On "Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS"

Seminar: Modern Database-systems for Machine Learning and Knowledge Discovery 2023

ANTON SACHNOV

Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS [3] is a paper addressing the acceleration of data analytics engines utilizing GPUs. Specifically, the focus is on heterogeneous CPU-GPU data analytics engines, as the authors introduce their own novel hybrid data analytics engine called Mordred. The most limiting factor of GPUs in data analytics is their memory capacity. This paper delves into two primary domains of enhancement when addressing this limitation. The authors discuss the importance of data placement in a heterogeneous CPU-GPU system and afterward, they address the challenges of heterogeneous query execution. The author's approach to implementing effective data placement is by introducing a semantic-aware fine-grained caching policy, which can not only cache data at sub-column granularity but also within the data prioritizes segments estimated as more semantically relevant. To make query execution on fine-grained data like this possible, the authors introduce a heterogeneous query executor. This type of query execution can also take data from both CPU and GPU into account and uses both units in parallel during execution. Evaluation of Mordred in benchmark tests has shown, that semantic-aware fine-grained caching outperforms traditional policies and Mordred itself outperforms coexisting GPU DBMS. The next step towards optimization and improvement for this system would be a full-fledged heterogeneous query optimizer, as currently, some operations could still present a bottleneck. Furthermore, extending the research towards new hardware technologies like NVLink or CXL could also offer new opportunities due to their increased interconnect bandwidth.

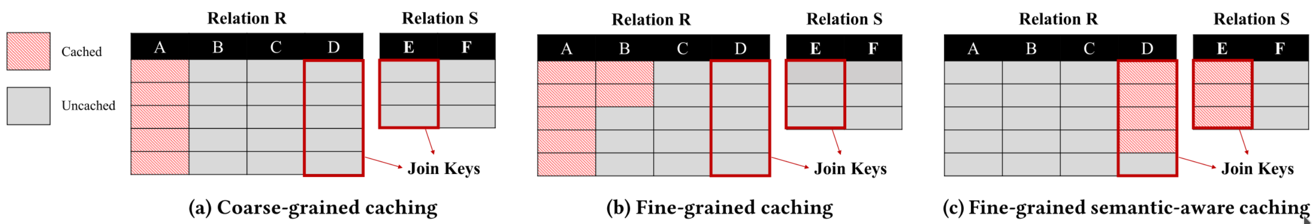
1 INTRODUCTION

While GPUs have great computational power, due to their massive parallelization ability and high memory bandwidth, their use for data analytics engines has always been limited by their small memory capacities (up to 80GB). This Constraint makes GPUs only leverageable for small data sets. While former studies have shown GPU databases to perform more than 10x faster compared to CPU counterparts, the standard approaches to mitigate the memory constraint can mostly be categorized with three strategies. The first strategy simply consists of increasing the number of GPUs, so a larger data set can be stored across multiple GPUs. This approach has the drawback of poor scalability, so while it is a perfectly fine approach in theory, it is not easily realizable as a practical or commercial solution. The second approach only employs the GPU to accelerate certain parts of the query, effectively treating it as a coprocessor. This design is not limited by the GPU memory as the main storage place of data is handled by the CPU. Transferring data to GPU on demand presents another weak link, this time it's the limitation of PCIe bandwidth being the new performance bottleneck. The third design category uses both GPU and CPU during query execution and capitalizes on data that resides in both units' main memories. The primary objective of this hybrid system is to avoid excessive data transfer between the units, thus preventing overloading of the PCIe link, while also utilizing as much of the computational power of both units as possible. Previously existing system designs have shown that implementing an efficiently running hybrid system requires additional complexity. Developing a suitable caching policy to ensure effective data placement and taking an extensive look into hybrid query execution is necessary to fully take advantage of the acceleration potential of hybrid systems. The pre-existing data analysis engines referenced in the paper had not yet implemented data placement strategies for heterogeneous query execution, or in some cases were relying upon traditional caching policies.
The research documented in this paper focuses on the author's own novel data analytics engine called Mordred and the comparative advantage it has to prior existing GPU DBMS. Apart from already existing optimizations like late materialization, operator pipe-lining, and segment skipping, Mordred's novel optimization techniques are centered around the two previously discussed issues of data placement and heterogeneous query execution. Concerning data placement, Mordred supports a semantic-aware fine-grained caching policy, which enables Mordred to store data at sub-column granularity and furthermore implements a weighted cost system, so the most profitable data for GPU accelerated query execution gets prioritized. To enable such a caching policy, Mordred has augmented the query execution, enabling it to carry out operations on subsets of columns. Another property of Mordred's query execution is heterogeneity, meaning it can separate queries into sub-queries, assign those sub-queries to either GPU or CPU, based on the data stored in their memory, and finally, execute them in parallel.

The following chapters of this paper will cover the authors' approach to data placement, specifically the algorithm by which semantic-aware fine-grained caching is conducted, and also their implementation and optimization of heterogeneous query execution. Afterward, there is going to be a summary of the original paper's evaluation and a concluding chapter pointing out the most influential aspects of the Mordred engine when evaluating it against contemporary systems.

2 DATA PLACEMENT

Mordred's data placement design is based upon pre-existing designs [2], where the whole data set is residing in CPU memory, while subsets of the main data set get mirrored and cached in GPU memory.



(a) Coarse-grained caching (b) Fine-grained caching (c) Fine-grained semantic-aware caching

Fig. 1. Visualization of different caching policies, example assumes cache size of 7 segments

This design specifically avoids disjoint data sets across both processing units, so flexibility with query scheduling can be maintained. Mirroring data sets allows for easier parallelization and for data in transfer to have lightweight representation, so the CPU is able to reconstruct data sets while in turn, the data transfer over the link is kept low. What differentiates Mordred's data placement design from pre-existing ones is the engine's caching policy, which evaluates and compares data sets on multiple factors. While previous systems relied on traditional caching policies like least-recently-used (LRU) and least-frequently used (LFU), the authors of this paper argue that primitive caching policies like lack nuance and can therefore not identify relevent data for maximum GPU acceleration potential. The paper then goes into the benefits of caching at sub-column granularity and then elaborates on the importance of caching semantically relevant data.

2.1 Fine-Grained Caching

The authors elaborate on two main constraints of coarse-grained caching, which stores data by the whole column. The first constraint is fragmentation. With GPU caches having a fixed size, storing one column may be enough to occupy enough space, so that another column would not fit in anymore, while also leaving this space unused. With the ability to cache sub-columns, caches can be filled up with subsets of other columns, allowing the caches to utilize their full capacity. The second constraint is the skewness of the column. Not every segment of a column is used with the same frequency. In practice, columns may have an uneven distribution for the hotness of their segments. To fully capitalize on the limited memory capacity of GPU caches, more frequently accessed data within columns should be prioritized, instead of storing the whole column together with the colder segments.

2.2 Semantic-aware Fine-Grained Caching

Implementation of fine-grained caching with conventional caching policies mitigates fragmentation and accounts for the skewness of data but does not yet capture data sets benefiting GPU acceleration the most, during query execution. Operators which are computationally complex and frequent during query execution, are suitable for GPU acceleration, so data participating in such operations should also be prioritized. Data segments are not only skewed when it comes to hotness but also the correlation they have with other columns. Some operations can only be executed on GPU if all of the relevant columns to the operation are also cached. If you take the join operator as an example, both join keys are required to be cached.

This property calls for a caching policy with a cost model, which considers data segments from multiple perspectives. The semantic-aware fine-grained caching policy proposed by the authors extends fine-grained LFU with weighted frequency counters, which reflect the estimated speed-up gained from caching those segments and also extends the cost estimation to semantically correlating segments.

The process of the semantic-aware weight update algorithm in Mordred consists of calling a function for each segment to update the segments' weighted frequency counters. The amount by which the frequency counter gets increased is calculated by a lightweight cost estimation function which, estimates the runtime of a given query, provided the segment in question is cached. This function is referred to as estimateQueryRuntime() in the paper and is going to be explained further in the next chapter. The query runtime estimation is then compared to the estimated runtime if the segments were not cached and the frequency counter is then increased by the difference between the uncached query runtime and the cached query runtime. Finally, for every segment which is correlated to the segment in question, the frequency counter is weighted by the same amount divided by the cardinality of the correlated segments. The authors define correlation for Mordred with three operators: selection, join, and group-by aggregation. Although it is worth mentioning, that this definition can still be extended in future developments.

2.3 Cost Model

This chapter briefly covers the idea and main principles behind the estimateQueryRuntime() function. This is a cost model which is present within the Crystal library and has been extended by the authors to support more complex queries and PCIe. The model mainly derives estimated execution time from theoretically required memory traffic. The accuracy has so far only been verified on simple memory operations, but its precision has been shown to be adequate for the purposes of a caching policy. Specific models done by the authors include filtering cost, probing runtime for hash joins, data transfer time for heterogeneous query execution over PCIe, materialization time to enable estimation of tuple reconstruction from intermediate representation, and merging time of final results from CPU and GPU. These models are all comprised of equations, which take read and write memory bandwidth, the size of input segments, cache line size, and interconnect bandwidth, to calculate estimates

Algorithm 1: Update the *weighted frequency counter* for segment *S*

1 **UpdateWeightedFreqCounter**(*segment S*)
   # estimate query runtime when S is not cached.
2 $RT_{uncached}$ = **estimateQueryRuntime**(*cached_segments* \ *S*)
   # estimate query runtime when S and segments correlated with S are cached.
3 RT_{cached} = **estimateQueryRuntime**(*cached_segments* ∪ *S* ∪ *correlated_segments*)
4 *weight* = $RT_{uncached} - RT_{cached}$
5 *S.weighted_freq_counter* += *weight*
6 **for** *C* **in** *correlated_segments* **do**
     # evenly distribute weight to all segments correlated with S
7   *C.weighted_freq_counter* += *weight* / |*correlated_segments*|

Fig. 2. Pseudocode for segment weight update function

of those sub-query element runtimes. The estimateQueryRuntime() function can divide a complete query into these models and then calculate the overall sum of the runtimes with negligible overhead.

3 HETEROGENEOUS QUERY EXECUTION

Data sets that are stored in segment-granularity introduce new complexity to query executions. Some operators cannot be executed in GPU due to the GPU memory only storing a subset of the required data for the execution. Existing systems have dealt with this problem by executing the query on GPU and then transferring the uncached data during query execution. This solution has two drawbacks, with the first being unoptimized inter-device data transfer, bringing back the PCIe bottleneck, and the second one being the underutilization of the CPU, in which cores are left unused during GPU query execution.

Mordred approach to query execution focuses on parallelism of both CPU and GPU while minimizing the required data transfer overall, as well within the memory-bound devices as well as over the interconnecting link. The goal is to capitalize on as much of the available computational power as possible and to utilize the cached data to the fullest.

3.1 Operator placement

Operator placement during query execution has been discussed in previous works, while specifically Breß et al. [1] presented a data-driven operator placement heuristic, where operators are executed in the corresponding system which stores all of the required data. If the CPU is storing the main data set, operators would only get executed in GPU if all of the required columns are cached in GPU, otherwise, the operator would get executed on the CPU. This heuristic approach has been shown to outperform operator placement on estimated costs and is more lightweight. Mordred adopts data-driven operator placement at segment granularity, meaning it can execute portions of the operator in the device containing all required input segments. Mordred can split every operator between CPU and GPU and place those operator partitions according to the location of the input segments.

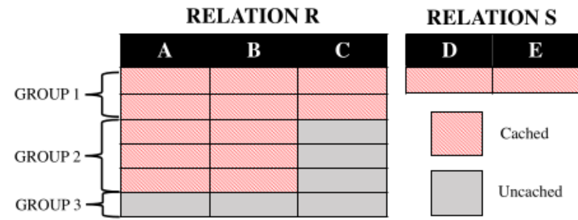


Fig. 3. Example of segment grouping

3.2 Segment-Level Query Plan

Mordred divides cached data sets into groups and executes them in parallel. To determine grouping, Mordred applies the data-driven operator placement heuristic to calculate the execution plan for each segment and puts segments with the same execution plans in a group. Operators are then fully or partially executed on those segment groups which have suitable input data, while the rest is executed in CPU. Figure 2 illustrates an example of a relation s which is fully cached in GPU, while r is only partially cached. Furthermore, there is a grouping of r which has group 1 storing two entire rows, group 2 storing three rows without column c, and group 3 is not cached in the GPU at all. Group 1 has basically a subset of the whole relation r so any operation on r which can be executed on subsets of relations, can be executed on r. Group 2 in turn could only perform operations that don't have c as a key column. Group 3 cannot be executed in the GPU, so any sub-queries with these segments would therefore be executed on the CPU.
Intermediary results from executing partial operations on segment groups are then sent back to the CPU in a lightweight representation. The CPU can then perform late materialization, reconstructing the results and merging them together with the CPU's dataset. The authors also point out that merging is typically lightweight, but with very large query results, merging could present a bottleneck. A concrete solution to this issue requires a complete heterogeneous query optimizer, which is deferred to future work by the authors.

4 EVALUATION

Mordred was evaluated by using Star Schema Benchmark(SSB), which has been widely used in previous data analytics research. The evaluation done by the authors is summarized in this chapter by highlighting these three aspects: (1) semantic-aware fine-grained caching policy and the comparative performance to traditional caching policies, (2) the performance increase gained from segment-level query execution, (3) the overall performance of Mordred as a data analytics engine compared to existing DBMS with GPU incorporation.

4.1 Caching Policy Performance on Standard SSB

In this first experiment, the authors varied the GPU cache size and evaluated the performance of different caching policies while executing SSB queries. The tested cache sizes ranged from 400 MB to 8.8 GB, with all columns which are accessed by queries fitting into an 8.8 GB cache. The query access distribution is uniform. LFU

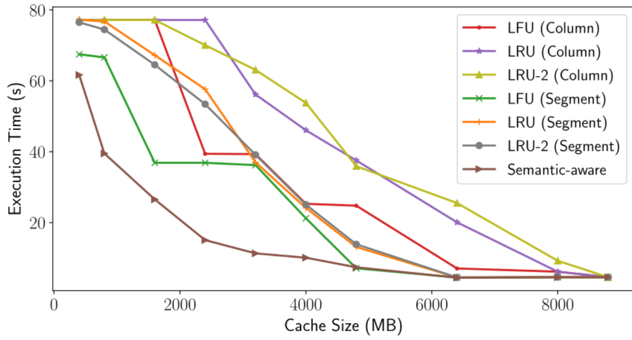


Fig. 4. Execution Time of Various Caching Policies with Different Cache Size (Uniform distribution with $\theta = 0$)

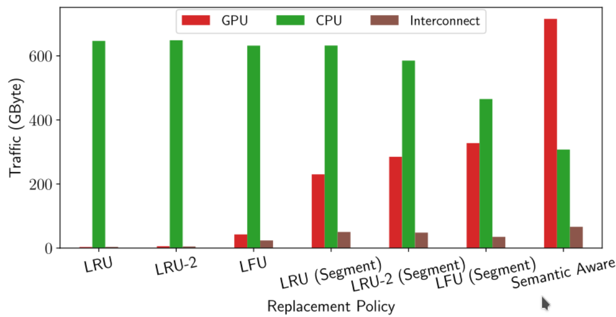


Fig. 5. Memory Traffic Breakdown for Each Caching Policy

caching policies have been shown to be faster than LRU, while the most difference is achieved by having the policy operate on a fine-grained level. Semantic-aware fine-grained caching outperformed every caching policy. Especially at small levels of cache, semantic-aware fine-grained caching heavily outperformed the other policies, due to it being the ability to identify hot data with greater precision. When investigating memory traffic in terms of traffic going through CPU, GPU, and the interconnect which is PCIe in this case, the interconnect traffic was low across all policies. The authors note that this is due to the data-driven operator placement heuristic. Semantic-aware caching had the highest GPU and interconnect traffic, and the lowest CPU traffic out of all policies. This shows that out of all other policies semantic-aware fine-grained caching utilized the GPU the most and had therefore the greatest acceleration.

4.2 Influence of Varying Query Access Pattern on Caching

This experiment tests semantic-aware caching on data sets with access skewness. The authors employed a Zipfian distribution with a tunable skewness parameter θ, to simulate non-uniform hotness in the test data sets. A larger θ indicates higher skewness. When the cache size is small fine-grained caching is more sensitive to skewness compared to column-granularity caching. The effectiveness of these policies in capturing the hot portion improves with

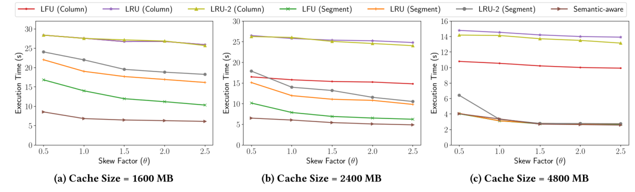


Fig. 6. Execution Time of Various Caching Policies with Varying Query Access Distribution

higher skew factors. Among the fine-grained policies, LFU generally performs better than LRU, indicating that access frequency is more effective than access timestamps for capturing skewness in query distribution. For all θ values semantic-aware caching consistently outperforms traditional caching policies. When the cache size is large the performance of fine-grained caching policies is comparable to semantic-aware caching. This is due to a bigger cache size fitting all of the hot data. However, for smaller cache sizes, the performance gap widens as only a subset of the hot portion can fit in. In such cases, semantic-aware caching can accurately identify critical segments that offer the greatest benefits from GPU acceleration.

4.3 Evaluating Segment-level Query Execution

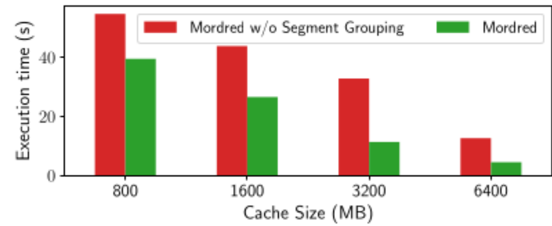


Fig. 7. Impact of segment grouping in Mordred

This chapter elaborates on the performance benefits of segment grouping for heterogeneous query execution. The conducted experiments have shown segment grouping to reduce the amount of launched kernels, as they are only launched in groups instead of for every segment individually. Segments with the same execution plan are launched with a single kernel call. Segment grouping can speed up the query by up to 3x, with the gain increasing with cache size. The authors also elaborate on merging and grouping as potential bottlenecks, with Mordred spending more time on both the larger the cache size gets. The percentage grows from 0.8% on merging and grouping compared to 99.2% of the time for execution with a cache of 0.8 GB, to 6.6% merging and grouping with 93.6% execution time at a larger cache size like 6.4GB. What is worth noting is that larger cache sizes have much faster execution. Following this analysis, the authors did not conclude merging or grouping to be performance bottlenecks.

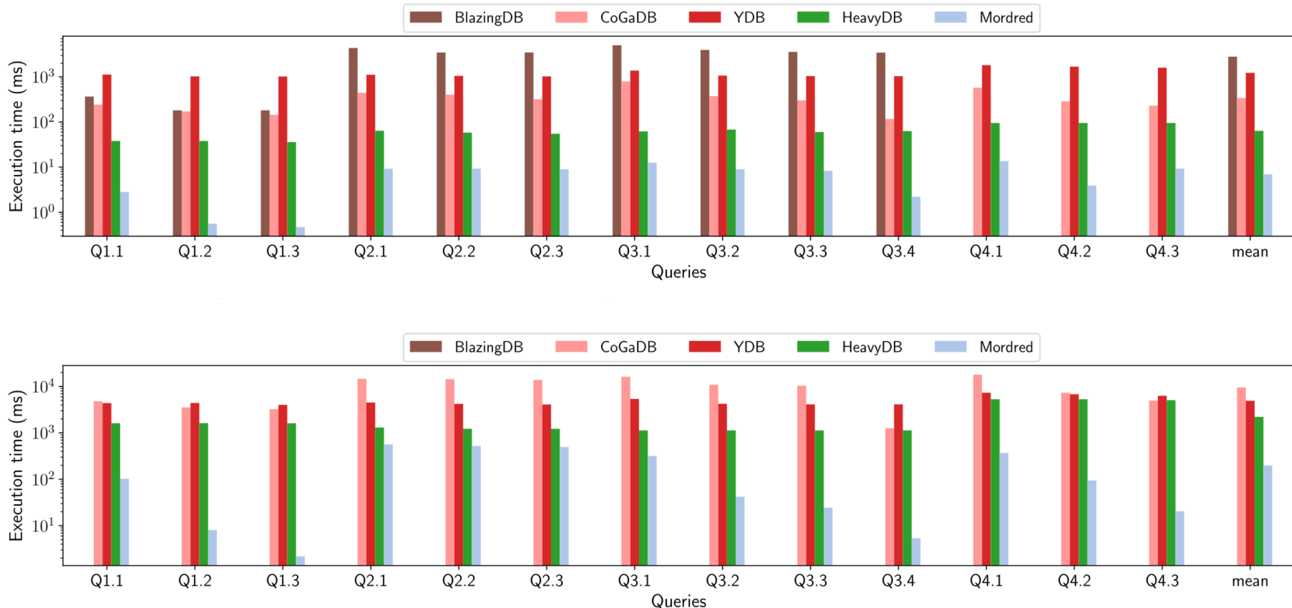


Fig. 8. SSB Query Performance of Different CPU/GPU DBMS with data fitting in GPU (top) and not fitting in GPU (bottom)

4.4 Comparison with Other CPU/GPU DBMS

To compare Mordred to contemporary systems, two sets of experiments have been conducted. The first set tests the DBMS with data that does fit in GOU, and the second set is done with data not fitting in GPU. When data fit in GPU, Mordred was outperforming every DBMS due to pre-existing optimization techniques which were implemented by the authors and optimizations specifically unique to Crystal, which enable more efficient use of GPU memory bandwidth. When data did not fit in GPU, Mordred was consistently outperforming the contemporary DBMS. According to the authors Mordred outperformed similar systems because of semantic-aware fine-grained caching, compared to contemporary systems, which did employ fine-grained caching, but with simpler LFU/LRU policies. Other DBMS were outperformed because they did not utilize fine-grained caching and therefore were affected by fragmentation. Because the caches were fragmented, the CPU was transferring data to the GPU more often resulting in suboptimal performance.

5 CONCLUSION

The main contributions of this paper to the field of CPU-GPU DBMS are data placement and heterogeneous query execution. The authors proposed semantic-aware fine-grained caching which is a caching policy specifically tailored to prioritize data that is most relevant for GPU acceleration. The other big contribution is the authors' heterogeneous query executor which has a focus on parallel use of CPU and GPU, while also leveraging the most out of the data located across both devices and operating in fine granularity. The author's novel hybrid engine Mordred implements both the caching policy and the query executor and was evaluated on Star Schema Benchmark, where it has outperformed existing Systems by multiple orders. The caching policy itself has outperformed traditional caching policies by 3x.

REFERENCES

[1] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1891–1906, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Sebastian Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[3] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. Orchestrating data placement and query execution in heterogeneous cpu-gpu dbms. *Proc. VLDB Endow.*, 15(11):2491–2503, jul 2022.