MonetDB/X100 - A DBMS In The CPU Cache

Marcin Zukowski, Peter Boncz, Niels Nes, Sándor Héman
Centrum voor Wiskunde en Informatica
Kruislaan 413, Amsterdam, The Netherlands
{M.Zukowski, P.Boncz, N.Nes, S.Heman}@cwi.nl

Abstract

X100 is a new execution engine for the MonetDB system, that improves execution speed and overcomes its main memory limitation. It introduces the concept of in-cache vectorized processing *that strikes a balance between the existing column-at-a-time MIL execution primitives of MonetDB and the tuple-at-a-time Volcano pipelining model, avoiding their drawbacks: intermediate result materialization and large interpretation overhead, respectively. We explain how the new query engine makes better use of cache memories as well as parallel computation resources of modern super-scalar CPUs. MonetDB/X100 can be one to two orders of magnitude faster than commercial DBMSs and close to hand-coded C programs for computationally intensive queries on in-memory datasets. To address larger disk-based datasets with the same efficiency, a new ColumnBM storage layer is developed that boosts bandwidth using* ultra lightweight compression *and* cooperative scans.

1 Introduction

The computational power of database systems is known to be lower than hand-written (e.g. C++) programs. However, the actual performance difference can be surprisingly large. Table 1 shows the execution time of Query 1 of the TPC-H benchmark run on various database systems and implemented as a separate program (with data cached in the main memory in all situations). We choose Query 1 as it is a simple single-scan query (no joins), that calculates a small aggregated result, thus mainly exposing the raw processing power of the system. The results show that most DBMSs are orders of magnitude slower than hand-written code.

DBMS “X”	MySQL 4.1	MonetDB/MIL	**MonetDB/X100**	hand-coded
28.1s	26.6s	3.7s	**0.60s**	0.22s

Table 1: TPC-H Query 1 performance on various systems (scale factor 1)

We argue that the poor performance of MySQL and the commercial RDBMS “X” is related to the Volcano iterator pipelining model [6]. While the Volcano approach is elegant and efficient for I/O bound queries, in computation-intensive tasks, its tuple-at-a-time execution makes runtime interpretation overhead dominate execution. That is, the time spent by the RDBMS on actual computations is dwarfed by the time spent interpreting

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

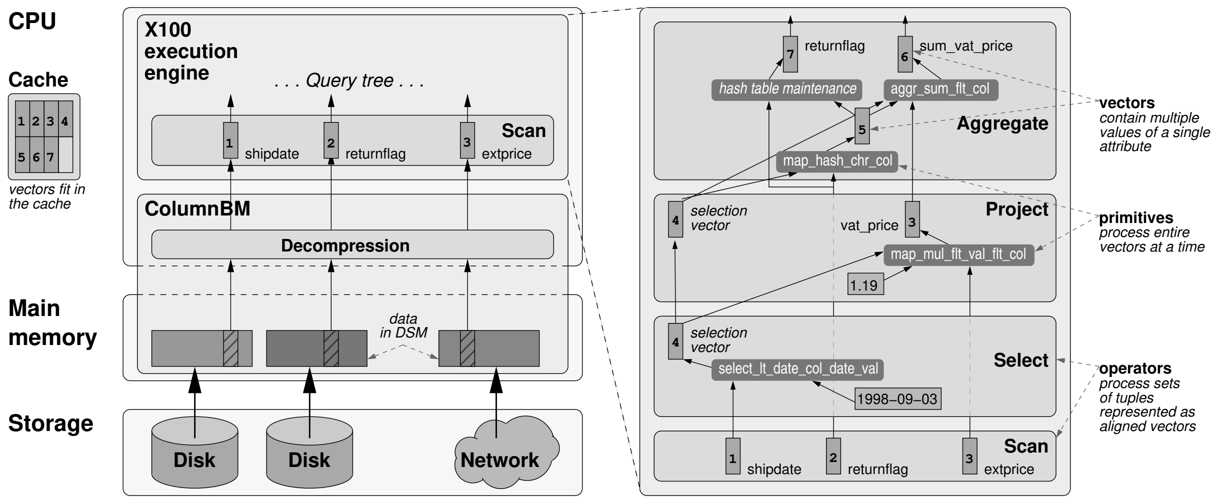

Figure 1: X100 – architecture overview and execution layer example

the expressions in a query tree, and looking up the needed attribute values in pages storing tuples in the N-ary storage model (NSM). As a result, quite a high number of CPU instructions are needed for each simple computational operation occurring in a query. Additionally, the instructions per cycle (IPC) efficiency achieved tends to be low, because the tuple-at-a-time model hides from the CPU most parallel execution opportunities, namely those provided by performing computations for *different* tuples in an overlapping fashion.

MonetDB [2], developed at CWI [1], is an alternative approach to building a query execution system. Its MIL algebra [3] minimizes the interpretation overhead by providing execution primitives that process data in a column-at-a-time fashion. Since MonetDB uses vertically decomposed storage model (DSM) [5], where columns are simple arrays, and each primitive performs only one fixed operation on the input data, the primitive implementation boils down to a sequential loop over aligned input arrays. Modern compilers can apply *loop pipelining* to such code, allowing MIL primitives to achieve high IPC on super-scalar CPUs. However, the column-at-a-time execution model introduces the extra cost of intermediate result materialization, which is only sustainable inside main memory, thus limiting the scalability of the system.

Figure 1 presents the architecture of X100, a new execution engine for MonetDB that combines the benefits of low-overhead column-wise query execution with the absence of intermediate result materialization in the Volcano iterator model. One of its distinguishing features is *vectorized execution*: instead of single tuples, entire *vectors* of values are passed through the pipeline. Another is *in-cache processing*: all the execution is happening within the CPU cache, using main memory only as a buffer for I/O and large intermediate data structures. These techniques make X100 execute efficiently on modern CPUs and allow it to achieve raw performance comparable to C code.

To bring the high-performance query processing of X100 to large disk-based datasets, a new *ColumnBM* storage manager is currently being developed. It focuses on satisfying the high bandwidth requirements of X100 by applying *ultra lightweight compression* for boosting bandwidth and *cooperative scans* for optimizing multi-query scenarios. Preliminary results show that it often achieves performance close to main memory execution without the need for excessive numbers of disks.

The outline of this article is as follows. In Section 2 we describe the hardware features important for the design of X100. Then, the system architecture is presented, concentrating on query execution in Section 3 and on the ColumnBM storage manager in Section 4. Finally we conclude and discuss future work.

[1] MonetDB is now in open-source, see `monetdb.cwi.nl`

2 Modern CPU features

In the last decade, a CPU clock-speed race resulted in dividing execution of a single instruction into an instruction pipeline of simple stages. The length of this pipeline continuously increases, going as far as 17 stages in AMD Opteron and 31 stages in Intel Pentium4. Another innovation found in modern super-scalar CPUs is the increase in number of processing (pipeline) units. For example, the AMD Opteron has 2 load/store units, 3 integer units and 3 floating point units, and allows for executing a mix of 3 instructions per cycle (IPC).

To fully exploit the pipelines, the CPU needs to know which instructions will be executed next. In the case of conditional branches, the CPU usually guesses which path the program will follow using a branch prediction mechanism and performs speculative execution of the chosen code. However, if a branch misprediction occurs, the pipeline needs to be flushed and the processing restarts with an alternative route. Obviously, with longer pipelines more instructions are flushed, increasing the performance penalty.

Another feature required to fully exploit available execution units is instruction independence. For example, when calculating `a=b+c;d=b*c;` both `a` and `d` can be calculated independently using two units. However, for `a=b+c;d=a*c;` processing of `d` has to be delayed until `a` is evaluated, resulting in just one processing unit being used. Figure 2 shows how *loop pipelining* eliminates this problem by interleaving the computation of multiple loop iterations. That is, `x[i]` is scheduled 8 instructions after the computation of `t0` started, making the result of `t0` immediately available for use. A similar effect can be achieved without compiler by a CPU that implements out-of-order execution, allowing it to execute instructions further up the stream (i.e. belonging to next loop iterations) while earlier instructions still wait on the result of another. Crucially, these techniques only work when a loop processes a large number of independent iterations. This property of the execution system, already present in MonetDB/MIL where one operation is executed on an entire column in a tight loop, is further improved in X100.

1) original code

```
for(i=0;i<n;i++) {
  t=a[i]+b[i];
  x[i]=t*c[i];
}
```

2) loop pipelining

```
for(i=0;i<n;i+=8) {
  t0=a[i]+b[i];
  t1=a[i+1]+b[i+1];
  ...
  t7=a[i+7]+b[i+7];
  x[i]=t0*c[i];
  x[i+1]=t1*c[i+1];
  ...
  x[i+7]=t7*c[i+7];
}
```

Figure 2: Loop Pipelining

Among the features of modern CPUs, cache memories received most attention from the DBMS community. A growing family of cache-conscious algorithms reduce access to main memory by tuning the access pattern of query processing algorithms and data structures [11, 1, 9, 8]. X100 goes even further by limiting its entire execution engine to the cache, as presented in the next section.

3 X100 architecture

The X100 engine is designed for *in-cache execution*, which means that the only “randomly” accessible memory is the CPU cache, and main memory is already considered part of secondary storage, used in the buffer manager for buffering I/O operations and large intermediate results. The other main principle behind X100 is *vectorized execution*, sketched in Figure 1:

- vertically decomposed tables are further partitioned horizontally into small chunks called *vectors*.
- a set of aligned vectors (one for each attribute), representing a set of tuples, is a single data unit in the execution pipeline.
- an optional *selection vector* contains the positions of the tuples currently taking part in processing. This removes the need for the extra after-selection projection steps present in MonetDB/MIL.
- the control logic of the operators is common for all data types, arithmetic functions, predicates etc.
- the actual data processing in the operators is performed by a set of *execution primitives* - simple, specialized and CPU-efficient functions.

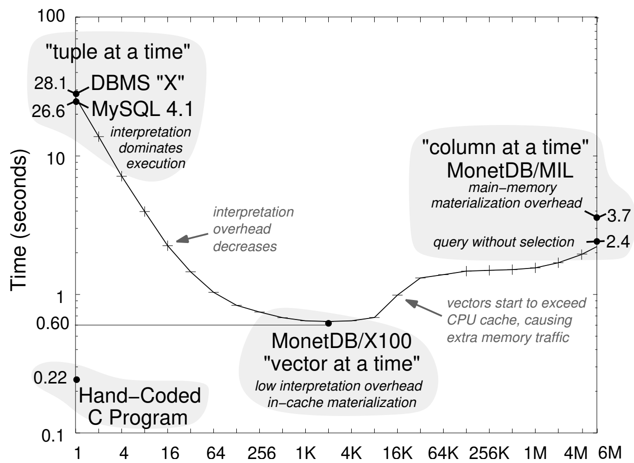

Figure 3: TPC-H Query 1 performance in X100 with varying vector size (in tuples, horizontal axis)

input count	time (us)	avg. cycles	**X100 primitive**
6M	13307	3.0	`select_lt_usht_col_usht_val`
5.9M	10039	2.3	`map_sub_flt_val_flt_col`
5.9M	9385	2.2	`map_mul_flt_col_flt_col`
5.9M	9248	2.1	**`map_mul_flt_col_flt_col`**
5.9M	10254	2.4	`map_add_flt_val_flt_col`
5.9M	13052	3.0	`map_uidx_uchr_col`
5.9M	14712	3.4	`map_directgrp_uidx_col_uchr_col`
5.9M	28058	6.5	`aggr_sum_flt_col_uidx_col`
5.9M	28598	6.6	`aggr_sum_flt_col_uidx_col`
5.9M	27243	6.3	`aggr_sum_flt_col_uidx_col`
5.9M	26603	6.1	`aggr_sum_flt_col_uidx_col`
5.9M	27404	6.3	`aggr_sum_flt_col_uidx_col`
5.9M	18738	4.3	`aggr_count_uidx_col`

Table 2: MonetDB/X100 performance trace during TPC-H Query 1 (primitives)

The execution primitive for multiplication of two vectors of floating point numbers might look like this:

```
int map_mul_flt_col_flt_col(int n, flt* res, flt* col1, flt* col2, int *sel)
{
  for(int i=0; i<n; i++)
      res[sel[i]] = col1[sel[i]] * col2[sel[i]];
  return n;
}
```

The simplicity of these primitives allows compilers to produce code that achieve IPC of over 2 and, as 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 [4]. Complex expressions must be executed in X100 by calling multiple primitives that each materialize intermediate results. While this materialization is in-cache, hence highly efficient, it still causes a high ratio of load-store instructions. For example, the multiplication primitive reads from `sel`, `col1` and `col2`, and writes in `res`, thus needing 4 load/stores for 1 ”work” instruction. A hand-coded C program does not need these load/stores as subexpression results are passed through CPU registers. To overcome this problem, primitives in X100 are *generated* from a so-called signature request and a code pattern. This allows X100 to generate *compound primitives* that execute an entire expression subtree (e.g. `(a*1.19+b)*0.95`). 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.

A key tuning factor in X100 is the size of the vectors, which can be chosen at run-time. Figure 3 shows the performance of MonetDB/X100 on TPC-H Query 1 with vector sizes varying from 1 (tuple-at-a-time) to 6M (the entire table: column-at-a-time). For better understanding, we also plotted the results of MySQL, DBMS ”X”, MonetDB/MIL and the hand-coded C program from Table 1. The performance of X100 at vector size 1 is highly similar to that of the relational DBMS systems. As the vector size is increased, performance of MonetDB/X100 improves by two orders of magnitude, clearly showing that interpretation overhead was the bottleneck. With further increase, however, the cache size boundary is crossed and cache misses start to occur. In other words, the materialized intermediate results become too big, making the query memory bandwidth bound. At size 6M, we indeed observe the performance of MonetDB/X100 to be suboptimal and highly similar to MonetDB/MIL. MonetDB/MIL is slower mainly due to materialization the of 99% selection of Query 1, which is avoided in X100 thanks to selection vectors. With this selection omitted, the performance of X100 exactly coincides with MIL.

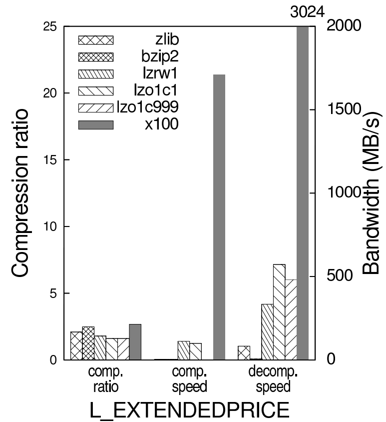

Figure 4: Performance of various compression algorithms on example TPC-H data

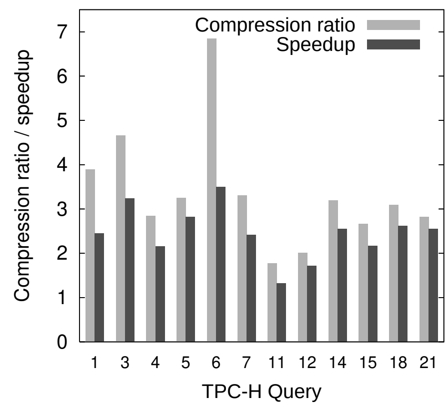

Figure 5: Compression ratio and performance improvement on a subset of TPC-H queries

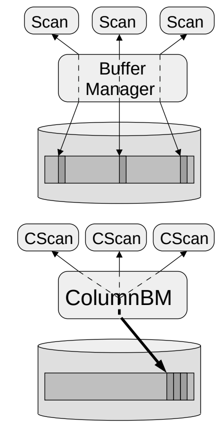

Figure 6: Scan processing in a traditional system and in the ColumnBM

4 ColumnBM

While the X100 execution engine is efficient in main memory scenarios, achieving similar performance for disk-based data is a real challenge. Due to its raw computational speed, MonetDB/X100 exhibits an extreme hunger for I/O bandwidth. As an extreme example, TPC-H Query 6 uses 216MB of data (SF=1), and is processed in MonetDB/X100 in less than 100 ms, resulting in a bandwidth requirement of ca. 2.5GB per second. For most other queries this requirement is lower, but still in the range of hundreds of megabytes per second. Clearly, such bandwidth is hard to achieve except by using expensive storage systems consisting of large numbers of disks. ColumnBM combines three techniques (DSM, compression and cooperative scans) to combat this problem.

DSM. ColumnBM vertically fragments tables on disk using DSM, which saves bandwidth if queries scan a table without using all columns. Its main disadvantage is an increased cost of updates: a single row modification results in one I/O per each influenced column. ColumnBM tackles this problem by treating data chunks as immutable objects and storing modifications in the (in-memory) delta structures, periodically updating the chunks [4]. During the scan, data from disk and delta structures are merged, providing the execution layer with a consistent state. While ColumnBM uses DSM, this is not strictly required by the X100 execution model. The rationale behind the in-cache column-wise layout (i.e. vectors) in X100 is not optimizing memory storage or reducing I/O bandwidth, but to allow a compiler to detect loop-pipelining opportunities in its execution primitives. To store data with a high-update rate, ColumnBM will also support the PAX [1] storage scheme, which stores entire tuples in disk blocks, but uses vectors to represent the columns inside such blocks.

Compression. While compression in databases was proposed by many researchers [7, 10], we introduce two novel techniques: *ultra lightweight compression* and *into-cache decompression*. Traditional compression algorithms usually try to maximize the compression ratio, making them too expensive for use in a DBMS. X100 introduces a family of new highly CPU-efficient compression algorithms, specifically designed to create a (de)compression kernel, that compiles into a pipelinable loop by making it simple, predictable and eliminating all if-then-else constructs. As Figure 4 shows, these new algorithms achieve a throughput of over 1 GB/s during compression and a few GB/s in decompression, beating speed-tuned general purpose algorithms like LZRW and LZO, while still obtaining comparable compression ratios. Preliminary experiments presented in Figure 5 show the speedup of the decompressed queries to be close to the compression ratio, which in case of TPC-H allows for a bandwidth (and performance) increase of a factor 3.

Most database systems employ decompression right after reading data from the disk, storing buffer pages in uncompressed form. This solution requires data to cross the memory-cache boundary three times: when it is delivered to the CPU for decompression, when uncompressed data is stored back in the buffer, and finally when it is used by the query. Since such approach would make X100 decompression routines memory-bound, ColumnBM stores disk pages in a compressed form and decompresses them just before execution, on a per-vector granularity. Thus (de)compression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk. This approach nicely fits the delta-based update mechanism, as merging the deltas can be applied after decompression, and chunks need to be re-compressed only periodically.

Cooperative Scans. While compression is tuned to improve performance of isolated queries, it is often the case that *multiple* queries are running at the same time, often fighting for disk bandwidth. If they read the same columns, ColumnBM applies *cooperative scans* [12], a technique in which queries, instead of reading data in a fixed sequential order, try to reuse data already buffered by other queries. Disk accesses are scheduled to satisfy the largest possible number of ”starving” queries. Preliminary results show that X100 on a single CPU can sustain a load of 30 I/O bound queries (TPC-H Query 6) without performance loss.

5 Conclusions and future work

In this article we presented X100, a novel execution engine for MonetDB. By applying vectorized in-cache execution, it efficiently exploits the features of modern CPUs and allows for raw in-memory performance one to two orders of magnitude higher than other systems. Additionally, we discussed ColumnBM, a new storage system that enables scaling MonetDB to large disk-based datasets.

X100 and ColumnBM are still in an experimental stage. Both components will play a role in our future research into architecture-aware query processing, e.g. looking at new CPU features such as SMT and CMT.

From a software point of view, X100 is part of the MonetDB software family so it will re-use its API infrastructure and query front-ends (SQL and XQuery). X100 is already being used in multimedia information retrieval on the TREC-VIDEO collections. Other applications on our agenda for X100 are data mining and analysis of astronomy datasets.

References

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

[2] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[3] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB J.*, 8(2):101–119, 1999.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, 2005.

[5] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, 1985.

[6] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[7] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.

[8] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE TKDE*, 14(4):709–730, 2002.

[9] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD*, 2000.

[10] M. Roth and S. van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, September 1993.

[11] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. VLDB*, 1994.

[12] M. Zukowski, P. A. Boncz, and M. L. Kersten. Cooperative scans. Technical Report INS-E0411, CWI, December 2004.