Parallel Continuous Preference Queries over Out-of-Order and Bursty Data Streams

Gabriele Mencagli, Massimo Torquati, Marco Danelutto, and Tiziano De Matteis

Abstract—Techniques to handle traffic bursts and out-of-order arrivals are of paramount importance to provide real-time sensor data analytics in domains like traffic surveillance, transportation management, healthcare and security applications. In these systems the amount of raw data coming from sensors must be analyzed by continuous queries that extract value-added information used to make informed decisions in real-time. To perform this task with timing constraints, parallelism must be exploited in the query execution in order to enable the real-time processing on parallel architectures. In this paper we focus on *continuous preference queries*, a representative class of continuous queries for decision making, and we propose a parallel query model targeting the efficient processing over out-of-order and bursty data streams. We study how to integrate *punctuation mechanisms* in order to enable *out-of-order processing*. Then, we present advanced scheduling strategies targeting scenarios with different burstiness levels, parameterized using the *index of dispersion* quantity. Extensive experiments have been performed using synthetic datasets and real-world data streams obtained from an existing real-time locating system. The experimental evaluation demonstrates the efficiency of our parallel solution and its effectiveness in handling the out-of-orderness degrees and burstiness levels of real-world applications.

Index Terms—Parallelism, Multicores, Data Streams, Continuous Preference Queries, Sliding Windows, Out-of-order Arrivals, Burstiness and Traffic Surges.

1 Introduction

Preference queries have received considerable attention over the last years for their increasing interest in decision making processes based on real-time data (e.g., intrusion detection systems and financial applications). They are executed for information filtering and extraction, in order to reduce the volume of data used for successive phases that combine various interests to make strategic decisions [1]. Their real-time execution represents a hot research topic [2], since such queries are usually time consuming and need expensive pairwise tuple-to-tuple comparisons (a *tuple* is a structured data item consisting in a record of attributes).

With the emergence of data streams, preference queries are continuously processed over unbounded sequences of transient data received at high velocity. A common approach is to process the query under a *sliding window* model [3], where the query is frequently re-evaluated over the most recent tuples received in the last time interval (e.g., expressed in seconds or milliseconds) [3], [4]. This periodic re-evaluation further increases the computational requirements and makes the use of *parallel processing* techniques on today's multicores a compelling choice to run the queries in real-time efficiently and in a scalable manner [5], [6].

However, the performance of parallel continuous queries may be hampered by the workload variations and the temporal properties of real-world data streams. One is the presence of *bursts* [7], i.e. sharp rises in the traffic volume, for example generated when some sensors detect critical events and respond with a temporarily increase in the sampling rate [7]. The parallel processing of temporal windows over bursty streams may suffer from an uneven load distribution, as the *extents* of different windows (i.e. the set of tuples whose timestamp is within the window temporal interval) may have a substantially different cardinality. The problem is exacerbated in case some tuples arrive at a data receiver *out-of-order*, i.e. not ordered according to their timestamps. Possible causes include the multiplexing of several physical streams or the use of unreliable communication protocols [8], [9]. In those cases the way the system detects when all the tuples of a window have likely been received is critical to provide live results with tolerable latency.

An approach to deal with workload variations, such as an increase or a decrease in the arrival rate, is to use *elastic* runtimes [5], [6], [10] able to scale up/down the resources (nodes/cores) hosting the query execution. Such resource reconfigurations may disturb the flow of data by producing latency spikes and throughput drops [6], [11]. Consequently, they must be triggered when some deterministic trends in the workload are observed or predicted at slow time-scales (e.g., minutes or tens of seconds) [5], [10]. Different instead is the nature of bursty workloads, where micro-congestion episodes happen at very fast time-scales (e.g., few milliseconds). As stated in Ref. [12], bursts may produce a detrimental effect in the query performance and may lead to ineffective elastic responses. Burstiness is a important feature to be addressed in next-generation stream processing frameworks, however it is often overlooked and the design of burst-tolerant strategies represents a still open challenge.

Two opposite models have been used to deal with out-of-order data streams [8], [9], [13], [14]: *i) in-order processing model* (IOP), where tuples are first buffered and then presented to the query in increasing order of their timestamps; *ii) out-of-order processing model* (OOP), where tuples are immediately forwarded to the query and processed in the arrival order. As shown in Ref. [14], the OOP model

• *G. Mencagli (first author), M. Torquati, M. Danelutto, and T. De Matteis are with the Department of Computer Science, University of Pisa, Italy, E-mail: {mencagli, torquati, danelutto, dematteis}@di.unipi.it.*

is capable of reducing the query latency and memory usage provided that: *a)* the query semantics allows the results to be incrementally computed by processing tuples in any order; *b)* some additional operator in the query plan must be able to recognize when all the tuples of a window have likely been received and the query can be re-evaluated. While solutions for this problem have been studied for simple queries like sliding-window aggregates [8], [13], no previous work copes with the problem of running preference queries in parallel environments and over out-of-order data streams.

In this paper, for the first time, we study a parallel model for continuous preference queries which is able to handle both bursty transmissions and out-of-order arrivals in data streams. We study a parallelization of the *pane-based model* presented in Ref. [15]. We show that this approach, originally thought to be sequential and designed for sliding-window aggregates, can be profitably applied to a set of preference queries that can be executed *"à la" map-reduce*, i.e. by separating the set of tuples of each window in different partitions called *panes*, and by merging the results of the panes to obtain the final results of the windows. The parallelization consists of two parallel stages working in pipeline. The first executes the query on the panes, the second computes the results of the windows. Both stages incorporate sophisticated scheduling strategies to cope with bursty data streams, and stream progress mechanisms to allow tuples to be processed out-of-order.

1.1 Contributions

The general contribution of this paper is to fill some existing gaps in the literature. From one side the impact of burstiness in continuous query processing has not been studied quantitatively, and countermeasures represent a novelty *per se*. On the other side, techniques for out-of-order data streams have a long history, however their evaluation in multicore parallel implementations is still missing. This paper covers these aspects by targeting preference queries, a class of computationally demanding queries. The specific contributions can be summarized as follows:

- ***Burst-tolerant scheduling strategies***: panes can be split to improve load balancing in presence of bursts. For preference queries, splitting too much the panes increases the number of tuples transmitted to the second stage. To solve this issue our scheduler dynamically applies the minimum splitting needed to successfully handle the current burstiness level. To this end, a proportional-integrative-derivative controller (PID) adjusts the splitting threshold autonomically. In the second stage, we design a scheduling strategy that neutralizes the impact of bursts as much as possible. The strategy uses feedbacks from the parallel workers and schedules tasks in order to optimize load balancing while respecting the computation semantics.
- ***Use of a parameterized burstiness model***: in order to have a systematic way to evaluate the capability to handle bursty periods with different intensities and duration, we adapt an existing approach [16] to inject burstiness at fine time-scales in the stream. The approach relies on a single parameter, the *index of dispersion*, and has been applied in the past for stress tests of client-server systems. Its application in the data stream processing domain is completely novel.
- ***Modularity of disordering handling:*** we localize the disordering handling mechanisms in the first stage. We show that the disordering handling logic is completely orthogonal to the scheduling strategies used to handle bursty periods, and can be configured to further decrease latency once the degree of parallelism is sufficient to avoid the query being a bottleneck.

An implementation is provided within the FastFlow[1] framework [17] for shared-memory architectures. The experiments, on synthetic and real-world datasets, show that our approach accommodates high levels of burstiness in an effective way while achieving satisfactory processing bandwidth and latency. Furthermore, we show that the parallel implementation tolerates high degrees of out-of-orderness and the latency reduction justifies the use of the OOP model.

The outline of this paper is the following. Sect. 2 provides background concepts. Sect. 3 shows the parallel query model. Sects. 4 and 5 describe the implementation strategies evaluated under synthetic datasets. Sect. 6 shows a final evaluation using real-world datasets. Finally, Sect. 7 describes related works and Sect. 8 provides the conclusions.

2 Background

In this section we describe the background of preference queries and the pane-based model [15]. Then, we identify the reasons that make this model suitable for supporting parallel and out-of-order processing.

2.1 Continuous Preference Queries

Let $\mathcal{D}$ be a d-dimensional sub-space of $\mathbb{R}^d$ with $d \geq 1$. A preference query $\mathcal{Q}$ finds the subset $\mathcal{S} \subseteq \mathcal{D}$ of the most relevant tuples according to some criterion. Some queries guarantee a fixed-cardinality output set, like the top-k query [18] that returns exactly the $k \geq 1$ most interesting tuples by using a user-defined scoring function. In contrast, the skyline query [19] uses an implicit criterion called Pareto dominance, i.e a tuple t dominates a tuples t' if it is better or equal in all the dimensions and better in at least one dimension. In case of high-dimensional datasets, this relation may produce large skylines (i.e. the output set of non-comparable tuples) because as d increases, for any tuple t it is more likely that exists another tuple t' where (t, t') are better than each other over different subsets of dimensions [20].

A recent effort has been made to define queries providing a fixed-size output set without using user-defined scoring functions that are in general difficult to be defined effectively. An example is the *top-δ dominant* query [20], in which a relaxation of the Pareto dominance called *k-dominance* is used, i.e. a tuple t in the k-dominant skyline is a tuple for which there does not exist any other tuple t' better or equal to t in $k \leq d$ dimensions and better in at least one of these dimensions. Accordingly, the size of the k-dominant skyline decreases with smaller values of k. The aim of the top-δ dominant query is to find the smallest $k \leq d$ such that there are more than $\delta \geq 1$ k-dominant skyline tuples.

1. FastFlow: http://mc-fastflow.sourceforge.net. The implementation source code can be found at https://github.com/ParaGroup/BT-PPQ.

Continuous preference queries over data streams are executed according to a *sliding window* model [4], where the query is evaluated over the most recent tuples. *Time-based windows* are most commonly used in stream processing applications [4]. It is possible to distinguish between:

- *time-based sliding windows* with a *temporal* activation mode: each window has a *length* of w time units and a new window is triggered every s time units (*slide*). For example, the query is evaluated over the tuples received in the last five seconds ($w = 5$) by producing a new result every one second ($s = 1$);
- *time-based sliding windows* with a *slide-by-tuple* activation mode: each window covers a timespan of w time units from the last received tuple, i.e. the window slides every new tuple ($s = 1$ tuple).

The first model is supported in most of the modern stream processing frameworks (e.g., Apache Storm [21]) and produces an output set $\mathcal{S}_i$ for each window $\mathcal{W}_i$. The second, applied to some continuous preference queries in the literature [22], [23], produces a sequence of incremental updates of a unique output set $\mathcal{S}$. In this work we will adopt the first model for the following reasons:

- in case of very fast input rates producing a result update every tuple is not realistic [13], [15]. Instead, the user may prefer to receive results at a regular frequency independently of the temporal characteristics of the stream like the presence of traffic bursts or lulls;
- the slide-by-tuple semantics may be difficult to be applied in out-of-order streams, as the arrival of a new tuple does not always imply that the maximum timestamp seen so far increases. Instead, with a temporal activation mode the user specifies how far apart are consecutive windows in terms of time units, independently of the arrival order of the tuples.

In the next part we will introduce a query processing model for sliding windows with temporal activation mode [15]. This approach will be the basis of our work.

2.2 The Pane-based Model

The *pane-based model* has been introduced for sliding-window aggregates [15], but the idea can be extended for a representative set of preference queries. The approach divides each window into a set of disjoint (tumbling) sub-windows called *panes*, computes the result of a sub-query on each pane, and merges the results of the panes to determine the result over each window, as depicted in Fig. 1.

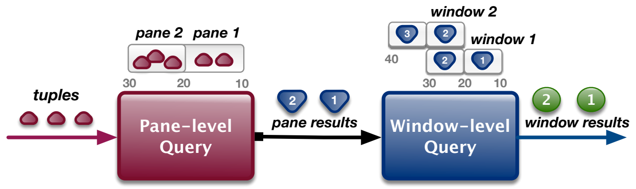


Fig. 1: Pane-based approach to sliding-window queries.

Let w and s be the window length and slide expressed in time units, $\mathcal{D}$ the set of all the tuples of the input stream and $t.ts$ the timestamp attribute (starting from zero) of a tuple t. Each pane is a tumbling window of length $L_p = GCD(w, s)$ time units [15], e.g., with windows of 5 seconds and slide of 2 seconds each window is partitioned into five panes each of length 1 second. The pane with identifier $i = 1, 2, \ldots,$ contains the set of tuples $\mathcal{P}_i \subseteq \mathcal{D}$ defined as follows:

$$\mathcal{P}_i = \Big\{ t \mid t \in \mathcal{D},\, t.ts \in \big[(i-1) \cdot L_p, i \cdot L_p\big) \Big\} \qquad (1)$$

Each tuple $t \in \mathcal{D}$ coming to the query operator belongs to exactly one pane, i.e. the one with identifier $i = \lceil t.ts/L_p \rceil$.

To apply the pane-based model the continuous preference query must be decomposable by the user into two sub-queries $\mathcal{Q} = \{\mathcal{G}, \mathcal{H}\}$. The first is the *pane-level query* (PLQ), which processes the panes and produces a result $\mathcal{R}_i = \mathcal{G}(\mathcal{P}_i)$ for each pane. The result of a pane is the subset of the pane tuples selected according to the used preference relation, i.e. $\mathcal{R}_i \subseteq \mathcal{P}_i$. The second sub-query is called *window-level query* (WLQ), and runs over all the results of the panes and computes the final results of the windows. Let $w_p = w/L_p$ and $s_p = s/L_p$ be the number of panes per window and slide respectively, and $\mathcal{W}_i$ the window with identifier $i = 1, 2, \ldots$. We define $\mathcal{Z}_i$ the set of the identifiers of the panes belonging to the i-th window:

$$\mathcal{Z}_i = \Big\{ j \mid j \in \mathbb{N},\ \big[(i-1)s_p\big] < j \leq \big[(i-1)s_p\big] + w_p \Big\} \quad (2)$$

The result of the i-th window is $\mathcal{S}_i = \mathcal{H}(\{\mathcal{R}_j\}_{j \in \mathcal{Z}_i})$ and represents the set of the most preferred tuples among to ones in the panes of the window, i.e. $\mathcal{S}_i \subseteq \bigcup_{j \in \mathcal{Z}_i} \mathcal{R}_j$.

The functions $\mathcal{G}$ and $\mathcal{H}$ are specific of the query. As an example, in the skyline query the PLQ computes the local skyline of each pane, and the global skyline of each window is computed by the WLQ using the local skylines of the panes. The same idea can be used for top-δ dominant query, where the set of k-dominant skyline points is always a subset of the skyline [20]. Analogously, in top-k queries the individual answers on the panes can be merged to produce the top-k answers of the windows [24].

The pane-based model reduces the query processing time [15]. In fact, each pane result $\mathcal{R}_i$ contains a subset (likely small) of the original input tuples of the pane. Consequently, the WLQ performs fewer tuple comparisons because it processes pane results instead of input tuples, and the results of panes shared among windows can be re-used by saving computation time. In addition, the pane-based approach has two useful properties:

Exploiting parallelism: the execution on different panes (PLQ) and windows (WLQ) is independent and they can be processed concurrently by multiple threads in order to increase the overall query throughput.

Handling disordered tuples: to enable OOP tuples must be processed in the arrival order. The pane-based approach is compatible with this vision provided that a proper mechanism is implemented in the PLQ to detect when all the tuples of a pane have likely been received. This aspect will be studied in Sect. 4.

3 Overview of the Parallel Solution

We designed a parallel implementation of the pane-based approach written on top of the FastFlow parallel programming framework [17] for streaming applications on multi-core systems.

FastFlow is a `C++11` template library. The FastFlow model allows the programmer to build networks of streaming operators. Single-threaded operators cooperate by exchanging data items (tasks) through clearly identified data paths. Each operator performs an infinite loop that: *i)* gets a task (a memory pointer to a data item) from its input queue; *ii)* executes a user code on the task; *iii)* puts a task/result (a pointer) into its output queue. FastFlow uses lock-free single-producer single-consumer queues to enable low-latency cooperation among operators [25].

FastFlow provides two basic parallel patterns to build graphs of operators. The first is the *pipeline* pattern, which allows the tandem composition of streaming operators working on different data items in parallel. The second is the *farm* pattern, where the same computation takes place on different data items in parallel. The pane-based approach is parallelized as a pipeline of two parallel stages (the PLQ and WLQ ones), each of them parallelized according to the farm parallel pattern as shown in Fig. 2. The two stages, although based on the same parallelism pattern, will be characterized by proper customized scheduling strategies as it will be explained in Sects. 4 and 5.

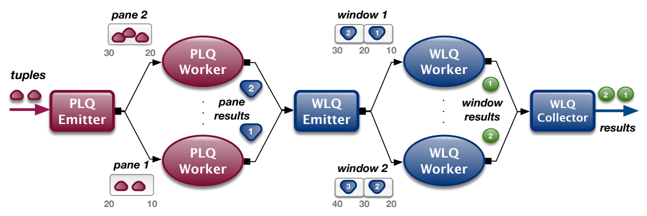


Fig. 2: Parallelization of pane-based sliding-window queries.

Each farm has an *emitter* operator in charge of splitting the input stream into several outbound streams, one for each *worker* operator. Workers are identical replicas of the same functionality. PLQ workers execute the pane-level query on the panes, while the workers within the WLQ perform the window-level query. The *collector* in the second farm is responsible to merge back the results of different windows into a single stream by restoring their correct order.

The whole parallel implementation can be instantiated by the user as a C++ object extending the FastFlow pipeline pattern, by providing as input arguments to the constructor the window specifications (window length and slide) and the sub-query functions of the PLQ and the WLQ stages.

4 Parallel Pane-Level Query

The PLQ parallel implementation is designed to withstand the arrival rate (input bandwidth) without being a *bottleneck* (this condition will be formalized later in this section). In this part we will study and evaluate the implementation choices applied to the PLQ emitter and workers.

4.1 PLQ Emitter

The PLQ emitter functionality is a sequential operator responsible to serve two distinct roles:

- *handling of out-of-order arrivals*: as studied in Ref. [14], OOP needs that input streams are *punctuated*, i.e. they are enriched with special meta-tuples used as a stream progress indicator. In our implementation the PLQ emitter generates such punctuations to the workers;
- *scheduling of input tuples* to the workers in order to exploit at best their computational capabilities.

In the following sections we will describe how these roles have been implemented in our approach.

4.1.1 Punctuation generation

The distinguishable aspect of out-of-order data streams is the presence of so-called *late arrivals* [8]. A tuple $t \in \mathcal{D}$ is a late arrival if it arrives after a tuple $t' \in \mathcal{D}$ such that $t'.ts > t.ts$. The presence of late arrivals makes difficult to determine when all the tuples of a pane have been received. A solution is to embed in the stream special meta-tuples called *punctuations* [26]. A punctuation contains no meaningful fields except a timestamp attribute. A punctuation with timestamp t_p means that no more tuples with timestamp lower than t_p are expected. Such a punctuation allows the PLQ to close all the panes with identifier smaller than $\lceil t_p/L_p \rceil$, i.e. their results can be produced to the WLQ.

The punctuation logic is localized in the PLQ emitter. One of the approaches that can be used to generate punctuations is based on the *K-slack* algorithm [27]. The original idea has been commonly used for IOP systems, where tuples are first buffered and then transmitted to the query in order of their timestamps. In that case the K-slack logic is used to determine how long the tuples must be buffered before being transmitted in order. Instead, in this work we forward immediately the tuples to the workers in their arrival order, and we use the K-slack algorithm to determine the value of punctuations that are periodically embedded in the stream.

The K-slack algorithm uses a delay parameter K initialized to zero. A variable t_{max} stores the maximum timestamp of the tuples received so far. For each newly received tuple t, $t.ts$ is compared with t_{max}. If t_{max} must be updated, the emitter performs the following actions:

- the value of K is updated to the maximum between its previous value and $d(t_i) = t_{max} - t_i.ts$ for all the tuples t_i received since the last update of t_{max};
- the emitter generates a punctuation with value $t_p = t_{max} - K$, which is *broadcasted* to all the PLQ workers.

This mechanism does not guarantee that the punctuation timestamps are monotonically increasing. Therefore, the PLQ emitter emits a new punctuation only if it is a real stream progress, i.e. its timestamp is greater than the one of the last emitted punctuation.

Punctuations are used to admit or drop the tuples that arrive at the query. For each received tuple, if its timestamp is greater than the value of the last emitted punctuation the tuple is scheduled to the workers (as described in the next section). Otherwise, the tuple is dropped and it will not contribute to the result of the corresponding pane.

4.1.2 Scheduling of tuples

The major role of the PLQ emitter is to schedule the tuples to the PLQ workers. A basic strategy (referred to as `PB_RR` in the sequel) consists in a round-robin assignment of panes to the workers, i.e. all the tuples of the i-th pane are scheduled to the worker with identifier $j = (i \mod n) + 1$, where $n \geq 1$ is the number of workers (the PLQ *parallelism degree*).

The main drawback of this scheduling strategy is its inability to keep the computational load always balanced among workers, potentially causing the PLQ to be a bottleneck. Load unbalancing can be caused by both *extrinsic* and *intrinsic* factors. Extrinsic factors are related to the temporal properties of the input stream, like its velocity and the presence of bursts. A burst can be defined as a time period in which the number of arrivals is abnormally over the average. In bursty scenarios the amount of tuples per pane can be very uneven, making the computation on some panes more time-consuming than the others. When such bursty behavior happens, the `PB_RR` scheduling may be unfair because several costly panes can be assigned to the same workers while other workers may remain idle.

The unbalance can be amplified by intrinsic factors peculiar to the query executed. We recall that the workers process the tuples incrementally by updating the corresponding pane result (the output set of the most preferred tuples in the pane). The number of tuples in the pane result may depend on the spatial distribution of the tuple attributes. For example, the skyline is small if the dataset is *correlated*, whereas *independent* or *anticorrelated* distributions produce larger skylines [19]. If the stream conveys tuples with a time-varying spatial distribution, it is be possible that panes with similar input cardinalities have a substantially different size of their skylines, and thus tuples of different panes may have different processing times. Such feature exacerbates the problem of finding an effective load distribution.

To remedy this problem, we need a scheduling strategy able to withstand the input bandwidth in presence of both the factors. Our solution is based on a *proactive splitting* of panes. The general idea is the following:

- the PLQ emitter records the number of tuples per pane transmitted to each worker and periodically estimates a *splitting threshold* θ expressed in terms of number of tuples. The threshold estimation requires a careful design that will be described later in this section;
- when the first tuple t of pane $\mathcal{P}_i$ is received, the worker w_j with the smallest number of tuples in its input queue at that time instant (called *least loaded worker*, briefly LLW) becomes the owner of that pane and t is scheduled to it;
- for each successive tuple t' of $\mathcal{P}_i$, if the current number of tuples of that pane transmitted to the owner is smaller than the actual value of the threshold θ, the tuple is still scheduled to the owner. Otherwise, the current LLW, let say w_k, becomes the new owner and t' is scheduled to it.

Therefore, more workers can hold different partitions of the same pane. We denote by $\mathcal{P}_{i,j}$ the j-th partition of the i-th pane and by $\mathcal{R}_{i,j}$ its result. We define the *splitting factor* $\sigma_s \geq 1$ *as the average number of existing partitions per pane*.

The approach has an intrinsic proactive nature due to the guess that the algorithm performs when a pane partition exceeds the current threshold. In fact, the emitter is not aware of the number of tuples that will contribute to the new partition, and the splitting can be ineffective if it will contain too few tuples. However, the LLW policy counteracts this effect by assigning new partitions to those workers where such small partitions have been assigned.

In general, there are two advantages and a main concern of using low values for the threshold θ:

- ✓ *distribution fairness*: low thresholds allow the emitter to distribute almost the same number of tuples to the PLQ workers, despite the presence of possible extrinsic factors like a severe burstiness;
- ✓ *fine computational grain*: the processing time per tuple depends on the current cardinality of the corresponding result of the pane partition. With low θ both the input cardinality (number of input tuples) and the result cardinality (tuples in the result) of a pane partition are kept as smallest as possible. This is an effective way to cope with intrinsics factors to load unbalance;
- ✗ *result size blowup*: the greater the number of partitions per pane the larger the number of tuples transmitted to the WLQ, as the PLQ selects the local "best" tuples within each partition. This increases the computational requirement of the WLQ and may vanish the computational advantage of the pane-based approach.

In contrast, high threshold values allow the PLQ to produce smaller result sets at the expense of a possibly unfair distribution and a coarser computational grain.

The work in Ref. [28] targets the execution of sliding-window aggregates (e.g., sum, max, avg, quantiles) over data streams exhibiting load variations (not properly bursts at fine time scales). That work uses a sort of splitting approach driven exclusively by some statistical estimates such as the average size of the most recently completed panes. The downside of this solution is that it does not correlate the obtained splitting factor with the actual query performance, and the used splitting may be too aggressive (or too conservative) with dramatic performance consequences for preference queries. Our idea is to make the threshold selection aware of the actual PLQ performance. In fact, as it will be shown in Sect. 4.3, *it is not essential that all the workers are perfectly balanced, but it is just sufficient that none of them are bottlenecks*. Therefore, the scheduling should adapt the threshold in such a way as to apply the minimum splitting to make the PLQ not a bottleneck. In this way the PLQ is capable of sustaining the input bandwidth without producing too large result sets to the WLQ. To this end, we use an adaptive splitting method (called `AS_PID`) based on a PID (*Proportional-Integrative-Derivative*) controller [29].

Adaptive splitting with PID. Fig. 3 shows the logical components within the emitter. The `Statistics_Manager` records the number of tuples that take part to the pane partitions and periodically estimates a *base threshold* θ_b, calculated as the sum of the average size of the most recent pane partitions closed by a punctuation plus the standard deviation, similarly to Ref. [28]. The `Scheduler` receives the tuples from the stream and dispatches them to the PLQ workers by using an *actual splitting threshold* θ calculated as $\theta = \alpha \cdot \theta_b$, where $\alpha > 0$ is an adaptation parameter.

The `Controller` is a discrete-time PID with a sampling interval of T time units. Every sampling interval the controller gets the PLQ *utilization factor* from the `Monitoring` component. The utilization factor ρ is defined as in Queueing Theory [30], i.e. it is the ratio of the arrival rate of

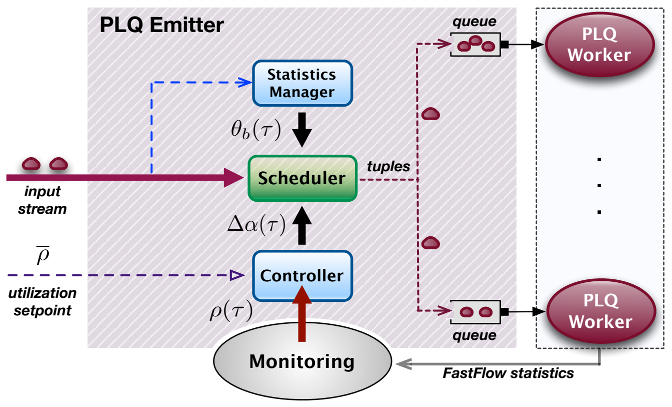


Fig. 3: Logical components of the PLQ emitter.

tuples over the average number of tuples that the PLQ is able to serve per time unit. The condition $\rho < 1$ states that the PLQ is not a bottleneck, while values greater than one indicate that the service rate is not high enough to withstand the input bandwidth. At each sampling interval τ, the controller determines an adjustment $\Delta\alpha(\tau)$—i.e. an increase or a decrease—of the $\alpha(\tau)$ parameter, and provides it to the `Scheduler` in order to compute the splitting threshold used for the next interval. The value of $\Delta\alpha(\tau)$ is chosen according to the PID control law stated as follows:

$$\Delta\alpha(\tau) = K_p\, e(\tau) + K_i \sum_{i=0}^{\tau} e(i) + K_d\, [e(\tau) - e(\tau-1)] \quad (3)$$

where $e(\tau) = \overline{\rho} - \rho(\tau)$ is the absolute error between a setpoint value $\overline{\rho}$ and the measured utilization factor. To remove the bottleneck without splitting too much the panes, the setpoint should be set to a value slightly smaller than one. The adjustment is the sum of three components: a term proportional to the last error, a term proportional to the sum of the last errors (integrative), and a term proportional to the last variation of the error (derivative). To set the weights K_p, K_i and K_d we use the Aström-Hägglund relay method [31] to tune the controller automatically during an initial *warm-up* period of the execution. Furthermore, we use classic techniques developed for PIDs in order to avoid the accumulation of past errors (*integral windup* [32]) which can occur during phases where the error is large independently of the controller action, e.g., when the PLQ is a bottleneck also using the minimum threshold, or when the operator has very low utilization also with a very high threshold.

The pseudo-code of the `Scheduler` component is shown in Alg. 1. The scheduling strategy obtains the last estimate of the utilization factor and the new value of $\Delta\alpha(\tau)$ at lines 3 - 5. The algorithm uses a data structure $\mathcal{PD}$ to store a descriptor for each pane. Each descriptor contains: *i)* the number of tuples currently assigned to the owner (a_tuples); *ii)* the identifier of the owner of the pane (a_worker); *iii)* an array of counters (i_sizes) representing the number of tuples of that pane currently scheduled to each worker (zero if the worker does not have a partition of that pane). The descriptors are purged from $\mathcal{PD}$ when panes are closed.

From line 9 to 15 the algorithm handles the case where no splitting is needed. If t is the first tuple received for pane

Algorithm 1 `AS_PID` Scheduling Strategy

```
1:  function PLQ_SCHEDULING($t, \theta_b$)
2:      if $curr\_time() > end\_sample\_time$ then
3:          $\rho(\tau)$ = MONITORING.GETUTILIZATION()
4:          $\Delta\alpha(\tau)$ = PID.GETOUTPUT($\rho(\tau)$)
5:          $\alpha(\tau) = \alpha(\tau) + \Delta\alpha(\tau)$
6:          $end\_sample\_time = curr\_time() + T$
7:      end if
8:      $i = \lfloor t.ts/L \rfloor$
9:      if $\mathcal{PD}[i].a\_tuples < \alpha(\tau) \cdot \theta_b$ then        ▷ Case 1: no splitting
10:         if $\mathcal{PD}[i].a\_tuples = 0$ then
11:             $w$ = GET_LLW_ID()
12:             $\mathcal{PD}[i].a\_worker = w$
13:         end if
14:         $\mathcal{PD}[i].a\_tuples = \mathcal{PD}[i].a\_tuples + 1$
15:         SEND_TO_WORKER($t, \mathcal{PD}[i].a\_worker$)
16:     else                                                   ▷ Case 2: splitting
17:         $w$ = GET_LLW_ID()
18:         $oldw = \mathcal{PD}[i].a\_worker$
19:         $\mathcal{PD}[i].a\_worker = w$
20:         $\mathcal{PD}[i].i\_sizes[oldw] = \mathcal{PD}[i].a\_tuples$
21:         $\mathcal{PD}[i].a\_tuples = \mathcal{PD}[i].i\_sizes[w] + 1$
22:         SEND_TO_WORKER($t, \mathcal{PD}[i].a\_worker$)
23:     end if
24: end function
```

i, the `Scheduler` selects the LLW (line 11) as the owner. Pane splitting case is handled from line 16 to 23.

PLQ utilization factor estimation. The estimation of the utilization factor by the `Monitoring` component is critical. FastFlow provides an API to monitor the workers activity [17]. Measurements that can be collected are:

- *work_time*: the total time spent in processing input tuples (it does not include the periods in which the worker was idle waiting for a new input tuple);
- *tasks_cnt*: the total number of tuples processed by a worker from the beginning of the execution.

These measurements are used to estimate the fraction of the last sampling interval in which worker i was active in processing tuples (denoted by ϕ_i) and the number of tuples processed q_i. The used symbols are summarized in Tab. 1.

Symbol	M or C	Description
ϕ_i	M	length of the fraction of the last sampling interval in which the i-th worker was active in processing tuples.
ϕ_{tot}	C	sum of the intervals ϕ_i of the workers, i.e. $\phi_{tot} = \sum \phi_i$.
q_i	M	number of tuples processed by the i-th worker in the last sampling interval.
q_{tot}	C	total number of tuples processed by the workers in the last sampling interval, i.e. $q_{tot} = \sum q_i$.
μ_i	C	number of tuples that the i-th worker would have been able to process during the last sampling interval.
λ_i	M	number of tuples scheduled to the i-th worker in the last sampling interval.
λ_{tot}	C	number of tuples scheduled by the PLQ emitter in the last sampling interval, i.e. $\lambda_{tot} = \sum \lambda_i$.
p_i	C	probability to schedule a tuple to the i-th worker, i.e. $p_i = \lambda_i / \lambda_{tot}$.

TABLE 1: Metrics used for estimating the utilization factor. In the second column the symbol M means that the metric is directly *measured* from the execution, C means that the metric value is *computed* using cost models.

To accommodate fluctuating input rates like in bursty scenarios, we configure the farm pattern in order to use *unbounded* FastFlow queues [25]. In this way if one of the worker is currently congested, the emitter is still capable of scheduling the tuples to the other workers at the arrival speed. This allows each steady-state arrival rate λ_i to the various workers to be derived independently of each other.

To determine the utilization factor of the i-th worker, we need an approximation of its *service rate* μ_i, i.e. the expected number of tuples that it is able to serve in a sampling interval assuming that the worker is never idle. This estimation must be carefully evaluated because of the incremental nature of the computation. In fact, it should be noted that the processing time of a tuple over an almost empty pane is low compared with the one of a tuple computed on panes that are about to be closed by a punctuation. We use a global average estimate of the processing time per tuple $\mathcal{C}$ computed as the ratio of ϕ_{tot} over the total amount of tuples processed by the workers, i.e. $\mathcal{C} = \phi_{tot}/q_{tot}$. The service rate is estimated as:

$$\mu_i = q_i + \frac{T - \phi_i}{\mathcal{C}} \tag{4}$$

that is, we add to the number of already processed tuples the estimated number of tuples that the worker would have been able to serve during the idle part of the last interval.

The use of unbounded queues allows the workers to act as independent systems, and the global utilization of the PLQ can be evaluated as the weighted average of the utilization factors of its workers, where the weights correspond to the distribution frequencies:

$$\rho = \sum_{i=1}^{n} \left(p_i \cdot \frac{\lambda_i}{\mu_i} \right) = \sum_{i=1}^{n} \frac{\lambda_i^2}{\lambda_{tot}\, \mu_i} \tag{5}$$

where $n \geq 1$ is the number of workers. As it will be shown experimentally in Sect. 4.3, this cost model will be able identify accurately when the PLQ is a bottleneck.

4.2 PLQ Workers

PLQ workers receive both tuples and punctuations from the emitter. They maintain a private data structure $\mathcal{RD}$ containing a result descriptor for each pane partition. A worker j can execute the following types of actions:

- *search*: determine whether the result of a pane with a specified identifier i exists in $\mathcal{RD}$;
- *insertion*: the result of a pane partition is created and inserted into $\mathcal{RD}$. The result is incrementally updated each time a new tuple of that pane is received;
- *removal*: when a punctuation t_p is received, the descriptors of panes with identifier $i < \lceil t_p/L_p \rceil$ are deleted from $\mathcal{RD}$ and the results transmitted to the WLQ.

While the search is performed each time a tuple arrives at the worker, the other actions are less frequent. We implement the $\mathcal{RD}$ container as a double-ended queue (`C++STL deque`), in order to reach a good compromise between efficiency and simplicity. Descriptors are kept ordered by the pane identifier in order to exploit a logarithmic time search, while removals are handled efficiently because we remove the first descriptors in the deque (the panes closed by a punctuations are always at the beginning of the container). Also insertions are handled efficiently, as they likely happen at the last positions of the deque (except for huge disordering). The pseudo-code of a worker is shown in Alg. 2.

Algorithm 2 PLQ worker

```
1:  function PROCESS_TUPLE($t$)
2:      $i = \lfloor t.ts/L \rfloor$
3:      if $t = punctuation$ then                    ▷ Case 1: $t$ is a punctuation
4:          $\Omega = \mathcal{RD}$.SEARCH_LOWER_BOUND($i$)
5:          for each $p \in \Omega$ do
6:              SEND_TO_WLQ($p.resultset$)
7:              $\mathcal{RD}$.REMOVE($p$)
8:          end for
9:      else if $t = tuple$ then                     ▷ Case 2: $t$ is a regular tuple
10:         if $\mathcal{RD}$.SEARCH_ENTRY($i$) $= false$ then
11:             $p =$ new $Pane(i)$
12:             $\mathcal{RD}$.INSERT($p$)
13:         else
14:             $p = \mathcal{RD}$.GET_ENTRY($i$)
15:         end if
16:         $p$.ADD_COMPUTE($t$)
17:     end if
18: end function
```

The worker handles punctuations from line 3 to 8. The function called at line 4 returns the set of pane entries with identifier lower than the input argument. The results of those entries are transmitted to the WLQ and the corresponding pane partitions are closed. In the code region from line 9 to 16 the algorithm processes a regular tuple. The corresponding pane partition is searched in $\mathcal{RD}$: if it is not found we create a new entry for the pane partition. Then, the worker updates the corresponding result at line 16.

4.3 PLQ Evaluation

In this section we describe the experimental platform used in this paper and how synthetic datasets are generated. In particular, we show the application of a methodology to parameterize the burstiness level of synthetic datasets. Then, we will demonstrate the effectiveness of the PID-based adaptive splitting scheduling described before.

4.3.1 *Test-bed architecture*

All the experiments shown in this paper have been obtained on a two-socket IBM server 8247-42L equipped with two Power8 processors each with ten cores (total 20 cores). Each core has eight thread contexts (SMT), private L1d and L2 caches per core of 64 KB and 512 KB, and a shared on-chip L3 cache of 8 MB per core (globally 80 MB per socket). The machine is installed with 64 GB of RAM.

The source code of the parallel implementation using the FastFlow library version `2.1` has been compiled with the `gcc` compiler version `4.8.4` with the `-O3` optimization flag enabled. Each benchmark has been repeated 20 times and the results show a small standard deviation (in the range of $1.5 - 4\%$ of the mean). For this reason and to improve readability we avoid showing the error bars in the plots.

The run-time system of our framework (FastFlow, see Sect. 3) is strongly based on non-blocking synchronization primitives that provide a very low-latency cooperation mechanism among threads [25]. However, when more threads are mapped onto the same physical cores such kind of synchronization usually generates a mutual interference among threads that can be detrimental for performance. For this reason: *i)* we limit the maximum query parallelism to the number of physical cores in order to have more stable and predictable results; *ii)* we will show that this parallelism degree will be sufficient in real-world scenarios.

4.3.2 Synthetic datasets

To stress our parallel implementation, we developed a configurable data generator that produces synthetic datasets of tuples. Each tuple has an *application timestamp* used to determine the pane/window boundaries, and a *generator timestamp* used to transmit tuples according to a chosen rate. The generator timestamp is obtained from the application one by adding a random delay (generated using a uniform distribution with mean D_{avg}). The tuples in the dataset are ordered and transmitted based on their generator timestamps. Therefore, a tuple t received after t' can have an application timestamp smaller than the one of t', i.e. it is a late arrival. In general, the higher the average delay the greater the number of late arrivals in the dataset.

We use the approach presented in Ref. [16] to inject bursty phases with different intensities. We use a two-state *Markovian Arrival Process* model (MAP(2)). The idea is depicted in Fig. 4. In the "normal" state the inter-arrival times are generated using an exponential distribution with rate λ_{normal} set to a value reproducing a condition of normal traffic; in the "bursty" state the rate is $\lambda_{burst} > \lambda_{normal}$, hence the arrivals are more closely spaced possibly generating bursts. The terms $p_{n,b}$ and $p_{b,n}$ denote the probabilities to change the state, while $p_{n,n} = 1 - p_{n,b}$ and $p_{b,b} = 1 - p_{b,n}$ are the probabilities to remain in the same state.

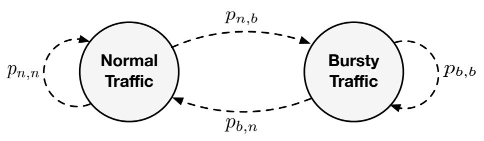
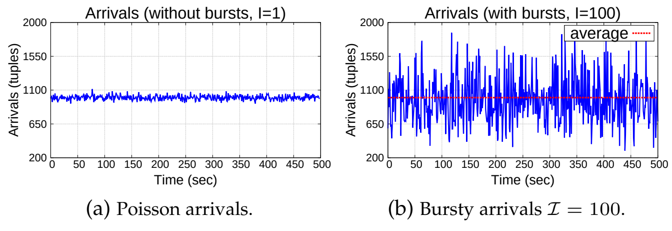


Fig. 4: Two-state MAP distribution modeling bursty arrivals.

This model allows generating both short temporal intervals with highly condensed arrivals and variations within a traffic burst. As in Ref. [16], we use the *index of dispersion* $\mathcal{I}$ as a regulator of the intensity of traffic surges. It is defined as $\mathcal{I} = SCV \cdot (1 + 2\sum_{i=1}^{\infty} r_k)$, where SCV is the squared coefficient of variation and r_k is the k-lag autocorrelation coefficient. As shown in Ref. [33], the index of dispersion has the fundamental property to grow proportionally with both variability and correlations, thus it can be used to generate a sequence of inter-arrival times reproducing bursty arrivals, i.e. the higher the value of $\mathcal{I}$ the higher the burstiness level.

To parameterize the MAP(2), we provide the desired mean inter-arrival time λ^{-1} and the target index of dispersion $\mathcal{I}$ as inputs of a non-linear optimizer that searches the probabilities $p_{n,b}$ and $p_{b,n}$ such that the MAP(2) model matches as much as possible the required index of dispersion. Further details about this approach can be found in Ref. [16]. Fig. 5a shows the case of Poisson arrivals (exponential inter-arrival times) with average rate of $1K$ tuples per second. In this case we have no bursts, and the index of dispersion is equal to the SCV of the distribution, i.e. $SCV = 1$. Fig. 5b shows a MAP(2) distribution, fitted with the above approach, having the same mean and an index of dispersion $\mathcal{I} = 100$, where the effect of burstiness is already remarkable.



(a) Poisson arrivals. (b) Bursty arrivals $\mathcal{I} = 100$.

Fig. 5: Effect of the burstiness in two synthetic datasets.

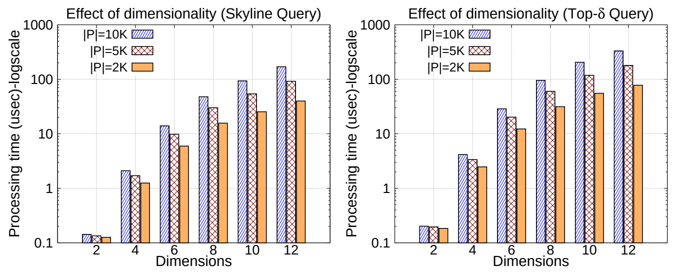
4.3.3 Results

The benchmarks in this section show the effect of dimensionality and data correlation in the computational grain, the effectiveness of our scheduling strategy in the PLQ, and the benefit of the out-of-order processing.

Effect of dimensionality and correlation. In this paper we focus on two computationally demanding preference queries, i.e. the skyline query and the top-δ dominant query (with $\delta = 100$). Coherently with some past works [22], [23], we study the case of panes containing $2 - 10K$ tuples on average, corresponding to arrival rates (denoted by λ) of tens of thousands/hundreds of tuples per second and pane lengths L_p of hundreds/thousands of milliseconds.

The implementation of the skyline query is based on the *block nested loop* algorithm (BNL) described in Ref. [19], which repeatedly scans the input set to compare all the pairs of tuples[2]. The top-δ dominant query has been implemented on top of the skyline one, where in addition we compute for each skyline tuple t the highest $k \leq d$ such that t is k-dominated by another tuple. The query returns the first δ tuples with the lowest k value.

Fig. 6 shows the average processing time per tuple (i.e. time to update the pane result given a new tuple t) by variating the dimensionality d. The processing time is proportional to both the number of tuples per pane, i.e. $|\mathcal{P}| = \lambda \cdot L_p$, and the number of attributes per tuple $d \geq 1$. The top-δ dominant query has a higher processing time, $1.5 - 2$ times higher than the skyline query.



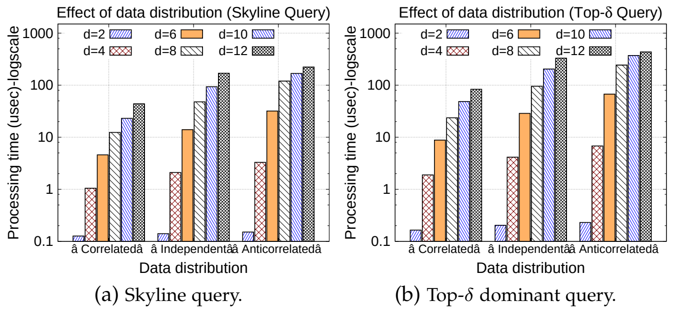
(a) Skyline query. (b) Top-δ dominant query.

Fig. 6: Effect of dimensionality in the processing time per tuple.

Fig. 7 shows the processing time per tuple by changing the data distribution. We consider correlated, independent (used for the experiments in Fig. 6) and anticorrelated distributions. The distribution affects the size of the pane results. In the skyline query, 74% of the tuples belong to the skyline with $d = 12$ and anticorrelated data, while in

2. More efficient algorithms, e.g., based on some index structures, can be used and easily integrated in our parallel query model.

the correlated case the skyline is composed of only 3% of the pane tuples. The lower the number of dimensions the higher the capability of the PLQ to incrementally filter out non-skyline tuples, thus the lower the processing time.



(a) Skyline query. (b) Top-δ dominant query.

Fig. 7: Effect of data distribution in the processing time per tuple. Case with $|\mathcal{P}| = 10\text{K}$.

Evaluation of the PLQ scheduling strategies. We study the skyline query under four scenarios all having $10K$ tuples per pane on average ($\lambda = 100K$ tuples/sec and $L_p = 0.1$ seconds). The first scenario has no bursts, the others have a *low*, *medium* and *high* level of burstiness[3] obtained by generating datasets with $\mathcal{I} = 1K$, $\mathcal{I} = 2.5K$ and $\mathcal{I} = 6K$. In the non-bursty scenario the tuples (eight attributes to have sufficient computational grain) are independently distributed, while in the bursty scenarios we periodically change the data correlation. In all the datasets we use an average delay of $D_{avg} = 200$ ms similar to the one considered in other works [8], [9]. In the last part of this section a thorough study of different disordering configurations will be presented.

We study the behavior of the `PB_RR` and `AS_PID` strategies by comparing them with other two strategies:

- `TB_RR`: tuples are distributed to the workers in a round-robin fashion. Therefore, all the panes are evenly partitioned among the workers, i.e. σ_s is equal to the parallelism degree;
- `AS`: a modification of the `AS_PID` strategy without PID. The actual threshold is directly the one from the `Statistics_Manager`, i.e. $\theta = \theta_b$, similarly to the solution proposed in Ref. [28].

Fig. 8a shows the minimum parallelism degree needed to sustain the input bandwidth (named *optimal parallelism degree*, denoted by n^*). In the non-bursty case (where the standard deviation of the pane size is 2.8% of the mean) the `PB_RR` strategy needs seven workers to withstand 100% of the input bandwidth. In all the bursty scenarios instead (with standard deviation equal to 123% of the mean in the highly-bursty case), `PB_RR` *is not able to remove the bottleneck with any parallelism degree*. In those cases we report the fraction of the input bandwidth sustained by the PLQ with the highest parallelism degree (17 workers[4]). The higher the burstiness level the lower the fraction of the input traffic managed. This result is not due to the lack of cores, but to load unbalancing that prevents to exploit the worker processing capabilities. This is confirmed by the results in Figs. 8d-f, where the utilization factor of `PB_RR` is always greater than one by increasing the number of workers. These results show the effectiveness of the utilization factor estimation method described in Sect. 4.1.2. As soon as we obtain $\rho < 1$, the PLQ is able to sustain 100% of the input bandwidth, i.e. on average $1/L_p$ pane results per second are produced by the PLQ.

Instead, the `TB_RR` strategy removes the bottleneck in all the scenarios. This outcome is due to the high splitting factor, as the panes are evenly partitioned among the workers. Since the processing time per tuple is proportional to the size of the corresponding pane partition of the destination worker, the `TB_RR` strategy reduces the computational grain compared with `PB_RR`. This advantage is offset by the increase in the number of selected tuples per pane (union of the local skylines of the pane partitions). Fig. 8g shows that in the highly-bursty scenario the average number of selected tuples per pane with 17 workers is 60% greater than in the `PB_RR` case. Therefore, the lower computational grain in the PLQ is actually paid in the WLQ, where the pane results are merged to compute the final window results.

The `AS_PID` strategy (with $\overline{\rho} = 0.9$ and $T = 1$ second) finds the right amount of splitting to remove the bottleneck with the used parallelism degree. If `AS_PID` is not able to achieve $\rho < 1$, it behaves like `TB_RR` by splitting the panes as much as possible, see Fig. 8b. The splitting factor, higher with a more intensive burstiness level, increases up to the optimal parallelism degree n^* and decreases by adding more workers. This is because, owing to the presence of more available resources, the scheduler needs to split less the panes to get closer to the utilization factor setpoint. Peculiar is the case of the non-bursty scenario, where with more than seven workers `AS_PID` avoids splitting the panes (seven workers is the n^* of `PB_RR`, see Fig. 8a). The precision of the PID in confirmed by the results in Figs. 8c-f, where the utilization factor is always near to the setpoint (error less than 2%) with parallelism degrees greater or equal to n^*.

The `AS` strategy divides the panes by using a statistic threshold without the PID intervention. The average splitting factor applied by `AS` is of 1.01, 1.48, 1.66 and 2.41 in the non-bursty scenario and in the three bursty scenarios. There are two situations where this strategy is ineffective:

- *the splitting factor with* `AS` *could be insufficient to remove the bottleneck*. As shown in Fig. 8f, with five workers the `AS` strategy obtains $\rho \approx 1.88$ with $\sigma_s = 2.41$ while `AS_PID` obtains $\rho \approx 0.91$ with $\sigma_s = 3.90$. In that case `AS_PID` transmits to the WLQ stage more tuples per pane than using the `AS` strategy (13% more, see Fig. 8g), however such splitting is necessary to remove the bottleneck (the `AS` strategy with five workers withstands only 53% of the input bandwidth);
- `AS` *can split too much the panes*. As an example, in the highly-bursty scenario with 17 workers the `AS` strategy achieves $\rho \approx 0.63$ with $\sigma_s = 2.41$, hence the PLQ is unnecessarily fast. Instead, `AS_PID` obtains $\rho \approx 0.89$ with $\sigma_s = 2.03$ and the results of the pane partitions are 11% smaller than with `AS`, see Fig. 8g.

Finally, Fig. 8h shows the *load unbalancing ratio* of the average CPU load of the most loaded worker over the one

3. Bursty workloads are characterized by an index of dispersion of several thousands, as stated in Refs. [16].
4. Two cores host two threads generating the stream and consuming the pane results, while another core hosts the PLQ emitter thread.

(a) Optimal no. of cores.

(b) Splitting factor with PID.

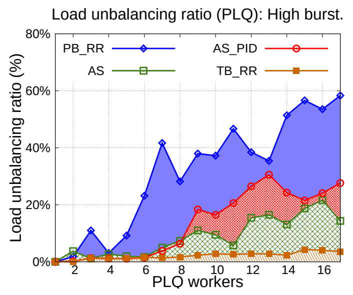
(c) Utilization factor (no burst.).

(d) Utilization factor (low burst.).

(e) Utilization factor (medium burst.).

(f) Utilization factor (high burst.).

(g) Select. tuples per pane (high burst.).



(h) Load unbalan. ratio (high burst.).

Fig. 8: Performance evaluation and comparison between the various scheduling strategies of the PLQ stage. Skyline query with high ($\mathcal{I} = 6K$), medium ($\mathcal{I} = 2.5K$) and low ($\mathcal{I} = 1K$) burstiness and $|\mathcal{P}| = 10$K.

of the least loaded worker. The higher the ratio the higher the load unbalance. In the highly-bursty case the ratio is high with PB_RR (up to 58% with 17 workers). TB_RR is always able to balance the load. Adaptive splitting techniques obtain intermediate results. The PID-based approach is less effective in terms of load balancing than AS. This is an expected result, because AS_PID applies the minimum splitting to remove the bottleneck and although not optimal the load balancing with AS_PID is sufficient to obtain $\rho < 1$.

In conclusion, these results confirm that the AS_PID scheduling strategy is effective in eliminating the bottleneck in the PLQ even in highly-bursty scenarios and without splitting too much the panes.

Comparison between IOP and OOP models. To compare the different processing models, we developed an IOP version of the PLQ in which tuples are internally buffered by the emitter using the standard K-slack buffer [27], and presented in increasing order of their application timestamps to the workers. Therefore, no punctuation is needed in this implementation. We introduce the following definition:

Definition 1 (Pane latency). *The latency of a pane is the time elapsed from the arrival of the first tuple of the pane until the last tuple has been computed.*

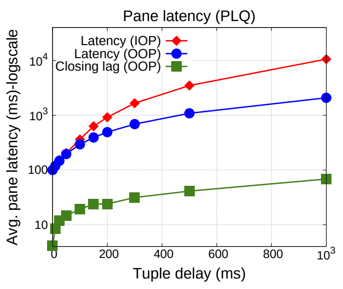
Fig. 9a shows the latency of the OOP and IOP versions in the highly-bursty scenario studied above. Since the processing time per tuple depends on the splitting factor, we use the TB_RR strategy with $n = 5$ workers in order to have a fair comparison and to remove the bottleneck. The results show that the higher the tuple delay the greater the latency reduction with OOP, while the latency is similar with low delays. The reason is due to the buffering performed by the emitter in the IOP version, while in OOP the tuples are immediately forwarded to the workers. In all cases the dropped tuples are about 0.01%. The latency reduction is remarkable: 68% and 80% with delays of 500 ms and 1 second. Nevertheless, though with smaller values, also the latency of the OOP version has an increasing slope. To understand this aspect we introduce the following definition:

Definition 2 (Closing lag). *The closing lag of a pane is the time elapsed from the computation on the last received tuple of the pane until the arrival of the punctuation that closes it.*

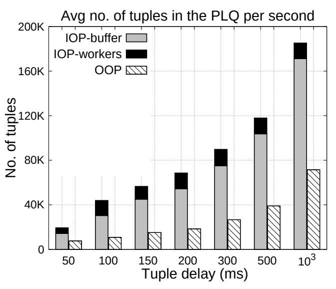
The average closing lag increases with higher delays (Fig. 9a) because punctuations are emitted less frequently. However, it represents a small fraction of the pane latency which is substantially lower than in the IOP case. We observe that the closing lag can be significantly lower than the average tuple delay, because the last late arrival of a pane can be received and computed slightly before the generation of the punctuation that closes the pane.

The case with a zero delay (a totally ordered stream) deserves a further consideration. In this case, according to the description in Sect. 4.1.1, a new punctuation is emitted every new tuple. Although this huge number of punctuations could be considered a high overhead, actually it has an imperceptible impact owing to the efficient way adopted by the FastFlow framework to exchange data among threads (i.e. passing pointers to punctuation messages through lock-free queues [25]) and the negligible processing time of punctuations that do not close any pane (few microseconds on average). In addition, our implementation can be further configured to limit the maximum number of punctuations to emit per second if needed.

Fig. 9b shows the average number of tuples present in the computation per second including: *i)* the tuples buffered in the K-slack buffer within the emitter (only in the IOP case); *ii)* the tuples in the pane results maintained internally by the workers; *iii)* the tuples enqueued in the FastFlow input queues of the workers. For the OOP case in the figure we show the overall number of tuples, while in the IOP case



(a) Pane latency.



(b) Stored tuples per second.

Fig. 9: IOP and OOP comparison in the highly-bursty scenario: average latency and number of tuples in the PLQ per second.

we separate the number of tuples at point *i)* (denoted by IOP-buffer) and the sum of the tuples at points *ii)* and *iii)* (denoted by IOP-workers).

In the IOP case most of the tuples are in the K-slack buffer, and this number increases with higher delays. In the OOP version the tuples are forwarded to the workers as they arrive. Since the result of a pane only contains the selected tuples (i.e. the skyline tuples), the sooner the tuples are processed by the workers the sooner the non-skyline tuples can be deleted from the results of the open panes. This is the reason for the lower number of tuples present in the PLQ (on average 60% less) with the OOP version.

Finally, we analyze the size of the $\mathcal{RD}$ data structure maintained by the workers in the OOP model, see Fig. 10. The number of open panes increases with higher delays and shorter panes. Even in the pessimistic case of very short panes and long delays, the number of open panes is at most in the order of hundreds and grows linearly with the average delay. This justifies the choice of the data structures used in the worker implementation, see Sect. 4.2.

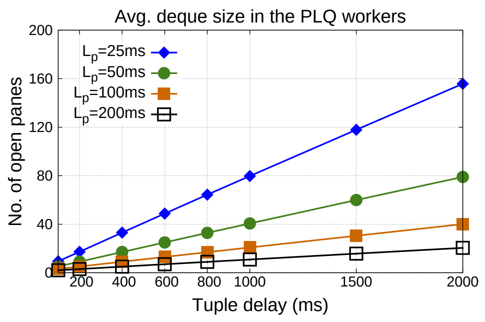


Fig. 10: Average size of the $\mathcal{RD}$ data structure (deque).

In conclusion, these experiments confirm that the OOP model is useful for continuous preference queries both for latency and space cost reasons.

5 Parallel Window-Level Query

The goal of the WLQ is to find the set of preferred tuples within each temporal window. To this end, the WLQ computes the window results by processing the results of the panes received from the PLQ, where most of the input tuples have likely been filtered out.

Once a result $\mathcal{R}_{i,k}$ of a pane partition is received from the PLQ, the WLQ emitter determines the identifiers of the windows that pane i belongs to. We denote this set by Φ_i, defined as follows (for $i > w_p$):

$$\Phi_i = \left\{ j \mid j \in \mathbb{N}, \left\lceil \frac{(i - w_p)}{s_p} \right\rceil < j \leq \left\lceil \frac{i}{s_p} \right\rceil \right\} \qquad (6)$$

For each window $\mathcal{W}_j$ with $j \in \Phi_i$ its result $\mathcal{S}_j$ is incrementally updated by processing the tuples in $\mathcal{R}_{i,k}$. We call this activity *window task* (denoted by wt). The WLQ in general executes more window tasks for each received result $\mathcal{R}_{i,k}$, one for each window in Φ_i. In the next section we will study specific strategies to schedule window tasks to the WLQ workers.

5.1 Scheduling of Window Tasks

A task is a data structure containing a memory pointer to the result of a pane partition and a pointer to the window result to update. Tasks must be scheduled without modifying the computation semantics. Specifically, *tasks on different windows can be executed in parallel, while tasks on the same window must be processed serially* (they modify the same window result $\mathcal{S}_j$). Our goal is to design a scheduling strategy that meets this constraint while exploiting at best the worker processing capabilities.

The idea is to change the initial structure of the WLQ farm shown in Fig. 2 by using the *feedback* pattern-modifier provided by FastFlow. WLQ workers provide feedbacks to the emitter through additional FastFlow queues as shown in Fig. 11. Feedbacks are used to inform the emitter that workers are ready to receive new tasks.

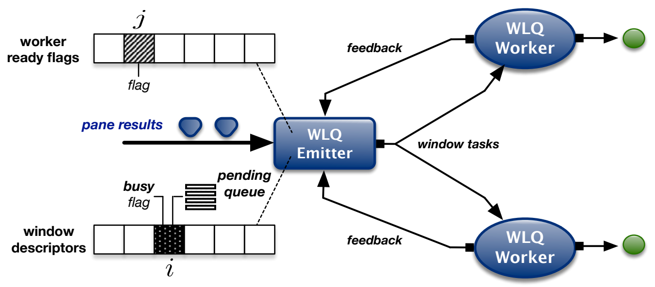


Fig. 11: WLQ emitter and feedback-based scheduling strategy of window tasks.

The pseudo-code of the scheduling strategy is described in Alg. 3. The WLQ emitter receives messages either from the PLQ (results of pane partitions) or from the WLQ workers (feedbacks). The emitter maintains a data structure $\mathcal{WD}$ that stores window descriptors containing: *i)* a flag which is false if there is a task of that window in execution in any worker (the window is *busy*); *ii)* a *queue* of pending tasks of that window, i.e. tasks that are waiting to be executed by an available worker. If the message comes from the PLQ, the algorithm generates a task for each window that that pane belongs to. Each task of a not-busy window is scheduled to an available worker. If a task modifies a busy window, or if there is no available worker, it is inserted into the *pending queue* of the corresponding window. This phase corresponds to the code region from line 2 to 13.

The case of a feedback message is managed from line 14 to 36. The feedback contains the identifier of the worker

Algorithm 3 Feedback-based Scheduling Strategy

```
 1: function WLQ_SCHEDULING($msg$)
 2:   if $msg.src = PLQ$ then                    ▷ Case 1: $msg$ comes from the PLQ
 3:     $\mathcal{P}_{i,k}$ = MSG.GETPANERESULT()
 4:     for each $j \in \Phi_i$ do
 5:       $worker\_id$ = GET_READY_WORKER()
 6:       if $\mathcal{WD}[j].notBusy$ and $worker\_id \neq -1$ then
 7:         $\mathcal{WD}[j].notBusy = false$
 8:         SET_NOT_READY($worker\_id$)
 9:         SEND_TO_WORKER(new $win\_task(\mathcal{P}_{i,k}, \mathcal{S}_j), worker\_id$)
10:       else
11:         $\mathcal{WD}[j]$. ADD_TO_QUEUE(new $WT(\mathcal{P}_{i,k}, \mathcal{S}_j)$)
12:       end if
13:     end for
14:   else                                       ▷ Case 2: $msg$ is a feedback message
15:     $worker\_id$ = MSG.GET_WORKER_ID()
16:     $w'$ = MSG.GET_WINDOW_ID()
17:     if $\mathcal{WD}[w']$. QUEUE_LEN $> 0$ then
18:       $wt = \mathcal{WD}[w']$. GET_PENDING()
19:       SEND_TO_WORKER($wt, worker\_id$)
20:     else
21:       $\mathcal{WD}[w'].notBusy = true$
22:       $served = false$
23:       for each $w \in \mathcal{WD}$ do
24:         if $\mathcal{WD}[w].notBusy$ and $\mathcal{WD}[w]$. QUEUE_LEN $> 0$ then
25:           $\mathcal{WD}[w].notBusy = false$
26:           $wt = \mathcal{WD}[w]$. GET_PENDING()
27:           SEND_TO_WORKER($wt, worker\_id$)
28:           $served = true$
29:           break
30:         end if
31:       end for
32:       if not $served$ then
33:         SET_READY($worker\_id$)
34:       end if
35:     end if
36:   end if
37: end function
```

and of the window of the last executed task. If there exist some pending tasks of that window, the first in queue is scheduled to the same worker sending the feedback (line 17 to 20). This strategy exploits at best temporal locality in the core's private caches by trying to schedule tasks of the same window to the same worker[5]. Otherwise, the emitter checks the presence of a pending task of any other not-busy window and, if presents, schedules it to the worker (line 23 to 31). Finally, if there are no pending tasks, the worker is marked as available (line 32 to 34).

5.2 WLQ Workers and Collector

The implementation of the WLQ worker is straightforward. For each window task received from the emitter, the worker executes the following actions: *i)* it computes the task by updating the result of the corresponding window; *ii)* if the task is the last of that window (this information is set through a flag in the task by the emitter, not shown in Alg. 3 for brevity), the worker transmits the window result to the collector; *iii)* the worker notifies its availability to receive new tasks by sending a feedback message to the WLQ emitter.

The WLQ collector receives window results from the workers. Its main goal is to temporarily buffer those results and emits them in increasing order of the window identifier.

5.3 WLQ Evaluation

In this section we evaluate the WLQ implementation using synthetic datasets generated as in Sect. 4.3.2. We will show the advantages of the feedback-based scheduler (`WIN_FB`) compared with a static strategy (`WIN_RR`) that preassigns windows to workers in a round-robin fashion, i.e. the tasks of the i-th window are executed by worker $j = (i \mod m) + 1$ where $m \geq 1$ is the WLQ parallelism degree.

WLQ performance and load balancing. We run the skyline query with $\lambda = 50K$ tuples/sec, $w = 1$ and $s = 0.2$ seconds ($L_p = 0.2$). We analyze the best performance achieved by the whole query parallelization with the `AS_PID` strategy for the PLQ and by comparing the use of the `WIN_RR` and `WIN_FB` strategies in the WLQ. Fig. 12 reports the performance achieved under different levels of burstiness, i.e. up to an index of dispersion of $\mathcal{I} = 8K$ which is a dramatic burstiness case [16]. We measure the ratio of the *offered bandwidth* (throughput) achieved by the parallelization over the (ideal) *required bandwidth* theoretically equal to $1/L_p$ windows per second. A value of the ratio close to one represents the optimal case where the actual throughput is similar to required bandwidth.

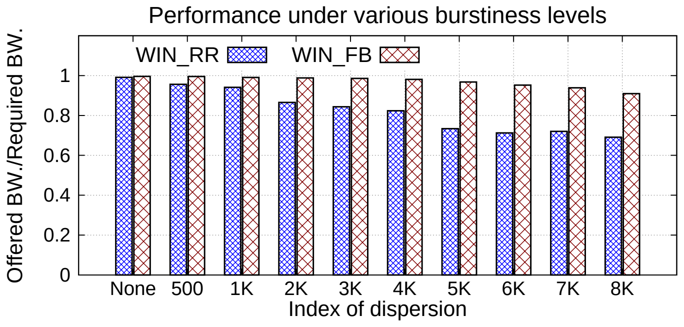
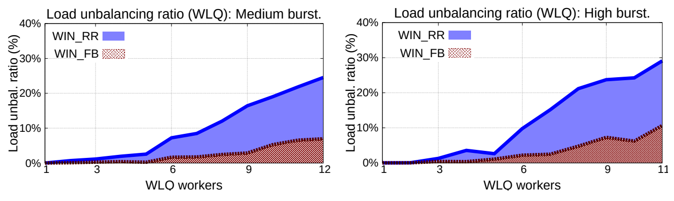


Fig. 12: Performance of the parallel skyline query with various burstiness levels.

In the `WIN_RR` case the ratio drops significantly with high burstiness and it is always lower than the one achieved by the `WIN_FB` strategy. The reason is that with the `WIN_RR` strategy it is possible that windows of bursty periods (so-called "heavy windows") are assigned to the same workers by impairing parallel efficiency and scalability. This is confirmed by the further experiments in Fig. 13, where we show the unbalancing factor by using a different number of WLQ workers in two cases, a medium-bursty case ($\mathcal{I} = 2.5K$, three workers in the PLQ) and a highly-bursty case ($\mathcal{I} = 6K$, four PLQ workers). With more than five WLQ workers the unbalancing factor with `WIN_RR` increases. Instead, the `WIN_FB` strategy always schedules tasks based on the actual availability of the workers to compute them and, hence, the load unbalancing factor remains significantly lower.



(a) Medium burstiness. (b) High burstiness.

Fig. 13: Load unbalancing ratio: comparison between the `WIN_FB` and `WIN_RR` scheduling strategies.

5. According to the FastFlow default setting, each thread is exclusively pinned onto an available core.

	Stream speed		*Out-of-ordering*			*Burstiness*	
	no. tuples	*arrival rate*	*no. late arrivals*	*avg. delay*	*max delay*	*index of dispersion*	*level*
Game1 (G1)	$7.12M$	$8,514$ tuples/sec	$5.94M$	34 ms	142 sec	$2,853$	severe
Game2 (G2)	$4.02M$	$10,037$ tuples/sec	$3.49M$	29 ms	171 sec	207	no bursts
Game3 (G3)	$12.64M$	$8,766$ tuples/sec	$10.67M$	26 ms	77 sec	$1,507$	moderate
Game4 (G4)	$7.67M$	$11,610$ tuples/sec	$6.66M$	24 ms	54 sec	155	no bursts
Game5 (G5)	$10.57M$	$18,850$ tuples/sec	$9.14M$	15 ms	50 sec	$1,845$	moderate
Game6 (G6)	$6.90M$	$18,462$ tuples/sec	$5.98M$	16 ms	71 sec	23	no bursts
Game7 (G7)	$24.58M$	$21,102$ tuples/sec	$21.64M$	20 ms	5.03 sec	89	no bursts
Game8 (G8)	$12.92M$	$23,071$ tuples/sec	$12.14M$	265 ms	5.18 sec	$3,062$	severe

TABLE 2: Characteristics of the real datasets: stream speed, out-of-order features and burstiness levels.

Very high burstiness levels also affect the `WIN_FB` strategy. In fact, we point out that all the tasks of the same window must be executed serially by the WLQ, thus the impact of burstiness cannot always be neutralized. With very high burstiness we experience a slight performance degradation (of 10% in the worst case). However, such dramatic levels of burstiness are often theoretical or likely characterize relatively short periods of the execution.

Pane-based vs. windowed approach. The advantage of the pane-based approach has been demonstrated in Ref. [15] for sliding-window aggregates. In this part we show that this approach is beneficial also for continuous preference queries. Fig. 14 shows the ratio of the execution time of the skyline query using panes over the execution time without panes (the lower the ratio the greater the benefit of the pane-based approach). This last is obtained by parallelizing the standard approach presented in Ref. [13], i.e. each tuple is scheduled to multiple workers that update in parallel the results of all the windows containing the tuple.

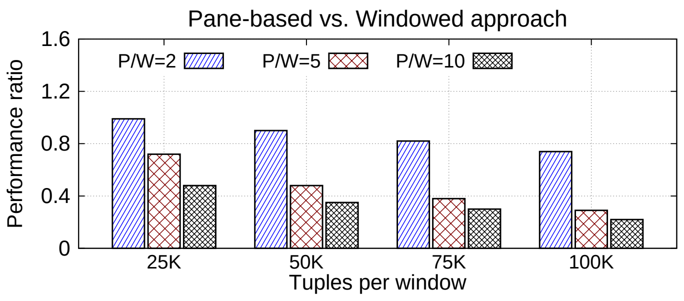


Fig. 14: Comparison between Paned vs. Windowed approach.

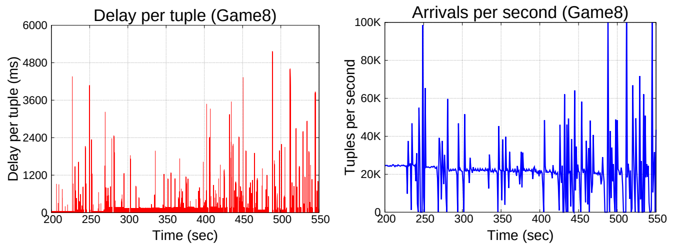
The pane-based approach reduces the query execution time. The PLQ filters out most of the tuples, and this reduces the total number of tuple comparisons. Instead, the windowed approach updates the result of each window that contains the tuple, and the higher the overlapping between windows (panes per windows, P/W) the greater the performance advantage in favor of the pane-based approach.

6 System Evaluation

In this final section we will evaluate the performance of our parallel query model on a set of real datasets exhibiting various disordering and burstiness characteristics.

6.1 Real Datasets

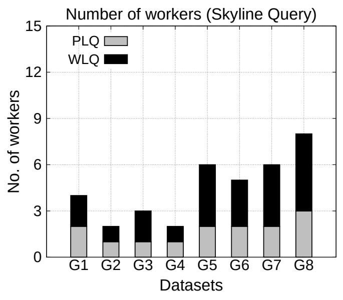
We use eight data streams obtained from the *Real-time Locating System* (RTLS) installed in the main soccer stadium in Nuremberg, Germany [34]. Sensor data (embedded in the shoes of the players and in the ball) are obtained during several training games with high data rate sensors tracking the positions and velocities of players (eight attributes per tuple). The characteristics of the streams are summarized in Tab. 2. Sensor data are gathered by the sink node of the sensor network where the query is supposed to be executed. The sink experiences some delay in receiving the readings from the sensors, which can be collected out-of-order. All the datasets have an average tuple delay of tens of milliseconds (with peaks of tens of seconds). Fig. 16a and Fig. 16b show the delay of the tuples and the arrivals per second in the second part of the execution of the `Game8` dataset, where this behavior is more evident.



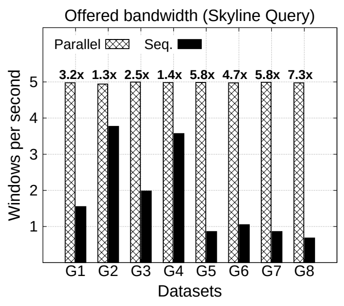
(a) Out-of-order pattern. (b) Arrivals per second.

Fig. 16: Out-of-order pattern and arrivals in `Game8`.

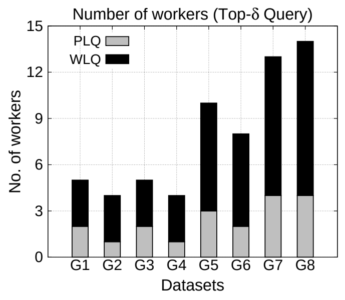
Differently from the synthetic datasets where the inter-arrival times are generated with a MAP(2) distribution and the tuple delays are uniformly distributed, in the real datasets the delays are non-uniformly distributed (like `Game8`, see Fig. 16a), and this may generate bursty arrivals to the sink node (see Fig. 16b). In fact, time periods characterized by bursty arrivals correspond to phases where the received tuples have a larger delay. Each dataset has a different level of burstiness depending on the temporal distribution of the tuple delays: the more uniformly distributed are the tuple delays the less is the index of dispersion. The index of dispersion can be estimated using an alternative definition, i.e. as the limit $\lim_{t\to\infty} Var(N_t)/E[N_t]$ where N_t is the number of arrivals during an interval of t time units. Based on this definition, to estimate the index of dispersion of an existing dataset we use the method proposed in Ref. [33], where the number of arrivals are measured for consecutive time intervals by increasing the interval length up to reach a given accuracy. The estimated indices of dispersion are shown in Tab. 2. Some datasets are characterized by low levels of burstiness while others like `Game1` and `Game8` have a severe burstiness.



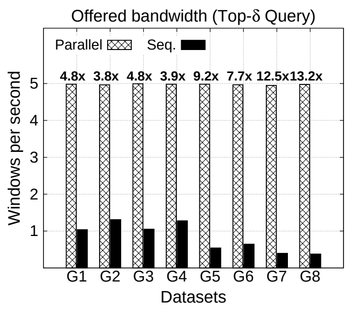
(a) No. of workers (Skyline Query).



(b) Offered bandwidth (Skyline Query).



(c) No. of workers (Top-δ Query).



(d) Offered bandwidth (Top-δ Query).

Fig. 15: Parallelism degrees of the PLQ and the WLQ and offered bandwidth with different preference queries.

6.2 Experiments

We study the execution of the skyline and the top-δ dominant queries. The real datasets are processed in real-time in order to extract complex analytics of the players performance that are notified to the team managers and visualized on mobile devices like a smartphone or a PDA. Typically, sensor readings are analyzed using a sliding window model with a refresh interval of several hundreds of milliseconds [8], [9]. For the sake of brevity, we execute both the queries in order to produce a result every 200 ms by processing the tuples received in the last second ($w = 1$, $s = 0.2$ and $L_p = 0.2$ seconds).

Parallelism degree and offered bandwidth. Figs. 15a and 15c show the number of workers needed by the PLQ and the WLQ stages to remove the bottleneck in each of the eight real datasets. For each stage we use the best scheduling approach, i.e. `AS_PID` for the PLQ, `WIN_FB` for the WLQ. As we can observe, the second stage needs more parallelism and the number of cores in the top-δ query is generally higher, owing to the higher processing time per tuple, as explained in Sect. 4.3. Furthermore, the last two datasets have a faster rate and thus they need more cores, while a slightly higher number of cores is usually necessary for datasets with a higher burstiness level.

Figs. 15b and 15d report the offered bandwidth by the parallel query with the number of workers described in Figs. 15a and 15c. In addition, we report the offered bandwidth by the single-threaded implementation of the pane-based approach, where the PLQ and WLQ stages are executed serially. The results confirm that the parallel implementation is able to reach the requested bandwidth of $1/L_p$ windows per second in all the datasets. Over each bar we report the *performance gain* that shows how much the offered bandwidth has been improved. It is computed as the ratio of the offered bandwidth by the parallel implementation over the one of the sequential query processing. The gain is always close to the whole number of utilized workers (PLQ+WLQ), demonstrating the effectiveness of our parallel query processing model in operating efficiently under real-world out-of-order and bursty streams.

Latency with different punctuation mechanisms. In this concluding part of the paper we show the latency of the parallel query processing. We define the latency of a window as done in Sect. 4.3 for the pane latency:

Definition 3 (Window latency). *The latency of a window is the time elapsed from the arrival of the first tuple of the window until the window result is finalized.*

Accordingly, the latency cannot be smaller than the window length and it may be actually higher depending on the disordering characteristics of the stream.

Once removed the bottleneck, the latency mainly depends on the punctuation mechanism. The basic K-slack mechanism introduced in Sect. 4.1.1 generates the punctuation that closes a pane when likely all the late arrivals have been received. To do that, the mechanism adapts to the maximum delay seen so far and this may produce very large latency results. Fig. 17 shows the latency of the skyline query using the number of cores shown in Fig. 15a. The latency obtained with K-slack is much more higher than the window length, and in general approaches the value of the maximum delay of the datasets reported in Tab. 2.

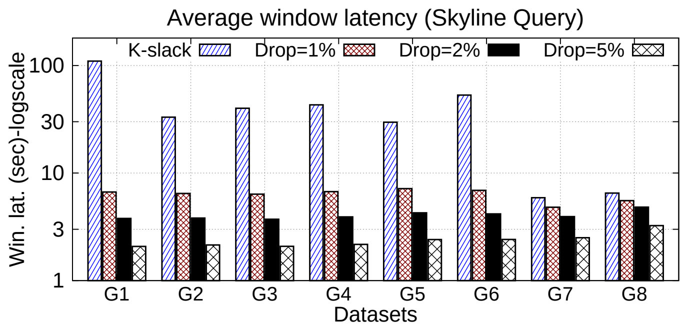


Fig. 17: Window latency with different punctuation mechanisms (Skyline Query).

Our parallel implementation can easily incorporate any punctuation mechanism in the PLQ emitter. To show the effect of different mechanisms, we repeated the experiments using an adaptive punctuation mechanism based on the AQ-K-slack presented in Ref. [8]. The idea is to anticipate the generation of punctuations so that the dropping ratio (of dropped tuples over the total number of received tuples) meets a user-defined value. Fig. 17 shows three ratios of 1%, 2% and 5%. As we can note, a low ratio (of 1%) allows the query to drastically reduce (of one order of magnitude) the average latency, and better results can be achieved with higher dropping ratios. In general, the acceptable level of dropping depends on the application requirements, and can be provided as a configuration parameter of the chosen punctuation mechanism.

7 Related Work

In this section we provide a review of some of the most representative papers covering topics similar to the ones studied in this work. We will highlight the overlapping and the differences with the approach presented in this paper.

Disorder handling approaches. The paper in Ref. [13] presents a framework for processing sliding-window aggregates over out-of-order streams. The authors introduce a technique called WID to map tuples to window extents. Tuples are processed on-the-fly and punctuation mechanisms determine when a window can be finalized. The approach does not explore parallel processing techniques and does not study the application to preference queries. An OOP architecture has been presented in Ref. [14]. The utilized punctuation mechanism (Gigascope heartbeats) assumes that all the input streams are ordered. The approach has been evaluated for traditional query operators (e.g., joins and union) executed in a sequential fashion, and shows the benefit in terms of latency reduction and memory saving compared with IOP frameworks. Almost all the existing papers do not explore the combined use of the OOP model with intra-operator parallel processing, as the operators are usually single-threaded. Intra-operator parallelism has been studied for sliding-window operators in Ref. [28], however under the assumption that all the tuples arrive in order.

The work in Ref. [35] focuses on the problem of dealing with late tuples that arrive after a window closes. Such tuples are usually dropped without being processed. Instead, the authors propose a partial processing technique. When the system receives a set of tuples older than the last emitted punctuation, it processes them by producing a partial result. Such late results are used to consolidate previously emitted results of closed windows. This is performed in a lazy fashion when these results are needed by the user. Although supporting parallel processing, this approach is not suited for real-time queries. A more suitable approach is the one proposed in Refs. [8], [9], where punctuations are emitted in a controllable way by regulating the dropping ratio. The idea is to relate the dropping ratio with the accuracy of query results. The approach is actually limited to simple sliding-window queries like sum, count or quantiles, and does not support intra-operator parallelism. The integration of such quality-driven approach to our parallel OOP architecture represents an interesting topic of our future research.

Handling load variations. A set of papers like the ones in Refs. [5], [6], [10] focus on elastic supports for data stream processing frameworks, in order to dynamically scale the used resources based on the actual/predicted incoming workload. Resource scaling usually needs complex state transfer protocols to migrate the query state without altering the computation semantics. As shown in Ref. [6], such protocols may generate latency spikes and transient throughput drops, and are suitable to handle long-/medium-term variations in the arrival rate in order to amortize the transient reconfiguration overhead with the performance benefit expected during the steady-state phase.

We have already studied elastic supports for stream processing in Ref. [6], and their integration is one of our next planned activities. Instead, this paper focused on scheduling strategies to handle bursty arrivals that cannot be handled satisfactorily with resource scaling only. In fact, bursty streams are characterized by traffic surges at short time-scales (e.g., few hundreds of milliseconds). As an example, the arrival pattern shown in Fig. 5b depicts a stationary process where the mean does not change over time. However, short-term bursty episodes characterize the arrival pattern throughout the execution. As demonstrated in Ref. [12], ignoring such fine-scale burstiness may provide an over-optimistic view of the elasticity behavior, which may turn out to be disastrous in practical scenarios. Although some existing papers [3]–[5], [10] cite burstiness as a critical issue in stream processing applications, they do not provide explicit solutions to cope with it. The recent work in Ref. [36] provides a formal model based on Petri nets for controlling which streams to admit for storage and analysis in the cloud. The goal is to maximize the revenue by respecting the desired QoS. The approach accounts for bursty streams, however the policy controls only which streams to admit for processing, while in our approach the streams must always be processed despite short-term traffic surges that may unbalance the load among cores.

8 Conclusions

Burstiness and out-of-orderness represent two of the major issues of real-world data streams. This paper coped with both the problems in the context of the parallel execution of preference queries. We proposed a parallelization of the pane-based approach which has been proven effective for sliding-window aggregates in the past. We studied sophisticated scheduling strategies that make use of interdisciplinary approaches from Control Theory. Furthermore, we allowed the out-of-order execution of input tuples through the use of punctuation mechanisms, and we stressed our implementation under various levels of burstiness injected in the synthetic datasets in a controllable way using a simple parameterization inspired by the work in Ref. [16].

The results showed the effectiveness of our approach in handling fast streams with high levels of burstiness and out-of-orderness. A wide set of real-world datasets have been used to assess the validity of our approach in real scenarios, confirming the results of the synthetic experiments.

Several future extensions of this work can be devised. Notably, the integration in our approach of elastic supports, like the ones that we presented in Ref. [6], is an interesting topic to fully provide our framework with mechanisms and strategies to deal with both bursty events at fine time-scales and long-/medium-term variations in the workload (trends, cyclic variations in the arrival rate).

Acknowledgments

We would like to thank Dr. Yuanzhen Ji, PhD (SAP, TU Dresden) for her valuable help in providing us the real datasets used for the final evaluation of our work.

This work has been partially supported by EU H2020-ICT-2014-1 project RePhrase (No. 644235).

References

[1] W. Choi, L. Liu, and B. Yu, "Multi-criteria decision making with skyline computation," in *2012 IEEE 13th International Conference on Information Reuse Integration (IRI)*, Aug 2012, pp. 316–323.

[2] L. H. U, N. Mamoulis, and K. Mouratidis, "Efficient evaluation of multiple preference queries," in *2009 IEEE 25th International Conference on Data Engineering*, March 2009, pp. 1251–1254.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/543613.543615

[4] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing*. Cambridge University Press, 2014, cambridge Books.

[5] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.295

[6] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 13:1–13:12. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851148

[7] Y. Drougas and V. Kalogeraki, "Accommodating bursts in distributed stream processing systems," in *Parallel Distrib. Proc., 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–11.

[8] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer, "Quality-driven processing of sliding window aggregates over out-of-order data streams," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. New York, NY, USA: ACM, 2015, pp. 68–79. [Online]. Available: http://doi.acm.org/10.1145/2675743.2771828

[9] Y. Ji, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Quality-driven disorder handling for concurrent windowed stream queries with shared operators," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '16. New York, NY, USA: ACM, 2016, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2933267.2933307

[10] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2012.24

[11] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611294

[12] S. Islam, S. Venugopal, and A. Liu, "Evaluating the impact of fine-scale burstiness on cloud elasticity," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 250–261. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806846

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 311–322. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066193

[14] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: A new architecture for high-performance stream systems," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 274–288, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453890

[15] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Rec.*, vol. 34, no. 1, pp. 39–44, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1058150.1058158

[16] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 149–158. [Online]. Available: http://doi.acm.org/10.1145/1555228.1555267

[17] M. Danelutto and M. Torquati, "Structured parallel programming with "core" fastflow," in *Central European Functional Programming School*, ser. LNCS, V. Zsók, Z. Horváth, and L. Csató, Eds. Springer, 2015, vol. 8606, pp. 29–75.

[18] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 11:1–11:58, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1391729.1391730

[19] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Data Engineering, 2001. Proceedings. 17th International Conference on*, 2001, pp. 421–430.

[20] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang, "Finding k-dominant skylines in high dimensional space," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 503–514. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142530

[21] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[22] Y. Tao and D. Papadias, "Maintaining sliding window skylines on data streams," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, no. 3, pp. 377–391, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2006.48

[23] M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos, "Continuous top-k dominating queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 5, pp. 840–853, May 2012.

[24] K. C.-C. Chang and S.-w. Hwang, "Minimal probing: Supporting expensive predicates for top-k queries," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002, pp. 346–357. [Online]. Available: http://doi.acm.org/10.1145/564691.564731

[25] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 662–673. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_65

[26] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, no. 3, pp. 555–568, Mar. 2003. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2003.1198390

[27] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 545–580, Sep. 2004. [Online]. Available: http://doi.acm.org/10.1145/1016028.1016032

[28] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13. New York, NY, USA: ACM, 2013, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2488222.2488258

[29] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[30] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of Queueing Theory*, 4th ed. New York, NY, USA: Wiley-Interscience, 2008.

[31] K. J. AAström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning*, 2nd ed. Instrument Society of America, Research Triangle Park, NC, 1995.

[32] K. J. Astrom and L. Rundqwist, "Integrator windup and how to avoid it," in *1989 American Control Conference*, June 1989, pp. 1693–1698.

[33] G. Casale, N. Mi, L. Cherkasova, and E. Smirni, "How to parameterize models with bursty workloads," *SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 2, pp. 38–44, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1453175.1453182

[34] C. Mutschler, H. Ziekow, and Z. Jerzak, "The DEBS 2013 grand challenge," 2013, dEBS, pages 289 - 294, 2013.

[35] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre, "Continuous analytics over discontinuous streams," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1081–1092. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807290

[36] R. Tolosana-Calasanz, J. A. Bañares, C. Pham, and O. F. Rana, "Resource management for bursty streams on multi-tenancy cloud environments," *Future Gener. Comput. Syst.*, vol. 55, no. C, pp. 444–459, Feb. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.future.2015.03.012

Gabriele Mencagli is an Assistant Professor at the Computer Science Department of the University of Pisa, Italy. He is co-author of more than 40 peer-reviewed papers appeared in international conferences, workshops and journals, and of one book. He is member of the Parallel Programming Models group at the same university. His research interests are in the area of parallel and distributed systems and data stream processing. He is also interested in studying interdisciplinary approaches for autonomic data stream processing applications and frameworks.

Massimo Torquati is an Assistant Professor at the Computer Science Department of the University of Pisa, Italy. He has published more than 60 peer-reviewed papers in conference proceedings and journals, mostly in the fields of parallel and distributed programming and runtime systems for HPC. His current research interests are on pattern-based parallel programming models, high-performance data stream processing, concurrent lock-free data structures and autonomic management in parallel systems. Currently, he is the main developer of the FastFlow parallel programming framework.

Marco Danelutto is a Full Professor at the Computer Science Department of the University of Pisa, Italy. His main research interests are in the field of parallel programming models, in particular in the area of parallel design patterns and algorithmic skeletons. He is author of more that 150 papers appearing in refereed international journals and conferences. He is the group leader of the Parallel Programming Model group at the same university, involved in a number of national and EU funded projects (CoreGRID, GRIDcomp, ParaPhrase, REPARA, RePhrase).

Tiziano De Matteis is a PostDoc at the University of Pisa, Department of Computer Science. During his PhD he studied the problem of providing parallel and autonomic supports to data stream processing applications. His principal research interests are related to parallel programming, autonomic computing and high performance data stream processing on homogeneous and heterogeneous architectures.