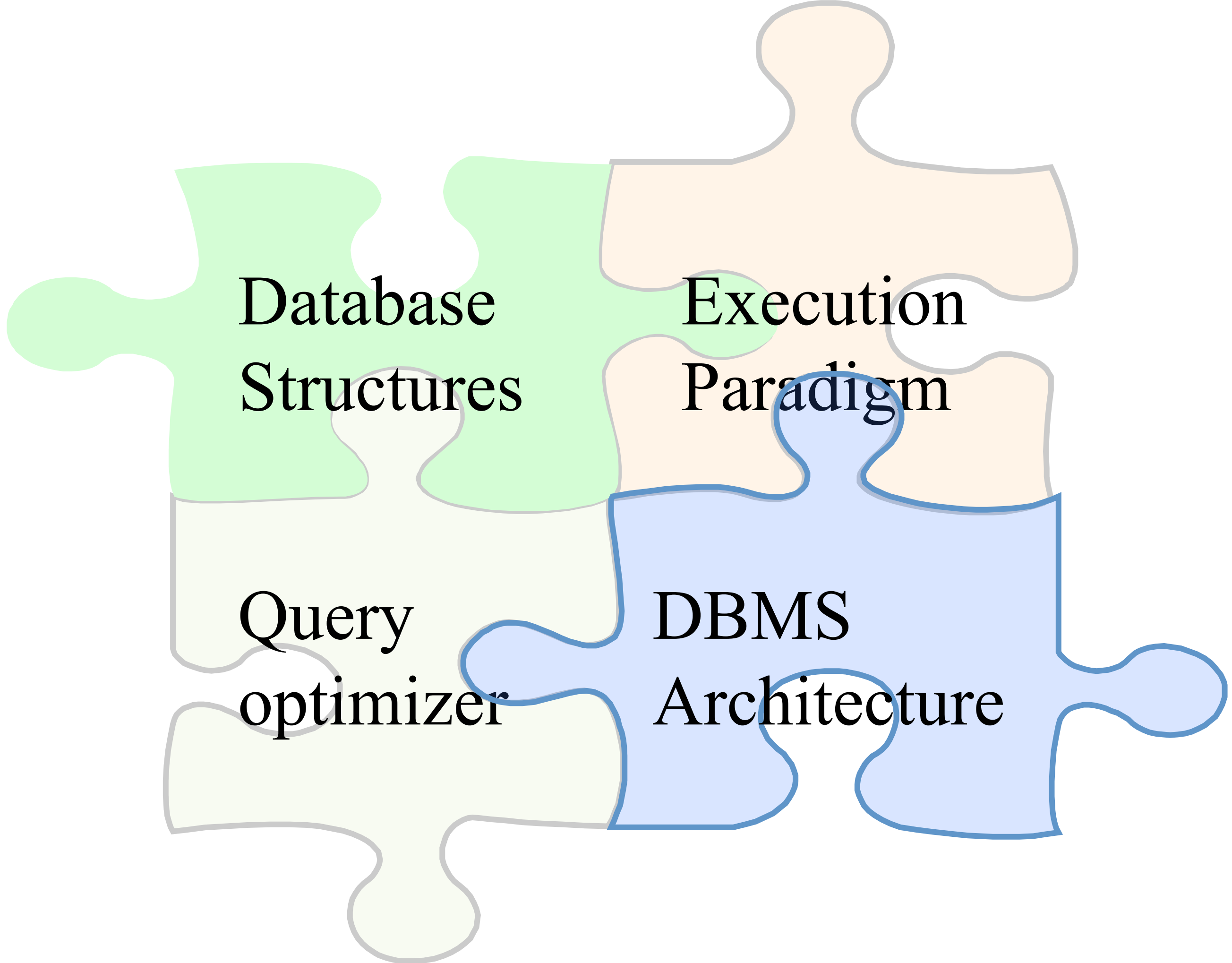
The MonetDB Architecture

Martin Kersten

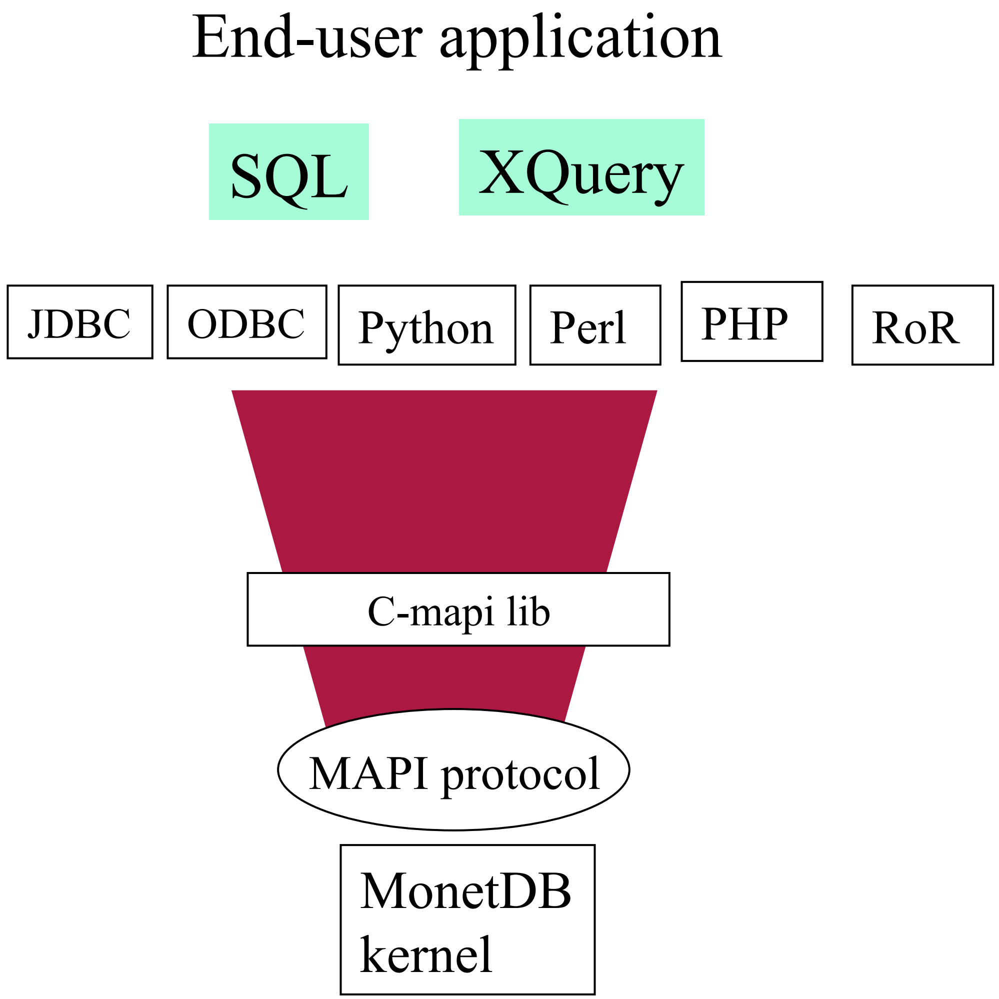
CWI

Amsterdam

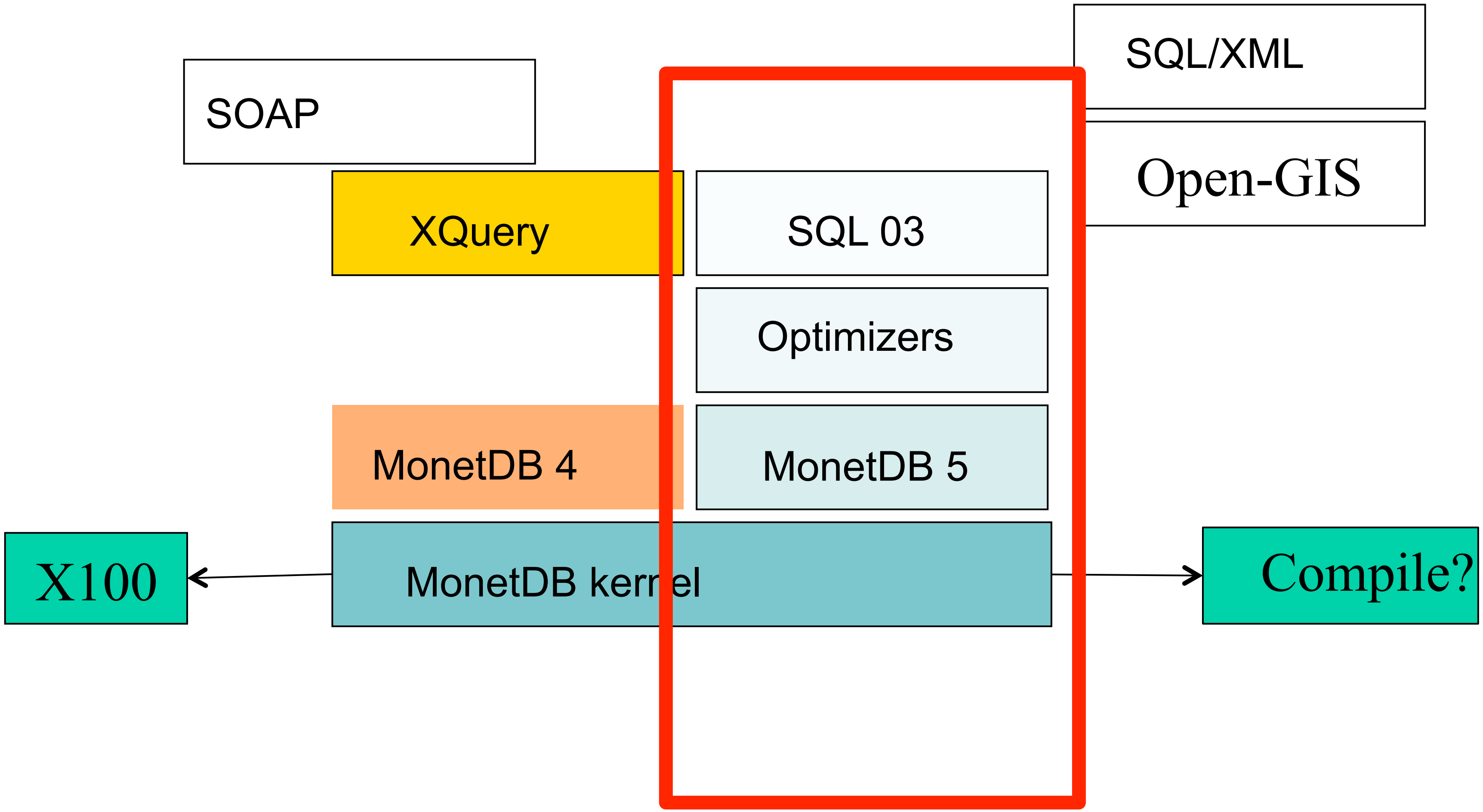
Try to keep things simple



MonetDB quickstep

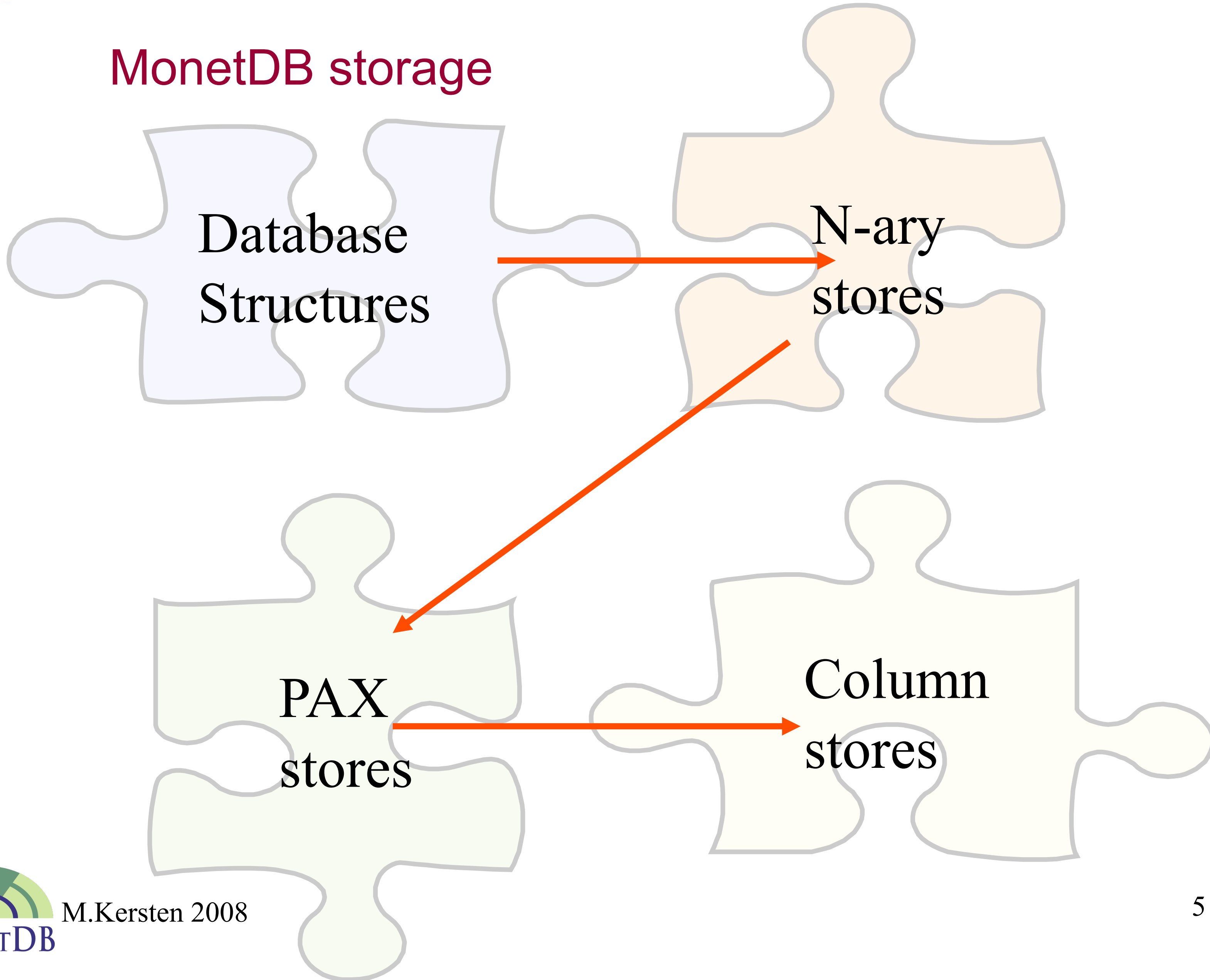


The MonetDB Software Stack

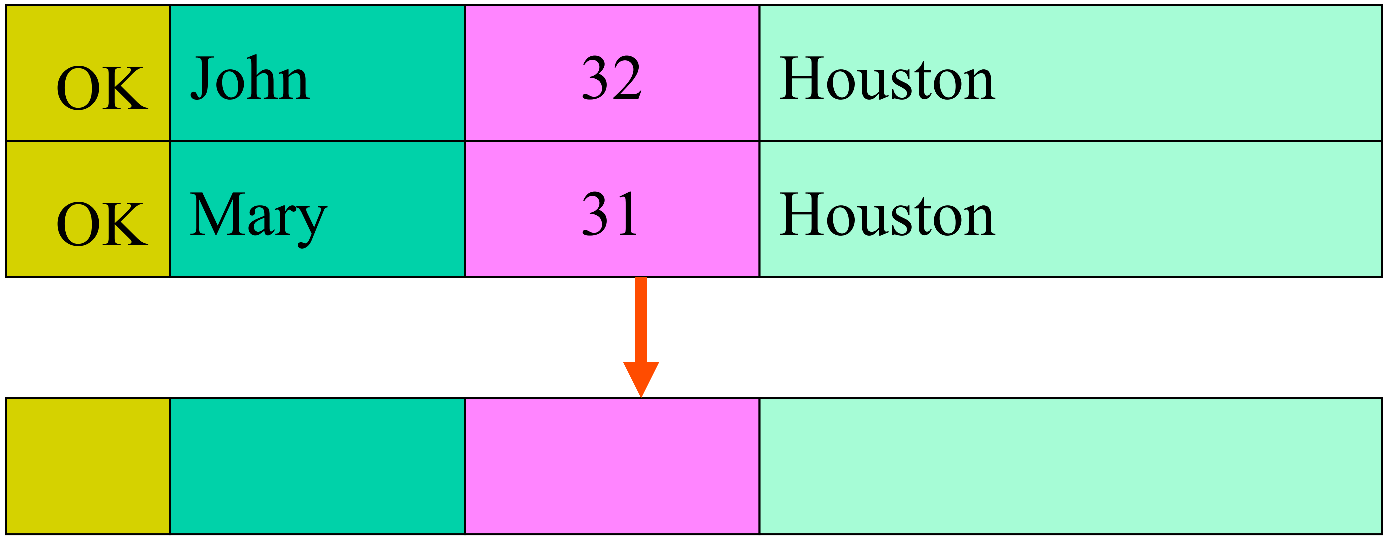


An advanced column-oriented DBMS

MonetDB storage

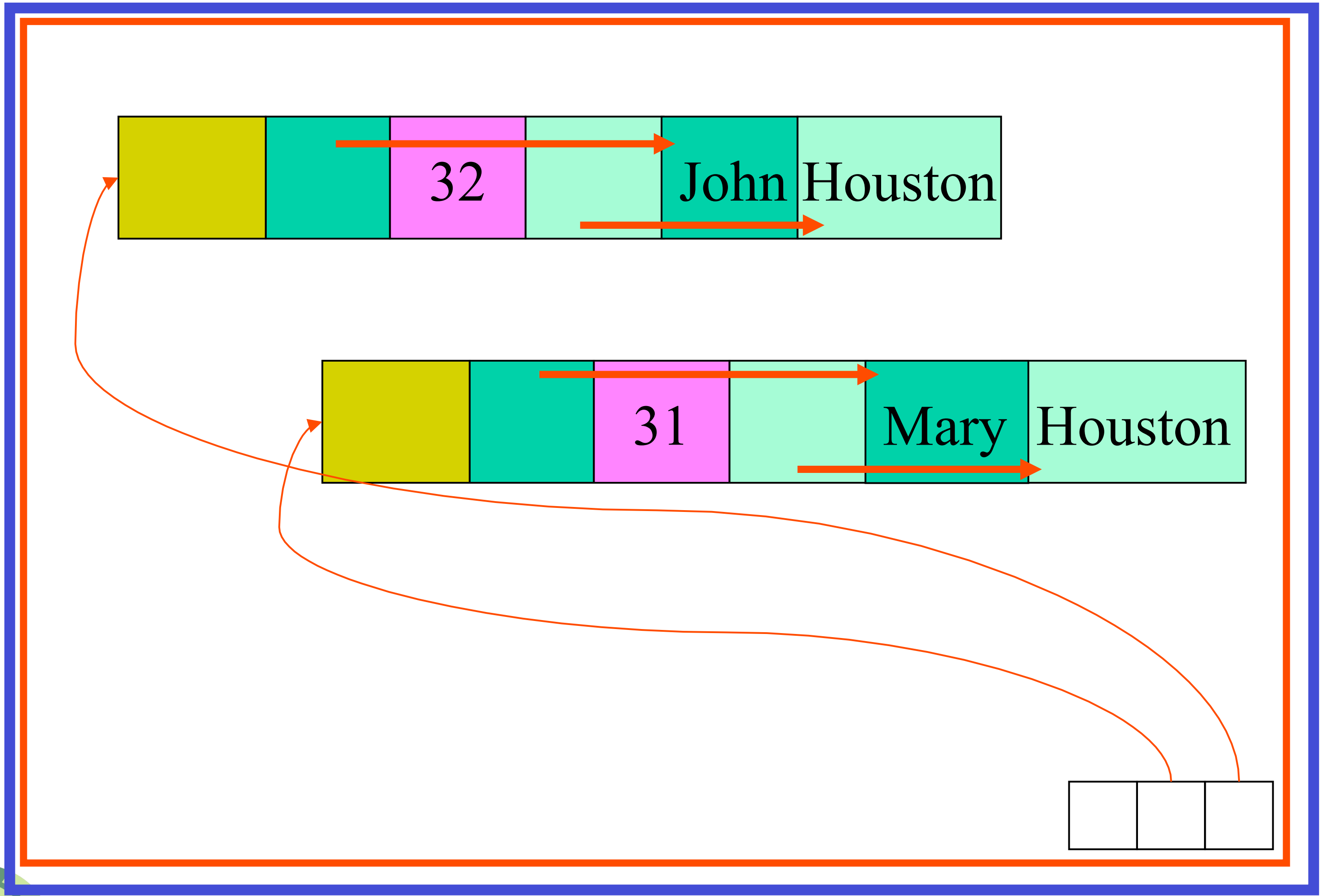


Early 80s: tuple storage structures for PCs were simple

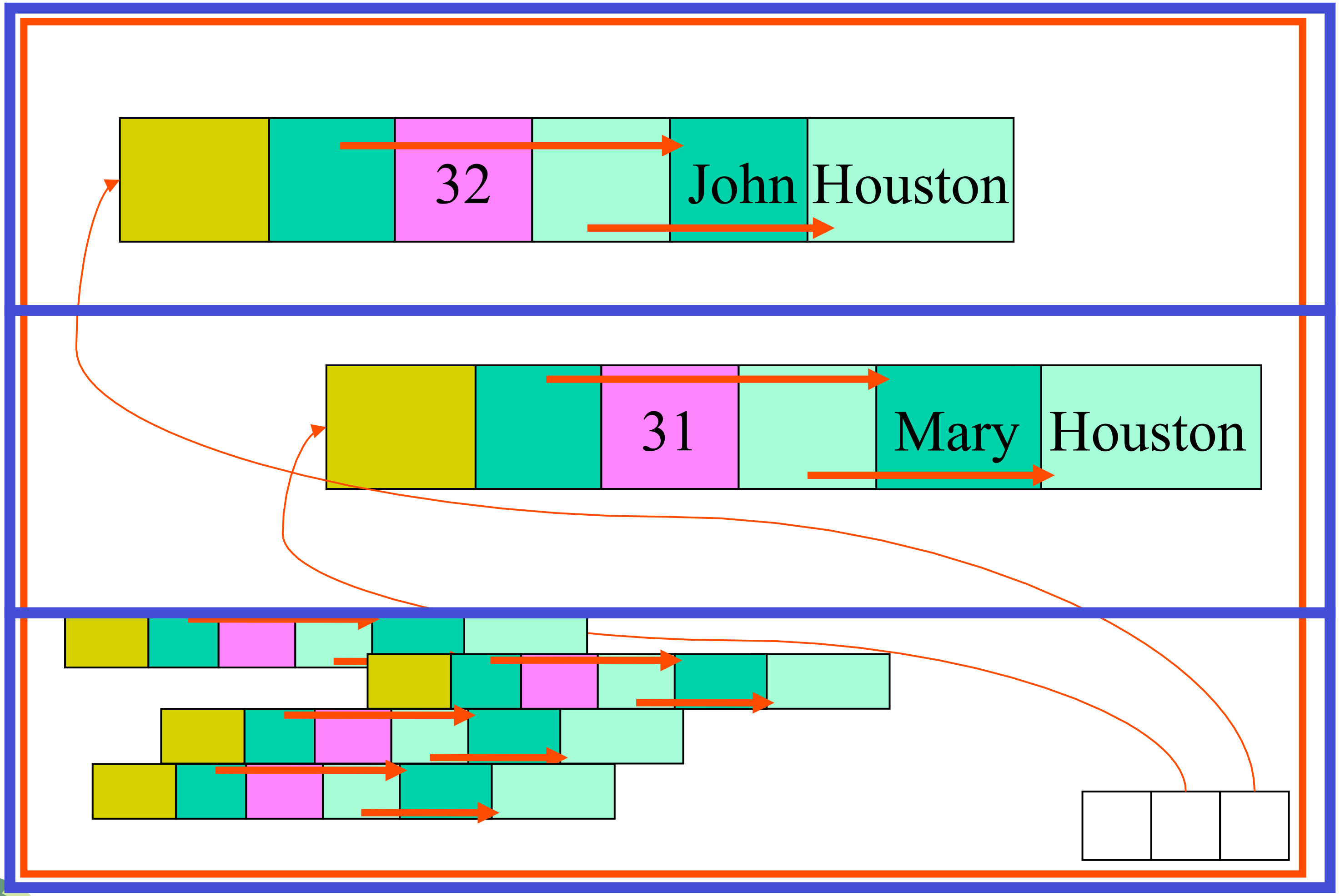


Easy to access at the cost of wasted space

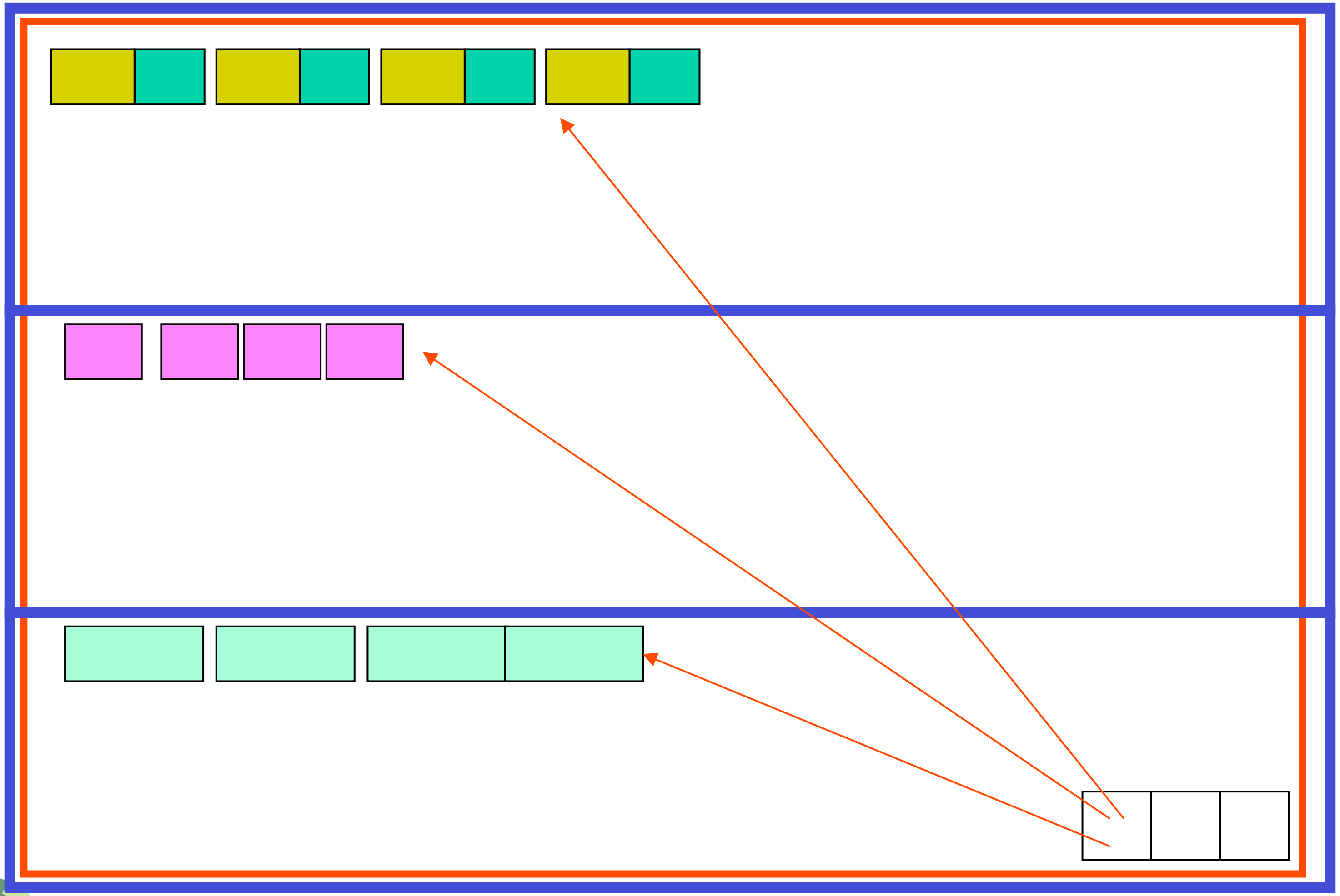
Slotted pages Logical pages equated physical pages



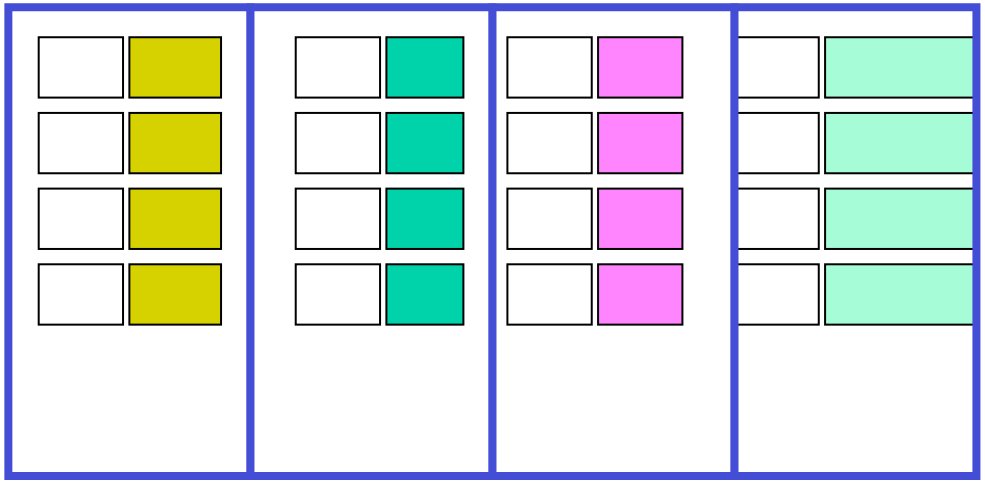
Slotted pages Logical pages equated multiple physical pages



Not all attributes are equally important



A column orientation is as simple and acts like an array



Attributes of a tuple are correlated by offset

Try to keep things simple

- MonetDB Binary Association Tables

ID	Day	Discount
10	4/4/98	0.195
11	9/4/98	0.065
12	1/2/98	0.175
13	7/2/98	0

OID	ID
100	10
101	11
102	12
103	13
104	14

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98
104	1/2/99

OID	Discount
100	0.195
101	0.065
102	0.175
103	0
104	0.065

Physical data organization

- Binary Association Tables

head	*tail*
100	10
101	11
102	12
103	13
104	14

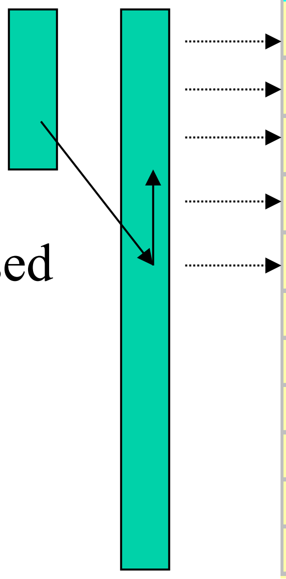
Dense sequence

Memory mapped files

Bat Unit fixed size

- Binary Association Tables accelerators

Hash-based access



OID	ID
100	10
101	11
102	12
103	13
104	14

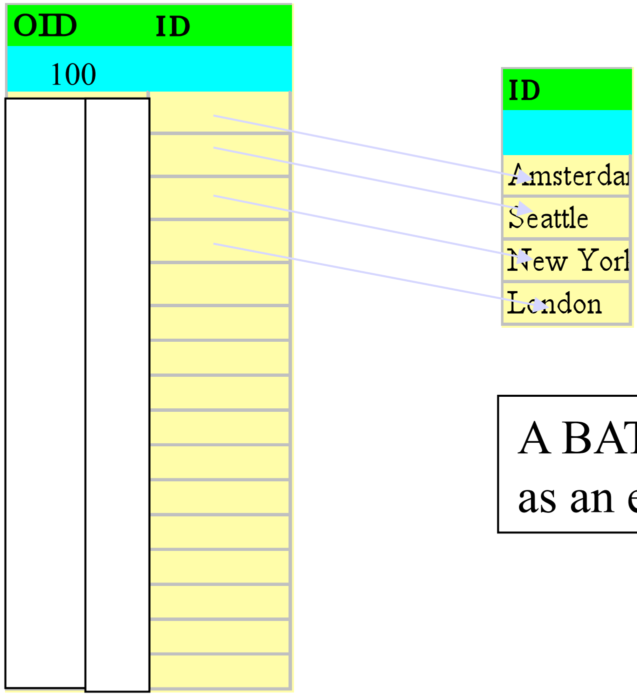
Column properties:
- key-ness
- non-null
- dense
- ordered

Try to avoid doing things twice

- Binary Association Tables storage control

Type remappings are used to squeeze space

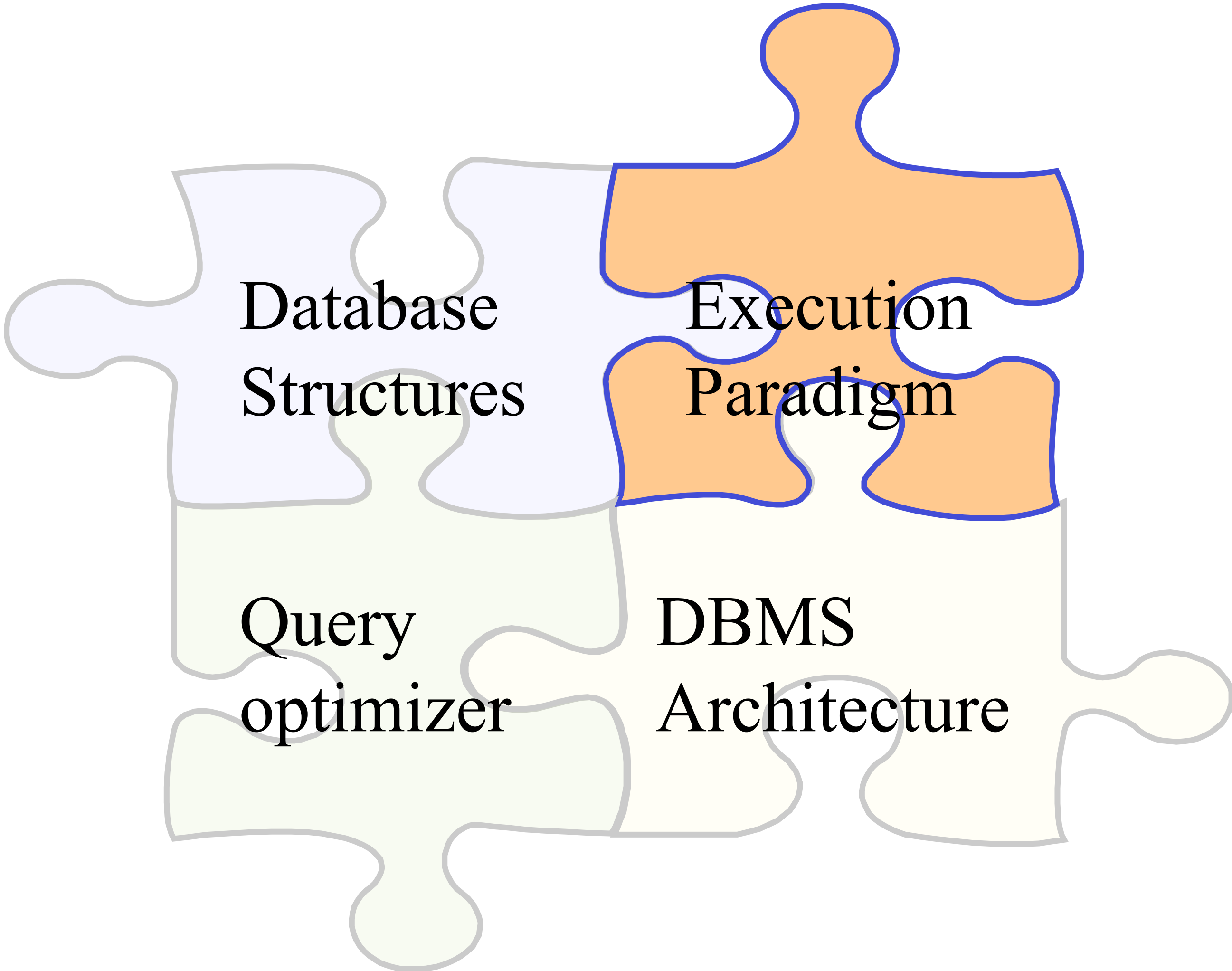
A VID datatype can be used to represent dense enumerations



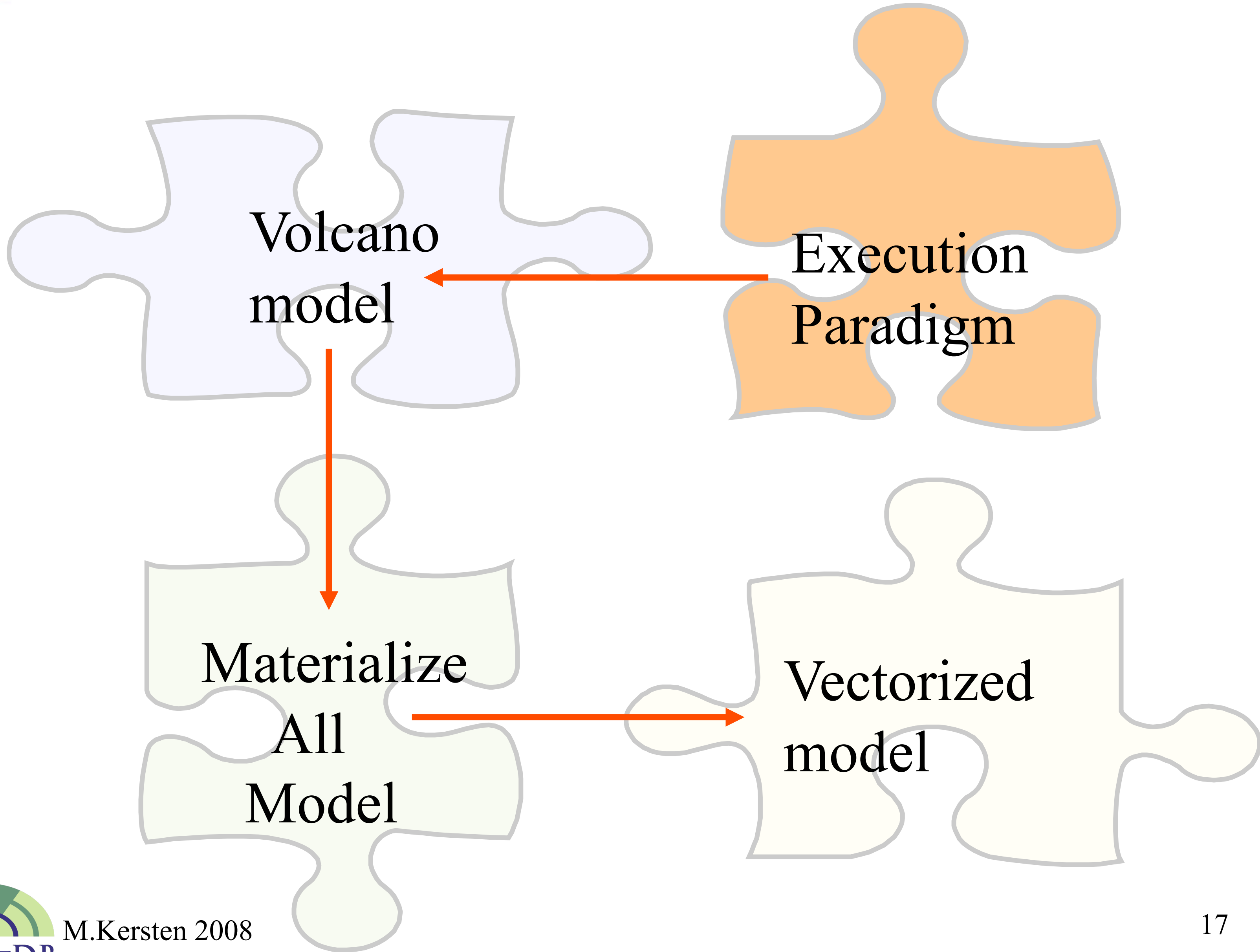
A BAT can be used as an encoding table

Mantra: Try to keep things simple

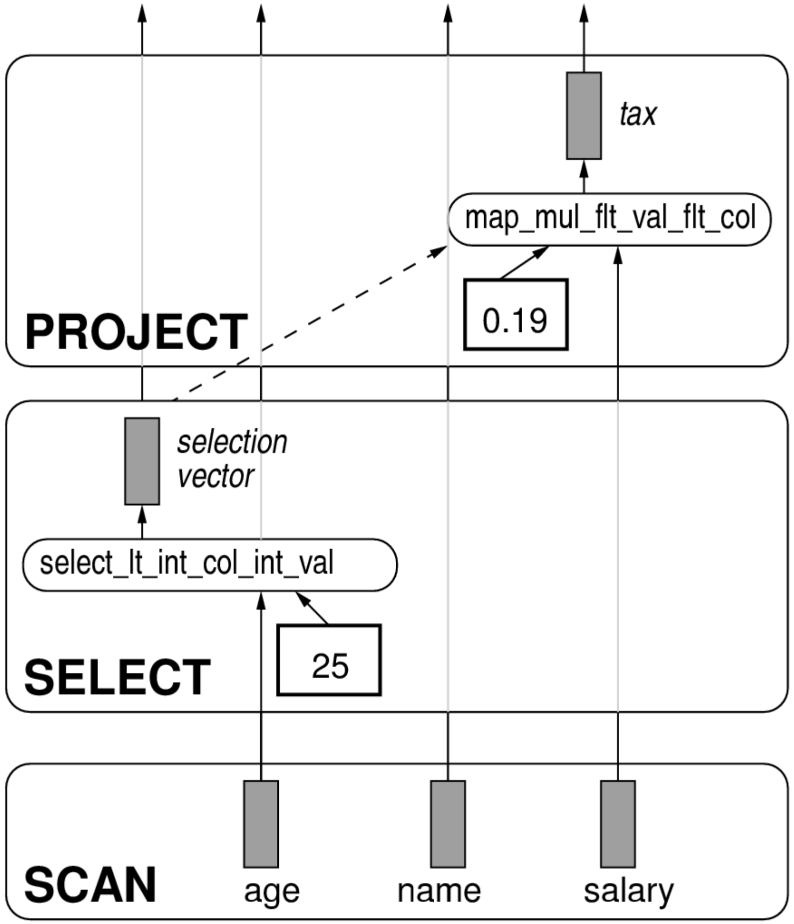
- Column orientation benefits datawarehousing
- Brings a much tighter packaging and improves transport through the memory hierarchy
- Each column can be more easily optimized for storage using compression schemes
- Each column can be replicated for read-only access


Database
Structures
Execution
Paradigm
Query
optimizer
DBMS
Architecture

Try to maximize performance



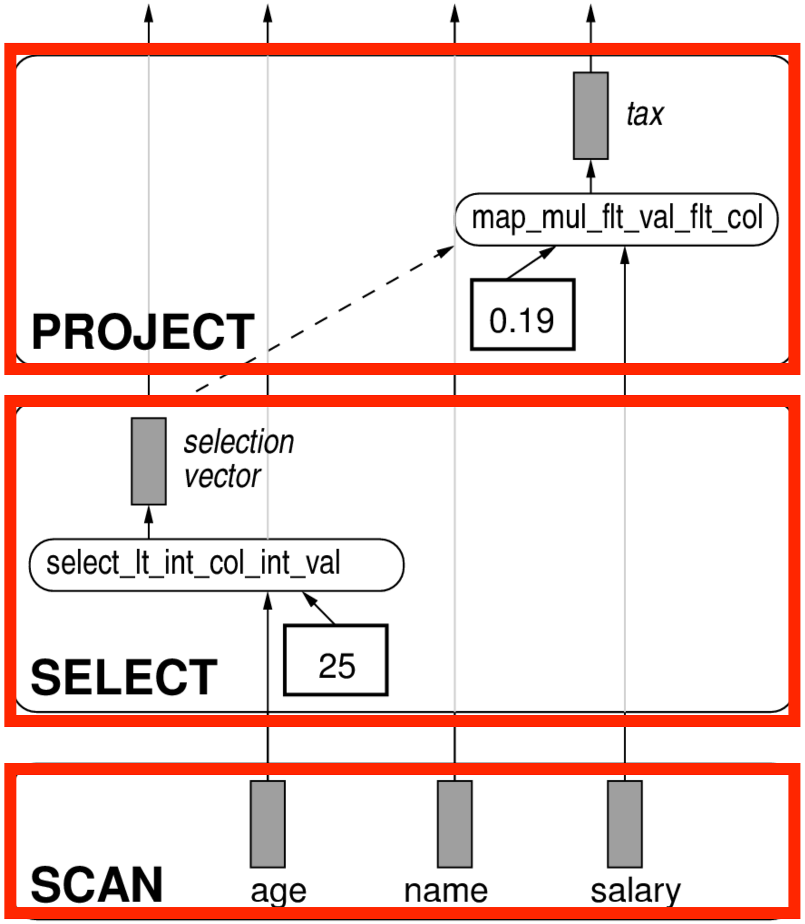
Volcano Engines



Query

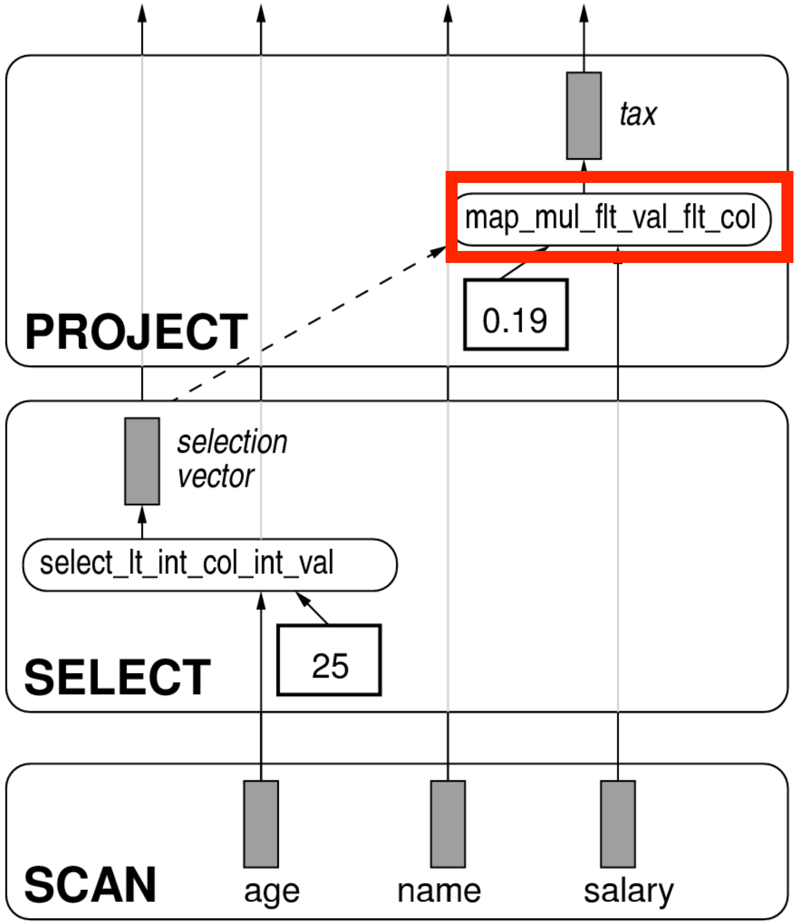
SELECT
name,
salary*.19 AS tax
FROM
employee
WHERE
age > 25

Volcano Engines



Operators

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



Primitives

Provide computational functionality

All arithmetic allowed in expressions,
e.g. multiplication

`mult(int,int)` ➔ `int`

Volcano paradigm

- The Volcano model is based on a simple pull-based iterator model for programming relational operators.
- The Volcano model minimizes the amount of intermediate store
- The Volcano model is CPU intensive and can be inefficient

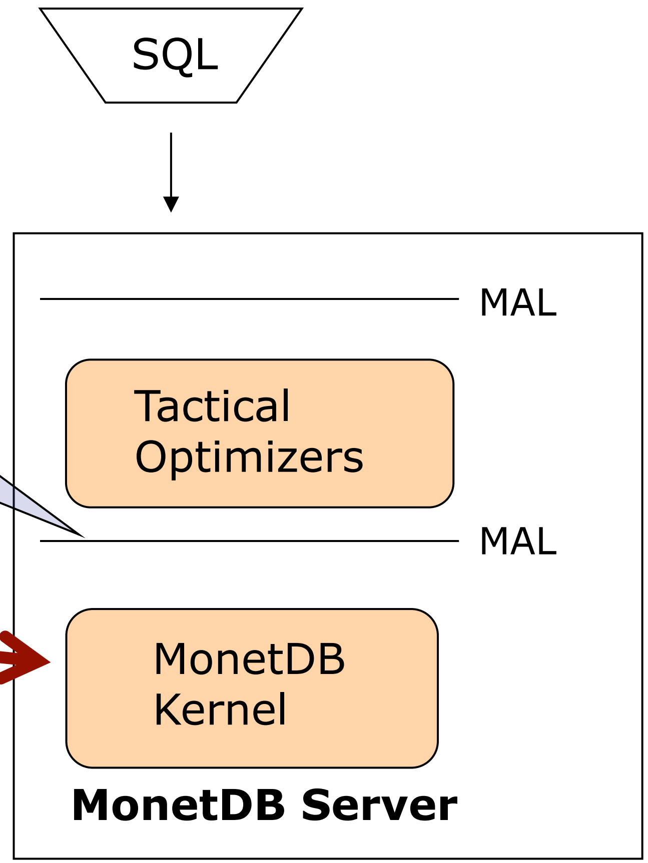
MonetDB paradigm

- The MonetDB kernel is a programmable relational algebra machine
- Relational operators operate on ‘array’-like structures

MonetDB quickstep

select count(*) from photoobjall;

```
function user.s3_1():void;
   X1:bat[:oid,:lng] := sql.bind("sys","photoobjall","objid",0);
   X20 := aggr.count(X1);
 sql.exportValue(1,"sys.","count_","int",32,0,6,X20,"");
end s3_1;
```



Kernel execution paradigms

- Tuple-at-a-time pipelined
- Operator-at-a-time

Operator implementation

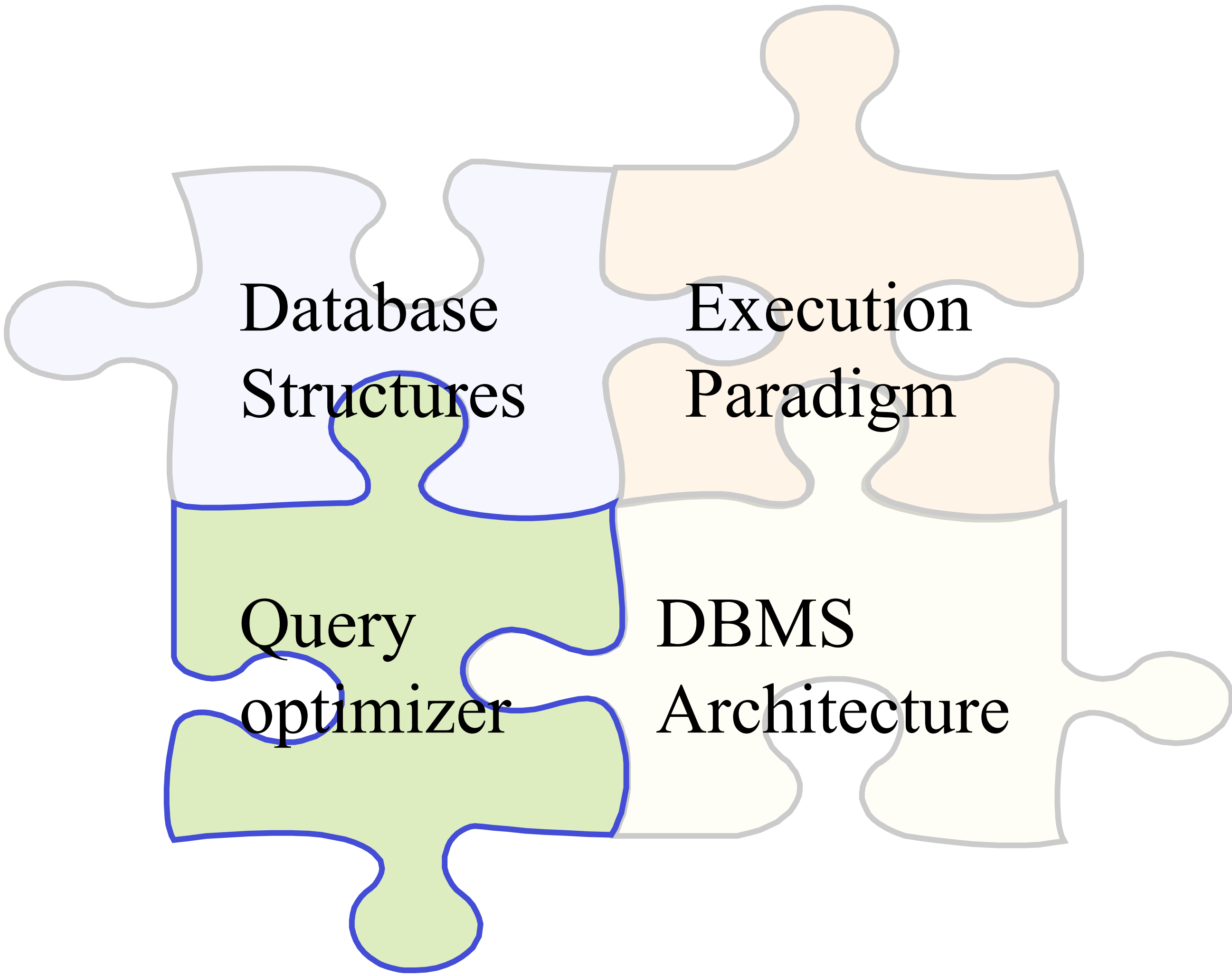
- All algebraic operators materialize their result
  - GOOD: small code footprints
  - GOOD: potential for re-use
  - BAD : extra storage for intermediates
  - BAD: cpu cost for retaining it

- Local optimization decisions
  - Sortedness, uniqueness, hash index
  - Sampling to determine sizes
  - Parallelism options
  - Properties that affect the algorithms

Operator implementation

- All algebraic operators materialize their result
- Local optimization decisions
- Heavy use of code expansion to reduce cost
  - 55 selection routines
  - 149 unary operations
  - 335 join/group operations
  - 134 multi-join operations
  - 72 aggregate operations

Try to avoid the search space trap



MonetDB quickstep

Strategic optimizer:

– Exploit the lanuage the language

– Rely on heuristics

Tactical MAL optimizer:

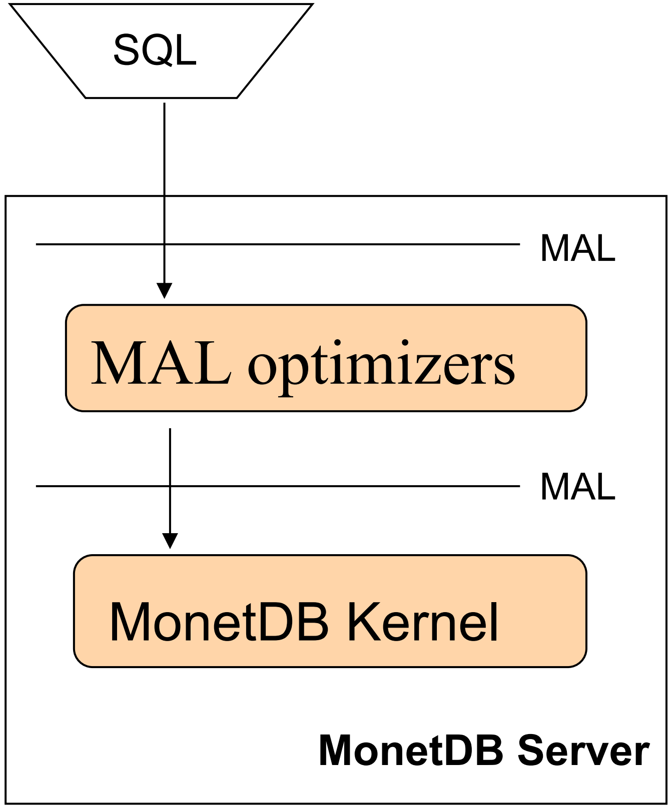
–Modular optimizer framework

–Focused on coarse grain resource optimization

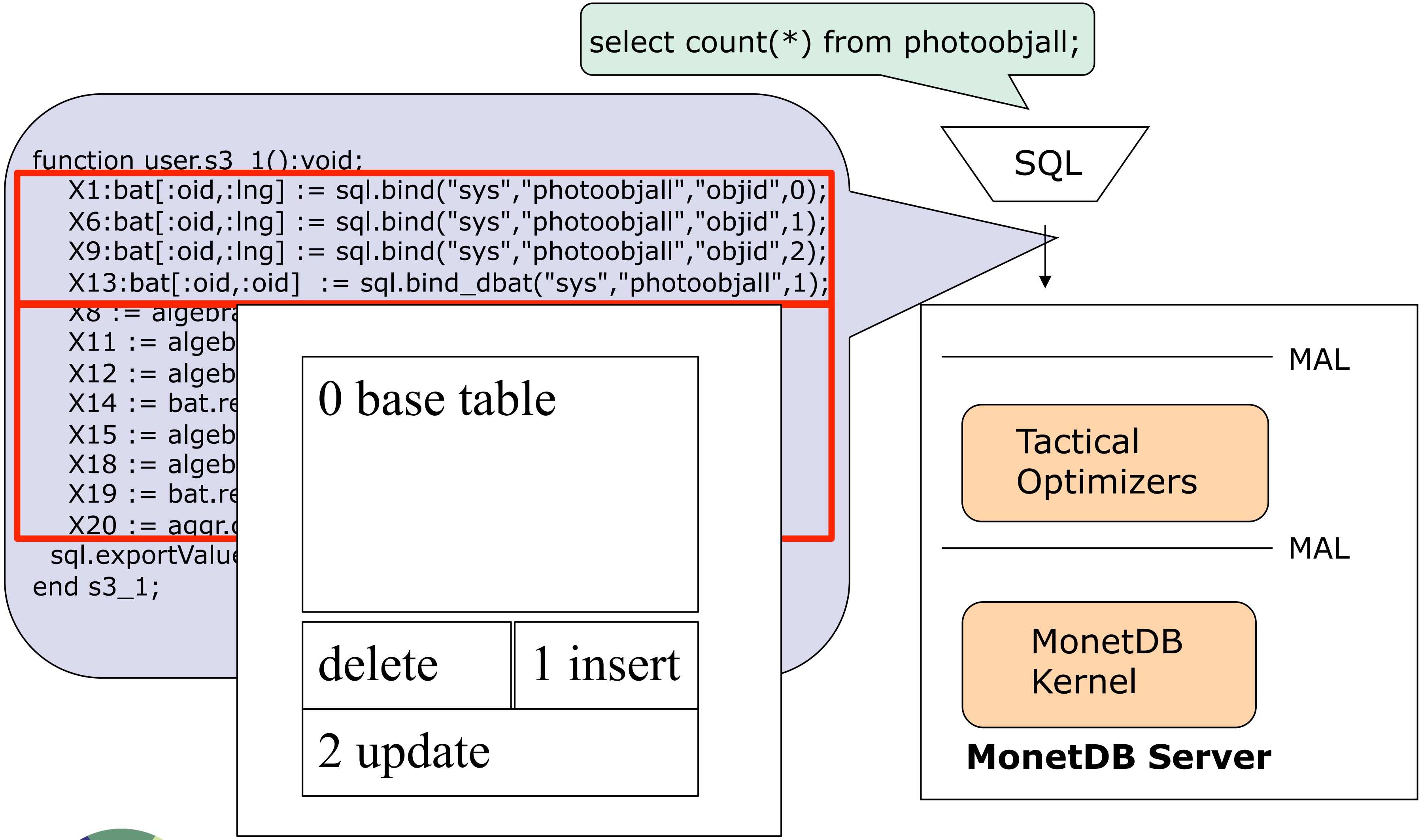
Operational optimizer:

– Exploit everything you know at runtime

– Re-organize if necessary



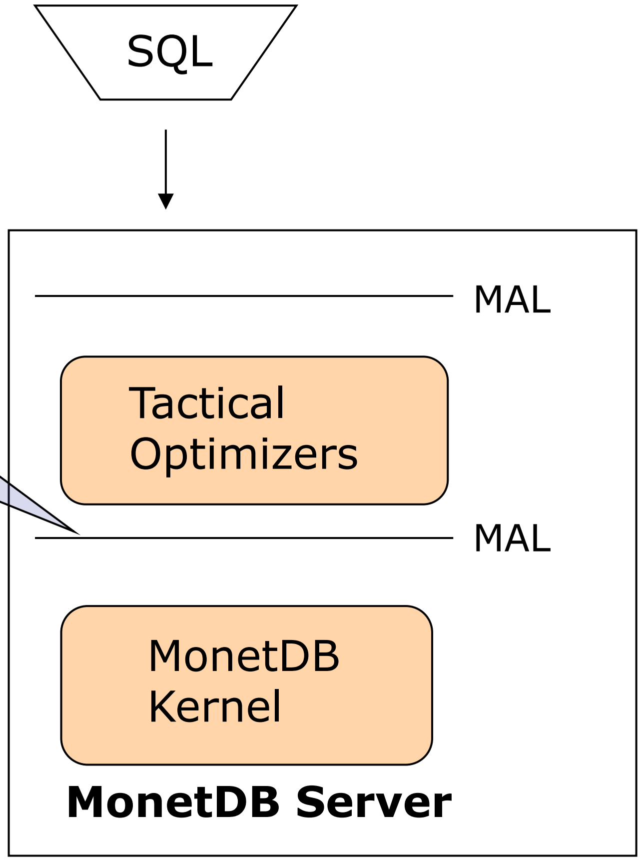
MonetDB quickstep



MonetDB quickstep

select count(*) from photoobjall;

```
function user.s3_1():void;
    X1:bat[:oid,:lng] := sql.bind("sys","photoobjall","objid",0);
    X20 := aggr.count(X1);
  sql.exportValue(1,"sys.","count_","int",32,0,6,X20,"");
end s3_1;
```



Optimizer pipelines.

sql> select optimizer;

inline,remap,evaluate,costModel,coercions ,emptySet,aliases,mergetable,deadcode,c onstants,commonTerms,joinPath,deadcode ,reduce,garbageCollector,dataflow,history,r eplication,multiplex

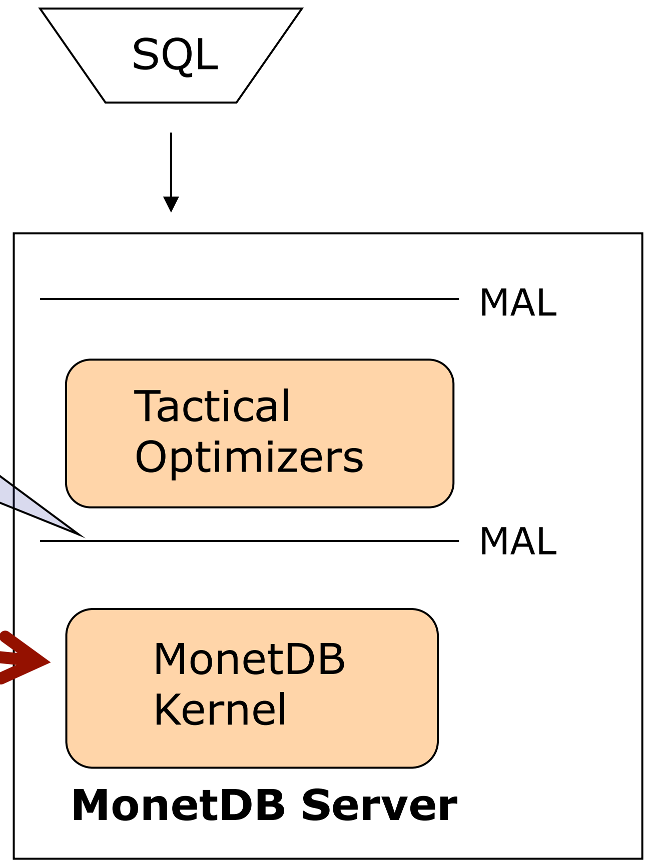
MonetDB quickstep

```
select count(*) from photoobjall;
```

```
function user.s3_1():void;
   X1:bat[:oid,:lng] := sql.bind("sys","photoobjall","objid",0);
   X20 := aggr.count(X1);
  sql.exportValue(1,"sys.","count_","int",32,0,6,X20,"");
end s3_1;
```

Kernel execution paradigms

- Tuple-at-a-time pipelined
- Operator-at-a-time



Query optimization

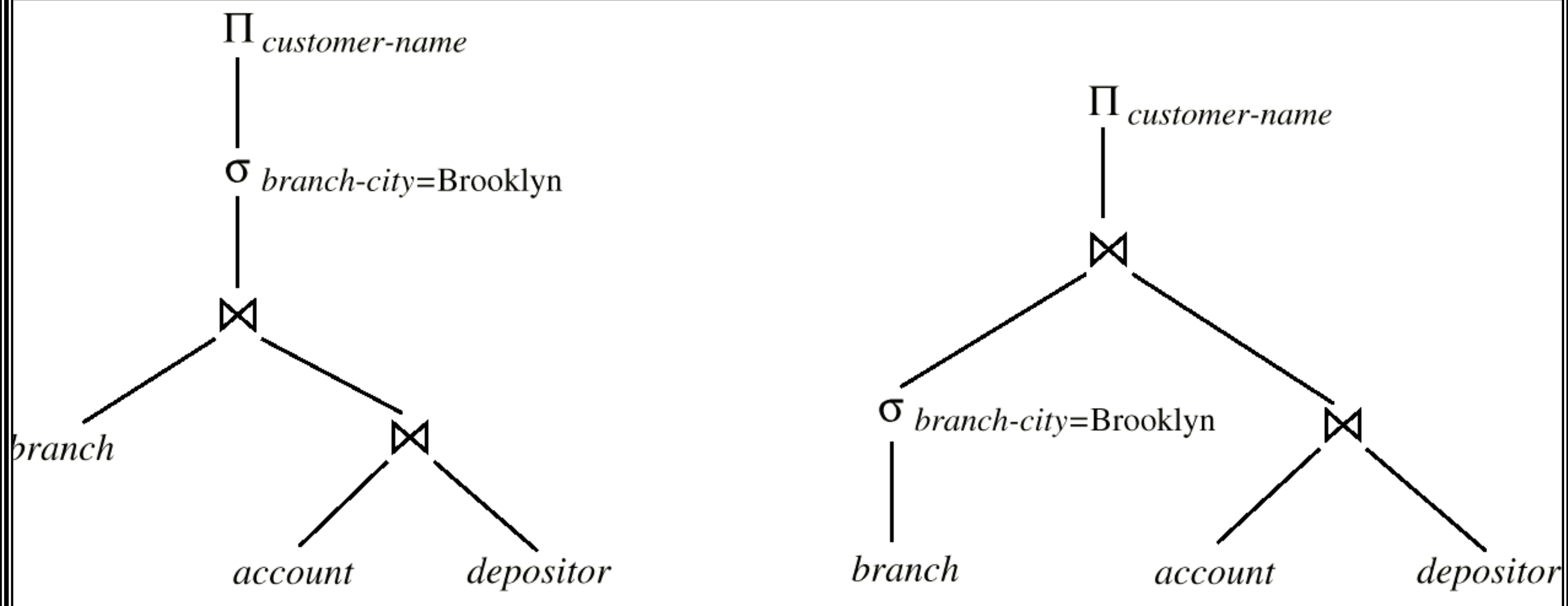
- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation (Chapter 13)

- Cost difference between a good and a bad way of evaluating a query can be enormous
  - Example: performing a *r* X *s* followed by a selection *r.A = s.B* is much slower than performing a join on the same condition

- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
  - Need to estimate statistics for intermediate results to compute cost of complex expressions

Introduction (Cont.)

Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently.



(a) Initial Expression Tree (b) Transformed Expression Tree

Introduction (Cont.)

- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions
     - Use **equivalence rules** to transform an expression into an equivalent one.
  2. Annotating resultant expressions to get alternative query plans
  3. Choosing the cheapest plan based on **estimated cost**
- The overall process is called **cost based optimization.**

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
2. 2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted. $\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$
4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

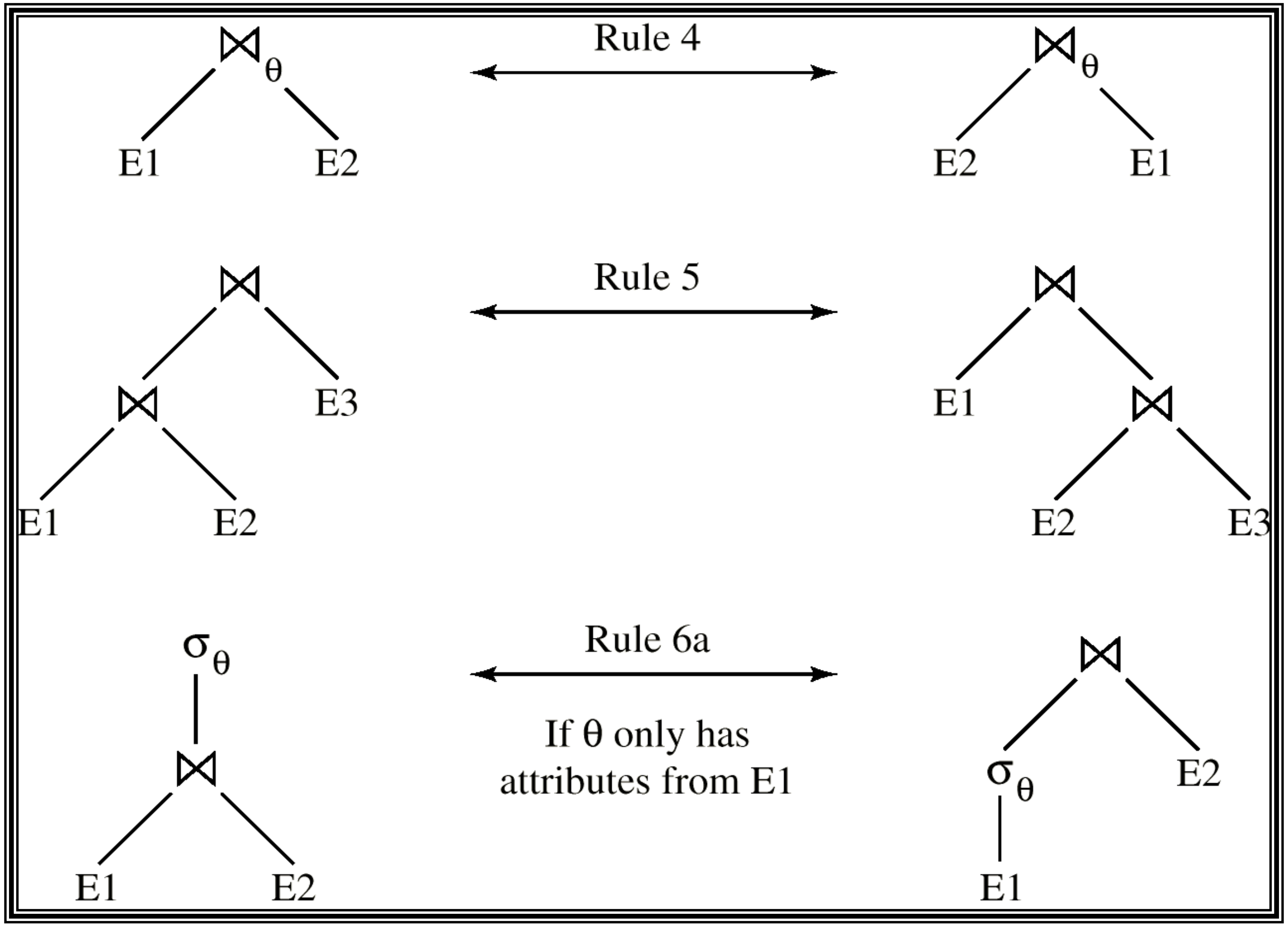
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 2 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

   $$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

   $$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

MonetDB

Equivalence Rules (Cont.)

8. The projections operation distributes over the theta join operation as follows:

   (a) if it involves only attributes from $L_1 \cup L_2$:

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\prod_{L_1}(E_1)) \bowtie_\theta (\prod_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \prod_{L_1 \cup L_2}((\prod_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\prod_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

- (set difference is not commutative).

11. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over ∪, ∩ and –.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for ∪ and ∩ in place of –

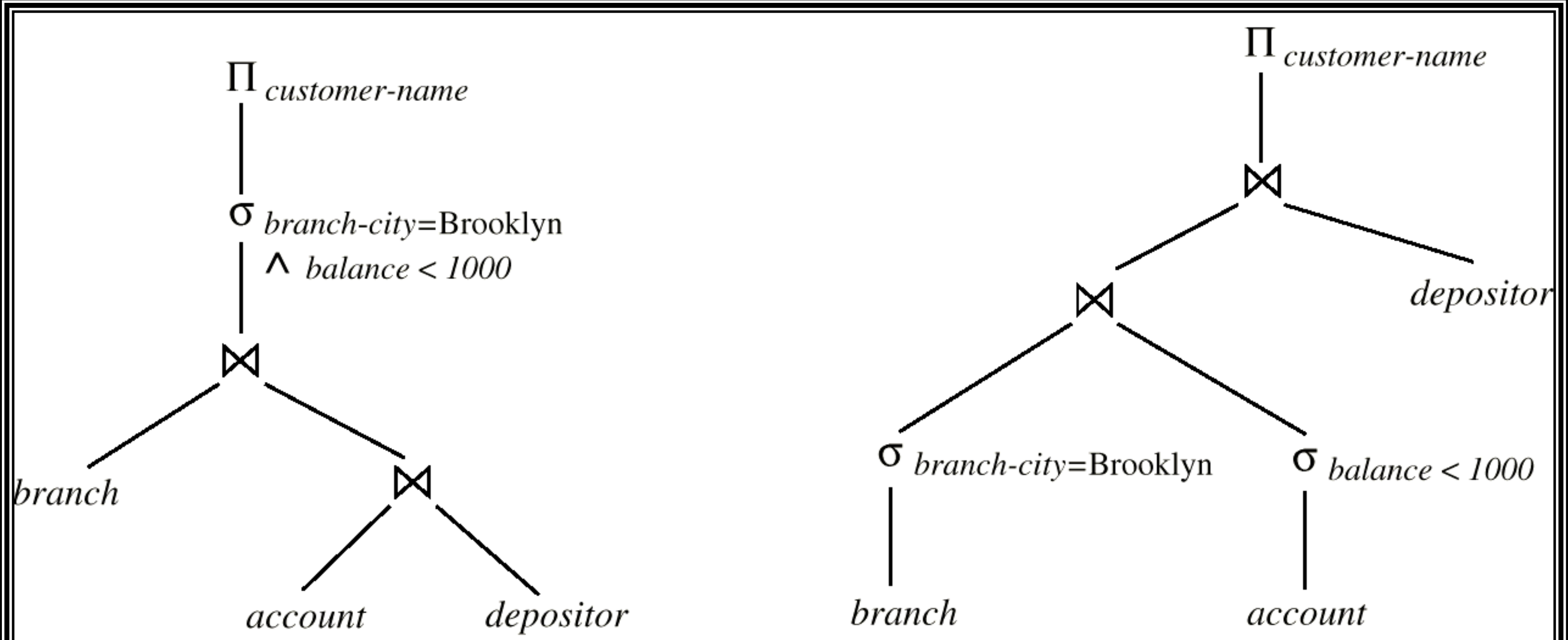
Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for ∩ in place of –, but not for ∪

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Multiple Transformations (Cont.)



(a) Initial Expression Tree

(b) Tree After Multiple Transformations

Optimizer strategies

- Heuristic
  - Apply the transformation rules in a specific order such that the cost converges to a minimum

- Cost based
  - Simulated annealing
  - Randomized generation of candidate QEP
  - Problem, how to guarantee randomness

Memoization Techniques

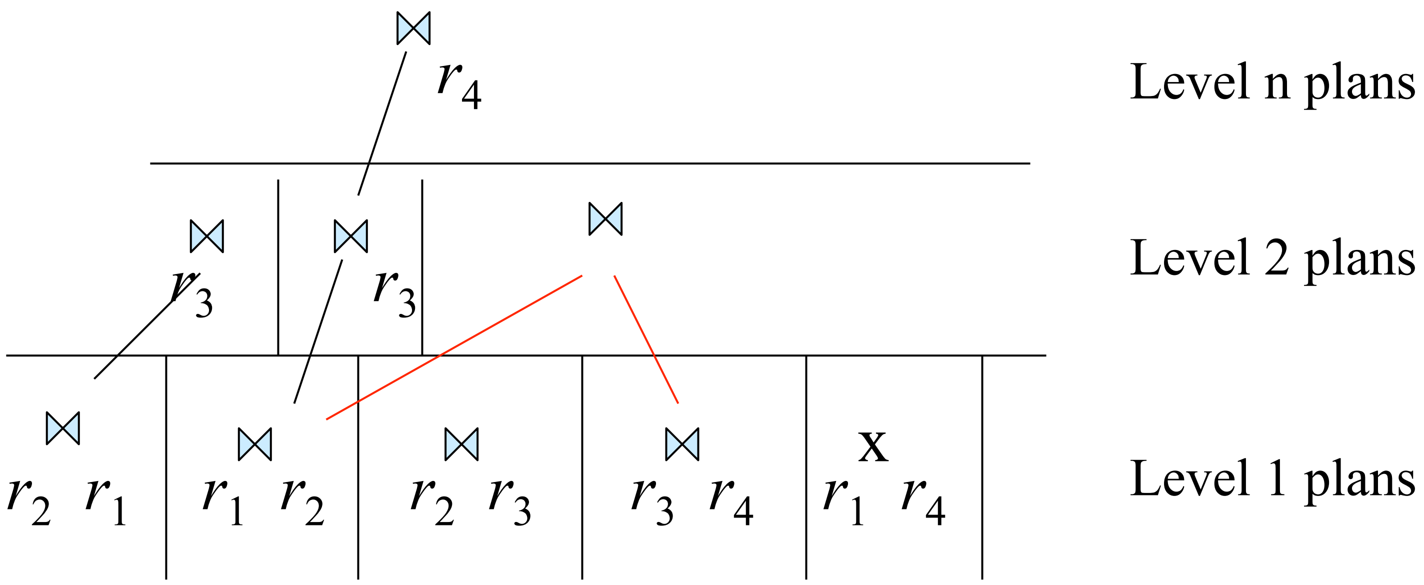
- *How to generate alternative Query Evaluation Plans?*
  - Early generation systems centred around a tree representation of the plan
  - Hardwired tree rewriting rules are deployed to enumerate part of the space of possible QEP
  - For each alternative the total cost is determined
  - The best (alternatives) are retained for execution

  - Problems: very large space to explore, duplicate plans, local maxima, expensive query cost evaluation.

  - SQL Server optimizer contains about 300 rules to be deployed.

Memoization Techniques

- *How to generate alternative Query Evaluation Plans?*
  - Keep a memo of partial QEPs and their cost.
  - Use the heuristic rules to generate alternatives to built more complex QEPs
  - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$



Ditching the optimizers

- Applications have different characteristics
- Platforms have different characteristics
- The actual state of computation is crucial

- A generic all-encompassing optimizer cost-model does **not** work

Try to disambiguate decisions

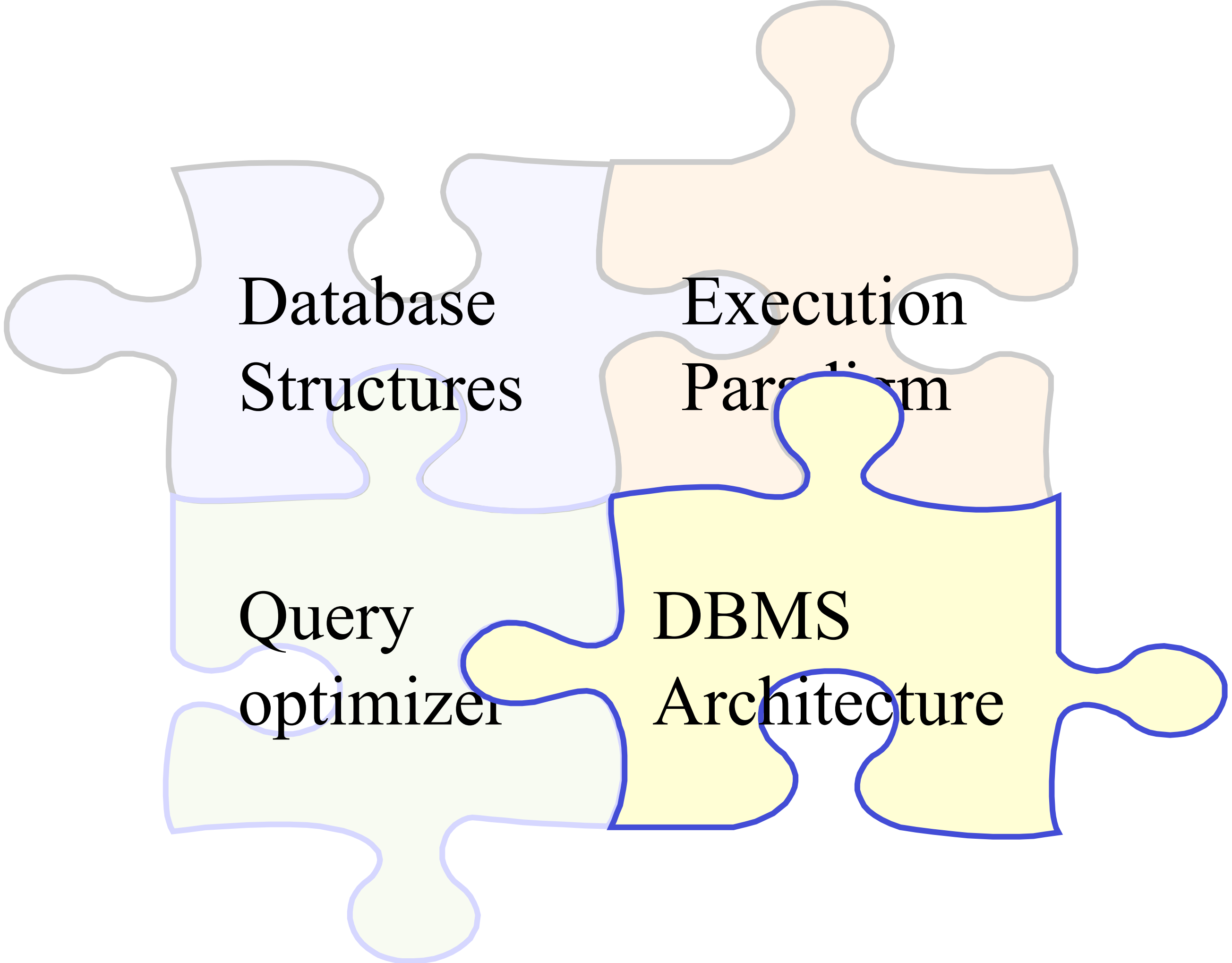
Code Inliner.
Constant Expression Evaluator.
Accumulator Evaluations.
Strength Reduction.
Common Term Optimizer.

Join Path Optimizer.
Ranges Propagation.
Operator Cost Reduction.
Foreign Key handling.
Aggregate Groups.

Code Parallizer.
Replication Manager.
Result Recycler.

MAL Compiler.
Dynamic Query Scheduler.
Memo-based Execution.
Vector Execution.

Alias Removal.
Dead Code Removal.
Garbage Collector.

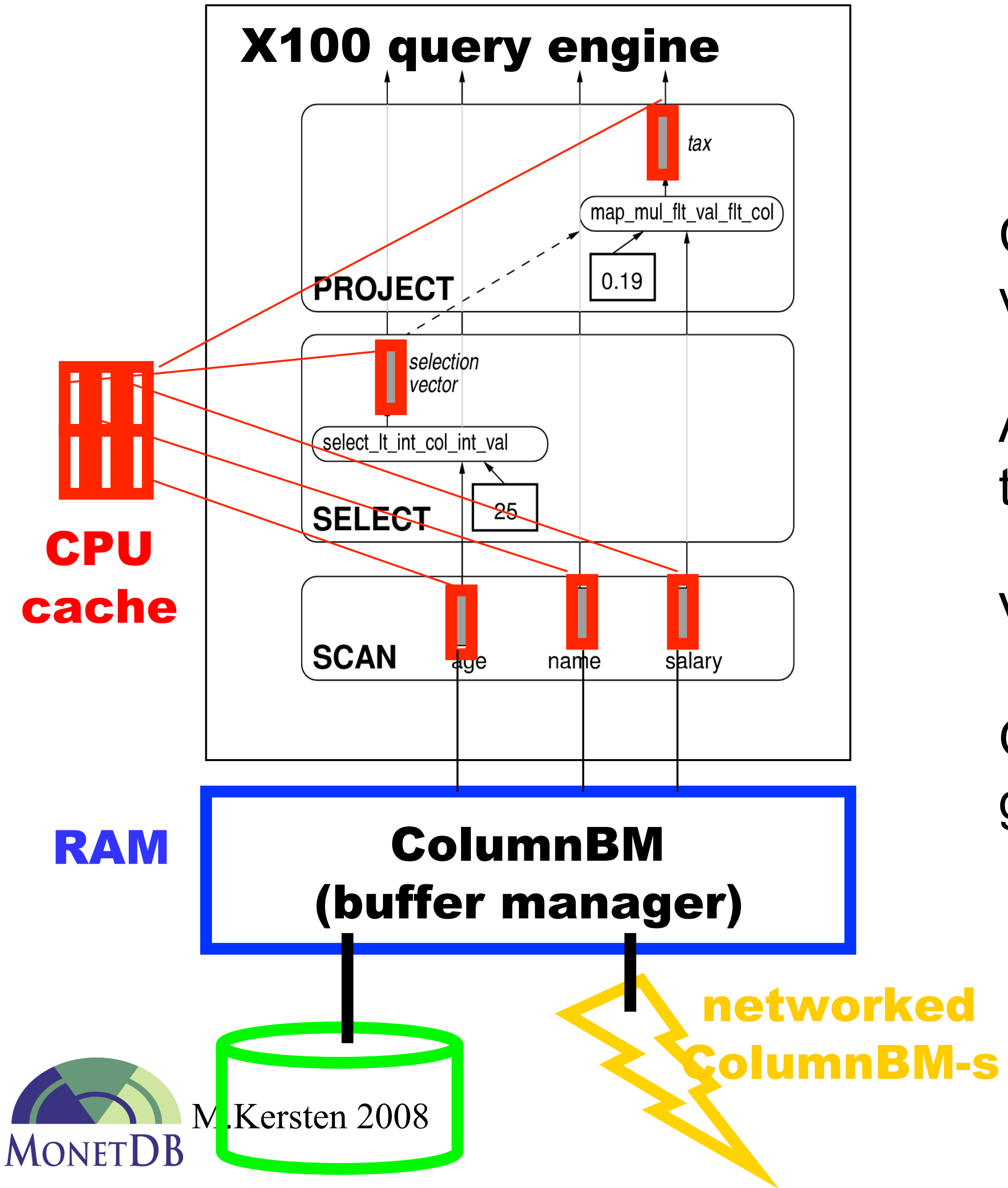

Database
Structures
Execution
Query
optimizer
DBMS
Architecture

Execution paradigms

- The MonetDB kernel is set up to accommodate different execution engines

- The MonetDB assembler program is
  - Interpreted in the order presented
  - Interpreted in a dataflow driven manner
  - Compiled into a C program
  - Vectorised processing
    - X100 project

MonetDB/x100



Combine Volcano model with vector processing.

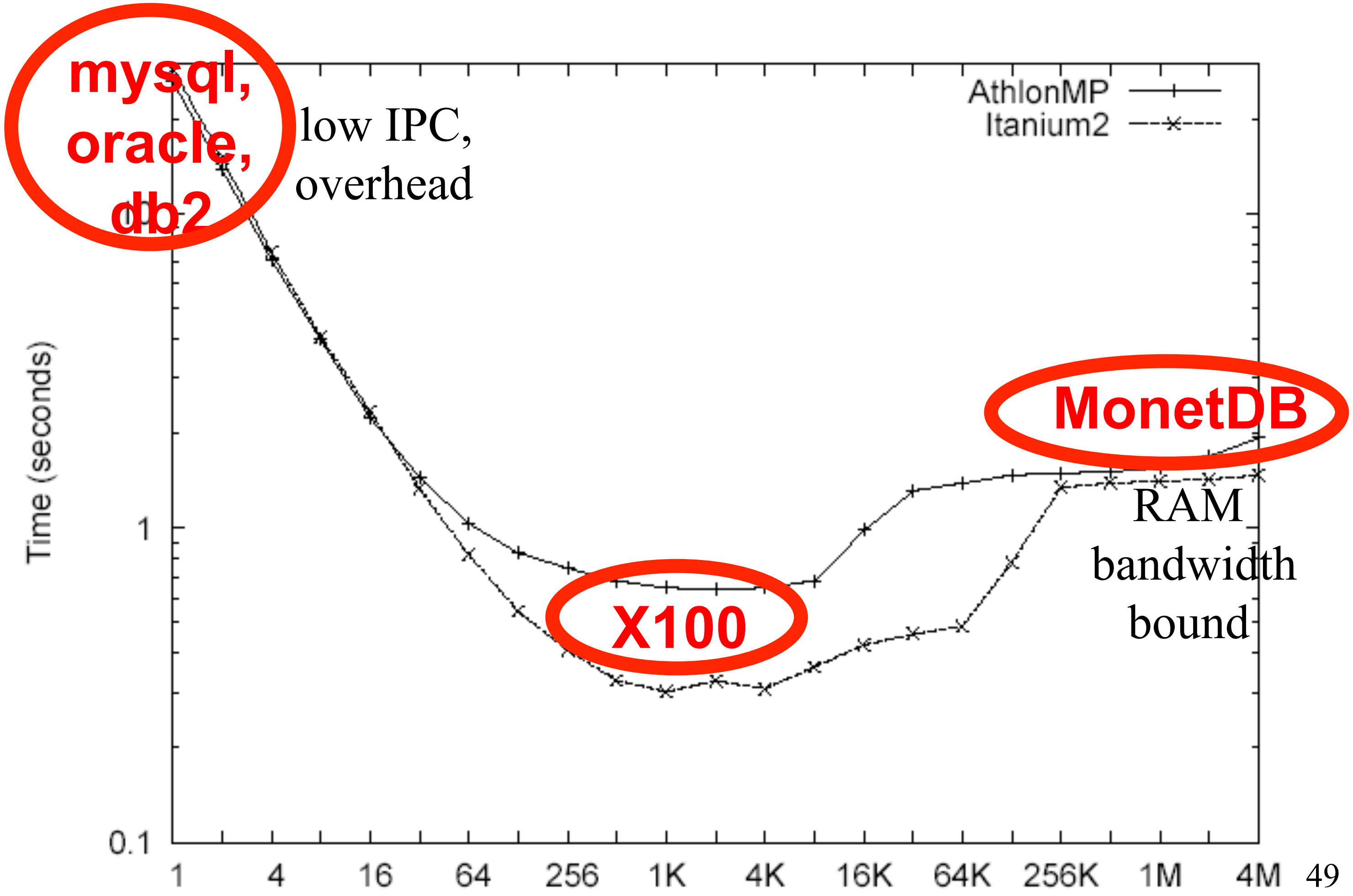
All vectors together should fit the CPU cache

Vectors are compressed

Optimizer should tune this, given the query characteristics.

No data from persistent store to the memory trash

- Varying the vector size on TPC-H query 1



Query evaluation strategy

- Pipe-line query evaluation strategy
  - Called Volcano query processing model
  - Standard in commercial systems and MySQL
- Basic algorithm:
  - Demand-driven evaluation of query tree.
  - Operators exchange data in units such as records
  - Each operator supports the following interfaces:– open, next, close
    - ***open()*** at top of tree results in cascade of opens down the tree.
    - An operator getting a ***next()*** call may recursively make ***next()*** calls from within to produce its next answer.
    - ***close()*** at top of tree results in cascade of close down the tree

Query evaluation strategy

- Pipe-line query evaluation strategy
  - Evaluation:
    - Oriented towards OLTP applications
      - Granule size of data interchange
    - Items produced one at a time
    - No temporary files
      - Choice of intermediate buffer size allocations
    - Query executed as one process
    - Generic interface, sufficient to add the iterator primitives for the new containers.
    - CPU intensive
    - Amenable to parallelization

Query evaluation strategy

- Materialized evaluation strategy
  - Used in MonetDB
  - Basic algorithm:
    - for each relational operator produce the complete intermediate result using materialized operands
  - Evaluation:
    - Oriented towards decision support queries
    - Limited internal administration and dependencies
    - Basis for multi-query optimization strategy
    - Memory intensive
    - Amendable for distributed/parallel processing

TPC-H

	MonetDB	PostgreSQL	MySQL
	5.2.3	8.2.6	5.0.45
SF	0.01		
oad	1097	1734	4409
	ms		
1	29	558	366
2	9	15	35
3	12	39	141
4	10	18	34
5	12	21	223
6	4	44	156
7	13	31	59
8	9	35	30
9	14	124	75
10	12	20	132
11	7	21	27
12	9	61	74
13	47	51	112
14	7	43	301
15	6	67	152
16	9	46	80
17	4	12	10
18	11	92	18
19	18	64	
20	12	1810	11
21	26	199	27
22	8	420	13

TPC-H
60K rows line_item table
Comfortably fit in memory
Performance in milliseconds

ATHLON X2 3800+ (2000mhz) 2 disks in raid 0, 2G main memory

TPC-H

	MonetDB	PostgreSQL	MySQL	MonetDB	PostgreSQL	MySQL
	5.2.3	8.2.6	5.0.45	5.2.3	8.2.6	5.0.45
SF	0.01			1		
oad	1097	1734	4409	96466	342103	140888
	ms			ms		
1	29	558	366	4004	61253	36305
2	9	15	35	104	3344	152870
3	12	39	141	935	18504	34137
4	10	18	34	663	3273	3187
5	12	21	223	867	758	19931
6	4	44	156	171	14011	6897
7	13	31	59	902	17395	5967
8	9	35	30	353	17101	2940
9	14	124	75	855	49769	9233
10	12	20	132	692	1014	12424
11	7	21	27	65	2491	39164
12	9	61	74	404	14303	6023
13	47	51	112	5532	6461	8128
14	7	43	301	350	13129	32832
15	6	67	152	110	15093	18023
16	9	46	80	487	8972	9086
17	4	12	10	176	> 1 hour	1042
18	11	92	18	698	35273	> 1 hour
19	18	64		1060	15208	297
20	12	1810	11	538	> 1 hour	16936
21	26	199	27	2483	47391	99153
22	8	420	13	246	> 1 hour	718

Scale-factor 1
6M row line-item table

Out of the box performance
Queries produce empty
or erroneous results

ATHLON X2 3800+ (2000mhz) 2 disks in raid 0, 2G main memory

TPC-H

	MonetDB	PostgreSQL	MySQL	MonetDB	PostgreSQL	MySQL	MonetDB	PostgreSQL	MySQL
	5.2.3	8.2.6	5.0.45	5.2.3	8.2.6	5.0.45	5.2.3	8.2.6	5.0.45
SF	0.01			1			2		
oad	1097	1734	4409	96466	342103	140888	211	733	303
	ms			ms			sec		
1	29	558	366	4004	61253	36305	6	121	80
2	9	15	35	104	3344	152870	1	6	308
3	12	39	141	935	18504	34137	3	31	91
4	10	18	34	663	3273	3187	1	6	17
5	12	21	223	867	758	19931	3	2	5755
6	4	44	156	171	14011	6897	0	23	27
7	13	31	59	902	17395	5967	2	37	2655
8	9	35	30	353	17101	2940	121	32	504
9	14	124	75	855	49769	9233	3	256	57
10	12	20	132	692	1014	12424	3	4	91
11	7	21	27	65	2491	39164	1	5	175
12	9	61	74	404	14303	6023	2	27	23
13	47	51	112	5532	6461	8128	20	15	132
14	7	43	301	350	13129	32832	2	27	15193
15	6	67	152	110	15093	18023	1	31	49
16	9	46	80	487	8972	9086	1	17	19
17	4	12	10	176	> 1 hour	1042	4	> 1 hour	27
18	11	92	18	698	35273	> 1 hour	13	125	> 1 hour
19	18	64		1060	15208	297	4	31	50
20	12	1810	11	538	> 1 hour	16936	1	> 1 hour	162
21	26	199	27	2483	47391	99153	131	107	1492
22	8	420	13	246	> 1 hour	718	3	> 1 hour	1755

ATHLON X2 3800+ (2000mhz) 2 disks in raid 0, 2G main memory

TPC-H

	MonetDB	PostgreSQL	MySQL	MonetDB	PostgreSQL	MySQL	MonetDB	PostgreSQL	MySQL	MonetDB	PostgreSQL	MySQL
	5.2.3	8.2.6	5.0.45	5.2.3	8.2.6	5.0.45	5.2.3	8.2.6	5.0.45	5.2.3	8.2.6	5.0.45
SF	0.01			1			2			5		
oad	1097	1734	4409	96466	342103	140888	211	733	303	793	3570	1233
	ms			ms			sec			sec		
1	29	558	366	4004	61253	36305	6	121	80	74	293	201
2	9	15	35	104	3344	152870	1	6	308	88	25	1028
3	12	39	141	935	18504	34137	3	31	91	29	79	421
4	10	18	34	663	3273	3187	1	6	17	21	13	112
5	12	21	223	867	758	19931	3	2	5755	425	13	> 1 hour
6	4	44	156	171	14011	6897	0	23	27	10	75	58
7	13	31	59	902	17395	5967	2	37	2655	47	99	> 1 hour
8	9	35	30	353	17101	2940	121	32	504	58	82	2818
9	14	124	75	855	49769	9233	3	256	57	85	538	> 1 hour
10	12	20	132	692	1014	12424	3	4	91	34	108	344
11	7	21	27	65	2491	39164	1	5	175	56	24	504
12	9	61	74	404	14303	6023	2	27	23	19	77	60
13	47	51	112	5532	6461	8128	20	15	132	40	61	802
14	7	43	301	350	13129	32832	2	27	15193	20	73	> 1 hour
15	6	67	152	110	15093	18023	1	31	49	5	60	113
16	9	46	80	487	8972	9086	1	17	19	26	36	49
17	4	12	10	176	> 1 hour	1042	4	> 1 hour	27	402	>1 hour	185
18	11	92	18	698	35273	> 1 hour	13	125	> 1 hour	21	69	>> 1 hour
19	18	64		1060	15208	297	4	31	50	14	571	> 1 hour
20	12	1810	11	538	> 1 hour	16936	1	> 1 hour	162	17	>1hour	489
21	26	199	27	2483	47391	99153	131	107	1492	188	302	> 1 hour
22	8	420	13	246	> 1 hour	718	3	> 1 hour	1755	84	>1hour	4

ATHLON X2 3800+ (2000mhz) 2 disks in raid 0, 2G main memory