Stream Joins and Dynamic Query Evaluation

WANG, Qichen
HUANG, Ziyue
Contents

1. Introduction
2. Dynamic Yannakakis Algorithm
3. Higher Order Incremental View Maintenance
Problem Definition

- **Input:**
  - Some input streams $R_1, R_2, \ldots, R_N$, represent $N$ different relations
  - Current time $t$

- **Output:**
  - $\Delta(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_N)$ at time $t$, as the incremental results of the join query
Challenges

- **Standard join no longer apply:**
  - Input/output arrives/is generated in a *continuous fashion*
  - No assumptions on input data
  - No indices on input data
  - Standard join algorithms no longer apply, due to their *blocking nature*  

- **Traditional solutions require explicitly create the join result:**
  - Most of the case, the join result will be very large and cannot store in memory.
  - Without storing the whole join result, the algorithm need to go through all the database and generate all the results with new data.
Traditional Solution: Yannakakis Algorithm

- An output-sensitive algorithm for $\alpha$-acyclic join query with running time $O(IN+OUT)$

**Algorithm:**

- Semi-join Phase: $O(IN)$
  - Bottom-up: from leaves to root, semi-join each relation with each of its children
  - Top-down: from root to leaves, semi-join each relation with its parent

- Join Phase: in arbitrary order, enumerate all the join result: $O(OUT)$
Yannakakis: An example

- Bottom-Up:
  - $T = T \bowtie U$
  - $R = R \bowtie T$
  - $R = R \bowtie S$

- Top-Down:
  - $S = S \bowtie R$
  - $T = T \bowtie R$
  - $U = U \bowtie T$

- After the semi-join phase, the remaining tuples are all live tuples, we can perform join in arbitrary order to output the result.
• Doing Yannakakis algorithm every time is costly;
• Storing the whole results requires too much space

• Research community seizes to devise small data structures that are able to extract the join query results quickly

• For example: Constant Delay Enumeration
A structure D supports CDE of S, if when ENUM(S) is invoked...

Property that enables the quick formation of the join result

\[ S = \{S_1, S_2, \ldots, S_n\} \]
Dynamic Query Evaluation: Dynamic Yannakakis Algorithm (Idris et al., 2017)

- **Problem Definition:**
  - Dynamic query evaluation of a given query $Q$ that has to be evaluated against a database $db$ that is constantly updated.
  - Database $db + \text{update } u + Q(db) \rightarrow \text{Compute } Q(db + u)$

- **Idea:** The Structure need to satisfy following requirement:
  1. Permits constant delay enumeration of $Q$’s result
  2. Check if a tuple $t$ belongs to the result of $Q$ in constant time
  3. Requires linear space to the size of $db$, independently of the $Q$’s result
  4. Given an update $u$, the structure is updated in $O(|db| + |u|)$, better in $O(|u|)$
Generalized Join Tree

- A width-1 GHD tree $T$ in which:
  1. All the atoms (relations) are leafs.
  2. Every interior node must have a guard.
- A GHD is acyclic, iff its atoms have a generalized join tree.

Hyperedge: subset of vars of all the vars of the atoms

Diagram:

- $[x, y, z]$ is guard for $[y]$
- $R$ is guard for $[x, y, z]$
- $[y, v]$ is guard for $[y]$
- $R$ is guard for $[x, y]$
Constant Delay Yannakakis (CDY)

- Observation: After the first bottom-up phase, traversal the query result can be enumerated with constant delay, without materializing it.

**Live tuples**: the surviving tuples that reside in the tree nodes after Bottom-Up Phase

**Algorithm (CDY)**
- Execute the Bottom-Up phase of Classical Yannakakis
- Using the index between nodes to perform the CDE

The CDE is essentially a multi-way hash join where:
1) Relation at the root is the probe relation
2) The other $R_n$ as built relation, accessible by the $L_n$ indices.
Example of Enumeration
Dynamic Yannakakis

- Such structure can support constant time enumeration, now the challenge is to maintain the structure.

- **Naïve approach:** Re-run CDY after each update. \( \rightarrow \) Time linear to the size of db!!!

- To avoid the overhead we can maintain the live tuples incrementally.

- They define following recursive queries:
Recursive Queries:

Let $T$ be a join tree.

To every node $n$ associate 2 queries: 1 over its variables, and 1 over its parents variables.

- If $n$ is a non-leaf node, the live tuples on all of its guards children.

For every leaf $a$ of the tree we have $\Psi^T_a := \pi_{\text{pvar}(a)} a$ and $\Lambda^T_a := a$.

The $\Lambda^T_n$ is the set of live tuples on tree node $n$.

Notice they divide the $\Lambda^T_a = \Join_{c \in ch(n)} \Psi^T_c$ into two parts, guard and non-guard.
T-representation (T-rep)

Let $T$ be a join tree and let $\text{db}$ a database. A T-rep is a data structure $D$ that for each node $n$ of $T$ contains:

- An index $L_n$ of $\Lambda_n(\text{db})$ on $\text{pvar}(n)$
- A relation $P_n$ that materializes $\Psi_n(\text{db})$
- A relation $G_n$ that materializes $\Gamma_n(\text{db})$
- For each non-guard child $c$ an index $G_{n,c}$ of $G_n$ on $\text{pvar}(n)$
The $P_n = \Psi_n(db)$ are the gray tables

The $L_n = \Lambda_n(db)$ are the white tables
Dynamic Yannakakis (DYN)

- The DYN maintains the T-reps under updates.
  1. Traverse the nodes of T in a bottom-up fashion when an update u arrives.
  2. During the traversal DYN materializes the deltas and apply the updates to each node

**Delta Computation:**

- If node n is an atom α → DYN uses the update u to compute $\Delta L_α = u_α$ and $\Delta P_α = \pi_{pvar(α)} u_α$ (the projection using a hash-based aggregation algorithm)

- If node n is a hyperedge → Uses a bound U that contains all tuples that can appear in $\Delta Γ_n(db, u)$ or $\Delta Λ_n(db, u)$. Then iterates through all tuples in U and using the fact that $Ψ_c(db + u)(t) = P_c(t) + \Delta P_c(t)$ for every tuple t, it calculates all the $\Delta L_n$, $\Delta P_n$ and $\Delta G_n$
Example of insertion on non-guard leaf
Example of deletion on a guard leaf
From the example, we can see that the updates on guard leaf is simple—>
*Only changing one tuple of its parents*

So if all nodes are the guard of its parents, then the query is simple.

\[ \text{DYN}(D, u) \text{ produces a T-rep for } db + u \text{ in time } O(|u|) \text{ (linear in } u) \text{ iff the } T \text{ is simple.} \]

\[ \text{DYN}(D, u) \text{ for a join tree } T \text{ in the worst case performs in time } O(|db| + |u|) \text{ similar to using CDY after every update } \Omega(|db + u|). \text{ But its performance in practice it much better than the upper bound.} \]
Complexity (2)

- Some scientists (Belholz et al., 2017) proof that with linear space, some query can achieve constant time update, but most of the query cannot.

- To achieve fast update time and query time, we might need to use more space to store some extra information.

- Here comes another approach: (H)IVM
Data Model and Query Language

- Generalizes multiset relations (as in SQL) to tuple multiplicities that are rational numbers

<table>
<thead>
<tr>
<th>R</th>
<th>( \langle A, B \rangle \rightarrow # )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle a, b_1 \rangle \rightarrow 2 )</td>
<td></td>
</tr>
<tr>
<td>( \langle a, b_2 \rangle \rightarrow -3 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S_1</th>
<th>( \langle B, C \rangle \rightarrow # )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle b_1, c_1 \rangle \rightarrow 2 )</td>
<td></td>
</tr>
<tr>
<td>( \langle b_1, c_2 \rangle \rightarrow -10 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S_2</th>
<th>( \langle B, C \rangle \rightarrow # )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle b_1, c_1 \rangle \rightarrow 3 )</td>
<td></td>
</tr>
<tr>
<td>( \langle b_1, c_2 \rangle \rightarrow 3 )</td>
<td></td>
</tr>
<tr>
<td>( \langle b_2, c_1 \rangle \rightarrow -11 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S_1 + S_2</th>
<th>( \langle B, C \rangle \rightarrow # )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle b_1, c_1 \rangle \rightarrow 5 )</td>
<td></td>
</tr>
<tr>
<td>( \langle b_1, c_2 \rangle \rightarrow -7 )</td>
<td></td>
</tr>
<tr>
<td>( \langle b_2, c_1 \rangle \rightarrow -11 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R \bowtie (S_1 + S_2)</th>
<th>( \langle A, B, C \rangle \rightarrow # )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle a, b_1, c_1 \rangle \rightarrow 10 )</td>
<td></td>
</tr>
<tr>
<td>( \langle a, b_1, c_2 \rangle \rightarrow -14 )</td>
<td></td>
</tr>
<tr>
<td>( \langle a, b_2, c_1 \rangle \rightarrow 33 )</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Sum}_{AC;1/2}(R \bowtie (S_1 + S_2)) | \langle A, C \rangle \rightarrow \#
\]

| | --- |
| \( \langle a, c_1 \rangle \rightarrow 21.5 \) |
| \( \langle a, c_2 \rangle \rightarrow -7 \) |
\( \sigma_{C < \text{Sum}_{A:B} RS} \)

select * from S

where S.C < (select sum(B) from R where R.A=S.A)
Higher-order Incremental View Maintenance

- Realtime analytics
- Algorithmic trading
- Scientific data analysis
Example 1

- Consider a query $Q$ that counts the number of tuples in the product of relations $R$ and $S$. For now, we only want to maintain the view of $Q$ under insertions.

- Denote by $\Delta R$ (resp. $\Delta S$) the change to a view as one tuple is inserted into $R$ (resp., $S$). Suppose we simultaneously materialize the four views. Suppose we simultaneously materialize the four views,

- $Q$ (0-th order),
- $\Delta R \ Q = \text{count}(S)$ and $\Delta S \ Q = \text{count}(R)$ (first order)
- $\Delta R(\Delta S \ Q) = \Delta S(\Delta R \ Q) = 1$ (second order, a “delta of a delta query”)
Example 1

| time point | insert into | $||R||$ | $||S||$ | $Q$ | $\Delta R Q$ | $\Delta S Q$ | $\Delta R \Delta S Q$, $\Delta S \Delta R Q$ |
|------------|-------------|---------|---------|-----|-------------|-------------|---------------------------------|
| 0          | –           | 2       | 3       | 6   | 3           | 2           | 1                               |
| 1          | S           | 2       | 4       | 8   | 4           | 2           | 1                               |
| 2          | R           | 3       | 4       | 12  | 4           | 3           | 1                               |
| 3          | S           | 3       | 5       | 15  | 5           | 3           | 1                               |
| 4          | S           | 3       | 6       | 18  | 6           | 3           | 1                               |
Delta of a Query

\[
\Delta Q(D, \Delta D) := Q(D + \Delta D) - Q(D)
\]

\[
\Delta (Q_1 + Q_2) := (\Delta Q_1) + (\Delta Q_2),
\]

\[
\Delta (\text{Sum}_{\overline{A};f} Q) := \text{Sum}_{\overline{A};f} (\Delta Q),
\]

\[
\Delta (Q_1 \bowtie Q_2) := ((\Delta Q_1) \bowtie Q_2) + (Q_1 \bowtie (\Delta Q_2))
\]

\[
\Delta (\sigma_\theta Q) := \sigma_\theta (\Delta Q).
\]
Example 2

Q = select sum(LI.PRICE * O.XCH)
from Orders O, LineItem LI where O.ORDK = LI.ORDK;

on insert into O values (ordk, custk, xch) do {
    Q += xch * Q_O[ordk];
    Q_LI[ordk] += xch;
}

on insert into LI values (ordk, partk, price) do {
    Q += price * Q_LI[ordk];
    Q_O[ordk] += price;
}
Example 2

\((\Delta \circ Q)[ordk] = \)

select sum(LI.PRICE) from Lineitem LI where LI.ORDK=ordk

Performing IVM of the deltas.
The triggers can be evaluated in constant time for single-tuple inserts.
Example 3

Given schema $R(AB)$, $S(CD)$, and query

select sum($A \times D$) from $R$, $S$ where $B = C$

\[
\text{Sum}_{} (A \times D) (\sigma_{B \equiv C} (R \bowtie S))
\]
Example 3

$$\Delta \text{Sum}_{\langle \rangle; A \ast D} (\sigma_{B=C}(R \Join S)) =$$

$$\text{Sum}_{\langle \rangle; A \ast D} \Delta (\sigma_{B=C}(R \Join S)) =$$

$$\text{Sum}_{\langle \rangle; A \ast D} (\sigma_{B=C} \Delta (R \Join S))$$

$$\Delta (R \Join S) = (\Delta R) \Join S + R \Join (\Delta S) + (\Delta R) \Join (\Delta S).$$

$$\text{Sum}_{\langle \rangle; A \ast D} (\sigma_{B=C}((\Delta R) \Join S))$$
Example 4

Assume that $\Delta R$ is an insertion of a single tuple $<A : x, B : y>$. The delta query

$$\sum_{A \neq D}(\sigma_{B = C}(\{<A : x, B : y>\} \bowtie S))$$

can be simplified to

$$\sum_{x \neq D}(\sigma_{y = C}S)$$
Binding Patterns

- Input variables or parameters without which we cannot evaluate these expressions
- Output variables, the columns of the schema of the query result
select * from R
where B < (select sum(D) from S where A > C)

\[
\text{Sum}_{*;1}(\sigma_{B<\text{Sum}_{;D}(\sigma_{A>C}(S))}R)
\]

output variables: all columns of the schema of R

In the subexpression \[
\text{Sum}_{;D}(\sigma_{A>C}(S))
\], A is an input variable.

Delta adds input variables parameterizing the query with the update.

In Example 4 for instance, the delta query has input variables x and y to pass in the update.
The degree $\text{deg}(Q)$ of query $Q$ is the number of relations joined together.

If $\text{deg}(Q) > 0$, then $\text{deg}(\Delta Q) = \text{deg}(Q) - 1$.

Turns a query into a set of update triggers that together maintain the view of the query and a set of auxiliary views.
procedure VT(Q, \(\vec{x}_{in}\)):
foreach relation name \(R\) used in \(Q\) do {
\(T_R\) += (foreach \(\vec{x}_{in}\) do \(Q[\vec{x}_{in}] += \Delta_R Q[\vec{x}_{in} \cdot D_R]\))
if deg(\(Q\)) > 0 then {
let \(D\) be a new variable of type relation of schema \(R\);
VT(\(\Delta_R Q\), \(\vec{x}_{in} \cdot D\))
}
}
two relations R and S and query Q has degree 2

Q += ΔRQ[Dr];
foreach D1 do ΔRQ[D1] += ΔRΔRQ[D1, Dr];
foreach D2 do ΔSQ[D2] += ΔRΔSQ[D2, Dr]
\[(\Delta_{\text{sgn}_R} R(x, y) \Delta_{\text{sgn}_S} S(z, u) Q)[x, y, z, u]\]

\[
\text{sgn}_R \text{sgn}_S \text{Sum} \langle \rangle \cdot x \cdot u (\sigma_{y=z} \{ \langle \rangle \})
\]

\[Q \quad \Delta_{+R(x, y)} Q[x, y];
\]

\[
\text{foreach } u \text{ do } \Delta_{+S(y, u)} Q[y, u] \quad \Delta_{+S(y, u)} Q[y, u] \quad \Delta_{-S(y, u)} Q[y, u] \quad \Delta_{-S(y, u)} Q[y, u] \\
\quad \{ \langle \rangle \mapsto x \cdot u \};
\]

\[
\text{foreach } u \text{ do } \Delta_{-S(y, u)} Q[y, u] \quad \Delta_{-S(y, u)} Q[y, u] \\
\quad \{ \langle \rangle \mapsto x \cdot u \};
\]