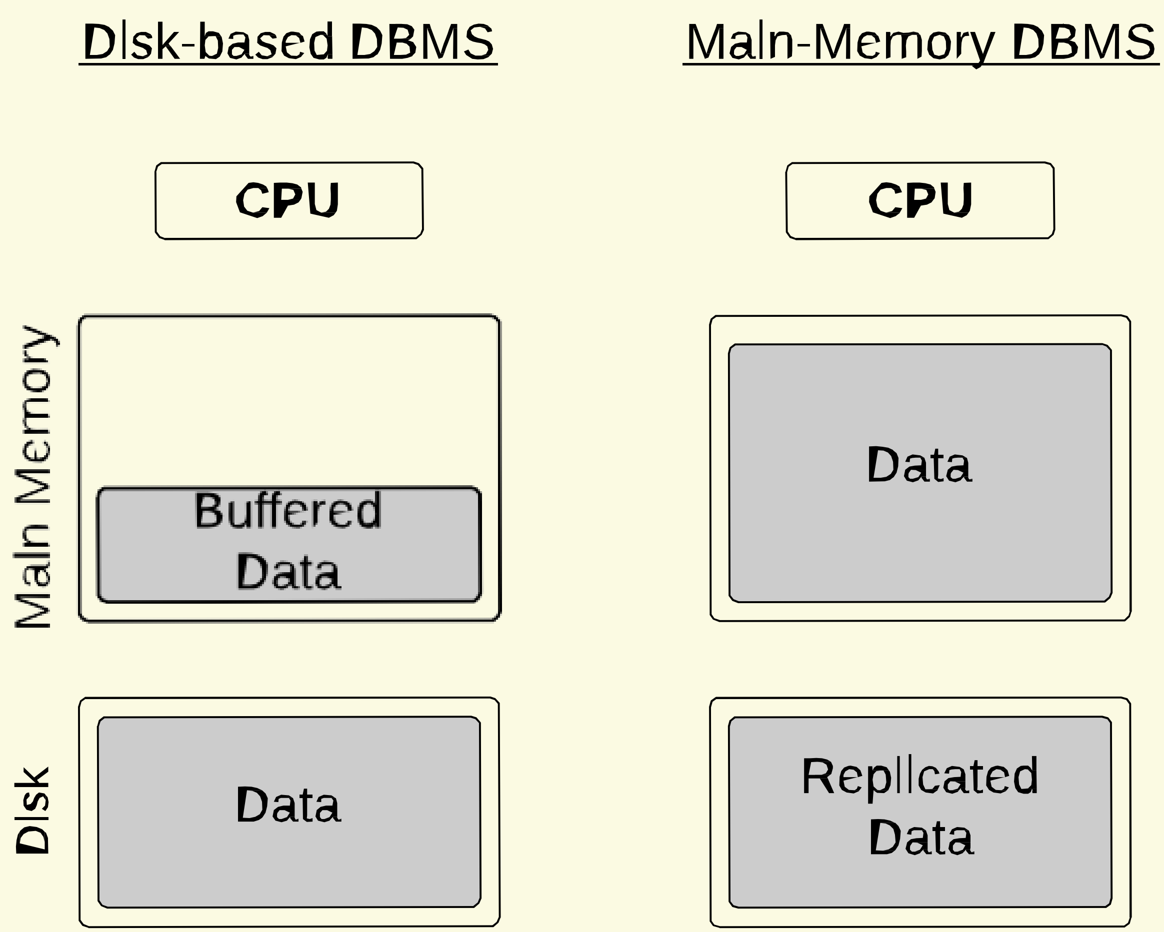
Cache cost – In-memory store

Data taken from [Hennessy and Patterson, 2012]

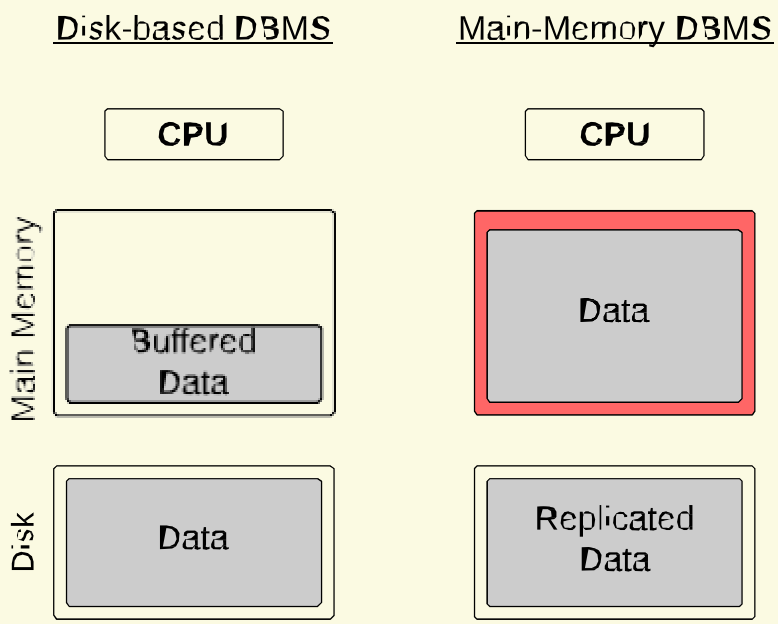
Slide 37

YW1 Yinghui Wu, 9/15/2016

Disk-based vs. Main-Memory DBMS



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

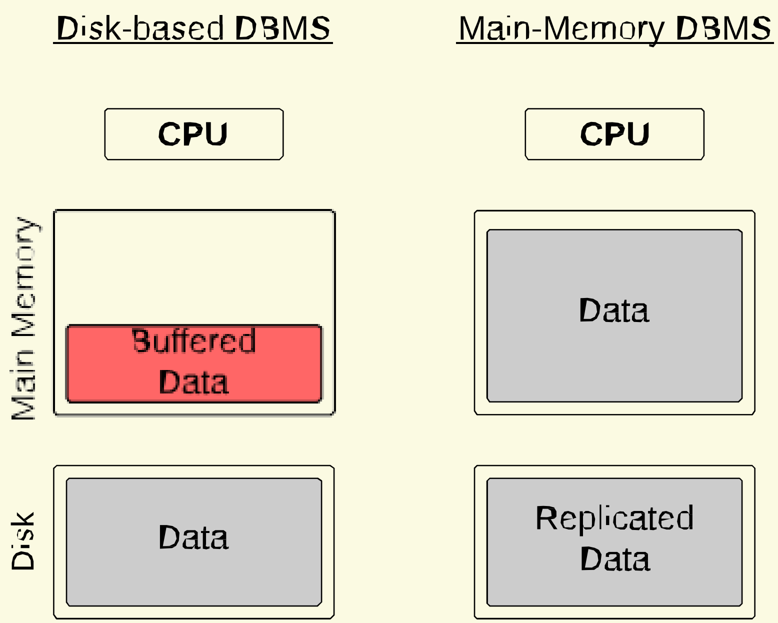


ATTENTION: Main-memory storage != No Durability

→ 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)

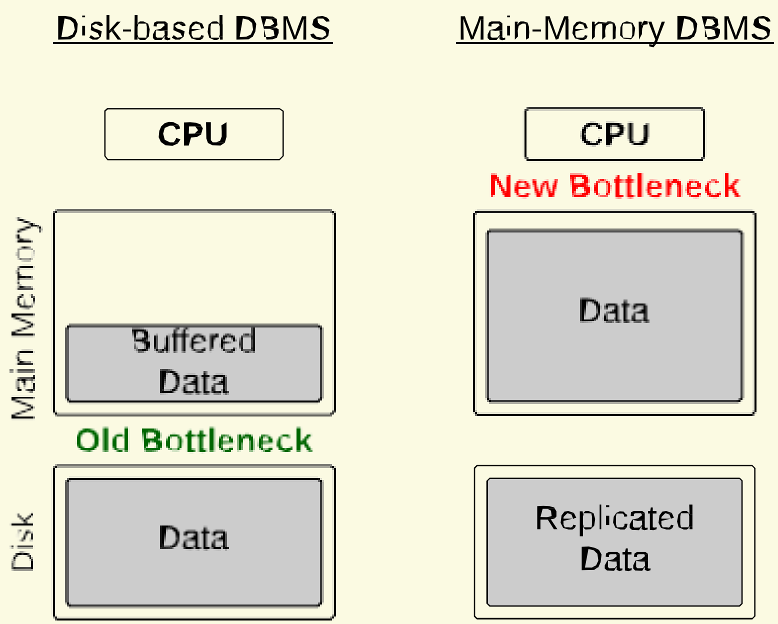


Having the database in main memory allows us to remove buffer manager and paging

→ Remove level of indirection

→ Results in better performance

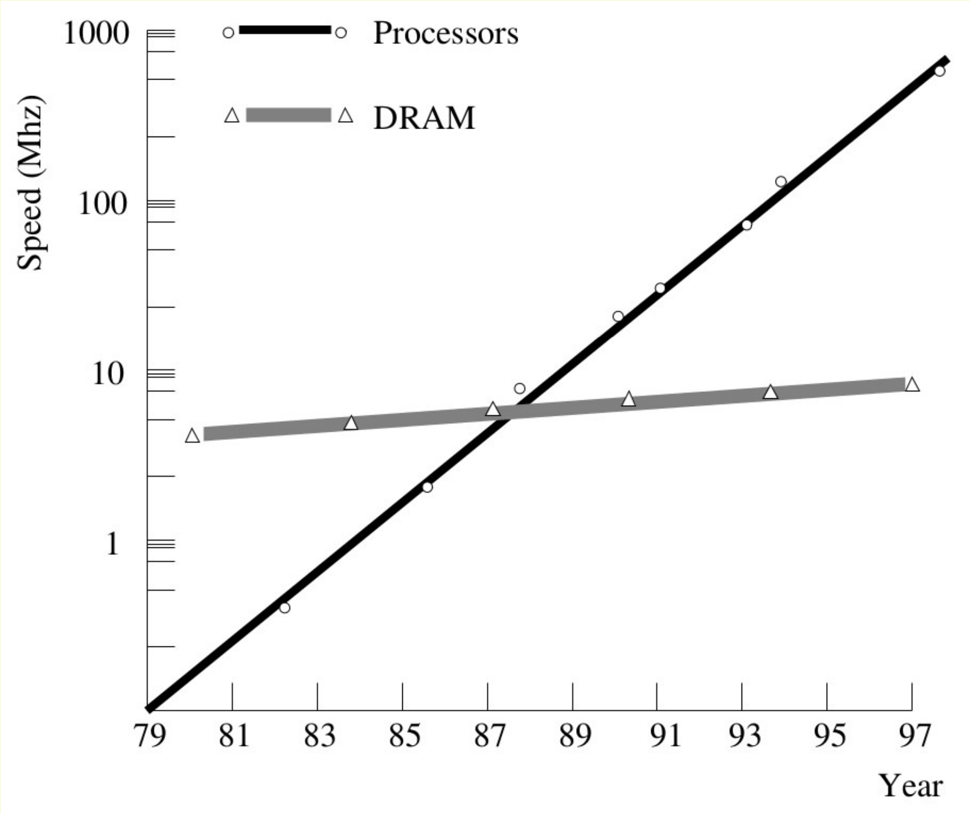
Disk-based vs. Main-Memory DBMS (4)



Disk bottleneck is removed as database is kept in main memory

→ Access to main memory becomes new bottleneck

The New Bottleneck: Memory Access

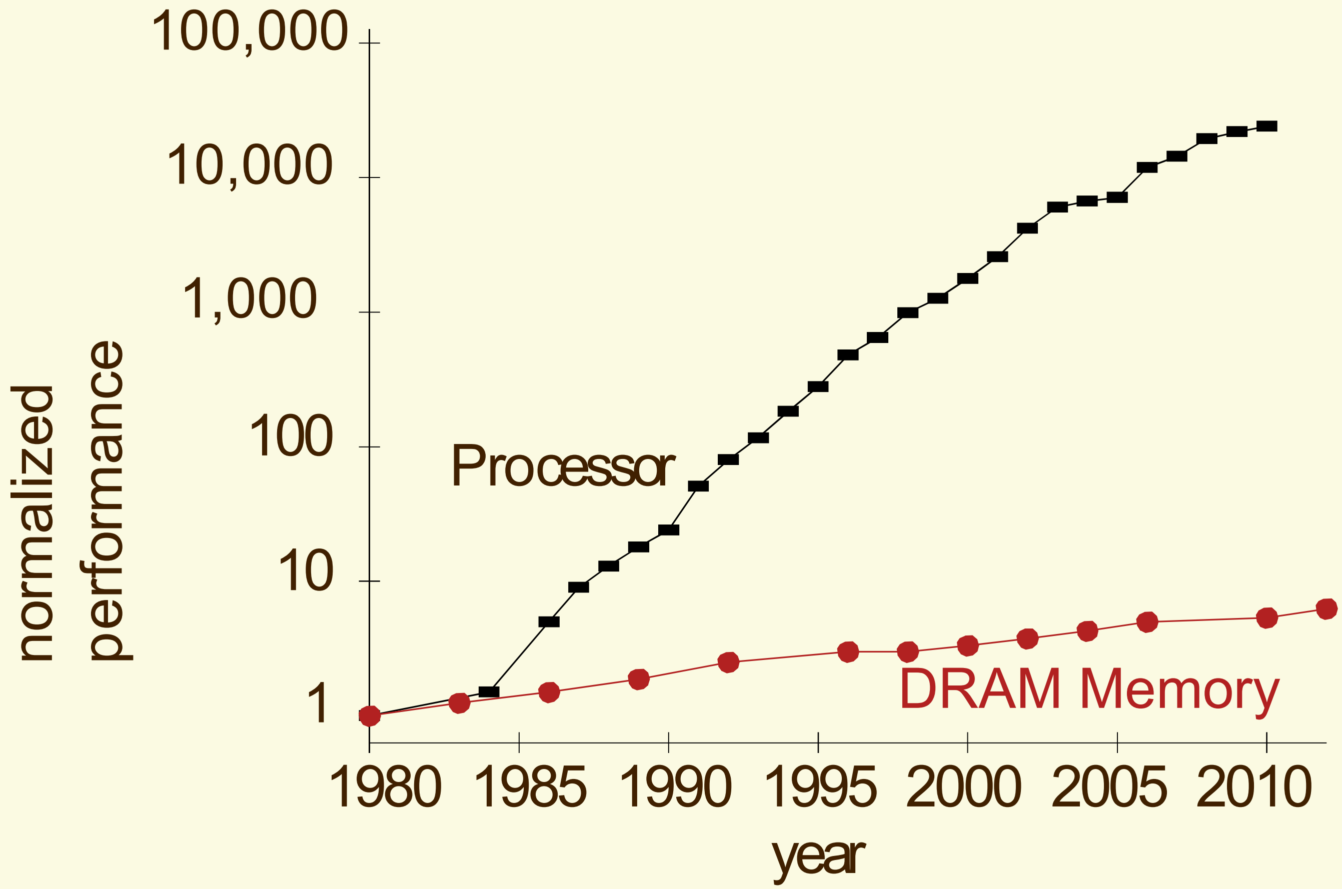


Accessing main-memory is much more expensive than accessing CPU registers.

→ Is main-memory the new disk?

Picture taken from [Manegold et al., 2000]

Memory Wall



Data taken from [Hennessy and Patterson, 2012]

Rethink the Architecture of DBMSs

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

- Many function calls → stack manipulation overhead[1] + instruction-cache misses
- Adverse memory access → data-cache misses

→ Be aware of the caches!

[1] Can be reduced by function inlining

Cache awareness

A Motivating Example (Memory Access)

Task: sum up all entries in a two-dimensional array.

Alternative 1:

```
for (r = 0; r < rows; r++)
for (c = 0; c < cols; c++) sum += src[r * cols +
c];
```

Alternative 2:

```
for (c = 0; c < cols; c++)
for (r = 0; r < rows; r++) sum += src[r * cols +
c];
```

Both alternatives touch the same data, but in different order.

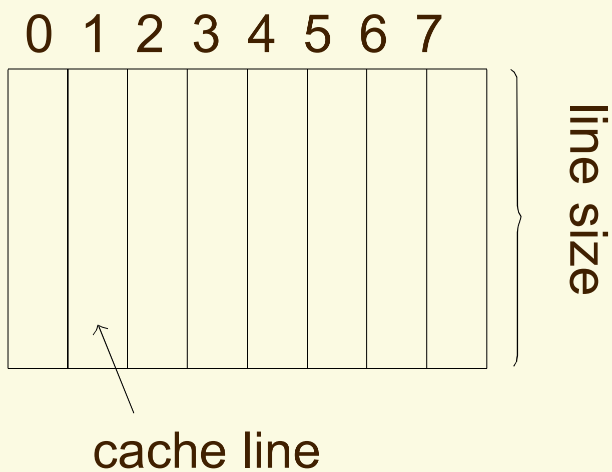
Principle of Locality

- ✓ 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.

Example: AMD Opteron

Data taken from [Hennessy and Patterson, 2006]

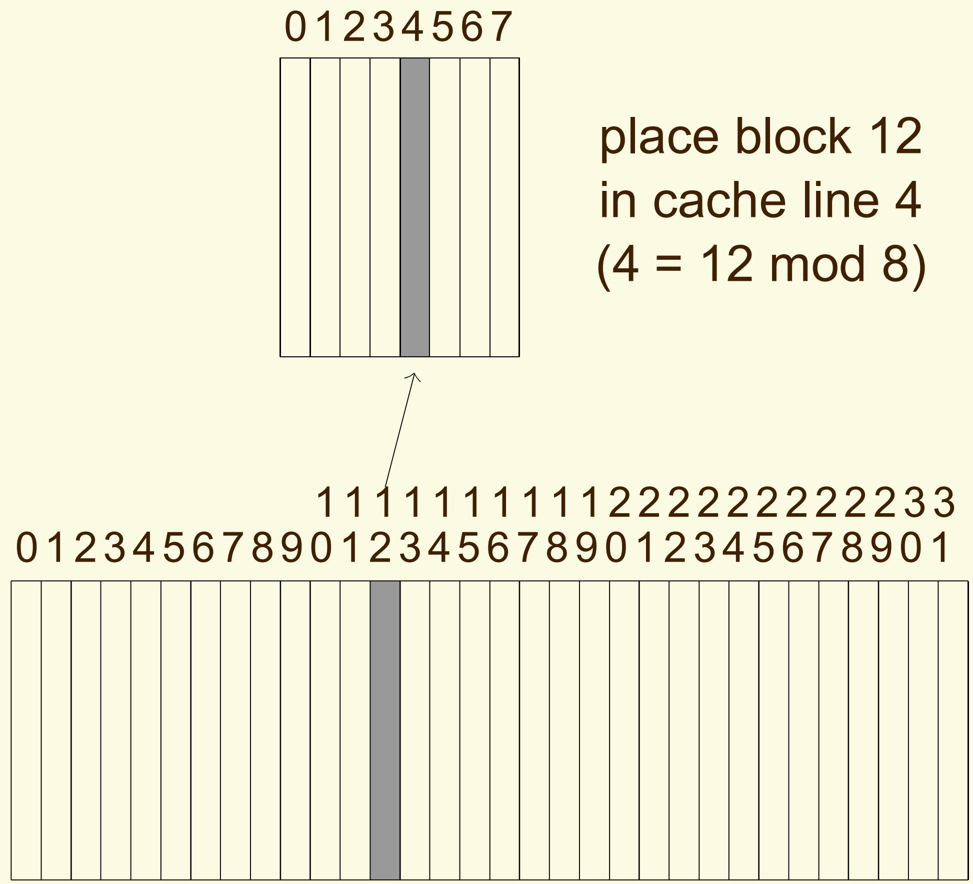
Example: AMD Opteron, 2.8 GHz, PC3200 DDR SDRAM

- **L1 cache:** separate data and instruction caches, each 64 kB, 64 B cache lines
- **L2 cache:** shared cache, 1 MB, 64 B cache lines
- **L1 hit latency:** 2 cycles (≈ 1 ns)
- **L2 hit latency:** 7 cycles (≈ 3.5 ns)
- **L2 miss latency:** 160–180 cycles (≈ 60 ns)

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)

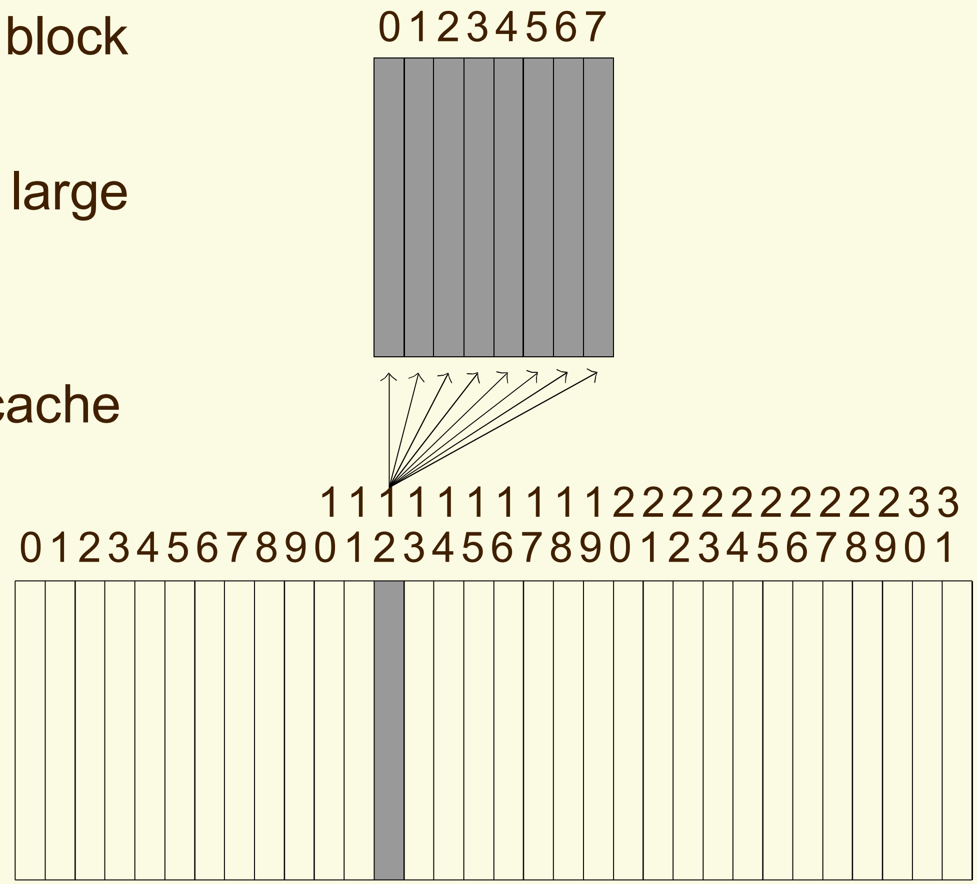
Block Placement: Fully Associative Cache

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,

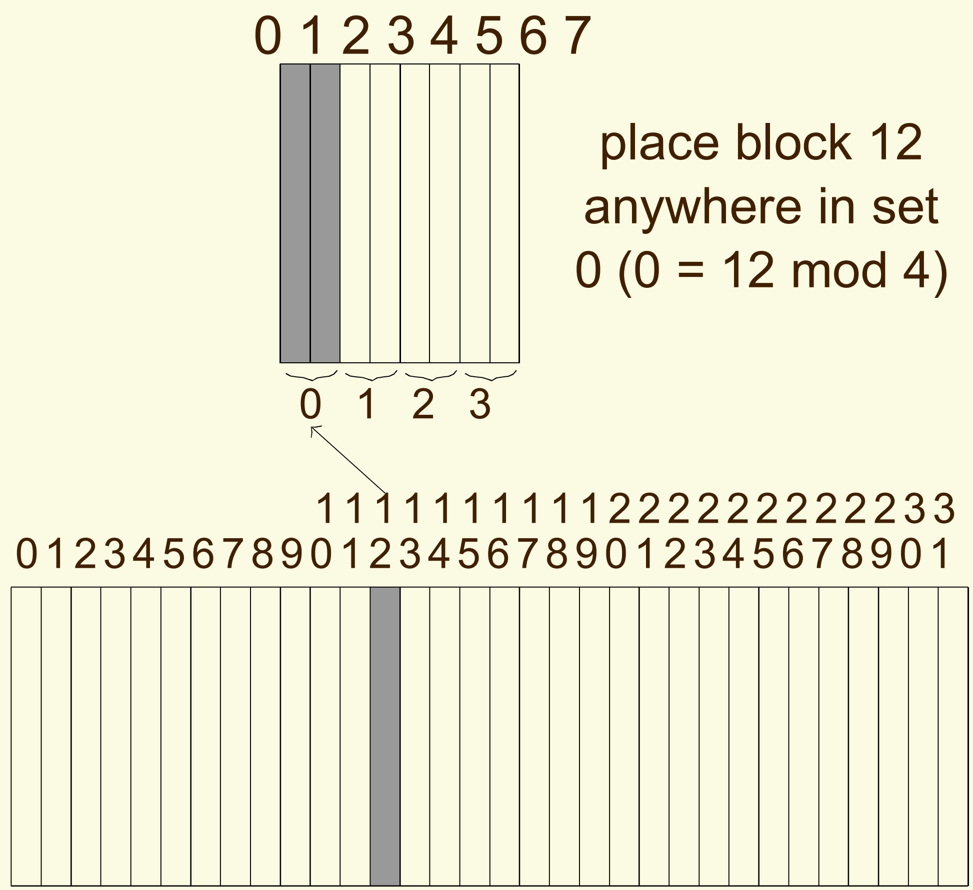
line size: 64 B: 65,536 cache lines.



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.



Example: Intel Q6700 (Core 2 Quad)

- Total cache size: **4 MB** (per 2 cores).
- Cache line size: **64 bytes**.

→ 6-bit offset ($2^6 = 64$)

→ There are 65,536 cache lines in total (4 MB ÷ 64 bytes).

- Associativity: **16-way set-associative**.

→ There are 4,096 sets ($65,536 \div 16 = 4,096$).

→ 12-bit set index ($2^{12} = 4,096$).

- Maximum physical address space: **64 GB**.

→ 36 address bits are enough (2^{36} bytes = 64 GB)

→ 18-bit tags ($36 - 12 - 6 = 18$).

tag	set index	offset
18 bit	12 bit	6 bit

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.

- DRAM provides high capacity, but long latency.
- SRAM has better latency, but low capacity.
- 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).

Systems will benefit from locality of data and code.

Processing models

Processing Models

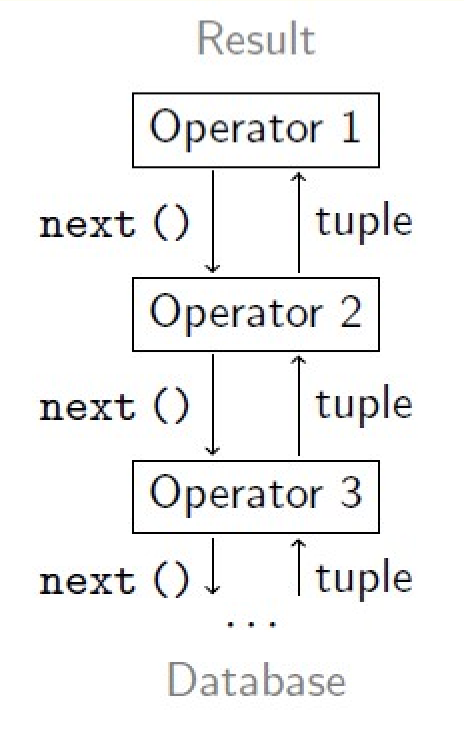
There are basically two alternative processing models that are used in modern DBMSs:

- 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.



select avg(A) from R where A < 100.

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**.

Observations

- Only **single tuple** processed in each call; **millions of calls**.
- Only **10 % of the time** spent on actual query task.
- Low **instructions-per-cycle** (IPC) ratio.

- Much time spent on field access.
  - Polymorphic operators
- Single-tuple functions hard to optimize (by compiler).

→ Low instructions-per-cycle ratio.

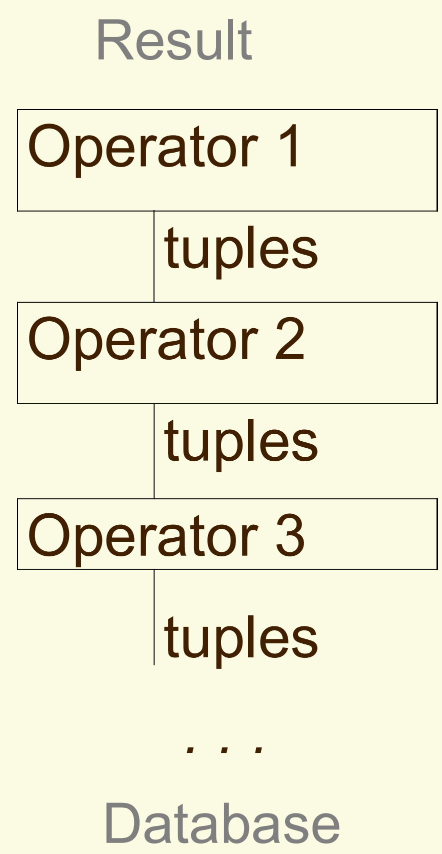
→ Vector instructions (SIMD) hardly applicable.

- Function call overhead
  - $\frac{38\,\text{instr.}}{0.8\,\frac{\text{instr.}}{\text{cycle}}} = 48\,\text{cycles}$ **vs.** 3 instr. for load/add/store assembly[4]

[4] Depends on underlying hardware

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.



select avg(A) from R where A < 100.

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.

Vectorized Execution Model

Idea:

•Use Volcano-style iteration,

but:

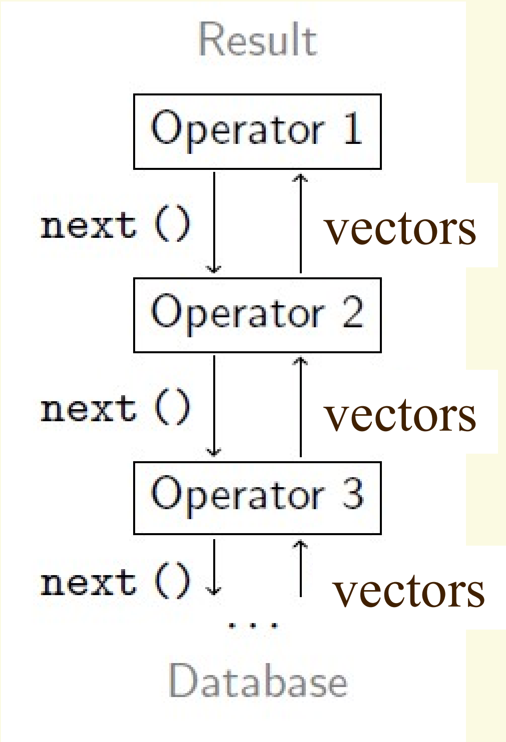
•for each next () call **return a large number of tuples**

→ 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.



Conclusion

- **Column store and in-memory DBMS**
- **Row**-stores store complete tuples sequentially on a database page
- **Column**-stores store all values of one column sequentially on a database page
- Depending on the workload column-stores or row-stores are more advantageous
  - Tuple reconstruction is overhead in column-stores
  - Analytical workloads that process few columns at a time benefit from column-stores

→ One data storage approach is not optimal to serve all possible workloads