Real-Time Syst (2017) 53:1–44
DOI 10.1007/s11241-016-9257-0

Real-time processing of streaming big data

Ali A. Safaei[1]

Published online: 1 August 2016

Abstract In the era of data explosion, high volume of various data is generated rapidly at each moment of time; and if not processed, the profits of their latent information would be missed. This is the main current challenge of most enterprises and Internet mega-companies (also known as the *big data* problem). Big data is composed of three dimensions: *Volume*, *Variety*, and *Velocity*. The velocity refers to the high speed, both in data arrival rate (e.g., streaming data) and in data processing (i.e., real-time processing). In this paper, the velocity dimension of big data is concerned; so, real-time processing of streaming big data is addressed in detail. For each real-time system, to be fast is inevitable and a necessary condition (although it is not sufficient and some other concerns e.g., *real-time scheduling* must be issued, too). Fast processing is achieved by parallelism via the proposed *deadline-aware dispatching* method. For the other prerequisite of real-time processing (i.e., real-time scheduling of the tasks), a *hybrid clustering multiprocessor real-time scheduling algorithm* is proposed in which both the partitioning and global real-time scheduling approaches are employed to have better schedulablity and resource utilization, with a tolerable overhead. The other components required for real-time processing of streaming big data are also designed and proposed as real time streaming big data (*RT-SBD*) processing engine. Its prototype is implemented and experimentally evaluated and compared with the *Storm*, a well-known real-time streaming big data processing engine. Experimental results show that the proposed RT-SBD significantly outperforms the Storm engine in terms of proportional deadline miss ratio, tuple latency and system throughput.

✉ Ali A. Safaei
aa.safaei@modares.ac.ir

[1] Department of Medical Informatics, Faculty of Medical Sciences, Tarbiat Modares University, Tehran, Iran

Keywords Streaming big data · Hybrid multiprocessor real-time scheduling · Clustering · Deadline-aware dispatching · Periodic continuous queries

1 Introduction

In many recent data-intensive applications, data volume and complexity are increasing fast. Search engines, social networks, e-science (e.g., genomics, meteorology and healthcare), financial (e.g., banking and mega-stores) are some examples of such applications. This, which is the major challenge of many enterprises and Internet-scale mega-companies is known as the *Big Data* problem. While volume is a significant challenge in managing big data, the focus must be on all of the dimensions of such data sets which are *Volume*, *Variety* and *Velocity* (also known as *3Vs)* (Chen Philip and Zhang 2014).

- *Volume*—by which we mean high volume of data (i.e., increase in data volumes and also a massive analysis)
- *Variety*—or in fact, high variety of data (many sources and types of data)
- *Velocity*—by which we mean high velocity (both how fast data is being produced and how fast the data must be processed to meet demand)

The *Volume* dimension, despite that mentions an ancient problem, but the scale has been changed nowadays (e.g., petabytes of new data each day). Also, the *Variety* dimension is focused on the semi-structured and unstructured data in new applications. *NoSQL* data models (e.g., *key-value, columnar, document-based, and graph*) and systems (e.g., *MongoDB, neo4j, etc.*) are provided to handle this kind of challenges.

But the *Velocity* dimension, which implicitly affects the volume dimension, consists of both *high speed of data arrival* rate, and the need for *high speed in data processing*. The most well-known instance of high-speed data is "*streaming data*", and high speed in processing (in this context) means "*real-time processing*". As an example, *eBay* addressing fraud from *PayPal* usage by analyzing real-time 5 million transactions each day (Lehner and Sattler 2013). Regards to the importance of the velocity dimension, this paper is concentrated on it (i.e., *real-time processing of streaming big data).*

1.1 Motivation

Streaming big data has some remarkable characteristics. In data stream systems, data is received as continuous, infinite, rapid, bursty, unpredictable and time-varying sequence. *Monitoring (e.g., network traffic, sensor networks, healthcare, etc.), surveillance, web-clicks stream, financial transactions, fraud and intrusion detection* are some applications of streaming big data. In most of these applications, QoS requirements (e.g., response time, memory usage, throughput, etc.) are extremely important; and time-critical processing can also be generalized to QoS requirements. In other words, most of streaming big data applications have real-time requirements (i.e., *deadline*) (Babcock et al. 2003). So, real-time processing of streaming big data (i.e., Velocity dimension of the big data problem), which is addressed in this paper is needed in many current real-world applications.

1.2 Challenges

In Stankovic etal. (1999) eight misconceptions in real-time data management are discussed. One of the most common and important misconceptions is: "*real-time computing is equivalent to fast computing.*" In fact, fast processing does NOT guarantee time constraints. In other words, although being fast is necessary but is not sufficient. For a real-time system, there is a need for other mechanisms (real-time scheduling, feedback control, etc.) to handle and satisfy time constraints.

Stonebraker et. al. (2005) introduced eight requirements for real-time processing of data streams. Two key requirements are "*fast operation*" and "*automatic and transparent distribution of processing over multiple processors and machines*". The requirements raise from the fact that single processor data stream processing systems are not capable of processing huge volume of input streams and cannot execute query operators continuously over them with satisfactory speed (Johnson et al. 2008; Safaei and Haghjoo 2010). So, solutions such as *parallel processing* over multiple processors have to be used for this bottleneck to reach the required processing speed for real-time processing of streaming big data.

Parallel processing of streaming data is studied in many works, mostly based on the *MapReduce* computational model (which is essentially good for batch processing) (Condie et al. 2010). We proposed parallel processing model of data stream queries in Safaei and Haghjoo (2010) and improved it in Safaei and Haghjoo (2012), by employing *Dispatching* instead of its event-driven operator scheduling. Also, we have discussed system architecture, practical challenges and issues for the underlying parallel system, as well as its implementation on multi-core processors in Safaei et al. (2012). Moreover, a proposed MapReduce based framework for parallel processing of data stream is presented in Safaei and Haghjoo (2014).

In this paper, fast operation necessary for real-time processing of streaming big data is achieved via employing our parallel query processing method presented in Safaei and Haghjoo (2010, 2012) (which is reviewed briefly in Sect. 3), and other required mechanisms are added as presented in Sect. 3.

One of the major preliminaries for employing parallelism is in possession of multiple processing elements such as cores in a multi/many –core CPUs, GPUs besides CPUs, and clusters of commodity machines, or even using the *Cloud* infrastructure. Besides providing required computing resources and performing parallel processing (to achieve the required processing speed for real-time processing of streaming big data as stated in Sect. 3), the most critical challenge is how to assign resources to the requests with respect to their deadlines (also known as *real-time scheduling*). Various real-time scheduling algorithms exist for single and multi-processor systems (Andersson and Jonsson 2003; Anderson and Srinivasan 2000; Bans et al. 2002). Optimal single processor real-time scheduling algorithms such as EDF[1] and RM[2] are not optimal for multiprocessor real-time systems (Dhall and Liu 1978; Carpenter et al. 2004). Since our parallel query processing method proposed in Safaei and Haghjoo (2010;

[1] Earliest deadline first.

[2] Rate monotonic.

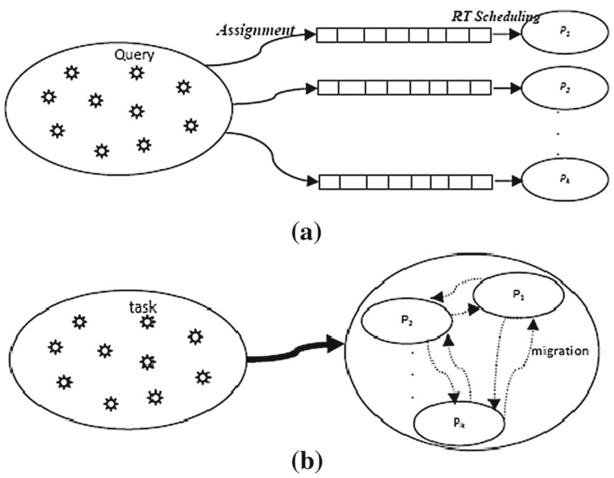


Fig. 1 Multiprocessor real-time scheduling approaches. **a** The partitioning approach. **b** The global approach

2012 and Safaei et al. 2012 is based on multiprocessing environment, here we employ multiprocessor real-time scheduling.

There are three approaches for real-time scheduling in multiprocessor systems: *Partitioning, Global* and *Hybrid* scheduling. In the partitioning approach, each processor has its own task waiting queue. The set of tasks is partitioned and each task is assigned to the proper processor (task waiting queue) according to the *allocation* algorithm. Each processor executes tasks in its task waiting queue according to its *real-time scheduling policy* (Fig. 1a). In the global approach, each task can be executed over all processors. In fact, a task which is started in a processor can *migrate* to any other processor to be continued (Fig. 1b) (Holman and Anderson 2006). Generally, online real-time scheduling in multiprocessor systems is a NP-hard problem (Carpenter et al. 2004).

The partitioning approach may not be optimal but is pragmatically suitable because: (1) Independent real-time scheduling policies can be employed for each task queue. Therefore, the multiprocessor real-time scheduling problem is simplified to single processor real-time scheduling. (2) As each task only runs on a single processor, there is no penalty in terms of migration cost; so, it has low run-time overhead which helps for better performance (Holman and Anderson 2006). (3) If a task overruns its worst-case execution time, then it can only affect other tasks on the same processor (Safaei and Haghjoo 2010).

The global approach has the ability to provide optimal scheduling due the migration capability, as well as spare capacity created when a task executes for less than its worst-case execution time can be utilized by all other tasks, not just those on the same processor. One the drawbacks of the Global approach is its considerable overhead. Furthermore, to have the optimal schedule, some preconditions must be held which is not possible in all applications (Holman and Anderson 2006). Generally, Global scheduling is more appropriate for open systems, as there is no need to run load balancing/task allocation algorithms when the set of tasks changes (Safaei and Haghjoo 2010).

To achieve the benefits of these two multiprocessor real-time scheduling approaches together, different *Hybrid* approaches have been proposed by researchers (Safaei and Haghjoo 2010). For example, *EKG* (Andersson and Tovar 2006), *Ehd2-SIP* (Kato and Yamasaki 2007), *EDDP* (Kato and Yamasaki 2008), *PDMS-HPTS* (Lakshmanan et al. 2009), *HMRTSA* (Srinivasan and Anderson 2004), *PFGN* (Safaei et al. 2011) and *PDMRTS* (Alemi et al. 2011) use *semi-partitioning* Hybrid approach which aims at addressing the fragmentation of spare capacity in partitioning approach to split a small number of tasks between processors (Safaei and Haghjoo 2010).

Another well-known Hybrid multiprocessor real-time scheduling approach is *Clustering* (Safaei and Haghjoo 2010). *Hybrid clustering* approach can be thought of as a generalized form of partitioning with the clusters effectively forming a smaller number of faster processors to which tasks are allocated. Thus capacity fragmentation is less of an issue than with partitioned approaches, while the small number of processors in each cluster reduces global queue length and has the potential to reduce migration overheads, depending on the particular hardware architecture. For example, processors in a cluster may share the same cache, reducing the penalty in terms of increased worst-case execution time, of allowing tasks to migrate from one processor to another (Safaei and Haghjoo 2010). It should be noted again that, the partitioning approach can be considered as a special form of the hybrid clustering approach (i.e., each cluster contains only one processor).

In this paper, processing of real-time continuous queries over streaming big data is issued by using hybrid clustering multiprocessor real-time scheduling, and parallel processing of query in each cluster of cores in multi-core CPU.

We have proposed parallel processing of a continuous query over processing elements (e.g., cores in a multi-core CPU) in Safaei and Haghjoo (2010), and improved the method in Safaei and Haghjoo (2012) by performing the scheduling continuously and dynamically (called *Dispatching* method, instead of the *event-driven* one in Safaei and Haghjoo (2010). Practical dimensions of our parallelism method over multi-core processors are presented in Safaei et al. (2012) (as is reviewed briefly in Sect. 3). Relying on the proposed parallel processing of a continuous query over a multi-core CPU (Safaei and Haghjoo 2010, 2012; Safaei et al. 2012), and using the hybrid clustering approach, in this paper, we have presented a prototype for real-time processing of continuous queries over streaming big data using multi-core processors (called *RT-SBD*[3] proceeding engine).

In other words, the primary focus of this paper is the velocity dimension of the big data problem which by the definition, regards to the real-time processing of streaming big data. A major prerequisite for real-time processing is to be fast and the parallel processing and dispatching method we have presented in previous papers are employed for this aim; But the proposed solution is achieved by some contributions that are designed to solve the problem objectively. Some of the most important contributions of this paper are:

- *Deadline-aware dispatching method* as the parallel processing method to provide the required fast processing, necessary to be real-time.

[3] Real time-streaming big data.

- *Hybrid clustering multiprocessor real-time scheduling algorithm* as the other prerequisite for real-time processing
- Proportional deadline miss ratio (*PDMR*) instead of the traditional DMR, as the most important metric for evaluation of real-time (stream) processing systems.
- Also, the prototype of the proposed real-time streaming big data processing engine is developed.

1.3 Structure of the paper

The paper is continued as follows: Real-time query processing in data stream management systems is formally defined in Sect. 2. The proposed real-time streaming big data processing engine (*RT-SBD*) is presented in Sect. 3, while some of its important properties are analyzed through this section. Details of simulation results and performance evaluation for the presented system is discussed in Sect. 4. In Sect. 5, related work is presented. Finally, we conclude in Sect. 6.

2 Problem formulation

In this section, real-time query processing in data stream systems is formalized.

2.1 Data model

A data stream is a continuous, infinite, rapid, bursty, unpredictable and time-varying sequence of data elements denoted as $S = <s_1, s_2, \ldots>$. For each data element, its arrival time to the system is appended.

Definition 1 *Stream* Let Φ denote set of data tuples and $\mathcal{T}$ denote discrete time domain. A stream is denoted as $S = (B, \leq_t)$ in which B is an infinite sequence of tuples in form of (τ, ts), where $\tau \in \Phi$ is a data tuple and ts $\in \mathcal{T}$ is its timestamp, and $\leq_t$ is a total order over B.

In this paper, we assume that deadlines are only assigned to queries (not to data streams).

2.2 Query model

In data stream systems, queries are mainly continuous or one-time. In real-time data stream systems, tasks are queries and queries are categorized as *periodic* and *aperiodic*. Periodic queries are modeled as *PQuery* in Wei et al. (2006a). In this model, for each registered query, instances are activated with a specified period and each instance must provide its results on a window including ω tuples within the determined deadline.

Aperiodic queries include continuous and one-time queries. In the former, instances are activated by arrival of each input tuple whilst in the latter are activated and executed only once. One–time query is considered as a periodic query with period value of ∞

(Li and Wang 2007) but in this case it should be resident in the system forever which imposes overhead to the system. Hence, in this paper we model one-time query in a different form. Set of queries in real-time system presented in this paper consists of combination of periodic (*PQuery*) and aperiodic (continuous and one-time) queries. Properties of each query type are:

- Periodic query: resident in the system, consisting of instances, each instance activated at a given time, each instance executed over a window of tuples.
- Continuous query: resident in the system, consisting of instances, each instance activated by arrival of the first tuple, each instance executed on one tuple.
- One-time query: not resident in the system, only one instance, activated by the first tuple arrival, executed over a window of tuples.

In fact, what is generated in the system and executed over tuples is the instance of query. So, we model query instance instead of the query itself.

According to the properties of query types, we define query instance below:

Definition 2 *Query instance* Query instance in RT-SBD is modeled as a sixtuple as $q = <i, j, D, T, \omega, p>$ in which: q is the *query instance*, i is the *query number*, j is the *instance number*, D is the *deadline*, T is the *period*, ω is the *window size*, p is the *priority*

$where\, i, j, \omega, p \in N^+ \, and \, D, T \in \mathcal{T}.$

Notice that for continuous query instances we have $T = 0$ and $\omega = 1$, and $T = \infty$ is used to indicate one-time query type.

A query (instance) which determines operators and their arrangements can be depicted as query-plan graph. In this paper, we use notations introduced in Safaei and Haghjoo (2010) for query plan. We also assume the quality of the queries can be traded off for timeliness by dropping some of their input tuples.

2.3 Real-time data stream system model

In order to clarify the problem, here we describe the real-time data stream system model issued in this paper.

(a) *Hard, firm or soft real-time*
Applications in which *RT-SBD* is employed determine type of real-time system. In database context, real-time systems are generally soft because it is often impossible to determine query duration accurately before execution (Babcock et al. 2003; Stankovic et al. 1999; Stonebraker et al. 2005; Johnson et al. 2008). On the other hand, in most common data stream real-time applications such as monitoring (health, network, etc.) and financial applications (stock exchange, etc.), results provided after the deadline are not profitable. As a result, the real-time streaming big data processing system presented in this paper is considered as *firm* real-time. Hence, we can define the *value* of each query instance as:

Definition 3 *Query instance value* For each query instance with deadline d_i and finishing time f_i, its value is:

$$v_i = \begin{cases} v_{max} & f_i \leq d_i \\ 0 & f_i > d_i \end{cases}$$

where v_{max} is the maximum value achieved by finishing without tardiness.

(b) *Release of tasks*
According to the *RT-SBD* query model, queries are released both periodically and aperiodically.

(c) *Dependency between tasks*
Queries are independent of each other. Multiple query processing and optimization is left for future work.

(d) *Priority assignment*
Priority assignment is dynamic, i.e., priority of query instances may change during runtime.

(e) *Preemption*
Preemption of query instances is allowed i.e., a higher priority query may postpone lower priority executing query.

(f) *Static or dynamic scheduling*
In order to be compatible with continuous and dynamic nature of data stream system (streams, queries and system conditions), scheduling is performed dynamically and based on system circumstances.

(g) *Open or close loop scheduling*
Since the whole system information about streams, queries, etc. is not available when scheduling begins, feedback control approach (close loop scheduling) is applied.

(h) *Single-processor or multiple-processors real-time scheduling*
Due to the type of parallelism in *RT-SBD*, multiprocessor real-time scheduling (hybrid clustering approach) is used.

3 The proposed real-time streaming big data processing engine

Although being fast is necessary for real-time systems but, to be fast does not mean to be real-time. Real-time system designers employ necessary mechanisms in the system architecture to support real-time scheduling and deadlines. The proposed architecture of *RT-SBD* (see footnote 3) processing engine is depicted in Fig. 2.

User queries and their characteristics (i.e., query type, deadline and period) are delivered to the *request manger* unit. It accepts queries with valid deadline, registers and sends it to the *scheduler* as well as the *deadline monitor* unit. The *scheduler* unit executes queries via the *parallel query processing engine* (which process each query in parallel over a multiprocessing environment) with respect to its deadline. The *deadline monitor* unit monitors system outputs continuously and restores the system to an acceptable status when necessary (e.g., when DMR threshold violation occurs).

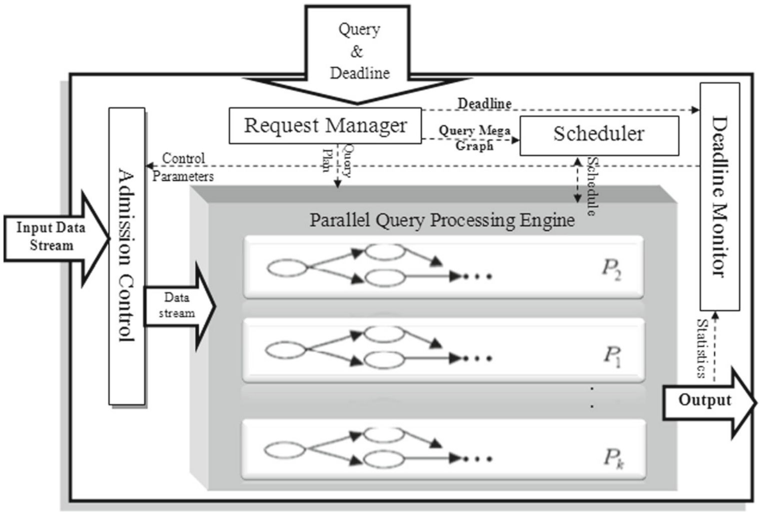


Fig. 2 Architecture of the proposed real-time streaming big data processing system (*RT-SBD*)

To do this, the *deadline monitor* unit sets the *data admission control* unit parameters in a manner that input data stream rate (i.e., system workload) degrades.

3.1 Request manager

Users deliver queries together with their characteristics, including query type (periodic, continuous or one-time), deadline and period (if necessary) to the request manger unit. The request manager unit determines whether the query can be processed with respect to its deadline or not; If so, it is accepted, registered, and its query plan is generated and sent to scheduler as well as deadline monitor unit (along with its deadline). Otherwise, the query is rejected.

In order to determine whether a query can be processed before its deadline or not, we define and implement a function to compute system response time when a new query is added to the ones resident in the system, and compare its result with the determined deadline.

Definition 4 *Query response time* in RT-SBD is the result of function response(q,ss,t) in which q is the query and ss is the system status at time t.

Note Exact definition and implementation of *response(q,ss,t)* is presented in Mohammadi (2010).

3.2 Parallel query processing engine

To achieve fast operation in *RT-SBD* we employ parallel query processing in a multi-processing environment as presented in our previous publications (Safaei and Haghjoo

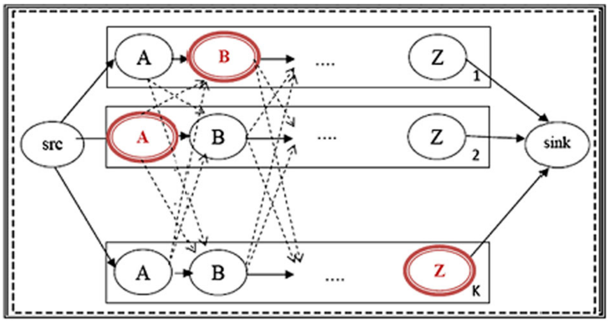


Fig. 3 Dispatching in parallel query processing engine unit

2010, 2012). Parallel query processing engine of *RT-SBD* contains ***k*** parallel logical machines that execute the query plan in parallel by using dynamic and continuous operator scheduling (i.e., dispatching). At first, a copy of the query plan is assigned to each machine. In this way, the query plan is recognized by all machines and they become capable of collaborating for parallel execution of operators. In this collaboration, if in the query plan, an operator ***A*** sends its output tuples to operator ***B***, then in each particular machine P_i , operator ***A*** is capable of sending its output tuples to ***B*** of all machines (Fig. 3) (Safaei and Haghjoo 2010). In initial operator scheduling, operators of a query plan are assigned to logical machines according to Eq. (1):

$$j = i \; \boldsymbol{mod} \; k \tag{1}$$

where *j* is the machine *Id*, *i* is the operator *Id*, *k* is the number of logical machines.

Each assigned operator should process its input tuples and then forward it to the next selected operator (bold circles in Fig. 3), to continue operation on the tuple as a pipeline. Selection of the next operator (machine) is done by computing workload of machines (according to Eq. (2)[4]) and selecting the minimum workload machine (Safaei and Haghjoo 2012). This is called the *Dispatching* of the (processed) data stream tuples.

$$\begin{aligned} \forall (a, b) \in E' \therefore a = O_f^i \wedge b \\ = O_h^{i+1} \left(\left(w(a, b) \leftarrow \sum_{l=1}^{k} \left(q\_count\left(O_l^-, O_h^-\right) \times e_{o_h^-}\right)\right)\right) \end{aligned} \tag{2}$$

So, in dispatching process, each operator (machine) *continuously* and *dynamically* performs these tasks (Safaei and Haghjoo 2012):

(a) processing of an input tuple,
(b) computing destination machines' weight and selecting the minimum weight machine as the next operator (machine),

[4] For each edge (a,b) from an operator to its immediate subsequent in the graph, the weight is processing time of all operators running on the destination operator's machine (i.e., aggregation of number of tuples waiting in input queue of each operator multiplied by the corresponding operator's execution time).

(c) forwarding the processed tuple to the selected machine in order to continue processing of the query.

We proved in Safaei and Haghjoo (2012) that this dynamic and continuous operator scheduling (i.e., *Dispatching*) provides minimum tuple latency. It means that, the parallel query processing engine of *RT-SBD* provides fast operation necessary for our real-time streaming big data processing engine. In addition, dispatching leads to degradation of fluctuations in system performance parameters (i.e., tuple latency, memory usage, throughput and tuple loss) (Safaei and Haghjoo 2012). This feature provides high adaptivity of the system against bursty nature of data streams.

Also, we have shown in Safaei and Haghjoo (2012) and Safaei et al. (2012) that, using two logical machines that collaborate with each other via the proposed *Dispatching* method can provide parallel processing of a continuous query efficiently. So, a cluster of two logical machines performing the dispatching method is considered as a processing element (called *Disp-2 engine*). Of course, this setting is used as the initial configuration of the proposed system; and the proper number of processing elements (processors or machines) as well as optimal number of processors in each cluster of the hybrid clustering multiprocessor real-time scheduling will be determined via experimental evaluations.

3.3 Scheduler

The used algorithm for real-time scheduling of queries is a *hybrid clustering multiprocessor real-time scheduling* proposed and introduced below.

By *Hybrid*, we mean that both partitioning and global scheduling approaches are used as a mixed (hybrid) approach. This is to reduce the overheads (by using the partitioning approach) as well as to have better utilization of processors (by using the global approach and migration), to process more tasks (i.e., processing tuples).

Also, by *Clustering* we mean that set of processors is clustered such that each cluster acts as a more powerful processing element (or processor). Determination of the proper number of processors in each cluster is discussed later. As mentioned in Sect. 3.2, processors in a cluster use the proposed dispatching method for (parallel) processing of the assigned task (query).

So, the proposed hybrid clustering multiprocessor real-time scheduling algorithm consists of two major levels: selecting the highest priority query (task) in set of queries to be scheduled and allocating it to the proper cluster of processors (based on the partitioning approach), and then, parallel processing of the allocated query by processors in the cluster via the dispatching method (which preforms migrations as in the global approach).

Detailed descriptions of these two levels are as follows.

(a) *Allocation*

In this phase, based on the partition approach, set of the queries (tasks) is partitioned into subsets of queries which each subset assigned to a particular cluster of processors (Fig. 4). Generally, the partitioning approach for multiprocessor real-time scheduling consists of two phases: the *allocation* phase that allocates a task among tasks set to the waiting queue of a desired processor and the *real-time* *scheduling* phase that selects the task with the highest priority for each processor, among tasks in its waiting queue (with respect to its real-time scheduling policy) to be processed over the corresponding processor. In real-time query scheduling of *RT-SBD*, each processor in the partitioning approach is substituted by a parallel query processing engine presented in Safaei and Haghjoo (2012) (i.e., a cluster of two logical machines that process each allocated query in parallel using dispatching method). In other words, the set of $\boldsymbol{k}$ logical machines in the parallel query processing unit of the *RT-SBD* architecture is partitioned into ***k/2*** sets or clusters. Each one (a *Disp-2* engine) contains two logical machines which process an assigned query in parallel by our dispatching method introduced in Safaei and Haghjoo (2012). So, from the scheduler unit point of view, each *Disp-2* engine is transparently a processing unit (a cluster of processors or cores of a multi-core CPU). In the allocation phase, queries are allocated to waiting queue of each *Disp-2* engine. Then, in the real-time scheduling phase, for each *Disp-2* engine, the highest priority query is selected to be processed over its relevant *Disp-2* engine (Fig. 4). Scheduling and executing of the selected query over a *Disp-2* engine is performed by the *Disp-2* engine itself as discussed in Safaei and Haghjoo (2012).

The scheduler unit of *RT-SBD* uses the *First-Fit* algorithm (Carpenter et al. 2004) in its allocation phase to allocate a query (instance) to waiting queue of *Disp-2* engines. Utilization factor of each query (i.e.,$\boldsymbol{e_i/T_i}$) is computed and compared with utilization factor of each of *Disp-2* engines. Query is allocated to the first *Disp-2* engine with query utilization factor not greater than its utilization factor. Also, in the real-time scheduling phase, the *EDF* real-time scheduling policy is used for waiting queue of all *Disp-2* engines.

Note execution time of each query (which is needed for computing utilization factor) is taken from the request manger unit. Since it computes query execution time via the *response(q,ss,t)*, it is not necessary for user to determine query execution time.

(b) *Deadline-aware dispatching*

At this level, in each cluster, the task (query) which is assigned to that cluster is executed (processed) on cluster's processors. Each cluster, as a *Dispatching-engine* uses the proposed dispatching method for (parallel) processing of the assigned query. Through the dispatching process, a tuple is processed by a processor and then forwarded to the best next processor to continue the processing of the tuple. Therefore, dispatching leads to the distribution of the steps of processing a query, over the processors in the cluster (i.e., *Migration*). Of course, in the global multiprocessor real-time scheduling approach, task is not decomposed for migration but instead, the time required for completion of the task is decomposed and shared by different processors.

In other words, in the classic task migration, the task as a whole, follows up its remained execution in another processor each time. But, in context of query processing over data stream, the proposed dispatching will result in migration; regards to the continues nature of the streams and queries, and with respect to this fact that a job of a task (which should migrate) is composed of operators and

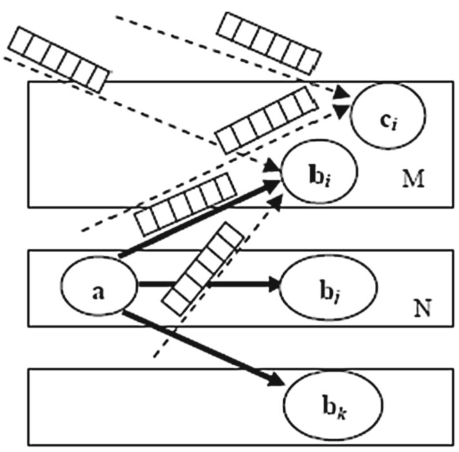


Fig. 4 Edges to next operators located in machines (workload of machine ***M*** when running some operators concurrently must be taken into account in weight updating by operator ***a***)

data, a job in processing a continues query can be the execution of an operator of the query on a data stream tuple.

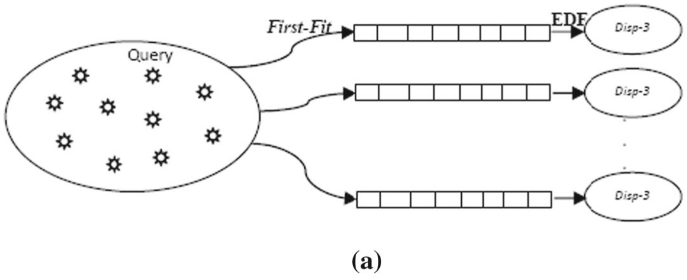
Accordingly, a job of a task (i.e., execution of an operator of the query on a data stream tuple) is migrated among different processors in the cluster, while dispatching method is used for (parallel) processing of the assigned query by the cluster's processors. Therefore, what is performed by the processors in a cluster (i.e., a *Disp-engine*) for (parallel) processing of the assigned query, will result in migration of the jobs in the scheduled task.

But, in the dispatching method proposed previously in Safaei and Haghjoo (2012) (also discussed in Sect. 3.2 and shown in Fig. 4), processing costs of the destination processor is the only criteria used for migration (which does not take jobs' deadline into the account). In order to resolve this defect, a deadline-aware dispatching method should be substituted that consider job's deadline when selecting the next processor to forward the processed tuple. In other words, in the deadline aware dispatching method, each processor in the cluster *continuously* and *dynamically* performs these tasks:

(a) processing of an input tuple,
(b) computing destination processors' weight, and selecting the best next processor based on the used forwarding policy,
(c) forwarding the processed tuple to the selected machine in order to continue processing of the query.

Computing the processing costs (weight) of the destination processors is done according to the Eq. (3).

$$\forall (a,b) \in E' \therefore a = O_f^i \wedge b = O_h^{i+1} \left(\left(w(a,b) \leftarrow \sum\nolimits_{l=1}^{k} \left(q\_count \left(O_l^-, O_h^- \right) \times e_{o_h^-} \right) \right) \right) \tag{3}$$



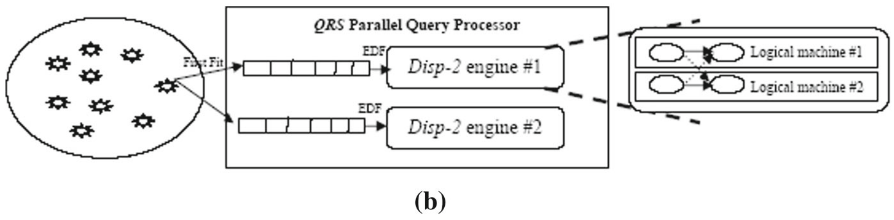


Fig. 5 Levels of the proposed hybrid clustering multiprocessor real-time scheduling. **a** *Allocation* level, **b** *deadline-aware dispatching* level

The *forwarding policy* must check that job's deadline will not be missed. Also, it is better to select the *best* next processor. Checking for meeting the deadline is performed when the predicate $w(a, b) + e_{t,k} < d_k$ (in which $e_{t,k}$ denotes execution time of tuple t of task k while $d_k$stands for task's deadline), holds for the next processor. If there is no such a processor, it means that the deadline may be missed; in this case, the tuple can be discarded or forwarded anyway (in case of firm or soft real-time systems, respectively).

There are some alternatives for selecting the *best* next processor; the best processor here may mean the fastest one (i.e., the processor with the minimum processing costs). Finding such processor is costly and similar to the worst-fit algorithm (with respect to processors' capacity). So, since the best-fit and first-fit (while are near identical) are generally better alternatives, it is more efficient to use the first-fit algorithm. Hence, the preferred forwarding policy can be such as this: *select the first next processor in which the predicate* $w(a,b)+e_{t,k} < d_k$ *holds*. The other alternatives are also studied and compared in the experimental evaluation of the proposed system (Sect. 4).

So, as stated, scheduling in *RT-SBD* consists of two levels: allocation of queries to the proper cluster, and deadline-aware dispatching of the allocated query; at the allocation level (using the portioning approach), queries are partitioned via the first-fit algorithm and assigned to the proper Disp-2 engine cluster. Each cluster selects from its waiting queue of queries based on the EDF algorithm. The selected query is processed by processors of that cluster via the proposed deadline-aware dispatching method (migration of jobs of the task, as in the global approach). The two levels of the proposed hybrid clustering multiprocessor real-time scheduling is depicted in Fig. 5.

To schedule aperiodic queries (continuous and one-time), a virtual query instance named *server* is created. Its execution time is dedicated to aperiodic query instances

(Kato and Yamasaki 2008). A virtual deadline is computed according to Eq. (4) and assigned to each aperiodic query instance (Li and Wang 2007; Kato and Yamasaki 2008).

$$vd\,(i) = \max\,(a\,(i)\,,\ vd\,(i-1)) + \frac{e\,(i)}{U_A} \tag{4}$$

$vd_{(i)}$ is virtual deadline for instance i, $a\,(i)$ is its arrival time, $e_{(i)}$ execution time, and U_A is utilization factor for aperiodic queries (Li and Wang 2007). So, aperiodic query instances and their virtual deadlines behave as periodic query instances and their deadlines.

After determining virtual deadline for aperiodic query instances, query instances are allocated to the waiting queue of relevant cluster (*Disp*-engine) based on the First-Fit algorithm. Then, the scheduler unit selects the highest priority query instance among its waiting queue with respect to their deadline or virtual deadline. A cluster processes each selected query instance in parallel using the proposed deadline-aware dispatching method.

3.3.1 Analysis

The goal of analyzing a real-time scheduling algorithm is determining its essential characteristics and analyzing its functionality in scheduling of tasks in the task system. The most important characteristics of a real-time scheduling algorithm are introduced below and the proposed hybrid clustering real-time scheduling algorithm is analyzed with respect to them.

In the real-time scheduling literature, a task system is *correct* if the release and eligibility times of all of its jobs are specified, and is *feasible* if there exist a schedule in which no job deadline is missed (Leontyev 2010). A hard real-time task system is called *schedulable* under a scheduling algorithm on a given platform if no deadline is missed, while for soft real-time task system, if the maximum task tardiness is bounded (Carpenter et al. 2004). For analyzing a real-time scheduling algorithm, we must consider task systems' utilization bound. Upper and lower bonds of utilization for different classes of real-time scheduling algorithms are categorized based on two important dimensions, *degree of migration allowed* and *the complexity of the priority scheme* (Table 1) (Carpenter et al. 2004).

Since the proposed real-time scheduling algorithm is a hybrid algorithm using two levels with two distinct approaches, its analysis is divided into two levels: analyzing allocation level and then deadline-aware dispatching level.

Note that, in both of the levels, the task system $q = <i, j, D, T, \omega, p>$ has an identical priority mechanism which is job level dynamic; while has different migration degree: at the allocation level (based on partitioning approach) has the degree of *no migration* (*i.e., partitioned*), and at the deadline-aware dispatching level (based on global approach) has the degree of *restricted migration*. (Migration inside the corresponding cluster at the operator's boundary).

Accordingly, it is in class of (2-1) (i.e., *job level dynamic, partitioning*) at its first level (with utilization bound of $U\,(\tau) \le \frac{(M+1)}{2}$) while is in class of (2-2) (i.e.,

Table 1 Known bounds on worst case achievable utilization (denoted *U*) for the different classes of scheduling algorithms (Carpenter et al. 2004)

	1: static	2: job-level dynamic	3: unrestricted dynamic
3: full migration	$\frac{M^2}{3M-2} \le U \le \frac{M+1}{2}$	$\frac{M^2}{2M-1} \le U \le \frac{M+1}{2}$	$U = M$
2: restricted migration	$U \le \frac{M+1}{2}$	$M - \alpha\,(M-1) \le U \le \frac{M+1}{2}$	$M - \alpha\,(M-1) \le U \le \frac{M+1}{2}$
1: partitioned	$\left(\sqrt{2}-1\right) M \le U \le \frac{M+1}{1+2^{\frac{1}{M+1}}}$	$U = \frac{M+1}{2}$	$U = \frac{M+1}{2}$

job level dynamic, restricted migration) at its second level (with utilization bound of $M - \alpha\,(M-1) \le U \le \frac{M+1}{2}$).

So, as discussed before, at the first level, after clustering of the processors (each cluster working as a *Disp-engine*), tasks of the task system $q = <i, j, D, T, \omega, p>$ are scheduled and allocated to the clusters via the First-Fit and then EDF algorithms.

So, the proposed hybrid clustering multiprocessor real-time scheduling algorithm has the following characteristics for its first level.

Theorem 1 *A query system modeled as* $q = <i, j, D, T, \omega, p>$ *is schedulable upon the RT-SBD.*

Proof for a task (query) system to be schedulable, total utilization factor of periodic and aperiodic tasks must not be greater than one (Eq. 5) (Li and Wang 2007).

$$U_p + U_A \le 1 \tag{5}$$

In *RT-SBD*, utilization factor of periodic and aperiodic tasks is computed as follows (see Sect. 3.4):

For periodic query instances $q = <i, j, D, T, m, p>$:

$$u_i = \frac{e_i}{T_i} \quad and \quad U_P = \sum\nolimits_{i=1}^{n} u_i \sum\nolimits_{i=1}^{n} \frac{e_i}{T_i}$$

and for continuous query instances $q = <i, j, D, T, 1, 0>$ or one-time query instances $q = <i, j, D, T, m, \infty>$:

$$U_A = 1 - U_P \tag{6}$$

So, according to Eq. 6, condition $U_P + U_A \le 1$ holds and query system modeled as $q = <i, j, D, T, m, p>$ is schedulable in *RT-SBD*. □

Theorem 2 *Query system modeled as* $q = <i, j, D, T, m, p>$ *on M Disp-2 engines is feasible under the hybrid clustering real-time scheduling approach.*

Proof The set of $\boldsymbol{k}$ logical machines are partitioned into $M = k/2$ subsets, each one as a *Disp-2* engine. They process the query instance selected by the real-time scheduling algorithm (i.e., *FF+EDF*) in parallel. In other words, the system consists of M processing units (clusters of cores) for executing real-time queries (tasks) by applying the partitioning approach.

In Lopez et al. (2000), it is proved that for task system τ on M processors using the partitioning approach ((job-level dynamic priorities, no migration)-restricted scheduling class), if $U(\tau) \leq \frac{(M+1)}{2}$, then τ is feasible. Since, the following condition holds for queries utilization (Eq. 5):

$$U_A + U_P \leq 1$$

Therefore, utilization is less than (or equal to) one (i.e., $U(\tau) \leq 1$). Also, M (i.e., number of processors) is an integer value (and greater than one, in a multiprocessor system). So, for all values of M, condition $U(\tau) \leq \frac{(M+1)}{2}$ will be held. For example, for the least value for the number of processors $M = 2$, we have $U(\tau) \leq \frac{3}{2}$. Generally speaking, since that increasing the number of processors causes value of $\frac{(M+1)}{2}$ to be increased, thus for all $M > 2$, condition $U(\tau) \leq \frac{(M+1)}{2}$ will be held, too.

$$((\forall M \in \mathbb{N}, M \geq 2) \wedge (U(\tau) \leq 1)) => U(\tau) \geq \frac{(M+1)}{2}$$

So, query system modeled as $q = <i, j, D, T, m, p>$ on M Disp-2 processing engine and using the hybrid clustering real-time scheduling is feasible. □

In real-time systems context, in addition to system's logical correctness and timeliness which are essential, *Fairness* is interesting in multiprocessor real-time scheduling, too (Tatbul et al. 2003). Fairness is regularly mooted in the global multiprocessor real-time scheduling approach, as the Proportional Fair (PFair) scheduling and algorithms such as PF (Baruah et al. 1995), PD Anderson and Srinivasan (2004) and PD^2 (Srinivasan 2003). Periodic task systems can be optimally scheduled on multiprocessors using PFair scheduling algorithms. Under PFair scheduling, each task must execute at a uniform rate, while respecting a fixed-size allocation quantum. Uniform rates are ensured by requiring the allocation error for each task to be always less than one quantum, where "error" is determined by comparing to an ideal fluid system (Tatbul et al. 2003).

In global multiprocessor real-time scheduling which support *PFairness,* a task's weight (i.e., utilization) determines amount of the processor share it requires (Tatbul et al. 2003). In other words, roughly speaking, supporting proportional fairness in multiprocessor real-time scheduling means that "*the more the processor's utilization capacity, the more the workload assigned*". So, workload is assigned proportional to the processors' utilization capacity (e.g., a subtask with weight δ is assigned to the processor with utilization capacity of δ).

Although in *RT-SBD* the hybrid clustering approach is employed as multiprocessor real-time scheduling, but *PFairness* can be supported via employing the *Best-Fit* algorithm in the allocation phase of its multiprocessor real-time scheduling.

Theorem 3 *Using the Best-Fit algorithm in allocation phase of real-time scheduling in RT-SBD supports PFairness.*

Proof scheduling in *RT-SBD* is performed in to levels: real-time scheduling of queries and parallel scheduling of the selected query operators. Employing the Best-Fit algorithm in the allocation phase of the first scheduling level means that a query is assigned to processor such that less utilization capacity would be remained empty. In other words, Best-Fit attempts to fill processor's utilization capacity as much as possible and to leave the least empty capacity amount (Carpenter et al. 2004). Therefore, a processor with higher capacity will have more assigned workload; this functionality exactly matches the *PFairness* in global multiprocessor real-time scheduling. For example, for a processor with utilization capacity of 5 units, a query with weight of either 5 units or the greatest value less than 5, will be assigned.

Also, in the second level (parallel scheduling of query operators), operators are initially assigned to all of the processors (i.e., logical machines in the *Disp.*) manually and according to Eq. (1). So, the algorithm is work-conserving[5] (Devi 2006); also, since in Dispatching (Safaei and Haghjoo 2012) each operator sends its processed to the next operator (machine) which has the minimum workload (Eq. 2), hence scheduling in this level is also done absolutely proportionally fair with respect to processor's workload. Therefore, *PFairness* can be provided via using Best-Fit algorithm as allocation phase in *RT-SBD*'s scheduler. □

Similarly, the second level of the proposed hybrid clustering multiprocessor real-time scheduling algorithm can be analyzed.

3.4 Deadline monitor

Before delivering each result tuple to the user (or application), it is investigated by the deadline monitor unit, and its processing time (difference between current time and tuple's timestamp) is compared with the corresponding deadline.

Since RTDBSs are generally soft real-time (Babcock et al. 2003; Stankovic et al. 1999; Stonebraker et al. 2005; Johnson et al. 2008), improving the average performance is the main goal. Deadline investigating units mainly consider DMR and enforce proper policy (mechanism) when DMR threshold is violated (feedback control approach). There are two drawbacks in this approach:

(a) In data stream context, computing DMR is partially different. Generally, DMR (as the most important parameter of a real-time system) is computed as:

$$DMR = \frac{number\ of\ rejected\ tasks + number\ of\ missed\ tasks}{total\ number\ of\ tasks} \tag{7}$$

5 An algorithm is said to be work conserving if it does not idle any processor when one or more jobs are pending, and non- work conserving, otherwise.

For each missed deadline task, number of missed tasks is incremented by one. However, in real-time data stream systems, when a query instance misses its deadline the value that must be incremented is different. In fact, since a query instance must be executed over a window of tuples (Golab 2004; Kramer 2009; Kontaki 2010), the processing stage in which deadline is missed is important. For example, suppose that a query instance has missed its deadline when it has processed only one tuple of its window, but another one has missed its deadline when it has processed all of its window tuples except one. These two must not have equal effect on computing DMR. So, in computing DMR, we increase proportion of missed queries with respect to number of tuples in corresponding window which are not processed due to missing deadline (instead of increasing blindly by one). Hence, we define and use*Proportional Deadline Miss Ratio* as follows.

Definition 5 *Proportional deadline miss ratio (PDMR)* is equal to ratio of deadline missing query instances (ratio of non-processed tuples within the window) (rejected) to total number of query instances:

$$PDMR = \frac{\sum_{rejected} 1 + \sum_{missed} \frac{(\omega - number\ of\ tuples\ processed\ within\ the\ deadline)}{\omega}}{total\ number\ of\ query\ instances} \tag{8}$$

(b) Undesirable system status is notified after some tasks have missed their deadlines. In *RT-SBD*, in addition to investigating PDMR threshold violation, deadline missing *estimation* mechanism is provided for periodic query type. The goal of deadline missing estimation mechanism is to provide an early estimation of deadline miss ratio in order to prevent missing deadlines.

According to definition of periodic query type (*PQuery* model), each query instance should be executed over a window of $\boldsymbol{\omega}$ tuples before the determined deadline (Wei et al. 2006a). So, the deadline monitor unit computes processing time of one tuple of the window and multiplies it by $\boldsymbol{\omega}$ to estimate missing of deadline:

$$rt\,(\omega) = tuple\_latency\,(\omega) * \omega \tag{9}$$

The query instance would miss its deadline if the computed processing time is greater than its deadline $(rt(\omega) > d)$. In other words, if the measured tuple latency is greater than d/ω (estimating that deadline would be missed), the deadline monitor unit sends proper parameters to the *data admission control* unit. The data admission control unit uses these parameters to adjust systems' input tuple admission rate. In fact, in case that deadline missing is estimated, the deadline monitor unit forces the data admission control unit to discard (drop) more input tuples. Decreasing the workload causes the system to degrade response time (tuple latency) and satisfy deadlines.

Corollary 1 *Increasing the rate of dropping input tuples leads to decreasing system tuple latency.*

Proof Operator path that a tuple should traverse to be processed is a path in query plan graph [*Query Mega Graph* (Safaei and Haghjoo 2010)] (Babcock et al. 2004).

Cost of this path which determines tuple latency is the summation of cost of nodes (operators) and edges (operator input queues) in this path (Safaei and Haghjoo 2010).

$$\forall t \in data\_stream\_tuples \Big(tuple\_latency\,(t) = \sum\nolimits_{1\leq i\leq |V|} cost\left(O^i_-\right) + \sum\nolimits_{1\leq i\leq |V|} \left(Buffer\_size\left(O^i_-\right)\right)$$

Since set of operators is the same for different operator paths of one query, decreasing the total length of operators input queues leads to decreasing tuple latency. So, by increasing the drop rate of input tuples, number of tuples entering the system and waiting in operators input queues is decreased and hence tuple latency is degraded. □

Corollary 2 *Increasing the rate of dropping input tuples leads to decreasing probability of missing deadlines.*

Proof According to the above, increasing input tuples drop rate decreases tuple latency. With respect to Eq. (9), decreasing tuple latency causes decreasing of the response time for the whole widow ($rt(\omega)$) and the condition ($rt(\omega) \leq d$) is more probable to hold. In other word, probability of missing deadline for the corresponding query instance would be decreased. □

Although system circumstances and tuple latency may change at each time instance, but this computation for each output tuples causes overhead to the system. In order to degrade this overhead, we perform deadline missing estimation for multiple tuples instead of one tuple as follows.

Definition 6 *Estimation Interleaving Factor (EIF)* Interleaving factor for estimating deadline missing in a window of ω tuples, denoted as EIF, indicates how many tuples are interleaved between each two deadline missing estimations. For instance, EIF=1 means that estimation is done for half of the tuples (one interleaved).

Corollary 3 *The more the EIF value, the less the system result quality.*

Proof As an example, assume that $EIF=\frac{\omega}{2}$ in which estimation is done only twice (i.e., at the beginning and middle of the window). In this scenario, we have lost half of input tuples. To compensate and to decrease probability of missing the deadline, we should drop more tuples from the remaining half of the window (corollary 2). Therefore, discarding the input tuples leads to degrade quality of system's output results. □

As a result, it is very important to determine a proper value for EIF and make a tradeoff between decreasing deadline missing probability as well as estimation overhead and the quality of output results. Effect of different values of *EIF* to *PDMR* is analyzed via experimental evaluation of *RT-SBD* in Sect. 4.

3.5 Data admission control

Generally, there are two main approaches for dealing with overload situations and decreasing the workload: *load shedding* and *data admission control*.

Load shedding is often performed via the query plan [e.g., by applying load shedding operators within the query plan as in the Aurora (Abadi et al. 2003)]. Despite the fact that the load shedding helps to decrease system's workload more effectively, it has some drawbacks for real-time systems as well as a considerable complexity. One major drawback of load shedding approach for a real-time DSMS[6] is that tuples are selected to be discarded in a stage of the query plan in which they have passed a portion of processing. Discarding tuples in this stage wastes the processing time consumed for providing these intermediate results. This time wasting is not acceptable for a real-time DSMS. Also, in many real-time applications, intermediate results are even more worthy than missing deadlines (Wei et al. 2006a). Roughly speaking, earlier load shedding (i.e., in the earlier stages of the query plan) would be more effective (Babcock et al. 2004).

In data admission control approach which is query independent and simpler, excess tuples are discarded before entering the query plan. Although in this approach excess tuples are discarded almost blindly but it does have less complexity and overhead. Also, discarded tuples are not processed, so no time is wasted for processing them.

In fact, load shedding is substituted by dynamic load balancing in *RT-SBD* since in each *Disp-2* engine, tuples waiting in input queues of an overloaded operator are redirected to the corresponding operator in another under-utilized machine (Safaei and Haghjoo 2010, 2012). So, in *RT-SBD*, overload situation is handled by data admission control, in addition to decreasing its probability (via dynamic load balancing).

The data admission control unit simply is a dropper. Its tuple dropping (discarding) rate is adjustable by parameters U_P and U_A, cumulative utilization factor of processor for periodic and aperiodic queries respectively. Static setup of these parameters is not suitable because system status will change during runtime (Wei et al. 2006a; Li and Wang 2007). So, using the feedback control mechanism, these parameters are adjusted dynamically by the deadline monitor unit (using *PDMR* as the measured variable for tuning input tuple drop rate). For real-time DSMS, classic *PID*[7] controller is not suitable due to irregular data arrival rate as well as variable selectivity of queries (Wei et al. 2006a, 2006b). In data admission control unit of *RT-SBD*, the *PI*[8] controller is used which is simple, and provides an acceptable response time to workload fluctuations (Li and Wang 2007). The proportional deadline miss ratios (*PDMRs*) are sampled periodically and compared against target value. The differences are passed to the *PI* controller to generate the data admission control signal ΔP_{AC}, which is subtracted from the current data admission ratio. The ΔP_{AC} is derived using Eq. (10) (Li and Wang 2007).

$$\begin{aligned}\Delta P_{AC} = P_{PDMR} \times (PDMR_{ST} - PDMR_{threshold})\\ + I_{PDMR} \times (PDMR_{LT} - PDMR_{threshold})\end{aligned} \tag{10}$$

In Eq. (10), $PDMR_{ST}$ and $PDMR_{LT}$ are the short-term and long-term proportional deadline miss ratios sampled in the last sampling period. $PDMR_{threshold}$ is the spec-

6 Data stream management system.

7 Proportional-integral-derivative.

8 Proportional-integral.

ified maximum proportional deadline miss ratio allowed by the application; P_{PDMR} and I_{PDMR} are two controller parameters which control the weights that short-term and long-term proportional deadline miss ratios have on the data admission control signal. How to tune the *PID* (also *PI*) controller to suit different system responses has different methods (e.g., manual tuning, *Ziegler–Nichols*, *Tyreus Luyben*, *Cohen–Coon*, using software tools, etc.) (Marisol et al. 2014; Astrom and Hagglund 1995). In this paper, the two controller parameters are handpicked to give the best system response.

Usually, a real-time system is expected to be predictable. Predictability in real-time systems means that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumption made (Stankovic and Ramamritham 1990). Of course, predictability is in fact a property required for the Hard Real-Rime (*HRT*) systems and not so necessary for soft or firm real-time systems. Moreover, data stream is unpredictable in its nature (Babcock et al. 2003). But, the proposed *RT-SBD* potentially can provide desired predictability in terms of timing constraints by the contributed feedback-control mechanism [i.e., EIF adjustment, cost prediction and computation as is performed in deadline-aware dispatching (Elliott et al. 2013), etc.].

According to the discussions about *RT-SBD's* architecture, a high level description of its operation is illustrated in Fig. 6. Corresponding to the main components of *RT-SBD*'s architecture, request manager, scheduler, parallel query processing, and deadline monitor are considered as the main components, with sub-routines stated in the form of pseudo code.

4 Performance evaluation

4.1 Experimental setup

The contributed real-time streaming big data processing engine prototype (*RT-SBD*) is implemented in *Java* (*Java SE 8 Update 66*) on *Linux* (*Ubuntu 14.04*). Also,*Virtualization* is used to emulate the needed underlying multiprocessor system. Machine virtualization allows a single physical machine to emulate the behavior of multiple machines, with the possibility to host multiple and heterogeneous operating systems (also known as guest OSs) on the same hardware. A *Virtual Machine Monitor* (*VMM*), or *hypervisor*, is the software infrastructure running on (and having full control of) the physical host and which is capable of running such emulation. However, in order to have an efficient access to the physical platform through the VMM, you should use real-time hypervisors which typically may allow applications to access to the physical machine, in virtualized environments (Beloglazov et al. 2012). So, *ESX6-update01-3029758 – x86_64a* VMware (which is known as a real-time hypervisor) is used as the VMM, running on *SuperMicro* server (*X8DTL*) machine with Intel *Xeon E5620* 2.4GHz processor (with 8 *Xeon cores*) and 30GB RAM.

Each logical machine of parallel query processing engine is considered as a core of the multi-core CPU. *RT-SBD* with clusters performing parallel query processing via the deadline-aware dispatching with (*M* processing machines in which *M* is determined via experimental evaluation) is evaluated and compared with the most popular real-time streaming big data processing engine, *Storm* (https://storm.apache.

```
RT_Scheduler()
1.  { if (enqueued_flag(query_set_queue)){
2.       $q_i$ ←dequeue(query_set_queue);
3.       k← first_fit(utilization_factor($q_i$),utilization_factor(Disp − $3_i$));
4.       enqueue($q_i$,query_waiting_queue(Disp − $3_k$));
5.       enqueued_flag(query_set_queue)=FALSE;
6.     }
7.  }

Parallel_Query_Processing()
1.  { For each query processing engine Disp-$3_i$ do in parallel:
2.    { ready_query← EDF(query_waiting_queue(Disp-$3_i$));
  // processing of selected query in parallel via the dispatching method
3.      generate query plan for the registered query
4.      generate k identical copies of query plan and send each one to a logical machine
5.      generate the Query Mega Graph (QMG)
6.     initial scheduling: assign the first operator of the query plan to the first machine
7.     repeat in parallel by each machine
8.            select the edge with minimum weight (i.e., minimum workload machine)
9.            repeat
10.              process tuple in front of the input queue
11.             send result tuple and (operator_Id+1) to destination machine of the selected edge
12.              Increment counter
13.           until counter> ω
14.          buffer and sort output tuple and deliver to user or application
15. until there is no more input tuple OR query is expired
16.   }
17. }

Deadline_Monitor()
1.  { for each output result tuple $t_i$:
2.       if (current_time>deadline_query_instance($t_i$))
3.            missed_deadline_counter+=(#tuples_missed_their_deadline)/(window_size);
4.           else{
5.               if (estimation_time)                //estimation interleaving is passed
6.                   if(window_size*(current_time-timestamp($t_i$))>deadline_query_instance($t_i$)) {
7.                       compute data admission control parameters w.r.t. equation (VII) and send them to it;
8.                      update estimation_time related parameters;
9.                      }
10.          }
11.   }
```

Fig. 6 Pseudo code of different components in *RT-SBD* architecture

org/). In the comparison, the contributed *RT-SBD* engine with the proposed clustering multiprocessor real-time scheduling approach which uses First-Fit algorithm (Carpenter et al. 2004) in its allocation phase (named *RT-SBD-FF*), *EDF* algorithm for selection of a query instance among the cluster's waiting queue to execute, and the contributed deadline-aware dispatching method for the selected query instance is evaluated. Moreover, in order to show how the other alternatives affect the system, the Best-Fit, Next-Fit and Worst-Fit algorithms (Carpenter et al. 2004) are also used as *RT-SBD's* deadline-aware dispatching phase (named *RT-SBD-BF, RT-SBD-NF* and *RT-SBD-WF*, respectively). Also, the case with *sequential* hybrid multiprocessor

Table 2 Range of selectivity values for some of the operators

	Min	Max	Average
Filter	0.002046	1.0	0.4843909
Join	1.164173	4.910254	2.9286148
Project	1.0	1.0	1.0

real-time scheduling (as we have introduced previously in Safaei et al. (2011) named *PFGN*[9]) is evaluated and compared. By default, *EDF* is used as the single processor real-time scheduling algorithm and in the *allocation* level of the proposed hybrid real-time scheduling algorithm.

To evaluated whatever discussed and analyzed theistically in Sect. 3.3.1 in an experimental manner, effects of the input dropping rate and the *EIF* value, are measured and discussed. Moreover, the system's operation is compared in different situations (e.g., by emitting each of the *admission control* and *deadline monitor* components, which changes the closed-loop control to open-loop).

The Linear Road Benchmark (Arasu et al. 2004) is used for determining data set and query set. Data set is generated using MIT traffic simulator (*MITSIM*) (Yang and Koutsopoulos 1996) (about 12e+6 streaming data tuple).

The query set consists of $\underline{17}$ different types (containing $\underline{4}$ operators to single operator queries). The operators are query operators mostly used in data stream applications (including selection, projection, aggregation, stream-to-relation and stream-to-stream windowed join, etc.). Table 2 shows selectivity values for some of these operators:

Deadline and period of queries are set as k times of the estimated query execution time ($1 \leq k \leq 10$). Evaluation duration is about 450 min (∼27,000,000 ms). 5 different runs of the above scenarios are made and their average value is measured and computed.

The most important evaluated parameters are:

- *PDMR*: proportional deadline miss ratio according to Eq. (8).
- *Tuple Latency*: difference between each tuple's arrival and departure time.
- *Throughput*: number of query instances processed in a time unit.
- *Memory Usage*: total amount of memory space consumed.
- *Tuple Loss*: number of discarded tuples.

Overheads such as communication or context-switching are negligible because the employed machines are cores of a multi-core CPU.

4.2 Experimental results

Experiment 1 *Configuration of the contributed system in term of number of processing machines in each cluster.*

Charts in Fig. 7 show efficiency of RT-SBD in terms of the measured parameters in different cases of the number of processing machines in a cluster (i.e., *single*, *dual*, *quad*, *octal*, and *hexa*).

9 Single real-time disp.

As can be inferred from charts in Fig. 7, the best case for the *RT-SBD* is using two processing machines in each of clusters (*named as QRS-Dual*); because it is performing more effectively in this case, by having minimum PDMR, minimum tuple latency, and maximum throughput, acceptable tuple loss, especially in their steady state.

Accordingly, the proposed RT-SBD processing engine prototype is set up and configured with clusters with two cores (i.e., *using Disp-2 engines*) that process the allocated query in parallel by using the (deadline-aware) dispatching method. By

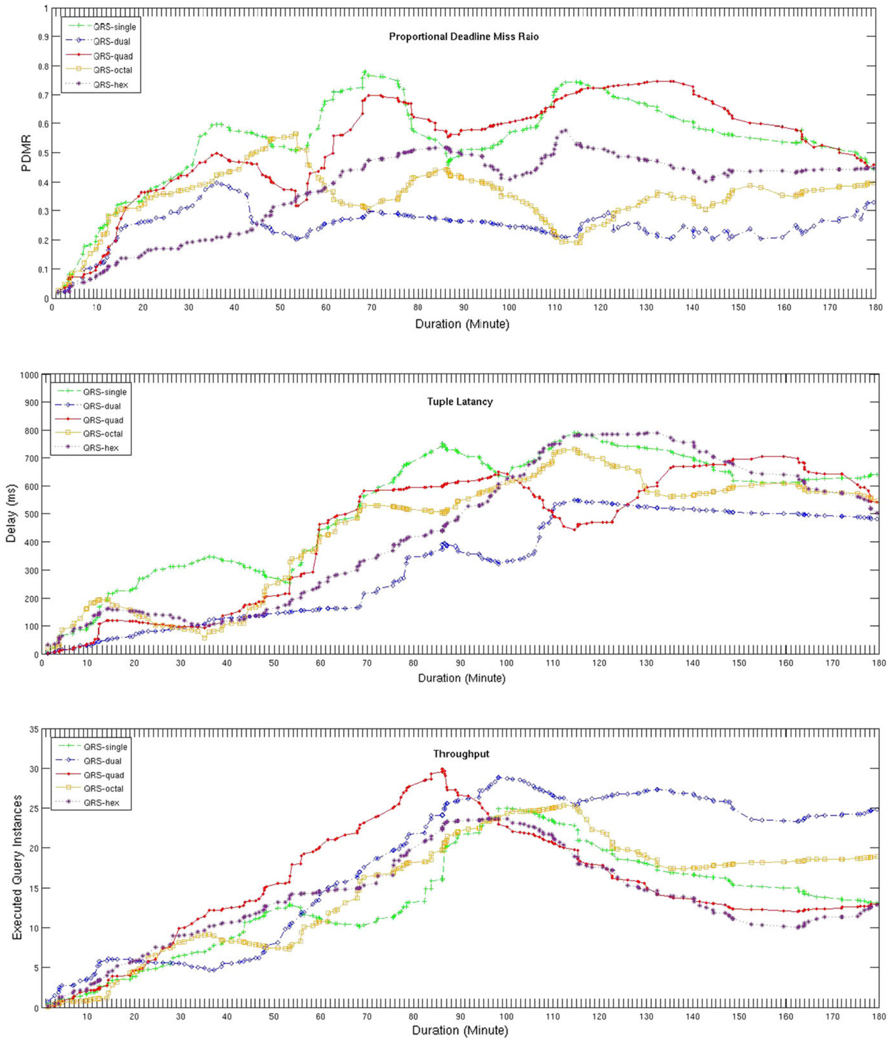


Fig. 7 *RT-SBD*'s efficiency parameters for different cases as number of processing machines in a cluster

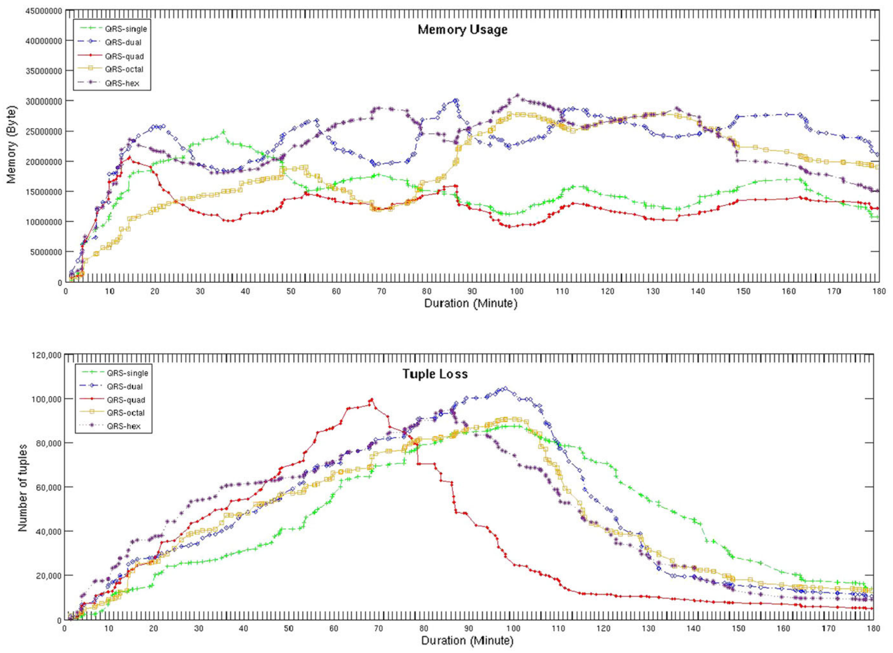


Fig. 7 continued

this configuration, the proposed *RT-SBD* prototype (named as *QRS*[10], with First-Fit, Best-Fit, Next-Fit and Worst-Fit Deadline-aware dispatching, *QRS-FF* and *QRS-BF*, *QRS-NF*, and *QRS-WF,* respectively) are compared with the Storm (https://storm.apache.org/) and the PFGN (Safaei et al. 2011).

Experiment 2 *Measuring performance parameters for the contributed system and comparison.*

Figures 8, 9, 10, 11, and 12 illustrate evaluation charts of *PDMR*, *tuple latency*, system throughput, *memory usage* and *tuple loss* in *RT-SBD* (with different configurations), *PFGN* and the *Storm*, respectively.

Deadline miss ratio (*DMR*) is the most important parameter in each real-time system. But as discussed in Sect. 3.4, in real-time streaming big data processing engines it is hanged into *Proportional Deadline Miss Ratio* (*PDMR*). According to Fig. 8, the proposed system, nearly in all configurations, outperforms *Storm* and*PFGN* in steady state.

Since the contributed *RT-SBD* is also evaluated in other cases that each of the deadline monitor and admission control units of *RT-SBD's* architecture are emitted (reported in the next figures), this case is explicitly distinguished by the label "(with Deadline Monitor & Admission Control)".

[10] Quick Real-time Stream processor.

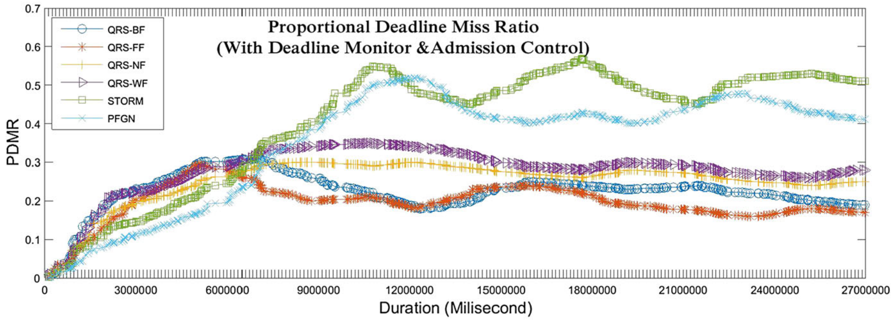


Fig. 8 PDMR in *RT-SBD* vs *Storm* and *PFGN*

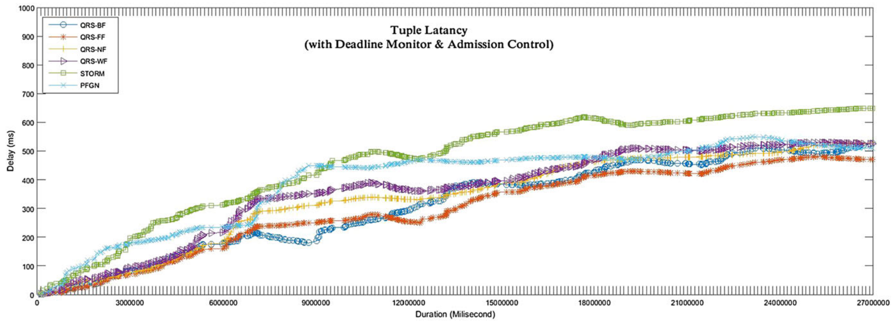


Fig. 9 Tuple latency in *RT-SBD* vs *Storm* and *PFGN*

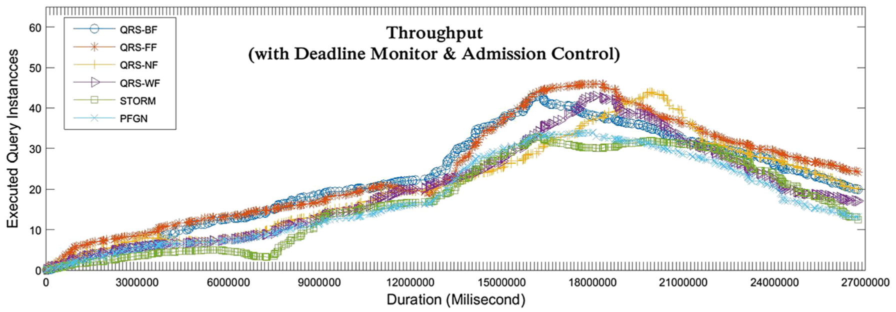


Fig. 10 Throughput in *RT-SBD* vs *Storm* and *PFGN*

In order to have more complete and accurate evaluation, *RT-SBD* is compared with *Storm* and*PFGN* in terms of other important and influencing parameters (Figs. 9, 10, 11, 12).

By these time-varing vlaues charts, it may be hard to judge about the performance of the compared alternatives. So, the average value of each parameter for each of the compared configurations and systems, are computed and represented in Figs. 13, 14, 15, and 16. But before that, as stated before, lets see what is the effect of feedback control mechanism used in the *RT-SDB*. In order to evaluate the contributed system

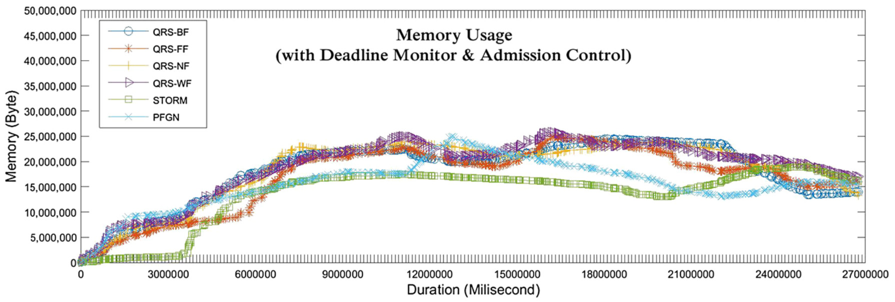


Fig. 11 Memory usage in *RT-SBD* vs *Storm* and *PFGN*

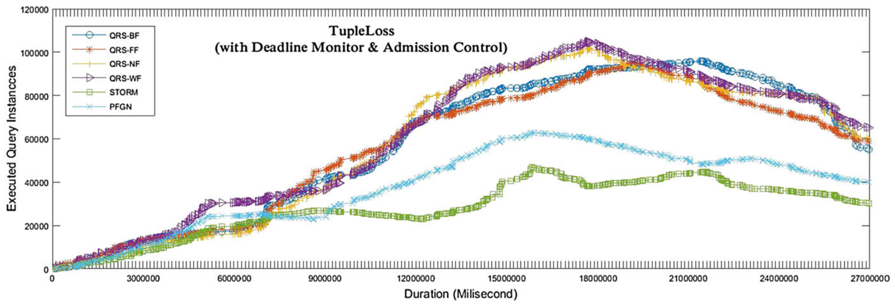


Fig. 12 Tuple loss in *RT-SBD* vs *Storm* and *PFGN*

in such other configurations, parameters are measured in the case that the deadline monitor unit, the admision control unit, and also both of them are emmited, and the closed-loop control becomes open-loop (charets are presented in the appendix).

As is shown in charts, value of *PDMR* and tuple latency are increased in the case of emitting one of the mentioned components. This is an obvious effect since the contributed feedback control mechanisms final goal is to reduce the *PDMR* as the most important measure. Also, this result confirms that the contributed parallel query processing (i.e., *deadline-aware dispatching*) is really deadline-aware; by ignoring systems PDMR and deadlines of tuples, dispatching is not working perfectly and tuple latency increases (Corollary 1).

These charts presented momentarily changes of the parameters' values. In order to have a more accurate analysis, average value of each parameter for each of the compared configurations and systems, are computed and represented in Figs. 13, 14, 15, 16, and 17.

Performance evaluation charts illustrated in Figs. 13, 14, 15, 16, and 17 generally show that *RT-SBD* makes a considerable improvement in terms of *PDMR* as well as tuple latency and system throughput whilst requires more system resources (e.g., memory space). In other words, compared alternatives can be ranked from the best to the worst as follows: *QRS-FF*, *QRS-BF*, *QRS-NF*, *QRS-WF*, *PFGN*, and the *Storm*. RT-SBD with First-Fit deadline-aware dispatching method as the best and recommended configuration has improvement in *PDMR* (∼50 % of the *Storm*), tuple latency (∼66 %

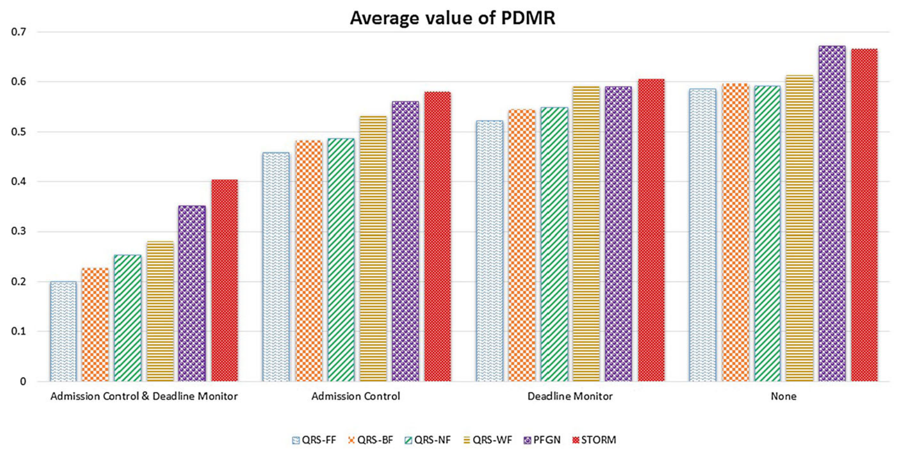


Fig. 13 Average value of PDMR in *RT-SBD* vs *Storm* and *PFGN*

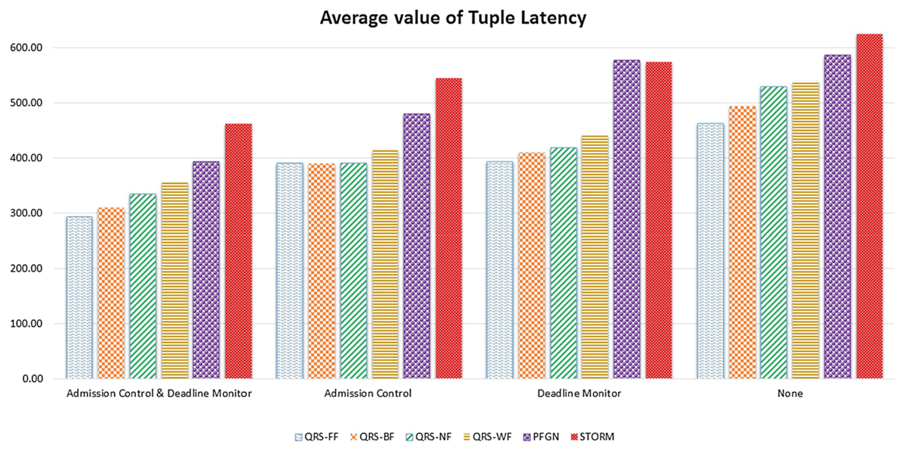


Fig. 14 Average value of tuple latency in *RT-SBD* vs *Storm* and *PFGN*

of the *Storm*), and throughput (∼1.4 of the *Storm*), while has some penalties in terms of memory usage (∼1.2 of the *Storm*) and tuple loss (∼1.9 of the *Storm*).

So, Experimental results illustrated that *RT-SBD* significantly outperforms the most popular streaming big data processing system, *Storm* (https://storm.apache.org/). Also, *RT-SBD* processing engine with First-Fit algorithm in its deadline-aware dispatching (i.e., *QRS-FF*) has the best performance win terms of PDMR, tuple latency and throughput. Furthermore, the First-Fit algorithm is more efficient than the Best-Fit since it does not perform selecting the best one. But, on the other hand, the RT-SBD with Best-Fit supports fairness (Theorem 3).

As shown in Figs. 14 and 18, achieving improvements of PDMR and tuple latency cause some overheads and costs, for example in term of more memory space required. However, experimental evaluation results show that the ratio of improvements in terms

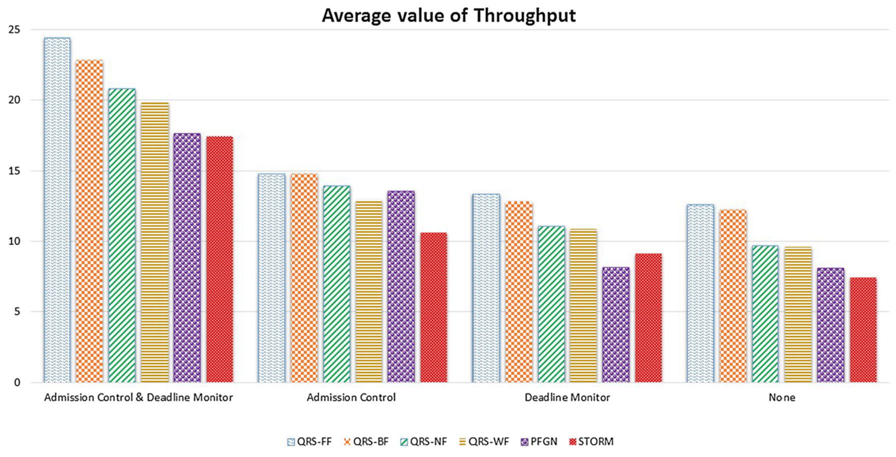


Fig. 15 Average value of system throughput in *RT-SBD* vs *Storm* and *PFGN*

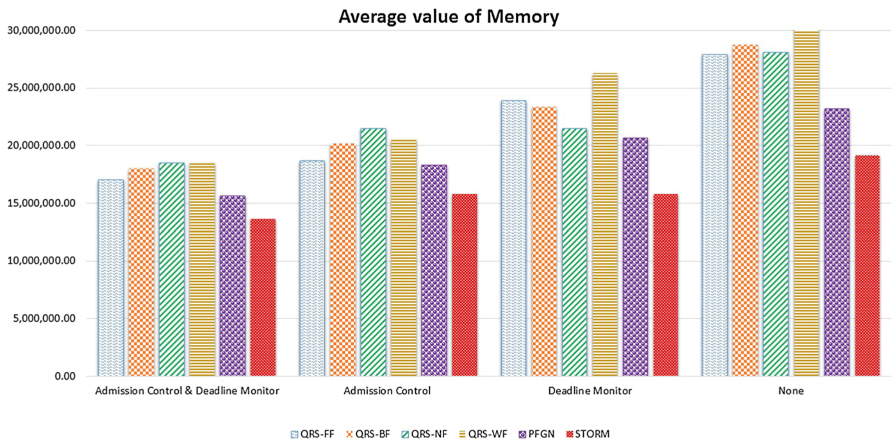


Fig. 16 Average value of memory usage in *RT-SBD* vs *Storm* and *PFGN*

of PDMR, tuple latency, and system throughput is very high compared to the costs and overheads.

Experiment 3 *Monitoring overhead versus timeliness (deadline missing estimation overhead versus PDMR).*

The other thing that should be noted is that, although tuple loss ratio may increase, but according to corollary 4, amount of tuple loss ratio depends on the value of the *EIF*. So, we can make a tradeoff between tuple loss ratio and other parameters such as the estimations overheads via setting proper value for *EIF*.

Figure 19 shows the average value of *PDMR* (for *QRS-FF* and *QRS-BF*) versus different *EIF* settings.

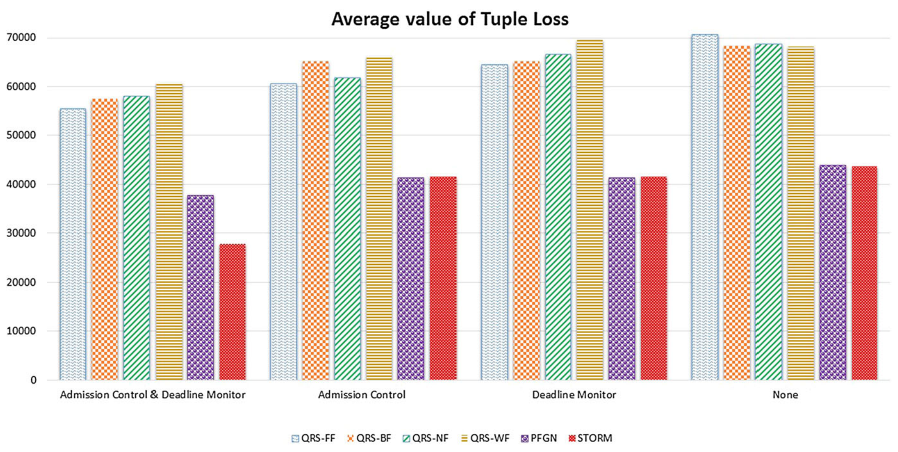


Fig. 17 Average value of tuple loss in *RT-SBD* vs *Storm* and *PFGN*

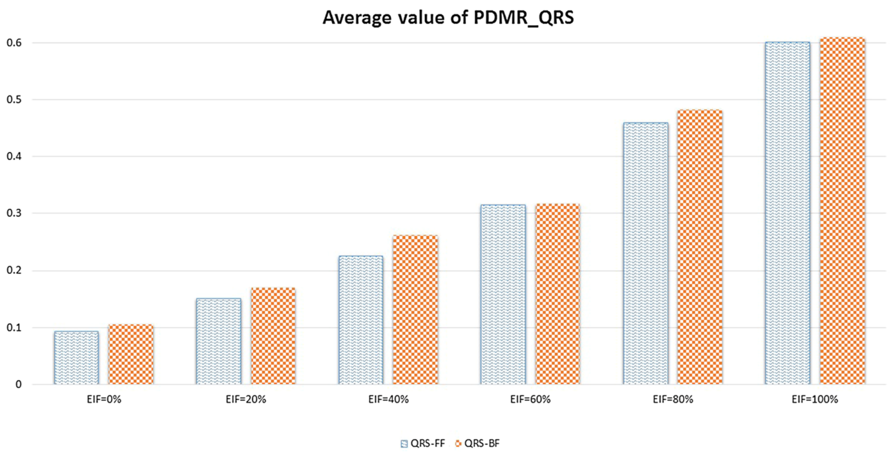


Fig. 18 *PDMR* versus *EIF* values

The results of Fig. 18 show *PDMR*'s growth by increasing of EIF value; *the more the EIF value, the less the system result quality (in term of timeliness)* as stated and proven in corollary 3.

Experiment 4 *Data quality versus timeliness (input tuple dropping rate versus PDMR).*

Also, in order to analyze what was stated and proven in corollary 2 about the relationship between input tuple drop rate (data completeness as a data quality metric) and PDMR (timeliness), Fig. 19 shows the average value of *PDMR* (for *QRS-FF* and *QRS-BF*) versus different *input tuple drop rates.*

As illustrated in Fig. 19, increasing the input tuple drop rate (performed by the admission control unit, based on the deadline monitor unit's signal), causes PDMR to be decreased.

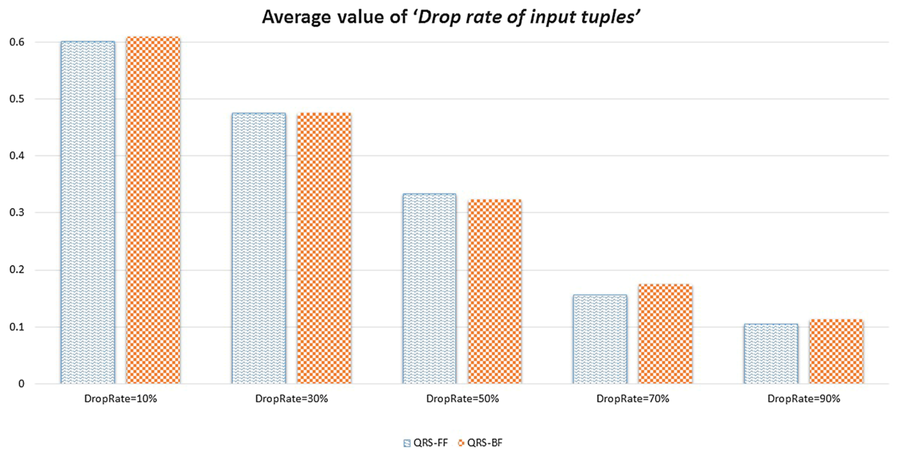


Fig. 19 *PDMR* versus *input tuple drop rate*

Experimental results (Figs. 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19) show that *RT-SBD* significantly outperforms the most popular streaming big data processing system, *Storm* (https://storm.apache.org/). Also, *RT-SBD* processing engine with First-Fit algorithm in its deadline-aware dispatching (i.e., *QRS-FF*) has the best performance win terms of PDMR, tuple latency and throughput, while these improvements has some tolerable system resource requirements (e.g., memory usage). Also, it is shown that emitting the contributed feedback control mechanism (and even each of its components), dramatically degrades systems performance.

5 Related work

A considerable research activity pertains to stream systems. Real-time query processing is essential in most streaming big data applications (e.g., surveillance, healthcare or network monitoring) (The STREAM Group 2003). Although a number of DSMS prototypes have been developed including STREAM (Ma et al. 2009) and Aurora (Abadi et al. 2003), but none of them satisfy real-time requirements. Aurora as a full-fledged DSMS aims to provide quality of service which is different from real-time requirements, it doesn't have any mechanisms for defining deadline, scheduling based on deadlines, etc. (Abadi et al. 2003). RTSTREAM (Wei et al. 2006a) claims to extend STREAM (Ma et al. 2009) to satisfy real-time requirements, but this extension is limited to periodic query model [*PQuery*), EDF real-time scheduling policy (not sufficient for a real-time DSMS (Bestavros and Nagy 1996)], and adding some phrases to the CQL language to declare deadline and period of queries (Wei et al. 2006a). We employ PQuery as well as aperiodic query type (i.e., continuous and one-time) in *RT-SBD*. There are many other systems and engines developed for processing of streaming big data. For example, S4 (Neumeyer et al. 2010) which is a general purpose, distributed, scalable, fault-tolerant, and pluggable platform written in Java and initially released by *Yahoo!*, but to be real-time was not the concern. Apache *Hadoop*

(Bu et al. 2010), known as the king of big data analytics, use the Map-Reduce computational model (Condie et al. 2010; Yang et al. 2007) and is essentially for batch processing. *Storm* (https://storm.apache.org/) is a free and open-source distributed and fault-tolerant engine for real-time computing of streaming big data (in fact, is a *CEP*[11] engine). Also, there are frameworks for developing applications for fast data processing such as Muppet (Wang et al. 2012; Safaei and Haghjoo 2014) and Esper (http://www.espertech.com/esper/). *Storm* is recently replaced by *Twitter Heron*; a new platform for real-time analytics to provide the required size scalability, has better debug-ability, better performance, and to be easier to manage—all while working in a shared cluster infrastructure (Kulkarni et al. 2015).

In Kleiminger et al. (2011) eight requirements are presented for real-time DSMSs. Fast processing, transparent and automatic distribution of processing across multiple processors and machines are the most important ones. Serial query processing in existing DSMS prototypes is not capable of executing continuous queries over continuous data streams with a satisfactory speed. We have presented parallel processing of continuous queries over logical machines in Safaei and Haghjoo (2010). The scheduling method employed in Safaei and Haghjoo (2010) is dynamic but event-driven (in overload situation). Considering the continuous nature of continuous queries and data streams, compatibility with this nature and adaptivity with time varying characteristics of data streams is very important. In Safaei and Haghjoo (2012), we introduced a dynamic continuous scheduling method (*dispatching*) to substitute the even-driven one presented in Safaei and Haghjoo (2010). Also, we have discussed system architecture, practical challenges and issues for the underlying parallel system, as well as its implementation on multi-core processors in Safaei et al. (2012). Employment of dispatching instead of event-driven scheduling provided system performance improvement as well as fluctuations reduction (Safaei and Haghjoo 2012). Although it is necessary for a real-time system to be fast, but it is not sufficient. Mechanisms such as defining deadlines, deadline-based scheduling and deadline satisfaction investigation are required (Kleiminger et al. 2011). Accordingly, in this paper, we added mechanisms for request management, parallel query processing, real-time scheduling, deadline monitoring and input data admission control to *RT-SBD*.

Request admission strategies are inspected in Jamin et al. (1993). In most paradigms it is assumed that execution requirements of requests are pre-specified (Bestavros and Nagy 1996). In Wei et al. (2007) an admission paradigm is proposed in which a compensating request is used for each unsuccessful one. This (compensation or even rollback) is not applicable for data stream due to its append-only nature. Query processing time estimation for QoS management of real-time streams is argued in Yang et al. (2007). Request manager unit of *RT-SBD* uses response time computation function proposed in Mohammadi (2010).

Parallelism in database systems is covered in Graefe et al. (1994). Parallel processing of continuous queries over data streams are vastly covered in Safaei and Haghjoo (2010, 2012).

[11] Complex-event processing.

The main contribution in a real-time system design is its real-time scheduling. History of important events and key results in real-time scheduling is reviewed in Baruah et al. (1996). Multiprocessor real-time scheduling which is totally different from traditional single processor real-time scheduling is classified into three approaches: global, partitioning and hybrid. Problems and algorithms related to these approaches are discussed in Carpenter et al. (2004). Despite optimality of *PFair* scheduling algorithms [such as PF (Baruah et al. 1995), PD (Anderson and Srinivasan 2004) and PD^2 (Srinivasan 2003)], partitioning is currently favored (Safaei et al. 2011). The reasons are: (*a*) *PFair* scheduling algorithms have excessive overhead due to frequent preemptions and migrations (*b*) *PFair* scheduling are limited to periodic tasks (*c*) though partitioning approaches are not theoretically optimal, they tend to perform well in practice (Safaei et al. 2011).

To achieve the benefits of these two multiprocessor real-time scheduling approaches together, different *Hybrid* approaches have been proposed by researchers (Safaei and Haghjoo 2010). For example, *EKG* (Andersson and Tovar 2006), *Ehd2-SIP* (Kato and Yamasaki 2007), *EDDP* (Kato and Yamasaki 2008), *PDMS-HPTS* (Lakshmanan et al. 2009), *HMRTSA* (Srinivasan and Anderson 2004), *PFGN* (Safaei et al. 2011) and *PDMRTS* (Alemi et al. 2011) use *semi-partitioning* hybrid approach which aims at addressing the fragmentation of spare capacity in partitioning approach is to split a small number of tasks between processors (Safaei and Haghjoo 2010). In Safaei et al. (2011), we discussed different alternatives of hybrid multiprocessor real-time scheduling algorithms derived from mixing the partitioning and global approaches; these two approaches can be employed sequentially or concurrently. The scheduling algorithm we introduced in Safaei et al. (2011) was a sequential one in which partitioning approach is used entirely and after that, global approach will be used for scheduling of the *remained* tasks. In contrast, the proposed hybrid clustering multiprocessor real-time scheduling is a concurrent one; at the allocation level, it uses partitioning approach while at the deadline-aware dispatching level it uses global approach for the *same* task which is under scheduling.

Acceleration of big data processing was already shown in the past with middleware technologies such as *iland* (Valls et al. 2013). Big data processing applied in surveillance and remote object tracking for critical spaces monitoring proved to be appropriately handled (meeting timing deadlines) with mainstream middleware. In order to implement and employ the contributed system and components, proper frameworks can be used. There are different frameworks with various capabilities to support real-time task; For example, RTSJ (Bollella and James 2000) as a secure platform with rich functionalities for real-time Java applications is extended in Kwon et al. (2012) to support various multiprocessor real-time scheduling algorithms. Also, *ExSched* (Åsberg et al. 2012) is developed as an external real-time scheduler framework which enables different schedulers to be developed using a uniform developing interface. Some other frameworks are developed for validation, test and analysis of real-time scheduling algorithms and scheduler implementation (Golatowski et al 2002). Hierarchical schedulers are also supported in frameworks such as HLS (Regehr and Stankovic 2001). Adaptivity for tasks which may frequently require significant share changes in multiprocessor real-time systems is issued in Block et al. (2008). Moreover, there are frameworks for managing GPUs in a real-time system e.g., *GPUSync* (Elliott et al. 2013).

Virtualization allows for server consolidation in data centers, where multiple operating systems that would leave their underlying hosts under-utilized can be moved to the same physical resources. This enables the achievement of a reduction of the number of required physical hosts (Beloglazov et al. 2012). But, despite the success of cloud computing for general-purpose computing, existing cloud computing and virtualization technology face tremendous challenges in supporting emerging soft real-time applications (Marisol et al. 2014). Machine virtualization (also referred to as processor virtualization) allows a single physical machine to emulate the behavior of multiple machines, with the possibility to host multiple and heterogeneous operating systems (called guest operating systems or guest OSs) on the same hardware. A virtual machine monitor (VMM), or hypervisor, is the software infrastructure running on (and having full control of) the physical host and which is capable of running such emulation.

For our multiprocessor real-time scheduling problem, using virtualization can be beneficial specially in experimental and practical issues.

6 Conclusion

Big data is the current challenge of data management, in research, academia, industry and technology. Velocity, as on the 3Vs in big data problem, refers to both high speed data (e.g., streaming data) and high speed (i.e., real-time) processing. In order to be real-time, it is inevitable to be fast. Most often, a single processor is not capable of processing query's operators continuously over infinite, continuous and rapid streaming data tuples with a satisfactory speed. In order to solve this shortcoming, we have presented parallel processing of continuous queries in a multiprocessing environment in Safaei and Haghjoo (2010) and enhanced it in Safaei and Haghjoo (2012). Fast operation is a necessary but not sufficient condition for real-time systems. Generally, there are two main approaches to task scheduling on multiprocessor platforms; partitioning and global scheduling. Under global scheduling a higher utilization bound can be achieved, but in practice the overheads of migrating tasks is high. On the other hand, under partitioned scheduling, besides simplicity and efficiency, existing scheduling and synchronization methods developed for uniprocessor platforms can more easily be extended.

The main contribution of this paper is handling of the *Velocity* characteristic of big data (i.e., real-time processing of streaming big data, as described in the introduction). To deal with this problem, its challenges are issued and proper solutions (preferably, using our previous works) are provided. A real-time streaming big data processing engine (named *RT-SBD*) is proposed in which required components are designed. It uses our parallelism method for fast processing of queries. Set of the processors is clustered such that processors in each cluster can process the assigned query in parallel, using the proposed (deadline-aware) dispatching method. Assignment of queries to the clusters is done via the allocation level of the proposed hybrid clustering multiprocessor real-time scheduling algorithm; a submitted query is accepted if its determined deadline can be satisfied. Each accepted query is assigned to the first processing unit with utilization factor not less than the query's utilization (i.e., *First-Fit* algorithm). Each cluster selects the highest priority query instance among its waiting queue according to the *EDF* policy and then, processes the selected query in parallel via the (deadline-aware) dispatching method (Safaei and Haghjoo 2012). Since the two levels of the multiprocessor real-time scheduling algorithm uses both the partitioning (at the allocation level, by FF+EDF allocation of queries to the proper cluster) and the global approaches (at the deadline-aware dispatching level, by migration of the operators of the assigned query among the processors in the cluster), the proposed real-time scheduling algorithm is categorized as a *hybrid clustering multiprocessor real-time scheduling algorithm*. *Proportional Deadline Miss Ratio* is computed and compared with its threshold as well as performing *deadline missing estimation*. So, input data tuple arrival rate (system workload) is reduced to decrease the probability of missing deadlines. *RT-SBD* prototype is implemented and its performance is evaluated and compared with the *Storm* (https://storm.apache.org/) and *PFGN* (Safaei et al. 2011) in terms of PDMR, tuple latency, system throughput, memory usage and tuple loss. Experimental results show that using First-Fit as deadline-aware dispatching level of multiprocessor real-time scheduling in *RT-SBD* has the best performance in terms of PDMR and tuple latency and throughput (even compared to the case with the Best-Fit). RT-SBD (for both of the cases) outperform *Storm*, and also *PFGN*. Generally, experimental results show that the presented real-time streaming big data processing engine provides significant improvements in terms of PDMR, tuple latency, throughput, memory usage and tuple loss.

So, the primary focus of this paper is the velocity dimension of the big data problem, which by the definition, regards to the real-time processing of streaming big data. A major prerequisite for real-time processing is to be fast and the parallel processing and dispatching method we have presented in previous papers are employed for this aim; But the proposed solution is achieved by some contributions that are designed to solve the problem objectively. Some of the most important contributions of this paper are:

- *Deadline-aware dispatching method* as the parallel processing method to provide the required fast processing, necessary to be real-time.
- *Hybrid clustering multiprocessor real-time scheduling algorithm* as the other prerequisite for real-time processing
  - In this proposed real-time scheduling algorithm, both the partitioning and global real-time scheduling approaches are employed to have better schedulablity and resource utilization, with a tolerable overhead.
- *PDMR* (Proportional Deadline Miss Ratio) instead of the traditional DMR, as the most important metric for evaluation of real-time (stream) processing systems.
- Also, the prototype of the proposed real-time streaming big data processing engine is developed.

Since there is a growing need for real-time streaming big data processing systems in industrial applications (e.g., smart cities, oil and gas, industrial automation, cybersecurity, and telecommunication), development of some commercial products relying on the proposed *RT-SBD* may be useful for such applications.

Anyway, some of the future works can be as follow:

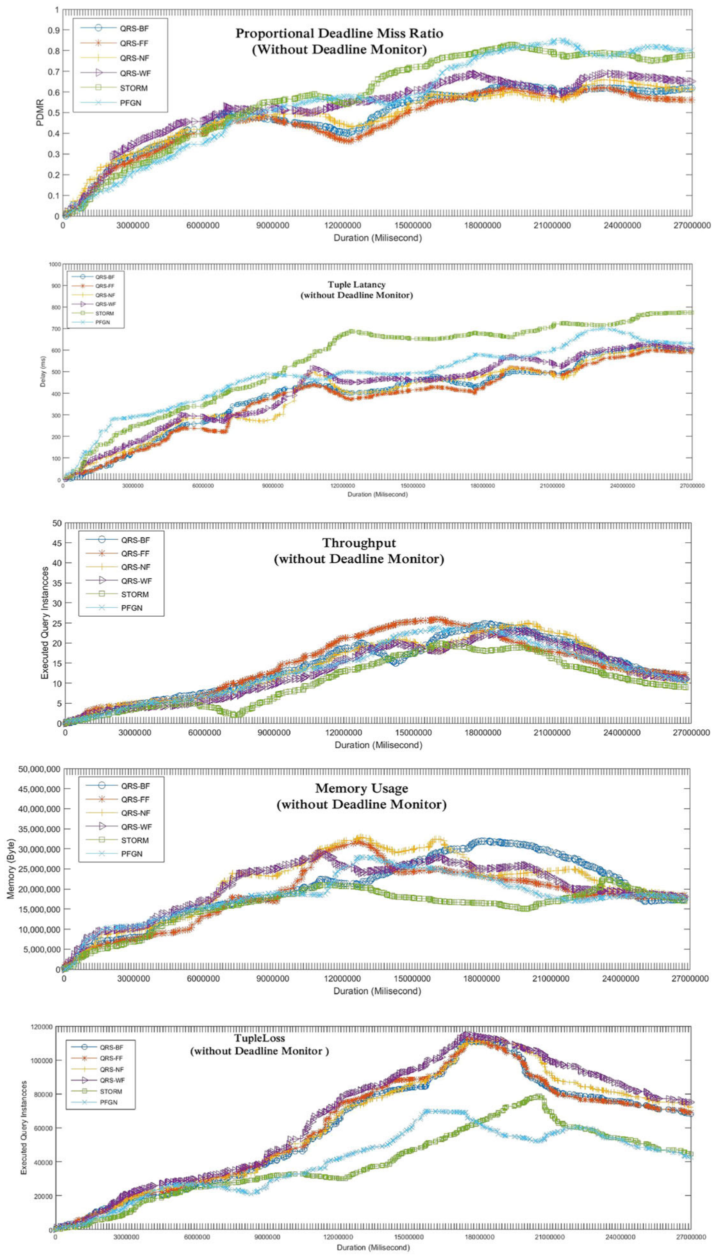
- Extension of *RT-SBD* for concerning other dimensions of big data rather than velocity (such as variety)

- Supporting fault-tolerance and dependability in *RT-SBD*, as most of the time critical applications require to support
- An integrated analysis and formal specification and verification of the proposed hybrid clustering multiprocessor real-time scheduling algorithm e.g., using real-time calculus
- Design and development of a cloud-based version of *RT-SBD* (*RT-SDB* proceeding -As-A-Service) since the 3rd generation of stream processing systems are mostly based on the Cloud infrastructure

Appendix: performance evaluation for different configuration of the contributed system

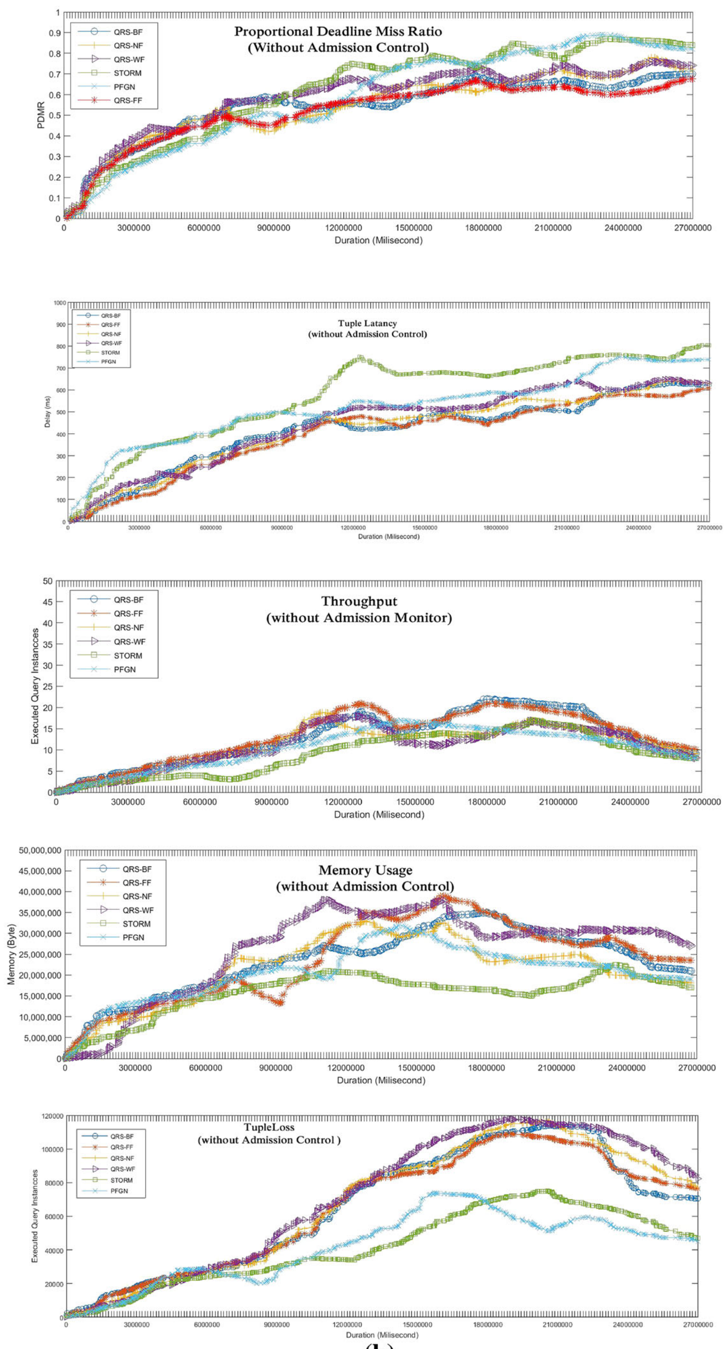
Experiment 5 *measuring performance parameters for the contributed system with different configuration and components.*

A stated in Sect. 4.2—Experimental results, by time-varing vlaues charts (Figs. 8, 9, 10, 11, 12), it may be hard to judge about the performance of the compared alternatives. So, the average value of each parameter for each of the compared configurations and systems, are computed and represented in Figs. 13, 14, 15, and 16. As a complementary experiment, what is the effect of feedback control mechanism used in the *RT-SDB* is issued; e.g., how is the systems performance while eminitin feedback control mechanism. In order to evaluate the contributed system in such other configurations, parameters are measured in the case that the deadline monitor unit, the admision control unit, and also both of them are emmited (Fig. 20a–c, respectively), and the closed-loop control becomes open-loop.



(a)

Fig. 20 Parameters when **a** the admission control unit, **b** deadline monitor unit, and **c** both of them in *RT-SBD* are emitted. **a** Parameters when the deadline monitor unit of *RT-SBD* is emitted, **b** parameters when the admission control unit of *RT-SBD* is emitted, *c* parameters when both (the admission control and deadline monitor units) of *RT-SBD* are emitted



(b)

Fig. 20 continued

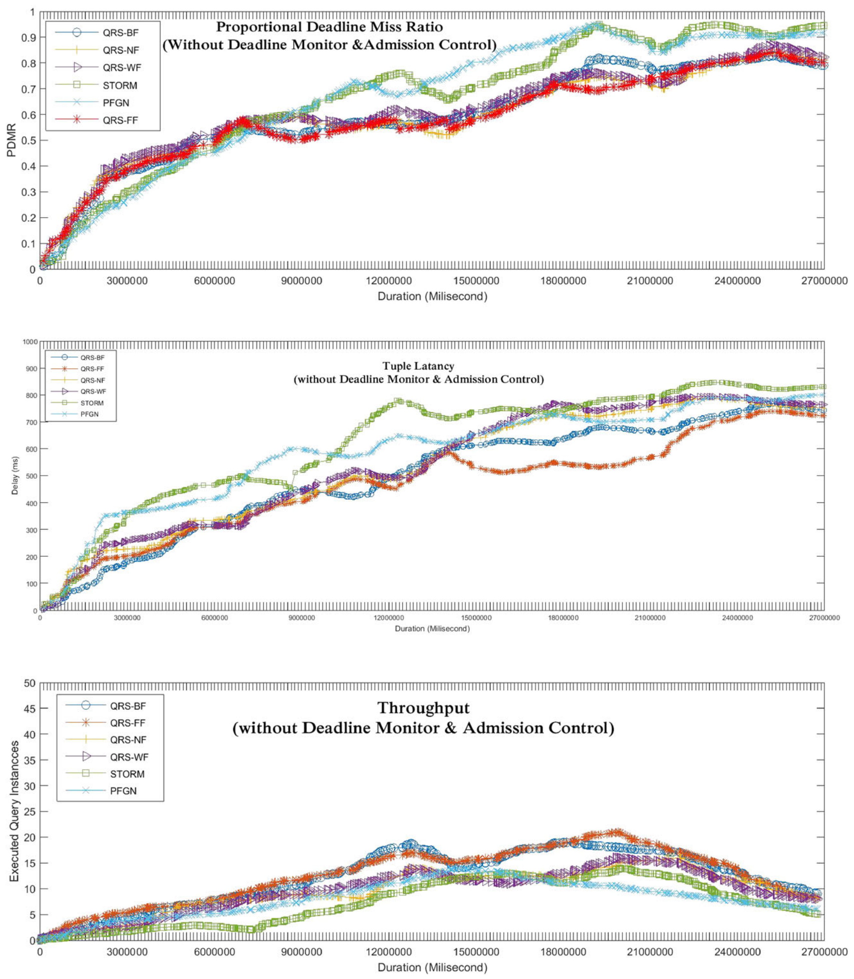


Fig. 20 continued

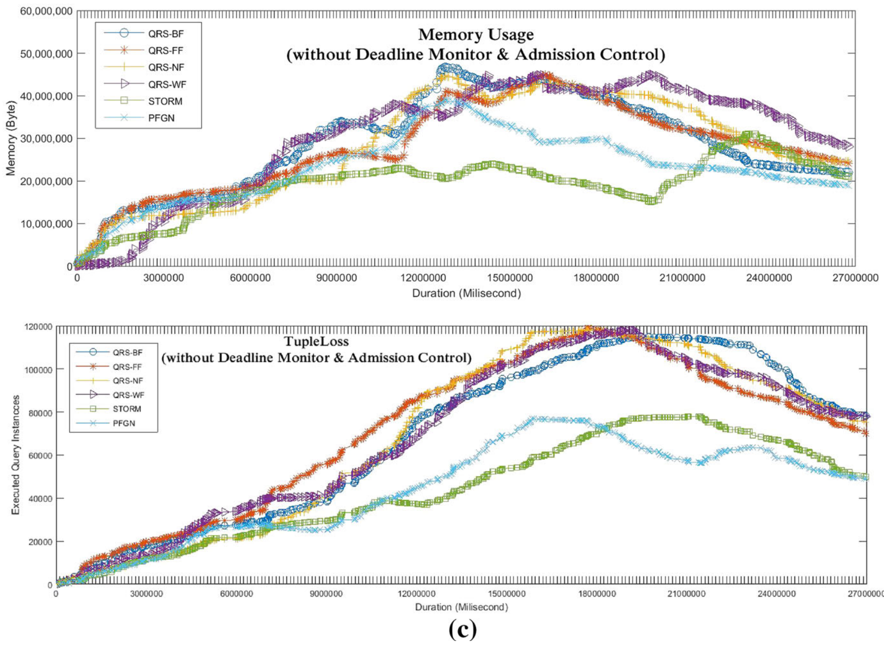


Fig. 20 continued

References

Abadi D et al (2003) Aurora: a new model and architecture for data stream management. VLDB J 12(2):120–139

Alemi M, Safaei AA, Hagjhoo MS, Abdi F (2011) PDMRTS: multiprocessor real-time scheduling considering process distribution in data stream management system. In: International conference on digital information and communication technology and its applications, pp 166–179

Anderson J, Srinivasan A (2000) Early release fair scheduling. In: Proceedings of the euromicro conference on real-time systems. IEEE Computer Society Press, Stockholm, pp 35–43

Anderson J, Srinivasan A (2004) Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. J Comput Syst Sci 68(1):157–204

Andersson B, Jonsson J (2003) The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50 percent. In: 15th euromicro conference on real-time systems (ECRTS'03), Porto, Portugal, 02–04 July

Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In: Proceedings of the international conference on embedded and real-time computing systems and applications (RTCSA)

Arasu A et al (2004) Linear road: a stream data management benchmark. In: Proceedings of the thirtieth international conference on very large data bases, vol 30. VLDB Endowment

Åsberg M et al (2012) Exsched: an external cpu scheduler framework for real-time systems. In: 2012 IEEE 18th international conference on embedded and real-time computing systems and applications (RTCSA). IEEE

Astrom KJ, Hagglund TH (1995) New tuning methods for PID controllers. In: Proceedings of the 3rd European control conference

Babcock B et al (2003) Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems

Babcock B et al (2004) Load shedding for aggregation queries over data streams. In: International conference on data engineering (ICDE)

Babcock B et al (2004) Operator scheduling in data stream systems. VLDB J 13(4):333–353

Bans JM, Arenas A, Labarta J (2002) Efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In: PDPTA, pp 809–815

Baruah N et al (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica 15:600–625

Baruah S, Gehrke J, Plaxton C (1995) Fast scheduling of periodic tasks on multiple resources. In: Proceedings of the 9th international parallel processing symposium, April 1995, pp 280–288

Beloglazov A, Abawajy J, Buyya R (2012) Energy-aware allocation heuristics for efficient management of data centers for cloud computing. Future Gener Comput Syst 28:755–768

Bestavros A, Nagy S (1996) An admission control paradigm for real-time databases. Technical Report BUCS-TR-96-902, Computer Science Department, Boston University, Boston

Bestavros A, Nagy S (1996) Value-cognizant admission control for RTDB systems. In: IEEE 16th real-time systems symposium, December 1996

Block A et al (2008) An adaptive framework for multiprocessor real-time system. In: Euromicro conference on real-time systems (ECRTS'08). IEEE

Bollella G, James G (2000) The real-time specification for Java. Computer 33(6):47–54

Bu Y et al (2010) HaLoop: efficient iterative data processing on large clusters. Proc VLDB Endow 3(1–2):285–296

Carpenter J et al (2004) A categorization of real-time multiprocessor scheduling problems and algorithms. In: Handbook on scheduling: algorithms, models and performance analysis

Chen Philip CL, Zhang C-Y (2014) Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci 275:314–347

Condie T et al (2010) MapReduce Online.In: NSDI, vol. 10, no. 4

Devi UC (2006) Soft real-time scheduling on multiprocessors. Ph.D. Thesis, University of North Carolina at Chapel Hill

Dhall S, Liu C (1978) On a real-time scheduling problem. Oper Res 26:127–140

Elliott GA, Ward BC, Anderson JH (2013) GPUSync: a framework for real-time GPU management. In: 2013 IEEE 34th real-time systems symposium (RTSS). IEEE

Golab L (2004) Querying sliding windows over on-line data streams. In: Proceedings of ICDE/EDBT Ph.D. workshop, March, pp 1–10

Golatowski F et al (2002) Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations. In: Proceedings of the 13th IEEE international workshop on rapid system prototyping, 2002. IEEE

Graefe G et al (1994) Extensible query optimization and parallel execution in volcano. Query processing for advanced database systems, Morgan Kafman

Holman P, Anderson J (2006) Group-based pfair scheduling. Real Time Syst 32(1–2):125–168

http://www.espertech.com/esper/. Accessed 20 May 2016

https://storm.apache.org/. Accessed 20 March 2015

Jamin S et al (1993) An admission control algorithm for predictive real-time service. LNCS 712(1993):347–356

Johnson T et al (2008) Query-aware partitioning for monitoring massive network data streams. In: SIGMOD

Kato S, Yamasaki N (2007) Real-time scheduling with task splitting on multiprocessors. In: Proceedings of the international conference on embedded and real-time computing systems and applications, pp 441–450

Kato S, Yamasaki N (2008) Portioned EDF-based scheduling on multiprocessors. In: Proceedings of the international conference on embedded software, pp 139–148

Kato S, Yamasaki N (2008) Scheduling aperiodic tasks using total bandwidth server on multiprocessors. EUC, vol 1, pp 82–89

Kleiminger W, Kalyvianaki E, Pietzuch P (2011) Balancing load in stream processing with the cloud.In: 2011 IEEE 27th international conference on data engineering workshops (ICDEW). IEEE

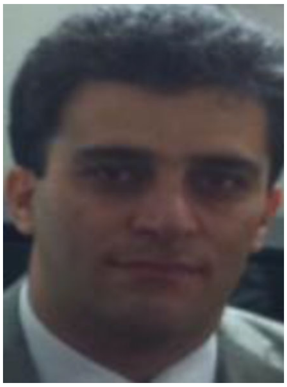
Kontaki M (2010) Continuous processing of preference queries in data streams. In: 36th international conference on current trends in theory and practice of computer science (SOFSEM)

Kramer J (2009) Continuous queries over data streams- semantics and implementation. kra

Kulkarni S et al (2015) Twitter heron: stream processing at scale. In: Proceeding of the ACM SIGMOD'15, pp 239–250

Kwon J, Cho H, Ravindran B (2012) A framework accommodating categorized multiprocessor real-time scheduling in the RTSJ. In: Proceedings of the 10th international workshop on java technologies for real-time and embedded systems. ACM

Lakshmanan K et al (2009) Partitioned fixed-priority preemptive scheduling formulti-core processors. In: Proceedings of the euromicro conference on real-time systems, pp 39–248
Lam W et al (2012) Muppet: MapReduce-style processing of fast data. Proc VLDB Endow 5(12):1814–1825
Lehner W, Sattler K-U (2013) Web-scale data management for the cloud. Springer, Berlin
Leontyev H (2010) Compositional analysis techniques for multiprocessor soft real-time scheduling. Ph. D. Thesis, University of North Carolina at Chapel Hill
Li X, Wang HA (2007) Adaptive real-time query scheduling over data streams. VLDB '07, 23–28 September, Vienna
Lopez J, Garcia M, Diaz J, Garcia D (2000) Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In: Proceedings of the 12th euromicro conference on real-time systems, June, pp 25–33
Ma L et al (2009) Real-time scheduling for continuous queries with deadlines. SAC'09, Honolulu, HI
Marisol G-V, Tommaso C, Chenyang L (2014) Challenges in real-time virtualization and predictable cloud computing. J Syst Architect 60:726–740
Mohammadi S (2010) Continuous query response time improvement based on system conditions and stream featuress. M.Sc. Thesis, Iran University of Science and Technology
Neumeyer L et al (2010) S4: distributed stream computing platform. 2010 IEEE international conference on data mining workshops (ICDMW). IEEE
Regehr J, Stankovic JA (2001) HLS: a framework for composing soft real-time schedulers. In: Proceedings of the 22nd IEEE real-time systems symposium (RTSS 2001). IEEE
Safaei AA, Haghjoo MS, Abdi F (2011) PFGN: a hybrid multiprocessor real-time scheduling algorithm for data stream management systems. In: Proceeding of international conference on digital information and communication technology and its applications, pp 180–192
Safaei AA, Alemi M, Haghjoo MS, Mohammadi S (2011) Hybrid multiprocessor real-time scheduling approach. Int J Comput Sci Issues 8(2):171
Safaei AA, Sharif-Razavian A, Sharifi M, Haghjoo MS (2012) Dynamic routing of data stream tuples among parallel query plan running on multi-core processors. J Distrib Parallel Databases 30(2):145–176. doi:10.1007/s10619-012-7090-6
Safaei AA, Haghjoo MS (2010) Parallel processing of continuous queries over data streams. Distrib Parallel Databases 28(2–3):93–118. doi:10.1007/s10619-010-70663
Safaei AA, Haghjoo MS (2012) Dispatching of stream operators in parallel execution of continuous queries. J Supercomput 61(3):619–641. doi:10.1007/s11227-011-0621-5
Safaei AA, Haghjoo MS (2014) Parallel processing of data streams. J Comput Sci Eng 11(2):11–29
Srinivasan A (2003) Effcient and flexible fair scheduling of real-time tasks on multiprocessors. Ph.D. Thesis, University of North Carolina, Chapel Hill
Srinivasan A, Anderson JH (2004) Efficient scheduling of soft real-time applications on multiprocessors. J Embed Comput 1(3):1–14
Stankovic JA et al (1999) Misconceptions about real-time databases. J Comput 32(6):29–36
Stankovic JA, Ramamritham K (1990) What is predictability for real-time systems? Real Time Syst 2(4):247–254
Stonebraker M et al (2005) The 8 requirements of real-time stream processing. SIGMOD Rec 34(4):42–47
Tatbul N et al (2003) Load shedding in a data stream manager. In: Proceedings of VLDB, pp 309–320
The STREAM Group (2003) STREAM: the Stanford stream data manager. IEEE data engineering bulletin, March 2003
Valls MG, Lopez IR, Villar LF (2013) iLAND: an enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. IEEE Trans Ind Inform 9(1):228–236
Wei Y et al (2007) QoS management of real-time data stream queries in distributed environments. In: IEEE international symposium on object-oriented real-time distributed
Wei Y, Son SH, Stankovic JA (2006a) RTSTREAM: real-time query processing for data streams. In: 9th IEEE international symposium on object/component/service-oriented real-time distributed computing, pp 141–150
Wei Y, Prasad V, Son SH, Stankovic J (2006b) Prediction-based QoS management for real-time data stream. In: Proceedings of IEEE real-time systems symposium (RTSS'06), December
Yang, H et al (2007) Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD international conference on management of data. ACM
Yang Q, Koutsopoulos HN (1996) A microscopic traffic simulator for evaluation of dynamic traffic management systems. Transp Res C 4(3):113–129



Ali A. Safaei was born in Semnan, Iran, in 1979. He received the B.Sc. and M. Sc. degrees in computer engineering in 2001 and 2004, respectively. He also received the Ph.D. degree in computer engineering from the Iran university of Science and Technology, Tehran, Iran, in 2011. From 2012 to 2014 Dr. Safaei served at Computer Engineering department of K. N. Toosi University of Technology, as an Assistant Professor. In 2014, he joined the Department of Bio-Medical Informatics, Tarbiat Modares University as an Assistant Professor. His current research interests include parallel and real-time processing of data streams, complex-event processing, big data management, medical data management, and crowdsourcing in information retrieval. He has published two books and more than 40 papers in the field of data management.