How to Win a Hot Dog Eating Contest: Incremental View Maintenance with Batch Updates

Milos Nikolic, Mohammad Dashti, Christoph Koch
DATA lab, EPFL

SIGMOD, 28th June 2016
REALTIME APPLICATIONS

- Web Analytics
- Sensor Networks
- Cloud Monitoring

EVENTS
Continuously arriving data

RUNTIME ENGINE
Continuously evaluated views

DECISION SUPPORT

ACTIONS
REALTIME SYSTEMS: REQUIREMENTS

LOW LATENCY PROCESSING
Incremental view maintenance
\[ Q(D + \Delta D) = Q(D) + \Delta Q(D, \Delta D) \]

COMPLEX CONTINUOUS QUERIES
SQL queries (w/ nested aggregates)
No window semantics

SCALABLE PROCESSING
Synchronous execution model
Q1: How does the size of update affect the performance of incremental computation?

Q2: (Idea) How to achieve efficient distributed incremental computation?
PROBLEM: DBMS & stream engines with classical IVM can have poor performance on fast, long-lived data

OUR APPROACH: Compilation of SQL queries into incremental engines

\[ \text{HIGH-PERFORMANCE INCREMENTAL COMPUTATION} \]

\[ \text{PERF: Million view refreshes/sec for single-tuple updates} \]
Relations: R(A,B), S(B,C)

\[ Q := \text{SELECT SUM}(R.A \times S.C) \]
\[ \text{FROM R, S} \]
\[ \text{WHERE R.B = S.B} \]

Optimized \( \Delta R Q \)

\[ \text{SUM}(L \times R) \]
\[ \text{SUM}(A) \text{ GROUP BY B} \]
\[ \text{SUM}(C) \text{ GROUP BY B} \]

Delta for \( \Delta R \)

Update \( Q \)

Optimized Delta \( \Delta R Q \)

\[ \text{SUM}(A \times C) \]

Delta \( \Delta R Q \)

\[ \Delta R \]

\[ S \]
Relations: $R(A,B), S(B,C)$

$Q := \text{SELECT SUM}(R.A \times S.C) \text{ FROM } R, S \text{ WHERE } R.B = S.B$

Optimized Delta $\Delta_R Q$

Optimized Delta $\Delta_S Q$
ON UPDATE R BY ΔR:

// Pre-aggregate batch
tmp[B] := SELECT B, SUM(A)
  FROM ΔR
  GROUP BY B

// Update Q
Q  += SELECT SUM(tmp.V * mS.V)
  FROM tmp, mS
  WHERE tmp.B = mS.B

// Update mR
mR[B] += SELECT * FROM tmp
ON UPDATE R BY ΔR:

// Pre-aggregate batch
tmp[B] := SELECT B, SUM(A)
FROM ΔR
GROUP BY B

// Update Q
Q += SELECT SUM(tmp.V * mS.V)
FROM tmp, mS
WHERE tmp.B = mS.B

// Update mR
mR[B] += SELECT * FROM tmp

void onUpdateR(List<T> dR) {
    // Pre-aggregate batch
    HashMap<int, int> tmp;
    foreach (dA, dB) in dR
        tmp[dB] += dA;

    // Update Q (of type int)
    foreach (k, v) in tmp
        Q += v * mS[k];

    // Update mR
    foreach (k, v) in tmp
        mR[k] += v;
}
void onUpdateR(int dA, int dB) {
    Q += dA * mS[dB];
    mR[dB] += dA;
}

BASELINE

void onUpdateR(List<T> dR) {
    // Pre-aggregate batch
    HashMap<int,int> tmp;
    foreach (dA, dB) in dR
        tmp[dB] += dA;

    // Update Q (of type int)
    foreach (k, v) in tmp
        Q += v * mS[k];

    // Update mR
    foreach (k, v) in tmp
        mR[k] += v;
}
SINGLE-TUPLE VS. BATCH IVM

TPC-H, 10GB stream, batch size = 1…100,000, C++

MAIN RESULTS

1) Best performance w/ medium batch sizes (= bite sizes)

2) Single-tuple processing faster for 5 queries; 7 queries within 20% of best-batch performance

3) Batch pre-aggregation can enable cheaper maintenance

4) OOM faster than DBMS
DISTRIBUTED IVM

DESIGN CHOICE 1:
Local $\rightarrow$ Distributed programs

CHALLENGE:
Dependencies among statements prevent arbitrary re-orderings

DESIGN CHOICE 2:
Synchronous execution model (on top of Spark)
OUR APPROACH

LOCATION TAGS: LOCAL, PARTITIONED BY KEY, RANDOM
Annotate each node in query plan with location info

LOCATION TRANSFORMERS: Insert communication operations into query plan to preserve query semantics

HOLISTIC OPTIMIZATION: Minimize network cost
CONCLUSION

Much more in the paper:

• Single-tuple vs. batch incremental processing *(single-tuple can be better!)* + more experiments

• Distributed IVM (+ optimization framework)

• IVM of queries with nested aggregates

• Code and data-structure specialization

Download: [http://www.dbtoaster.org](http://www.dbtoaster.org)