Teddies: Trained Eddies for Reactive Stream Processing

Kajal Claypool[1,⋆] and Mark Claypool[2]

[1] MIT Lincoln Laboratories
claypool@ll.mit.edu
[2] Worcester Polytechnic Institute, Worcester, MA
claypool@cs.wpi.edu

Abstract. In this paper, we present an adaptive stream query processor, Teddies, that combines the key advantages of the Eddies system with the scalability of the more traditional dataflow model. In particular, we introduce the notion of adaptive packetization of tuples to overcome the large memory requirements of the Eddies system. The Teddies optimizer groups tuples with the same history into data packets which are then scheduled on a per packet basis through the query tree. Corresponding to the introduction of this second dimension – the packet granularity – we propose an adaptive scheduler that can react to not only the varying statistics of the input streams and the selectivity of the operators, but also to the fluctuations in the internal packet sizes. The scheduler degrades to the Eddies scheduler in the worst case scenario. We present experimental results that compare both the reaction time as well as the scalability of the Teddies system with the Eddies and the data flow systems, and classify the conditions under which the Teddies' simple packet optimizer strategy outperforms the per-tuple Eddies optimizer strategy.

Keywords: Stream Database Systems, Adaptive Query Processing, Adaptive Scheduler.

1 Introduction

The proliferation of the Internet, the Web, and sensor networks have fueled the development of applications that treat data as a continuous stream rather than as a fixed set. Telephone call records, stock and sports tickers, streaming data from medical instruments, and data feeds from sensors are examples of streaming data. As opposed to the traditional database view where data is fixed (passive) and the queries considered to be active, in these new applications data is considered to be the active component, and the queries are long-standing or continuous. In response, a number of systems have been developed [BW01,MWA+03,BBD+02,CCC+02,CDTW00] to address this paradigm shift from traditional database systems to now meet the needs of query processing over streaming data.

⋆ This work was done while the author was at University of Massachusetts, Lowell.

J.R. Haritsa, R. Kotagiri, and V. Pudi (Eds.): DASFAA 2008, LNCS 4947, pp. 220–234, 2008.

Stream query optimizers represent a new class of optimization problems. The most egregious problem faced by stream query optimizers is the need for query plan adaptation (in some cases continuous adaptation) in reaction to the turbulence exhibited by most data streams. The term turbulence here refers to the fluctuations in fundamental data statistics such as the selectivity, as well as to the variability in the network, where bandwidth, quality-of-service, and latency can modulate widely with time. The turbulence in data streams thus makes a static optimizer, which chooses a query plan once-and-for-all at query set-up time, inadequate. For example, in traditional systems the optimizer makes decisions on scheduling and resource allocation based on the fundamental statistics of the data such as its selectivity. However, in the dynamic case statistics that lead to optimal query plan selection at one point in time may no longer be prudent at a later time. A stream query optimizer must, thus, be able to adapt to changing conditions to deal with this dynamicity. To address this problem, two broad strategies for a dynamic stream query optimizer have been proposed: the *data flow* [CCC+02,MWA+03] and the *Eddies* [AH00,MSHR02] models.

A data flow optimizer typically selects a query plan topology based on past (or expected) stream statistics, and arriving data is processed *en masse* using this query plan topology. While the data flow optimizer is adaptive in theory, in practice it is often difficult to change the overall topology of the query plan without introducing significant delays into the system. The novel Eddies architecture [AH00,MSHR02,Des04,DH04] was proposed to introduce a higher degree of adaptability into the query optimizer. The Eddies model functions at the granularity of individual tuples, electing to find the most efficient evaluation path on a per tuple basis with respect to the conditions that prevail at the time. Thus, unlike the data flow and the traditional models, the Eddies model need not compute apriori the optimal query plan topology for the entire data set, rather it can calculate the optimal path based on current tuple statistics as it processes each tuple. The fast reaction time, the adaptability offered by the Eddies model, comes at a price. As its scheduler operates on a per tuple basis, the Eddies processor cannot take advantage of bulk processing of similar data. Moreover, the metadata required for each tuple to keep track of its progress through the logical query plan makes the Eddies system prohibitively expensive in terms of the memory requirements, thereby limiting its scalability.

In this paper, we propose Teddies, **T**rained **eddies**, – a hybrid system that incorporates the adaptability of the Eddies system with the scalability of the more traditional data flow model. The fundamental building block of the Teddies system is the *adaptive packetization* of tuples to create a tight inner processing loop that gains efficiency by bulk-processing a "train" of similar tuples simultaneously. Corresponding to this new dimension – the packet granularity – we introduce an *adaptive scheduler* that reacts to not only the time varying statistics of the input streams and the selectivity of the operators, but also to the fluctuations in the internal packet sizes. The scheduler, thus, schedules only those packets that are sufficiently filled as determined by a threshold value for each packet type. Additionally, the threshold value for each packet type is adjusted dynamically,

growing if the system is creating many tuples of a particular type, and shrinking otherwise. This ensures that there is no starvation – packets containing tuples of all types are eventually scheduled.

The Teddies system uses packets in a manner similar to data flow systems, while at the same time applying an adaptive scheduling mechanism that does not tie down the overall processing to a single query tree topology. The cost of the Teddies system is the introduction of latency into the system since some tuples may have to remain in their respective queues for longer periods of time waiting for tuples with similar metadata to fill the queue to the threshold. We ran a series of experiments to measure the overhead of our adaptive packetization algorithm, as well as a series of experiments to isolate the performance benefits of packetization with respect to the varying cost of a single routing decision. The results show the promise of the Teddies approach over both the data flow and Eddies approaches.

The rest of the paper is organized as follows: Section 2 presents an overview of the data flow and the Eddies system to set the context for the rest of the paper; Section 3 gives an overview of the Teddies system and details its main components. Section 4 outlines our experimental methodology and presents our preliminary experimental results; Section 5 briefly describes the related work; and Section 6 summarizes our conclusions and provides possible future work.

2 Background: Data Flow and Eddies Architecture

2.1 Data Flow Query Processor

In the *data flow* framework, stream operators have dedicated buffers at their inputs, and the query plan chosen by the optimizer is instantiated by connecting the output of some operators to the input of another operator. Figure 1 gives a pictorial depiction of a multi-way join query topology in the data flow model. The operators are scheduled to consume tuples in their input queues in such a way so as to optimize the throughput of tuples in the system. Various strategies and algorithms are used to schedule the operators. However, the topology of the query plan is difficult to change once it has been set up, as the amount of computational state that resides in the operator input queues at any point of time cannot easily be disentangled. Hence, a change to a potentially more advantageous query topology, involving reconnection of some of the operator queues, cannot be done except at special points of time. One possible strategy to address this is to: (1) block all upstream tuples from entering that portion of the query tree that is to be modified; (2) drain all the remaining tuples in the intermediate queues; (3) reconnect the queues in a new plan; and then (4) unblock the upstream tuples to resume execution. If there is a large amount of intermediate state, this blocking strategy is both costly and inefficient. The latency in the reaction time of the data flow architecture is the main motivation for the introduction of the Eddies framework. The Eddies model explicitly keeps track of the intermediate state of tuples in the input queues to enable query

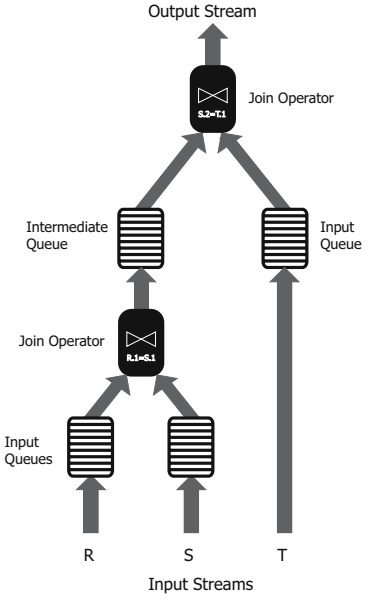

Fig. 1. The Dataflow Query Processing Architecture

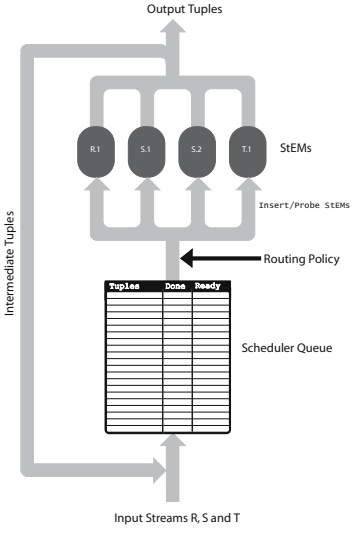

Fig. 2. The Eddies System Architecture

topology rearrangement on the fly without blocking any processing and hence introducing any latency in the system.

2.2 The Eddies Query Processor

The Eddies stream query processor is a highly adaptive architecture for computing stream queries. In Eddies, tuples – input tuples or intermediate tuples – are routed individually through the query plan. Each tuple has associated with it a set of metadata that tracks the progress of the tuple through the query plan. This metadata is unique to each individual query and its structure is determined when the query is first introduced into the system. A query plan consists of the operators that compose the query, such as versions of SELECT, PROJECT, and JOIN operators.

The Eddies system relaxes the requirement to choose a particular query topology by decomposing the join operators, in this case, into sub-operators – called SteMs – and then routing the tuples through the SteMs [Ram01,MSHR02]. There is one SteM associated with each stream attribute that participates in the join. For example, consider the join $R \bowtie_{R.1=S.1} S \bowtie_{S.2=T.1} T$ shown in Figure 2. Here the join attributes $R.1$, $S.1$, $S.2$, and $T.1$ each have an associated SteM. The SteM keeps track of the window of attribute values that have arrived on the stream and allow subsequent tuples from the other streams to probe these stored attribute values to search for a match. The SteM may be implemented, for example, as a hash table on the attribute values to allow for efficient probing. In the case

of the join operator, the order in which tuples probe the SteMs is irrelevant as long as each tuple passes through each SteM exactly once. This holds from the associativity and the commutativity property of the join operator. Intermediate tuples that are produced as output from a SteM – representing a partial join between the input tuple and any of the tuples stored in the SteM that match the input tuple's join attribute – do not need to revisit the SteMs already visited by their constituent tuples. In this way the intermediate tuples, as they pass through all of the SteM operators, build up the final output tuples one SteM at a time. Once a tuple has visited all of the SteMs, it is output from the system as a tuple in the full join.

The Eddies system, thus, routes tuples through the SteM operators, and maintains one large queue of intermediate tuples. Each intermediate tuple has associated metadata (see `Scheduler Queue` in Figure 2) that keeps track of which SteMs the tuple has already visited and which SteMs the tuple needs to visit in the future. When a tuple bubbles to the top of the queue it is eligible for routing. The scheduler examines the metadata and then schedules one of the SteM operators that the tuple needs to visit. Because this process occurs for each intermediate tuple in the system, and indeed represents the main inner loop of the processing of the system, the decision to which SteM to send a tuple must be made efficiently.

The Eddies system uses an *adaptive scheduler*, one version of which keeps track of the number of tuples produced minus the number of tuples consumed for each query operator. In this version of the Eddies adaptive scheduler, a lottery based scheme is used whereby operators gain lottery tickets by consuming input and lose lottery tickets by producing output. For each tuple to be scheduled, the scheduler holds a lottery, with the winning operator selected to process the tuple [MSHR02,DH04]. By scheduling the operator that produces the fewest while consuming the most, the system can react to changing conditions by making a local decision for each tuple. Since each tuple need not follow the same path through the SteM operators, it is possible for the system to find the most efficient path for a given tuple with respect to the prevailing conditions.

3 Teddies Adaptive Query Processor

The Eddies system, while highly reactive, is limited in its scalability by the fine granularity of its routing decisions - that is by its tuple routing. To overcome the drawbacks of the Eddies system, we propose the Teddies system.

Teddies is an adaptive query processing system that leverages the fine-grained adaptivity of pure Eddies with the efficiencies gained from the bulk processing of query operators. A fundamental concept of the Teddies system is a *packet* wherein tuples with the same history (metadata) are grouped together. The Teddies scheduler thus works at the granularity of packets, making routing decisions per packet as opposed to the per tuple routing decisions made by the Eddies system. In a heavily loaded system, this packetization of tuples enables the scheduler to make a reduced number (by a factor of the packet size) of

routing decisions per tuple, achieving a corresponding increase in the system efficiency. The potential gains of this amortization is harnessed by the scheduler to enhance the packet routing policies that are employed. In addition, when a particular operator is scheduled by the packet router, the run-time cost of the operator per-tuple is amortized across the number of tuples in a packet. However, the trade-off of grouping tuples into packets is that of a reduced adaptivity, as potentially a finer-grained scheduling policy may respond with minimal delay to the changing characteristics of the input data, and of increased latency for some tuples, as some tuples may now have to wait longer in queues before being scheduled.

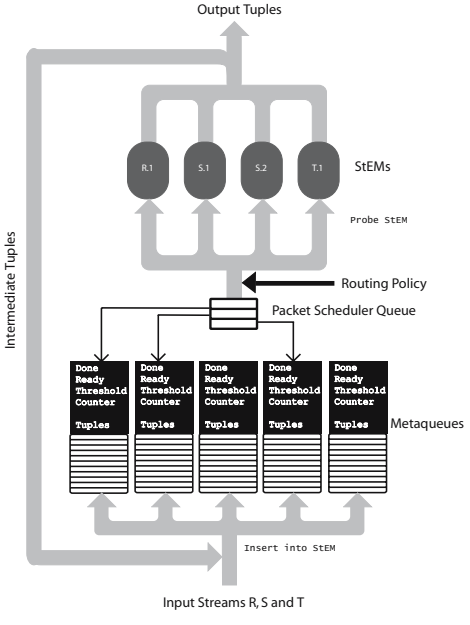

Fig. 3. Teddies System Architecture Illustrating a Three-Way Join Between Three Streams R, S, and T - $R \bowtie_{R.1=S.1} S \bowtie_{S.2=T.1} T$

The overall Teddies design consists of two aspects: the set of data structures used to schedule packets and the algorithm that implements the adaptive packetization scheme.

3.1 The Metaqueue and Scheduler Queue Data Structures

Figure 3 gives an overview of the flow of tuples through the Teddies system for an equi-join $R \bowtie_{R.1=S.1} S \bowtie_{S.2=T.1} T$ defined on three input relations.

The Teddies scheduler design separates the single Eddies scheduler queue into multiple data structures: the *scheduler* queue and the set of *metaqueues*. The

scheduler queue maintains references to those metaqueues which are currently eligible for scheduling. The metaqueues, which are the fundamental units that are scheduled, group together tuples with the same metadata history. Thus, in the Teddies system the metaqueues are the specific mechanism used to packetize the data. Figure 4 shows an example metaqueue, where each metaqueue is a set of tuples along with some associated metadata. The associated metadata consists of the READY and DONE bitmaps, which have the same semantics as in the Eddies system. However, in Teddies this metadata is now shared by *all the tuples* in the metaqueue. Additional metadata is used to keep track of the state of the metaqueue, including (1) *occupancy* – which counts the total number of tuples currently contained in the metaqueue; (2) *threshold* – which represents the number of tuples that need to be in the metaqueue before it can be scheduled; and (3) *counter* – which is a value maintained by the scheduler to prevent starvation. (The precise policy used by the scheduler is described in Section 3.2.).

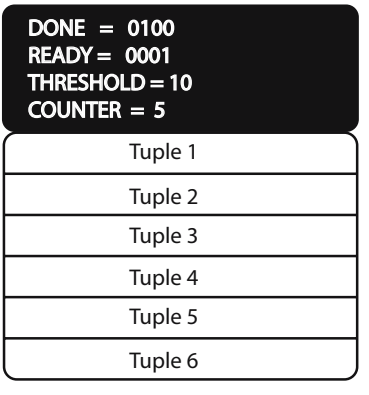

Fig. 4. The Metaqueue Data Structure

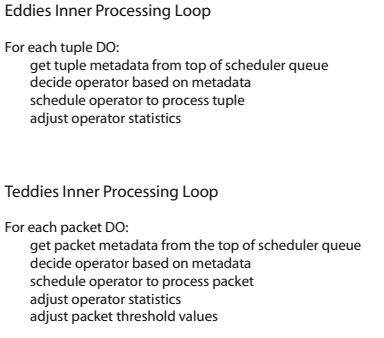

Fig. 5. The Inner Processing Loops for the Eddies and the Teddies Systems

In Teddies, new input tuples are inserted into the SteM operators *before* they are inserted into the proper metaqueues. This ensures that the insert/probe invariant is maintained: if the input tuples are already ordered by timestamp, then whenever a tuple probes a SteM, all tuples with an earlier timestamp are guaranteed to be present in the SteM, no matter in what order the metaqueues are actually scheduled. This invariant must be maintained to ensure that no potential output tuples are lost. This is in contrast to the policy employed by the Eddies system. In the Eddies system, an input tuple is inserted into the SteMs at the same time that the tuple first probes a SteM: in this manner the insert/probe invariant for the Eddies is maintained.

3.2 The Adaptive Packetization Algorithm

The second major aspect of the Teddies query processor is the adaptive scheduler that provides two functionalities: (1) dynamically adjusting the level of packetization of each individual metaqueue; and (2) making all packet routing decisions – that is deciding which query operator to schedule next for a given packet. This is in contrast to the Eddies adaptive scheduler that focuses only on the routing of individual tuples.

The adaptive packetization algorithm utilizes the metaqueue and scheduler queue statistics to dynamically adjust the number of tuples allowable in each metaqueue. The scheduler queue contains references to unique metaqueues, where every metaqueue is either on the scheduler queue or is currently ineligible to be scheduled. For each round of the scheduler, the top metaqueue of the scheduler queue is processed. The READY bit of the top metaqueue is examined and the router policy is applied. Based on the outcome of the router policy, the entire train of tuples in the metaqueue are processed *en masse* by the selected query plan operator. All tuples output from this operator are either output from the system or placed into a metaqueue based on the READY and DONE bits of the input metaqueue. After all of its tuples are thus processed, the top metaqueue is removed from the scheduler queue, its occupancy is decremented by the number of tuples processed, its counter is reset to zero, and the counter value of any currently unscheduled metaqueue is incremented by one. At this point, as a result of this adjustment, another metaqueue may become eligible to be scheduled; if so, it is placed at the end of the scheduler queue where it awaits its turn to be processed.

The metaqueue selection, i.e., the criterion used to decide the eligibility of a given metaqueue to be scheduled, is based on the three values introduced in Section 3.1: the occupancy, the threshold, and the counter. At the end of each round of scheduler execution, if the occupancy value is greater than or equal to its threshold value, the metaqueue is moved from the unscheduled state and placed at the end of the scheduler queue. The threshold value is a dynamic quantity that is adjusted as follows: if the occupancy of a metaqueue is at least twice its current threshold when the metaqueue is finally processed from the head of the scheduler queue, then the threshold is doubled. On the other hand, if a metaqueue's counter value, after being incremented during a round of the scheduler, is greater than a preset limit (which is a preset system parameter), then the threshold value of this metaqueue is halved. In this way, those scheduled metaqueues tend to accumulate many tuples per round of the scheduler increase their threshold for more efficient processing, and, on the other hand, those unscheduled metaqueues that tend to accumulate few tuples per round lower their threshold to the point that they become eligible to be scheduled, thus avoiding starvation. The dynamics of the threshold value are thus determined by the rate at which a metaqueue gains occupants: high-rate metaqueues accumulate relatively more tuples before being scheduled, allowing for efficient train-processing, while low-rate metaqueues do not starve as their threshold values are lowered in response.

The second function of the Teddies scheduler is implemented by the algorithm that selects the operator to process the metaqueue on the top of the scheduler queue. This part of the scheduler is identical in essence to the Eddies scheduler with one distinction. Consider the inner loop processing of the Eddies and the Teddies system shown in Figure 5. Both schedulers schedule based on the READY bits of the entity on the top of the scheduler queue. However, the Teddies inner loop has an additional step – ("*adjust packet threshold values*") – not shared by the Eddies inner processing loop. This step invokes the adaptive packetization algorithm. Moreover, it should be noted that although all other steps are common to both inner loops, the Teddies inner loop runs fewer times if the data tuples are effectively grouped into packets. This allows the adaptive scheduler to potentially gain efficiencies by trading off the cost of performing the extra packetization step with the reduction in the total number of times the scheduler itself is invoked.

4 Experimental Evaluation

4.1 Experimental Setup and Methodology

To evaluate the Teddies design, we implemented both the Teddies and the Eddies architectures and ran sample queries to compare their relative performance. To ensure an even comparison, the implementations of Teddies and Eddies used as much of the same codebase and data structures as possible. As described in Section 3, the main difference between the two implementations is that Teddies uses a set of metaqueues to store the intermediate tuples while the Eddies system uses a single queue for its intermediate tuples. The tuple routing mechanism was also shared between the two systems, by suitably modifying the Eddies code to handle metaqueues instead of tuples. In addition, for the Teddies system, we modified the query operators (the SteM operators) to allow processing of a train of tuples at a single invocation.

For all experiments, we use a windowed, three-way equality join defined on three input streams (i.e., three relations ordered by timestamp), and varied the join selectivities by changing the size of the defining window. The input streams are multiple integer-valued tuples randomly generated from a uniform distribution. Consequently, each of the possible paths through the three-way join query tree have the same cost. Since all possible query plans have the same cost, the actual scheduling policy used by both the Teddies and the Eddies implementations was not exercised. In addition, since we are interested in measuring the performance difference between single tuple routing and packet tuple routing, we ensured that all routing decisions have equal cost by using a simple random routing policy. The cost of packet routing is the sum of the cost of maintaining the metaqueue threshold data and the cost of routing each packet. Hence, by minimizing the routing component cost, we can isolate the cost/benefits of the packet scheme.

Each experiment consists of measuring the total time required to compute the join query $R \bowtie_{R.1=S.1} S \bowtie_{S.2=T.1} T$ for a given batch size and window size. Batch

processing of tuples in this manner gives a measure of the maximum data rate sustainable by the system, as the system never has to spin idly waiting for the arrival of new input data. However, this batch processing introduces the complication of how to introduce new data into the system. The mechanisms used to introduce new tuples into both systems are similar – whenever the respective schedulers cannot proceed because the scheduler queue is empty of intermediate tuples (in the Eddies case) or because the scheduler queue contains no metaqueues above their thresholds (in the Teddies case), a new batch of tuples is added to the scheduler queue (Eddies) or to appropriate metaqueues (Teddies). Thus both systems use a priority scheme whereby new tuples are introduced only when there are no more intermediate tuples available to be scheduled. The number in the new batch of tuples is adjustable and represents the "back pressure" effect that the input has on the system.

4.2 Results and Analysis

The focus of our experiments was to compare the cost of the Teddies system with that of the Eddies system. To this end, the experiments measured the total time required to process a given batch of tuples from randomly generated data sets under varying conditions.

The first set of experiments compared the total runtime cost of the Teddies and the Eddies systems for evaluating the three way join $R \overset{\bowtie}{_{R.1=S.1}} S \overset{\bowtie}{_{S.2=T.1}} T$ with varying join selectivity. We controlled the join selectivity by adjusting the SteM window size – we start with a window size that yields an average selectivity of 1.0, and in each subsequent experiment we double the selectivity by doubling the window size. For this experiment, we used a simple random router policy to consign tuples to operators. The selection of a low cost router was made primarily to highlight the overhead cost of the adaptive packetization mechanism proposed for Teddies.

Figure 6 shows the results of our experiments. The x-axis depicts the join selectivity and the y-axis the runtime ratio. The runtime ratios are *normalized* by dividing the Teddies runtime cost by the total Eddies runtimes (so that Eddies always has ratio 1.0). As can be seen from Figure 6, the total runtime cost of the Eddies system is better than the total runtime cost of the Teddies system for lower join selectivities (between 1.0 and 16.0). However, the Teddies system outperforms the Eddies system at higher selectivities (greater than 32.0). At lower selectivities the opportunities for packetization are fewer, that is, each SteM operator produces few tuples that can be grouped together to form packets. The overhead of the Teddies adaptive packetization is significant in this case. However, higher selectivities result in the formation of larger packets that trade-off the overhead cost of the adaptive processing with the benefits of batch processing, resulting in overall performance improvements.

The second set of experiments were targeted towards measuring the adaptivity of the packetization scheme that has been implemented as a core feature of the Teddies system, and for affirming the results shown in Figure 6. Here, we varied the join selectivities in a manner identical to the join selectivity variation for

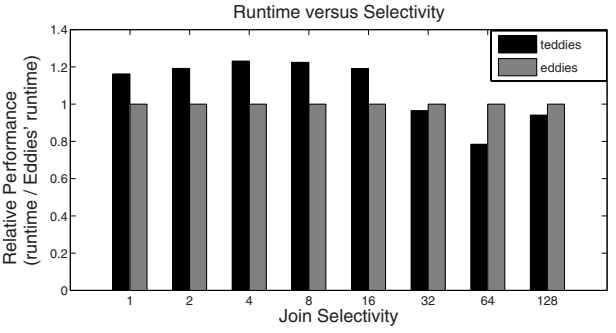

Fig. 6. A Comparison of the Runtimes of Teddies with the Eddies Baseline

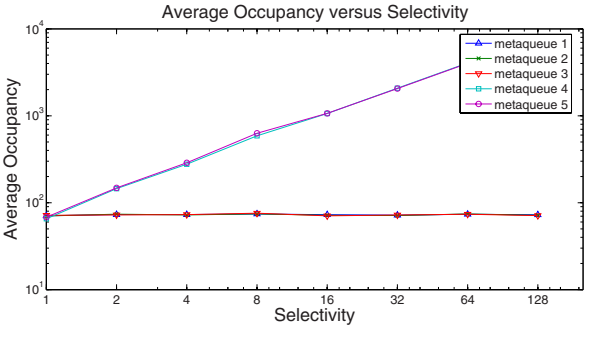

Fig. 7. The Average Occupancy of the Metaqueues for the Join Query

the first set of experiments (Figure 6) and recorded the average occupancies of the metaqueues. To obtain this measure, we instrumented the code as follows – every time a metaqueue is scheduled to be processed by a SteM operator, its occupancy value is recorded and the resulting running average is later computed.

Figure 7 depicts the average occupancy for each of the five metaqueues involved in the evaluation of the three way join $R \overset{\bowtie}{}_{R.1=S.1} S \overset{\bowtie}{}_{S.2=T.1} T$. Here, the x-axis depicts the join selectivity, while the y-axis shows the average occupancy of the metaqueues in terms of the number of tuples. It can be seen that the metaqueue behavior divides into two primary groups. The first three metaqueues handle only input tuples, and hence have an average occupancy that is independent of the join selectivity. However, the last two metaqueues handle tuples output from the first set of SteM operators. These two metaqueues have an average occupancy that exhibits a relationship linear to the selectivity of the SteMs.

The last set of experiments was geared towards measuring the effect of the cost of the routing decision, i.e., the scheduling cost, on the total runtime cost of the Teddies and the Eddies systems. Both the Teddies and the Eddies systems were used to evaluate the three-way join $R \overset{\bowtie}{}_{R.1=S.1} S \overset{\bowtie}{}_{S.2=T.1} T$. However, for this

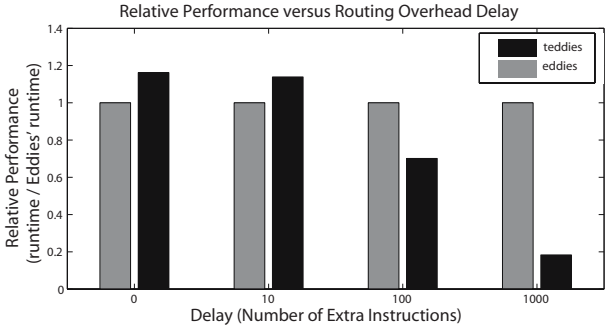

Fig. 8. The Effect of Scheduler Cost

set of experiments, we fixed the join selectivity at 1.0. We simulated the extra scheduling cost by augmenting the random tuple router with an increasing number of dummy instructions every time a routing decision needed to be made. Each dummy instruction simply increments a counter by one. In this manner, we simulate the cost of a more sophisticated tuple router that needs to adapt its decision making to the changing environment.

Figure 8 shows the results of our experiments. Here, the x-axis represents the delay in terms of the total number of extra instructions added per routing decision. The y-axis represents the normalized runtime ratio obtained by dividing the runtime costs by the Eddies runtime cost. Figure 8 shows that as the cost-per-tuple of the Eddies scheduler increases, even in the low selectivity regime (the selectivity here was fixed at 1), the Teddies scheduler achieves significant performance benefits.

Collectively the above graphs (Figure 6–8) present an overall picture of how the Teddies system compares with the Eddies system. Figures 6 and 7, show that the Teddies system performs better when the metaqueues are large – that is when the packet sizes are large. Operating in the high selectivity regime allows Teddies to amortize the cost of maintaining the metaqueues across many tuples. On the other hand, operating in the low selectivity regime when the occupancies of the metaqueues are relatively small, the Teddies adaptive packetization scheme has significant overhead compared to Eddies. However, we point out in Figure 8 that with any reasonable scheduler – a more complex scheduler that presumably makes smarter tuple routing choices – the adaptive packetization of Teddies can provide significant performance benefits even in the lower selectivity regions.

5 Related Work

Adaptive query processing deals primarily with on-the-fly re-optimization of query plans and has been studied in the context of both static databases [Ant96,GC94,INSS92,KD98] and with renewed vigor for continuous queries in stream systems [ZRH04,CCC+02,MWA+03].

Re-optimization strategies in static databases range from those that utilize a run-time statistics collector and reconfigure only the unprocessed portion of the running query plan to improve performance [KD98], to strategies that select and run multiple query plans in parallel and dynamically migrate the running query plans to better performing alternatives [GC94,INSS92]. These approaches have limited direct applicability for stream systems. For instance, reconfiguration of the unprocessed portion of a running query plan is impractical for continuous queries, as the query plan is likely to already be in its execution cycle before migration is needed. The parallel strategy [GC94,INSS92] has been adapted to a stream system [ZRH04], but in of itself is technically infeasible as it is near impossible to apriori define a set of plans for continuous queries.

Migrations strategies in stream systems range from a pause-drain-resume strategy [CCC$^+$02,CCR$^+$03] to adaptation of a parallel strategy [ZRH04] to a tuple-based routing strategy [AH00,MSHR02,DH04]. The pause-drain-resume strategy [CCR$^+$03] has the ability to shutdown processing along a segment of the plan in order to reorder operators within this portion. On completion of the reordering, tuples can resume flow through the previously shutdown segments. A drawback of this strategy is the difficulty of handling stateful operators such as the join operator. Zhu et al. [ZRH04] have developed a moving state and a parallel track strategy to explicitly handle migration of stateful query plans.

Madden et al. [AH00,MSHR02] have developed a novel tuple-based routing called Eddies that shifts away from the traditional re-optimization strategies and makes an optimized decision for each individual tuple. The Eddies system [AH00,MSHR02,Des04,DH04] was designed to allow fine-grained adaptivity at the tuple level. Each tuple in the Eddies model is routed through the query plan independently from other tuples. Tuples are tagged with metadata that allows the scheduler to determine the query operators for a given tuple.The fundamental design choices for an Eddies system implementation include history maintenance for each tuple as it moves through the system, implementation of "stateful" operators such as the join operators, and routing policy decisions for each tuple.

The current Eddies implementation uses bitmaps associated with each tuple to store the metadata. Two types of stateful join operators have been proposed for the Eddies systems: the SteM [Ram01,MSHR02] operator, which implements a version of the symmetric hash join but stores only input tuples in its state hash tables; and the `STAIR` [DH04] operator that now also stores intermediate tuples in the state hash tables. Storing intermediate tuples in the state enhances efficiency due to reduction of repeated probing. However, it complicates the book-keeping needed to ensure that duplicate tuples are not produced.

An efficient and quickly computed routing policy is crucial for the Eddies system, as a routing decision must be made on every tuple inside the inner loop of the system's operation. One such policy [MSHR02] is the lottery based policy whereby an operator either gains or loses tickets based on how many tuples it consumes or produces. Each time through the inner scheduling loop, a lottery must be held to determine which operator wins the next tuple. In addition,

statistics of each operator's behavior must be kept during execution in order to compute the number of tickets each operator holds at any point in time. Other routing policies based on the distribution of attribute values and an estimate of each operator's selectivity have also been proposed in the literature [DH04]. An approach similar to ours that provides virtual batching of tuples has also been proposed [Des04]. Here the scheduler updates the routing decision for each possible tuple history periodically. The period is a fixed size value and each tuple is still routed individually, but the lottery algorithm (or other routing decision algorithm) need not be run for each individual tuple.

6 Conclusions and Future Work

In this paper we have introduced a new type of adaptive query processor. The Teddies adaptive query processor is a hybrid architecture combining the adaptivity of the Eddies model with the efficient bulk processing of the data flow model. We have introduced an adaptive packetization algorithm that fits in the Eddies design scheme but allows for the accumulation of similar tuples into metaqueues, which are then scheduled *en masse* by the query plan router. Our experiments have shown that the packetization algorithm is able to produce large packets in the case of high operator selectivity. Our experiments also show that the bare overhead of packetization is beneficial in the high selectivity regime. Finally, our experiments have shown that the benefits of scheduling tuples in aggregate is beneficial when the cost of making a single routing decision is high, even in the low selectivity regime.

In the future, we plan on investigating the adaptivity of the Teddies architecture. The packetization process gives an adaptive query processor an extra dimension with which to find the most efficient query processing strategy. The above experiments show how this extra dimension given to the adaptive query processor may prove beneficial in the case of high input data rates.

Acknowledgements. The authors would like to thank Henry Kostowski of University of Massachusetts, Lowell, for his contributions to this paper.

References

AH00. Avnur, R., Hellerstein, J.: Eddies: Continuously Adaptive Query Processing. In: SIGMOD, pp. 261–272 (2000)

Ant96. Antoshenkov, G.: Dynamic Optimization of Index Scane Restricted by Booleans. In: International Conference on Data Engineering, pp. 430–440 (1996)

BBD+02. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: Principles of Database Systems (PODS) (2002)

BW01. Babu, S., Widom, J.: Continuous Queries over Data Streams. In: Sigmod Record (2001)

CCC+02. Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams - A New Class of Data Management Applications. In: Int. Conference on Very Large Data Bases, pp. 215–226 (2002)

CCR+03. Carney, D., Cetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., Stonebraker, M.: Operator Scheduling in a Data Stream Manager. In: Int. Conference on Very Large Data Bases (2003)

CDTW00. Chen, J., DeWitt, D., Tian, F., Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: SIGMOD, pp. 379–390 (2000)

Des04. Deshpande, A.: An initial study of overheads of eddies. ACM SIGMOD Record 33(1), 44–49 (2004)

DH04. Deshpande, A., Hellerstein, J.M.: Lifting the Burden of History from Adaptive Query Processing. In: Int. Conference on Very Large Data Bases (2004)

GC94. Graefe, G., Cole, R.: Optimization of Dynamic Query Evaluation Plans. In: International Conference on Management of Data (SIGMOD), pp. 150–160 (1994)

INSS92. Ioannidis, Y., Ng, R.T., Shim, K., Sellis, T.: Parameteric Query Optimization. In: International Conference on Very Large Databases (VLDB), pp. 103–114 (1992)

KD98. Kabra, N., De Witt, D.J.: Efficient Mid-Query Re-Optimization of Sub-optimal Query Execution Plans. In: International Conference on Management of Data (SIGMOD), pp. 106–117 (1998)

MSHR02. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continuously Adaptive Continuous Queries over Streams. In: SIGMOD (2002)

MWA+03. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Resource Management, and Approximation in a Data Stream Management System. In: Conference on Innovative Data Systems Research (2003)

Ram01. Raman, V.: Interactive Query Processing. PhD thesis, UC Berkeley (2001)

ZRH04. Zhu, Y., Rundensteiner, E., Heineman, G.: Dynamic plan migration for continuous queries over data streams. In: SIGMOD (2004)