# Rethinking the Design of Distributed Stream Processing Systems

Yongluan Zhou[†] Karl Aberer[†] Ali Salehi[†] Kian-Lee Tan[‡*]

[†]EPFL, Switzerland [‡]National University of Singapore

## Abstract

*In this paper, we present a novel architecture to support large scale stream processing services in a widely distributed environment. The proposed system, COSMOS, distinguishes itself by its loose coupling and communication efficiency. To exploit the sharing of data transfer incurred by different queries and to break the tight coupling of the distributed nodes, a new communication paradigm, content-based network, is employed. We discuss the design and the challenges of this system.*

## 1. Introduction

There is an emerging interest from the database community to build large scale stream processing services in a widely distributed environment [2, 4, 13]. In such systems, the communication cost can be very high as it may involve inter-country and even intercontinental communication. Furthermore, streams are typically of a very high rate and are transferred persistently. Hence, achieving communication efficiency should be an important objective in the system design. To achieve this goal, we should carefully design the communication substrate.

Unfortunately, existing systems do not pay much attention to this problem. They simply adopted the unicast communication paradigm and focused on optimization algorithms that allocate the operators of each user query along the overlay path from the source to the end user [4, 13]. There are several problems with these existing systems.

First, it is hard to exploit users' common data interest to minimize the communication cost. For example, two users in two nearby countries in Europe, respectively, may be interested in the stock market of the New York Exchange and submit their queries to two servers in their own countries respectively. One approach is to plan the two queries separately. The requested source streams as well as the intermediate result streams of these two queries would be transferred separately even though they may share a large amount of common contents. This incurs unnecessary overheads because these streams may have similar transfer path due to the proximity of their destinations. With a large number of user queries, such overhead would be overwhelming. While performing multiple query optimization may alleviate this problem, it impairs the system's scalability. For example, in [2], the authors proposed to generate a giant operator graph for all the queries submitted to the system. However, no scalable algorithm has been proposed to achieve this goal so far.

Second, queries and stream sources are exposed to each other. The sources not only have to transfer data for every relevant query but also have to keep track of all of them. Such tightly-coupled architecture is undesirable for a large-scale system.

Looking from a different angle, these two problems can be alleviated if we have a "smarter" communication substrate that can automatically exploit the opportunities to share communication among different queries and break the tight-coupling between the sources and users.

The above observation brings our attention to Content-based network (CBN) which is a multi-cast like networking method emerging in recent years [8]. In a CBN, each datagram consists of several attribute-value pairs. A node in the network can express its data interest as a few selection predicates on the attributes of the datagram. The sources and the destinations are not known to each other. The sources simply pass the datagrams to the CBN and the datagrams will be routed by the network based on the data interest of the receivers. One can see that a CBN has the merits of a multicast network (i.e., communication for common items is shared, and the data sources and receivers are loosely coupled) and achieves high communication efficiency by providing a powerful interface to express data interest.

In this paper, we propose the design of our scalable distributed stream processing system: COSMOS (COoperative and Self-tuning Management of Streaming data). COSMOS provides an efficient stream query processing service for a large number of users. Contrary to existing systems, COSMOS employs a CBN as its communication substrate.

*Kian-Lee Tan is partially supported by research grant R-252-000-237-112 from National University of Singapore.

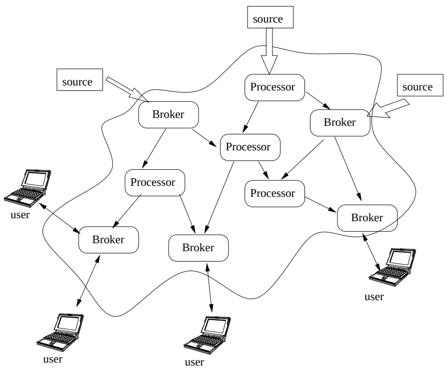



**Figure 1. Overview of COSMOS**

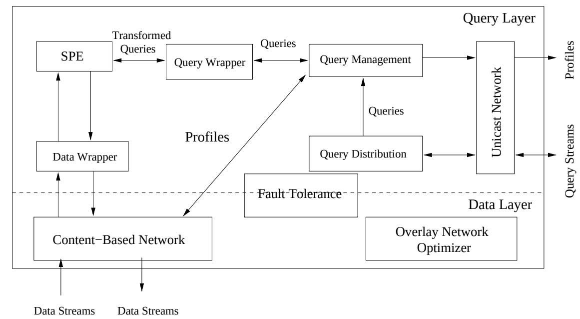



**Figure 2. Architecture of a processor**

Figure 1 shows the overview of COSMOS. The service is backed by a number of distributed servers interconnected with a widely distributed overlay network. These servers are autonomous and may join or leave the system anytime. Furthermore, they have different capabilities due to their different hardware and software configurations. Some of these servers are only used to route data across the network while others are equipped with stream processing engines (SPE) and hence is able to process complex continuous queries. We refer to the former type of servers as *brokers* and the latter ones as *processors*. A number of data sources continuously publish their data to the network through the servers. User queries submitted to the system are specified in high level SQL-like language statements such as CQL [16].

This paper mainly discusses the various issues to efficiently employ CBN into a distributed stream processing system. The rest of this paper is organized as follows. Section 2 provides an overview of the COSMOS system. Section 3 presents the data layer. Section 4 presents the query layer. Section 5 presents some results from a preliminary study. Finally, we conclude in Section 6

## 2. COSMOS System Overview

Figure 2 shows the architecture of a processor in the system. There are two layers of modules in the architecture: the query layer and the data layer. The "ordinary" brokers only have the data layer module. Contrary to existing systems, each processor in COSMOS can be under different administrations and run by different entities. Hence, COSMOS allows different stream processing engines (SPE) or different versions of the same SPE to be installed in different processors. Existing single site SPEs such as TelegraphCQ [9], STREAM [16] and Aurora [7] can be employed in COSMOS. For each type of SPE, a data wrapper and a query wrapper can be plugged into the system to translate the data and the queries between COSMOS and the SPE.

In COSMOS, a user first connects to a broker/processor which works as the proxy for the user and is responsible for retrieving the result stream from the network and sending it back to the user. User queries are specified in an SQL-like language similar to CQL. They are handled by the query layer. A user query is first distributed to a processor, say $n_i$, by the load management service (provided by the Query Distribution module in Figure 2) for processing. The query management module of $n_i$ will analyze the query, and a new query or a modification of an existing query is sent to the SPE. The details of this procedure will be presented in the following sections.

Source data are transferred around the system through the data layer, which is mainly composed by a content-based network. The data sources advertise the source streams that they provide and the processors subscribe to these source streams by submitting the data interest profiles to the data layer. Furthermore, the processors would also advertise the result streams that they generate and users will subscribe to these result streams through the content-based network.

The fault tolerance function of COSMOS is also divided into two layers. The module at the query layer is responsible for recovering the processing of queries from failures, while the one at the data layer is targeted at providing highly available data transmissions service. The former function has already been extensively studied in existing stream systems [6, 11, 15]. Due to the adoption of different communication infrastructures, these literatures do not study the latter function. However, we will not discuss the fault tolerance function further in this paper due to the space limit.

## 3. Data Layer

In most existing stream management systems, data streams are modeled as relations that are continuously being appended. However, traditional CBN does not have the notion of "relation". Hence we have to first enhance the CBN to be aware of streaming relations. Each stream is assigned a unique name in COSMOS. In our current system, if the number of streams is small, the schema information of the streams will be flooded to every node upon its arrival. Otherwise, we use a DHT architecture to store the schema information while using the unique stream name as the hashing key.

### 3.1. Data Interest/Profile Subscription

Following traditional CBN, we can compose a data interest profile as follows. Each profile is a disjunction of a few datagram filters. Each filter is defined on one stream and is only applicable to this stream. Furthermore, a filter is a conjunction of constraints on the values of a set of attributes from the stream that the filter is defined on. A datagram is said to be covered by a filter, if the datagram is from the data stream of the filter and satisfies all the constraints in the filter. Furthermore, a datagram is covered by a profile if it is covered by any filters in the profile.

To exploit more opportunities to reduce communication cost, we extend CBN to perform projections. Early projection can save the cost of transmitting unnecessary attributes. Hence, in addition to the filters mentioned above, each profile also contains one set of attribute names for each of its requesting streams. When a node receives a datagram, it first finds out which stream the datagram is from and then evaluates the corresponding filters on the datagram. For each profile that has a filter being satisfied by the datagram, the projecting attribute set of the corresponding stream is retrieved and the projection operation is done on the datagram.

In summary, a profile $p_i$ is a triple $\langle \mathcal{S}, \mathcal{P}, \mathcal{F} \rangle$, where $\mathcal{S}$ is a set of stream names, $\mathcal{P}$ specifies the set of attributes of streams in $\mathcal{S}$ that are of interest, and $\mathcal{F}$ is a set of filters applied to streams in $\mathcal{S}$ in a similar form as traditional subscription profiles.

### 3.2. Overlay Network Optimizer

In a CBN, nodes are organized into an overlay network. The structure of the overlay network is critical to the communication efficiency. Its optimality depends on a lot of system parameters such as the capability and workload of the servers, overlay link delay, etc.

The overlay network optimizer periodically monitors the status of the network and performs the reorganization of the overlay network if necessary. Currently the nodes in COSMOS are organized into multiple overlay dissemination trees. Each optimizer module at each node monitors the workloads and connections of its neighbors in the overlay trees. By using a configurable cost function defined on these parameters, it estimates whether a local reorganization of the overlay trees is beneficial [18, 19].

## 4. Query Layer

As mentioned before, the query management module is responsible for composing the data interest profiles for the local processor to retrieve the data for processing and the profiles for the users to retrieve the results.

For each query, a profile is composed for retrieving the source data. The selection predicates applied to each individual source stream are extracted to compose the filters of the profile. Then a projection predicate is composed by using all the attributes in the query. Consider the following query as an example:

```
SELECT R.A, S.C
FROM   R [Now], S [Now]
WHERE  R.B=S.B AND R.A>10
```

Then the profile $\langle \mathcal{S}, \mathcal{P}, \mathcal{F} \rangle$ to retrieve the source data can be composed as follows: $\mathcal{S} = \{R, S\}$, $\mathcal{P} = \{R.A, R.B, S.B, S.C\}$, $\mathcal{F} = \{R.A > 10\}$.

In existing stream processing engines, different result streams are generated for different queries and transferred to the users independently. This is because users are assumed to be directly connected to the server in traditional systems. Following this approach, we can also compose one profile for each user to retrieve the result stream. First, a unique stream name is assigned to the result stream. Then a profile can be composed by using this unique stream name without filter and projection predicates.

However, this approach does not exploit the sharing of result stream delivery among different queries and may result in large communication overhead in our system as illustrated by the following example.

Table 1 lists a few queries drawn from an auction stream monitoring application specified using CQL [16]. The schema of the two streams are:

- OpenAuction (itemID, sellerID, start_price, timestamp)
- ClosedAuction(itemID, buyerID, timestamp)

Consider the join queries, $q_1$ and $q2$, presented in Table 1. We can see that the result tuples of $q_1$ and $q_2$ have overlaps in their result streams (since the auctions closed within five hours contain those closed within three hours). Furthermore, their projection attributes also overlap. Consider an overlay structure depicted in Figure 3(a). Nodes $n_3$ and $n_4$ issue two queries $q_1$ and $q_2$ respectively. These two queries are allocated to node $n_1$ which is equipped with a stream processing engine (SPE). Using traditional techniques, their result streams, $s_1$ and $s_2$, will be transmitted separately as shown in Figure 3(a). Hence the overlapping contents of $s_1$ and $s_2$ will be transmitted twice over the link between $n_1$ and $n_2$.


(a) Non-Share (b) Share

**Figure 3. Result Stream Delivery**

**Table 1. Example queries**

| | |
|---|---|
| $q_1$: Report all auctions that closed within three hours of their opening | |
| SELECT | O.* |
| FROM | OpenAuction [Range 3 Hour] O, ClosedAuction [Now] C |
| WHERE | O.itemID = C.itemID |
| $q_2$: Report the items and buyers of auctions closed within five hours of their opening | |
| SELECT | O.itemID, O.timetamp, C.buyerID, C.timestamp |
| FROM | OpenAuction [Range 5 Hour] O, ClosedAuction [Now] C |
| WHERE | O.itemID = C.itemID |
| $q_3$: Report all auctions that closed within five hours of their opening and their buyers | |
| SELECT | O.*, C.buyerID, C.timestamp |
| FROM | OpenAuction [Range 5 Hour] O, ClosedAuction [Now] C |
| WHERE | O.itemID = C.itemID |

Note that existing multi-query optimization techniques, such as [12, 5] also suffer from this problem. For example, one shared join operator can be created for the above two queries. However this join operator would still generate two separate result streams for the two queries respectively.

To solve the problem, we should send one result stream $s_3$ to $n_2$, which is the superset of both $s_1$ and $s_2$, and "split" it into two streams $s_1$ and $s_2$ at node $n_2$ (Figure 3(b)). To implement this scheme, one approach is to re-engineer a "specialized" SPE which can generate one result stream for multiple queries. However, such an intrusive approach is not desirable as it requires complex "low-level" software development and tightly coupling interaction between the SPE and the distributed system.

Instead, we propose a query reformulation approach. For a group of queries that have overlapping results, our method composes a new query $q$ that contains all the queries in this group, i.e. the result of $q$ is a superset of the result of each query in this group. For example, we can create a new query $q_3$ listed in Table 1 which contains $q_1$ and $q_2$ and issue $q_3$ to the SPE at $n_1$ instead of $q_1$ and $q_2$. The result stream $s_3$ will be "split" at $n_2$ using the filtering mechanism of the data layer. More specifically, the following two profiles are sent to $n_2$ by $n_3$ and $n_4$ respectively:

- $p_1$: $\mathcal{S} = \{s_3\}, \mathcal{P} = \{O.*\}, \mathcal{F} = \{-3(hour) \leq O.timestamp - C.timestamp \leq 0\}$.
- $p_2$: $\mathcal{S} = \{s_3\}, \mathcal{P} = \{O.itemID, O.timetamp, C.buyerID, C.timestamp\}, \mathcal{F} = \{-5(hour) \leq O.timestamp - C.timestamp \leq 0\}\rangle$

Tuples that pass $p_1$ will be sent to $n_3$ and those that pass $p_2$ will be sent to $n_4$. The mechanism of composing of $q_3$ and the two profiles will be explained later.

In our approach, each processor maintains a number of query groups such that queries inside each group have overlapping results and it is beneficial to rewrite these queries into one query $q$ which contains all the member queries $q_i$. Such a query $q$ is called the representative query of the query group. The benefit of the rewriting can be estimated as $\sum_i C(q_i) - C(q)$, where $C(q)$ is the estimated rate(bps) of the result stream of $q$.

Query containment and equivalence is a fundamental problem which has been extensively studied in the literature, e.g. [14]. We, however, need to extend these techniques to the continuous stream query context. Some related literature studies the use of views to answer user queries [10]. This direction studied how to rewrite a query such that the given views of the underlying relations can be utilized to answer the original query. However, our work is kind of the other way round. We have to compose a "view" of the streams that can be utilized to answer multiple queries using the simple filtering mechanism in a CBN.

First of all, we should extend the query containment and equivalence definition of traditional queries to continuous stream queries. Traditionally, a query is said to contain another query if its result data contains that of the other one. However, in the continuous query context, the result data are continuously generated and hence this traditional definition is no longer applicable. To address this problem, we assume there is an application discrete time domain $\mathcal{T}$ where the timestamps of the input stream data are drawn from. The temporal result data of a query $q$ evaluated on a stream instance $S$ at the time instance $\tau \in \mathcal{T}$ is denoted as $q(S, \tau)$, which is the result of evaluating $q$ over all the data from $S$ with timestamps smaller or equal to $\tau$.

**Definition 1** *A continuous query $q_1$ is contained by another continuous query $q_2$, denoted by $q_1 \sqsubseteq q_2$, if for all stream instances $S$, $q_1(S, \tau)$ is a subset of $q_2(S, \tau)$ at any application time instance $\tau$.*

The second problem is how to determine the containment relationship between two continuous queries. We assume that there is an approach to determine such relationship between two traditional non-continuous queries. The major difference between continuous stream query and traditional query is the introduction of window semantics. In a typical continuous query over data streams, each source stream is associated with a window predicate. In this paper, we only consider the time-based sliding window predicate introduced in CQL [16]. A window predicate $w(T)$ takes a positive time-interval $T$ as a parameter and defines a temporal relation composed by tuples arrived within the last $T$ time units, where $T$ ranges from zero to infinity. Note that if all window predicates have a parameter $T = \infty$, we can use the traditional approach to determine the containment relationship by simply ignoring the window predicates.

For queries with window predicates, we have the following lemma and theorems.

**Lemma 1** *For a query with only a window-based join operation of two streams $S_1$ and $S_2$ with window sizes of $T^1$ and $T^2$ respectively, two tuples $t_1$ from $S_1$ and $t_2$ from $S_2$ generate a join result tuple $t$ if and only if both the following conditions are true:*

*(1) they satisfy the join predicates;*

*(2)* $-1 \cdot T^1 \leq t_1.timestamp - t_2.timestamp \leq T^2$.

**Theorem 1** *A select-project-join continuous query $Q_1$ is contained by another select-project-join continuous query $Q_2$ if both the following conditions are true:*

*(1)* $Q_1^\infty \sqsubseteq Q_2^\infty$*, where $Q_i^\infty$ is a query resulted from setting all the window sizes of $Q_i$ as $\infty$;*

*(2)* $T_1^i \leq T_2^i$*, where $T_j^i$ is the window size of the $i$th stream of query $Q_j$.*

**Theorem 2** *An continuous aggregate query $Q_1$ is contained by another continuous aggregate query $Q_2$ if both the following conditions are true:*

*(1)* $Q_1^\infty \sqsubseteq Q_2^\infty$*, where $Q_i^\infty$ is a query resulted from setting all the window sizes of $Q_i$ as $\infty$;*

*(2)* $T_i^1 = T_i^2$*, where $T_i^j$ is the window size of the $i$th stream in query $Q_j$.*

With the above lemma and theorems, we can generate the representative query for each group of queries. For simplicity, queries in a group should involve the same set of streams (i.e. with the same FROM clause in SQL) and the same aggregation functions (if any) with the same grouping conditions. The representative query is generated by merging the query predicates (selection predicates, join predicates, window predicates, etc.).

Profiles are also generated for the users to retrieve their query results from the result stream of the representative query. It is actually to re-tighten the constraints that have been "loosen" in the representative query.

The benefit of the grouping of queries is estimated as the difference between the expected output data rate of the representative query and the sum of the original queries. An incremental greedy algorithm is used to optimize the query grouping, where each new query is assigned to the query group that can achieve the maximum benefit.

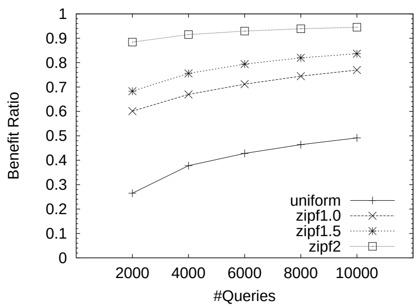
## 5. Preliminary Experiments

In this preliminary experiment, we examine our query management techniques, i.e. the query merging techniques presented in Section 4.1. The system is implemented in Java. We adopt the stream processing system: GSN (Global Sensor Network, `http://gsn.sourceforge.net/`), as the underlying SPE. The experiment is conducted in a Linux server with 2 Dual-Core 2.66GHz Intel CPU and 4G memory.

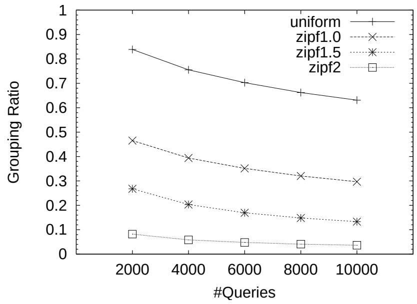
We use the sensor dataset generated by the SensorScope project (`http://sensorscope.epfl.ch`), which measures key environmental data such as air temperature and humidity etc. In the experiments, we use 63 streams and emulate the streaming scenario by using their timestamp information. Queries are generated by randomly selecting the involved streams, their window sizes and the filtering predicates based on a distribution (uniform or zipfian).

The CBN is simulated in the experiments. The topology generator BRITE (`http://www.cs.bu.edu/brite/`) is used to generate a power law network topology with 1000 nodes. Then a minimum spanning tree is constructed as the dissemination tree. All the experiments are repeated 20 times with different random queries and the average results are reported.

In Figure 4(a), we present the *benefit ratio* at the point when a certain number of queries have been inserted. *Benefit ratio* is computed as the percentage of communication cost that is reduced by the query merging algorithms in comparing to that without merging. The distribution used to generate the queries varies from uniform to zipfian with different skew parameters. One can see that, with more number of queries added to the system, there are more opportunities for the query merging approach to explore the sharing of communication and hence a higher benefit ratio can be achieved. Another interesting point is that query merging is more beneficial with a skewed query distribution. The reason is obvious. With more queries interested in the same subset of data, the probability that we can merge the queries would be higher. Figure 4(b) provides another perspective on the experimental results. The *grouping ratio* is the ratio of the number of query groups to the total number of queries. Generally, the lower the grouping ratio, the higher the benefit ratio could be.




(a) Benefit Ratio




(b) Grouping Ratio

**Figure 4. Query Grouping Performance**

# 6. Conclusion

In this paper, we revisit the design of a scalable distributed stream processing system. The proposed novel architecture leverages the recent work from the networking community, CBN, to enhance the system's scalability. The new architecture can achieve both loose coupling and high communication efficiency. The issues on how to efficiently utilize CBN for stream processing are discussed. Preliminary experimental result suggests that our techniques can efficiently utilize the capability of CBN to optimize the system performance.

# References

[1] `http://serl.cs.colorado.edu/~carzanig/siena/`.

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

[3] K. Aberer and M. Hauswirth and A. Salehi Infrastructure for data processing in large-scale interconnected sensor networks. In *MDM*, 2007.

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

[5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[6] M. Balazinska et al. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD*, 2005.

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

[8] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *Infrastructure for Mobile and Wireless Systems*, 2001.

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

[10] A.Y. Halevy. Answering queries using views: A survey. In *VLDB J.*, 2001.

[11] J.-H. Hwang, M. Balazinska et al. A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, 2005.

[12] S. Madden et al. Continuously adaptive continuous queries over streams. In *SIGMOD* , 2002.

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

[14] Y. Sagiv, M. Yannakakis. Equivalences Among Relational Expressions with the Union and Difference Operators. *J. ACM*, 1980.

[15] M. A. Shah et al. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD*, 2004.

[16] The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 2003.

[17] Y. Zhou, B. C. Ooi, et al. Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In *CoopIS*, 2006.

[18] Y. Zhou, B. C. Ooi, et al. Adaptive Reorganization of Coherency Preserving Dissemination Trees for Streaming Data. In *ICDE*, 2006.

[19] Y. Zhou, B. C. Ooi and K.-L. Tan. Disseminating streaming data in a dynamic environment: an adaptive and cost-based approach. *The VLDB Journal*, accepted for publication.