Continuous Queries (CACQ) over Streams

presented by:
T.J. Green
March 27, 2007

Slides by Sam Madden, 2002
With Mahul Shah, Joseph Hellerstein, and Vijayshankar Raman

CACQ Introduction

- Proposed continuous query (CQ) systems are based on static plans
  - But, CQs are long running
  - Initially valid assumptions less so over time
  - Static optimizers at their worst!
- CACQ insight: apply continuous adaptivity of eddies to continuous queries
  - Dynamic operator ordering avoids static optimizer danger
  - Process multiple queries simultaneously
  - Interestingly, enables sharing of work & storage

Outline

- Background
  - Motivation
  - Continuous Queries
  - Eddies
- CACQ
  - Contributions
  - Example driven explanation
- Results & Experiments

Motivating Applications

- “Monitoring” queries look for recent events in data streams
  - Sensor data processing
  - Stock analysis
  - Router, web, or phone events
- In CACQ, we confine our view to queries over ‘recent-history’
  - Only tuples currently entering the system
  - Stored in in-memory data tables for time-windowed joins between streams

Continuous Queries

- Long running, “standing queries”, similar to trigger systems
- Installed; continuously produce results until removed
- Lots of queries, over the same data sources
  - Opportunity for work sharing!
  - Global query optimization problem: hard!
  - Idea: adaptive heuristics not quite as hard?
    - Bad decisions are not final
  - Future work: finding an optimal plan (adaptively)
CACQ Query Model

- Monotonic queries, from point when query is registered
  - Terr et al., SIGMOD 1992
- Streaming answers
- Non-blocking operators
  - Windowed Symmetric Joins
  - Windows in tuples or time

Eddies & Adaptivity

- Eddies (Avnur & Hellerstein, SIGMOD 2000): Continuous Adaptivity
- No static ordering of operators
- Policy dynamically orders operators on a per tuple basis
- done and ready bits encode where tuple has been, where it can go

Outline

- Background
  - Motivation
  - Continuous Queries
  - Eddies
- CACQ
  - Contributions
  - Example driven explanation
- Results & Experiments

CACQ Contributions

- Adaptivity
  - Policies for continuous queries
  - Single eddy for multiple queries
- Tuple Lineage
  - In addition to ready and done, encode output history in tuple in queriesCompleted bits
  - Enables flexible sharing of operators between queries
- Grouped Filter
  - Efficiently compute selections over multiple queries
- Join Sharing through State Modules (SteMs)

Explication By Example

- First, example with just one query and only selections
- Then, add multiple queries
- Then, (briefly) discuss joins

Eddies & CACQ: Single Query, Single Source

- Use ready bits to track what to do next
  - All 1’s in single source
- Use done bits to track what has been done
  - Tuple can be output when all bits set
- Routing policy dynamically orders tuples
Multiple Queries

Outputting Tuples

Grouped Filter

Work Sharing via Tuple Lineage

Tradeoff: Overhead vs. Shared Work
Joins in CACQ

- Use symmetric hash join to avoid blocking
- Use State Modules (StMs) to share storage between joins with a common base relation
- Detail about effect on implementation & benefit in paper

Processing Joins Via State Modules

- Idea: Share join indices over base relations
- State Modules (StMs) are:
  - Unary indices (e.g. hash tables, trees)
  - Built on the fly (as data arrives)
  - Scheduled by CACQ as first class operators
- Based on symmetric hash join

Routing Policies

- Previous system provides correctness; policy responsible for performance
- Consult the policy to determine where to route every tuple that:
  - Enters the system
  - Returns from an operator
  - Basic Ticket Policy
    - Give operators tickets for consuming tuples, take away tickets for producing them
    - To choose the next operator to route, run a lottery
  - More selective operators scheduled earlier
  - Modification for CACQ
    - Give more tickets to operators shared by multiple queries (e.g. grouped-Maria)
    - When a shared operator outputs a tuple, charge it multiple tickets
    - Intuition: cardinality reducing shared operators reduce global work more than unshared operators
  - Not optimizing for the throughput of a single query!

CACQ Review

- Efficient mechanism for processing multiple simultaneous monitoring queries over streaming data sources
  - Share work by processing all queries within a single eddy
- Continuous adaptivity via eddies & routing policy
  - Queries come & go, but performance adapts without costly multiquey reoptimization
- Maximize ability to work share by explicitly encoding lineage
- Share selections via grouped filter
- Share join state via StMs

Outline

- Background
  - Motivation
  - Continuous Queries
  - Eddies
- CACQ
  - Contributions
  - Example driven explanation
- Results & Experiments
Evaluation

- Real Java implementation on top of Telegraph QP
  - 4,000 new lines of code in 75,000 line codebase
- Server Platform
  - Linux 2.4.10
  - Pentium III 733, 756 MB RAM
- Queries posed from separate workstation
  - Output suppressed
- Lots of experiments in paper, just a few here

Results: Routing Policy

CACQ vs. NiagaraCQ*

- Performance Competitive with Workload from NCQ Paper
- CACQ outperforms NCQ:

\[
\text{SELECT stocks.sym, articles.text}
\]

\[
\text{FROM stocks, articles}
\]

\[
\text{WHERE stocks.sym = articles.sym AND stocks.sid < stocks.sid}
\]

CACQ vs. NiagaraCQ #2

- No shared subexpressions, so no shared work!

CACQ vs. NiagaraCQ Graph

- CACQ: sharing and adaptivity for high performance monitoring queries over data streams
- Features
  - Adaptivity
  - Adapt to changing query workload without costly multi-query reoptimization
  - Work sharing via tuple lineage
  - Without constraining the available plans
  - Computation sharing via grouped filter
  - Storage sharing via S6M6s
- Future Work
  - More sophisticated routing policies
  - Batching & query grouping
  - Better integration with historical results (Chandrasekaran, VLDB 2002)
Questions?

Experiment: Increased Scalability

Tuple & Query Data Structures

- Per tuple bitmaps:
  - queryCompleted
  - What query has this tuple been output to or rejected by?
  - done
  - What operators have been applied to this tuple?
  - ready
  - What operators can be applied to this tuple?
- Per query bitmaps:
  - completionMask
  - What operators must be applied to output a tuple to this query?

<table>
<thead>
<tr>
<th>Tuple (10, 110, ...)</th>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

CACQ Query Model

- SELECT R.a, S.b from R, S where R.c = x and R.d[n] = S.c[n]
- Landmark queries
- Streaming answers
- Band Joins
- Windows in tuples or