Adaptive Input Admission and Management for Parallel Stream Processing

Cagri Balkesen
Systems Group, ETH Zurich, Switzerland
cagri.balkesen@inf.ethz.ch

Nesime Tatbul
Systems Group, ETH Zurich, Switzerland
tatbul@inf.ethz.ch

M. Tamer Özsu
University of Waterloo, Canada
tamer.ozsu@uwaterloo.ca

ABSTRACT

In this paper, we propose a framework for adaptive admission control and management of a large number of dynamic input streams in parallel stream processing engines. The framework takes as input any available information about input stream behaviors and the requirements of the query processing layer, and adaptively decides how to adjust the entry points of streams to the system. As the optimization decisions propagate early from input management layer to the query processing layer, the size of the cluster is minimized, the load balance is maintained, and latency bounds of queries are met in a more effective and timely manner. Declarative integration of external meta-data about data sources makes the system more robust and resource-efficient. Additionally, exploiting knowledge about queries moves data partitioning to the input management layer, where better load balance for query processing can be achieved. We implemented these techniques as a part of the Borealis stream processing system and conducted experiments showing the performance benefits of our framework.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

Data streams; Adaptive Admission Control; Parallelism

1. INTRODUCTION

Stream processing has matured into an influential technology over the past decade. Numerous applications of stream processing have emerged, ranging from the classical cases such as financial market monitoring to more novel ones such as social feed monitoring [25] or crowd-sourced sensing [27]. A common class of these applications is characterized by a large number of autonomous streaming data sources that are highly dynamic in nature, which leads to input data workloads that dynamically fluctuate over time. This dynamism can be due to several reasons, including: *(i)* intermittent disconnection of data sources due to low connectivity or power, lack of activity, etc.; *(ii)* variability of the rates of each source both among each other and over time; and *(iii)* the skew in the data values that they report. On the other hand, the aggregate input volumes generated by these sources can also get very high and can threaten the quality-of-service (QoS) in a real-time stream processing engine (SPE) by overloading its resources. Therefore, an SPE must be equipped with adaptive admission control and query processing techniques in order to be able to serve these applications in a scalable and resource-efficient manner.

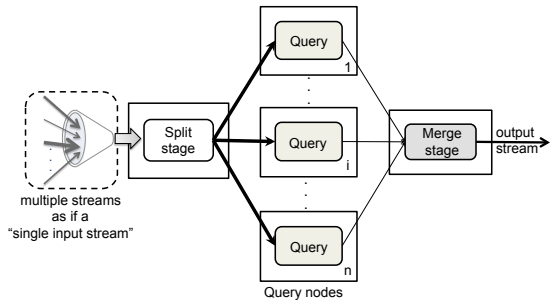


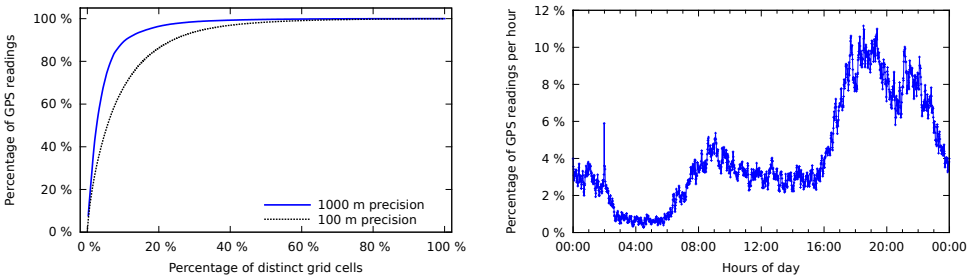
Figure 1: Classic Split-Merge parallelism model.

There has been much recent work on scaling stream processing using parallelization techniques [3, 11, 14, 15, 23, 30]. A significant portion of these rely on a classic partitioned parallelism model, where input streams are partitioned and routed to multiple processing units to be processed in parallel [7]. Figure 1 shows the general parallelization framework that these approaches follow.

While there has been much focus on stream partitioning and parallel query evaluation (the middle part of Figure 1), an important problem that has been ignored is the admission control and management of large number of dynamic input data streams. Since partitioned parallelism requires splitting data streams to multiple processing nodes in a load-balanced manner, it is usually assumed that all the input data streams will arrive at a split stage as a single physical data stream as shown in Figure 1. This model is not realistic, and it introduces scalability issues in the split stage, which itself might require parallelization [3, 30].

In this paper, we propose a general framework for explicit management and admission control of input data streams as first-class entities in a parallel SPE (the left half of Figure 1). In a partitioned parallelism setting, this involves three main

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.



(a) Spatial distribution (b) Temporal distribution

Figure 2: Skewed and dynamic behavior of data sources in the Uber trace.

tasks: *(i)* accepting and assigning individual input streams to one or more split nodes; *(ii)* continuously monitoring behavior of each input stream; and *(iii)* setting the number of split, query, and merge nodes accordingly. It has been shown that binding all input streams to a single split node would introduce scalability issues [30, 3]. On the other hand, binding each stream to a separate split node would not be resource-efficient. Thus, the first important goal in this problem is to minimize the number of split nodes (and deciding the number of query and merge nodes accordingly). Furthermore, the dynamic nature of the inputs require continuous monitoring of the inputs and performing the admission control tasks adaptively. However, changes in input stream assignment would lead to delay and buffering penalties. Therefore, the second important goal is to minimize the cost of stream redirections.

Adaptive admission control and management of input streams in parallel stream processing systems is important, as it provides the opportunity to observe and react to high and dynamically changing loads as soon as they are received by the system, even before reaching the query processing stage. If done right, it would maximize the scalability and resource-efficiency of the system while maintaining QoS guarantees.

This problem is challenging for several reasons. First, optimal input stream assignment can be reduced to the bin-packing problem, which is known to be NP-hard [6]. Second, minimizing the cost of stream redirections requires having knowledge about future behaviors of inputs, which is usually not available in advance. Therefore, heuristic solutions that predict these behaviors are needed.

Moreover, as our motivating example shall demonstrate, forecasting is not the only way to learn the characteristics of input streams. In many areas, such as financial markets, there are quite well-known periods (*i.e.*, opening-closing hours, crises times, etc.) when high-volume streams will stress SPEs. In fact, many factors such as peak rates, periodicity, rate behavior (rising/falling trend) can be known in advance. By declaratively integrating such knowledge about streams as meta-data into the input stream management, the system can be made more robust and resource-efficient. The following example demonstrates the characteristics and significance of applications that our framework addresses.

1.1 Motivating Application

Large-scale sensor network applications have been prominent use cases of data stream processing. With the emergence of smartphones that are capable of collecting a variety of information such as GPS location, it is now possible to treat everyone carrying a smartphone as a "sensor" in a wide range of useful applications that are called "crowd-sourced sensing applications". For example, in crowd-sourced traffic monitoring, smartphones or in-car GPS systems are used as input data sources that report location information on a continuous basis (*e.g.*, Waze social mobile application [27] or the Mobile Millennium project [12]). Real-time traffic monitoring applications then rely heavily on continuous spatio-temporal aggregation and deep analysis of high-volume streams reporting vehicle location and speed. This can be modeled using the following continuous query:

```
Input(Time, VehicleID, Long, Lat, Speed, ReportType)

SELECT getGridCoordinates(Long, Lat, @precision) AS AreaID,
       COUNT(*), AVG(Speed), MAX(Speed), isQueueEnd(*)
FROM Input [RANGE 60 MINUTE SLIDE 1 MINUTE]
WHERE ReportType = 'traffic'
GROUP BY AreaID;
```

We have analyzed a publicly available real-world trace of 1.2 million GPS position reports from a black car dispatching company called *Uber* that operates in San Francisco [26]. In Figure 2, we plot the distribution of GPS readings from this data trace over space (based on 100m x 100m or 1000m x 1000m tiles over the city map) and time (based on different times of day). While the distribution of GPS reports for different tiles follows a highly skewed, Zipf-like distribution (cf. Figure 2(a)), it also varies significantly throughout the day (cf. Figure 2(b)) due to the changing number of data sources (cars in traffic sending reports) as well as their update rates depending on the peak hours and locations.

As this analysis demonstrates, real-time monitoring applications such as the traffic monitoring case described above, impose a number of unique challenges for stream processing. First of all, it is widely projected that the number of smartphones and GPS devices used by drivers will immensely increase over the next few years [8]. As such, these applications typically involve a very large number of data stream sources. Second, these data sources are highly dynamic and transient (*i.e.*, they can join or leave in an unpredictable way). Third, they may exhibit skewed workloads in terms of their update rates and reported values (*e.g.*, more frequent reports from areas where the traffic is moving faster or where mobile connectivity is stronger). Finally, in order for the results to be useful, live reports must be aggregated and analyzed in a correct and timely manner.

1.2 Scope and Contributions

In this paper, we propose a general framework for adaptive admission control and management of input streams for parallel stream processing. The framework takes as input any available information about input stream behaviors and

Figure 3: Input stream admission and management.

requirements of the query processing layer, and adaptively decides how to assign streams to split nodes. The framework treats input streams as first-class citizens of the system, thereby improving the efficiency and robustness of the query processing layer. We also show how our framework can extend a partitioned parallelism framework and effectively be applied to parallelizing sliding window aggregation queries in SPEs.

2. INPUT STREAM ADMISSION AND MANAGEMENT

In data stream processing, workloads of long running queries fluctuate over time with input rates sometimes increasing by orders of magnitude (cf. Figure 2(b)). When workloads are so unpredictable, it is not possible to provision required resources for query processing in advance.

Our adaptive input stream admission and management framework tries to avoid over-provisioning of resources without sacrificing QoS requirements for highly dynamic streaming workloads. The first major problem for such workloads is the high number of distributed input sources with time-varying input rates. This problem, which is more pronounced in a parallel processing setting, requires dynamically changing the entry points of individual streams to the system. The second problem is that streams are also highly transient, meaning they join/leave over time, which requires an admission control mechanism. Finally, based on the overall volume of streams that enter the system, capacity of the processing layer requires adjustment.

An overview of our framework is shown in Figure 3. In contrast to a classic admission control, our framework eventually accepts all the streams and therefore admission is expected to be always successful. In doing so, the framework utilizes knowledge from both sources and queries in a novel manner. Streams are profiled at runtime and their statistics are used for making forecasts. At the same time, users can also specify meta-data about streams. On the other hand, query knowledge such as windowing and grouping information are inferred from queries. Finally, query load information is also monitored at runtime. We describe the framework and its interaction with the input streams layer in detail in this section. Sections 3 and 4 describe its interaction with the query processing layer.

2.1 The Input Stream Assignment Problem

It is impractical to bind multiple, highly dynamic and transient input streams to a single split node [3, 30]. However, binding only a single stream to each split node may extremely overuse resources. Our solution is to periodically check the behavior of streams along with their meta-data and dynamically re-assign some of the streams to different split nodes. The re-assignment of streams is driven by quality requirements of the query and the behavior of streams.

The problem can be formalized as follows. Assume we are given N input streams with average rates $\{R_1, \ldots, R_N\}$ over a certain period, where the maximum of rates is always less than the node processing capacity C, *i.e.* in the worst case a single stream can be processed by a single split node ($\forall_i R_i \leq C + \epsilon$). We are also given M split nodes each with capacity C, where $M \leq N$. The problem is to partition streams (and rates) into a minimum number k of subsets $M_1, M_2, \cdots, M_k$ such that $\sum_{R_i \in M_j} R_i \leq C + \epsilon, \forall j = 1, \ldots, k$. This problem can be reduced to the bin-packing problem, which is known to be NP-hard, but for which there exist heuristic solutions with guaranteed performance bounds [6]. Some of the well-known heuristic solutions are First Fit Decreasing (FFD) and Best Fit Decreasing (BFD). FFD first sorts items in decreasing order of size and then inserts each item to the first bin that it fits. BFD on the other hand keeps bins sorted in increasing order of free space. Whenever something is inserted to a bin, the bin's order on the list changes with the remaining space.

2.2 Near-Optimal Input Stream Assignment

Time varying stream rates make our problem more challenging than bin-packing where optimization decisions have to be checked periodically. BFD and FFD assume that all bins are initially empty and items can be assigned to any of them. In our context, applying BFD/FFD causes reoptimization from scratch, resulting in a reassignment of all the streams regardless of their previous assignments. This is clearly undesirable, since it has high cost – moving a stream is not a cheap operation as it has delay and buffering penalties on the client side as the experiments demonstrate (Section 6.2). Hence, as a second goal of the optimization, we also need to minimize redirections of streams, *i.e.*, trade off resource optimality against redirection overhead.

Algorithm 1 Input Stream Assignment Strategy Skeleton

```
1: for all split node S do
2:   if S.isOverloaded() then
3:     streamsToMove.add( S.pickStreamsToMove() );
4:   else if S.acceptsNewStream() then
5:     bins.add(S);
6:   end if
7: end for
8: streamsToMove.add( getNewInputStreams() );
9: items ⇐ sort(streamsToMove, PlacementOrder());
10: modifications ⇐ runBFDVariant(items, bins);
11: for all n in modifications.newNodes do
12:   addNewNode(n);
13: end for
14: for all m in modifications.streams do
15:   if m.isNewStream() then
16:     m.assignStream(m.destNode);
17:   else
18:     m.redirectStream(m.srcNode, m.destNode);
19:   end if
20: end for
```

Algorithm 1 shows the skeleton of our optimization strategy that is executed periodically. The algorithm iterates through all the split nodes and moves away streams from the overloaded nodes. The streams to be moved along with the new streams are then connected to nodes that have enough capacity. In case of insufficient capacity, new split nodes are added to the system. The stream assignments to nodes are

carried out using a variant of the BFD bin-packing algorithm with the goal of using as few nodes as possible. However, the concrete strategy is driven by the heuristics used. Different concrete optimization strategies can be created by customizing the implementation of underlined methods whose general tasks are the following:

isOverloaded: Determines whether a node needs consideration for moving some of its input streams because of insufficient capacity.

pickStreamsToMove: On overload, some of the streams must be moved away from the corresponding node. This method identifies the order in which streams will be moved away from the node until the capacity constraint is satisfied.

acceptsNewStream: Determines whether a given node has enough capacity to accept new input streams (called *accepting node*).

PlacementOrder: Identifies the order in which streams that are waiting to be assigned are considered for placement to accepting nodes.

Our framework's concrete optimization strategy is a specialized implementation of the skeleton (*i.e.*, it customizes the underlined methods). The core of our strategy is based on forecasting rates and utilizing meta-data about streams. In the next two sub-sections, we describe these two components and then finally our concrete optimization strategy.

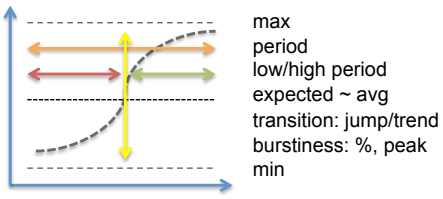


Figure 4: Modelling behavior of input streams.

2.3 Forecasting the Input Rate

Part of our solution for input stream assignments is based on predictions about future behavior of input streams rather than their history. History is only employed to make good quality forecasts about future rates of streams. As forecasts become more accurate, the system is better optimized until a point in future instead of a point which lies in the past.

The forecasting model of our system is as follows. Assume that we are given a series of periodic observations of an input stream's rate, denoted by X_i for each period i. The problem is forecasting, at period t, the rate at period $t+k$, denoted by $\hat{X}_{t+k}$. We use Holt-Winters forecasting technique [5], which is a type of exponential smoothing[1]. We especially use Holt's linear trend model (also known as double exponential smoothing) that works particularly well in practice for reasonable term forecasts. As observations arrive, the value for the local mean level (L) and trend (T) for the input rate is updated continuously as follows:

$$L_t = \alpha \cdot X_t + (1-\alpha)(L_{t-1} + T_{t-1})$$

$$T_t = \gamma \cdot (L_t - L_{t-1}) + (1-\gamma) \cdot T_{t-1}$$

[1]For a survey of time series forecasting, see [4].

Using the updated values of level and trend, the rate forecast at time t, for period $t+k$ is computed as:

$$\hat{X}_{t+k} = L_t + k \cdot T_t$$

The smoothing parameters α and γ can be selected by fitting historical data (we refer the reader to [5] for details). As we shall demonstrate in our experiments, this forecasting technique works quite well for our framework. However, one should note that any sophisticated forecasting technique can be easily plugged into our system, and can be used instead.

2.4 Exploiting External Meta-Data of Streams

In many cases, input data streams often have predictable or known characteristics. Thus, as a first step, our system builds a model for each stream and tries to forecast future rates. This is enabled by continuously monitoring the rate and behavior of each stream. However, beyond that, certain external knowledge can be exploited to improve the modelling and forecasting of streams.

Figure 4 depicts the stream modelling mechanism that we use in our system. Basically, ten essential characteristics of streams are modelled by this mechanism as shown in Figure 4. In the beginning, before a stream is connected to the system, the client can provide information about any of these characteristics as meta-data of the stream. The more information the client provides, the better will be the model in improving the stream assignment optimizations. However, if the client does not give any information about streams, then the model is based on default values of these characteristics that are based on average observations among existing streams. For instance, if the client does not provide any information about the expected or average rate of the stream, then the system assumes a value based on average rates of all existing streams in the system. Among these characteristics, period values give information regarding periodicity of the stream behavior. On the other hand, transition behavior is another important clue about the changes in stream rates. Having information such as "`transition=jump`" means that the stream rate will jump/fall to max/min value. This is very valuable information since it is hard to forecast it by any statistical model. When "`transition=trend`" is given, we can expect that the transition between low and high values will occur with an observable trend, which also greatly improves forecasting accuracy.

2.5 Delta Rate Forecasting with Meta-data

Our concrete optimization strategy for input stream admission and management is called the *Delta Rate Forecasting with Meta-data* **DRF-M**. We employ the rate forecasting-based heuristic (Section 2.3) in our strategy which is enhanced by utilizing stream meta-data. We begin describing it by explaining the concrete implementations of the underlined methods in skeleton strategy shown in Algoritm 1:

isOverloaded: **DRF-M** implementation estimates the latency using an exponentially weighted moving average window of queue size samples, and returns true when this value goes beyond the given threshold, *i.e.*, the QoS metric.

pickStreamsToMove: The main *heuristic* we employ in **DRF-M** is to move the streams with highest expected rate increase, as these are the ones that have the most potential to cause trouble. We call this heuristic *delta-rate-forecast* and it is defined as $\Delta_{RF} = \hat{X}_{t+k} - \hat{X}_{t+1}$. However, if there are meta-data available about the stream, then meta-data

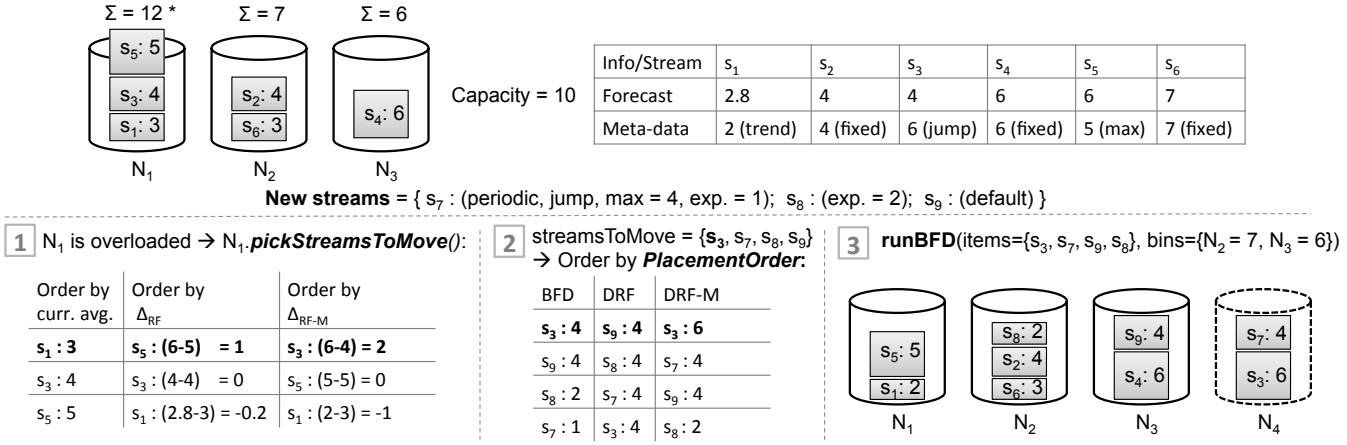


Figure 5: Input stream admission and management in action.

based stream rate expectation is used instead of the forecast. For example, if a meta-data about the maximum achievable rate of an input stream is available, then that value can be plugged into the equation instead of a forecast. Hence, in this case $\Delta_{RF-M} = X^m_{t+k} - \hat{X}_{t+1}$, where X^m_{t+k} denotes the meta-data based stream rate expectation at period $t+k$. We then order all the streams of an overloaded node with their $\Delta_{RF}/\Delta_{RF-M}$ and pick streams to move in this order until total rate forecast falls below node processing capacity (*i.e.*, $\Sigma_{RF} \leq C/\Sigma_{RF-M} \leq C$).

acceptsNewStream: To determine whether a node can accept new input streams, our implementation sums up rate forecasts/expectations of a node's input streams (Σ_{RF}) and returns true if Σ_{RF} is below a certain capacity level.

PlacementOrder: In the BFD algorithm, items to be placed in bins are normally considered in decreasing order of size [6]. In our strategy, we consider streams in order of decreasing rate forecasts or meta-data based rate predictions for the next period, $\hat{X}_{t+1}$ or X^m_{t+k} depending on availability.

Figure 5 illustrates the execution of our optimization strategy in action. For this illustration, we assume 3 split nodes ($N_1..N_3$) containing a total of 6 input streams ($s_1..s_6$) deployed on them. The table on the top right shows the forecasts along with meta-data based predictions for stream rates. The capacity of each node is given in terms of tuples that can be processed per unit time (which is 10) and the rates of streams are given as number of tuples that flow per unit time. As the first step of the algorithm, the load of each split node is computed and N_1 is found to be overloaded ($12 > 10$). The optimization strategy needs to pick streams from N_1 to be assigned to other nodes. Using forecasts and meta-data based predictions from the table, Δ_{RF-M} is computed and s_3 stream is chosen as the victim. Note that if only Δ_{RF} or current average rates were used, the decision would have been different (*i.e.*, 1^{st} and 2^{nd} columns in step 1), and much worse since the chosen streams (as we can see from the meta-data information) are not likely to cause a problem for their nodes, and their redirection is redundant. The second step of the algorithm begins by considering the newly added input streams (cf. line 8 in Algorithm 1) in addition to s_3. This time streams to be moved are ordered according to the *PlacementOrder* heuristic using DRF and meta-data, which is shown on the right-most column in the table. Note the difference in order when using just DRF in the 2^{nd} column (*i.e.*, no meta-data about new streams, hence all assume a default rate of 4) or decreasing order of average rates (plain BFD) in the 1^{st} column. In the third step, the streams to be moved are considered in the determined order from step 2 and the BFD bin-packing algorithm is executed. As a result, a new split node is added (line 12 in Algorithm 1) and the new stream s_7 and stream s_3 are assigned to that node. Other new streams s_8 and s_9 are assigned to nodes N_2 and N_3, and one optimization period of the algorithm completes.

3. ADAPTIVITY IN PROCESSING LAYER

3.1 QoS Model

Our input stream admission and management framework takes QoS requirements for queries as additional input and uses them in determining the degree of parallelization. More specifically, QoS is specified in terms of maximum result latency (L time units) in the output stream.

We define result latency as the time difference between the arrival timestamp of the last tuple contributing to a query result tuple r and the generation of r's timestamp. All result tuples must have result latency of at most L time units.

3.2 Setting Number of Query and Merge Nodes

The modifications of streams (new assignments or redirections) may result in changing the number of split nodes. As the aggregate input volume changes, this in turn requires adjusting the number of processing nodes (*i.e.*, query and merge nodes) fed by the split nodes. Basically, the optimization decision propagates from the input streams layer to the query processing layer.

The QoS latency metric guides parallelization level (d) adjustment. The optimization goal is to keep the latency during period $i+1$ below the target. As a first step, the input stream controller determines the number of split nodes in period $i+1$ (n_s^{i+1}), by considering all streams with their historical, forecasted, and meta-data predicted rates as described in Section 2. Next, the optimizer utilizes total rate expectations from the first step to identify the number of required processing (n_p^{i+1}) and merge (n_m^{i+1}) nodes at period $i+1$. The total rate forecasts of input streams, Σ_{RF} or Σ_{RF-M} and the per-tuple processing cost of the query op-

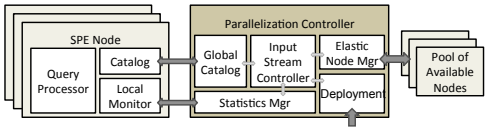


Figure 6: The parallelization controller.

erator (c_{op}) are used to determine the number of processing nodes required for the next period, n_p^{i+1}, using Equation 1a (where H denotes processing headroom of SPE not used for operator execution and C_{op} is the processing capacity of a single node running the query operator).

$$\frac{\Sigma_{RF}}{n_p^{i+1}} \cdot c_{op} = 1.0 - H \tag{1a}$$

$$\frac{n_p^{i+1} \cdot C_{op}}{n_m^{i+1}} \cdot c_{merge} = 1.0 - H \tag{1b}$$

Similarly, the number of merge nodes, n_m^{i+1}, is determined using Equation 1b, assuming that all the processing nodes will run close to saturation in the next period. After determining n_p^{i+1} and n_m^{i+1}, the system adjusts parallelization level of a query by instantiating/de-instantiating replicas of the operators. This is supported by dynamic query modifications at runtime [24].

3.3 Architecture and Implementation

In order to implement the techniques proposed in Sections 2 and 3.2, we designed the Parallelization Controller (PC) as a new component that is responsible for dynamic resource management and continuous query optimization in our framework. Figure 6 shows the general architecture of the PC. Queries and streams are admitted to the system via PC's deployment sub-component. The PC has a statistics manager that periodically collects runtime statistics from active SPE (in our implementation, Borealis [1]) nodes. The global catalog keeps track of all deployed queries with their specifications and locations. The input stream controller is responsible for admission and management of all input streams and continuously optimizing their assignments to split nodes. Lastly, the elastic node manager manages a pool of available nodes and is responsible for handling node add/remove requests by the PC. Next, we will briefly discuss the implementation of stream redirection, dynamic query modifications, and state handover.

Stream Redirection: To be able to dynamically redirect streams to different nodes, our system needs to have control over each external input stream. To achieve this, we attach an interface to each stream with several methods. Essentially, the external input client must implement the interface and support its methods. The interface provides methods to suspend/resume the data flow, to establish/drop a connection and to redirect a stream. Whenever a re-assignment of the stream is required, PC calls necessary methods on the relevant client.

Dynamic Query Modifications: Dynamic query modifications involve adding/removing operator replicas to/from a running query and require suspending certain parts of the pipeline. First, to add a new operator, the running split operator is choked for a short period in order to modify it at runtime. Next, a new output stream is added to the choked split operator. After the modification, the split is resumed without activating the new output stream until the full query modification is complete. In the next step, the new operator is instantiated by retrieving its replica's specification and deployment description from the catalog. The query processor in the new node deploys the operator, its input/output streams and schemas locally. In the third step, the merge operator is choked for a while to add the new input stream. As the last step, merge is resumed and the inactive new output stream of the split operator is activated. Tuples begin flowing on the new stream path from split to merge. Removal of an operator instance follows similar steps.

State Handover: In principle, our parallelization model does not require migrating unprocessed input state during adaptation or load-balancing. Query nodes process all their existing input until the choke point and only ship their partial results to the merge nodes as if they were executing normally, which is then followed by a `stream_end` punctuation to inform the merge operator.

4. INPUT- AND QUERY-AWARE STREAM PARTITIONING

In this section, we describe the integration of stream partitioning techniques and query-awareness into the admission and management of input streams. Stream partitioning and query knowledge is extracted from queries and utilized at the input stream management layer that provides an opportunity to observe and react to dynamic fluctuations in inputs as early as possible. Furthermore, the integration results in robust load-balancing in the processing layer, which is a key requirement for effective parallelization.

There are two main approaches to stream partitioning in the literature: *(i)* content-insensitive, *(ii)* content-sensitive. The former applies to queries with windows and takes only windowing semantics (*i.e.*, size and slide) into account without considering the values that appear in those windows, while the latter applies to queries with key-based processing (*e.g.*, GROUP-BY aggregates) and divides the streams according to the values of those keys.

Given query Q, if it includes key-based processing, then we apply content-sensitive partitioning ("frequency-aware hash-based partitioning"). Otherwise, we apply content-insensitive partitioning (rate-aware pane-based partitioning").

4.1 Frequency-aware Hash-based Partitioning

For queries where evaluation is inherently done on logical partitions identified by keys (*e.g.*, GROUP-BY attributes), typically a hash function is applied over the relevant attributes to identify a processing node for each tuple. This technique can distribute data in a load-balanced manner when the hash function is carefully chosen and the data are uniformly distributed.

In a dynamic setting where there can be a large number of dynamically changing data sources, processing nodes and time-varying fluctuations in the distribution of the data, traditional hashing would result in relocation of keys every time data sources or processing nodes change and would not suffice to achieve load-balancing (cf. Figure 13). What we need, instead, is a hashing technique that not only preserves load-balance in the presence of a data skew, but also minimizes the change in key assignments to processing nodes as data sources or processing nodes join/leave. We, therefore, utilize a consistent-hashing [16] based technique in a novel

manner to balance load among processing nodes. It is important to note that using consistent hashing alone does not completely solve the data skew problem. It provides uniform distribution of keys to nodes, where each node will have the number of keys close to the mean number of keys per node. Under data skew, some of the keys appear more frequently than others over a time interval. In this case, nodes that are assigned the more frequent keys end up receiving more load than the others. In order to deal with the data/frequency skew problem, we propose a revised consistent hashing technique where the most frequent keys are divided into sub-keys in a more fine-grained manner and are assigned to multiple processing nodes. We call this stream partitioning approach *frequency-aware hash-based partitioning*.

Our algorithm proceeds in periods of fixed-size time intervals. During each interval, we maintain the frequencies of the K most frequent keys. We partition each such key i into p_i parts, where p_i is proportional to key i's frequency. As tuples arrive for key i, we suffix them with a partition number from 1 to p_i in a round-robin fashion (*i.e.*, key i becomes $i\#j, 1 \leq j \leq p_i$). This generates p_i distinct keys from each key i, which are treated as if they were separate keys in assigning their corresponding tuples to the processing nodes in the consistent hash table (see Figure 8). It is important to note that this process is carried out on each split node independently so that there is no need for extra communication among the split nodes and hence the mechanism does not need a fully distributed hash table as in peer-to-peer systems. Note that our frequency-aware hash-based partitioning technique is general enough to be applied in other domains that employ consistent hashing (*e.g.*, in distributed key-value stores like Cassandra [18], where temporally skewed accesses are also commonplace).

4.2 Rate-aware Pane-based Partitioning

Our content-insensitive stream partitioning makes novel use of the pane-based technique for parallel processing of sliding window queries. Given a query with window size w and slide s, the main idea is to divide windows into non-overlapping panes of size $gcd(w, s)$ as shown in Figure 7. Pane-based partitioning is a better approach than window-based partitioning when windows overlap. In this case, independent partitions can be created without replicating the overlapping tuples across partitions [3]. In pane-based partitioning, each tuple belongs to exactly one pane.

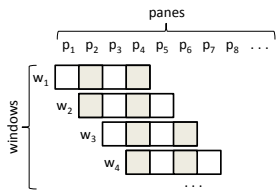


Figure 7: Panes.

In our framework, we assign window-id's and pane-id's to tuples in the split operator (like in [19]), and then distribute panes to the query nodes in a round-robin order. Distributing tuples on a pane-by-pane basis does not work well when input streams are bursty. The reason is that, for time-based windows, bursty or fluctuating input rates may lead to uneven window and pane sizes in terms of number of tuples contained. In this case, we lose control of load-balance over query nodes. This problem is similar to the skew problem in traditional parallel databases, except that, in our case the skew is due to input rates instead of input data distributions. To remedy this problem, we propose a *rate-aware pane-based partitioning* technique. The *rate-awareness* comes from the fact that the system continuously monitors the expected window and pane sizes and their statistical properties by keeping track of "recently" arrived tuples and their rates. In our approach, a pane is further partitioned into sub-panes if its size is larger than the average size of "recently" arrived panes plus the standard deviation. In this case, the tuples exceeding this average are sent to the next query node in the round-robin sequence. On the other hand, for panes whose size is smaller than this average, our algorithm compensates by routing additional panes/sub-panes to the processing node where such small panes are assigned. This solution smoothly integrates into our framework and automatically preserves load-balance of pane-based partitioning. The overhead we pay in return is continuously maintaining an estimate for average pane size and standard deviation over recent stream history, which is relatively small.

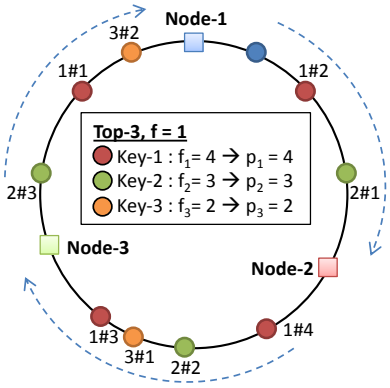


Figure 8: Consistent hashing with further key partitioning.

5. QUERY PROCESSING IN ACTION

In this section, we show how our framework can extend a partitioned parallelism model and be effectively applied to parallelizing sliding window aggregation queries in SPEs.

5.1 Query Types

We assume a generic parallelism model as shown in the middle part of the Figure 1. We focus on Select-Map-Aggregate (SMA) queries as our workload to demonstrate the effectiveness of our techniques, as illustrated by the example query of Section 1.1. This query involves a Selection with predicate `ReportType = 'traffic'`, a Map with a user-defined transformation function `getGridCoordinates(Long,-Lat,@precision)`, and a time-based window with size and slide of 60 and 1 minutes, respectively, over which a distributive (`COUNT()`), an algebraic (`AVG()`), a distributive (`MAX()`) and a user defined aggregate function `isQueueEnd()` are applied. The user-defined Map function transforms given GPS coordinates and precision level to a geodetic coordinate system. The user-defined aggregate detects whether a given road segment has a traffic queue-end [9].

5.2 Pane and Window Meta-data

In our framework, split nodes also take part in query processing. Split nodes annotate tuples with meta-data to in-

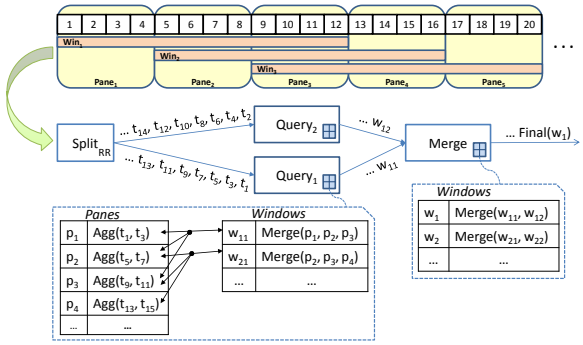


Figure 9: Parallel aggregation pipeline.

form the downstream query, merge, and union nodes about pane- and window-related information that is needed in query evaluation. These include pane-id's and window-id's that are added to the tuples [19], as well as pane-close or window-close punctuations (*i.e.*, additional tuples injected into the streams) signalling the end of a given pane or window.

Consider a query with window size w and window slide s. For count-based windows, a tuple with index i is a window-closer if $((i \geq w) \wedge ((i - w) \bmod s \equiv 0))$. For time-based windows, the same condition can be checked by using i's time field's values. However, when there are multiple tuples with the same time value, the first tuple following a sequence of tuples holding this condition should actually signal a window-close event. A similar discussion applies to pane-close events. Once a tuple is seen that meets the window-close or the pane-close condition, a corresponding punctuation tuple is broadcast to all query nodes. If there are n split nodes, a query node should close a window/pane upon receiving a total of n such punctuations to ensure that all inputs contributing to that window/pane (each might be coming through any one of the n split nodes) are considered.

5.3 Evaluation of Aggregates

Pane-based aggregation decomposes a given query into two sub-queries: *(i)* pane-level sub-query (PLQ) that runs over the panes, and *(ii)* window-level sub-query (WLQ) that runs over the results of the panes. The decomposed evaluation enables parallel processing of a window aggregation as follows. We first partition the input tuples of a window into d partitions. Each of the partitions is processed by a query node that contains a copy of the aggregate operator. Each query node proceeds as if the tuples received for the partition represent the complete window, and generates a partial aggregate for its entire window. This computation benefits from the sharing of pane results and moreover needs reduced internal buffering for distributive and algebraic aggregates. Let us explain this mechanism with an example.

Figure 9 illustrates a parallel aggregation pipeline with 2 query nodes and 1 merge node. The query is a user-defined aggregate with $w = 12$ and $s = 4$ tuples. Therefore, each pane has $w_p = s_p = gcd(12, 4) = 4$ tuples and there are 3 panes per window. For this example, the stream is partitioned with a simple tuple-by-tuple round-robin partitioning. Every query node receives its subset of tuples for corresponding pane partitions (p_i) as a stream from the split operator and is responsible for locally computing the PLQ's for those. PLQ results, (p_i, value) pairs, are stored in a `Panes` table. On the other hand, a `Windows` table keeps track of window results as (w_{ij}, value) pairs that are evaluated from the PLQ results by applying the merge function, i denoting the window-id and j denoting the index of the node. A Panes table entry has a complete result after all the `pane_close` punctuations arrive. A Windows table entry becomes ready for output when all of its n panes have complete results in the corresponding Panes table. In the example, partial window results w_{11} and w_{12} will be output to the merge node from query nodes 1 and 2, respectively. After all query nodes send a `win_close` punctuation to the merge node, the corresponding Windows table entry becomes ready for final output.

6. EXPERIMENTAL EVALUATION

The goal of this experimental study is to investigate how well our framework achieves its goals. The results demonstrate that: *(i)* the number of processing nodes that our system uses is close to optimal, *(ii)* load is balanced at all times with low overhead, and *(iii)* the adaptivity layer meets QoS bounds.

6.1 Experimental Setup

We implemented our adaptive input stream admission and management techniques as an extension to the Borealis distributed stream processing engine [1]. Additionally, we enriched the query execution framework of Borealis by integrating our techniques for a data partitioned evaluation of SMA queries.

All the experiments were conducted on a shared-nothing cluster of machines, where each machine has an Intel® Xeon® L5520 2.26 Ghz Quadcore CPU and 16GB of main memory. The nodes, each running Debian Linux, are connected by a Gigabit Ethernet.

6.1.1 Workloads

To evaluate our system, we used the query from the motivating example introduced in Section 1.1. The query is a fairly expensive SMA query, which includes a costly mapping function (`getGridCoordinates()`) doing floating point intensive transformation on geographic coordinates and it evaluates four aggregations, one of them being fairly complex (`isQueueEnd()`). Lastly, in order to have a representative number of panes, we fixed the ratio of window size to slide at 100, where actual sizes are varied based on the chosen input rate.

The workload data that we used for our experiments is adapted from the real world traces of Uber [26] that is publicly available [13]. The data is a sample of $1.2M$ position reports collected from black cars in San Francisco. However, it misses some of the attributes such as `Speed`, `ReportType` given in the schema described in Section 1.1 and hence we artificially generated these values (assuming 100% selectivity for the predicate). In order to increase input rates, we replayed the real trace in a faster way. As a result, the data we used in the experiments follow the real trace in terms of skew with the only difference being the input rates.

Additionally, we also synthetically generated different temporal patterns based on the distributions of the real dataset. The constant workload (**constw**) consists of k streams with uniformly distributed rates between a min and a max rate. Tuples arrive with exponentially distributed inter arrival times with the mean equal to the chosen average rate. Pair-

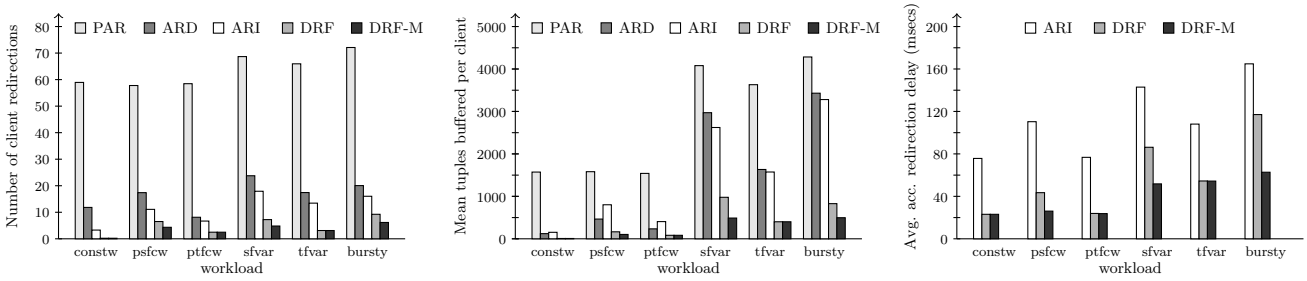


Figure 10: (a) Mean number of client redirections, (b) mean number of buffered tuples, and (c) redirection delay at client side.

step (**psfcw**) and pair-trend (**ptfcw**) fluctuating constant workloads consist of k streams that fluctuate in pairs but opposite of each other while the total load stays constant. We created two workloads where the total load varies. In the step fluctuating version (**sfvar**), rates suddenly change between high and low, whereas in trend fluctuating (**tfvar**) the switch happens with an observable trend. Lastly, for the bursty workload, we used the classical on-off model. During an active period, tuples arrive periodically with a certain rate, whereas in an idle period no data arrives. Duration of active and idle periods follow an exponential distribution. As the total number of streams, we have picked 80. Finally, we define the **load level** as the number of machines that can handle the entire load at hand.

6.1.2 Optimization Strategy Variations

The main optimization strategy of our framework, **DRF-M**, is described in Section 2.5. For comparison, we implemented other strategies by customizing the optimization algorithm shown in Algorithm 1. The baseline algorithm is called *Plain Average Rate BFD* (**PAR**), which simply considers all streams as moveable and all nodes as initially empty. Moreover, it uses only last period average rates instead of forecasts. The other algorithms we consider differ from our strategy in the way they pick streams. *Average Rate Increasing BFD* (**ARI**) picks streams in increasing order of their last period average rates, whereas *Average Rate Decreasing BFD* (**ARD**) picks them in decreasing order. Both rely on last period average rates instead of rate forecasts. Lastly, in order to demonstrate the effectiveness of the meta-data information about streams, we have also implemented **DRF** which relies on just *delta-rate-forecast* heuristic without considering meta-data about the streams.

6.2 Performance of Input Stream Admission and Management

In this experiment, we demonstrate the effectiveness of our input stream admission and management technique **DRF-M**. We discuss the performance in terms of the number of client redirections, the buffering amount at the client, and the delay contribution in the overall query evaluation.

6.2.1 Client Redirections and Buffering

Figure 10(a) shows mean client redirections and Figure 10(b) shows mean client buffering caused by different strategies under different workloads. First, PAR considers reassignment of all streams at each period regardless of the workload. As a result, it shuffles around most of the streams (note that the total number is 80) all the time causing high amount of client side buffering. Second, the DRF strategy moves streams early by utilizing the forecasts and prevents the problems in advance. This reduces stream movement, as initial movements are provisioned for future rates. As a result, client side buffering is also reduced. The benefit of forecasting is apparent in comparing *sfvar* and *tfvar* workloads where forecasts are more accurate in *tfvar*. Comparing ARI and ARD, we see that ARI performs slightly worse than ARD as it first considers moving low rate streams, which dictates it to move more streams around. Compared to all other techniques, DRF is much better in reducing the number of client redirections and the amount of buffering at the client side. Furthermore, DRF-M, which integrates meta-data knowledge about streams, performs even better. In this experiment, we provided information about peak-rates, trend of rates and burstiness period. In cases where predictions do not perform sufficiently well (*i.e.*, *psfcw*, *sfvar* and *bursty*), DRF-M further improves the performance compared to DRF.

6.2.2 Delay Contribution

Figure 10(c) shows average accumulated delays caused by stream redirections per optimization period. We only include ARI, DRF and DRF-M as others have extremely high delays. The delays are in line with the other experiments, fewer number of redirections and buffering in DRF and DRF-M keep the accumulated delay lower than others. Additionally, DRF-M becomes more robust by integrating meta-data even with a bursty workload.

6.3 Resource Efficiency

In this experiment, we investigate how different strategies perform in terms of resource usage compared to the optimal case that requires solving an instance of the bin-packing problem at each optimization period. To find the optimal number of nodes (*i.e.* bins), we used an approximate method called *wasted-space residual optimality* [17] that provides a tight bound for optimal number of bins required.

Figure 11 shows the result of this experiment. PAR performs very close to the optimal. The reason is that it applies a global optimization in stream assignment by considering all streams at once and does not consider the previous assignments of streams. However, despite being close to optimal in terms of resource efficiency, we have shown in Section 6.2 that global optimization in PAR performs unacceptably poor for admission and management of input streams. Second, rate forecasting can be negative in terms of resource usage as forecasts often tend to overestimate the input rates. DRF can perform similar to the non-forecasting versions

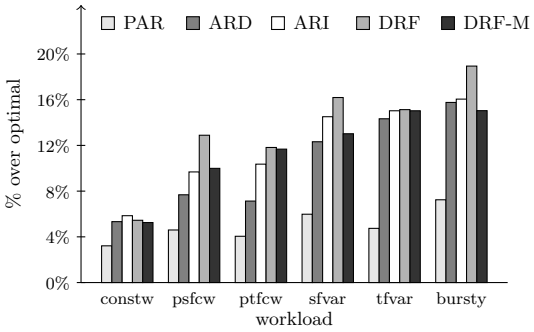


Figure 11: % resource utilization above optimal.

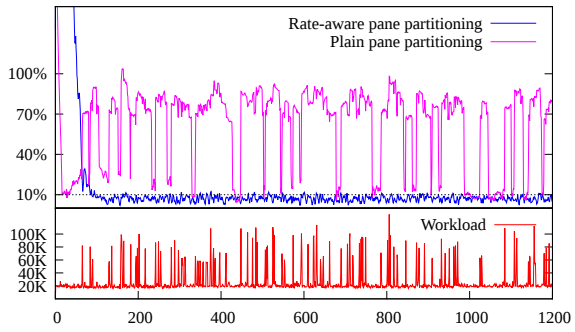


Figure 12: Load-balancing in content-insensitive partitioning.

(ARD, ARI) when forecasting is more accurate, for instance in workloads with an explicitly observable trend (*ptfcw*, *tfvar*). Overall, DRF only needs around 15% more resources than optimal to handle a fluctuating workload with trend. More importantly, our main optimization strategy DRF-M performs even closer to the optimal for *psfcw*, *sfvar* and *bursty* workloads where using meta-data proves better than just forecasts.

To summarize the results of Sections 6.2 and 6.3, the DRF-M strategy is very successful in keeping the delay below a certain threshold by avoiding excessive stream reassignments in a running query, while it requires only ≈5-10% more resources than other strategies and ≈15% more than the optimal.

6.4 Impact of Query-awareness in Stream Management for Parallelization

In this section, we evaluate the integration of stream partitioning techniques and query-awareness into the admission and management of input streams. The experiments in this section demonstrate the effectiveness of these techniques in terms of robust load-balancing and low overhead, which are key requirements for effective parallelization.

6.4.1 Content-insensitive Partitioning

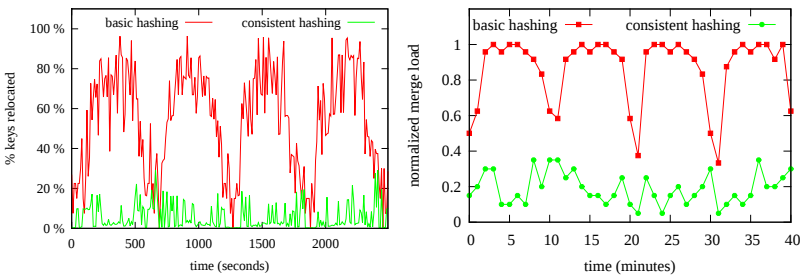
Content-insensitive partitioning utilizes window size and slide information from the query and introduces use of panes as discussed in Section 4.2. In this experiment, we evaluate our pane-based partitioning techniques. We use a single splitter node with either plain or rate-aware pane partitioning to compare them. The metric used in the experiment is the **load-balancing ratio** defined as the ratio of mean and standard deviation of CPU loads observed over 16 nodes that split feeds. As the workload, which is specific to this experiment, we use an input stream with a rate uniformly distributed between 18K and 22K for every second. To model the rate-skew, the input rate transiently jumps to 5 times the current rate with a probability of 0.1 at each second. Figure 12 shows the workload at the bottom and the results at the top. First, in both cases the load-balancing ratio is high at the beginning as nodes are idle and most of them do not have enough data to process. However, this period is longer in rate-aware partitioning as it needs a warm-up period to statistically estimate the number of tuples in a pane. After that, rate-aware pane partitioning continuously keeps track of pane-size statistics and dynamically further partitions a pane if there is need. As a result, the load-balancing ratio is slightly affected by the rate-skew problem and, after the warm-up, the system is able to keep the average load-balancing ratio **below 7 %** during the entire run. However, plain pane partitioning is highly sensitive to rate-skew problem as seen in Figure 12. Whenever the rate jumps, the load-balancing ratio also jumps up to 100 % causing a high load-imbalance. During the entire run, the average load-balancing ratio is ≈60 %.

6.4.2 Content-sensitive Partitioning

Content-sensitive partitioning utilizes key-based constructs from the query (*i.e.*, `GROUP-BY` statement in our case) as discussed in Section 4.1. In this experiment, we study the load-balancing performance of our key-based partitioning technique under varying skew. In each experiment, after a warm-up period for collecting key frequency statistics, we measure the mean and the standard deviation of load on the processing nodes. In all experiments, the maximum number of distinct keys is 16K. In the first experiment, we used the real data distributions from the motivating example. In the other ones, we vary the frequency of keys at each experiment by using a Zipf distribution with different parameters. Figure 15 shows the load-balancing property of our key-based partitioning technique on both real data trace and synthetic data with different Zipf and top-K parameters. K on x-axis indicates the number of the most frequent keys that are considered for further partitioning. The percent shown on the y-axis denotes the load-balancing ratio. First of all, as shown previously in Figure 2(a), the real data trace follows a highly skewed Zipfian-like distribution and, not surprisingly, the load-balancing performance of our techniques work equally well on real trace and synthetic data generated with a Zipf distribution. Second, as the skew increases, variation of load between different processing nodes increases significantly if we do not apply further partitioning (*i.e.*, $K=0$). Since there are a few hotspot keys, choosing $K=5$ reduces the ratio below 50 % in all the cases. When K increases, *e.g.*, at $K=100$, the variation of load decreases tremendously, almost nearing the keys per node ratio. However, increasing K beyond a certain point does not help as the load-balancing ratio is determined by the keys per node ratio. As a conclusion, by only monitoring and further partitioning the 100-200 most frequent keys we can achieve robust load-balancing even under extreme skew.

6.4.3 Why basic hashing would not suffice?

One might tend to think that basic hash-partitioning may suffice for adaptive parallelization. Assume a hash function



(a) Impact on key relocations (b) Problem in aggregation

Figure 13: Problem with basic hashing.

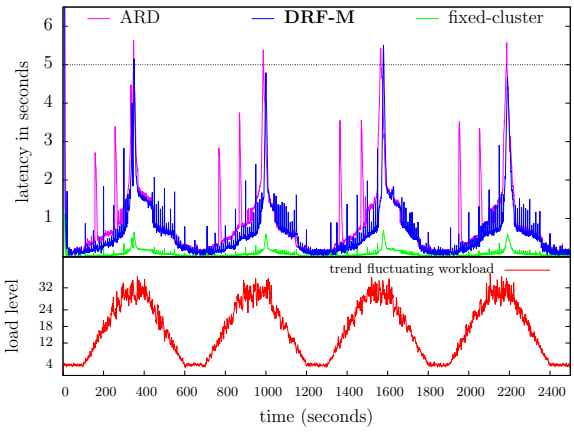


Figure 14: Adaptivity vs. Latency.

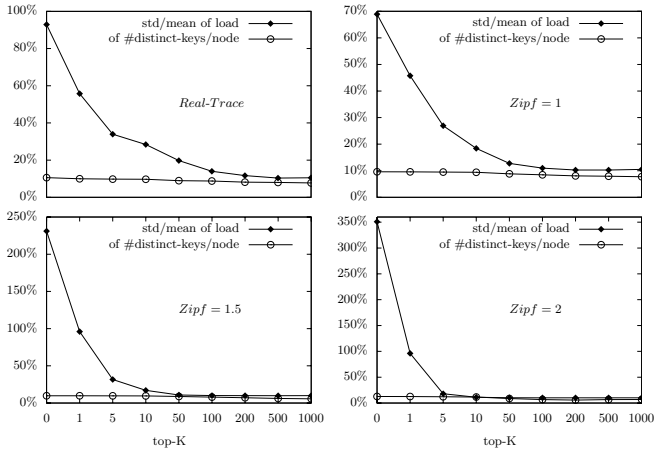


Figure 15: Load-balancing in content-sensitive partitioning under skew.

$h(k) = hash(k)\%N$ where N denotes the number of nodes. We use the same workload as shown in the lower part of Figure 14, where the number of processing nodes changes significantly over time. As N changes at each optimization point, the value of the hash function changes frequently as well. We call each such change a ***key relocation***, which causes tuples of a pane to be distributed to excessively many nodes. In an extreme case, no significant aggregation might be done on processing nodes. Even a single pane result must be produced by merging many sub-pane results from several nodes, which in turn increases the merge cost. Figure 13 shows the experiments we conducted to study this effect. Figure 13(a) shows the percentage of all keys that are relocated. In basic hashing, almost all of the keys are relocated as the number of nodes changes. Our technique keeps the number of relocated keys significantly lower. As a result, our technique achieves a lower load level on merge nodes as shown in Figure 13(b).

6.4.4 *Load-balancing Overhead*

Frequency-aware partitioning needs to keep track of the most frequent K keys for further partitioning. At the implementation level, we use an approximate frequency estimation algorithm on a sample of the stream. In the split node, average per tuple processing cost overhead in comparison to basic hashing is $\approx$ **17 %**. On the other hand, pane-based partitioning with rate-awareness needs to keep track of pane size statistics. In this case, per tuple processing cost overhead compared to plain round-robin is only $\approx$ **4 %**. Overall, the load is balanced at all times for a small overhead.

6.5 Adaptivity and QoS Preservation

In this experiment, we study adaptivity, in particular the end-to-end latency behavior when our optimization strategy (DRF-M) is used. For comparison, we use ARD since it is one of the best. The workload used in this experiment is a fluctuating workload with a trend. It begins with a workload saturating 4 machines and goes up to 32. Inspired by the Linear Road Benchmark [2], the QoS metric chosen has a maximum latency of 5 seconds. Results are shown in Figure 14. At the beginning, all the generated streams are assigned randomly, which causes a spike. Latency drops later until load begins to increase. The major difference between the two algorithms is that DRF-M looks at the future (by forecasts), while ARD looks at the past (by average values). Using forecasts, DRF-M optimizes early and keeps the latency low. The early optimizations cause transient peaks but reduce latency later. ARD performs optimizations even earlier than DRF-M, which causes latency to be higher and peaks to be lengthy. When the load-level reaches the max, both algorithms need to shuffle around many streams to achieve optimal assignment. During that time, streams are very fast and this causes a lot of buffering on the clients and nodes resulting in short QoS violations. Finally, we also include the latency measurements with a static cluster of 32 nodes in which nodes are never dynamically added/removed to/from the query. As expected, it has a lower latency but comes with extremely high resource usage. For example, it uses 32 nodes at times when only 3 nodes would suffice for the workload at hand.

6.6 Summary

The experiments show that our system can meet the QoS requirements of fluctuating workloads of highly dynamic and transient input streams while being resource efficient. Integration of input meta-data and query-awareness into the input management enabled our partitioning techniques to achieve robust balancing of load for skewed data and rates with low overhead for maximum utilization of parallelism.

7. RELATED WORK

The advances in parallel processing platforms and the emergence of the pay-as-you-go economic model of the new cloud-based infrastructures motivate the need for elastic scal-

ability in stream processing systems. In order to address this requirement, Recent works introduce capabilities into SPEs to allow flexible scaling up and down of the processing capacity in response to the workload fluctuations [10, 11, 21].

In general, the elastic scalability feature is tightly coupled with the parallelization model. Pipelined parallelism is one of the common models to provide inter-operator/inter-query parallelism (*e.g.*, [29]). In this case, load-balancing/adaptivity requires moving operators and state across different nodes. In order to provide elastic parallelism during runtime, Gulisano et al. [10, 11] use a similar model to pipelining by proposing a technique to split queries into subqueries for allocating them to independent sets of nodes.

The partitioned parallelism model, by contrast, splits input streams into disjoint partitions, each of which is processed by a replica of the query in a parallel fashion. Partitioning usually provides fine-granular intra-operator parallelism and achieves much better load-balancing. In order to provide elastic parallelism in an SPE, Schneider et al. [21] use a partitioned parallelism model that proposes methods for streaming operator elasticity on multi-core CPUs. In one of the earlier works in this area, Flux generalizes the Exchange and RiverDQ approaches of traditional parallel databases to provide online repartitioning of streaming operators such as group-by aggregates [23]. Ivanova et al. [14] instead focus on data partitioning for content-insensitive streaming operators such as windowed aggregates. Very recently, to address the unscalability of partitioning in this work, Zeitler et al. [30] propose a parallelized stream splitting operator (*parasplit*) for massive-volume streams. In addition, once data parallelism is identified, extracting it from queries automatically with a compiler and runtime system constitutes an important step [22].

In terms of admission control, a recent work discusses an auction-based "query admission" to a cloud-resident SPE to increase the system's economic utilization [20]. However, this work mainly focuses on the admission of different user's queries and does not discuss any issues related to the admission control and management of input streams. Another closely related work discusses resource management and admission control for SPEs [28]. By treating the input streams as dummy nodes, the problem of admission control is transformed into a routing and resource-allocation problem with the objective of maximizing the overall system utility. However, the assumptions of this work are rather restrictive, including a fixed-size cluster without elasticity, fixed-rate streams, and no consideration of input stream characteristics. Lastly, Gulisano et al. [11] provide on-demand provisioning for computing resources based on past observations of node loads. However, they also do not consider multiple input streams and their characteristics in their problem.

8. CONCLUSIONS

In this paper, we presented a framework for adaptive admission control and management of input streams for parallel data stream processing. The main goal of this framework is to treat large numbers of dynamic external input streams as first-class citizens of a stream processing engine. By explicitly controlling their admission and managing them, SPEs can better and timely react to dynamically changing workloads. Our main contributions include a near-optimal input stream assignment technique that employs forecasting and meta-data, stream partitioning and adaptivity techniques that employ query knowledge early in the input layer for automatically minimizing the size of the cluster and maintaining load balance. Our results show that these techniques are effective in achieving the goals of our framework.

9. REFERENCES

[1] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB*, 2004.

[3] C. Balkesen and N. Tatbul. Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams. In *DMSN*, 2011.

[4] P. J. Brockwell and R. A. Davis. *Time Series: Theory and Methods*. Springer-Verlag, 1991.

[5] C. Chatfield and M. Yar. Holt-Winters Forecasting: Some Practical Issues. *Journal of the Royal Statistical Society*, 37(2), 1988.

[6] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation Algorithms for Bin Packing: A Survey*. 1997.

[7] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6), 1992.

[8] Gartner. Forecast: Mobile Communications Devices by Open Operating System, Worldwide, 2008-2015.

[9] S. Geisler et al. A Data Stream-based Evaluation Framework for Traffic Information Systems. In *GIS-IWGS*, 2010.

[10] V. Gulisano et al. StreamCloud: A Large Scale Data Streaming System. In *ICDCS*, 2010.

[11] V. Gulisano et al. StreamCloud: An Elastic and Scalable Data Streaming System. *TPDS*, 2012.

[12] J. C. Herrera, , et al. Evaluation of Traffic Data Obtained via GPS-enabled Mobile Phones: The Mobile Century Field Experiment. *Transportation Research Part C: Emerging Technologies*, 18(4), 2010.

[13] Infochimps. http://www.infochimps.com/datasets/uber-anonymized-gps-logs/.

[14] M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. In *VLDB*, 2005.

[15] T. Johnson et al. Query-aware Partitioning for Monitoring Massive Network Data Streams. In *SIGMOD*, 2008.

[16] D. Karger et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.

[17] R. E. Korf. A New Algorithm for Optimal Bin Packing. In *AAAI*, 2002.

[18] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *OSR*, 44(2), 2010.

[19] J. Li, D. Maier, et al. Semantics and Evaluation Techniques for Window Aggregates in Data streams. In *SIGMOD*, 2005.

[20] L. A. Moakar, P. K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs. Admission control mechanisms for continuous queries in the cloud. In *ICDE*, 2010.

[21] S. Schneider et al. Elastic Scaling of Data Parallel Operators in Stream Processing. In *IPDPS*, 2009.

[22] S. Schneider et al. Auto-parallelizing stateful distributed streaming applications. In *PACT*, 2012.

[23] M. A. Shah. *Flux: A Mechanism for Building Robust, Scalable Dataflows*. PhD thesis, U.C. Berkeley, 2004.

[24] K. Sheykh-Esmaili et al. Changing Flights in Mid-air: A Model for Safely Modifying Continuous Queries. In *SIGMOD*, 2011.

[25] A. Silberstein et al. Feeding Frenzy: Selectively Materializing Users' Event Feeds. In *SIGMOD*, 2010.

[26] Uber. http://www.uber.com/.

[27] Waze. http://www.waze.com/.

[28] C. H. Xia, D. Towsley, and C. Zhang. Distributed resource management and admission control of stream processing systems with max utility. In *ICDCS*, 2007.

[29] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *ICDE*, 2005.

[30] E. Zeitler and T. Risch. Massive Scale-out of Expensive Continuous Queries. In *VLDB*, 2011.