MonetDB:

A high performance database kernel

for query-intensive applications

Peter Boncz

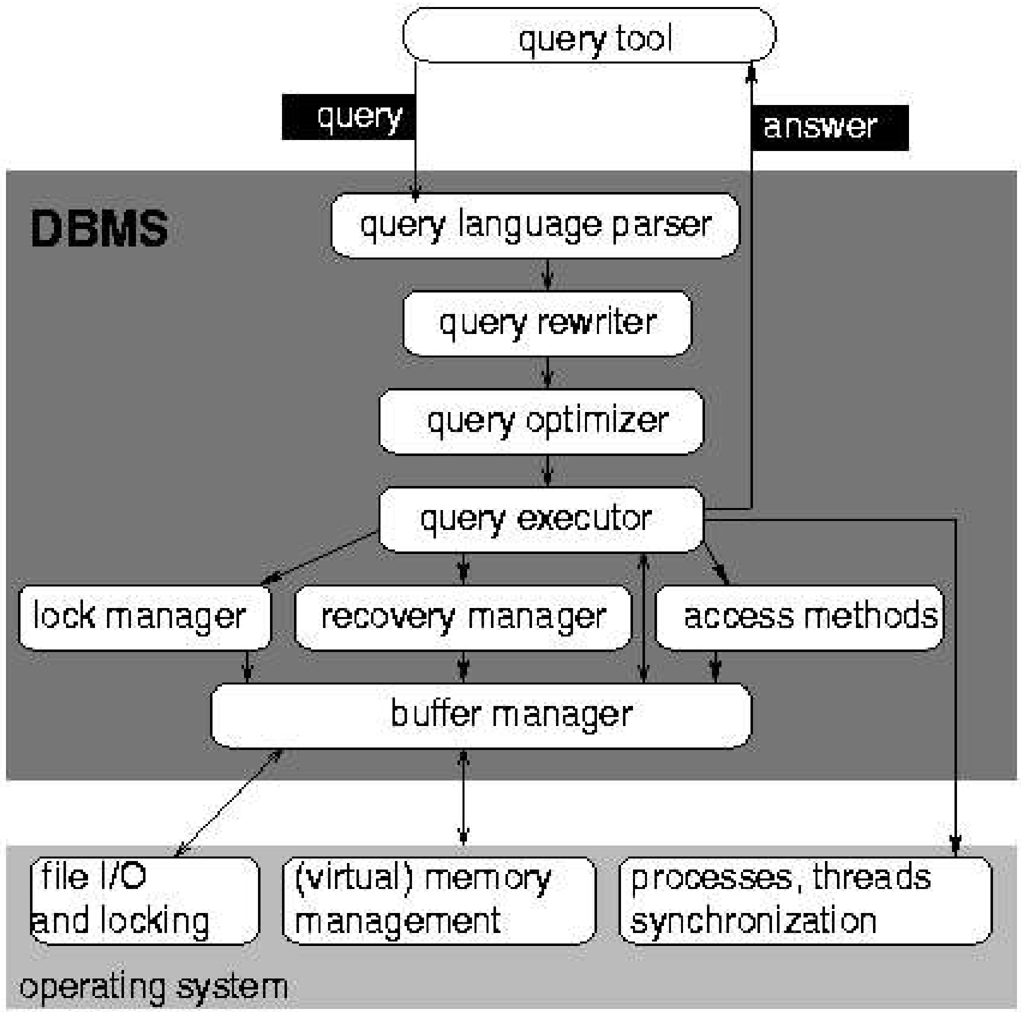
CWI
Amsterdam
The Netherlands

boncz@cwi.nl

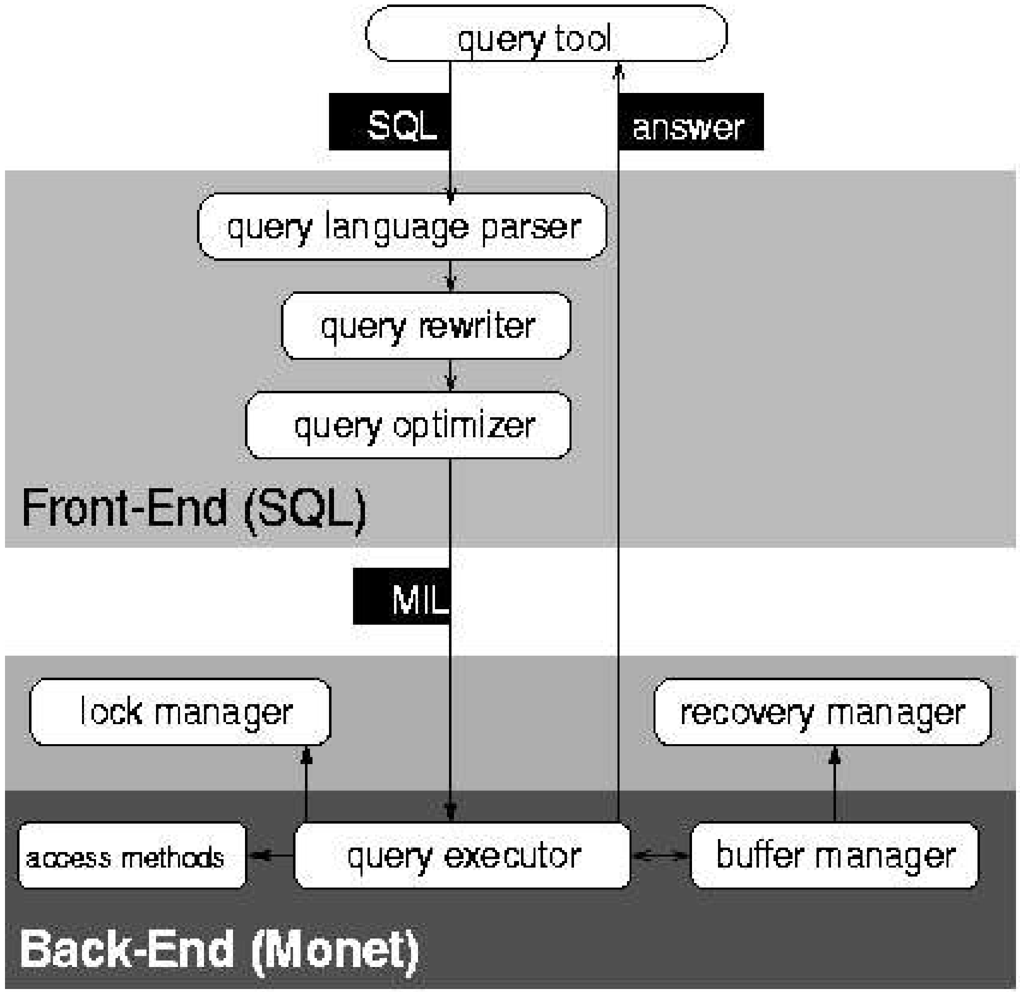
Contents

- The Architecture of MonetDB
- The MIL language with examples
- Where is MonetDB good for?
- Implementation Highlights
- Reference Material

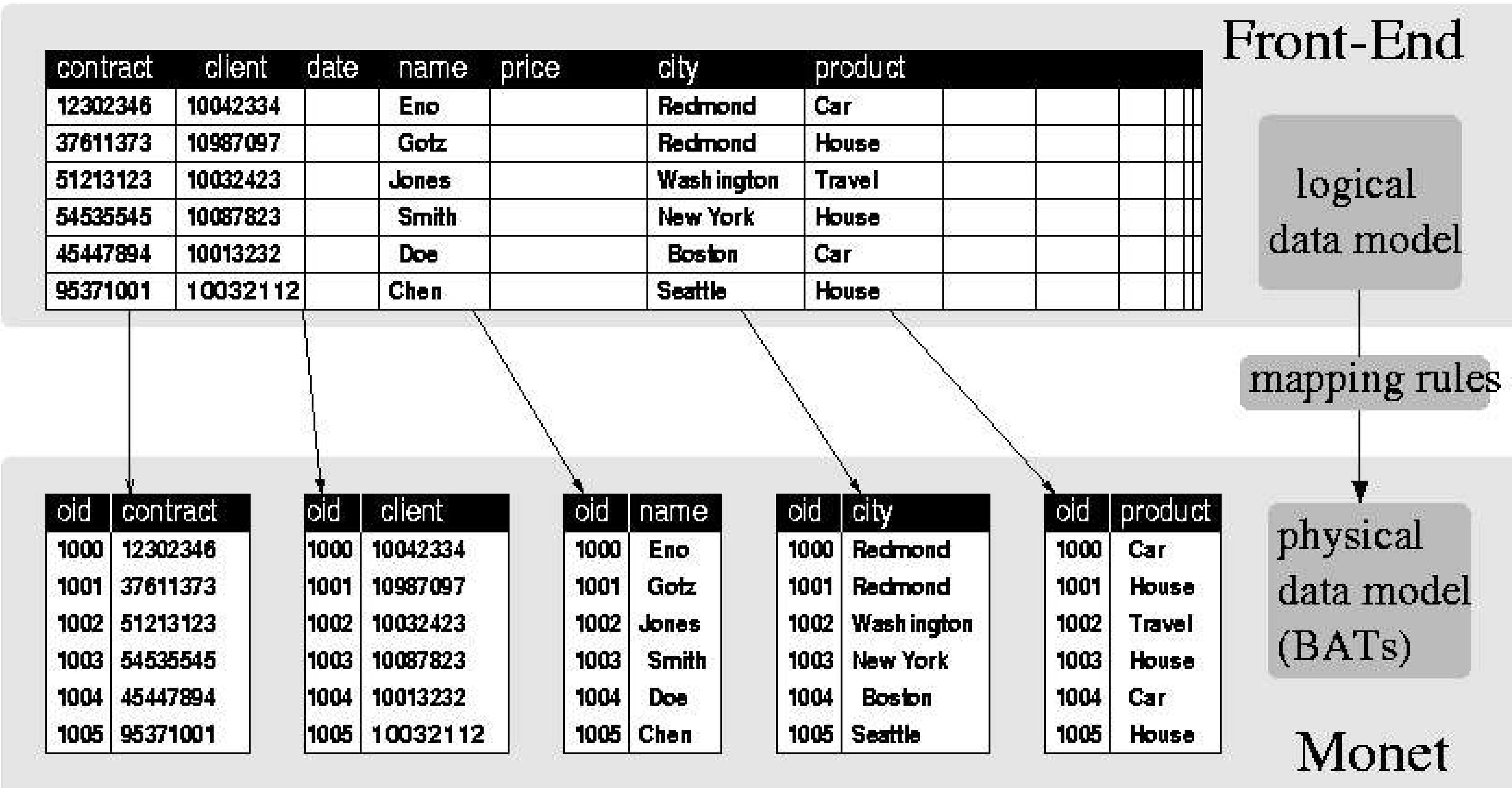
DBMS Architecture



MonetDB Architecture



Storing Relations in MonetDB



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

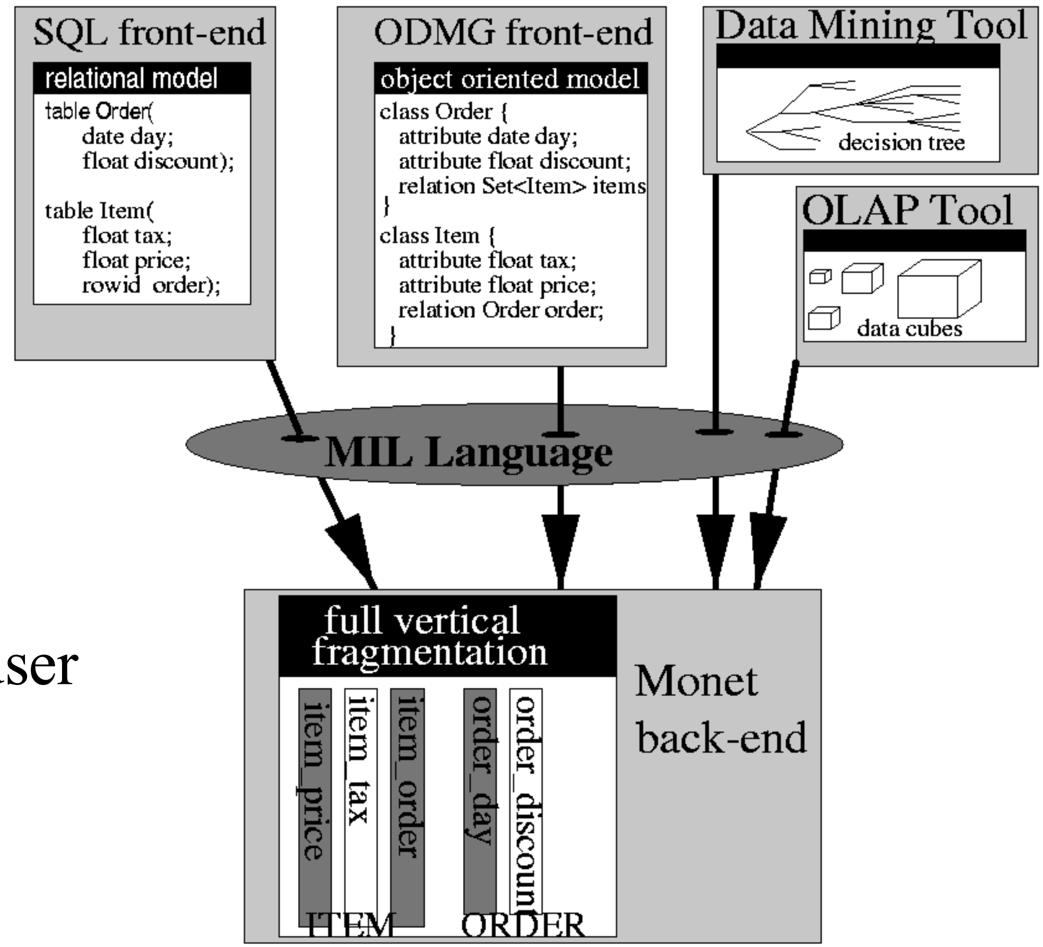
MonetDB: architecture

Front-end/back-end:

• support multiple data models

• support multiple end-user languages

• support diverse application domains



MonetDB: query language

MIL= MonetDB Interpreter Language

- algebraic language
- closed algebra on BATs
- bulk operations
- run time optimizations through scripting

MonetDB: architecture

"RISC"-approach to database systems

• very very simple data model, which is neutral to the high-level model the end user sees, so it can be used for many such high-level data models

• a limited number of query/update primitives, in order to keep the architecture simple, which ultimately enhances performance.

SQL => MIL example

SELECT SUM(price*tax), category, brand

FROM orders

WHERE date between 1-1-2000 and 1-2-2000

GROUPBY category, brand;

Date	price	tax	category	brand
31-12-1999	150.25	05.00	squash	nike
01-01-2000	150.00	05.00	squash	puma
15-01-2000	200.00	10.00	tennis	nike
29-01-2000	100.00	10.00	tennis	nike
01-02-2000	136.50	05.00	squash	nike

SUM(price*tax)	category	brand
07.50	squash	puma
30.00	tennis	nike

MIL Example

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

MIL Example

v1 := order_date.**select**(date("1-1-2000"), date("1-2-2000")).**mirror**;

order_date		v1		
[100, 31-12-1999]	=====>	[101, 01-01-2000]	=====>	[101, 101]
[101, 01-01-2000]	**select**	[102, 15-01-2000]	**mirror**	[102, 102]
[102, 15-01-2000]		[103, 29-01-2000]		[103, 103]
[103, 29-01-2000]				
[104, 01-02-2000]				

MIL Example

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

```
v2 := v1.join(order_price);
v3 := v1.join(order_tax);
```

```
v1            order_price          v2
[ 101, 101 ]  [ 100, 150.25 ]  =====>  [ 101, 150.00 ]
[ 102, 102 ]  [ 101, 150.00 ]   join   [ 102, 200.00 ]
[ 103, 103 ]  [ 102, 200.00 ]          [ 103, 100.00 ]
              [ 103, 100.00 ]
              [ 104, 136.50 ]          join( [X,Y], [Y,Z]) => [X,Z]
                                       |____|
v1            order_tax            v3
[ 101, 101 ]  [ 100, 05.00 ]  =====>  [ 101, 05.00 ]
[ 102, 102 ]  [ 101, 05.00 ]   join   [ 102, 10.00 ]
[ 103, 103 ]  [ 102, 10.00 ]          [ 103, 10.00 ]
              [ 103, 10.00 ]
              [ 104, 05.00 ]
```

MIL Example

v1 := order_date.select(date(“1-1-2000”), date(”1-2-2000”)).mirror;

v2 := v1.join(order_price);

v3 := v1.join(order_tax);

v4 := [*](v2,v3);

v2	v3		v4
[101, 150.00]	[101, 05.00]	===================>	[101, 07.50]
[102, 200.00]	[102, 10.00]	**multiplex *(flt,flt)**	[102, 20.00]
[103, 100.00]	[103, 10.00]		[103, 10.00]

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

[Y,c],..,[Y,d]   [Y, f(c,..,d)]

MIL Example

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

v2 := v1.join(order_price);

v3 := v1.join(order_tax);

v4 := [*](v2,v3);

v5 := **group**(order_category, order_brand).**reverse**;

order_category	order_brand				v5
[100, squash]	[100, nike]	=====>	[100, 100]	======>	[100, 100]
[101, squash]	[101, puma]	**group**	[101, 101]	**reverse**	[101, 101]
[102, tennis]	[102, nike]		[102, 102]		[102, 102]
[103, tennis]	[103, nike]		[103, 102]		[103, 103]
[104, squash]	[104, nike]		[104, 100]		[100, 104]

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”)).mirror;

v2 := v1.join(order_price);

v3 := v1.join(order_tax);

v4 := [*](v2,v3);

v5 := group(order_category, order_brand);

v6 := v5.mirror.**unique**;

v5		v6		
[100, 100]	=====>	[100, 100]	=====>	[100, 100]
[101, 101]	mirror	[101, 101]	**unique**	[101, 101]
[102, 102]		[102, 102]		[102, 102]
[102, 103]		[102, 102]		
[100, 104]		[100, 100]		

MIL Example

v1 := order_date.select(date(“1-1-2000”), date(”1-2-2000”)).mirror;

v2 := v1.join(order_price);

v3 := v1.join(order_tax);

v4 := [*](v2,v3);

v5 := group(order_category, order_brand).reverse;

v6 := v5.mirror.unique;

v7 := v5.join(v4).**{sum}**(v6);

v5 v4 v6 v7

[100, 100] [101, 07.50] ===> [101, 07.50] [100, 100] ===> [100, 00.00]

[101, 101] [102, 20.00] join [102, 20.00] [101, 101] **pump** [101, 07.50]

[102, 102] [103, 10.00] [102, 10.00] [102, 102] **{sum}** [102, 30.00]

[102, 103]

[100, 104] pump {f} assembles a mirrored BAT for each set, and calls f(BAT)

sum(empty)=>0.00 sum([07.50,07.50])=>07.50 sum ([10.00,10.00])=>30.00

[20.00,20.00]

MIL Example

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

v7 order_category order_brand

[100, nil] [100, squash] [100, nike] ==========> **00.00 squash nike**

[101, 07.50] [101, squash] [101, puma] **multiplex 07.50 squash puma**

[102, 15.00] [102, tennis] [102, nike] **sprintf(..) 30.00 tennis nike**

[103, tennis] [103, nike]

[104, squash] [104, nike]

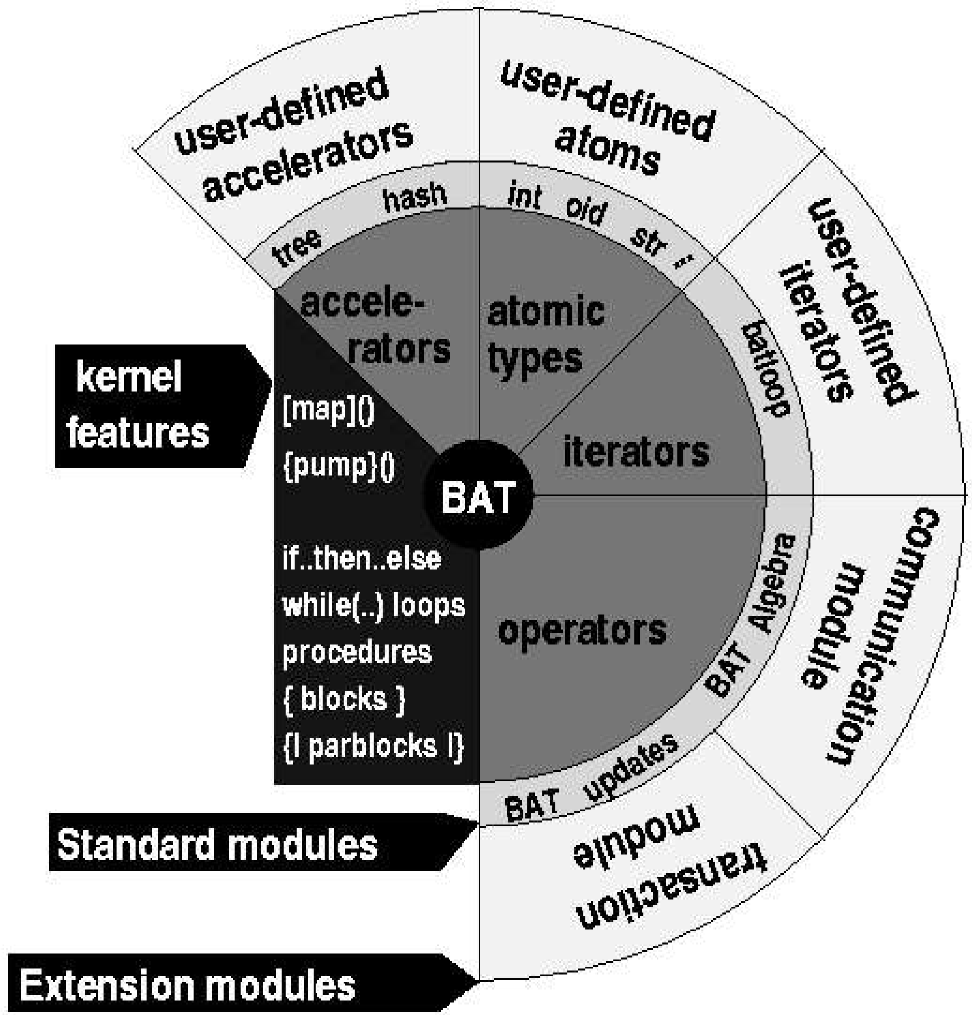
MonetDB: extensibility

MEL= MonetDB Extension Language

- add new datatypes to MIL
  - url, image, point, polyline, etc..
- add new commands to MIL
  - intersection(polygon, polygon) : polygon
  - convolution(image, ..params..) : image
- add new search accelerators to MIL
  - GIS: R-tree
  - image processing?

MEL specifications packaged in **modules**

MIL Extensibility



What is MonetDB Good for?

- Query-intensive application
- very high performance demanding
- complex data models
- complex query primitives

MonetDB: performance

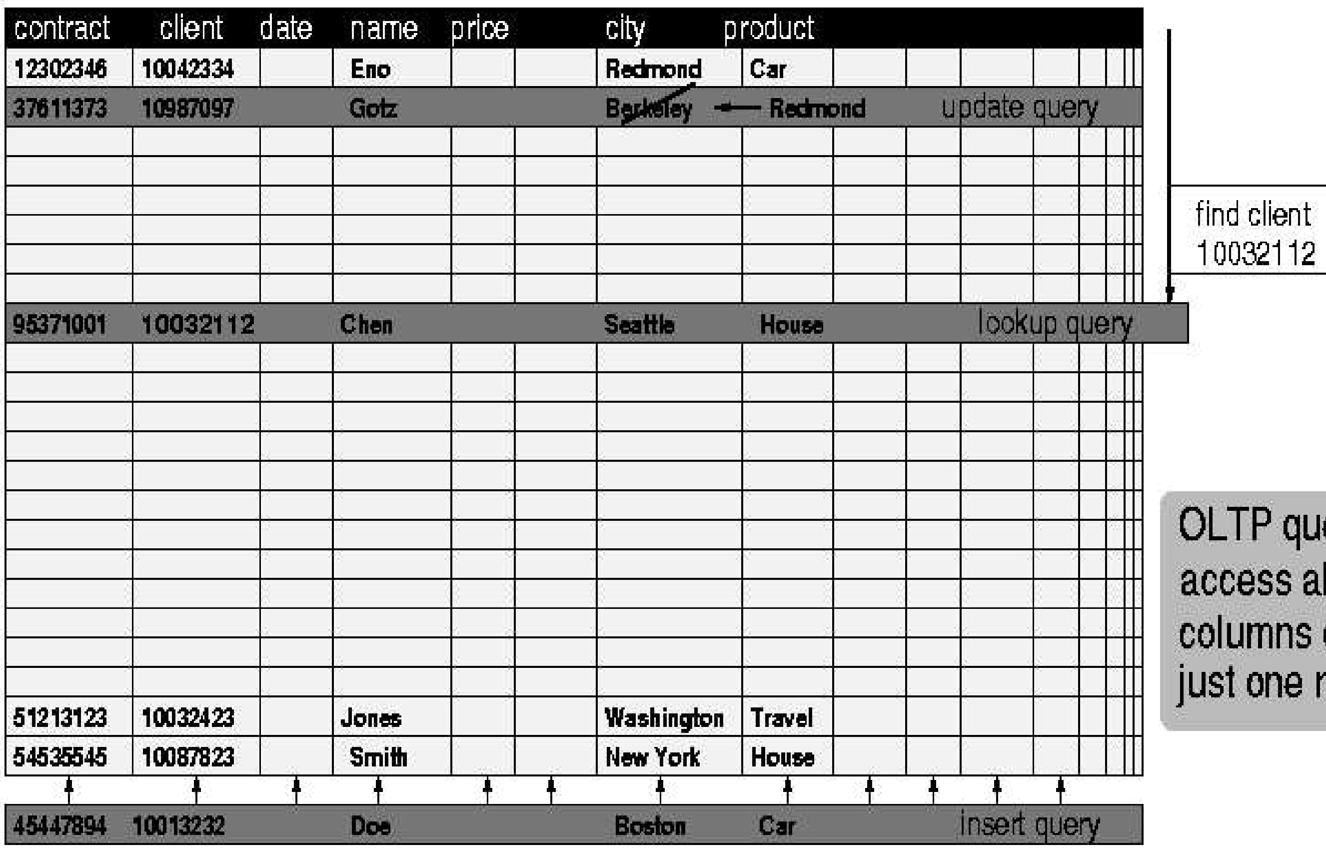
Emphasis on efficient implementation

- implementation techniques (code expansion)
- memory cache optimizations
- O(>=1) faster than normal DBMS

Parallel Processing

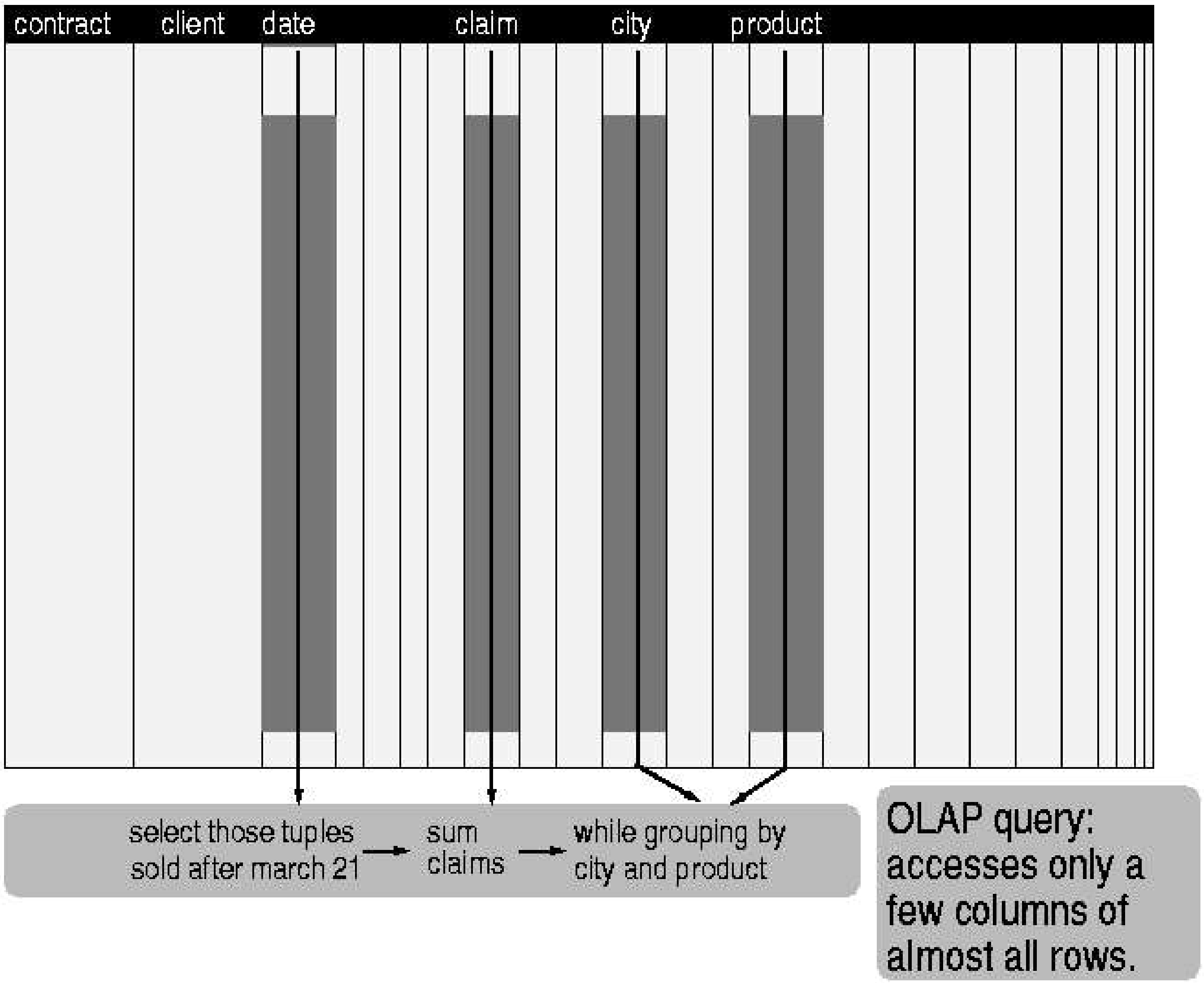
- SMP: multi-threaded + MIL language support
- MPP: multi-server shared-nothing over tcp/ip connections

Transactions (OLTP)



OLTP queries: access all columns of just one row.

OLAP, Data Mining



Relational Mapping

Order Table

id	day	discount
10	4/4/98	0.175
11	9/4/98	0.065
12	1/2/98	0.175
13	9/4/98	0.000
14	7/2/98	0.000
15	1/2/98	0.065

order_id

oid	int
100	10
101	11
102	12
103	13
104	14
105	15

order_day

oid	date
100	4/4/98
101	9/4/98
102	1/2/98
103	9/4/98
104	7/2/98
105	1/2/98

order_discount

oid	float
100	0.175
101	0.065
102	0.175
103	0.000
104	0.000
105	0.065

Item Table

order	price	tax
10	04.75	0.10
10	11.50	0.00
11	10.20	0.00
11	75.00	0.00
11	02.50	0.00
12	92.80	0.10
13	37.50	0.10
13	14.25	0.00
13	17.99	0.00
14	22.33	0.00
14	42.67	0.10

item_order

oid	int
1000	10
1001	10
1002	11
1003	11
1004	11
1005	12
1006	13
1007	13
1008	13
1009	14
1010	14

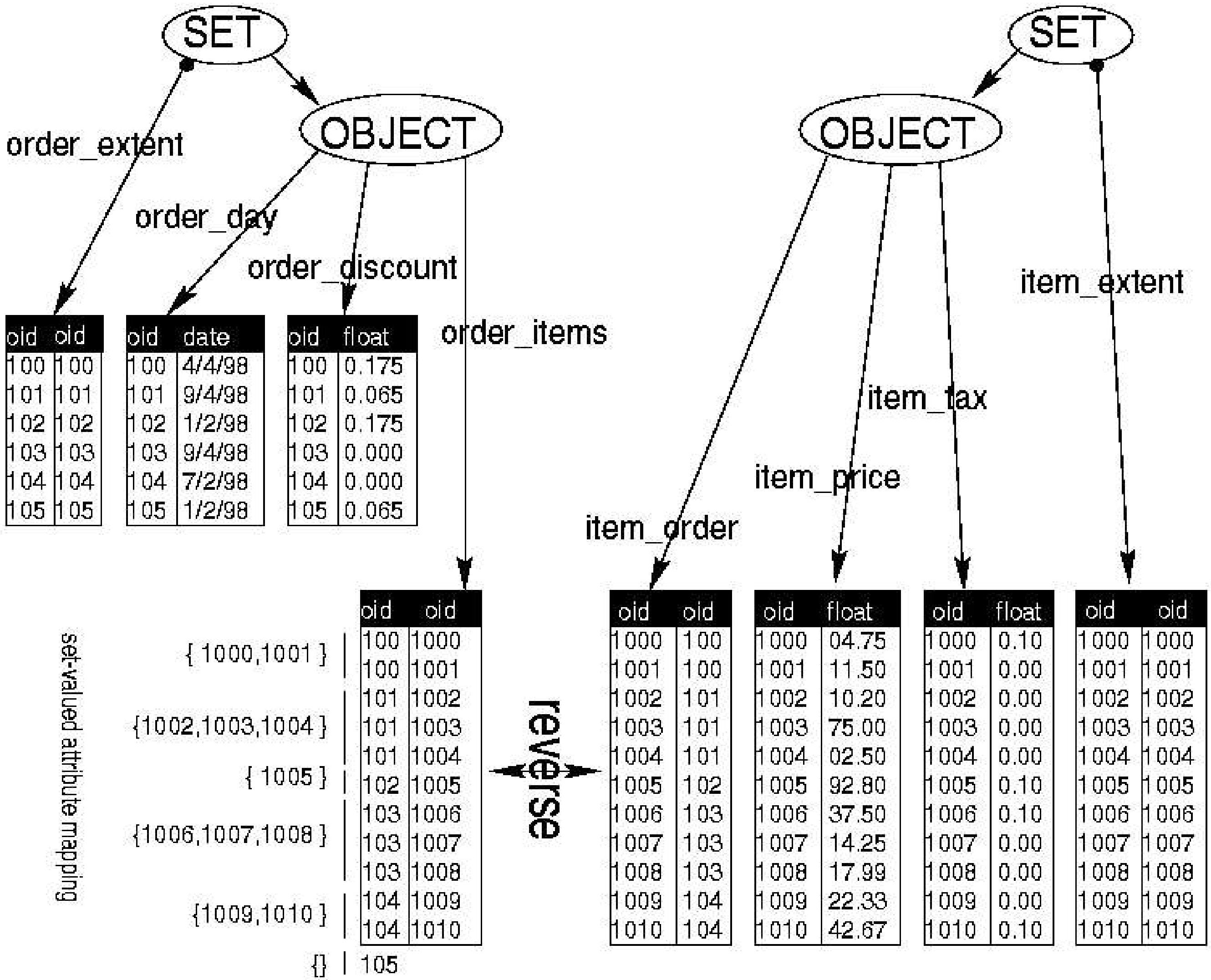
item_price

oid	float
1000	04.75
1001	11.50
1002	10.20
1003	75.00
1004	02.50
1005	92.80
1006	37.50
1007	14.25
1008	17.99
1009	22.33
1010	42.67

item_tax

oid	float
1000	0.10
1001	0.00
1002	0.00
1003	0.00
1004	0.00
1005	0.10
1006	0.10
1007	0.00
1008	0.00
1009	0.00
1010	0.10

Object-Oriented Mapping



oid	oid
100	100
101	101
102	102
103	103
104	104
105	105

oid	date
100	4/4/98
101	9/4/98
102	1/2/98
103	9/4/98
104	7/2/98
105	1/2/98

oid	float
100	0.175
101	0.065
102	0.175
103	0.000
104	0.000
105	0.065

oid	oid
100	1000
100	1001
101	1002
101	1003
101	1004
102	1005
103	1006
103	1007
103	1008
104	1009
104	1010

oid	oid
1000	100
1001	100
1002	101
1003	101
1004	101
1005	102
1006	103
1007	103
1008	103
1009	104
1010	104

oid	float
1000	04.75
1001	11.50
1002	10.20
1003	75.00
1004	02.50
1005	92.80
1006	37.50
1007	14.25
1008	17.99
1009	22.33
1010	42.67

oid	float
1000	0.10
1001	0.00
1002	0.00
1003	0.00
1004	0.00
1005	0.10
1006	0.10
1007	0.00
1008	0.00
1009	0.00
1010	0.10

oid	oid
1000	1000
1001	1001
1002	1002
1003	1003
1004	1004
1005	1005
1006	1006
1007	1007
1008	1008
1009	1009
1010	1010

New Domains: GIS

- New data types (point, polygon, etc..)
- New search accelerators (R-Tree, etc..)
- New primitives boolean intersects(polygon,polygon)
- Complex topological structures -
  stored in DCELs that are decomposed over BATS
  queries are efficient due to MonetDB high join speed

New Domains: Multimedia

- New data types (url, image, etc..)
- new search accelerators (color histograms)
- new primitives (similarity search)
- complex data structures:
  bayesian inference networks (information retrieval)
  again decomposed in BATs and efficient to query

Implementation Highlights

- motivation based on hardware trends
- data structures
- algorithms

=> focus on join algorithms

=> focus on memory cache optimizations

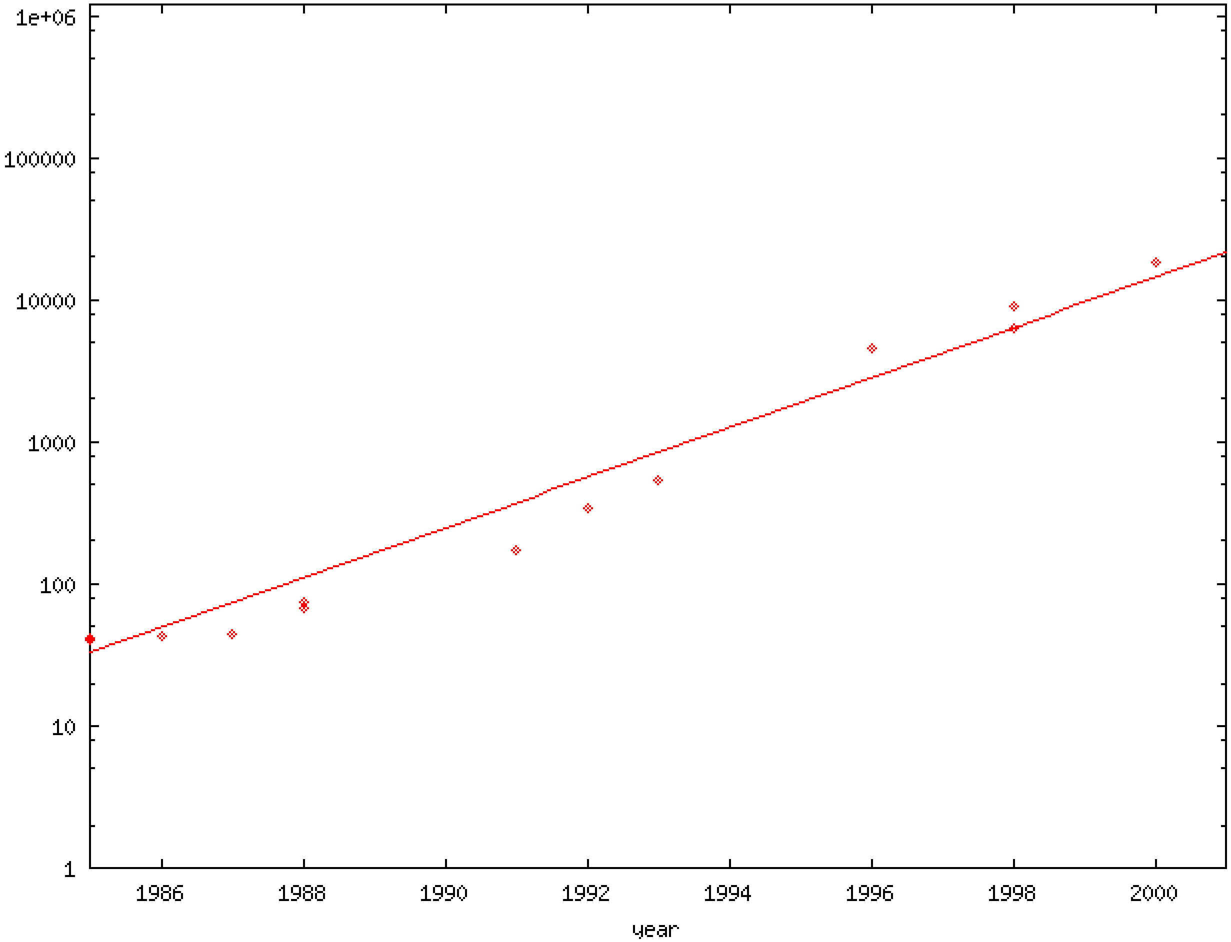
Computer Hardware Trends

year	computer model	processor			memory		
		type	MHz	# par. units	STREAM/copy (bandwidth)	typical size (MB)	latency (ns)
1989	Sun 3/60	68020	20	1	6.5	16	
1990	Sun 3/80	68030	20	1	4.9	16	
1991	Sun 4/280	Sparc	17	1	9.6	16	160
1992	Sun ss10/31	superSparc I	33	3	42.9	32	
1993	Sun ss10/41	superSparc I	40	3	48.0	32	
1994	Sun ss20/71	superSparc II	75	3	62.5	64	870
1995	Sun Ultra1 170	ultraSparc I	167	5	225.2	128	225
1996	Sun Ultra2 2200	ultraSparc II	200	5	228.5	256	225
1996	SGI Power Chall.	R10000	195	5	172.7	128	610
1997	SGI Origin 2000	R10000	250	5	332.0	256	424
1998	SGI Origin 2000	R12000	300	5	336.0	256	404
1992	Intel PC	80486	66	1	33.3	8	
1993	Intel PC	Pentium	60	2	47.1	8	161
1994	Intel PC	Pentium	90	2	46.4	8	161
1995	Intel PC	Pentium	100	2	85.1	8	161
1996	Intel PC	Pentium	133	2	84.4	16	161
1996	Intel PC	PentiumPro	200	5	140.0	16	203
1997	Intel PC	PentiumII	300	5	188.2	32	145
1998	Intel PC	PentiumII	350	5	279.3	32	145
1998	Intel PC	PentiumII	400	5	304.0	32	145
1999	Intel PC	PentiumIII	600	5	379.2	64	135
2000	Intel PC	PentiumIII	733	5	441.9	128	135
1999	AMD PC	Athlon	500	9	373.5	64	217
2000	AMD PC	Athlon	800	9	387.9	128	217

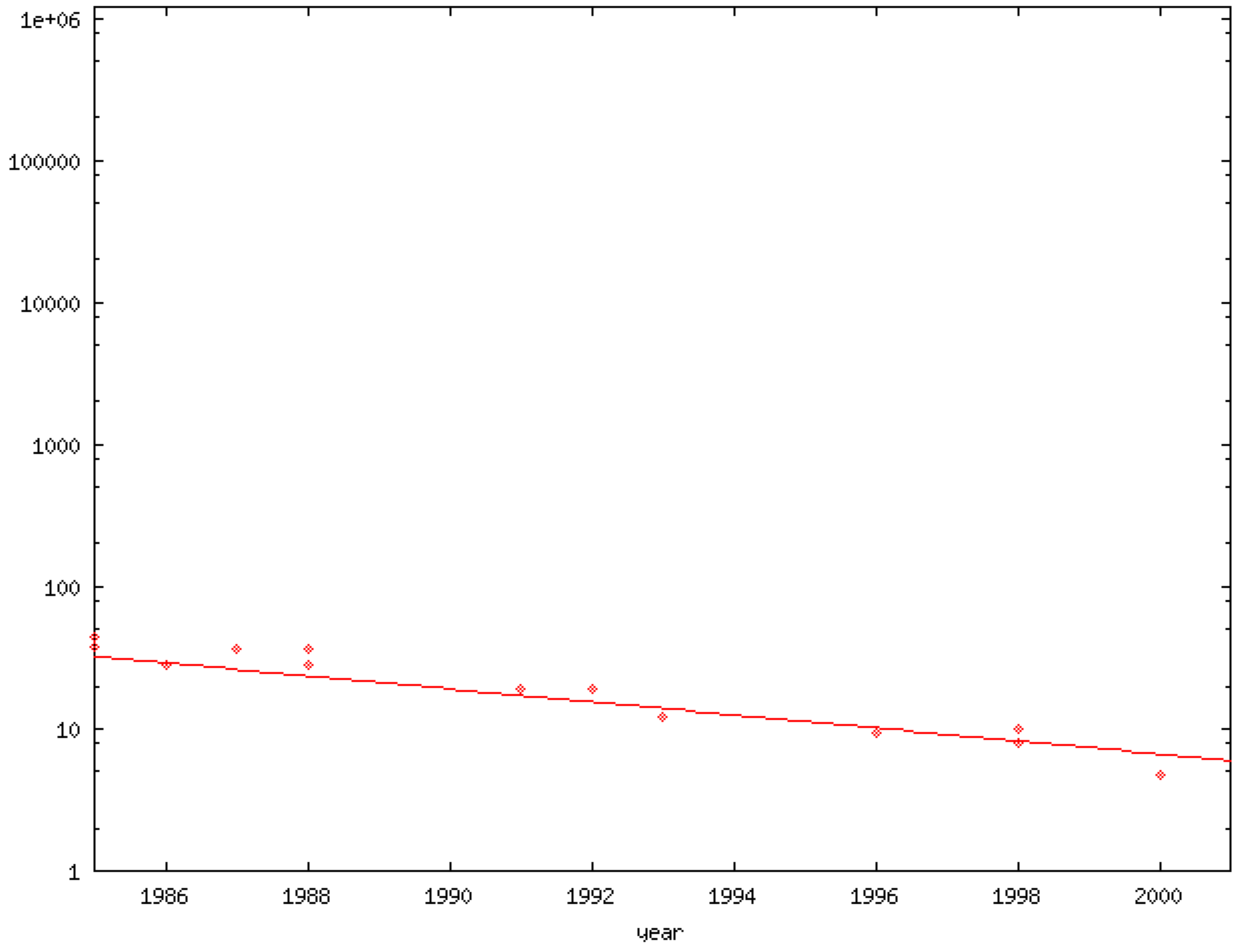
Disk Hardware Trends

year	hard disk model		size (Mb)	latency (ms)	rotations /minute	bandwidth (MB/s)	cache (MB)
1979	Tandon	TM-602	5	75	3600	0.62	0
1981	Tandon	TM-252	10	85	3600	0.62	0
1983	Seagate	ST-225	20	65	3600	0.62	0
1984	CMS	LT LD20	21.4	75	2640	0.9	0
1984	Mitsubishi	MR522	25.5	85	3536	0.9	0
1985	Quantum	Q540	40	45	3550	0.9	0
1985	Seagate	ST-251	42	38	3600	0.9	0
1986	CDC	94205-5	43	28	3597	0.9	0
1986	Microscienc	HH-1050	43	28	3436	0.9	0
1987	Rodime	RO3055	45	36.3	3600	0.9	0
1988	Rodime	RO5090	74.6	36.3	3600	0.9	0
1988	Micropolis	1325	67.1	28	3600	0.9	0
1991	CDC	94221-190	170	19	3597	1.1	0
1992	CDC	94171-344	344	19	3597	1.5	0
1993	Tandy	250-4168	540	12	4498	1.5	0
1996	Quantum	Empire	4500	9.5	5400	3.6	0
1998	Quantum	Fireball	6400	10	5400	14	0.5
1998	Quantum	Atlas II	91008	8	7200	14	2
2000	Quantum	Atlas10K II	18400	5.0	10000	24	8

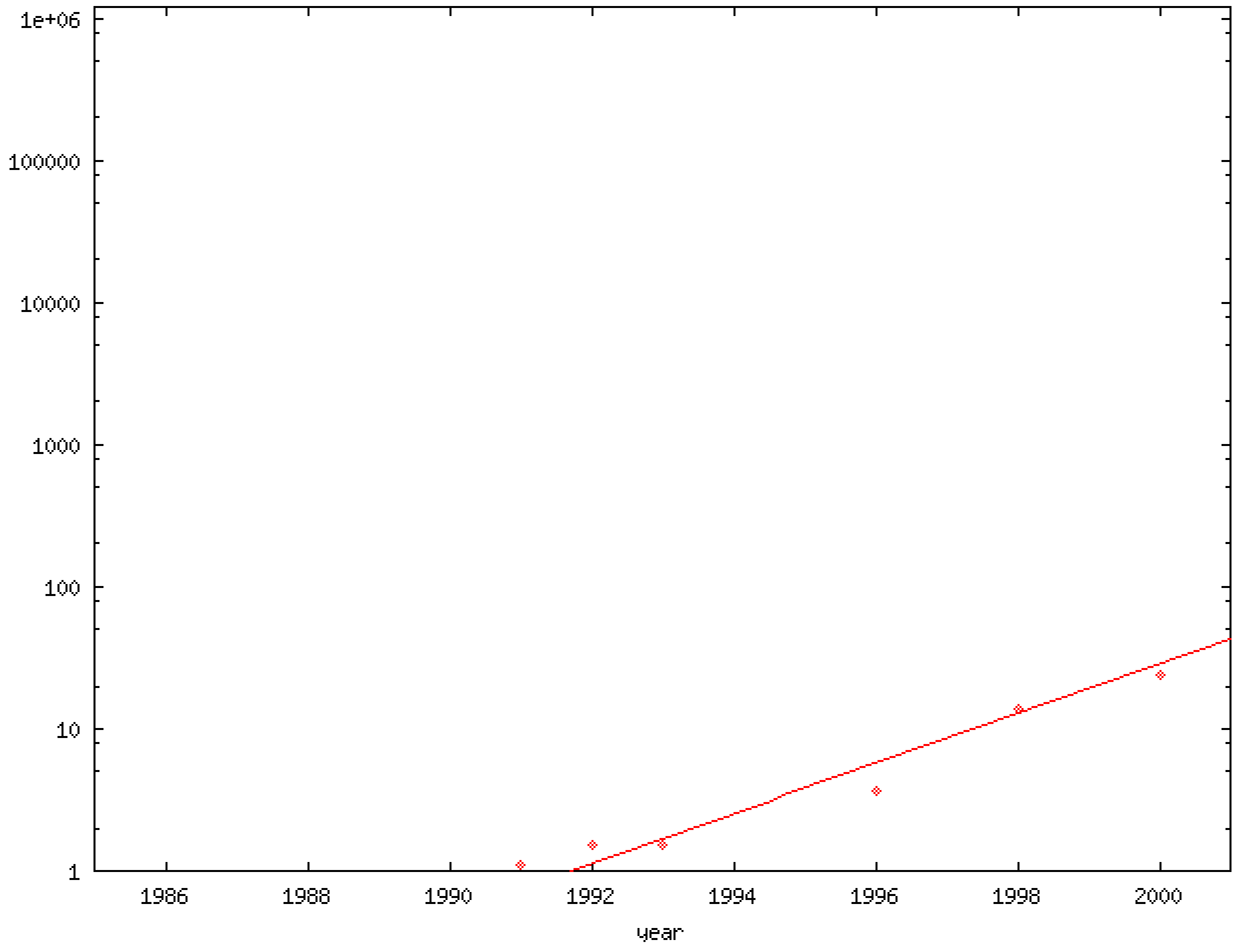
Disk capacity (MB)



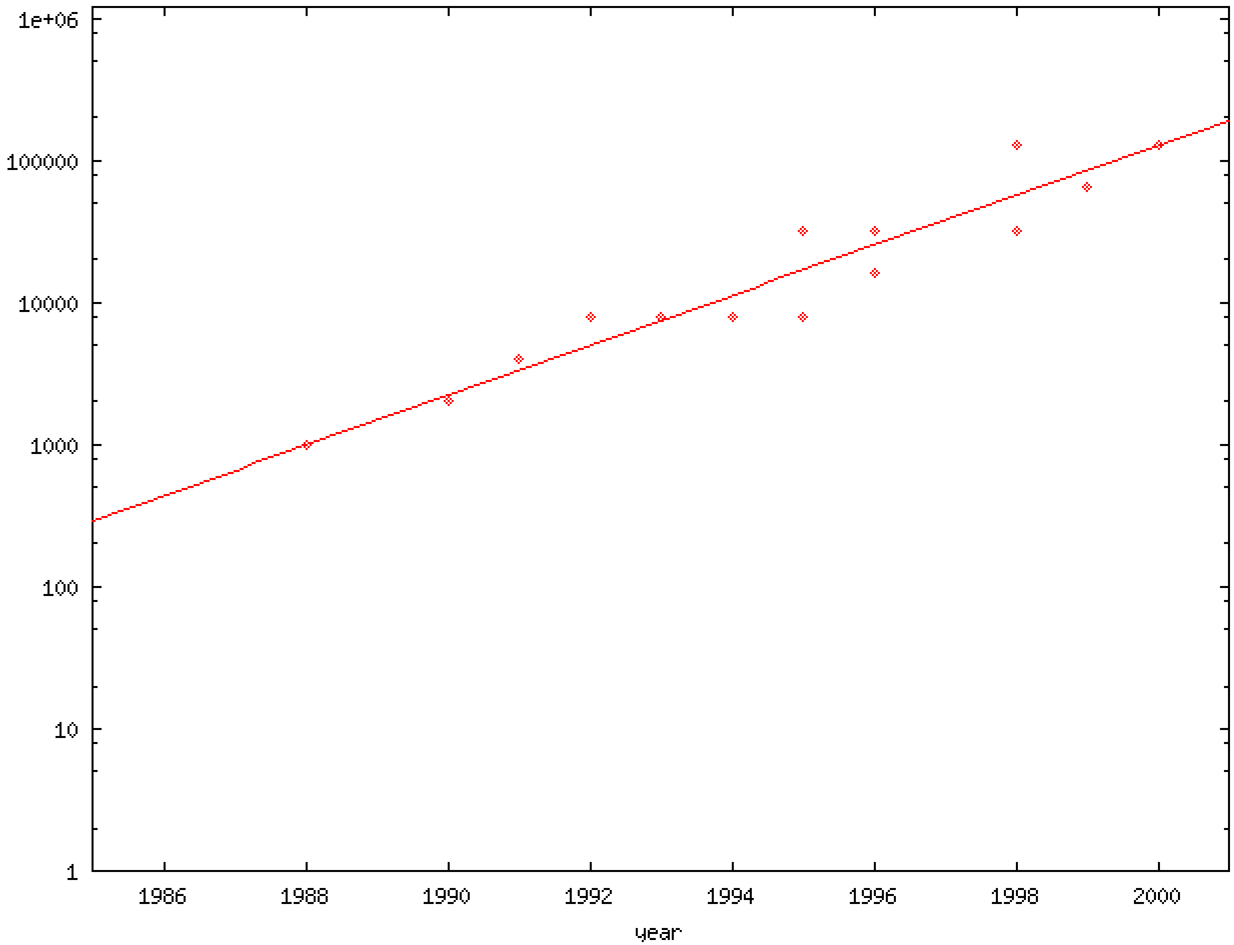
Disk latency (ms)



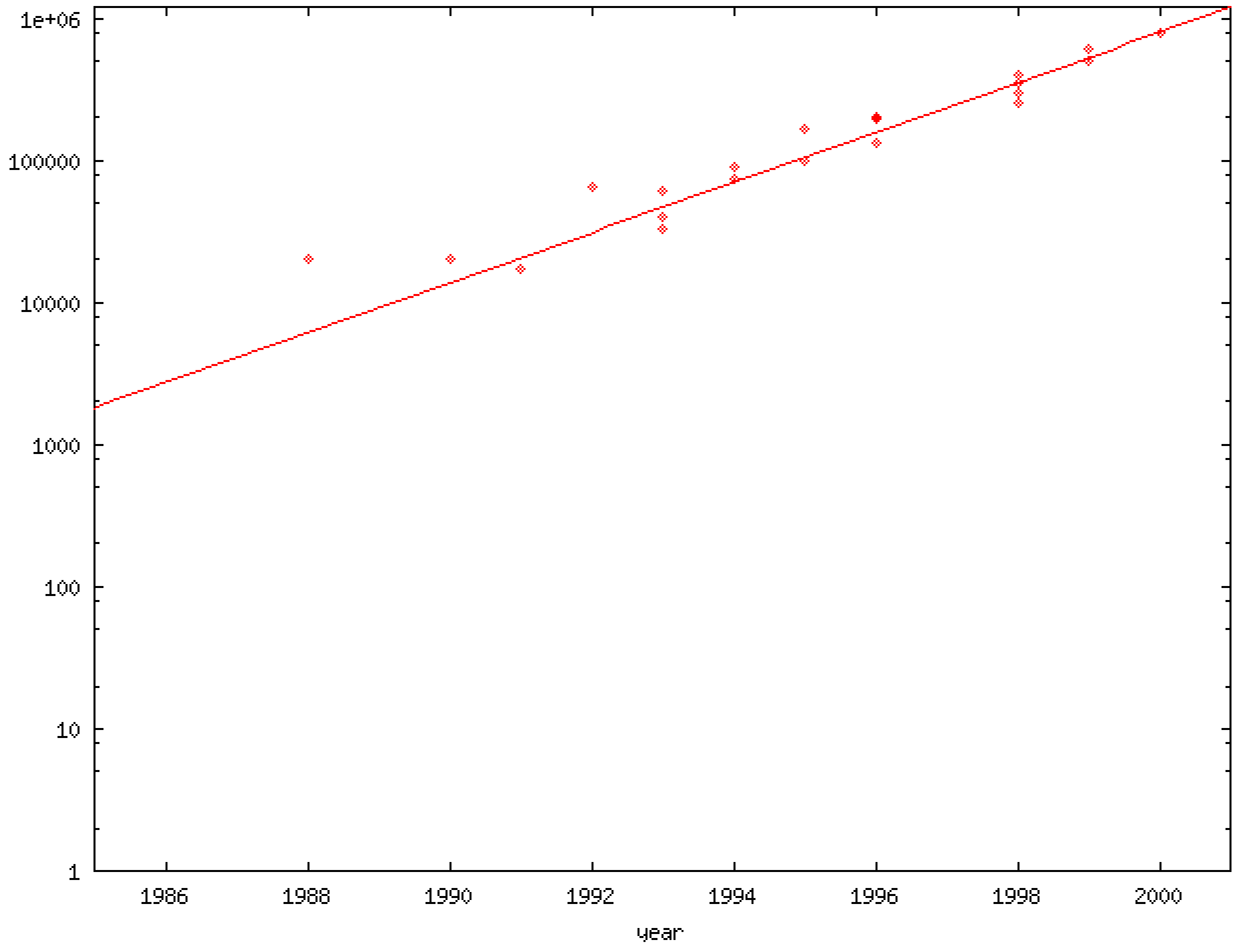
Disk bandwidth (MB/s)



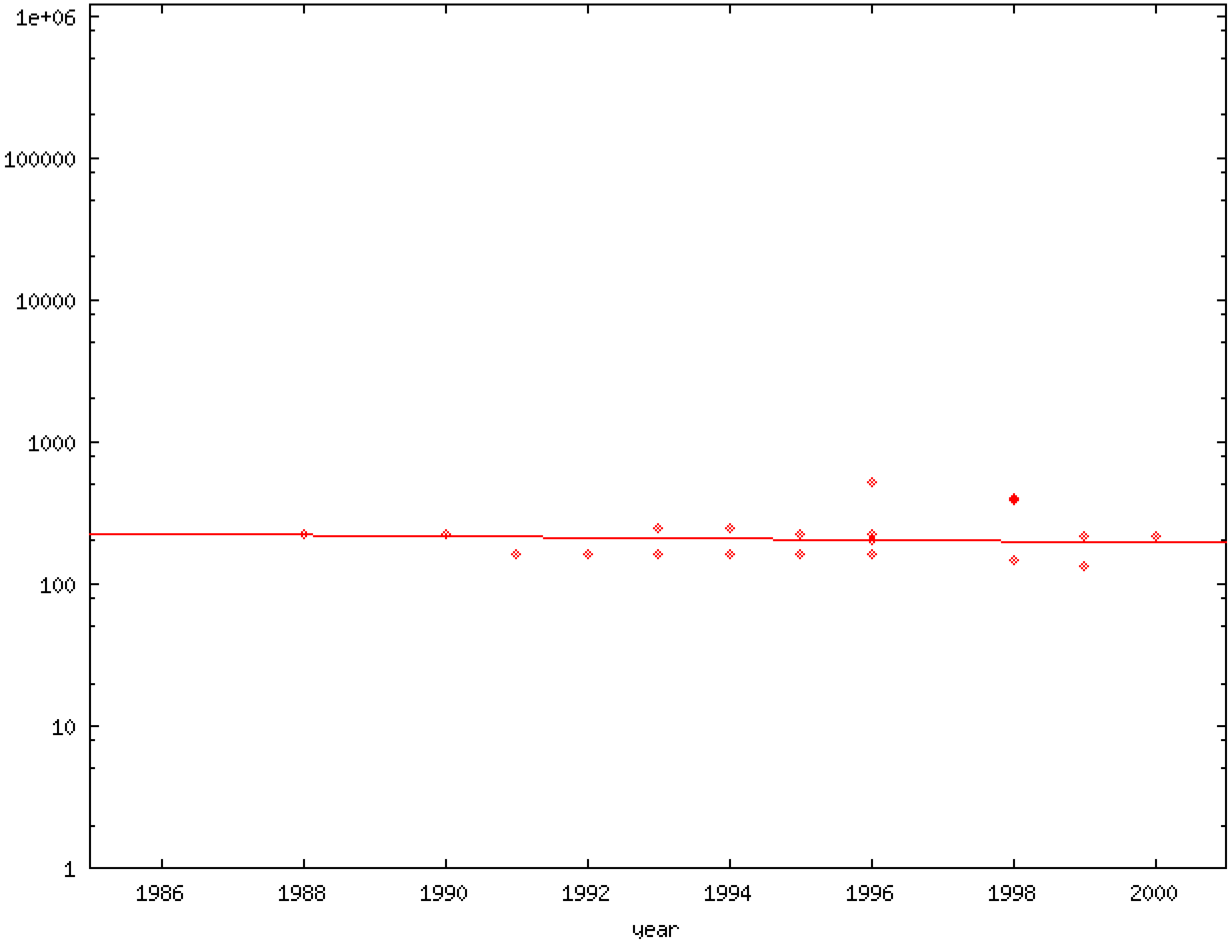
typical memory size (MB)



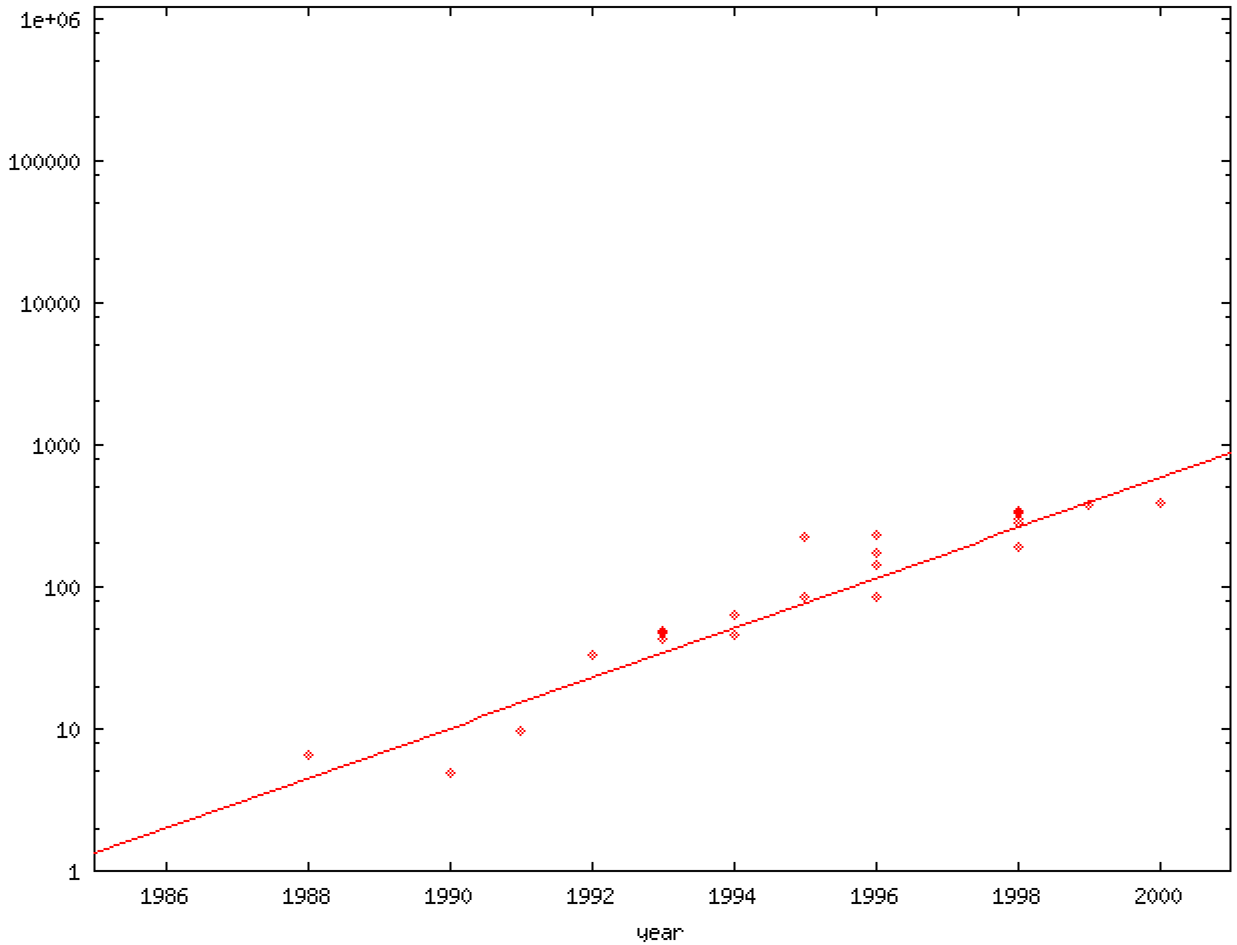
CPU clockspeed (Mhz)



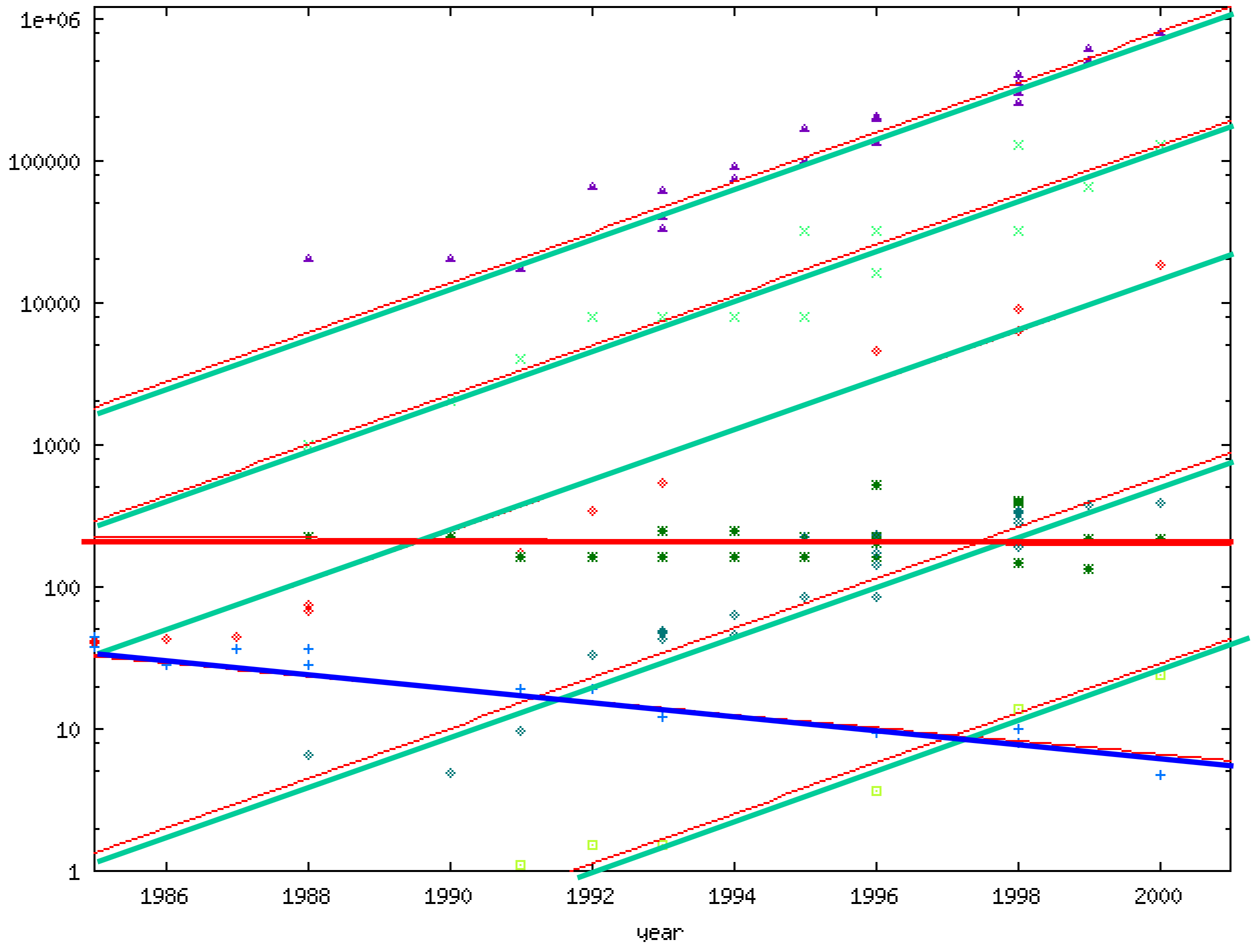
Memory latency (ns)



Memory Bandwidth (MB/s)



Hardware Trends



50% p/year:

- cpu speed
- mem size
- mem bandwidth
- disk bandwidth

1% p/year:

- mem latency

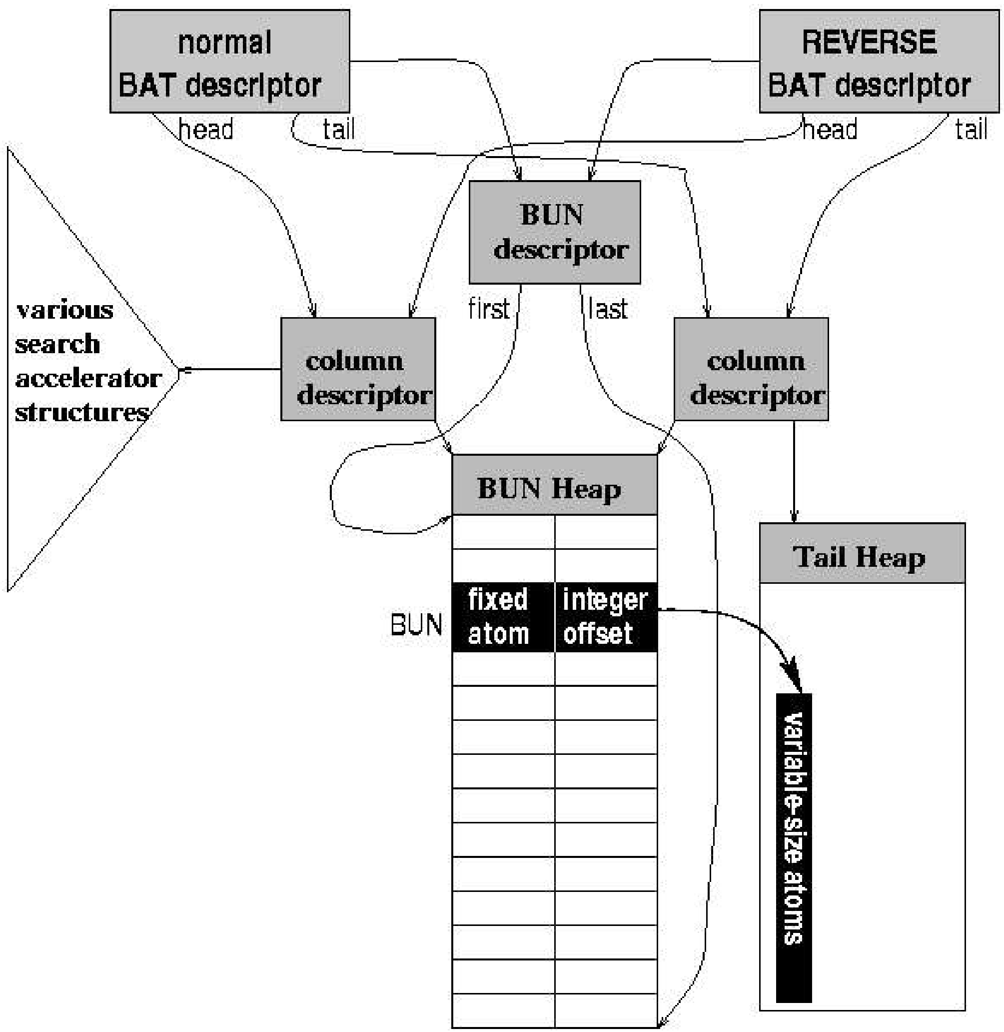
10% p/year:

- disk latency

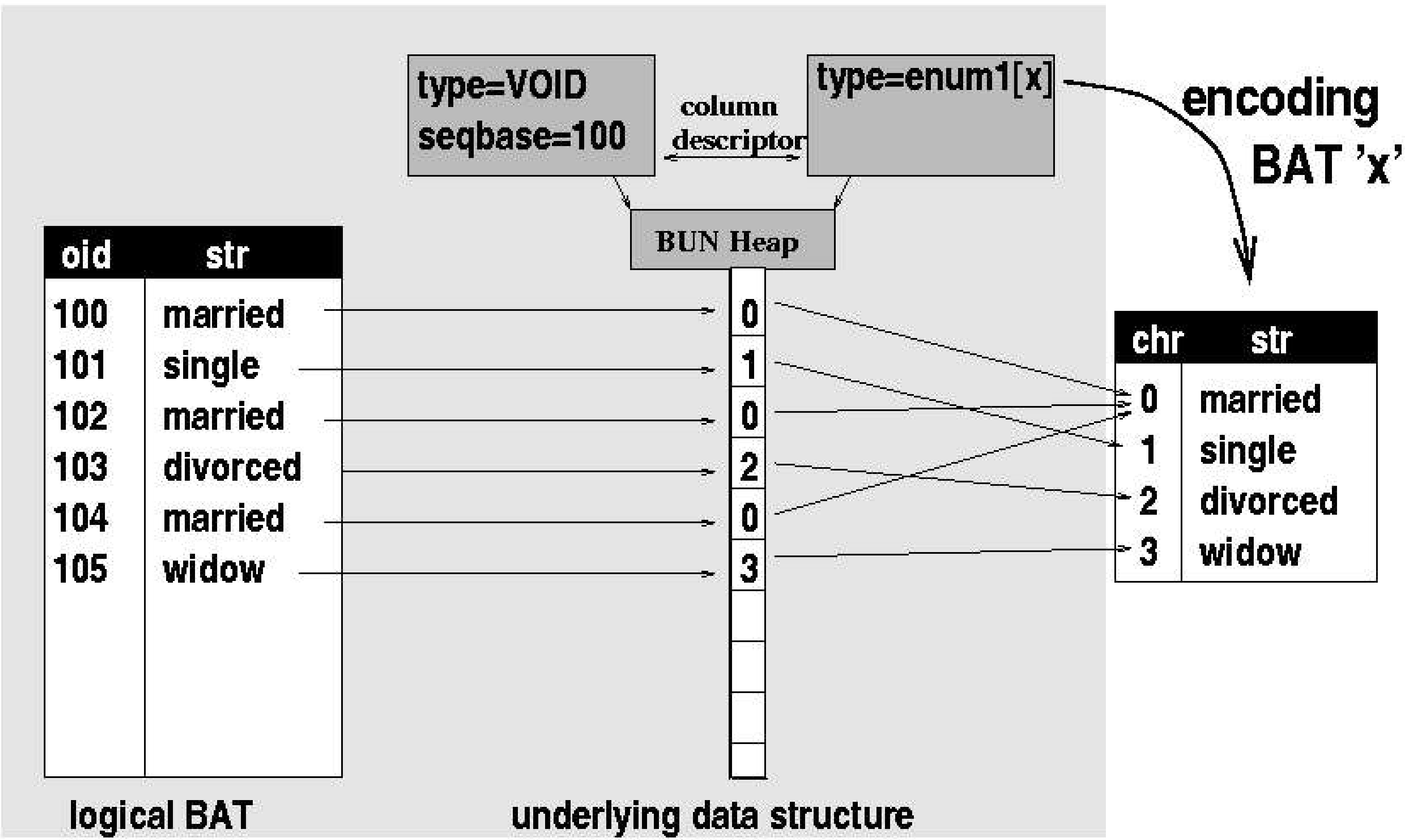
Latency is the enemy!

- Commercial DBMS products (oracle, DB2, SQLserver) stem from OLTP roots
- focus on minimizing random I/Os => depend on latency!
- MonetDB: built for bulk access
- optimize CPU and memory performance

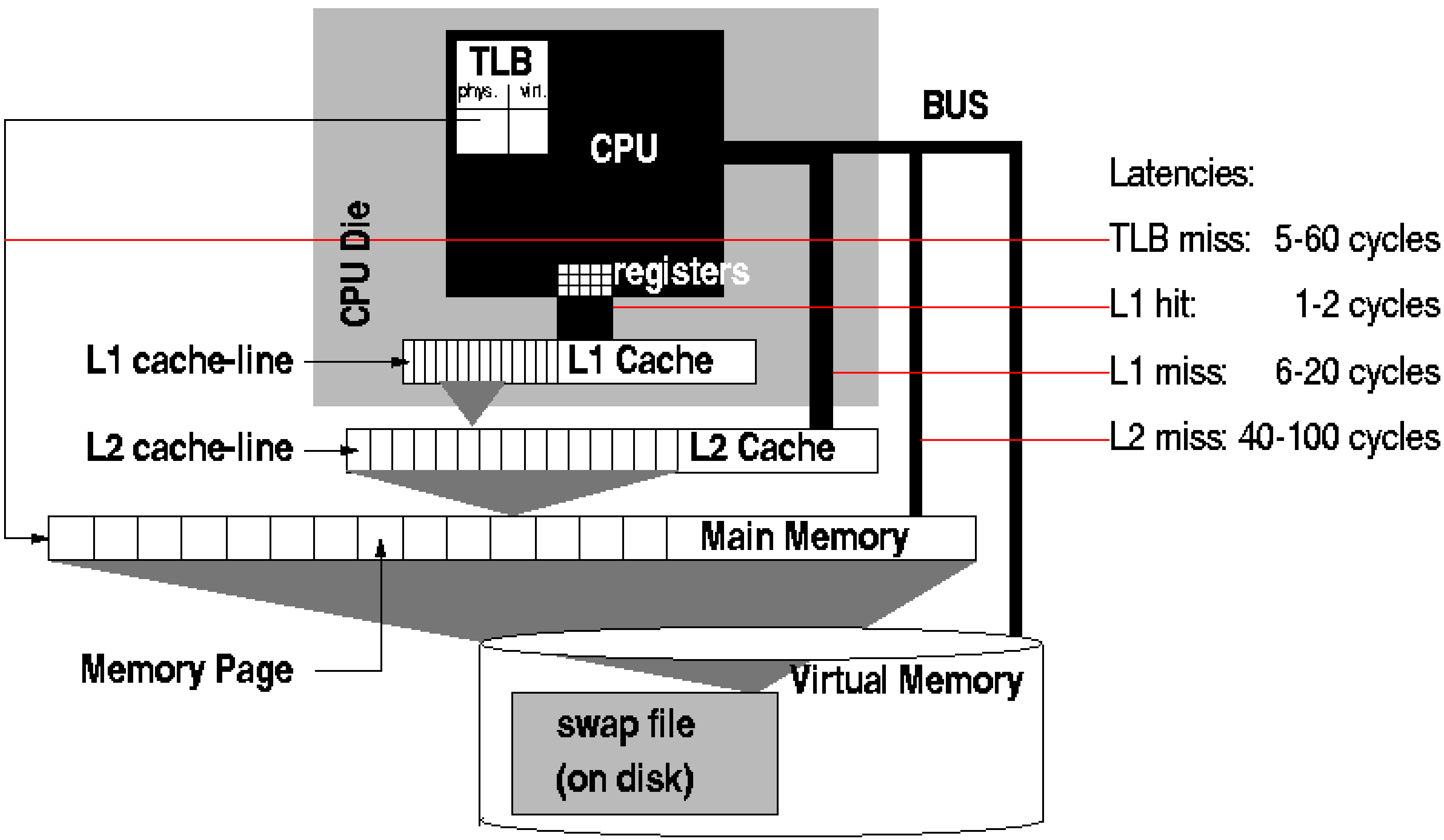
BAT Data structure



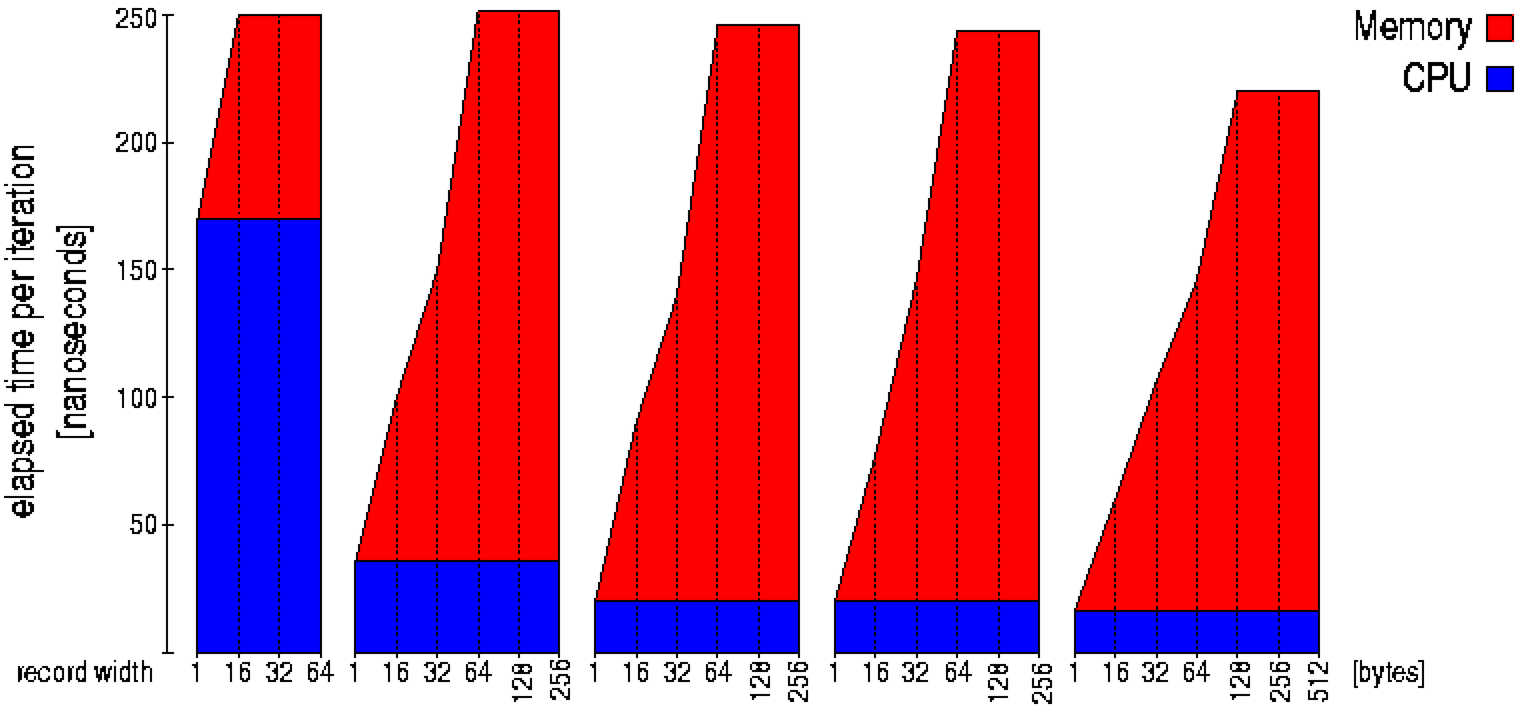
BAT Storage Optimizations



Memory Access in Hierarchical Systems



Simple Scan Experiment



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

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: join)

Join ([X,Y],[Y,X]) => [X,Z] **Algorithms**

- Nested loop

```
for all tuples X in INNER
    for all tuples Y  in OUTER
        if X == Y INSERT
```

- Void join

```
for all tuples X in INNER
    Y = lookup(X-base)
        if X == Y INSERT
```

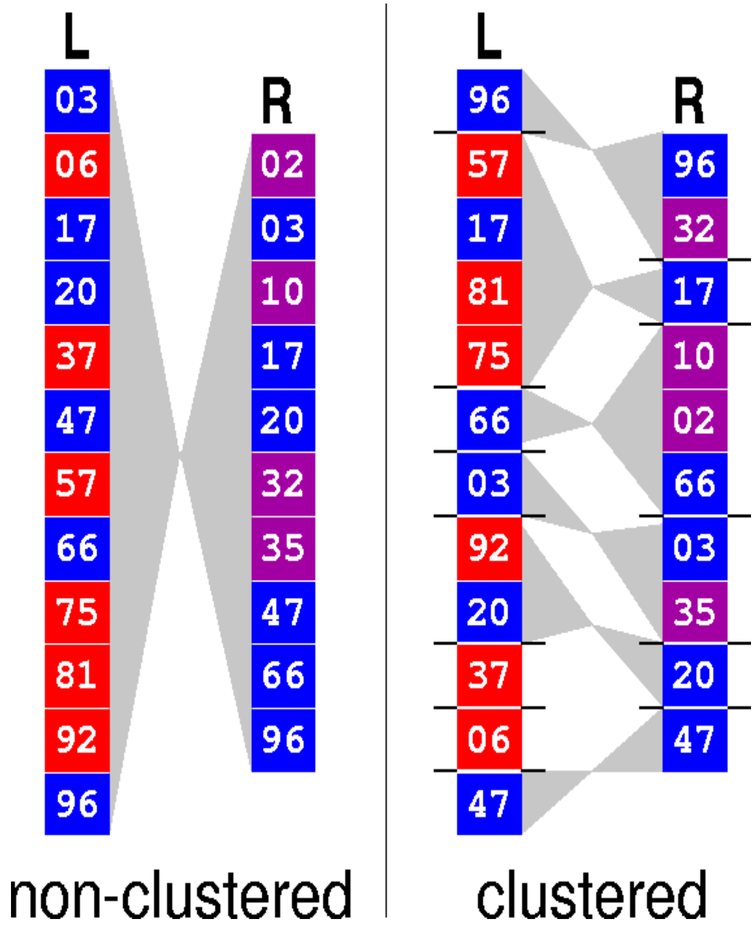
- merge join

```
sort(INNER)
sort(OUTER)
scan INNER and OUTER
    if X == Y INSERT
```

- **hash join**

Partitioned Joins

- Cluster both input relations
- Create clusters that fit in memory cache
- Join matching clusters
- Two algorithms:
  - Partitioned hash-join
  - Radix-Join
    (partitioned nested-loop)



Partitioned Joins: Straightforward Clustering

- Problem:

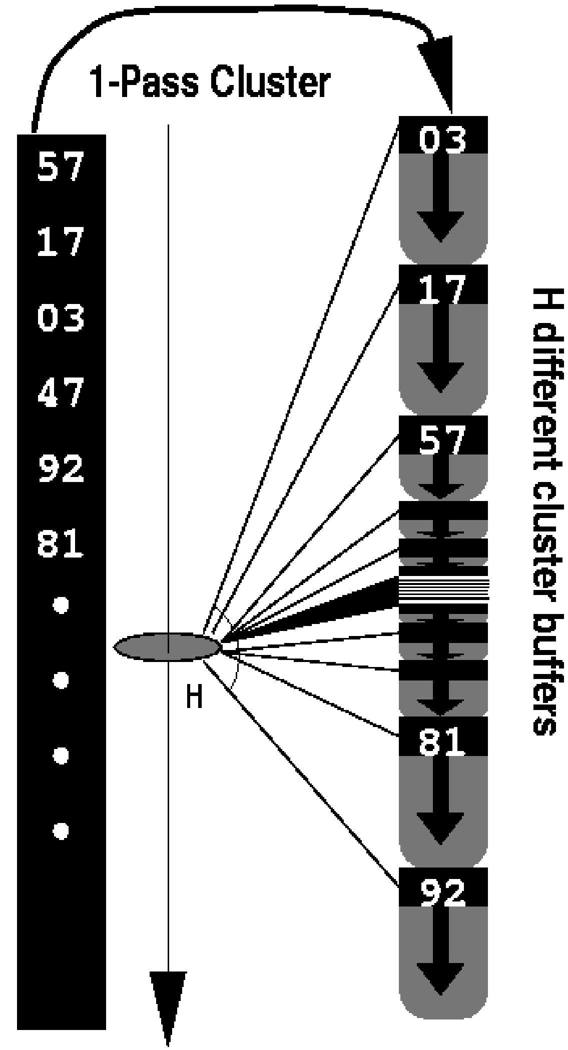
  Number of clusters exceeds number of

  – TLB entries ==> TLB trashing

  – Cache lines ==> cache trashing

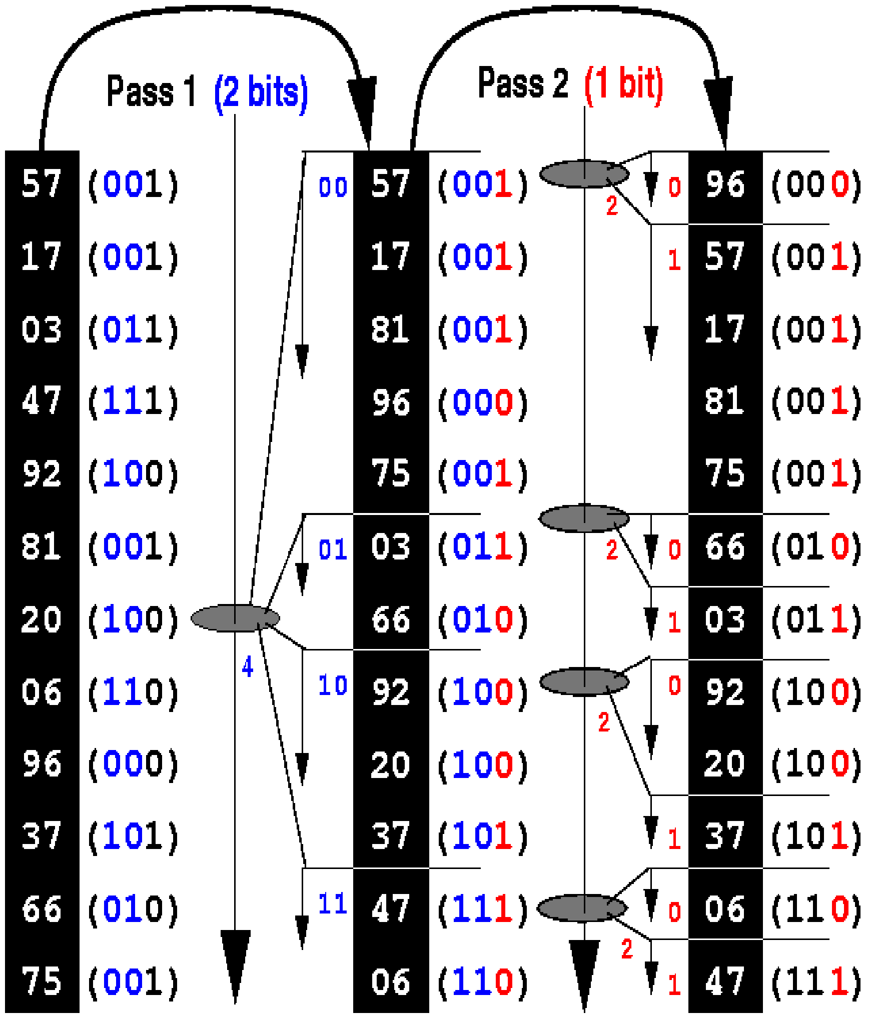
- Solution:

  Multi-pass radix-cluster



Partitioned Joins: Multi-Pass Radix-Cluster

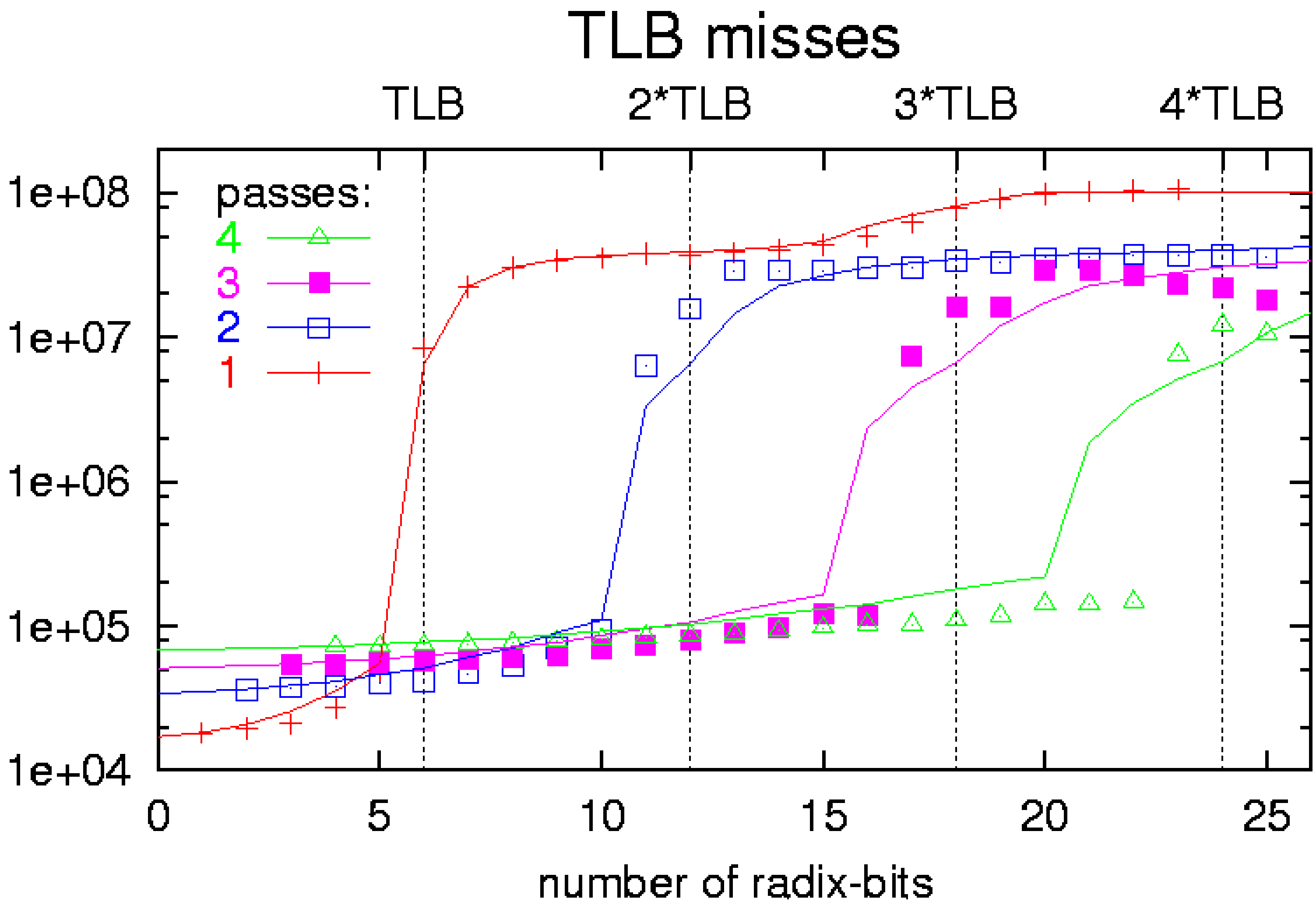
- Multiple clustering passes
- Limit number of clusters per pass
- Avoid cache/TLB trashing
- Trade memory cost for CPU cost
- Any data type (hashing)



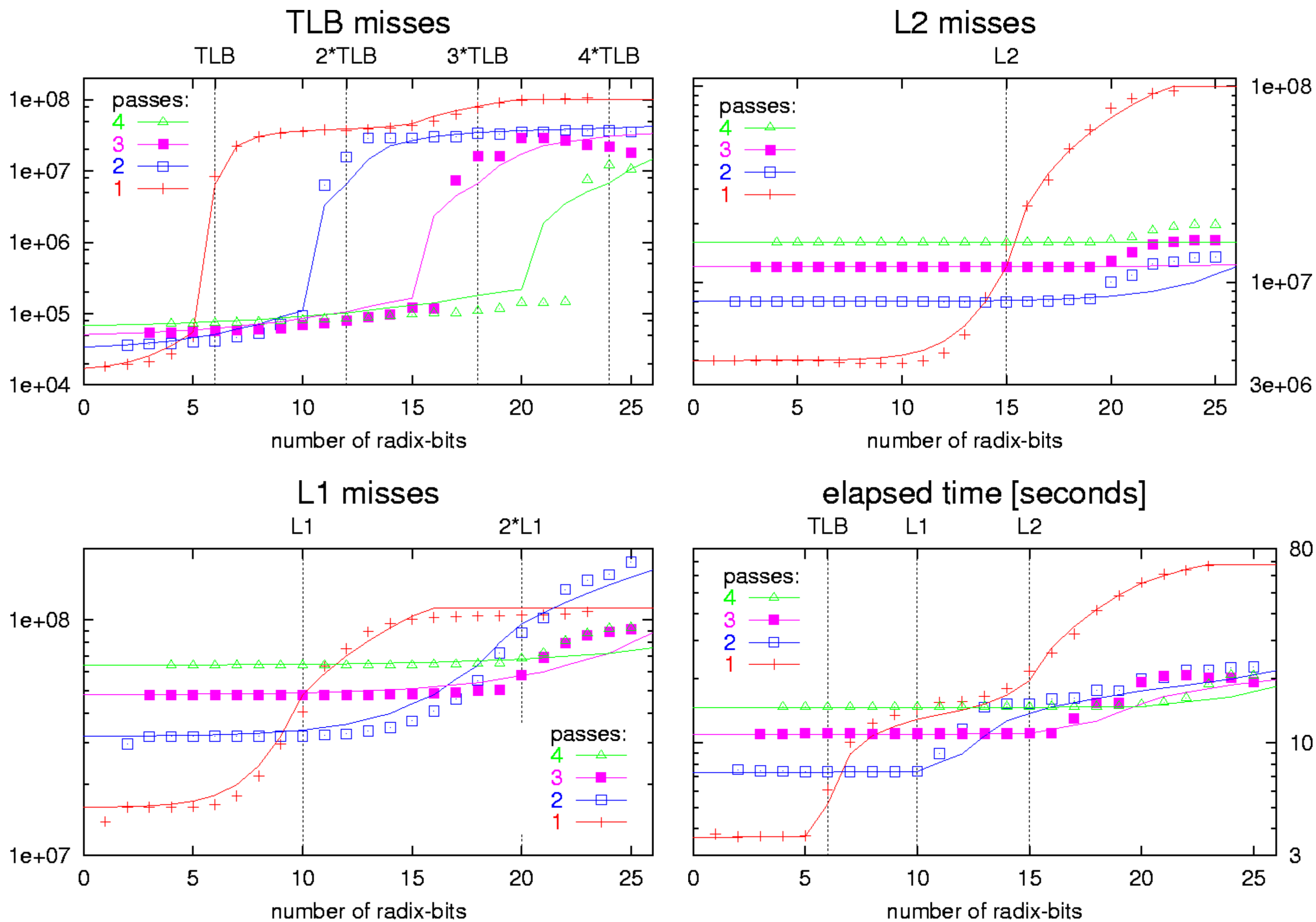
MonetDB Experiments: Setup

- Platform:
  - SGI Origin2000 (MIPS R10000, 250 MHz)
- System:
  - MonetDB DBMS
- Data sets:
  - Integer join columns
  - Join hit-rate of 1
  - Cardinalities: 15,625 - 64,000,000
- Hardware event counters
  - to analyze cache & TLB misses

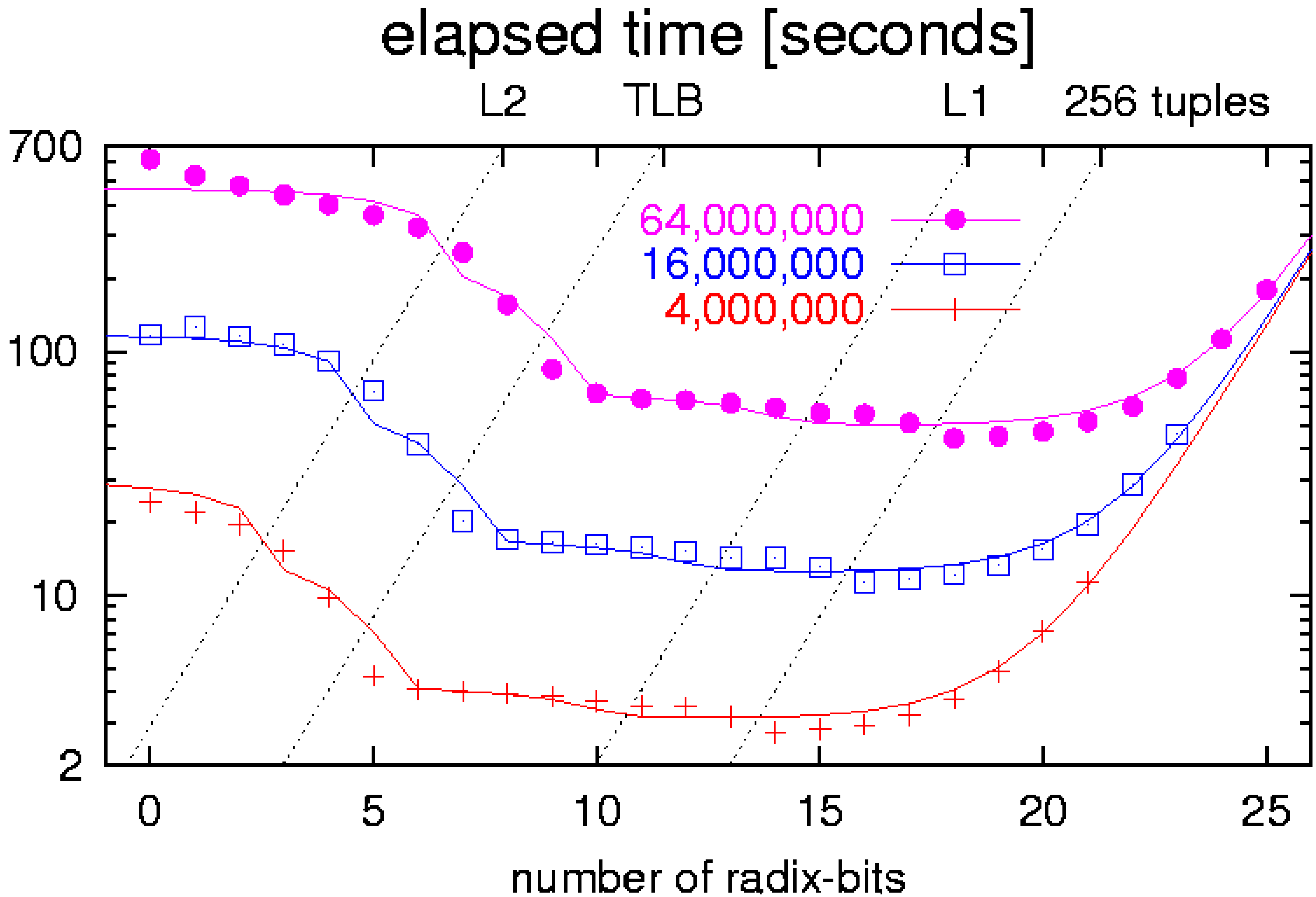
MonetDB Experiments: Radix-Cluster (64,000,000 tuples)



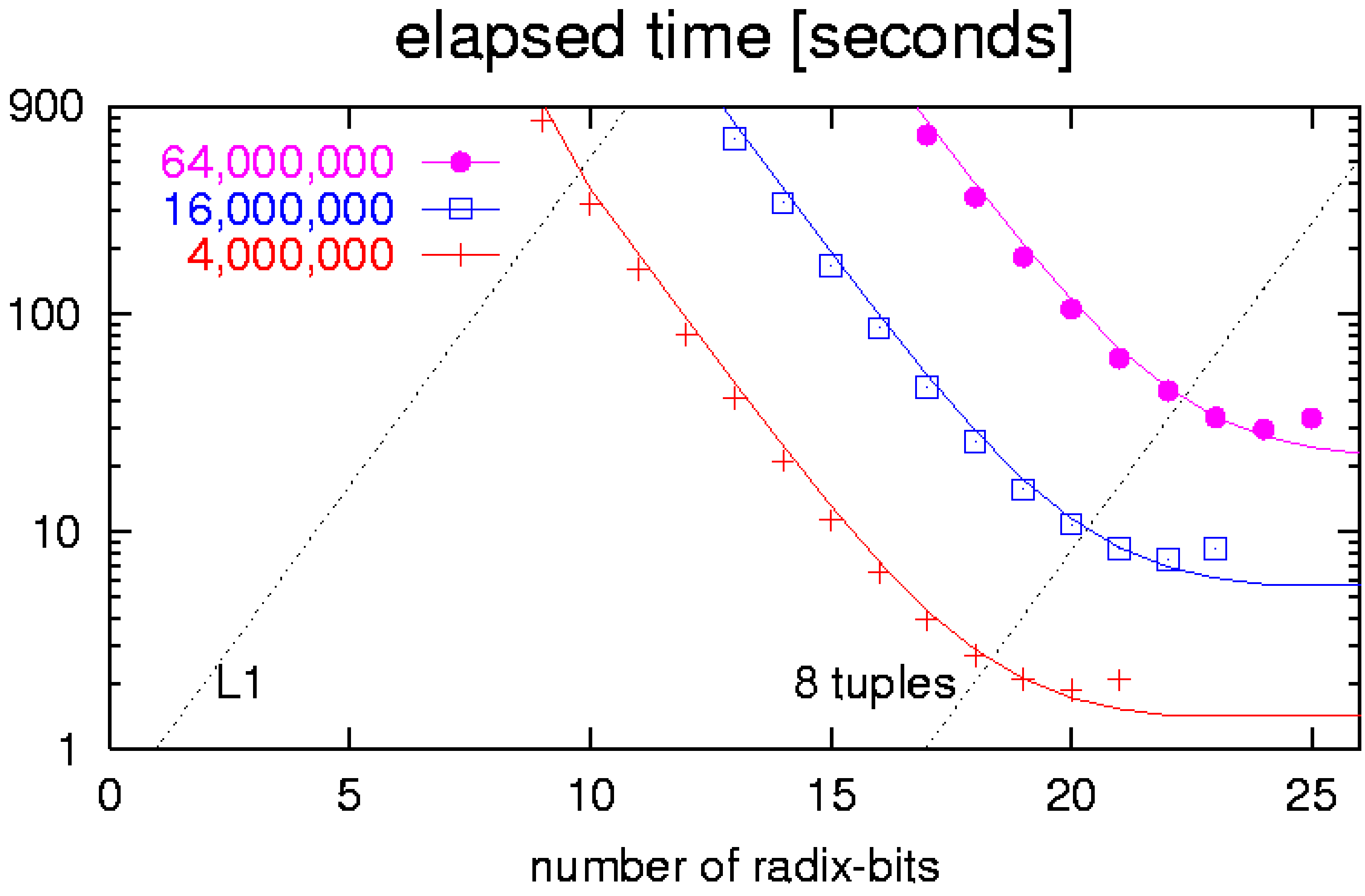
Accurate Cost Modeling: Radix-Cluster



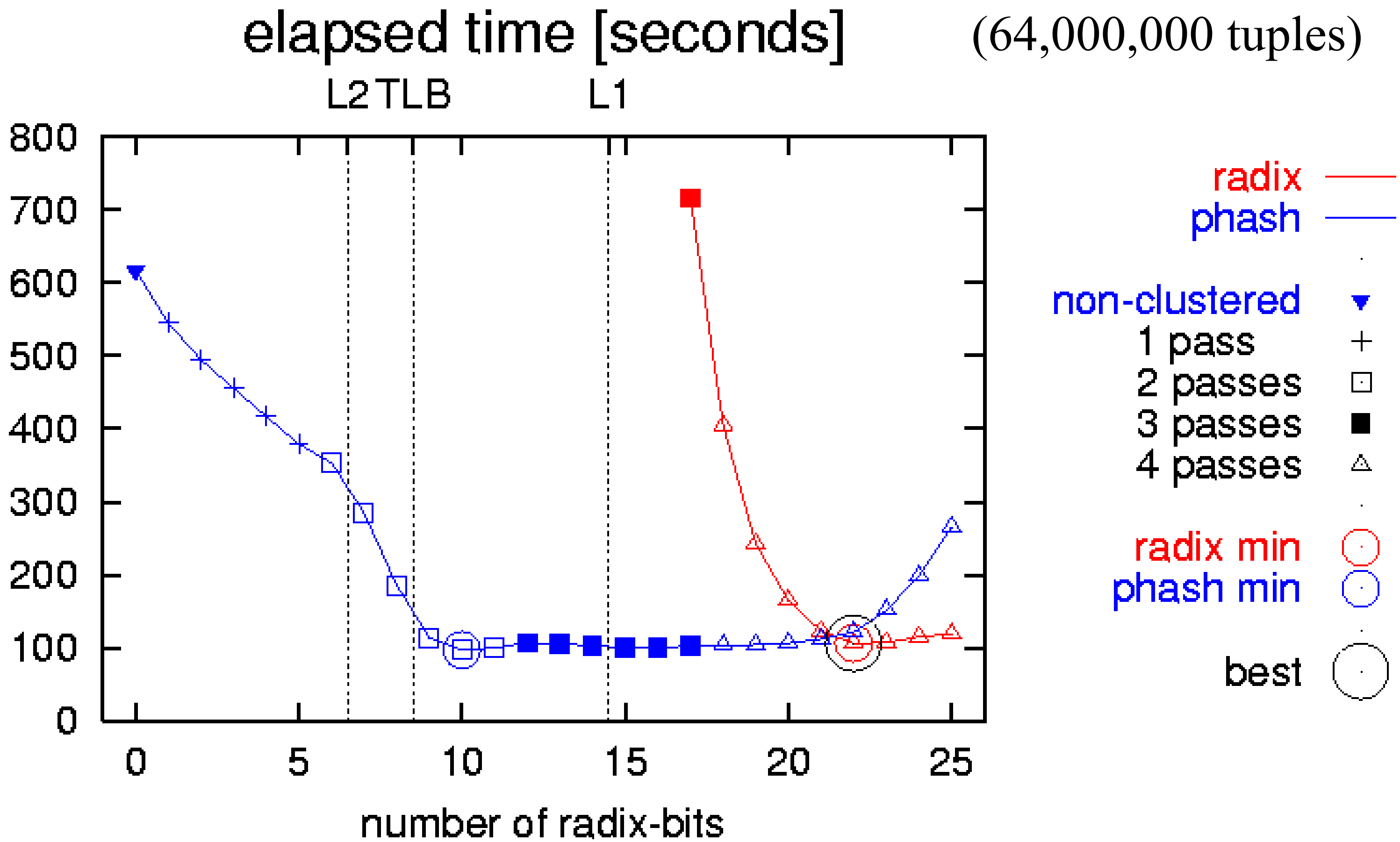
MonetDB Experiments: Partitioned Hash-Join



MonetDB Experiments: Radix-Join



MonetDB Experiments: Overall Performance



Reference Material

- http://www.monetdb.com
- EDBT'96: GIS extensions, SEQUOIA
- ICDE'98: MOA object-oriented query mapping, TPC-D
- VLDB'98: Data Mining Benchmark vs Oracle
- VLDB journal'99: MIL language definition
- VLDB'99: cache-optimized join
- VLDB'00: super-scalar CPU join