X100
Architektura nowoczesnego systemu bazodanowego

Marcin Żukowski

(marcin@cwi.nl)

MIMUW, 2009-05-28

Polska trudna język

- Prezentacja po polsku
  - Eksperyment ☺
  - Przepraszam za pomyłki
- Slajdy po angielsku

Application focus

- Two major DBMS application types
- Transaction processing – not today ☺
- **Data-analysis applications**
  - Data warehousing, reporting etc.
  - Scientific data, information retrieval

- A lot of technical content… wake up!

Outline

- Traditional database performance
  - Improvements in MonetDB
- X100
  - Query execution
  - Storage

Motivation

TPC-H benchmark (1GB), Query 1

- Selects 98% of a fact table (6M rows), performs simple aggregations
- Performance:
  - C program: ?
  - MySQL: 26.2s
  - DBMS "X": 28.1s

Motivation

TPC-H benchmark (1GB), Query 1

- Selects 98% of a fact table (6M rows), performs simple aggregations
- Performance:
  - C program: **0.2s**
  - MySQL: 26.2s
  - DBMS "X": 28.1s

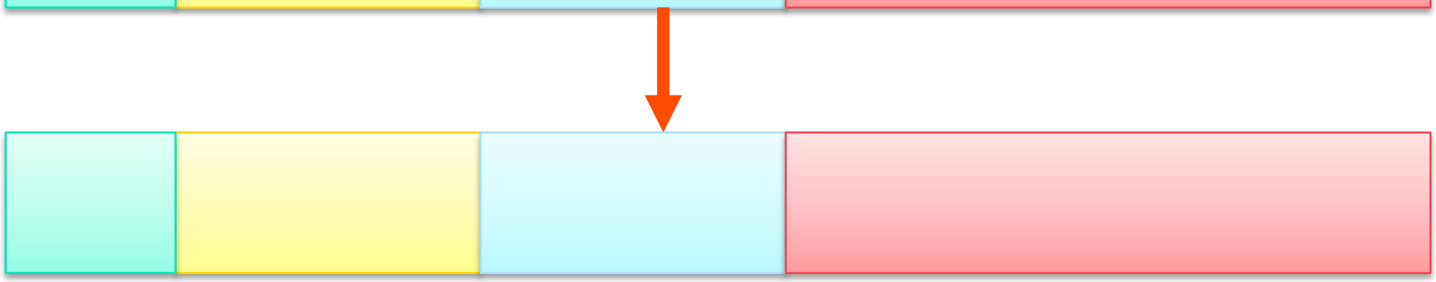
Database performance

- Why so slow?
  - Inefficient data storage format
  - Inefficient query processing model

N-ary storage model (NSM)

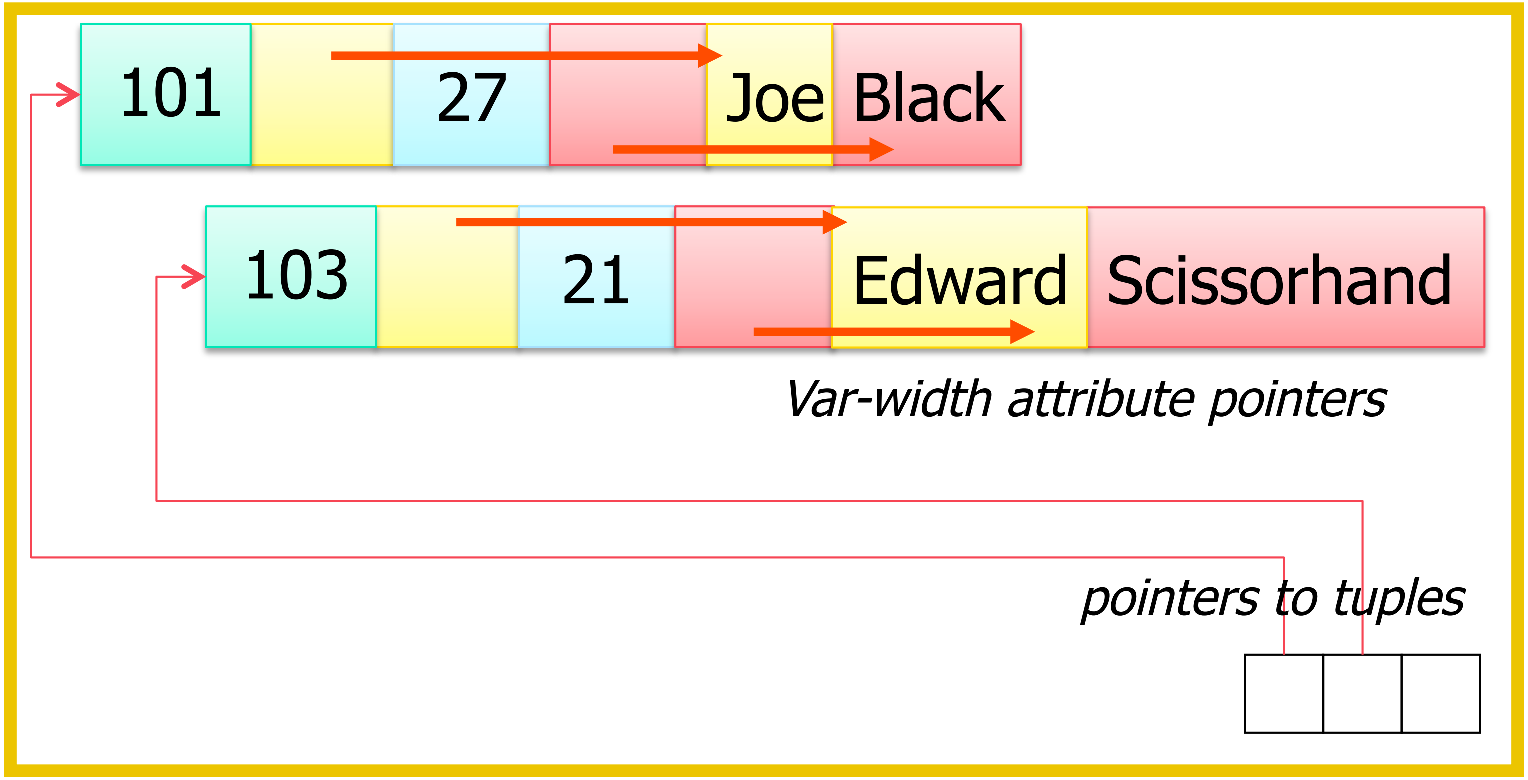
- Attributes in a record

101	Joe	27	Black
103	Edward	21	Scissorhand



Real-life NSM implementation

- Pages on disk – example:

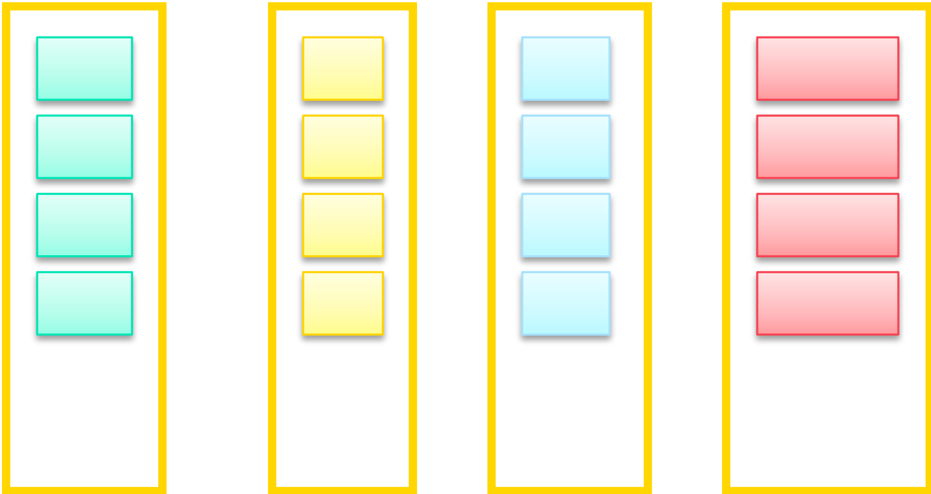


NSM problems

- **Always read all the attributes**
  - Poor bandwidth and buffer-space use
  - Terrible on disk
  - Bad in memory
- Complex tuple structure and navigation
  - e.g. compressing out null fields

Column stores to the rescue!

- Store attributes separately

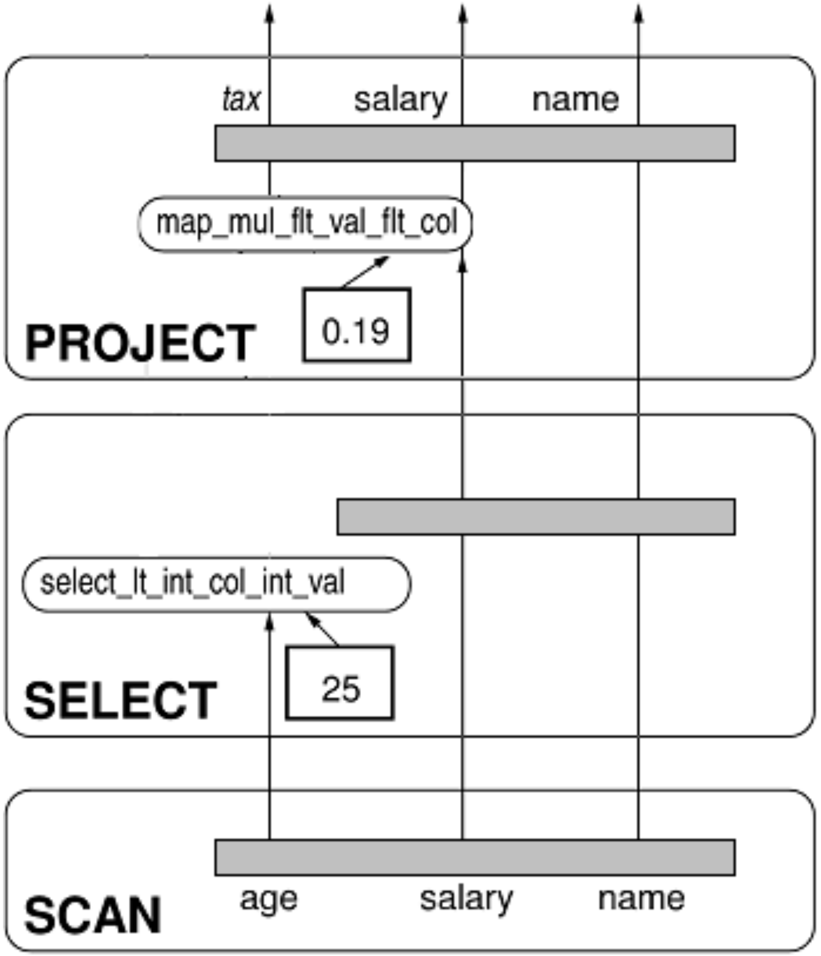


- Read only attributes used by a query

“Traditional” column stores

- Data path
  - Read columns from disk
  - Convert into NSM
  - Use NSM-based processing
- Examples: Sybase IQ, Vertica
- Not enough!
  - **Only I/O problem addressed**

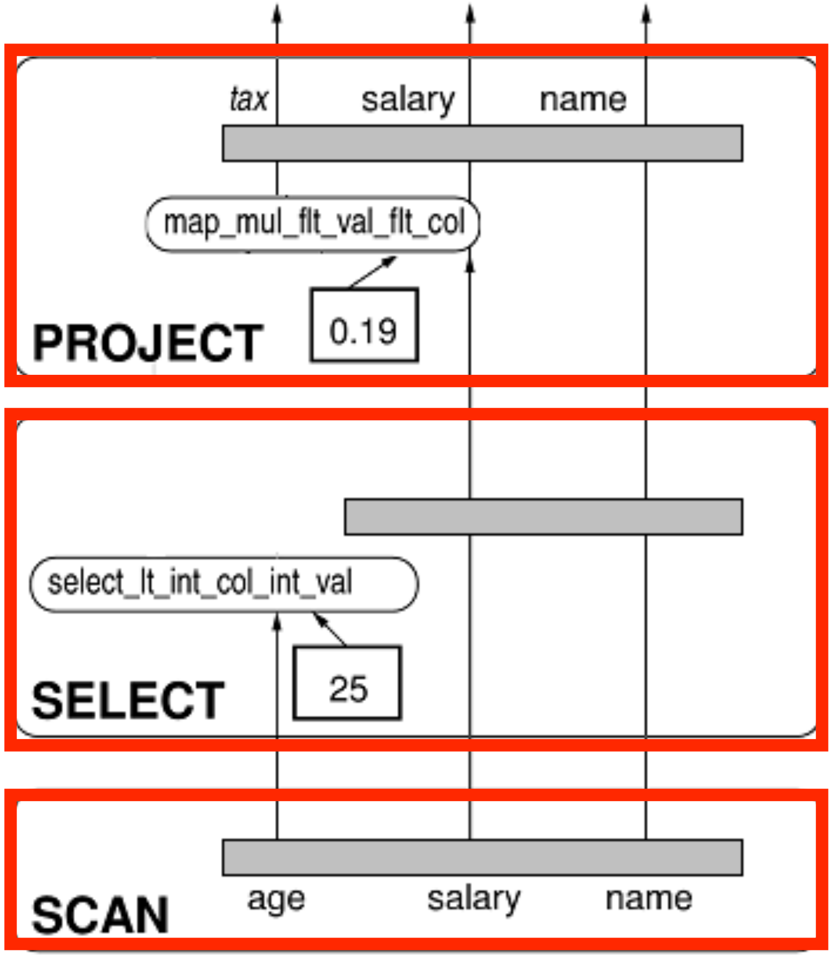
How databases run a query



Query

SELECT
name,
salary*.19 AS tax
FROM
employee
WHERE
age > 25

Database operators

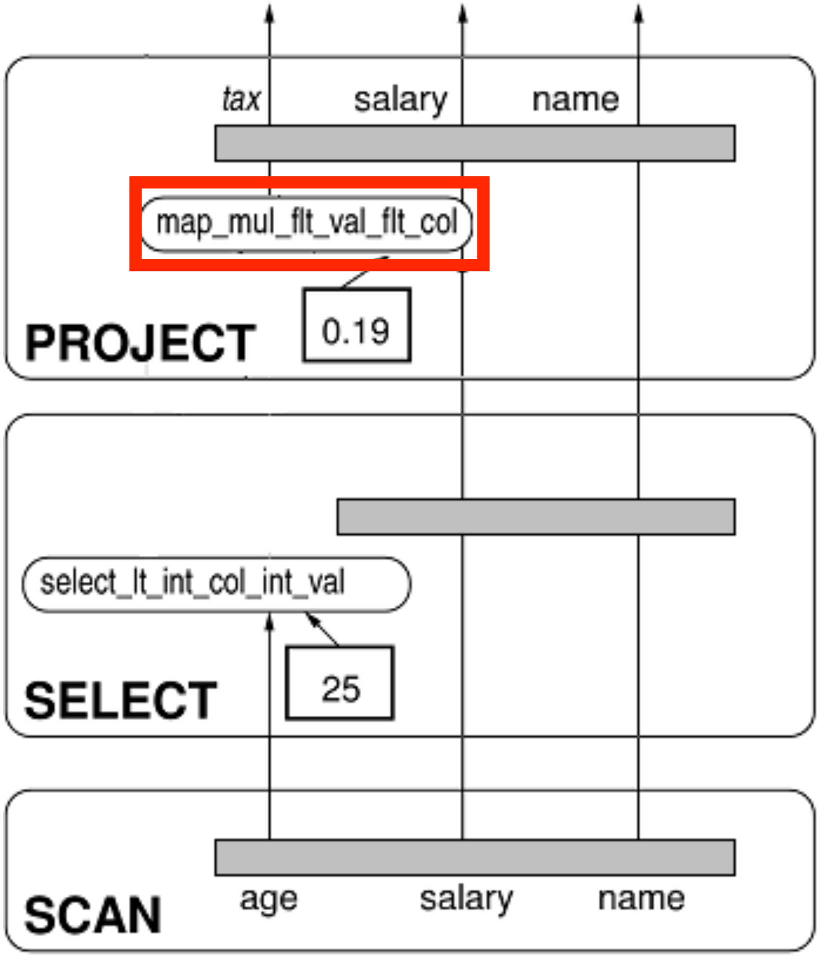


Tuple-at-a-time iterator interface:
- open()
- next(): tuple
- close()

next() is called:
- for each operator
- for each tuple

Complex code repeated over and over

Primitive functions



Provide data-specific computational functionality

Called once for every operation on every tuple.

Even worse with complex tuple representation

Perform one operation (e.g. addition) in one call

DBMS performance - IPT

- Lots of repeated, unnecessary code
  - Operator logic
  - Function calls
  - Attribute access
  - ***Most*** *instructions* ***interpreting*** *a query*
  - ***Very few*** *instructions* ***processing*** *actual data!*
- **High instructions-per-tuple (IPT) factor**

Modern CPUs

- New CPU features over the last 20 years
  - RAM too slow - instruction and data cache
  - Complex CPU pipelines – branch sensitivity
  - Superscalar features – multiple instructions at once
  - SIMD instructions (SSE)
- Great for e.g. multimedia processing…
- … but bad for database code!

DBMS performance - CPI

- CPU-unfriendly code
  - Complex code: function calls, branches
  - Poor use of CPU cache (both data and instructions)
  - Processing one value at a time
  - Compilers can't help much ☹
- **High cycles-per-instruction (CPI) factor**

DBMS performance

- Performance factors:
  - High instructions-per-tuple
  - High cycles-per-instruction
  - **Very high cycles-per-tuple (CPT)**
- Others can do better
  - Scientific computing, mulitmeda, …
- How can we?

MonetDB

- MonetDB – 1993-now, developed at CWI
  - In-memory column store
  - Focused on computational efficiency
- Predecessor of X100

MonetDB: a column store

- "save disk I/O when scan-intensive queries need a few columns"

MonetDB: a column store

- ~~"save disk I/O when scan-intensive queries need a few columns"~~
- "reduce interpretation overheads to improve computational efficiency"

MonetDB in action

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

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

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

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

MonetDB in action

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

MonetDB in action

SELECT **id, name, (age-30)*50 a**
FROM **people**
WHERE **age > 30**

```
int
select_gt_float(    oid* res,
                    float* column
                   float val, int n)
{
    for(int j=0,i=0; i<n; i++)
        if (column[i] >val) res[j++] = i;
    return j;
}
```

CPU Efficiency depends on "nice" code
- no function calls
- few dependencies (control,data)
- compiler support

Compilers love simple loops over arrays
- loop unrolling, loop pipelining
- automatic SIMD

Simple, hard-coded operators

MonetDB: a column store

- ~~"save disk I/O when scan-intensive queries need a few columns"~~
- "reduce interpretation overheads to improve computational efficiency"
  - Hard-coded, specialized operators (thousands!)
  - No function calls
  - Array-based processing

MonetDB problem

```
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(3 nil)

sel_age

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

[-](,30)

tmp

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

MATERIALIZED intermediate results

Materialization problem

- Extra main-memory bandwidth
  - Performance is sub-optimal…
  - … but still faster than anything else (5 years ago ☺)
- Reduces scalability
  - Can’t afford writing to disk
  - **Only effective for limited data sizes and not all query types**

MonetDB: a Faustian Pact

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

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

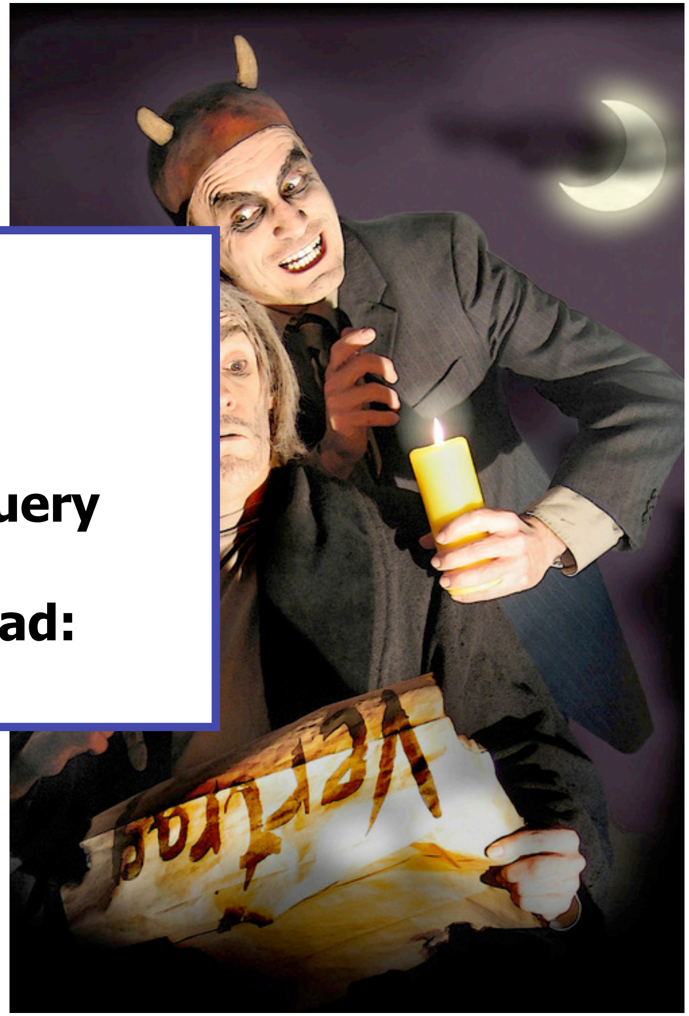
MonetDB: a Faustian Pact

- You want efficiency
  - Simple hard-coded operators
- I take sca
  - Result r

MonetDB
Supports SQL and XQuery

Open-source download:
monetdb.cwi.nl

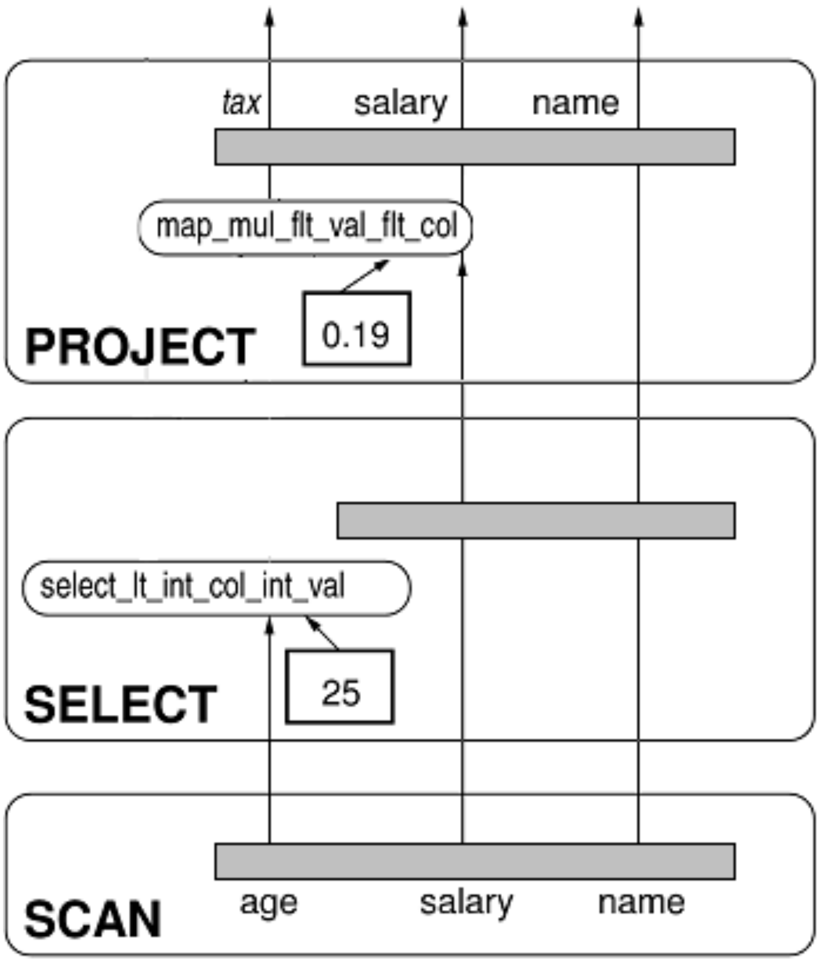
- C pro
- **MonetDB**: 3.7s
- MySQL: 26.2s
- DBMS "X": 28.1s



X100

- My PhD thesis
- Motivation:
  - let's fix MonetDB scalability problem…
  - … and improve the performance on the way ☺
- Core ideas:
  - New execution model
  - High performance column storage

Typical Relational DBMS Engine



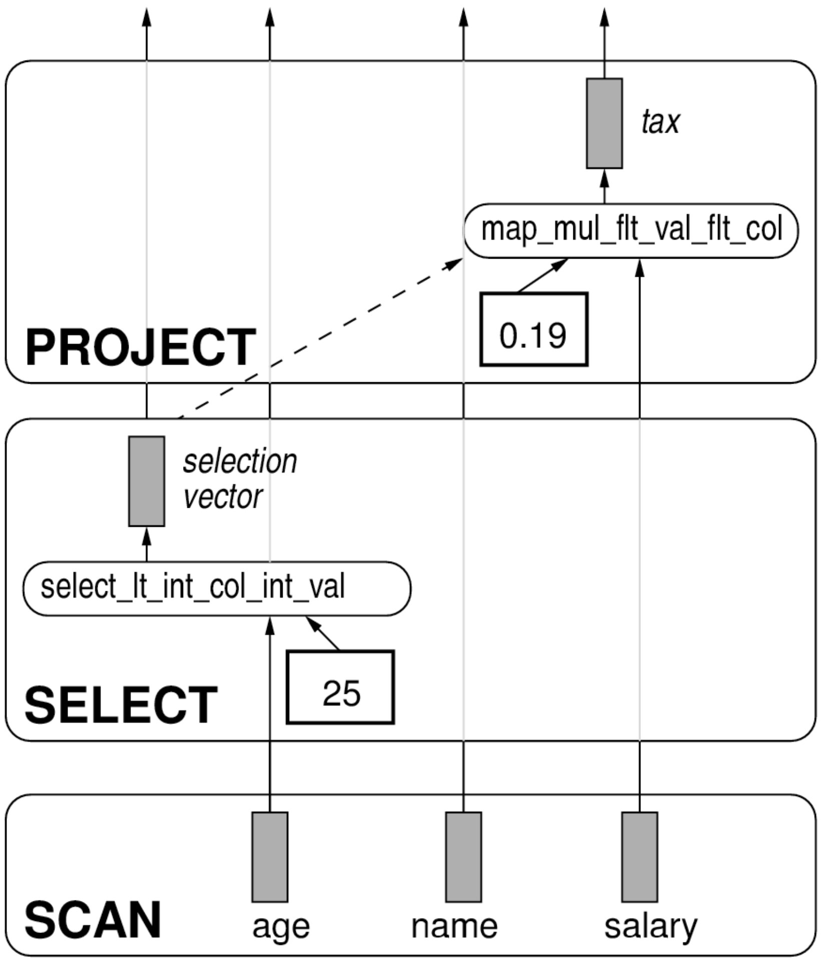
Iterator model is nice

Query

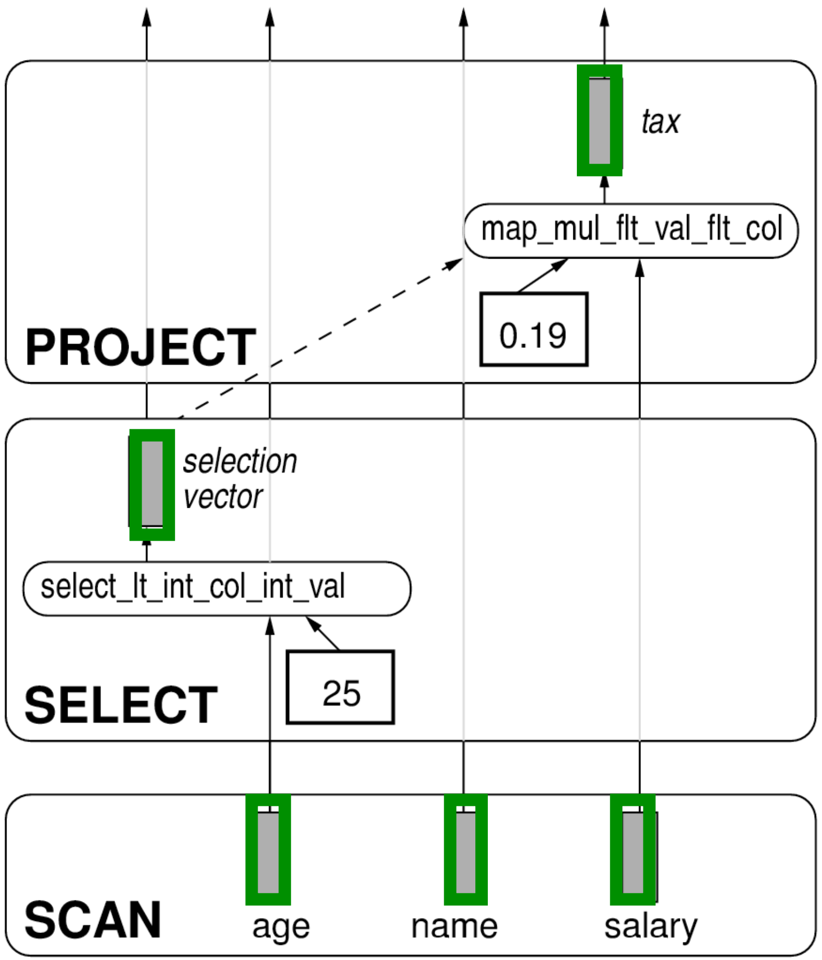
SELECT
name,
salary*.19 AS tax
FROM
employee
WHERE
age > 25

Tuple-based processing is bad

X100: "Vectors"



X100: "Vectors"

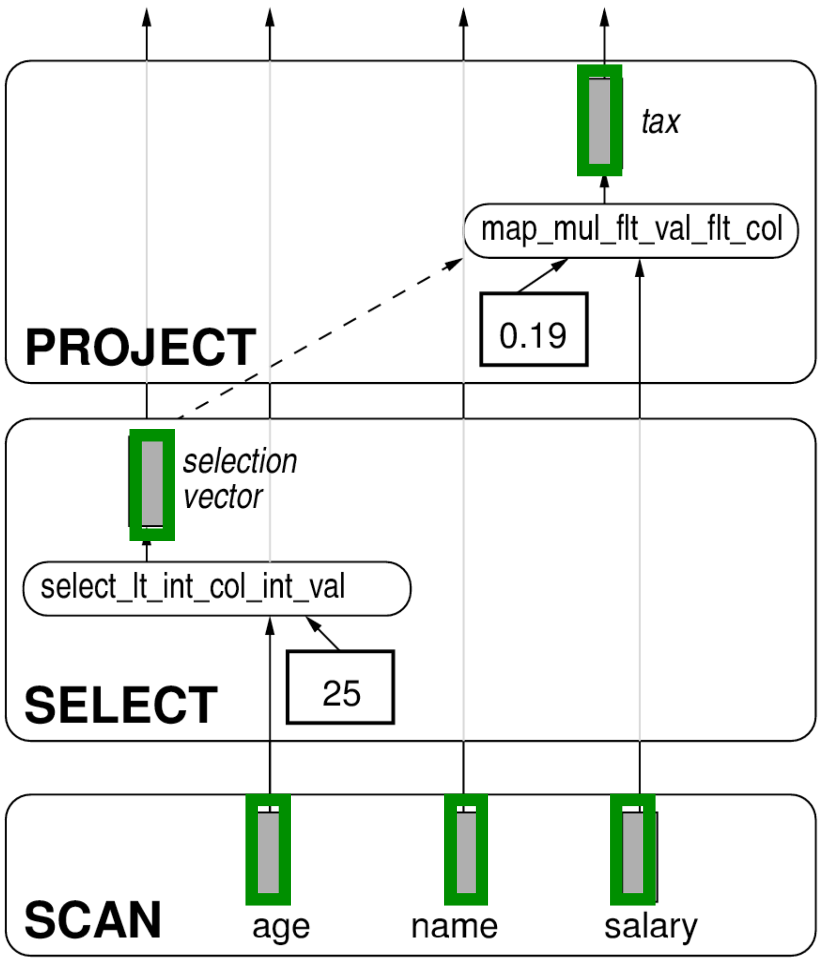


Vector contains data of ***multiple*** tuples (~100-1000)

All operations consume and produce entire vectors

Effect: much less operator.next() and primitive calls.

Vectors



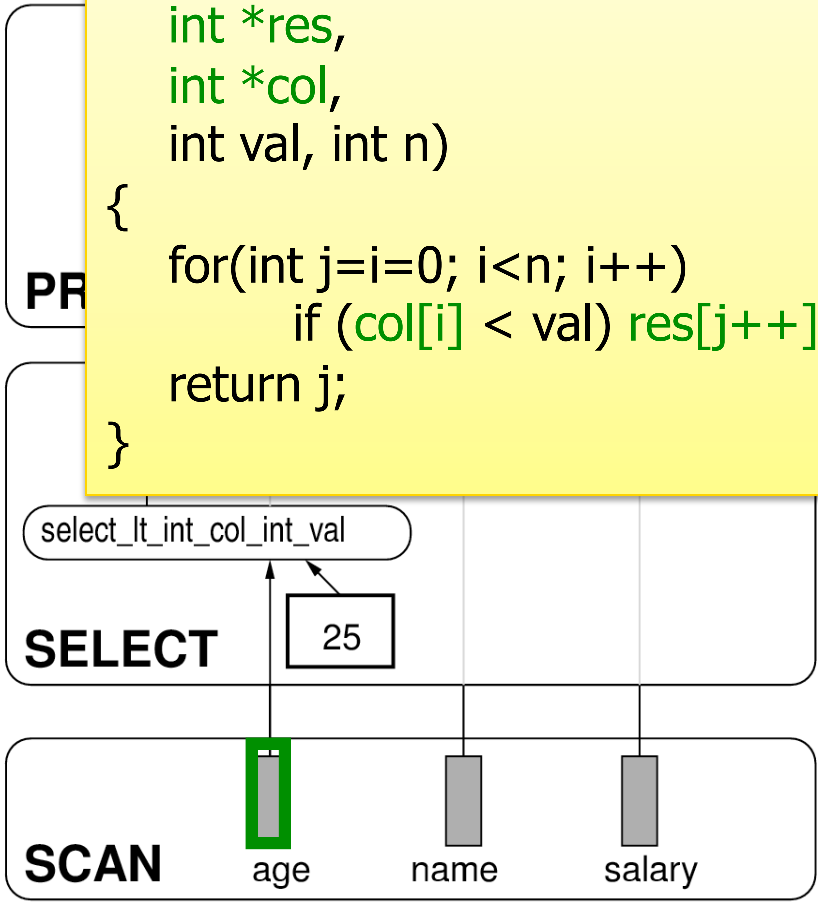
Column slices as unary arrays

Not because:
Columns are better for storage than rows
(though we still think it often is)

But because:
- simple and efficient
- SIMD friendly layout
- **Assumed cache-resident**

Vectorized Primitives

```
int
select_lt_int_col_int_val (
    int *res,
    int *col,
    int val, int n)
{
    for(int j=i=0; i<n; i++)
        if (col[i] < val) res[j++] = i;
    return j;
}
```



Most primitives take just **0.5 (!) to 10** cycles per tuple

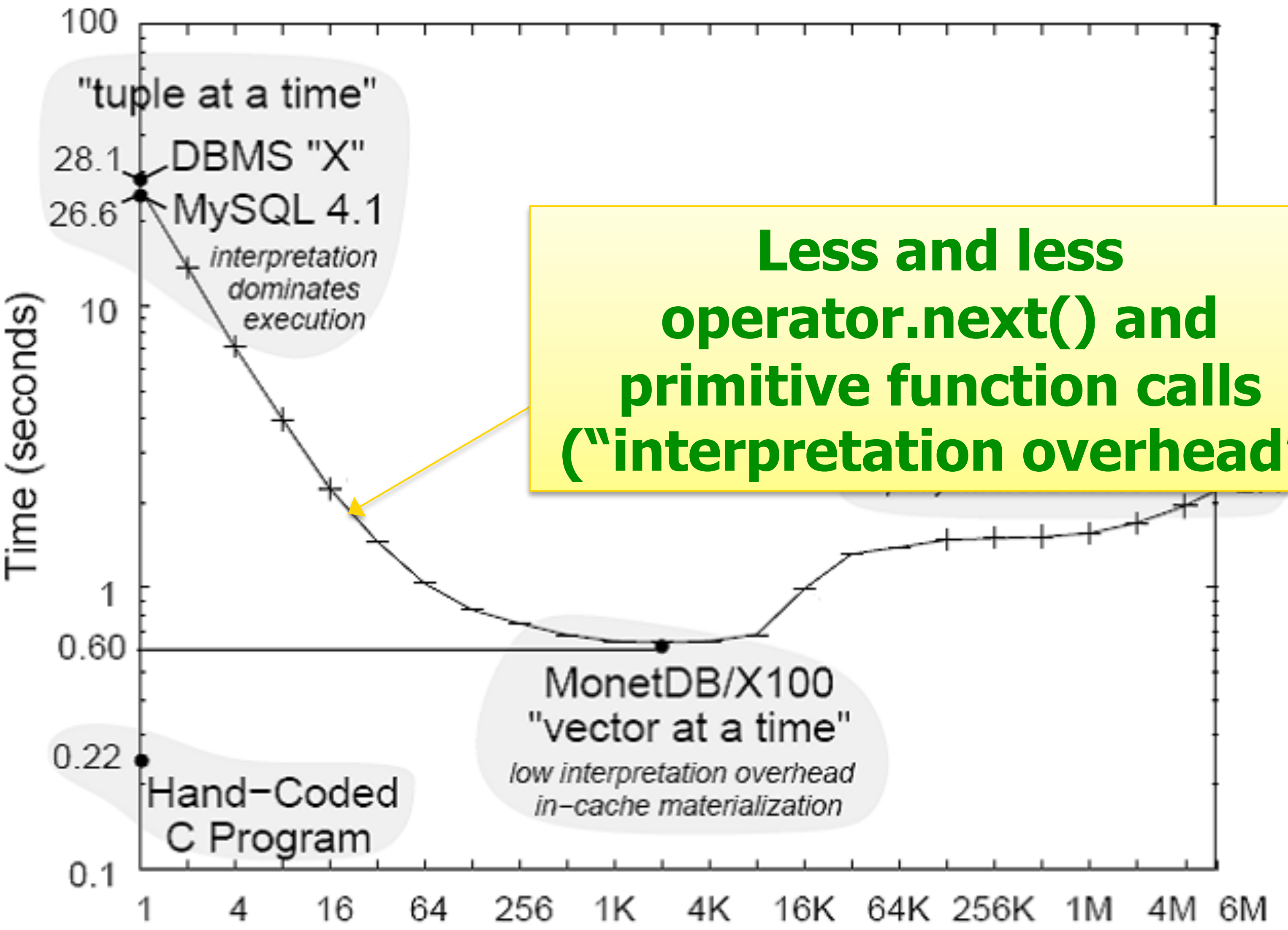
10-100+ times faster than tuple-at-a-time

X100

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

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

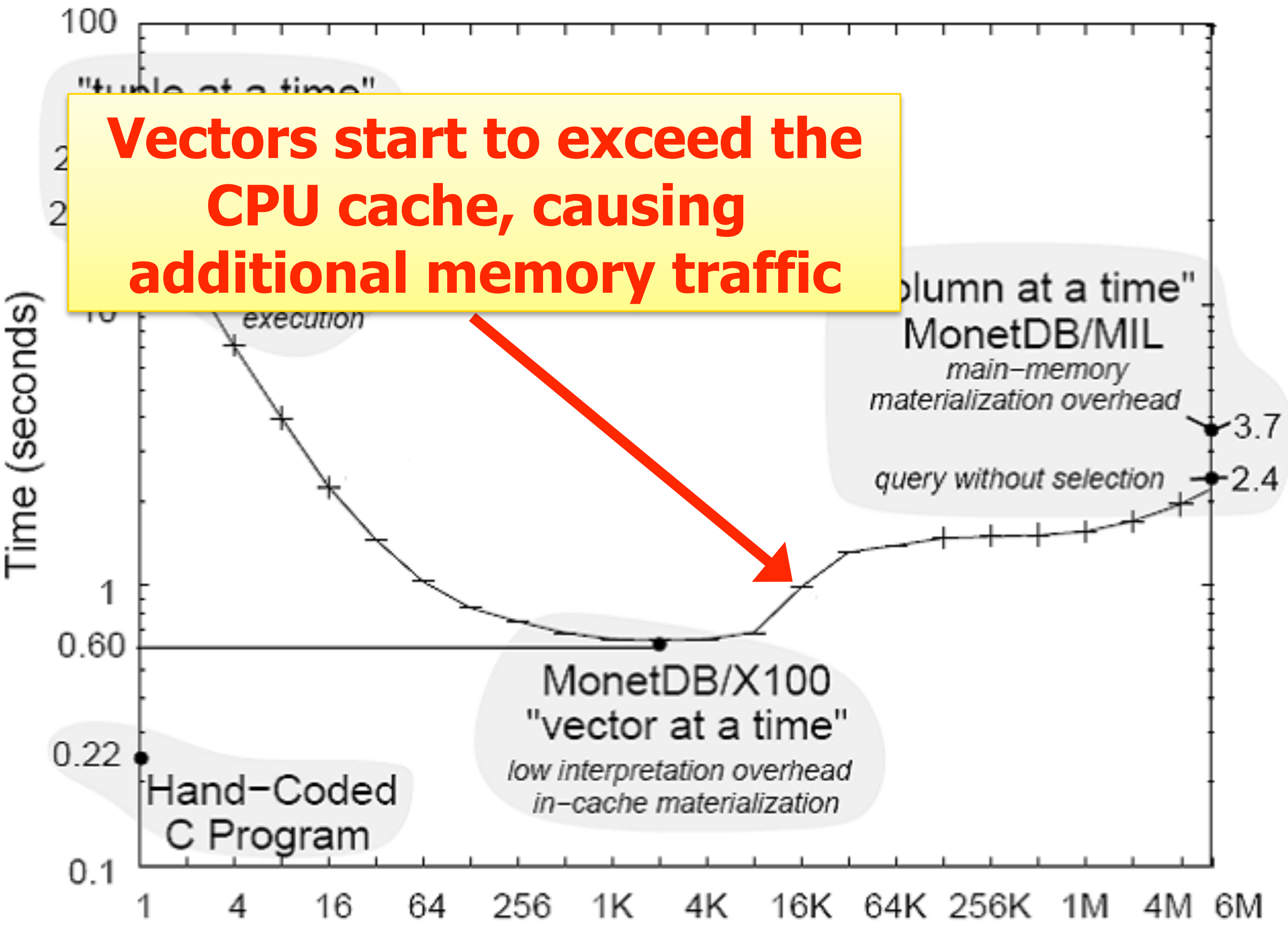
Varying the Vector size



Less and less operator.next() and primitive function calls ("interpretation overhead")

Varying the Vector size

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



Why is X100 so fast?

- **Reduced interpretation overhead**
  - 100+ times fewer function calls
- **Good CPU cache use**
  - High locality in the primitives
  - Cache-conscious algorithms
- **No Tuple Navigation**
  - Primitives only see arrays
- **Vectorization allows algorithmic optimization**
- **CPU and compiler-friendly function bodies**
  - Multiple work units, loop-pipelining, SIMD…

Feeding the Beast

X100 uses ~100 cycles per tuple for TPC-H Q1

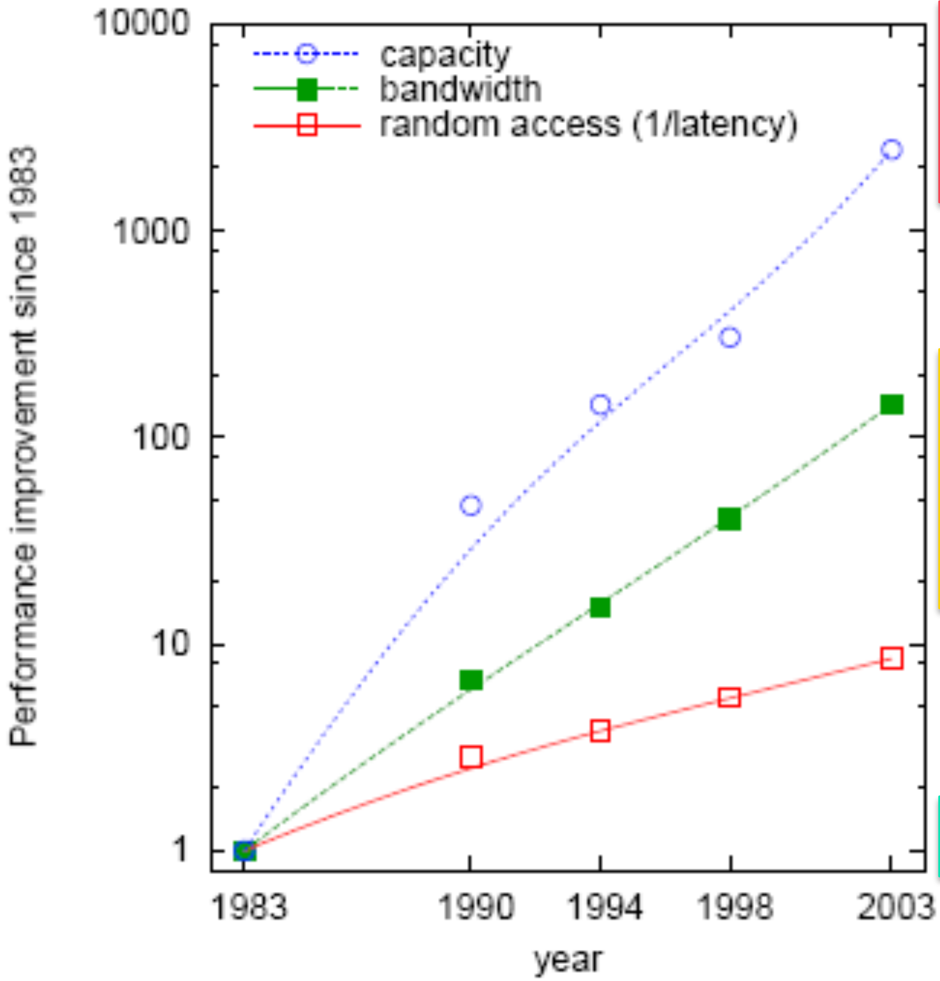
- Q1 has ~30 bytes of used columns per tuple
- 3GHz CPU core

  eats 900MB/s

No problem for RAM

But disk-based data?

Using Disk in the 21th century



Databases traditionally depend on **secondary indices** resulting in many random disk accesses

Poor **random disk acces**s needs to be compensated with more and more disk heads.
(tens, hundreds… thousands!)

Focus on **scanning**!

Feeding the Beast (1)

Two ideas pursued:

- Lightweight compression to enhance disk bandwidth
- Maximizing disk scan sharing in concurrent queries.

Compression to improve I/O bandwidth

- 0.9GB/s query consumption
- 1/3 CPU for decompression → 1.8GB/s needed

Algorithm	Decompression Bandwidth
BZIP	10 MB/s
ZLIB	80 MB/s
LZO	300 MB/s

→ new lightweight compression schemes

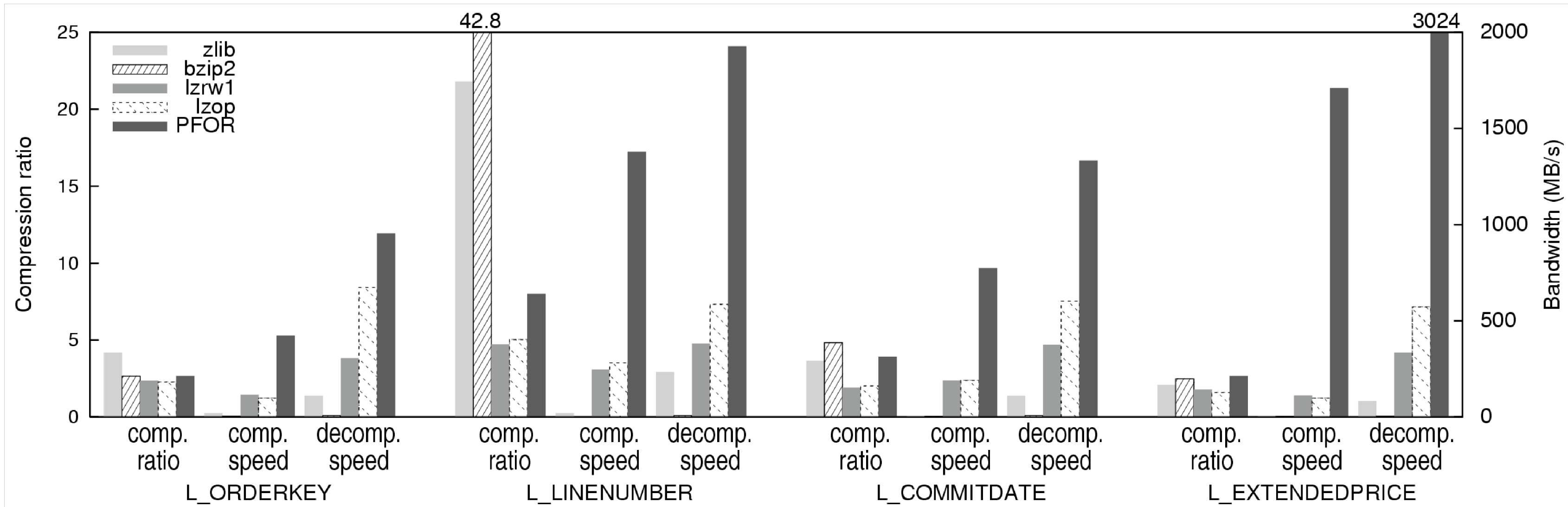
Key Ingredients

- Compress relations on a per-column basis
  - Easy to exploit redundancy
- Keep data compressed in main-memory
  - More data can be buffered
- Decompress vector at a time
  - Minimize main-memory overhead
- Use light-weight, CPU-efficient algorithms
  - Exploit processing power of modern CPUs

CPU-friendly decompression

- Tuples classified into “hits” and “misses”

```
void decompress(size_t n, char* in, int *out, int *misses,  int first_miss)
  for (i =0; i < n; i++)               // decode all values
    out[i] = DECODE( in[i] );       // including misses
  for (i = first_miss, j = 0; i < n; i += in[i])   // patch misses
    out[i] = misses[j++];          // using exception table
```



TPC-H 100 GB

Decent improvement with fast disks

TPC-H query	X100 on 1 CPU					DB2 – 8 CPUs
	Compression ratio	4 disks		12 disks		142 disks
		Speedup	Time (s)	Speedup	Time (s)	Time (s)
01	4.33	4.41	69.6	1.29	50.9	111.9
03	3.04	3.10	11.3	1.48	6.0	15.1
04	8.15	7.58	2.4	2.67	1.8	12.5
05	3.81	3.55	15.3	1.06	16.2	84.0
06	4.39	4.50	10.7	2.35	4.6	17.1
07	1.71	1.66	72.0	0.84	40.8	86.5

Linear speedup with slow disks

Competes with DB2 using ~10x less resources

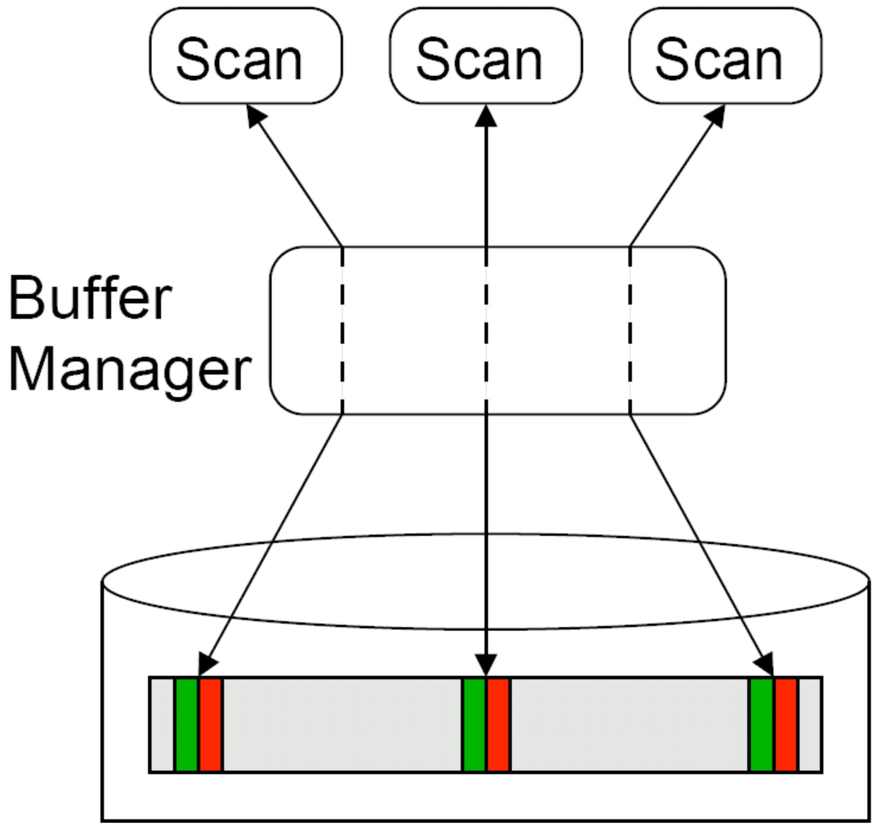
Feeding the Beast (2)

Two ideas pursued:

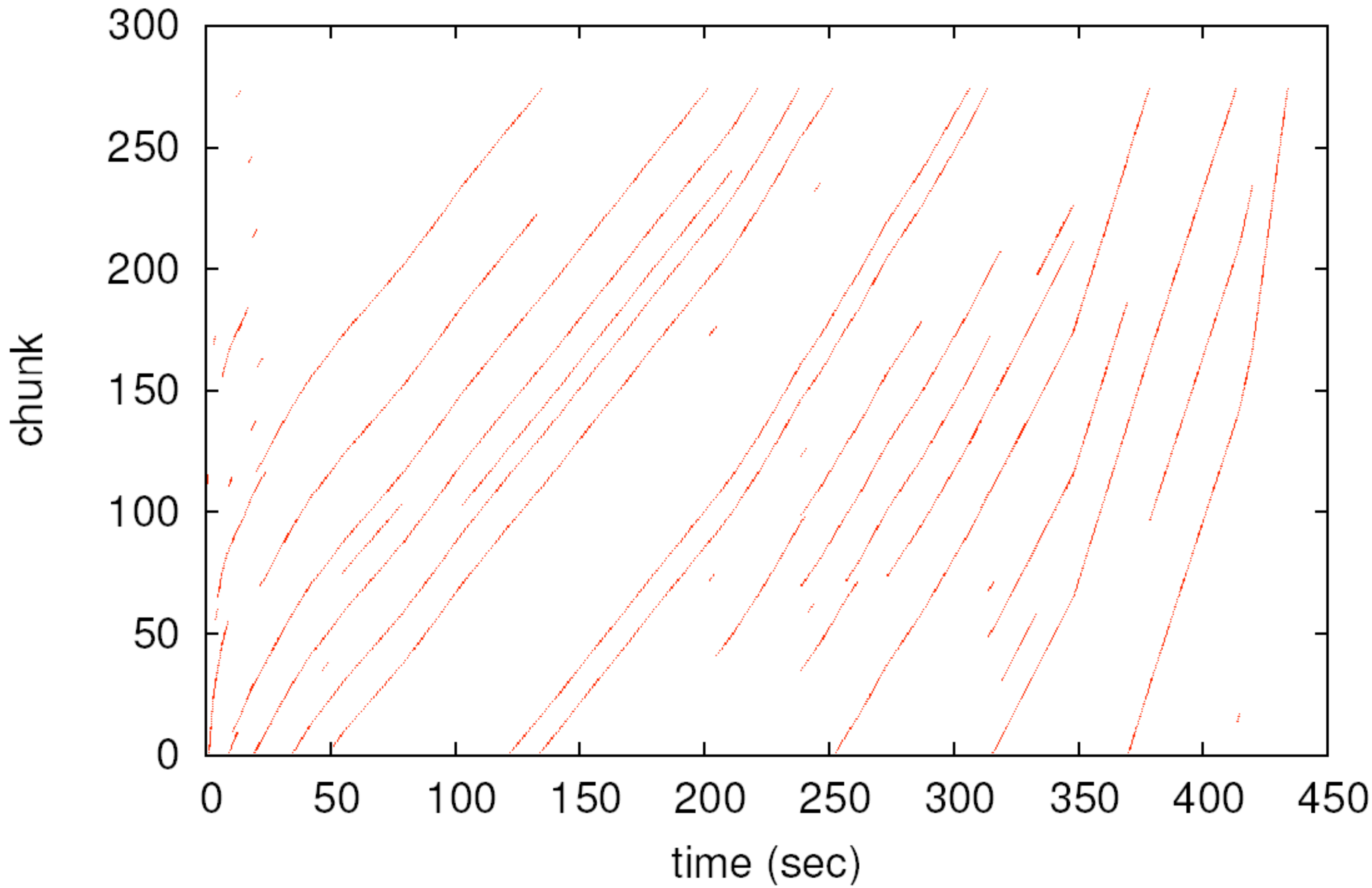
- Lightweight compression to enhance disk bandwidth
- Maximizing disk scan sharing in concurrent queries.

Concurrent scans

- Multiple queries scanning the same table
  - Different start times
  - Different scan ranges
- Compete for disk access and buffer space
- FCFS request scheduling: poor latency

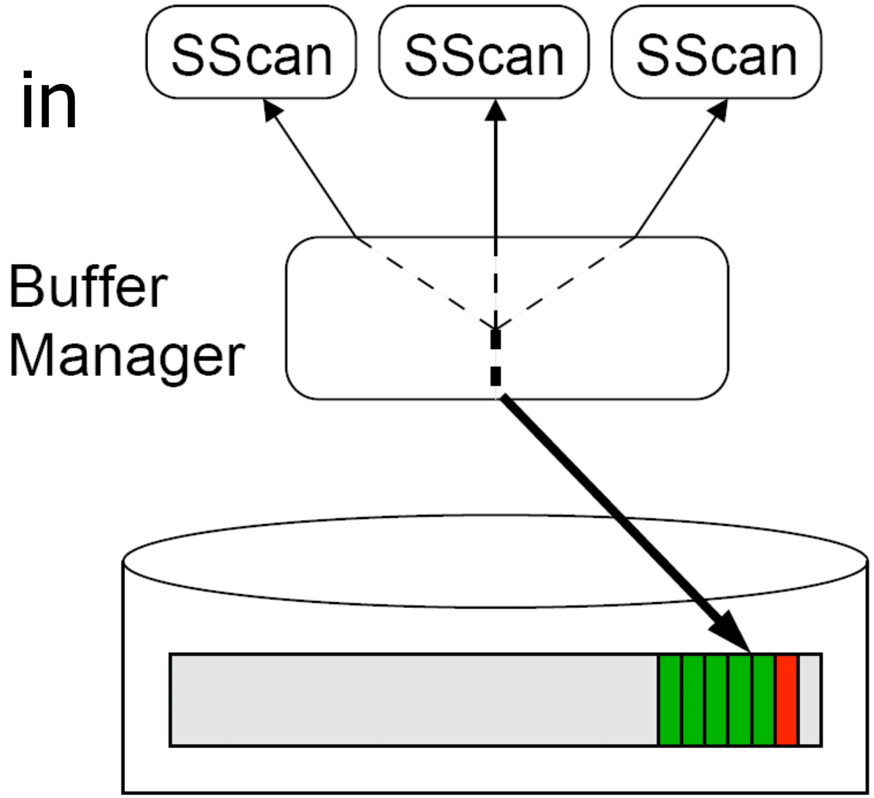


“Normal” scans in real life

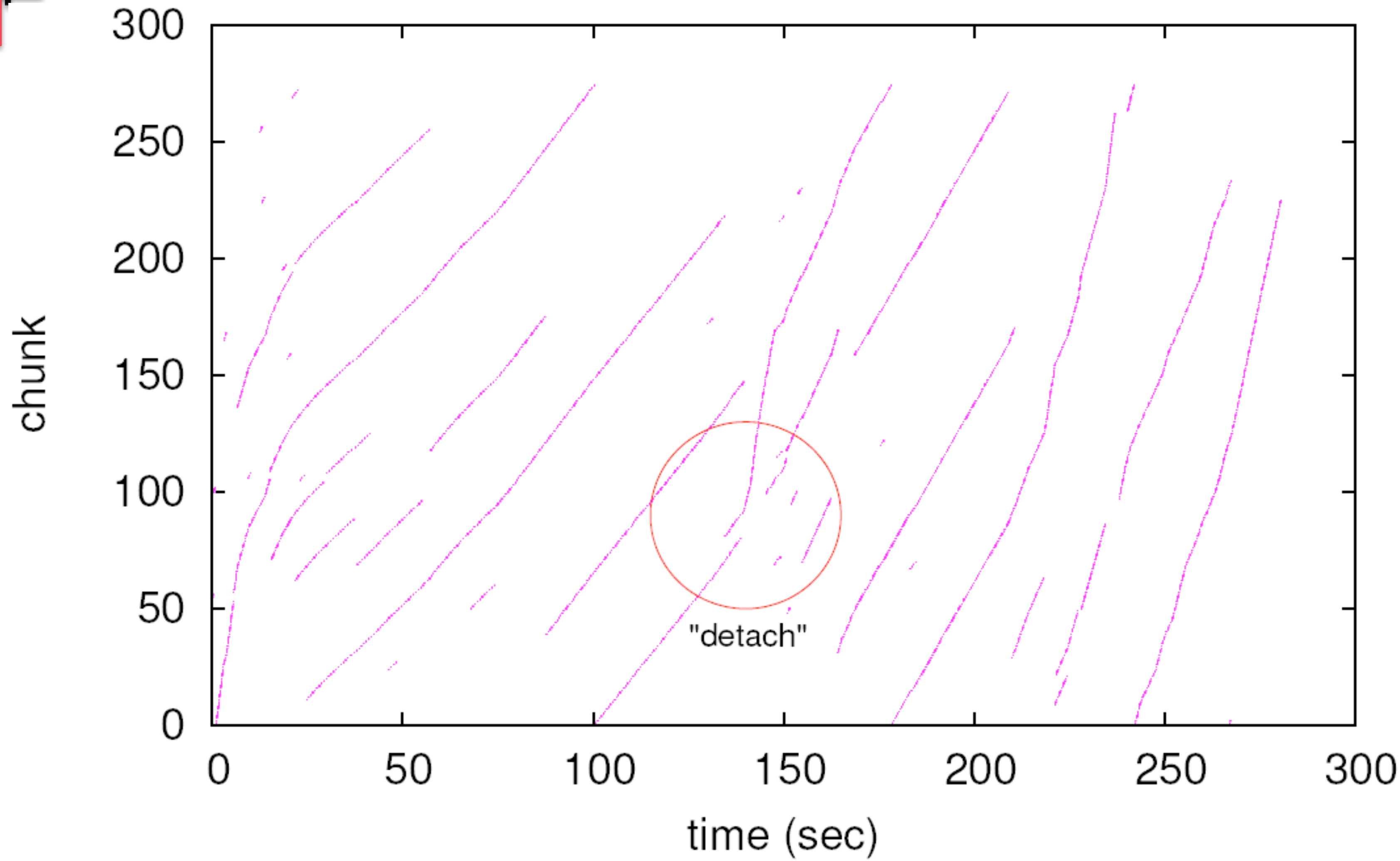


Shared scans

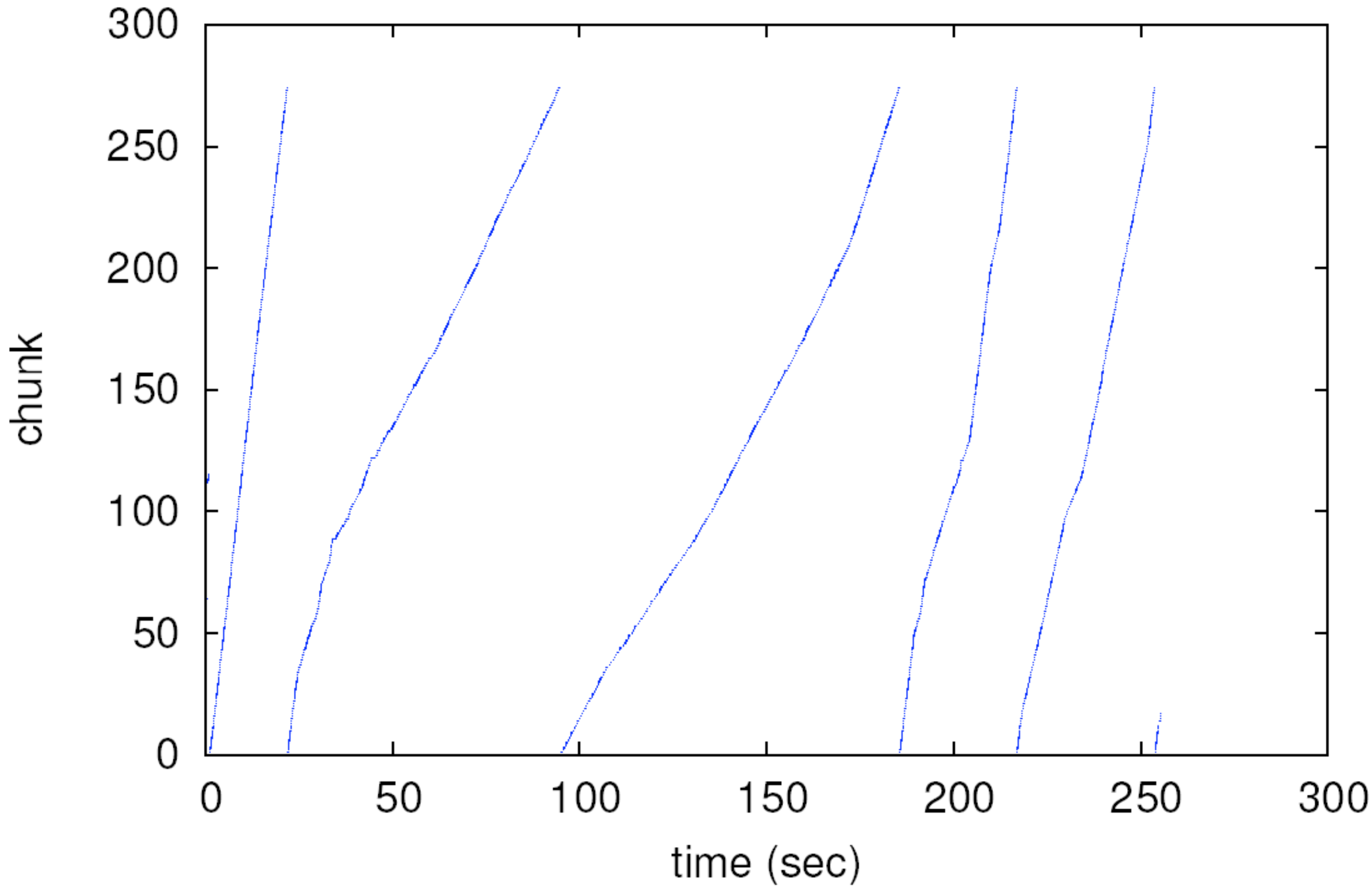
- Observation: queries often do not need data in a sequential order
- Idea: make queries "share" the scanning process
- Two existing types:
  - Attach
  - Elevator



“Attach” in real life



"Elevator" in real life



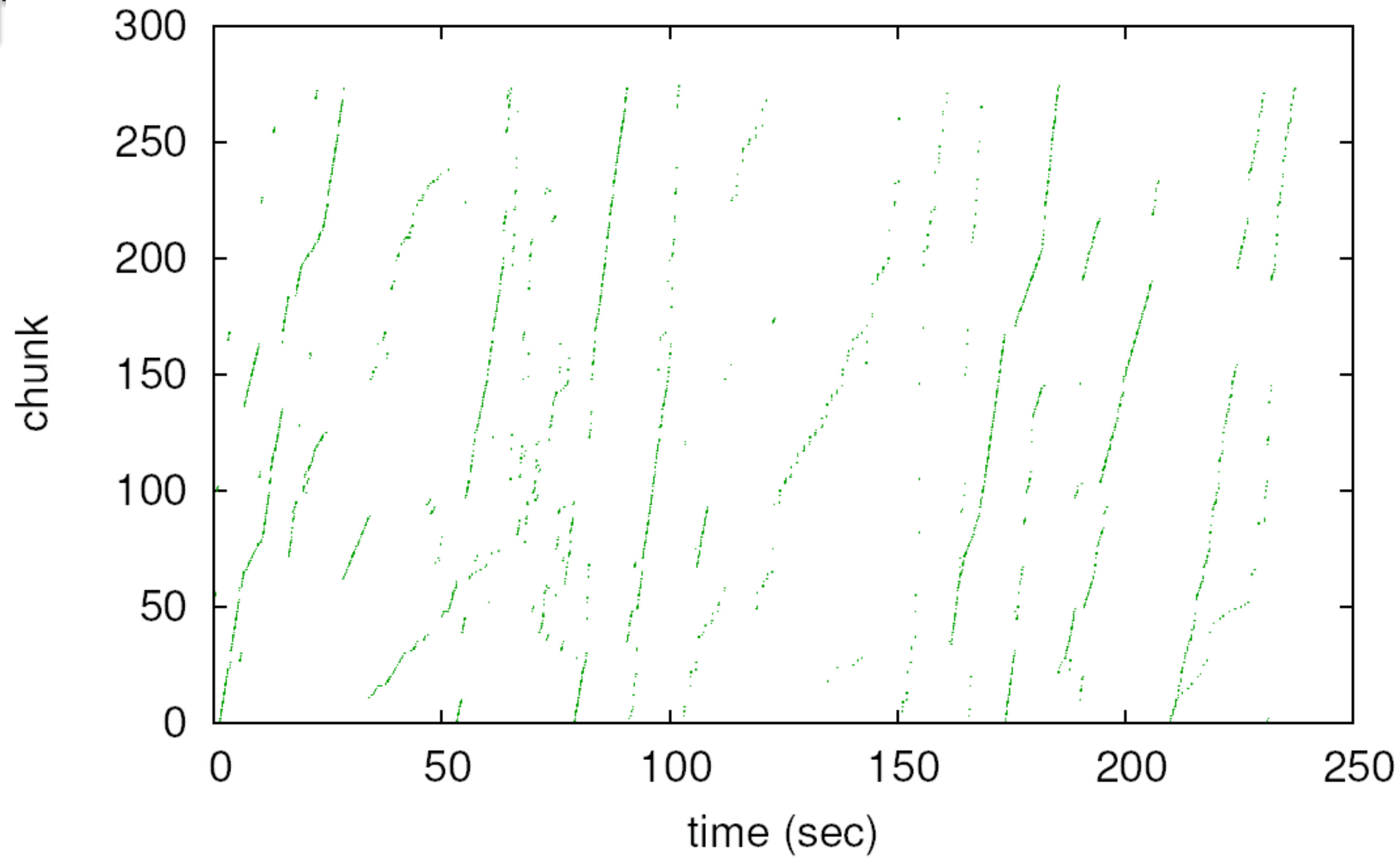
Existing shared scans

- Benefits
  - Less I/O operations
  - Better data reuse
- Problems
  - Sharing decisions static (when a query starts)
  - Misses opportunities in a dynamic environment
  - Not sensitive to different query types

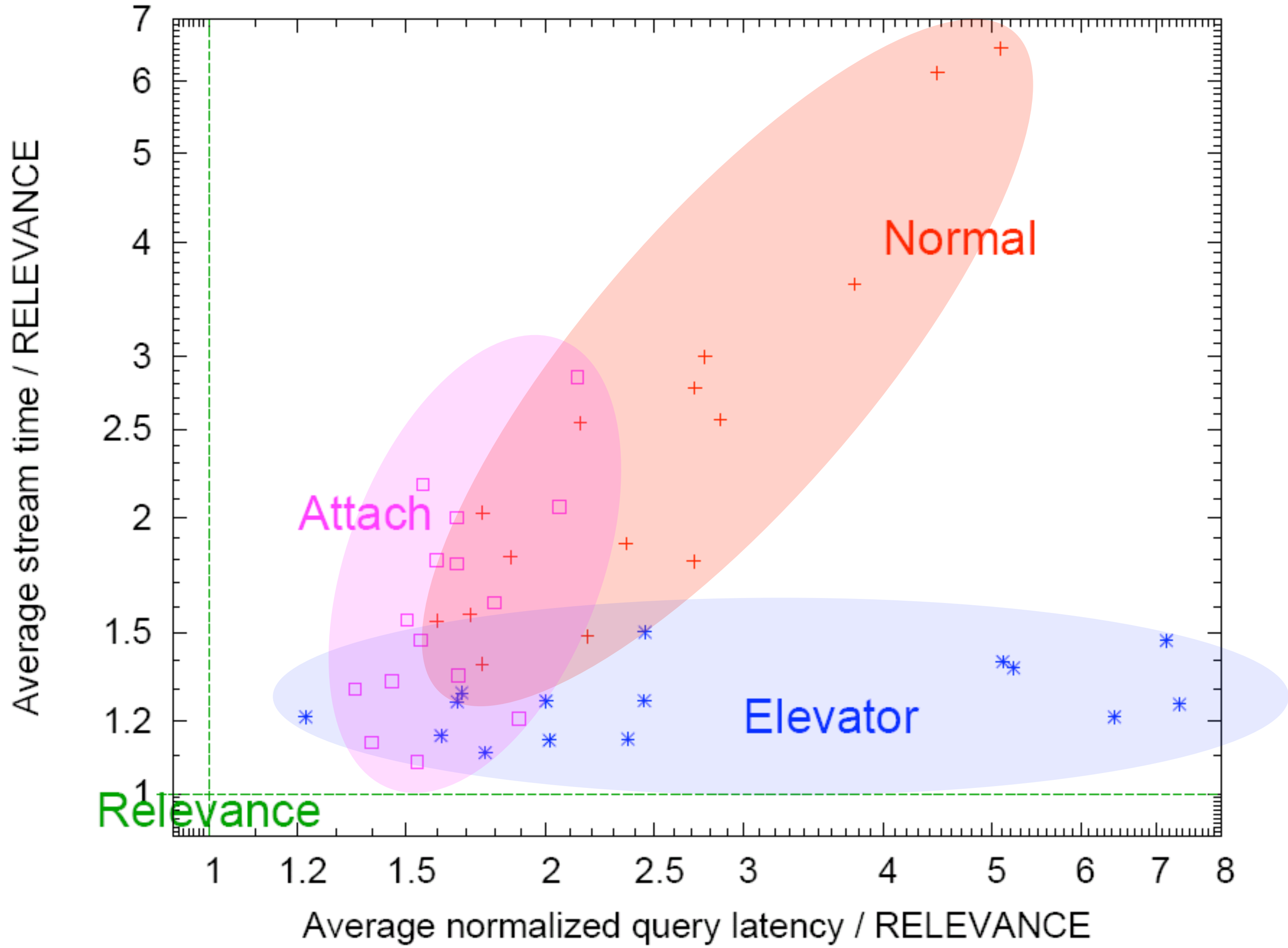
“Relevance” scans

- Core ideas
  - Dynamically adapt to the current situation
  - Allow fully arbitrary data order
- Goals:
  - Maximize data sharing
  - Optimize latency and throughput
  - Work for different types of queries

"Relevance" in real life



Results



Normal
Attach
Elevator
Relevance
Average stream time / RELEVANCE
Average normalized query latency / RELEVANCE
7
6
5
4
3
2.5
2
1.5
1.2
1
1
1.2
1.5
2
2.5
3
4
5
6
7
8

Conclusions

- Presented X100
  - A new database kernel
  - Uses block-oriented iterator model (vectorization)
    - works amazingly well
- So fast, must reduce hunger for hard disk bandwidth
  - Column storage specialized in sequential access
  - + Lightweight compression schemes (give ~~ factor 3)
  - + Cooperative bandwidth sharing (gives ~~ factor 2)
- Good performance results
  - Fastest raw 100GB TPC-H performance around (** not fair)
  - Beats IR systems on Terabyte TREC

Literature

- Monet:
  - *"Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications"*
    P.Boncz, PhD thesis 2002
    http://old-www.cwi.nl/htbin/ins1/publications?request=intabstract&key=Bo:DISS:02
- X100:
  - *"MonetDB/X100: A DBMS In The CPU Cache"*
    M.Zukowski, P.Boncz, N.Nes, S.Heman, DeBull 2005
  - *"MonetDB/X100: Hyper-Pipelining Query Execution"*
    P.Boncz, M.Zukowski, N.Nes, CIDR 2005
  - *"Super-Scalar RAM-CPU Cache Compression"*
    M.Zukowski, S.Heman, N.Nes, P.Boncz, ICDE 2006
  - *"Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS"*
    M.Zukowski, S.Heman, N.Nes, P.Boncz, VLDB 2007
  - All these and more available at http://homepages.cwi.nl/~marcin/

The End

Thank you!

Questions?

(If too shy to ask now, write to marcin@cwi.nl)