PermJoin: An Efficient Algorithm for Producing Early Results in Multi-join Query Plans

Justin J. Levandoski Mohamed E. Khalefa Mohamed F. Mokbel

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{justin,khalefa,mokbel@cs.umn.edu}

***Abstract*— This paper introduces an efficient algorithm for Producing Early Results in Multi-join query plans (*PermJoin*, for short). While most previous research focuses only on the case of a single join operator, *PermJoin* takes a radical step by addressing query plans with multiple join operators. *PermJoin* is optimized to maximize the early overall throughput and to adapt to fluctuations in data arrival rates. *PermJoin* is a non-blocking operator that is capable of producing join results even if one or more data sources are blocked due to slow or bursty network behavior. Furthermore, *PermJoin* distinguishes itself from all previous techniques as it: (1) employs a new flushing policy to write in-memory data to disk, once memory allotment is exhausted, in a way that helps increase the probability of producing early result throughput in multi-join queries, and (2) employs a novel *state manager* module that adaptively switches operators between joining in-memory data and disk-resident data in order to maximize overall throughput.**

I. Introduction

Traditional join algorithms (e.g., see [1], [2]) are designed with the implicit assumption that all input data is available beforehand. Furthermore, traditional join algorithms are optimized to produce the entire query result. Unfortunately, such algorithms are not suitable for emerging applications and environments that call for a new join algorithm design that is: (1) applicable in cases where input data is retrieved from remote sources through slow and bursty network connections and (2) optimized to produce early join results in a non-blocking manner while not sacrificing performance in processing the complete query result. Examples of such applications include web-based environments, where data is gathered from multiple remote sources and may exhibit slow and bursty behavior [3]. In addition, web users prefer early query feedback, rather than waiting an extended period for the complete result. Another vital example is scientific experimental simulation where experiments may take up to days to produce large-scale results. In such a setting, a join query should be able to function while the experiment is running, and not have to wait for the experiment to finish. Also, scientists prefer to receive early feedback from long-running experiments in order to tell if the experiment must halt and be restarted due to unexpected results [4]. Other applications that call for new non-blocking join algorithms that produce early results include streaming applications, workflow management, data integration, parallel databases, sensor networks, and moving object environments.

This work is supported in part by the Grant-in-Aid of Research, Artistry, and Scholarship, University of Minnesota, DTC Digital Technology Initiative Program, University of Minnesota, and DTC Intelligent Storage Consortium (DISC), University of Minnesota

Toward the goal of maintaining high result throughput in emerging environments, several research efforts have been dedicated to the development of non-blocking join operators (e.g., see [3], [5], [6], [7], [8], [9], [10]). However, with the exception of [6], these algorithms focus on query plans containing a single join operator. The main focus of these algorithms is to optimize for high throughput *locally* at each operator. Optimizing for *local* throughput does not necessarily contribute to the goal of maximizing *overall* throughput in a multi-join query plan. Considering the complete query plan calls for a new set of optimization techniques that maximize early throughput in multi-join query plans.

In this paper, we propose a new efficient non-blocking join algorithm for Producing Early Results in Multi-Join query plans (*PermJoin* for short). *PermJoin* goes beyond the idea of single-operator non-blocking join algorithms and explores a holistic approach to optimization techniques for maximizing throughput in multi-join query plans in new and emerging environments. To this end, *PermJoin* exploits a set of techniques that considers all join operators in a query plan, rather than viewing each operator as a separate entity. *PermJoin* employs the symmetric hash join algorithm [10] to join incoming data in memory. Once runtime memory is exhausted, *Permjoin* intelligently selects a percentage of data to be flushed to disk based on *expected* query throughput contribution. Furthermore, *PermJoin* manages the state of each join operator, alternating between memory and disk processing, in order to maximize the throughput of the multi-join query plan. It is important to note that *PermJoin* is capable of producing *complete* and *exact* query results, making it suitable for applications that do not tolerate approximations.

PermJoin distinguishes itself from all other non-blocking join algorithms in two *novel* aspects: (1) *PermJoin* employs a novel adaptive memory flushing technique that is triggered once memory is full. Unlike previous flushing techniques for non-blocking join algorithms, the flushing technique in *PermJoin* maximizes the *overall* query throughput for a multi-join query plan by *predicting* the throughput contribution of in-memory data. (2) *PermJoin* employs a novel *state manager* module, that has the ability to switch any join operator back and forth between joining in-memory data and disk-resident data based on the operation most beneficial to the query throughput. Such a *state manager* module does not exist in previous non-blocking join algorithms.

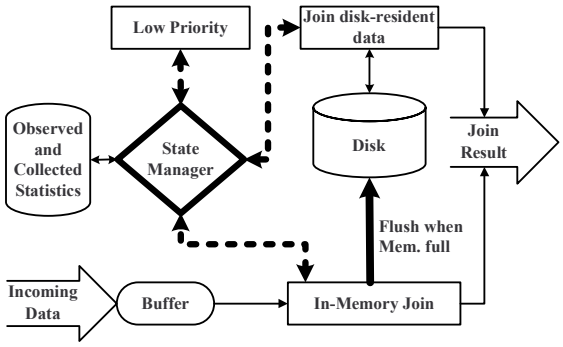


Fig. 1. Overview of the *PermJoin* Algorithm

II. Related work

The symmetric hash join [10] is the most widely used non-blocking join algorithm for producing early join results. However, it was designed for cases where all input data fits in memory. With the massive explosion of data sizes, several research attempts have aimed to extend the symmetric hash join to support disk-resident data [3], [5], [7], [8], [11]. These methods employ in-memory hash buckets to manage data. A percentage of these buckets are flushed to disk either individually [3], [8] or in groups [7], [9] when input data exhausts memory allocated for the query. All of these algorithms employ a symmetric hash join in order to achieve their non-blocking behavior. The main difference between them lies in the *flushing algorithm* used to free memory and maintain a high throughput. However, these techniques focus only on the case of a single join operator with no applicable extension for multi-join query plans. *PermJoin* employs a novel flushing algorithm that explicity considers the *expected* overall contribution of data for *multiple* join operators in a query plan to maximize throughput in real-time environments.

The closest work to *PermJoin* is the state-spilling method [6]; it is, to the authors' knowledge, the only work that addresses multi-join query plans. The basic idea behind state spilling is to score each hash partition group (i.e., pairs of corresponding hash buckets) in the query plan based on its past contribution to the query result. When memory is full, the partition group with the lowest score is flushed to disk. *PermJoin* distinguishes itself from this approach in two novel respects. First, *PermJoin* employs a novel flushing technique that takes into account both the *input* and *output* characteristics at each join operator. Using this information, the flushing algorithm predicts the contribution of data residing in the hash buckets. Second, *PermJoin* employs a novel *state manager* that switches each operator in the query plan between joining in-memory and disk-resident data in order to maximize the overall query throughput.

III. PermJoin: An Overview

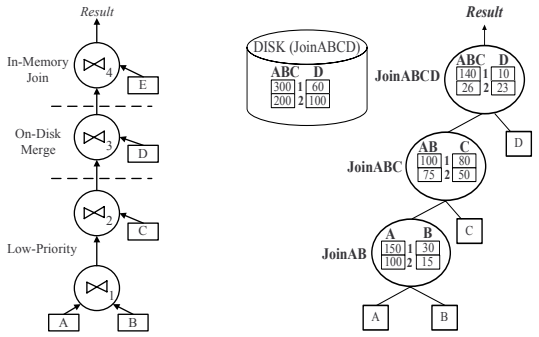
This section gives an overview of the *PermJoin* algorithm, centered around the following two novel methods: (1) A new *memory flushing* algorithm, designed with the goal of evicting data from memory that will contribute to the result throughput the *least*, and optimized for *overall* (rather than local) early result throughput. (2) A *state manager* module designed with the goal of placing each operator in a state that will positively affect result throughput. Each operator can function in an *in-memory*, *on-disk*, or *blocking* state.

Figure 1 gives a state diagram for *PermJoin* with its novel aspects highlighted in bold. In this work, we consider left-deep query plans with m binary join operators ($m > 1$), as depicted in Figure 2.As depicted in the diagram, whenever a new tuple R_S is received by the input buffer from source S of operator O, the state manager determines how the tuple is processed. If O is currently *not* in memory, then R_S will be temporarily stored in the buffer until O is brought back to memory. Otherwise, R_S will be immediately used to produce early results by joining it with in-memory data. Initially, all joins operate in memory. Once memory becomes full, *PermJoin* frees memory space by flushing a percentage of in-memory data to disk (depicted by the bold line from in-memory join to disk in Figure 1). The core of the *PermJoin* algorithm includes the *state manager* module (depicted by a bold diamond in Figure 1). The *state manager* is a continuously running thread that aims to place each join operator in one of three states (depicted by dotted bold lines): (1) Joining in-memory data, (2) Joining disk-resident data, or (3) Low Priority, i.e., producing results only if resources are available. At any time, the *state manager* may opt to switch a particular join operator from one state to another. In general, *PermJoin* consists of the following four components:

Memory Flushing. In *PermJoin*, the symmetric hash join [10] is used to produce early results in online environments. If memory allotment for the query plan is exhausted, data is flushed to disk to make room for new input, thus continuing the production of early results. We propose the *Adaptive-GlobalFlush* algorithm that aims to produce a high early overall throughput for multi-join query plans. The *Adaptive-GlobalFlush* policy flushes partition *groups* simultaneously (i.e., corresponding hash partitions from both hash tables). The main idea behind *AdaptiveGlobalFlush* is to consider partition groups across *all* join operators in concert, by iterating through all possible groups, scoring them based on their *expected* contribution to the overall result throughput, and finally flushing the partition group with the *lowest* score, i.e., the partition group that is *expected* to contribute to the overall result throughput the *least*.

To accomplish its goal, the *AdaptiveGlobalFlush* algorithm takes into account the following three characteristics of data in the query plan: (1) *Global Contribution* of the data, i.e., the ability to contribute to the overall throughput, (2) *Data Arrival Patterns*, i.e., changes in data arrival rates to the query plan, and (3) *Data Properties*, i.e., join attribute distribution or whether the data is sorted. A main drawback of previous flushing techniques is the inability to consider all three properties [6], [7], [8]. The *AdaptiveGlobalFlush* algorithm overcomes this drawback by considering all three properties.

State Manager. The main responsibility of the state manager is to place each join operator in the most beneficial state in terms of maximizing the overall query throughput. As a motivating example for the *state manager*, consider the query pipeline in Figure 2(a) with four interconnected join operators:



(a) State Example (b) Memory and Disk State

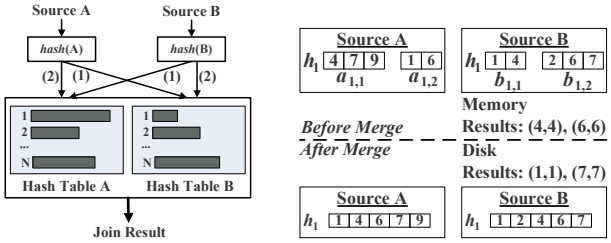
Fig. 2. Multi-Join Plan Examples

a base operator $Join_1$ with inputs sources A and B, an operator $Join_2$ with inputs AB (the output of $Join_1$) and source C, an operator $Join_3$ with inputs ABC and source D, and a root operator $Join_4$ with inputs $ABCD$ and source E. During query runtime, sources A and B may be transmitting data, while sources C, D, and E are blocked. In this case, query results can only be generated from the base operator $Join_1$. The overall query results produced by $Join_1$ rely on the selectivity of the three operators above in the pipeline (i.e. $Join_2$, $Join_3$, and $Join_4$). If the selectivity of these operators is low, merging disk-resident data at either $Join_3$ or $Join_4$ may be more beneficial in maximizing the overall query throughput than performing an in-memory join at the base operator $Join_1$. Thus, the *state manager* may decide to place $Join_4$ in the in-memory (i.e., default) state, $Join_3$ in the on-disk merge state, while $Join_1$ and $Join_2$ are placed in a low-priority state.

The *state manager* accomplishes its task by invoking a daemon process that traverses the query pipeline from top to bottom. During this traversal, it attempts to determine the operator O closest to the root that will produce a higher overall throughput in the on-disk state compared to its in-memory state. If this operator O exists, it is immediately directed to its on-disk merge state to process results. Meanwhile, all operators below O in the pipeline are directed to temporary block while all operators above O in the pipeline continue processing tuples in memory.

The design rationale behind the *state manager* is fundamentally different than that of the *AdaptiveGlobalFlush*. With *AdaptiveGlobalFlush*, the goal is to predict the least valuable in-memory data to flush to disk in order for the query to run effectively. Due to the changing nature of the query during runtime, data once determined to be the least beneficial by the flushing algorithm may turn out to be valuable *later* in runtime. For example, in Figure 2(b), the data on disk at $JoinABCD$ may be able to produce a higher result throughput than keeping all operators in their in-memory states. Thus, the rationale behind the *state manager* is to implement an *efficient* algorithm in order to *find* and *use* disk-resident data that becomes beneficial to the overall throughput.

In-Memory join. *PermJoin* employs the symmetric hash join algorithm to produce early join results [10]. Figure 3(a) gives the main idea of the symmetric hash join. Each input source



(a) Symmetric Hash Join (b) On-Disk Merge

Fig. 3. In-Memory Join and On-Disk Merge

(A and B) maintains a hash table with hash function h and n buckets. Once a tuple r arrives from input A, its hash value $h(r)$ is used to probe the hash table of B and produce join results. Then, r is stored in the hash bucket $h(r_A)$ of source A. A similar scenario occurs when a tuple arrives at source B. Symmetric hash join is typical in many join algorithms optimized for early results [3], [7], [10].

On-disk join. *PermJoin* operators in the disk merge state employ a disk-based sort-merge join to produce results (e.g., see [6], [7]). Figure 3(b) gives an example of the sort-merge join in which partition group h_1 has been flushed twice. The first flush resulted in writing partitions $a_{1,1}$ and $b_{1,1}$ to disk while the second flush resulted in writing $a_{1,2}$ and $b_{1,2}$. Results from joining disk-resident data are produced by joining (and merging) $a_{1,1}$ with $b_{1,2}$ and $a_{1,2}$ with $b_{1,1}$. Partitions $(a_{1,1}, b_{1,1})$ and $(a_{1,2}, b_{1,2})$ do not need to be merged, as they were already joined while residing in memory; the results produced from in-memory and on-disk operations are labeled in Figure 3(b). Flushing partition groups and using a disk-based sort-merge has the *major* advantage of not requiring timestamps for removal of duplicate results [6], [7]. Thus, once a partition group is flushed to disk it is not used again for in-memory joins.

REFERENCES

[1] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–170, 1993.

[2] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *TODS*, vol. 11, no. 3, pp. 239–264, 1986.

[3] T. Urhan and M. J. Franklin, "XJoin: A Reactively-Scheduled Pipelined Join Operator," *IEEE Data Engineering Bulletin*, vol. 23, no. 2, pp. 27–33, 2000.

[4] G. Abdulla, T. Critchlow, and W. Arrighi, "Simulation Data as Data Streams," *SIGMOD Record*, vol. 33, no. 1, pp. 89–94, 2004.

[5] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld, "An Adaptive Query Execution System for Data Integration," in *SIGMOD*, 1999.

[6] B. Liu, Y. Zhu, and E. A. Rundensteiner, "Run-time operator state spilling for memory intensive long-running queries," in *SIGMOD*, 2006.

[7] M. F. Mokbel, M. Lu, and W. G. Aref, "Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results," in *ICDE*, 2004.

[8] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis, "RPJ: Producing Fast Join Results on Streams through Rate-based Optimization," in *SIGMOD*, 2005.

[9] S. Viglas, J. F. Naughton, and J. Burger, "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources," in *VLDB*, 2003.

[10] A. N. Wilschut and P. M. G. Apers, "Dataflow Query Execution in a Parallel Main-Memory Environment," in *PDIS*, 1991.

[11] G. Luo, C. Ellmann, P. J. Haas, and J. F. Naughton, "A scalable hash ripple join algorithm," in *SIGMOD*, 2002.