Scalable and Adaptable Distributed Stream Processing

Yongluan Zhou
National University of Singapore

Abstract

In this paper we introduce a new architectural design of a large scale distributed stream processing system. The system adopts a two layer architecture. Based on the locality and the natural administrative dependency of the processors, the processors are naturally partitioned into multiple independent entities. Processors within each entity compose the first layer while all the entities comprise the second one. Tightly coupled cooperation is employed within each entity due to the availability of central administration and their close locality. On the contrary, the entities cooperate with each other in a loosely coupled way. Challenges are identified in each layer of the architecture and techniques are proposed to solve them.

1. Introduction

Emerging applications, such as stock tickers, sports tickers, financial monitoring and network management, have fueled much research interest in designing stream processing engines [1, 5, 10]. These systems support complex continuous queries over push-based data streams. In the applications, such as financial market monitoring, which have potentially large number of clients, we envision that there would be a lot of business entities that provide stream processing services for a huge number of clients. One example of such kind of entity is `raderbo`.com. Instead of only providing stock quotes, it can evaluate user specified queries and deliver the results in a real time manner. Each of this kind of entities installs and runs its own stream processing engine. To enhance scalability, each entity can employ a cluster of processors. These processors are typically interconnected by a fast local network, under a central administration and expected to employ the same processing model.

A more ambitious service is to integrate the processing power and capabilities of the different entities to provide a central access portal to all the clients. We assume the participating entities cooperates based on business agreements and they are incited to process queries assigned to them. For example, an entity can be paid based on the length of time when it is executing the queries. We also assume there is a

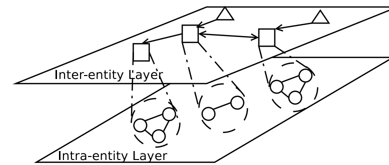

Figure 1. Two layer network

known global schema of the data. Each participating entity only needs to install a wrapper which is responsible to cooperate with other entities. Due to the possibility of employing different processing engines, different entities may have different data model, processing model as well as user interfaces. And it is not expected that the entities will surrender their administrations. Furthermore, the entities are typically interconnected by a widely distributed network. This brings different requirements to the architectural design. Instead of being tightly-coupled as the intra-entity processors, the inter-entity cooperation is preferred to be loosely coupled due to the lack of central administration.

Under the above observations, we can see that the network is naturally partitioned into two layers: the intra-entity layer and the inter-entity layer. This structure is illustrated in Figure 1, where the rectangles denote the entities and the circles denote the processors within the entities. The triangles are to represent the stream sources. This thesis aims at designing a scalable and adaptable distributed stream processing system under the above architecture. The properties of the system include the followings:

- The inter-entity cooperation is loosely-coupled. Each entity can operate its own processing engine and process the queries assigned to it without much interactions with other entities.
- The task of disseminating streams from the sources to the widely distributed entities are emphasized in our system. This differs from existing work which mainly focus on the query processing task. Adaptive data dissemination techniques are applied to efficiently disseminate the data streams from the sources to the widely distributed entities.
- Queries are adaptively allocated to the entities to minimize the communication cost of data stream dissemination

and to share the load among the entities.

• Within each entity, we assume there is a central administration over the whole entity. These processors are expected to employ the same processing engine and hence share a uniform data model, processing model and user interface. As such, we propose the processors cooperate in a more closely-coupled way to enhance efficiency. The proposed techniques are platform independent and hence can be applied to any existing processing engines. Note that this architecture is only one of all the possibilities. Each entity is not forced to employ any specific architecture due to the fact that they cooperate in a loosely coupled manner.

• A platform independent intra-entity dynamic operator placement scheme is proposed. Queries assigned to an entity are dynamically partitioned into multiple fragments which are distributed to the processors for processing.

• An adaptive distributed operator ordering architecture is proposed to adaptively change the ordering of operators that are distributed to multiple processors within an entity.

The rest of this paper is organized as follows. The architecture design for the inter-entity layer and the intra-entity layer are presented in Section 3 and Section 4 respectively. Section 5 presents the related work and Section 6 concludes the paper.

2. The Degree of Coupling

Although distributed stream processing has attracted significant research attention recently, we do not notice any explicit discussion of the degree of coupling for such a system. In general, the coupling of a distributed system is related to the degree of cooperation between the distributed nodes. We focus on the cooperation in the two main services provided by the system: *stream transfer* and *query processing*. The stream transfer service disseminates the data streams to the diverse processing nodes while the query processing service evaluate the submitted queries over the streaming data.

Table 1 presents the different types of cooperation in these two services. Here, we use a coarse-grained categorization. Inside each category, the adoption of different techniques would also affect the degree of coupling. To provide the stream transfer service, the distributed nodes in a non-cooperated system can connect to the sources directly and rely on the sources to transfer data to them. Alternatively, to enhance the communication efficiency, the nodes can be organized into an overlay network and transfer the streams to the nodes through their cooperation. For the query processing service, the processing nodes can be isolated and process their queries independently. To address the problem of load imbalance, the nodes can cooperate to share their processing loads. We further subdivide the cooperation into two subcategories: load sharing at the query level and the one at the operator (or finer) level. In the former one, load is distributed in the unit of the whole queries while it is in the unit of operators (or partitions of each operator) in the latter one. Generally speaking, the system is more tightly coupled in the latter case, because it requires the processing nodes to adopt the same data model and processing model.

Stream transfer	Query processing		
	Isolated	load sharing	
		query level	op or finer level
non-cooperated	all single-site engines	unexplored	[9, 11, 6]
Cooperated	[13]	Section 3	[2, 7, 8], Section 4

Table 1. Different degree of cooperation

The main advantage of a loose coupling system is the ease of deployment. With a looser coupling, the changes at one node would have less impact on the others. This results in a lot of desirable system characteristics. For example, we can adopt different stream processing engines in different nodes and upgrade the version of the engine in one node without the synchronization of the others. On the other hand, with a tighter cooperation, higher efficiency can be achieved, such as the load sharing and cooperated stream transfer mentioned above. Hence when we are designing the architecture of the system, the degree of coupling should be carefully chosen. We choose different degree of coupling in different layers of our system. Details are given in the following sections.

3. Inter-entity Layer

In the inter-entity layer, because the entities are under independent administrations, loosely coupled cooperation is preferred. First, entities may join or leave at any time which is out of control even without failure. Second, different processing engines might be installed in different entities because of different business decisions. These engines may have different data models and processing models. Thus, many tightly coupled cooperation techniques cannot be directly tapped upon. For instance, dynamically distributing the operators of a query to multiple entities violates the loosely-coupling property. Moreover, this may also not be feasible, e.g., moving a window join operator from the STREAM system to a TelegrahCQ system is hard to implement, because it relies on a special data structure “synopsis” implemented in STREAM which is not only manipulated by the join operator itself but also other operators before or after the join operator. Furthermore, even the entities use the same engine, one may upgrade its engine without informing

the others. The would also bring problems unless forward and backward compatibility is implemented. For a practical solution, we choose to operate at query level, i.e., a query is processed within a single entity. In the following subsections, we present how the entities cooperate to disseminate the data streams and distribute the user queries.

3.1. Data Stream Dissemination

Given that queries allocated to the various entities, the streams have to be transferred to the entities to feed the queries. A straightforward approach is to let the source nodes to feed the entities directly. However, relying solely on the sources to transfer data is not scalable to the number of entities. To address this issue, we allow the entities to cooperate with each other in transferring data streams rather than only relying on the sources. The entities are organized into multiple hierarchical tree structure, which is widely adopted in large scale data dissemination systems. Each parent entity in a tree is responsible to transfer the upstream data to its children. Hence each entity only needs to transfer streams to a limited number of entities. The shapes of these trees have significant impact on the dissemination efficiency which deserve further study. The second issue is how each entity forward data to its children. The simplest approach is to forward all the received data to its children. This incurs a lot of unnecessary data transfer if a child does not require all the data. Due to the large volumes and continuity of streaming data, minimizing the communication cost in the system is critical. We allow each entity to express its data requirement which will be used to perform early filtering and transforming at its ancestors. This brings the issue of how to represent the data interest of the different queries as well as how to efficiently compute the aggregation of data interest from different queries.

3.2. Query Distribution

The above discussion assumes that the queries are already allocated to the entities. To distribute queries, we should consider the following issues.

3.2.1. Coordinator Tree Construction. Queries in our application may arrive very quickly. We refer to them as the query streams. The query allocation algorithm should be scalable to fast query streams. To address this problem, we adopt a hierarchical coordinator-based approach. The coordinators are organized as a hierarchical tree. Queries are distributed level by level down the tree. An internal coordinator distributes query to its child coordinators. The queries are finally distributed to the entities by the leaf coordinators. A higher level coordinator distributes queries based on coarser information. In this paper, we adapt the distributed mechanism proposed in [3] to dynamically construct a hierarchical tree of coordinators. The mechanism tries to maintain a tree with the following properties: (1) the size of the cluster in each level (except the root and the second to root level, where the size is less than $3k-1$) is between k to $3k-1$; (2) the parent of a cluster is the geographical center. The tree is constructed incrementally and dynamically.

1. When a new node requests to join the network, its request will be first directed to the root coordinator. For each node that receives a join request, if it is the leaf coordinator, it will add the joining node as its child node. Otherwise, it will identify the child coordinator closest to the joining node and direct the request to that child.
2. If a node leave the network, a message is sent to its parent and children (if any). If it is a coordinator, a new parent is reselected among its remaining children. Furthermore, heartbeat messages are sent periodically among the parent and children to detect any node failure.
3. If a coordinator finds out that the number of its children exceeds $3k-1$, it will partition the cluster into two clusters, each of size at least $\lfloor 3k/2 \rfloor$, such that the radii among the two clusters are minimized. The center of the two clusters are selected as the two new parents.
4. If the number of children of a coordinator x falls below k, it will send a merge request to the closest sibling y. The sibling adds all the children of x to its children.
5. Periodically, a new parent will be selected if the current parent is no longer the center among its cluster.

3.2.2. Load Distribution. To achieve maximum system utilization and minimum processing latencies, load balancing among the processors is desirable. Moreover, the data interest of different queries may significantly overlap. The transferring of these data can be shared among the queries to reduce the communication cost. This suggests that queries with similar data interest should be allocated to the entities that are close to each other in the dissemination trees. To solve the above two problems, we model the query distribution problem as a graph partitioning problem. Each vertex in the query graph corresponds to a query and there is an edge between two vertices if there is overlap in their data interest. A vertex is weighted by the workload incurred by the query and an edge is weighted with the estimated arrival rate (bytes/second) of the data of interest to both end vertices (queries). The problem is modeled as: given a graph $G = (V, E)$ and the weights on the vertices and edges, dynamically partition V into k disjoint partitions such that each partition has a specified amount of vertex weights and the *weighted edge cut*, i.e. the total weight of the edges connecting vertices in different partitions, is minimized.

Figure 2 illustrates an example query graph that comprises 5 queries. The weights of the vertices and edges are drawn around them. If, for example, we have to allocate the queries to two entities, we can consider two plans: (a) allocate Q_3 and Q_4 to one entity and the rest to another; (b) allocate Q_3 and Q_5 to one entity and the others to another. Both the two plans can achieve load balance. However,

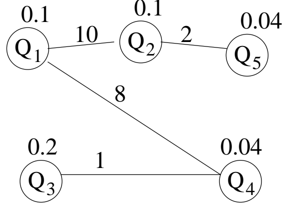

Figure 2. Query Graph

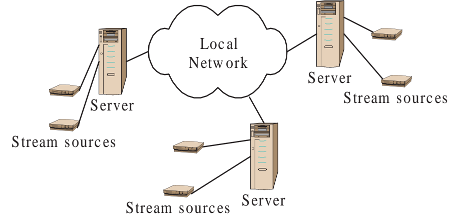

Figure 3. The structure of an entity

plan (b) has a better communication efficiency, where only 3 (bytes/second) of duplicate data are transferred to both nodes, while in plan (a) 8 (bytes/second) of duplicate data are transferred. Note that only considering allocating similar queries together may not result in good performance. As can be seen from the above example, Q_3 and Q_5 are not similar in their data interest but allocating them together results in a better scheme.

At runtime, the system is subject to changes, e.g., the evolving of the data interest of the queries and the load incurred by the queries, arrival or leave of queries etc. Hence, the query graph should be adaptively repartitioned if a better solution can be found. One approach is to repartition the query graph from scratch. This may result in a relatively optimal partitioning but with a long decision making time and a large number of query movements. Another approach is to cut some vertices from the overloaded partitions to other underloaded partitions without considering the relationship of overlap in data interest. This approach can achieve small query migration time and decision making time. However, communication efficiency might be unsatisfactory due to the large overlap of data interest between different partitions. Hence a desirable approach should be able to achieve a trade-off between these two extremes.

4. Intra-entity Layer

Within each entity, processors are under a central administration and interconnected by a fast local network, which eases the employment of tightly coupled techniques. Because we assume entities may adopt different processing engines, the techniques proposed should be independent on the actual employed engines.

The first problem is how to receive the data feed by the upstream entity and forward them to the downstream entities. Relying on a single processor to receive all the streams is not scalable. Hence, we assign a processor as the delegation of each data stream that is sent to the entity. The delegation processor is responsible to route the streams to other processors in the same entity as well as to transfer the streams to the child entities. Figure 3 shows an overview of the structure of an entity. Given a delegation scheme, there are two important query optimization issues: operator placement and operator ordering.

4.1. Operator Placement

To achieve better performance, we adopt a finer grained load distribution scheme than that used in the inter-entity layer. A single query is dynamically partitioned into multiple query fragments and distributed to the processors for processing to minimize the delay of query results. At a closer look, the delay d_k includes the time used in evaluating the query (denoted as p_k), the time waiting for processing as well as the time it is transferred over the network connections. For a specific processing model and a particular operator ordering, we regard the evaluation time p_k as the inherent complexity of the query. Since different queries may have different inherent complexities, the value of d_k cannot reflect correctly the relative performance of different queries. For example, a query may experience a long delay because its evaluation time is long. However, in a multi-query and multi-user environment, we wish to tell the relative performance of different queries. Hence we propose a new metric *Performance Ratio (PR)* to incorporate the inherent complexity of a query. The PR of a query q_k is defined as $PR_k = \frac{d_k}{p_k}$. Our objective is to minimize the worst relative performance among all the queries, i.e. $PR_{max} = \max_{1 \leq k \leq Q} PR_k$, where Q is the total number of queries.

We identified several heuristics to achieve the above goal. First, an arrived tuple has to wait for processing while a processor is busy. And the length of the busy period of a processor depends on the workload imposed upon the processor. Hence, to minimize the waiting time, load balancing among the processors is desirable. Second, the communication delay of a tuple is equal to the product of the transfer latency and the times that the tuple is transferred over the network. This suggests the second heuristic: distribute operators of a query to a restricted number of nodes so that communication overhead of a query is limited. We call the maximum of this number as the distribution limit of that query. Third, the bandwidth of each processor is limited and we have to avoid high communication traffic. Hence the third heuristic is to minimize the communication traffic under the first two heuristics.

4.2. Operator Ordering

To address this problem, distributed plan adaptation techniques [12] can be adapted to our context. In [12], we proposed a distributed query processing architecture which facilitates the adaptation of operator orderings. However, it relies on a specific processing engine: TelegraphCQ. In this paper, we extend it to a platform independent scheme. There is an Adaptation Module (AM) that intercepts the input and output stream of the processing engine. Given an operator placement, the output stream of a processor may need to be processed by multiple processors in different orders. A set of candidate downstream processors are generated when a query fragment is (re)placed onto a processor. The AM continuously collects statistics of these candidate processors, such as workload, selectivities of the query fragments and the bandwidth usage etc. Based on these statistics, the AM adaptively chooses the immediate downstream processor for an output tuple.

5. Related Work

We categorize exisiting work on stream processing systems based on the types of the adopted cooperatoion techniques, which are listed in Table 1.

Early work on stream processing focus on building single-site stream processing engines[5, 4, 10]. Recently, attention has been paid to distributed stream processing. Flux [9] and Borealis [11] adopted similar system architecture and assumptions. A cluster of processors is employed to enhance the scalability of the processing engine. The network connections are assumed to be very fast and hence the communication cost is ignored. The techniques focus on sharing the load between the processors in a fine granularity. However, the streams are disseminated directly from the sources to the processors. Furthermore, the load distribution problems in the above two pieces of work are essentially partitioning problems, as all the processors are identical in terms of the assignment of operator/stream partitions. However, our intra-entity operator placement problem is an assignment problem (due to the stream delegation scheme), which requires different solutions.

Project Medusa [6] also proposed an architecture to integrate multiple administratively independent entities. In their load distribution algorithm, it is assumed that the entities employ the same type of processing engine so that operators of a query can be distributed to multiple entities for processing. Furthermore, their architecture does not address the problem of transferring the data steams to large number of processors and rely solely on the sources to disseminate the streams. Literatures [2], [7] and [8] proposed techniques of in-network stream processing. Query operators are allocated along the path from the sources to the clients to enhance the communication efficiency. The major goal of the load sharing is to minimize the communication cost instead of load balancing. Furthermore, they assume an overlay mesh already exists and focus on searching the place to allocate the query operators along the data transfer path. On the contrary, [13] focuses on the problem of constructing an overlay structure to transfer streaming data to a large number of processing servers without considering load sharing.

6. Conclusion

In this paper, we reexamined the problem of designing a scalable distributed stream processing system. A new architectural design is presented. The system composed by two layers: the inter-entity layer and the intra-entity layer. A few performance issues in the new system structure are identified. Initial solution to these issues are presented. We are now trying to integrate the system with various single site processing engines and then plan to deploy it onto real network environment

References

[1] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] Y. Ahmad and U. Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, 2004.

[3] S. Banerjee et al. Scalable application layer multicast. In *SIGCOMM*, 2002.

[4] D. Carney et al. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.

[5] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[6] M. Cherniack et al. Scalable distributed stream processing. In *CIDR*, 2003.

[7] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.

[8] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[9] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[10] The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*.

[11] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[12] Y. Zhou, B. C. Ooi, K.-L. Tan, and W. H. Tok. An adaptable distributed query processing architecture. *Data Knowl. Eng.*, 53(3):283–309, 2005.

[13] Y. Zhou, B. C. Ooi, K.-L. Tan, and F. Yu. Adaptive reorganization of coherency-preserving dissemination tree for streaming data. In *ICDE*, 2006.