TelegraphCQ: An Architectural Status Report

Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande
Michael J. Franklin, Joseph M. Hellerstein, Wei Hong[†], Samuel R. Madden
Fred Reiss, Mehul A. Shah

University of California, Berkeley, CA 94720
[†]Intel Research Laboratory, Berkeley, CA 94704
sailesh@cs.berkeley.edu

Abstract

We are building TelegraphCQ, a system to process continuous queries over data streams. Although we had implemented some parts of this technology in earlier Java-based prototypes, our experiences were not positive. As a result, we decided to use PostgreSQL, an open source RDBMS as a starting point for our new implementation. In March 2003, we completed an alpha milestone of TelegraphCQ. In this paper, we report on the development status of our project, with a focus on architectural issues. Specifically, we describe our experiences extending a traditional DBMS towards managing data streams, and an overview of the current early-access release of the system.

1 Introduction

Streaming data is now a topic of intense interest in the data management research community. This has been driven by a new generation of computing infrastructure, such as sensor networks, that has emerged because of pervasive devices. At Berkeley, we are building TelegraphCQ, a system for continuous dataflow processing. TelegraphCQ aims at handling large streams of continuous queries over high-volume highly variable data streams. In this paper, we focus on the architectural aspects of TelegraphCQ in its current release; please see [3] for the details of the novel query execution techniques that underlie the system.

As part of our earlier work [2, 6, 7, 4] in the Telegraph project, we built a system for adaptive dataflow processing in Java [1]. Although our research on adaptivity and sharing showed significant benefits, our experience in using Java in a systems development project was not positive [10]. After considering a few alternatives, we decided to use the PostgreSQL open source database system as a starting point for our new implementation. A continuous query system like TelegraphCQ is quite different from a traditional query processor. We found, however, that a significant amount of PostgreSQL code was easily reusable. We also found the extensibility features of PostgreSQL very useful, particularly the ability to load user-defined code and the rich data types such as intervals.

Challenges: As discussed in [3], sharing and adaptivity are our main techniques in implementing a continuous query system. Doing this in the codebase of a conventional database posed a number of challenges:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- **Adaptivity:** While the TelegraphCQ query executor uses a lot of existing PostgreSQL functionality, the actual interaction between query processor operators is significantly different. We rely on our prior work on adaptive query processing with operators such as *eddies* for tuple routing and operator scheduling, as well as *fjords* for inter-operator communication.

- **Shared continuous queries:** TelegraphCQ aims at amortizing query-processing costs by *sharing* the execution [6] of multiple long-running queries. This requirement resulted in a large-scale change to the conventional process-per-connection model of PostgreSQL.

- **Data ingress operations:** Traditional federated database systems fetch data from remote data sources using an operator in a query plan. Typically, such an operator uses user-defined (often "in the factory") *wrapper* functions that fetch data across the network. For a shared query processing system however, it is vital that all operators are non-blocking, and it is not possible to guarantee this with traditional wrappers.

Status: At the time of writing, we have just completed an alpha milestone – an early access release of TelegraphCQ. In its current state, TelegraphCQ supports a DDL statement, `CREATE STREAM`, that can be used to create archived or unarchived streams. Sources can register themselves with TelegraphCQ and push data to specific streams. This data is used to produce tuples using stream- and source- specific *wrapper* functions registered in the system. In addition, continuous queries can be registered that work over these streams. The queries can involve individual streams, joins over multiple streams and joins over streams and tables. The DML for these queries includes primitive sliding window syntax that supports grouped aggregate operations over these sliding windows. Please visit `http://telegraph.cs.berkeley.edu` for more information.

The rest of this paper is organized as follows. We start with a description of how a user interacts with TelegraphCQ in Section 2. In Section 3 we present an overview of TelegraphCQ. Next, in Section 4 we show how user-defined wrappers can be created in the system to interface with external data sources. We conclude with remarks on future work in Section 5.

2 Using TelegraphCQ

In this section we describe how users and applications can interact with TelegraphCQ. In addition to the full features of PostgreSQL, interactions with TelegraphCQ involve the following:

- Creating archived and unarchived streams.
- Creating sources that stream data to TelegraphCQ.
- Creating user-defined wrappers.
- Issuing continuous queries.

The streams that are created in the system identify objects that may be queried and data that may be processed, and possibly archived. The data sources are independent programs that continually send data to a specific named stream in TelegraphCQ. For each stream, there is a user-defined wrapper that is registered with TelegraphCQ and is designed to understand the data sent by the source. Finally, users and applications connect to TelegraphCQ so that they can issue continuous queries over these streams.

Creating streams: In TelegraphCQ, streams can be created and dropped using the new DDL statements, `CREATE STREAM` and `DROP STREAM` respectively. These statements are similar to those used to create and drop tables. A requirement in defining a stream is that there *must* exist exactly one column of a time type which serves as a timestamp for the stream. This column is specified with a new `TIMESTAMPCOLUMN` constraint. For example, an archived stream of readings of average speeds measured by traffic sensor stations in freeways may be defined as:

```
CREATE STREAM measurements (tcqtime TIMESTAMP TIMESTAMPCOLUMN,
                            stationid INTEGER,
                            speed REAL) TYPE ARCHIVED
```

Creating sources: TelegraphCQ expects that a data source will initiate a network connection with it and advertise the name of the stream for which it undertakes to provide data. More than one source can simultaneously push data to the same stream. The stream name is identified in the first few bytes sent to TelegraphCQ after establishing the network connection. There are no other restrictions on the format of the stream data, as all subsequent network operations on the connection are the responsibility of the user-defined wrapper functions associated with the stream.

Creating wrappers: Each wrapper consists of three user-defined functions (`init`, `next` and `done`) that are called by TelegraphCQ to process data from external sources. This is described in more detail in Section 4. The functions are registered in TelegraphCQ using the standard PostgreSQL `CREATE FUNCTION` statement. For example the `init` function of the `measurements` wrapper should be named `measurements_init` and can be declared using the following DDL.

```
CREATE FUNCTION measurements_init(INTEGER) RETURNS BOOLEAN
AS 'libmeasurements.so',measurements_init' LANGUAGE 'C';
```

Continuous queries: Once streams have been created in the system, users can issue long-running continuous queries over them. When data streams in, the results get sent to the appropriate issuers. A query is identified as being continuous if it operates over one or more streams. Such queries do not end until cancelled by the user. The queries can be simple SQL queries (without subqueries) with an optional *window* clause. The interval that is specified as a string in the window clause may be anything that PostgreSQL can convert automatically into an *interval* type. The window clause serves to restrict the amount of data that participates in the query over time. This is particularly important for join and aggregate queries since streams are unbounded. An example query that involves a stream and a table is:

```
SELECT   ms.stationid, s.name, s.highway, s.mile, AVG(ms.speed)
FROM     measurements ms, stations s
WHERE    ms.stationid = s.stationid
GROUP BY ms.stationid, s.name, s.highway, s.mile
WINDOW   ms ['10 minutes']
```

This query joins `measurements`, a stream of sensor readings with `stations`, a normal relation. The query continually reports average speeds recorded by each station in a sliding 10 minute window.

These queries can be submitted to TelegraphCQ using `psql`, the interactive client. Other programmatic interfaces such ODBC and JDBC can also be used, but the applications using those must submit continuous queries by declaring named cursors. Otherwise the PostgreSQL call-level interface buffers the results of a query and blocks until all results have been received.

3 Overview of the system

In this section we present a short overview of the TelegraphCQ system, and show how we leverage the functionality of PostgreSQL.

Figure 1 shows the basic process structure of PostgreSQL. PostgreSQL uses a process-per-connection model. Data structures shared by multiple processes, such as the buffer pool, latches, etc. are located in shared memory. A *Postmaster* process forks new server processes in response to new client connections. Amongst the different components in each server process, the *Listener* is responsible for accepting requests on a connection and returning data to the client. When a new query arrives it is parsed, optimized, and compiled into an access plan that is

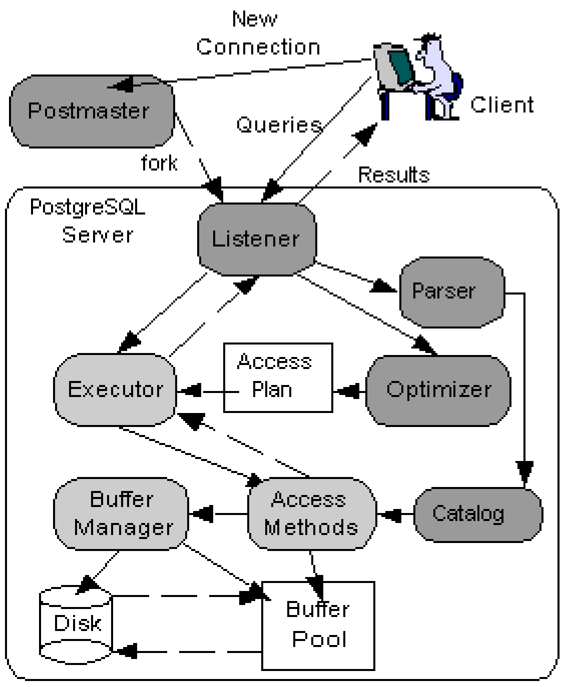

Figure 1: PostgreSQL Architecture

then processed by the query *Executor* component. The components that have only been changed minimally in TelegraphCQ are shaded in dark gray in Figure 1. These include: Postmaster, Listener, System Catalog, Query Parser and Optimizer[1]. Components shown in light gray (the Executor, Buffer Manager and Access Methods) are being leveraged with significant changes. In addition, by adopting the front-end components of PostgreSQL we also get access to important client-side call-level interface implementations (not shown in the figure) such as ODBC and JDBC.

Our chief challenge in using PostgreSQL is supporting the TelegraphCQ features it was not designed for: streaming data, continuous queries, shared processing and adaptivity. The biggest impact of our changes is to PostgreSQL's process-per-connection model. In TelegraphCQ we have a dedicated process, the TelegraphCQ Back End (BE), for executing shared long-running continuous queries. This is in addition to the per-connection TelegraphCQ Front End (FE) process that fields queries from, and returns results to, the client. The FE process also serves non-continuous queries and DDL statements. TelegraphCQ also has a dedicated Wrapper ClearingHouse process that ensures that data ingress operations do not impede the progress of query execution.

Figure 2 depicts a bird's eye view of TelegraphCQ. The figure shows (as ovals) the three processes that comprise the TelegraphCQ server. These processes communicate using a shared memory infrastructure. The TelegraphCQ Front End contains the Listener, Catalog, Parser, Planner and "mini-Executor". The actual query processing takes place in a separate process called the TelegraphCQ Back End. Finally the TelegraphCQ Wrapper ClearingHouse is used to host the data ingress operators which make fresh tuples available for processing, archiving them if required.

As in PostgreSQL, the Postmaster listens on a well-known port and forks a Front End (FE) process for each fresh connection it receives. The listener accepts commands from a client and based on the command, chooses where to execute it. DDL statements and queries over tables are executed in the FE process itself. Continuous

[1]Currently we use the PostgreSQL optimizer to create the plan data structures; we are planning to bypass this step for continuous queries in future.

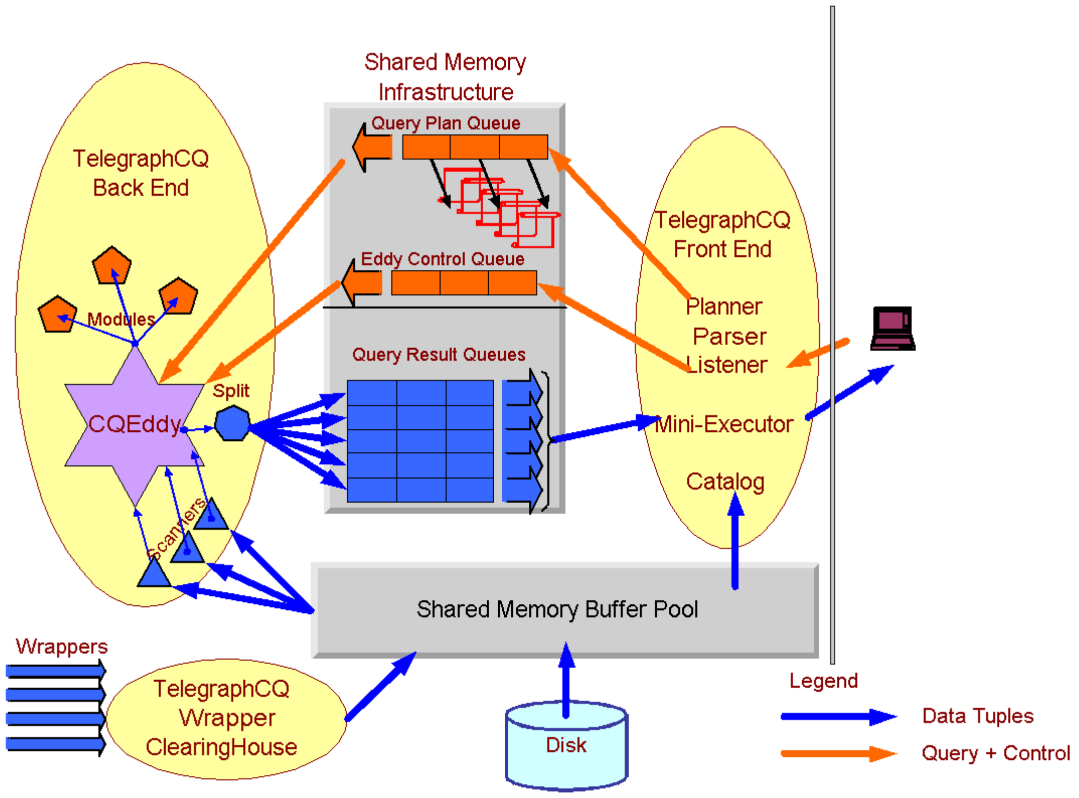

Figure 2: TelegraphCQ Architecture

queries, those that involve streams as well as streams and tables, are "pre-planned" and sent via the Query Plan queue (in shared memory) to the Back End (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. Once the FE has handed a query off to the BE, it produces an FE-local minimal query plan that the mini-executor runs to continually dequeue results from its Query Result queue and send back to the connected client.

Since a continuous query never ends, clients should submit such queries as part of named cursors. We have added a continuous query mode to `psql`, the standard PostgreSQL interactive client. In this mode, `SELECT` statements are automatically converted into named cursors which can then be iterated over to continually fetch results.

4 Wrappers

In this section we describe how data from external sources can be streamed into TelegraphCQ. We plan to support the following kinds of sources:

- Pull sources, as found in "traditional" federated database systems.
- Push sources, where connections can be initiated either by TelegraphCQ (TelegraphCQ-Push) or by the data source itself (Source-Push).

With pull and TelegraphCQ-push sources, TelegraphCQ connects to the source. In contrast, Source-Push sources connect to a well-known port of TelegraphCQ, advertise the stream they are "supplying" and commence data delivery. In our current release we only support Source-Push sources.

As with traditional federated systems, we use user-defined wrapper functions to massage data into a TelegraphCQ format (essentially that of PostgreSQL). In our current implementation, there is a one to one correspondence between a wrapper and a stream, although eventually we will ease this restriction. In TelegraphCQ, however, the system handles the network interface, and the user-defined wrapper limits itself to reading data off a network socket.

Since it is expensive to create a fresh process for each data source, we have a single process, the TelegraphCQ Wrapper ClearingHouse (WCH) that manages data acquisition. The WCH is responsible for the following operations:

- Accepting connections from external sources, and obtaining the stream name they advertise.
- Loading the appropriate user defined wrapper functions.
- Calling these wrapper function when data is available on a connection.
- Processing result tuples returned by a wrapper according to the stream type.
- Cleaning up when a source ends its interaction with TelegraphCQ.

The WCH accepts connections from new data sources, and based on the stream name the source advertises, loads and executes the appropriate user-defined wrapper code. Since the single WCH process handles multiple sources, each network socket, and hence each wrapper function, must be non-blocking.

The wrapper code is called either in response to a data-ready indication on the wrapper's network socket, or if there is any data as yet unprocessed by the wrapper. The latter condition can arise, as the wrapper only returns one tuple at a time, in a manner akin to the classical iterator [5] model. Each call to a wrapper may not even result in a fresh tuple, as the data required to form a tuple might not be available all at once (e.g. a large XML document that streams in across many network packets). So user defined wrappers should read data from the network, buffering it if necessary.

Once the wrapper has enough data to form a tuple, it converts it into the PostgreSQL data types expected by the target stream. Such conversion often relies on calls to PostgreSQL's own data conversion and construction functions, and so the wrapper writers are expected to have at least minimal familiarity with PostgreSQL datatypes and data manipulation functions. In particular, the wrapper must return data to the system as an array of PostgreSQL `Datum` items that can be used to create a data tuple in the PostgreSQL format. In future the WCH will be extended to provide both mapping and data conversion services to wrappers to make their implementation easier, perhaps based on the Potter's Wheel [8] system.

A wrapper consists of three separate user defined functions: `init` for initialization, `next` to produce new records, and `done` for cleanup. They are loaded from a shared library using the PostgreSQL function manager. Each of these functions take a single argument that is treated as a pointer to a `WrapperState` structure, and return a boolean value to indicate success or fatal error. The `next` function returns processed tuples through a field in the `WrapperState`. Most wrapper functions will need to maintain their own additional state. This is done through another private field of the `WrapperState`. Wrapper functions will also typically allocate and free memory respectively in their `init` and `done` functions using the PostgreSQL region-based memory management infrastructure.

The WCH and wrappers use other fields to communicate state information and return values. For instance the wrapper needs to indicate to the WCH that it has as yet unprocessed data and must be called even if its network connection remains idle.

The final responsibility of the WCH is to make freshly created tuples available to the rest of the system. This is done by placing the tuples in buffers within the standard PostgreSQL buffer pool (analogous to the way buffers would be allocated for insertions to a traditional table). In the case of archived streams these are flushed to disk as part of the normal buffer manager operation. Subsequently, normal scan operators in the TelegraphCQ executor fetch these archived tuples and process them. Unarchived streams can be handled in a similar way, the difference being that dirty buffers are not written out to stable storage. In the current release, however, we use a separate shared memory queue for unarchived streams.

5 Conclusion and future work

This paper describes the current status of TelegraphCQ, a streaming data management system that processes continuous queries. This corresponds to an early access release of TelegraphCQ that we completed in March 2003. After our experience building the first version of Telegraph in Java, we chose to extend PostgreSQL, a proven open source database system that is implemented in the C programming language and runs on a wide variety of platforms. Starting off with PostgreSQL has helped us quickly ramp up the development of TelegraphCQ. We have benefited from very many useful features that already exist in PostgreSQL. The main message we have at this point in the TelegraphCQ project is that it is feasible to build a highly adaptive streaming dataflow system with shared query processing that uses building blocks from a conventional relational data base. An important next step is to analyze the performance of our system.

TelegraphCQ is very much a work in progress, and we are actively working on adding more features to the system. As part of this effort, we are addressing a number of open questions that have arisen. Some of these are:

- **Adaptive adaptivity:** Adaptivity clearly has a benefit in the hostile environments that streaming dataflow systems must operate in. These benefits, however, probably come with a concomitant cost. What are these costs, and is it possible for our system to adapt the *amount* of adaptivity dynamically ?

- **Storage system:** In a streaming system, data continually floods in, in a fashion unrelated to the queries in the system and how they are being processed. When data is no longer *orchestrated* through the system, can we revisit the role of the storage manager ?

- **Interactions between continuous and historical queries:** In TelegraphCQ we currently only support continuous queries over data streams. In future we plan to support historical queries by running them separately, say in the TCQ Front End process. However, it is not clear how to process queries that are combinations of a stream's newly arriving data and historical data (e.g. combine the last hour's traffic sensor data with the average data for this hour over the last year).

- **Scalability and Availability:** The Flux operator [9] is designed to enhance TelegraphCQ with partitioned parallelism, adaptive load-balancing, high availability and fault tolerance. The load-balancing work (Lux) has been validated in simulations from the prior version of Telegraph; the full-featured Flux implementation in TelegraphCQ is underway.

- **Signal/noise streams:** To handle overload situations, a number of research groups are considering schemes that downsample (drop) or synopsize (lossily compress) items in a stream. In this vein, we are exploring a new algebra that combines both the fully-propagated items and the synopsized components. Our intent is for this combined approach to provide a characterization of both the signal and noise produced when downsampling or synopsizing the stream. We are also exploring opportunities for joint optimization of the fully-propagated query plan and the synopsis plan.

6 Acknowledgements

We are grateful for comments and suggestions from other members of the database systems research group at Berkeley, as well as researchers who are building similar systems in other groups. This work has been supported by National Science Foundation awards IIS-0208588, IIS-0205647, IIS-0086057 and SI0122599 as well as funding from Microsoft, IBM, Intel and the UC MICRO program.

References

[1] ARNOLD, K., GOSLING, J., AND HOLMES, D. *The Java Programming Language*, 3rd ed. Addison-Wesley, 2000.

[2] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *Proceedings of ACM SIGMOD Conference* (2000), pp. 261–272.

[3] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of Conference on Innovative Data Systems Research* (2003).

[4] CHANDRASEKARAN, S., AND FRANKLIN, M. J. Streaming queries over streaming data. In *Proceedings of VLDB Conference* (2002).

[5] GRAEFE, G. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2 (June 1993), 73–170.

[6] MADDEN, S. R., SHAH, M. A., HELLERSTEIN, J. M., AND RAMAN, V. Continuously adaptive continuous queries over streams. In *Proceedings of ACM SIGMOD Conference* (2002).

[7] RAMAN, V., DESHPANDE, A., AND HELLERSTEIN, J. M. Using state modules for adaptive query processing. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[8] RAMAN, V., AND HELLERSTEIN, J. M. Potter's wheel: An interactive data cleaning system. In *Proceedings of VLDB Conference* (2001), pp. 381–390.

[9] SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[10] SHAH, M. A., MADDEN, S. R., FRANKLIN, M. J., AND HELLERSTEIN, J. M. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record 30*, 4 (December 2001), 103–114.