CS848 Presentation Report

TelegraphCQ: Continuous Dataflow Processing for an Uncertain World

Zhaohua Li
20156677

School of Computer Science
University of Waterloo

February 14, 2006

Abstract

Increasingly pervasive networks fuel the research and development of new data management technologies to deal with the streaming data and continuous queries that are typical of data stream applications. TelegraphCQ is the next generation of Telegraph System, which is a highly adaptive streaming dataflow system with shared processing that uses building blocks from conventional relational database systems.

1 Introduction

With Web, Internet, and increasingly pervasive networks, applications in which the data is modeled best as transient data streams have become widely recognized. Examples of such applications include financial applications, network monitoring, sensor networks, and others. The characteristics of the data stream model and queries on it put new challenges to traditional data management technology [1,3]:

- Some or all of the input data that are to be operated on are not available for random access from disk or memory. Instead, the data arrive online in the form of continuous data streams. So in the query processing of streaming data, the data is continuously "pushed" to the query processor, but not "pulled" from the disk like in the traditional database.
- The arrival rate of the data stream may be very high and bursty, and most of the applications require real-time response to the queries, this puts constrains on processing time and memory usage.
- Time and ordering are very important in many data streams
- Volatility and unpredictability of the environment might cause data lost, unavailability of load information, statistics and cost estimates about the data
- Continuous queries (CQ) is an important class of queries for data stream applications, since these applications have a monitoring or filtering aspect and the queries are evaluated continuously as data streams continue to arrive. Algorithms for CQ with acceptable processing time for real-time application are on demand.
- Ad-hoc queries complicate the system. These queries might come from the users who directly interact with the application in many cases. To correctly answer these queries the historical data may be needed, and potentially they have been discarded already.

Telegraph project at UC Berkeley starts to address some of these challenges outlined above by using adaptive query processing in early 2000. The first version of Telegraph supports Federated Facts and Figures (FFF). It made no attempt to support shared processing over streams. Even though the two recent prototypes, CACQ and PSoup, demonstrated substantial advantages of combining the adaptive framework with the shared processing over streams, they had significant limitations, such as (1) they restricted their processing to data that could fit in memory, (2) they did not investigate scheduling and resource management issues for queries with little or no overlap, (3) they did not explore the opportunities for varying the degree of adaptivity to tradeoff flexibility and overhead, and (4) they didn't explicitly deal with the Quality of Service (QoS) for adapting to resource limitation.

Building on these initial prototypes, the TelegraphCQ is the completely redesigned and reimplemented version with the focus on support for shared, continuous query processing over static tables and data streams.

Let's first look at the building blocks in Telegraph system.

2 Adaptive Building Blocks

The main components of Telegraph are shown in **Figure 1** [1]. It consists of a set of dataflow modules or operators that produce or consume records in a manner that is analogous to the operators used in the traditional database query engines.

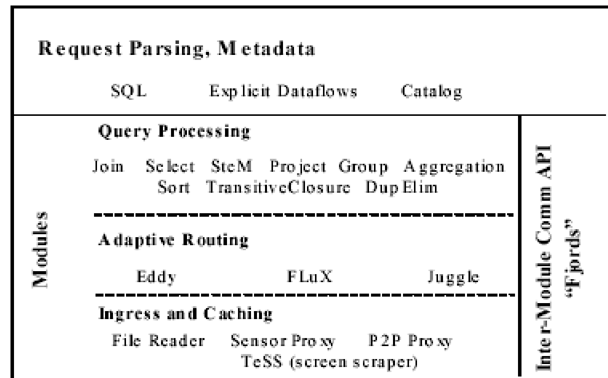

Figure 1: Telegraph Architecture

2.1 Ingress and Caching Modules

These modules are interfaced with the external data sources. They can reach out to remote data sources and may also cache data locally to hide the network delays. They might have more sophisticated functions, such as sending responses to the network.

2.2 Adaptive Routing Modules

Telegraph constructs a query plan that contains adaptive routing modules, which are able to "re-optimize" the plan on a continuous basis while a query is running.

- **Eddies**: An Eddy operator continuously routes tuples among a set of other modules according to a routing policy. Eddy determines the order in which to apply operators by observing their recent cost and selectivity and routing tuples accordingly [5].
- **Juggle:** The Juggle operator performs the online reordering for prioritizing records by content, the interesting records are processed first, and the user can dynamically change the definition of "interesting" during the course of an operation [7].
- **Flux:** Flux module is used for scaling a query in the volatile environment. It is placed between a producer-consumer operator pair in a pipelined, partitioned dataflow. Its functions are including: (1) Data partitioning and routing across a shared nothing cluster, (2) Load balancing: it online repartition the input stream and internal operator states on the consumer side, (3) Fault tolerance: Flux module can replicate an operator's internal state and in-flight data, in case of failure, it automatically recovers the in-flight data and operator state and continues processing without human intervention [1].

2.3 Query Processing Modules

In Telegraph, query processing is performed by routing tuples through query modules. These modules are pipelined, non-blocking versions of standard relational operators, such as selections, projections, aggregation, joins, etc. In addition, telegraph uses a special type of module known as a State Module (SteM).

A SteM is a temporary repository of tuples that span the same set of tables. It corresponds to a half-join, and encapsulates a dictionary data structure over tuples from a table, and handle *build* (insert) and *probe* (lookup) request on that dictionary [2]. It provides a shared data structure for materializing and probing the data accessed from a given table, regardless of the number of access methods or join algorithms involving that table
In addition, by splitting joins into SteMs it allows pipelined computation and sharing of states between joins in different queries.

2.4 Fjords: Inter-Module Communication

Fjord [1, 6] is an inter-module communication API that binds the various modules together to form a query plan. The key advantage of Fjords is that it provides supporting for integrating streaming data, which is pushed into the system with disk-based data, which is pulled by traditional operators. It is put between the producers and the consumers in a query plan, and allows pairs of modules to be connected by various types of queues. Non-blocking enqueue and dequeue for push-queue make it possible to return the control to the consumer when the queue is empty. This is very important for data stream query processing.

3 Initial CQ Approaches

3.1 CACQ [1, 5]

Continuously Adaptive Continuous Query (CACQ) is the first continuous query engine to exploit the adaptive query processing framework of Telegraph. It provides the significant performance benefits to evaluate continuous queries, not only because of its adaptivity, but also because of the aggressive sharing of work and space among the multiple queries.

In CACQ prototype, first, the eddy operator provides continuous adaptivity to the changes in query workload, data delivery rates, and the overall system performance. Second, the path that each tuple takes through the operators – its lineage – is explicitly encoded in the tuple; this allows operators from many queries to be applied to a single tuple. Third, the using of grouped filters makes it possible to apply multiple selections over a single tuple. Finally, the joins are split into unary operators, SteMs, which allow pipelined join computation and sharing of state between joins in different queries.

3.2 PSoup [1, 4]

PSoup is built on the ideas developed in CACQ. It extends CACQ in two main ways: (1) it allows queries to access historical data and (2) it adds support to disconnected operations, that is, the client registers a query specification with the system, and he/she can revoke the query intermittently to retrieve the current results.

PSoup treats data and queries symmetrically. It views the execution of a stream of queries over a stream of data as a join of the two streams (see **Figure 2**). When a new query comes in, its SELECT-FROM-WHERE clause is extracted and inserted into the Query SteM, then applied to the old data stored in Data SteM. This can explain why PSoup can apply queries to historical data. On the other hand, when the new data come in, they are inserted into the Data SteM, and applied to all the queries in the Query SteM. PSoup continually runs the data/query join, and materializes the results in a special Results Structure. This is the key to support the disconnected operations in PSoup.

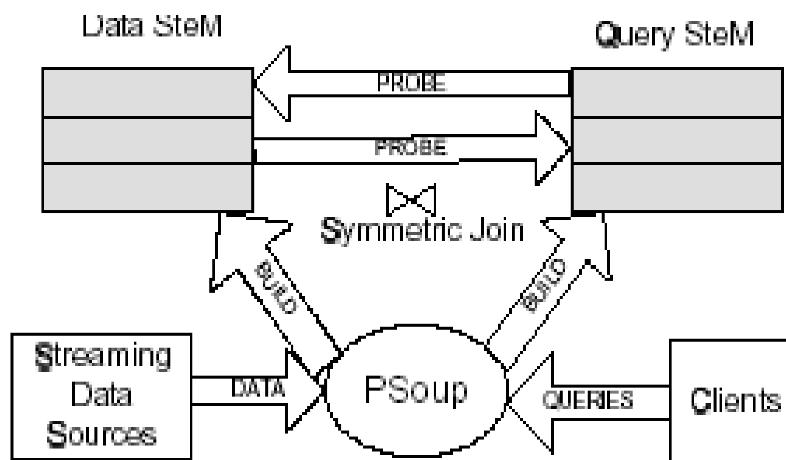

Figure 2: PSoup

4 TelegraphCQ

CACQ and PSoup have a number of limitations as we stated in the introduction session. The new generation of Telegraph system, TelegraphCQ, addresses these limitations, and focused on [1]:

- Scheduling and resource management for groups of queries
- Support for out-of-core data
- Variable adaptivity
- Dynamic QoS support
- Parallel cluster-based processing and distribution

Let's first take a look at the window semantics in TelegraphCQ

4.1 Window Semantics in TelegraphCQ

Since the length of data stream is unbounded, blocking operations such as joins, and aggregations, can only run over finite windows on the data stream. Two popular windowing schemes in the context of stream query processing are landmark and sliding window. TelegraphCQ support much more general windows than landmark and sliding windows. This is done using a for-loop construct to declare the sequence of windows over which the user desires the answer to be queried. The syntax of the for-loop is as follows:

```
For (t=initial_value; continue_condition(t); change(t))
{
  WindowsIs(Stream_A, left_end(t), right_end(t));
  WindowsIs(Stream_B, left_end(t), right_end(t));
  WindowsIs(Stream_C, left_end(t), right_end(t));
  …
}
```

There is a WindowIs statement for each stream in the query in the for-loop. Using this window scheme, we can construct snapshot queries, landmark queries, sliding queries and temporal Band-Join.

4.2 TelegraphCQ Design Overview

PostgreSQL open source database system is used as a starting point for the TelegraphCQ system. The architecture of TelegraphCQ is shown in **Figure 3** [8]**.**

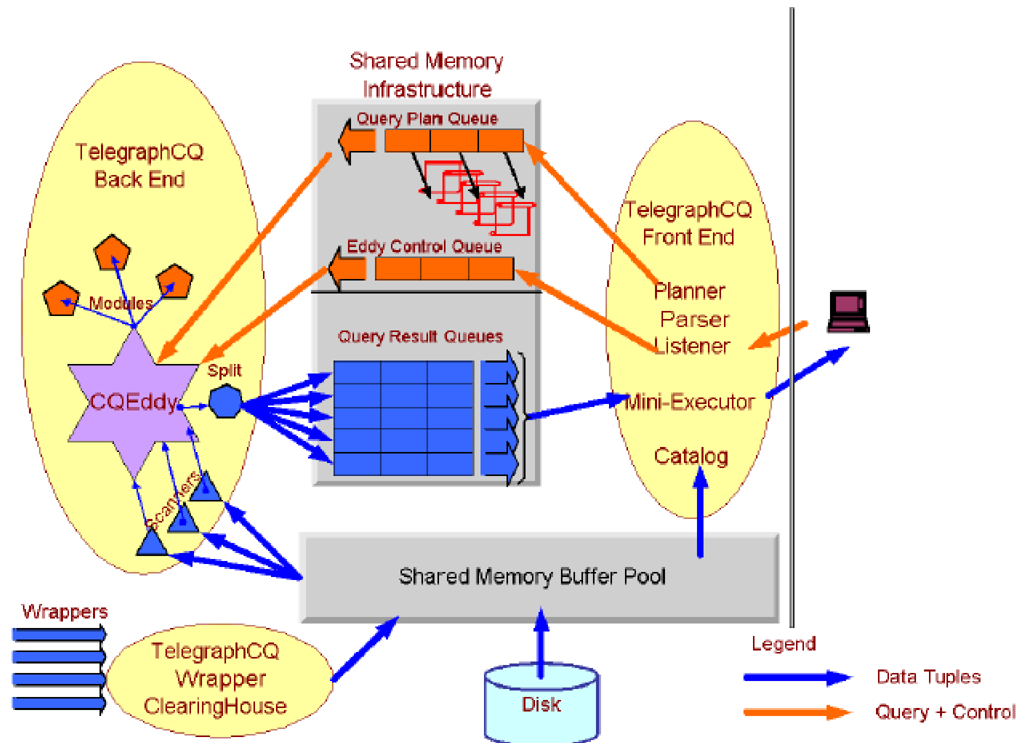

Figure 3: TelegraphCQ Architecture

The figure shows (as ovals) the three processes that comprise the TelegraphCQ server. These processes communicate using a shared memory infrastructure. TelegraphCQ Front End (FE) includes planner, parser, listener, and mini-executor. The actual query processing happens in the TelegraphCQ Back End (BE), which consists of the multi-threaded TelegraphCQ Executor. And the TelegraphCQ Wrapper ClearingHouse is used to host the data ingress operators, which make fresh tuples available for processing.

While the client sets up a connection to the TelegraphCQ system, and a FE process is forked. Then, the listener in FE process accepts queries from the client, and chooses where to execute them. DDL statement and queries over static tables are executed in the FE process itself. Continuous queries are pre-planned and sent via the Query Plan queue in shared memory to the BE process. The BE executor continually dequeues fresh queries and dynamically folds them into the current running query. Query results are in turn placed in the client-specific Query Result queues.

The data sources of TelegraphCQ system can be push sources, as found in traditional federated database systems, and pull sources, where Wrapper or data source itself initiate the connection to the TelegraphCQ system. In either case, the Wrapper process is responsible for “pulling” and “pushing-client” sources.

5 Conclusion and Future Work

This report describes TelegraphCQ system, a streaming data management system that processes continuous queries. It emphasizes the adaptivity and sharing with the query processing. There are a number of open questions that have arisen. Some of them are [1, 8]:

- **Adapting adaptivity**: Per-tuple and per-operator routing decision could consume significant portion of overall execution time. Is it possible that the system can dynamically adjust the degree of adaptivity?
- **Storage system:** How does the storage manager stream remote data into the disk and to the executor through the buffer pool? And what is the underlying file management system?
- **Egress modules:** It is desirable to provide the egress modules to manage and deliver the query results in order to handle the different modalities of client interactions.
- **Query grouping and sharing:** The TelegraphCQ executor partitions queries into execution objects (EO). The queries in the same EO tend to have a high degree of overlap. But how do we determine how much overlap is required to group a new query into an existing execution object?

References

[1] Chandrasekaran, S., Cooper, O., and Deshpande, A., et al., TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In CIDR, 2003

[2] Raman, V., Deshpande, A., and Hellerstein, J.. Using State Modules for Adaptive Query Processing. In ICDE (2003)

[3] Babcock, B., et al., Models and Issues in Data Stream Systems. In PODS (2000)
[4] Chandrasekaran, S. and Franklin, M., Streaming Queries over Streaming Data. In VLDB (2002)
[5] Madden, S., and Shah, M., and Hellerstein, J., and Raman, V., Continuously Adaptive Continuous Queries over Streams. In SIGMOD (2002)
[6] Madden, S., and Franklin, M., Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In ICDE (2002)
[7] Raman, V., Raman, B., and Hellerstein, J., Online Dynamic Reordering for Interactive Data Processing. In VLDB (1999)
[8] TelegraphCQ Getting Started Guide
http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/gettingstarted.html