Microarchitectural Analysis of Graph BI Queries on RDBMS

Rathijit Sen
Gray Systems Lab, Microsoft
Redmond, United States
rathijit.sen@microsoft.com

Yuanyuan Tian
Gray Systems Lab, Microsoft
Mountain View, United States
yuanyuantian@microsoft.com

ABSTRACT

We present results of microarchitectural analysis for LDBC SNB BI queries on a relational database engine. We find underutilization of multicore CPUs, inefficient instruction execution, data access overheads at the on-chip cache hierarchy, data TLB overheads, and overall low (but short-term high) memory bandwidth utilization. Using huge pages increased query performance by up to 65% and workload performance by 23%.

ACM Reference Format:
Rathijit Sen and Yuanyuan Tian. 2023. Microarchitectural Analysis of Graph BI Queries on RDBMS. In *19th International Workshop on Data Management on New Hardware (DaMoN '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3592980.3595321

1 INTRODUCTION

The high demand for graph technologies in the market has been fueled by the rapid growth of network and graph data, resulting in the emergence of a plethora of graph databases [44]. Besides native graph databases, like Neo4j [31] and TigerGraph [11], which are built from scratch just for handling graph workloads, many argue that relational database systems (RDBMSs) can well support graph workloads [12, 48]. In fact, many graph database solutions on market today, including Microsoft SQL Graph [29], Oracle Spatial and Graph [33], and IBM Db2 Graph [45], provide graph query capabilities on top of an existing relational database. There are also many research endeavors in efficiently supporting graph queries on top of RDBMSs [13, 20, 43], like VoltDB [10], DuckDB [34], etc. The counter argument from the native graph database camp is: while it is technically possible to run graph queries on an RDBMS, it may not be the most efficient or effective way to handle graph data.

As discussed in [44], settling the argument of native vs RDBMS-based graph databases is at least challenging if not impossible. Beyond performance, many factors, such as the composition of graph and non-graph workloads in the application, the data transfer and transformation cost in the end-to-end data pipelines, and the actual design and implementation of the system, also contribute to the choice of a particular graph database solution for an end user. Nevertheless, it is still interesting and important to understand in a performance perspective how efficient graph queries are executed in a relational database. There has been a number of experimental studies benchmarking various graph databases [21, 23, 26], including some RDBMS-based graph databases. All of these studies were performed at a high level, measuring latency and throughput of graph queries and operations. The results from different studies often generated different conclusions in terms of which graph databases are more performing. None of them reason about how graph queries are effectively using the hardware platform or inefficiencies thereof that are limiting performance. To answer this question, we believe that it is necessary to understand the performance of a graph database at a microarchitectural level, i.e. hardware resource utilization and bottlenecks when executing graph queries, being inspired by prior studies [5, 36, 38–40] for more conventional workloads.

As far as we know, this work is the first attempt to analyze the performance of executing graph queries in a relational database at the microarchitectural level. Since prior work has demonstrated DuckDB as a promising RDBMS for supporting graph queries [20, 43], we pick DuckDB to study in this paper. For query workload, we use the widely adopted the LDBC Social Network Benchmark (LDBC-SNB) [6]. In particular, we focus on the complex BI queries, which access a significant portion of the graph and are designed with a variety of performance choke points [6, 41, 42].

Our microarchitectural analysis reveals the following insights.

- Processor core pipelines are significantly underutilized waiting to fetch and issue instructions and for data accesses to complete. Core utilizations and/or instruction execution efficiencies (instructions per cycle) are low.
- Cache-conscious and TLB-conscious query processing are important for higher performance. Huge pages can improve query performance by reducing TLB misses and miss overheads.
- While queries have a moderate average bandwidth requirement, short-term peak bandwidth demand can be much higher. Bandwidth demands are likely to increase as future optimizations mitigate pipeline inefficiencies and increase core utilizations.

2 EXPERIMENTAL SETUP

Systems and tools: We use a dual-socket Intel Gold 6226R (Cascade Lake) server with a total of 32 physical (64 logical) cores and 750GB memory, and running Ubuntu 20.04.4 LTS. We generate LDBC SNB BI datasets and parameters [2] for scale factors (SFs) 10 and 100 and create in-memory databases with DuckDB [34] v0.7.1. We use Linux perf, Intel Vtune Profiler [15] v2023.1.0, Performance Counter Monitor [17], and Memory Latency Checker [16] v3.10 to gather microarchitectural statistics, usage, and performance metrics.

Queries: We use query implementations for Umbra [1] as a starting point, then make some changes, e.g., to account for the undirected (bi-directional) nature of the *Friends* relationship from the Person_knows_Person data, and rewrite some path queries. We run each query with 30 settings of parameter values.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DaMoN '23, June 18–23, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0191-7/23/06…$15.00
https://doi.org/10.1145/3592980.3595321

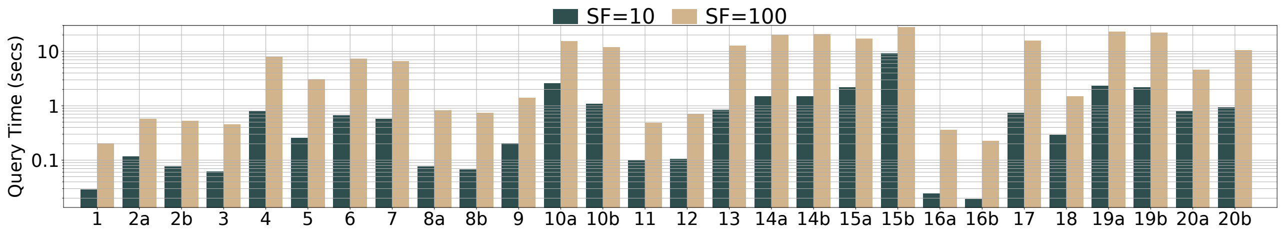

Figure 1: Average query runtimes per setting of parameter values. Query numbers on x-axis.

Server and client processes: We run DuckDB inside a Pyro5 [3] server daemon process, which runs the query requested by a lightweight client. To minimize communication overhead, the server runs the query sequentially for 30 times, once for each parameter setting, before signaling completion to the client. Each query run is multithreaded, but only one query executes at a time. We do a system-wide collection of microarchitectural performance metrics.
Precomputations: Inspired by the schema used for Umbra, we compute and materialize the results of several common computations. This assumes that identification and materialization of commonly-used computations is feasible. One of the main precomputations identifies the root message and associated properties of a *Comment* (a *Comment* can be a reply of another *Comment* or *Post*). We implement this with a recursive CTE. Other precomputations include merging information for some tables. The precomputations take around 12 secs for SF=10 and 2 mins for SF=100.

3 ANALYSIS AND FINDINGS

Runtime and peak resident memory: Figure 1 shows the average query run time per parameter setting excluding precompute time. Some queries have variants (a and b) that differ only in the properties of parameter values. The runtimes span a wide range for each SF. The longest-running queries are Q10 (with high-cardinality group by, joins, correlated subqueries, path computations), Q13 (with high-cardinality group by, joins, correlated subqueries), Q14 (with low-cardinality group by, joins, correlated subqueries, ranking), Q15, Q17 (with joins, negative patterns), Q19, and Q20.

Q15, Q19, and Q20 compute weighted shortest paths. Using recursive CTEs was too computationally expensive for Q15 and Q19, so we use repeated joins to compute shortest paths but limit[1] the number of hops to 6 and 3. We use recursive CTE for Q20, limited[1] to 4-hop paths. The limits are fine for this particular dataset, except for Q15b that failed to find the shortest path for 2 of 30 parameter settings at SF=10 and 3 of 30 settings at SF=100 and would need larger hop-count limits (therefore, more runtime and memory) that may not be feasible to statically determine. Thus, optimizing path queries [43], in particular, efficient computations of weighted shortest paths, is an important optimization for this workload.

The peak resident memory size was 54 GB for SF=10 and 248 GB for SF=100. The large memory requirement relative to the SF can make it challenging to use accelerators with small device memories in a naive way for such workloads.

Core utilization and execution efficiency: Figure 2 shows the average logical core utilization and IPC (instructions per cycle) for each query over its run time. The former indicates occupancy of logical cores while the latter indicates instruction execution efficiency. We see higher utilization for SF=100 compared to SF=10, likely due to each thread having more data to process and relatively smaller query startup and completion overheads. However, even at SF=100, some queries have low utilization, e.g., Q7 (with low-cardinality aggregates, overlap between outer and inner queries, negative patterns), Q11 (triangle counting), Q6, Q13, Q17, Q18. IPC for most queries is < 1, which is considered low (thus, inefficient). The results suggest that there is room for future query processing optimizations to improve both the work distribution to parallel threads and the execution efficiency of each thread.
Pipeline Utilization: Figure 3 shows the breakdown of processor pipeline slots usage, a characterization that is used in the top-down analysis technique [19, 46], into the following categories.

- **Retiring**: Pipeline slots occupied due to retiring (completion) of instructions. This represents useful work.
- **Frontend Bound**: Pipeline slots unutilized due to inadequate supply of instructions to issue from the fetch unit, e.g., due to instruction cache misses, micro-op cache misses, etc.
- **Bad Speculation**: Pipeline slots wasted due to incorrect speculative execution, e.g., due to incorrect branch predictions.
- **Memory Bound**: Pipeline slots blocked waiting for memory operations to complete, e.g., stalled due to data cache miss.
- **Core Bound**: Pipeline slots blocked waiting for core functional units to complete operations.

For higher performance, we expect to see a higher percentage for the Retiring category and lower percentages for the others. However, we observe that across queries, the retiring percentage is not very high with the medians being 31% for SF=10 and 25% for SF=100. The lowest value (8%) occurs for Q20b at SF=100. The Frontend and Memory Bound categories together constitute the major portion of pipeline slots not utilized for useful work. Losses due to incorrect speculation are quite small. Core functional units are not a big bottleneck either. Q7, SF=10, is the most core bound (19%). Overall, many pipeline slots are starved or blocked waiting for instruction fetch and issue or for memory operations to complete, while core computational capacity is not a large bottleneck. We speculate that other hardware backends with somewhat weaker cores but more efficient instruction delivery and data access could be more performant for such workloads.
Load stalls: We now look deeper into the memory bound category. Figure 4 shows the breakdown of cycles with stalls due to memory load operations. Note that each cycle corresponds to multiple pipeline slots, equal to the pipeline width [14], so the distribution of events across pipeline slots and cycles are not the same. L1-, L2-, L3-, and DRAM-Bound categories indicate stalls due to loads that

[1]Thus, Q15, Q19, Q20 are not fully implemented since the spec does not specify limits.

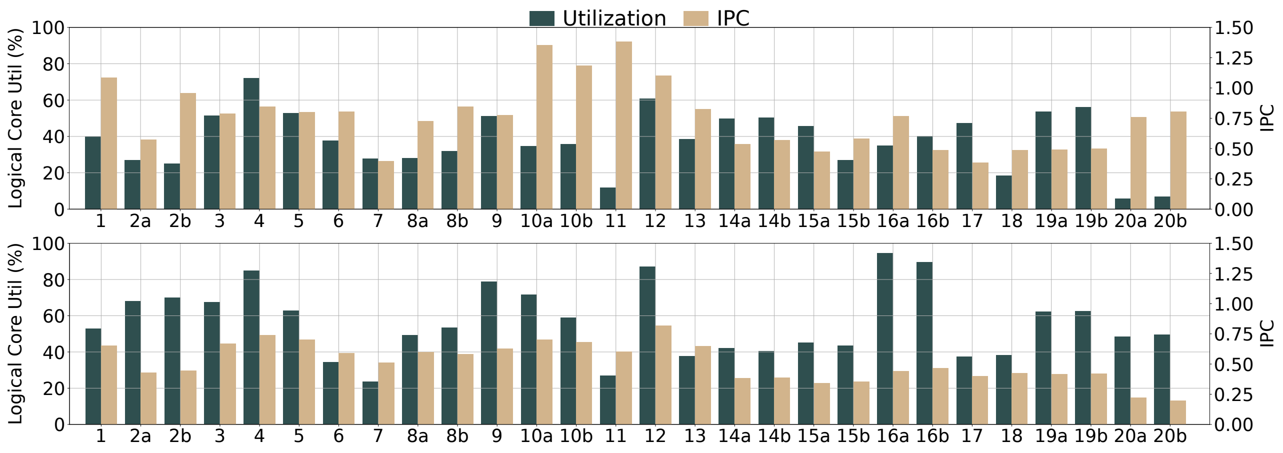

Figure 2: Average logical core utilization and Instructions Per Cycle (IPC). SF=10 (upper), 100 (lower). Query numbers on x-axis.

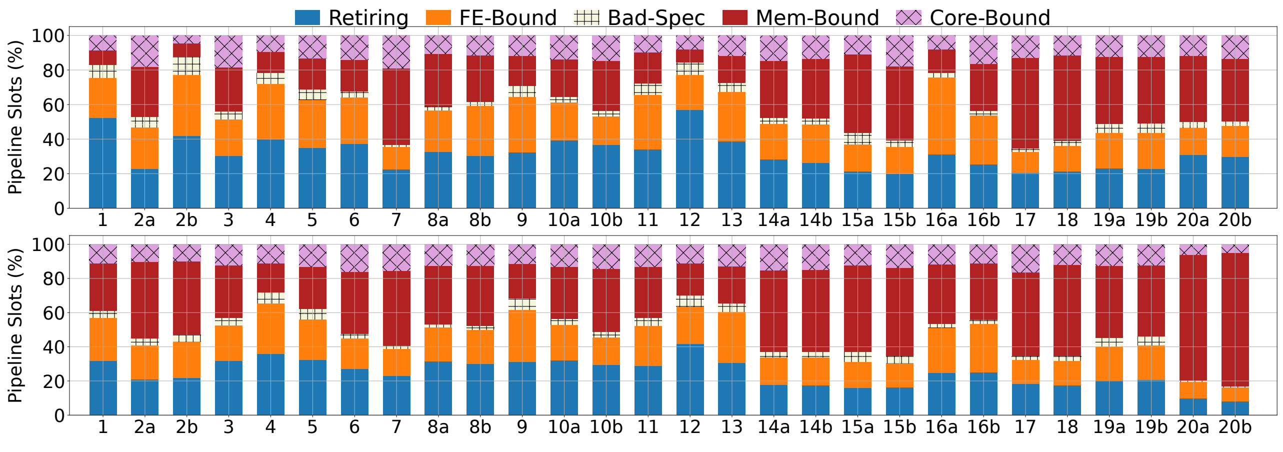

Figure 3: Breakdown of pipeline slots usage. SF=10 (upper), 100 (lower). Query numbers on x-axis.

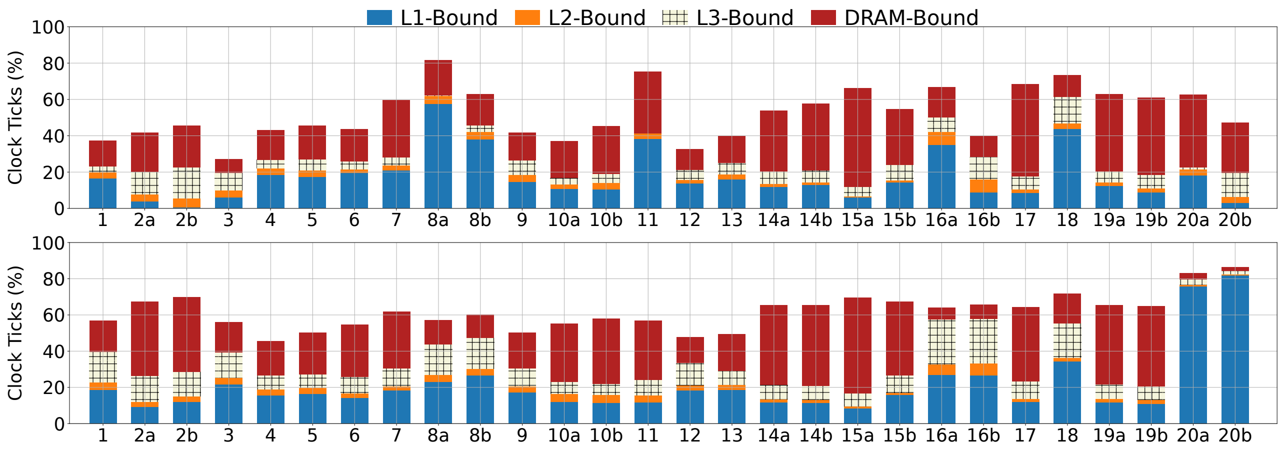

Figure 4: Breakdown of cycles with load (memory read) stalls. SF=10 (upper), 100 (lower). Query numbers on x-axis.

are serviced at that level of the memory hierarchy [14]. L3-Bound also includes inter-core coherence stalls [14]. While DRAM-Bound stalls are not surprising, we also see a significant fraction of cycles with on-chip cache-bound accesses, including at the L1 (that can be caused by inter-instruction data dependencies [14]). Optimizing the on-chip cache hierarchy and having cache-conscious query processing techniques [7, 8, 27, 28, 35, 37] remain important for query performance, as well as for getting full benefits from potential reductions in off-chip bottlenecks, e.g., with the increased DRAM bandwidths available in next-generation processors [18].

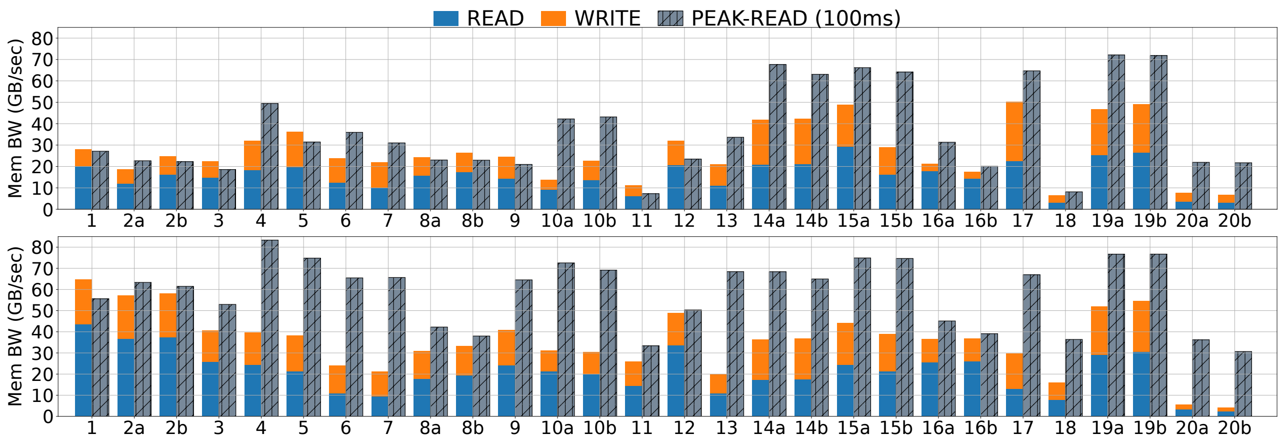

Figure 5: Avg. read, write and peak read (100ms windows) memory bandwidths. SF=10 (upper), 100 (lower). Query nos. on x-axis.

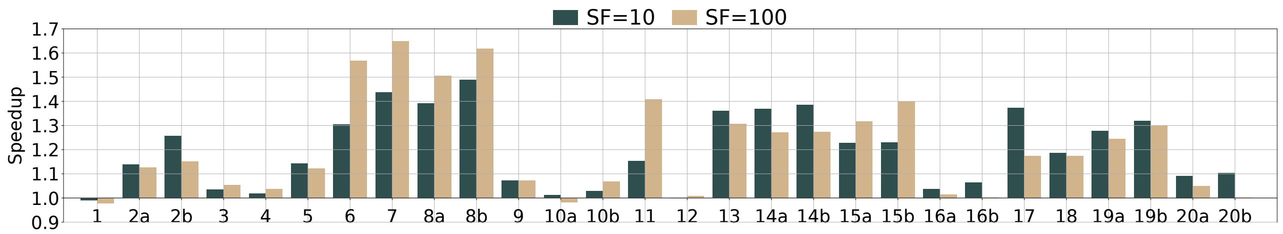

Figure 6: Query speedup with THP. Speedup > 1 implies performance gain. Query numbers on x-axis.

DRAM access performance: Figure 5 shows the DRAM read (includes prefetch) and write[2] bandwidths averaged over the query run time, and peak of average read bandwidths (since usually, reads are more on the critical path) averaged over 100ms time intervals. Although average end-to-end query bandwidths are low, several queries, e.g., Q4, Q13, Q14, Q17, Q20 etc. have much larger peak bandwidths that the memory subsystem should support to avoid potential query slowdowns. Q18 (with negative patterns, undirected cyclic subgraph), Q20 have a relatively smaller average bandwidth demand although being memory bound, indicating latency bottlenecks in data movement across the memory hierarchy. For most queries, cycles having stalls due to DRAM latency are larger than those due to DRAM bandwidth limits on this server. Software prefetches, initiated during intervals when bandwidth is available, might help to mitigate latency stalls [9].

NUMA bottlenecks: The per-socket peak DRAM bandwidth on this server is ~100 GB/sec, but remote DRAM accesses (to the local DRAM at the other socket) see a peak bandwidth of ~34 GB/sec and latency increase of at least 50%. We see 33–75% (median: 54%) remote DRAM accesses across queries, suggesting room for data placement and task scheduling optimizations [24, 25].

TLB misses and speedup with huge pages: We observe significant DTLB load misses that cause page walks. The highest miss rate for such misses occurs for Q15a with an MPKI (misses per Kilo instruction) of 8.7 at SF=10 and 9.3 at SF=100, while the workload-level (all queries) MPKI is 3.9 for SF=10 and 5.3 for SF=100. To reduce these misses and associated overheads, we investigate the effect of using huge pages with the Transparent Huge Page (THP) capability in Linux [4]. THP tries to automatically allocate 2MB pages instead of the default 4KB pages. With THP, we see MPKI reductions, e.g., MPKI for Q15a reduced by 76.3% for SF=10 and 35.9% for SF=100, and at the workload-level it reduced by 47% and by 18.7% respectively. As we show in Figure 6, we get significant speedups for several queries, e.g., at SF=100, Q6: 56.8%, Q7: 64.9%, Q8a: 50.6%, Q8b: 61.7%, Q11: 40.7%, Q14a: 27%, Q14b: 27.3%, Q15a: 31.8%, Q15b: 39.9%, etc. Q20b, SF=100, and a few other queries did not speed up indicating the presence of additional bottlenecks. The speedup for the overall workload (sum of runtimes for all queries) was 21.3% for SF=10 and 22.7% for SF=100. Considering differences in guidance regarding THP for different database systems [30, 32, 47], we suggest caution in using THP in general, but urge further exploration of judiciously using huge pages and TLB-conscious query processing [8, 22, 27] to improve performance.

4 CONCLUSION

Microarchitectural analysis of LDBC SNB BI queries on a modern server revealed resource underutilization and inefficiencies in data access and instruction execution. Huge pages can improve performance significantly, and more opportunities remain for co-optimizing the microarchitecture and query processor for further gains. We believe that microarchitectural analyses can complement algorithmic and software analyses to support queries on RDBMSs more efficiently, in helping to select suitable hardware backends, and by providing insights for using them effectively.

[2]The queries do not modify input data, but their execution creates intermediate results that are written to the cache hierarchy, and when evicted cause writes to memory.

REFERENCES

[1] 2023 (last accessed). LDBC SNB BI Umbra implementation. https://github.com/ldbc/ldbc_snb_bi/tree/main/umbra.

[2] 2023 (last accessed). LDBC SNB Datagen (Spark-based). https://github.com/ldbc/ldbc_snb_datagen_spark/.

[3] 2023 (last accessed). Pyro5 5.14. https://pypi.org/project/Pyro5/.

[4] 2023 (last accessed). Transparent Hugepage Support. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

[5] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs On A Modern Processor: Where Does Time Go? *PVLDB* (1999).

[6] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[7] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[8] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, Vol. 99. 54–65.

[9] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (aug 2007), 17–es. https://doi.org/10.1145/1272743.1272747

[10] Volt Active Data. 2023 (last accessed). VoltDB. https://www.voltdb.com.

[11] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[12] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR '15*.

[13] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. GRFusion: Graphs As First-Class Citizens in Main-Memory Relational Database Systems. In *SIGMOD '18*. 1789–1792. https://doi.org/10.1145/3183713.3193541

[14] Intel. 2023 (last accessed). CPU Metrics Reference. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/cpu-metrics-reference.html.

[15] Intel. 2023 (last accessed). Get Started with Intel VTune Profiler for Linux OS. https://www.intel.com/content/www/us/en/docs/vtune-profiler/get-started-guide/2023/linux-os.html.

[16] Intel. 2023 (last accessed). Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[17] Intel. 2023 (last accessed). Intel Performance Counter Monitor (Intel PCM). https://github.com/intel/pcm.

[18] Intel. 2023 (last accessed). Intel Xeon CPU Max Series. https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html.

[19] Intel. 2023 (last accessed). Top-down Microarchitecture Analysis Method. https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html.

[20] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2022. GRainDB: A Relational-core Graph-Relational DBMS. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p57-jin.pdf

[21] Salim Jouili and Valentin Vansteenberghe. 2013. An Empirical Comparison of Graph Databases. In *2013 International Conference on Social Computing*. 708–715. https://doi.org/10.1109/SocialCom.2013.106

[22] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*. 1–10. https://doi.org/10.1145/3076113.3076115

[23] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlýnková. 2013. Experimental Comparison of Graph Databases. In *Proceedings of International Conference on Information Integration and Web-Based Applications & Services (IIWAS '13)*. 115–124. https://doi.org/10.1145/2539150.2539155

[24] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754. https://doi.org/10.1145/2588555.2610507

[25] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf

[26] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation. *PVLDB* 12, 4 (2018), 390–403. https://doi.org/10.14778/3297753.3297759

[27] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (jul 2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[28] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. Optimizing database architecture for the new bottleneck: memory access. *The VLDB journal* 9 (2000), 231–246. https://doi.org/10.1007/s007780000031

[29] Microsoft. 2023 (last accessed). Graph processing with SQL Server and Azure SQL Database. https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017.

[30] Microsoft. 2023 (last accessed). Performance best practices and configuration guidelines for SQL Server on Linux. Leave Transparent Huge Pages (THP) enabled. https://learn.microsoft.com/en-us/sql/linux/sql-server-linux-performance-best-practices?view=sql-server-ver16#leave-transparent-huge-pages-thp-enabled.

[31] Neo4j. 2023 (last accessed). Neo4j Graph Data Platform. https://neo4j.com/.

[32] Oracle. 2023 (last accessed). Database Installation Guide for Linux. Disabling Transparent HugePages. https://docs.oracle.com/en/database/oracle/oracle-database/19/ladbi/disabling-transparent-hugepages.html.

[33] Oracle. 2023 (last accessed). Spatial and Graph features in Oracle Database. https://www.oracle.com/database/technologies/spatialandgraph.html.

[34] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[35] Kenneth A. Ross. 2009. *Cache-Conscious Query Processing*. Springer US, Boston, MA, 301–304. https://doi.org/10.1007/978-0-387-39940-9_658

[36] Rathijit Sen and Karthik Ramachandra. 2018. Characterizing resource sensitivity of database workloads. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 657–669. https://doi.org/10.1109/HPCA.2018.00062

[37] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 510–521.

[38] Utku Sirin. 2021. *Micro-architectural Analysis of Database Workloads*. Ph. D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-10363

[39] Utku Sirin and Anastasia Ailamaki. 2020. Micro-Architectural Analysis of OLAP: Limitations and Opportunities. *PVLDB* (2020), 840–853. https://doi.org/10.14778/3380750.3380755

[40] Utku Sirin, Pınar Tözün, Danica Porobic, Ahmad Yasin, and Anastasia Ailamaki. 2021. Micro-Architectural Analysis of in-Memory OLTP: Revisited. *The VLDB Journal* 30, 4 (mar 2021), 641–665. https://doi.org/10.1007/s00778-021-00663-8

[41] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. 2018. An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In *GRADES-NDA at SIGMOD/PODS*. ACM, 9:1–9:11. https://doi.org/10.1145/3210259.3210268

[42] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *PVLDB* 16, 4 (dec 2022), 877–890. https://doi.org/10.14778/3574245.3574270

[43] Daniël ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *Proceedings of the Conference on Innovative Data Systems Research*. https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf

[44] Yuanyuan Tian. 2023. The World of Graph Databases from An Industry Perspective. *SIGMOD Record* 51, 4 (jan 2023), 60–67. https://doi.org/10.1145/3582302.3582320

[45] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 345–359. https://doi.org/10.1145/3318464.3386138

[46] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[47] Wenbo Zhang. 2020. Transparent Huge Pages: Why We Disable It for Databases. https://www.pingcap.com/blog/transparent-huge-pages-why-we-disable-it-for-databases/.

[48] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1165–1180. https://doi.org/10.1145/3035918.3035943