UNIVERSITY OF SOUTH FLORIDA
COLLEGE OF ENGINEERING

CheetahDB®: A System for High-Throughput Database Processing on GPUs

Hao Li,† Chengcheng Mou,† Napath Pitaksirianan,† Ran Rui,† Zhila Nouri-Lewis,† Mehrad Eslami,† Ruoya Sheng,✿ Shan Lei,✿ Jing Wang,✿ and Yicheng Tu†✿

† Department of Computer Science and Engineering, University of South Florida, Tampa, FL, USA
✿ Cheetah Data Systems, Inc., Tampa, FL, USA

Cheetah Data Systems

Abstract

GPU database has been an active topic in academic research as well as industrial practice. However, existing systems have not shown significant performance advantages over CPU-based in-memory DBMSs. We argue that two main factors contributed to such difficulties: (1) the CUDA programming model, by focusing on HPC-type workloads, requires non-trivial basic research to address the many technical challenges in developing a DBMS system software; (2) I/O bottleneck between host and GPU offsets the performance gain of on-board query processing.

CheetahDB is a high-performance in-memory DBMS generated from NSF-supported research at the database group in University of South Florida (USF) and commercialized by Cheetah Data Systems, Inc. CheetahDB addresses the above challenges via a complete rethinking of the software architecture of a DBMS under today's multi-core hardware environment. Specifically, we redesigned and optimized a multitude of DBMS components such as relational operator processing, query optimizer, query executor, buffer management, data indexing, and resource allocation. To address the I/O bottleneck issue, our query processing model minimizes data transmission between host and decide, maximizes overlap between computation and I/O, and more important, adapts a novel multi-query optimization scheme to optimize resource sharing among the workload level. Putting all efforts into one system design, CheetahDB delivers query processing performance at least one order of magnitude higher than competing systems, CPU-based or GPU-based. We believe our work ends the debate *whether GPUs are advantageous over CPUs in processing database workloads* with a definite "yes."

Introduction of CheetahDB

- In-memory DBMS, columnar storage
- Standard SQL query interface (will support up to SQL:1999)
- Optimized for OLAP and Data Stream Processing scenarios
- Research funded by NSF (Career Award to Yicheng Tu)
- Commercialization started in 2018
- IP suite: One PCT, three US patents in the pipeline, more to come

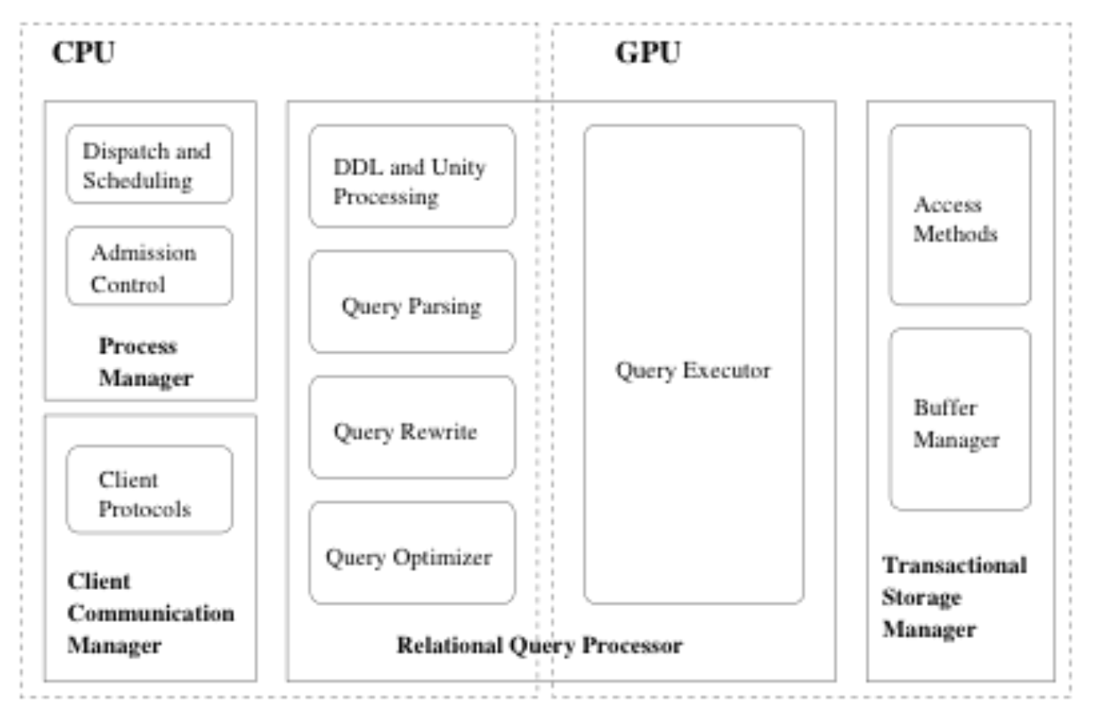

Fig. 1 A sketch of the CheetahDB Architecture

CheetahDB Design Philosophy

- Balance the core and I/O utilization: feed the CUDA cores with more work!
  - Optimize towards a workload rather than a single query (Fig. 8)
- If data must be transmitted, overlap it with in-core computation
- Concurrent kernel processing is inevitable in DBMSs: optimize the parameters!
- Still reduce the total amount of work: use indexes!
- CUDA does not support OS-type functionalities! Find ways to deal with them.
- Memory allocation not flexible in CUDA? Use a page-based buffer pool
- Kernel development: need all the tricks we learned in CUDA programming

Relational Operators in CheetahDB: Join [3]

- Efficient sort-merge join and hash join (Fig. 2) implementations
- Maximize hardware utilization by various optimizations (Fig. 3)
- Sophisticated designs to achieve load-balance
- Support out-of-core processing with data larger than GPU memory (Fig. 4)
- Support multiple GPUs working cooperatively (Fig. 5)

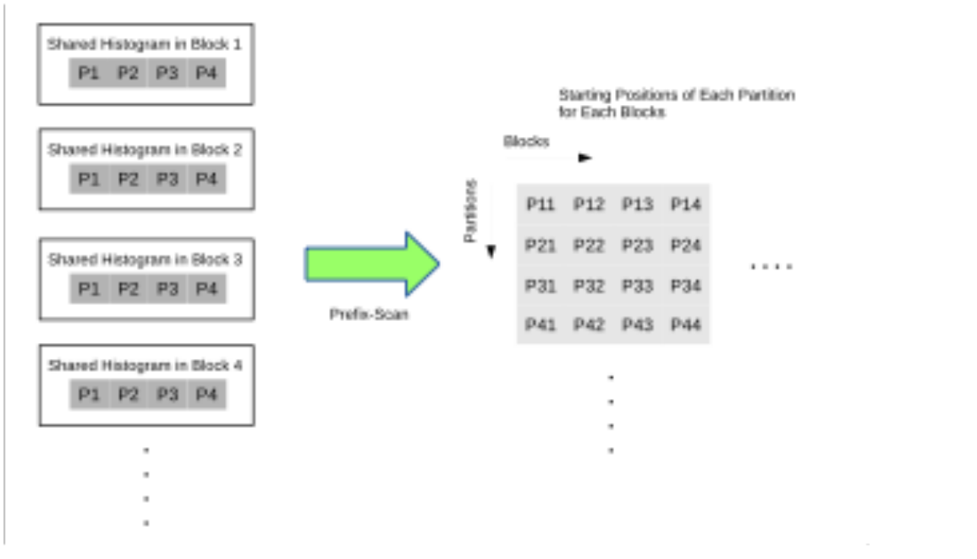
Shared Histogram in Block 1
Shared Histogram in Block 2
Shared Histogram in Block 3
Shared Histogram in Block 4
Starting Positions of Each Partition for Each Blocks
Prefix Scan

Fig. 2 Shared histogram – a key idea in our design of Partitioning and Reordering in GPU hash join

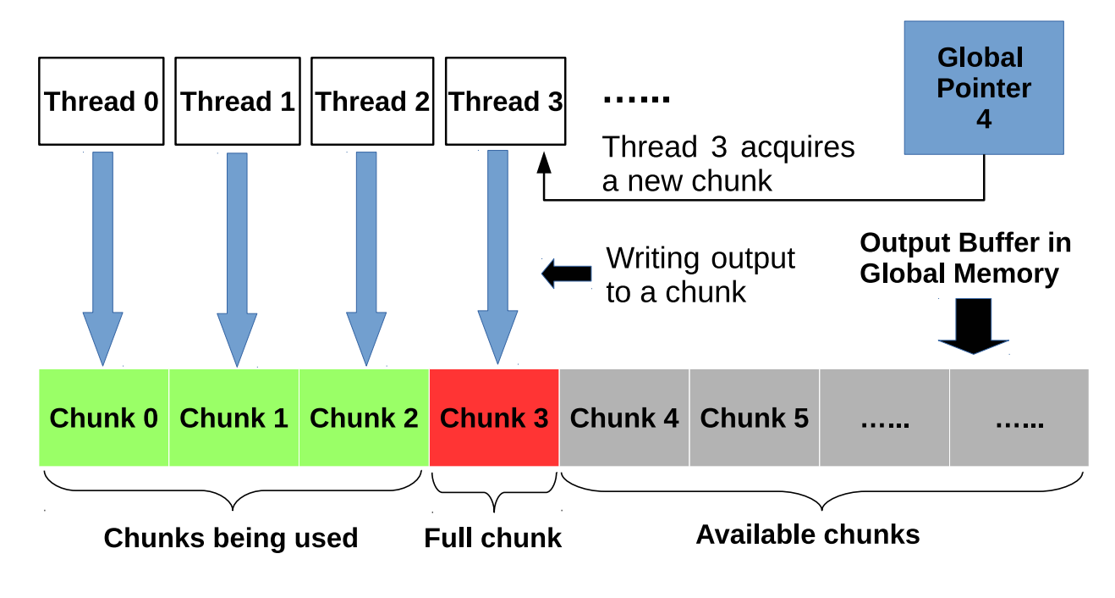

Fig. 3 Output data is acquired from a buffer pool instead of allocated beforehand, this solves the problem of output with sizes unknown

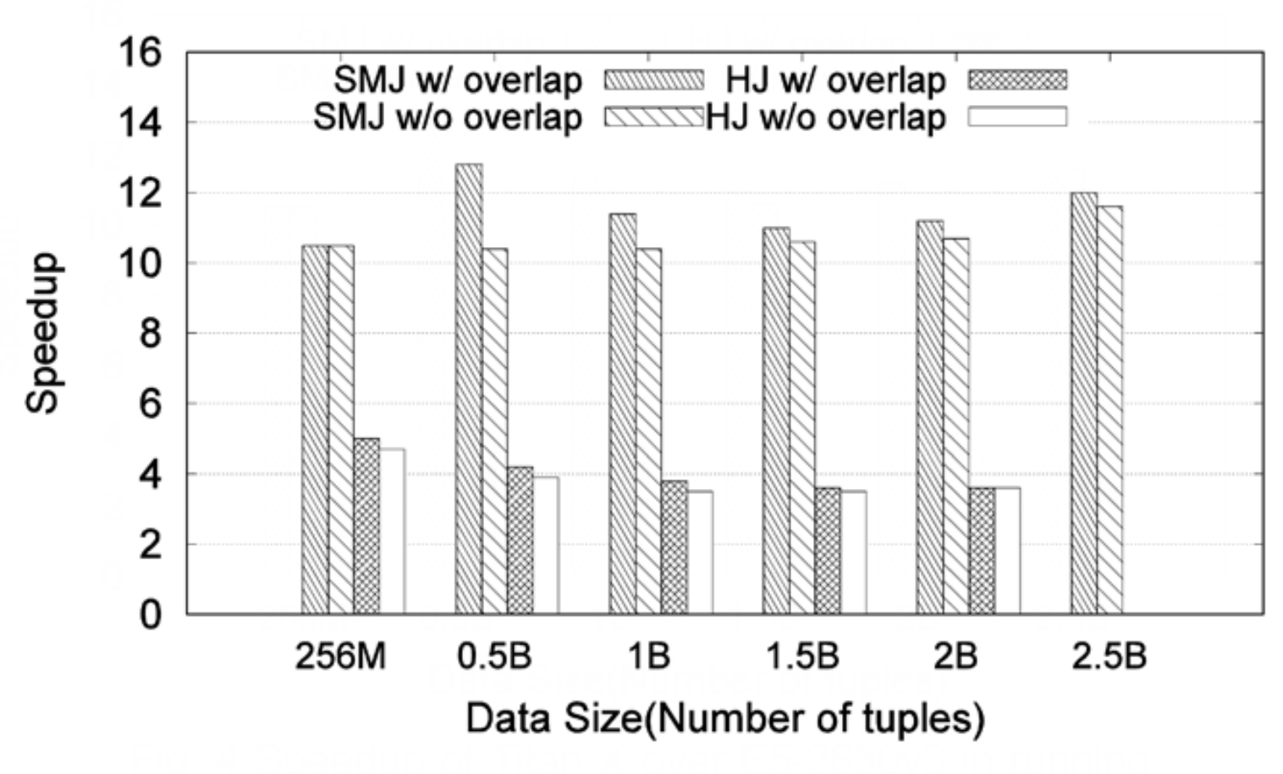

Fig. 4 Speedup of Titan X over E5-2630v3 in running join code with large tables. SMJ: sort-merge join; HJ: Hash Join

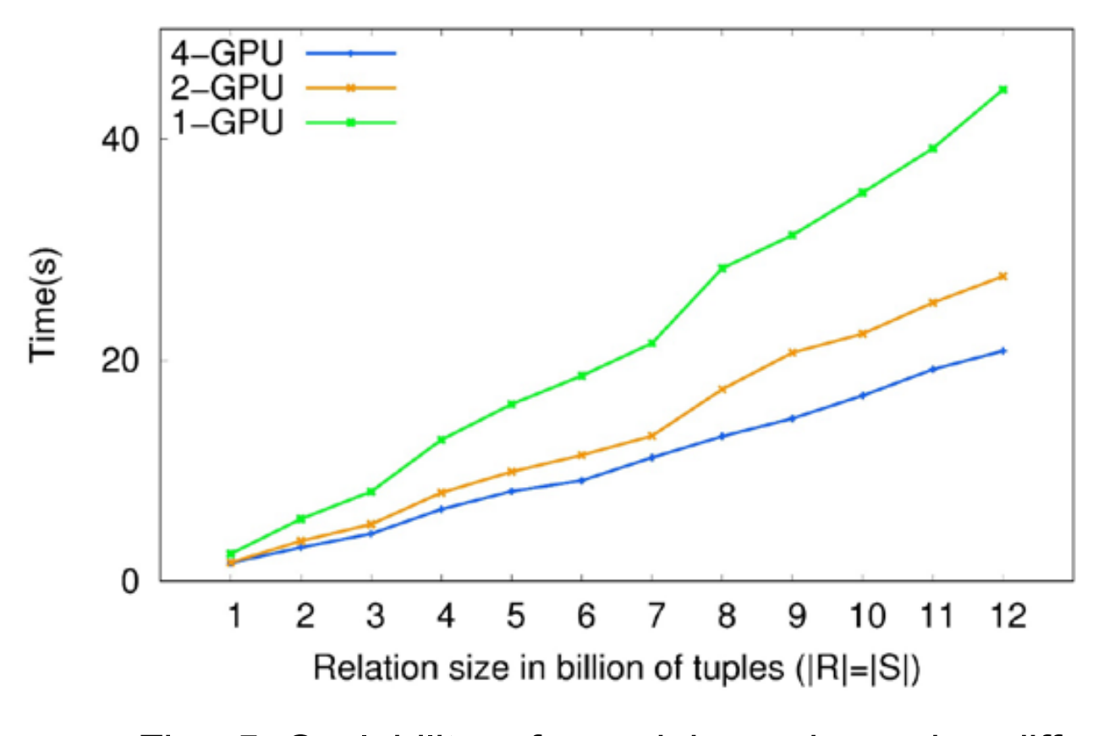

Fig. 5 Scalability of our join code under different number of GPU cards.

Relational Operators in CheetahDB: Group-By/Aggregates

- Sorting-based and hash-based parallel Group-By kernels
- Optimized design using shuffle instructions and multi-run grouping method
- Group-By supports popular data types (i.e., int, long, double float, char, etc.)
- Aggregates (count, average, sum, min, and max) are integrated along with Group-By kernels for more efficient I/O
- Group-By queries are processed up to 4 times faster than a similar system from Company Y – a leading GPU database company based in silicon valley (Fig. 6)
- Composite query (join + Group-By + aggregate) results are not obtained – company Y's system delivered incorrect results
  - Fig. 7 shows the Join-only results

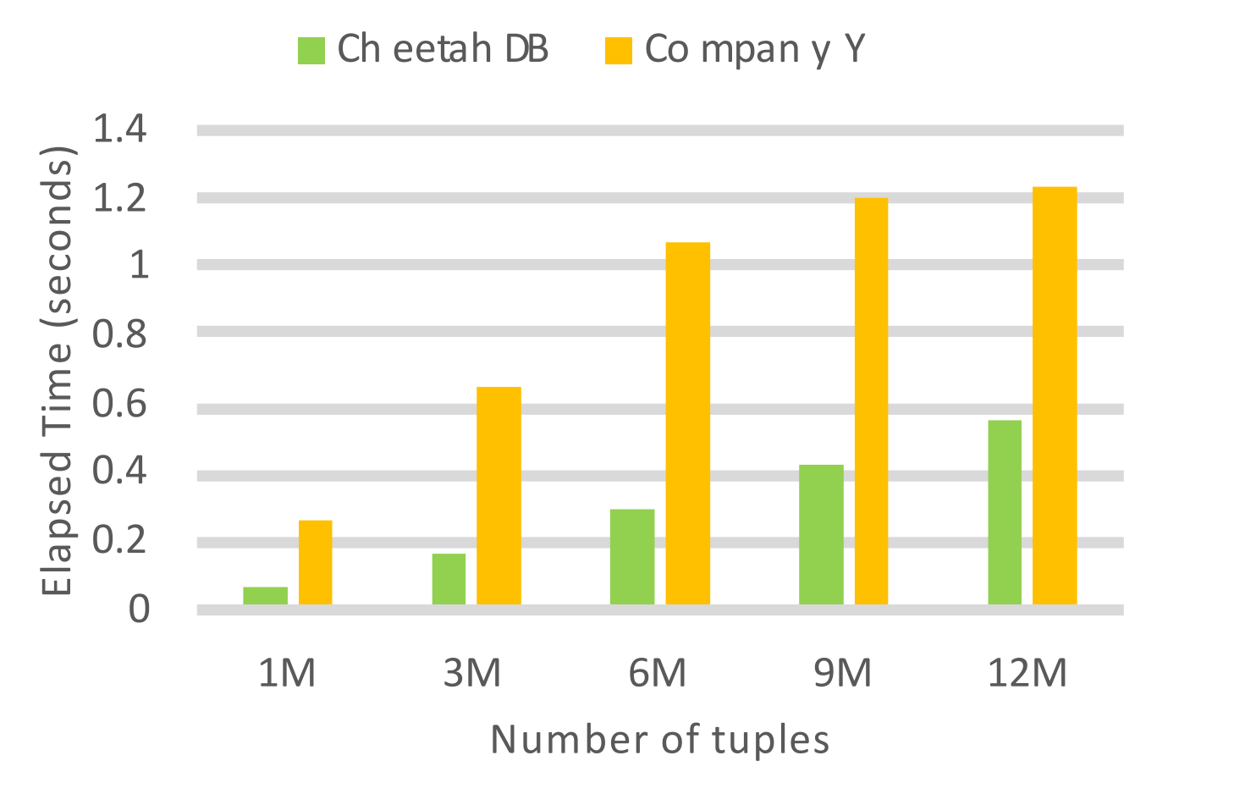

Fig. 6 Performance comparison of group-by + aggregate queries) between CheetahDB and Company Y, the data type of group-by column is 64-bit integer, and aggregate column is float

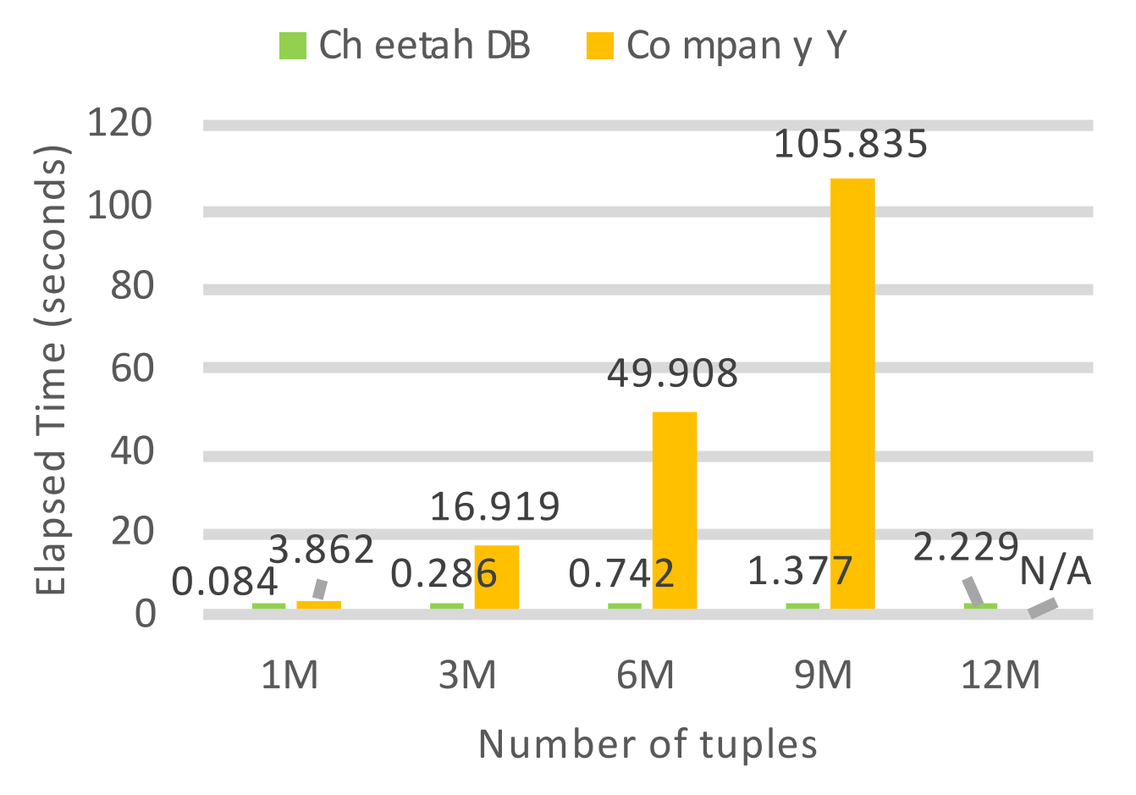

Fig. 7 Performance comparison of running queries Joining two tables between CheetahDB and Company Y

Parallel index-based search with B+-tree

- Support key-search and range-search operation
- Suitable for running large number of concurrent (search) queries (Fig. 8)
- We build the GPU B+-trees by our dynamic allocator and maximize the performance of the GPU index-based searching operation by preprocessing the queries
- We preprocess the queries by grouping similar queries and assigning each group to a CUDA block
- The results show that the searching time is up to 7.4 times from the best multi-core CPU index-based searching implementation

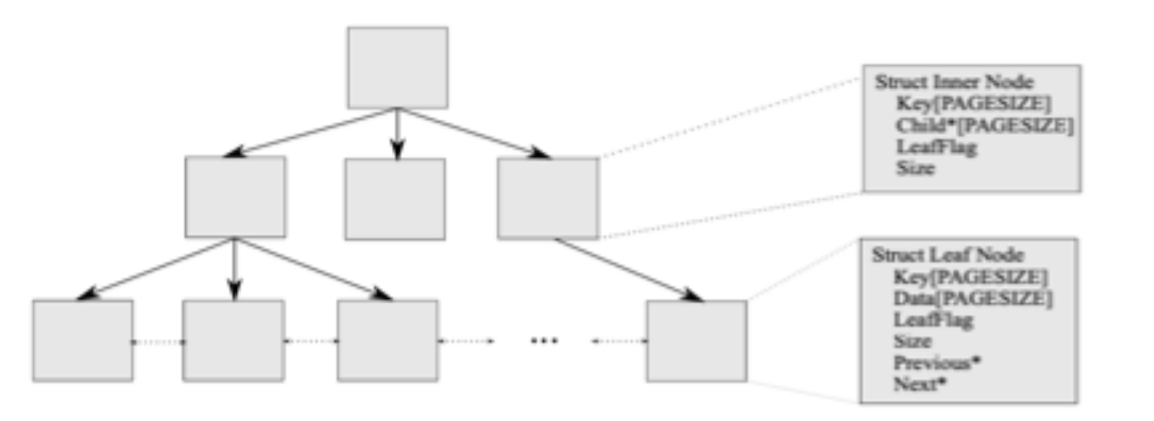

Fig . 7. Tree construction procedure in GPUs, each node is stored in a separate data page

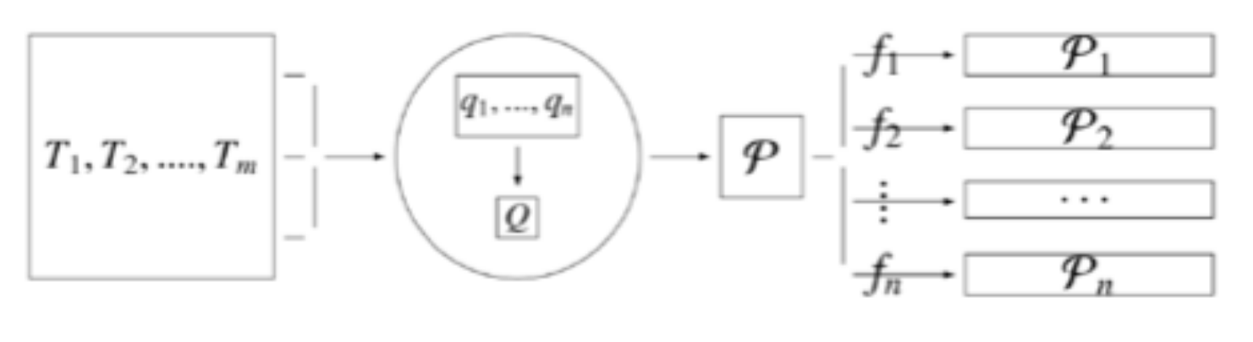

Fig . 8. The CheetahDB concurrent query processing system model. $T_1, T_2, \ldots, T_m$ are input tables, $p_1, p_2, \ldots, p_n$ are output tables for the n queries

G-PICS: An Extensible Module for Building Spatial Trees [2]

- Supports all regular space partitioning tree types: quad/oct-tree, kd-tree
- Data-driven threading to achieve high parallelism
- Efficient query processing algorithms: point search, range search, within distance, k-NN, spatial joins
- Support dynamic data: efficient tree updates with cost proportional to data dynamics
- Multi-GPU support

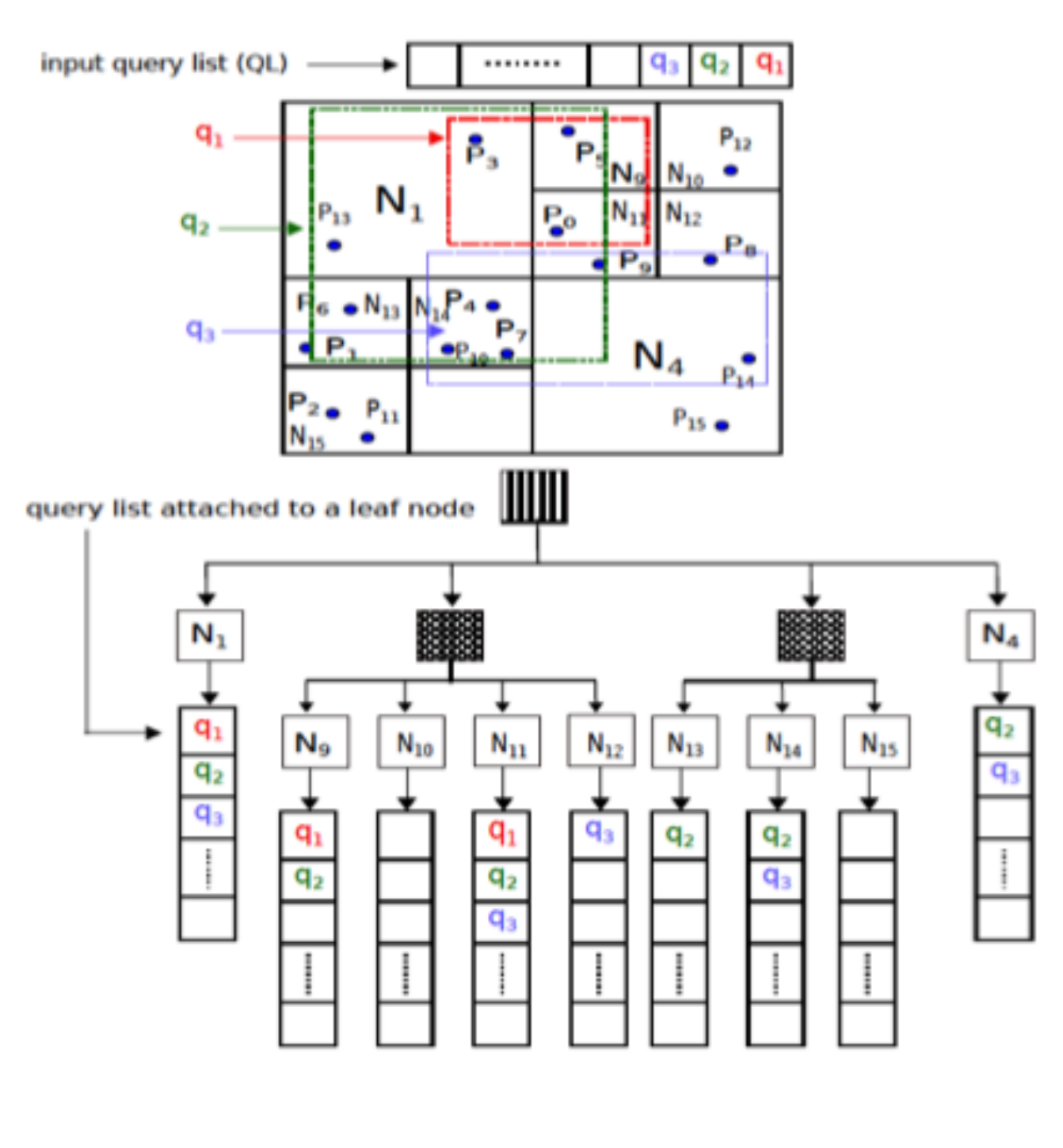

Fig. 9. Example of a quad-tree built by G-PICS and one step in query processing

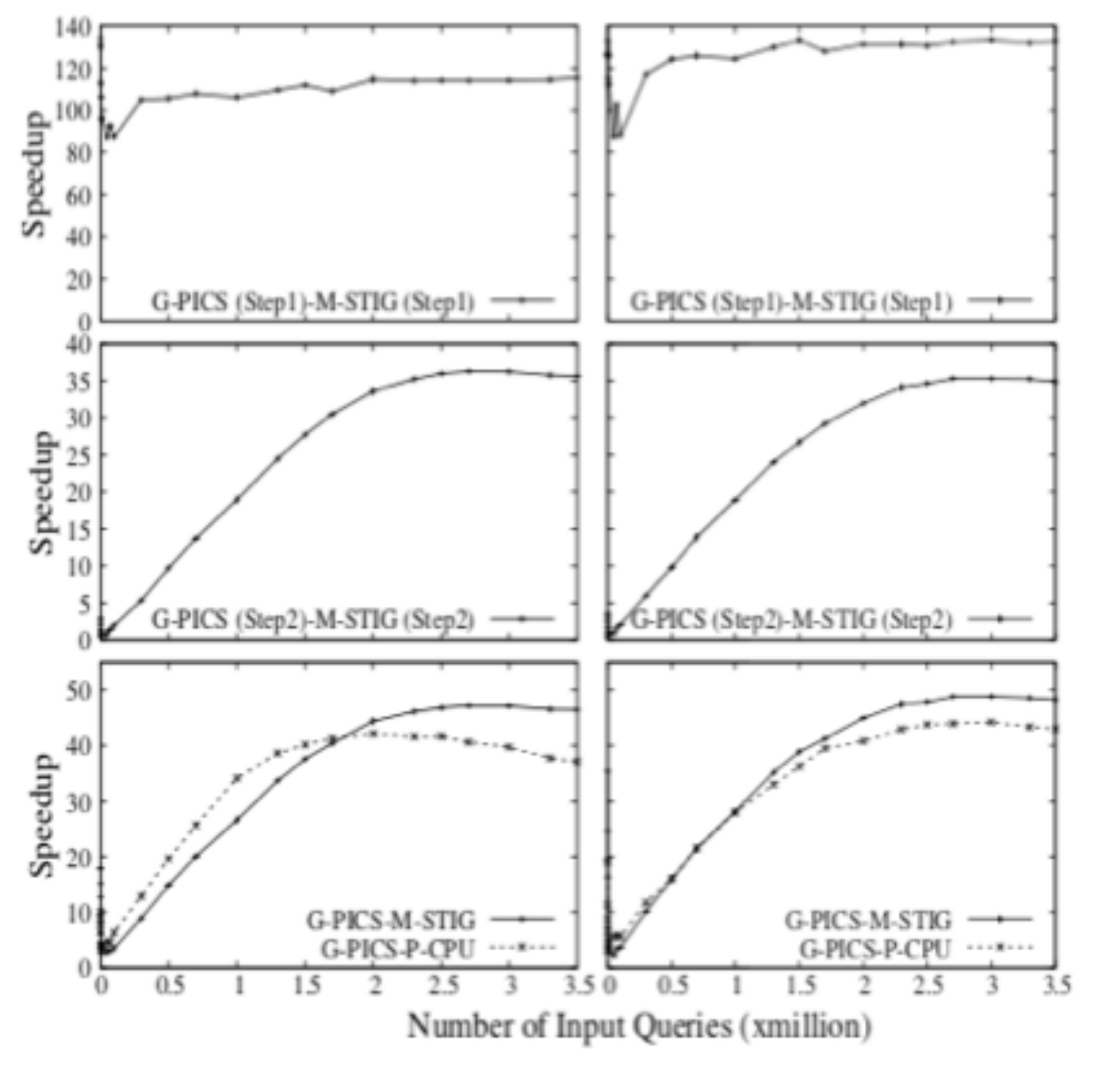

Fig. 10. Performance of G-PICS Spatial Query Processing over best-known GPU code (M-STIG) and highly optimized CPU code (P-CPU)

Resource Allocation Among Multiple Kernels [1]

- Concurrent processing of kernel is both necessary (i.e., multiple queries) and feasible (CUDA Streams)
- Aims at optimal parameter configuration for launching kernels
- Identify the GPU resources that related to execute multi-queries concurrently
- Formulate the resource allocation problem to a two-stage mixed programming model
- Exact and Heuristic Algorithms are developed to solve the models efficiently

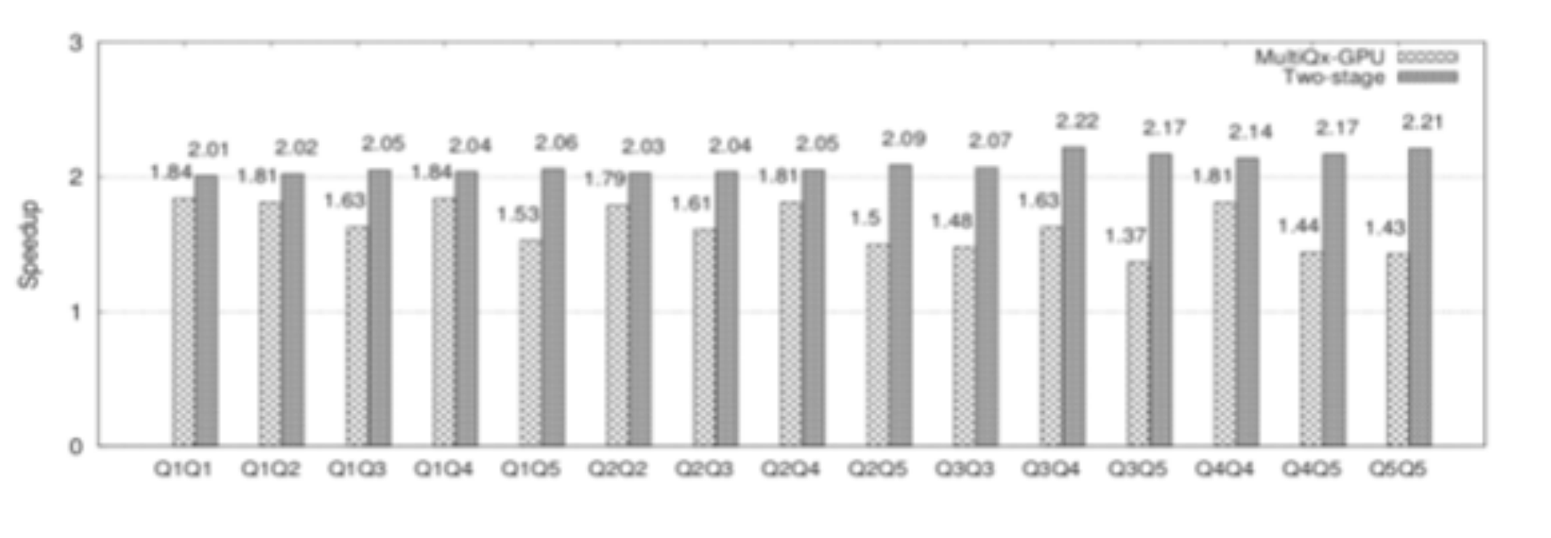

Fig. 11. Speedup of two query combinations that MultiQx-GPU [5] Optimization and Two-stage Model over sequential solution

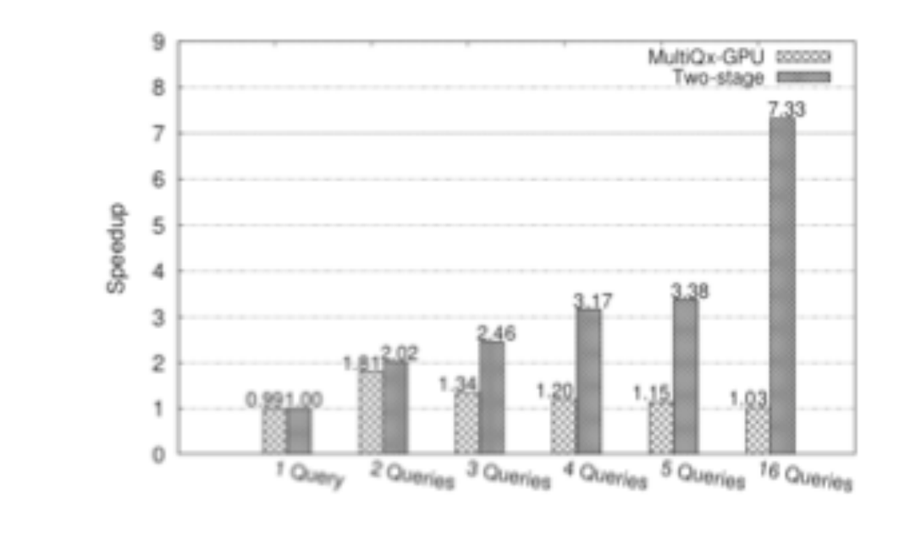

Fig. 12. Speedup of different number of queries that *MultiQx-GPU* Optimization and *Two-stage Model* over sequential solution

Experimental Comparison to Competing Systems

- Results of two sets of experiments, each set highlighting the effectiveness of our two major innovations: novel GPU-based DBMS and combined query processing

Single-Query Performance:

- Record the running time of the four most popular queries identified in a real-world financial database supporting risk management applications
- Compare CheetahDB with the most popular GPU databases from Company Y and Company Z, as well as a mainstream CPU-based in-memory database system from Company X
- Outperforms the other three systems in all queries under different table sizes

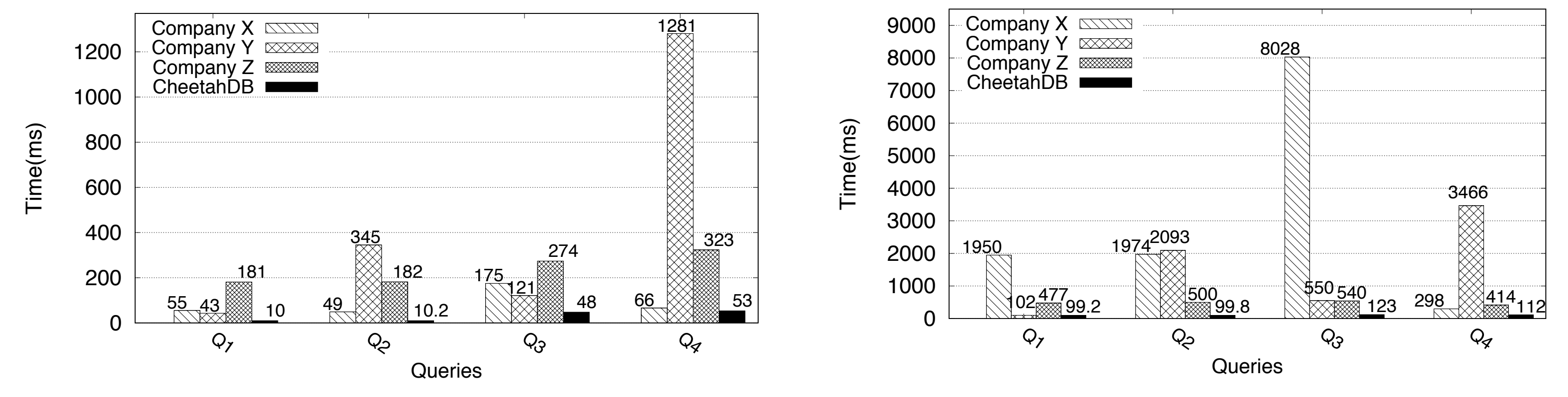

Fig. 13 . Performance of CheetahDB prototype over competitors under two sizes of the facts (driver) table (Left: 500K records; Right:10M records). The database design consists of a star-shaped structure with four tables, we fix the size of two tables to be 4M and 200K records, respectively

Performance in Workload Processing:

- Queries are generated from the TPC-H benchmark under three database sizes: SF1, which is an in-memory database setup; SF100, which is a traditional disk-based database due to its large size; and SF10, which can only be partially put into memory
- Compare CheetahDB with Company M, a record keeper in many TPC-H test results
- Outperforms Company M in all cases, with a speedup up to 36X

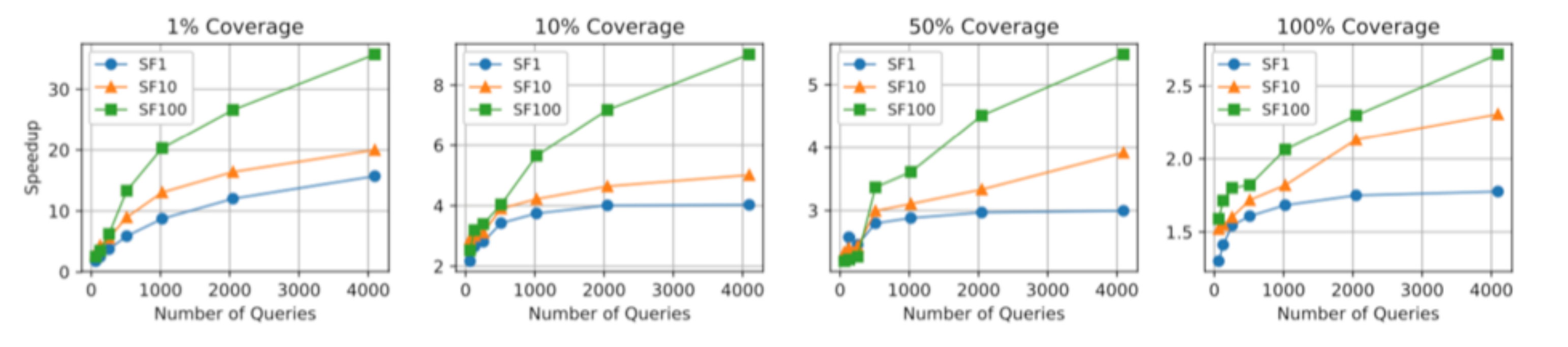

Fig. 14. .Speedup of CheetahDB over Company M under different database sizes and query numbers

Conclusions

- Existing GPU-based databases fall short in efficient query processing
- The CheetahDB approach and system fills the performance gap via novel DBMS architecture suitable for GPUs
- CheetahDB can be an order of magnitude faster than any existing in-memory DBMS

References

[1] H. Li, Y. Tu, and B. Zeng, 2019, Concurrent query processing in a GPU-based database system. *PloS one* 14.4 (2019)
[2] Z. Nouri and Y. Tu. GPU-Based Parallel Indexing for Concurrent Spatial Query Processing. SSDBM 2018.
[3] Ran Rui and Yi-Cheng Tu. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM '17).
[4] A. Shahvarani and H. Jacobsen. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. SIGMOD 2016
[5] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. 2014. Concurrent analytical query processing with GPUs. Proceedings of the VLDB Endowment 7, 11 (2014), 1011–1022.