Distrib Parallel Databases (2010) 28: 93–118
DOI 10.1007/s10619-010-7066-3

# Parallel processing of continuous queries over data streams

**Ali A. Safaei · Mostafa S. Haghjoo**

Published online: 29 June 2010


**Abstract** In this paper, we propose parallel processing of continuous queries over data streams to handle the bottleneck of single processor DSMSs. Queries are executed in parallel over the logical machines in a multiprocessing environment. Scheduling parallel execution of operators is performed via finding the shortest path in a weighted graph called *Query Mega Graph* (*QMG*), which is a logical view of $K$ machines. By lapse of time, number of tuples waiting in queues of different operators may be very different. When a queue becomes full, re-scheduling is done by updating weight of edges of *QMG*. In the new computed path, machines with more workload will be used less. The proposed system is formally presented and its correctness is proved. It is also modeled in PetriNets and its performance is evaluated and compared with serial query processing as well as the *Min-Latency* scheduling algorithm. The presented system is shown to outperform them w.r.t. tuple latency (response time), memory usage, throughput and also tuple loss- critical parameters in any data stream management systems.

**Keywords** Query plan · Data stream · Parallel execution · Tuple latency

## 1 Introduction

In many recent applications, data is not stored (as in traditional DBs) but is received as streams which are infinite, rapid, unpredictable and time-varying sequences of

Communicated by Ahmed K. Elmagarmid.

A.A. Safaei (✉) · M.S. Haghjoo
Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
e-mail: safaeei@iust.ac.ir

M.S. Haghjoo
e-mail: haghjoom@iust.ac.ir

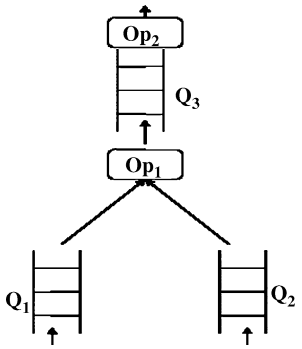



**Fig. 1** A query plan generated for a query on data stream

data elements. Traditional DBMSs are not able to process data streams with such characteristics. To handle this problem, data stream management systems (DSMSs) are introduced [1–3].

On the other hand, despite traditional DBMSs, queries are continuously executed over the non-stopping data streams which are rapid and bursty [4, 5].

The above conditions give rise to the fact that a single processor is not capable to process this volume of data and execute continuous queries over them with satisfactory speed. In other words, a single processor used for serial execution of queries in DSMSs causes reduction of output performance and is compelled to discard data stream elements [6].

In this paper, parallel execution of queries based on a multiprocessor platform is presented to eliminate system bottleneck (*i.e.,* single processor). Parallelism in query processing offers better performance, availability and extensibility [7].

Parallelism in query processing consists of the following phases:

*I. Converting query into query plan*

In a DSMS, a continuous query plan is decomposed into a set of operators such as project, select, join, etc. as in a traditional DBMS. The output of one operator is buffered in a queue, which consequently acts as the input to another operator (if not the final one) (Fig. 1) [1, 2]. The query plan can be conceptualized as a directed acyclic graph (DAG) in which a node represents a pipelined operator and a directed edge between two nodes represents the queue connecting those two operators. The edge also represents precedence relationship between nodes.

Regarding mass volumes of data stream in most applications, a negligible improvement in query plan or its parallel execution causes considerable performance improvement.

Each tuple of data stream traverses a sequence of operators in the query plan, called "*operator path*" [8].

Generating and optimizing query plan is discussed in [9–11].

*II. Parallel execution of sub-queries*

Generally, parallelism could be achieved via two approaches [12]:

- *Parallel data* (data partitioning): data set is partitioned into some fragments and each one is assigned to a machine. So, all machines do identical processes on distinct (partitioned) data.

- *Parallel control*: the whole set of data is delivered to each machine which performs parts of the total task on the whole set of data. In other words, data set is shared among machines but each one performs a distinguished task on it.

Since the whole set of data stream is *not* available in most applications, in this paper we present parallel query processing of data streams using parallel control approach.

Our main contributions are:

- Parallel processing of continuous queries in DSMSs based on a multiprocessing platform, together with its system architecture (Sect. 2).
- Operator scheduling algorithm for parallel execution of query plan (Sect. 3).
- Parallel query processing algorithm in the proposed data stream management system (Sect. 4).
- Complexity analysis of the presented parallel query processing algorithm (Sect. 5).
- Resource allocation of multiple query processing (Sect. 6).
- Presented system is modeled by the Petri Networks and its performance is evaluated (Sect. 7).

Finally, we consider related work in Sect. 8 and conclude in Sect. 9.

## 2 Parallel query processing

Queries in data stream management systems are usually continuous. For a registered query, DSMS generates its query plan. With respect to continuous nature of query and data stream [13, 14], each operator must process data stream tuples continually while there are several operators in a query plan. So, a single processor is *not* able to execute operators on data stream tuples simultaneously and should serialize them. This is a serious bottleneck in fast query processing. In order to handle this bottleneck, in this paper, parallel query processing of data streams is presented. The proposed system architecture is illustrated in Fig. 2.

In the parallel query processing engine, there are $k$ identical processors (or logical machines) collaborating each other. Machines might be physical (*e.g.* processors in a multi-processor system or nodes of a cluster) or logical (*e.g.* threads running on a multi-core CPU). Details of the desired platform are out of the scope of this paper. We assume that in a multiprocessing environment, we have $k$ identical processes (called "*logical machines*" or "*machines*" hereafter) which can work in parallel with collaboration. These machines prepare desired platform to process continuous queries in parallel.

In order to process queries in parallel on these $k$ machines, query plan is generated and identical copies of it are sent to each one of them. This informs all of the machines about the operators and their preferences. Two jobs are remained: assigning operators to each machine and how the machines collaborate with each other. These two jobs are handled by scheduling algorithm in Sect. 3.

*Example 1* Consider query $Q_1$ and 3 logical machines below. As shown in Fig. 3, a copy of query plan (Fig. 3.a) is assigned to each logical machines (Fig. 3.b):

$$\mathbf{Q_1}: \textbf{\textit{SELECT}}\ x\ \textbf{\textit{FROM}}\ s\ \textbf{\textit{WHERE}}\ p$$

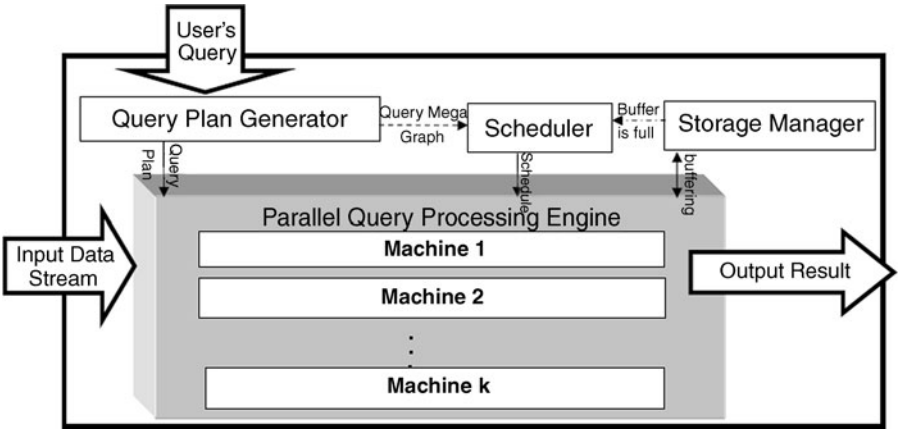
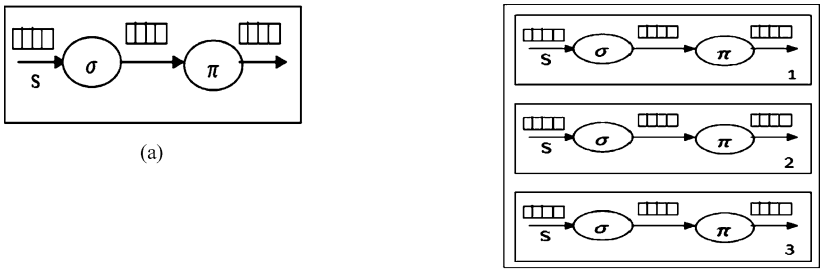


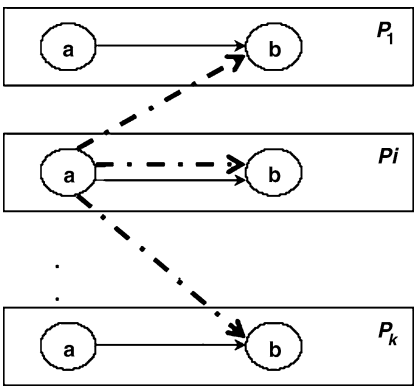

**Fig. 2** The proposed system architecture for parallel query processing over data stream




(a) (b)

**Fig. 3** (**a**) Query plan for query $Q_1$ and (**b**) assigning 3 copies to each logical machine

**Fig. 4** Collaboration of logical machines for processing operators of query plan




For logical machines to collaborate with each other, 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. 4).

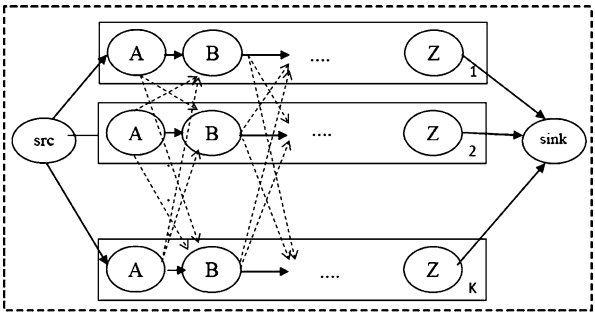



**Fig. 5** *Query Mega Graph* created from a query plan including operators $A$–$Z$ over $K$ logical machines

Accordingly, query plan generator unit generates *Query Mega Graph* and gives it to the scheduler unit for parallel query scheduling over $k$ existing logical machines as elaborated below.

## 2.1 Query Mega Graph (QMG)

### *2.1.1 Generating Query Mega Graph*

A DAG is created by generating $k$ identical copies of a query plan and duplicating edges between all machines. Two special nodes are assumed: a *source* and a *sink* node before and after nodes of the main query plan respectively. An edge originating from the source node links it to each node in first nodes of the main query plan. Also, an edge links last nodes of the main query plan to the sink node. The created graph is called *Query Mega Graph (QMG)*. Figure 5 illustrates a *QMG* created from a query plan including operators $A$–$Z$ over $k$ logical machines.

In a *Query Mega Graph*, an edge also represents the queue between its two operators. The weight of the edge is *number of data elements* in the queue. Hence, *Query Mega Graph* is a weighted DAG which also shows work load distribution in the system and is used for scheduling operator execution to achieve parallel query processing as elaborated in Sect. 3.

### *2.1.2 Formal definition*

**Definition 1** A query plan is a DAG $G_{QP} = \langle V, E \rangle$ in which:

$$V = \{o \mid o \in \text{stream query processing operators}\}$$

$$E = \{\langle A, B \rangle \mid A, B \in V\}$$

**Lemma 1** *Query plan is a multi-stage graph.*

*Proof* Each operator in a query plan processes tuples received from its immediate pre-operator and delivers results to its immediate post-operator. Operators of query plan can be partitioned into distinct stages in which each edge of query plan graph *only* connects a node from stage $i$ to a node in stage $i + 1$ due to standard relational operators execution. □

**Notation** $O_j^i$ denotes operator $O$ which is the $i^{\text{th}}$ operator of main query plan in machine $j$. Accordingly, $O_-^i$ denotes the $i^{\text{th}}$ operator of main query plan, $O_j^-$ means operator $O$ in machine $j$ and $O_-^{|V|}$ denotes the last operator of main query plan.

**Definition 2** *Query Mega Graph* created from $G_{QP} = \langle V, E\rangle$ is a triple $QMG = \langle V', E', W'\rangle$ such that:

$$V' = \{src, sink\} \cup \bigcup_{\substack{1\le i\le |V|\\ 1\le j\le K}} O_j^i$$

$$\begin{aligned} E' = {} & E \cup \langle x, y\rangle | \forall A, B \in V(\langle A, B\rangle \in E) \\ & \Rightarrow \quad (\forall i = 1, 2, \ldots, k, \forall j = 1, 2, \ldots, k, i \neq j(\langle A_i^-, B_j^-\rangle \in E' \\ & \wedge (x = A_i^- \wedge y = B_j^-))) \vee \big(\forall i = 1, 2, \ldots, k\big(\big(x = src \wedge y = O_i^1\big) \\ & \vee \big(x = O_i^{|V|} \wedge y = sink\big)\big)\big)\big\} \\ W' : {} & E' \to \mathbb{Z}^+ \quad \text{such that } \forall a, b \in V'(W'(a, b) = q\_count(a, b)) \end{aligned}$$

$q\_count(a, b)$ returns number of tuples waiting in queue of edge $(a, b)$.

**Note** Number of nodes and edges in *QMG* can be computed as:

$$|V'| = (|V|.K) + 2 \tag{1}$$

$$|E'| = (|E|.K) + (K - 1).\sum_{i=1}^{|V|} fan\_out_i + 2K$$

$fan\_out_i$ is the number of edges outgoing from node $i$.

**Theorem 1** *Query Mega Graph is semantically equivalent to the main query plan.*

*Proof* We must show that each operator path existing in *QMG* is equivalent to operator path of the main query plan and vice-versa. We show that each edge in *QMG* is equivalent to an edge in the main query plan *or* it doesn't make any disorder in processing the query.

With regard to formal definition of *QMG*, for edges not connected to the *source* or *sink* we have:

$$\begin{aligned} & \forall x, y((\langle x, y\rangle \in E') \wedge (x \neq src \wedge y \neq sink)) \\ & \quad \Rightarrow \quad \exists i, j, k, l, A, B\big(\big(x = A_j^i \wedge y = B_k^l\big) \wedge \langle A_-^i, B_-^l\rangle \in E'\big) \end{aligned}$$

Also, edges in *QMG* that are connected to the *source* or *sink*, shift tuples to proper nodes only (*i.e.,* received tuples to the first operator the main query plan and deliver

result tuples from the last operator of main query plan to user) without any extra processing:

$$\begin{aligned}&\forall x, y((\langle x, y\rangle \in E') \wedge (x = src \vee y = sink))\\ &\quad\Rightarrow \quad \nexists X(\forall O \in V((\langle src, X\rangle \in E' \wedge \langle X, O^1_-\rangle \in E')\\ &\quad\vee (\langle O^{|V|}_-, X\rangle \in E' \wedge \langle X, sink\rangle \in E')))\end{aligned}$$ □

To process a tuple as fast as possible by the query plan, we should employ operator path in *Query Mega Graph* with the minimum *tuple latency* (or the shortest path in *QMG*).

**Lemma 2** *The difference between tuple latency of each two various operator paths in QMG, is dependent only on the total delay of edges establishing each operator path* (*independent of nodes of operator path*).

*Proof* Amount of latency a tuple is confronted while traversing an operator path generally can be considered as processing delays (by operators *i.e.,* nodes of graph) in addition to queuing delays (by queues *i.e.,* edges of graph) [32]:

$$\forall t \in data\_stream\_tuples$$

$$tuple\_latency(t) = \sum_{1\le i\le|V|} cost\left(O^i_-\right) + \sum_{1\le i\le|V|} \left(Buffer\_size\left(O^i_-\right)\right)$$

With respect to Theorem 1, for each tuple, every operator path existing in *Query Mega Graph* is equal to operator path which it should traverse in the main query plan. So, paths a tuple can use are the same in terms of operators, preference of operators and total operators delay. Therefore, sum of operator delays is equal in various operator paths. Hence, tuple latency in each path of *QMG* is dependent only on delay of edges establishing that path. □

**Theorem 2** *The shortest path in Query Mega Graph has minimum tuple latency.*

*Proof* According to Definition 2 and Lemma 2, amount of latency a tuple confronts in traversing a path is equal to total delay it suffers in queues between operators of that path, which is equal to the total number of data elements in queues (*i.e.,* sum of weight of edges establishing the path). By definition, the shortest path in *Query Mega Graph* has minimum total weight of edges (or minimum number of data elements waiting in queues). So, data stream tuples traversing this path suffers minimum tuple latency. □

## 3 Scheduling algorithm

Generating *Query Mega Graph* from main query plan enables parallel query processing. In order to employ this feasibility, an appropriate scheduling for parallel execution should be established. Since, among feasible operator paths in *QMG*, minimum

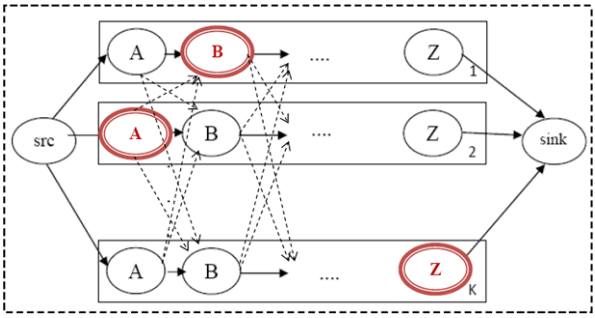



**Fig. 6** A sample outcome of *Dijkstra's* shortest path algorithm over *QNG* of Fig. 5

tuple latency will be achieved from the shortest path (Theorem 2), this scheduling algorithm determines the shortest path in *QMG*. In fact, *QMG* is a logical view of $k$ logical machines, each of which has the capability of executing the query plan. But to use them for executing query operators in parallel, we schedule operators and dispatch parallel query plan execution to the machines.

Operator scheduling algorithm in the presented system mainly consists of finding the shortest path in *Query Mega Graph* from *source* to *sink*. For instance, *Dijkstra's* shortest path algorithm is applicable here.

The *Dijkstra's* shortest path algorithm finds a path as a sequence of query plan operators with nodes distributed over $k$ existing logical machines. Figure 6 shows a sample outcome of *Dijkstra's* shortest path algorithm over *Query Mega Graph* of Fig. 5.

According to the proposed architecture (Fig. 2), the scheduler finds shortest path in *Query Mega Graph* received from query plan generator unit and sends a triple $\langle p, o, s\rangle$ (*predecessor* machine, *operator_id*, *successor* machine) to each machine. For example in Fig. 6, after determination of the shortest path, triple $\langle 2, B, 5\rangle$ is sent to machine 1.

Each machine receives data tuples from its *predecessor*, executes desired operators on them, and sends the result tuples in to its *successor* machine.

**Theorem 3** *Outcome of finding shortest path in Query Mega Graph is tantamount to parallel scheduling of query plan operators.*

*Proof* Assigning copies of query plan to logical machines just helps them to be aware of operators and their precedence. Outcome of finding shortest path in *Query Mega Graph* is an operator path that is equivalent to the main query plan (Theorem 1) and is used to execute query plan in parallel. So, this path is the main query plan in runtime which is distributed over existing logical machines. In other words, query plan is no longer executed on a single machine, but is executed in parallel over $k$ existing logical machines. Moreover, operators are assigned to machines such that query processing has minimum tuple latency, because of using the shortest path in *Query Mega Graph*. □

Accordingly, we use terms *Dijkstra's* shortest path algorithm and our proposed parallel operator scheduling algorithm interchangeably hereafter.

### 3.1 Initial scheduling

Initially, all of the edges in *QMG* have zero weights (which means queues are empty). Hence, the path found by routing algorithm (*i.e., Dijkstra's* shortest path algorithm) might be placed entirely on *one* machine. In order to balance workload over all of the machines at the beginning, initializing weight of edges in *QMG* could be done such that routing algorithm results in a path distributed over all existing machines. To avoid the overload of assigning weights to edges of *QMG*, operators are initially assigned manually (*i.e.,* without using routing algorithm) to machines using the following equation:

$$j = i \bmod k \tag{2}$$

($j$: machine's *Id*, $i$: operator's *Id*, $k$: number of existing logical machines).

### 3.2 Re-scheduling

We need to re-schedule because different operators in query plan (*e.g. selection*, *projection* and *join*) need different execution time *i.e.,* some operators (*e.g. projection*) are faster than some others (*e.g. join* that is a blocking operator in general). By arriving more and more tuples, system will be led to a status that some queues are full while some others are nearly empty. For example, input queue of a join operator may have 8 tuples whilst a projection operator has 2 tuples in its input queue. Hence, by lapse of time, weight of different edges in *QMG* may be very different.

In common DSMSs, when input queue of an operator becomes full, subsequent tuples are dropped. In our presented system, re-scheduling (re-routing in *QMG*) is done to handle overload situations and to improve system performance. For example, suppose that input queue of an operator $o$ in machine $A$ is full. In this situation, re-scheduling is done and new shortest path is generated in which operator $o$ is not located in machine $A$ anymore. At this leisure, machine $A$ can process remaining tuples in its input queue.

This is a dynamic load balancing (instead of load shedding) in which system tries to process intermediate tuples (which some amount of system time has spent for processing them) instead of discarding them.

Updating weight of edges in *QMG* before re-routing should be done with load balancing considerations. For example if operator $a$ in machine $A$ has $m$ tuples and operator $b$ in machine $B$ has $n$ tuples, such that $n \ll m$, then after rescheduling, operators $a$ and $b$ are swapped between machines. It means that for new arriving tuples, $a$ must be executed on $B$ and $b$ on $A$ while concurrently, execution of $a$ in $A$ and $b$ in $B$ would be continued for old tuples remaining in their input queues. So, after re-routing in *QMG,* new arriving data stream tuples will traverse newly found shortest path whilst the remaining tuples in input queues of old path will traverse the same old path concurrently. In this way, machines that were executing *heavy* operators are assigned *light* operators and complete processing on tuples remaining in their input queues.

Having different paths, may cause tuples to lose their incoming orders *i.e.,* a new incoming data stream tuple may traverse a path with nearly empty queues and reach

to the *sink* node whilst some tuples with lower input timestamp are still waiting in queues of the old shortest path. This is not acceptable and needs to preserve input ordering of data stream tuples (Sect. 3.4).

It may look better to redirect tuples remaining in input queues of the old path into beginning of queues of the new shortest path, after re-scheduling, because if so, incoming order of data stream tuples would be preserved and also there is just one operator for each machine to execute. This is not reasonable as proved in the following theorem.

**Theorem 4** *Moving tuples from input queues of the old path to front of corresponding queues in the new path is the same as using the old path.*

*Proof* By moving tuples from input queues of the old path to front of corresponding queues in the new path, all of the tuples in a queue in the old path are moved to front of queue of corresponding operator in the new path. So, each new tuple arriving to this queue should wait until all old tuples are processed. Therefore, tuple latency for new arriving tuples is equal to tuple latency in the old path. This is the same as using the old path, *i.e.* re-scheduling has no effect. □

According to Theorem 4, after re-scheduling (re-routing in *QMG*) new arriving data stream tuples will traverse newly found shortest path whilst tuples waiting in queues of operators in the old path continue traversing old path, concurrently.

### 3.3 Weight updating in *QMG*

When re-scheduling becomes necessary, weight of edges in *QMG* should be updated before, to decrease workload of more busy machines to let them complete their remaining work. Accordingly, the total number of tuples waiting in input queues of all operators in each machine $A$ is accumulated with the weight of *ALL* edges in *QMG* with destination node in $A$, *i.e.*:

is used as weight of edges in re-routing *QMG* (re-scheduling)

### 3.4 Preserving order of tuples

In most data stream applications, it is desirable to preserve incoming ordering of data stream tuples when delivered as system output. As input data stream tuples order may be changed during query processing of the presented parallel system, we must assure that output tuples preserve incoming order of data stream tuples. The problem is solved by buffering and sorting output tuples before delivering. Using sliding window mechanism, $N$ output tuples are buffered and stored in ascending order of their input timestamp.

An example of output queue is illustrated in Fig. 7.

Output queue is part of system's limited memory space. So, its size and efficient management is very important.

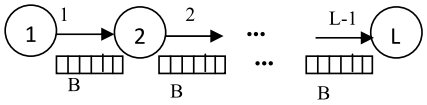
Also, complexity and overhead of sorting output queue is a challenging issue in system performance.


**Fig. 7** An example of output queue

**Theorem 5** *To preserve incoming order of data stream tuples, output queue should have a minimum size of* $(L-1) \times B$.

$L$: *number of levels in main query plan*
$B$: *size of buffers (queues) between each two operators*

*Proof* Incoming order of tuples may be modified in situations after re-routing when a queue in the path is full. A new path is selected for new arriving tuples while old tuples (waiting in queues) traverse the same old path. A query plan including $L$ levels has $L-1$ edges in its corresponding operator path. Each edge has a queue of size $B$ (below):




□

In some applications preserving order of input streams is not important. In such cases, we can relax the value $(L-1) \times B$ via modeling and simulation (Sect. 7). Moreover, the overhead of sorting output buffer can be relaxed via *Insertion* algorithm. For each tuple, it just finds the proper location in output buffer and inserts the tuple. Also, circular queue is a good mechanism for output buffer management here.

## 4 Parallel query processing algorithm

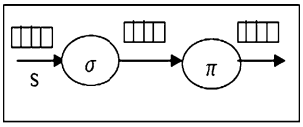
According to the proposed system architecture and the above discussions, parallel query processing algorithm is stated below.

For example, assume that query $Q_1$ is submitted to be processed in parallel over 4 logical machines.

$$\mathbf{Q_1}: \textbf{\textit{SELECT}}\ x\ \textbf{\textit{FROM}}\ s\ \textbf{\textit{WHERE}}\ p$$

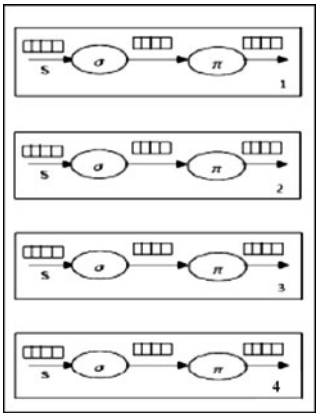
According the parallel query processing algorithm, figures show what is happening step by step:

(1) *Generate query plan for registered query.*

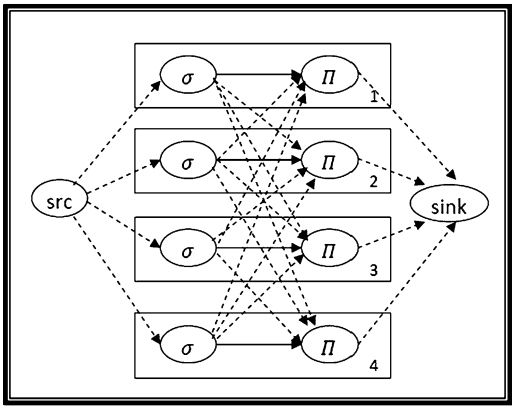


1. *Generate query plan for registered query*
2. *Generate k identical copies of query plan and send each one to one of the machines*
3. *Generate Query Mega Graph*
4. *Initial scheduling: assign each operator of the main query plan to the corresponding machine according to* (1) *(send proper triple* $(p, o, s)$ *to each machine)*
5. *Repeat until there is no more input tuple or query expired:*
   - *5.1. Process tuples (by each operator (machine))*
   - *5.2. Buffer and sort output tuple and deliver them to user or application*
   - *5.3. If a queue is full, then:*
     - *5.3.1. Update weight of edges in QMG*
     - *5.3.2. Re-scheduling (re-routing in QMG): find the shortest path in QMG and send proper triple* $(p, o, s)$ *to each machine in the new path*
     - *5.3.3. New arriving data stream tuples traverse new computed path while the old ones (tuples waiting in queues of the old path) traverse the same old path*

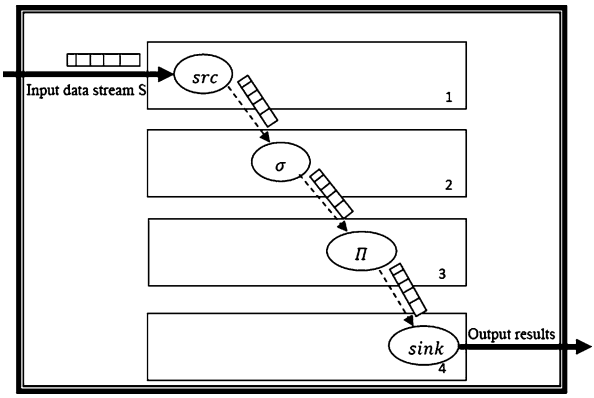
(2) *Generate k identical c Generate k identical copies of query plan and send each one to one of the machines.*




(3) *Generate Query Mega Graph.*



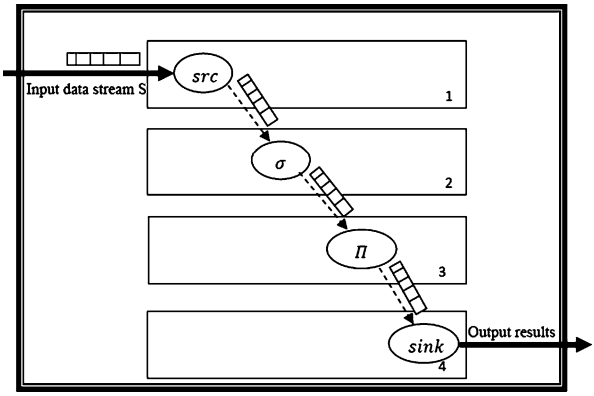
(4) *Initial scheduling: assign each operator of the main query plan to the corresponding machine according to (1) (send proper triple ( p , o , s ) to each machine).*




(5) *Repeat in parallel:*

*5.1 Process tuples (by each operator)*

*5.2 Buffer and sort output tuple and deliver them to user or application (by the sink)*




*5.3. If a queue is full, then:*

*5.3.1. Update weight of edges in QMG*

*{since selectivity and execution time of σ are so much more w.r.t. Π, by lapse of time, system will led to a situation in which input queue of σ is full whilst others are partially empty}*

*Suppose:*

$$w(src_1^-, \sigma_2^-) = 10$$

$$w(\sigma_2^-, \Pi_3^-) = 2$$

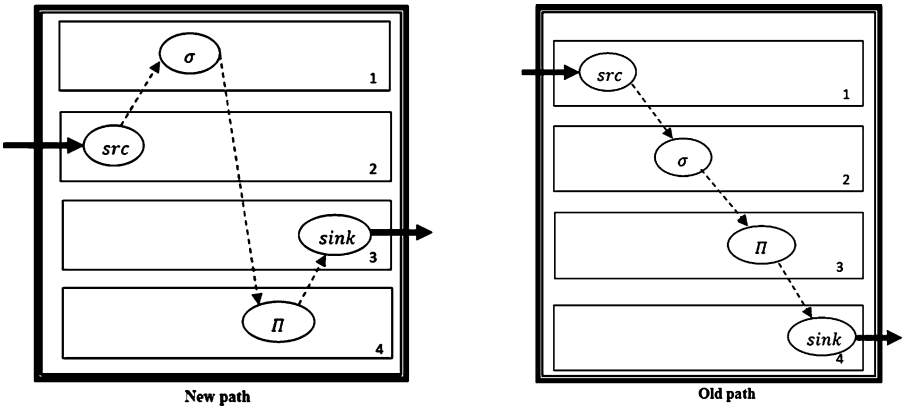
$$w(\Pi_3^-, sink_4^-) = 6$$

and

$$B = 10$$

{*Overload situation is happened and re-scheduling is performed*}

5.3.2. *Re-scheduling* (*re-routing in QMG*): *find the shortest path in QMG and send proper triple* $(p, o, s)$ *to each machine in the new path scheduling: assign each operator of the main query plan to the corresponding machine according to* (1) (*send proper triple* $(p, o, s)$ *to each machine*).




{*new and old paths are running concurrently until old path becomes full. Then, only new path will work until next overload situation*}
*UNTIL there is no more input tuple or query expired*

According to the proposed parallel query processing algorithm, Fig. 8 shows pseudo codes for different parts of this algorithm.

## 5 Time complexity analysis

Time complexity of parallel query processing algorithm is analyzed from the following three perspectives:

(a) *Machines executing query plan operators*

Time complexity of these algorithms (*i.e.,* stream operators such as *selection*, *projection*, *join* over tuples in a queue of size $n$) is assumed $O(n^2)$.

(b) *The sink node*

The machine that executes sink node of the *QMG* sorts output buffer's tuples. So, its time complexity is $O(n.\log n)$ in which $n$ is *output buffer size*.

```
Procedure Generate_Query_Plan (Q)
{
 G_QP<V,E> := Make_OptQuery_Plan(Q);
}

Procedure Generate _ Query _ Mega _Graph (G_QP)
{
  For i := 1 to k
        Send-to-process (Make_Copy(G_QP), i);
  QMG<V',E',W> := Make-QMG(G_QP, K);
}

Procedure Initial_Scheduling (G_QP)
{
    For i := 1 to |V|                    // initial scheduling (manual routing)
        { j := i mod k;
          Routing-Table[j][1].p:=j-1;
          Routing-Table[j][1].o:=i;
          Routing-Table[j][1].s:=j+1;
        }
}

Procedure Parallel_Query_Processing ()
{
  Repeat until (there is any tuple left && query is not finished)
   { Do in parallel
        {In each process 1 ≤ j ≤ k
              For n := 1 to z
                If (Routing-Table[j][n].empty≠ FALSE) {
                      Receive-tuple-from-process(Routing-Table[j][n].o);
                      Execute-on (Routing-Table[j][n].o);
                      Send-tuple-to (Routing-Table[j][n].s);
                }
        Sort-tuple-by-asc-timestamp (Sink.InputBuffer);
        Deliver (Dequeue (Sink.InputBuffer));        //delivers an output tuple to the App.
        }
  }

Procedure Re-Scheduling ()
{
    If (Isfull_any _ InputBbuffer())          // updating the weight of edges in query mega graph
        { For i := 1 to k
            For j := 1 to k , i ≠ j
              For a := 1 to |v|
                For b := 1 to |v|
                    If (a,b) ∈ E {
                          C := Σ_{i=1}^{k} W_{i,i}(a, b);
                          W_{j,i}(a, b): = W_{j,i}(a, b) + C;
                    }
        Find_Shortest_Path (QMG);
        }
}

Procedure Find_Shortest_Path (G)                  // finding the shortest path and assigning triple (p,o,s) to each
{ Dikjesta_shortest_path (G);                              //  machine in new path
 m := (z+1) mod d;            // z is number of concurrent operators in a machine in runtime, d is its upper bound
 For i := 1 to k
       { Routing-Table[i][m].empty=FALSE;
         Update (Routing-Table[i][m].p);          // handling multiple operators (old one and new ones in the
         Update (Routing-Table[i][m].o);                    // new path) in each machine
         Update (Routing-Table[i][m].s);
       }
}
```

**Fig. 8** Pseudo codes for different parts of the parallel query processing algorithm

(c) *The scheduler unit*

Time complexity of algorithms that the scheduler unit executes is equal to sum of time complexity of steps which scheduler performs, *i.e.*:

- Generating query plan, sending its copies to logical machines, and also generating *QMG* which are done statically only once at the beginning (negligible).
- Initial scheduling which consists of determining machines for each query plan operator with $O(|V|)$ (according to (1), and sending proper triple $(p, o, s)$ to each machine with $O(K)$.
- Steps that scheduler executes dynamically (2) as below:

$$O(|E'|) + O(|V'|.\log|V'|) + O(K) \tag{3}$$

They are updating *QMG*'s edges, finding the shortest path in *QMG*, and sending proper triple $(p, o, s)$ of the new path to machines respectively.

Accordingly, the most costly steps of the proposed parallel query processing algorithm refer to the re-scheduling (3). With respect to (1) and (3), since $K$ is a constant and small value (Sect. 7), growing of main query plan dimensions (*i.e.,* $|V|$ and $|E|$), leads to increasing re-scheduling time quasi linearly.

## 6 Resource allocation in multiple query processing

To process multiple queries in the presented system, computational resources can be allocated to queries in one of the following approaches ($M$ is the maximum number of queries that can be executed simultaneously over $K$ logical machines while $K$ is a constant value):

(a) *Dynamic allocation*

All of the resources are allocated to the first registered query at the beginning. When a new query arrives, $1/M$ of resources is revoked and allocated to it.

(b) *Static allocation*

A quota equal to $1/M$ of $K$ is allocated to each query.

(c) *Hybrid allocation*

Minimum amount of resources allocated to each query is $K/M$. Also, free resources are assigned to existing queries (although they will be revoked when a new query arrives).

The first approach is complex to implement. The second is simple and also observes fairness, but has low resource utilization. The third one has the best performance but the most complexity.

In the presented system, static allocation approach is employed.

## 7 Performance evaluation

We have modeled the proposed parallel query processing system and evaluated it via simulation. Our experimental setup is briefly described below.

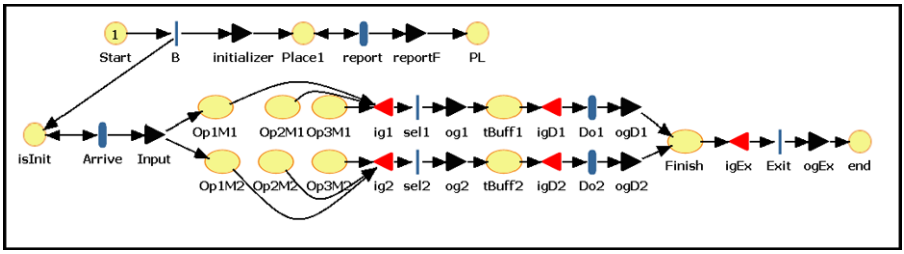



**Fig. 9** PetriNets model of the presented system

The presented system is modeled in PetriNets for performance evaluation. Petri nets and their extensions are appropriate formalisms for modeling and analyzing concurrent systems. Stochastic activity networks (SANs) are powerful and flexible extensions of Petri nets. These models combine the representation of concurrency, timeliness, fault-tolerance and degradable performance in a single model [16]. Colored stochastic activity networks (CSANs) are a high-level extension of SANs which provide features for hierarchical modeling and data manipulation by introducing macro activity and colored place [16]. For modeling purpose, we have used PDETool [15, 17] which is an extensible multi-formalism modeling and simulation framework that currently supports CSAN in addition to some other stochastic extensions of Petri nets. Figure 9 shows our systems' PetriNets model. At the beginning, there exists a token in place *start* which causes firing activity $B$ to initialize the model. Then, activity *Arrive* fires periodically that models the packet arrival into the system. Each colored place $Op_iM_j$ is a *FIFO* queue that contains tuples in the queue of $operation_i$ in $machine_j$. To perform the appropriate operation on a tuple in $machine_j$ ($Op_1M_j$, $Op_2M_j$ or $Op_3M_j$), the activity $sel_j$ is fired to carry the tuple into the colored place $tBuff_j$ (temporal buffer).

The color for each token tuple is defined in a way that such a token can store all the required properties of a tuple including its initial time, data, etc.

To perform an operation in $machine_j$, an activity $Do_j$ fires to model the operation on the selected tuple in place $tBuff_j$. This is done based on the selectivity of the tuple with cost time $t$. The result tuple(s) is enqueued in the proper queue (determined by execution of the algorithm in the output gate $og_j$) or leaves the system by adding into the place Finish.

The activity *report* which is uniformly fired every second, logs the current snapshot of the system to help in model evaluation stage. In addition, some steady-state rewards (such as performance measures) are defined and calculated by the tool.

### 7.1 Experimental setup

The model is implemented in PDETool [17] and all the experiments were run on a dedicated dual processor Alpha machine with 2GB of RAM. Algorithm presented in this paper (that improve query processing speed via parallelism) is compared with ordinary serial query processing (*Serial*—means without parallelism) [8] and also with the *Min-Latency* scheduling algorithm used in the Aurora DSMS prototype [18].

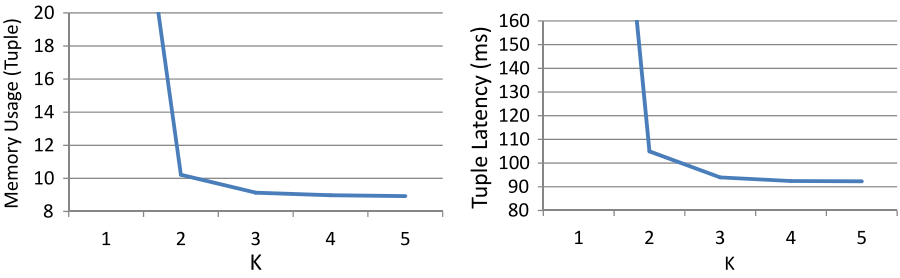



**Fig. 10** Comparison of system performance while increasing number of logical machines

Test queries consist of selection, projection and join (stream-to-relation and stream-to-stream) operators. Each operator is modeled with two characteristics, execution time (values between 2 and 5 milliseconds) and selectivity (values between 0.8 and 1.2). A query is considered as a sequence of 3 operators, join, selection and projection. The query set consists of 12 queries of different types ranging from the simplest queries (single operator) to the complicated ones (with 3 different operators). We used a synthetic data generator to generate input data stream. The input data arrival times are generated as synthetic bursty traffic by flows that begin according to a Poisson process (with mean inter arrival time equal to 1 time unit) and then send packets continuously for some duration chosen from a heavy-tailed distribution. The Pareto distribution is used for packet durations, which has a probability mass function given by $P(x) = \alpha k^{\alpha} x^{-\alpha-1}$, for $k > 0$, $x \geq k$. We used $k = 1$ and $\alpha = 2.3$ in our experiments as in [8]. Also, the attribute values are generated uniformly from a numeric domain.

Note that in our performance evaluation, when we use $k$ logical machines each with speed of $s$ in our parallel query processing engine, single machine used in serial query processing methods has speed of $k.s$ instead of $s$.

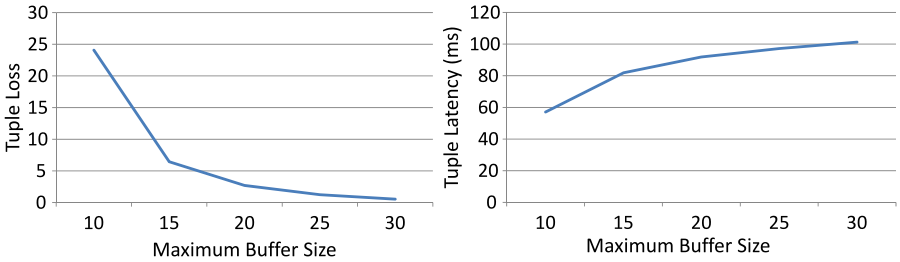
Based on [33], measured parameters are:

- *tuple latency* (*or response time*): difference between departure and arrival time for each tuple
- *memory usage*: number of tuples which are stored in all of the queues in the query plan at run time
- *system throughput*: number of output tuples
- *tuple loss*: number of tuples which are discarded from queues of query plan

## 7.2 Experimental results

Before analyzing performance of the presented system and comparing it with the other methods, we analyze our system performance for different values of $k$ (*i.e.*, number of logical machines). Figure 10 shows system's memory usage and tuple latency for $1 \leq k \leq 5$.

As shown in Fig. 10, the difference between single machine and two logical machines (parallel query processing) is very considerable. In general, a direct relationship holds between number of logical machines and system performance. Although




a) tuple latency vs. maximum buffer size

b) tuple loss vs. maximum buffer size

**Fig. 11** Tuple latency and tuple loss for different sizes of operator's queue

employing four logical machines ($k = 4$) would have better performance, but $K = 2$ is selected to present performance improvement whilst employing minimum computational resources.

**Note** In context of parallelism, increasing number of employed computational resources does not necessarily result in considerable performance improvement. This is known as speedup and Amdahl's law [38–41]. Amdahl's law, also known as Amdahl's argument, is used in parallel computing to predict the theoretical maximum speed up using multiple processors [wikipedia].

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. For example, if a program needs 20 hours using a single processor core, and a particular portion of 1 hour cannot be parallelized, while the remaining promising portion (95%) can be parallelized, then regardless of how many processors we devote, the minimum execution time cannot be less than that critical 1 hour. So, in this situation the speed up is limited up to 20× [wikipedia].

Furthermore, size of operator's queue, $B$, has a significant impact on system performance; the more the queue size, the more the tuple latency and the less the tuple loss. Figure 11 shows these parameters for different values of $B$.

Although, it is better for the presented system—which aims to minimize tuple latency—to use smaller queue size, but making a tradeoff, $B = 20$ is selected as operator's queue size.

Hereafter, parallel query processing using two logical machines and queue size of 20 tuples (called *PQP*-2) is compared with ordinary query processing using the Round Robin scheduling algorithm (called *SR*) [8], and serial query processing using the *Min-Latency* scheduling algorithm (called *ML*) [18].

Figure 12 illustrates memory usage, tuple latency, throughput and tuple loss parameters versus duration (simulation time or data stream tuples arrival).

Figure 12 shows that, although the presented system is proposed to minimize tuple latency, but it also makes a considerable improvement in terms of memory usage, throughput and tuple loss. Also, average values of these parameters, computed from five-times simulation is shown in Fig. 13.

Results shown in Fig. 13 state the fact that the presented parallel system has a significant improvement over ordinary query processing as well as the *Min-Latency*

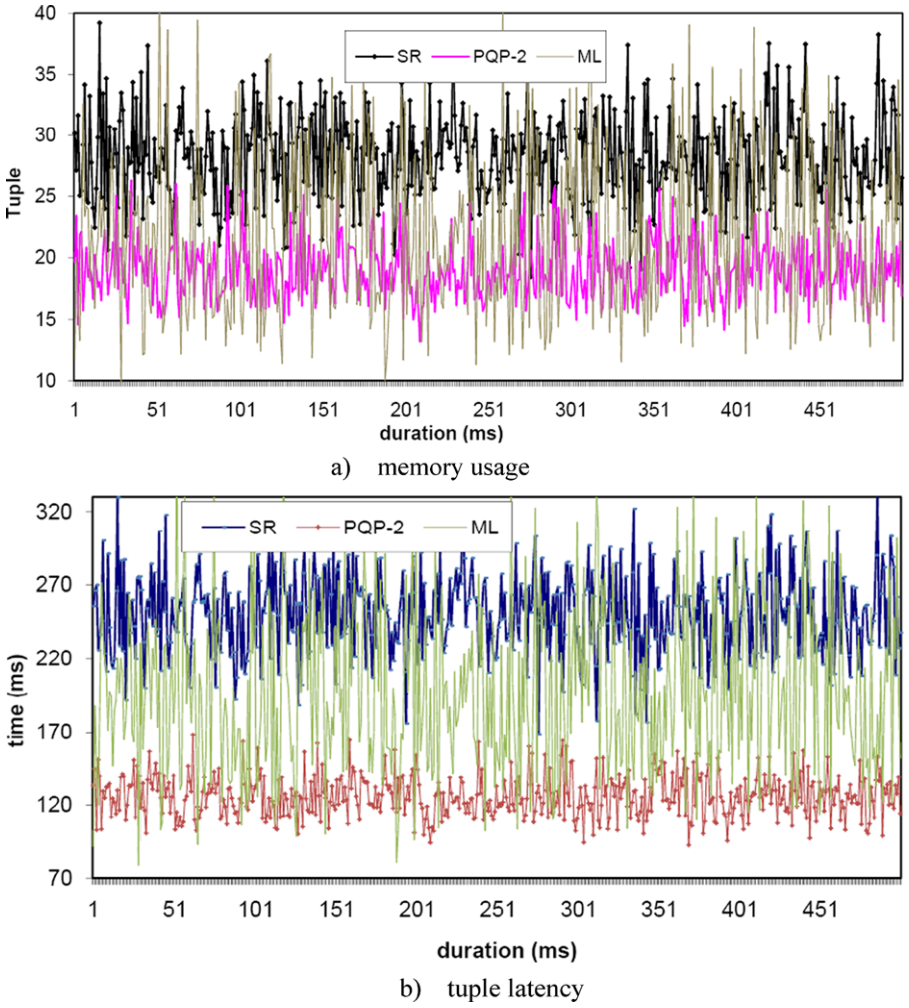
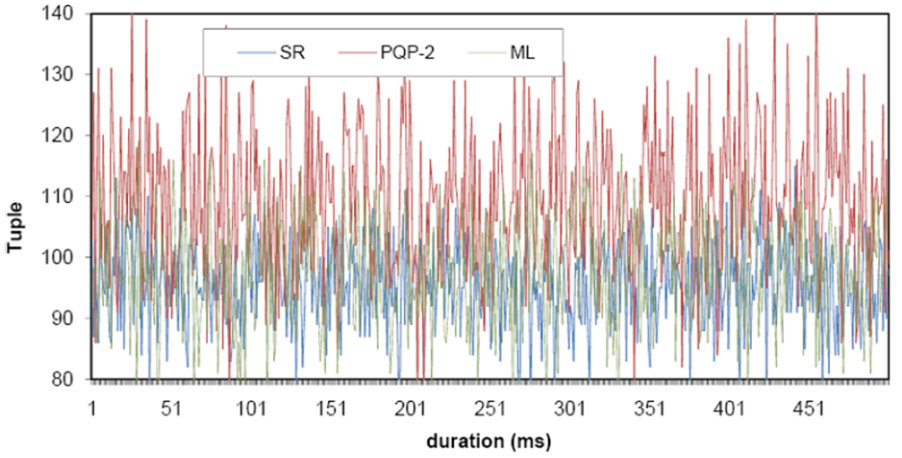



**Fig. 12** Comparison of measured parameters vs. simulation time

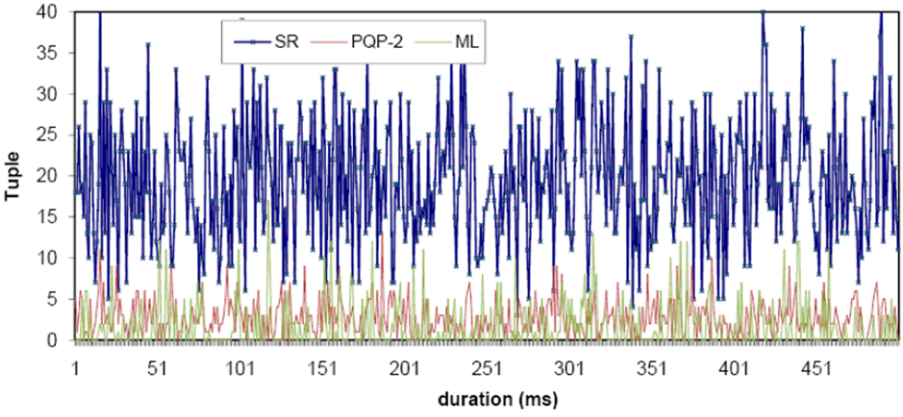
scheduling algorithm in terms of tuple latency and memory usage. Although, throughput and tuple loss parameters of the *ML* method are much better than *SR* method, but the presented method has the best performance. Standard deviation of measured parameters is computed and illustrated in Fig. 14.

Less value of deviation means to have a more stable and uniform operation. As shown in Fig. 14, standard deviation of measured parameters in the presented method is less than compared methods except in term of throughput.

In order to evaluate system performance against increasing number of queries, memory usage, tuple latency, throughput and also tuple loss of the presented system are compared in Fig. 15.




c) throughput




d) tuple loss

**Fig. 12** (*Continued*)

Figure 15 shows that in the presented system, memory usage, tuple latency and throughput grow linearly against increasing number of queries, but tuple loss grows nearly exponentially.

According to the experimental results, especially Figs. 12, 13 and 14, it can be concluded that *PQP-2* outperforms ordinary serial data stream query processing and also scheduling algorithm employed in the Aurora DSMS prototype claiming to minimize tuple latency.

## 8 Related work

In primitive DSMSs such as the STREAM, aiming at compatibility with the relational data model, query processing was experiencing a slow procedure (due to converting stream to relation, executing relational operators and converting to stream) [2].

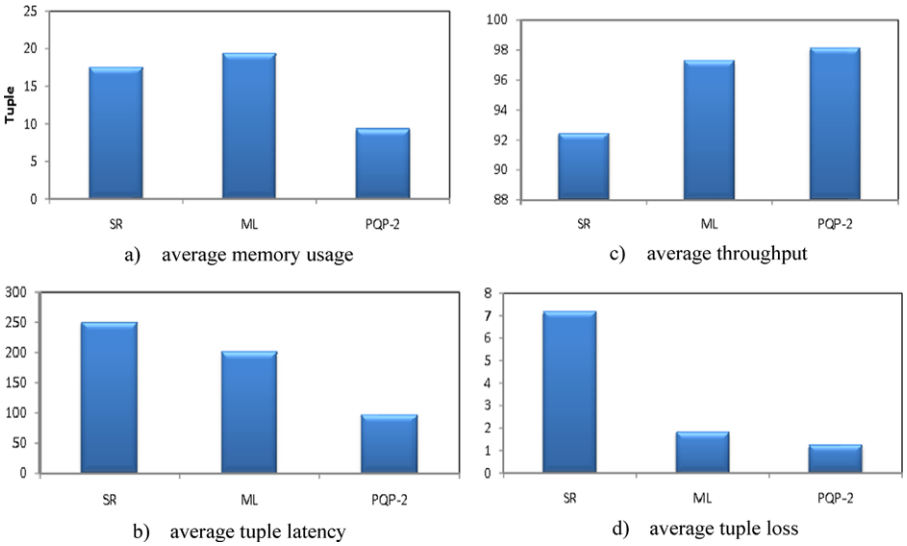



**Fig. 13** Average value of measured parameters computed from 5-times simulation

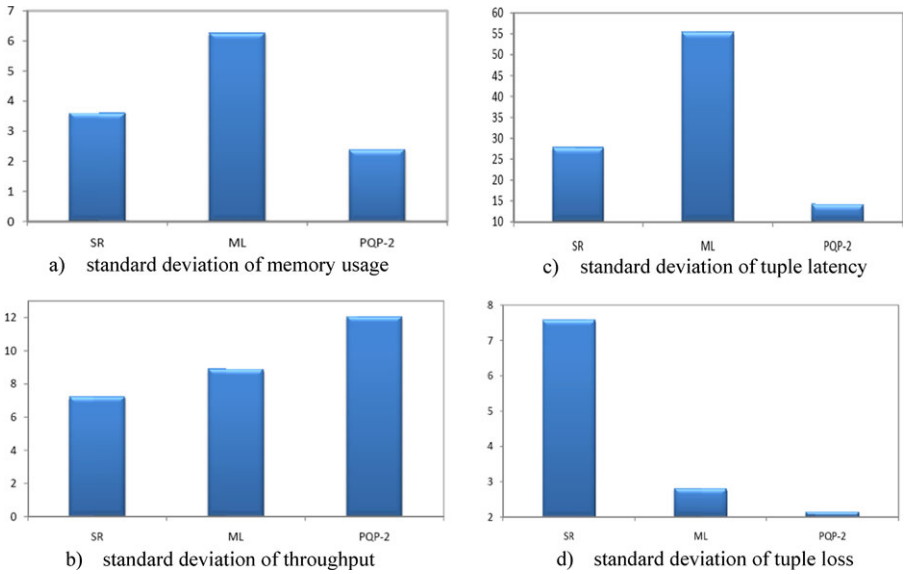



**Fig. 14** Standard deviation of measured parameters

Fast query processing in data stream management systems, as a challenging issue, has been considered by researchers in form of scheduling algorithm [8, 18, 31]. Although different scheduling strategies are proposed in context of data stream systems but most of them are not profitable. Some of them such as *FIFO* and *Round Robin* are very simple and show no adaptivity to the bursty nature of data stream [8]. Many of

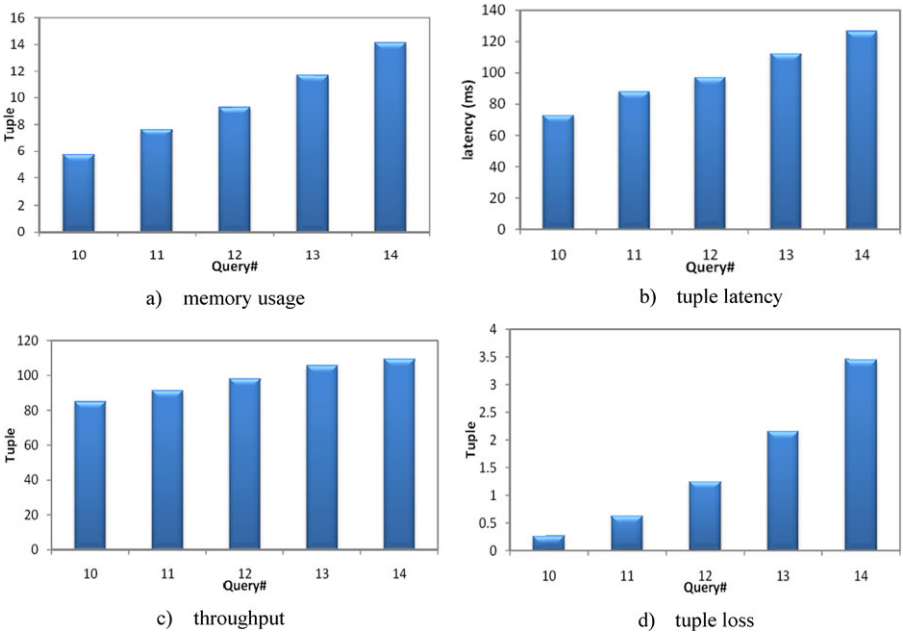



**Fig. 15** Measured parameters versus number of queries in the presented system

scheduling algorithms strive to optimize one of the several critical parameters in data stream management systems *i.e.*, tuple latency, memory usage, throughput and tuple loss. Among manifold scheduling algorithms proposed in data stream query processing context, many have concentrated on efficient memory usage [19–21]. The *Chain* scheduling algorithm which is employed in the STREAM DSMS, aims to minimize memory requirements but it results in high output latency [20]. Extensions of the *Chain* (*e.g., Chain-Flush*) are designed to simultaneously achieve low memory usage and low latency [8]. Some others consider a complex hybrid function of metrics such as memory usage and response time [22]. Aurora prototype provides QoS consideration by employing different scheduling algorithms as inbox traversing method [3] It uses a two-level scheduling approach; first, determining which query to process and then, how to process it. For the latter one, three algorithms for traversing the operator path are introduced in [18] which differ in the performance parameter they strive to optimize: throughput, tuple latency, and memory usage. *Min-Latency* [18] is such an algorithm that is used to minimize tuple latency and we compared it with our proposed algorithm.

Dynamic load distribution over multiple machines is provided in the Borealis [23, 24]. Although this is done in two static and dynamic phases, but reducing end-to-end delay is issued just in static phase in which requires a coordinator to gather load information in a learning time period. In dynamic phase, it just tries to achieve a good balance between the load migration overhead and the quality of the new operator mapping plan.

Many data stream query processing researches concern adaptive query processing [25]. This may help to handle the overload situations but the single processor bottle-

neck would be left. One of the well-known solutions to handle this bottleneck is parallelism. Parallelism is considered in a vast spectrum of platforms in traditional DB systems [26, 27] but is missed awhile in data stream management systems [28–30]. The system presented in this paper aims to solve the mentioned bottleneck via parallel query processing in a multiprocessing environment.

In [34], in order to improve adaptivity and scalability, an operator named flux is proposed for partitioning of context-sensitive operators (*e.g., join* and *groupby*). The flux operator performs horizontal (intra-operator) adaptivity by online re-partitioning and distributing the load and operators' state across a cluster. The *Eddy* architecture enables fine-grained (inter-operator) adaptivity via eliminating query plan. The *Eddy* operator re-routes each tuple through operators dynamically. High amount of overhead of the eddy architecture is its most important drawback [34]. In distributed version of *Eddy* [35], it is not a bottleneck anymore but the storage overheads (*e.g.,* storing done bitmap along with each tuple) as well as processing costs (*e.g.,* updating bitmaps and gathering statistical information such as selectivity and execution cost for each operator) are leftover which are considerable w.r.t. distribution of the system.

In [36], using a classifier, a distinct path is computed and employed for each subset of data with distinguishing characteristics. This strategy is a solution between the two major query optimization approaches: employing a single monolithic execution plan and planless execution such as the eddy architecture. Nehme et al. [37] improve its adaptivity via the concept-drift mechanisms in machine learning.

## 9 Conclusion

Since in DSMSs, queries are continuously executed over the non-stopping data streams which are rapid and bursty, a single processor is not capable to process this volume of data and execute continuous queries over them with satisfactory speed. In other words, a single processor used for serial execution of queries in DSMSs causes reduction of output performance and is compelled to discard data stream elements. Parallel query processing presented in this paper is a solution for problems arising from this bottleneck in such systems. Scheduling of parallel execution of operators in our presented system is performed via finding the shortest path in a weighted graph called *Query Mega Graph* which is a logical view of $K$ logical machines. By lapse of time, number of tuples waiting in queues of different operators may be very different. When a queue becomes full, re-scheduling is done by updating weight of edges of *QMG*. In the new computed path, machines with more workload are used less. Preserving input ordering of data stream tuples is a challenging issue discussed.

Parallel query processing algorithm, its complexity analysis and resource allocation in case of multiple query processing is presented. The presented system is formally stated and its correctness is proved. Also system performance is evaluated and compared with *Serial* query processing and the *Min-Latency* scheduling algorithms. Even though it can be concluded that more logical machines causes better performance, but minimum resources are employed in our evaluation ($K = 2$). The system improves tuple latency, memory usage, throughput and tuple loss parameters.

Some of the future works are listed below:

- More efficient algorithms for scheduling operators and preserving tuples ordering
- Optimization of resource allocation in case of multiple query processing
- Employing the presented parallel query processing engine in real-time DSMSs
- Distributing the presented parallel system
- Reliability analysis of the presented system

## References


1. Babcock, B., et al.: Models and issues in data stream systems. In: Proceeding of PODS, Invited paper (2002)
2. The STREAM Group.: STREAM: The Stanford stream data manager. IEEE Data Engineering Bulletin, March (2003)
3. Abadi, D., et al.: Aurora: A new model and architecture for data stream management. VLDB J. **2**(12), 120–139 (2003)
4. Babcock, B.: Models and issues in data stream systems. In: Proceeding of PODS (2002)
5. Golab, L., Ozsu, M.T.: Issues in data stream management. SIGMOD Records (2003)
6. Tatbul, N., et al.: Load shedding in a data stream manager. In: Proceedings of VLDB'03, Germany, pp. 309–320 (2003)
7. Chekuri, C., et al.: Scheduling problems in parallel query optimization. In: Proceeding of PODS (1995)
8. Babcock, B., et al.: Operator scheduling in data stream systems. VLDB J. **13**(4), 333–353 (2004)
9. Hong, W.: Parallel query processing using shared memory multiprocessors and disk arrays. Ph.D. thesis (1992)
10. Hasan, W., et al.: Open issues in parallel query optimization. ACM SIGMOD Rec. **25**(3), 28–33 (1996)
11. Graefe, G., et al.: Extensible Query Optimization and Parallel Execution in Volcano. Query Processing for Advanced Database Systems. Morgan Kaufmann, San Mateo (1994)
12. DeWitt, D.J., Gray, J.: Parallel database systems: The future of high performance database processing. Commun. ACM **36**(6), 377–387 (1992)
13. Babu, S., Widom, J.: Continuous queries over data streams. In: SIGMOD Record (2001)
14. DeWitt, C., Naughton: Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In: Proceeding of ICDE (2002)
15. Movaghar, A.: Performability modeling with stochastic activity networks, Ph.D. dissertation, University of Michigan (1985)
16. Abdollahi Azgomi, M., Movaghar, A.: Coloured stochastic activity networks: Definitions and behavior. In: Proceeding of UKPEW'04, UK, pp. 297–308 (2004)
17. Khalili, A., et al.: PDETool: A multi-formalism modeling tool for discrete-event systems based on SDES description. In: Lecture Notes in Computer Science, vol. 5606, pp. 343–352. Springer, Berlin (2009)
18. Carney, D., et al.: Operator scheduling in a data stream manager. In: Proceedings of the VLDB, Germany, pp. 838–849 (2003)
19. Arasu, et al.: Characterizing memory requirements for queries over continuous data streams. In: Proceeding of the PODS (2002)
20. Babcock, B., et al.: Chain: Operator scheduling for memory minimization in data stream systems. In: Proceeding of the ACM SIGMOD (2003)
21. Babu, et al.: Exploiting $k$-constraints to reduce memory overhead in continuous queries over data streams. Technical Report, November (2002)
22. Ghalambor, M., Safaeei, A.A., Azgomi, M.A.: DSMS scheduling regarding complex QoS metrics. In: IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 10–13 May 2009
23. Abadi, D.J., et al.: The design of the Borealis stream processing engine. In: Proceeding of the CIDR (2005)
24. Xing, Y., et al.: Dynamic load distribution in the Borealis stream processor. In: Proceeding of ICDE (2005)
25. Babu, S.: Adaptive query processing in data stream management systems. Ph.D. thesis (2005)

26. Graefe, G.: Volcano—An extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994)
27. Apers, P.M.G., et al.: PRISMA/DB: A parallel, main memory relational DBMS. IEEE Trans. Knowl. Data Eng. **4**(6), 4–24 (1992)
28. Beringer, J., Hullermeier, E.: Online clustering of parallel data streams. Int. J. Data Knowl. Eng. **58**(2), 180–204 (2006)
29. Kramer1, J.: Dynamic plan migration for snapshot-equivalent continuous queries in data stream systems. In: Proceeding of ICSWN (2006)
30. Zhu, Y., et al.: Dynamic plan migration for continuous queries over data streams. In: Proceeding of SIGMOD (2004)
31. Safaei, A.A., et al.: Using finite state machines in processing continuous queries. Int. Rev. Comput. Softw. **4**(5), 551–556 (2009)
32. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: Proceeding of the VLDB (2003)
33. Chakravarthy, S., Pajjuri, V.: Scheduling Strategies and Their Evaluation in a Data Stream Management System. LNCS, vol. 4042. Springer, Berlin (2006)
34. Shah, M.A., et al.: Flux: An adaptive partitioning operator for continuous query systems. In: Proceeding of the ICDE (2003)
35. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: Proceeding of the VLDB (2003)
36. Nehme, R.V., et al.: Query mesh: Multiroute query processing technology. In: Proceeding of the VLDB (2009)
37. Nehme, R., et al.: Self-tuning query mesh for adaptive multi-route query processing. In: Proceeding of the ACM EDBT (2009)
38. Alqadi, Z.A.A., et al.: Performance analysis and evaluation of parallel matrix multiplication algorithms. World Appl. Sci. J. **5**(2), 211–214 (2008)
39. Akl, S.G., et al.: Data-movement-intensive problems: Two folk theorems in parallel computation revisited. Theor. Comput. Sci. **95**, 323–337 (1992)
40. Almasi, G.S., Gottlieb, A.: Highly Parallel Computing. Benjamin/Cummings, Redwood City (1989)
41. Cosnard, M., Trystram, D.: Parallel Algorithms and Architectures. International Thompson Computer Press, Boston (1995)