Wander Join: Online Aggregation via Random Walks

Authors: Feifei Li, Bin Wu, Ke Yi, Zhuoyue Zhao

Presenter: Gowtham Rajasekaran
Introduction

• OLTP
  ❖ Enables the real-time execution of large numbers of database transactions.
  ❖ Focus on Correctness and Efficiency.

• OLAP
  ❖ Answer multi-dimensional analytical queries on large data swiftly in computing.
  ❖ Large data and many tables.
  ❖ Efficiency and Correctness tradeoff?
Analytical Query (TPC-H)

```
SELECT n_name, SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM customer, orders, lineitem, supplier, nation, region
WHERE c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND l_suppkey = s_suppkey
  AND c_nationkey = s_nationkey
  AND s_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'ASIA'
  AND o_orderdate >= MDY(1,1,1994)
  AND o_orderdate < MDY(1,1,1994) + 1 UNITS YEAR
GROUP BY n_name
ORDER BY revenue DESC
```

Query to get revenue volume done through local suppliers.

Joins are expensive

“Lot of time to get 100% accurate answer”
Online Aggregation

- Online aggregation offers a flexible tradeoff between query efficiency and accuracy.
- Analytical queries do not really need a 100% accurate answer.
- The Database provides an estimated answer with some type of quality assurance at start, then improve accuracy as time goes on.
- Eg: Estimate for the query is 100 with a ± 5% confidence interval (between [95,105]) with a confidence level of 92%.

\[
\Pr[\hat{Y} - \varepsilon < Y < \hat{Y} + \varepsilon] > 0.95
\]

Confidence Interval  Confidence Level

A Sample Online Aggregation interface.
Ripple Join [Haas, Hellerstein, SIGMOD 99]

- Repeatedly take samples in a round robin fashion from each table, and only performs the join on the sampled tuples.
- Tuples need to be stored in random order.
- Assuming two relations R1, R2.
  - Select a random tuple a from R1
  - Join with previously sampled tuples from R2.
  - Select a random tuple b from R2
  - Join with previously sampled tuples from R1.
  - Repeat
Ripple Join variants

- **Block Ripple Join**: Sample units are blocks of tuples at size $\alpha$.

- **Index Ripple Join**: If there is an index on the join attributes of R2, the index ripple join uses the index to identify the tuples in R2 that join with a given random tuple selected from R1 during a sampling step.

- **Hash Ripple Join**: All the sampled tuples are kept in hash tables in memory. Thus, calculating the join condition of a new sampled tuple with previous sampled tuples is very fast (saving I/O). But it is not scalable due to size limitations.
Cons of Ripple Join

- Performance of ripple join depends on the fraction of the randomly selected tuples that could actually join (Low fraction especially in natural joins).
- It demands that the tuples in each table be stored in a random order.
Wander Join

- The key idea is to model a join over k tables as a join graph, and then perform random walks in this graph.
- The vertices in the graph indicate the tuples and the edges denotes that the two tuples can join.
- Randomly sample a tuple only from one of the tables.
- Conducts a random walk starting from that tuple. In every step of the random walk, only the “neighbors” of the already sampled tuples are considered.
- Random walks provide unbiased estimators for various aggregation functions.
Consider an example on the natural join between 3 tables R1, R2, R3:

\[ R1(A, B) \bowtie R2(B, C) \bowtie R3(C, D) \]

- A path can be randomly sampled by first picking a vertex in R1 uniformly at random, and then “randomly walking” towards R3.

- Which path to select?

- Consider \( a_1 \Rightarrow b_1 \Rightarrow c_1 \). The probability this path will be sampled is \( \frac{1}{7} \times \frac{1}{3} \times \frac{1}{2} \approx 2.4\% \)

- Consider \( a_6 \Rightarrow b_6 \Rightarrow c_6 \). The probability this path will be sampled is \( \frac{1}{7} \times 1 \times 1 \approx 14\% \)

- What if the value of D in tuple \( c_6 \) is very large, leading to an overestimate
Horvitz-Thompson estimator

- Wander join uses the Horvitz-Thompson estimator to remove the bias and get an unbiased estimator of the aggregate function.
- Suppose path $\gamma$ is sampled with probability $p(\gamma)$ and $v(\gamma)$ be the expression to be aggregated on $\gamma$
- The sampling probability is represented as
  
  $$p(\gamma) = \frac{1}{|R_1|} \times \frac{1}{d_2(t_1)} \times \frac{1}{d_3(t_2)}$$

  where $d_{i+1}(t_i)$ is the number of tuples in $R_{i+1}$ that join with $t_i$.

Unbiased estimator of $\Sigma v(\gamma) = \frac{v(\gamma)}{p(\gamma)}$
What if the random walk gets stuck?

- Consider the case below where we randomly walk from vertex $a_1$ to $b_3$. There is no path from $R_2$ to $R_3$.
- The sampled path is not rejected.
- It returns 0 as its aggregation estimate for the Horvitz Thompson estimator to remain unbiased.
### SQL Query

```sql
SELECT SUM(Price)  
FROM Customers C, Orders O, Items I  
WHERE  
  C.Nation = 'China'  
  C.CID = O.BuyerID  
  O.OrderID = I.OrderID
```

### Sum of prices of products bought from China

<table>
<thead>
<tr>
<th>Nation</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>1</td>
</tr>
<tr>
<td>US</td>
<td>2</td>
</tr>
<tr>
<td>China</td>
<td>3</td>
</tr>
<tr>
<td>UK</td>
<td>4</td>
</tr>
<tr>
<td>China</td>
<td>5</td>
</tr>
<tr>
<td>US</td>
<td>6</td>
</tr>
<tr>
<td>China</td>
<td>7</td>
</tr>
<tr>
<td>UK</td>
<td>8</td>
</tr>
<tr>
<td>Japan</td>
<td>9</td>
</tr>
<tr>
<td>UK</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BuyerID</th>
<th>OrderID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OrderID</th>
<th>ItemID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>301</td>
<td>$2100</td>
</tr>
<tr>
<td>2</td>
<td>304</td>
<td>$100</td>
</tr>
<tr>
<td>3</td>
<td>201</td>
<td>$300</td>
</tr>
<tr>
<td>4</td>
<td>306</td>
<td>$500</td>
</tr>
<tr>
<td>3</td>
<td>401</td>
<td>$230</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>$800</td>
</tr>
<tr>
<td>2</td>
<td>201</td>
<td>$300</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>$200</td>
</tr>
<tr>
<td>4</td>
<td>301</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>201</td>
<td>$600</td>
</tr>
</tbody>
</table>
### SELECT Query

```
SELECT SUM(Price) 
FROM Customers C, 
     Orders O, 
     Items I 
WHERE 
  C.Nation = 'China' 
  C.CID = O.BuyerID 
  O.OrderID = I.OrderID
```
SELECT SUM(Price) 
FROM Customers C, Orders O, Items I 
WHERE 
  C.Nation = 'China' 
  C.CID = O.BuyerID 
  O.OrderID = I.OrderID
SELECT SUM(Price) 
FROM Customers C, 
Orders O, 
Items I 
WHERE 
C.Nation = 'China' 
C.CID = O.BuyerID 
O.OrderID = I.OrderID
SELECT SUM(Price) FROM Customers C, Orders O, Items I WHERE C.Nation = 'China' AND C.CID = O.BuyerID AND O.OrderID = I.OrderID

Unbiased estimator: \[
\frac{\$500}{\text{sampling prob.}} = \frac{\$500}{1/3 \cdot 1/4 \cdot 1/3}
\]
Wander join for Acyclic queries

- To solve queries whose joins are not Chain, we model a Join query graph.
- Tables are represented as vertices and joins represented as edges.
- Fix a walk order such that each table in the walk order must be adjacent (in the query graph) to another one earlier in the order.
- Example: \([R1, R2, R3, R4, R5]\) and \([R2, R3, R4, R5, R1]\).
- \([R3,R4,R5,R2,R1]\) cannot be a walk order because \(R3\) cannot reach \(R4\) before going through \(R2\).
- After selecting the walk order, wander join performs the random walks as before the only difference is that a random walk may now consist of both “walks” and “jumps”.
- In the order \([R1, R2, R3, R4, R5]\), After reaching \(R3\), wander join jumps to \(R4\) through \(R2\).
Wander join for Cyclic queries

- Wander join extends the algorithm used for acyclic queries to solve cyclic queries.
- Find any spanning tree which covers all the vertices (tables) in the graph.
- Perform the random walks on this spanning tree.
- After sampling path $\gamma$ on the spanning tree, we need to check if the path satisfies the non-spanning tree edges too.
- In the given cyclic graph, Consider the walk order [R1, R2, R3, R4, R5], after sampling this path, wander join also checks if the non-spanning tree edge [R3, R5] join.
- If the tuples in R3 and R5 don’t join then the algorithm returns 0 as the estimate.
Indexing justification

- Indexes are a crucial factor in Random walk approach.

- Consider an example on the natural join between 3 tables R1, R2, R3:
  \[ \text{R1}(A, B) \Join \text{R2}(B, C) \Join \text{R3}(C, D) \]

- R2 needs to have an index on its B attribute, and R3 needs to have an index on its C attribute.

- Insufficient indexing will influence/limit random walk orders.
  - Add Directed edges in join query graph if there is an index on the join attribute.

- Assuming that having plenty of indexes is reasonable.
  - Query speed up
  - Not much cost and overhead due to a data warehousing environment.
Walk Plan optimizer

Factors affecting Walk order performance

- Structure of the join data graph.
  Eg: In the given graph sampling probability of $[R1, R2, R3] = 28.6\%$, $[R3, R2, R1] = 100\%$.

- Selection predicate
  If there is a selection predicate on an attribute and there is a table with an index on that attribute, it is preferable to start from that table.

- Non uniformity in Data Distribution
  Even if the success probability of the random walks is the same, different walk orders may result in different non-uniformity.
Walk Plan optimizer

- Enumerate all walk orders.
- Conduct ~ 100 trial random walks for each order.
- Measure the variance of all orders.
- Select the best plan which has minimum variance.
- All trials runs are still useful.
Convergence comparison on Standalone implementation

Q7

Q10
Wander Join in PostgreSQL (Sufficient memory)

(a) PostgreSQL full join

(b) Wander join in PostgreSQL
Running on Insufficient Memory (4GB)

- Insufficient memory incurs a heavy penalty due to random I/O.
- Growth is still logarithmic.
- Fundamentally: Random sampling at odds with hard disks.
<table>
<thead>
<tr>
<th></th>
<th>SF$^1$</th>
<th>sufficient memory</th>
<th>limited memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DBO</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cl$^3$ AE$^4$</td>
<td>Cl AE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>10</td>
<td>32.24</td>
<td>107.27</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>74.29</td>
<td>249.94</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>65.17</td>
<td>428.39</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>90.23</td>
<td>707.04</td>
</tr>
<tr>
<td>Q7</td>
<td>10</td>
<td>33.62</td>
<td>103.3</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>73.03</td>
<td>205.7</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>57.82</td>
<td>326.35</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>77.92</td>
<td>445.86</td>
</tr>
<tr>
<td>Q10</td>
<td>10</td>
<td>40.43</td>
<td>146.57</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>98.96</td>
<td>326.67</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>109.19</td>
<td>697.06</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>138.87</td>
<td>829.97</td>
</tr>
</tbody>
</table>
# Wander Join vs Ripple Join

<table>
<thead>
<tr>
<th></th>
<th>Wander Join</th>
<th>Ripple Join</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sampling method</strong></td>
<td>Independent but non-uniform</td>
<td>Uniform but non-independent</td>
</tr>
<tr>
<td><strong>Index needed?</strong></td>
<td>Yes</td>
<td>Index / Random storage</td>
</tr>
<tr>
<td><strong>Convergence time</strong></td>
<td>~ 3s</td>
<td>~50s</td>
</tr>
<tr>
<td>(20GB data, 3 tables)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Logarithmic</td>
<td>Slightly less than linear</td>
</tr>
</tbody>
</table>
| **System implementation** | ● PostgreSQL (finished)  
                          | ● Oracle (in progress)  
                          | ● SparkSQL (in progress)  
                          | ● Informix (internal project)  
                          | ● DBO                        |
Conclusion

- Query sampling immediately solves the online aggregation problem.
- Wander join and ripple join have demonstrated that a non-uniform or a non-independent sample can be used to estimate the aggregate with quality guarantees.
- Results show that wander join outperforms ripple join and DBO by orders of magnitude in speed for achieving the same accuracy.
References

- Christopher Jermaine, et al. *Scalable Approximate Query Processing with the DBO Engine*. In SIGMOD ’07.
- Slides adapted from Authors presentation slides.