TelegraphCQ: A Data Stream Management System

Neil Conway (neilc@samurai.com)
Introduction

- What is data stream management?
- A query language for streams
- TelegraphCQ architecture
- Query execution
Most traditional database systems are optimized for one-time queries on mostly-static data.

- Long-lived data, short-lived queries

This is a poor fit for applications that want to manipulate real-time unbounded *streams* of data.

- Long-lived queries, short-lived data

Examples: real-time analysis of financial data (stock trades, fraud detection), sensor network data, network traffic analysis (clickstreams, intrusion-detection), . . .
Possible solution: insert incoming data into a DBMS, perform analysis, and then periodically trim/archive the data.

Inefficient: unnecessarily hits disk, and recomputes the analysis from scratch each time.

Better would be to incrementally update the result set of a continuous query to reflect latest stream tuples.

Popular alternative: “roll your own” from scratch.

Labour intensive.

Goal: simplify streaming applications by building a generic data stream management system (DSMS).
TelegraphCQ

- There has been considerable interest in streams among academia
- One such project is TelegraphCQ from UC Berkeley
  - Open-source DSMS prototype, based on the PostgreSQL database system
- Several other academic streaming projects: STREAM (Stanford), Aurora (MIT), Nile (Purdue), Medusa (Brown), ... 
- Several TelegraphCQ folks have started Amalgamated Insight Inc.
- There are several other stream-related startups, notably StreamBase (Stonebraker)
What is a stream?

- A stream is an infinite bag of \( \langle \text{timestamp}, \text{tuple} \rangle \) pairs.
- Stream tuples can be externally timestamped, or timestamped by the system.
- A DSMS manages both streams and (typically) conventional database objects.
- “Push” stream: stream content is supplied by client applications that send it directly to the DSMS.
  - For example, by connecting via TCP.
- In “pull” streams, the DSMS fetches content from a remote source and converts it into a stream of tuples.
- A derived stream is a stream defined by a query on other database objects.
Queries on streams

- A DBMS typically handles ad hoc, one-time queries
  - Client submits query, DBMS evaluates it and returns result set

- In streaming applications, we are more interested in continuous queries
  - Long-running (months or years not uncommon)
  - Continuous: result set changes over time
  - Typically represents a condition of interest ("do $x$ when condition $y$ or $z$ is satisfied"), or a running statistical summary of a stream ("show me the $k$ most active stock symbols in the last $t$ minutes, computed every $n$ seconds")
Query Language

- A declarative query language is a Good Thing
- STREAM and TelegraphCQ implement variants of CQL, the Continuous Query Language
- CQL is a straightforward extension to SQL for manipulating streams
- Doesn’t specify some important technical details
- Currently under development: standardized streaming query language, “StreamSQL”
CQL Basics

- Pragmatism: we have a perfectly good relational query language, so reuse that
  - Query optimization and execution for traditional relational query languages is also very well understood

- Two basic kinds of things: streams and relations
- SQL defines relation $\rightarrow$ relation operators
- CQL defines operators for stream $\rightarrow$ relation and relation $\rightarrow$ stream
Stream → Relation

- Streams are unbounded. To deal with a finite portion of the stream, we apply a *window clause* to it to produce a time-varying relation.

- Time-based window: the tuples that appear in a specified time interval in the stream
  
  \[
  \text{RANGE '5 minutes' SLIDE '30 seconds'}
  \]

- Tuple-based window: most recent \(n\) tuples in the stream
  
  \[
  \text{ROWS 10 SLIDE '60 seconds'}
  \]

- Partitioned window: given a set of attributes, groups stream tuples on those attributes, then computes the union of a window clause applied to all the groups
  
  \[
  \text{PARTITION BY stock.symbol ROWS 5}
  \]
Relation $\rightarrow$ Stream

3 types of $R \rightarrow S$ operators:

1. The $IStream$ of $R$ contains a tuple $s$ at time $t$ when $s$ is in $R_t - R_{t-1}$

2. The $DStream$ of $R$ contains a tuple $s$ at time $t$ when $s$ is in $R_{t-1} - R_t$

3. The $RStream$ of $R$ contains a tuple $s$ at time $t$ when $s$ is in $R_t$
Joins

- Stream-relation joins are common in practice
  - For each new stream tuple, do a table (index) lookup
  - 1 stream, $n$ relations
  - Divides plan into streaming and non-streaming components

- Stream-stream joins can be useful
  - Probably requires compatible window clauses
CQL Example

(Source: Stanford CQL Query Repository)

Network traffic analysis: “Every 5 minutes, sum the number of bytes and number of packets devoted to HTTP traffic for each distinct IP address.”

```
SELECT RSTREAM(src_ip, SUM(len), COUNT(*))
FROM packets [RANGE '5 Minutes'
             SLIDE '5 Minute']
WHERE dest_port = '80'
GROUP BY src_ip
```
Online auction fraud detection: “Every 90 seconds, compute the highest bid made in the last 10 minutes.”

```sql
SELECT RSTREAM(item_id, bid_price)
FROM bid [RANGE '10 Minutes'
    SLIDE '90 Seconds']
WHERE bid_price =
    (SELECT MAX(bid_price)
    FROM bid [RANGE '10 Minutes'
        SLIDE '90 Seconds'])
```
**TelegraphCQ**

- **Adaptivity**: continuous queries might run for months. Static planning decisions will become invalid.
  - Therefore, don’t use a static query plan.

- **Sharing**: many typical systems will execute hundreds of continuous queries at a time. Sharing the work required to evaluate these queries is necessary.
  - Happily, long-running continuous queries are easier to share.
  - Concurrency is also simplified with streams.

- Need to allow continuous queries to be easily added and removed from a running system.

- Try to avoid hitting disk.
TelegraphCQ Architecture

- Context: PostgreSQL uses a fairly traditional Unix daemon architecture
  - A single persistent parent process, the *postmaster*
  - The postmaster forks a new child process called a *backend* to handle each new client connection
  - A SysV IPC shared memory region is used to communicate between backends
    - It contains various caches, notably the shared buffer pool
  - Some additional server processes: autovacuum daemon, background writer, checkpoint process
Process Architecture

- New TelegraphCQ processes:
  - *Wrapper Clearing House*: manages stream I/O (push/pull) and format conversion
  - *TCQ Backend*: single process that executes all the streaming queries as part of a single query plan

- Communication between processes via shared memory queues

- Client connects to normal Postgres backend; continuous queries are planned by the backend, then sent via shared memory to the TCQ backend

- Results returned via another shared memory queue
Global Query Plan

- TCQ backend is responsible for evaluating all the continuous queries in the system.
- Construct a single query plan containing all the operators in all the queries.
  - Continuous queries from a PostgreSQL backend folded into the shared query plan.
  - Commonalities between queries can be exploited by using a single shared operator to implement parts of more than one query.
- Determining how to walk the graph of operators for a given stream input tuple is called *tuple routing*.
Tuple Routing

- The optimal path might change over time: operator cost, operator selectivity, stream arrival rates, ... are all variable.

- Therefore, don’t do any static planning: instead, per-tuple adaptive routing.

- A stream tuple includes “routing metadata”, describing the operators it has visited, the queries it is still visible to, and its signature (underlying base tuples).
  - We don’t materialize join tuples, for more routing flexibility.

- Once a tuple fails a predicate for a query, mark it as invisible to that query (but continue routing tuple!)
Tuple Routing, cont.

- Split joins into two halves (STeM): “build” and “probe”
- Decide which operator to send a tuple to next based on runtime statistics about operator costs / selectivies, plus the tuple routing metadata
- Current implementation is not parallel, but the design should parallelize well
Shared Evaluation

- This architecture naturally leads to implementing parts of multiple queries with a single operator.
- Sharing predicates is fairly easy for $\leq, <, >, \geq, =, \neq$.
- Joins can be shared by splitting them into StEMs.
- Aggregates can be shared pretty effectively.
  - Even aggregates with different predicates and window clauses can be shared.
- Two-phase aggregation.
Interesting Streaming Problems

- Graceful degradation under load
  - The rate of arrival of a given stream is often highly variable
  - Sometimes necessary to provision hardware for average load, not peak load
  - How to degrade gracefully?
  - Options: spill excess tuples to disk, summarize excess tuples (e.g. via histograms), or discard them

- High-availability and clustering
More Problems

- “Hybrid” queries (stream-table joins)
  - How does this change query optimization/execution, especially in the non-streaming portion of the query?
  - How do we avoid the downsides of static planning?
  - Sharing?

- Streams and transactions
  - When do rows in base tables become visible?
  - Transaction-like semantics for streaming queries?

- Historical queries, archived streams