Query Execution in Main Memory DBMSs

- Postgres on Disk: 100.0 s
- Postgres on Ramdisk: 96.0 s
- In-Memory DBMS: 0.7 s
Query Execution in Main Memory DBMSs
select
    l_returnflag, l_linenstatus,
    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_linenstatus
order by
    l_returnflag, l_linenstatus;
Query Execution in Main Memory DBMSs

- Postgres on Disk: 100.0 s
- Postgres on Ramdisk: 96.0 s
- In-Memory DBMS: 0.7 s
BRANCHING CAN BE BAD!

Section 5.2: Implementing the vectorized execution model

Figure 5.6: Performance of control-dependency based and data-dependency based selection routines (Core 2 Duo)

```c
void aggr_count_int_vec_int_vec(int *result, int *groupids, int n) {
    for (int i = 0; i < n; i++)
        result[groupids[i]] += 1;
}
```

In this code, each tuple depends on the previous one, causing data stalls in the CPU pipeline. One approach to reduce these stalls, is to use multiple copies of the `result` array, and make different tuples update different versions of it.

```c
void aggr4_count_int_vec_int_vec(int **result, int *groupids, int n) {
    for (int i = 0; i < n; i += 4) {
        result[0][groupids[i+0]] += 1;
        result[1][groupids[i+1]] += 1;
        result[2][groupids[i+2]] += 1;
        result[3][groupids[i+3]] += 1;
    }
}
```

The latter solution, while minimizing data dependencies between iterations, increases the memory consumption for `result` arrays by a factor 4. Still, if such extra cost is acceptable, this approach allows for a significant performance improvement. For example, on our Core2Duo test machine it improved the performance from already very good 2.76 cycles/tuple (with 256 groups) to 2.05 cycles/tuple. On some architectures this difference can be significantly larger.

Another solution to the data dependency problem is to combine multiple operations into one primitive. For example, in some scenarios, multiple aggregates are computed at one processing stage – such a situation occurs in TPC-H query 1 [HNZB07]. Then, it is possible to compute e.g. 4 aggregates in one primitive:

```c
void multiaggr_sum_int_vec4_int_vec(int **result, int **values, int *groupids, int n) {
    for (int i = 0; i < n; i++) {
        ...
    }
}
```
FUNCTION CALLS ARE BAD!

Summing 2Gig worth of integers, 1 Call per Integer

- Inline: 227 ms
- Single Call: 1,104 ms
- Four Unique Calls: 5,165 ms
Furthermore, operators are connected by materializing intermediate results as temporary tables inside the buffer pool and streaming them to subsequent operators.
Our approach has multiple benefits: 

(a) the number of function calls during query evaluation is minimized; 
(b) the generated code exhibits increased data locality, therefore making optimal use of cache-resident data; 
(c) code generation and compilation allow the use of compiler optimization techniques targeting each individual query, an extra optimization level on top of conventional query optimization; and 
(d) the generated code approaches the performance of hard-coded evaluation plans. The model is flexible and does not affect other or-
The measured performance translates to a block that lacks function calls and is tailored towards efficient grouping, and aggregation operations in a single succinct code. This is due to the holistically generated code: it inlines all selection, magnitude, reaching a number of four and the other in Figure 8(a) show the holistic aggregation algorithm is map aggregation. The results of distinct value cardinalities equal to six, the most appropriate vertical partitioning and outperform all query. Thus, the expectation is for MonetDB to benefit from cache lines, with only a few fields actually needed by any functions.

These include highly selective join predicates that cannot be evaluated as join teams, as well as aggregation operations. This is expensive over the benchmark tables, as sorting. The holistic optimizer stages all inputs before further coded implementation, thus achieving maximum efficiency in positivity that MonetDB/X100's outperforms PostgreSQL and System X by a substantial factor, compared to MonetDB. Compared to the DSM-based systems, MonetDB/X100’s results prove the viability of holistic evaluation in like MonetDB: through vertical partitioning only the required like MonetDB: through vertical partitioning only the required...
NO!

"column at a time"

MonetDB/MIL

\[ \sim 18x \]
This allows X100 to generate primitives in a just-in-time fashion. To overcome this problem, the execution primitive for multiplication of two vectors of floating point numbers might look like this:

```c
(a * b) // X100
```

Table 2 shows, spend only a few cycles per tuple. In the case of a multiplication of two values, X100 spends 2 cycles per tuple, whereas MySQL spends 49. Complex expressions must be executed in X100 by calling "tuple at a time" and "column at a time". Currently, primitive generation is predefined, but the ultimate goal is to allow a query optimizer to generate and compile compound primitives in a just-in-time fashion. The hand-coded C program does not need these load/stores as subexpression results are passed through CPU registers.
HYPER VS. HAND-CODED

Naïve Hand-Coded: 250 ms
Hyper: 623 ms