<u>A</u>dvanced <u>D</u>ata <u>M</u>anagement for Data Analysis

Stefan Manegold

Data Management @ LIACS

Group leader Database Architectures
Centrum Wiskunde & Informatica (CWI)
Amsterdam

s.manegold@liacs.leidenuniv.nl
http://www.cwi.nl/~manegold/

ADM: Agenda

- 07.09.2022: Lecture 1: **Introduction**
- 14.09.2022: Lecture 2: **SQL Recap**

  (plus Assignment 1 [in groups; 3 weeks]: TPC-H benchmark)
- 21.09.2022: Lecture 3: **Column-Oriented Database Systems (1/6) - Motivation & Basic Concepts**
- 28.09.2022: Lecture 4: **Column-Oriented Database Systems (2a/6) - Selected Execution Techniques (1/2)**
- 05.10.2022: Lecture 5: **Column-Oriented Database Systems (2b/6) - Selected Execution Techniques (2/2)**

  (plus Assignment 2 [in groups; 4 weeks]: Compression techniques)
- 12.10.2022: Lecture 6: **Column-Oriented Database Systems (3/6) - Cache Conscious Joins**
- 19.10.2022: Lecture 7: **Column-Oriented Database Systems (4/6) - "Vectorized Execution"**
- ~~26.10.2022~~: ***No lecture!***
- 02.11.2022: Lecture 8: **DuckDB: An embedded database for data science (1/2) (guest lecture & *hands-on*)**

  (plus Assignment 3 [individual; 2 weeks]: Analysing NYC Cab dataset with DuckDB)
- 09.11.2022: Lecture 9: **DuckDB: An embedded database for data science (2/2) (guest lecture & *hands-on*)**
- 16.11.2022: Lecture 10: **Branch Misprediction & Predication**

  (plus Assignment 4 [individual; 2 weeks]: Predication)
- 23.11.2022: Lecture 11: **Column-Oriented Database Systems (5/6) - Adaptive Indexing**
- 30.11.2022: Lecture 12: **Column-Oriented Database Systems (6/6) - Progressive Indexing**

ADM: Literature

- **Column-Oriented Database Systems (4/6) - “Vectorized Execution”**

  - “MonetDB/X100: Hyper-Pipelining Query Execution”. Boncz, Zukowski, Nes. CIDR’05.

  - “Buffering Database Operations for Enhanced Instruction Cache Performance”. Zhou and Ross. SIGMOD’04.

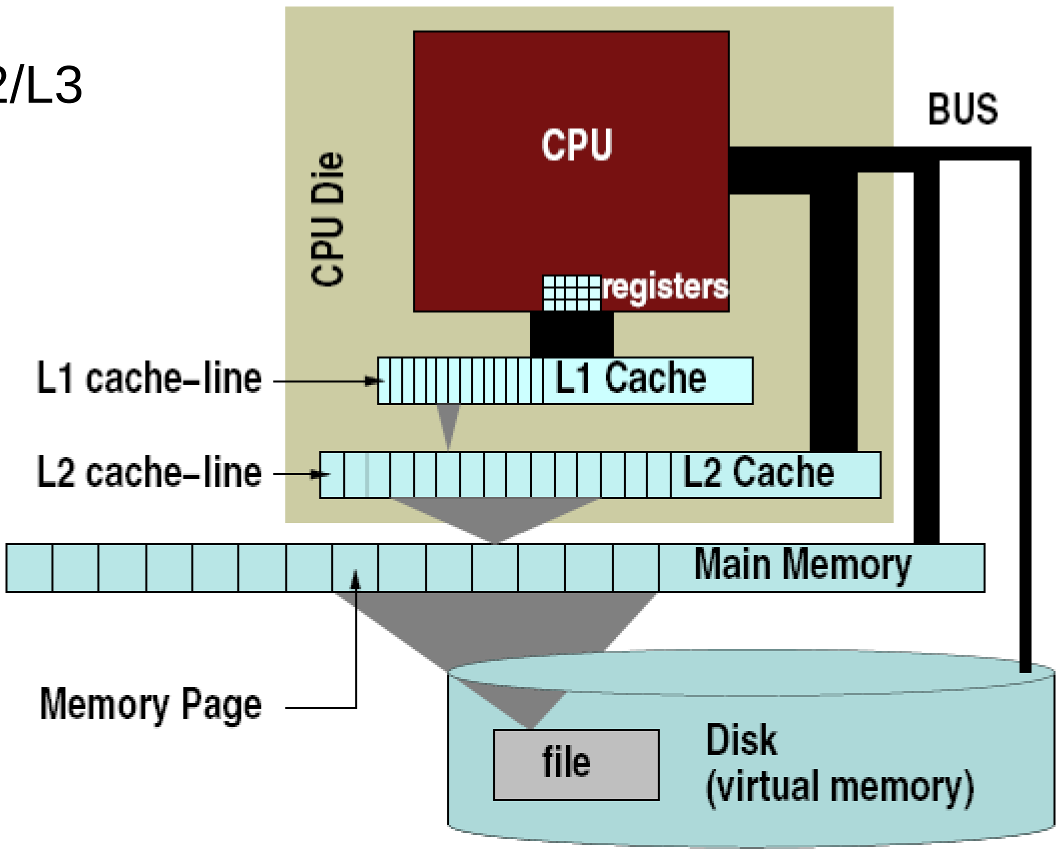
  - “Block oriented processing of relational database operations in modern computer architectures”. Padmanabhan, Malkemus, Agarwal. ICDE’01.

  - “Balancing Vectorized Query Execution with Bandwidth Optimized Storage”. Zukowski. PhD Thesis. CWI 2008.

CPU Architecture

Elements:

- Storage
  - CPU caches L1/L2/L3
- Registers
- Execution Unit(s)
  - Pipelined
  - SIMD

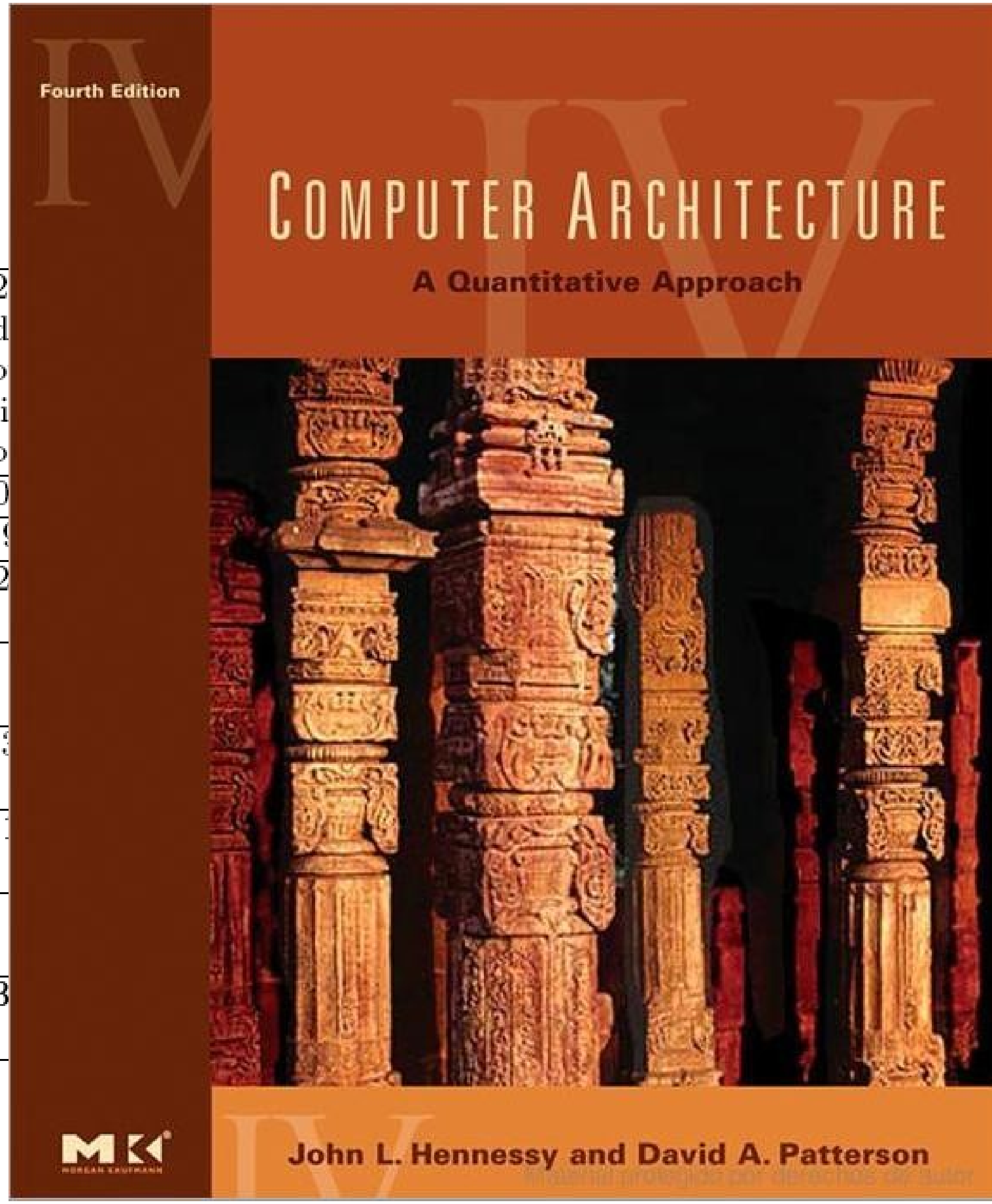


CPU Metrics

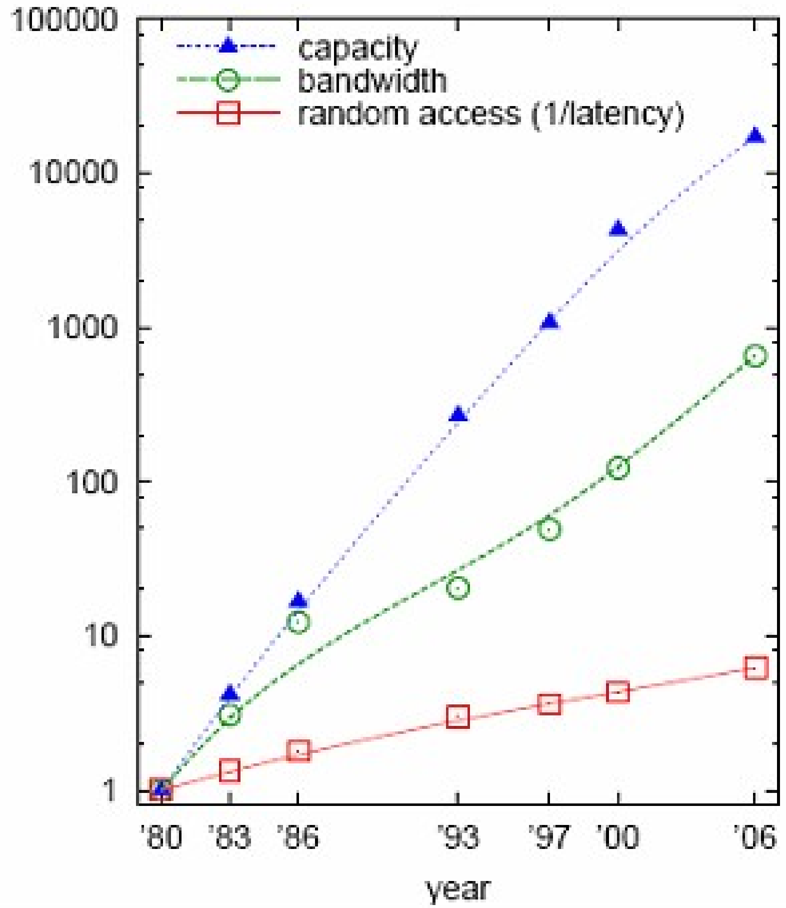
Processor	16-bit address/, bus, micro-coded	32-bit address/ bus, micro-coded	5-stage pipeline, on-chip I&D caches FPU	2-way super-scalar, 64-bit bus	Out-of-order, 3-way super-scalar	Out-of-order, super-pipelined, on-chip L2 cache	Multi-core
Product	80286	80386	80486	Pentium	PentiumPro	Pentium4	CoreDuo
Year	1982	1985	1989	1993	1997	2001	2006
Transistors (thousands)	134	275	1,200	3,100	5,500	42,000	151,600
Latency (clocks)	6	5	5	5	10	22	12
Bus width (bits)	16	32	32	64	64	64	64
Clock rate (MHz)	12.5	16	25	66	200	1500	2333
Bandwidth (MIPS)	2	6	25	132	600	4500	21000
Latency (ns)	320	313	200	76	50	15	5

CPU Metrics

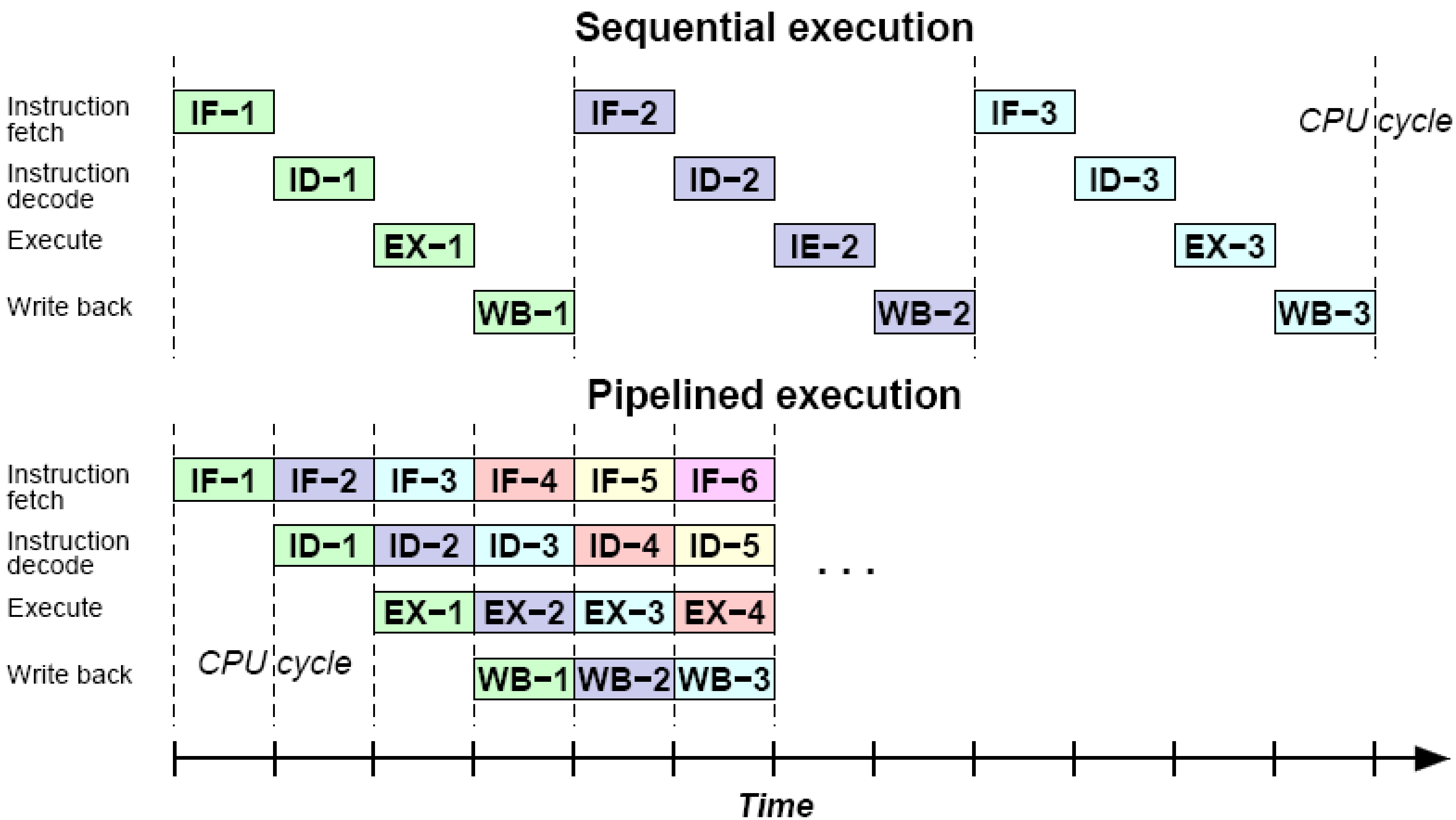
Processor	16-bit address/, bus, micro-coded	32 add b mi co	core
Product	80286	80	Duo
Year	1982	19	6
Transistors (thousands)	134	2	00
Latency (clocks)	6		
Bus width (bits)	16	3	
Clock rate (MHz)	12.5	1	3
Bandwidth (MIPS)	2		00
Latency (ns)	320	3	



DRAM Metrics



Super-Scalar Execution (pipelining)

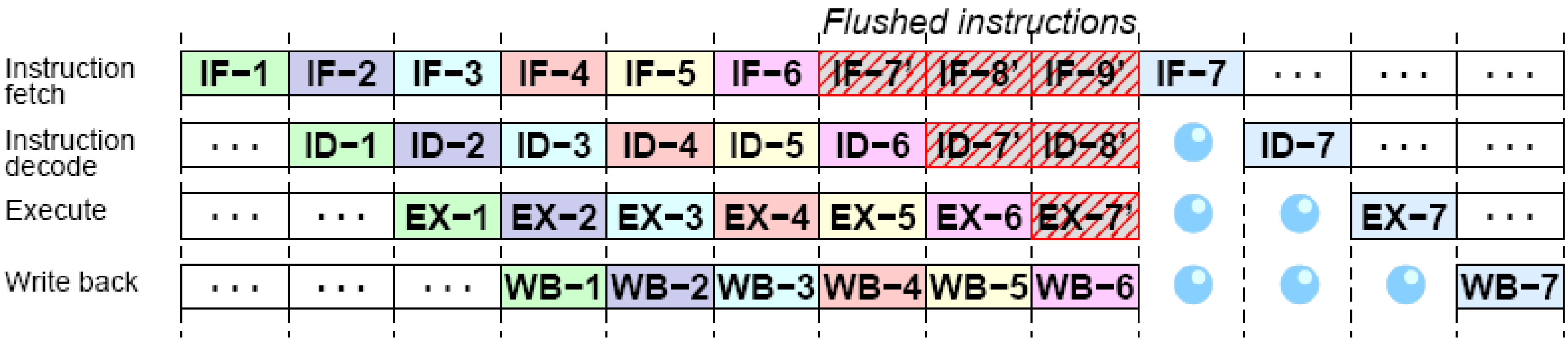


(See also https://en.wikipedia.org/wiki/Instruction_pipelining)

Hazards

- Data hazards
  - Dependencies between instructions
  - L1 data cache misses
- Control Hazards
  - Branch mispredictions
  - Computed branches (late binding)
  - L1 instruction cache misses

Result: bubbles in the pipeline



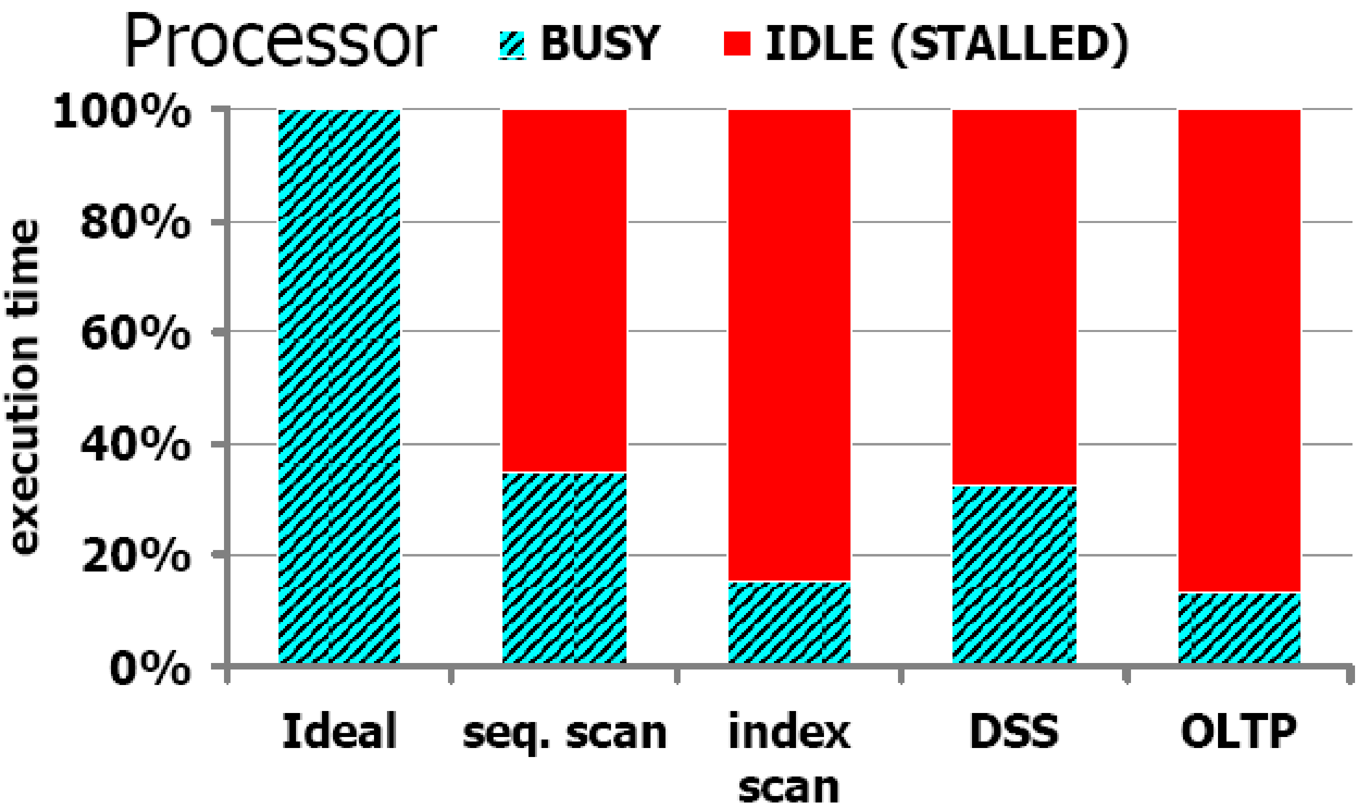
Out-of-order execution addresses data hazards

- control hazards typically more expensive

(See also https://en.wikipedia.org/wiki/Instruction_pipelining)

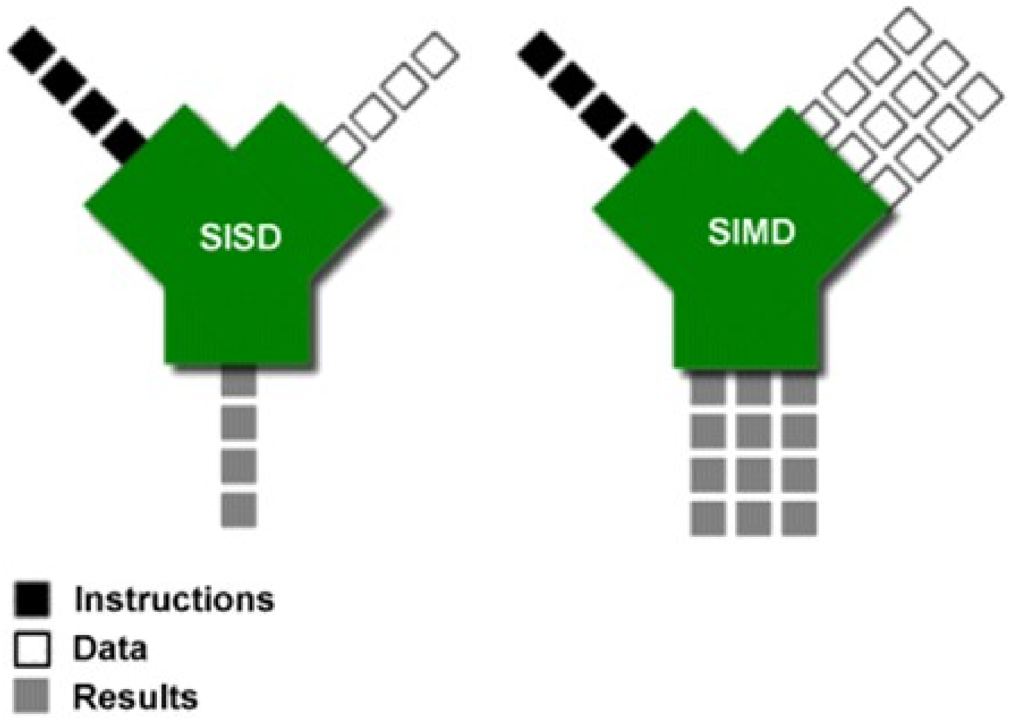
Database Architecture causes Hazards

- DB workload execution on a modern computer



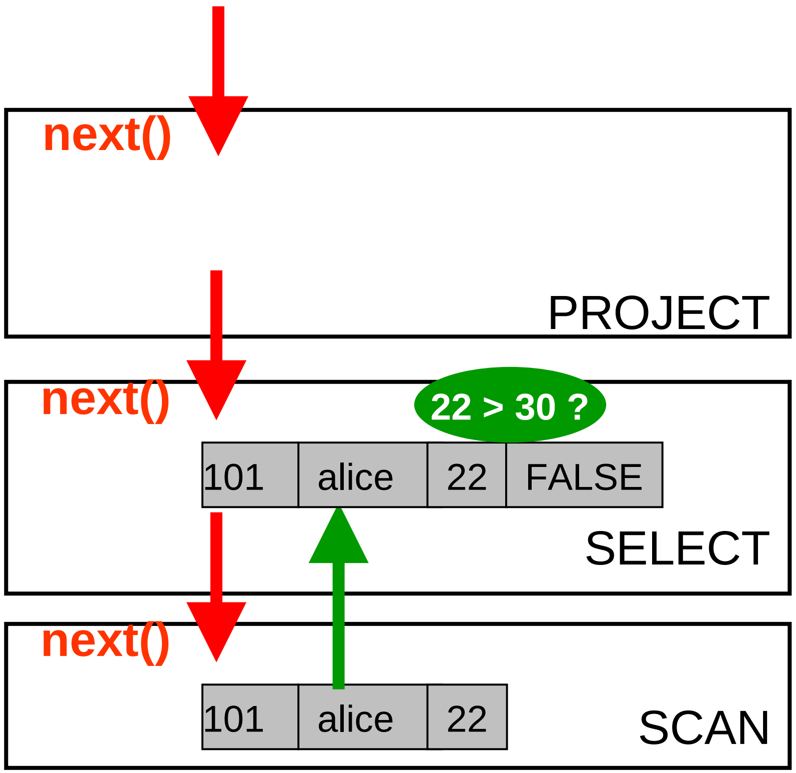
"DBMSs On A Modern Processor: Where Does Time Go? "
Ailamaki, DeWitt, Hill, Wood, VLDB'99

SIMD



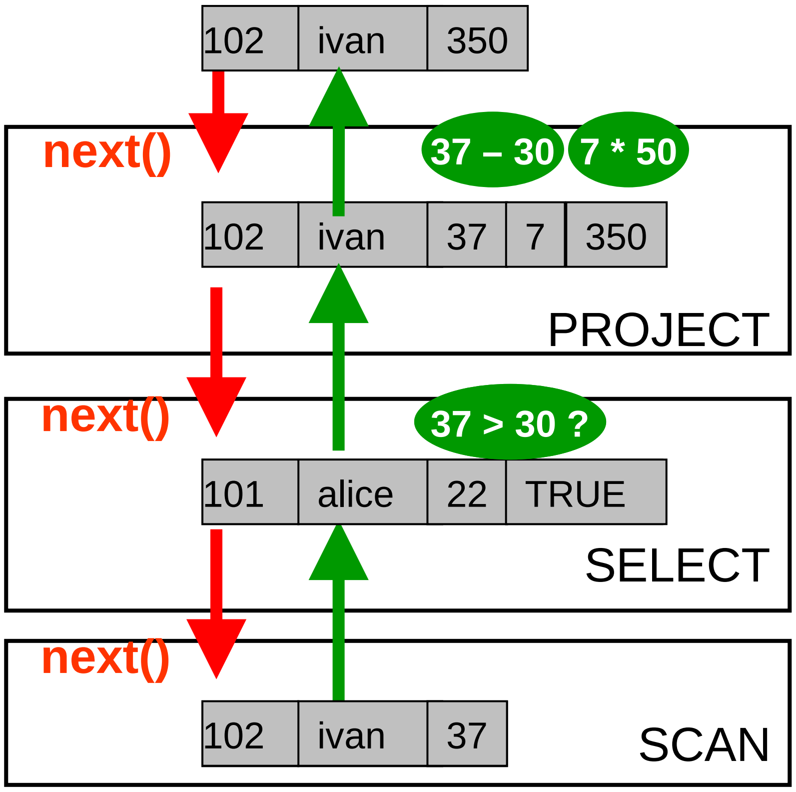
- Single Instruction Multiple Data
  - Same operation applied on a vector of values
  - MMX: 64 bits, SSE: 128bits, AVX: 256bits
  - SSE, e.g. multiply 8 short integers

A Look at the Query Pipeline



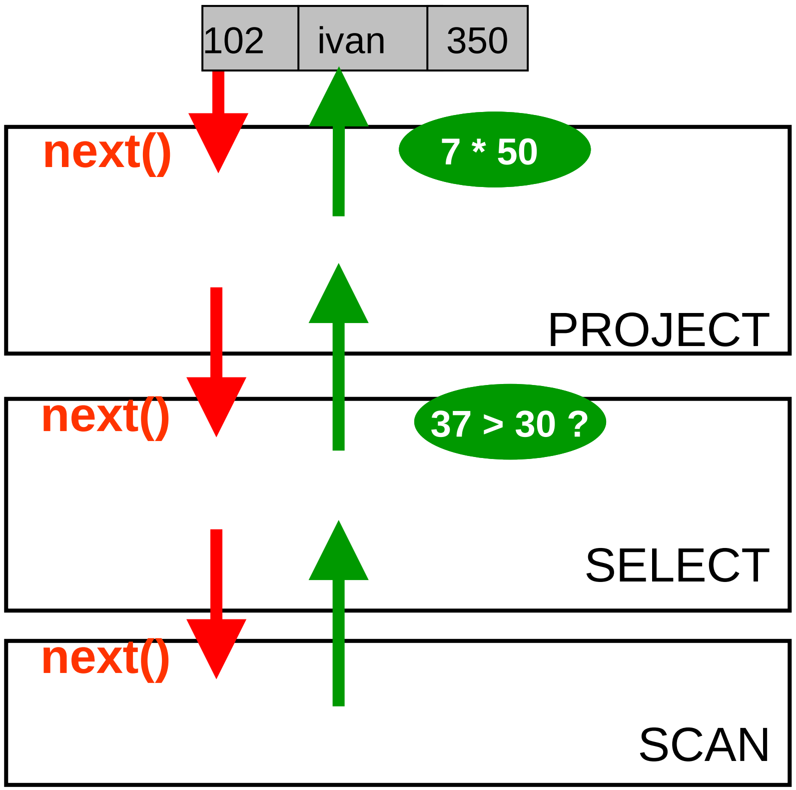

```
SELECT   id, name
         (age-30)*50 AS bonus
FROM     employee
WHERE    age > 30
```

A Look at the Query Pipeline



SELECT id, name
(age-30)*50 AS bonus
FROM employee
WHERE age > 30

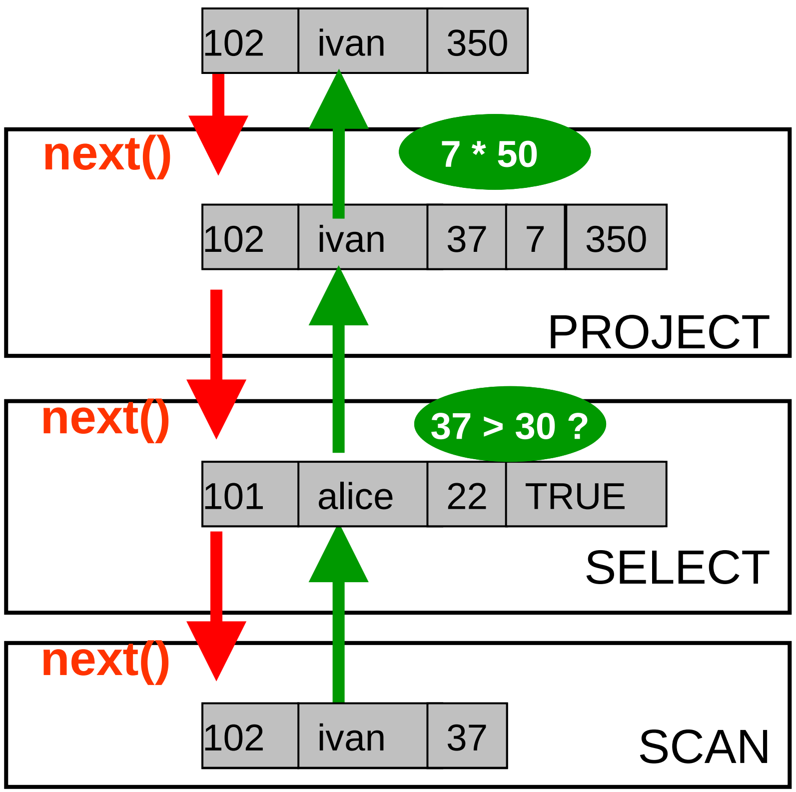
A Look at the Query Pipeline



Operators

Iterator interface
-open()
-next(): tuple
-close()

A Look at the Query Pipeline



Primitives

Provide computational functionality

All arithmetic allowed in expressions,
e.g. Multiplication

7 * 50

`mult(int,int)` ➔ `int`

Database Architecture causes Hazards

- Data hazards
  - Dependencies between instructions
  - L1 data cache misses
- Control Hazards
  - Branch mispredictions
  - Computed branches (late binding)
  - L1 instruction cache misses

SIMD

Out-of-order Execution

work on one tuple at a time

Large Tree/Hash Structures

Code footprint of all operators in query plan exceeds L1 cache

Data-dependent conditions

Next() late binding method calls

Tree, List, Hash traversal

Complex NSM record navigation

PROJECT

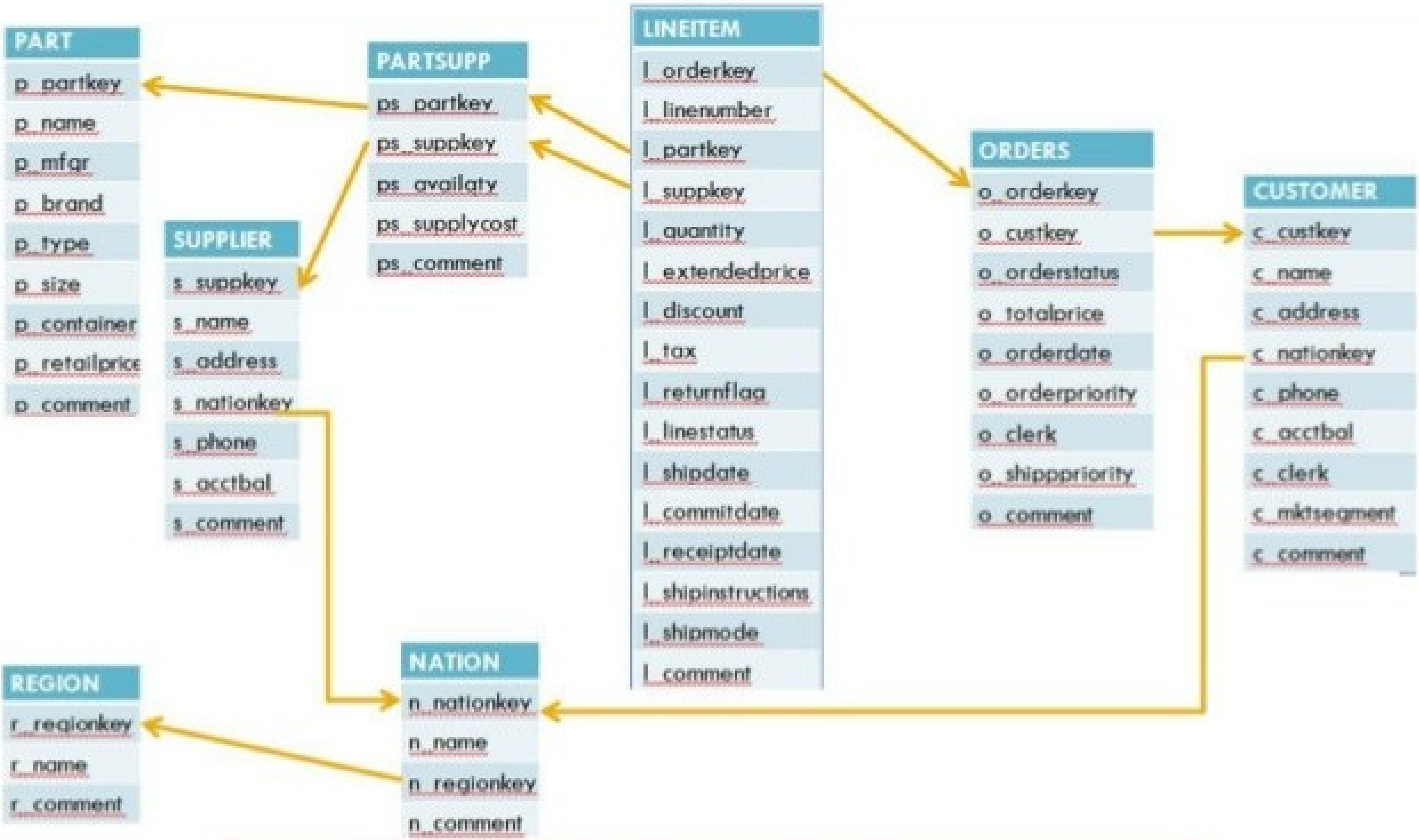
SELECT

SCAN

Operators

Iterator interface
-open()
-next(): tuple
-close()

TPC-H Database Schema



TPC-H Query 1

```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

DBMS Computational Efficiency

TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all
- Results:
  - C program: ?
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

DBMS Computational Efficiency

TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all
- Results:
  - C program: **0.2s**
  - MySQL: 26.2s
  - DBMS "X": 28.1s

"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

monetdb a column-store

- ~~“save disk I/O when scan-intensive queries need a few columns”~~
- “avoid an expression interpreter to improve computational efficiency”

RISC Relational Algebra

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

Simple, hard-coded semantics in operators

markH()

(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

[−](,30)

tmp

(void)	(int)
0	7
1	15
2	1
3	12
4	5

[*](50,)

sel_bonus

(void)	(int)
0	350
1	750
2	50
3	600
4	250

CPU ☺? Give it “nice” code !

- few dependencies (control,data)
- CPU gets out-of-order execution
- compiler can e.g. generate SIMD

One loop for an entire column

- no per-tuple interpretation
- arrays: no record navigation
- better instruction cache locality

```
batcalc_minus_int(int* res,
                  int* col,
                  int  val,
                  int n)
{
    for(i=0; i<n; i++)
        res[i] = col[i] - val;
}
```

monetdb a column-store

- ~~“save disk I/O when scan-intensive queries – need a few columns”~~
- “avoid an expression interpreter to improve computational efficiency”

How?

- RISC query algebra: hard-coded semantics
  - Decompose complex expressions in multiple operations
- Operators only handle **simple arrays**
  - No code that handles slotted buffered record layout
- Relational algebra becomes **array manipulation language**
  - Often SIMD for free

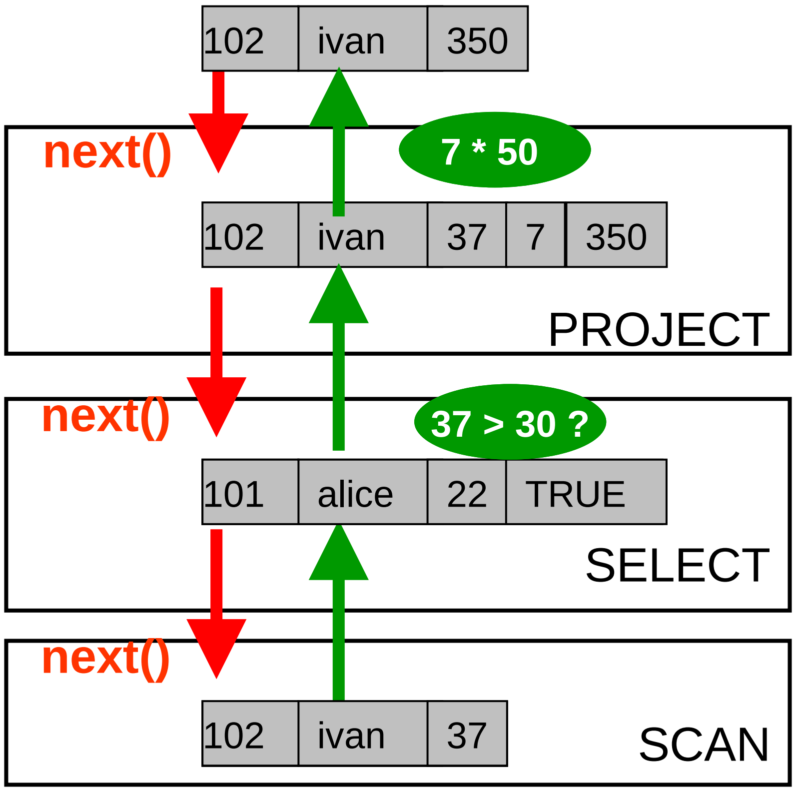
  - Plus: use of *cache-conscious* algorithms for Sort/Aggr/Join

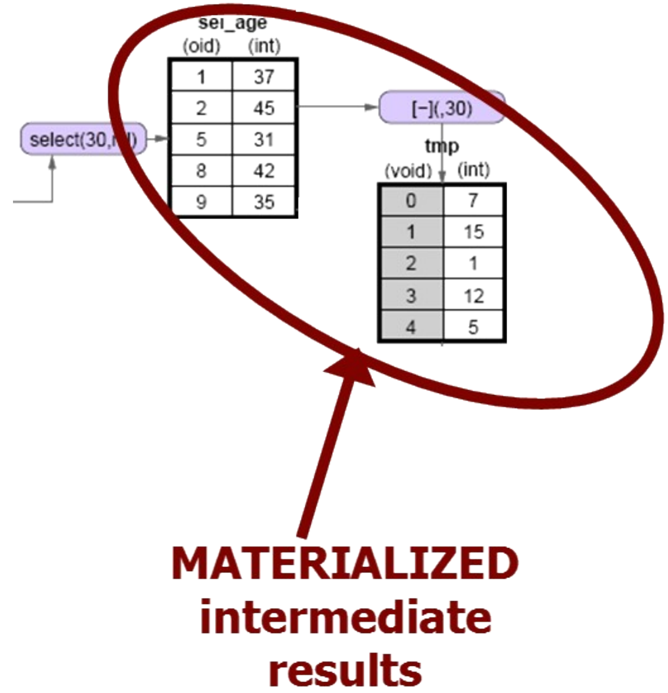
monetdb a Faustian pact

- You want efficiency
  - Simple hard-coded operators
- I take scalability
  - Result materialization

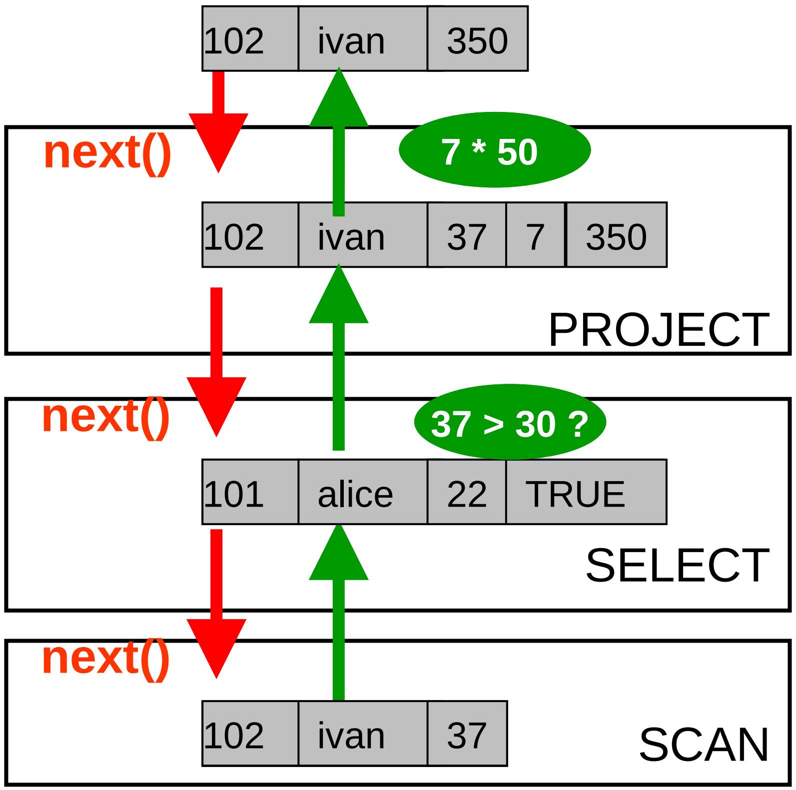
C program:	0.2s
MonetDB:	3.7s
MySQL:	26.2s
DBMS “X”:	28.1s

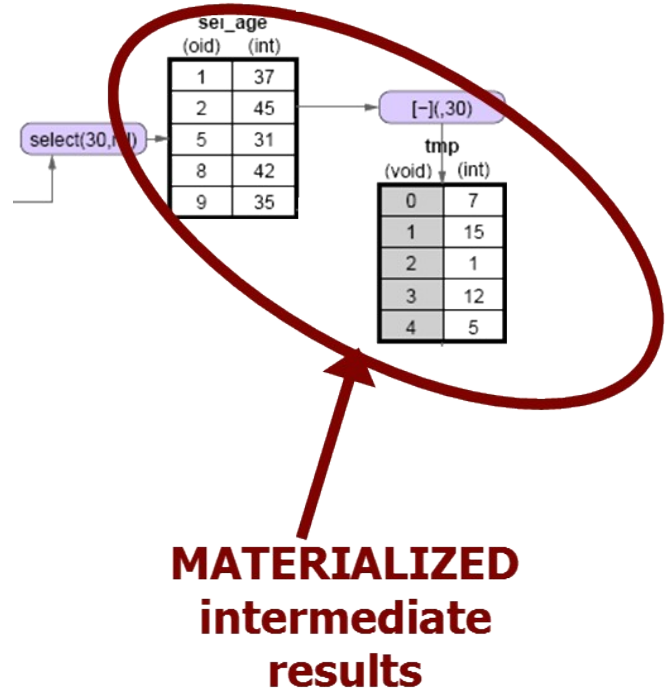
Pipelining vs Materialization





Pipelining vs Materialization





=> vectorized query processing

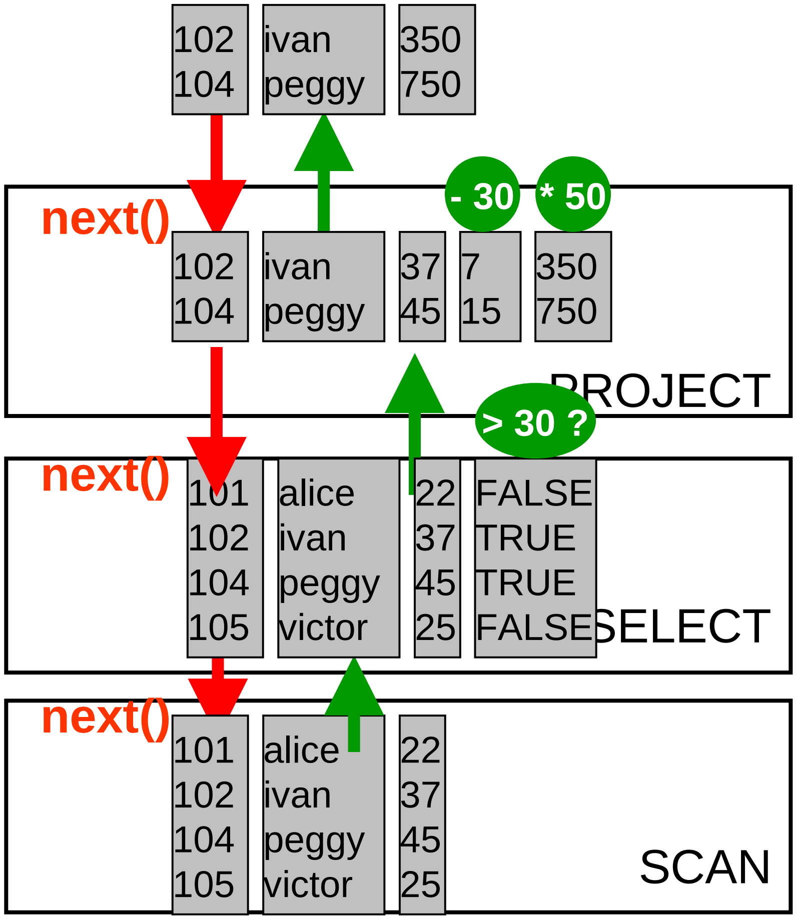
MonetDB spin-off: MonetDB/X100 vectorwise

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Observations:

next() called much less often ➔ more time spent in primitives less in overhead

primitive calls process an array of values in a **loop**:



“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

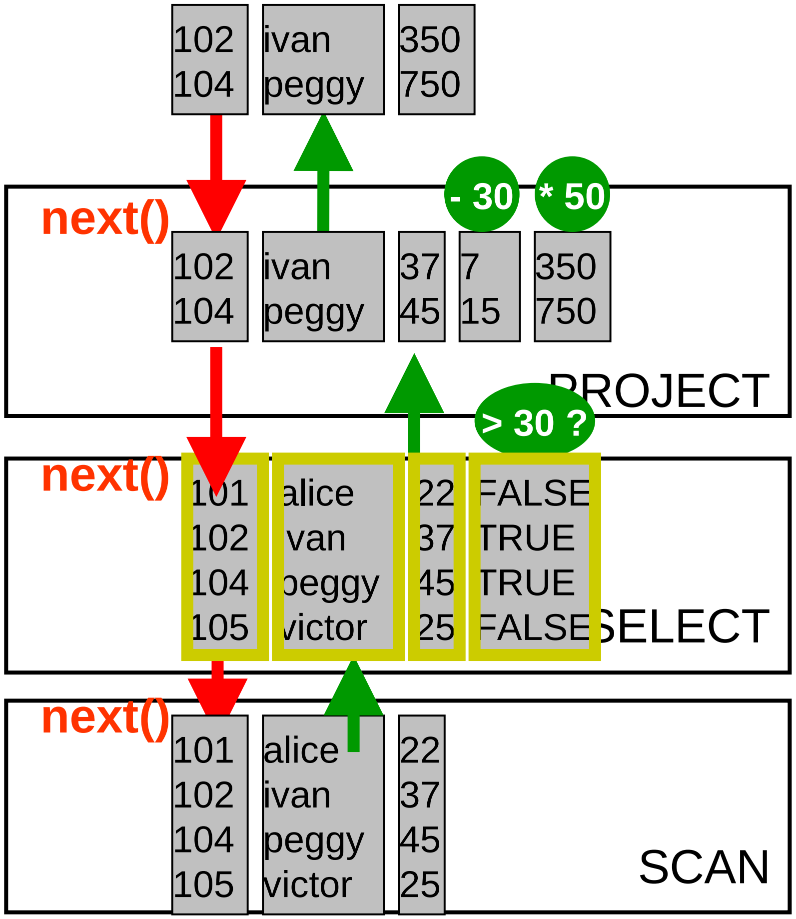
vectorwise

“Vectorized In Cache Processing”

vector = array of ~1000

processed in a tight loop

CPU cache Resident



“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

vectorwise

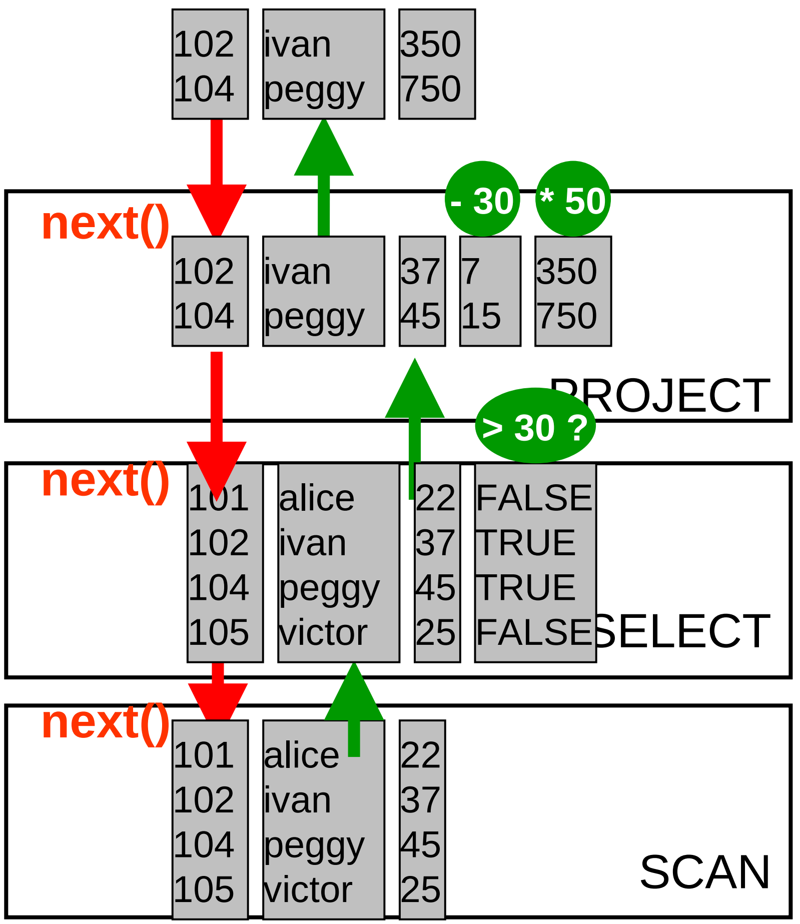
Observations:

next() called much less often ➔ more time spent in primitives less in overhead

primitive calls process an array of values in a **loop**:

CPU Efficiency depends on “nice” code
- out-of-order execution
- few dependencies (control,data)
- compiler support

Compilers like simple loops over arrays
- loop-pipelining
- automatic SIMD



“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Observations:

next() called much less often ➔ more time spent in primitives less in overhead

primitive calls process an array of values in a **loop**:

CPU Efficiency depends on “nice” code
- out-of-order execution
- few dependencies (control,data)
- compiler support

Compilers like simple loops over arrays
- loop-pipelining
- automatic SIMD

> 30 ?

FALSE
TRUE
TRUE
FALSE

```
for(i=0; i<n; i++)
  res[i] = (col[i] > x)
```

- 30

7
15

```
for(i=0; i<n; i++)
  res[i] = (col[i] - x)
```

*** 50**

350
750

```
for(i=0; i<n; i++)
  res[i] = (col[i] * x)
```

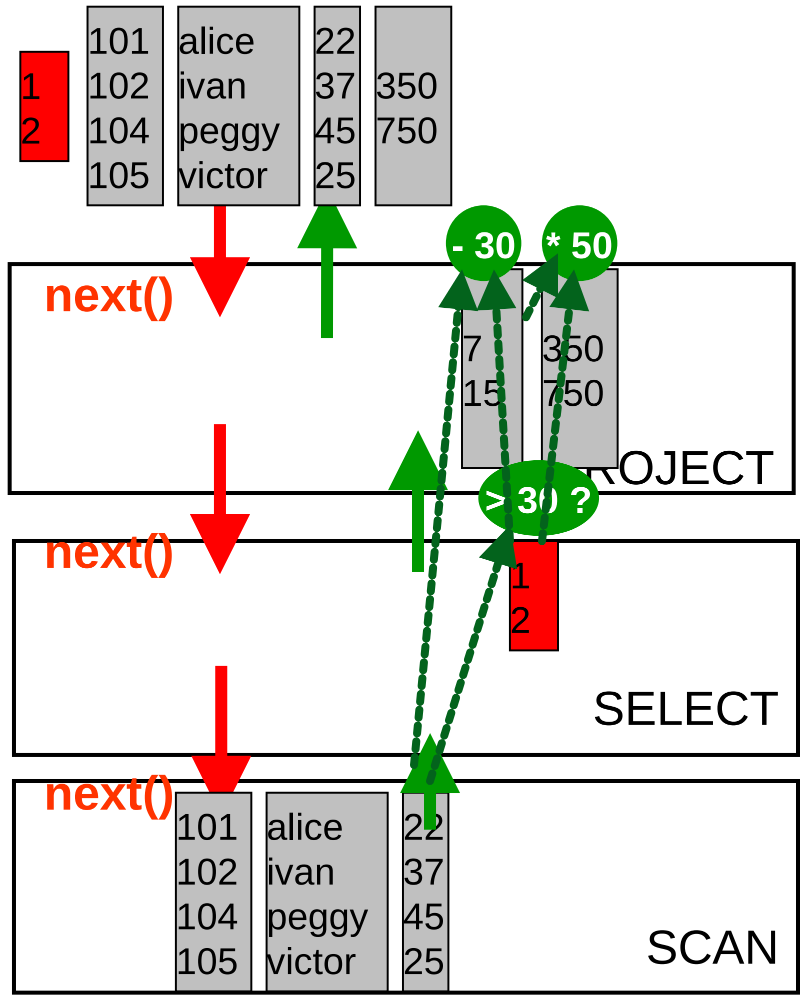
“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

vectorwise

Tricks being played:

- Late materialization

- Materialization avoidance using selection vectors



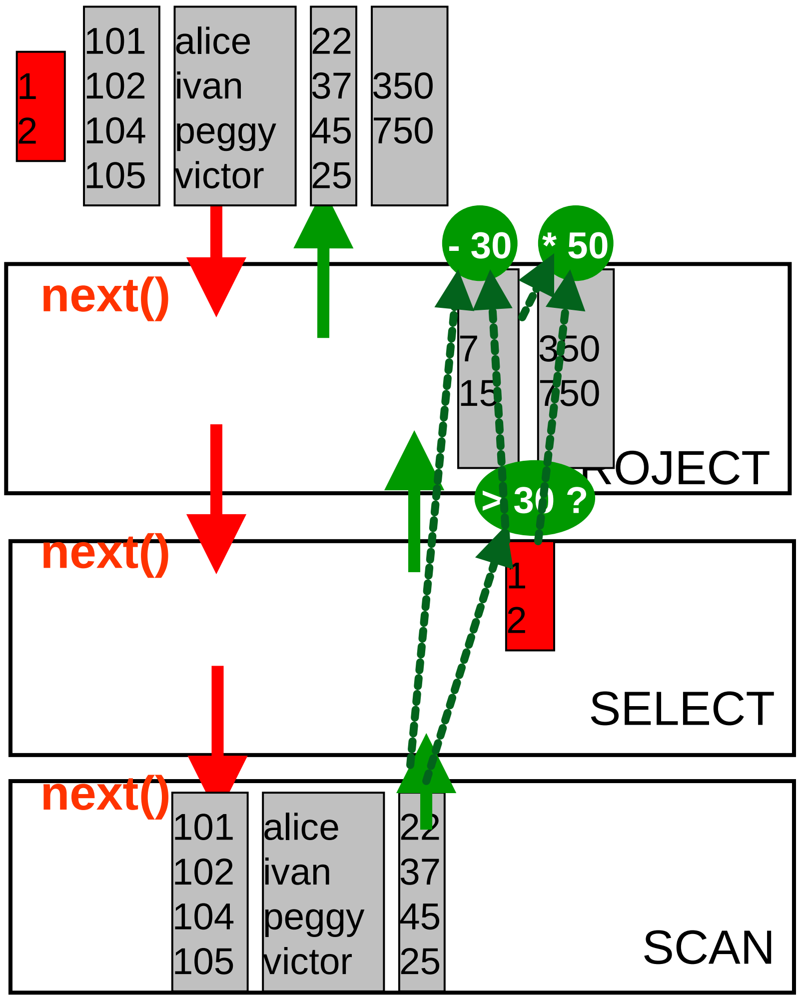
“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

```
map_mul_flt_val_flt_col(
    float *res,
    int*   sel,
    float  val,
    float *col, int n)

{
    for(int i=0; i<n; i++)
          res[i] = val * col[sel[i]];
}
```

selection vectors used to reduce vector copying

contain selected positions



vectorwise

MonetDB/X100

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

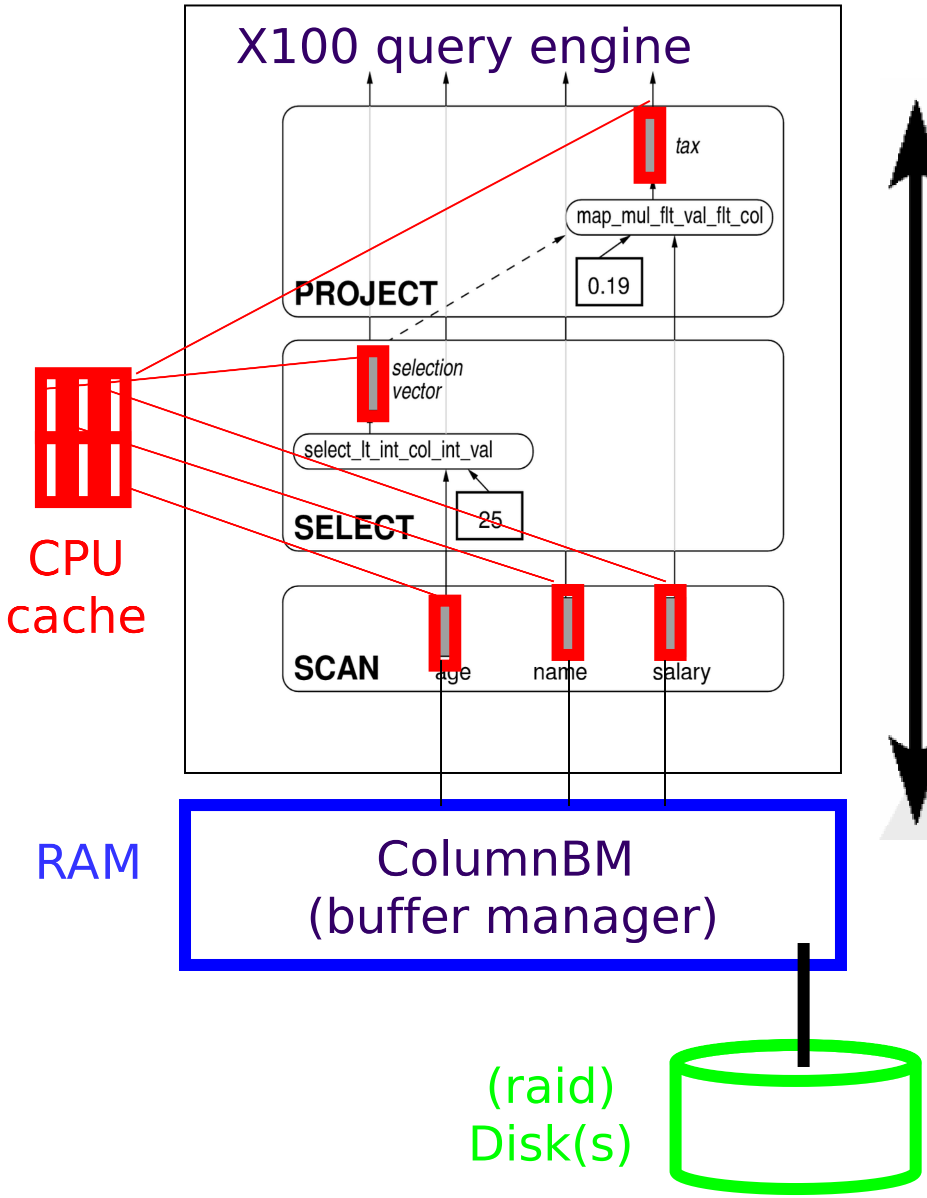
- Both efficiency
  - Vectorized primitives
- and scalability..
  - Pipelined query evaluation

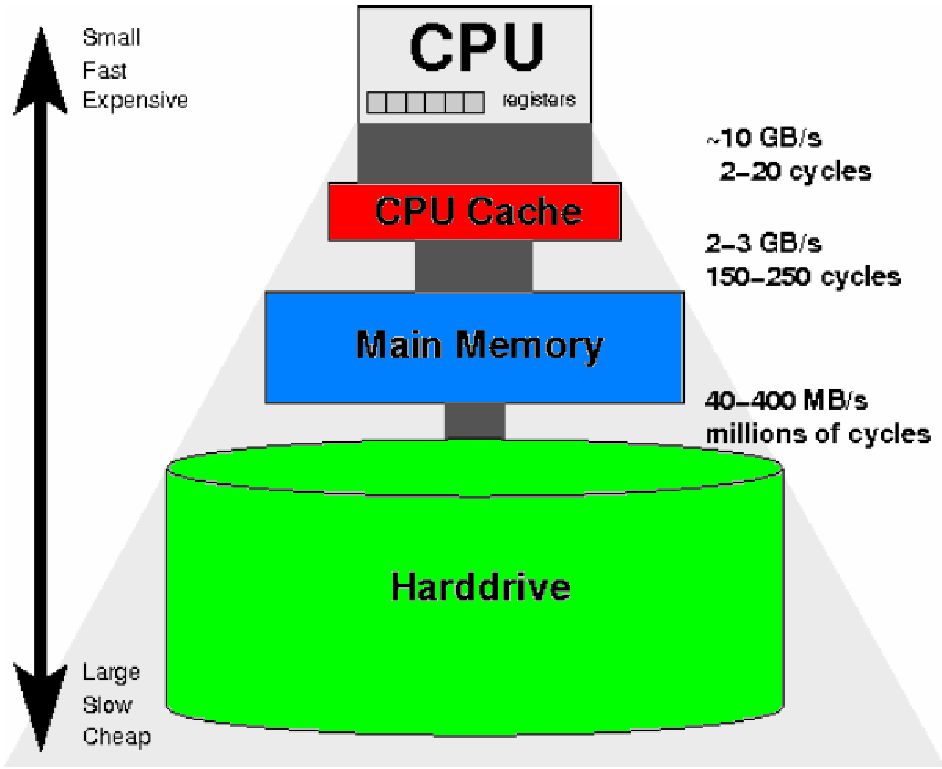
- C program: 0.2s
- MonetDB/X100: 0.6s
- MonetDB: 3.7s
- MySQL: 26.2s
- DBMS “X”: 28.1s

vectorwise

Memory Hierarchy

"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

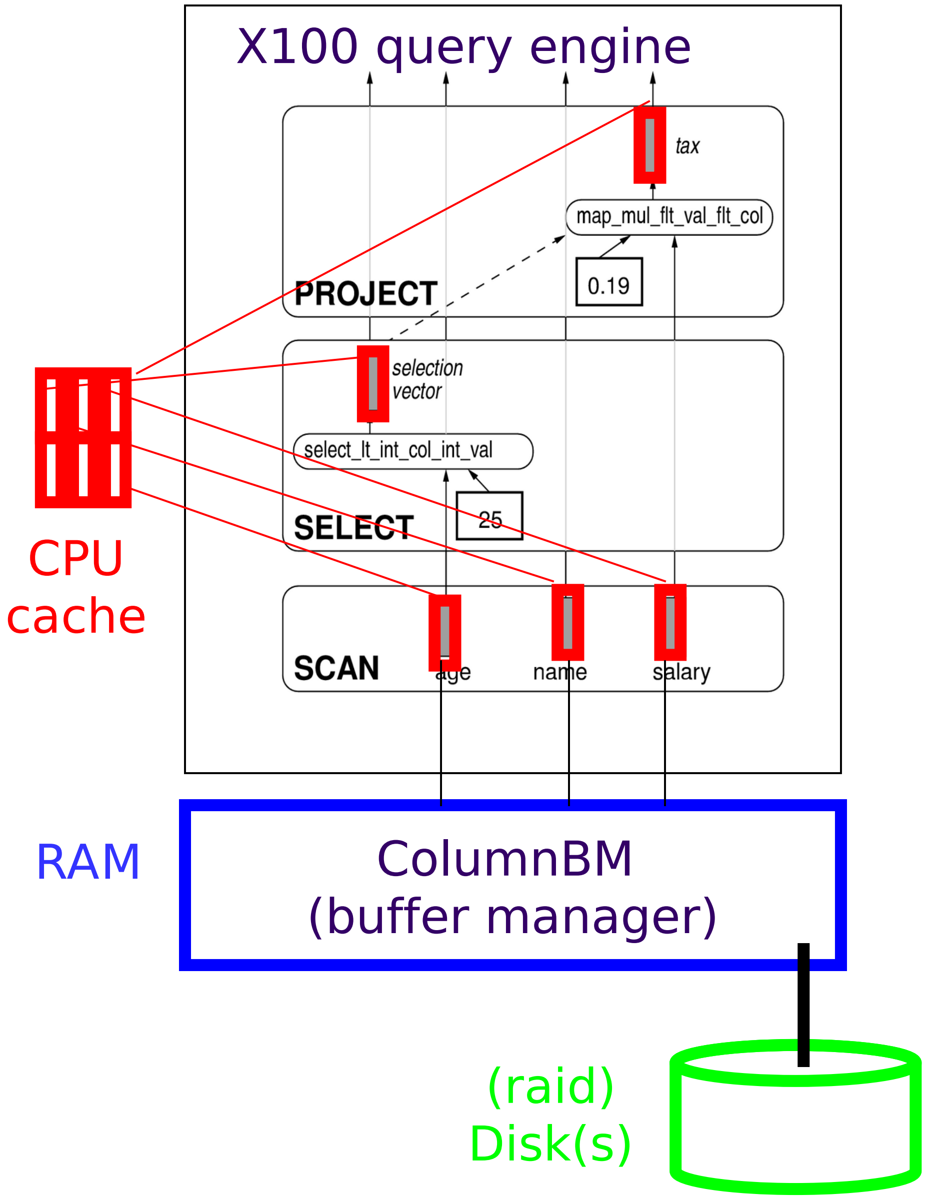




vectorwise

Memory Hierarchy

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05



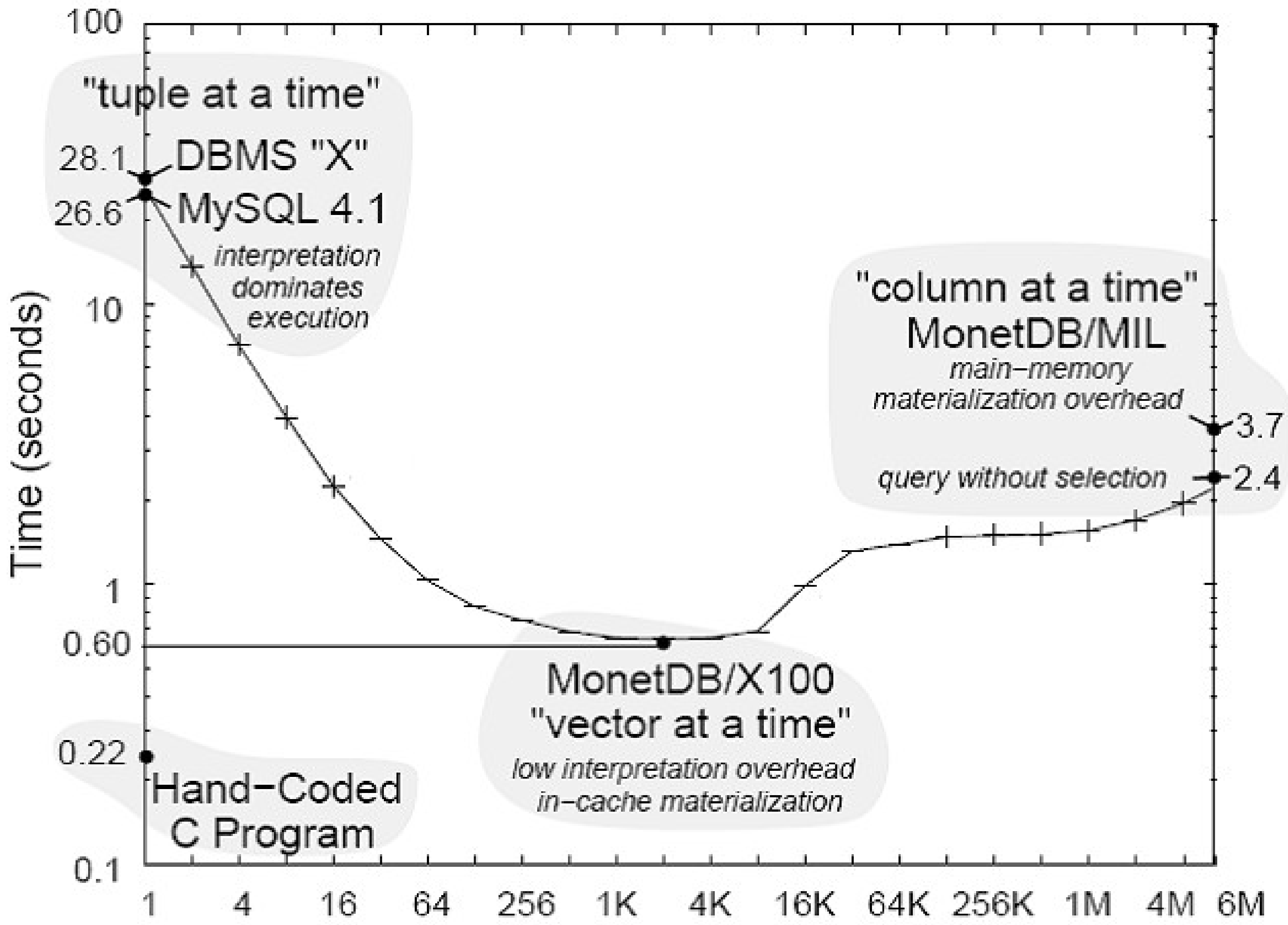
Vectors are only the in-cache representation

RAM & disk representation might actually be different

(vectorwise uses both PAX & DSM)

Varying the Vector size

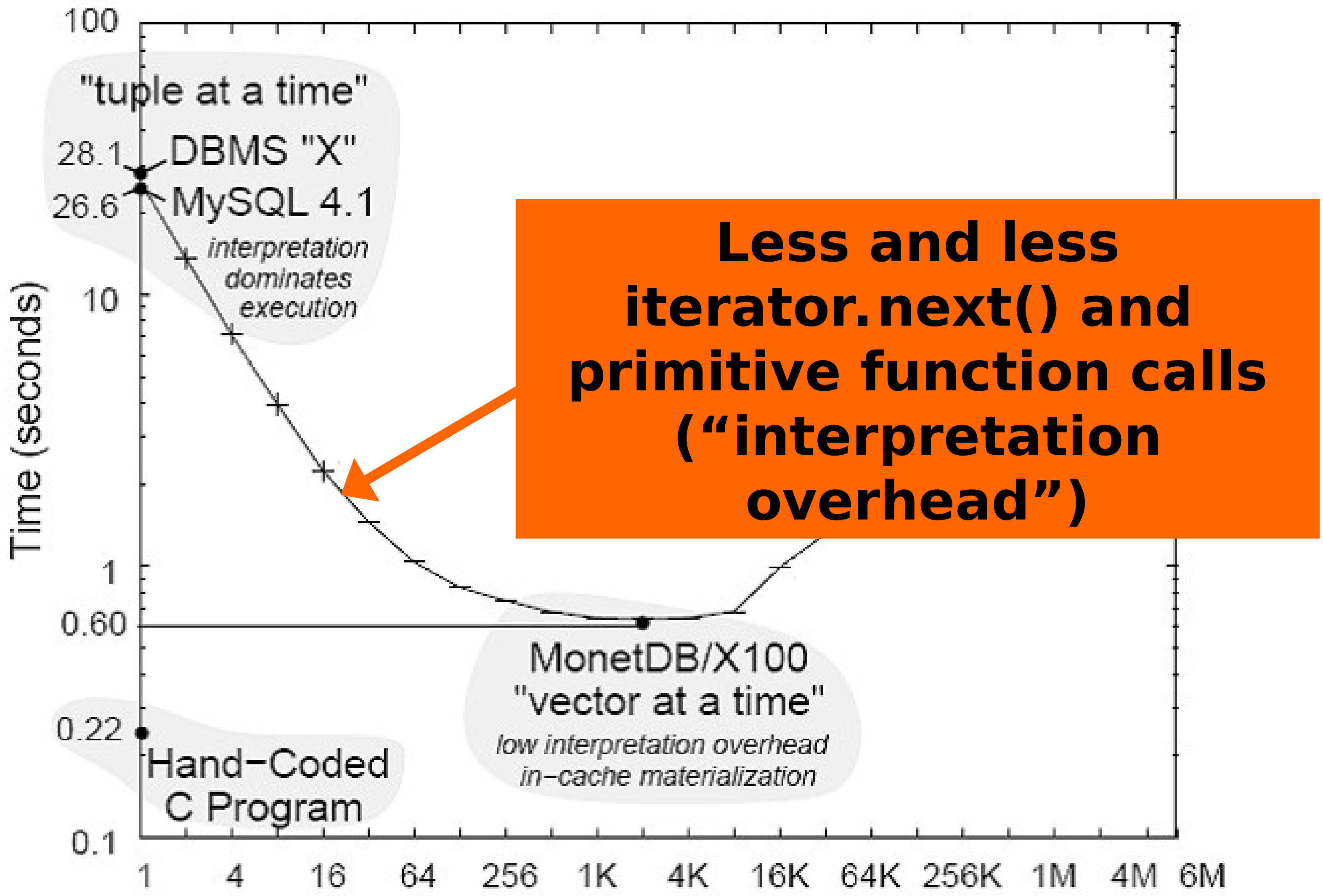
"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05



vectorwise

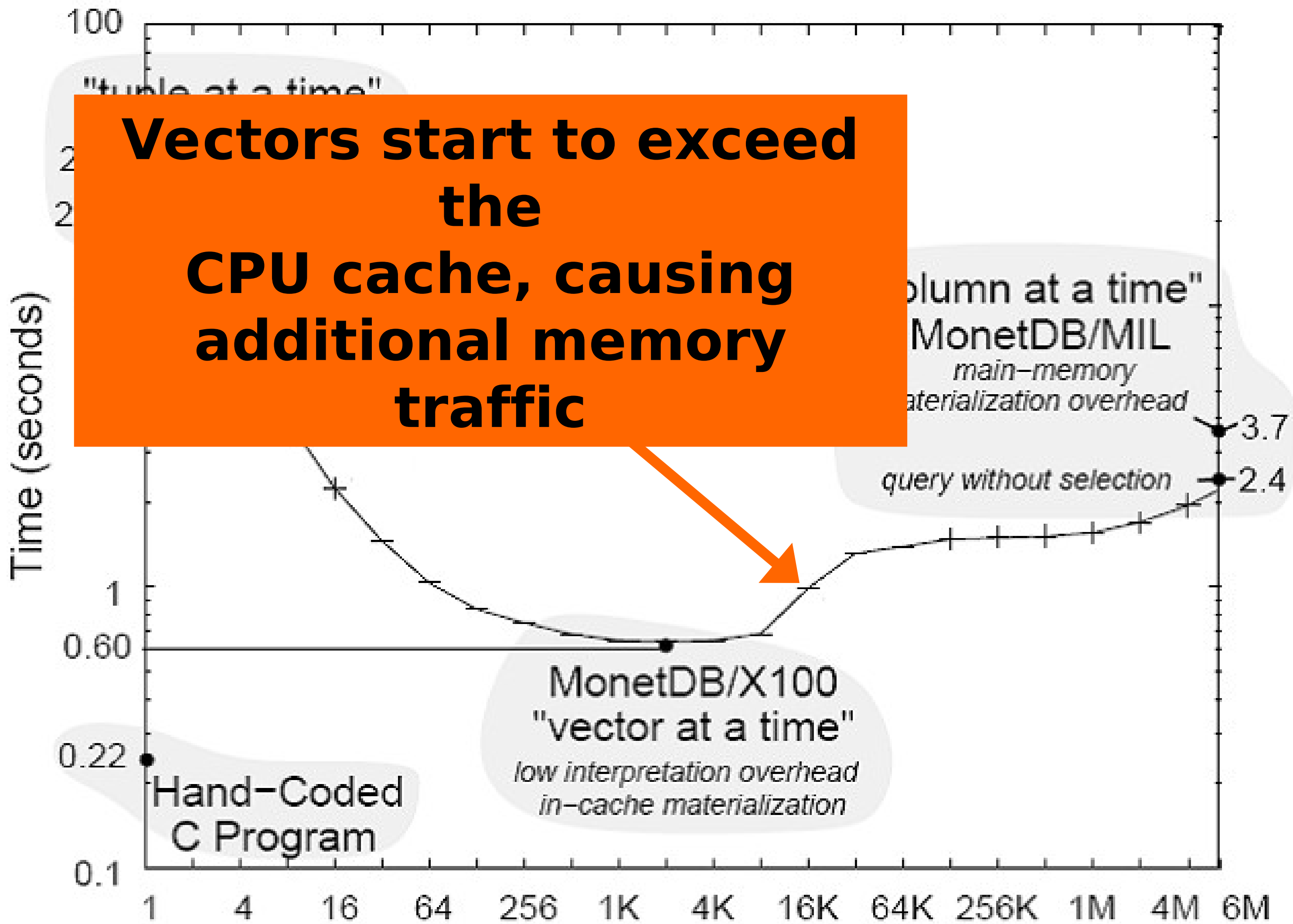
“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Varying the Vector size



“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

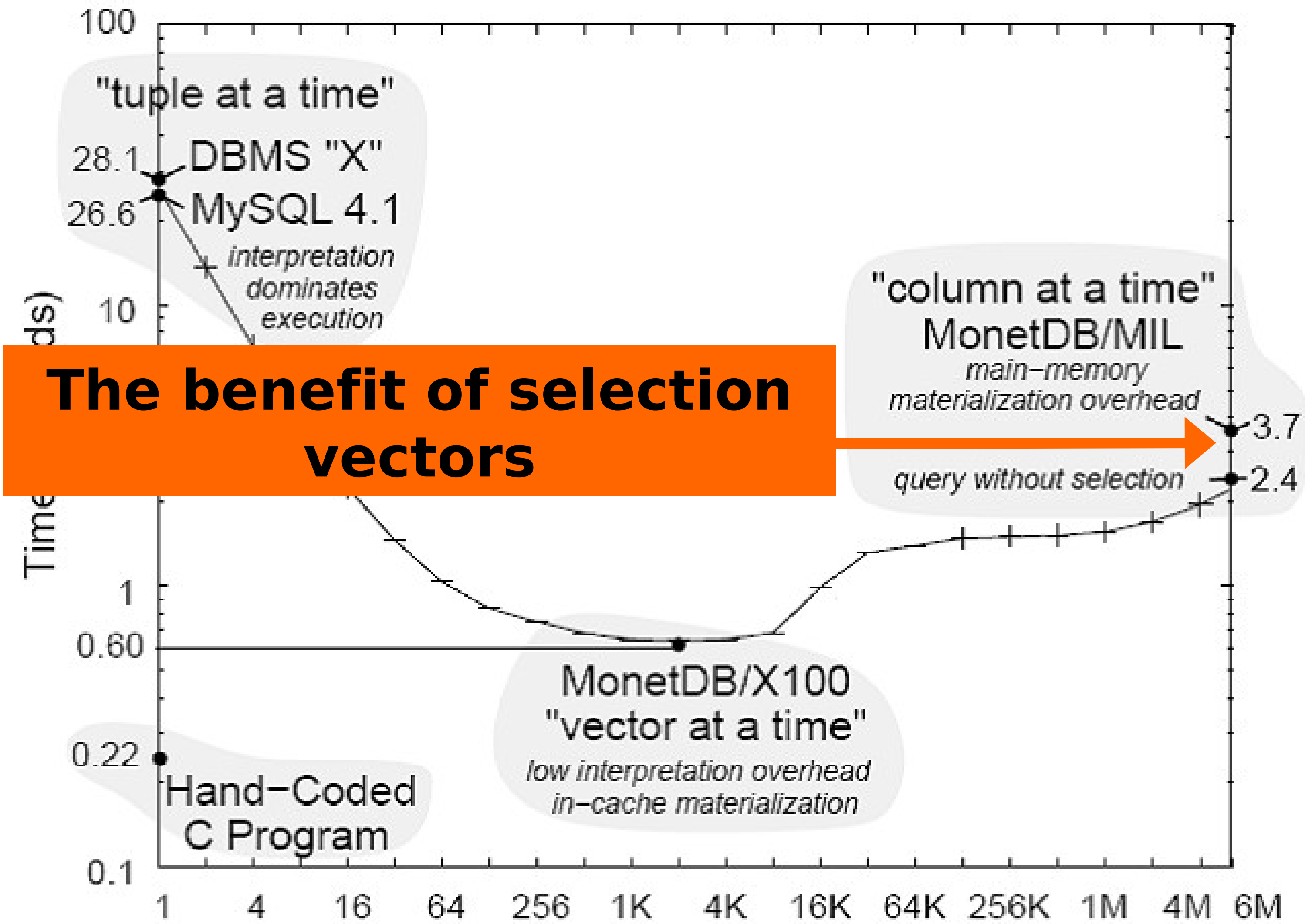
Varying the Vector size



Vectors start to exceed the CPU cache, causing additional memory traffic

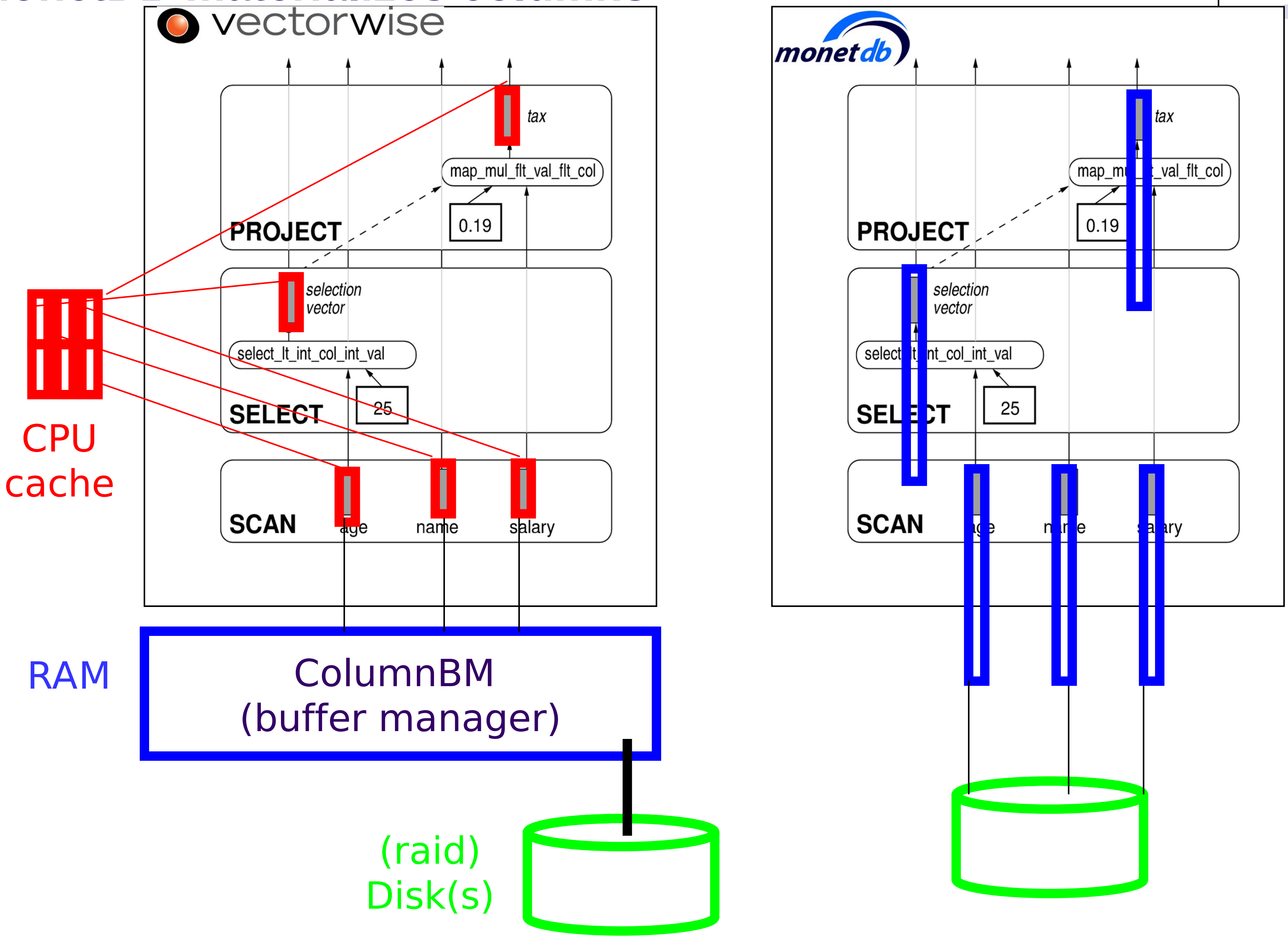
“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Varying the Vector size



MonetDB materializes columns

"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05



vectorwise

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

Benefits of Vectorized Processing

“Buffering Database Operations for Enhanced Instruction Cache Performance” Zhou, Ross, SIGMOD’04

“Block oriented processing of relational database operations in modern computer architectures” Padmanabhan, Malkemus, Agarwal, ICDE’01

- **100x less Function Calls**
  - iterator.next(), primitives
- **No Instruction Cache Misses**
  - High locality in the primitives
- **Less Data Cache Misses**
  - Cache-conscious data placement
- **No Tuple Navigation**
  - Primitives are record-oblivious, only see arrays
- **Vectorization allows algorithmic optimization**
  - Move activities out of the loop (“strength reduction”)
- **Compiler-friendly function bodies**
  - Loop-pipelining, automatic SIMD

vectorwise

"Balancing Vectorized Query Execution with Bandwidth Optimized Storage " Zukowski, CWI 2008

Vectorizing Relational Operators

- Project
- Select
  - Exploit selectivities, test buffer overflow
- Aggregation
  - Ordered, Hashed
- Sort
  - Radix-sort nicely vectorizes
- Join
  - Merge-join + Hash-join

ADM: Literature

- **Column-Oriented Database Systems (4/6) - “Vectorized Execution”**
  - “MonetDB/X100: Hyper-Pipelining Query Execution”. Boncz, Zukowski, Nes. CIDR’05.
  - “Buffering Database Operations for Enhanced Instruction Cache Performance”. Zhou and Ross. SIGMOD’04.
  - “Block oriented processing of relational database operations in modern computer architectures”. Padmanabhan, Malkemus, Agarwal. ICDE’01.
  - “Balancing Vectorized Query Execution with Bandwidth Optimized Storage”. Zukowski. PhD Thesis. CWI 2008.

ADM: Agenda

- 07.09.2022: Lecture 1: **Introduction**
- 14.09.2022: Lecture 2: **SQL Recap**

  (plus Assignment 1 [in groups; 3 weeks]: TPC-H benchmark)
- 21.09.2022: Lecture 3: **Column-Oriented Database Systems (1/6) - Motivation & Basic Concepts**
- 28.09.2022: Lecture 4: **Column-Oriented Database Systems (2a/6) - Selected Execution Techniques (1/2)**
- 05.10.2022: Lecture 5: **Column-Oriented Database Systems (2b/6) - Selected Execution Techniques (2/2)**

  (plus Assignment 2 [in groups; 4 weeks]: Compression techniques)
- 12.10.2022: Lecture 6: **Column-Oriented Database Systems (3/6) - Cache Conscious Joins**
- 19.10.2022: Lecture 7: **Column-Oriented Database Systems (4/6) - "Vectorized Execution"**
- ~~26.10.2022~~: ***No lecture!***
- 02.11.2022: Lecture 8: **DuckDB: An embedded database for data science (1/2) (guest lecture & *hands-on*)**

  (plus Assignment 3 [individual; 2 weeks]: Analysing NYC Cab dataset with DuckDB)
- 09.11.2022: Lecture 9: **DuckDB: An embedded database for data science (2/2) (guest lecture & *hands-on*)**
- 16.11.2022: Lecture 10: **Branch Misprediction & Predication**

  (plus Assignment 4 [individual; 2 weeks]: Predication)
- 23.11.2022: Lecture 11: **Column-Oriented Database Systems (5/6) - Adaptive Indexing**
- 30.11.2022: Lecture 12: **Column-Oriented Database Systems (6/6) - Progressive Indexing**

Bring your own laptop!

ADM: Literature (5/6)

- **DuckDB: An embedded database for data science**
  - “DuckDB: an Embeddable Analytical Database”. Mark Raasveldt & Hannes Mühleisen. SIGMOD’19. Demo.
  - “Data Management for Data Science - Towards Embedded Analytics”. Mark Raasveldt & Hannes Mühleisen. CIDR’20.
  - “Integrating Analytics with Relational Databases”. Mark Raasveldt. PhD Thesis, Leiden University & CWI, 2020.
  - https://duckdb.org