On Producing High and Early Result Throughput in Multi-join Query Plans

Justin J. Levandoski, *Member, IEEE*, Mohamed E. Khalefa, *Member, IEEE* and Mohamed F. Mokbel, *Member, IEEE*

Abstract

This paper introduces an efficient framework for producing high and early result throughput in multi-join query plans. While most previous research focuses on optimizing for cases involving a single join operator, this work takes a radical step by addressing query plans with multiple join operators. The proposed framework consists of two main methods, a *flush algorithm* and *operator state manager*. The framework assumes a symmetric hash join, a common method for producing early results, when processing incoming data. In this way, our methods can be applied to a group of previous join operators (optimized for single-join queries) when taking part in multi-join query plans. Specifically, our framework can be applied by (1) employing 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) employing a state manager that adaptively switches operators in the plan between joining in-memory data and disk-resident data in order to positively affect the early result throughput. Extensive experimental results show that the proposed methods outperform the state-of-the-art join operators optimized for both single and multi-join query plans.

I. Introduction

Traditional join algorithms (e.g., see [1], [2], [3]) 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 require results as soon as possible. Such environments call

J.J. Levandoski, M.E. Khalefa, and M.F. Mokbel are with the Department of Computer Science and Engineering, Unversity of Minnesota - Twin Cities, 200 Union Street SE, Minneapolis, MN 55455. E-mail: {justin,khalefa,mokbel}@cs.umn.edu

```
SELECT  R.type,
        AVG(R.rating+C.service+Z.value)
FROM    YelpReview R,
        CityPageData C,
        ZagatData Z
WHERE   R.city="Minneapolis" AND
        R.id=M.id AND
        R.id=Z.id
GROUP   BY R.type
```

(a) Query

```
+-----------------------+
|Percent completed: 45% |
+-----------------------+
type      | avg | confidence | interval
----------+-----+------------+----------
Chinese    4.345     93        0.12432
Mexican    4.119     95        0.11982
French     4.543     95        0.14453
Sushi      3.413     95        0.14873
```

(b) Answer with online aggregation

Fig. 1. Data integration query

for a new non-blocking join algorithm design that is: (1) applicable in cases where input data is retrieved from remote sources through slow and bursty network connections, (2) optimized to produce as many *early* results as possible (i.e., produce a high early result throughput) in a non-blocking fashion, while not sacrificing performance in processing the complete query result, and (3) operates *in concert* with other non-blocking operators to provide early (i.e., online) results to users or applications. In other words, any blocking operation (e.g., sorting) must be avoided when generating of online results.

Numerous system environments require a non-blocking join algorithm capable of adapting to fluctuations of input arrival rates. As a prime example, data integration systems (e.g., Tukwila [4]) are databases built explicitly for processing queries over autonomous, remote (i.e., network-bound) data sources. A popular example of a remote data source are web services [5], where data is available through public a public web interface. Consider the query in Figure 1(a) in a data integration environment asking about the average aggregated rating score per restaurant type taken over the sum of three attributes from three different sources.

Note the following properties of this query. (1) The restaurant rating data is taken from three distinct web services transmitted over a network, i.e., overall rating from Yelp, service rating from City Pages, and value rating from Zagat. (2) The query necessitates a join between the *three* sources based on restaurant id (or any unique identifier). (3) The query does not use a blocking operation (e.g., sorting), but instead uses *online aggregation* [6], where the AVG operator is able to start computation as tuples stream into the operator. Figure 1(b) depicts the result of the online aggregation query, where averages are reported with a confidence and interval to reflect their accuracy. In order to function in a non-blocking manner, the online aggregation operator must receive tuples as soon as possible, meaning (a) the join operation must be non-blocking and (b) the join must ideally maintain a high throughput while dealing with a potentially slow

and bursty network connection, so that the answer accuracy reaches an acceptable threshold as soon as possible. We note online aggregation operators must also receive tuples in random order to guarantee statistical bounds. In most data integration environments, remote tuples are received in random order. Furthermore, the join feeding the online aggregation operator must produce result tuples in random order - a property followed by the methods we present in this paper.

Another vital motivating scenario is scientific experimental simulation. This scenario is particularly interesting as (1) some large-scale scientific data is modeled as a stream, meaning data may be generated on separate remote machines and transmitted to the query processor [7], and (2) experiments may take up to several days to produce large-scale results due to the sheer size of the input data. 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. Furthermore, scientists prefer to receive *early* feedback from long-running experiments in order to tell if the experiment must halt and be restarted with different settings due to unexpected results. These early results do not require a particular sort order [7], [8].

Other applications that necessitate the production of early results, and where are join framework is applicable, include streaming applications [9], [10], workflow management [11], parallel databases [12], spatial databases [13], sensor networks [14], [15], and moving object environments [16].

Toward the goal of producing a high early result throughput in new and emerging online environments as those just described, several research efforts have been dedicated to the development of non-blocking join operators (e.g., see [17], [6], [18], [13], [19], [20], [21], [10], [22]). However, with the exception of [18], these algorithms focus on query plans containing a *single* join operator. Thus, the optimization techniques employed by these join operators focus on producing a high early result throughput *locally*, implying that each operator is *unaware* of other join operators that may exist above or below in the query pipeline. Optimizing for *local* throughput does not necessarily contribute to the goal of producing a high *overall* early result throughput in multi-join query plans.

In general, the ability of join queries to produce high early result throughput in emerging environments lies in the optimization of two main tasks: (1) The ability to handle large input sizes by *flushing* the *least* beneficial data from memory to disk when memory becomes exhausted during runtime, thus allowing new input tuples to produce early results, and (2) The ability to

adapt to input behavior by producing results in a state (e.g., *in-memory* or *on-disk*) that *positively* affects early result throughput. In this paper, we explore a holistic approach to optimizing these two tasks for multi-join query plans by proposing two methods that consider join operators *in concert*, rather than as separate entities. The first method is a novel flushing scheme, named *AdaptiveGlobalFlush*, that intelligently selects a portion of in-memory data that is *expected* to contribute to result throughput the *least*. This data is written to disk once memory allotment for the query plan is exhausted, thus increasing the probability that new input data will help increase overall result throughput. The second method is a novel *state manager* module that is designed to fetch disk-resident data that will *positively* affect result throughput. Thus, the *state manager* directs each operator in the query pipeline to the most beneficial state during query runtime. These two methods assume that symmetric hash join, a common non-blocking join operator [18], [19], [22], is used to compute join results. This assumption makes the *AdaptiveGlobalFlush* method and the *state manager* module compatible with previous operators optimized for single-join query plans. Also, while these two methods are mainly designed to produce high early throughput in multi-join query plans, they also ensure the production of *complete* and *exact* query results, making them suitable for applications that do not tolerate approximations.

To the authors' knowledge, the state spilling approach [18] is the only work to have considered multi-join query plans. However, the work presented in this paper distinguishes itself from state-spilling and all other previous work through two *novel* aspects. First, the *AdaptiveGlobalFlush* algorithm attempts to maximize *overall* early result throughput by building its decisions over time based on a set of simple collected statistics that take into account both the *data input* and *result output*. Second, the *state manager* is a novel module that does not exist in previous work, and is designed specifically for multi-join query plans. The *state manager* has the ability to switch any join operator back and forth between joining *in-memory* data and *disk-resident* data during runtime based on the operation most beneficial for the set of pipelined join operators to produce early result throughput. When making its decision, the *state manager* module maintains a set of accurate, lightweight statistics that help in *predicting* the contribution of each join operator state. In general the contributions of this paper can be summarized as follows:

1) We propose a novel flushing scheme , *AdaptiveGlobalFlush* , applicable to any hash-based join algorithm in a multi-join query plan. *AdaptiveGlobalFlush* helps produce high early result throughput in multi-join query plans while being adaptive to the data arrival patterns.

2) We propose a novel *state manager* module that directs each join operator in the query pipeline to join either in-memory data or disk-resident data in order to produce high early result throughput.
3) We provide experimental evidence that our methods outperform the state-of-the-art join algorithms optimized for early results in terms of efficiency and throughput.

The rest of this paper is organized as follows: Section II highlights related work. Section III presents an overview of the methods proposed in this paper. The *AdaptiveGlobalFlush* method is described in Section IV. Section V presents the *state manager* module. Correctness of *AdaptiveGlobalFlush* and the *state manager* is covered in Section VI. Experimental evidence that our methods outperform other optimization techniques is presented in Section VII. Finally, Section VIII concludes the paper.

II. Related work

The symmetric hash join [22] 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. Such algorithms can be classified into three categories: (1) hash-based algorithms [4], [23], [19], [24], [20], [21] that flush in-memory hash buckets to disk either individually or in groups, (2) sort-based algorithms [17], [25] in which in-memory data is sorted before being flushed to disk, and (3) nested-loop-based algorithms [6] in which a variant of the traditional nested-loop algorithm is employed. Also, several methods have been proposed to extend these algorithms for other operators, e.g., MJoin [10] extends XJoin [21] for multi-way join operators while hash-based joins have been extended for spatial join operators [13]. However, these join algorithms employ optimization techniques that focus only on the case of a single join operator, with no applicable extension for multi-join query plans.

In terms of memory flushing algorithms, previous techniques can be classified to two categories: (1) Flushing a single hash bucket. Examples of this category include XJoin [21] that aims to flush the largest memory hash bucket regardless of its input source and RPJ [20] that evicts a hash bucket based on an extensive probabilistic analysis of input rates. (2) Flushing a pair of corresponding buckets from both data sources. Examples of this category include

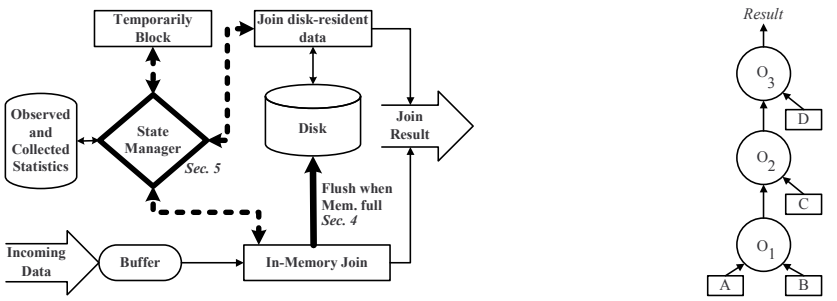
hash-merge join [19] that keeps the memory balanced between the input sources and state spilling [18] that attempts to maximize the overall early result throughput of the query plan. Our proposed *AdaptiveGlobalFlush* method belongs to the latter category, as flushing a pair of buckets *eliminates* the need for extensive timestamp collection. Timestamps are necessary when flushing a single bucket to avoid duplicate join results [20], [21].

A family of previous work can be applied to streaming environments where input data is potentially *infinite* (e.g., [26], [27], [28]). Such algorithms aim to produce an approximate join result based on either load shedding [29], [30] or the definition of a sliding window over incoming streams [31], [32]. Since our methods are concerned with *finite* input and *exact* query results, discussion of infinite data streams is outside the scope of this paper.

To the authors' knowledge the closest work to ours is PermJoin [33] and the state-spilling method [18]. PermJoin does not discuss specific methods for producing early results in multi-join query plans, only a generic framework. The basic idea behind state spilling is to score each hash partition group (i.e., symmetric hash buckets) in the query plan based on its current contribution to the query result. When memory is full, the partition group with the lowest score is flushed to disk. Once all inputs have *finished* transmission, the state-spill approach joins disk-resident data using a traditional sort-merge join. Our proposed *AdaptiveGlobalFlush* method differs from the state-spilling flush approach in that it is *predictive*, taking into account both input and output characteristics to make an optimal flush decision. Furthermore, our proposed *state manager* is not found previous work, as it continuously operates during *query runtime* to place operators in an optimal state with regard to overall throughput. State-spill and other previous work consider changing operator state only when sources block or data transmission has terminated.

III. Overview

This section gives an overview of the two novel methods we propose for producing a high early result throughput in multi-join query plans. These methods are: (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.



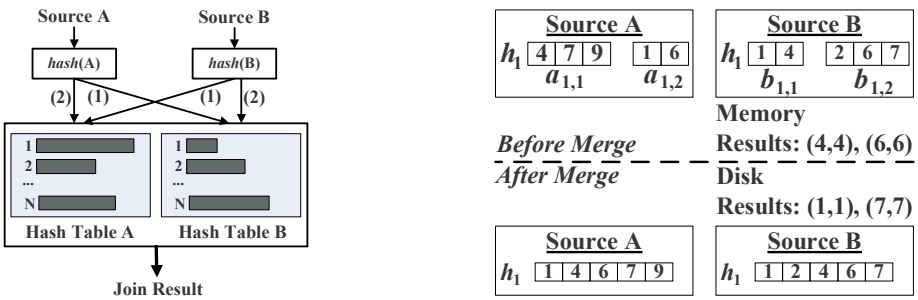
(a) Join Operation Overview

(b) Example Plan

Fig. 2. Join Overview and Example Query Plan

An overview of how our *memory flushing* algorithm and *state manager* can be added to existing non-blocking hash-based join algorithms is given in the state diagram in Figure 2(a). 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, the tuple 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 join operators function in memory. Once memory becomes full, the *memory flushing* algorithm frees memory space by flushing a portion of in-memory data to disk. During query runtime, each join operator may switch between processing results using either memory-resident or disk-resident data.

Memory Flushing. Most hash-based join algorithms optimized for early results employ a flushing policy to write data to disk once memory is full. In a multi-join query plan, policies that optimize for *local* throughput will likely perform poorly compared to policies that consider all operators together to optimize for overall throughput. For example, given the plan in Figure 2(b), if a local policy constantly flushes data from O_1, then downstream operators (O_2 and O_3) will be starved of data, degrading overall result throughput. We introduce a flushing policy, *AdaptiveGlobalFlush*, that can be added to existing hash-based join algorithms when they take part in a multi-join query plan. *AdaptiveGlobalFlush* opts to flush pairs of hash buckets, where flushing a bucket from an input source implies flushing the corresponding hash bucket from the opposite input at the same time. *AdaptiveGlobalFlush* evicts pairs of hash buckets that have the *lowest* contribution to the overall query output, using an accurately collected set of statistics



(a) Symmetric Hash Join

(b) On-Disk Merge

Fig. 3. Join and Disk Merge Examples

that reflect both the data input and query output patterns. Details of *AdaptiveGlobalFlush* are covered in Section IV.

State Manager. The main responsibility of the state manager is to place each join operator in the most beneficial state in terms of producing high early result throughput. These states, as depicted in Figure 2(a) by rectangles, are: (1) Joining in-memory data, (2) Joining disk-resident data, or (3) Temporary blocking, i.e., not performing a join operation. As a motivating example, consider the query pipeline given in Figure 2(b). During query runtime, sources A and B may be transmitting data, while sources C and D are blocked. In this case, query results can only be generated from the base operator O_1. The overall query results produced by O_1 rely on the selectivity of the two operators above in the pipeline (i.e. O_2 and O_3). If the selectivity of these operators is low, merging disk-resident data at either O_2 or O_3 may be more beneficial in maximizing the overall result throughput than performing an in-memory join at the base operator O_1. Thus, the *state manager* may decide to place O_3 in the in-memory (i.e., default) state, O_2 in the on-disk merge state, while O_1 is placed in a low-priority state. Details of the *state manager* are covered in Section V.

Architecture assumptions. We assume the in-memory join employs the symmetric hash join algorithm to produce join results [22]. Figure 3(a) gives the main idea of the symmetric hash join. Each input source (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 [19], [21], [22]. We also assume a join operator in the disk merge state employs a disk-based sort-merge join to produce results (e.g., see [18], [19]). 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 [18], [19]. Thus, once a partition group is flushed to disk it is not used again for in-memory joins.

In this work, we consider left-deep query plans with m binary join operators ($m > 1$) where the base operator joins two streaming input sources while all other operators join one streaming input with the output of the operator below. An example query plan is given in Figure 2(b). We choose to study left-deep plans as they are common in databases. Our methods can apply to bushy trees, where our flush algorithm and state manager can be applied to leaf nodes of the plan. Future work will involve adapting our methods to non-leaf nodes in bushy multi-join query plans. Further, an extension of our framework to the case of query plans consisting of more than one m-way join operator is straightforward.

IV. Memory Flushing

In online environments, most join operators optimized for early throughput produce results using symmetric hash join [19], [21], [22]. 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. In this section, we present the *AdaptiveGlobalFlush* algorithm that aims to produce a high early overall throughput for multi-join query plans. As discussed in Section III, the *AdaptiveGlobalFlush* 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*.

In general, a key success to any flushing policy lies in its ability to *produce a high result throughput*. In other words, the ability to flush hash table data that contributes the *least* to the result throughput. To avoid the drawbacks of previous flushing techniques [18], [19], [20], we identify the following three important characteristics that should be taken into consideration by any underlying flushing policy in order to *accurately* determine in-memory data to flush:

- ***Global contribution*** of each partition group. Global contribution refers to the number of *overall* results that have been produced by each partition group. In multi-join query plans, the *global contribution* of each partition is continuously changing. For instance, any non-root join in the plan is affected by the selectivity of join operators residing above in the operator chain. These selectivities may change over time due to new data arriving and possibly being evicted due to flushing. The ability to successfully *predict* the *global contribution* of each partition group is a key to the success of a flushing policy in producing result throughput.
- ***Data arrival patterns*** at each join operation. Due to remote data transmission through unreliable network connections, input patterns (i.e., arrival rates or delays) can significantly change the overall throughput during the query runtime. For example, higher data arrival rates imply that a join operator (and hence, its partition groups) will contribute more to the overall throughput. Likewise, a lower data arrival rate implies less contribution to the overall throughput. Thus, a flushing policy should consider the fluctuations of input patterns in order to *accurately predict* the overall throughput contribution of a partition group.
- ***Data Properties.*** Considering data properties is important as they directly affect the population, and hence the result production, for each partition group in a join operator. Such properties could be join attribute distribution or whether the data is sorted. For instance, if input data is sorted in a many-to-many or one-to-many join, a partition group j may contribute a large amount to the overall result throughput within a time period T as data in a sorted group (i.e., all have join attribute N) may all hash to j. However, after time T has passed, j may not contribute to the result throughput for a while, as tuples from a new sorted group (i.e., with join attribute $N + 1$) now hash to different partition K. Therefore, the flushing policy should be able to adapt during query runtime to data properties in order to *accurately predict* the population of a partition group.

Class	Statistic	Definition
Size	$prtSize_{Sj}$	Size of hash partition *j* for input *S*
	$grpSize_j$	Size of partition group *j*
	$tupSize_S$	Tuple size for input *S*
Input	$input_{tot}$	Total input count
	$unique_j$	No. unique values in part. group *j*
	$prtInput_{Sj}$	Input count at partition *j* of input *S*
Output	$obsLoc_j$	Local output of partition group *j*
	$obsGlo_j$	Global output of partition group *j*

TABLE I

STATISTICS

A. Adaptive Global Flush Algorithm

AdaptiveGlobalFlush is a novel algorithm that aims to flush in-memory hash partition groups that contribute the *least* to the overall result throughput. The decisions made by the *AdaptiveGlobalFlush* algorithm mainly depend on the three characteristics just discussed; namely, *global contribution*, *data arrival patterns*, and *data properties*. The main idea of the *AdaptiveGlobalFlush* algorithm is to collect and observe statistics during a query runtime to help the algorithm choose the *least* useful partition groups to flush to disk. Flushing the *least* useful groups increases the probability that new input data will continue to help produce results at a high rate. We begin by discussing the statistical model that forms the "brains" behind the decision made by *AdaptiveGlobalFlush* to evict data from memory to disk. We then discuss the steps taken by *AdaptiveGlobalFlush*, and how it uses its statistical model, when flushing must occur. Throughout this section, we refer to the running example given in Figure 4 depicting a pipeline query plan with two join operators, $JoinAB$ and $JoinABC$.

1) Statistics: The *AdaptiveGlobalFlush* algorithm is centered around a group of statistics observed and collected during the query runtime (summarized in Table I). These statistics help the algorithm achieve its ultimate goal of determining, and flushing, the partition groups with the least potential of contributing to the overall result throughput. In general, the collected statistics serve three distinct purposes, grouped into the following classes:

Size statistics. The size statistics summarize the data residing inside a particular operator at

any particular time. For each operator O, we keep track of the size of each hash partition j at each input S, denoted $prtSize_{Sj}$. In Figure 4, the operator $JoinAB$ has four partitions (i.e., two partition groups) with sizes $prtSize_{A1}$ = 150, $prtSize_{B1}$ = 30, $prtSize_{A2}$ = 100, $prtSize_{B2}$ = 15. We also track the size of each pair of partitions $grpSize_j$, this value is simply the sum of the symmetric partitions $grpSize_j$ = $prtSize_{Aj}$ + $prtSize_{Bj}$. For example, in Figure 4, the size of partition groups at $JoinAB$ are $grpSize_1$ = 150 + 30 = 180 and $grpSize_2$ = 100 + 15 = 115. Finally, we keep track of the tuple size for the input data at each source S, assuming the tuple size is constant at each join input. In Figure 4, we observe a tuple size of $tupSize_A$=10 and $tupSize_B$=8.

Input statistics. The *input* statistics summarize the properties of incoming data at each operator. Due to the dynamic nature of data in an online environment, the input statistics are collected over a time interval [t_i,t_j] ($i < j$) and updated directly after the interval expiration (we cover these maintenance in Section IV-B). Specifically, for each operator O we track the total number of input tuples received from all inputs, $input_{tot}$. Also, for each partition group j in operator O, we track the statistic $unique_j$ that indicates the number of unique join attribute values observed in j throughout the query runtime. Finally, for each input S and hash partition j, we track the number of tuples received at each partition j, $prtInput_{Sj}$. For the example in Figure 4, we assume the following values have been collected over a certain period of time: $input_{tot} = 100$, $unique_2 = 5$, $prtInput_{A2} = 6$, and $prtInput_{B2} = 4$.

Output statistics. Opposite of the input statistics, the *output* statistics summarize the results produced by the join operator(s). Like the *input* statistics, the *output* statistics are collected and updated over a predefined time interval. Specifically, for each partition group j in each join operator, we track two output statistics, namely, the *local output* ($obsLoc_j$) and the *global output* ($obsGlo_j$). In a pipeline query plan, the *local output* of a join operator O is the number of tuples sent from O to the join above it in the operator tree. In Figure 4, all tuples that flow from $JoinAB$ to $JoinABC$ are considered *local output* of $JoinAB$. The collected statistic $obsLoc_j$ in operator O is the number of tuples in the *local output* of O produced by partition group j. The collected statistic $obsGlo_j$ is the number of tuples in the *global output* (i.e., the *local output* of the root join operator, $JoinABC$ in Figure 4), produced by partition group j. To track this information, an identifier for the partition group(s) is stored on each tuple. Thus, each tuple stores a compact list of identifiers of the partition groups where it "lived" during query runtime. We

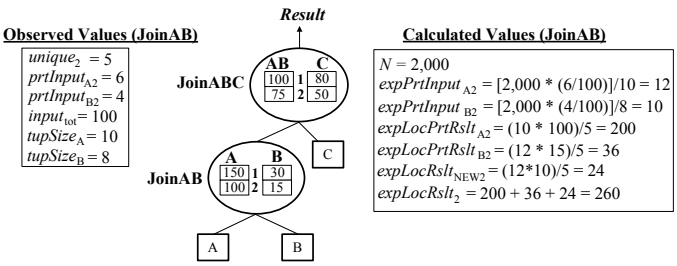


Fig. 4. Flush Example Plan

note that storing identifiers versus re-computing the join to find the original hash group (as done in [18]) is a trade-off between storage overhead and computation costs. We choose the former method, which is more suitable for larger tuples, whereas a hash group re-computation algorithm can be implemented for smaller tuples. Once a tuple is output from the root join operator, it is considered to contribute to the global output, and hence the observed global output values for all the partition group identifier(s) stored on that tuple are incremented by one.

2) AdaptiveGlobalFlush Algorithm: The *AdaptiveGlobalFlush* algorithm takes as input a single parameter N, representing the amount of memory that must be freed (i.e., flushed to disk). The idea behind *AdaptiveGlobalFlush* is to evaluate each hash partition group at each join operator and *score* each group based on its *expected contribution* to the global output. The algorithm then flushes the partition group(s) with the *lowest* score until *at least* the amount of memory N is freed and its data written to disk. We classify this process into four steps. In the rest of this section, we provide the ideas and intuition behind each step, and finish by providing a detailed look at the *AdaptiveGlobalFlush* algorithm.

Step 1: Input estimation. The *input estimation* step uses the *input statistics* (covered in Section IV-A1) in order to *estimate* the number of tuples that will arrive at each operator input hash partition group S_j until the next memory flush. Since each flushing instance aims to flush enough tuples to free an amount N of memory, we know the next flush instance will occur after tuples with a combined size of N enter the *entire* query plan. Thus, the goal of input estimation is to predict *where* (i.e., which partition group) each of these new tuples will arrive. We note that the size N encompasses both new tuples streaming into the query plan, and intermediate results generated by joins lower in the operator tree. Estimating the number of tuples that will arrive at each partition group is the first step in gaining an idea of how "productive" a each

group will be.

Step 2: Local output estimation. The *local output estimation* step uses the *size statistics* and *input estimation* from Step 1 in order estimate the contribution of each partition group to the *local output* of its join operator until the next flushing instance. The intuition behind this step is as follows. By tracking size statistics, we know, for each binary join operator, the size of each partition j for each input A and B. Further, through input estimation, we have an idea of how many new tuples will arrive at partition A_j and A_b. Thus, to calculate *local output*, this step estimates a partitions contribution to the *local output* as the combination of three calculations: (1) The output expected from the newly arrived tuples at partition A_j joining with tuples already residing in partition B_j. (2) The output expected from the newly arrived tuples at partition B_j joining with tuples already residing in partition A_j. (3) The output expected from the newly arrived tuples at partition A_j joining with the newly arrived tuples at partition B_j.

Step 3: Partition group scoring. The *partition group scoring* step uses the observed *output statistics* and the *local output estimation* from step 2 in order to score each partition group j across all operators O, where the lowerst score implies that j is the *least* likely to contribute to the overall *global* result output. The rationale for this score is to maximize the *global output* per number of tuples in the partition group. For example, consider two partition groups, P with size 20 and Q with size 10, each having an expected global output of 25. In this case, Q would be assigned a better score as it has a higher *global output* count per number of tuples. The intuition behind this approach is that using *local output estimation*, we know how many local results will be produced by each partition group j. Furthermore, through our observed output statistics, we can form a ratio $\frac{obsLoc_j}{obsGlo_j}$ that allows us to estimate how many of these local results will become global results. Finally, we can normalize this estimated global contribution by the partition size to derive a final score giving us the global output per partition size.

Step 4: Partition group flushing. The final partition group flushing step uses the scores from the *partition group scoring* in step 3 to flush partition groups to disk. This step simply chooses the *lowest* scoring partition group across *all* operators, and flushes it to disk. It iterates until tuples totaling the size N have been flushed to disk.

Detailed Algorithm. We now turn to the details behind *AdaptiveGlobalFlush*. Algorithm 1 gives the pseudo-code for the algorithm, where each step previously outlined is denoted by a comment. The algorithm starts by iterating over all partition groups within all operators (Lines 2

Algorithm 1 AdaptiveGlobalFlush Algorithm

1: **Function** AdaptiveGlobalFlush(*N*)
2: **for all** Operators O **do**
3: A ← O.sideA; B ← O.sideB; $SizeFlushed \leftarrow 0$
4: **for all** Partition Groups $j \in$ O **do**
5: /* Step 1: Input estimation */
6: $expPrtInput_{Aj} \leftarrow (N \cdot \frac{prtInput_{Aj}}{input_{tot}})/tupSize_A$
7: $expPrtInput_{Bj} \leftarrow (N \cdot \frac{prtInput_{Bj}}{input_{tot}})/tupSize_B$
8: /* Step 2: Local output estimation */
9: $expLocPrtRslt_{Aj} \leftarrow \frac{(prtSize_{Aj} \cdot expPrtInput_{Bj})}{unique_j}$
10: $expLocPrtRslt_{Bj} \leftarrow \frac{(prtSize_{Bj} \cdot expPrtInput_{Aj})}{unique_j}$
11: $expLocRslt_{NEWj} \leftarrow (\frac{expPrtInput_{Aj} \cdot expPrtInput_{Bj}}{unique_j})$
12: $expLocRslt_j \leftarrow (expLocPrtRslt_{Aj} + expLocPrtRslt_{Aj} + expLocRslt_{NEWj})$
13: /* Step 3: Partition group scoring */
14: $expGloRslt_j \leftarrow expLocRslt_j(\frac{obsGlo_j}{obsLoc_j})$
15: $grpScore_j \leftarrow \frac{expGloRslt_j}{grpSize_j}$
16: **end for**
17: **end for**
18: /* Step 4: Partition group flushing */
19: **while** $SizeFlushed \leq N$ **do**
20: $P_{Sj} \leftarrow$ partition group from input S with lowest $grpScore_j$
21: Flush P to disk
22: $SizeFlushed += sizeof(P_{Sj}) \times tupSize_S$
23: **end while**

to 4 in Algorithm 1).

Step 1 calculates an estimate for the number of new input tuples for each side S of the partition group j, denoted as $expPrtInput_{Sj}$. This value is calculated by first multiplying the *expected* input size of all tuples in the query plan N by the observed ratio that have hashed to side S of j, formally $prtInput_{Sj}/input_{tot}$. Second, this value is divided by the tuple size $tupSize_S$ for input S to arrive at the number of tuples. As an example, in Figure 4, assuming that N=2,000 the *expected* input for each partition in group 2 of $JoinAB$ is: $expPrtInput_{A2} = [2000 \times (6/100)]/10 = 12$ and $expPrtInput_{B2} = [2000 \times (4/100)]/8 = 10$, assuming a tuple size of 10 and 8 at input A and B, respectively.

Step 2 (Lines 9 to 12 in Algorithm 1) estimates a partition group's local output as follows. (1) The output from new tuples arriving at side B hashing with existing data in partition side A is computed by multiplying the two values $expPrtInput_{Bj}$ (from step 1) and $prtSize_{Aj}$ (a size statistic). However, to accommodate for the fact that a hash bucket j may contain up to $unique_j$

different values, we divide the computed value by $unique_j$. As an example, in Figure 4, this value for partition $A2$ at $JoinAB$ is $expLocPrtRslt_{A2} = (10 \times 100)/5 = 200$. (2) The output from new tuples arriving at side A hashing with existing data in partition side B is computed in a symmetric manner by exchanging the roles of A and B. For example, in Figure 4, this value for partition $B2$ at $JoinAB$ is $expLocPrtRslt_{B2} = (12 \times 15)/5 = 36$. (3) The output estimate from newly arrived tuples from both A and B is computed by multiplying $expPrtInput_{Aj}$ by $expPrtInput_{Bj}$ and dividing by $unique_j$. For example, in Figure 4, this value for partition group 2 at $JoinAB$ is $(12 \times 10)/5 = 24$. Finally, the *local output* estimation for partition group j, $expLocRslt_j$ is the sum of the previously three computed values. In the example given in Figure 4, this value for partition group 2 at $JoinAB$ is $expLocRslt_2 = 200 + 36 + 24 = 260$

Step 3 (Lines 14 to 15 in Algorithm 1) proceeds by first estimating the global output of partition group j (denoted $expGloRslt_j$) by multiplying $expLocRslt_j$ by the ratio $\frac{obsLoc_j}{obsGlo_j}$. For example, assume that in Figure 4, the global/local ratio for partition group 2 in $JoinAB$ is 0.5 (i.e. half of its *local output* contributes to the *global output*). Then the expected global output for this partition group is $expGloRslt_2 = 260/2 = 130$. Finally, the algorithm assigns a score ($grpScore_j$) to partition group j by dividing the *expected global output* by the size of the partition group (Line 15 in Algorithm 1).

In *Step 4* (Lines 19 to 23 in Algorithm 1), the algorithm flushes partition groups with lowest score, $grpScore_j$, to disk until the desired data size N is reached.

B. Scoring Accuracy

The quality of the *AdaptiveGlobalFlush* algorithm depends on the *accuracy* of the collected statistics. The *size* statistics (i.e., $grpSize_j$ and $prtSize_{Sj}$) are *exact* as they reflect the *current* cardinality that changes only when new tuples are received or memory flushing takes place. The *input* statistic $unique_j$ is *exact* as it reflects a running total of the unique join attributes that arrive to partition group j. Also, we assume the statistics $tupSize_S$ is static, i.e., that tuple size will not change during runtime. However, other *input* statistics (i.e., $prtInput_{Sj}$ and $input_{tot}$) and *output* statistics (i.e., $obsLoc_j$, and $obsGlo_j$) are observed between a time interval [t_i,t_j] ($i < j$). In this section, we present three methods, namely, *Recent*, *Average*, and *EWMA* for maintaining such *input* and *output* statistics. Figure 5 gives an example of each method for the statistic $obsLoc_j$ over four observations made from time t_0 to t_4.

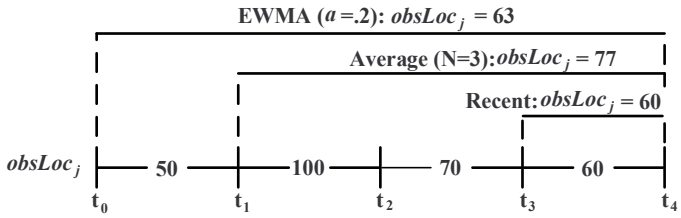


Fig. 5. Statistic Maintenance Sample

Recent. Using this method, only the most recent observation is stored. In Figure 5, the *Recent* method stores only the value 60 observed between time period t_3 to t_4. The main advantage of this method is ease of use as no extra calculations are necessary. This method is not well suited for slow and bursty behavior.

Average. In this method, the average of the last N observations is stored, where N is a predefined constant. In Figure 5, the *Average* method stores the average of the last $N = 3$ observations from time period t_1 to t_4, which is 77. The *average* method is straightforward as it considers the last N time periods to model the average behavior of the environment. It does not adhere to the fact that the recent behavior should weigh more than an old behavior.

Exponential Weighted Moving Average (EWMA). Using the *EWMA* method, each past observation is aged as follows: If the statistic at time period t_i has value $stat_i$ while at the time period $[t_i, t_{i+1}]$, we observe a new value $ObsVal$, then the value of the collected statistic is updated to be: $stat_{i+1} = \alpha * stat_i + (1 - \alpha) * ObsVal$ where α is a decay constant ($0 < \alpha < 1$). By choosing a smaller α, this method forgets past observations quickly and gives a greater weight to recent observations. The opposite is true for a larger α. In Figure 5, the *EWMA* method (with α=0.2) tracks all previous observations, but weights each new observation by a factor of 0.8 while weighting old observed values by 0.2. The *EWMA* method along with its parameter α gives more *accurate* model for statistic maintenance.

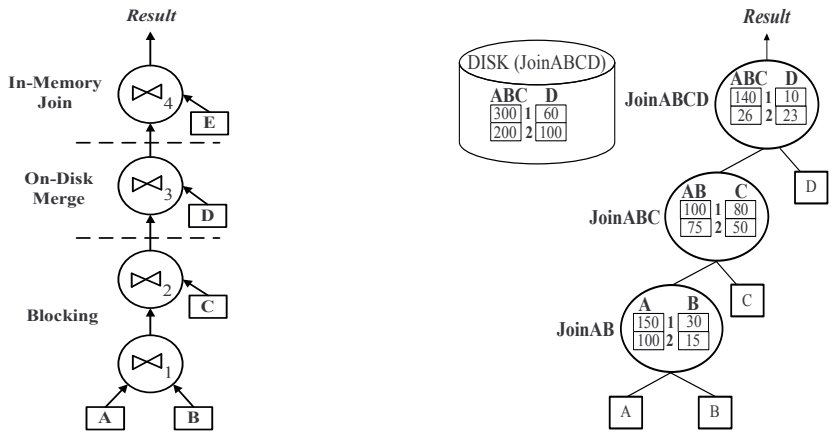
C. Discussion

The *AdaptiveGlobalFlush* algorithm is designed to produce a high overall throughput in multi-join query plans by considering the three main characteristics mentioned at the beginning of this section, namely, *global contribution*, *data arrival patterns*, and *data properties*. In terms of *global contribution*, the usage of the collected statistics $obsGlo_j$ (observed global output form j) and

$obsLoc_j$ (observed local output form j) enables us to continuously monitor the contribution of each partition group j. In terms of *data arrival patterns*, the usage of the statistic $prtInput_{Sj}$ (the number of tuples *expected* to arrive at partition j for source S) and $input_{tot}$ (the *expected* total number of input tuples) allow the algorithm to *accurately* model data arrival patterns and hence take them into account when *predicting* the global throughput of each partition group j. Finally, this same statistic enables the flushing algorithm to accommodate different *data properties*, as tracking input at the partition level allows the algorithm to predict *where* data will be arriving (i.e., hashing) in each hash table.

The statistics collected and calculated by *AdaptiveGlobalFlush* are designed to be lightweight due to its application in join processing. For instance, instead of calculating the *exact* intermediate input for all non-base join operations coming from operator output lower in the query plan, we track the input independently at each input using $prtInput_{Sj}$. By simplifying this statistical tracking, the flushing algorithm does not have to derive *correlation* between partition groups at each operator to calculate input to intermediate operators. As an example, consider a query plan with three joins as given in Figure 6(b). In order calculate the intermediate input to $JoinABCD$, we would need to derive the correlation between each partition group in $JoinAB$ to the partition groups in $JoinABC$ (i.e., where tuples in partition group n of $JoinAB$ will hash in $JoinABC$). Similarly, the correlation between groups in $JoinABC$ and $JoinABCD$ is needed. While calculating intermediate input this way would be more accurate than our approach, we feel the trade off between accuracy and algorithmic complexity is important in this case.

Comparison to State-of-the-Art Algorithms. Other flushing algorithms used by the state-of-the-art non-blocking join algorithms lack at least two of the three important aspects achieved by the *PermJoin*. For example, state-spilling flushing [18] only considers *global contribution* which is calculated based on the previous observed output, however, it does not adapt to *data arrival patterns* or *data properties* that affect the global throughput during the query runtime. The hash-merge join flushing algorithm [19] only considers *data arrival patterns* when attempting to keep hash tables balanced between inputs, however, *global contribution* and *data properties* are not taken into consideration. Finally, the RPJ flushing policy [20] adapts only to *data arrival patterns* and *data properties* by using a probability model to predict data arrival rates at each hash partition. However, *global contribution* is not considered by RPJ flushing. Furthermore, the RPJ policy is designed only for single-join query plans with no straightforward extension for



(a) State Example

(b) Memory and Disk State

Fig. 6. Multi-Join Plan Examples

multi-join plans.

V. The State Manager

While the goal of the *AdaptiveGlobalFlush* algorithm is to evict data from memory that will contribute to the result throughput the *least*, the goal of the *state manager* is to fetch disk-resident data that will positively affect result throughput. The basic idea behind the *state manager* is, at any time during query execution, to place each operator in one of the following states: (1) In-memory join, (2) On-disk merge, or (3) Temporarily blocking (i.e., waiting for a join operator above in the pipeline to finish its merge phase). The *state manager* constantly monitors the overall throughput potential of the *in-memory* and *on-disk* states for each join operator. Overall throughput of intermediate operators (i.e., non-root operators) refers to the number of intermediate results produced that propagate up the pipeline to form an overall query result, whereas results at the root operator *always* contribute to the overall query result.

The *state manager* accomplishes its task by invoking a daemon process that traverses the query pipeline from top to bottom. During traversal, it determines 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. Figure 6(a) gives an

example of a pipeline where the *state manger* finds that join operator $\bowtie_3$ is the closest operator to the root capable of maximizing the query global throughput when joining disk-resident data rather than in-memory data. Thus, the *state manger* directs $\bowtie_3$ to the on-disk state while all the operators above $\bowtie_3$ (i.e., $\bowtie_4$) are directed to join in-memory data and all operators below $\bowtie_3$ (i.e., $\bowtie_1$ and $\bowtie_2$) are temporarily blocked.

The main rationale behind the state-manager approach is threefold. (1) Top-down traversal serves as an *efficiency* measure by shortcutting the search for operators where disk-resident operations are more beneficial to overall throughput than in-memory operations. (2) Each operator in a query pipeline produces an overall throughput greater than or equal to the lower operators in the query plan. Thus, increasing the overall throughput of an operator higher in the pipeline implies an increase in the overall throughput of the multi-join query. For example, in Figure 6(a) the *state manager* places $\bowtie_3$ in the on-disk merge state without traversing the pipeline further, as the overall result throughput contributions of $\bowtie_1$ and $\bowtie_2$ are bounded by $\bowtie_3$, i.e., all intermediate results produced by $\bowtie_1$ and $\bowtie_2$ must be processed by $\bowtie_3$ before propagating further. Thus, the overall result throughput produced by in-memory operations at $\bowtie_1$ and $\bowtie_2$ cannot be greater than the overall in-memory throughput produced by $\bowtie_3$. (3) While performing on-disk merge operations multiple operators in the query plan (e.g., $\bowtie_3$ and $\bowtie_2$) could also benefit result throughput, we feel placing a single operator in its on-disk state is sufficient due to the rationale just discussed. Furthermore, a single operator performing an on-disk join *limits* context switches, random I/O, and seeks that can result when multiple requests for disk operations are present.

In the rest of this section, we present the functionality behind the state manager. We first provide the high-level state-manager algorithm. We then discuss the details behind the state manager's statistics and calculations. We finish with a discussion of the relative advantages of the state manager.

A. State Manager Algorithm

Algorithm 2 provides the pseudo code for the *state manager* module. As previously mentioned, the *state manager* algorithm performs a top-down traversal of all join operators in the pipeline. At each operator O, it compares the potential overall throughput for the in-memory and on-disk states. If the on-disk state is found to have a greater potential overall throughput at an operator O, the state manager directs O to the on-disk merge state immediately. All operators above O

Algorithm 2 State Manager

```
1:  Function StateManager()
2:  O ← root join operator , mergeGroup ← φ
3:  /* Stage 1 - Evaluate states */
4:  while mergeGroup = φ AND O ≠ φ do
5:      memThroughput ← memThroughput(O) {Sec V-B1}
6:      diskThroughput ← Max(dskThroughput(j)) by iterating through all disk partition groups j ∈ O {Sec V-B2}
7:      if ((diskThroughput × dskConst) > memThroughput) then
8:          mergeGroup ← diskThroughput.j
9:      else
10:         O ← next operator in pipeline
11:     end if
12: end while
13: /* Stage 2 - Direct operators */
14: if O = φ then
15:     for all Operators O3 in Pipeline do O3.state=IN-MEMORY
16: else
17:     O.merge = mergeGroup,O.state = ON-DISK
18:     for all Operators O1 above O do O1.state=IN-MEMORY
19:     for all Operators O2 below O do O2.state=BLOCK
20: end if
```

remain in the in-memory state, while all operators below O are directed to temporarily block. In general, the *state manager* executes in two main stages (1) *Evaluate states* and (2) *Direct operators*. We now outline the functionality of each stage.

Stage 1: Evaluate states. The *evaluate states* stage performs a top-down traversal of the query plan, starting at the root operator (Line 2 in Algorithm 2). For the example given in Figure 6(b), the algorithm would traverse from $JoinABCD$ down to $JoinAB$. The traversal ends when either an operator is found to switch to the on-disk state, or all operators have been visited (Line 4 in Algorithm 2). For the current operator O, the algorithm compares the overall throughput produced by the in-memory state to the throughput for the on-disk merge state (Lines 5 to 7 in Algorithm 2). Details behind in-memory and on-disk throughput calculations are covered shortly in Section V-B. This process begins by first calculating the in-memory throughput for O (Line 5 in Algorithm 2). The on-disk merge state can only process one disk-resident partition group at a time. Thus, when comparing the on-disk throughput to in-memory throughput, the algorithm iterates over all disk-resident partition groups j in O to find the j that will produce a maximal overall disk throughput (Line 6 in Algorithm 2). The algorithm then compares the

overall throughput for an on-disk merge to the overall in-memory throughput for operator O. Since producing join results from disk is much slower than producing join results from memory, the *state manager* accounts for this fact by multiplying the calculated disk throughput by a disk throughput constant, $dskConst$. The value $dskConst$ is a tunable database parameter, and represents the maximum amount of tuples per time period (e.g., 1 second) that can be read from disk (Line 7 in Algorithm 2). If the on-disk throughput is maximal, the algorithm will mark the disk partition group (Line 8 in Algorithm 2). By marking j, the algorithm ends the traversal process and moves to the next stage. If the algorithm does not mark a partition group, the traversal continues until all operators have been visited.

Stage 2: Direct operators This stage of the algorithm is responsible for directing each operator in the query pipeline to the appropriate state based on the analysis in stage 1. Stage 2 starts by checking if the traversal in stage 1 found an operator O to move to the on-disk state (Line 14 in Algorithm 2). If this is not the case, all operators in the query pipeline are directed to the in-memory state (Line 15 in Algorithm 2). Otherwise, we will direct O to its on-disk state where it is asked to merge the marked disk partition group from stage 1 (Line 17 in Algorithm 2). Also, all operators above O in the pipeline will be directed to the in-memory state while all operators below O in the pipeline will be directed to block (Lines 18 to 19 in Algorithm 2).

B. Throughput Calculation

From the high-level overview, we can see the core of the *state manager* is based on an accurate estimate of the throughput potential for the *on-disk* and *in-memory* states of each join operator in the query plan. We now cover the statistics and calculations necessary to estimate these throughput values.

1) Memory Throughput: The overall result throughput produced by the in-memory state at operator O can be *estimated* as the sum of all observed global output values for its partition groups. Formally, this *estimation* is calculated as: $memThroughput(O) = memConst(\sum_{j=1}^{|O|} obsGlo_j)$ where $|O|$ represents the number of partition groups in operator O and $obsGlo_j$ is the observed global output estimate over a predefined time interval (using a method from Section IV-B) for a partition group j in O (see Table I). Much like our disk constant *dskConst* presented in Stage 1 of the state manager algorithm, the memory throughput calculation uses a tunable parameter *memConst* that models the maximum number of results that can be produced in-memory over a

constant time period (e.g., 1 second).

2) Disk Throughput: The on-disk merge state allows only one disk-resident partition group to be merged at a time, meaning the overall disk throughput at an operator O is represented by the disk group that produces the *maximal* overall throughput when merged. Thus, the overall disk throughput *estimation* is done per disk-resident group.

Collected statistics. The disk throughput calculation relies on two *exact* statistics (i.e., they are not maintained using methods from Section IV-B). First, a *size* statistic that maintains, for each operator, the number of disk-resident tuples from input S for each partition j ($dskSize_{Sj}$). As with in the in-memory *size* statistics, $dskSize_{Sj}$ is an *exact* statistic as it represents the current cardinality. The other collected statistic maintains the current number of local results *expected* from an operator O if it were to merge a disk partition group j ($dskLocRslt_j$). The statistic $dskLocRslt_j$ is initially set to zero, *increases* with every flush of partition group j, and reset to zero when the disk partition group j is merged. Upon flushing partition group j (with sides A and B) to disk, the statistic $dskLocRslt_j$ is increased by the number of results *expected* to be produced by j which can be computed as follows: The flushed partition from side B ($partSize_{Bj}$; a collected in-memory *size* statistic), must eventually join with data *already* residing on disk (from a previous memory-flush) from side A ($dskSize_{Aj}$; a collected on-disk *size* statistic) while the flushed partition from A ($partSize_{Aj}$) joins with disk-resident data from B ($dskSize_{Aj}$). These estimations are possible because data is flushed in partition groups; new data being flushed from one partition side has not yet joined with disk-resident data from the other side. Since a partition group may contain tuples with multiple join attributes, the expected result must be divided by the unique number of join attributes observed in the partition group ($unique_j$; a collected in-memory *input* statistic). Formally, the statistic $dskLocRslt_j$ is increased by the following value when a flush occurs: $((dskSize_{Aj} \times partSize_{Bj}) + (dskSize_{Bj} \times partSize_{Aj}))/unique_j$. As an example, in Figure 6(b) (assuming $unique_j = 5$), if partition group 1 from $JoinABCD$ were to be flushed to disk, the statistic $dskLocRslt_1$ would be increased by: $((140 \times 60) + (300 \times 10))/5 = 2280$.

Overall disk throughput calculation Estimating the overall disk throughput for merging a disk partition group j involves two steps. The first step calculates the *expected* global output for the merge ($dskGloRslt_j$). Similar to the computation made by the *AdaptiveGlobalFlush* algorithm, we can derive this value by multiplying the *expected* local output ($dskLocRslt_j$; a collected on-disk statistic) by the observed global/local output ration ($obsGlo_j/obsLoc_j$; a collected in-

memory statistic). Formally, this equation is: $dskGloRslt_j = dskLocRslt_j \times (obsGlo_j/obsLoc_j)$. In Figure 6(b), the global/local output ratio at $JoinABCD$ is 1, as it is the root join operator. Therefore, if $dskLocRslt_1 = 2280$ for partition group 1 at $JoinABCD$, then $dskGloRslt_1$ is also 2280. The second step involves calculating a value for the number of global results expected per tuples read from disk. This calculation is similar to the scoring step of the *AdaptiveGlobalFlush* algorithm as we divide the expected global throughput by the size of the expected data read from disk. Formally, this equation for overall disk throughput is: $dskThroughput(j) = (dskGloRslt_j/(dskSize_{Aj} + dskSize_{Bj}))$. As an example, in Figure 6(b) (assuming $dskLocRslt_1 = 2280$), the expected disk throughput for partition group 1 at $JoinABCD$ is $dskThroughput(ABCD1) = (2280/(300 + 60)) = 6.33$. It is important to note that since producing join results from disk is much slower than producing join results from memory, the estimation for the throughput of an on-disk merge must be bounded by the time needed to read data from disk. In Section V-A, we described how the *state manager* accounts for this bound using the tunable parameter *dskConst* that represents disk throughput.

C. Discussion

One main goal of join algorithms optimized for early result throughput is the ability to produce results when sources block. When an operator is optimized for a *single-join* query plans, the decision to join data previously flushed to disk is straightforward as it is used to produce results *only* when *both* sources block [19], [20]. For multi-join queries, the state-spilling algorithm [18] does not consider using disk-resident data to produce results while input is streaming to the query. Disk-resident data is used by state-spilling in its *cleanup* phase, producing results when *all* input has terminated. Our proposed *state manager* goes beyond the idea of using disk-resident data only in the cleanup phase by invoking the disk merge operation when it is expected to increase the overall result throughput of the query. In this regard, the *state manager* is a novel algorithm for *multi-operator* query plans that considers the use of disk-resident data to maintain a high result throughput *outside* the cleanup phase.

If the *state manager* finds that the on-disk state can produce a greater overall throughput than the in-memory state at an operator O, it immediately directs O to enter its on-disk state. While this case is certainly true when *both* inputs to the operator O block, it may be true otherwise. For instance, input rates at operator O may slow down, decreasing its ability to produce overall

results. If the operator O is not a base operator, then one of these input rates is directly affected by the throughput of an operator located below in the pipeline. Thus, merging disk-resident data at operator O is beneficial if it is guaranteed to generate tuples that will increase the overall result throughput.

Also, we note that the design rationale behind the *state manager* is fundamentally different than that of the *AdaptiveGlobalFlush* algorithm presented in Section IV. 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 nature of the input data in emerging environments (i.e., remote transfer, no pre-processing), data that was flushed to disk at one point may turn out to be valuable at a *later* point during query runtime. Thus, the rationale behind the *state manager* is to implement an *efficient* algorithm that uses a top-down query plan traversal in order to *find* and *use* disk-resident data that becomes beneficial to the overall throughput.

Input Buffering. The *state manager* is able to place an operator in an *on-disk* or *blocking* state even if data is still streaming into the query plan at or below the on-disk operator. One consequence of this freedom is *buffer overrun*, i.e., exhausting the buffer memory for incoming tuples. One method to avoid buffer overrun is to ensure that switching operators to the on-disk and blocking phase will not cause buffer overrun. Using our collected statistics, we can easily perform this check. Let *BufSpace* be the available space in the buffer. Further, let *DskTime* be the time necessary to perform an on-disk merge. *DskTime* can be calculated as the time needed to read an on-disk partition group j. Using our disk size statistics, this value is $\frac{dskSize_{Aj}+dskSize_{Bj}}{dskConst}$, where *dskConst* is our tunable parameter modeling disk throughput. Also, recall from Section IV-A1 that we measure (over time interval [t_i,t_j]) the incoming count to each partition j for each join input S as $prtInput_{Sj}$. Let *InputSum* be the sum of all $prtInput_{Sj}$ values for partitions residing in join operators at or below the operator about to be placed in an on-disk state. Then, we can model the time until the next buffer overrun as $BufTime = \frac{(t_j-t_i)BufSpace}{InputSum}$. Then, the state manager can reliably place the operators in the on-disk and blocking states without the threat of a buffer overrun if the value *BufTime* is greater than *DskTime*.

Cleanup. Because the *AdaptiveGlobalFlush* algorithm and the *state manager* are designed for multi-join query plans, results that are *not* produced during the in-memory or on-disk phase require special consideration. In this vein, we outline a *cleanup* process for each operator that is compatible with our framework to ensure that a *complete* result set is produced. In this phase,

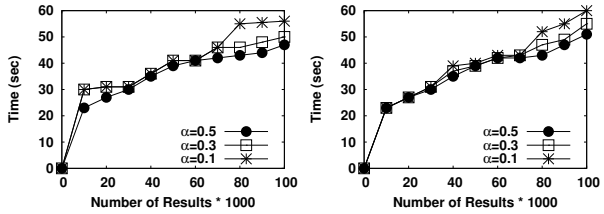
each operator begins cleanup by flushing all memory-resident data to disk after receiving an end-of-transmission (EOT) signal from *both* inputs. After this flush, all disk-resident partition groups are merged, producing a complete result set for that operator. The order of operator cleanup is important, due to the top-down dependency of pipeline operators. An example of this dependency is given in Figure 6(b). In order for $JoinABC$ to produce a complete result set, it must receive all input from source C and $JoinAB$ before beginning the cleanup phase. Cleanup order starts at the base operator and ends at the root of the query plan. To enforce cleanup order, we assume each operator in the chain can send an EOT signal to the next pipeline operator after completing its cleanup phase.

VI. Correctness

The correctness of both *AdaptiveGlobalFlush*, and the *state manager* can be mapped to previous work. In terms of producing *all* results, we use symmetric hash join to produce in-memory results. Further we flush partition *groups* and use a variance of sort-merge join to produce disk results. Using these methods ensures that a *single* operator will produce all results [18], [19]. Furthermore, using these methods also ensures that a single operator will produce a unique result *exactly* once in either the in-memory or on-disk states [19]. If the *base* operator in a query plan guarantees production of all results exactly once, the ordered *cleanup phase* ensures that all results from the base operator will propagate up the query plan. It follows that the this *cleanup phase* ensures that using our methods in a multi-join query plan will produce *all* results *exactly* once.

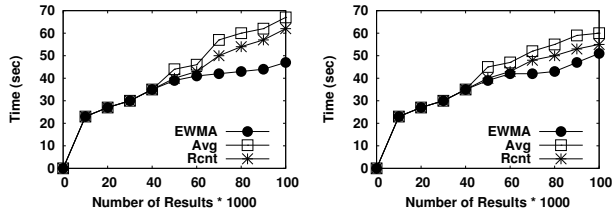
VII. Experimental Results

This section provides experimental evidence for the performance of the *AdaptiveGlobalFlush* algorithm and the *state manager*, by implementing them along with symmetric hash join [22]. We compare this implementation to optimizations used by the state-of-the-art non-blocking join operators optimized for multi-join queries (i.e., state-spill [18]) as well as single-join queries (i.e., Hash-Merge Join (HMJ) [19] and RPJ [20]). Unless mentioned otherwise, all experiments use three join operations with four inputs A, B, C, and D as given in Figure 6(b). All input arrival rates exhibit slow and bursty behavior. To model this behavior, input rates follow a Pareto distribution; widely used to model slow and bursty network behavior [34]. Each data



(a) $ABCD_B$ (b) ABC_B and D_S

Fig. 7. α Value



(a) $ABCD_B$ (b) ABC_B and D_S

Fig. 8. Maintenance methods

source contains 300K tuples with join key attributes uniformly distributed in a range of 600K values, where each join key "bucket" is assigned to a source with equal probability. Thus, given M total join key buckets assigned to each join input S, there will be X ($X < M$) overlapping join key buckets and ($M - X$) non-overlapping join key buckets at each operator. Memory for the query plan is set to 10% of total input size. For our propsed methods, statistical maintenance (see Section IV-A1) is performed every five seconds ($t = 5$), a value determined in initial tuning experiments not included for space reasons. For flushing policies, the percentage of data evicted is constant at 5% of total memory.

The outline of this section is as follows. Section VII-A studies the effect of different maintenance methods. Section VII-B studies performance in environments with different data input behavior. Section VII-C studies the memory needs for producing early results. Section VII-D tests different data flush sizes. Section VII-E studies the effect of join ratios. Finally, Section VII-F studies the scalability for query plans with a large number of joins.

A. Maintenance Methods

This set of experiments studies the effect of different maintenance methods on the performance of our methods. First, we study the effect of different α values when using the Exponential Weighted Moving Average (*EWMA*) method for managing statistics. Second, we study the impact of each maintenance method presented in Section IV-B.

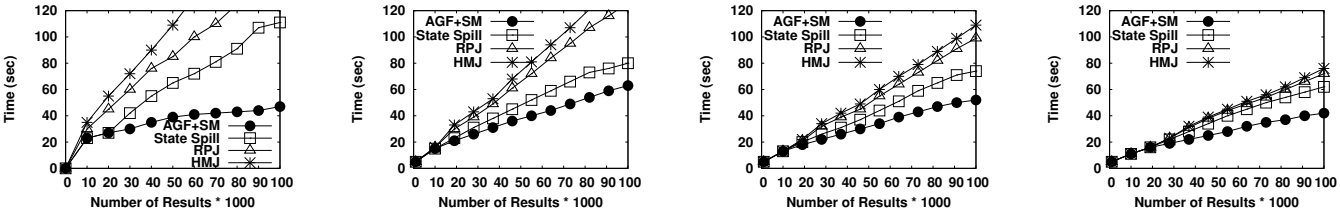
Different α Values. Figure 7(a) gives the performance of each α value for the default input behavior. Throughout the rest of this section, the subscript B represents slow/bursty input, while S represents a steady input at 200 tuples/sec. In this environment, the α value of 0.5 helps maintain a steady throughput performance to the first 500K results, while other values lead to

fluctuation in performance. Figure 7(b) gives the performance of each α value when inputs A, B, and C are slow and bursty, while D is steady. Here, the α value of 0.5 still provides the best performance. Due to this outcome, in rest of this section all experiments using the *EWMA* method will have a default α value of 0.5.

Different Maintenance Methods. Figure 8(a) gives the performance of each maintenance method for the default input behavior. In these experiments, the *average* method uses an N value of 5, while the *EWMA* method uses an α value of 0.5. The *EWMA* method gives the best performance as it produces a stable overall early throughput. The *average* and *recent* methods show relative performance degradation while producing the first 40K results. Figure 8(b) gives the performance of each method when inputs A, B, and C exhibit slow and bursty behavior, while input D is steady, confirming that the *EWMA* method delivers the most consistent performance.

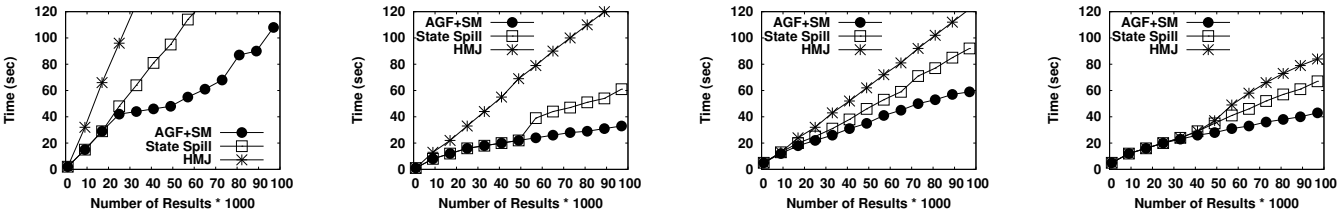
B. Input Behavior

Figure 9 gives the performance of *AdaptiveGlobalFlush* and the *state manager* (abbr. AGF+SM) compared to that of state-spill, HMJ, and RPJ when run in environments with four different input behaviors. In these tests, we focus on the first 100K results produced, corresponding to *early* query feedback. Figure 9(a) gives the results for the default behavior (all inputs slow/bursty), and demonstrates the usefulness of AGF+SM. *AdaptiveGlobalFlush* is able to keep beneficial data in memory, thus allowing incoming data to produce a better overall *early* throughput. Meanwhile the *state manager* helps to produce results due to the slow and bursty input behavior where sources may block. The state-spill flush policy does not adapt to the slow and bursty input behavior in this case, and thus flushes beneficial data to disk early in the query runtime. Also, state-spill does not employ a *state manager* that is capable of finding *on-disk* data at each operator to help produce results. State-spill only invokes its disk phase once *all* inputs terminate. HMJ and RPJ, optimized for single-join queries, base their flushing decisions on maximizing output *locally* at each operator. However, these flushing policies are unaware of how data at each operator is being processed by subsequent operators in the pipeline, causing tuples to flood the pipeline that are unlikely to produce results higher in the query plan. HMJ invokes its disk phase once *both* inputs to an operator block. However, in a multi-join query, this scenario will likely occur only at the *base* operator, as one input for each intermediate operator is generated from a join operation residing below in the pipeline. In this case, disk



(a) $ABCD_B$ (b) A_S,BCD_B (c) AB_S,CD_B (d) ABC_S,D_B

Fig. 9. Input Behavior



(a) $ABCD_B$ Mem 5% (b) $ABCD_B$ Mem 15% (c) AB_SCD_B Mem 5% (d) AB_SCD_B Mem 15%

Fig. 10. Query Memory Size

results from the base operator can be pushed upward in the pipeline that will not produce results at subsequent operators, and cause flushing to occur higher in the pipeline when inputs are blocking, also degrading performance. On the other hand, RPJ performs better than HMJ due to its ability to switch from memory to disk operations based on its statistical model that predicts local throughput. For multi-join queries, a *state manager* is needed to manage processing at all operators *in concert*.

Figures 9(b) through 9(d) give the performance of AGF+SM compared to state-spill HMJ, and RPJ for three, two, and one input(s) exhibiting slow and bursty behavior, respectively. In this left-right progression, as the the inputs exhibit more of a steady behavior, the disk phase is used less, allowing us to pinpoint the relative behavior of each flushing policy only. While the performance of each algorithm converges from left to right, AGF+SM consistently outperform state-spill by at least 30%, RPJ by 40%, and HMJ by 45%, in producing *early* results. AGF+SM also exhibit a constant steady behavior in both slow/bursty and steady environments. Finally, we note that these experiments show the benefit of optimization techniques for multi-join query plans, due to the performance of state-spill and AGF+SM compared to that of RPJ and HMJ. Also, due to the

(a) Flush Progression

(b) Flush 10%

Fig. 11. Data Flush Size

similar behavior of HMJ and RPJ compared to state-spill and AGF+SM, subsequent experiments use *only* HMJ to represent algorithms optimized for single-join query plans.

C. Memory Size

Figure 10 gives the time needed for AGF+SM, state-spill, and HMJ to produce 100K results for query plan memory sizes 5% above and below the default memory size (i.e., 10%) for two different environments. Figures 10(a) and 10(b) give the performance for inputs exhibiting the default behavior, while figures 10(c) and 10(d) give the performance for inputs A and B exhibiting steady behavior while C and D are slow and bursty. For smaller memory (Figures 10(a) and 10(c)), flushing occurs earlier causing more flushes during early result production. In this case, *AdaptiveGlobalFlush* is able to *predict* beneficial in-memory data and keep it in memory in order to maintain a high early overall throughput. Furthermore, when all inputs are slow/bursty (Figure 10(a)), the *state manager* helps in early result production. With larger memory (Figures 10(b) and 10(d)), more room is available for incoming data, hence less flushing occurs early, meaning the symmetric-hash join is able to process most early results when memory is high. For both input behaviors, state-spill and AGF+SM perform relatively similar early on when more memory is available. Meanwhile, HMJ performs relatively better for a larger memory sizes, but its flushing and disk policy are still a drawback in multi-join query plans when inputs are slow/bursty compared to a steady environment.

D. Data Flush Size

Figure 11 give the performance for different data flush sizes. Figure 11(a) shows the time needed to produce the first 100K results when smaller and larger percentages of total memory

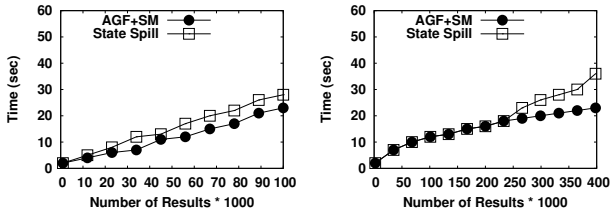
are evicted per flush operation, while Figure 11(b) focuses on the specific case of 10% data flush. When the percentage is very low (1%), flushes occur often, causing delays in processing. In general, writing *less* data to disk (in a reasonable range) during a flush operation leads to a greater amount of early results as more data is available in-memory for arriving tuples. However, more overall flush operations are performed during query runtime leading to a large number of buckets on disk with low cardinality, which is less beneficial to the disk-merge phase. Writing *more* data to disk during a flush *should* lead to less amount of early results as less in-memory data is present for arriving tuples, while less flushes will be performed overall. Both state-spill and HMJ produce less early results in this case. However, AGF+SM performance remains stable. The reason for this performance is mainly due to the *state manager*. When *AdaptiveGlobalFlush* is required to write *more* tuples to disk at every flush, the *state manager* is able to quickly reclaim beneficial data to produce early results.

E. Join Ratio

Figure 12(a) gives the time period for the *PermJoin* and state-spill algorithms in producing the first 100K results with a small join ratio of 1:2 at each operator, meaning the output at each operator is double the size of its input. The input behavior is set to *steady* for all inputs, to test each algorithm's reaction to join ratio. The *PermJoin* algorithm performs well due to the ability of the *AdaptiveGlobalFlush* algorithm to *predict* beneficial partition groups and keep them in memory, thereby filling the query pipeline with data expected to aggregate to the overall result. The state-spill algorithm has lower relative performance, as partition groups at each operator generally contribute less to the overall early throughput. Figure 12(b) gives the time period for both algorithms in producing the first 400K results with a large join ratio (1:4). We focus on 400K results here as a larger join ratio implies a large number results will be produced early. In this case, *PermJoin* and state-spill perform relatively well through the first 200K results. Since flushing occurs often with a large join ratio, the difference in the *PermJoin* and state-spill flushing algorithms can be seen after the first 200K results.

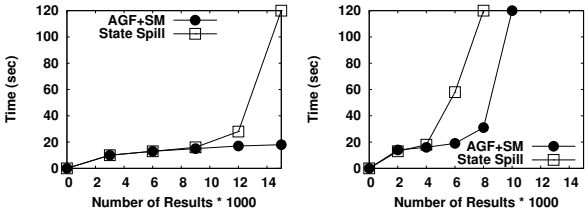
F. Scalability

Figure 13 gives the time needed for AGM+SM and state-spill to produce the first 15K results in 5-join (6 inputs) and 6-join (7 inputs) query plans. In these tests, HMJ was omitted as



(a) Small Join Ratio (b) Large Join Ratio

Fig. 12. Join Ratio



(a) 5-Join Query (b) 6-Join Query

Fig. 13. Number of Joins

no results were produced in the first 120 seconds. Due to the higher selectivity from more join operations, we plot the first 15K results as cleanup begins earlier (steep slopes in figures) than in previous experiments. Here, AGF+SM is able to delay the cleanup phase longer. If the *AdaptiveGlobalFlush* algorithm makes an incorrect decision and flushes beneficial data at an intermediate operator, the *state manager* is able to propagate these beneficial tuples by placing each operator in the in-memory or on-disk state in order to produce more results before cleanup. The state-spill algorithm does not make use of disk-resident data while producing these early results. Furthermore, state-spill relies only on its flushing algorithm that scores partition groups based on global vs. local contribution over size. This scoring method generally does not perform well with a large number of joins where selectivity is high in slow/bursty environments. Here, the ability to *predict* beneficial data in both the flush algorithm and disk states is important. Thus, state-spill exhibits lower relative performance in these cases.

VIII. Conclusion

This paper introduces a framework for producing high early result throughput in multi-join query plans. This framework can be added to existing join operators optimized for early results by implementing two novel methods. First, a new flushing algorithm, *AdaptiveGlobalFlush*, flushes data *predicted* to contribute to the overall early throughput the *least* by adapting to important input and output characteristics. This adaptation allows for considerable performance improvement over the methods used by other state-of-the-art algorithms optimized for early results. Second, a novel module, termed a *state manager*, adaptively switches each operator in the multi-join query plan to process results in two states (in-memory or on-disk) to maximize the overall early throughput. The *state manager* is a novel concept, as it is the first module to consider the efficient use of

disk-resident data in producing early results in multi-join queries. Extensive experimental results show that the proposed methods outperform the state-of-the-art join operators optimized for both single and multi-join query plans in terms of efficiency, resilience to changing input behavior, and scalability when producing early results.

REFERENCES

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

[2] P. Mishra and M. H. Eich, "Join Processing in Relational Databases," *ACM Computing Surveys*, vol. 24, no. 1, pp. 63–113, 1992.

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

[4] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld, "An Adaptive Query Execution System for Data Integration," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1999.

[5] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query Optimization over Web Services," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2006.

[6] P. J. Haas and J. M. Hellerstein, "Ripple Joins for Online Aggregation," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1999.

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

[8] J. Becla and D. L. Wang, "Lessons Learned from Managing a Petabyte," in *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2005.

[9] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent Streaming Through Time: A Vision for Event Stream Processing," in *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2007.

[10] S. Viglas, J. F. Naughton, and J. Burger, "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

[11] D. T. Liu, M. J. Franklin, G. Abdulla, J. Garlick, and M. Miller, "Data-Preservation in Scientific Workflow Middleware," in *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 2006.

[12] D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Commununications of ACM, CACM*, vol. 35, no. 6, pp. 85–98, 1992.

[13] G. Luo, J. F. Naughton, and C. Ellmann, "A Non-Blocking Parallel Spatial Join Algorithm," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2002.

[14] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid, "Stream Window Join: Tracking Moving Objects in Sensor-Network Databases," in *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 2003.

[15] S. Schmidt, M. Fiedler, and W. Lehner, "Source-aware Join Strategies of Sensor Data Streams," in *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 2005.

[16] G. S. Iwerks, H. Samet, and K. P. Smith, "Maintenance of K-NN and Spatial Join Queries on Continuously Moving Points," *ACM Transactions on Database Systems , TODS*, vol. 31, no. 2, pp. 485–536, 2006.

[17] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[18] B. Liu, Y. Zhu, and E. A. Rundensteiner, "Run-time operator state spilling for memory intensive long-running queries," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2006.

[19] 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 *Proceedings of the International Conference on Data Engineering, ICDE*, 2004.

[20] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis, "RPJ: Producing Fast Join Results on Streams through Rate-based Optimization," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2005.

[21] 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.

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

[23] G. Luo, C. Ellmann, P. J. Haas, and J. F. Naughton, "A scalable hash ripple join algorithm," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2002.

[24] W. H. Tok, S. Bressan, , and M.-L. Lee, "RRPJ : Result-Rate based Progressive Relational Join," in *Proceedings of the International Conference on Database Systems for Advanced Applications, DASFAA*, 2007.

[25] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "On producing join results early," in *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, 2003.

[26] J. Kang, J. F. Naughton, and S. Viglas, "Evaluating Window Joins over Unbounded Streams," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2003.

[27] U. Srivastava and J. Widom, "Memory-Limited Execution of Windowed Stream Joins," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2004.

[28] J. Xie, J. Yang, and Y. Chen, "On Joining and Caching Stochastic Streams," in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2005.

[29] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

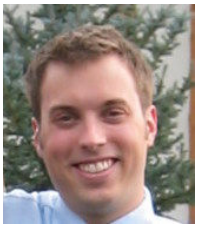
[30] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao, "Load Shedding in Stream Databases: A Control-Based Approach," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2006.

[31] L. Golab and M. T. Özsu, "Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

[32] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, "Scheduling for Shared Window Joins over Data Streams," in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

[33] J. J. Levandoski, M. E. Khalefa, and M. F. Mokbel, "PermJoin: An Efficient ALgorithm for Producing Early Results in Multi-join Query Plans," in *ICDE Poster*, 2008.

[34] M. E. Crovella, M. S. Taqqu, and A. Bestavros., *Heavy-tailed probability distributions in the world wideweb*. Chapman Hall, 1998, ch. A practical guide to heavy tails: statistical techniquesand applications.



Justin J. Levandoski received his Bachelor's degree at Carleton College, MN, USA in 2003, and his Master's degree at the University of Minnesota, MN, USA in 2008. He is pursuing a PhD degree in computer science at the University of Minneosta. His PhD research focuses on embedding preference and contextual-awareness inside the database engine. His general research interests focus on extending database technologies to handle emerging applications.

Mohamed E. Khalefa received the BS and MS degrees in Computer Science and Engineering in 2003 and 2006, respectively, from Alexandria University, Egypt. He is currently pursuing a PhD in database systems at the University of Minnesota. His research interests include non-traditional query processing, preference queries, uncertain data processing, and cloud computing.

Mohamed F. Mokbel (Ph.D., Purdue University, 2005, MS, B.Sc., Alexandria University, 1999, 1996) is an assistant professor in the Department of Computer Science and Engineering, University of Minnesota. His main research interests focus on advancing the state of the art in the design and implementation of database engines to cope with the requirements of emerging applications (e.g., location-based applications and sensor networks). His research work has been recognized by two best paper awards at IEEE MASS 2008 and MDM 2009. Mohamed is a recipient of the NSF CAREER award 2010. Mohamed has actively participated in several program committees for major conferences including ICDE, SIGMOD, VLDB, SSTD, and ACM GIS. He is/was a program co-chair for ACM SIGSPATIAL GIS 2008, 2009, and 2010. Mohamed is an ACM and IEEE member and a founding member of ACM SIGSPATIAL. For more information, please visit http://www.cs.umn.edu/ mokbel.