ColumnBurst: A Near-Storage Accelerator for Memory-Efficient Database Join Queries

Gongjin Sun
gongjins@uci.edu
Department of Computer Science
University of California, Irvine

Sang-Woo Jun
swjun@ics.uci.edu
Department of Computer Science
University of California, Irvine

ABSTRACT

We present ColumnBurst, a memory-efficient, near-storage hardware accelerator for database join queries. While the paradigm of near-storage computation has demonstrated performance and efficiency benefits on many workloads by reducing data movement overhead, memory-bound operations such as relational joins on unsorted data have been relatively inefficient with fast modern storage devices, due to the limited capacity and performance of memory available on the near-storage processing engine. ColumnBurst delivers very high performance even on such complex queries, while staying within the memory performance and capacity budget of what is typically already available on off-the-shelf storage devices. ColumnBurst achieves this via a compact, hardware implementation of sorting-based group-by aggregation and join algorithms, instead of the conventional hash-based algorithms. We evaluate ColumnBurst using an FPGA-based prototype with 1 GB of slow on-device DDR3 DRAM, and show that on benchmarks including TPC-H queries with join queries on unsorted columns, it outperforms MonetDB on a 6-core i7 with 32 GB of DRAM by over 7×, and ColumnBurst using a near-storage hash join algorithm by 2×.

CCS CONCEPTS

• **Information systems → Storage architectures**; *Join algorithms*; • **Hardware → Hardware accelerators**.

ACM Reference Format:
Gongjin Sun and Sang-Woo Jun. 2020. ColumnBurst: A Near-Storage Accelerator for Memory-Efficient Database Join Queries. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410494

ACM ISBN 978-1-4503-8069-0/20/08.
https://doi.org/10.1145/3409963.3410494

1 INTRODUCTION

As the scale of data analytics requirements as well as the performance of secondary storage continue to increase, near-storage processing is becoming an attractive paradigm of making effective use of the high performance of modern storage devices. Fast storage devices built using NAND-flash or Non-Volatile Memory technologies are shifting the bottleneck from storage to other parts of the system, such as the processor or the interconnect between the processor and storage. For example, off-the-shelf SSDs have shown an internal bandwidth of almost 2× compared to their interconnect [15]. Near-storage processing augments the storage device with a computation offload engine, which both adds computation capacity as well as reduces the data transported over the interconnect. Many previous research have shown great performance benefits, often orders of magnitude improvements over analytics software running purely on the host CPU [9, 12, 14, 18, 22, 24].

While a vast selection of analytics operations has been explored on near-storage accelerator platforms with success, memory-intensive operations, especially with complex access patterns on large working sets, have been relatively inefficient with fast modern storage devices due to the limited memory resources on the near-storage processing engine [15]. For example, a hash-based algorithm for group-by aggregation and relational joins on unsorted data require random access on a hash table exceeding the size of on-chip memory. Meanwhile, a typical storage device is equipped with a small amount of performance-restricted memory devices such as Low-Power DDR4 (LPDDR4), resulting in lower random access performance as benchmarked in Section 4. Furthermore, for modern SSDs with NAND-flash and Intel X-Point, most of that memory is already used by the SSD controller for important flash management functions. As a result, near-storage offloading of such complex operations is either avoided [7, 9, 12, 13, 15, 21], only achieves performance similar to host CPU [25], targets slow SATA SSDs [26], or requires costly addition of memory resources which affect the cost-effectiveness of near-storage accelerators [10].

ColumnBurst is a memory-efficient accelerator for efficiently offloading database join queries near storage, targeting solid-state storage such as NAND-flash. In order to address the memory resource issue, and to achieve high performance on complex queries, ColumnBurst uses a hardware implementation of a sorting-based group-by aggregation algorithm on a small amount of memory, instead of the conventional hash-based aggregation and join algorithm. This also forms the basis of its join algorithm. ColumnBurst's aggregation algorithm interleaves sorting and aggregation operations whenever possible. While merging is only possible for inner joins with a unique key, it can often dynamically reduce data volume when applicable. Section 4 shows that many queries of interest have this feature.

The benefit of sorting-based aggregation is twofold: (1) Unlike hash-based algorithms which require fine-grained random access, merge-sort in hardware has a completely sequential access pattern into memory, making the best use of its bandwidth. (2) The aggregation results are sorted by the key, meaning downstream join operations can turn into simple merge joins.

We have constructed an Field-Programmable Gate Array (FPGA)-based prototype implementation of ColumnBurst, which includes 1 GB of on-board DDR3 DRAM SODIMM card. We compare its performance to MonetDB [11] and MySQL [20] on a 6-core i7 workstation with 32 GB of memory, as well as a version of ColumnBurst which implements a naive hash-based join algorithm. Besides microbenchmarks for component evaluation, we evaluated these systems on a subset of queries from the TPC-H benchmark: query 6, which demonstrates conventional filtering and aggregation offloading, and queries 13 and 14, which demonstrate the major focus of ColumnBurst, join performance on unsorted data, with various selectivity ratios. On datasets scaling from scale factor 100 (100 GB) to 500 (500 GB), we show that ColumnBurst outperforms MonetDB by almost 7×, and our hash-based accelerator by almost 3× for high selectivity queries. Relative performance of MySQL was multiple orders of magnitude lower than all other configurations.

While we have focused on these queries as they help isolate and compare operations of interest, complex queries with more join stages will also benefit from ColumnBurst, either by sharing the on-board memory between multiple join operations, chaining join operations, or by offloading only the earlier part of the query plan. An intelligent query planner should be able to take advantage of ColumnBurst, but this is a topic for future work.

The rest of this paper is organized as follows: Section 2 describes ColumnBurst's join algorithm based on sorting-based group-by aggregation, and Section 3 describes the architecture of ColumnBurst in detail. Section 4 presents our evaluation results, and we conclude with discussions and future work in Section 5.

2 SORTING-BASED NEAR-STORAGE ACCELERATOR FOR JOIN OPERATIONS

When joining two unsorted database tables, either a **sorting-based**, or a **hash-based** algorithm typically is used. Contrary to previous research showing hash-based join algorithms having better performance compared to sort-based join algorithms [2, 8, 16, 23], our experience with hardware accelerators showed that sort-based join algorithms often outperform hash-based algorithms, due to the performance limitations of on-board memory resources. The low-power memory devices on storage devices have relatively low random access performance, due not only to device component characteristics, but also due to its long minimum burst lengths [17]. As a result, the random-access intensive hashing step often becomes a bigger performance bottleneck compared to sorting, especially when the computation aspect of sorting can be efficiently offloaded to the accelerator while also enjoying a sequential access pattern to memory. This observation agrees with the trends presented by previous research [2, 16] that predicted the performance relation between hash and sort based algorithms may change as the processing power relative to memory performance continues to increase.

Sorting-Based Aggregation: ColumnBurst uses an accelerator implementation of a sorting based group-by aggregation algorithm, which also forms the basis of its join operation. The completely sequential access pattern of merge-sort avoids the random-access performance penalty of hashing. ColumnBurst implements a high fan-out merge-sorter and aggregator to minimize the number of data accesses, while interleaving sorting and merging operations to dynamically reduce data [1, 3].

The benefits of interleaving merging is especially beneficial for a high fan-out hardware merge-sorter. The throughput of a tree of merge-sorters is limited by the fixed-rate datapath of the root node. If merging is done at intermediate nodes of the tree, the data that the root node must process can be reduced drastically.

Aggregator Accelerator for Joins: ColumnBurst uses this aggregator implementation to perform relational join operations when the join column of one or more tables is not ordered. In a sort-merge join algorithm, the unordered tables are first sorted according to the join column, after which it turns into a simple ordered merge join operation.

ColumnBurst first uses its aggregator accelerator to sort and aggregate the unordered table according to the join column. Joins are most efficient when the *cardinality*, or the ratio of unique values in the column, of the join column

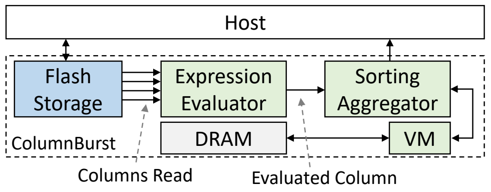

Figure 1: Overall architecture and data flow of ColumnBurst

is low. In such a situation, the resulting aggregated table will be much smaller than if the original table had simply been sorted, reducing the subsequent merge-join overhead, as well as allowing more data to fit on the on-board memory.

3 COLUMNBURST ARCHITECTURE

Figure 1 shows the architecture of ColumnBurst. The ColumnBurst architecture consists of hardware implementations of three components: Expression evaluator, Sorting Aggregator, and Virtual Memory. Columns of interest are read from storage into the expression evaluator, which can perform some early filtering as well as expression evaluation including arithmetic between columns. The resulting data is streamed to the sorting aggregator, which either performs an aggregation operation on the stream, or a join operation between the input stream and an aggregated table cached in memory by a previous aggregation operation. The results of the aggregator can be sent to host, or stored in memory for further joining. The virtual memory system provides convenient and high-performance on-board DRAM access for the sorting aggregator.

3.1 Expression Evaluator

In order to do a realistic evaluation of ColumnBurst, its architecture includes a *Expression Evaluator*. The expression evaluator is simply a hardware environment to make sure we read a realistic amount of data from the storage for a query, in order to accurately characterize the effects of storage performance. It will be removed when porting ColumnBurst to a full-fledged query accelerator.

The expression evaluator assumes table data is stored in a column-oriented way, where each column is stored in a separate file, and ingests multiple streams of columns at once via multiple ports. The expression evaluator can be configured in a limited way to perform filtering and arithmetic operations according to a user query, and emits a single stream of key-value pairs for the aggregator. Due to the simple design of the expression evaluator, it cannot take advantage of advanced features like indices, but can support enough queries to demonstrate ColumnBurst's performance.

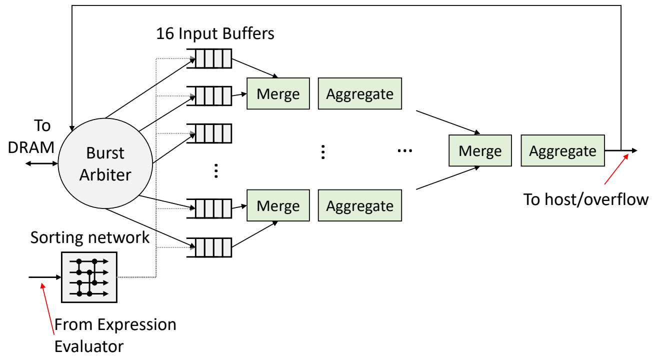

Figure 2: ColumnBurst uses a 16-way sort-aggregator to perform group-by aggregation

3.2 Sorting Aggregator

The sorting aggregator module performs the sorting-based aggregation operation in on-board memory, using a wide fan-out merge sorter tree interleaved with programmable aggregate function modules. In the current ColumnBurst prototype, the sorting aggregator implements a 16-way merge sorter tree. The internal structure of the sorting aggregator module is shown in Figure 2. Each intermediate merge and aggregate modules can emit up to one key-value pair per cycle, but the effective data ingestion rate is typically much higher thanks to data being merged at intermediate layers. We decided against a multi-rate sorter in order to keep on-chip resource usage low enough for a low-cost FPGA.

The initial input from the expression evaluator is first fed through a sorting network before being scattered across the input buffers. For example, the current prototype implements a 8-tuple sorting network, resulting in sorted blocks of length 8 fed through the merge sorter tree.

On-Board DRAM Layout: In order to simplify memory management, the location of each sorted block in the on-board memory is pre-determined regardless how much data aggregation managed to reduce. For example, the output of the initial merge sort phase on the input data from the expression evaluator is a sorted block with up to $8 \times 16 = 128$ elements. While the number of elements may be smaller if aggregation was effective, the blocks are stored at offsets of multiples of 128 element units. This way, the locations of blocks to read and write are deterministic. If aggregation was effective at any merge-sort phase resulting in the number of stored elements smaller than the space allocated, the end of the valid data is marked with a *null* element, prompting the rest of the block to be ignored.

Memory Layout For Multi-Step Sort-Aggregate: Because the input stream for aggregation will be larger than the on-board memory capacity for most interesting analytical queries, the stream will need to be processed in multiple parts. To simplify memory management, the total usable

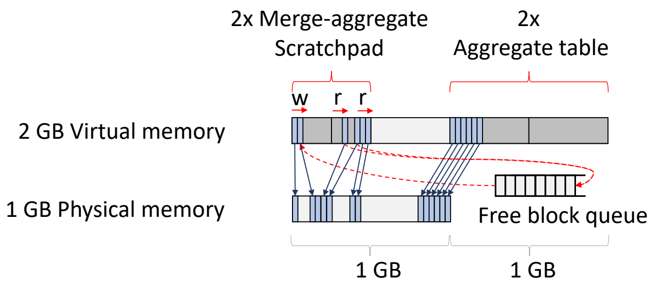

Figure 3: Mapping from 2 GB virtual memory to 1 GB physical memory, during merge-aggregate in scratchpad.

physical memory is divided into two halves: the first half is used as scratchpad memory for the aggregator, and the second half is used to store the total aggregated data so far. The scratchpad half is further divided into two halves, which are used as double buffers for the sort-aggregator, where each half is used as source and destination buffers alternately. For example, for a 1 GB DRAM device, the maximum amount of aggregated data that can be held is 512 MB. The remaining 512 MB is divided into two halves of 256 MBs, and sorting is done by copying data back and forth between them.

While this memory layout is simple, it has an issue: At the last merge sort phase, when sorted blocks in the 512 MB region and the aggregated block in the 512 MB region are merged together, there is no contiguous 512 MB block to write the result to. In order to address these issues, ColumnBurst implements a very simple virtual memory mapping described in Figure 3.

Virtual memory is organized into large pages of 8 MB, and the virtual memory manager maps a virtual memory address much larger than available physical memory. A large page configuration allows simple management of block mappings on-chip. The aggregate table can be merged and by copying data between two large virtual memory regions, dynamically mapping physical memory for output, and reclaiming input blocks that have been consumed so that there is always physical memory available for mapping. In the worst case, data ingestion can happen in such a way that 16 input blocks will be returned at the same time, in which case 16 blocks' worth of output data will be emitted before the 16 input blocks can be returned to the free queue. When merging the 256 MB of aggregated data into the 512 MB block, one of the two scratchpads are now unused. Because 16× 8 MB blocks add up to 128 MB, there is always enough physical memory available with space for the block map to spare.

When total aggregated data exceeds the memory capacity allocated (e.g., 512 MB), the later parts of the sorted stream are sent to the software fallback via the overflow path seen in Figure 2.

4 EVALUATION

In this section we provide an evaluation of ColumnBurst, using an FPGA-based prototype implementation. We first provide the environment and configuration in which we evaluated the ColumnBurst prototype, as well as other systems for comparison. We also provide evaluations on the benefits of interleaving sorting with aggregation, in our aggregation algorithm, in terms of data reduction and the resulting performance improvement. Finally, we provide performance comparisons against existing systems using queries from the TPC-H [5] benchmark.

4.1 Evaluation Environment

We evaluate ColumnBurst by comparing the performance of its prototype implementation against two software DBMSs, MySQL [20] and MonetDB [4], as well as a modified version of ColumnBurst which uses a hash-based aggregation method. MySQL is one of the most widely used DBMSs [6], while MonetDB is one of the fastest available column stores [19]. Both software were tuned using on-line resources to the best of our abilities for maximum performance.

ColumnBurst Prototype: We have implemented a prototype version of ColumnBurst on a BlueDBM near-storage accelerator development platform deployment [14] plugged into the PCIe Gen2 ×8 of a host server. The prototype storage device consists of 1 TB of NAND-flash storage, with a raw connection to a Xilinx VC707 FPGA development board over the FPGA Mezzanine Card (FMC) connection. The peak measured bandwidth between the flash array and the FPGA board is 2.4 GB/s, and the latency is 75 μs. The raw connection between the array of flash chips and the FPGA allows us to experiment with drastic changes in the FTL, including the block-level management we are using in ColumnBurst. All ColumnBurst accelerator modules are implemented in VC707's FPGA.

Resource and Power: The accelerator portion, despite the low effort put into optimization, only consumed an aggregate of less than 200K LUTs and 164 KB of on-chip BRAM while running at 250 MHz. This amount of resources can fit in a low-cost Artix 7 FPGA, with less than 10 Watts of typical power consumption [27].

On-Board DRAM: The VC707 FPGA board is also equipped with a 1 GB DDR3 SODIMM card, which is a good proxy for embedded DRAM modules on modern storage devices. Not only does the last-generation DDR3 SODIMM card have similar read latency and bandwidth to the current-generation Low-Power DDR4 (LPDDR4) DRAM modules, the DDR3 controller in ColumnBurst groups the 8 banks in the memory module into a single interface, resulting in a 64-byte access granularity which matches the minimum burst length of modern LPDDR4 DRAM modules [17]. Bank grouping also

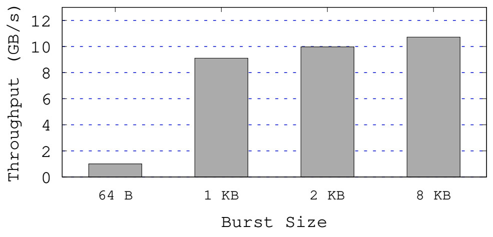

Figure 4: Random access performance of the on-board DDR3 DRAM by burst size

	ColumnBurst	SATA	NVMe
Description	Custom Storage	4x SATA-III RAID-0	2x NVMe RAID-0
Capacity	1 TB	4 TB	2 TB
Bandwidth			
Rand. 4 KB	1.2 GB/s	0.7 GB/s	3.2 GB/s
Rand. 8 KB	2.4 GB/s	1 GB/s	3.2 GB/s
Seq. 4 KB	2.4 GB/s	1.5 GB/s	3.5 GB/s

Table 1: The storage profiles of the evaluation environments

collects the row buffers from all banks into a single wider row buffer, which mimics the characteristics of mobile DRAM. Figure 4 shows the measured random read performance of the on-board DRAM module according to the read request sizes. At the minimum burst size, 64 Bytes, the bandwidth of the DRAM module is an order of magnitude lower than when read at the row buffer size of 8 KB.

CBHash: CBHash is a modified version of ColumnBurst, which uses a hash-based aggregation and join algorithm instead of a sorting-based one. However, since we did not have an efficient implementation of a DRAM hash with BRAM caching ready, we estimated the upper bound performance of hash-based join and aggregation algorithms by generating a trace of DRAM read/write requests using a software simulation of a direct-mapped write-back cache, and measuring the time required to simply perform all I/O operations on the on-board DRAM. Since this method does not suffer from non-I/O performance bottlenecks such as read-after-write hazards or sudden I/O bursts that force back-pressure on the previous processing stages, its results represent the ideal upper bound on the performance of CBHash.

System for Software Execution: All software was executed on a workstation with 6-core (12 threads) i7-8700K running at 3.70 GHz, as well as four 8 GB DDR4-2133 DIMMs adding up to 32 GB. Two sets of backing storage were provided, a RAID-0 array of four SATA-III SSDs adding up to 4 TB, and a RAID-0 array of two PCIe NVMe SSDs adding up to 2 TB. The performance characteristics can be seen in Table 1. The BlueDBM storage has 8 KB physical pages, so 4 KB random reads halves the available bandwidth. The performance relationship between SATA, BlueDBM, and NVMe are consistent throughout all tests. MySQL was run on the SATA raid, while MonetDB was run on both SATA and NVMe raid configurations. This means compared to MonetDB on NVMe RAID, ColumnBurst is already starting out with a storage performance disadvantage.

4.2 Evaluation Workloads

We evaluated the systems introduced above, on three benchmarks from TPC-H [5]: Query 6, Query 13, and Query 14. TPC-H is a benchmark suite that focuses on analytics performance of database systems. Query 6 is a single-table aggregation query chosen to evaluate the usual aggregation, filtering, and expression evaluation performance of the near-data accelerator. Query 13 includes a join query between two tables, which has a filtering predicate with a high selectivity, meaning not many rows are filtered out. Query 14 includes a join query between two tables, which has a filtering predicate with a very low selectivity, where most rows are filtered out. Queries 13 and 14 involved only two tables, where one join column is ordered while the other is not, which well-isolates the effects of near-storage join acceleration. As of now, all query plans were hand-created so that filtering and arithmetic expressions were mapped to their corresponding accelerator modules.

The queries were run on three different sizes synthesized using the TPC-H dbgen tool : Scale 100 (100 GB), Scale 250 (250 GB) and Scale 500 (500 GB).

4.3 Effects of Interleaving Sort and Aggregation

We first present the benefits of our aggregation algorithm's choice of interleaving sorting with aggregations. Figure 5 shows the ratio of data read and written at every phase of 16-way merge sort, from the aggregation operation during TPC-H query 13, for three different data scales. It shows the amount of data sharply decreasing at the last two merge-sort phases before the table is fully sort-aggregated, significantly reducing the data I/O required from DRAM.

While the data I/O overhead of the first few sort phases could be removed using an array of sorter using on-chip memory, we decided to not use it in favor of conserving on-chip memory and logic resources.

4.4 TPC-H Query 6

Figure 6 presents the performance evaluation of various systems on the TPC-H Query 6, for datasets of Scale 100 and 250. Query 6 is a single-table query with arithmetic expressions and aggregations, and ColumnBurst is able to process all data at line-rate. CB represents ColumnBurst

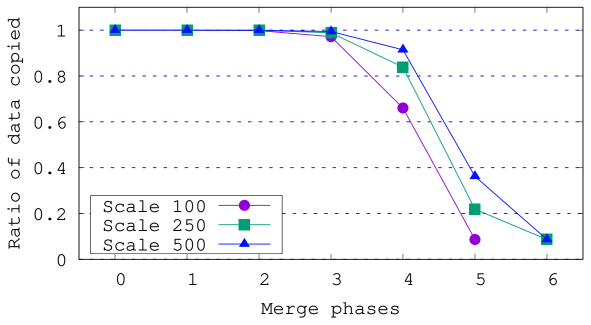

Figure 5: Dynamic reduction of data during a 16-way merge sort by interleaving with aggregation, for TPC-H query 13

performance, while MonetDB$_S$ and MonetDB$_P$ represents MonetDB running on SATA and NVMe RAIDs, respectively.

Performance results for MySQL on Scale 250 are omitted because execution took too long (Hours), and Scale 500 results are omitted because on our system, MonetDB was unable to load the `lineitem` table of the Scale 500 dataset. While ColumnBurst managed to complete execution on the Scale 500 dataset, there was no system to compare against, and is omitted. Because there is no group-by aggregation or join operations, ColumnBurst results are not divided into hash or sort-based implementations. Performance was measured after queries were run multiple times, putting the database into a "warm" state where in-memory caches are populated.

MonetDB performs very well on Scale 100 for both storage configurations, where all data required for the query is cached in memory, but at Scale 250, MonetDB on NVMe RAID demonstrates roughly double the performance compared to SATA. When cold-started without the benefits of cached tables, both instances of MonetDB on Scale 100 showed slower performance, which matches the trend seen in Scale 250. ColumnBurst shows best performance at Scale 250 despite the storage performance disadvantage, as the accelerators allow line-rate processing. MySQL performance was relatively low, even compared to MonetDB$_S$.

4.5 TPC-H Query 13

Figure 7 presents the performance of various systems on the TPC-H Query 13, for datasets of Scale 100 to Scale 500. Query 13 joins two of the larger tables, `customer` and `orders`, where `orders` is unordered according to the join column. There is a string-matching selection predicate on `orders`, but the selectivity is high, where roughly about 77% of rows is kept.

We see that CBSort demonstrates the best performance across all data scales consistently demonstrating almost 8× speed-up compared to MonetDB on NVMe, and almost 3× speed-up compared to CBHash. The performance of CBHash was bottlenecked by the low random access performance of the on-board DRAM, and the majority of the execution time was spent on memory I/O. There was no significant performance difference between MonetDB on SATA or NVMe RAIDs, as this query is mostly computation-bound. One interesting observation was that on Scale 500, MonetDB on SATA slightly outperformed the NVMe system consistently across multiple executions. We predict this is due to some complex interplay between storage access and computation. MySQL continued to demonstrate relatively low performance.

4.6 TPC-H Query 14

Figure 8 presents the performance of various systems on the TPC-H Query 14, for datasets of Scale 100 and 250. Query 14 joins two of the largest tables, `lineitem` and `part`, where the largest table `lineitem` was unordered according to the join column. There is a date-comparison selection predicate on `lineitem`, and the selectivity is very low, where only 1.3%

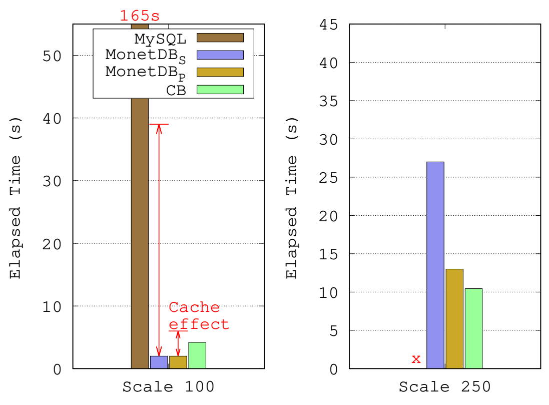

Figure 6: Elapsed time on the TPC-H query 6 on various platforms. Lower is better

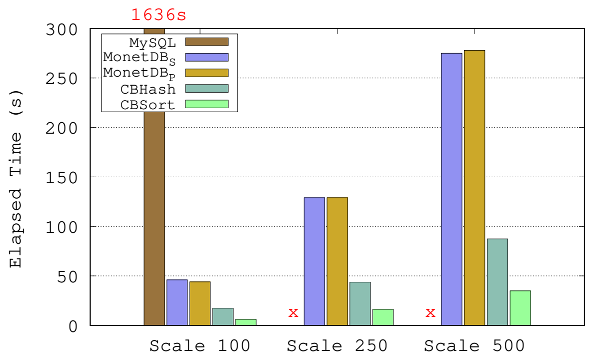

Figure 7: Elapsed time on the TPC-H query 13 on various platforms. Lower is better

of data is kept. Scale 500 was omitted for the same reason as query 6, where MySQL was too slow and MonetDB was unable to load `lineitem`.

At Scale 100, MonetDB outperformed all other systems after the cache was warmed up, while all systems showed similar performance at Scale 250. Cold-started MonetDB without cache effects demonstrated slower performance, somewhat matching the patterns seen with Scale 250. Interestingly, CBHash outperformed CBSort slightly. This is because the amount of hash access was so low due to the low selectivity that the hash access could be hidden within file read latency. However, because this also meant sorting overhead was also low, the performance difference was marginal.

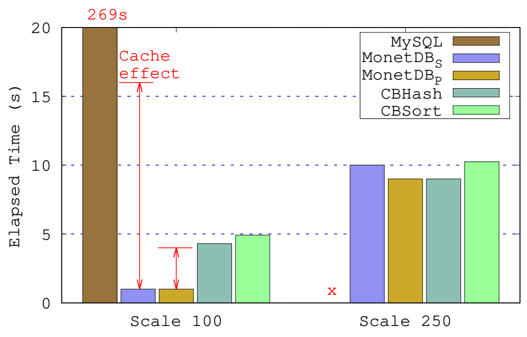

Figure 8: Elapsed time for the TPC-H query 14 on various platforms. Lower is better

The performance relations change significantly with a different selectivity. Figure 9 shows the performance of various systems on query 14 on data Scale 250, with varying selectivity. The left graph shows performance with selectivity between 0.013 (original) and 0.1. It shows that even when selectivity is only 0.1, the sort-based system demonstrates better performance compared to the hash-based one. The right of Figure 9 shows the results from when the selection predicate is completely removed, with a breakdown of where the latency comes from. It can be seen that with such a large table to hash or sort (`lineitem` is the largest table in the dataset by far), the time spent by the accelerator becomes dominant. The accelerator performance is bottlenecked by memory for CBHash and accelerator throughput for CBSort.

4.7 Discussion on Aggregated Table Width

For both queries 13 and 14, the aggregated table in memory consisted of only two columns: the join column and a aggregated data column generated by early expression evaluation. While early expression evaluation was possible for the queries we tested with, it may not always be possible with complex queries, in which case the aggregated table may need to have multiple columns. While ColumnBurst can be trivially extended to support wider aggregated tables,

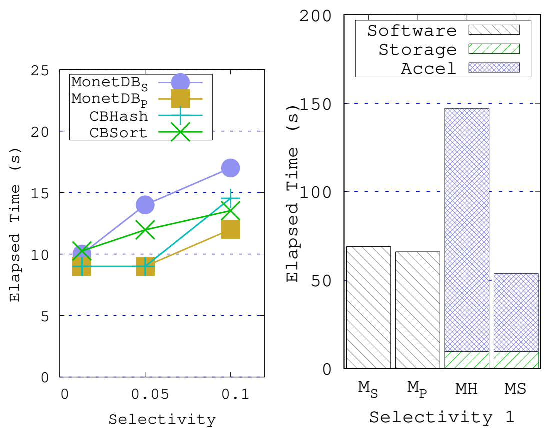

Figure 9: Elapsed time for the TPC-H query 14 with varying selectivity. Lower is better

it presents a potential performance implication on hashing vs. sorting. A large part of the performance overhead of hashing comes from the access granularity mismatch, as memory needs to be accessed in bursts, which is either 32 or 64-Byte units for LPDDR4. The performance difference between hashing and sorting may not be as pronounced if the hash elements are larger. However, if the table becomes too wide, the problem will quickly become memory bandwidth-bound, and it would be better off falling back to software with faster memory. Furthermore, the performance difference will become even more pronounced for some popular join queries where one table simply acts as the filter for the other, and only the join column needs to be kept in memory.

5 CONCLUSION AND DISCUSSION

In this paper, we have presented ColumnBurst, a memory-efficient near-storage database join accelerator, which uses a hardware implementation of a sorting-based aggregation and join algorithm to efficiently use the performance-restricted memory available on storage devices. By using algorithms with sequential access patterns on performance-restricted memory resources, ColumnBurst was able to outperform much costlier systems with fast CPUs and large amounts of memory. We are actively working on expanding ColumnBurst with more capabilities, such as an automatic query planner and column compression. We hope the discoveries demonstrated by ColumnBurst will help design future cost-effective database systems with near-storage acceleration.

ACKNOWLEDGEMENTS

This work was partially funded by NSF (CNS-1908507)

REFERENCES

[1] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)* 1, 2 (1976), 97–137.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[3] Dina Bitton and David J DeWitt. 1983. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)* 8, 2 (1983), 255–265.

[4] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[5] Transaction Processing Performance Council. (Accessed Sep 25, 2019). *TPC-H*. http://www.tpc.org/tpch/.

[6] DB-Engines. Sep 2019 (Accessed Sep 25, 2019). *DB-Engines Ranking*. https://db-engines.com/en/ranking.

[7] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[8] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. 1994. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering* 6, 6 (1994), 934–944.

[9] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 153–165.

[10] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins.. In *CIDR*.

[11] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* (2012).

[12] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1202–1213.

[13] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.

[14] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2749469.2750412

[15] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 1–12.

[16] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[17] Minho Kim. 2014 (Accessed Sep 25, 2019). *Evolutionary Migration from LPDDR3 to LPDDR4*. https://www.jedec.org/sites/default/files/Minho_SK%20hynix_CES_14_new.pdf.

[18] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. 2017. ExtraV: Boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1706–1717.

[19] MonetDB. April 2014 (Accessed Sep 25, 2019). *Citus Data cstor_fdw (PostgreSQL Column Store) vs. MonetDB TPC-H Shootout*. https://www.monetdb.org/content/citusdb-postgresql-column-store-vs-monetdb-tpc-h-shootout.

[20] MySQL. 2019 (Accessed Sep 23, 2019). *MySQL*. https://www.mysql.com/.

[21] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218.

[22] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sonmez. 2017. AxleDB: A novel programmable query processing platform on FPGA. *Microprocessors and Microsystems* 51 (2017), 142–164.

[23] Donovan A Schneider and David J DeWitt. 1989. *A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment*. Vol. 18. ACM.

[24] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. 2018. CompStor: An In-storage Computation Platform for Scalable Distributed Processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1260–1267.

[25] Satoru Watanabe, Kazuhisa Fujimoto, Yuji Saeki, Yoshifumi Fujikawa, and Hiroshi Yoshino. 2019. Column-oriented Database Acceleration using FPGAs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 686–697.

[26] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[27] Xilinx. May 2015 (Accessed Sep 25, 2019). *Xilinx 7 Series FPGA Power Benchmark Design Summary*. https://www.xilinx.com/publications/technology/power-advantage/7-series-power-benchmark-summary.pdf.