An Introduction To Data Stream Query Processing

Neil Conway
<nconway@aminsight.com>

Amalgamated Insight, Inc.

May 24, 2007

Outline

Outline

What's wrong with database systems?

The Need For Data Stream Processing

What's wrong with database systems?

Nothing, but they aren't the right solution to every problem

The Need For Data Stream Processing

What's wrong with database systems?

Nothing, but they aren't the right solution
to every problem

What are some problems for which
a traditional DBMS is an awkward fit?

Financial Analysis

- Electronic trading is now commonplace
  - Trading volume continues to increase rapidly
- Algorithmic trading: detect advantageous market conditions, automatically execute trades
  - Latency is key
- Visualization
  - A hard problem in itself

Financial Analysis

- Electronic trading is now commonplace
  - Trading volume continues to increase rapidly
- Algorithmic trading: detect advantageous market conditions, automatically execute trades
  - Latency is key
- Visualization
  - A hard problem in itself

Typical Queries

- 5-minute rolling average, volume-waited average price (VWAP)
- Comparison between sector averages and portfolio averages over time
- Implement models provided by quantitive analysis

Network Monitoring

- Network volume continues to increase rapidly
- Custom solutions are possible, but roll-your-own is expensive
  - Ad-hoc queries would be nice
- Can we build generic infrastructure for these kinds of monitoring applications?

Sensor Networks

Pervasive Sensors

"As the cost of micro sensors continues to decline over the next decade, we could see a world in which everything of material significance gets sensor-tagged." – Mike Stonebraker

- Military applications: real-time command and control
- Healthcare
- Habitat monitoring
- Manufacturing

Other Examples

Real-Time Decision Support

Turnaround-time for traditional data warehouses is often too slow

- "Business Activity Monitoring" (BAM)

Other Examples

Real-Time Decision Support

Turnaround-time for traditional data warehouses is often too slow

- "Business Activity Monitoring" (BAM)

Fraud Detection

- Sophisticated, cross-channel fraud
- Real-time

Other Examples

Real-Time Decision Support

Turnaround-time for traditional data warehouses is often too slow

- "Business Activity Monitoring" (BAM)

Fraud Detection

- Sophisticated, cross-channel fraud
- Real-time

Online Gaming

- Detect malicious behavior
- Monitor quality of service

Data Stream Management Systems

Database Systems

Mostly static data, *ad-hoc* one-time queries

- Fire the queries at the data, return result sets
- "Store and query"
- Focus: concurrent reads & writes, efficient use of I/O, maximize transaction throughput, transactional consistency, historical analysis

Data Stream Management Systems

Database Systems

Mostly static data, *ad-hoc* one-time queries

- Fire the queries at the data, return result sets
- "Store and query"
- Focus: concurrent reads & writes, efficient use of I/O, maximize transaction throughput, transactional consistency, historical analysis

Data Stream Systems

Mostly transient data, continuous queries

- Fire the data at the queries, *incrementally* update result streams
- Data rates often exceed disk throughput

Complex Event Processing (CEP)

- Data stream processing emerged from the database community
  - Early 90's: "active databases" with triggers
- Complex Event Processing is another approach to the same problems
  - Different nomenclature and background
  - Often similar in practice

Outline

Data Streams

- A stream is an infinite sequence of $\langle tuple, timestamp \rangle$ pairs
  - Append-only
  - New type of database object
- The timestamp defines a total order over the tuples in a stream
  - In practice: require that stream tuples have a special `CQTIME` column
- Different approaches to building stream processing systems
  - This talk: relation-oriented DSMS. Specifically, TelegraphCQ, AmInsight, StreamBase, . . .

CREATE STREAM

- Exactly 1 column must have a CQTIME constraint
  - CQTIME can be system-generated or user-provided
- With user-provided timestamps, system must cope with out-of-order tuples
  - "Slack" specifies maximum out-of-orderness

Example Query

```
CREATE STREAM trades (
   symbol varchar(5),
   price  real,
   volume integer,
   tstamp timestamp CQTIME USER GENERATED SLACK '1 minute'
) TYPE UNARCHIVED;
```

Types of Streams

Raw Streams

Stream tuples are injected into the system by an external data source

- E.g. stock tickers, sensor data, network interface, ...
- Both push and pull models have been explored

Types of Streams

Raw Streams

Stream tuples are injected into the system by an external data source

- E.g. stock tickers, sensor data, network interface, ...
- Both push and pull models have been explored

Derived Streams

Defined by a query expression that yields a stream

Archived Streams

Allows historical and real-time stream content to be combined in a single database object

Language Design Philosophy

- Pragmatism: relational query languages are well-established
  - Relational query evaluation techniques are well-understood
  - Everyone knows SQL
- Therefore, add stream-oriented extensions to SQL
  - Pioneering work: CQL from Stanford STREAM project

Language Design Philosophy

- Pragmatism: relational query languages are well-established
  - Relational query evaluation techniques are well-understood
  - Everyone knows SQL
- Therefore, add stream-oriented extensions to SQL
  - Pioneering work: CQL from Stanford STREAM project

Kinds Of Operators

Relation $\rightarrow$ Relation: Plain Old SQL

Stream $\rightarrow$ Relation: Periodically produce a relation from a stream

Relation $\rightarrow$ Stream: Produce stream from changes to a relation

Note that $S \rightarrow S$ operators are not provided.

Continuous Queries

Fundamental Difference

The result of a continuous query is an unbounded stream, not a finite relation

Continuous Queries

Fundamental Difference

The result of a continuous query is an unbounded stream, not a finite relation

Typical Query

1. Split infinite stream into pieces via windows
   - $S \rightarrow R$
2. Compute analysis for the current window, comparison with prior windows or historical data
   - $R \rightarrow R$
3. Convert result of analysis into result stream
   - $R \rightarrow S$
   - Often implicit (use defaults)

Stream $\longrightarrow$ Relation Operators: Windows

- Streams are infinite: at any given time, examine a finite sub-set
- Apply window operator to stream to periodically produce visible sets of tuples

Stream → Relation Operators: Windows

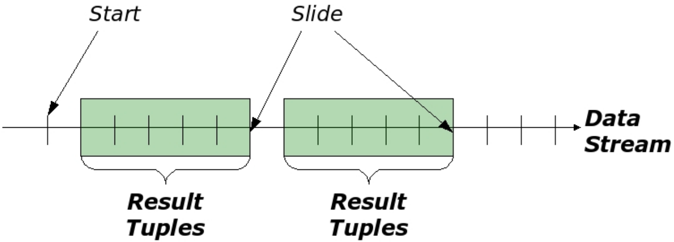
- Streams are infinite: at any given time, examine a finite sub-set
- Apply window operator to stream to periodically produce visible sets of tuples

Properties of Sliding Windows

Range: "Width" of the window. Units: rows or time.

Slide: How often to emit new visible sets. Units: rows or time.

Start: When to start emitting results.



Example Query

Description

Every second, return the total volume of trades in the previous second.

Query

```
SELECT   sum(volume) AS volume,
         advance_agg(qtime) AS windowtime
FROM     trades < VISIBLE '1 second' ADVANCE '1 second' >
```

Another Example

Description

Every 5 seconds, return the volume-adjusted price of MSFT for the last 1 minute of trades.

Query

```
SELECT   sum(price * volume) / sum(volume) AS vwap,
         sum(volume) AS volume,
         advance_agg(qtime) AS windowtime
FROM     trades < VISIBLE '1 minute' ADVANCE '5 seconds' >
WHERE    symbol = 'MSFT'
```

More About Windows

Aggregation

Useful aggregate: *advance_agg*(*CQTIME*)

- Timestamp that marks the end of the current window
- Similar aggregates for "beginning of window", "middle of window" might also be useful

More About Windows

Aggregation

Useful aggregate: *advance_agg*(*CQTIME*)

- Timestamp that marks the end of the current window
- Similar aggregates for "beginning of window", "middle of window" might also be useful

Other Window Types

Landmark: Fixed left edge, "elastic" right edge. Periodically reset. ("All stock trades after 9AM today.")

Partitioned: Divide stream into sub-streams based on partitioning key(s), then apply another $S \rightarrow R$ operator to the sub-streams.

Relation → Stream Operators

Types of Operators

- `ISTREAM`: the tuples added to a relation
- `RSTREAM`: all the tuples in a relation
- `DSTREAM`: the tuples removed from relation

Relation → Stream Operators

Types of Operators

`ISTREAM`: the tuples added to a relation

`RSTREAM`: all the tuples in a relation

`DSTREAM`: the tuples removed from relation

Defaults

- `ISTREAM` for queries without aggregation/grouping
- `RSTREAM` for queries with aggregation/grouping
- `DSTREAM` is rarely useful

Mixed Joins

Common Requirement

Compare stream tuples with historical data

- System must provide both tables and streams!
- Elegantly modeled as a join between a table and a stream

Mixed Joins

Common Requirement

Compare stream tuples with historical data

- System must provide both tables and streams!
- Elegantly modeled as a join between a table and a stream

Implementation

- Stream is the right (outer) join operand; left (inner) operand is arbitrary Postgres subplan
  - For each stream tuple, join against non-continuous subplan

Mixed Join Example

Description

Every 3 seconds, compute the total value of high-volume trades made on stocks in the S & P 500 in the past 5 seconds.

Example Query

```
SELECT   T.symbol, sum(T.price * T.volume)
FROM     s_and_p_500 S,
         trades T < VISIBLE ‘5 sec’ ADVANCE ‘3 sec’ >
WHERE    T.symbol = S.symbol
     AND T.volume > 5000
GROUP BY T.symbol
```

Composing Streams

- The tuples in a stream can be viewed as a series of events
  - E.g. "The temperature in the room is 20°", 25°, 30°, ...
- The output of a continuous query is another series of events, typically higher-level or more complex
  - E.g. "The room is on fire."

Composing Streams

- The tuples in a stream can be viewed as a series of events
  - E.g. "The temperature in the room is 20°", 25°, 30°, ...
- The output of a continuous query is another series of events, typically higher-level or more complex
  - E.g. "The room is on fire."
- Therefore, streams can be composed in various ways:
  - Stream views
    - Macro semantics
  - Derived streams
  - Subqueries
  - Active tables

Derived Streams

- A derived stream is a database object defined by a persistent continuous query
- Unlike a stream view, always active
- Similar to a materialized view

Example Query

Description

Every 3 seconds, compute the "volume-weighted average price" (VWAP) for all stocks traded in the past 5 seconds.

Query

```
CREATE STREAM vwap (symbol varchar(5),
                    vwap   float,
                    vtime  timestamp cqtime) AS
(SELECT     symbol,
            sum(price * volume) / sum(volume),
            advance_agg(qtime)
 FROM       trades < VISIBLE '5 seconds' ADVANCE '3 seconds' >
 GROUP BY   symbol);
```

Subqueries

- One-time subqueries can be used in continuous queries, of course
- Continuous subqueries are planned and executed as independent queries
  - Essentially inline derived streams
- Require that subqueries yielding streams specify `CQTIME`
- Planned: `WITH`-clause subqueries

Active Tables

- An active table is a table with an associated continuous query
- Two modes of operation:

  Append: New stream tuples appended to table at each window
  Replace: At each new window, truncate previous table contents

Event Language

Example Query

```
SELECT      'Shoplifting!', D.loc, D.id
FROM        Store S C D PARTITION BY id
WHERE       S.loc = 'shelf' and C.loc = 'checkout'
            AND D.loc = 'door'
EVENT       AND (FOLLOWS(S, D, '1 hour'),
                 NOT PRECEDES(C, D, '1 hour'));
```

Outline

Basic Requirements

Adaptivity

Static query planning is undesirable for long-running queries

- Either replan or use adaptive planning

Basic Requirements

Adaptivity

Static query planning is undesirable for long-running queries

- Either replan or use adaptive planning

Shared Processing

Essential for good performance: 100s of queries not uncommon

- Long-lived queries make this more feasible

Basic Requirements

Adaptivity

Static query planning is undesirable for long-running queries

- Either replan or use adaptive planning

Shared Processing

Essential for good performance: 100s of queries not uncommon

- Long-lived queries make this more feasible

Graceful Overload Handling

Stream data rates are often highly variable

- Often too expensive to provision for maximal data rate
- Therefore, must handle overload gracefully

System Architecture

- Modified version of PostgreSQL
- One-time queries executed normally
- Continuous queries planned and executed by the `CqRuntime` process

System Architecture

- Modified version of PostgreSQL
- One-time queries executed normally
- Continuous queries planned and executed by the `CqRuntime` process
- Stream input: `COPY`, or submitted via TCP to `CqIngress` process
  - libevent-based, simple `COPY`-like protocol
- Stream output: cursors, active tables, `CqEgress` process
- Communication between processes done via shared memory queue infrastructure
  - Message passing done via SysV shmem and locks

Shared Runtime

- New continuous query is defined $\rightarrow$ shared runtime via shared memory
- Runtime plans the query, folds query into single shared query plan
  - Not a traditional tree; graph of operators

Shared Runtime Main Loop

1. Check for control messages: add new CQ, remove CQ, ...
2. Check for new stream tuples
   - Route each stream tuple through the operator graph (CPS)
   - Push output tuples to result consumers

Shared Evaluation

- Continuous query evaluation done by a network of operators in the shared runtime
- If multiple queries reference the same operator, we can evaluate it only once
  - Better than linear scalability!
- Each operator keeps track of the queries it helps to implement

Implementing Shared Evaluation

Sharing Predicates

- Simple cases: $<$, $\leq$, $=$, $>$, $\geq$, $\neq$
    - Construct a tree that divides domain of type into disjoint regions
    - For each tuple: walk the tree to find the region the tuple belongs in
        - Region implies which queries the tuple is still visible to
- Immutable functions can also be shared relatively easily

Implementing Shared Evaluation

Sharing Predicates

- Simple cases: $<, \leq, =, >, \geq, \neq$
  - Construct a tree that divides domain of type into disjoint regions
  - For each tuple: walk the tree to find the region the tuple belongs in
    - Region implies which queries the tuple is still visible to
- Immutable functions can also be shared relatively easily

Sharing Joins, Aggregates

Can also be done

- Even between queries with varying windows and predicates
- Requires some thought (say, a PhD thesis or two)

Adaptive Tuple Routing

- Given a new tuple, how do we route it through the graph of operators?

Adaptive Tuple Routing

- Given a new tuple, how do we route it through the graph of operators?
- Traditional approach: statically choose an "optimal" route for each stream
    - Hard optimization problem
    - Need to re-optimize when new queries defined or system conditions change (e.g. operator selectivity)

Adaptive Tuple Routing

- Given a new tuple, how do we route it through the graph of operators?
- Traditional approach: statically choose an "optimal" route for each stream
  - Hard optimization problem
  - Need to re-optimize when new queries defined or system conditions change (e.g. operator selectivity)
- TelegraphCQ approach: adaptive per-tuple routing
  - Push tuples one at a time through the operator graph; choose order of operators at runtime

Implementing Adaptive Routing

- For each tuple, maintain lineage
  - "What operators has this tuple visited?"
  - "Which queries can still see this tuple?"
- Implication: can't push down projections
- Make routing decisions on the basis of simple run-time statistics

Handling Overload

- Common scenario: peak stream rate >> average stream rate ("bursty")
- The system should cope gracefully

Handling Overload

- Common scenario: peak stream rate $>>$ average stream rate ("bursty")
- The system should cope gracefully
- Three alternatives:
  1. Spool tuples to disk, process later
     - But stream rates often exceed disk throughput

Handling Overload

- Common scenario: peak stream rate >> average stream rate ("bursty")
- The system should cope gracefully
- Three alternatives:
  1. Spool tuples to disk, process later
     - But stream rates often exceed disk throughput
  2. Drop excess tuples

Handling Overload

- Common scenario: peak stream rate $>>$ average stream rate ("bursty")
- The system should cope gracefully
- Three alternatives:
  1. Spool tuples to disk, process later
     - But stream rates often exceed disk throughput
  2. Drop excess tuples
  3. Substitute statistical summaries for dropped stream tuples

Handling Overload

- Common scenario: peak stream rate >> average stream rate ("bursty")
- The system should cope gracefully
- Three alternatives:
  1. Spool tuples to disk, process later
     - But stream rates often exceed disk throughput
  2. Drop excess tuples
  3. Substitute statistical summaries for dropped stream tuples
- Quality of Service (QoS)

Outline

Open Source DSMS

Esper

DSMS engine written in Java (GPL). SQL-like stream query language.

- `http://esper.codehaus.org`

TelegraphCQ

Academic prototype from UC Berkeley, based on PostgreSQL 7.3

- PostgreSQL's SQL dialect, plus stream-oriented extensions
- BSD licensed; `http://telegraph.cs.berkeley.edu`

StreamCruncher

DSMS engine written in Java. Free for commercial use (*not* open source).

- `http://www.streamcruncher.com`

Proprietary DSMS

StreamBase

A Stonebraker company. Founded in 2003.

Proprietary DSMS

StreamBase

A Stonebraker company. Founded in 2003.

Other Startups

- Coral8
- Apama (purchased by Progress Software in 2005)
- and more ...

Proprietary DSMS

StreamBase

A Stonebraker company. Founded in 2003.

Other Startups

- Coral8
- Apama (purchased by Progress Software in 2005)
- and more . . .

Established Companies

TIBCO BusinessEvents, Oracle BAM

Amalgamated Insight

- Based on the experience gained from TelegraphCQ
  - New codebase
- Application components:
  1. Continuous Query Engine
     - Modified version of PostgreSQL (currently 8.1.9+)
  2. Integration Framework
     - Connectors, input/output converters, query management
  3. Visualization
- Closed Series A funding in June 2006
- 1.0 release will be available Real Soon Now (currently RC3)
  - Lesson: PostgreSQL is a huge competitive advantage
- We're hiring :-)

Outline

1. The Need For Data Stream Processing
2. Stream Query Languages
3. Query Processing Techniques For Streams
   - System Architecture
   - Shared Evaluation
   - Adaptive Tuple Routing
   - Overload Handling
4. Current Choices For A DSMS
   - Open Source
   - Proprietary
5. **Demo**
6. Q & A

Outline

Q & A

Thank You.

Any Questions?