Dynamic Pipeline Scheduling for Improving Interactive Query Performance*

Tolga Urhan

Propel Corp.
urhan@propel.com

Michael J. Franklin

University of California Berkeley
franklin@cs.berkeley.edu

Abstract

Interactive query performance is becoming an important criterion for online systems where delivering query results in a timely fashion is critical. Pipelined execution is a promising query execution style that can produce the initial portion of the result early and in a continuous fashion. In this paper we propose techniques for delivering results faster in a pipelined query plan. We distinguish between two cases. For cases where the tuples in the query result are of the same importance we propose a dynamic rate-based pipeline scheduling policy that produces more results during the early stages of query execution. For cases where the result tuples have varying degrees of importance, we propose a dynamic tuple regulation algorithm that produces more important tuples during the early stages of query execution. Experimental results show that the proposed approaches significantly improve the interactive behavior in both cases.

1 Introduction

The explosive growth of the Internet and the World Wide Web has made tremendous amounts of data available on-line. Emerging e-commerce applications provide online users easy access to data that may be geographically distributed. Success of many such applications depends on how fast they start producing the *initial/relevant* portion of the result rather than how fast the entire result is computed. Such interactive behavior is desirable in cases where the user only wants to get an idea about the result quickly, or if a subset of the tuples in the result are likely to be sufficient to satisfy the request. Interactive behavior is even more critical in unpredictable communication environments (such as the Internet, mobile networks etc.) where frequent delays or lengthy disconnections render computing the entire result impractical.

Traditional query processing techniques fail to deliver good interactive behavior for wide-area online applications for two reasons:

1. *They cannot cope well with delays in receiving data* - In traditional systems query results are often delivered after most of the input has been received. This results in poor performance if delays are experienced in getting the data.

2. *The execution typically proceeds in a stop-and-go fashion* - Traditional implementations of some relational operators have more than one execution phase which have to be executed sequentially (such as the build and probe phases of hash join). Since they typically produce all or most of their result in the final stage they fail to deliver good interactive behavior unless the initial stages proceed quickly.

1.1 Pipelined Execution

Pipelined execution is emerging as a promising query execution style for cases where producing the initial portion of a result early is important. A fully pipelined execution style (i.e., where all the operators in the query plan execute in a pipelined fashion) delivers result tuples as soon as they are computable from input tuples while allowing the delays to be overlapped by other work. Pipelining works only as long as operators are non-blocking, i.e., they do not stage the data (either in memory or on disk) without producing results for a long time.

In a pipelined query plan the way operators are scheduled and how the flow of data is controlled can lead to significantly different kinds of output behavior, though the cost of the query plan may not be radically affected. In this paper we propose techniques for controlling the flow of data in a pipelined query plan to

*This work was supported in part by the National Science Foundation under NSF grant IIS00-86057, by DARPA under contract number N66001-99-2-8913, and by contributions from IBM, Microsoft, and Siemens.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

further speed up the delivery of the tuples to the user. We distinguish between two query types based on the how the users prefer the result to be delivered and address them separately:

1. *Result tuples are of the same importance* - From the user perspective no output tuple is more preferable than another output tuple. For such a query the best behavior is to output as many result tuples as quickly as possible.

2. *Output tuples have different degrees of importance from the user's perspective* - For such a query the best output behavior is to return *more important* results as early as possible.

For the first type of queries we propose a *Rate-based Pipeline Scheduling* algorithm that prioritizes and schedules the flow of data between pipelined operators so that the result output rate is maximized. The scheduling algorithm takes into account the characteristics of operators, such as their cost and productivity (which could be dynamically changing) when making scheduling decisions. For the second type of queries we develop an *Importance-based Tuple Regulation* algorithm that prioritizes the processing of individual tuples in order to produce more important tuples as fast as possible. It allocates more resources to tuples that are important.

In this paper, we focus on non-blocking pipelined hash joins. The reasons are twofold. First, hash joins are frequently used in complex query plans and are used to combine data from multiple sources. Second, join operators lead to complex scheduling problems since they proceed in more than one operational stage.

2 Overview of Pipelined Hash Joins

The earliest work in non-blocking pipelined hash joins is the Symmetric Hash Join (SHJ) [HS93, WA90, WA91] which was designed to allow a high degree of pipelining in parallel databases. SHJ builds two hash tables, one for each source. When a tuple arrives on one of the inputs, it is first inserted into the hash table for that input, and then immediately used to probe the hash table of the other input. A result tuple will be produced as soon as a match is found.

2.1 The Two Stages Double Pipelined Hash Join

SHJ requires enough memory for both of its inputs. This requirement prevents SHJ from being used effectively for complex queries with large inputs. The Double Pipelined Hash Join (DPHJ) of the Tukwila project addresses this problem by extending the symmetric hash join to use less memory by allowing parts of the hash tables to be moved to secondary storage [IFFL+99]. It does this by partitioning the inputs. The input tuples are organized in partitions based on their join attribute values. When memory fills up a partition is picked and tuples belonging to that partition are flushed to disk.

DPHJ proceeds in two stages; namely, a *Regular Stage* followed up by a *Cleanup Stage*. The Regular Stage works similarly to SHJ, and joins incoming tuples with the ones already in the memory. It is also responsible for flushing partitions to disk when memory is exhausted. The Regular Stage fails to join matching pairs of tuples if one of them arrives after the other one has been flushed to disk. The Cleanup Stage identifies such pairs using a marking algorithm and joins them by bringing in flushed partitions one-by-one and joining them. It is activated when both inputs are exhausted.

2.2 Three Stages of XJoin

The *XJoin* operator was originally proposed in order to achieve good query response in the presence of slow and bursty delays [UF00]. It is similar to DPHJ in the way it deals with limited memory: A Regular Stage similar in spirit to that of DPHJ manages partitions and performs memory-to-memory joins. The Cleanup Stage runs in a similar fashion. In order to cope with delays, XJoin employs a reactively initiated stage (called the *Reactive Stage*) that is activated when the Regular Stage is blocked due to unavailable input. It uses the tuples from the disk-resident portion of one of the partitions to probe the memory resident portion of the corresponding partition of the other source. The Reactive Stage continues as long as tuples from both inputs are delayed. It allows joining tuples on disk with tuples in memory, and produces more results. The Reactive Stage has the potential of increasing the cost of the query, however, since it runs during delays the extra overhead is effectively hidden. XJoin uses a tuple marking algorithm based on timestamps to prevent duplicate results that can occur due to the overlapping nature of its stages.

3 Rate-based Pipeline Scheduling

Having described pipelined hash join algorithms we now provide a dynamic, pipeline scheduling algorithm based on output rates to improve the execution of a plan containing non-blocking pipelined hash join operators. The algorithm schedules the processing of input tuples so that the *initial portion* of the result is computed quickly. Our solution has the following fundamental characteristics.

- *Stream-based approach* - Rather than scheduling operators, we schedule "streams". A stream is a unit of execution that consumes tuples from an input and produces a result, which is then delivered to another consumer stream (Figure 1). Adopting a stream based approach allows making decisions tailored to the characteristics of streams. In the

case of pipelined hash joins we have the four possible streams for each join operator: Two streams each running the Regular Stage one for each of the two inputs, one stream for the Cleanup Stage (for DPHJ and XJoin only), and one stream for the Reactive Stage (for XJoin only).

- *Rate-based scheduling decisions* - The scheduling policy we propose aims at increasing the rate at which result tuples are produced in order to achieve better interactive performance. When making decisions, it takes into account various characteristics of input streams, such as processing costs, expected output cardinality etc.

- *Dynamic* - The schedule is determined dynamically during run-time and can change when the system behavior changes due to inputs that are blocked, operators that finish their execution, etc.

In a pipelined execution model the unit of execution is the processing of a single tuple on a single stream. The propagation of an input tuple along adjacent streams may ultimately cause a result tuple to be produced at the top. Figure 1 shows an example of a pipelined query plan which has many such streams. A scheduling policy is needed to determine which stream to process when more than one has available tuples. Scheduling of streams must be done in a way that improves the interactive behavior.

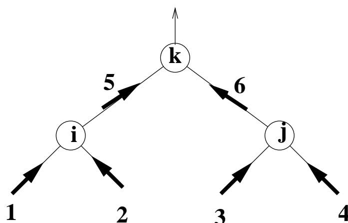

Figure 1: A query plan can have many streams

We address this problem by taking a *rate-based* approach to scheduling individual streams. In the rate-based approach we manage the scheduling of streams so that more *result* tuples are produced at the *top* of the query plan in a given unit of time. More specifically the algorithm we present schedules at each instance, the stream with the maximum expected output rate. The output rate of a stream refers to the rate at which that stream contributes to the *final* result.

3.1 Computing Output Rates

Equation 1 gives the expected global output rate, $OR^{global}(s)$, of a stream s, where $O^{global}(s)$ is the expected number of *result* tuples due to processing *one tuple* on stream s *and* its ancestors, and $C^{global}(s)$ is the time cost of processing the input tuple and propagating the result up the query plan.

$$OR^{global}(s) = \frac{O^{global}(s)}{C^{global}(s)} \quad (1)$$

Notice that all streams below the root produce intermediate tuples. Their output has to be processed by parent streams before a final output tuple can be produced (i.e., the output has to be propagated up to the top operator). We factor the effect of this propagation into the computation of the O^{global} and C^{global} functions:

$$\begin{aligned} O^{global}(s) &= pr_s \times pr_{parent(s)} \times \ldots \times pr_{top} \\ C^{global}(s) &= Cost(s) + Cost(parent(s)) \times pr_s + \\ &\quad Cost(parent(parent(s))) \times pr_s \times pr_{parent(s)} + \ldots \end{aligned}$$

In the equations above, the term $parent(x)$ denotes the parent of stream s to which stream s sends its output. $Cost(s)$ is the time cost of processing one input tuple along stream s; *top* denotes the topmost stream which returns the result tuples; pr_s denotes the productivity of a stream, i.e., how many output tuples it produces per input tuple processed. The productivity differs from the selectivity in that, selectivity describes how many result tuples will *eventually* be produced, whereas productivity describes how many tuples will be produced *at that point* in the execution. The productivity of a stream strongly depends on the selectivity of the join predicate as well as how many tuples have been read by the two streams.

$O^{global}(s)$ is trivially computed by multiplying the productivity estimates of all the streams an input tuple to stream s has to travel in order to propagate all the way to the top operator. The computation of $C^{global}(s)$, on the other hand, simply totals the local cost of propagating an input tuple along each stream starting from stream s. Cost figures for individual streams are scaled by multiplying them by the number of input tuples that the stream is expected to receive and process. Computing the value of $Cost(s)$ can be done by keeping runtime cost statistics for stream s.

3.1.1 Scheduling the Regular Stage

The productivity of a Regular Stage stream s, denoted as pr_s, can be computed as follows:

$$pr_s = \sigma_{join} \times CurCard(sibling(s))$$

In the equation above, $sibling(s)$ denotes the stream with whose tuples stream s joins (i.e., in Figure 1 the sibling of stream 1 is stream 2), and σ_{join} denotes the selectivity of the join predicate between these two streams. $CurCard(s)$ denotes the number of tuples that have been read so far by stream s. Intuitively, each tuple processed by a regular stage stream s can potentially join with any of the $CurCard(sibling(s))$

number of tuples received by its sibling with a probability given by the join selectivity, i.e., σ_{join}.

Notice that even in a static system the value of $OR^{global}(s)$ changes due to the ever-increasing value of $CurCard$. As a result of this, the scheduling order changes dynamically and it may be required to switch between streams even if the current stream is not blocked. This phenomenon is the reason for adopting a dynamic scheduling strategy.

3.1.2 Scheduling the Cleanup Stage

The Cleanup Stage Stream joins the accumulated tuples of two inputs and produces only the output that has not been produced by the two Regular Stages that operate on those inputs. Therefore its productivity is:

$$pr_s = Card_{left} \times Card_{right} \times \sigma_{join} - N_{produced}$$

In the equation above $Card_{left}$ is the cardinality of the left input (similarly for the right) and $N_{produced}$ is the number of tuples that have been produced by the Regular Stages. The Cleanup Stage becomes runnable only after the two inputs of the join operator have been fully received.

3.2 Scheduling the Reactive Stage

The Reactive Stage joins tuples accumulated on disk with tuples in the memory and produces only the output tuples that were not produced by the Regular Stages. The productivity can be computed using the number of memory- and disk-resident tuples (respectively denoted as N_{built} and N_{probed}), join selectivity, and the number of tuples that have been previously produced.

$$\begin{aligned} pr_s &= N_{probed} \times N_{built} \times \sigma_{partition} - N_{produced} \\ \sigma_{partition} &= \sigma_{join} \times N_{partitions} \end{aligned}$$

Notice that instead of using the join selectivity σ_{join} we used $\sigma_{partition}$, i.e., the partition selectivity. When the tuples are organized into partitions the probability of having two matching tuples increases. Intuitively, a tuple T_A that is in the same partition as another tuple, T_B has more chance of matching tuple T_B than an arbitrary tuple T_C .

By comparing the output rates of all Reactive Stage streams it is possible to find which Reactive Stage to execute if more than one could be executed. However, a more interesting side-effect of comparing output rates of streams is that a Reactive Stage could be activated even when there are no delays. This happens when a Reactive Stage stream has an output rate larger than other first stage streams. This case could occur when enough tuples have accumulated on the disk but there was not sufficient delay to trigger the execution of the Reactive Stage.

Notice that this approach goes against the original policy used to activate the Reactive Stage, i.e. when the inputs are delayed. The original policy for activating the Reactive Stage tries to overlap the extra cost of the Reactive Stage by scheduling it to run during delays. This helps control the overall cost of the query while producing more tuples earlier. When there are no delays, only the first stage, which joins memory-resident tuples is scheduled. This approach, however, seriously reduces the number of result tuples produced when the memory is limited.

Using output rate information for scheduling streams solves this problem by allowing the Reactive Stage to be executed when its expected output rate becomes large enough. Since this could happen even in the absence of delays the extra cost of executing the Reactive Stage will increase the total cost of the query and will delay the completion of the query. Executing the stage, however, allows more output to be produced earlier.

3.3 Implementation Issues

The rate based scheduling algorithm schedules at each instance, the stream with the maximum expected output rate. The output rates, however, can change dynamically during the execution of a query plan. Therefore the scheduling of the streams must dynamically adapt to the ever-changing output rates.

Ideally the output rates would be recomputed every time a tuple is processed by a stream. Such fine grained scheduling can be, however, extremely costly. Our approach is to invoke the scheduler periodically (e.g., once every second) in order to avoid this overhead. When invoked, the scheduler re-computes output rates of all streams and schedules the stream with the maximum output rate among the eligible streams. If the scheduled stream becomes ineligible unexpectedly before the next run of the scheduler, the eligible stream with the next higher priority is scheduled.

3.4 Experiments

We compared the interactive performance of the Rate-based approach to three other policies: *RoundRobin*, *CheapestFirst* , and *EqualTime*. The RoundRobin policy employs a FIFO scheduling algorithm. It simply schedules the next stream that is in the ready queue when the currently running stream releases the CPU. CheapestFirst schedules the stream which has the cheapest tuple processing cost and does not take into account the number of output tuples generated. EqualTime schedules the runnable streams in a way that allocates time equally among the streams. In our experiments the scheduler is activated once every second for RateBased, CheapestFirst, and EqualTime policies. In our experiments the scheduler is activated once every second for *RateBased*, *CheapestFirst* , and *EqualTime* policies.

We have implemented a non-blocking pipelined hash join algorithm that has all three stages mentioned in Section 2. The experiments were performed on a Sun Ultra Sparc 5 workstation with 128 MB of RAM.

Our database consists of relations each containing 50K tuples that are populated similar to the Wisconsin benchmark [BDT83] tables using different random seeds. The tuple sizes are fixed at 100 bytes/tuple unless otherwise noted. In the experiments we used 2- and 4-way join queries. The join predicates are one-to-one, hence the result for both queries contains 50K tuples.

Since the output behavior of each stream depends on its cost characteristics we have introduced artificial tuple processing costs associated with each of the input relations. For the 4-way join case processing an input tuple from three inputs cost 2, 5, and 10 times more than the fourth input. For the 2-way join one of the inputs is 5 times more costly to process than the other. Using different processing costs allow us to examine the abilities of scheduling policies when faced with different kinds of streams. The cost statistics are dynamically collected as the query is executing.

In the experiments we use an implementation of XJoin in order to see the performance of scheduling policies where all three types of stages are subject to scheduling. We use a small memory setting: Each XJoin operator is allocated the minimum amount of memory it requires.

3.4.1 Results of Experiments

Figures 2 and 3 show the cumulative response times of queries with 4 and 2 input relations. For the case with 2 relations there are 4 streams that can be scheduled (2 Regular Stage streams, 1 Reactive Stage Stream, and 1 Cleanup Stage Stream which becomes runnable only at the end). For the case with 4 relations there are 16 streams. The x-axis shows a count of the result tuples produced and the y-axis shows the time in seconds at which that result tuple was produced. The reason we show the response time for all 50,000 output tuples is to be able to see what effect each policy has on the completion time of the query (i.e. the time last tuple is delivered). However, our focus is on the interactive performance of scheduling decisions, therefore, most of the discussion in this section will apply to the left hand region of the graphs where response times for the initial result tuples are shown.

As can be seen in the graphs, *RateBased* provides the best interactive performance for both cases whereas others suffer from bad decisions. This is especially true for the 4-relation case (Figure 2) in which scheduling policies have to take into account all 16 streams, and hence, have more effect in the output behavior. For that case, *RateBased* provides upto 50% reduction in response time for the initial portion of the result. *RateBased* produces the initial 1% of the result (i.e., 500 tuples) in 41 seconds whereas *EqualTime*, *RoundRobin*, and *CheapestFirst* produce the same amount of tuples in 56, 61, and 117 seconds, respectively.

CheapestFirst is the worst in terms of the interactive performance in both cases and also suffers from a poor completion time for the 4 relation case. This is surprising considering that it schedules the lower-cost streams more often. The reason for its poor performance is twofold: First, it does not schedule expensive Regular Stage Streams that fetch input tuples. This reduces the availability of potentially matching tuples that are needed to produce output. Second, it prefers running the Reactive Stages over running the expensive Regular stages. This not only increases the cost of the query plan with only minor improvements in the output rate, but also prevents expensive stages from being scheduled even if cheaper Regular Stages block (i.e. while waiting for IO etc).

EqualTime gives equal time to all the streams and allows expensive streams to get a chance to execute. This behavior results in the production of more potentially matching tuples, so it produces more result tuples. *RoundRobin* is surprisingly good for the 2 relation case although it has poor interactive performance for the 4 relation case. As with *EqualTime* it allows all streams to get a chance to execute, therefore its interactive performance is better than *CheapestFirst*.

A discussion of the effect of the Reactive Stage on query cost is in order. As has been mentioned above, the execution of the Reactive Stage increases the number of result tuples produced at the expense of increasing the execution cost, and hence, the completion time (which is otherwise constant for a query plan no matter what scheduling policy is used to schedule). By looking at the completion times in the graphs it is possible to tell which policies schedule the Reactive Stage more often then the others. For the 4 relation case *CheapestFirst* ends up invoking the Reactive Stage more often and for the 2 relation case *EqualTime* does.

We have also performed experiments for the large memory case where all input tuples can fit into the memory and saw that Rate-based scheduling policy continued to provide the best interactive performance (not shown due to space limitations).

4 Importance-based Tuple Regulation

Rate based pipeline scheduling aims to produce initial results as quickly as possible, however, it is not optimized to speed up the delivery of important tuples. An effective interactive query system must also take into account the preference of the users in terms of which tuples are more important and *present the important tuples before the less important ones.*

One way to achieve this kind of output behavior is to let the user specify the importance criteria using an ORDER BY clause. The execution engine then places a sort operator at the top of the query plan that orders the result according to the importance criteria.

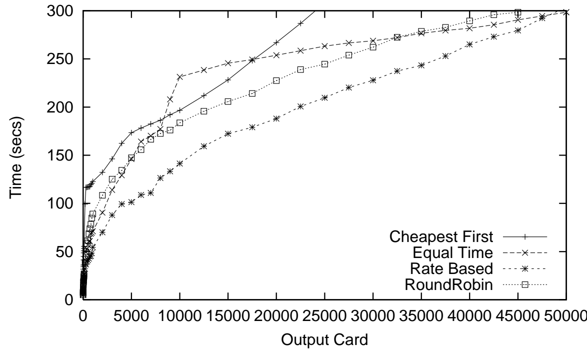

Figure 2: 4 input relations - 16 streams to schedule.

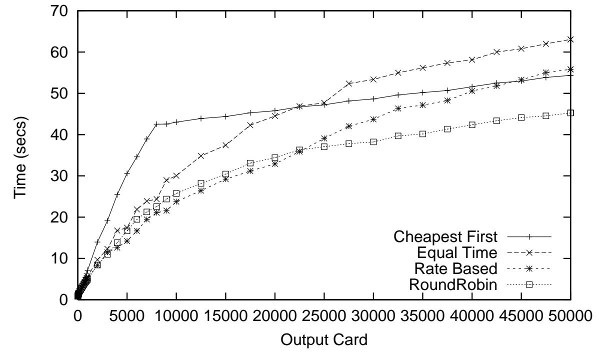

Figure 3: 2 input relations - 4 streams to schedule.

Although this approach returns important tuples before less important ones it has two drawbacks. First, sorting is usually implemented as a blocking operation and produces the sorted result only towards the end of the query execution. Second, the query plan beneath the sort operator usually does not take into account the relative importance of tuples it processes and may end up delivering important tuples late.

The correct approach for handling the first issue (i.e., blocking sort) is to use a sort algorithm that allows the user to "peek" at the result at any point during the query execution and see a sorted list of tuples that has been produced so far by the query plan. The well known insertion sort algorithm can be used to achieve this effect.

In this section we focus on solving the second problem, i.e., processing tuples within the query plan so that more important tuples are output earlier. This can be achieved by giving more emphasis to tuples that are more important (i.e., that are located at the top of the result set with respect to a user defined ordering) and allocating more resources to them. This behavior may come at the expense of a significant delay in the production of less important tuples, but this hardly matters if the decision that will be made by the user eventually depends on the more important tuples.

Rather than making a binary decision about the importance of a tuple (i.e. important vs. unimportant) and processing only the important ones, we take a probabilistic approach. In this approach a tuple is processed with some probability that increases with its importance. In other words, an important tuple will have a higher chance of being processed immediately than a less important tuple. As a result more important tuples are allowed to get ahead of less important ones, while occasionally allowing less important tuples to be processed too. Delivering some of the less important tuples, as opposed to not delivering them at all, provides the user a more global view of the result.

A recent work that is closely related to ours is the "juggle". Juggle is a best-effort reordering algorithm that attempts to solve similar problems as we do. However, the scenarios at which juggle and our methods are aimed differ somewhat, as do the mechanisms they use. We describe the Juggle operator in detail in Section 5.

4.1 Selective Input Processing (SIP)

In order to produce important tuples early, our algorithm selectively processes each tuple with a probability that increases with its importance. SIP works right above a scan operator: When an input tuple is received its importance value (called $i-value$) is computed. The $i-value$ of a tuple is defined to be between 0 and 1 with 0 representing the least important tuple and 1 representing the most important tuple in the result set. Then a random value between 0 and 1 is picked. If the $i-value$ of the tuple is greater than this random number, the tuple is processed. Otherwise the tuple is temporarily staged in memory or on disk (if memory is not available). Staged tuples are processed at the end of the query execution after all the input has been read (Figure 4.)

The importance of the result is derived from the sorting column. Needless to say only tuples that contain the sorting column can be selectively processed as outlined above (e.g., Tuples of Table A in the example). Tuples which do not contain the sorting column are processed normally. This behavior is revised in the next section when we present variations of the basic algorithm.

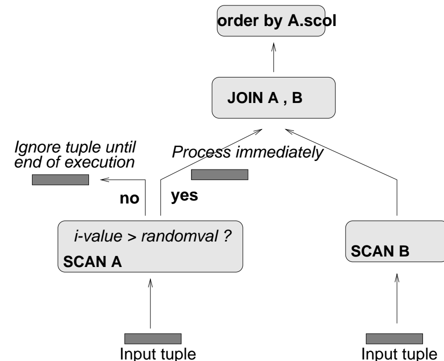

Figure 4: Selective processing of input tuples.

Computing importance

The $i - value$ of a tuple is directly related to its expected position in the result set. We use the "rank" of a tuple, which changes between 0 and 1, to indicate its position in the result in ratio to the expected result cardinality. A rank value of 0.05 would mean that the tuple is expected to be at the 50th position among 1000 result tuples. The rank of a tuple can be computed (approximately) by using the value of its sorting column and statistics (such as the value distribution) that are maintained for this column. A mapping function then converts the rank to an $i - value$. We consider two such functions:

- *Conservative Mapping (CM)*: $i - value = 1 - rank$. Conservative mapping relates the $i - value$ linearly to the expected rank of the tuples.

- *Aggressive Mapping (AM)*: $i - value = 1 - \sqrt{2 \times rank - rank^2}$. When this function is plotted it outlines a lower-left quadrant of a unit circle touching x and y axes at positions (1,0) and (0,1). AM is more selective than CM for all the tuples, i.e., it causes fewer tuples to be processed, and is biased towards important ones.

Notice that computing the $i - value$ of a tuple implicitly relies on query statistics. How these statistics are computed, however, is outside the scope of this paper. The reader is referred to a recent work on mid-query re-optimization of query plans [KD98] for dynamic collection of statistics during query execution.

In this paper our focus is on ways of enforcing the tuple flow so that the importance of tuples (whatever it may be) is used effectively to achieve good interactive output behavior. Therefore, these two functions should be considered only as a rough measure of the importance of tuples rather than as a perfect measure.

4.2 Selective Join Processing (SJP)

Even though SIP partially reorders input tuples it can only be applied at the scan operator which reads the table with the sort column. The output behavior of a query plan involving other operators (such as joins), however, cannot be controlled solely by ordering tuples of one source. As an example, consider a join in which the table with the sort column arrives very early. In that case the output behavior will be determined by the later arriving table because most of the join result will be produced when the tuples of the second table arrive and therefore the output order will be determined by how these tuples arrive.

We address this problem by moving reordering logic from scan operators to join operators. The reason is that the order in which results are output is more dependent on the order in which tuples are matched by the join operators rather than the order that they are delivered by the scan operator.

The resulting algorithm, called Selective Join Processing (SJP), is similar to (SIP) in that it processes tuples selectively using the mapping functions described before. Rather than deciding whether to process an input tuple, it, instead, decides whether two matching tuples should be *joined* based on the importance of the result tuple that would be generated. Notice that in order to estimate the importance of the *result* tuple one of the inputs of the join operator must supply the sort column. This means that the SJP algorithm can be employed at all the join operators that are consumers of the input supplying the sort column.

SJP works as follows: When an input tuple, say T_1, matches a previously received tuple from the other input, say T_2 , SJP first computes the $i - value$ of the result tuple that will be generated (using one of the mapping functions). Then it decides whether to actually join these two tuples with some probability based on the $i - value$. For the case when two tuples are to be joined, a result tuple is generated as usual. For the case when tuples are not joined, the later arriving tuple (in this case T_1) is marked as "ignored" and will not be used again until both inputs have been fully fetched. Tuples marked as ignored are then processed to compute the remainder of the result.

4.3 Further improvements - Value Chaining

SJP regulates tuple flow at all join operators that have an input supplying the rank column. Depending on the shape of the query plan, however, a great portion of the tuple flow may be left unregulated. For instance, in the first plan in Figure 5 SJP can be employed at operators J1, J2, and J3, whereas it is only applicable at operator J1 in the second plan. This means that considerable work may have been spent on computing tuples by operators J2, and J3 that will eventually be ignored because they match with an unimportant tuple at operator J1.

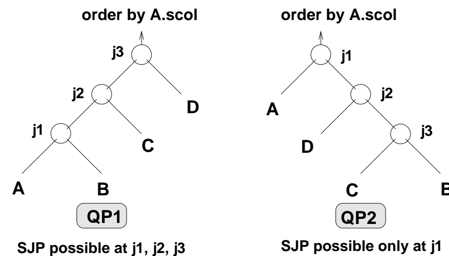

Figure 5: Limited applicability of SJP.

A technique we call Value Chaining, solves this problem for certain kinds of queries, by allowing tuple regulation at other parts of the query plan that do not have access to the sorting column. The key observation is that the importance of the tuples of a table (say table B in Figure 5) can be *uniquely* inferred if there is a primary key - foreign key join between the sorting table (i.e. table A) and table B in the form

of `B.fkey = A.pkey` (i.e., a functional join) and that a matching tuple from A has arrived. In that case a B tuple joins with only one A tuple and therefore its importance can be uniquely identified (i.e., it will be the same as that of the matching A tuple). In other words the importance of an A tuple is propagated to a matching B tuple, allowing tuple regulation at operator above the scan of B (i.e., at J2, and J3 in the figure). Value Chaining requires a hash table to store the primary key and the sorting column of the sorting relation. When a tuple is received from the sorting relation, the value of its sort column and the primary key column are stored together in a hash table, hashed by the primary key value. When a tuple is received from table B we use the value of its foreign key to lookup the hash table. If an entry is found then we use the value of the corresponding sort column to compute the importance of a tuple from table B. If no entry is found then the tuple from table B is processed normally.

The Value Chaining method can be used in conjunction with either SIP or SJP. When used with SIP, the lookup occurs for each $Tuple_B$ when it arrives. With SJP the lookup occurs when $Tuple_B$ is about to be joined with another tuple.

4.4 Experimental Results

In order to show the effectiveness of importance-based tuple reordering we have implemented proposed algorithms in PREDATOR and used the same machine as was used in the previous set of experiments described in Section 3.4. The database is also populated similarly. We used simple 2- and 3-way join queries. Queries involve an ORDER BY statement which is assumed to describe the relative importance of tuples.

4.4.1 Basic Performance of SIP and SJP

In the first set of experiments we investigate the basic output behavior of four policies (combination of two reordering policies, *SIP* and *SJP*, and two mapping functions Conservative Mapping, *CM*, and Aggressive Mapping, *AM*). We have also included in our experiments the output behavior of a do-nothing approach (labeled as *NoOrder*) which does not employ any reordering of the tuples at all. Lastly, we have included the behavior of the do-nothing approach in the *idealized* case when the tuples of the sort table arrive already in sorted order. The output behavior of this imaginary case is labeled as *SortedInput*. Inclusion of this case is only meant for investigating the output behavior for the ideal case when the input was already sorted.

We used two variations of a query that joins two input tables. The first query involves joining two tables of equal size each having 50K tuples. The second query involves joining a small table having 10K tuples with another table having 50K tuples. The smaller table supplies the ordering column. The second variation is used to examine the case where the ordering table is received quickly (since it has fewer tuples). In that case the order in which the result tuples are produced is mainly determined by the other table.

Figures 6 and 7 show experimental results for the first variation, and Figures 8 and 9 show the results for the second variation. Each graph shows, for each of the approaches, the output characteristics of the query execution after the query has executed for a certain time (5 seconds, and 25 seconds for both queries). As such, each graph is a "snapshot" of the query result taken at a specific time.[1] The x-axis contains the policies and y-axis indicates the portion of the result (in percentages) output by the policies at that point in time.

For each of the policies five bars are plotted. The first bar represents how much of the most important tuples has been output. We assume that the top 1% of the tuples with respect to the ordering criteria represent the most important tuples. The next three bars represent tuples that are in the top 3, 10, and 30% of the result (i.e., progressively less important tuples.) The fifth bar represents how much of the entire result has been output at that point in time.

Variation 1 - Query with equally sized relations

Figures 6 and 7 show the results with a query joining two equal sized tables. The graphs show that employing a tuple regulation algorithm based on the importance values speeds up delivery of the important tuples compared to *NoOrder*. For instance, after 5 seconds *SJP/AM* has produced 30 % more important tupes than *NoOrder* has. After 25 seconds the improvements are bigger: All four tuple regulation policies are able to return more than twice as many important tuples as *NoOrder*. Also, aggressive versions of SIP and SJP (i.e., *SIP/AM* and *SJP/AM*) perform better than their conservative counterparts (*SIP/CM* and *SJP/CM*) in terms of the amount of important tuples returned. This behavior is, however, at the expense of producing fewer tuples overall. For example, at 25 seconds, about 90 percent of the most important tuples (i.e., the ones in the top 1 percentile) are returned by *SJP/AM*, but only about 25 percent of all the result has been returned by the same policy.

In the idealized case, when the input tuples arrive already sorted, about 22 percent of all the most important tuples are returned in the first 5 seconds (*SortedInput* in Figure 6). The performance of *SortedInput*, however, degrades considerably later performing the same as the *NoOrder*. Figure 7 shows that both *SIP/AM* and *SJP/AM* return about 50 % more important tuples than *SortedInput*. This is both surprising and counterintuitive, as one would assume the query with an already sorted input should outperform one that partially reorders tuples at all.

The reason for this phenomenon is as follows: Af-

[1] The two variants take 46, and 53 seconds respectively if no tuple regulation is applied. The overhead due to employing SIP algorithm increases the completion time by about 2 seconds for both variants, whereas the overhead of SJP increases the completion time by about 4 seconds.

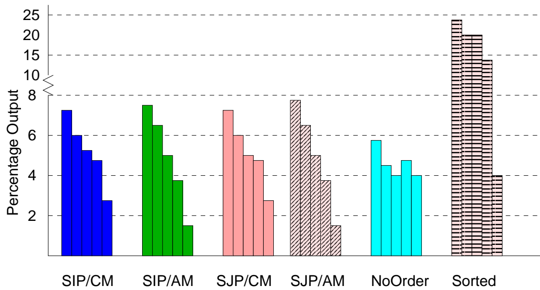

Figure 6: Output after 5 secs. Variation 1.

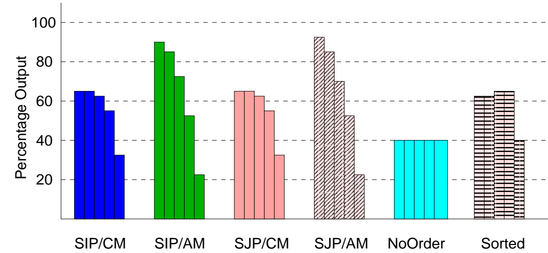

Figure 7: Output after 25 secs. Variation 1.

ter the query has executed 25 seconds *SortedInput* will have received all the tuples in the first four groups (i.e., top 1%, 3%, 10% and 30%) and some of the remaining tuples (since the tuples arrive in sorted order). Those sorted tuples that have not yet been matched with a tuple from the other (non-sorted) input will be produced in the order non-sorted tuples arrive without taking into account the importance of tuples. Since, processing less important tuples delays processing more important tuples the output behavior is penalized. Tuple regulation algorithms avoid this trap by dropping (i.e. hiding) less important tuples thereby saving join overhead that is best allocated to more important tuples.

Variation 2 - Query with small sorting relation

In this variation the table which contains the ordering column is read quickly because it has less tuples. Most of the result is produced when the tuples of the bigger table arrive. This means that the ordering of the result tuples will depend on the ordering of the tuples from the bigger table. We have excluded conservative version of *SIP* and *SJP* from the graphs since they perform worse than their aggressive counterparts.

The figures show that *SJP/AM* performs much better than other policies. This time, however, *SortedInput* is unable to provide good performance at all. This is due to the fact that the order in which join results are produced is more dependent on the order of the tuples received from the bigger table. In other words the sort order does not propagate to the top of the plan. *SIP/AM* also suffers from the same problem. It operates on the sorting table (i.e. the smaller one). However, the entire ordering table has been read by the 25th second, and *SIP/AM* ceases to be effective beyond that point. This behavior manifests itself as the plateau in the second figure. *SJP/AM*, however, continues to regulate tuple flow well after that point.

These two experiments show that SJP has better control over the tuple flow in a query plan. Moreover, aggressively dropping less important tuples improves the delivery of more important tuples.

We also performed experiments to measure the effectiveness of the Value Chaining (VC) technique (not shown due to space limitations). We found that VC improved the delivery of the important results in types of query plans for which it was designed. When used in conjunction with SJP, VC increased the number of important tuples produced by SJP about 50% during the early stages of query execution.

5 Related Work

The Juggle operator [RRH99] is a pipelined best-effort reordering operator whose goal is also to produce important results faster. It takes an unordered set of tuples and produces a result that is nearly sorted. Input tuples are buffered in a sorted list which is updated as new tuples arrive. When the parent operator requests a tuple the Juggle operator returns the tuple at the top of the sorted list. The main difference between the Juggle operator and the tuple regulation algorithms is that, the Juggle operator is sensitive to the rate at which its parent operator consumes the output of the Juggle operator: It returns a tuple when its parent asks for one. If the parent operator requests tuples faster than Juggle operator can produce them, then no reordering will be possible. This is because as soon as the Juggle operator receives a tuple there will be a pending request by the parent. As such juggle degenerates into a “no-op”. Another difference is that the Juggle operator is optimized for cases in which the user preferences change dynamically due to user actions (such as browsing a long list of results and jumping from one region of the result to another region). Our approach is optimized for the case where the importance of the tuples do not change.

The Ripple Join operator [HH99, Hel97] works similar to the three join algorithms described in this paper; SHJ, DPHJ, and XJoin are a type of Ripple Join. Ripple Join was initially developed to approximate query aggregates in a real-time fashion. It, however, imposes a scheduling for processing its inputs in order to provide certain statistical guarantees.

The work described in [WA91] extends authors' earlier work on Symmetric Hash Join with an optimization framework. Although the authors do not address the scheduling issues explicitly, they analyze the behavior of tuple flow in a fully pipelined query plan and its relation to the output characteristics.

The *eddies* [AH00] work relates to our rate-based pipeline scheduling work. An eddy is a query mechanism, and encapsulates a set of pipelined operators and adaptively routes tuples through them. Thus, it

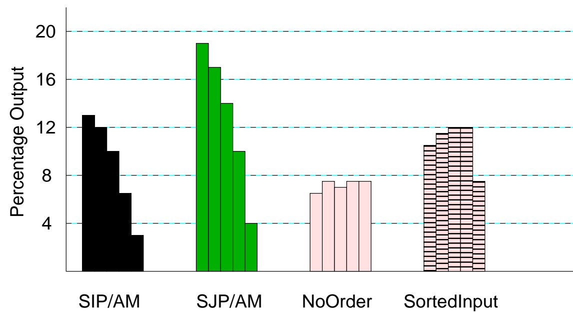

Figure 8: Output after 5 secs. Variation 2.

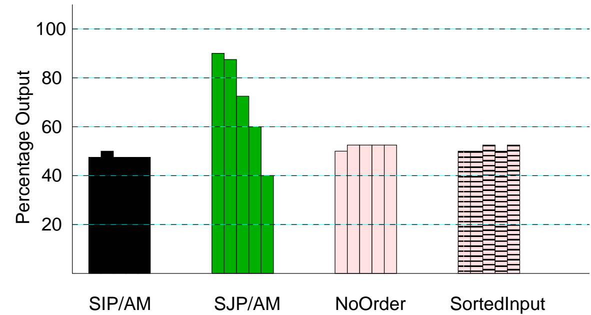

Figure 9: Output after 25 secs. Variation 2.

effectively reorders *operators* in a query plan and aims to reduce the cost of the query execution and *query completion time*. Eddies are designed for cases where it is not possible to get a good query plan or good estimates. Our techniques are intended for less extreme, but still dynamic, situations. Also their objective is slightly different than ours – our Rate-based pipeline scheduling mechanism may slightly increase the query completion time if doing so will improve the delivery of the initial results.

Carey and Kossmann [CK97, CK98] proposed ways of improving the "TOP N" queries where the user is only interested in the topmost N tuples with respect to some ordering criteria. Their algorithms only focus on minimizing the time it takes to *finish* producing first N tuples. They do not focus on giving query results incrementally, as they become available.

6 Conclusions

In this paper we have proposed scheduling algorithms that improve the interactive performance of pipelined queries. We have considered two kinds of queries and addressed them separately. For queries where the tuples in the query result are of the same importance we proposed a dynamic rate-based pipeline scheduling policy that produces more results during the early stages of query execution. For cases where the result tuples have varying degrees of importance, we proposed a dynamic tuple regulation algorithm that produces more important tuples during the early stages of query execution. Our results show that the proposed approaches significantly improve the interactive behavior of these two types of queries.

Pipelined query execution is emerging as a powerful alternative to traditional query processing techniques for scenarios where good user experience is critical. Although scheduling and optimization issues have been extensively studied for decades, techniques for improving the interactiveness of pipelined query execution have only recently been investigated. The need for such techniques will continue to increase with the widespread adoption of E-commerce applications that interact with customers, decision makers, and business partners.

References

[ABF+97] L. Amsaleg, P. Bonnet, M. Franklin, A. Tomasic, and T. Urhan Improving Responsiveness for Wide-Area Data Access. *IEEE Data Eng. Bul.*, 20(3).

[AH00] R. Avnur, J. Hellerstein. Eddies:Continuously Adaptive Query Processing. *ACM SIGMOD Conf.*,2000.

[BDT83] D. Bitton, D. J. Dewitt, C. Turbyfill. Benchmarking Database Systems, a Systematic Approach. *VLDB Conf.*, 1983.

[BM96] R. Bayardo, and D. Miranker. Processing Queries for the First Few Answers. *Proc. 3rd CIKM Conf.*, 1996.

[CK97] M. J. Carey, and D. Kossman. On Saying "Enough Already!" in SQL. *ACM SIGMOD Conf.*, 1997.

[CK98] M. J. Carey, and D. Kossman. Reducing the Breaking Distance of an SQL Query Engine. *VLDB Conf.*, 1998.

[HH99] P. J. Hass, J. M. Hellerstein. Ripple Joins for Online Aggregation. *ACM SIGMOD Conf.*, 1999.

[Hel97] J. M. Hellerstein. Online Processing Redux. *Data Eng. Bul.*, 20(3).

[HS93] W. Hong, M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9-32, 1993.

[IFFL+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, D. S. Weld. An Adaptive Query Execution System for Data Integration. *ACM SIGMOD Conf.*, 1999.

[KD98] N. Kabra, D. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans *ACM SIGMOD Conf.*, 1998.

[RRH99] V. Raman, B. Raman, J. M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. VLDB Conf., 1999.

[SLR97] P. Seshadri, M. Livny, R. Ramakrishnan. The Case for Enhanced Abstract Data Types. *23rd VLDB Conf.*, 1997.

[SP97] P. Seshadri, M. Paskin PREDATOR: An OR-DBMS with Enhanced Data Types. *ACM SIGMOD Conf.*, 1997.

[UF00] T. Urhan, M. J. Franklin XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Eng. Bul.*, 23(2).

[WA90] A. N. Wilschut, and P. M. G. Apers. Pipelining in Query Execution. *Conf. on Databases, Parallel Architectures, and their Applications*, 1991.

[WA91] A. Wilschut, and P. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. *PDIS Conf.*, 1991.