ORDER PRESERVING STREAM PROCESSING IN FOG COMPUTING ARCHITECTURES

K. Vidyasankar

Department of Computer Science, Memorial University of Newfoundland,
St. John's, Newfoundland, Canada

ABSTRACT

A Fog Computing architecture consists of edge nodes that generate and possibly pre-process (sensor) data, fog nodes that do some processing quickly and do any actuations that may be needed, and cloud nodes that may perform further detailed analysis for long-term and archival purposes. Processing of a batch of input data is distributed into sub-computations which are executed at the different nodes of the architecture. In many applications, the computations are expected to preserve the order in which the batches arrive at the sources. In this paper, we discuss mechanisms for performing the computations at a node in correct order, by storing some batches temporarily and/or dropping some batches. The former option causes a delay in processing and the latter option affects Quality of Service (QoS). We bring out the trade-offsbetween processing delay and storage capabilities of the nodes, and also between QoS and the storage capabilities.

KEYWORDS

Fog computing, Order preserving computations, Quality of Service

1. INTRODUCTION

Internet of Things (IoT) is about making things smart in some functionality, and connecting and enabling them to perform complex tasks by themselves. A "thing" is any object of interest with some communication capability. IoT applications include Connected Vehicles, Smart Grid, Smart Cities, HealthCare and, in general, Wireless Sensors and Actuators Networks [1]. Billions of devices are expected to be made smart in the very near future. They will produce massive amounts of data, requiring enormous amount of computations. In cloud-based IoT environment, the computations are delegated to the cloud. The cloud is certainly scalable with respect to processing capability and storage. However, many applications require quick real time computations and local actuations, and the latency involved in communicating with the cloud is not tolerable. Further, sending huge amount of data to the cloud requires high network bandwidth and incurs considerable delay. In addition, in many applications, 24/7 connectivity to the cloud may not be available. To overcome these constraints, a *fog computing* architecture has been proposed recently [1, 2, 3, 4]. It consists of *edge* nodes that generate and possibly pre-process (sensor) data, *fog* nodes that do some processing quickly and enable any actuations that may be needed, and *cloud* nodes that may perform further, detailed analytics for long-term and archival purposes.

Fog computing typically involves continuous processing of stream data that are input to the edge devices. The data consist of tuples. They are processed in batches of tuples. Each processing instance at a node uses some input batches and produces an output batch which is sent to the

Dhinaharan Nagamalai et al. (Eds) : COMIT, AISCA - 2019
pp. 37 – 48, 2019. © CS & IT-CSCP 2019 DOI: 10.5121/csit.2019.90104

parent of that node (except at the cloud level) for further processing. The computation to be done on a batch is decomposed into sub-computations to be executed at the different nodes in the fog architecture. Edge and fog nodes typically have limited storage, compute and network connectivity capabilities. Hence, the computations need to be distributed carefully among the processing nodes. Guaranteeing consistency of the executions is very important. Consistency issues arise for sub-computations at the individual nodes as well as the entire computations on individual batches and computations over sequences of input batches.

In this paper, we consider each sub-computation at a node as a transaction. We also assume serial executions of these transactions in each node. We relate consistency to serializability of these transactions at every node. In several applications, the computations on the sequence of batches are expected to preserve the order in which the batches arrive from the sources. This is the *consecutive serializability* requirement for the transactions. In some cases, the sub-computations at some nodes, especially at lower levels of the hierarchy, may not be required tofollow batch order, that is, the sequence can be *saga*[5], with the order being restored at higher levels. This helps also for scalability where the computations at a level can be distributed over multiple nodes and the results forwarded to a single node in the next higher level. Then the input batches in the higher level may not arrive according to the batch order. Unreliable network connectivity may also produce out-of-order message delivery. In this paper, we focus on achieving consecutive serializability at a node in the presence of out-of-order message delivery. We do this by storing some input batches temporarily and/or dropping some batches. The first option requires storage capacity and also causes delay in processing whereas the second option affects the accuracy of the continuous executions. This affects Quality of Service (QoS). We identify some QoS parameters that are relevant in this context. We discuss different execution options that offer trade-offs between QoS and storage capacities of the processing nodes.

We consider the simple case of inputs from a single source in Section 2. We consider individual executions of the input batches as well as their combined executions. We consider processing batches from multiple input sources in Section 3 and multiple heterogeneous input sources in Section 4. We discuss some related works in Section 5. We conclude in Section 6.

2. SINGLE INPUT SOURCE

We use the basic definitions given in Vidyasankar [6]. We consider a hierarchy (rooted tree) **V** of nodes v. It consists of n levels. In this section, we consider the simple case of a single input source. Then, the hierarchy is a simple path of length $n-1$. The node in the path in j^{th} level will be v_j. Here, v_n refers to the cloud, v_1 to the edge and the intermediate nodes to the fog. We assume that stream data is generated at level 0. The edge devices at level 1 themselves may generate some or all of this data. We separate the generation into another level for notational convenience. Each node $v_j$has *processing capability* $P_j$and *storage capacity* S_j, each expressed in appropriate units. The source input batches are numbered sequentially. We refer to the i^{th}batch as b_i. Each batch b_i is processed in one or more nodes. The computation for b_i is referred to as $C(b_i)$. We consider a decomposition of $C(b_i)$ into sub-computations as

$$C(b_i) = c_{1,i}+c_{2,i}+ \cdots +c_{n,i}.$$

Such decompositions will be based on the semantics of the applications and of the computations. He reeach $c_{j,i}$is to be executed at level j, in the given sequential order of the levels. The sum of the computations until level j is referred to as $C_{j,i}$(with capital C). That is,

$$C_{j,i}= c_{1,i}+ \cdots +c_{j,i}.$$

Then, $C_{n,i}= C(b_i)$.

As stated earlier, we assume in this paper that the individual $c_{j,i}$'s are executed atomically and serially in each level j. We denote the processing requirement and storage requirement for $c_{j,i}$as $p(c_{j,i})$ and $s(c_{j,i})$, respectively. Obviously, we must have $P_j \geq p(c_{j,i})$ and $S_j \geq s(c_{j,i})$. With each $c_{j,i}$, we associate an input batch $In(c_{j,i})$ and an output batch $Out(c_{j,i})$.

Several (devices in) nodes may have limited range of transmission. Nodes have to be placed such that dataflow from one level to the next is possible. To facilitate this, some nodes could be placed just to receive data from the lower level and send it to upper level. (This may involve storing some data temporarily.) We call these *relay* nodes. Sub-computation done in such a node will be *nil*. Output batch of this computation is the same as the input batch.

We discuss serializable executions of $C_{j,i}$'s. We define the following.

(1) **C** is the set of computations $c_{j,i}$'s for a given set of batches.
(2) $\prec B$ is the batch order.
(3) $\prec L$ is the level order.
(4) $\prec$ is $\prec B \cup \prec L$.
(5) A *history* **H** over (**C**, $\prec$) is a sequence of $c_{j,i}$'s in **C** obeying $\prec$.
(6) A history **H** is *globally serial* if it is a sequence of $C(b_i)$'s, that is, all the $c_{j,i}$'s for each I occur consecutively in **H**. It is *globally serializable* if it is equivalent to a globally serial history.

Some batches may be processed only partially. That is, $C(b_i)$ may only be $C_{k,i}(b_i)$, for some k, $k<n$. The above definition applies to such computations also.

2.1 INDIVIDUAL PROCESSING

We first consider processing of the batches individually at each level. Then, consecutive serializability of $C_{j,i}$'s, at each level j, is guaranteed if $c_{k,i}$'s are executed at each level k between 1 and j serially according to the batch order. (Recall that we are assuming atomic execution of each $c_{j,i}$.) In the following, we look at the ways of obtaining serial order effectively when output batches from one level may arrive at the next level out of order.

If $c_{j,i}$'s are not conflicting with each other, then an out-of-order execution is serializable. Then, inputs may be processed as they arrive and the corresponding outputs sent to the next level. This option is very favourable for horizontal scalability. Batches may be split and processed in multiple nodes in the same level provided the combined computations will constitute $c_{j,i}(b_i)$. However, an out-of-order execution together with an out-of-order message delivery from the current level to the next might amplify the extent of the out-of-order in the arrival of batches in the next level. (The *extent* of out-of-order can be characterized in many ways: (i) how late a batch arrives, that is, the number of batches with greater ids that come before this batch, (ii) how early a batch arrives, namely, the number of batches with smaller ids that come after this batch, (iii) the number of late or early arriving batches, (iv) averages over thedelay or too early arrival, etc.)

In the following, we consider the case where $c_{j,i}$'s are conflicting.

- Out-of-order inputs (messages) can be kept in a *pending set*, and the executions themselves can be done in correct order when the respective batches arrive. This involves waiting, causing delay in execution, and requires storage space for the pending set. Depending on the extent of the out-of-order, both the delay and the required amount of storage space will vary.

- Without pending set, executions can be done for batches arriving in increasing order of their ids, as they arrive, and late-arriving batches can be ignored (dropped). This implies that the

dropped batches are processed only partially, up to the previous level. No storage space is required here. In the example sequence (1,8,4,2,5,7,9,3), batches (with ids) 1, 8 and 9 will be processed and the remaining will be ignored.

- Without pending set, executions can be done for batches in the correct consecutive order of their ids. Out-of-order batches (those that arrive too early) can be ignored. In the above example sequence (1,8,4,2,5,7,9,3), batches 1,2 and 3 will be processed and the remaining ignored.

- A limited storage space can be kept for the pending set and early-arriving out-of-order messages that cannot be added to the pending set can be ignored. For example, with (1,8,4,2,5,7,9,3), if storage space is available only for three batches, after batches 8,4 and 5, batches 7 and 9 might be ignored. (Other options regarding which three batches to store can also be exercised.) Similarly, out-of-order messages arriving later than a certain amount of delay can be ignored.

We define *drop ratio* as the number of batches dropped compared to the total number of batches. We note that, in the above options, a trade-off exists between drop ratio and storage space, and between drop ratio and processing delay. Message loss is equivalent to dropping the message due to excessive delay.

If network connectivity is disrupted intermittently and hence output batches cannot be transmitted immediately after the executions, then the following options exist:

- When storage space is available, the options are the following. Here, two pending sets are used, one for storing input batches and the other for storing output batches.
  - Store output batches in the *output pending set* and send several of them together when connectivity is restored. Continue processing the input batches. (This option can be followed even when network connectivity is available, if transmitting several batches together will be cheaper than sending them one at a time.)
  - Stop processing until the output batch is sent, and store the incoming batches in the *input pending set.*
  - Store output batches in the output pending set until connectivity becomes available, and also store input batches in the input pending set until they can be processed. Continue processing. This option is suitable when input arrives from the lower level as a set of batches.
- When sufficient storage space, for output batches and/or pending sets, is not available, the options are the following.
  - Drop the output batch thus terminating the processing of the corresponding batch and continue processing the input batches.
  - Stop processing until the output batch is sent, and drop the batches that are incoming in the mean time, thus terminating their executions.

Here also, we observe a trade-off between drop ratio and storage space. Nodes in the hierarchy could be heterogeneous. Different nodes may follow different options. Out-of-order execution may be acceptable at some levels, and correct order required at certain levels. This amounts to $c_{j,i}$'s being non-conflicting at the former levels and conflicting in the latter ones. Then, the execution options may be chosen appropriately. We also note that allowing some out-of-order execution will reduce drop ratio. That is, there is a trade-off between the extent of the out-of-order and drop ratio.

2.2 COMBINING MULTIPLE BATCHES

At any level, several input batches may be combined and processed together. That is, the computation at v_j could be $c_j(b_{i:k})$, combining c_j for batches b_i to b_k. The combined output will be sent to v_{j+1}. An example is when the frequencies of executions at different levels are different. For instance, c_1 may be performed every 5 seconds and c_2 performed every 10 seconds. Then, the outputs of two executions of c_1 may be processed together in one execution of c_2. Another situation is when network connectivity is not always available to send data from one level to the next level and the output batches corresponding to several computations kept and sent together when connectivity becomes available.

In the following discussions, we use examples where three batches are combined.

2.2.1 NON-OVERLAPPING GROUPINGS.

(1) Grouping of consecutive batches:

Wait until all the relevant batches arrive and then process. Delay and storage space considerations discussed in the single batch processing case are applicable here also. In addition, we need to consider the following.

(a) Suppose batches 1, 2 and 3 are to be grouped, and 2 arrives very late (or does not arrive due to message loss). Then, we can drop that batch. Then, we can do one of the following:

- drop batches 1 and 3, that is, the entire group;
- do the computation for batch 1 alone (if the application semantics allows it) and combine batch 3 with 4 and 5; or
- combine batches 1, 3 and 4 if the application semantics allows grouping of a broken sequence of batches.

All these options relate to QoS differently: (i) absence or presence of broken sequences and, in the latter case, the number or percentage of broken sequences and (ii) fixed or variable size groupings.

(b) The batches for latter groupings may become available before those for earlier groupings (for example, (4,5,6) before (1,2,3)). If the computations are not conflicting, they can be processed in any order. Otherwise, they have to be processed in the correct batch order. If (4,5,6) grouping is processed first and then we find that batch 2 has to be dropped, the options are dropping 1 and 3 also or processing them either individually or by combining them.

(2) Grouping of non-consecutive batches:

As and when sufficient number of batches are available, the grouping can be done. The only storage space required will be for the batches waiting for the grouping.

2.2.2 OVERLAPPING GROUPINGS.

An example is (1,2,3), (2,3,4), (3,4,5), etc. After 1 and 2, suppose 5 arrives. Then wait for 3. When 3 arrives, combine (1,2,3). Then, wait for 4, etc. Whichever batches need to arrive, wait for them. Here, suppose 3 does not arrive for a long time and so it is dropped. Then, groupings (1,2,4), (2,4,5), etc. can be considered. Another possibility is dropping (1,2,3), (2,3,4) and (3,4,5), namely, all the originally intended groupings with 3. The choice would depend on the application

semantics of 'consecutive' batches. Here also, different options affect QoS differently. The delay and the storage space factors are the same as with non-overlapping groupings.

3. MULTIPLE HOMOGENEOUS INPUT SOURCES

In this section, we consider multiple input sources, all producing similar data that are to be processed the same way. The hierarchy is a tree. We consider a general height-balanced tree. (The discussion in the next section covers arbitrary trees.) We again separate the data generation part into level 0 and each source feeds to, that is, sends its output to a *distinct* node in level 1. Thus, each node in level 1 has one child.

We consider the case where, *at each level, each node performs the same computation*. (This restriction is also relaxed in the next section.) Each node in level 1 will process its source input and send its output to its parent. Each node in level j, for $1 < j < n$, will process the inputs from all its children and will send a single output batch, at the end of processing, to its parent.

We will first consider the case where all sources generate data synchronously. We refer to one such set of batches as a *batch-set*. We first consider synchronous processing of the batch-sets. That is, at each step, one batch arrives from each child and the set of these batches is processed. The batch-sets are indexed sequentially. A batch-set with index i is referred to as B_i. A computation at a node v_j in each level j combines the computations $c_j(x)$ of all the source input batches x in a batch-set that are input to the descendants of v_j in level 1. The computation required for B_i is $C(B_i)$, decomposed into $c_1(B_i) + c_2(B_i) + \cdots + c_n(B_i)$.

We now consider out-of-order message delivery from one level to another. We assume that the communication between any two nodes (a parent and a child) is independent of the communications between other pairs of such nodes. Therefore, the extent of the out-of-order will vary with respect to messages from different children. In the following, we illustrate the options with an example where the messages from only one child arrive out-of-order and messages from all other children arrive in correct order. We consider the example sequence (1,8,4,2,5,7,9,3) for messages from child x. Input batch from x with batch id k is denoted x_k.

- The executions are done in correct consecutive order when all the inputs for the corresponding batch-set have arrived. Until then, the incoming batches are kept in *separate* pending sets, one for each child. For the sequence (1,8,4,2,5,7,9,3), after processing B_1, the pending set for x will store inputs for batch-set ids 8 and 4 and pending sets for the other children will store 2 and 3 until x_2 arrives. Then, the computation can be done for B_2. After three further steps, the pending set for x will have (8,4,5,7,9) and other pending sets will have (3,4,5,6,7). On arrival of x_3, B_3 will be processed, followed by B_4 and B_5, waiting for x_6 for the processing of B_6, and so on. This involves waiting, causing delay in execution, and requires considerable storage space for the pending sets.

- We can reduce the size of the pending sets considerably as follows. Executions can be done in the correct consecutive order of the batch-sets with the batches arriving from children in the correct order, and *not waiting* for the batches of that batch-set from other children; when these batches arrive later, they are ignored. Out-of-order batches from other children with greater ids (those arriving too early) are stored in the pending sets, and used when their turns arrive. In our sequence (1,8,4,2,5,7,9,3), after B_1, batch-sets B_2 and B_3 will be processed without the inputs from child x. Batches 8 and 4 will be stored in the pending set for x. Then, B_4 will be processed with the newly arriving batches from other children and the one stored in the pending set for x, ignoring x_2. Batch-set B_5 will be

processed with batches from all children, and B_6 with inputs from all except x, storing 7 in the pending set of x, and so on.

  - This implies executions on partial batch-sets. This affects QoS relating to whether there are executions on partial batch-sets and, if so, a measure of the *density* of the partial sets, for example, how many batches, how well they represent various geographical regions, etc.
  - The execution can be subject to receiving batches from a minimum number of children, to make it meaningful. Otherwise, no execution may be done, resulting in dropping the entire batch-set in that level. This affects QoS differently: the drop ratio can be categorized as *batch drop ratio* and *batch-set drop ratio*.

- A variation in the above option is dropping the out-of-order inputs (those with greater ids, 8 in the above example), instead of storing in the pending set. That is, all out-of-order messages are dropped. Then, no pending sets are kept.

Combinations of the above options are possible, especially when the extent of the out-of-order is expected to be small. The first option of keeping the batches in the pending sets until all the inputs of the next batch-set arrive can be used for a while. At some stage, if the storage space becomes insufficient or the delay becomes too much, executions with partial batch-sets can be done. If the computations on different batch-sets are not conflicting, then the batch-sets can be processed soon after all their input batches are received. For example, in the sequence (1,8,4,2,5,7,9,3), B_4 can be processed without waiting for B_3 (in the case of not opting for executions on partial batch-sets). This will also reduce the number of entries in the pending sets.
We note that, as illustrated in the above example, if the inputs from even one child are out-of-order, the inputs from all other children have to be kept in the pending sets for correct, consecutive, order of execution.

Allowing for non-synchronous arrival of input batches (at any level, including the source level) and hence non-synchronous execution is straightforward. The batches from each input can be kept in the respective pending sets and when a batch-set is complete it can be processed. The processing could be in the correct order or any order. At some stage, an incomplete batch-set can either be dropped or processed as such.

If network connectivity is disrupted intermittently, the options discussed in Section 2 are applicable here also. We recall that the options are storing output batches, storing input batches, and dropping batches before or after the current computation. The requirement of storage space for pending input batches and/or output batches is inevitable. Less space will be needed for output batches due to (i) storage of one batch per computation in contrast to all input batches for that computation and (ii) computations such as aggregation producing outputs that are likely to be much smaller in size than any input or at least all inputs put together. Here also, nodes in the hierarchy could be heterogeneous and may follow different options.

Considerations for overlapping and non-overlapping groupings of several batch-sets are similar to those for grouping batches from a single source case. Several QoS parameters can be applied for the groups for different options. Some of them for non-overlapping groupings where out-of-order messages are ignored are:

- the number of complete batch-sets;
- the number of missing batch-sets;
- minimum number of batches in a processed batch-set; and
- average number of batches from a child.

For example, in a grouping of 5 batch-sets from 4 children, the quantities mentioned above for the sets of batches ((1,1,1,1),(2,2,-,2),(-,3,-,3),(5,-,5,5),(-,6,-,6)) will be 1 (for the first batch-set), 1 (for the fourth batch-set), 2 (with respect to the third batch-set) and 14/6 (with 6 batch-sets), respectively. (Here, "-" denotes messages arriving out-of-order messages and hence being dropped.) Different aggregations for several groups of batch-sets can also be considered.We note that grouping of individual out-of-order messages may result in reduced out-of-order among messages relevant for the entire group. For example, with grouping of three batch-sets from three children, for the batches arriving in the sequence (1,2,1), (2,1,3) and (3,3,2), there is no out-of-order messages with respect to the entire group.

4. HETEROGENEOUS INPUTS

We assume an arbitrary rooted tree for the fog infrastructure. Leaf nodes could be at different levels. In the following, we will assume that each source input is different and that computation performed at each node is different.

We will first consider the processing of a batch-set, consisting of one batch per source. In general, each input batch will be processed first individually and then together with other input batches (or the batches derived from them). For example, we consider a fog architecturewhere each of the three inputs x, y and z is processed individually first, then (derived batches from) x and y are processed together and then all the three are processed together. We refer to the computation done on a set S of batches as $C(S)$. Each of these computations is decomposed into sub-computations and then grouped into c_j's for execution at respective nodes. Let the corresponding sequences of computations be $C(x)$,$C(y)$,$C(z)$,$C(x,y)$ and $C(x,y,z)$. For each $C(S)$, the analysis as in the homogeneous case can be applied. We focus on the nodes where batches from different subtrees are combined. We refer to them as *merging nodes*. For simple exposition, we will take a single sub-computation for each set S, namely, $c_1(x)$,$c_1(y)$,$c_1(z)$,$c_2(x,y)$ and $c_3(x,y,z)$.

First, we consider synchronous arrival and synchronous execution of batch-sets B_i consisting of $\{x_i,y_i,z_i\}$.At merging nodes, we assume that if input batches from one or more children are not available, then the computation cannot be done. With out-of-order batch arrival, the options are the following:

- Store the batches in the pending sets and process a batch-set when it is complete, that is, when all the input batches corresponding to that batch-set have arrived.
- At any (synchronous) step, if all the input batches in the expected batch-set are not available, ignore the batch-set.

The options when network connectivity is disrupted are the same as in the homogeneous inputs case. We note that computations on the dropped batch-sets will not be done in any ancestors. This was called *ancestral-abort* in [7].

We now consider asynchronous arrival of the individual batches.For simple illustration, we consider the execution of $c_{2,i}(x,y)$, where only the batches (derived from) x and y are combined.We refer to the batches as x-batch and y-batch for convenience. The frequencies of generation of x- and y-batches may be different. We assume that the computation is triggered each time a new x- or y- or both batches arrive. In the first case, the most recent y-batch is used, in the second case, the most recent x-batch and in the last case both new batches are used for the computations. To be able to identify consistent pairing of the batches, we assume a *global* time stamping of the batches. We assume integer counter values as timestamps and index the batches with these timestamps. An example sequence of arrival of the batches, in correct order, is given in

Table 1. Here, for example, y_3 is paired with x_1, and also with x_4.We note that the indices of x-batches (similarly, y-batches) may not be continuous.

Table 1: Time stamped Sequence

x_1	$y_1y_2y_3$
$x_4x_5x_6$	y_6
...	...

Table 2: Indexed Time stamped Sequence

$x_{1,1}x_{1,2}x_{1,3}x_{2,4}x_{3,5}x_{4,6}$ ...	$y_{1,1}y_{2,2}y_{3,3}y_{3,4}y_{3,5}y_{4,6}$ ...

Now, we consider out-of-order message delivery. Then,x-batches and y-batches may arrive out of order at the merging node. We can store the batches in the respective pending sets, wait for a while for late-arriving batches and then order the batches correctly and pair them. For example, at some stage, if we assume that all batches with timestamps less than or equal to 6 have arrived, then the sequence shown in Table 1 can be formed and used for pairing. Batches arriving very late, very much out-of-order, can be ignored. Suppose, for instance, that y_3 has not arrived yet when we do the pairing. Then, we will end up pairing x_4 with y_2 and also x_5 with y_2. This causes inconsistency, in addition to not being able to use y_3, and QoS is affected.

Suppose that after y_1, y_6 arrives, perhaps after some time. Then, we will not know whether there are some y_i's in between. Suppose, with each y_j, the batch-id of the previous y-batch is sent. Then, on the arrival ofy_6, we would know about the existence ofy_3. However, until y_3 arrives, we will not know the existence of y_2. Thus, some mechanism can be implemented to indicate possible late arrivals of at least some batches.

To avoid the inconsistencies mentioned above, batches can be indexed with both batch number (independent of timestamp) and global timestamp as in Table 2. The batches with the same timestamp can be combined. We note that this is as in the case of synchronous execution of batch-sets. The timestamp synchronizes the batch-sets.

5. RELATED WORKS

Consistencies of continuous executions have been discussed widely in the literature in the context of stream processing. The sub computations are treated as transactions in Conway [8], Meehan et al. [9] and Botan et al. [10]. Serializability of the entire computation on a batch, treated as a composite transaction, is discussed in Gürgen [11] and Oyamada et al. [12]. Serializability of continuous queries is discussed in Vidyasankar [13]. Distributing computations in fog architectures has been described in Andrade et al. [14], Mortazavi et al. [15] andVidyasankar [6]. The property that computations at some levels are non-conflicting and hence they need not be order preserving has been used in Transactional Topologies [16]. Order preserving computations have been discussed in stream processing in Li et al. [17] and Shen et al. [18], and for Big Data Streams in Xhafa et al. [19].

6. DISCUSSION AND CONCLUSION

In fog architectures, stream inputs are processed in several stages at nodes in different levels. In a hierarchical structure, at each node, the computation is over the input batches that arrive from the children, producing an output batch which is sent to the parent. In this paper, we have addressed several issues relating to obtaining order preserving executions when messages do not arrive in correct order. We have discussed mechanisms for performing computations in correct order, by storing some batches temporarily and/or dropping some batches. The former option causes a delay in processing and the latter option affects QoS. We have brought out some trade-offs between processing delay and storage capabilities of the nodes, and also between QoS and the storage capabilities. Here, only transient storage of batches is considered, not persistent store for the results of the computation.

We have identified several QoS parameters that are relevant in this context. All of them deal with the non-inclusion of a batch in a computation. This can be identified at the node where the computation is done and its effect on the appropriate QoS parameter can be used at that node and also transmitted to the parent as part of the *context* associated with the output batch. The context may include batches explicitly with ids or just implicitly, for example, that a batch has been dropped. For doing this, batches need to be indexed. A natural place for indexing is the source level. However, in many applications, sources connect to *gateways* that filter the source input batches and pre-process them. The batches that are dropped in that level may not be relevant for the computations. Hence the (output) batches at the gateway level may be indexed. The gateways are expected to have the contextual information such as id, location, and other deployment details of the sources (sensors). Hence, indexing at that level may be more comprehensive than at the source level.

When the batches are processed individually throughout the entire hierarchy, the initial indexing may be adequate. However, when several batches are combined and processed together at some level, new index could be given to the output batch. (One example is assigning the largest batch index of that group, when the batches arriving at the next level need not have consecutive indexes.) That is, depending on the context that is attached to the output, separate independent indexing can be used at different levels, rather than carrying the initial index throughout.

Several non-hierarchical fog infrastructures have been proposed in the literature, for example, clustered, vehicular and smart-phone in [3]. These can be modeled by extending a hierarchy by replacing single nodes with clusters of nodes where the nodes within a cluster can communicate with each other in peer-to-peer fashion. Several sub-computations may be assigned to a cluster. They will be executed by one or more nodes in the cluster based on their processing and storage capabilities. Message transmissions among the nodes in a cluster may not have delay, loss and out-of-order properties. Even if they do, a cluster on the whole may have adequate capacity to store input and/or output batches to do the computations in correct order. Thus, our considerations in this paper need to be applied to inter-cluster communications only.

ACKNOWLEDGEMENTS

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grant 3182.

REFERENCES

[1] F. Bonomi, R. Milito, J. Zhu & S. Addepalli (2012)"Fog computing and its role in the internet of things", *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pp 13–16, New York, NY, USA, ACM.

[2] F. Bonomi, R. Milito, P. Natarajan & J. Zhu (2014) "Fog computing: A platform for internet of things and analytics", In N. Bessis and C. Dobre, editors, *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp169–186, Springer International Publishing, Cham.

[3] C. Chang, S. N. Srirama& R. Buyya (2017)"Indie fog: An efficient fog-computing infrastructure for the internet of things",*Computer*, Vol. 50, No. 9, pp 92–98.

[4] A. V. Dastjerdi& R. Buyya(2016)"Fog computing: Helping the internet of things realize its potential",*Computer*, Vol. 49, No. 8, pp 112–116.

[5] K. Vidyasankar (1991)"Unified theory of database serializability", *FundamentaInformatica*, Vol. 1, No. 2, pp 145-153.

[6] K. Vidyasankar (2018a)"Distributing computations in fog architectures", *TOPIC'18 Proceedings*. Association for Computing Machinery.

[7] K. Vidyasankar (2018b)"Atomicity of executions in fog computing architectures",*Proceedings of the Twenty Seventh International Conference on Software Engineering and Data Engineering (SEDE-18)*.

[8] N. Conway (2008)"Transactions and data stream processing", *Online Publication*, pages 1–28. http://neilconway.org/docs/stream_txn.pdf.

[9] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, & H. Wang (2015) " S-store: Streaming meets transaction processing",*Proc. VLDB Endow.*, Vol. 8, No. 13, pp 2134–2145.

[10] I. Botan, P. M. Fischer, D. Kossmann, & N. Tatbul (2012)"Transactional stream processing", *Proceedings EDBT*, ACM Press.

[11] L. Gürgen, C. Roncancio, S. Labbé& V. Olive (2006)"Transactional issues in sensor data management", *Proceedings of the 3rd International Workshop on Data Management for Sensor Networks (DMSN'06), Seoul, South Korea*, pp 27–32.

[12] M. Oyamada, H. Kawashima, & H. Kitagawa (2013)"Continuous query processing with concurrency control: Reading updatable resources consistently", *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pp 788–794, New York, NY, USA, ACM.

[13] K. Vidyasankar (2017) "On continuous queries in stream processing", *The 8th International Conference on Ambient Systems, Networks and Technologies (ANT-2017), Procedia Computer Science*, pp 640–647. Elsevier.

[14] L. Andrade, M. Serrano& C. Prazeres (2018)"The data interplay for the fog of things: A transition to edge computing with IoT",*Proceedings of the 2018 IEEE International Conference on Communications (ICC)*, IEEE Xplore.

[15] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips & E. de Lara (2017)"Cloudpath: A multi-tier cloud computing framework", *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pp 20:1–20:13, New York, NY, USA, ACM.

[16] storm.apache.org/releases/1.0.6/Transactional-topologies.html.

[17] Jin Li , Kristin Tufte, VladislavShkapenyuk, VassilisPapadimos, Theodore Johnson & David Maier (2008) "Out-of-Order Processing: A new Architecture for high-performance stream systems", PVLDB '08, pp 274-288, VLDB Endowment.

[18] Zhitao Shen, Vikram Kumaran, Michael J. Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche& Kim Macpherson (2015) "CSA: Streaming engine for internet of things", *Data Engineering bulletin*, Vol. 38, No. 4, pp 39-50, IEEE Computer Society.

[19] F. Xhafa, V. Naranjo, L. Barolli& M. Takizawa (2015)"On streaming consistency of big data stream processing in heterogeneous clusters", *Proceedings of the 18th International Conference on Network-Based Information Systems*. IEEE Xplore.