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

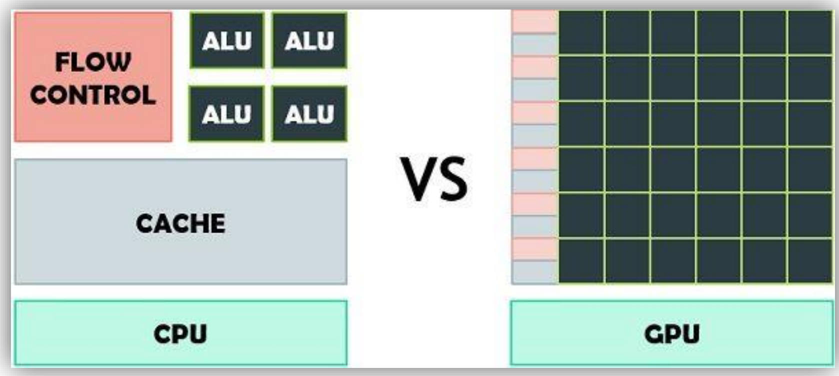
Image sources:
https://productnation.co/my/15665/best-graphics-card-gpu-malaysia/
https://developer.nvidia.com/blog/benchmarking-deep-neural-networks-for-low-latency-trading-and-rapid-backtesting-on-nvidia-gpus/

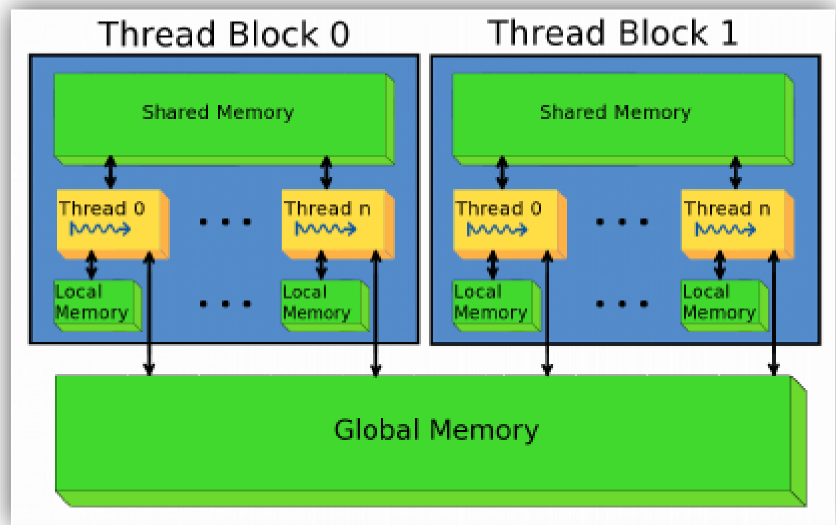
Outline

- Introduction and Background
  - Basic GPU architecture
  - GPU for data analytics
  - Previous GPU solutions
  - Heterogenous CPU-GPU DBM
  - Mordred novel hybrid CPU-GPU data analytics engine introduced by the paper
- Optimizations implemented by Mordred engine
  - Data placement: Semantic-Aware Fine-Grained Caching
  - Heterogenous query execution
- Evaluation
- Conclusion

GPU Architecture

- Global memory at bottom of GPU memory hierarchy (up to 80 GB and 2TB/s bandwidth on modern GPUs)
- Most basic compute unit: streaming multiprocessors (SM)
- One SM has multiple cores with access to same shared memory (SMEM)
- the L1 and L2 caches/access global memory
- L1 cache is local to an SM and the L2 cache is shared by all SMs





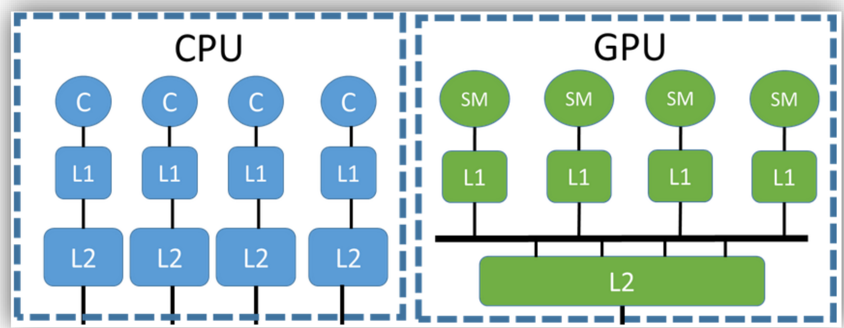


Image sources:
https://techdifferences.com/difference-between-cpu-and-gpu.html
https://www.researchgate.net/figure/Standard-GPU-memory-hierarchy_fig4_271134412
https://www.bsc.es/research-development/research-areas/computer-architecture-and-codesign/memory-hierarchy-gpu

GPU for Data Analytics

- Potential for acceleration:
  - massive parallelism
  - high memory bandwidth
  - more than 10× speedup over the CPU counterparts
- Main limitation:
  - small memory capacity
  - only small workloads fit in and can then be accelerated

Mitigating Memory Limitation

- GPU is primary execution engine
  - Working sets are stored in one or multiple GPUs
  - multiple GPUs for larger aggregated memory
- GPU as a coprocessor
  - data resides on the CPU
  - transferred to GPU on demand during query execution (GPU as accelerator)
  - systems do not suffer from limited GPU memory capacity
  - Limited bandwidth on PCIe => another bottleneck
- Heterogeneous CPU-GPU query execution
  - CPU and GPU are both used in special query execution
  - Partial execution on CPU avoids excessive data transfer to GPU
  - Focus of this paper (Mordred data analytics engine)

Data Placement and Query Execution in Mordred

- Data Placement
  - CPU maintains a copy of the entire database, subset of data cached in GPU memory
  - semantic-aware cache replacement policy
    - Fine granularity caching
    - cost based performance model estimates benefit of caching

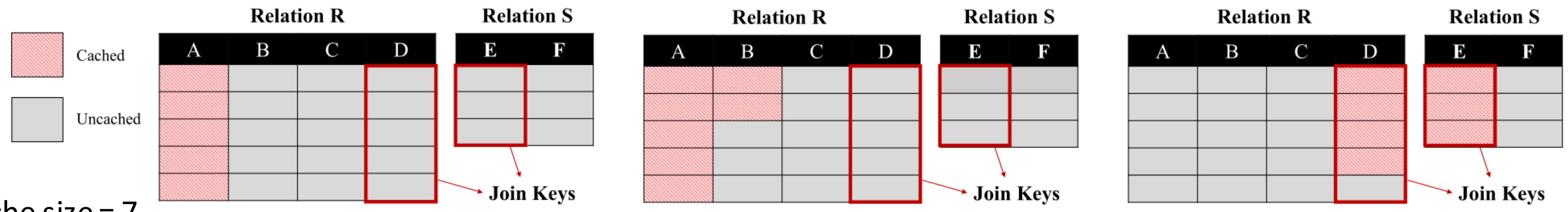
- Heterogeneous Query Execution
  - segment-level query plan allows for fine-grained heterogeneous execution
  - Other general optimization techniques (late materialization, operator pipelining etc)

Data Placement in Mordred

- Mordred maintains a copy of all data in CPU
- No disjoint datasets compared to alternatives
- Flexible query scheduling
  - CPU can process queries when GPU can't
  - CPU can reconstruct results => reduce PCIe traffic

Data Placement: Fine-Grained Caching

- Previous LRU and LFU replacement policies are not optimal for GPU acceleration
- Problem is caching at column granularity
  - Fragmentation
  - Does not capture access skewness
  - Hotter sub-column data cannot be prioritized in caching



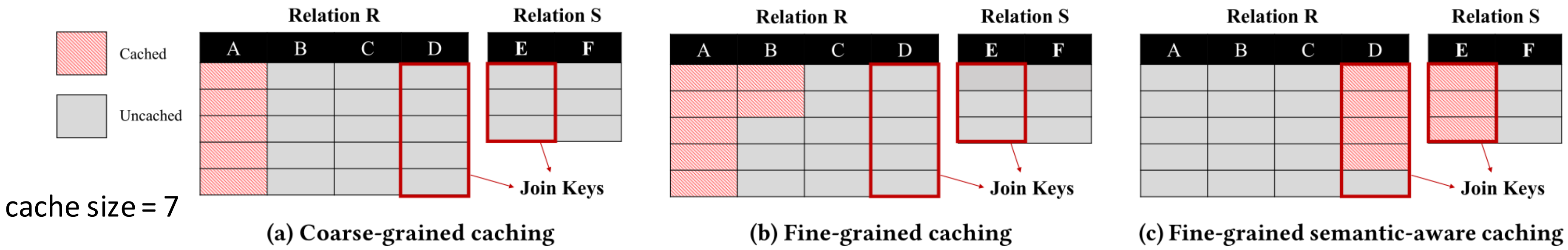
cache size = 7

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

Data Placement: Semantic-Aware Caching

- Sub-column LRU/LFU cannot identify data benefiting most from GPU
- Consider correlation between multiple columns when caching
  - Join needs both keys cached etc.

- Extend LFU with weighted frequency counters



Cache Replacement Policy

- Cost model captures:
  - relative speedup of caching a segment
  - correlation among segments from different columns
  - Correlation depends on the performed operator (selection, join, and group-by aggregation)

- estimateQueryRuntime()
  - Simple model to predict runtime
  - assumption that the CPU/GPU memory and PCIe bandwidth are the performance bottleneck

---

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

UpdateWeightedFreqCounter(*segment S*)

  # estimate query runtime when S is not cached.

  $RT_{uncached}$ = **estimateQueryRuntime**(*cached_segments* $\setminus$ *S*)

  # estimate query runtime when S and segments correlated with S are cached.

  RT_{cached} = **estimateQueryRuntime**(*cached_segments* $\cup$ *S* $\cup$ *correlated_segments*)

  weight = $RT_{uncached} - RT_{cached}$

  S.weighted_freq_counter += *weight*

  for *C* **in** *correlated_segments* **do**

    # evenly distribute weight to all segments correlated with S

    C.weighted_freq_counter += *weight* / |*correlated_segments*|

Cost Models: estimateQueryRuntime()

- Derives execution time mostly from assumed memory traffic
- Model has only been verified on simple operators
- Mordred extends model to more complex queries and to support PCIe
- Example: Filtering cost

$$\mathit{filter\ runtime} = \frac{size(int) \times N}{B_r} + \frac{size(int) \times N \times \sigma}{B_w}$$

N = |input segments|
σ = selection predicate
Br = read memory bandwidth
Bw = write memory bandwidth

Heterogenous Query Execution

- fine-grained caching adds extra complexity of query execution
  - possible that only subset of data required by operator exists in GPU memory
  - Existing systems with fine-grained caching still execute entire query on GPU, transfer uncached data to GPU during execution

- Goals of Mordred query execution:
  - Minimize inter-device data transfer
  - Minimize CPU/GPU memory traffic
  - Fully exploit parallelism in both CPU and GPU

Operator Placement

- Previously: Data driven operator placement heuristic
  - operator is executed in GPU only if all input columns are cached in GPU
- Mordred applies this at sub-column granularity
  - executes portions of the operator in the device where input segments reside
  - Single operator can be split to run in both CPU and GPU

Segment-Level Query Plan

- Mordred groups segments and executes them in parallel
  - Grouping of segments is based on data-driven operator placement heuristic
  - Segment groups are then executed in parallel
  - After execution finish all results are sent back to CPU merged

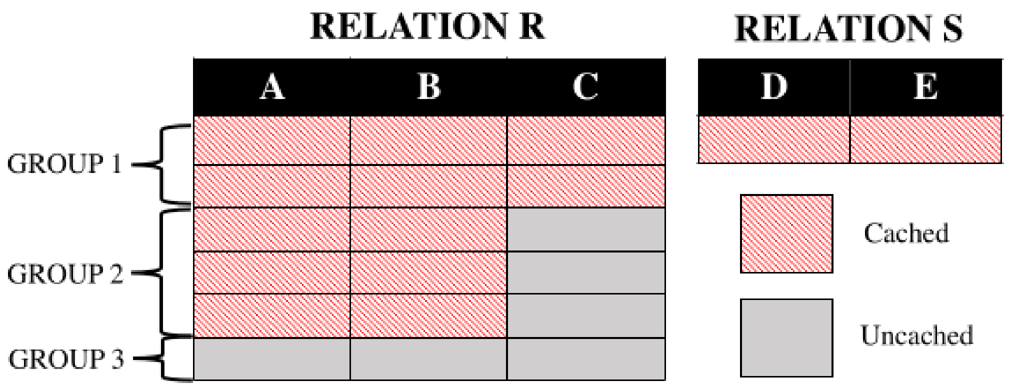


Figure 2: Example of Segment Grouping.

4.1.3 Example of Query Execution.

```
Q0: SELECT S.D, SUM(R.C) FROM R,S
    WHERE R.B = S.D AND R.A > 10 AND S.E > 20
    GROUP BY S.E
```

Evaluation: Caching Policy

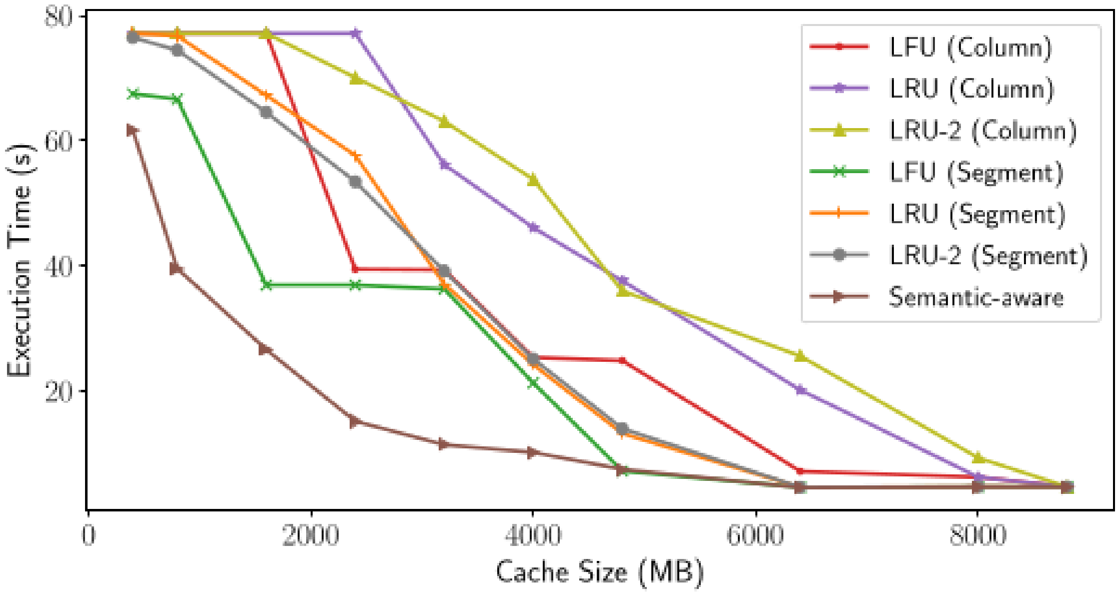
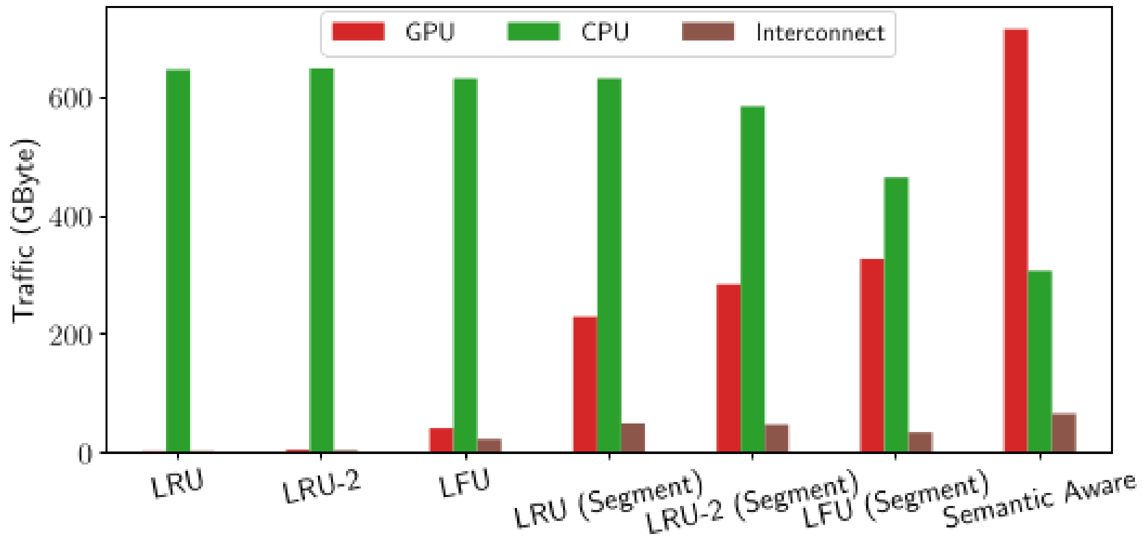
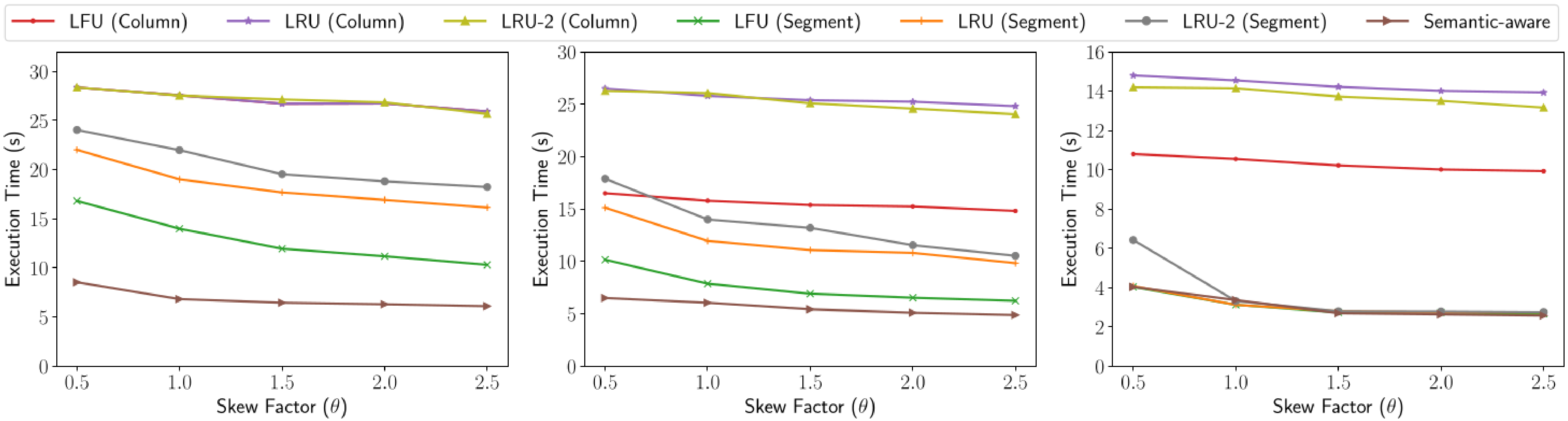


Figure 5: Execution Time of Various Caching Policies with Different Cache Size (Uniform distribution with $\theta = 0$)



Cache can hold 20% of accessed collumns

Figure 6: Memory Traffic Breakdown for Each Caching Policy



(a) Cache Size = 1600 MB **(b) Cache Size = 2400 MB** **(c) Cache Size = 4800 MB**

Figure 7: Execution Time of Various Caching Policies with Varying Query Access Distribution

Comparison with Other CPU/GPU DBMS

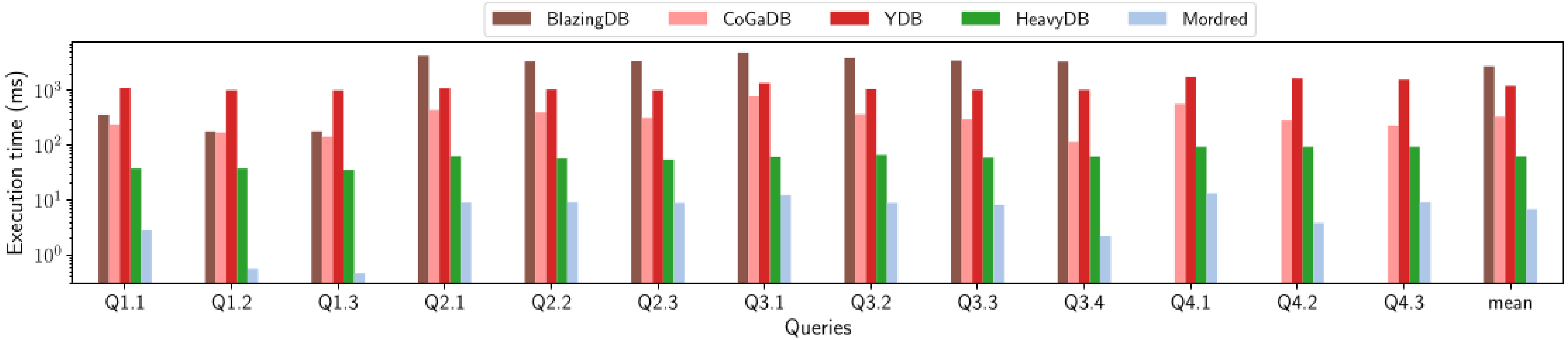


Figure 13: SSB Query Performance of Different CPU/GPU DBMS (Data fits in GPU)

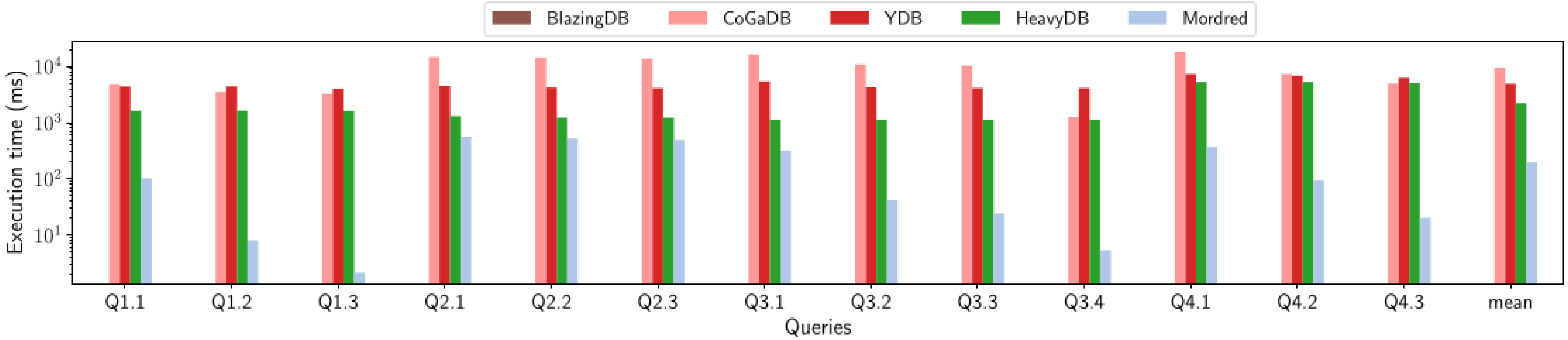


Figure 14: SSB Query Performance of Different CPU/GPU DBMS (Data does not fit in GPU)

Conclusion

- Two main contributions:
  - Data placement
    - introduce semantic-aware fine-grained caching policy
  - Heterogenous query execution
    - can fully exploit data in both devices
    - coordinate query execution at a fine granularity
- Evaluation:
  - semantic-aware caching policy manages to outperform the best traditional caching policy by 3×
  - Mordred manages to outperform existing GPU databases by an order of magnitude