MonetDB / X100

state-of-the-art / future
high-performance DBMS kernels from CWI

Peter Boncz,
Niels Nes, Stefan Manegold, Marcin Zukowski,
Sjoerd Mullender, Martin kersten

Centrum voor Wiskunde en Informatica (CWI)
Amsterdam
The Netherlands

boncz@cwi.nl

Contents

- What is MonetDB/X100
- Open-Source MonetDB Status
- MonetDB Internals
  - Architecture, Data Model, Query Language
  - Cache-Conscious Query Processing Algorithms
- Research Direction
  - MonetDB Evaluation
  - X100: new technology inside MonetDB
- Reference Material

What is Monet?

- CWI prototype DBMS
  - my Ph.D. subject (due 31 may 2002)
  - open source version is called **MonetDB**

Monet: Applied CS Research

- a decade of “query-intensive” application experience
  - *image retrieval*: Peter Bosch ➔ ImageSpotter
  - *audio/video retrieval*: Alex van Ballegooij ➔RAM
  - *XML text retrieval*: de Vries / Hiemstra ➔TIJAH
  - *biological sequences*: Arno Siebes ➔ BRICKS
  - *XML databases*: Albrecht Schmidt ➔ XMARK
    Grust / vKeulen ➔ Pathfinder
  - *GIS*: Wilco Quak ➔ MAGNUM
  - *data warehousing / OLAP / data mining*
    SPSS ➔ DataDistilleries
    Univ. Massachussetts ➔ PROXIMITY

Monet research group successfully spun off and grew DataDistilleries

What is X100?

- New High-Perf DBMS kernel technology
  - “beyond cache-conscious processing” / “throughput computing”
  - focus on minimizing IPT (CPU instructions per tuple)
  - exploit SMT parallelism & other new hardware features
  - scaling out to non-memory resident problems (huge RAID arrays)
- will be part of future MonetDB versions
  - gradual phase-in
  - will re-use Monet front-ends (SQL,RAM,..), plus all MonetDB APIs

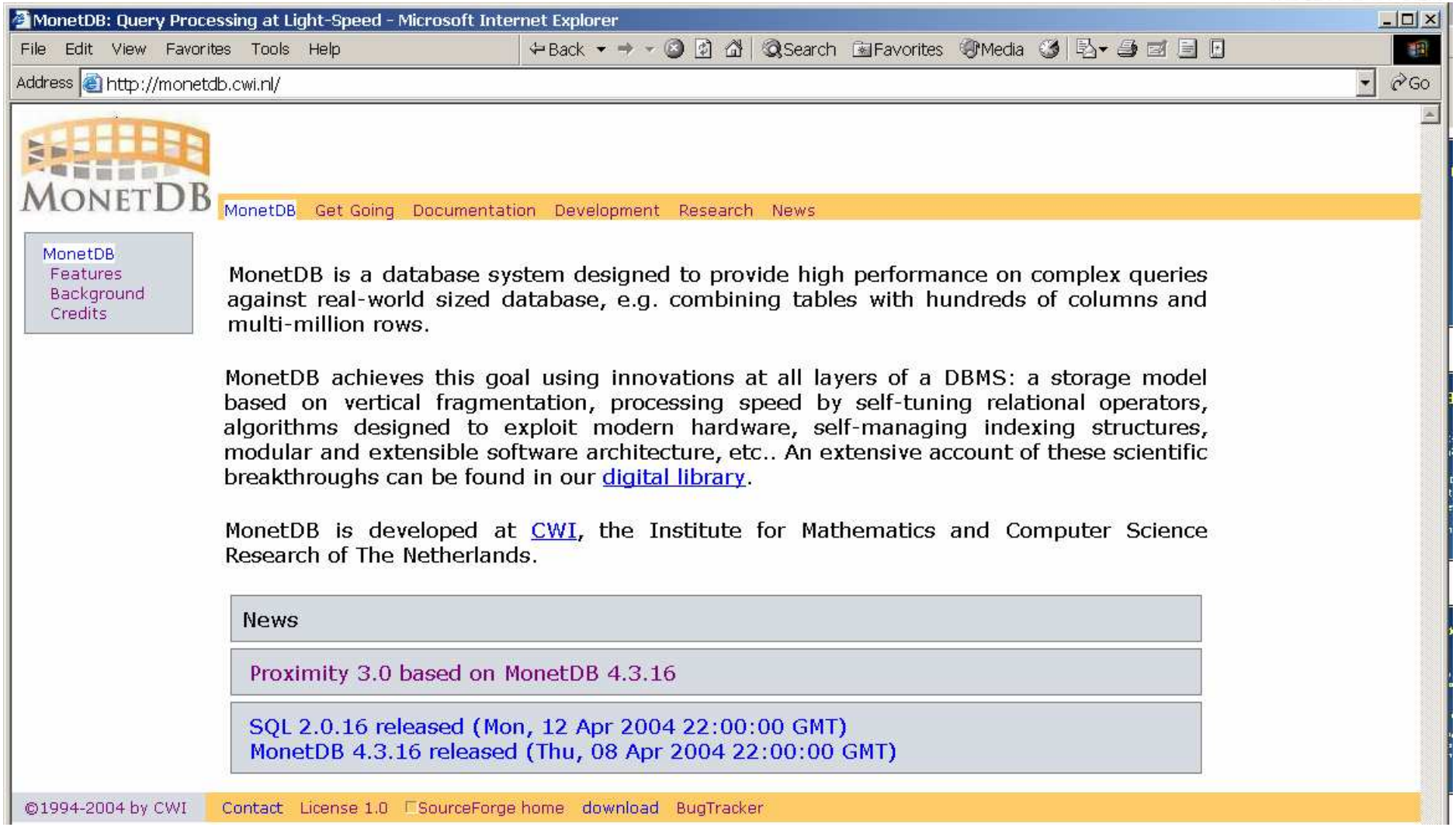
Pronunciation: “Times-100”

MonetDB Open Source

- MonetDB on Sourceforge
- Project Homepage
  - http://monetdb.cwi.nl
- Developers website:
  - http://sf.net/projects/monetdb

RoadMap

- 14-apr-04: initial Beta release
- 30-sep-04: first official release


MonetDB: Query Processing at Light-Speed - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://monetdb.cwi.nl/
Go
MONETDB
MonetDB Get Going Documentation Development Research News
MonetDB
Features
Background
Credits
MonetDB is a database system designed to provide high performance on complex queries against real-world sized database, e.g. combining tables with hundreds of columns and multi-million rows.
MonetDB achieves this goal using innovations at all layers of a DBMS: a storage model based on vertical fragmentation, processing speed by self-tuning relational operators, algorithms designed to exploit modern hardware, self-managing indexing structures, modular and extensible software architecture, etc.. An extensive account of these scientific breakthroughs can be found in our digital library.
MonetDB is developed at CWI, the Institute for Mathematics and Computer Science Research of The Netherlands.
News
Proximity 3.0 based on MonetDB 4.3.16
SQL 2.0.16 released (Mon, 12 Apr 2004 22:00:00 GMT)
MonetDB 4.3.16 released (Thu, 08 Apr 2004 22:00:00 GMT)
©1994-2004 by CWI
Contact License 1.0 SourceForge home download BugTracker

MonetDB/SQL

SQL to MIL frontend

- Most of SQL-92
- Some of SQL-99
- Nested sub-queries
- Join indices for foreign keys
- Only simple heuristic optimisations (still)

MonetDB APIs

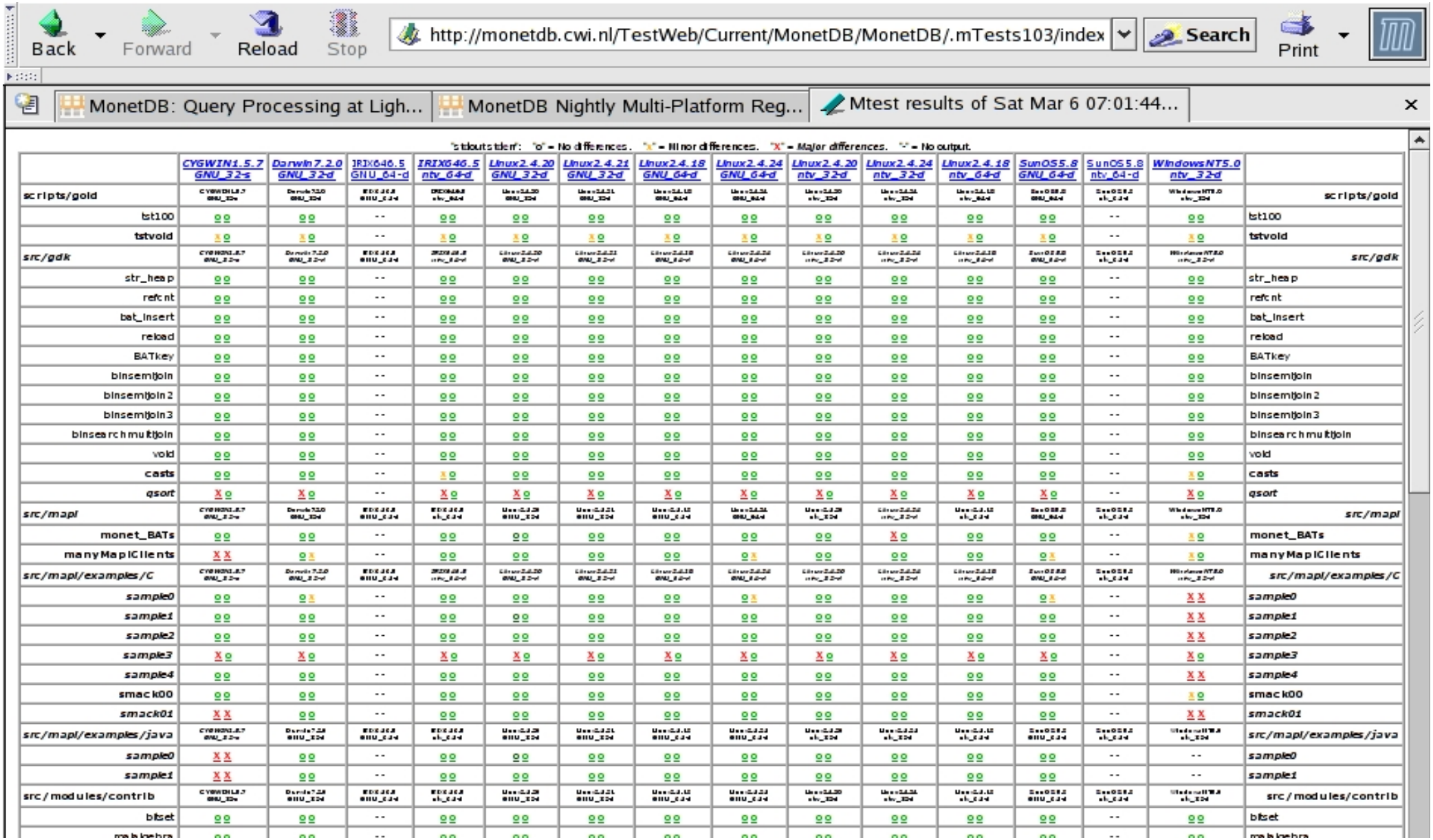
new APIs

- ODBC
- JDBC
- php
- Perl/DBI
- Python
- Haskell

Mknife

- graphical database browser for MonetDB

Nightly Testing



Contents

- What is MonetDB/X100
- Open-Source MonetDB Status
- MonetDB Internals
  - Architecture, Data Model, Query Language
  - Cache-Conscious Query Processing Algorithms
- Research Direction
  - MonetDB Evaluation
  - X100: new technology inside MonetDB
- Reference Material

MonetDB Motivation (1/2)

- relational DBMS dominates the scene
  - Oracle, SQLserver, DB2
- databases a solved problem?

MonetDB Motivation

- relational DBMS dominates the scene
    - Oracle, SQLserver, DB2
- databases a solved problem? **No!**

Problems:

- performance
    - new ‘query intensive’ applications (data mining, et al)
- extensibility
    - new applications (GIS,text,image,audio,video,XML)

MonetDB Motivation (2/2)

- are relational DBMS fit for the job?
    - developed in end 1970's begin 1980's

Motivation (2/2)

- are relational DBMS fit for the job?
    - developed in end 1970's begin 1980's

- hardware has changed
    - CPUs get faster but more vulnerable
    - capacity and bandwidth follows Moore's law
    - latency becomes a bottleneck (I/O **and** RAM)

Motivation (2/2)

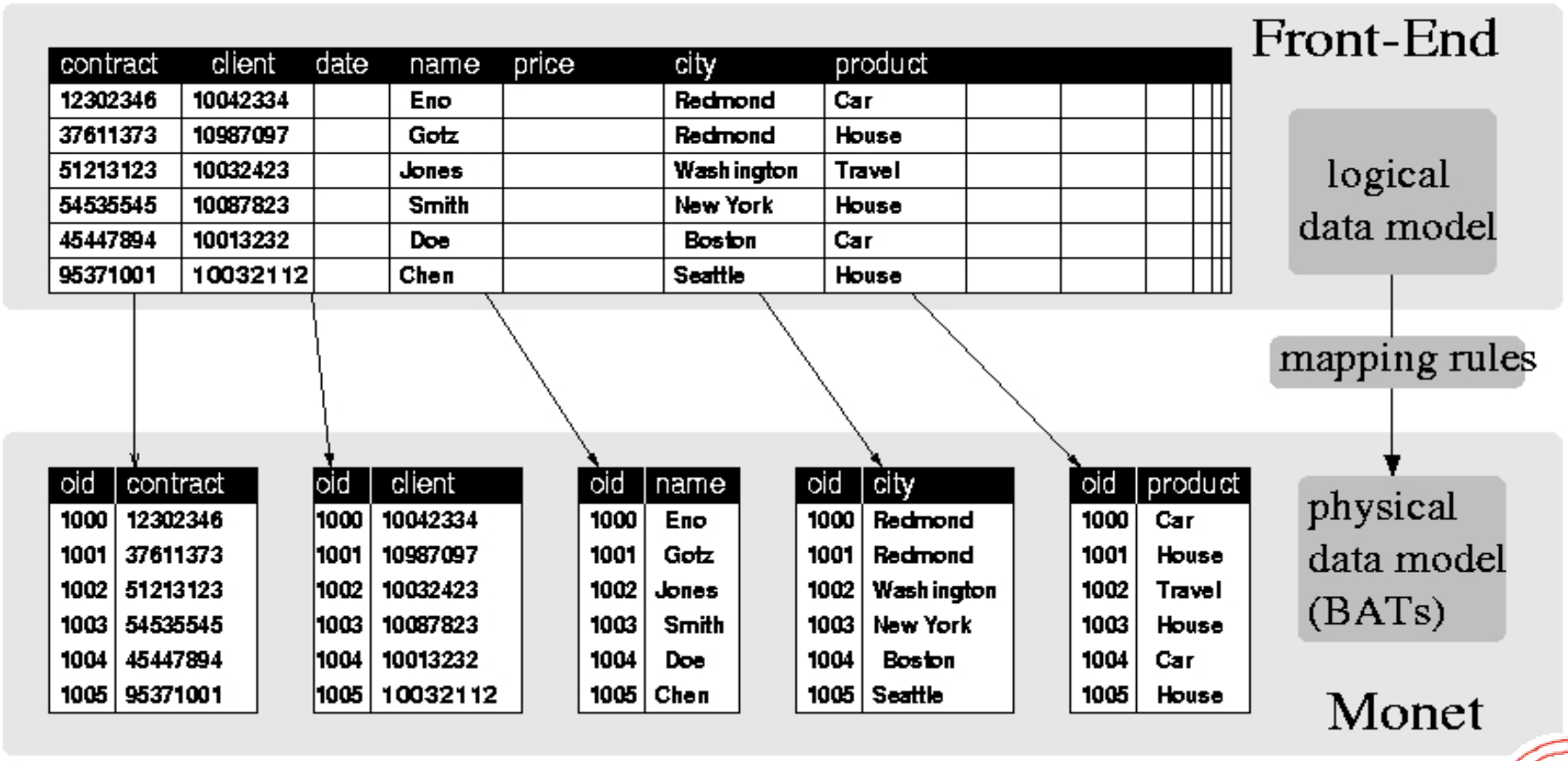
- are relational DBMS fit for the job?
  - developed in end 1970's begin 1980's

- hardware has changed
  - CPUs get faster but more vulnerable
  - capacity and bandwidth follows Moore's law
  - latency becomes a bottleneck (I/O **and** RAM)
- applications have changed
  - RDBMS tuned for transaction processing
  - not query-intensive
  - focuses on business domain mostly

How is Monet Different

- full vertical fragmentation: always!
  - everything in binary (2-column) tables
  - prevents expensive table scans in OLAP and Data Mining
- the RISC approach to databases
  - simple data model, simple query language, KISS KISS KISS…
  - no-freedom algebra primitives ➔ no interpreter ➔ happy compiler/CPU
  - explicit transaction management ➔ DIY approach to ACID
- CPU and memory cache optimized
  - programming team experienced in main memory DBMS techniques
  - use of scientific programming optimizations in DBMS kernel

Binary Association Tables (BATs)



contract	client	date	name	price	city	product
12302346	10042334		Eno		Redmond	Car
37611373	10987097		Gotz		Redmond	House
51213123	10032423		Jones		Washington	Travel
54535545	10087823		Smith		New York	House
45447894	10013232		Doe		Boston	Car
95371001	10032112		Chen		Seattle	House

oid	contract
1000	12302346
1001	37611373
1002	51213123
1003	54535545
1004	45447894
1005	95371001

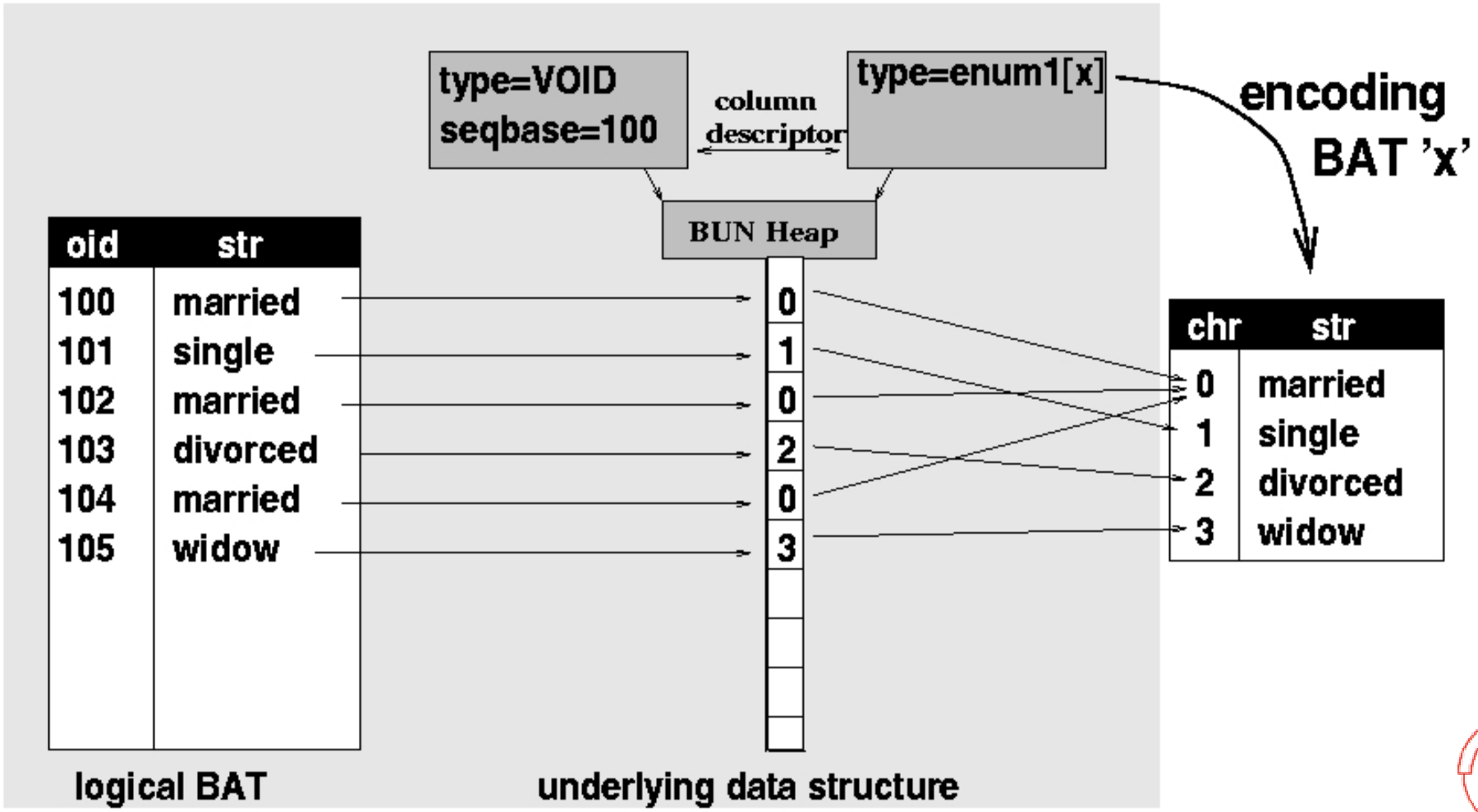
oid	client
1000	10042334
1001	10987097
1002	10032423
1003	10087823
1004	10013232
1005	10032112

oid	name
1000	Eno
1001	Gotz
1002	Jones
1003	Smith
1004	Doe
1005	Chen

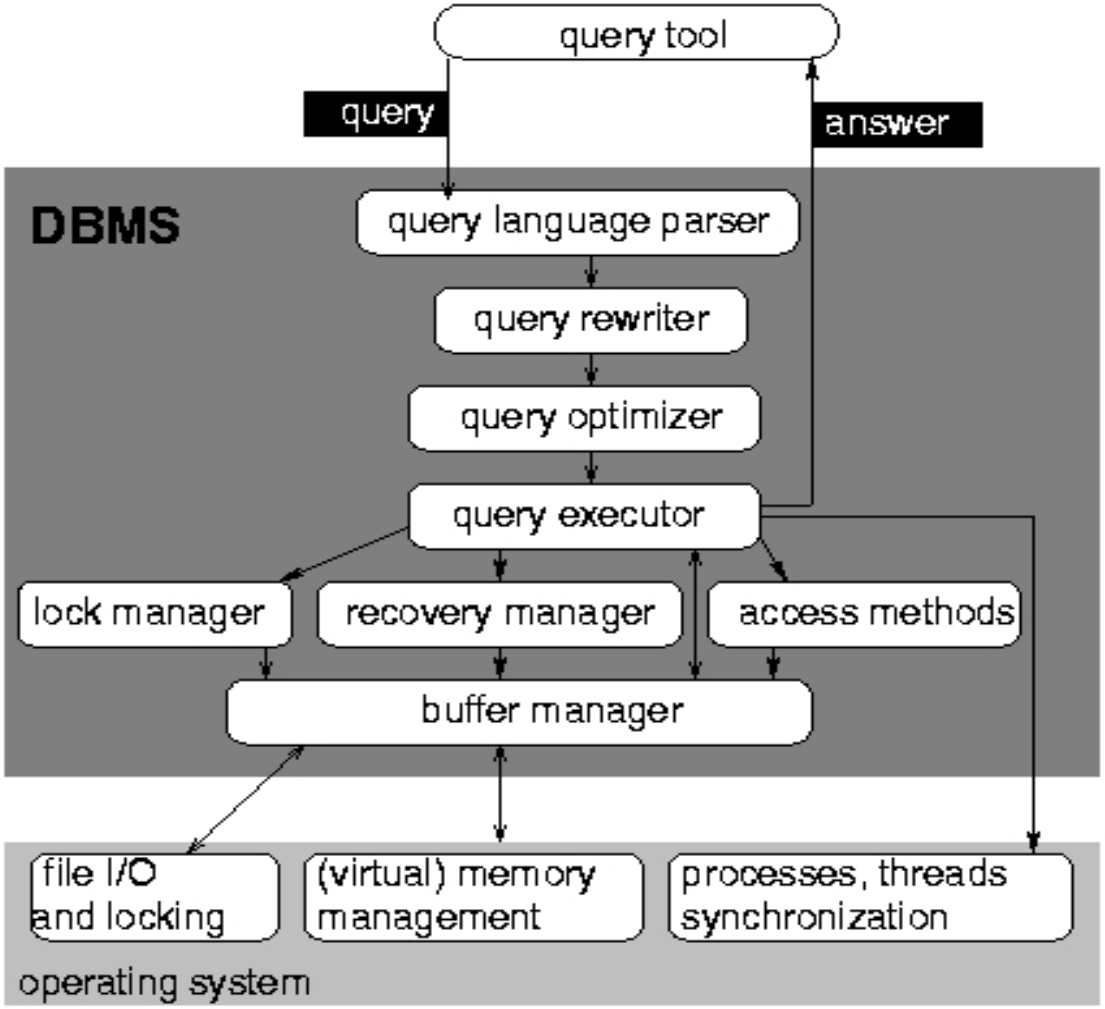
oid	city
1000	Redmond
1001	Redmond
1002	Washington
1003	New York
1004	Boston
1005	Seattle

oid	product
1000	Car
1001	House
1002	Travel
1003	House
1004	Car
1005	House

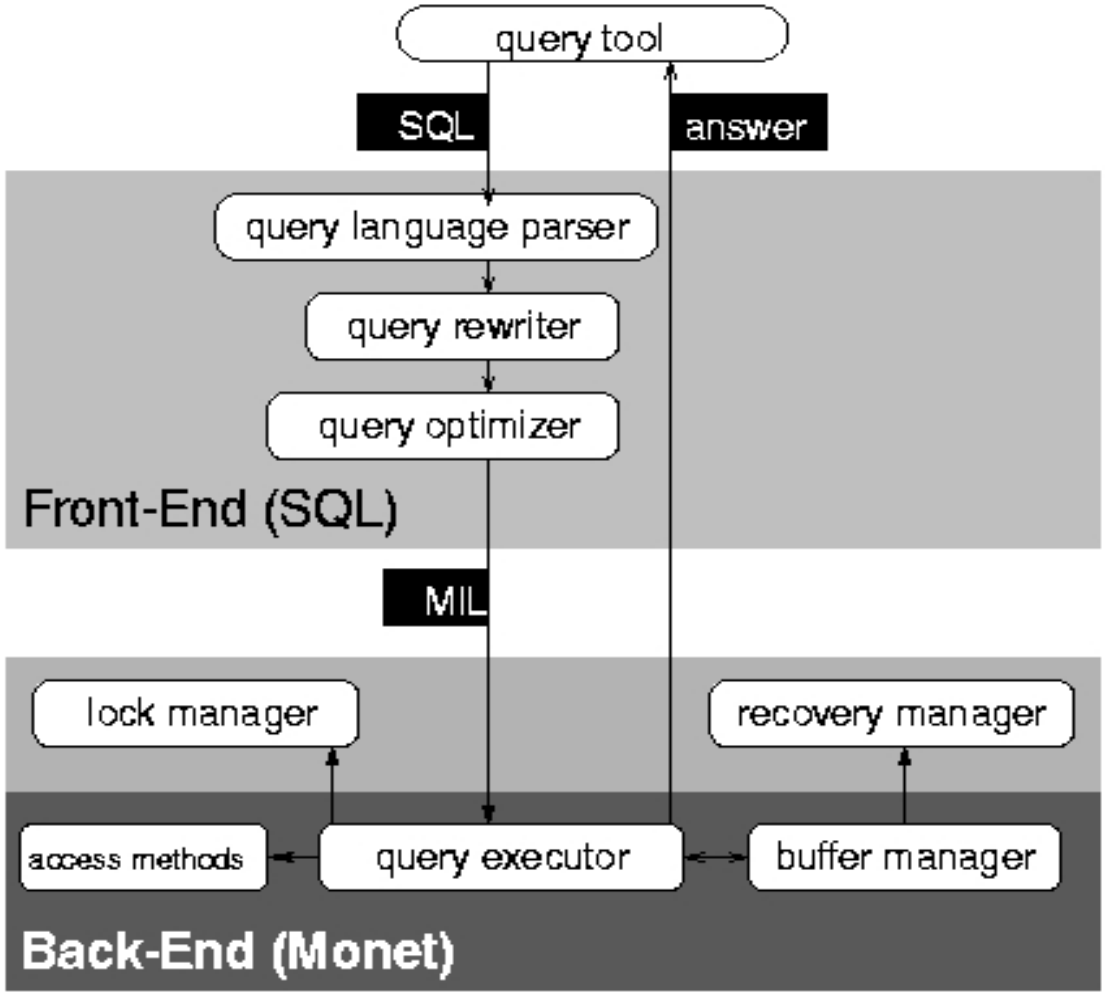
BAT storage as thin arrays



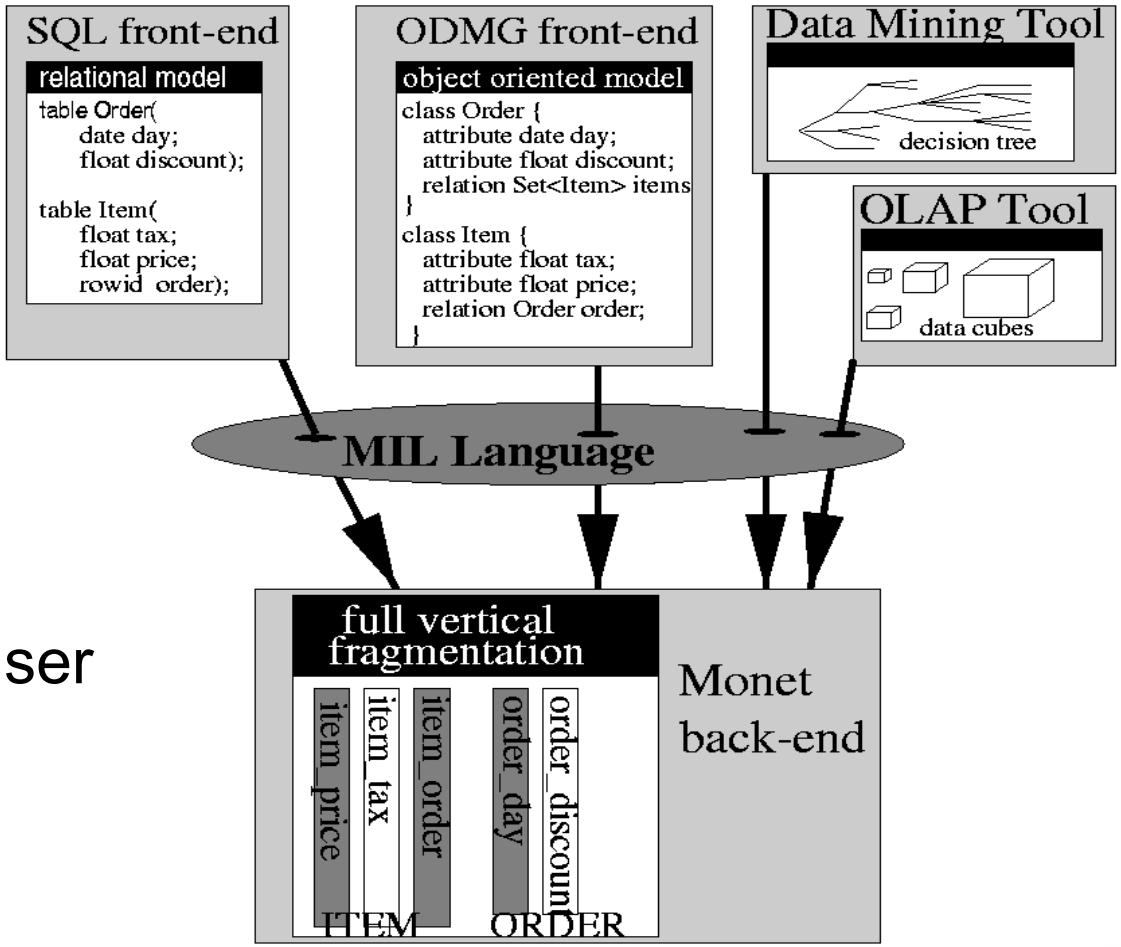
DBMS Architecture



Monet: DBMS Microkernel



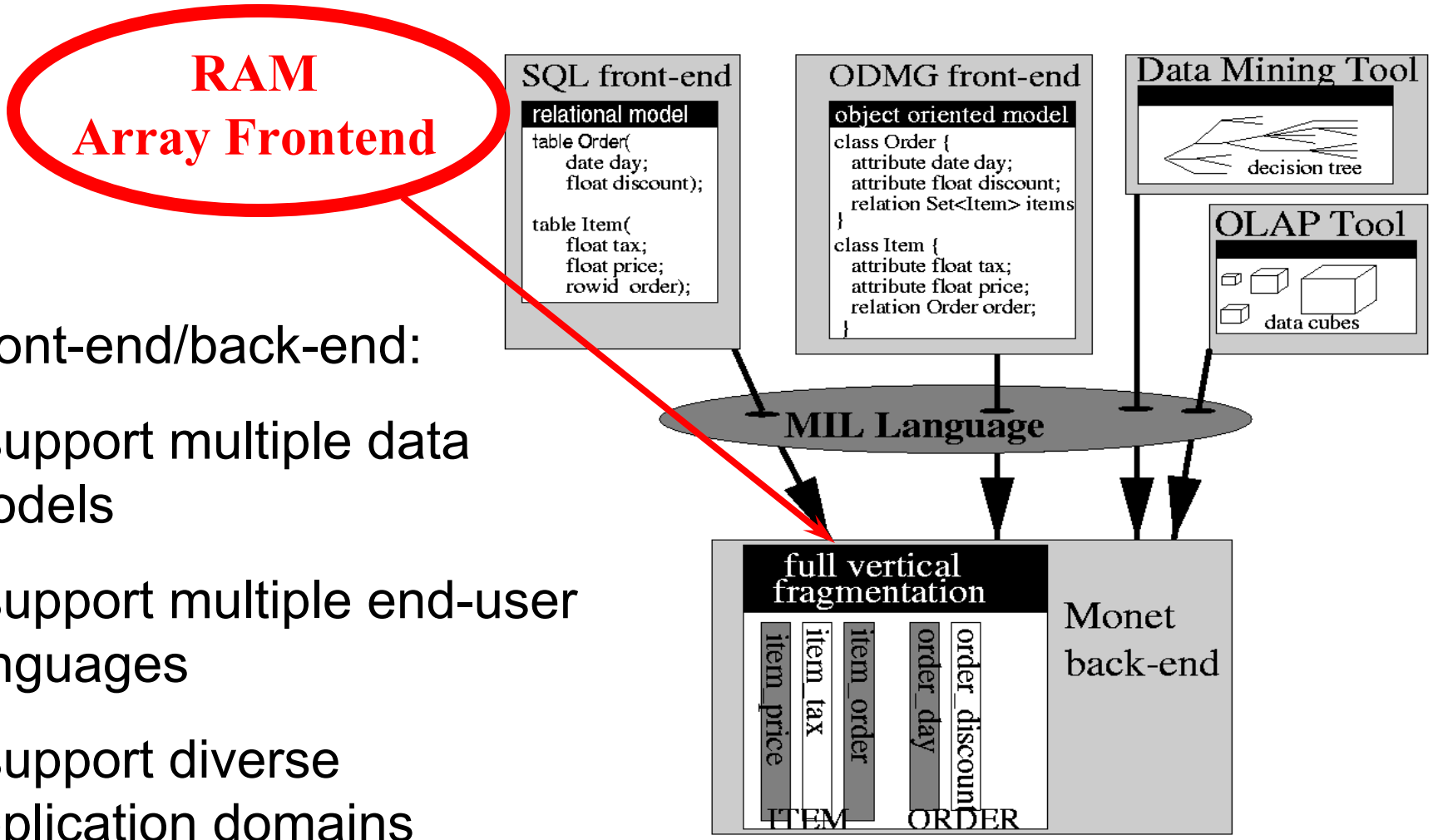
MonetDB: extensible architecture



Front-end/back-end:

- support multiple data models
- support multiple end-user languages
- support diverse application domains

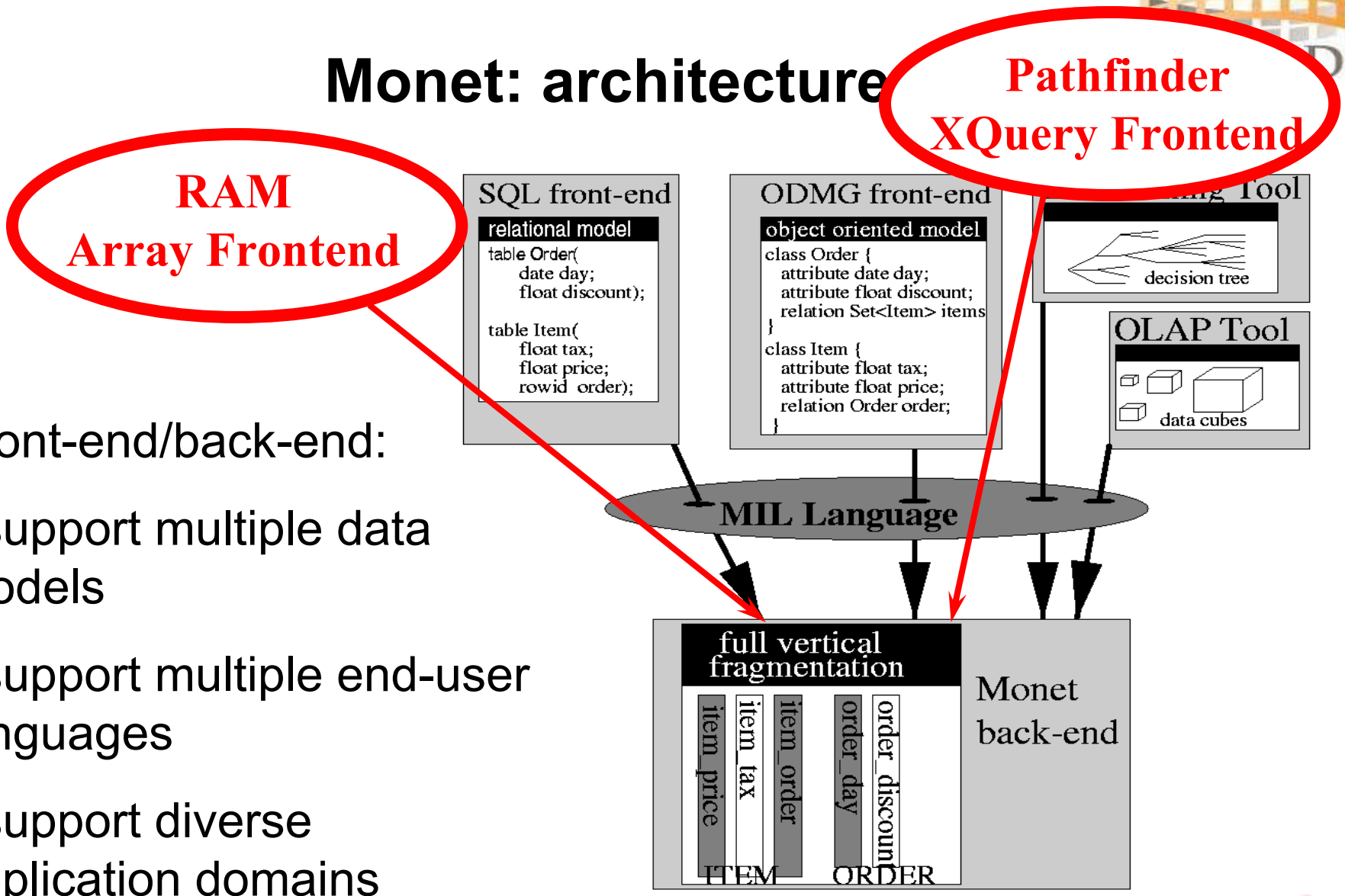
MonetDB: extensible architecture



Front-end/back-end:

- support multiple data models
- support multiple end-user languages
- support diverse application domains

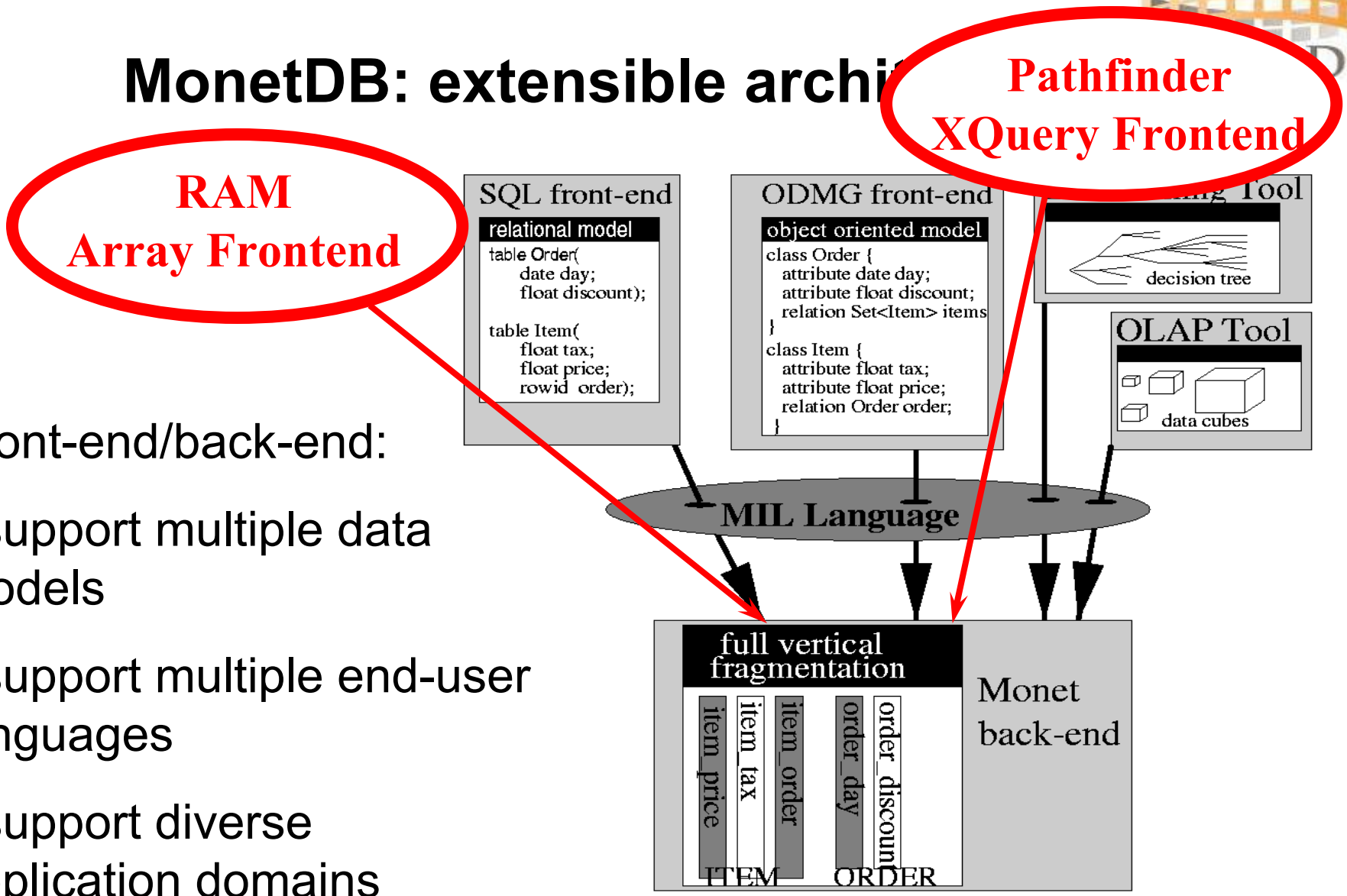
Monet: architecture



Front-end/back-end:

- support multiple data models
- support multiple end-user languages
- support diverse application domains

MonetDB: extensible archi



Front-end/back-end:

- support multiple data models
- support multiple end-user languages
- support diverse application domains

Contents

- What is MonetDB/X100
- Open-Source MonetDB Status
- MonetDB Internals
  - Architecture, Data Model, Query Language
  - Cache-Conscious Query Processing Algorithms
- Research Direction
  - X100: new technology inside MonetDB
- Reference Material

MonetDB: query language

MIL= Monet Interpreter Language

- algebraic language
- closed algebra on BATs
- bulk operations: no-degree-of-freedom

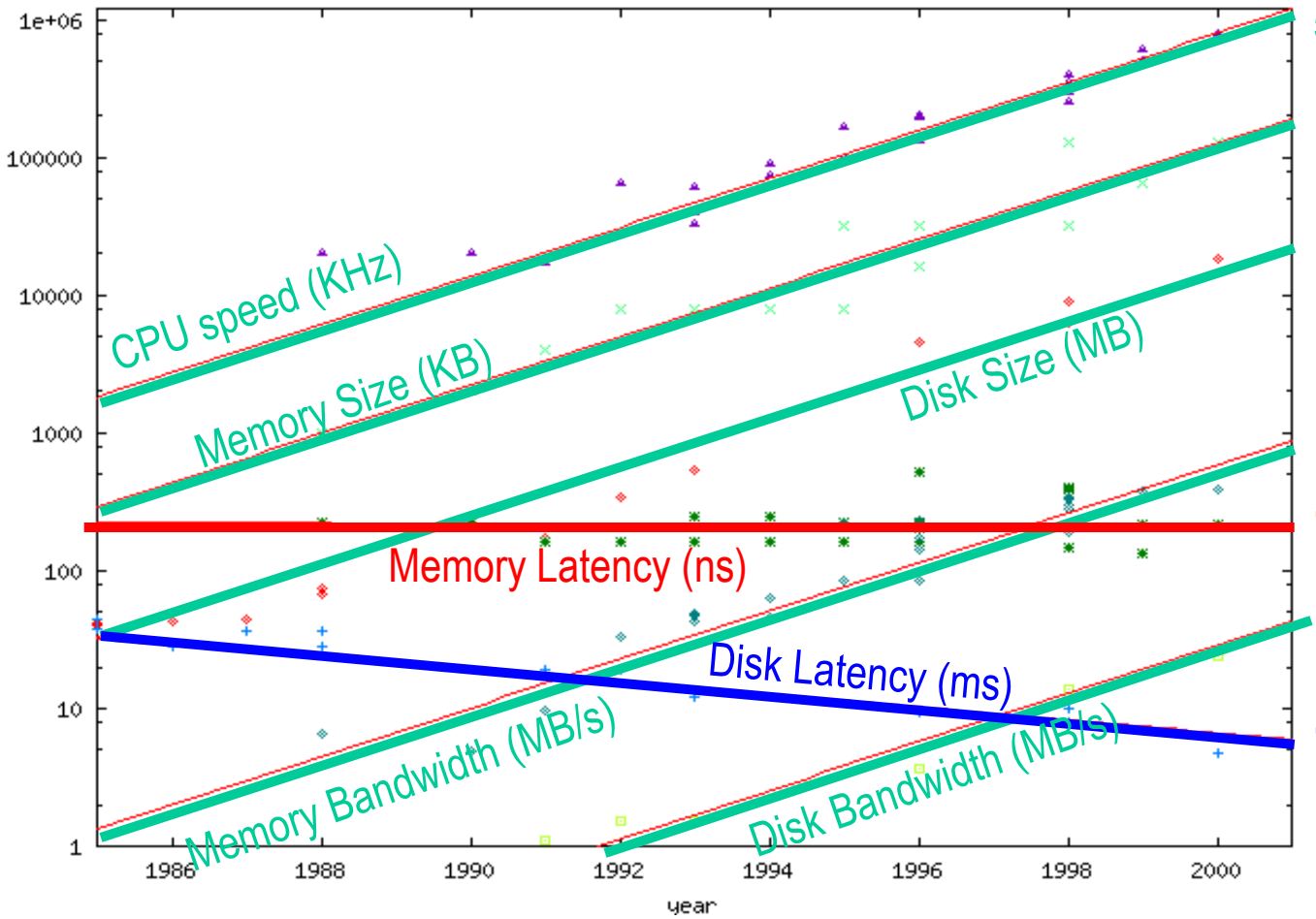
  ➔ hard-compiled primitives extract high CPI from modern CPUs

  ➔ Complex queries without interpretation cost

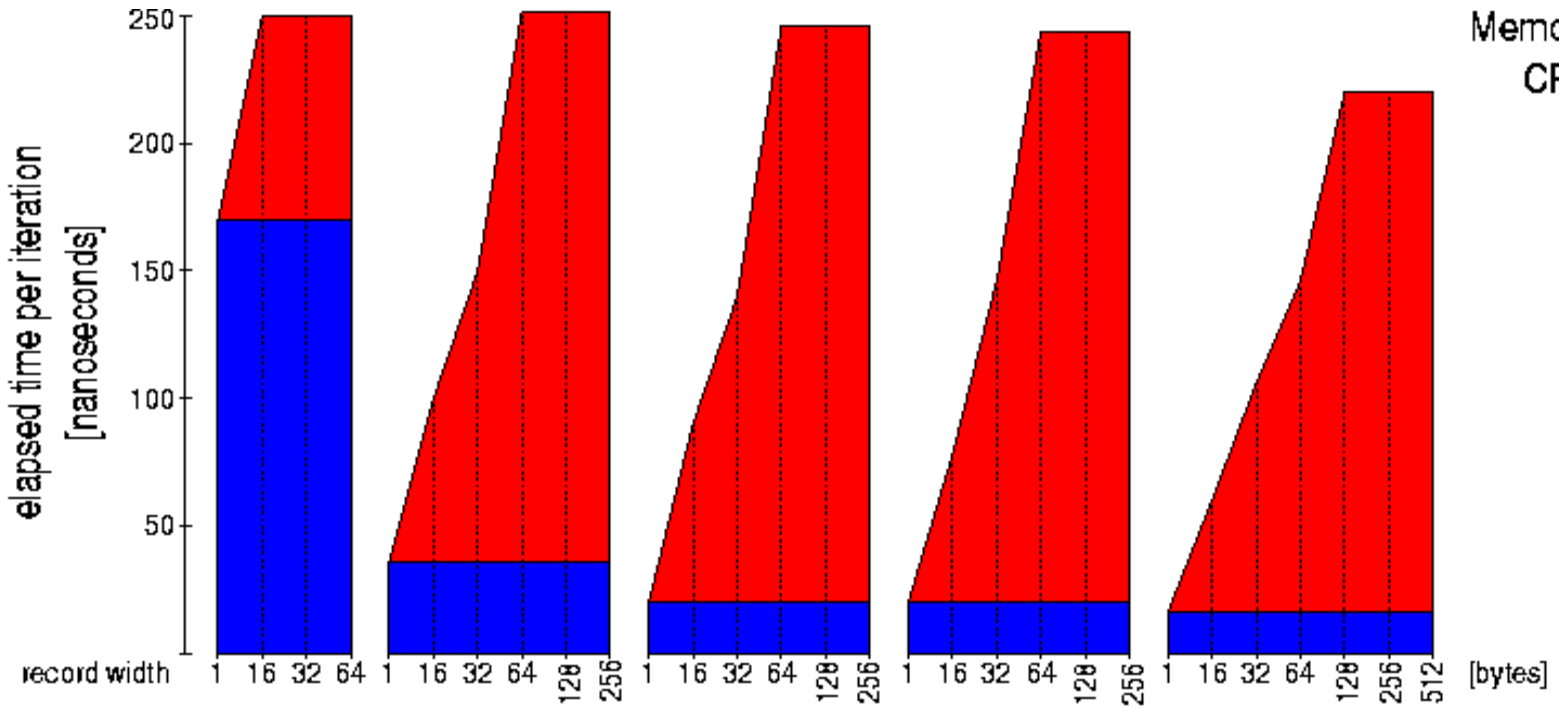
Contents

- What is MonetDB/X100
- Open-Source MonetDB Status
- MonetDB Internals
  - Architecture, Data Model, Query Language
  - Cache-Conscious Query Processing Algorithms
- Research Direction
  - X100: new technology inside MonetDB
- Reference Material

Hardware Improvement

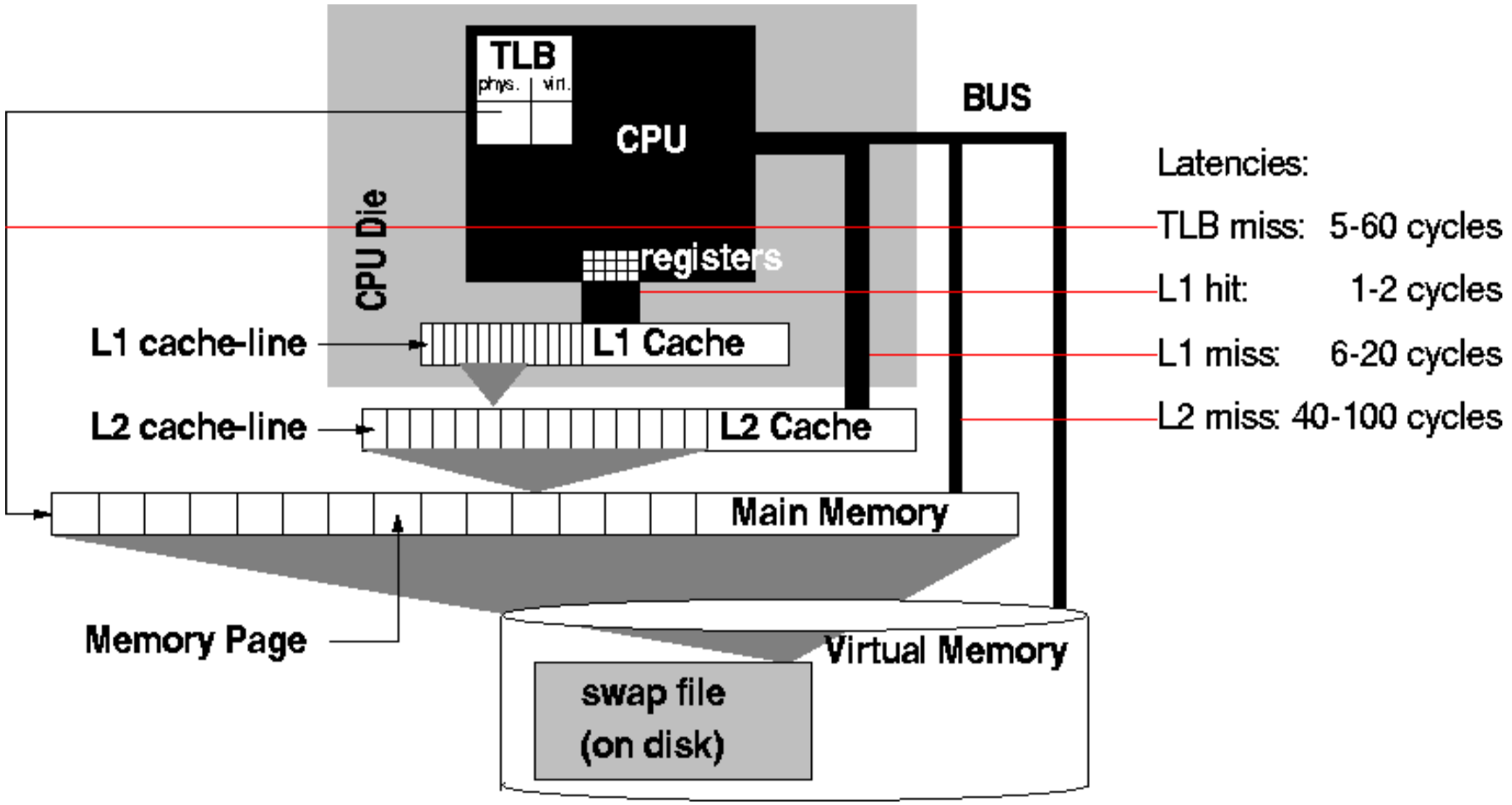


Memory Latency in Simple Query



year	1992	1996	1997	1998	1998
system	Sun LX	Sun Ultra	SunUltra	DEC Alpha	Origin2000
CPU type	Sparc	UltraSparc	UltraSparcII	Alpha	R10000
CPU speed	50 MHz	200 MHz	296 MHz	500 MHz	250 MHz
cache line size (L1/L2)	16 / -	16 / 64	16 / 64	32 / 64	32 / 128

Memory Access in Hierarchical Systems



Consequences for DBMS

- Memory access is a bottleneck
- Prevent cache & TLB misses
- Cache lines must be used fully

- DBMS must optimize
  - Data structures
  - Algorithms (focus: hash join)

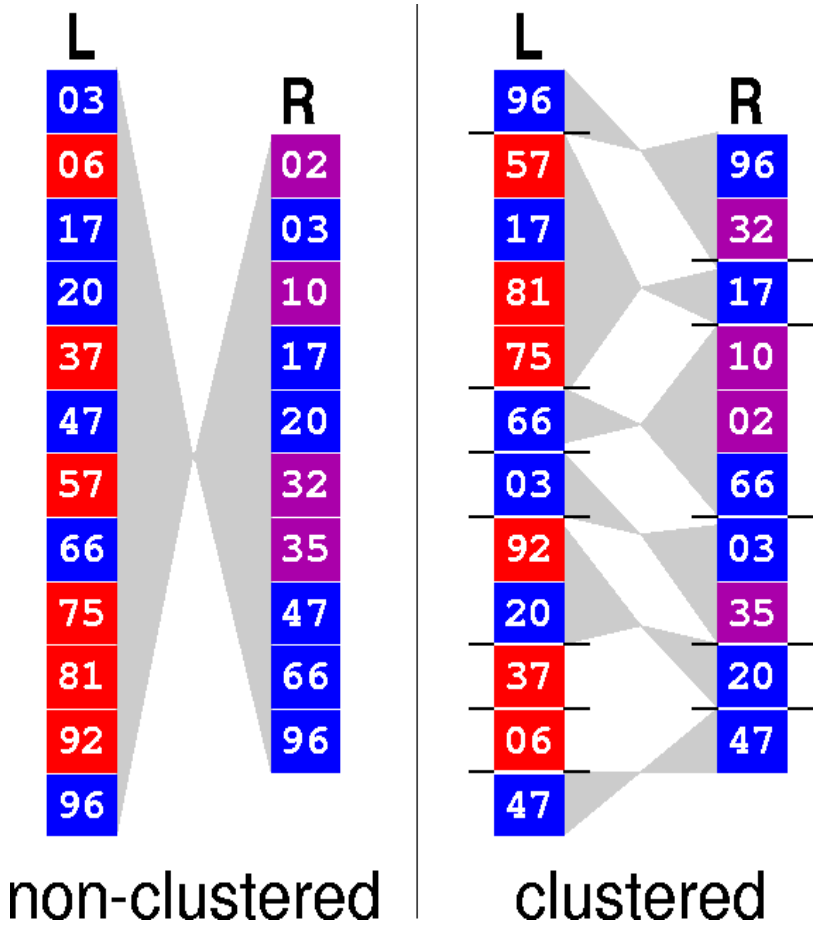
Partitioned Hash Join

- PHASE 1:

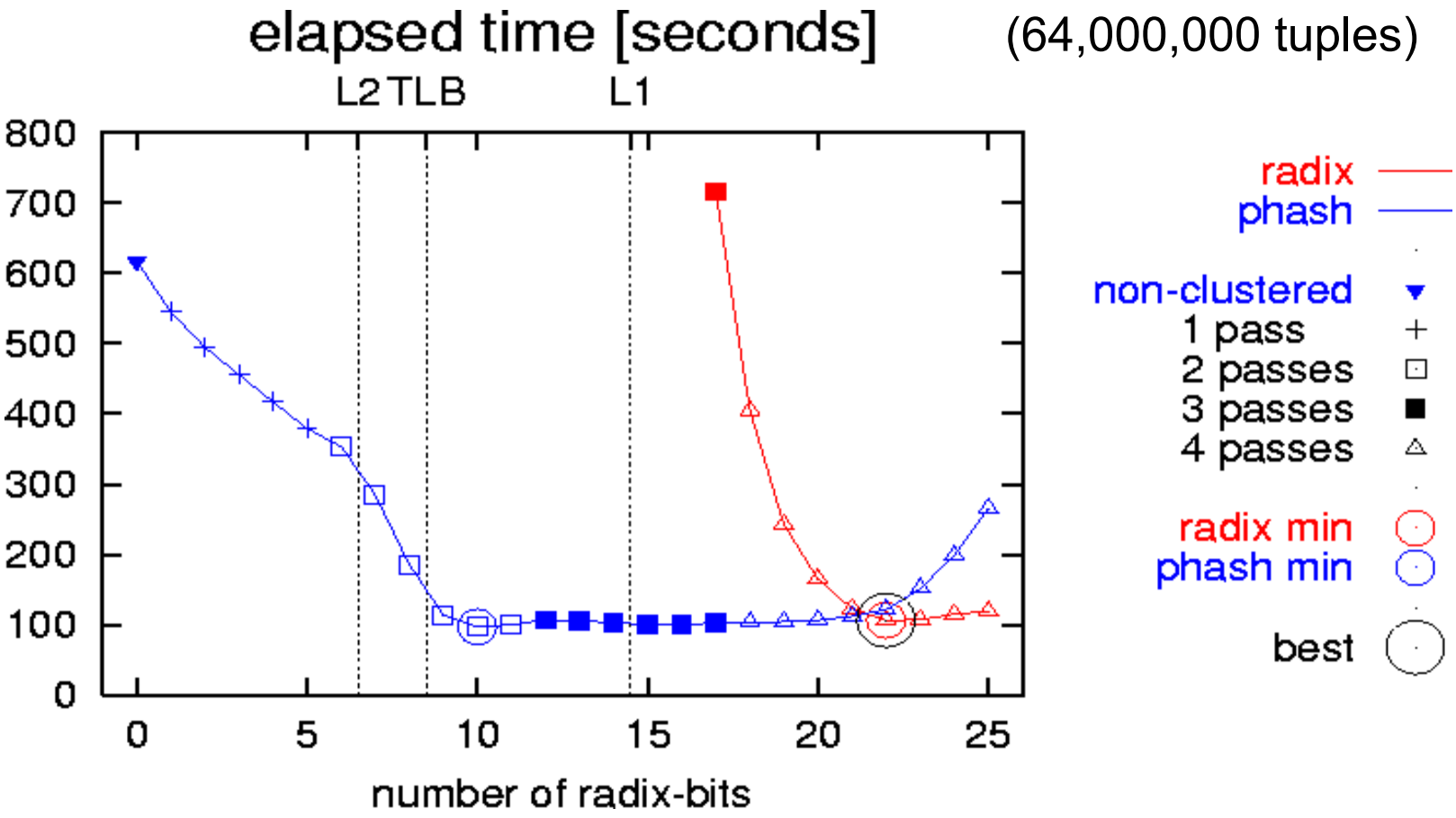
  cluster both input relations
  - Create clusters that fit in memory cache

- PHASE 2:

  Join matching clusters
  - Works fine because (outer+hash) fits the cache



Overall Join Performance

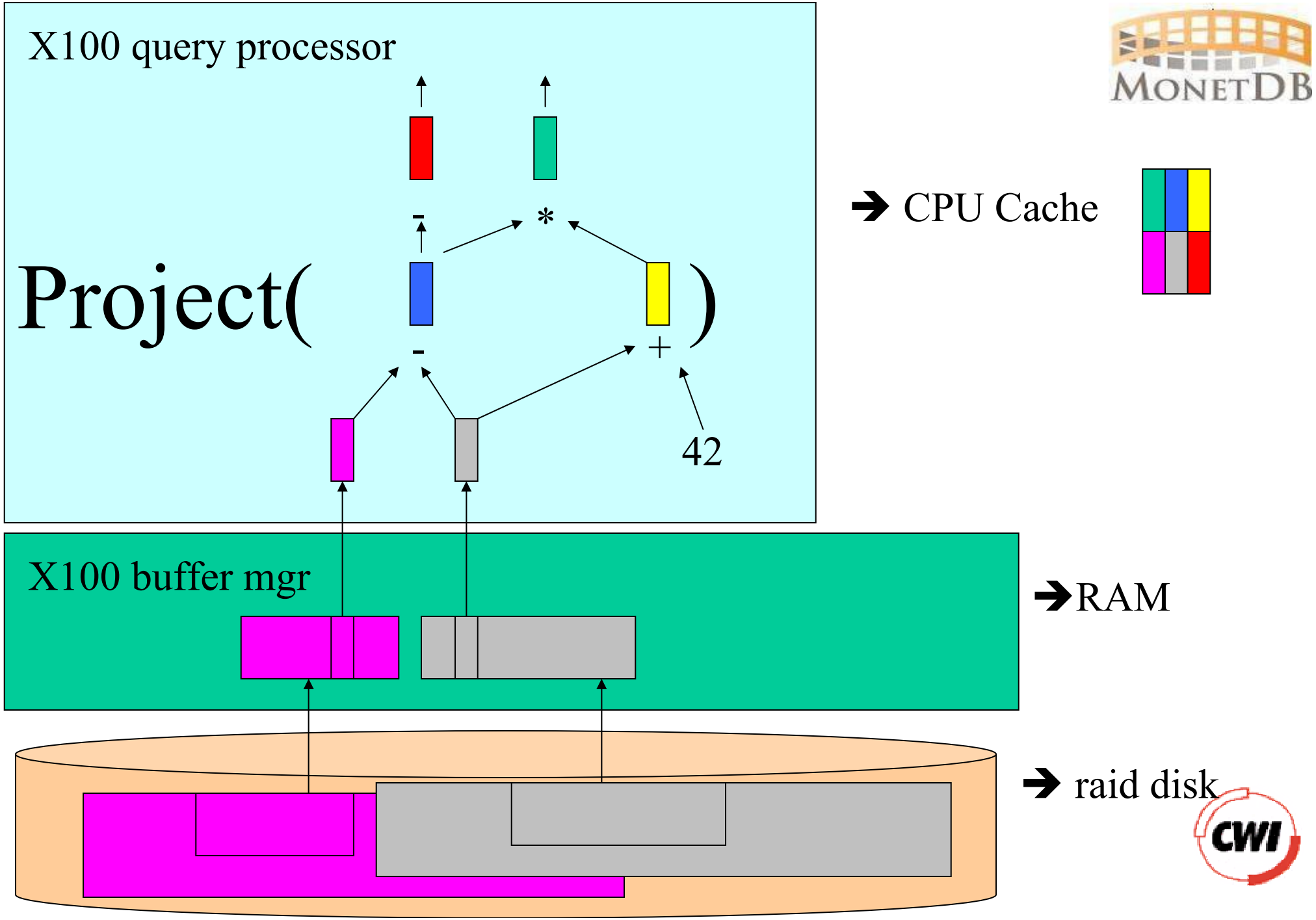


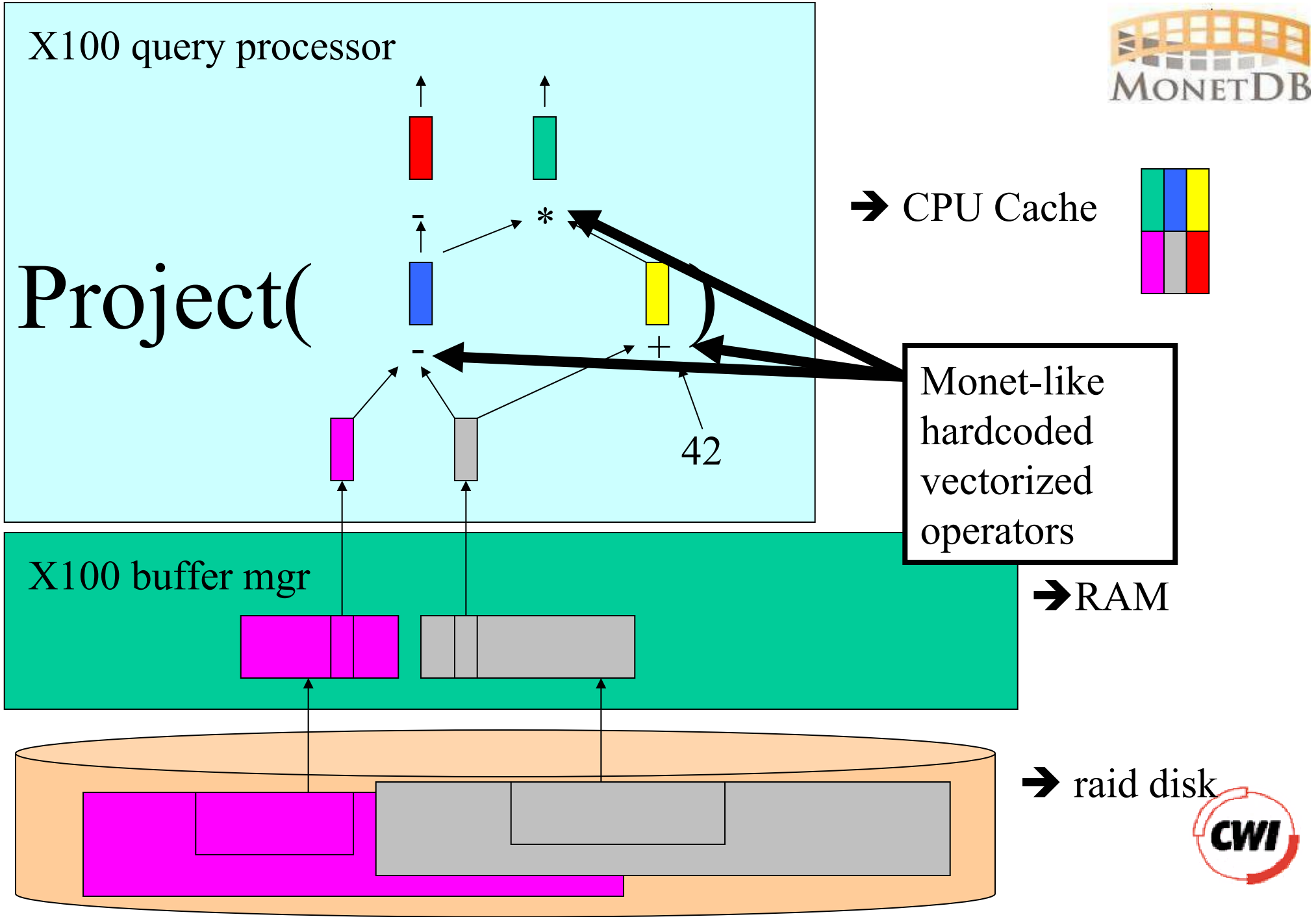
X100: main ideas

- Minimize Bandwidth (Materialization)
  - Pipelined materialization only in the CPU cache
  - Process on small vertical slices (‘vectors’)

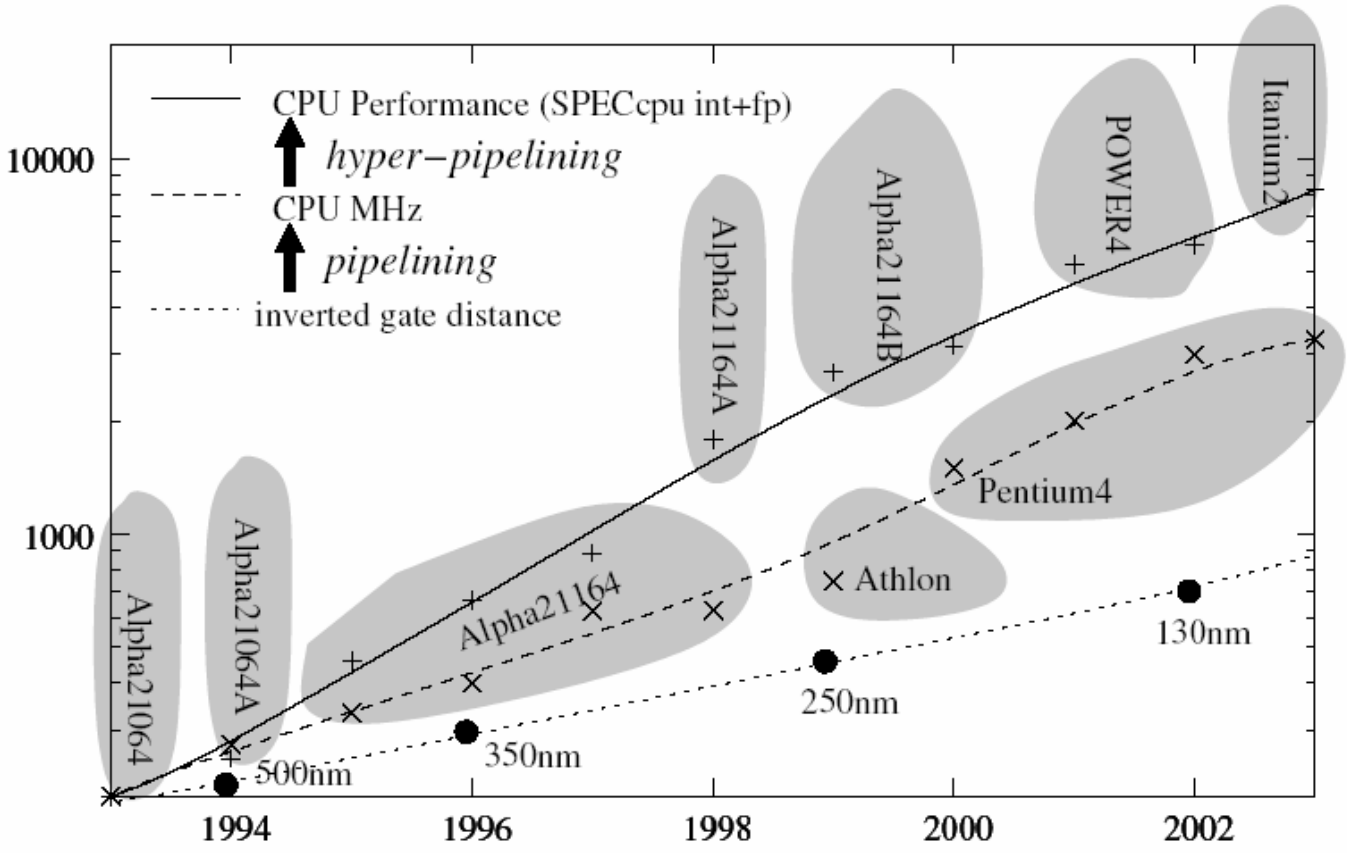
- CPU efficiency: *meet the smallprint conditions*
  - Hardcoded operators on unary arrays
  - Vectorize relational algorithms such as Hash-Aggregation
  - Use on-the-fly compilation to compile complex predicates

- I/O scalability: *try to stay CPU-bound*
  - Proper buffer mgrt with async prefetched bulk sequential I/O
  - Use lightweight compression


X100 query processor
Project( )
-
*
-
+
42
X100 buffer mgr
➔ CPU Cache
➔RAM
➔ raid disk
MONETDB
CWI


X100 query processor
Project(
-
*
-
+
42
)
Monet-like
hardcoded
vectorized
operators
X100 buffer mgr
➔ CPU Cache
➔RAM
➔ raid disk
MONETDB
CWI

CPU trends

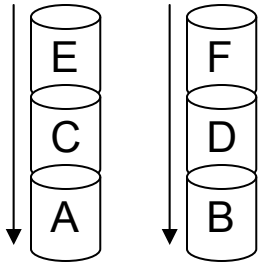

CPU Performance (SPECcpu int+fp)
hyper−pipelining
CPU MHz
pipelining
inverted gate distance
10000
1000
1994
1996
1998
2000
2002
Alpha21064
Alpha21064A
Alpha21164
Alpha21164A
Alpha21164B
POWER4
Itanium2
Athlon
Pentium4
500nm
350nm
250nm
130nm

CPU trends

- From CISC to hyper-pipelined:
  - 8086: CISC
  - 486: 2 execution units
  - Pentium: 2 x 5-stage pipelined units
  - Pentium3: 3 x 7-stage pipelined units
  - Pentium4: 12 x 20-stage pipelined execution units

operator sequence:
A,B,C,D,E,F

filled pipelines:



CPU trends

- filling the pipeline:
  - put in next instruction before previous finished
  - branch-mispredictions flush the pipeline! ☹
- Independent instructions go in parallel to different pipelines
  - Dependent instructions create holes ☹

- Problems:
  - conditional branches (if-then-else)
  - (dereferenced) function calls
  - Late binding in C++

➔ compilers fail!

“good”: scientific code (SPEC, matrix multiplication)
“bad”: generic DBMS code

MonetDB/X100 and CPU

- We need:
  - predictable instructions
  - independent instructions
- Relational algebra operators are *vectorized*:
  - Basic operators work on arrays, not single values
  - Tuples are independent ➔ tell the CPU so
  - No degree-of-freedom ➔ loop pipelining
- **Advantage wrt interpreted query engine: >10X**

TPC-H Benchmark Q1: very low cycles/tuple

input count	total MB	time (us)	BW MB/s	avg cycles	X100 primitive
6M	30	8518	352	1.9	map_fetch_uchr_col_flt_col
6M	30	8360	358	1.9	map_fetch_uchr_col_flt_col
6M	30	8145	368	1.9	map_fetch_uchr_col_flt_col
6M	35.5	13307	266	3.0	select_lt_usht_col_usht_val
5.9M	47	10039	468	2.3	map_sub_flt_val_flt_col
5.9M	71	9385	755	2.2	map_mul_flt_col_flt_col
5.9M	71	9248	767	2.1	map_mul_flt_col_flt_col
5.9M	47	10254	458	2.4	map_add_flt_val_flt_col
5.9M	35.5	13052	27	3.0	map_uidx_uchr_col
5.9M	53	14712	360	3.4	map_directgrp_uidx_col_uchr_col
5.9M	71	28058	253	6.5	aggr_sum_flt_col_uidx_col
5.9M	71	28598	248	6.6	aggr_sum_flt_col_uidx_col
5.9M	71	27243	260	6.3	aggr_sum_flt_col_uidx_col
5.9M	71	26603	266	6.1	aggr_sum_flt_col_uidx_col
5.9M	71	27404	259	6.3	aggr_sum_flt_col_uidx_col
5.9M	47	18738	250	4.3	aggr_count_uidx_col

Focus: scaling to disk

- example: video analysis
  - TRECVID: 600GB**?**
- example: data warehousing
  - TPC-H: 300GB

- Main-memory loads ➔Monet
- Disk-bound loads ➔ X100?

Is there any problem?

X100/Monet/Matlab is too fast:

- Well tuned algorithms spend only a couple of cycles per (double) value ➔500MB/s – 1500 MB/s

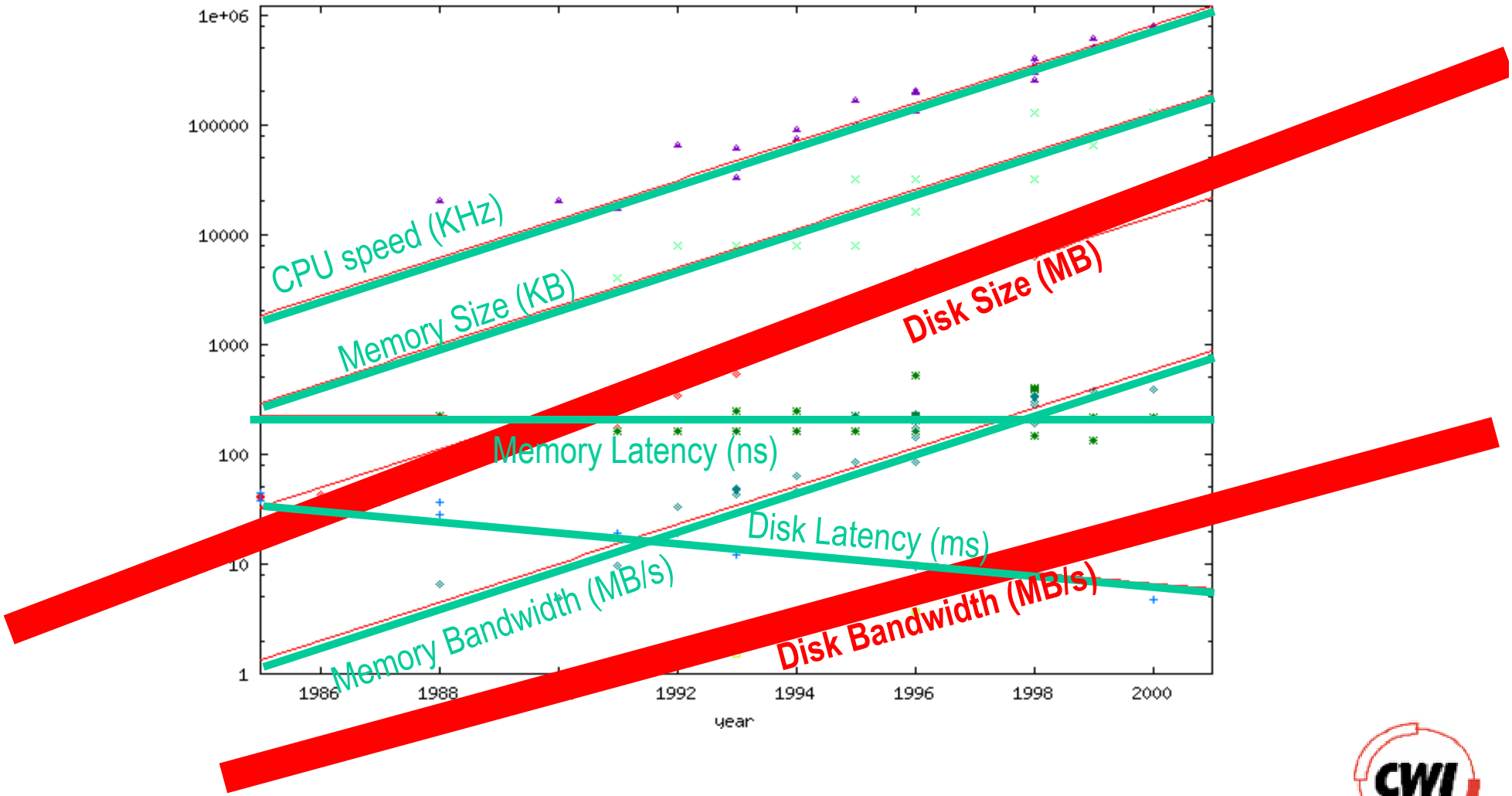
How fast are disks anyway?

- IDE: 10-40 MB/s
- SATA/SCSI: 30-60MB/s
- RAID-0: 100-200MB/s
- RAID-5: 80-130MB/s
- Combined RAIDS: 300-600MB/s

So, yes there is a performance gap

- Closing it requires very expensive RAID Solutions ☹
- Gap is expected to grow..

Hardware Improvement



sequential disk access

Disk access

- Sequential: read from start to end
- Random: read anywhere

Random disk access is very expensive (HOPELESS)

- Legacy database systems concentrate on this problem

X100 concentrates on sequential disk access

- Video queries are only as fast as you can read from disk
- Boost throughput with **lightweight compression**

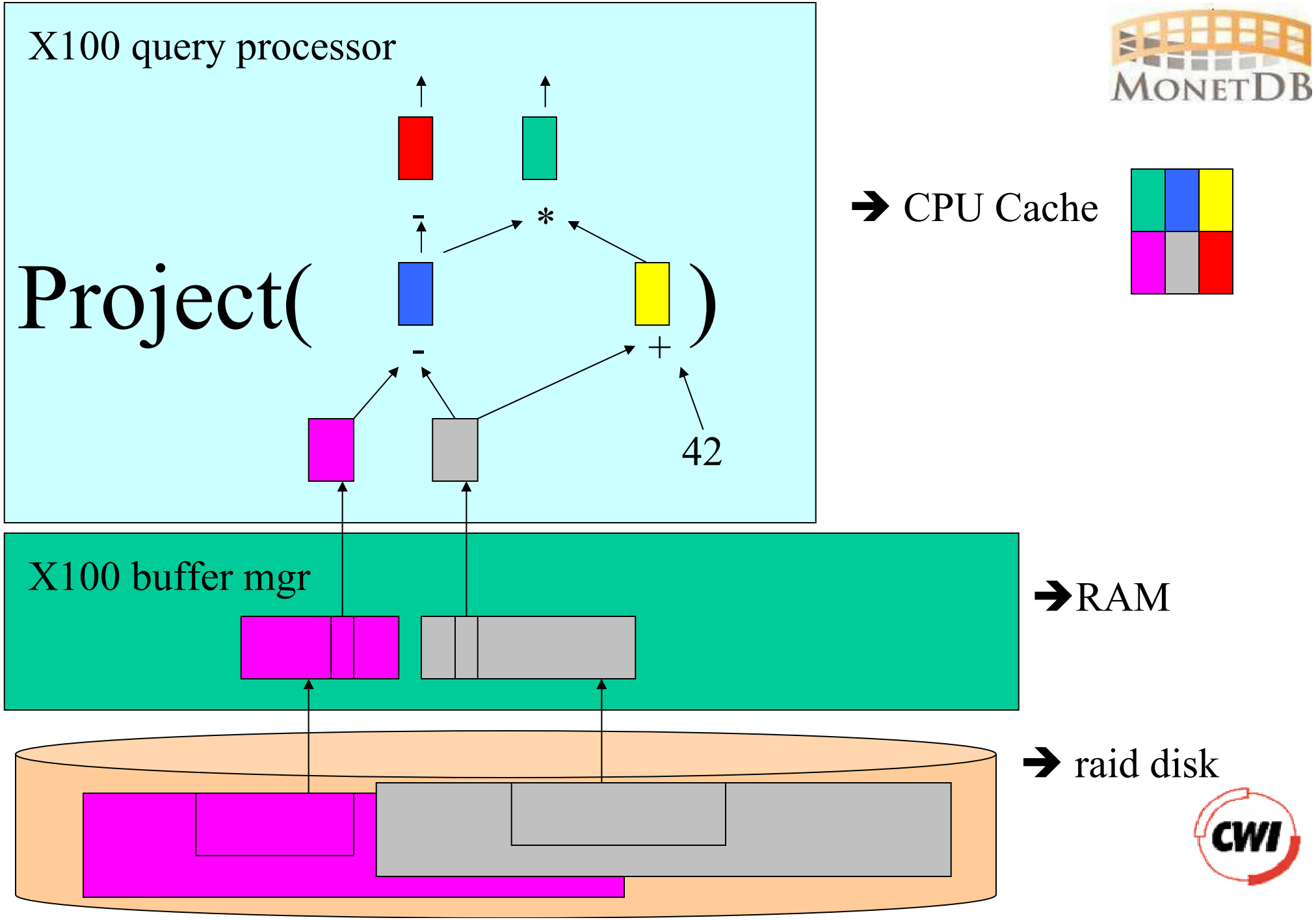
Compression in databases

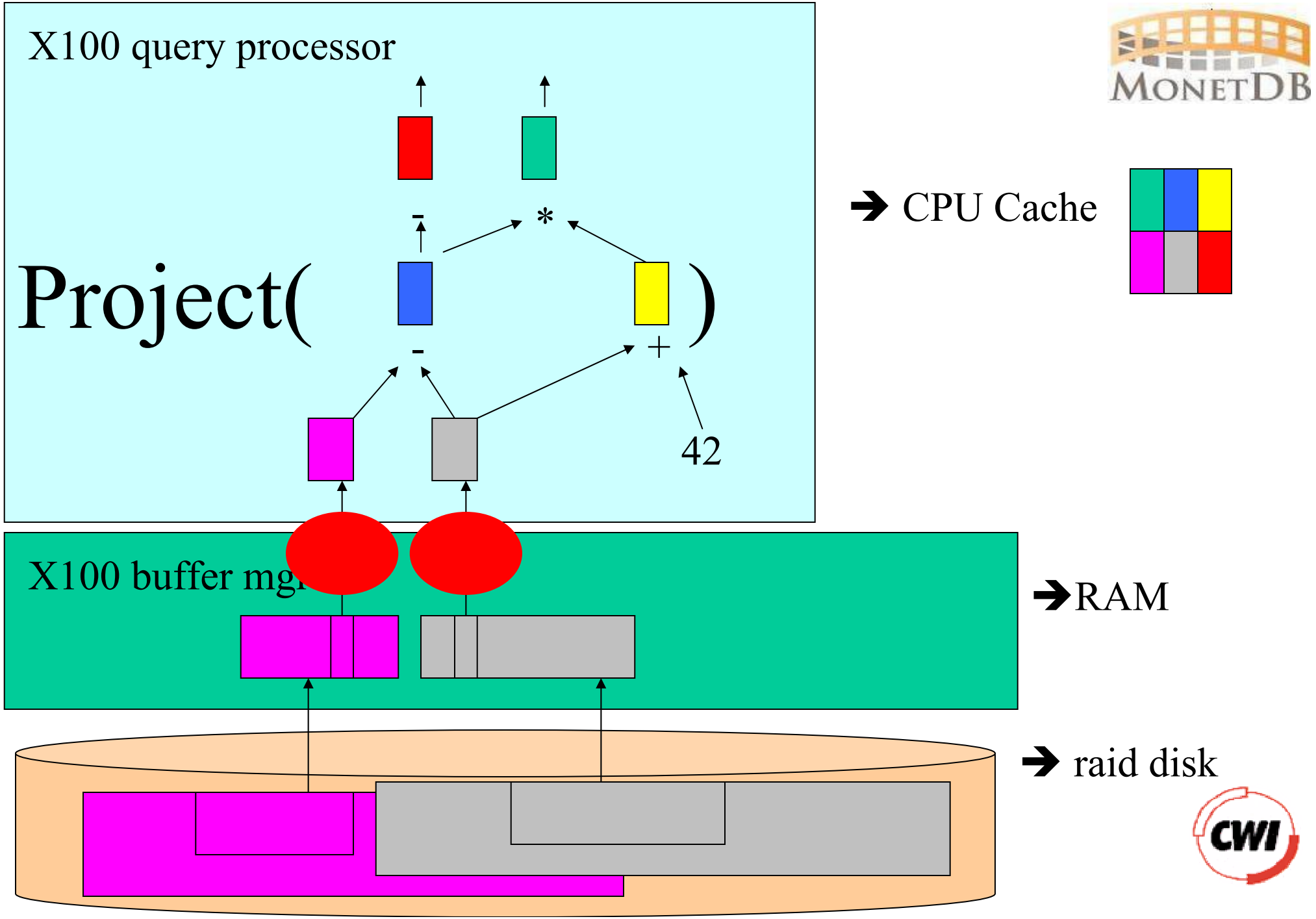
Some work done:

- [Goldstein,Larson] propose encoding tables
- Others concentrate on saving space (not performance)

Unique Selling Points

- Combine compression with vertical fragmentation
  - Uniform type strongly simplifies compression
- Skip RAM
  - (De)compress **between RAM and cache**
  - Not between disk and RAM

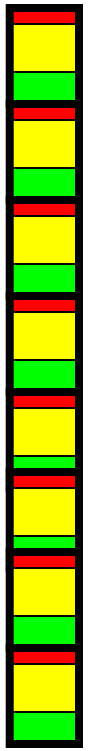

X100 query processor
Project(
)
-
*
-
+
42
X100 buffer mgr
➔ CPU Cache
➔RAM
➔ raid disk
MONETDB
CWI

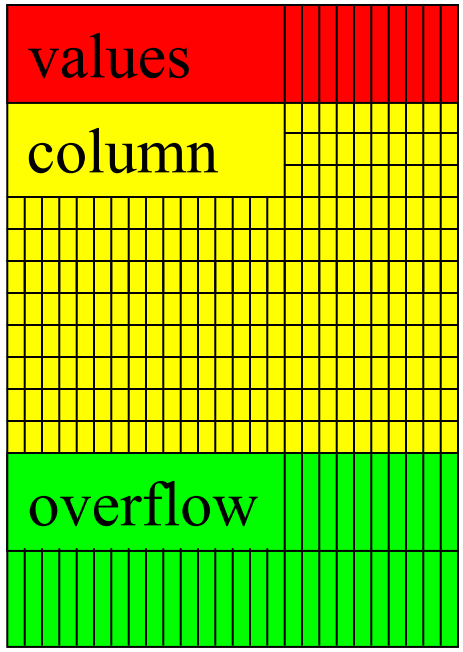

X100 query processor
Project(
)
-
*
-
+
42
X100 buffer mgr
➔ CPU Cache
➔RAM
➔ raid disk
MONETDB
CWI

Ultra-Lightweight Compression

Goal: improve throughput by compression ratio

- relatively low compression ratio
  - Typical cases: factor 3
- but: very fast!
  - Decompress one value each CPU cycle
  - Low extra CPU cost






```
LOOP:
   if (column[cur] == 255)
        out[cur] = overflow[miss++]
   else
        out[cur] = values[column[cur]]
```

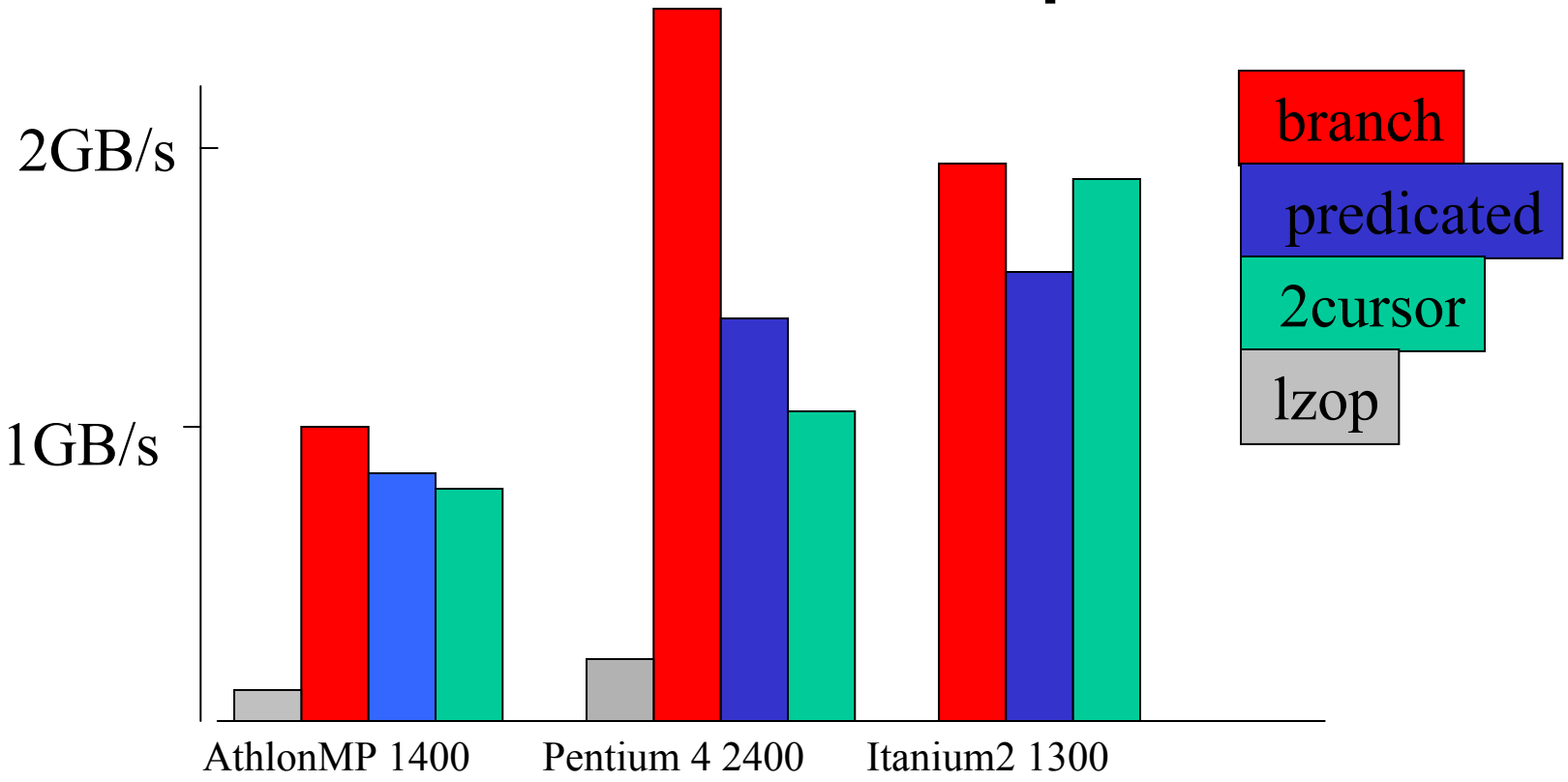
Ultra-Lightweight Compression

Goal: improve throughput by compression ratio

- relatively low compression ratio
  - Typical cases: factor 3
- but: very fast!
  - Decompress one value each CPU cycle
  - Compress in only a few cycles
  - Low extra CPU cost

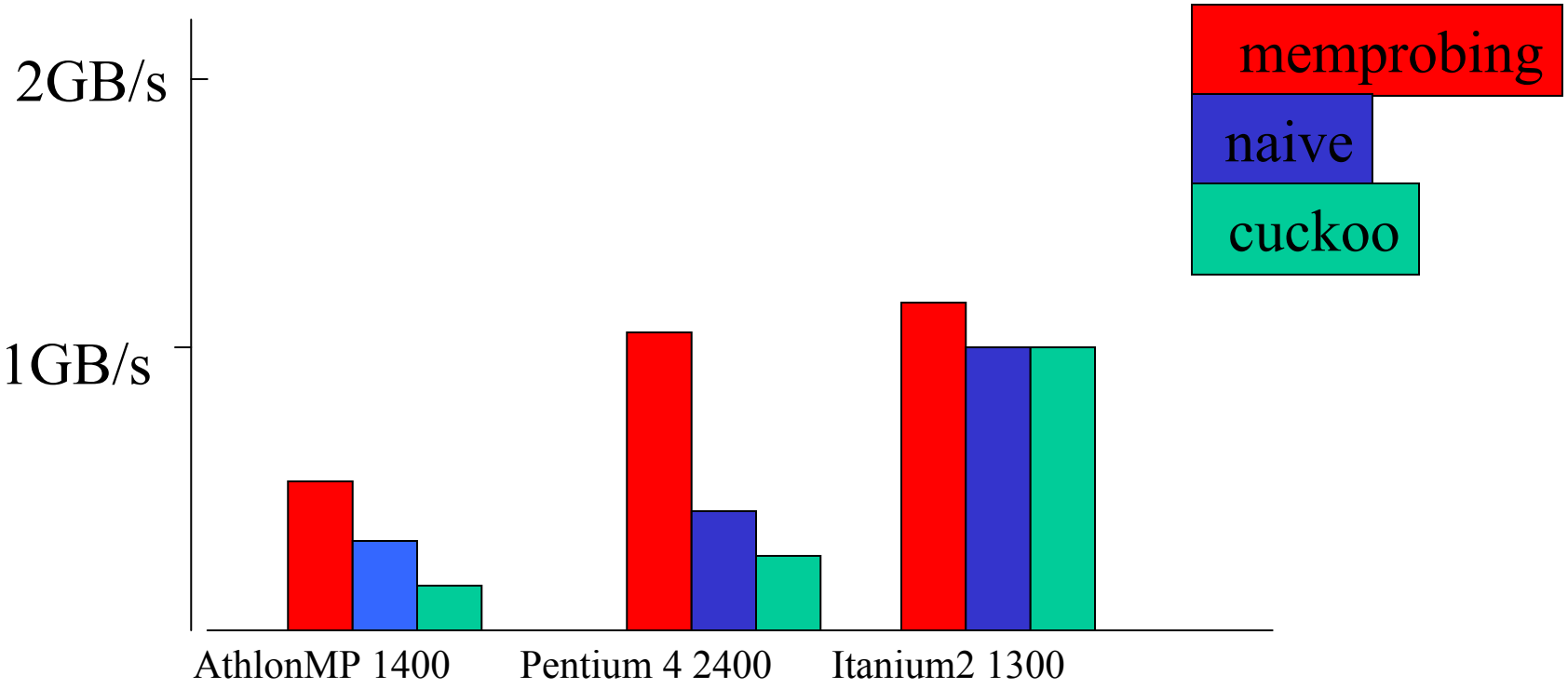
- TPC-H:
  - decompression factor 3.5
  - boosts db1 disk bandwidth from 120MB/s to 420MB/s
  - accelerates query by almost factor 3.5 (tentative)

GB/s Decompression



- TPC-H:
  - decompression factor 3.5
  - boosts db1 disk bandwidth from 120MB/s to 420MB/s
  - accelerates query by almost factor 3.5 (tentative)

GB/s Compression



Secret to succes is avoiding **branch mispredictions** in hash to get code for a to-be compressed value

Naïve = “ultra-fast” MonetDB Hash (63% first lookup hit)

Memorized Probing = new semi-perfect hash table (100% hit)

MonetDB/MIL		MonetDB/X100, 1CPU			DB2, 4CPU
Q	SF=1	SF=1	SF=1	SF=100	SF=100
1	3.72	0.50	0.31	30.25	229
2	0.46	0.01	0.01	0.81	19
3	2.52	0.04	0.02	3.77	16
4	1.56	0.05	0.02	1.15	14
5	2.72	0.08	0.04	11.02	72
6	2.24	0.09	0.02	1.44	12
7	3.26	0.22	0.22	29.47	81
8	2.23	0.06	0.03	2.78	65
9	6.78	0.44	0.44	71.24	274
10	4.40	0.22	0.19	30.73	47
11	0.43	0.03	0.02	1.66	20
12	3.73	0.09	0.04	3.68	19
13	11.42	1.26	1.04	148.22	343
14	1.03	0.02	0.02	2.64	14
15	1.39	0.09	0.04	14.36	30
16	2.25	0.21	0.14	15.77	64
17	2.30	0.02	0.02	1.75	77
18	5.20	0.15	0.11	10.37	600
19	12.46	0.05	0.05	4.47	81
20	2.75	0.08	0.05	2.45	35
21	8.85	0.29	0.17	17.61	428
22	3.07	0.07	0.04	2.30	93
	AthlonMP		Itanium2		

TPC-H Benchmark: overall results

MonetDB/X100 Conclusions

- Active development, encouraging first results
  - Innovations in kernel architecture, algorithms, compression
- Performance Target close to machine optimum
  - Close to or better than handcoded C
  - Is what scientific computing community expects!!
- Will address non-memory resident problems
  - RAID-tuned buffer mgr tuned for sequential access
  - Accelerated by ultra-lightweight disk compression
  - SMP/SMT built in
  - Not discussed: shared-nothing plans (they exist!)

MonetDB Conclusions

- Stable platform, high performance
  - supports standard APIs/languages
  - Supports non-standard APIs that may better suit your application (e.g. MIL, RAM) + extensible
- Open platform
  - Mozilla License
  - Hosted at sourceforge
- CWI db-arch team proven support record
  - A decade of experience in many application fields
  - Access to kernel hackers with unique HP expertise

Reference Material

- http://monetdb.cwi.nl
- EDBT'96: GIS extensions, SEQUOIA
- ICDE'98: MOA object-oriented query mapping, TPCD
- VLDB journal'99: MIL language definition
- VLDB'98: data mining performance (DD Benchmark)
- VLDB'99, VLDB journal'00: cache-optimized join
- VLDB'00, TKDE'02: super-scalar CPU join optimizations
- VLDB'02, cost modeling framework
- VLDB'04, cache-optimized joins with projection

SQL => MIL example

```
SELECT SUM(price), category, brand
FROM orders
WHERE date between 1-1-2000 and 1-2-2000
GROUPBY category, brand;
```

date	price	category	brand
31-12-1999	150.25	squash	nike
01-01-2000	150.00	squash	puma
15-01-2000	200.00	tennis	nike
29-01-2000	100.00	tennis	nike
01-02-2000	136.50	squash	nike

SUM(price*tax)	category	brand
150.00	squash	puma
300.00	tennis	nike

MIL Example

order_date[void,date]

```
[ 101,31-12-1999 ]
[ 101,01-01-2000 ]
[ 102,15-01-2000 ]
[ 103,29-01-2000 ]
[ 104,01-02-2000 ]
```

order_price[void,flt]

```
[ 101,150.25 ]
[ 102,150.00 ]
[ 103,200.00 ]
[ 104,100.00 ]
[ 105,136.50 ]
```

order_category[void,str]

```
[ 101,squash ]
[ 102,squash ]
[ 103,tennis ]
[ 104,tennis ]
[ 105,squash ]
```

order_brand[void,str]

```
[ 101,nike ]
[ 102,puma ]
[ 103,nike ]
[ 104,nike ]
[ 105,nike ]
```

MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).reverse.mark;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
v5 := group(v2,v3);
v6 := v5.reverse.mirror.unique;
v7 := {sum}(v6,v5,v4);
[printf]("%s\t%s\t%d\n", v7, v3, v4);
```

MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
```

```
order_date                                                          v1
[ 100,31-12-1999 ] =====> [ 101,01-01-2002 ]  ===> [101,0 ] ======> [ 0,101 ]
[ 101,01-01-2000 ] select [ 102,15-01-2002 ]  mark [102,1 ] reverse [ 1,102 ]
[ 102,15-01-2000 ]        [ 103,29-01-2002 ]       [103,2 ]         [ 2,103 ]
[ 103,29-01-2000 ]
[ 104,01-02-2000 ]
```

MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
```

```
v1          order_brand       v2
[ 0,101 ] [ 100,nike ] ===> [ 0,puma ]
[ 1,102 ] [ 101,puma ] join [ 1,nike ]
[ 2,103 ] [ 102,nike ]      [ 2,nike ]
          [ 103,nike ]
          [ 104,nike ]
```

```
join( [X,Y], [Y,Z]) => [X,Z]
        |____|
```

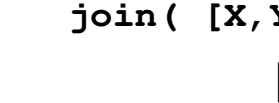
MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
```

```
v1          order_brand       v2
[ 0,101 ] [ 100,nike ] ===> [ 0,puma ]
[ 1,102 ] [ 101,puma ] join [ 1,nike ]
[ 2,103 ] [ 102,nike ]      [ 2,nike ]
          [ 103,nike ]
          [ 104,nike ]
```

```
v1          order_category        v3
[ 0,101 ] [ 100,squash ] ===> [ 0,squash ]
[ 1,102 ] [ 101,squash ] join [ 1,tennis ]
[ 2,103 ] [ 102,tennis ]      [ 2,tennis ]
          [ 103,tennis ]
          [ 104,squash ]
```

join([X,Y], [Y,Z]) => [X,Z]



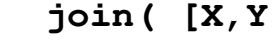
MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
```

```
v1          order_brand       v2              v1          order_price          v4
[ 0,101 ] [ 100,nike ] ===> [ 0,puma ]     [ 0,101 ] [ 100,150.25 ] ===> [ 0,150.00 ]
[ 1,102 ] [ 101,puma ] join [ 1,nike ]     [ 1,102 ] [ 101,150.00 ] join [ 1,200.00 ]
[ 2,103 ] [ 102,nike ]      [ 2,nike ]     [ 2,103 ] [ 102,200.00 ]      [ 2,100.00 ]
          [ 103,nike ]                               [ 103,100.00 ]
          [ 104,nike ]                               [ 104,136.50 ]

v1         order_category         v3
[ 0,101 ] [ 100,squash ] ===> [ 0,squash ]
[ 1,102 ] [ 101,squash ] join [ 1,tennis ]
[ 2,103 ] [ 102,tennis ]      [ 2,tennis ]
          [ 103,tennis ]
          [ 104,squash ]
```

join([X,Y], [Y,Z]) => [X,Z]



MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
v5 := group(v2,v3);
```

v2	**v3**		**v5**	
[0,squash]	[0,puma]	=====>	[0,0]	(squash,puma) ➔ 0
[1,tennis]	[1,nike]	**group**	[1,1]	(tennis,nike) ➔ 1
[2,tennis]	[2,nike]		[2,1]	

group([X,a],[X,b]) returns **[X,gid]** with gid a X for each unique [a,b]

MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
v5 := group(v2,v3);
v6 := v5.reverse.mirror.unique;
```

```
v5                                                    v6
[ 0,0 ] ======> [ 0,0 ] =====> [ 0,0 ] =====> [ 0,0 ]
[ 1,1 ] reverse [ 1,1 ] mirror [ 1,1 ] unique [ 1,1 ]
[ 2,1 ]         [ 1,2 ]        [ 1,1 ]
```

MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
v5 := group(v2,v3);
v6 := v5.reverse.mirror.unique.mark.reverse;
v7 := {sum}(v6,v5,v4);
```

v6	**v5**	**v4**		**v7**
[0,0]	[0,0]	[0,150.00]	=====>	[0,150.00]
[1,1]	[1,1]	[1,200.00]	**pump**	[1,300.00]
	[2,1]	[2,100.00]	**{sum()}**	

pump {f}(sets,grouping,values) assembles a BAT for each set, and calls f(BAT)

0 ➔ sum([150.00,150.00]) = 150.00

1 ➔ sum([200.00,200.00]) = 300.00
[100.00,100.00]

MIL Example

```
v1 := order_date.select(date("1-1-2000"),date("1-2-2000")).mark.reverse;
v2 := v1.join(order_brand);
v3 := v1.join(order_category);
v4 := v1.join(order_price);
v5 := group(v2.v3);
v6 := v5.reverse.mirror.unique;
v7 := {sum}(v6,v5,v4);
[sprintf]("%s\t%s\t%d\n", v7, v3, v4);
```

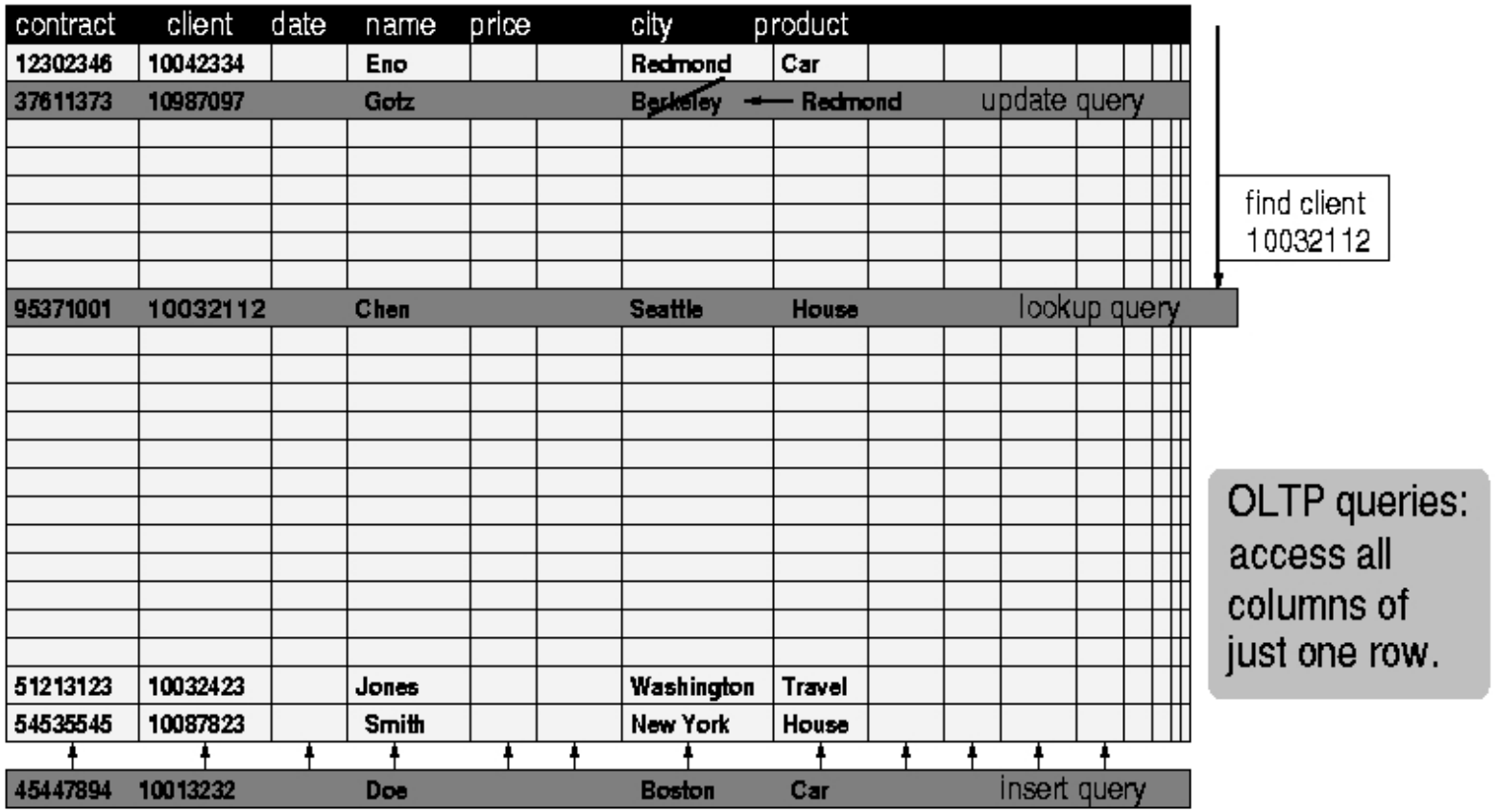
multiplex [f()] ([X,a],..,[X,b]) => [X,f(a,..,b)]
[Y,c],..,[Y,d] [Y, f(c,..,d)]

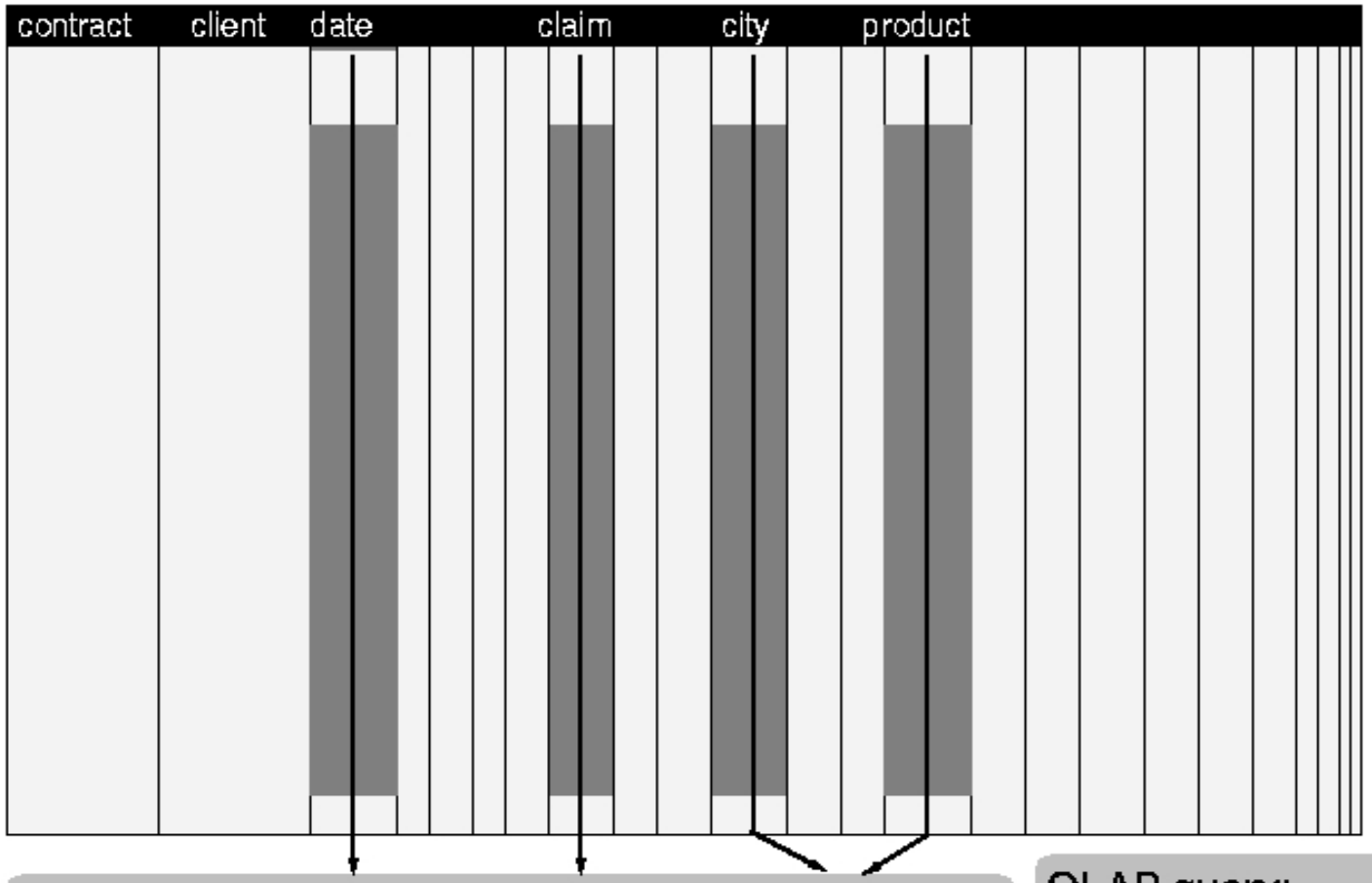
v7	**v8**	**v9**		
[0,150.00]	[0,squash]	[0,nike]	========>	[0,"150.00 squash puma"]
[1,300.00]	[1,tennis]	[1,puma]	**multiplex**	[1,"300.00 tennis nike"]
			[sprintf()]	

Contents

- What is MonetDB/X100
- Open-Source MonetDB Status
- MonetDB Internals
  - Architecture, Data Model, Query Language
  - Cache-Conscious Query Processing Algorithms
- Research Direction
  - MonetDB Evaluation
  - X100: new technology inside MonetDB
- Reference Material

Transactions (OLTP)



OLTP queries:
access all
columns of
just one row.

OLAP, Data Mining