MJoin: A Metadata-Aware Stream Join Operator

Luping Ding, Elke A. Rundensteiner, George T. Heineman
Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
{lisading, rundenst, heineman}@cs.wpi.edu

ABSTRACT

Join algorithms must be re-designed when processing stream data instead of persistently stored data. Data streams are potentially infinite and the query result is expected to be generated incrementally instead of once only. Data arrival patterns are often unpredictable and the statistics of the data and other relevant metadata often are only known at runtime. In some cases they are supplied interleaved with the actual data in the form of stream markers. Recently, stream join algorithms, like Symmetric Hash Join and XJoin, have been designed to perform in a pipelined fashion to cope with the latent delivery of data. However, none of them to date takes metadata, especially runtime metadata, into consideration. Hence, the join execution logic defined statically before runtime may not be well suited to deal with varying types of dynamic runtime scenarios. Also the potentially unbounded state needs to be maintained by the join operator to guarantee the precision of the result. In this paper, we propose a metadata-aware stream join operator called MJoin which is able to exploit metadata to (1) detect and purge useless materialized data to save computation resources and (2) optimize the execution logic to target different optimization goals. We have implemented the MJoin operator. The experimental results validate our metadata-driven join optimization strategies.

Keywords: Metadata, XML Stream, Join Algorithms, Optimization, Constraint, XQuery Subscription.

1. INTRODUCTION

1.1 New Challenges in Stream Processing

As processing of stream data like online news, sensor data and network monitoring data has attracted increasing attention in recent years [3] [10] [2] [4], join processing, being one of the most expensive query operators, has received renewed interest. As illustrated below, the traditional strategies employed for join processing must be re-examined for this new stream context. Unlike the conventional join being evaluated over persistently stored data, the stream join operator faces the following new challenges:

1. **Potentially unbounded state and initial delay.** Since the input data streams are potentially infinite and arrive on the fly during query execution time, join, being a stateful operator, needs to maintain a potentially unbounded growing state for the already-processed data to evaluate the future-arriving data. In addition, traditional join algorithms using an asymmetric execution model may take an extremely long time to build the "inner" state without being able to emit any initial result.
2. **Dynamic runtime scenarios.** Metadata such as the cardinality of the data stream may not be available to the query optimizer before runtime. Either the query system must dynamically estimate such metadata statistics or in some cases such metadata may be explicitly provided at runtime interleaved with the data, then called punctuations [12]. Since data sources are often distributed over a wide area network, the data arrival rates may greatly fluctuate. Hence a static join algorithm chosen before runtime may not be effective in such dynamic scenarios.
3. **Multiple optimization goals.** The query may run continuously and the result is generated incrementally instead of once only. Also other operators may be running at the same time. Thus for individual operators like join, the minimal time to the completion of the query is no longer the only optimization goal. Other objectives like minimal memory overhead or a stable output rate of dynamic metadata may instead become the important objectives.

1.2 Related Work

Some existing stream-oriented pipelined join operators aim to tackle these problems. [14] presents a symmetric hash join operator (SHJ). SHJ incrementally maintains in-memory state (hash buckets) for both input streams. It handles newly-arriving tuples from one stream by joining them with the current state of the other stream. Therefore, SHJ avoids the infinite long "build" phase of an asymmetric join. It also masks slow data arrival rates by proceeding with available data from either input stream instead of being stalled by a slow-delivered

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

data source. XJoin [13] extends SHJ to also handle any potential memory overflow, lifting the unrealistic assumption for the whole join state to be memory-resident. It provides a three-stage policy to switch the join processing between the memory-resident and disk-resident portion of the state to cope with any slow delivery of stream data. That is, the in-memory join is always given the highest priority. The blocking period during delayed data delivery is exploited to make progress on the join related to secondary-storage-resident tuples. Similar to XJoin, double-pipelined hash join [8] is another symmetric join strategy with overflow resolution.

However, none of the above join strategies take metadata about the stream, neither static nor dynamic, into consideration. Therefore, they will neither recognize nor drop no-longer-needed data from the state. Thus they don't tackle the problem of a potentially infinite state. Their execution logic is not optimized in accordance with the available metadata. Moreover, some of these join strategies only aim for a single optimization goal, namely, throughput, while other optimization goals may be more appropriate for stream data.

In response, [10] proposed the idea of exploiting constraints on data streams to reduce the memory overhead of the query operators by purging useless data from the state. [1] further provides a query execution algorithm that has state purging rules based on static constraints. These two works only consider a subset of static constaints, namely, integrity constraints and clustered or ordered arrival patterns known before runtime. They don't deal with intra-operator scheduling nor target different optimization goals. Tucker et al. [12] describe a framework for stream operators in the presence of puncutations, the runtime metadata. They define state purging and punctuation propagation functions for stream operators. However, no metadata-driven optimized join algorithm has been proposed to date.

Semantic query optimization [5] [9] is the counterpart of metadata-aware query optimization in the traditional query processing context. It uses knowledge of the semantics of the data to transform a query into an equivalent query which can be processed more efficiently. It is based on the fact that all the base data and metadata are available before runtime. It is also typically a plan-level instead of an operator-level optimization.

1.3 Our Approach

In this paper, we present MJoin, a metadata-aware stream natural join operator, which has the following features:

1. Use a multi-subtask execution model, with each subtask being equipped with a family of alternate execution strategies. Driven by metadata, one of the strategies will be chosen based on optimization heuristics.

2. Reduce resource requirements by correctly and in a timely manner detecting and purging useless data from the join state.

3. Be able to target different optimization goals.

4. Handle memory overflow by applying secondary-storage-involved overflow resolution techniques.

To the best of our knowledge, MJoin is the first join strategy that exploits both static and dynamic metadata to optimize the execution logic and also target different optimization goals. We have implemented MJoin in Raindrop [11], an XQuery subscription system developed at WPI. We also report on experimental studies that validate our metadata-driven optimization strategies.

The rest of the paper is organized as follows: Section 2 gives a running example. Section 3 provides background knowledge. Section 4 gives an overview of the MJoin operator. The detailed description of the join algorithm is presented in Section 5. Section 6 shows the experimental results and we conclude our work in Section 7.

2. RUNNING EXAMPLE

As running example throughout this paper, we consider an Online News Management System (Figure 1(a)), which publishes news as well as collects user access data, analyzes both news information and access records to get statistics. The *News Publisher* generates the general description about news and submits them as XML stream (Figure 2). The *Access Monitor* generates an XML stream of user access records of each news item from the time it is posted until it is dropped off the top 10 latest news. After that, a punctuation is inserted into the stream signaling "no more access record for news item with a serial number equal to some value will be generated" (Figure 3). The *Data Analyzer* enables the data analyst to issue XQueries over the XML streams. For example, the XQuery in Figure 4 correlates the user's IP address with the keywords of the news this user has read to establish the interrelationship between user interest and the keywords of the news items. In the corresponding query plan shown in Figure 1(b), a natural join operator joins the stream $NewsGeneral$ (denoted as IS_a) with the stream $AccessRecord$ (denoted as IS_b) on *sno*. This is a one-to-many join on the key of the stream IS_a and the foreign key of the stream IS_b.

The leaf operators of the query plan are special operators called *Convert*, which is similar to X-scan operator in Tukwila [7]. A *Convert* operator takes an XML stream and converts it to a stream of tuples. Those tuples will conform to the fixed schema shown in Figure 1(b). Therefore, the input data of Join operator are two potentially infinite streams of tuples.

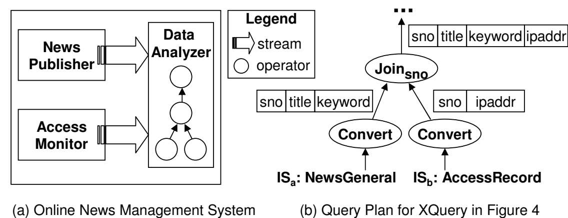

Figure 1: Running Example.

```
<stream:NewsGeneral>
  <info>
    <sno> 79 </sno>
    <title> Cashless society gets mixed reviews </title>
    <keyword> Smart Card </keyword>
  </info>
    ...
</stream:NewsGeneral>
```

Figure 2: General News Information Stream.

```
<stream:AccessRecord>
  <record>
    <sno> 79 </sno>
    <ipaddr> 64.58.76.224 </ipaddr>
  </record>
  <record>
    <sno> 81 </sno>
    <ipaddr>207.68.172.234</ipaddr>
  </record>
  <record>
    <sno> 79 </sno>
    <ipaddr> 202.112.39.8 </ipaddr>
  </record>
  <punct:record>
    <sno> 79 </sno>
    <ipaddr> * </ipaddr>
  </punct:record>
  ...
</stream:AccessRecord>
```

Figure 3: User Access Record Stream.

```
FOR $ar in stream("AccessRecord")/record
RETURN {
  FOR $ng in stream("NewsGeneral")/info[sno=$ar/no]
    RETURN
      <data> $ng/keyword, $ar/ipaddr </data>
}
```

Figure 4: Example XQuery.

3. BACKGROUND

3.1 Join Semantics and State

We use the **semantics** of continuous queries over multiple data streams defined in [10]. The *active set* of a stream natural join J at time T_t consists of all tuples that have been processed before time T_t. Then the answer of J at any time T_t, A(J, T_t), is defined as the answer resulting from evaluating J using standard natural join operator semantics over the active set of J at time T_t.

A *stateful* operator, like join, needs to materialize the whole or at least a partial set of already-processed data to be able to process the future-arriving data. Such materialized dataset is called *state*. In MJoin algorithm, each data item in the state is called *state tuple*, which corresponds to a whole input tuple. We denote the state corresponding to the input streams IS_a and IS_b by S_a and S_b, respectively.

3.2 Metadata

Metadata for a database defines how the data is stored, including its schema definition, integrity constraints and index information. In the stream context, schema information and integrity constraints can be provided at stream registration phase. Then they are available at the static query optimization phase. Some information may not be known before runtime, possibly only being delivered in the form of runtime metadata, including (1) the number of tuples of each stream seen thus far; and (2) the domain of future data.

Our goal is to design a join operator that is able to make use of both static and on-the-fly metadata. We now classify metadata by summarizing important dimensions of metadata introduced in the literature. As Figure 5 shows, *granularity* specifies the level at which the metadata would be applied, including query (like window constraints), schema (like integrity constraints), and instance level (like punctuations [12] interleaved inside the stream). *Time* characterizes when the metadata is *available* to our system, either statically at query optimization time or dynamically at runtime. The *scope* dimension refers to the time period during which the metadata will be valid. Global metadata would be valid for the whole data stream, while local metadata is only valid for a specified substream. In [10], they distinguish between static and dynamic constraints. By the classification of Figure 5, all global constraints obtained before runtime are static constraints, and all other constraints are dynamic.

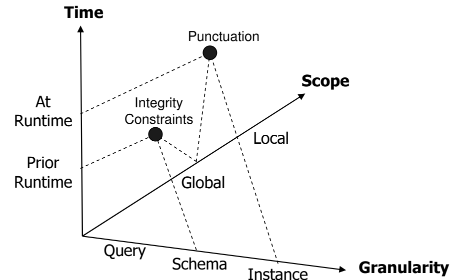

Figure 5: Dimensions of Metadata.

Below we will introduce two types of metadata exploited by MJoin.

Integrity constraints [6] are schema level metadata. We assume they are available before runtime and are global. We consider a join between a key and a foreign key, which implies a one-to-many join. Since the data arrival rates are unpredictable, we don't have any restrictions on the relative arrivals of the key and its matching foreign keys.

Punctuations [12] are predicates on stream elements which specify what items will not be seen past the punctuation. Punctuations are interleaved inside the actual data stream to mark the end of substreams. Hence, they serve as dynamic domain information for future data. By logically breaking an infinite stream into a combination of finite streams, the cardinality of substreams becomes known. We classify a punctuation as runtime instance-level metadata, which is globally valid from declaration onwards until the end of the stream.

In MJoin, we continue to use the punctuation model and semantics defined in [12]. A concrete example of the punctuation has been presented in the running example (inside the stream in Figure 3). We also assume all

metadata is provided by the stream generator, either when the stream is registered or at runtime. The MJoin operator will make use of them, whenever available, but it has no control over their content nor their arrival patterns.

4. MJOIN OPERATOR OVERVIEW

Internal to the join operator design, a set of subtasks are performed within an infinite loop. These subtasks include:

1. Retrieve a tuple *tup* from some input stream (e.g., IS_a);
2. Search the state S_b using *tup* for matching tuples, join them with *tup*, and output the result;
3. Insert *tup* into the state S_a;
4. Retrieve a punctuation *punct* from some input stream (e.g., IS_b);
5. Purge useless state tuples from the state S_a using *punct*.

We have various strategies designed for performing each subtask, with each strategy being optimized for certain metadata and specific optimization goal.

4.1 Stream Scheduling

The traditional hash join algorithm [6] proceeds in a "sequential-build-and-probe" mode, which relies on the fact that the entire dataset is finite and available before query execution. However, when dealing with potentially infinite stream data which gradually becomes available at runtime, such strategy does not apply because of the huge initial delay for building the inner dataset. Instead, the join algorithms which proceed in "interleaved-build-and-probe" mode [14] [13] [8] are proposed for the stream context. At each time, a tuple from either input stream is retrieved to probe the state of the other stream and then is inserted into the corresponding state. *Stream scheduling* determines from which input stream the next tuple should be retrieved from. We consider two stream-scheduling strategies.

Random scheduling randomly retrieves a tuple from one input stream to process, hence treating both inputs symmetrically. This is easy to control. However, in the case that the two inputs are not symmetric, like a one-to-many join, asymmetric stream scheduling may perform better than symmetric ones.

We design the *priority-driven scheduling*, an asymmetric strategy. It breaks both input streams into a sequence of substreams. Each time a substream from each stream is picked to form a substream pair. Within such substream pair, the traditional hash join algorithm is reused, that is, the smaller dataset is chosen as "inner" one to build the state, and tuples from the other dataset are used to probe the state. Therefore, in the one-to-many join from IS_a to IS_b, IS_a is assigned a higher priority to be consumed next. This costs less overhead in building and probing than the opposite choice or random scheduling for the following reasons: (1) Every tuple from IS_b can be dropped after it has been joined once. Ideally, if tuples from both streams arrive in the same order with respect to the join attribute, then each time a tuple from IS_b is used to probe S_a, the matching tuple is likely to be already there. Hence the materialization cost of tuples from IS_b is avoided; (2) By materializing an equal number of tuples from both inputs, S_a would cover a wider data range. This may improve the chance of newly-arriving tuples from IS_b to get joined immediately.

4.2 Tuple Scheduling

Unlike stream scheduling which concerns itself with making a decision about which stream to read data from, *tuple scheduling* affects the order in which tuples from the same input stream are processed.

Sequential scheduling is the default tuple scheduling strategy. Here the order of processing tuples from the same stream is also the order of retrieving them. The advantage of sequential scheduling is simplicity. However, in some cases we want to break the coupling of retrieving and processing tuples to achieve a better output rate of both data and punctuations.

We design *value-oriented scheduling* which schedules the processing of tuples based on the relationship between their values and the available punctuations. For example, in a many-to-many join with punctuations over both input streams, once a punctuation arrives from IS_a, the corresponding punctuation would be expected from IS_b so that we can propagate such punctuation (although it may never come). The prerequisite to propagating a punctuation is that all possible output tuples matching the punctuation have been generated. Therefore, tuples from IS_b which match the to-be-propagated punctuation will be assigned higher priority to be processed, while the processing of tuples which likely won't contribute to the punctuation propagation is delayed.

We design the rule for switching between these two tuple scheduling strategies. By default, sequential scheduling is used. When some punctuation arrives, value-oriented scheduling is activated and works until: (1) the punctuation expires (all the tuples punctuated by this punctuation have finished the join) and no valid punctuation exists or (2) stream scheduling switches to process another stream since stream scheduling overrides tuple scheduling.

4.3 On-the-fly Dropping and State Purging

By checking metadata, MJoin can achieve multi-fold gains in reducing memory overhead:

1. A tuple may never need to be inserted into the state if it won't have any chance to join with any future tuple, which we call *on-the-fly dropping*.
2. Once a state tuple is known to no longer be able to join with any future tuple, it is removed from the state, which is called *state purging*.

Either on-the-fly dropping or state purging can only happen when it is known from the metadata that no existing matching tuple from the opposite stream is left un-joined nor possibly still coming in the future. During

the evaluation of the join, if we can get such *purging indication* from metadata frequently, either newly-arriving tuples don't even need to be inserted into the state or useless state tuples can be purged in a timely manner. Hence the memory overhead can be reduced without compromising the precision of the result. Depending on the metadata, the purging indication can be explicit or implicit. Static constraints would specify implicit purging indications as follows:

1. A one-to-many join from IS_a to IS_b implies that whenever a value occurs at IS_a once, it won't occur any more. Hence there is no need to maintain the matching state tuples from IS_b.

2. A clustered arrival implies that whenever the cluster is over, no data with the clustered join value will come. Hence the matching state tuples from the other input can be purged.

3. An ordered arrival is a special case of clustered arrival. It not only specifies that no data with such join value will come later, but also implies that no data with a join value less than (increasing order) or greater than (decreasing value) such value will come any more past this cluster.

In addition, each punctuation explicitly specifies a purging indication. In our algorithm, on-the-fly dropping based on a purging indication is always done immediately. This avoids the overhead of inserting tuples into the state. In addition, two state-purging strategies are proposed to be tuned for different optimization goals. State purging can be done whenever the purging indication is obtained. A performance gain will likely be achieved by shrinking the state. However, the purging itself carries some cost, that is, the state needs to be scanned to find the purgable tuples. Hence this represents a delicate tradeoff between time versus space. Accordingly, we design two purging strategies: *Immediate purging* will be performed when a purging indication is obtained; while *batch purging* (also called *lazy purging*) may purge the state at a better time (when the number of existing punctuations reaches some threshold or the input stream is temporarily blocked so that no new input data is available for a while) or until necessary (when the size of the state reaches some size limitation).

Immediate purging can guarantee the minimal memory overhead at any time as well as the earliest possible punctuation propagation, while batch purging can potentially gain a better data output rate by reducing the number of scanning operations over the state and potentially batching the purge. Depending on the optimization goal, a different purging strategy may be chosen.

.

5. JOIN ALGORITHM

MJoin extends XJoin [13] by introducing metadata-aware optimization. We assume that all metadata is indeed accurate. We only deal with punctuations in the form of "from now on, no more tuple with a join value satisfying this *predicate* will arrive in the future". In this section, we will describe the MJoin algorithm in dealing with a one-to-many join from IS_a to IS_b with punctuations on the join attribute of tuples from IS_b (1-NP). Based on this, we briefly describe the modification of the 1-NP algorithm when dealing with a one-to-many join from IS_a to IS_b with a clustered arrival of tuples from IS_b (1-NC), which is a special case of 1-NP.

5.1 Framework

Figure 6 shows the framework of the MJoin operator.

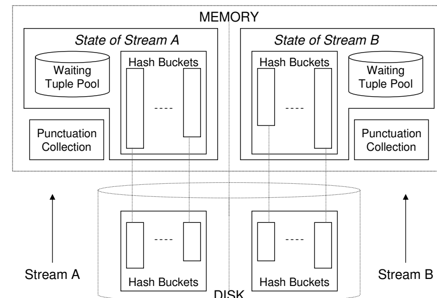

Figure 6: Framework of MJoin

Input/output queues. Each input/output stream of the join operator corresponds to a FIFO queue (denoted as Q_a, Q_b and Q_o, with Q_a and Q_b corresponding to IS_a and IS_b respectively), which can hold both data and punctuations.

Punctuation collection. The operator maintains a punctuation collection for each input stream. The collection contains all punctuations that have already arrived but have not yet been propagated. Each punctuation item specifies a predicate about the data.

State structure. The state contains hash buckets and a waiting tuple pool. *Hash buckets* hold all materialized data from the corresponding input stream. Similar to XJoin, each hash bucket has a memory-resident portion and a disk-resident portion. The *waiting tuple pool* contains *waiting tuples*, which have been retrieved from the queue, but haven't yet been processed due to value-oriented tuple scheduling. Waiting tuples are treated the same as those tuples which still sit in the queue, hence called waiting tuples.

Punctuated range. In order to purge useless data, we need to determine the data range specified by all the punctuations received so far from each input stream. We call it the *punctuated range* (denoted as PR_a and PR_b respectively). No new tuple with the join value falling into the punctuated range will come in the future. Then assume for example, a tuple t_a is retrieved from IS_a and finishes the join with all matching tuples received thus far from IS_b. If the join value of t_a falls into PR_b so that it won't be able to join with any future tuple from IS_b, then t_a can be dropped on the fly.

5.2 Overall Execution Logic

Join Stages. Similar to XJoin, MJoin proceeds in three stages. The *in-memory* stage joins matching tuples in the memory-resident state of both input streams.

The *memory-disk* stage joins the memory-resident state of one input with the disk-resident state of the other input. The *cleanup stage* joins any pairs of matching tuples which haven't gotten joined during the first two stages yet due to a state flush.

The in-memory stage always has the highest priority to run. It includes two different scenarios: (1) the newly-retrieved tuple will join with all matching tuples in the memory-resident state of the other input stream; (2) when the tuple scheduling switches from value-oriented to the sequential strategy, the waiting tuple pool will be cleaned out by processing all waiting tuples in the same way as (1). The memory-disk stage is activated whenever the in-memory stage blocks. And the cleanup stage starts when neither in-memory stage nor memory-disk stage can proceed. At each stage on-the-fly dropping and state purging may be applied.

State purging rules. In MJoin, state purging is invoked by one of the following events: (1) a punctuation is retrieved (using immediate purging); or (2) the maximal number of unpurged punctuations is reached (using batch purging); or (3) the memory is full and unpurged punctuations exist (batch purging); or (4) both input queues are empty (batch purging).

5.3 1-NP Join Algorithm

In the 1-NP join, we assume the stream generator will insert punctuations into the stream to signal the end of substreams, as shown in Figure 3. Such assumption is made only for simplifying the description of our algorithm, but it is not required. If no punctuation arrives, the execution logic described in Section 5.2 will be performed.

Stream/tuple scheduling. Since this is a one-to-many join, priority-driven stream scheduling is applied. In addition, triggered by the available punctuation, value-oriented tuple scheduling may be applied accordingly. Once started, MJoin operator will alternately retrieve substreams from Q_a and Q_b. Each queue is assigned a fixed retrieval quota, denoting the maximal number of tuples to be retrieved consecutively from a queue. At the beginning, the retrieval iterator of Q_a is 0. MJoin keeps retrieving tuples from Q_a until either Q_a becomes empty or the iterator of Q_a reaches the quota. Then Q_b begins to be consumed until an event happens similarly as above. Once the retrieval quota of one queue is reached, the quota of the other queue will be reset to 0. If the iterators of both queues are within the range [0, retrieval quota) and neither queue is empty, the head tuple of Q_a will be retrieved because Q_a is assigned higher scheduling priority.

Execution logic. The execution logic is determined based on two fundamental rules given below.

Rule 1: If MJoin is optimized to maximize the tuple output rate, batch purging will be applied. The state purging rule is stated in Section 5.2. The execution logic is defined as follows:

1. The MJoin algorithm is running in an infinite loop. Each time the stream and tuple scheduling strategies will decide how to retrieve tuples from queues and in which order to process them. After a tuple or a punctuation is retrieved and chosen to be processed, either step 2, 3 or 4 will be performed.

2. A tuple t_a retrieved from Q_a is used to probe S_b. If no matching tuple is found, t_a is inserted into S_a. MJoin continues to retrieve the next tuple. If matching tuples are found, t_a joins with all of them and then purges them immediately. If the join value of t_a falls into PR_b and the matching bucket in S_b doesn't have any disk-resident portion, then t_a is dropped as well. Otherwise it is inserted into S_a. The corresponding pseudocode for this step is shown in Figure 7.

3. A tuple t_b retrieved from Q_b is used to probe S_a. If a matching tuple t_a is found, t_b is joined with t_a and then dropped. Otherwise t_b is inserted into S_b.

4. A punctuation p retrieved from Q_b is inserted into the punctuation collection.

5. Whenever a tuple is inserted into the state, the status bit for purging will be checked to see if state purging is needed.

```
Procedure stage-mem-a (Tuple t_a) {
    /* Compute the hash value. */
    Object key = hash(t_a.sno);
    /* Get corresponding hash bucket. */
    Bucket bucket = S_b.getBucket(key);
    if (bucket ≠ ∅)
        foreach t_b in bucket
            /* Join t_a with matching tuples from S_b. */
            if (t_a.sno == t_b.sno) {
                Qo.enqueue(join(t_a, t_b));
                /* Purge matching tuple from S_b. */
                S_b.purge(t_b); }
    /* Judge whether to drop t_a on the fly. */
    if (t_a.sno ∈ PR_b) /* PR_b is punctuated range of S_b. */
        if (bucket.hasDiskPart())
            S_a.add(t_a);
    else
        S_a.add(t_a); }
```

Figure 7: Join Algorithm in Pseudocode.

Rule 2: If MJoin is optimized to minimize the memory overhead, immediate purging will be applied to purge the state whenever a punctuation is retrieved from Q_b. The rest of the execution logic is the same as above.

Duplicate issue. Although the multi-stage join algorithm is applied, for a one-to-many or one-to-one join, MJoin doesn't have the potential problem of producing spurious duplicate tuples, which is unlike XJoin. Since any tuple from IS_b will be dropped immediately after it is joined with the matching tuple from IS_a, it has no chance to contribute to more than one output tuple. Hence the execution logic is simpler than XJoin.

5.4 Further Optimization of 1-NC Join

1-NC (one-to-many-clustered) join is a special case of 1-NP join, which has stricter constraints on the stream IS_b. Hence it provides opportunity for further optimization. For example, since tuples with the same join value

arrive consecutively, the stream scheduling strategy can be changed accordingly to speed the output rate. Whenever a tuple from IS_b gets joined, MJoin will continue to retrieve the subsequential tuples from IS_b until a different value occurs to signal the end of the cluster. In doing so, more cost of probing and inserting tuple into the state has been saved.

6. EXPERIMENTAL VALIDATION

We now provide experimental results to explore the effectiveness of our metadata-based optimization.

6.1 Experimental Setup

We ran the experiments on an Intel(R) Celeron(TM) 733 MHz processor, with 512 MBytes real memory, running Windows2000 and Java 1.4.1 01. To have repeatable experiments, we create *NewsGeneral* and *Access−Record* streams beforehand and feed them to the MJoin operator at runtime. A punctuation is inserted into the *AccessRecord* stream whenever a news item is rolled out of the top 10 latest news and the access monitor no longer generates user access records for it. Within each punctuated substream, at most 10 distinct *sno* values exist representing the serial numbers of the news being monitored. Each distinct *sno* value may correspond to multiple access records, which arrive in a random order. A substream of *AccessRecord* (only *sno* value) is shown below. P(*value*) represents the punctuation. For example, P(3) means no more *sno* with value 3 will come afterwards.

← 9,12,5,11,7,4,12,6,4,7,3,12,P(3),5,11,10,7,4,7,4,9,12,P(4),8,...←

We test two arrival patterns of the input streams. In the *ordered* pattern, both input streams arrive in the increasing order with respect to *sno* of the *NewsGeneral* stream and the punctuated *sno* of the *AccessRecord* stream. In the *random* pattern, *NewsGeneral* stream arrives in some random order with respect to the *sno* value.

Due to space limitations, here we only show the experiments which test the in-memory behavior of the 1-NP join running in the "CPU-limited mode", that is, the input queues never become empty. All experiments are run at least 10 times and the averages over all runs are plotted.

6.2 Results

Experiment 1: Stream and tuple scheduling. We first compare the performance of two combinations of stream and tuple scheduling, that is, priority-driven stream scheduling with value-oriented tuple scheduling vs. random stream scheduling with sequential tuple scheduling. The latter is the default strategy used by most existing stream join algorithms [13] [14]. We evaluate MJoin using these two scheduling combinations over the two stream arrival patterns stated above. The time used to output a certain number of tuples is recorded. This time also covers the time for immediate purging and punctuation propagation. Figure 8 shows the result in the case of *ordered* and *random* arrival patterns respectively. We can see that in both cases, priority-driven stream scheduling with value-oriented tuple scheduling always outperforms the other scheduling combination.

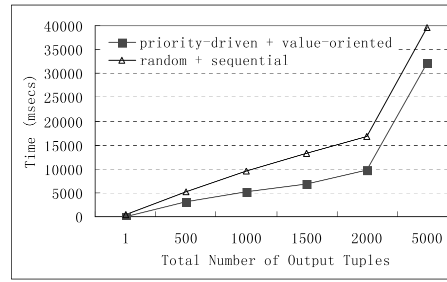

Figure 8: Stream/Tuple Scheduling Strategies.

Experiment 2: MJoin vs. XJoin. Second, we compare the result generating time and memory overhead of MJoin with XJoin. In this experiment, we eliminate the punctuation propagation task from MJoin because it is not supported by XJoin. Batch purging is applied here instead of immediate purging because the granularity of punctuations is rather small. We now purge conservatively after every 100 punctuations are received.

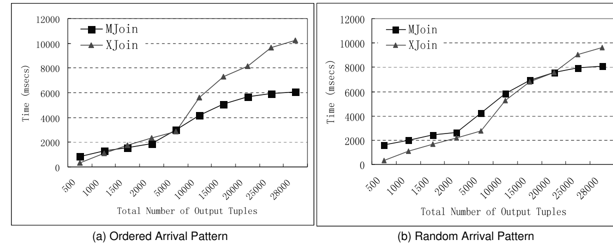

Figure 9: Execution Time, MJoin vs. XJoin.

Figure 9 shows that until some threshold (the total number of output tuples reaches 7500) is reached, MJoin costs somewhat more time than XJoin because the time used by MJoin to purge state dominates over the extra probing time taken by XJoin. As more data streams in, without purging, the state maintained by XJoin increases significantly. The probing time keeps increasing as well. MJoin, however, purges useless data in a timely manner so that it costs less time than XJoin. Especially, when memory overflow occurs in XJoin, MJoin will achieve more time savings because disk I/Os are significantly more expensive than only in-memory operations. Also from Figure 10, we can see that for the *random* arrival pattern, by getting rid of no-longer-needed data in a timely manner, the memory overhead can be significantly reduced as more data streams in. The result over *ordered* arrival pattern is similar to the *random* one.

Experiment 3: Immediate vs. batch purging. From Figures 9 and 10, we see that by applying batch purging, the memory overhead can be effectively reduced and the execution time won't deteriorate severely and can even be better as more data streams in. Now

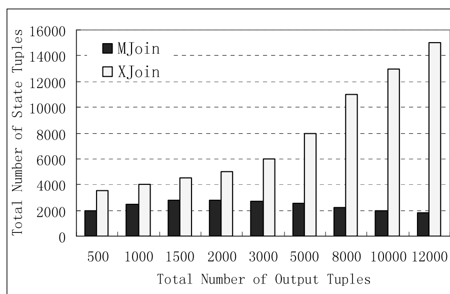

Figure 10: Memory Overhead, MJoin vs. XJoin.

we compare the performance of immediate purging with batch purging. Same as above, we evaluate MJoin by applying state purging after every 1 or 100 punctuations respectively over the data streams which arrive in *random* pattern. The execution time to get a certain number of output tuples is recorded.

Figure 11 shows that for such a frequently arriving and small-granularity punctuation case, immediate purging will cause more time than batch purging because of the cost for scanning the state. Hence we conclude that although state purging can help to reduce memory overhead and further reduce the state probing cost, since it also costs extra effort to scan the whole state, we need to find a good time to do it.

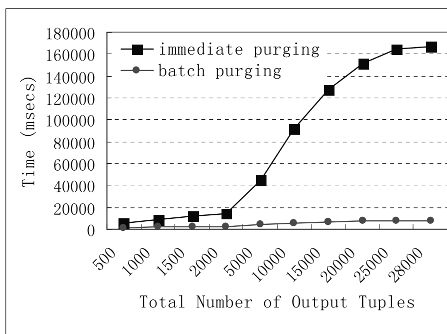

Figure 11: Immediate vs. Batch Purging.

7. CONCLUSION

In this paper, we have presented MJoin, a metadata-aware stream join operator, which extends XJoin by introducing optimization of the execution logic according to both static and dynamic metadata. It is also able to target different optimization goals. It purges useless data from the state to reduce the requirement of computing resources. Accordingly, different scheduling and purging strategies are being proposed. We have implemented MJoin in the Raindrop system. We also show the experimental results of heuristic-based intra-operator optimization.

In the future, we plan to work on a cost-based stream join operator optimization strategy by exploiting metadata instead of the heuristics-based approach presented here.

8. REFERENCES

[1] S. Babu and J. Widom. Exploiting k-Constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford University, Nov 2002.

[2] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[3] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214, 2002.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, May 2002.

[5] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. In *VLDB*, pages 687–698, 1999.

[6] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems, The Complete Book*. Prentice Hall, 2002.

[7] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Technical Report CSE000502, University of Washington.

[8] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD*, pages 299–310, 1999.

[9] J. J. King. Quist: A system for semantic query optimization in relational databases. In *VLDB*, pages 510–517. IEEE Computer Society, 1981.

[10] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, 2003.

[11] H. Su, B.Pielech, L. Ding, J. Jian, Y. Zhu, and E. A. Rundensteiner. Raindrop: A uniform query paradigm for processing xqueries on XML streams. Submitted for publication, 2003.

[12] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Punctuating continuous data streams. www.cse.ogi.edu/dot/niagara/pstream/punctuating.pdf, 2002.

[13] T. Urhan and M. Franklin. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[14] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.