Scheduling of Intermittent Query Processing

Saranya Chandrasekaran[1,2(✉)] and S. Sudarshan[2]

[1] U R Rao Satellite Centre, ISRO, Bengaluru, India
[2] IIT Bombay, Mumbai, India
{saranyac,sudarsha}@cse.iitb.ac.in

Abstract. There are many application queries on windows defined over a stream of tuples that must be processed within specified deadlines which are after the window end. Stream processing is usually done either on a tuple-by-tuple basis or in micro-batches. Processing queries over large windows using stream processing engines can be very inefficient since there is often a significant overhead per tuple or micro-batch. Conversely, processing all tuples at the end of the window may result in missed deadlines, and idling of system resources before the window end. We present scheduling schemes for queries on large windows, using large batches, and using priority schemes based on query deadlines and slack time. Our scheduling scheme handles multiple concurrent queries without any prior knowledge of the future query requirements. The proposed scheduling algorithms have been implemented as a custom scheduler, on top of Apache Spark. Our performance study with TPC-H queries shows that our approach of processing can achieve significant computation time reduction compared to naively using Spark Streaming and can also handle stringent deadline cases efficiently.

Keywords: Stream processing · Intermittent query processing

1 Introduction

Many applications carry out analysis on data streams and require results within a specified deadline, i.e. in real time. Stream Processing Engines (SPEs) are widely used for doing such real time analytics. These systems are characterized by high input data rates and usually run a large number of concurrent queries. Stream Processing Engines usually do tuple-by-tuple processing or processing in micro batches. However, doing computation eagerly is not needed for many applications, which perform aggregation on large windows.

Our work was motivated by a leading E-Commerce site in India where analysis was performed on data collected over the day and the results must be made available at some time in the morning of the following day; they wished to move the analysis window from daily to every few hours to support faster responses. The same queries were run on successive windows, i.e. they are recurring queries; as per Zhang et al. [17] 60% of queries in Microsoft SCOPE are recurring in

R. Chbeir et al. (Eds.): IDEAS 2024, LNCS 15511, pp. 363–376, 2025.
https://doi.org/10.1007/978-3-031-83472-1_24

nature. Also, Wang et al. [13] describe Grosbeak, a data warehouse implemented at Alibaba, to handle similar requirements where daily analysis queries must be processed within certain deadlines.

Stream processing engines such as Apache Spark and Apache Flink, update the aggregate as and when a new tuple arrives or when a micro batch of tuples arrives. This method of processing eagerly can lead to significant overheads. Since the results of queries are needed only at the deadline, tuples can be processed in larger batches. In the example considered, if the deadline of the query is say 2 h after the end of the day, one option is to start computation after the end of the window. However, in general, there may not be sufficient time to process the entire data in the gap between the end of the window and the query deadline. In such cases, tuples may be collected for a duration of say one hour, and then processed, and partial aggregates can be finally aggregated at the end. Such batched computation helps in not only reducing the overall computation cost but also in meeting the deadline.

The problem addressed in this paper is that of finding an appropriate batch execution schedule that meets the required query deadlines while minimizing the cost. Here, cost refers to the total time required to process the query. Tang et al. [8] introduce the concept of intermittent query processing, where parts of the query are executed on parts of the input at intermediate points, and the intermediate results are combined at the end to get the final result. Shang et al. [7], Tang et al.[10] propose query optimization by utilizing the query slackness. However, [7,8,10] do not consider query deadlines. Wang et al.[13] propose incremental computation over the available data. Though Wang et al. [13] discuss scheduling, they do not provide further details on how the schedule is generated.

We consider the problem of batching and scheduling of multiple queries, each with its deadline, which runs in a time-shared manner. The system may be dynamic and queries may be added at any time. Our contributions are:

1. We first address the problems of finding batch sizes that keep the computation cost within a predefined multiple of the minimum possible cost (when run as a single batch).
2. We then consider the problem of scheduling the batches of each query based on its input availability and deadlines.
3. We consider both fixed and varying input rates for tuple arrival.
4. We have implemented the scheduling schemes on a custom query scheduler module, built on top of Apache Spark.
5. Our performance study carried out under different scenarios on TPC-H data/queries demonstrate that our optimizations provide significant benefits in terms of reducing cost while meeting the query deadlines.

The special case of batching for the single query case is described in the full version of this paper [1]. The rest of this paper is organized as follows. Section 2 gives the problem description and explains the factors that affect query scheduling. Techniques for batching and scheduling are described in Sect. 3. Section 4 gives an overview of the related work. Implementation aspects are described in Sect. 5. Experimental results are presented in Sect. 6. Section 7 summarizes the conclusions and future work.

2 Problem Description

In this section, we describe the problem specifications and the factors that impact the scheduling of queries with intermittent query processing.

Input data arrives as a stream, and queries analyze the data over a certain time duration and the output is expected within a deadline. The system is assumed to be a soft real time system, where missing the deadline reduces the utility of the results. Our techniques endeavor to complete query execution within the deadline, provided it is feasible.

Query scheduling depends on the ability to model the query execution time. Since the queries are recurring in nature, the time cost model can be derived from historical data. We model the input data rate for computing when batches will be ready for processing, but in Sect. 3.3 we also consider situations where arrival rates may vary from the predicted rate.

Stream processing systems usually allow multiple queries to be processed simultaneously. In this paper, we assume that queries are independent of each other. We assume that queries compute aggregates on windows, and that queries can be computed in an incremental fashion: more specifically, we assume that partial aggregates can be computed on parts (batches) of a stream, and the partial results can be combined later to get the final result. For example, the query to determine the total purchases of each item can be computed by computing partial aggregates for each batch, and later combining the partial aggregates to get the final aggregate values. While aggregation of partial aggregates can also be done intermittently to reduce the final aggregation cost, in this paper, we restrict ourselves to strategies where partial aggregates are combined only in a single final aggregation step.

We assume that each query runs on one input stream and can join the stream with multiple stored or static relations. Extensions to handle some cases of joins between multiple streams are discussed in Sect. 5. We also assume that only tuples that are available at the start of the execution of a given batch are processed in that batch.

We assume that queries run on a time-shared system, where only one query runs at a time. The algorithm proposed in this paper can also be used for a cluster environment, where the same query will be executed in all the cluster nodes, but extensions to allow dividing of resources between queries are part of future work.

The parameters of a query that affect scheduling decisions and relevant notations are given in Table 1. The tuple input rate is assumed to be constant initially, but extended later to handle variable input rates.

The computation cost depends on the batch size and number of batches. A simple linear cost model combines a per tuple processing cost, and a per-batch overhead cost, as follows:

$$Cost = (NumTuples * TupleProcCost) + (NumBatches * OverheadCost)$$

Since actual computation costs may be non-linear, we use a piecewise linear model as an approximation. The model is learnt from actual query execution. Details of cost modelling as applied to TPC-H queries are explained in Sect. 5.

Table 1. Query Parameters

Notation	Description
queryID	Unique Identifier for the query
windStartTime	Time at which tuple arrival starts
windEndTime	Time at which tuple arrival stops
$deadline_Q$	Time by which the query processing must be completed
inputStream	Denotes the query input stream
inputRate	Rate at which tuples arrive for inputStream
numTupleTotal	Total number of tuples to be processed
minCompCost	Time required for processing all the tuples as a single batch
slackTime	The maximum time beyond which the processing cannot be delayed without missing the deadline

3 Scheduling in Dynamic Scenario

In a data analytics system, there may be multiple queries running with the same or different deadlines. Queries may or may not use the same input data stream, and queries may be added or removed from the system arbitrarily. The input rates for each stream and the total number of tuples in the window may also vary.

In this section, we consider such a dynamic scenario. The methodology for determining the batch size is explained in Sect. 3.1. Section 3.2 explains the scheduling scheme with a fixed arrival rate, while Sect. 3.3 extends the scheduling scheme to handle variable input rates.

3.1 Determining Batch Size

The Minimum Computation Cost, minCompCost, which is the time required to process all the tuples as a single batch, can be computed from the cost model for the query. The slack time for the query if computed as a single batch after the window end can be computed as:

$$slackTime = deadline_Q - windEndTime - minCompCost$$

If the slack time is positive, the query can be scheduled after window end but no later than:

$$schStartTime = deadline_Q - minCompCost$$

If the slack time is negative, the query processing cannot be delayed until the end of the window. Instead, the query has to be processed in multiple batches, starting before the window end time.

Analytical queries perform aggregations of data in each window. Processing tuples of one window in multiple batches results in partial aggregation being done on each batch. Hence once all the batches have been processed, the intermediate aggregation results need to be aggregated to get the final aggregation results; we call this step the *final aggregation step*.

In the static case where the arrival rates are fixed, it is possible to break up the input into different batch sizes, and schedule batch execution and final aggregation in such a way that costs are minimized while deadlines are met; details are in the full version of this paper [1]. However, such an approach may delay computation even if the system is idle.

In the dynamic scenario, the scheduler does not have a priori knowledge about future queries. Delaying query execution for the appropriate batch size may result in avoidable missing of deadlines, since new queries may be added to the system at any time. To handle the dynamic scenario, our approach is to process queries intermittently, i.e. whenever the number of tuples available for processing exceeds some minimum batch size. The scheduling of batches is done keeping query deadlines in mind.

The minimum batch size referred to as MinBatchSize, is determined based on the Resource Slack Factor δ_{RSF}. The goal is to pick a minimum batch size such that the overall computation cost is not increased by more than a factor δ_{RSF}. Let N denote the total number of tuples, and $minCompCost_{BatchSize=x}$ denote the computation cost for processing with a batch size of x tuples. Note that the lowest cost is obtained with $x = N$, i.e. processing all the tuples in a single batch. Then MinBatchSize is set to the smallest batch size x such that:

$$minCompCost_{BatchSize=x} \leq \delta_{RSF} * minCompCost_{BatchSize=N}.$$

The parameter δ_{RSF} can be set based on the system utilization. If the system is lightly loaded then a larger δ_{RSF} can be used, allowing for smaller batches. Extensions to automatically adjust δ_{RSF} based on the system load are part of future work.

3.2 Scheduling Using Minimum Batch Size

Once the MinBatchSize is determined, queries can be processed using scheduling techniques such as Least Laxity First (LLF), Earliest Deadline First (EDF), Shortest Job First (SJF), or Round Robin (RR). We assume the system is non-preemptive while processing a single batch of tuples.

For any new queries added to the system, the MinBatchSize is determined as explained earlier, with an additional requirement that the time for processing a batch must be at most some value, denoted as C_{max}. Since the scheduler is non-preemptive, C_{max} ensures that the system can start processing any new query with a delay of not more than C_{max} in case the query has very low slack time. The value of C_{max} has to be decided based on the application latency requirements.

We now consider scheduling with LLF. A query batch can be scheduled at a point in time if the number of tuples available at that time is greater than or equal to MinBatchSize. For each schedulable query i in the list of current queries, $qList$, with batch size x, its SlackTime or laxity at the current point in time is determined as $deadline_{Q_i} - currentTime - CompCost_{Q_{i(batchSize=x)}}$. The query with the least laxity is given the highest priority and its batch is scheduled for execution.

Once a batch has been processed both CPU and memory are released. The intermediate results of the batch are stored on disk. This is unlike streaming data systems which typically retain intermediate results in memory. If all tuples

of a query have been processed, then the final aggregation is done and the query is removed from q_{list}.

EDF, SJF, and RR based scheduling can be implemented with small variations of the LLF implementation. A discussion on the *schedulability* of a given set of jobs, i.e. feasibility of execution of the jobs within the specified deadlines, may be found in the full version of our paper [1] but is omitted here for lack of space.

3.3 Handling Variable Input Rate

So far we have assumed that the input data rate and the total number of tuples in the window are both predictable, i.e., known ahead of time. In practice these can vary, and handling these uncertainties is explained in this section. Scheduling using LLF is explained below. Scheduling using EDF, SJF and RR approaches can be done similarly.

Consider the scenario where the total number of tuples is fixed, but the input rate varies. Here, after MinBatchSize determination, the expected time point at which MinBatchSize will be ready as per the input rate is also estimated. A query batch is considered schedulable if either the input has reached MinBatchSize or the time point has crossed the estimated time for availability of MinBatchSize tuples; in the latter case, a query batch is schedulable even if there are fewer tuples than MinBatchSize. Schedulable queries are sorted based on their slack time, and the query with the least slack time is scheduled for processing. If the actual input rate is faster than or equal to the predicted model then processing will be triggered as and when the required batch size is ready. If the actual input rate is slower, then processing gets triggered based on the estimated input available time, thereby trying to meet the deadline by processing the available tuples instead of waiting for the MinBatchSize readiness. Processing using the available tuples reduces the risk of missing the query deadline.

For dynamic systems where both the input rate and the total number of tuples can vary, we can estimate the expected total number of tuples in the window using any appropriate estimator, which can take into account the actual input rate. Laxity is then computed based on the updated estimates for the total number of tuples in the window. Then the query with the least estimated slack time is processed.

4 Related Work

Many stream processing engines run on the YARN infrastructure. Vavilapalli et al. [11] describe scheduling schemes supported by YARN such as FIFO, Capacity, and Fair, but none of these schemes considers deadlines. Stream processing engines such as Apache Spark and Flink process tuples eagerly with some fixed minimum batch size. However, they too do not consider deadlines. Tuning of batch size is done by some stream processing systems to achieve a balance between throughput and latency, but not in a deadline aware manner, unlike

our work. Ye et al. [15] propose an optimization technique for Spark Streaming configurations without considering deadlines.

Tang et al.[8], Shang et al. [7] point out that the slack period available in queries can be utilized to reduce resource consumption. Tang et al. [8] propose intermittent processing of queries which is triggered at some time interval or based on the number of tuples accumulated. Shang et al. [7] have built a database system namely CrocodileDB which processes queries intermittently based on user inputted frequency. Tang et al. [9] define a new metric, Incrementability, to denote the amount by which a query supports incremental operations. However, none of these papers considers query deadlines.

Grosbeak [13] schedules analysis jobs in non peak hours based on the history of resource utilization. The job is processed in batches which is similar to our approach, but details on scheduling are not discussed. Wang et al. [14] discuss optimization of intermittent query processing, but unlike our case, they do not consider absolute deadlines or scheduling. Zhang et al. [16] show that join queries processed in a lazy manner can perform better than eager processing, but deadlines are not considered.

In hard real time systems where the incoming tuple has to be processed within a certain time to make critical decisions, each tuple is modeled with a deadline. This is different from our problem statement where all tuples in a query have to be processed within a common deadline. Ou et al. [6] propose Tick scheduling where a tick denotes a set of tuples that have the same deadline, but they do not consider the minimization of computation cost by batching.

Scheduling in real time systems has been widely explored and some of the prominent algorithms are EDF, LLF, etc. While EDF and LLF scheduling only aim at completing the query within its deadline, our approach reduces the overall computational cost of each query by processing queries in batches. Other deadline aware scheduling algorithms (see for example the survey [4]) do not consider batching.

In a cluster environment, choosing the optimal resources (e.g. nodes) to minimize cost while meeting deadlines, is considered in [2,3,5,12]. In all these approaches, once the resources are allocated, either Tuple-by-tuple processing or micro batch processing is used.

To the best of our knowledge, our work is the first of its kind which combines batching and scheduling to honor deadlines while minimizing the cost.

5 Implementation Details

The scheduling schemes proposed in this paper have been implemented by building a Custom Scheduler over the Apache Spark architecture. Our scheduling algorithms are agnostic to the underlying stream processing engine. The Custom Scheduler consists of a Query Repository, Schedule Optimizer and Query Scheduler components.

The Query scheduler runs periodically, whenever a query batch completes execution, or if the system is idle, it rechecks query batch readiness periodically.

Since batch sizes are chosen to ensure that no batch takes more than C_{max} time for execution, the scheduler will be invoked within a maximum interval of C_{max} from the previous invocation. The scheduler first checks if any new query has been submitted, and if so it invokes the Schedule Optimizer to compute the MinBatchSize for the query, which does so using the cost model for the query, along with the chosen δ_{RSL}, and C_{max}. The scheduler then selects the queries whose batches are ready for processing and determines the query to be processed based on the chosen scheduling strategy.

When the Query scheduler schedules a query for execution, the Schedule Optimizer invokes the appropriate query operations. Query Repository contains the actual query operations which are to be carried out for each query. As the current implementation uses Spark, it consists of the spark operations which are executed for each batch and the ones which are executed as part of the final aggregation. If any other stream processing framework is used then its corresponding query operations can be implemented in the Query Repository component of our Custom Scheduler. As each batch is processed the intermediate results are stored in a file.

The Schedule Optimizer also keeps track of the batches processed and invokes the final aggregation once all batches of a query have been processed.

For handling queries with joins the following strategy is adopted. For the stream to static join, as the static data does not change, each batch is joined against the static data to get the join results. Typically join conditions in queries ensure that matching tuples from different streams will have timestamp values that are the same or within some bound. For simplicity, our implementation assumes that the corresponding tuples from two streams are available in the same batch. For example, with the TPC-H schema, Orders and their associated Lineitem tuples are assumed to be in the same batch. This assumption can be relaxed, but implementation details depend on the stream-processing application used.

To derive the time cost model, each query is executed individually to measure the execution time at different input batch sizes. Based on observed execution times, a piece-wise linear cost model was arrived at. Similarly, a piece-wise linear model varying on the number of batches was designed to fit the final aggregation cost. As described in [17], most production environments run the same set of queries repeatedly and hence we can build the cost model for queries when they are first executed on the system, and use the cost model when the same query is executed again.

Though file-based input is widely used in many applications as it enables easy information exchange, streaming data platforms such as Kafka are a commonly used alternative. We explored input from a Kafka system by creating two Kafka topics namely Orders and Lineitem with 36 partitions to support parallel processing. We read from Kafka using both the stream and batch approaches. It was observed that the cost incurred using Kafka is at least 3 times more than using file-based inputs. Also, we observed that between Kakfa streaming and batching, streaming incurs considerably more cost. However, processing in batch mode

significantly reduces the cost incurred compared to stream processing, whether we use files as input or the Kafka platform. For our performance studies in Sect. 6, we used file-based inputs to avoid the overheads encountered in reading data from Kafka.

The Custom scheduler does not add any significant time overhead to the overall query processing as the time taken is in the order of milliseconds. Determination of MinBatchSize is done only once for each query.

6 Performance Evaluation

In this section, we present the performance evaluation for our scheduling strategies. Queries were run on a Spark cluster deployed in a standalone mode having 2 Intel Xeon Silver 4116 Processors (2.10 GHz) with 250 GB of RAM. Spark context was configured with 48 cores and 20 GB of memory.

As explained in Sect. 4, there is no prior work that does both batching and scheduling. Hence, in this section, we compare our methods against the standard Spark Streaming and the traditional scheduling algorithms to assess the effect of batching and scheduling respectively. We also further evaluate our method with stringent deadline cases along with variable input rates.

We use a modified version of the TPC-H Dataset of 25 GB. To simulate the input data stream, a timestamp has been added to each record in the relations Orders and Lineitem which are considered as streaming relations. The other relations are considered as static relations. The input stream consists of 4500 files inputted at the rate of 1 file of Orders and 1 file of Lineitem per second. Our study considers a subset of 9 of the TPC-H queries (including queries with stream-to-stream joins, i.e. between Orders and Lineitem, and stream-to-static-relation joins), along with 4 custom queries shown in Table 2.

6.1 Comparison of Custom Scheduler Against Spark Streaming

We first consider the cost reduction obtained due to batching with our approach as well as using Spark Streaming. The cost of execution of a query refers to the

Table 2. Custom Queries

QueryID	Query
CQ1	*SELECT count(*) as totalOrders FROM orders*
CQ2	*SELECT count(*) as totalOrders, orderPriority FROM orders GROUP BY orderPriority*
CQ3	*SELECT count(*) as totalItems, suppKey FROM lineItems GROUP BY suppKey*
CQ4	*SELECT count(*) as totalItems, partKey FROM lineItems GROUP BY partKey*

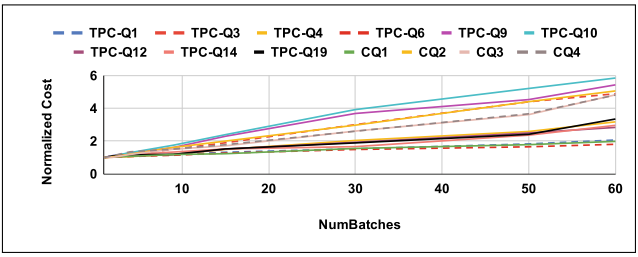


Fig. 1. Cost versus Number of Batches

sum of the query execution time of all the batches and the final aggregation cost. We study the effect of the increase in computation cost as the number of batches increases. The batch size of x in our experiments refers to the number of files processed as part of a single batch. Since there are 4500 files, the number of batches is $4500/x$. All 13 queries were evaluated for different batch sizes. For each of the queries, the minimum cost required for processing it in a single batch is taken as the baseline. The cost incurred when processed with different batch sizes has been normalized w.r.t. this baseline and shown in Fig. 1. It can be observed that the more the number of batches, the more the overall computation cost.

To compare our approach against Spark Streaming, the queries were processed using a Streaming job in Apache Spark with the default i.e. immediate and different batch intervals of 5, 10, 30 and 40 min, with the window aggregation duration of 4500 s. In addition, experiments were done in a one-shot mode where all files were processed in one go. Figure 2 shows the cost incurred for each query, for each batch interval, normalized to the cost of computation in a single batch. It can be observed that the computation cost decreases as the batch interval increases. The least computation cost incurred by Spark Streaming is with the one shot mode of processing. Among all the queries, TPC-Q14 (data labels marked in the figure) has the least normalized computation cost

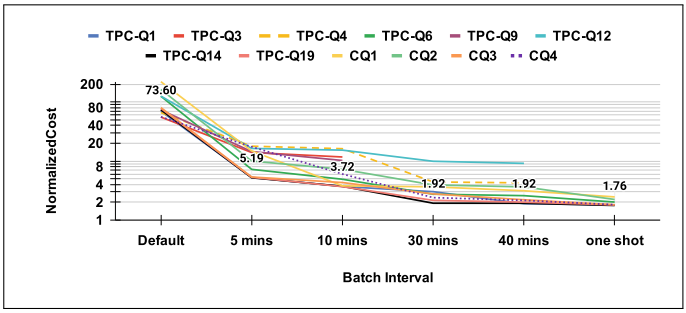


Fig. 2. Normalized Cost (log scale) of Batch processing versus Spark Streaming

with Spark Streaming, which is 1.76 times more than the cost incurred when all tuples are processed in a single batch using our approach.

With Spark Streaming, TPC-Q3, TPC-Q4, TPC-Q9, TPC-Q10 and TPC-Q12 failed for one shot computation and for runs with batch intervals of 30 and 40 min. All these failed queries have join on Orders and Lineitem. Spark Streaming keeps the data in memory for doing the join operations. Thus, as the batch interval increases in Spark Streaming, the amount of data to be stored in memory increases. These cases failed even with the increased memory of 45GB. In contrast with our approach all queries executed successfully with Spark context of 20 GB memory.

Since big data systems need to run multiple queries, the following experiment was carried out where all the TPC-H and the Custom queries were run simultaneously. In our approach, concurrent queries use time-sharing, with one batch of one query executing at a time. Spark Streaming could not support concurrent execution since it ran out of memory. Hence, multiple runs were used, where each run streamed the data to a subset of the queries. The cost incurred in each of the runs was summed up to get the total computation cost. Spark Streaming experiments were done using the default and 10-min batch intervals; for larger intervals, queries TPC-Q3, TPC-Q4, TPC-Q9, TPC-Q12 failed.

Spark streaming costs are compared against the total computation cost incurred using our dynamic mode of scheduling with 50% δ_{RSF} factor using LLF. The cost incurred by Spark Streaming for default and 10-min batch intervals were, respectively, 60 and 12 times the cost using our approach. Thus our approach of batching performs much better than running Spark Streaming for large window operations for multiple queries simultaneously.

6.2 Evaluation of Custom Scheduler for Different Deadlines

Next, we ran experiments to evaluate the performance of the Custom Scheduler with respect to meeting deadlines. For the dynamic scenario, all the TPC-H and the custom queries were considered simultaneously with δ_{RSF} factor of 50% and C_{max} of 30 s. All queries were set with the same window start time and window end time. We chose an arbitrary sequence of queries and set their deadlines such that each query ran as a single batch. The deadline of the first query is set as C_{max} plus the time required for processing all tuples in a single batch starting at the window end. Deadlines of other queries are set as the time required for processing all tuples in a single batch starting at the previous query's deadline. We refer to this set of deadlines as 1D. Further, cases with reduced deadlines were generated, where the deadline was set to window end time plus 0.8, 0.6, 0.4, 0.2, and 0.1 times the assigned gap from the window end time to the deadline for 1D case.

Experiments were carried out for all the above cases using EDF, LLF, SJF and RR. SJF failed for 0.2D and 0.1D and RR failed for 0.4D, 0.2D and 0.1D as some of the queries could not meet their deadlines. EDF and LLF passed all cases except for 0.1D as there is no feasible solution for 0.1D with $\delta_{RSF} = 50\%$. The

fact that SJF and RR failed on multiple cases shows that scheduling strategies based on deadline or slack time are essential to meet query deadlines.

6.3 Comparison with EDF and LLF Without Minimum Batch Size

To demonstrate the importance of having a minimum batch size, we ran experiments using EDF and LLF approaches without a minimum batch size, queries are considered schedulable with all available tuples. Since tuples are input in units of files, which in our experiments consist of around 9300 tuples, 1 file could also be viewed as the minimum batch size. With this approach, EDF and LLF failed to meet deadlines for the 0.4D case, while with our approach of computing minimum batch size with $\delta_{RSF} = 50\%$, deadlines were met down to 0.2D, with both EDF and LLF. Also, even for runs where deadlines were met since the queries with the earliest deadline/least laxity were scheduled more frequently with the small batch sizes, the costs incurred by EDF and LLF were 10 and 7 times more compared to the case where minimum batch size was set with $\delta_{RSF} = 50\%$. Thus our methodology minimizes cost while meeting the deadlines.

6.4 Evaluation of Custom Scheduler With Variable Input Rates

Next, we carried out experiments to assess the impact of variable input rates on the scheduling, where the scheduler cannot predict the arrival rate. Figure 3 shows the data (in units of number of files) that are received at different points in time. FR denotes the fixed rate of arrival case while VR1 to VR4 shows variable rates of input. While both VR1 and VR2 are faster compared to FR, VR2 contains bursty input. Both VR3 and VR4 are slower than FR, and some tuples arrive after the window end of FR.

Results for the case of 0.1D deadlines, with $\delta_{RSF} = 100\%$ are shown in Fig. 4a. With both EDF and LLF the scheduler could complete all queries within their deadlines for all cases as in Fig. 4a. SJF and RR completed all queries for VR1 and VR2 but failed for FR, VR3 and VR4 as some queries missed their deadlines. Similarly, experiments were carried out with variable input rates for 0.2D, $\delta_{RSF} = 50\%$ and the results are shown in Fig. 4b. EDF, LLF and SJF passed all cases while RR failed for VR3 and VR4.

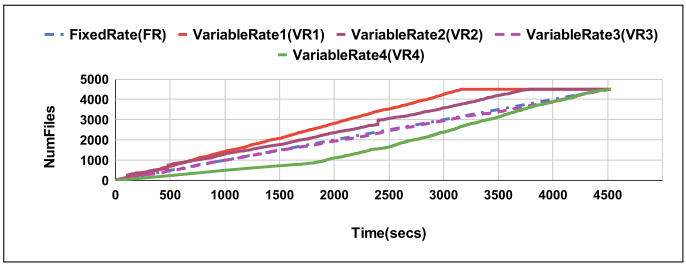
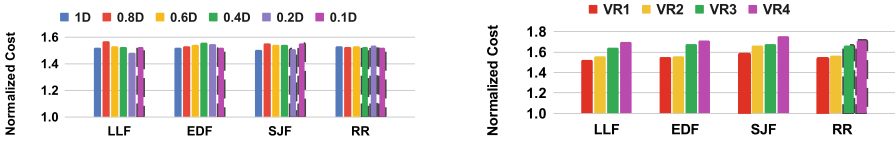


Fig. 3. Variable Input Rate For Multi Query Scenario



(a) 0.1D Cases with δ_{RSF}=100% (b) 0.2D Cases with δ_{RSF}=50%

Bars bordered in Black outline denote missed deadlines

Fig. 4. TPC-H, Custom Queries in Dynamic Scenario with Variable Input Rates

When δ_{RSF} is increased from 50% to 100%, the overall computation cost increases due to a reduction in the MinBatchSize. This can be observed in the Figs. 4a and 4b where the normalized computation cost is around 1.5 and 2.0 for δ_{RSF} of 50% and 100% respectively. Also for slower input profiles(VR3 and VR4), as the total number of batches is more the normalized computation cost is more compared to the cost incurred for the other input profile cases.

Thus, processing on very small batches may not lead to benefits. Further, it may be noted that partial aggregation within a batch is beneficial only if the aggregation result is significantly smaller than the input batch size, which requires the batch to contain on average multiple tuples for each group.

The results show that our scheduling algorithms complete the benchmark queries within their deadlines while keeping the overall computation cost not more than δ_{RSF} fraction compared to the computation cost of processing all tuples in a single batch. The results confirm that EDF and LLF perform better in meeting the deadlines compared to SJF and RR for both fixed and variable input rates. Also, our approach handles stringent deadlines better than just using EDF and LLF with a default batch size that is set without considering δ_{RSF}.

7 Conclusion and Future Work

We have presented techniques for determining appropriate batch sizes and scheduling of multiple queries under a dynamic environment, while handling the uncertainties in the input rate. The results presented in the performance section show that our methods perform better in terms of reducing the overall computation cost while meeting deadlines compared to Spark streaming as well as EDF and LLF with a default small batch size.

There are several directions for future work. Our scheduling techniques can be extended for a cluster setup where resources can be added dynamically to complete queries within the deadline. The cost model proposed in this paper can be correlated to the monetary value that would be required for resource allocation. Our current scheduling model runs one batch of one query at a time across all available resources. This can be extended to support the simultaneous execution of different queries on different subsets of nodes in the cluster.

References

1. Chandrasekaran, S., Sudarshan, S.: Scheduling of intermittent query processing. arXiv:2306.06678 (2024)
2. Cheng, D., et al.: Adaptive scheduling of parallel jobs in spark streaming. In: Proceedings of the IEEE Conference on Computer Communications (INFOCOM), pp. 1–9 (2017)
3. Dimopoulos, S., Krintz, C., Wolski, R.: Justice: a deadline-aware, fair-share resource allocator for implementing multi-analytics. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), pp. 233–244 (2017)
4. Hedayati, S., et al.: MapReduce scheduling algorithms in Hadoop: a systematic study. J. Cloud Comput. **12**(1), 143 (2023)
5. Islam, M., Karunasekera, S., Buyya, R.: dSpark: deadline-based resource allocation for big data applications in Apache Spark. In: Proceedings of the 13th IEEE International Conference on e-Science (e-Science), pp. 89–98 (2017)
6. Ou, Z., et al.: Tick scheduling: a deadline based optimal task scheduling approach for real-time data stream systems. In: Proceedings of the 6th International Conference in Web-Age Information Management (WAIM), pp. 725–730 (2005)
7. Shang, Z., et al.: CrocodileDB: efficient database execution through intelligent deferment. In: Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR) (2020)
8. Tang, D., et al.: Intermittent query processing. Proc. VLDB Endow. **12**(11), 1427–1441 (2019)
9. Tang, D., et al.: Thrifty query execution via incrementability. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1241–1256 (2020)
10. Tang, D., et al.: CrocodileDB in action: resource-efficient query execution by exploiting time slackness. Proc. VLDB Endow. **13**(12), 2937–2940 (2020)
11. Vavilapalli, V.K., et al.: Apache Hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC) (2013)
12. Wang, G., Xu, J., Liu, R., Huang, S.: A hard real-time scheduler for Spark on YARN. In: Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 645–652 (2018)
13. Wang, Z., et al.: Grosbeak: a data warehouse supporting resource-aware incremental computing. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 2797–2800 (2020)
14. Wang, Z., et al.: Tempura: a general cost-based optimizer framework for incremental data processing. VLDB J. **32**(6), 1315–1342 (2023)
15. Ye, Q., Liu, W., Wu, C.Q.: Nostop: a novel configuration optimization scheme for Spark Streaming. In: Proceedings of the 50th International Conference on Parallel Processing (ICPP), pp. 1–10 (2021)
16. Zhang, S., et al.: Parallelizing intra-window join on multicores: an experimental study. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 2089–2101 (2021)
17. Zhang, W., et al.: Deploying a steered query optimizer in production at Microsoft. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 2299–2311 (2022)