

VPipe: Virtual Pipelining for Scheduling of DAG Stream Query Plans

Song Wang, Chetan Gupta, and Abhay Mehta

Hewlett-Packard Laboratories, Austin, TX, USA
([songw](mailto:songw@hp.com)|[chetan.gupta](mailto:chetan.gupta@hp.com)|[abhay.mehta](mailto:abhay.mehta@hp.com))@hp.com

Abstract. There are data streams all around us that can be harnessed for tremendous business and personal advantage. For an enterprise-level stream processing system such as CHAOS [11] (Continuous, Heterogeneous Analytic Over Streams), handling of complex query plans with resource constraints is challenging. While several scheduling strategies exist for stream processing, efficient scheduling of complex DAG query plans is still largely unsolved. In this paper, we propose a novel execution scheme for scheduling complex directed acyclic graph (DAG) query plans with meta-data enriched stream tuples. Our solution, called Virtual Pipelined Chain (or VPipe Chain for short), effectively extends the “Chain” pipelining scheduling approach to complex DAG query plans.

Key words: stream data processing, performance and scalability

1 Introduction

With data streams all around us that can be harnessed for tremendous business and personal advantage, recent years have witnessed a rapid increase in attention towards data stream management systems (DSMS). DSMS have generated tremendous excitement [1, 8, 12, 17, 19], leading to both academic systems and initial adoptions in the industry. However, the design and development of successful enterprise-level DSMS, which can execute a large number of continuous queries over fast streams, still faces a lot of challenges. Among these challenges, resource management is one of the most critical ones. Resource constraints, especially the memory and CPU constraints, force complex system consideration for scheduling, when facing non-trivial continuous query workloads. Otherwise either the system cannot catch up with the stream speed or the performance measurements cannot reach the requirements.

In this paper, we focus on effective operator scheduling of complex DAG shaped stream query plans with resource constraints. Tree shaped query plans are commonly found in a DSMS to represent single continuous queries, with the root as the data sink, the leaves as the data sources of input streams and historical data, the nodes of the tree as stream operators, and the arcs denoting the queue connecting the operators. However, for an enterprise-level DSMS system, more complex DAG (Directed Acyclic Graph) shaped query plans are very com-

mon and are critical to handlings large continuous query workload. Consider the following examples:

Example 1 (Stream Data Warehouse).

For fast (or close to real-time) online multi-dimensional analysis, stream data warehouse has been proposed to extend traditional OLAP operations to streams [13]. OLAP has also been extended to S-OLAP [16] to achieve sequence analysis over stream of events. The core part of the stream data warehouse is a cube where the cuboids represent some aggregations. All these cuboids form a lattice, i.e. a DAG. When new stream tuples arrive, the cuboids need to be updated, which is equivalent to execution of a large continuous query workload.

Example 2 (Computational Stream Infrastructure).

DSMS has increasingly been used as generic platform for computational streaming applications. In the query language for composition of data-flow graphs, the synchronization operator is a must to express correct semantics. The *Barrier* operator in the SPADE language used in System-S [9] is one such example. Efficient execution of a data-flow involving the barrier operators requires special designed synchronization processing.

Operator scheduling for continuous queries has been tackled in several research efforts (e.g., [2, 6, 7, 14, 18, 20, 21]). However, when applying these approaches on complex DAG shaped query plans, new challenges at runtime further complicate operator scheduling.

- *Input Delay Synchronization (IDS)*: DAG shaped query plans may have operators with large number of fan-in connections. Examples are multi-way joins and union operators. Those operators usually run under the in-order processing semantics [10]. That is, the input stream tuples are processed in the order of their timestamps, irrespective of the connections they come from. Thus, a tuple on arrival, may have to wait for the arrival of other tuples.
- *Shared Output Synchronization (SOS)*: DAG shaped query plans may also have operators with large number of fan-out connections. For example, the copy operators¹ are used to duplicate the shared output. The same output tuples need to be sent to multiple down stream operators. The output tuples cannot be kicked out of the buffer until all down stream operators have consume them.

As an example, Fig. 1 shows an DAG shaped query plan. The input streams are first joined together by operator J_1 to explore the correlation between the streams. Then, operator J_2 is used as a self join to explore the sequential properties of the joined result from J_1 . Also, the joined result from J_1 is filtered by different predicates to serve other queries. While scheduling for the join operators and the union operators, the IDS needs to be considered, while for scheduling the copy operators, the shared output synchronization needs to be considered.

¹ Some DSMS uses the copy operator implicitly by associating it to up-stream operators.

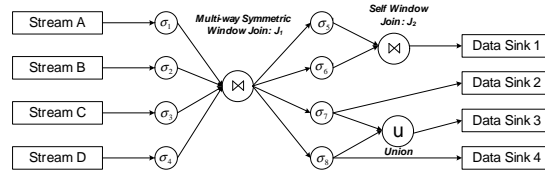


Fig. 1. An Example DAG Shaped Query Plan.

The synchronized stream processing greatly restricts freedom of scheduling. (1) To reduce resource usage or reduce scheduling cost, operator-based scheduling, like Chain [2] and HNR [20], first partition the query plan into linear pipelines or segments as the basic units for scheduling. However, at runtime IDS will cause multiple input operators block and breaks the pipeline, which eventually reduces the usefulness of the optimal partitioning. (2) At runtime, SOS implicitly requires synchronized scheduling of multiple query segments sharing the same stream input. It is not clear as to how the synchronization seamlessly works with the proposed scheduling metrics. (3) To reduce runtime scheduling cost, tuples are processed in batches after each scheduling. A larger batch size is preferred for its lower scheduling cost. However, the larger the batch, the longer the idle waiting time caused by IDS and more the buffer needed for SOS. Determining the optimal batch size at runtime further complicates the problem.

In this paper, we propose a novel execution scheme for scheduling complex DAG query plans by optimal pipelining. Our solution, called Virtual Pipelined execution (or VPipe for short), effectively extends existing scheduling approaches to complex DAG shaped query plans with little runtime cost. The insight is to transform a DAG plan into a linear plan virtually, which is of significant benefit for pipelined operator scheduling.

Our approach enriches each stream tuples with a bitmap recording its origin and path it goes through. Each operator then check the bitmap to determine its actions, process or bypass, on the stream tuples at runtime. Thus, with the help of the enriched stream tuples, a DAG shaped query plan can be treated as one linear plan for scheduling. Our approach thus can address the IDS and SOS problems without sacrificing the optimization opportunities for scheduling.

Our approach is generic in the sense that it is beneficial for all pipelined operator scheduling. Different scheduling algorithm can choose unique metrics to form the VPipe and partition the VPipe accordingly. While any pipelined scheduling can be plugged in and work together with VPipe, we will use the Chain scheduling to illustrate our VPipe approach in the rest of this paper.

Our Contributions:

- We introduce the novel VPipe execution scheme for DAG shaped query plan scheduling.
- We extend the “Chain” scheduling to DAG shaped query plan with the VPipe execution scheme. VPipe Chain combines all the features desired in original Chain scheduling and avoids all the synchronization problems listed above.

- The VPipe Chain scheduling is analytically evaluated and tuned based on a cost model.
- The proposed techniques are evaluated and results of performance comparison of our proposed techniques with state-of-the-art strategies are reported.

Organization of Paper: The rest of the paper is organized as follows: Section 2 briefly reviews the “Chain” scheduling and defines the problem tackled. Section 3 introduces the VPipe execution scheme. Sections 4 present the cost based analysis. Section 5 reports the experimental results. Section 6 contains related work while Section 7 concludes the paper.

2 Preliminaries

2.1 Review of the Chain Scheduling

The Chain scheduling [2] is a near optimal scheduling strategy in terms of minimizing runtime queue memory usage of single stream queries. The insight here is trying to first schedule a pipeline of operators which can reduce the runtime memory requirements the most.

Chain scheduling has been proven to be effective at minimizing runtime memory usage for single stream queries. We refer the reader to [2] for a comprehensive description of the algorithm and proofs. However, Chain scheduling faces several challenges when applied on complex DAG shaped query plans:

- To solve the IDS problem, Chain has to postpone the scheduling of a pipeline (which involves a multi-input operator) until none of the input queues are empty. However, at runtime the pipeline may still have to be broken in the middle if one input queue gets exhausted first. This situation is exacerbated even more, when a large fan-in operator is involved.
- To reduce the memory requirement of SOS, the Chain scheduling introduces deadlines (maximum allowable latency) to avoid starvation. Thus hopefully correlated pipelines will be scheduled close to each other. However it is unclear what is the optimal maximum latency with respect to runtime statistics.

Fig. 2 shows part of an example progress chart of the DAG query plan in Fig. 1. Here we assume that the selection and union operators take same time to process one input tuple while the joins take twice that time. Due to sharing, not all the selections can be pushed down below the joins. To achieve the goal of minimizing the runtime memory requirement, longer pipelines with blocking operators in the middle are preferred by the Chain scheduling (shown by dotted lines). Thus, the synchronizing problem caused by IDS has to be considered. The more joins involved in a pipeline, the harder for Chain scheduling to run a pipeline without interruptions.

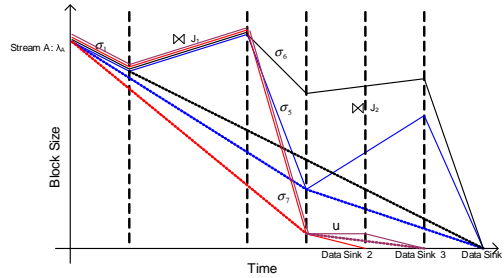


Fig. 2. Progress Chart of the DAG Query Plan in Fig. 1.

2.2 Problem Definition

In this paper, we consider *abstract stream operator (AOP)* which achieves a relational transformation of the input stream tuple(s) into the output stream tuple(s). The operators act as both a stream consumer and a stream producer. The operator may have inner buffer to capture the window semantics. This abstraction is sufficient to capture the semantics of common relational operators like select, project, join, aggregate, and also user-defined-function (UDF). Each AOP is triggered by streamed input when scheduled.

An AOP runs repeatedly over each of the input stream tuples and maps the input tuple to the output tuple(s). Adjacent AOPs can be composed together to form a *pipeline*. Two cascaded AOPs, OP_1 and OP_2 , are pipelined, meaning that the output of OP_1 is consumed by OP_2 one-by-one without waiting for OP_1 to process next input stream tuples. Such pipeline can be extended further downstream. At runtime, intermediate stream tuples will only be buffered at the start operator of each pipeline. Each pipeline is the smallest unit for scheduling.

AOPs are connected with each other by queues, forming a *synchronized DAG plan*. In this paper, we will concentrate on the case of a fixed query plan which does not change over time.

Each stream tuple has an associated timestamp identifying its arrival time at the system. Similar to [4], we assume that the timestamps of stream tuples are globally ordered. In a synchronized DAG plan, the stream tuples are processed in the order of their timestamps. This in-order processing captures the semantics of the symmetric multi-way joins defined in [3].

If a single data stream is input to multiple queries, we assume that multiple copies of the reference (or pointers) of the stream tuples are created by the system and fed to each of the queries separately. Consequently, a tuple can be removed from the system only when all the references are consumed (i.e. SOS is assumed).

Like Chain, we assume that the selectivities and per-tuple processing times of each AOP are known. For a multiple input AOP, like the multi-way join, the selectivities is calculated for each input and may not be identical for all the input streams. We also assume the arrival rate of each input stream is known to the scheduler.

As this work deals with operator level scheduling, we do not discuss the details of stream processing algorithms used in each AOP. Clearly, stream processing algorithms and plan-level optimizations are orthogonal to our focus.

3 The VPipe Execution Scheme

Intuitively, the DAG shaped query plan shows the logical dependence between connected operators. Different intermediate streams (inputs or outputs from an operator) are dependent on each other. With enriched stream tuples, we can transform the DAG plan into a linear one with paying an extra cost of bitmap matching.

In a DAG query plan, there are operator paths from the data sources to data sinks. For each of the queues connecting two operators, we can assign a unique bitmap for the purpose of identification of the tuple. A stream tuple passing through a queue at run-time is enriched by the associated bitmap. By checking the bitmap, a stream operator can know which queue the incoming tuple comes from. This allows the operator to judge whether or not to process the tuple. Similarly, from the bitmap, the operator can know where the newly generated result tuple should go. Thus, the bitmap encoded in the tuple itself can be used to represent the operator path during the processing. A direct consequence is that all the enriched stream tuples can form one queue and also form a long pipeline. Thus, there is only one single queue that connects any two operators. One can imagine it to be like single physical pipeline, with valves at regular distances, where the pipeline represents the single queue of tuples and valves are equivalent to our AOPs. In this way we reduce the complex DAG query plan into a linear pipeline of operators. Now, this linear pipeline can be partitioned into segments according to the Chain algorithm.

3.1 Change of Operator Logic

To achieve VPipe execution scheme, a multiple input AOP $aop(tuple, position)$ will be executed as follows.

When a new tuple a arrives on aop

1. *Bypass Check*: Check $a.bitmap$ with aop 's bypass bitmap. If match, bypass, otherwise goto 2.
2. *Process*: Check $a.bitmap$ with aop 's i -th input bitmap. If match, execute $aop(a, i)$.
3. *Mark Result*: Set the output tuple with aop 's output bitmap.

Fig. 3. Execution of an AOP in VPipe.

We pay an extra cost for bitmap matching in the VPipe execution scheme and extra $enqueue()$ and $dequeue()$ operations. By restricting the size of the bitmap to a word (32 bits), the bitmap matching and setting can be done efficiently

in one CPU cycle. Compared to the cost of common AOPs, the extra cost is negligible in practice.

Bitmap Assignment

Every AOP has one or multiple input bitmaps, corresponding to different input queues. The bypass bitmap is any bitmap except those input bitmaps. Using bitwise operation, the bitmap checking can be easily achieved. For example, suppose the bitmap is 4 bits long and the three input bitmap of AOP aop is 0001, 0010 and 0011. Let x denote the bitmap of the incoming stream tuple. Then if $x \text{ AND } 0011 == 0000$, this tuple is bypassed by aop .

Statistic Calculation

Like Chain, we assume that the selectivities and per-tuple processing times of each AOP are known. For a multiple input AOP, like the multi-way join, the selectivities are calculated for each input and may not be identical for all the input streams. We also assume the arrival rate of each input stream is known to the scheduler. With these statistics, we can easily calculate the relevant statistics of AOPs in the VPipe. Figure 4 shows an example.

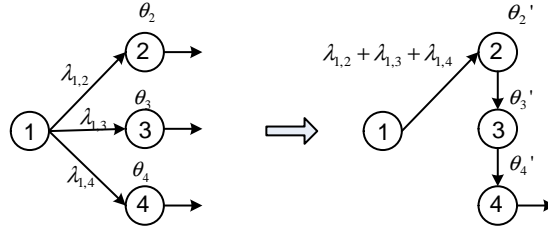


Fig. 4. Calculation of Statistics: an Example.

Suppose the VPipe is formed as $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Then the selectivity θ'_2 can be calculated as follows:

$$\begin{aligned} \theta'_2 &= (\lambda_{1,2}\theta_2 + \lambda_{1,3} + \lambda_{1,4})/(\lambda_{1,2} + \lambda_{1,3} + \lambda_{1,4}) \\ \theta'_3 &= (\lambda_{1,2}\theta_2 + \lambda_{1,3}\theta_3 + \lambda_{1,4})/(\lambda_{1,2}\theta_2 + \lambda_{1,3} + \lambda_{1,4}) \\ \theta'_4 &= (\lambda_{1,2}\theta_2 + \lambda_{1,3}\theta_3 + \lambda_{1,4}\theta_4)/(\lambda_{1,2}\theta_2 + \lambda_{1,3}\theta_3 + \lambda_{1,4}) \end{aligned} \quad (1)$$

Similarly, we can calculate some other metrics such as the computation time of the AOPs in the VPipe. Note that in the calculation, we need not consider the down stream operators.

Application of the Chain on VPipe

The last step is to directly apply the Chain scheduling algorithm on the linear VPipe. Now the DAG plan can just be treated as a one input stream query plan by the Chain scheduling.

4 Stochastic Analysis of Chain

In this section, we first introduce the analytical system model and main methodologies used to analysis the asynchronized chain scheduling and VPipe scheduling. We focus on the memory consumption as the major measurement of performance. With comparison with a base line system, the benefit and penalty of using VPipe is shown analytically. In summary, for the Chain scheduling when the system workload is high, the memory consumption due to the IDS and SOS issues will increase accordingly.

4.1 System Model Basis

We assume the data-flow based query plan is a DAG with one or more stream inputs and sinks. We model the run-time system using $M/G/1$ queuing system with priority. That is, we assume the stream tuple arrivals are independent Poisson arrivals (i.e. the M) with arrival rate λ_i . We also assume the service time of operators follows arbitrary distribution (i.e. the G) with known expected values $E_i(B)$ and variances. For simplification we only consider one server (i.e. the 1) in this paper. We only consider a stable system, i.e. total system utilization rate $\rho = \sum_i \lambda_i E_i(B) < 1$.

For the Chain scheduling, we only consider tuple based scheduling, i.e. batch size $\rightarrow 1$. This is the ideal case for the Chain scheduling since we ignore context changing cost for small batch size. Scheduling of the operator segments is modeled using priority in the queuing system.

The interested measurements from the system model are as follows. The expected value of these measurements are good enough for our purpose. The subscription i may be associated with them for individual queues.

- The queue length $E(L^q)$ before each operator segments, since the queue length determines the memory consumption.
- The sojourn time $E(S)$ of stream tuples, since it determines the response latency. Sometimes we use the queue waiting time $E(W)$ instead of $E(S)$. They have a direct relation $E(S) = E(W) + E(B)$.

To retrieve expected measurements, we use the *mean value approach* to build analytic model. This approach heavily uses following two rules.

- The Little's Law. Little's law gives a very important relation between queue length and waiting time, i.e. $E(L^q) = \lambda E(W)$.
- The PASTA property. The PASTA (Poisson Arrivals See Time Averages [22]) property states that for queueing systems with Poisson arrivals, i.e. $M/*/*$ systems, the arriving customers find on average the same situation in the queueing system as an outside observer looking at the system.

4.2 Case 1: System Analysis for SOS Synchronization

To ease the presentation of our analysis method, we begin with a simplified SOS-base case where the DAG has a copy operator followed by two downstream AOPs, as shown in Figure 5.

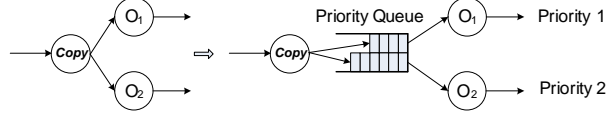


Fig. 5. A simplified SOS-based System Model.

We denote the arrival rate as λ and the mean service time as $E(B_1)$ for operator O_1 , $E(B_2)$ for operator O_2 . Here the O_1 and the O_2 may represent one AOP, or a linear segment of AOPs, which will be scheduled as one unit in the Chain scheduling. Without loss of generality, we assume O_1 has higher priority than O_2 and will be scheduled first by the Chain scheduling. Let's denote the utilization rate by $\rho_1 = \lambda E(B_1)$ and $\rho_2 = \lambda E(B_2)$. This is a typical non-preemptive priority queue setting. With this in place, we can state the following:

From the PASTA property and the property of a stable system, an arriving tuple for O_1 finds with probability ρ_1 that a tuple is being processed by O_1 , with probability ρ_2 that a tuple is being processed by O_2 . Also, at the arrival time, this tuple finds on the average $E(L_1^q)$ tuples in the queue for O_1 . Thus,

$$E(W_1) = E(L_1^q)E(B_1) + \rho_1 E(R_1) + \rho_2 E(R_2).$$

The residual processing time, which means the remaining service time for the one being processed, is

$$E(R_i) = \frac{E(B_i^2)}{2E(B_i)} = \frac{\sigma_{B_i}^2 + E(B_i)^2}{2E(B_i)} = \frac{1}{2}(c_{B_i}^2 + 1)E(B_i), i = 1, 2.$$

Here c_{B_i} is commonly called as *coefficient of variation* and defined as $\frac{\sigma_{B_i}}{E(B_i)}$. This is commonly used to measure the variance of a random variable normalized by its mean. From Little's law, we also have $E(L_1^q) = \lambda E(W_1)$. Thus

$$E(W_1) = \frac{\rho_1 E(R_1) + \rho_2 E(R_2)}{1 - \rho_1}$$

For a tuple arriving at O_2 , it needs to wait an extra amount of time to allow for processing priority 1 tuples at O_1 that arrive later even while it is waiting in the queue. That is,

$$E(W_2) = E(W_1) + E(L_2^q)E(B_2) + E(W_2)\rho_1.$$

We can get

$$E(W_2) = \frac{E(W_1)}{1 - \rho_1 - \rho_2}$$

Now we consider an ideal baseline system for comparison, where no priority is assigned to the operators. We assume the processing time of O_1 and O_2 are independent thus $E(B_{base}) = E(B_1) + E(B_2)$ and $\sigma_{base}^2 = \sigma_1^2 + \sigma_2^2$. Similarly, we can get

$$E(W_{base}) = \frac{\rho_{base} E(R_{base})}{1 - \rho_{base}}.$$

Following [2], we use accumulative memory consumption as the measurement. That is the memory cost is measured as $E(L^q)E(W)$. In Figure 5, the queue of priority 1 is always contained in the queue of priority 2. So we only count the queue of priority 2 for memory consumption. We now calculate the ratio of memory cost between the two systems. Let $r = \frac{E(B_1)}{E(B_2)} = \frac{\rho_1}{\rho_2}$. Since $\rho_{base} = \rho_1 + \rho_2$, we have,

$$\begin{aligned} \frac{Mem_{sos}}{Mem_{base}} &= \left(\frac{E(W_2)}{E(W_{base})} \right)^2 \\ &= \left(\frac{c_{B_1}^2 r^2 + c_{B_2}^2 + r^2 + 1}{(1 - \rho_1)(c_{B_1}^2 r^2 + c_{B_2}^2 + r^2 + 2r + 1)} \right)^2 \end{aligned} \quad (2)$$

What the above equation shows is that, the higher the ratio, the more memory cost exists for the Chain scheduling because of the SOS issue. Intuitively, when ρ_1 is close to 1, i.e. the system is heavily loaded with O_1 , then the ratio will increase significantly. The ratio is plotted in Figure 6 for several typical coefficient of variations. We also assume $c_{B_1} = c_{B_2}$, denoted as c_{B_i} in the figure. From Figure 6, we can see that the memory cost will increase dramatically when the utilization rate of O_1 is high.

4.3 Case 2: System Analysis for IDS Synchronization

For ease of analysis, we begin with a simplified IDS case where the DAG is a binary stream join, as shown in Figure 7.

Here we assume that the two operator segments s_1 and s_2 are obtained from the chain approach as the scheduling units. Without loss of generality, we assume s_1 has a steeper slope than s_2 in the progress chart, i.e., s_1 has higher priority for scheduling than s_2 . Figure 8 shows the queuing system model for this case. According to the Chain scheduling, the model runs with following non-preemptive priority rules shown in Figure 9.

Figure 9 shows the possibility that the high priority pipeline may be broken due to synchronization requirement. The memory cost of the Chain scheduling then is increased, since the broken pipeline needs more memory than the optimal “lower envelope”. In this system model, the probability P , that a tuple from segment s_1 will be blocked at the join J_1 since the counter tuple from segment

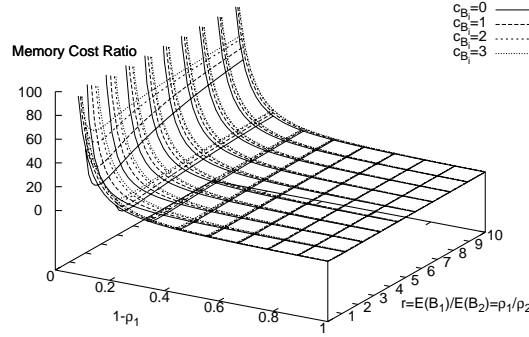


Fig. 6. Memory Cost Comparison: Chain vs. Synchronized Scheduling.

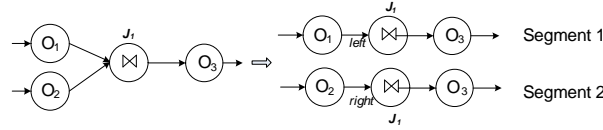


Fig. 7. An Simple IDS Case: Binary Join with Two Operator Segments.

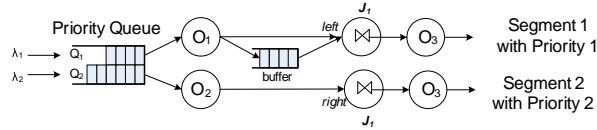


Fig. 8. Queuing System Model for Figure 7

-
- When the system fetch a new tuple t to process
1. Check Q_1 . If $length(Q_1) > 0$, process tuple t from the head of Q_1 ; else goto step 3.
 2. Check Q_2 . If $length(Q_2) == 0$, process t_1 through pipeline $O_1 \rightarrow J_1 \rightarrow O_3$; else process t_1 through O_1 and put t_1 into the buffer.
 3. Process tuple t from the head of Q_2 , through pipeline $O_2 \rightarrow J_1 \rightarrow O_3$;
 4. Release all the tuples in the buffer having timestamp smaller than t and process them through pipeline $J_1 \rightarrow O_3$.
-

Fig. 9. Priority Rule for Execution of an IDS Operator.

s_2 having smaller timestamps has not been processed, is the key measurement of the increased memory cost. To estimate the memory consumption, we use λ_i to denote the mean stream arrival rate of segment i and $E(B_{O_i})$ to denote the mean service time of operator O_1 and O_2 . Also we use $E(B_{J_1, O_3})$ for combined service time of O_J and O_3 . Note that since s_1 has higher priority than s_2 , all the tuples from s_2 will not be blocked at J_1 according to the Chain scheduling.

Given P , we can calculate the expected service time for segment 1 $E(B_1)$ and segment 2 $E(B_2)$ as follows, since part of the job of segment 1 is postponed and conducted in segment 2.

$$\begin{aligned} E(B_1) &= E(B_{O_1}) + (1 - P)E(B_{\bowtie, O_3}) \\ E(B_2) &= E(B_{O_2}) + E(B_{\bowtie, O_3}) + \frac{\lambda_1}{\lambda_2}PE(B_{\bowtie, O_3}) \end{aligned}$$

Since the priority execution does not change the total workload, we still have $\lambda_1 E(B_1) + \lambda_2 E(B_2)$ as:

$$\lambda_1(E(B_{O_1}) + E(B_{\bowtie, O_3})) + \lambda_2(E(B_{O_2}) + E(B_{\bowtie, O_3}))$$

Here we assume that all the service time distributions are independent thus the coefficient of variation and $E(R_1)$ $E(R_2)$ can be calculated as a function of P . Following the same routing shown for case 1, the expected queue length $E(W_1)$ and $E(W_2)$ can also be calculated.

In the rest of this section, we discuss a special case for ease of exposition. Here we assume, that the service time follows exponential distribution, i.e., the coefficient of variation being 1. We also assume $E(B_{O_1}) = E(B_{O_2})$, and normalized as being 1, while $E(B_{\bowtie, O_3}) = n \gg 1$. This simplification is reasonable since the join cost usually dominates the workload. Thus, we have $E(W_2)$ as a function of P :

$$\begin{aligned} E(W_2) &= \frac{\rho_1 E(R_1) + \rho_2 E(R_2)}{(1 - \rho_1)(1 - \rho_1 - \rho_2)}, \text{ where} \\ \rho_1 &= \lambda_1(1 + n - Pn) \\ \rho_2 &= \lambda_2(1 + n + \frac{\lambda_1}{\lambda_2}Pn) \\ E(R_1) &= 1 + n - Pn \\ E(R_2) &= 1 + n + \frac{\lambda_1}{\lambda_2}Pn \end{aligned} \tag{3}$$

From the PASTA property, we also have P equals to the probability that the length of Q_2 is not 0. That is $P = 1 - P(\text{length}_{Q_2} = 0)$. Now we treat the Q_2 and following operators as a single $M/G/1$ system without the effect of Q_1 . Thus $P = \lambda_2 E(\tilde{B})$ and we get another equation of $E(W_2)$ as a function of P . Here we do not have a closed form for the variance of the service time in the second system and thus only obtain an estimation based on coefficient of variation being 1. Thus we have:

$$E(W_2) = \frac{P^2}{\lambda_2(1 - P)} \tag{4}$$

From Equations 3 and 4, we can solve the P . For the special case where $\lambda_1 = \lambda_2 = \lambda$, we have the following relation:

$$\lambda n P^3 + P^2 + (2\lambda^2 n^2 - 3\lambda n)P - 2\lambda^2 n^2 = 0$$

Figure 10 shows the curve $P(\lambda n)$. Since overall work flow is $2\lambda(n+2) < 1$ for a stable system, we pick $\lambda n < 0.5$. From the figure, we can see that the P will increase quickly when the workload increases. Intuitively, when the workload is high, the system is busy processing the tuples from queue Q_1 and more and more tuples will be blocked at the join operator, waiting for the processing of Q_2 tuples.

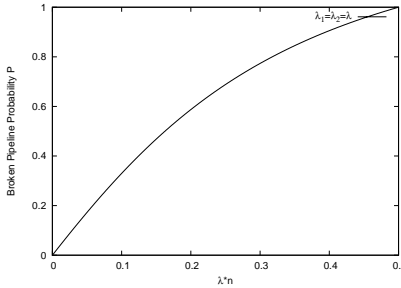


Fig. 10. Increasing Probability of Broken Pipeline with Rising Workload

5 Performance Study

In this section, we present an experimental study that showing the benefit of the VPipe approach comparing with the Chain scheduling. To focus on the performance of the scheduling method instead of effects from implementation differences, our experimental results are obtained from simulations. All our simulations are done using the Extend Simulation Environment.

We perform two different experiments. In the first experiment, we focus on the response time, i.e. the service time for each stream tuples in the running system. As shown in the analysis in Section 4, the service time is a key measurement of the system memory cost. The longer the service time is, the large the accumulative memory cost of the tuples in the system. In the second experiment, we focus on the probability P that the pipeline in the Chain scheduling will be broken due to the synchronization requirement. This experiment is designed to verify our cost model derived in Section 4.

We model the queuing system shown in Figure 7 in the Extend Simulator and measure the performance with respect different arrival rate of the streams. The result shown is the average over all stream tuples passing through the system.

5.1 Experiment 1: Response Time Comparison

Figure 11 shows the experiment results for our VPipe scheduling and the Chain scheduling on the system model in Figure 7. The average response time increases

along with the arrival rate. In the Chain scheduling, the tuples going through Q_1 has higher priority while tuples going through Q_2 has lower priority. The result shows that the response time in our VPipe system lies in the middle between the “Chain-High-Priority” and the “Chain-Low-Priority”. However, overall our VPipe approach wins the Chain scheduling for all the arrival rates, which shows the superior of the VPipe scheduling over the Chain for various workload.

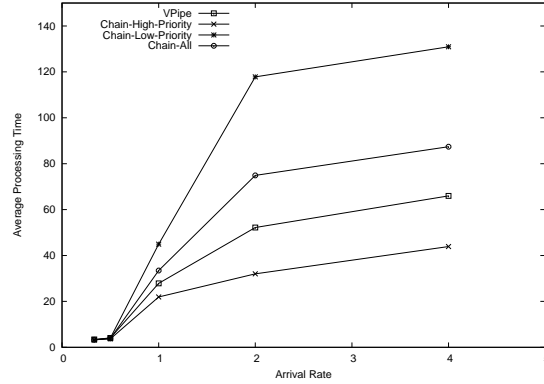


Fig. 11. Response Time Comparison.

5.2 Experiment 2: Broken Pipeline Probability

Figure 12 shows the experiment results for the broken pipeline probability P measured in the Chain scheduling on the system model in Figure 7. The high the P is, the more tuples will run through a broken pipeline which is sub-optimal comparing to the intended pipeline in the Chain scheduling. Thus more memory will be used when P is high. Figure 12 clearly shows that the P will increase sharply along with the system workload, which exactly verifies our cost model developed in Sectionsec.cost.

6 Related Work

Operator scheduling for continuous queries has been tackled in several research efforts (e.g., [2, 6, 7, 14, 18, 20, 21]). Many metrics have been considered for optimization: minimizing the runtime memory, minimizing the average latency, minimizing average slowdown (stretch), etc. However, these approaches do not tackle the special synchronization problem inherent in the complex DAG shaped query plans.

Existing works [5, 15] tackled the idle waiting time problem of IDS by generating heartbeats regularly or on-demand and used them as punctuation to

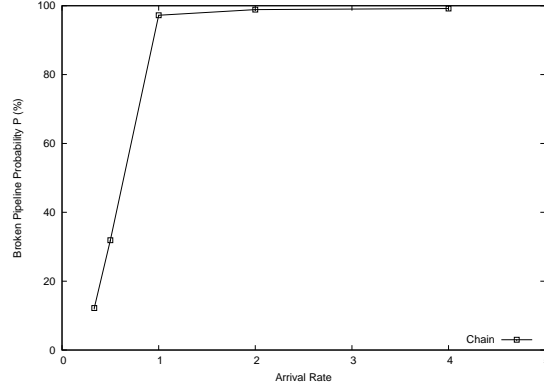


Fig. 12. Broken Pipeline Probability.

trigger execution of other operator segments. However, the fundamental problem of partitioning the DAG-shaped plan across IDS operators is not solved. Thus, opportunities to achieve optimal scheduling goals with longer pipeline are lost.

Previous work [6] tried to solve the SOS problem by scheduling all the first operators after the SOS operator in one group. Although this heuristic approach seems intuitive, it breaks the pipelines at the SOS operators and at all the first operators after the SOS operators. Thus this approach may limit the optimization opportunities to use longer pipeline.

7 Conclusion

Resource management is a key requirement for the performance of runtime DSMS. In this paper, we present a novel scheme for the pipelining execution of a complex DAG query plan. Using stochastic analysis, we show the drawback of the Chain scheduling algorithm and eventually extend the Chain scheduling for complex DAG plans. The experimental study demonstrates the performance benefit of using our solution.

References

1. D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, pages 120–139, 2003.
2. B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264, 2003.
3. S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

4. S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.
5. Y. Bai, H. Thakkar, H. Wang, and C. Zaniolo. Optimizing timestamp management in data stream management systems. In *ICDE*, pages 1334–1338, 2007.
6. Y. Bai and C. Zaniolo. Minimizing latency and memory in dsms: a unified approach to quasi-optimal scheduling. In *SSPS*, pages 58–67, 2008.
7. D. Carney, U. etintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.
8. S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, pages 269–280, 2003.
9. B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD Conference*, pages 1123–1134, 2008.
10. L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
11. C. Gupta, S. Wang, I. Ari, M. Hao, U. Dayal, A. Mehta, M. Marwah, and R. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC'09, to appear*, 2009.
12. M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, and et. al. Nile: A Query Processing Engine for Data Streams. In *ICDE*, page 851, 2004.
13. J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream cube: An architecture for multi-dimensional analysis of data streams. *Distrib. Parallel Databases*, 18(2):173–197, 2005.
14. Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *BNCOD*, pages 16–30, 2004.
15. T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in gigascope. In *VLDB*, pages 1079–1088, 2005.
16. E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. Olap on sequence data. In *SIGMOD*, pages 649–660, 2008.
17. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, pages 245–256, 2003.
18. T. S. B. Pielech and E. A. Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *IDEAS*, pages 445–454, July 2005.
19. E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *VLDB Demo*, pages 1353–1356, 2004.
20. M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, pages 511–522, 2006.
21. T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, Sep 2001.
22. R. W. Wolff. Poisson arrivals see time averages. *Operation Research*, 30(2):223–231, 1982.