Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases: Supplemental Material

Craig Chasseur
University Of Wisconsin
chasseur@cs.wisc.edu

Jignesh M. Patel
University Of Wisconsin
jignesh@cs.wisc.edu

ABSTRACT

This document is a supplement to the authors' paper *Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases* [4]. It contains additional experimental results which demonstrate the interaction between join processing and storage organization.

1. JOIN PROCESSING

Processing of join queries in a main-memory engine is a very active area of research. Well-known join algorithms such as sort-merge join [1, 8] and hash-join [2, 3, 5] are being adapted and tuned to highly parallel multi-core CPUs in the main memory environment. Joins are a higher-level query processing operation that, in general, are beyond the single-relation access plan-centric focus of the paper. Nevertheless, it is known that widely-used join algorithms do interact with the storage manager in a predictable way, and storage manager performance affects join performance. For instance, the previous study which we base our experiments on showed that hash-join performance is closely related to the scan performance of base tables [7], since both building a hash table and scanning the outer table to probe it involve a linear scan of the base tables in the join.

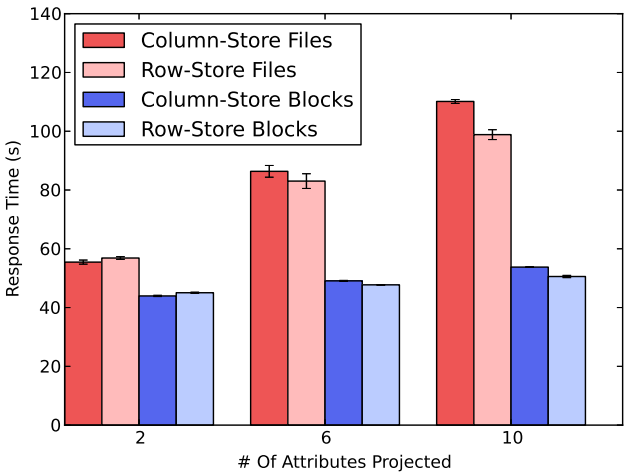
We evaluated a hash join in Quickstep, with both row-store and column-store data layout in both blocks and files. We use an outer table consisting of 750 million tuples based on the Narrow-U schema, but replace one column with a unique key. The inner table has a similar schema with 75 million tuples (1/10 as many, the same ratio of table sizes used in the join queries **Q10-Q17** of the Wisconsin benchmark [6]). Queries are of the form:

```
SELECT outer.col_a, ..., inner.col_a, ...  FROM
outer, inner WHERE outer.join_key = inner.join_key;
```

We vary the number of "payload columns" projected from each table and materialized in the output. As with our other experiments, we use 20 worker threads operating in parallel (for both the build and probe phases of the hash-join).

In all cases, we build a single global hash-table on the join key of the inner table (using a single global hash-table has been shown to outperform using smaller partitioned hash-tables when joining [3]). We use a concurrent hash-table implementation which allows us to build the global hash table in parallel, with multiple threads scanning different blocks or file partitions of the inner table and inserting entries in the hash table.

We show results for the actual join query in Figure 1. The query executes in two stages. In the first, worker threads scan the inner table in parallel and insert entries into the global hash table. In the second, worker threads scan the outer table in parallel and probe the hash table for a match on the join key. When a match is found, the projected columns are fetched from the inner and outer tables and materialized as a new row in the output (we do not claim that the projection method here is optimal, and acknowledge that aspects such as early vs. late materialization is an active an ongoing area of research [10], and beyond the scope of the paper). During the build stage, access on the inner table follows a linear scan pattern, while insertions into the hash table follow a random-access pattern. During the join stage, access on the outer table follows a linear scan pattern, and access on the hash table and the inner table follows a random-access pattern. The memory-access pattern of a hash-join therefore has some characteristics of a scan (scanning the inner table when building the hash table and the outer table when joining) and some which are similar to index-access (random-access probing of the hash table and fetching inner-table tuple values for projection). For all queries, the hash-build stage took about 7.5 seconds



Organization	Blocks vs. Files
Tuple Storage Layout	Column-Store vs. Row Store
Indexing	None
Predicate	JOIN
Projection Width	Varies

Figure 1: Hash Join Performance

with files and 8.5 seconds with blocks[1], with the remaining bulk of the execution time (and most of the difference in performance) spent in the probe-and-project stage.

Block-based organization consistently outperforms file-based organization, due to improved locality of access in smaller blocks resulting in better cache behavior (consistent with our results for selection queries). For instance, when projecting 10 columns (5 from each table), row-stores in files incur 5.38 billion L3 cache misses and 14.4 billion L2 cache misses, but row stores in blocks incur only 4.58 billion L3 misses and 5.83 billion L2 misses. We also see a slight performance advantage for row-stores over column-stores when projecting several columns, since the random access to column values in the inner-table hits all the desired values on one or two contiguous cache lines in a row store, whereas they are in several disjoint cache lines in different column stripes in a column store. For example, when projecting 10 attributes in file-based organization, column-stores incur 6.41 billion L3 cache misses, but row-stores incur only 5.38 billion.

Observation 13. *For hash-join queries, block-based organization consistently outperforms file-based organization. For wider projections, row-stores slightly outperform column stores.*

It should be noted that, in many analytics applications, there is a "star schema" where a large central fact table is connected to several smaller dimension tables by primary key-foreign key relationships. Queries on joins of the fact table with one or more dimension tables are extremely common. A widely-used optimization is to denormalize the star schema by pre-joining the fact table with the dimension tables and storing the result as a materialized view which single-table selection-projection and aggregate-grouping queries are run on. Some systems (for instance Vectorwise [9]) automatically apply this optimization by introducing a "join index" data structure, which is effectively a materialized view which the DBMS automatically creates based on the primary-foreign key relationships in the database schema.

Overall, results with Quickstep show that performance of in-memory hash-joins is very closely related to scan performance of the underlying tables, as well as the random-access performance of the inner table. Applying the common technique of pre-joining tables also effectively turns many join queries into scan queries.

Acknowledgments

This work was supported in part by National Science Foundation grant IIS-1250886 and a gift donation from Google.

2. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, pages 1064–1075, 2012.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Oszu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. *ICDE*, 2013.

[3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. *SIGMOD*, pages 37–48, 2011.

[4] C. Chasseur and J. M. Patel. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *VLDB*, 2013.

[5] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116 – 127, March 2004.

[6] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. *The Benchmark Handbook for Database and Transaction Systems. Morgan Kaufmann*, 1993.

[7] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *VLDB*, pages 502–513, 2008.

[8] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *VLDB*, pages 1378–1389, 2009.

[9] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *VLDB*, pages 1648–1653, 2009.

[10] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. *ICDE*, 2013.

[1]The slightly reduced performance for blocks in the build phase suggests that the block-based organization might actually be *too* parallelization-friendly relative to files during the build phase, since building the global hash table requires threads to sometimes synchronize when accessing shared hash-table buckets. Tuning the number of threads used in the hash-build phase below the number used in the probe-and-project phase is an interesting possible optimization, but general hash-join algorithm tuning is a broad area of research beyond the scope of the paper.