RC25117 (W1102-073) February 27, 2011
Electrical Engineering

IBM Research Report

Parallel Implementation of Relational Database Management Systems Operations on a Multi-Core Network-Optimized System on a Chip

Elahe Khorasani, Brent D. Paulovicks, Vadim Sheinin, Hangu Yeo
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Parallel Implementation of Relational Database Management Systems Operations on a Multi-Core Network-Optimized System on a Chip

Elahe Khorasani, Brent D. Paulovicks, Vadim Sheinin, and Hangu Yeo

Abstract

In a commercial Relational Database Management System (RDBMS), sort and join are the most demanding operations and it is quite beneficial to improve the performance of external sort and external join algorithms that handle large input data sizes. This paper proposes parallel implementations of multi-threaded external sort and external hash join algorithms to accelerate IBM DB2, one of leading RDBMSs, in sort and join operations using an IBM Power Edge of Network (IBM PowerEN™) Peripheral Component Interconnect Express (PCIe) card. The PowerEN™ PCIe card is used as an accelerator, and is based on PowerEN™ chip that runs at 2.3 GHz. The PowerEN™ chip consists of sixteen embedded 64-bit PowerPC cores, and each of sixteen cores supports four hardware threads. The main advantage of using features of PowerEN™ such as multithreaded multi-core, 10 GE Ethernet interface and hardware compression and decompression coprocessor is to execute the algorithms in parallel and reduce the size of data stored externally, and hence reduces the execution time as well as the size of external storage and bandwidth

between the accelerator and external disk. Simulations are done on PowerEN™ PCIe card using input records streamed from DB2, and results are compared with the performance of DB2 sort and join operations. The preliminary results show that the proposed parallel execution of external sort and join algorithm speeds up the DB2 sort and join performance about two times.

Introduction

Relational database management system now form the majority of database management systems, and lets users create, update, and find specific information much easier than the original flat databases without working sequentially through the entire information. Relational database stores data based on the relationship among the elements of data, and hence allows users to access to data easier than conventional database models such as hierarchical database and network database models. In the relational database system, the data is stored in the form of a table, and the related data are stored across multiple tables. Each row of the table is called a record, and each record contains fields which are columns of the table. A key is a logical way to identify and access a record in a table, and distinguishes one record from another. The key can be an individual column or a group of columns to differentiate

records from one another. It is important to store data into the database and retrieve data out of database efficiently and quickly. In relational database, structured query language (SQL) is a standard language to access and manipulate databases. The SQL allows the users to create databases, create tables in a database, insert and delete records, execute queries on the data, and supports arithmetic, equality, and logical operators to perform operations on the data stored in the RDBMS.

The sort and join operations are classic standard relational database operations, and are the most demanding operations of RDBMS for building indexes, binary searches, grouping, aggregation, etc, and hence obviously it is beneficial to improve performance of those operators by parallelizing the operations. There have been efforts to implement parallel versions of the algorithms for multi-core single instruction multiple data (SIMD) processors or hardware accelerators such as graphics processing units (GPUs). Bitonic sort and radix sort algorithms are well suited for SIMD processors and GPUs [2]-[6], and quicksort is reported to scale well when the number of core increases using hyperthreading technology [7]. There have been considerable studies on the parallel hash join algorithm [8]–[11], and parallelism was easily exploited for the high performance hash join operation.

Although a single pass in-memory sort or join operations are the fastest, but not always the fastest one with limited resources (limited main memory size). To handle huge input data size, multi-pass operations are more appropriate. External sort algorithm is applicable when the data to be sorted is too large to fit in the primary memory, and external merge sort is one of the most popular algorithms [1]. The data to be sorted is divided into runs so that the size of the run is small enough to fit into the main memory. Each run is sorted within the main memory using a sort algorithm, and the sorted runs are stored on the external storage device until all the runs are sorted. The sorted runs are read from the external storage, merged together to form fewer but larger run lists, and this merge process continues until the data is completely sorted. Hash join algorithm is commonly used in database systems to implement equijoins efficiently.

In an equijoin, equality is an operator to compare key values. The hash join algorithm consists of two phases, build phase and probe phase. The build phase creates an in-memory hash table using records of the smaller table (build table). During the probe phase, the hash table built in the build phase is probed using records of the larger table (probe table) to

generate joined output records from the pairs of matching records. Hybrid hash join algorithm is an external join operation developed to handle the case where the input tables are too big to be stored in the available main memory. In the hybrid hash join algorithm, the two input tables are partitioned such that a pair of partitions, one partition from each table, can fit in the main memory. Once the input tables are partitioned and stored in the external storage deice, pairs of build and probe partitions are loaded in the main memory, and joined sequentially.

The PowerEN™ chip [12] is a network-optimized processor system on a chip developed by IBM. The chip consists of four AT chiplets where each chiplet has four 64-bit embedded PowerPC A2 cores and 2 MB shared eDRAM L2 cache (8 MB L2 cache per chip), two memory controllers, four 10 GE Ethernet interfaces and five acceleration engines. Each core supports four hardware threads (64 total threads per chip) which share L1 and L2 caches, and provides memory management unit (MMU) and separate instruction and data cache controllers and arrays. The five acceleration engines include Host Ethernet acceleration for network protocol processing, Compression/Decompression engine, Cryptographic co-processor, Extensible Markup Language (XML) engine and Regular Expression/Pattern-matching engine. The compression engine works as a coprocessor to the A2

core, and used Lempel-Ziv (LZ77) compression followed by the Huffman Coding. The compression engine runs at 10 Gbps with the compression ratio between two and five when tested with various TPC-H tables generated as depicted in Table 1. The PowerEN™ PCIe card is an integrated board design based on PowerEN™ chip intended to be used for software developer platform, and x3650 M2 is used as a host for the card.

In this paper, we implemented efficient parallel external sort algorithm and external join algorithm (hybrid hash join algorithm) designed to accelerate the performance of DB2 in the area of sort and hash join operations by taking advantage of features of PowerEN™ chip. The DB2 code is altered so that the input records are intercepted to feed the sort and hash join processes that runs on an external multi-core accelerator, where they are implemented in a highly parallel manner and the results are returned to DB2. Since the speed of a single threaded external sort or join process is mainly limited by the speed of the processor and speed of data transfer rate to the disk, the implemented algorithms are mainly designed to take advantages of multi-threaded execution, compression and decompression units and 10 GE Ethernet interfaces.

Overview of DB2 Sort and Hash Join

From a very high level view, when a sort or hash join of one or more tables are required DB2 creates a sort (SORT) or hash join (HSJN) process. If the table data is not already in the buffer pool (the area allocated in the main memory by data base manager for cashing the table or index data), it is read from storage into the buffer pool, based on the size of the buffer pool, for fast access. In the case of sort, table records are "inserted" from the buffer pool into the SORT one row at a time. The SORT processes the input rows by inserting them in the in-memory sort list. If the whole table's data does not fit in the sort memory it is called external sort. In external sort, as the sort memory gets exhausted, temporary tables of the sorted lists are created and spilled to the disk to be merged later before returning to the client. Each one of these temporary tables is called a sorted run. After the last row is processed, the "fetch" process starts, in which records are retrieved from the sorted list and returned to the client one row at a time. If the sort is external, the spilled sorted runs have to be merged so that the retrieved rows are in order.

HSJN follows a similar process. First the build table is inserted into HSJN one row at a time and grouped into logical partitions based on a

hash code. If memory is exhausted the partitions are spilled to the disk. Then the probe table is inserted one row at a time. As the probe records are inserted, if the record's hash code matches the build hash code that is in memory the keys are compared for a match. If a match is found it is returned to the client. If the record's hash code does not match the in-memory build hash code the probe record is sent to the disk for later processing. When the entire probe table is scanned, the leftover process begins, in which all the records that had been spilled to the disk are processed for finding matches to be returned to the client.

Algorithm Implementation

The external sort and hybrid hash join algorithms are implemented in two passes, and each pass is implemented using multiple threads to take advantages of multi-core designs and parallelism at the chip level. In the first pass, the external sort algorithm (hybrid hash join algorithm) produces sorted runs (hashed partitions) using the records streamed from DB2, and the sorted runs (hashed partitions) are compressed and stored on a file server through the 10 GE Ethernet interface. In the second pass, the sorted runs (hashed partitions) are read from the file server, decompressed, and merged (joined). The sorted records are returned back to DB2 to complete the external sort process, and the join process packs

matched records using a record from the build table and a record from the probe table, and the packed records are returned to DB2. Figure 1 depicts an overview of aforementioned DB2 sort and join process accelerated with a PowerEN™ PCIe card. The sort and join requests are made by DB2 (sort or join client), and the records are streamed from DB2 to the accelerator through 10 GE interface. Both the DB2 and file server are integrated on IBM System X3650 M2 with two Intel Xeon 5500 (Nehalem) processors (x3650).

External Sort Algorithm

The external sort algorithm is implemented based on the quick sort and merge algorithm. The quick sort algorithm [13] is considered to be one of the simplest and fastest algorithms, and the algorithm uses a divide-and-conquer method to sort data. The records to be sorted is segmented into two segments by choosing a comparison element (pivot value) so that all records whose key values are less than the pivot value is assigned to the first segment, and all records whose key values are greater than the pivot value is assigned to the second segment. The two segments are further segmented recursively using the same procedure until each segment consists of a single record only. The quick sort algorithm does not perform well when there is huge discrepancy in size between the two

newly created segments, and it is important that each step produces two segments of equal size.

The external sort algorithm is depicted in Figure 2. During the first pass, the input records are sent to the accelerator and collected sequentially into run buffers allocated in the main memory (typical run buffer size is 512 MB), and the run buffers are sorted concurrently. When each run buffer has enough records and is ready to be sorted, each run is split into sixteen short, fixed-length segments sub-runs (32 MB). The sub-runs contain a “sort” record generated from each input record. Unlike the input record which might be variable length, a “sort” record is fixed length (multiple of 8 bytes in length), and the key values (possibly multiple keys) are extracted from the input record and translated from floating point, character, decimal, etc values into unsigned integer values and packed into the sort record. Each sub-run is then quick sorted in parallel by sixteen threads, and the sixteen sub-runs are merged in parallel by sixteen threads. Using the pointers in the “sort” records, the Asynchronous Data Mover (ADM), which is one of accelerators on PowerEN™, is used to reorder the original input records into output buffers, output buffers are passed to the compression engine, and the compressed buffers are written to the file server. During the second pass, all the runs are retrieved from the file server and merged.

First, the first buffer (64 KB) from each run is read into memory, and each buffer is passed to the decompression engine. After the decompression is done, the last record in each buffer is examined and the smallest key value is determined. All the buffers are merged in parallel into a single (sorted) intermediate buffer. The intermediate buffers are now merged up to the smallest key value found, and the merged records are returned to the client. The aforementioned process of read and merge buffers from each run is repeated until the runs are exhausted.

External Hash Join Algorithm

The external join algorithm implementation is based on the hybrid hash join algorithm. Using the classic hash join algorithm, the joining of two input tables creates output records by combining columns of two input tables using the join predicate. When the key values are matched, the combined rows using each matched rows of the two tables are generated as matching results. The simple join algorithm requires that the hash table built using the smaller table fits into memory. The grace hash join algorithm [14] is introduced to handle the case when the memory available is too small to hold the hash table. The grace hash join algorithm partitions the two input relations so that each partition

can fit in the memory, and pairs of build and probe partitions are joined sequentially and independently. The hybrid hash join algorithm [15] is very similar to the grace hash join algorithm, but the hybrid hash join algorithm is more optimized one that takes advantage of the available memory. The major difference between the grace hash join algorithm and the hybrid hash join algorithm is that the records belong to the first build partition are used to build a hash table in the memory without being written to the disk, and the record belong to the first probe partition are used to probe the hash table to perform actual join of the first pair of partitions.

The external join algorithm is depicted in Figure 3. During the first pass, the input records are sent to the accelerator from either a record generator or DB2, and the records are collected in hash buffers allocated in the main memory, and the records in the hash buffers are hashed into a number of partitions concurrently using multiple threads. For example, when each hash buffer has enough records and is ready to be hashed, a thread is created, and the records in the buffer are hashed into a number of partitions using the thread. The number of partitions is pre-estimated using the estimated number of build records and estimated size of the records provided by the DB2 so that the hash table

fits into the main memory. Unlike external sort, the join algorithm requires two input tables (build table and probe table), and each input table is partitioned into the same number of disjoint partitions using the same hash function one after another. The hash function converts either a variable length character strings or 4 byte key values into 4 byte hash codes. The records having hash codes within the same range belong to the same partition as shown in Figure 3. The hashed records as well as hash codes are sent to output buffers corresponding to each partition, each output buffer is passed to the compression engine when it is filled with hashed records, and compressed records are stored the file server. On the second pass, the pairs of partition files (one build partition and one probe partition) are decompressed and loaded into the main memory, and joined recursively using multiple threads. Each partition will be sub-partitioned into a number of sub-partitions, and hence a joining of two large partitions becomes multiple joining of smaller sub-partitions using multiple threads, each thread processes joining of each sub-partition independently. The output of matched records are concatenated and returned to the client. This process of joining each pair of partition is repeated until the partitions are exhausted on the file server.

To reduce the amount of records stored on the file server, during the first pass, bloom filter is built while hashing the build records. Bloom filter is a bit filter representation of the set of keys which can be queried to check if a key is present. The corresponding bit for hashed joining key value of record is set while hashing each build record. Then while hashing probe records, the joining key is hashed using the same set of hash functions, and the filter is checked to see whether the corresponding bit was set. If the bit was not set, the probe record is filtered out, and is not stored in the file server. Use of bloom filter reduces the amount of records stored on the file server and reduces "hash probe" time dramatically especially when there is only small number of matches between two input tables.

DB2 Sort and Hash Join Accelerator

Sort and Hash Join are high demanding processes in terms of resources, and accelerating them will improve the overall performance of DB2. In our implementation the DB2 code is altered to export the sort and hash join processes to an external multi-core accelerator, where they are implemented in a highly parallel manner and the results are returned to DB2. In both cases, the DB2's internal sort and hash join implementations are bypassed. The SORT and HSJN processes are

intercepted in the insert and retrieval of rows and the data flow is directed to and from the accelerator. The changes to the DB2 code are limited to the SORT and HSJN components and do not affect the rest of DB2. The interface between the DB2 and the Accelerator is implemented in a shared library.

In the SORT process, the “insert” phase is intercepted and the record data is directed to the accelerator instead of the DB2 routine. When all the data is sent, the “fetch” routine is intercepted and sorted records are retrieved from the external accelerator instead of the internal buffers that are managed by DB2. Because there is no sorting or merging in DB2, memory allocation, and managing of temporary tables are all bypassed. The initial table processing and formatting the records, as well as unformatting the results and sending them to the user remain unaffected by the changes.

In the HSJN process, the routines that manage build and probe tables are intercepted and the records for both tables are sent to the accelerator. There is no intermediate retrieval of matched records from the accelerator. After the last row of the probe table is sent, and the hash join process is completed by the accelerator, all of the matched records

are retrieved. Similar to the sort process, because DB2 does not implement the hash join process there is no memory management or temporary storing of spilled records. Processing of the input table records and returning the results to the client are not affected.

Simulation Results

The simulation results of external sort and hybrid hash join algorithms are compared in Table 2 - Table 5. Table 2 compares performance (sort rate) of external sort algorithm executed on two platforms, DB2 and PowerEN™ PCIe card. Performance is measured in megabytes per second (MB/sec), and relative sort performance in depicted in Table 2 and Table 3. Table 2 shows the scalability behavior of multithreaded external sort algorithm when the number of threads increases, and Table 3 shows that the accelerator sorts the input records two times faster than DB2 sort. If four simultaneous sort requests are made to the accelerator, four concurrent sort operations are executed in the sort accelerator. Each sort operation successfully finishes each sort request, and the aggregated sort rate measured was about seven times faster than DB2 single sort rate (R_{DB2}).

Table 4 compares performance of external join algorithm executed in the join accelerator using various numbers of threads. Input table sizes of 26 GB and 30 GB are used for build and probe tables. The “hash build” is an execution time (T_B) to map build records into partitions, and the “hash probe” is an execution time (T_P) to map probe records into the same number of partitions as build partitions. It was tested that the build records and probe records are evenly distributed across the same number of partitions. Unbalanced record distribution across the partitions may diminish the performance of parallelized join operation. The partitioned data is compressed using a software compression module before stored on the file server (x3650). A record generator (x3650) is used to request a join operation to the join accelerator (x3650). The record generator generates TPC-H tables and streams input records through the 10 GE interface to the join server, and receives matched records from the join server. This implementation is to test the scalability behavior of multithreaded hash join algorithm when the number of threads increases. The test results show that most of the speedup is obtained during the second pass for the join (T_J), and the first pass (T_B and T_P) yields speedup less than that of the first pass. This is not surprising since bloom filter pre-filters out unmatched probe records and optimizes the first pass, and only small amount of

records are stored on the file server. Table 5 compares relative performance of DB2 hash join and external join operation executed on PowerEN™ PCIe card. The join server implemented on the PowerEN™ PCIe card uses hardware compression and decompression coprocessors to reduce data size stored on the file server. The test results show that the multithreaded join implementation on the PowerEN™ PCIe card accelerates DB2 hash join operation by a factor of two.

Conclusion

The goal of the work was to accelerate DB2 relational database operations which cannot be processed in a single pass of operation due to limitations on resources. We have concentrated on parallel implementation of external sort and external hash join algorithms using a PowerEN™ PCIe card developed by IBM as an accelerator, and integrated it with a host machine running DB2 as a client. In this paper, we have presented multi-threaded implementation of external sort and join operations to allow multiple threads to execute the operations concurrently. Our experimental results demonstrate that stand alone sort and join are good candidates to be offloaded to the accelerator and achieve almost by a factor of two acceleration of the DB2 performance with the help of compression and decompression engine in conjunction

with parallel implementation of the algorithms. One interesting extension of the work would be to offload more than a single operation to the accelerator so that TPC-H queries benefited from the acceleration even more.

References

[1] D. E. Knuth, The Art of Computer Programming, Vol. 3: *Sorting and Searching*, Addison-Wesley, 1973.

[2] A. S. Arefin and M. A. Hasan, "An Improvement of Bitonic Sorting for Parallel Computing," *Proceedings of the 9th WSEAS International Conference on Distributed Computing*, Athens, Greece, July 11-16, 2005.

[3] J. Chhugani, A.D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU architecture," in *Proceedings of the VLDB Endowment*, August 2008, pp 1313-1324.

[4] N. Ramprasad and P. K. Baruah, "Radix Sort on the Cell Broadband Engine," *International Conference on High Performance Computing (HiPC)*, 2007.

[5] M. Zagha and G. E. Blelloch, "Radix Sort for Vector Multiprocessors," *Proceedings Supercomputing*, November, 1991, pp 712-721.

[6] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *23rd IEEE Internal Parallel and Distributed Processing Symposium*, May 2009, pp 1-10.

[7] R. Parikh, "Accelerating Quicksort on the Intel Pentium 4 Processor with Hyper-Threading Technology," http://software.intel.com, 2008.

[8] S. Azadegan and A. Tripathi, "A Parallel Join Algorithm for SIMD Architectures," *Journal of Systems and Software*, Vol. 39, December 1997, pp 265-280.

[9] H. Lu, K. L. Tan, and M. C. Sahn, "Hash-based Join Algorithms for Multiprocessor Computers with Shared Memory," *Proceedings of the Sixteenth International Conference on Vary Large Database*, 1990.

[10] P. Garcia and H. F. Korth, “Database Hash-Join Algorithms on Multithreaded Computer Architectures,” *Proceedings of the 3rd Conference on Computing Frontiers,* 2006.

[11] T. P. Martin, P. A. Larson, and V. Deshpande, “Parallel Hash-Based Join Algorithms for a Shared-Everything Environment,” *IEEE Transactions on Knowledge and Data Engineering,* Vol. 6, October 1994

[12] D. P. LaPotin, S. Daijavad, C. L. Johnson, S. W. Hunter, K. Ishizaki, H. Framke, H.D. Achilles, D. P. Dumarot, N. A. Greco, and B. Davari, “Workload and Network-Optimized Computing Systems,” *IBM Journal of Research and Development,* Vol. 54, No. 1, January 2010.

[13] A.R. Hoare, Algorithm 64: Quicksort, Commun. ACM, 4(7):321, 1961.

[14] J. Goodman, “An Investigation of Multiprocessor Structures and Algorithms for Database Management,” *Technical Report UCB/ERL M81/33,* University of California, Berkeley, May 1981.

[15] D. J. Dewitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, “Implementation Techniques for Main Memory Database Systems,” *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data,* Vol. 14, June 1984.

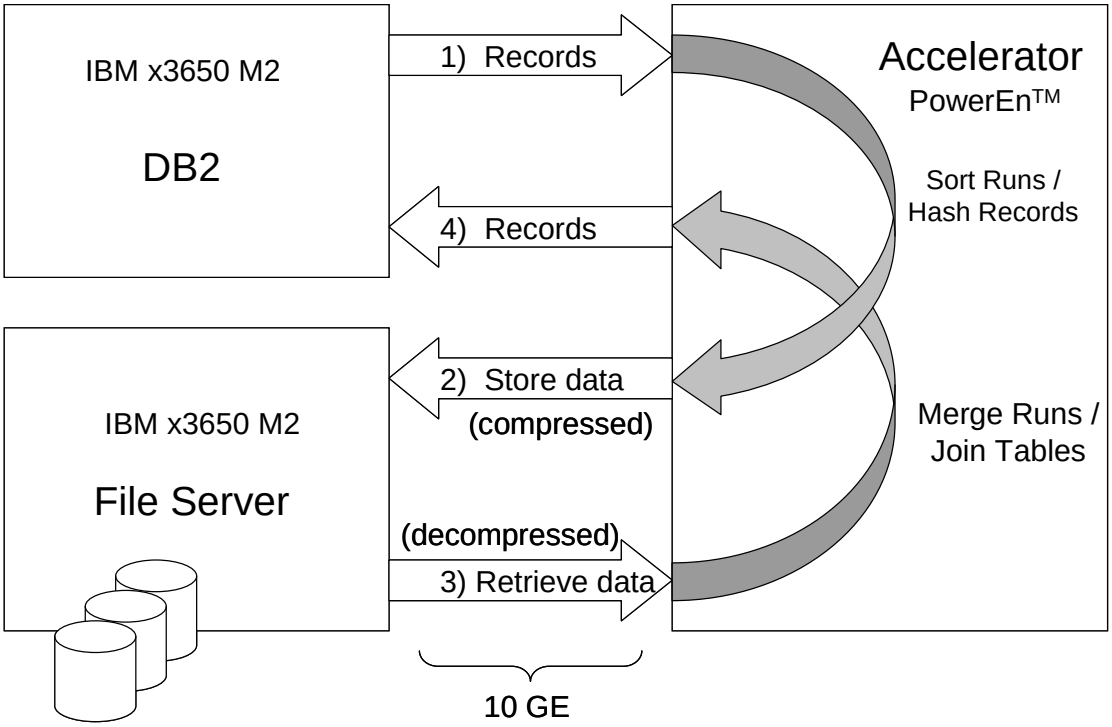

Figure 1 Overview of external sort and external join implementation using an accelerator.

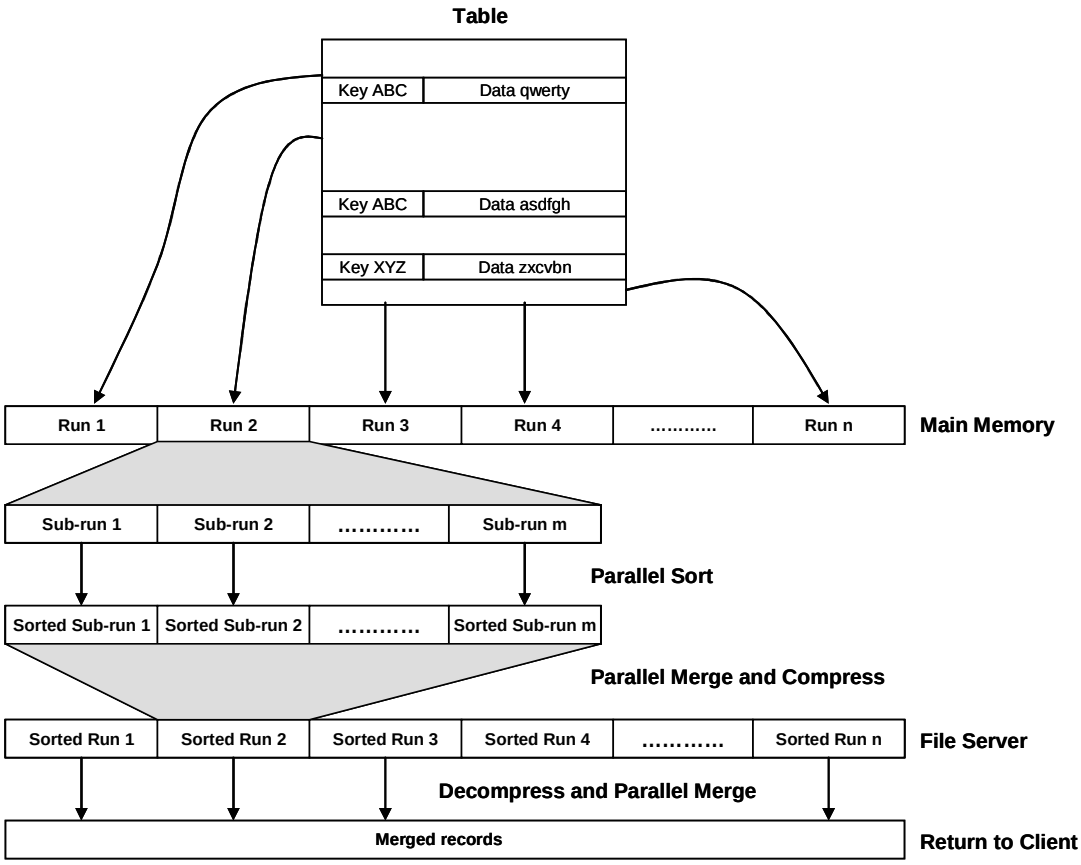

Figure 2 External sort algorithm

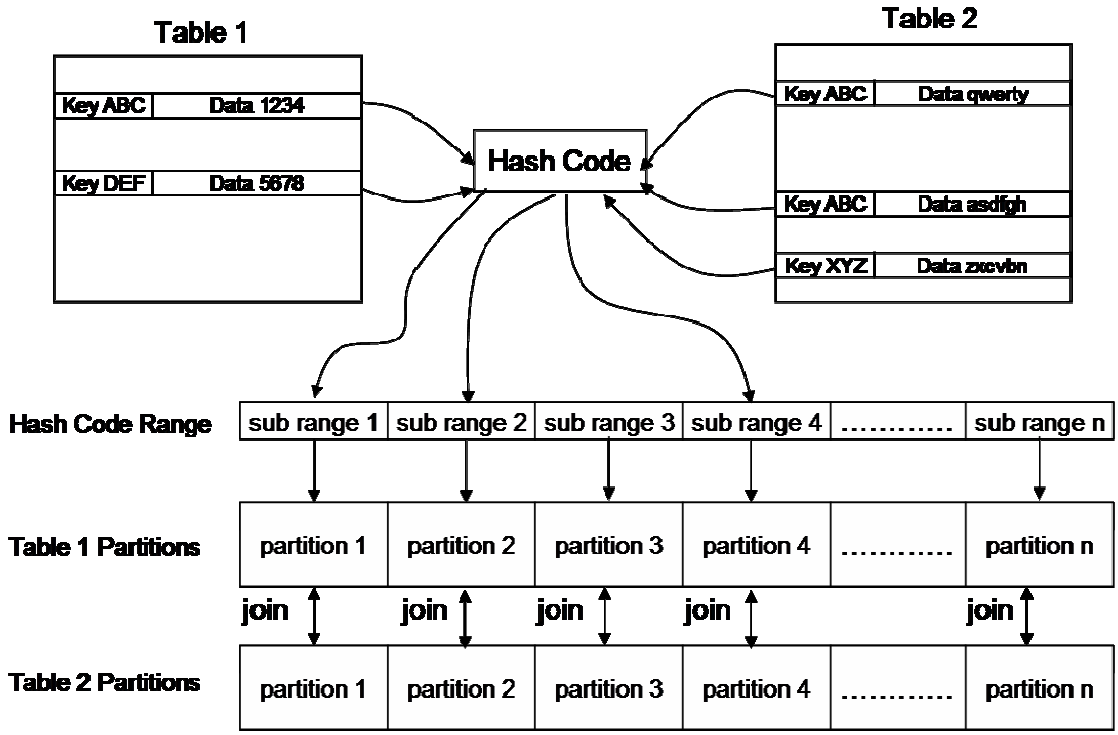

Figure 3 External join algorithm.

Table 1 Comparison of compression ratio using various TPC-H tables)

(GZIP on x3650 M2 vs hardware compression engine on PowerEN™)

TPC-H Tables	GZIP on x3650	Compression accelerator on PowerEN™
Customer	2.4	2.4
Orders	3.2	3.2
Lineitem (all)	3.2	3.2
Lineitem (2 cols)	2.9	2.9
Lineitem (3 cols)	5.1	5.0

Table 2 Test results of external sort algorithm (sort and merge) using different number of threads. T_S and T_M are execution times measured for each module with a single thread.

# threads	Sort	Merge
1	T_S	T_M
2	0.5 x T_S	0.5 x T_M
4	0.25 x T_S	0.26 x T_M
8	0.13 x T_S	0.13 x T_M
16	0.1 x T_S	0.07 x T_M
32	0.07 x T_S	0.05 x T_M

Table 3 Comparison of external sort algorithm (DB2 vs. accelerator). R_{DB2} is a sort rate measured on DB2. The compression was turned on DB2 and PowerEN™.

Sort Client	Sort Accelerator	# sort	Sort Rate
DB2	None	1	R_{DB2}
DB2	PowerEN™	1	2.11 x R_{DB2}
DB2	PowerEN™	4	7.22 x R_{DB2}

Table 4 Test results of external join algorithm using different number of threads (record generator, accelerator and file server implemented on x3650). T_B, T_P, T_J and T_{Total} are execution times measured for each module with a single thread.

# threads	Hash build	Hash Probe	Join	Total
1	T_B	T_P	T_J	T_{Total}
2	0.92 x T_B	0.93 x T_P	0.70 x T_J	0.82 x T_{Total}
4	0.77 x T_B	0.92 x T_P	0.36 x T_J	0.59 x T_{Total}
8	0.64 x T_B	0.92 x T_P	0.18 x T_J	0.45 x T_{Total}
16	0.5 x T_B	0.88 x T_P	0.10 x T_J	0.35 x T_{Total}
32	0.23 x T_B	0.92 x T_P	0.05 x T_J	0.22 x T_{Total}

Table 5 Comparison of test results of external join algorithm. $T_{PowerEN}$ is total join time measured on PowerEN™

Join Client	Join Accelerator	compression	Total Join Time
DB2	PowerEN™	hardware	$T_{PowerEN}$
DB2	None	software	1.89 x $T_{PowerEN}$