The Value of Merge-Join and Hash-Join in SQL Server

Goetz Graefe
Microsoft, Redmond, WA 98052-6399
GoetzG@Microsoft.com

Abstract

Microsoft SQL Server was successful for many years for transaction processing and decision support workloads with neither merge join nor hash join, relying entirely on nested loops and index nested loops join. How much difference do additional join algorithms really make, and how much system performance do they actually add? In a pure OLTP workload that requires only record-to-record navigation, intuition agrees that index nested loops join is sufficient. For a DSS workload, however, the question is much more complex. To answer this question, we have analyzed TPC-D query performance using an internal build of SQL Server with merge-join and hash-join enabled and disabled. It shows that merge join and hash join are both required to achieve the best performance for decision support workloads.

1.0 Introduction

For a long time, most relational database systems employed only nested loops join, in particular in the form of index nested loops join, and merge join. The general rule of thumb, stated over 20 years ago [Blasgen and Eswaran 1977], is that nested loops join is good if at least one join input is small, and merge join is good for two large inputs. Given that an input is either small or large, who needs more join algorithms?

Some database systems, in particular those that targeted online transaction processing (OLTP) applications, did not even use merge join. All join operations were executed by the nested loops algorithm. The most sophisticated join algorithm sorted the outer input and built an index on the inner input on the fly. Given the very similar disk reference patterns, we call index nested loops join with a sorted outer input a *poor man's merge join*.

Starting in about 1984, hash-based algorithms found intense interest among database researchers [Kitsuregawa et al. 1993, DeWitt at al. 1984, Sacco 1986, Shapiro 1986], and many hash join variants were invented. Product development teams were slow to adopt these new algorithms. Some relational database systems still have not added hash joins to their repertoire. One reason has been that query optimization technology wasn't sufficiently extensible to permit easy integration of a new algorithm.

Given the success of database systems that don't even have merge join, what is the value of hash and merge joins? We believe that the value of these algorithms strongly depends on both the workload and the set of available indexes. Moreover, the most desirable indexes most likely depend on the available algorithms. In other words, any comparison of these algorithms should be based on indexes specifically designed for the workload and the available algorithms.

Choosing the best indexes by hand for a single complex query with any assurance of optimality is hard. For an entire workload consisting of multiple complex queries, it is very, very hard (basically impossible). Such complex workloads require an automated tool to evaluate the alternatives and find an optimal design. SQL Server includes such a tool, called the "index tuning wizard" [Chaudhuri and Narasayya 1998], as well as variants of nested loops join, merge join, and hash join. The tool heuristically explores very many combinations of indexes, and relies on the query optimizer to estimate execution costs for given or collected workload. While the tool might run for minutes or even hours for workloads much more complex than TPC-D, it can design index sets for entire workloads with a reasonable assurance of optimality.

This study relies on the index-tuning wizard to design optimal index sets for a specific workload with complex queries (TPC-D, TPC-H, TPC-R) and for specific combinations of available join algorithms. After establishing a baseline using a small set of indexes, it compares the performance and assesses the value of merge join and hash join using an index set optimized for the entire, complex workload.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

2.0 Experiment setup and environment

Like many performance studies, ours leaves many parameters and definitions constant for all our experiments. The database is the TPC-D verification database with 100 MB of raw data (scale factor 0.1) [TPC-D]. There are not-null constraints on all columns, a primary key constraint on each table, and all suitable referential integrity constraints are declared and enforced in the database.

All experiments were performed using a desktop PC running Windows NT Server 4.0 and a functionally complete and partially tuned development build of Microsoft SQL Server 7.5. The hardware was a 450 MHz Pentium II CPU, 128 MB of memory, and a single 13.4 GB EIDE disk drive. Other than a minimized instance of Enterprise Manager (the graphical administration tools) and a minimized instance of Query Analyzer (the interactive SQL query tool used to drive the experiments), there are no other applications running on the machine. Each run started with both the I/O buffer and the procedure cache (compiled query plans) empty. The TPC-D queries were run in the order Q1, Q2, …, Q17, Q13-old, Q18, …, Q22. SQL Server's memory usage is set to 32 MB in all runs, which is shared among buffer, procedure cache, query memory (sort workspace, hash tables), etc. While the software supports parallel query processing, this feature was not employed in this study because there is only a single CPU. Asynchronous I/O is exploited extensively, including pre-fetching multiple needed records at a time, e.g., in index nested loops join.

The query optimizer plans each query for minimal resource consumption. Its cost functions presume a mostly cold buffer. For an entire workload, this means the query optimizer and the tuning wizard optimize system load or elapsed time of the entire workload. Therefore, these are the measures we report in the experiments below.

Since release 7.0, SQL Server automatically samples the database to create statistics on columns desired by the query optimizer. In order to eliminate the cost of statistics creation, we optimized the entire batch of queries once before running an experiment, and then cleared out both the I/O buffer and the procedure cache before running each set of queries.

Since the product is still under development, we do not report actual elapsed times. All reported performance numbers are scaled to the elapsed time of the entire workload using the most simplistic physical database design studied. Given the improvements in hardware performance, relative numbers serve the purposes of this comparison study just as well as absolute numbers could.

3.0 Effects in a simple database design

The first physical database design we evaluate is particularly simple. The only indexes are those required as part of the primary key definitions as well as indexes on foreign keys and on date columns. All indexes are non-clustered indexes. The purpose of evaluating this design is to establish a baseline for comparison to other physical database designs and demonstrate how important index choices are for any comparative analysis of query evaluation algorithms.

The performance of all TPC-D requests against this physical database design is shown in Table 1, with all times normalized to indicate percentages of the elapsed time of entire run using only nested loops join (see column 1). The columns represent the set of join algorithms available to the query optimizer. Algorithms are named by the abbreviations NLJ, MJ, and HJ. NLJ implies naïve nested loops join, index nested loops join, and poor man's merge join, where applicable. The merge join algorithm is very standard, using a temporary file to generate all matches in many-to-many joins. The hash join is a fairly sophisticated implementation, exploiting bit vector filters, role reversal, recursion for very large inputs, and "teams" of hash operators. The latter technique provides in hash-based query processing most of the long-known benefits of "interesting orderings" in sort-based query processing [Selinger et al. 1979, Graefe et al. 1998].

Table 1 – Performance for the simple physical database design. Results are elapsed time scaled so that NLJ sums to 100.

	NLJ only	NLJ+MJ	NLJ+HJ	All
Query 1	2.85	2.96	1.15	1.18
Query 2	0.16	0.16	0.17	0.27
Query 3	3.41	1.38	0.82	0.83
Query 4	3.35	0.88	0.78	0.79
Query 5	4.39	1.85	0.81	0.80
Query 6	0.73	0.63	0.62	0.63
Query 7	3.85	1.25	0.80	0.80
Query 8	6.28	2.97	0.82	0.83
Query 9	40.34	3.59	1.28	1.33
Query 10	7.01	1.41	1.02	0.99
Query 11	2.55	0.49	0.21	0.21
Query 12	1.29	1.03	0.76	0.76
Query 13	0.30	0.29	0.31	0.30
Query 14	1.12	0.79	0.67	0.67
Query 15	1.54	1.49	1.22	1.25
Query 16	0.17	0.19	0.17	0.16
Query 17	0.40	0.53	0.35	0.35
Query 13 (old)	2.24	0.45	0.41	0.40
Query 18	5.65	3.29	3.26	3.23
Query 19	0.76	0.76	0.71	0.72
Query 20	5.13	5.95	5.85	5.79
Query 21	6.40	10.11	7.44	7.55
Query 22	0.10	0.18	0.09	0.18
Total	100.00	42.61	29.71	30.00

Some observations are immediately obvious from Table 1. By looking at the totals, it is clear that for this fairly restricted index set, SQL Server's old set of query processing algorithms are not sufficient. Adding merge join or hash join gives a 2-fold or 3-fold performance improvement respectively.

Hash join has a substantial advantage over merge join only. In fact, a query processor using only merge join for large inputs is 40% slower than one using hash join (30 vs. 42). Thus, if there are very few indexes in the database, or if the existing indexes don't serve a query very well, hash join has substantial value. In a way, this is not surprising, given that the hash table in a hash join is nothing but an in-memory on-demand index.

A more surprising observation is that the total for the column "NLJ+HJ" is comparable to that for all algorithms. For this query set, Merge-Join does not add anything to a "NLJ+HJ" system.

This is in part due to weaknesses in the beta-quality optimizer and execution engine used in these experiments. Compare the elapsed times for query 2. Clearly, when given a choice, the optimizer wrongly chooses merge join or hash join. Also, query 21 is executed most efficiently using nested loops join.

Nonetheless, the optimizer frequently makes good use of merge join and hash operations. For example, query 1 shows the effectiveness of hash grouping. Queries 3, 4, 5, 7, 8, etc. benefit substantially from the additional join algorithms being available to the optimizer. Query 9 in particular performs very poorly, and it alone accounts for ½ of the difference in the column totals.

4.0 Effects in optimized database designs

The experiment in the previous section compared the performance of the different combinations of available join algorithms on a fixed physical database design. However, it is well known that desirable algorithms and desirable indexes affect each other.

Consider an index set specifically optimized for the entire workload using the index-tuning wizard for and for the query processor including all join algorithms. The index-tuning wizard limits index creation either by the cost of update operations included in the workload or by the available disk space. In our optimization, we did not include update operations but limited the space available for all indexes in the database to twice the data space. Table 2 shows the chosen set of indexes created or retained from the simple database design for the given query set, in addition to retaining non-clustered indexes on all primary keys. Note that there is no index on the "nation" table or the "customer" table, and that there is only a single clustered index in the entire database. Note also that the wizard clearly focuses on creating covering indexes to enable index-only scans.

Table 2 – Indexes as optimized for all algorithms

Table	Clustered	Columns
Region	No	R_RegionKey
Supplier	No	S_SuppKey, S_NationKey
Part	No	P_PartKey
PartSupp	No	PS_PartKey, PS_SuppKey, PS_AvailQty, PS_SupplyCost
Orders	No	O_OrderDate, O_OrderKey, O_CustKey
	No	O_CustKey, O_OrderKey
	No	O_OrderKey, O_OrderStatus
LineItem	Yes	L_ShipDate
	No	L_ShipDate, L_Quantity, L_ExtendedPrice, L_Discount, L_Tax, L_ReturnFlag, L_LineStatus
	No	L_PartKey, L_OrderKey, L_SuppKey, L_Quantity, L_ExtendedPrice, L_Discount
	No	L_OrderKey, L_SuppKey, L_PartKey, L_CommitDate, L_ReceiptDate
	No	L_OrderKey, L_ExtendedPrice, L_Discount, L_ReturnFlag
	No	L_OrderKey, L_Quantity

The performance of all TPC-D requests against this physical database design is shown in Table 3 with all times normalized to the same base line as the previous experiment.

Interestingly, even though the index set was optimized for the query processor with all algorithms, the more limited query processors benefit, too. The most limited query processor using only NLJ improved by a factor of 2.3 (100 to 44.70), whereas the most complete query processor improved only by a factor 1.5 (30 to 19.94). The limited query processor depends most heavily on useful indexes, and the complete query processor degrades gracefully if optimal indexes are missing. Moreover, an improvement of 50% is substantial and worthwhile the cost of running the tuning wizard.

Table 3 – Performance for an optimized physical database design

	NLJ only	NLJ + MJ	NLJ+HJ	All
Query 1	2.59	2.71	0.91	0.96
Query 2	0.09	0.10	0.15	0.16
Query 3	1.67	0.87	0.86	0.88
Query 4	1.54	0.84	0.87	0.82
Query 5	4.36	2.75	0.88	0.88
Query 6	0.10	0.07	0.07	0.08
Query 7	5.40	1.12	0.75	0.76
Query 8	2.27	0.28	0.20	0.20
Query 9	4.58	1.84	1.58	1.58
Query 10	2.60	1.00	0.95	0.87
Query 11	2.03	0.57	0.09	0.09
Query 12	3.31	3.29	1.84	1.83
Query 13	0.26	0.24	0.22	0.21
Query 14	0.35	0.08	0.05	0.05
Query 15	0.26	0.24	0.13	0.12
Query 16	0.18	0.20	0.16	0.16
Query 17	0.04	0.04	0.03	0.03
Query 13 (old)	1.54	0.48	0.40	0.40
Query 18	2.46	1.17	1.15	0.94
Query 19	1.73	1.68	1.67	1.67
Query 20	0.36	0.29	0.24	0.26
Query 21	6.86	6.81	12.28	6.77
Query 22	0.13	0.19	0.10	0.21
Total	44.70	26.85	25.59	19.94

It is also interesting to see that the totals for the query processors using merge join only and hash join only are very similar, although the performance for individual queries is quite varied. For example, queries 5, 11, and 12 benefit significantly from hash joins, whereas query 21 benefits from merge joins. The complete query processor chose the optimal query plan for each of the queries, and its performance is about 25% better than the performance of the query processors using either only merge join or only hash join (20 vs. 25).

5.0 Summary and conclusions

In summary, we found that merge join and hash join are both required to achieve the best performance for decision support workloads. To a surprising degree, careful index design alleviates the problem substantially, but it requires an automated tool for complex workloads. Even then, navigating indexes is not competitive with a query processor that includes a full complement of query evaluation algorithms.

However, a query processor using only nested loops join quite successfully processes only the required records. Thus, nested loops join tends to increase the elapsed time (as reported above), but it also reduces the CPU time. A more complete version of this study will compare both elapsed and CPU times as well as investigate the effectiveness of asynchronous I/O for scanning and fetching.

6.0 References

Blasgen and Eswaran 1977. Mike W. Blasgen, Kapali P. Eswaran: Storage and Access in Relational Data Bases. IBM Systems Journal 16(4): 362-377 (1977).

Chaudhuri and Narasayya 1998: Surajit Chaudhuri, Vivek Narasayya: Microsoft Index Tuning Wizard for SQL Server 7.0. ACM SIGMOD Conference 1998: 553-554.

DeWitt at al. 1984. David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation Techniques for Main Memory Database Systems. ACM SIGMOD Conference 1984: 1-8.

DeWitt et al. 1993. David J. DeWitt, Jeffrey F. Naughton, J. Burger: Nested Loops Revisited. Proc. Parallel and Distributed Information Systems 1993: 230-242.Graefe et al. 1998. Goetz Graefe, Ross Bunker, Shaun Cooper: Hash Joins and Hash Teams in Microsoft SQL Server. VLDB Conference 1998: 86-97.

Masaru Kitsuregawa, Hidehiko Tanaka, Tohru Moto-Oka: Application of Hash to Data Base Machine and Its Architecture. New Generation Computing 1(1): 63-74 (1983).

Sacco 1986. Giovanni Maria Sacco: Fragmentation: A Technique for Efficient Query Processing. ACM Trans. on Database Systems 11(2): 113-133 (1986).

Selinger et al. 1979: Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. ACM SIGMOD Conference 1979: 23-34.

Shapiro 1986. Leonard D. Shapiro: Join Processing in Database Systems with Large Main Memories. ACM Trans. on Database Systems 11(3): 239-264 (1986).

TPC-D. Transaction Processing Performance Council, www.tpc.org.