Improving I/O Bandwidth for Data-Intensive Applications

Marcin Zukowski

Centrum voor Wiskunde en Informatica
Kruislaan 413, 1090 GB Amsterdam, The Netherlands
M.Zukowski@cwi.nl

Abstract. High disk bandwidth in data-intensive applications is usually achieved with expensive hardware solutions consisting of a large number of disks. In this article we present our current work on software methods for improving disk bandwidth in ColumnBM, a new storage system for MonetDB/X100 query execution engine. Two novel techniques are discussed: *superscalar compression* for standalone queries and *cooperative scans* for multi-query optimization.

1 Introduction

High-performance data processing applications like OLAP, data mining and scientific data analysis require high disk bandwidth to provide enough data to the execution layer. Table 1 presents hardware details of the systems currently leading in the TPC-H 100GB benchmark [15] for the most common 4-CPU configuration. These systems achieve high bandwidth thanks to a large number (42-112) of disks typically connected through a high-end, expensive infrastructure. As a result, storage subsystem might constitute up to 80% of the entire system cost.

Table 1. Component costs in 4-way TPC-H 100GB systems

CPUs	RAM	Disks
4xPower5 1650MHz (9%)	32GB (13%)	42x36GB=1.6TB (78%)
4xItanium2 1500MHz (24%)	32GB (15%)	112x18GB=1.9TB (61%)
4xXeon MP 2800MHz (25%)	4GB (3%)	74x18GB=1.2TB (72%)
4xXeon MP 2000MHz (30%)	8GB (7%)	85x18GB=1.6TB (63%)

Using over 1TB of storage for 100GB of data might be questionable in real life situations. In many cases a simple RAID systems consisting of 4-12 drives would probably be a more practical solution. Since the bandwidth provided by such systems might not be sufficient for high-performance data processing, in this article we propose a set of software techniques trying to bridge this gap.

Table 2. Execution time of TPC-H Query 1 (scale factor 1) on various systems

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

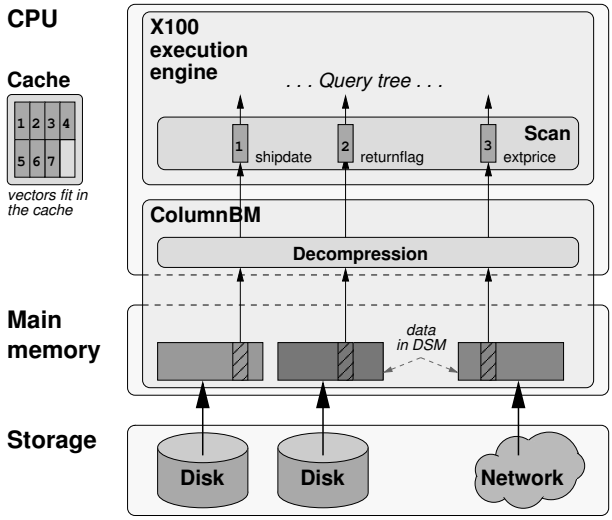

Fig. 1. ColumnBM and X100 overview

1.1 MonetDB/X100 query engine

Our interest in improving the performance of a storage layer stems from our recent research on X100 [2, 18], a high-performance query execution engine for the MonetDB [1] system. Thanks to a new *vectorized in-cache execution* model and a good use of modern CPU features, X100 achieves performance an order of magnitude higher than traditional DBMS and close to hand-written solutions, as presented in Table 2.

Due to its raw computational power, X100 exhibits unusually high bandwidth requirements. As an example, TPC-H Query 6 uses 216MB of data 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 value is lower, but still in the range of hundreds of megabytes per second. While such bandwidth is possible in main memory scenarios, achieving similar performance for disk-based data is a real challenge.

1.2 ColumnBM storage manager

To scale X100 performance to disk, we currently work on ColumnBM, a dedicated storage system for MonetDB. As Figure 1 shows, it combines the bandwidth of multiple storage devices, both disks and remote machines.

ColumnBM stores data using a vertically *decomposed storage model* [4] (DSM). This saves bandwidth if queries scan a table without using all columns. The main disadvantage of this model is an increased cost of updates: a single row modification results in one I/O per each influenced column. To tackle this problem ColumnBM uses a technique similar to differential files [14]. Vertical columns are divided into large data chunks (>1MB) that are treated as immutable objects. Modifications are stored in the (in-memory) delta structures, and chunks are updated only periodically. During the scan, data from disk and delta structures are merged, providing the execution layer with a consistent state.

To further improve data bandwidth ColumnBM applies two techniques. *Superscalar compression* allows improving performance of standalone queries by almost-transparent data decompression. *Cooperative scans* coordinate the work of multiple running queries so that they share I/O and achieve higher overall bandwidth.

1.3 Outline

The paper is organized as follows. In Section 2 we present the compression techniques introduced in ColumnBM, concentrating on *memory-to-cache* decompression and *ultra lightweight* algorithms. In Section 3 we present our recent work on scan processing for traditional *N-ary storage model* (NSM) and discuss problems related to applying this work to DSM used in ColumnBM. Finally, in Section 4 we conclude and present future work.

2 Superscalar RAM-cache compression

In I/O bound queries the CPU uses only a part of its processing power. The remaining part can be used to decompress the data read from the disk, resulting in a higher perceived disk bandwidth and improved overall performance:

$$R = \begin{cases} Br & : \ \frac{Br}{C} + \frac{Br}{Q} \leq 1 \quad \text{(I/O bound)} \\ \frac{QC}{Q+C} & : \ \frac{Br}{C} + \frac{Br}{Q} \geq 1 \quad \text{(CPU bound)} \end{cases}$$

r = compression ratio
B = I/O bandwidth
Q = processing bandwidth
C = decompression bandwidth
R = result query bandwidth

Compression is beneficial if the fraction of the CPU time used by the decompression ($\frac{Br}{C}$) is small. In an example scenario we use a RAID system delivering 300MB/s, compression ratio of 3, and a 3 GHz CPU. To limit the decompression overhead to 30% of the CPU time, the decompression routines need to provide bandwidth of C=3GB/s or higher.

2.1 Ultra lightweight compression

Traditional compression algorithms usually maximize the compression ratio at the cost of speed, making them too expensive for a DBMS. ColumnBM introduces a family of specialized compression routines that use simple and predictable code with minimized number of conditional branches. As a result they

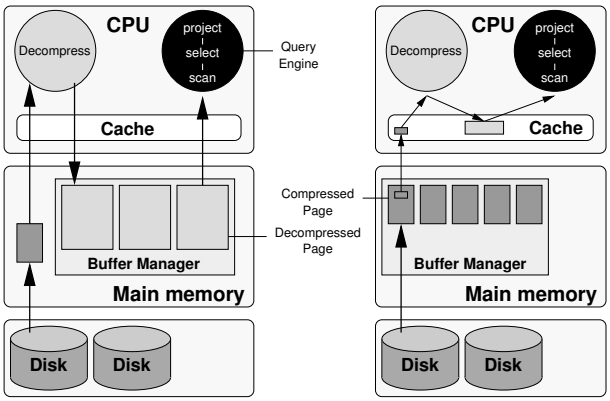

Fig. 2. Disk-to-memory and memory-to-cache decompression

execute efficiently on modern *superscalar* CPUs and they achieve a throughput of over 1 GB/s during compression and a few GB/s during decompression, beating speed-tuned general purpose algorithms like LZRW and LZO, while still obtaining comparable compression ratios.

2.2 Memory-to-cache decompression

Most database systems employ decompression right after reading data from the disk, storing buffer pages in an uncompressed form. As the left-hand part of Figure 2 shows, this 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 an approach would make decompression routines memory-bound, ColumnBM stores disk pages in a compressed form and decompresses them just before execution on a per-vector granularity. Thus, decompression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk, as presented in right-hand part of Figure 2. This approach fits nicely the delta-based update mechanism, as merging the deltas can be applied after decompression, and chunks need to be re-compressed only periodically.

2.3 Future research

ColumnBM compression currently concentrates on lossless compression of numerical data. Preliminary experiments show the speedup of the I/O bound TPC-H queries to be close to the compression ratio (ca. 3), proving that decompression overhead is minimal. In the future we plan to work on high-performance compression algorithms for other data types and value distributions. Moreover, we want to investigate techniques allowing the system to decide automatically when to compress the data and which algorithm to apply.

2.4 Related work

The benefits of applying compression in database systems were identified in [8, 13]. The implementation of such a system was described in [16], including query optimization also discussed in [3]. Various forms of compression have also been applied in commercial systems, including Oracle [12], Sybase IQ and others.

Lightweight compression in databases has been proposed in [16, 7]. In our work we go further by using an entire family of compression algorithms specialized for various data types and value distributions. Moreover, thanks to the CPU-friendly design our algorithms achieve significantly higher performance and minimize the decompression overhead.

3 Cooperative scans

While compression improves performance of isolated queries, it is usually the case that multiple queries are running at the same time competing for disk bandwidth. If many queries process the same table, each of them issues I/O requests independently and gets only a fraction of the available disk bandwidth. Additionally, the buffer manager concentrates on keeping the most recently used data in the buffer pool, possibly evicting pages that could soon be reused by a running scan.

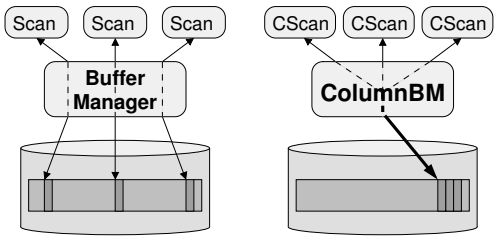

Fig. 3. Scan processing in a traditional system and with cooperative scans

In [17] we described our preliminary research on the idea of *cooperative scans*, presented in Figure 3. In this approach queries, instead of enforcing one particular data delivery order, cooperate to share as much bandwidth as possible. We presented a number of variants of this idea, differing in complexity and behavior in various scenarios. Preliminary results on PostgreSQL and ColumnBM show that (assuming sufficient processing power) multiple queries running in parallel can achieve performance close to a standalone case. For example, X100 on ColumnBM was able to sustain the same response time processing up to 30 instances of the TPC-H Query 6.

3.1 Future research

Algorithms presented in [17] provide a solution in the case of scans over a single relational table using an N-ary storage model (NSM). In our future research we plan to address a few issues that make the situation more complex: scans over multiple tables, scans over DSM tables, and compression.

When queries process multiple tables, the system needs to allocate I/O resources and buffer space for each running scan. We plan to introduce an algorithm that would dynamically adapt to the current system state, looking at information like the number of interested and starving queries for each table.

With vertically decomposed tables, applying per-column scheduling is obviously not possible, as different attributes need to be delivered in the same order to constitute a full tuple. If queries read disjoint or identical subsets of the relation attributes, these sub-relations could be processed like separate NSM tables. For queries with only partially overlapping attributes, we plan to develop a new scheduling algorithm that would exploit the DSM properties. For example, the system should not only determine which vertical table chunk to read, but also which attributes and in what order.

The final problem is related to our choice of memory-to-cache decompression presented in Section 2.2. In the case of multiple queries reading the same page from the buffer manager, each needs to decompress it separately, increasing overall decompression cost. As a result, a set of queries that were I/O bound might easily become CPU bound. Another solution would be for the first query to materialize a decompressed page it in the buffer manager, and let the others read uncompressed data. The choice of when to apply this strategy mainly depends on the number of queries interested, the speed of decompression routines, and the in-memory materialization cost.

3.2 Related work

Multi-query optimization traditionally concentrated on reusing common subexpression results both in materializing [9] and pipelining [5] scenarios. Various ideas have also been proposed to improve scan performance, e.g. by allowing a query to compute multiple results in a single scan [6], or scheduling similar queries together [11, 10]. While we are not aware of any scientific publications discussing ideas close to cooperative scans, similar solutions seem to have been incorporated into commercial systems including Red Brick warehouse, Teradata database and MS SQL Server (as *shared scans*). Unfortunately, no implementation details are available, making it hard to compare them to our proposal.

4 Conclusions and future work

In this article we presented the state of our research on improving disk performance for data intensive applications. Two techniques were presented: super-scalar compression and cooperative scans. While not fully implemented, they

already provide significant performance benefits. In the future we will address the unresolved problems that were discussed, mainly automatic data compression and cooperative scans for multi-table DSM queries. We plan to evaluate our ideas by comparing the performance of the ColumnBM storage manager working with traditional strategies and the described extensions. We will use both micro-benchmarks to determine the raw performance of the introduced techniques as well as the TPC-H benchmark to simulate their behavior in real life scenarios.

References

1. P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.
2. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.
3. Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. *SIGMOD Rec.*, 30(2), 2001.
4. G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, Austin, USA, 1985.
5. N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. *J. Comput. Syst. Sci.*, 66(4), June 2003.
6. J. C. et al. NonStop SQL/MX primitives for knowledge discovery. In *Proc. KDD*, San Diego, CA, USA, 1999.
7. J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. ICDE*, 1998.
8. G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.
9. S. Manegold, A. Pellenkoft, and M. L. Kersten. A Multi-Query Optimizer for Monet. Technical Report INS-R0002, January 2000.
10. W. Müller and A. Henrich. Reducing I/O Cost of Similarity Queries by Processing Several at a Time. In *Proc. MDDE*, Washington, DC, USA, 2004.
11. K. O'Gorman, D. Agrawal, and A. E. Abbadi. Multiple query optimization by cache-aware middleware using query teamwork (poster paper). In *Proc. ICDE*, San Jose, CA, USA, 2002.
12. M. Pöss and D. Potapov. Data Compression in Oracle. In *Proc. VLDB*, Berlin, Germany, 2003.
13. M. Roth and S. van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, September 1993.
14. D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3), 1976.
15. Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002. `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`.
16. T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3), September 2000.
17. M. Zukowski, P. A. Boncz, and M. L. Kersten. Cooperative scans. Technical Report INS-E0411, CWI, December 2004.
18. M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100: A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, June 2005.