Wander Join: Online Aggregation for Joins

Feifei Li [1], Bin Wu [2], Ke Yi [2], Zhuoyue Zhao [3]
[1]University of Utah [2]Hong Kong University of Science and Technology [3] Shanghai Jiao Tong University
lifeifei@cs.utah.edu {bwuac, yike}@cse.ust.hk zzy7896321@sjtu.edu.cn

ABSTRACT

Joins are expensive, and online aggregation over joins was proposed to mitigate the cost, which offers a nice and flexible tradeoff between query efficiency and accuracy in a continuous, online fashion. However, the state-of-the-art approach, in both internal and external memory, is based on ripple join, which is still very expensive and may also need very restrictive assumptions (e.g., tuples in a table are stored in random order). We introduce a new approach, *wander join*, to the online aggregation problem by performing random walks over the underlying join graph. We have also implemented and tested wander join in the latest PostgreSQL.

1. INTRODUCTION

Joins are the common operations in relational databases. In interactive data analytics, users often need the database system to quickly answer ad hoc queries with complex joins involving multiple tables over gigabetyes or even terabytes of data. For instance, the TPC-H benchmark specifies 22 queries, 17 of which are joins and the most complex one involves 8 tables. Unfortunately, even the leading relational database system may take hours to answer such complex queries, especially when working with large data.

A line of work known as "online aggregation" [8] was proposed, with the observation that users would prefer approximate answers with quality guarantees (in the form of confidence interval) if they can be returned much more quickly. The quality of the approximate answers improves over time. The user can immediately stop the query whenever the quality is satisfactory, saving a lot of computation resources.

```
SELECT SUM(l_extendedprice * (1 - l_discount))
FROM customer, orders, lineitem
WHERE c_mktsegment='BUILDING'
  AND c_custkey=o_custkey
  AND l_orderkey=o_orderkey
```

Consider the query above based on Q3 in TPC-H, which queries the total value of the orders placed by the customers in certain market segment. It involves natural joins over 3 tables. Running the full join to complete on TPC-H data with scale factor of 20 takes over 2 minutes while online aggregation achieves a 2.4% confidence interval within 1 second. If the user is satisfied with the result, he/she can stop the query. Or the user can wait for a bit longer to get a more accurate answer.

Prior work of online aggregation uses the ripple join algorithm to perform online aggregation over joins. However, no commercial or full-fledged open-source database systems have adopted ripple join due to its inherent limitations and weaknesses, which we will discuss in Section 2. We introduce a new approach called *wander join*, which outperforms ripple join significantly and is easy to implement in a full-fledged database engine. We implemented wander join in PostgreSQL. Users can pose online aggregation queries with new keyword `ONLINE` through PostgreSQL CLI. We also connected it to Apache Zeppelin, which allows users to see the convergence of estimators over time through a web interface.

We give the problem formulation and the background of online aggregation in Section 2. In Section 3 we describe the key ideas of wander join. In Section 4, we present the system implementation of wander join in PostgreSQL. In Section 5, we show some evaluation results to illustrate the performance advantage of wander join over ripple join and full join and present a demonstration plan of wander join. Lastly, we survey the related work in Section 6.

2. FORMULATION AND BACKGROUND

Problem formulation. The type of queries we aim to support is SQL queries of the form

```
SELECT g, AGG(expression)
FROM R_1, R_2, ..., R_k
WHERE join conditions AND selection predicates
GROUP BY g
```

where `AGG` can be any of the standard aggregation functions (e.g. `SUM`, `AVG`, `COUNT`, `VARIANCE`) and `expression` can involve any attributes of the tables. The `join conditions` consist of equality or inequality conditions between pairs of the tables, and `selection predicates` can be applied to any number of the tables.

At any point in time during query processing, the algorithm should output an estimator $\tilde{Y}$ for `AGG(expression)` together with a confidence interval, i.e.,

$$\Pr[|\tilde{Y} - \texttt{AGG(expression)}| \leq \varepsilon] \geq \alpha.$$

Here, ε is called the *half-width* of the confidence interval

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
 ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2899413

and α the *confidence level*. The user should specify one of them and the algorithm will continuously update the other as time goes on. The user can terminate the query when it reaches the desired level. Alternatively, the user may also specify a time limit on the query processing, and the algorithm should return the best-effort estimate within the limit, together with a confidence interval.

Ripple join. The ripple join algorithm [6] is central to prior work on online aggregation over joins. It repeatedly takes independent random samples from each table in a round-robin fashion and stores them in the memory. When a new tuple is retrieved, it is joined with all tuples stored in the memory to get samples from the join results. Clearly, these samples are not independent, nonetheless the sample mean can still serve as an unbiased estimator but the formulas for confidence intervals are complex and differ from standard statistical formulas [5, 7].

In spite of the nice properties of online aggregation, no commercial or full-fledged open-source database systems support this technique because of two weaknesses of the ripple join algorithm: (1) The performance highly depends on the fraction of the sampled tuples that actually join, which often can be extremely low in practice. (2) The algorithm requires that the tuples retrieved from the tables are in random order. However, tuples are usually clustered according to primary key to facilitate other operations. The system implementation of ripple join, DBO [4, 13] actually has to be built from scratch (independent from any existing database engines) to support the random-order storage of tuples.

3. WANDER JOIN

In this section, we illustrate how wander join works by an example of the natural join between 3 tables R_1, R_2, R_3:

$$R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D), \tag{1}$$

where $R_1(A, B)$ means that R_1 has two attributes A and B, etc. The natural join returns all combinations of tuples from the 3 tables that have matching values on their common attributes. We assume that R_2 has an index on attribute B, R_3 has an index on attribute C, and the aggregation function is $\texttt{SUM}(D)$.

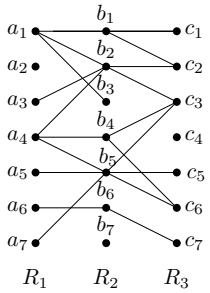


Figure 1: The 3-table join data graph: there is an edge between two tuples if they can join.

We model the join as a graph called *join data graph*. Figure 1 shows a possible join data graph of the 3-table natural join. Where each tuple is modeled as a vertex and there is an edge between two tuples whenever they can join, i.e. have the same value on their common attributes. Each join result is then represented by a path of length 2 from some vertex in R_1 to some vertex in R_3. Sampling a join result boils down to sampling such a path from the graph.

Sampling a path from the graph does not require the graph to be explicitly constructed. Rather, we first pick a vertex in R_1 uniformly at random and then perform a random walk towards R_3. In each step of the random walk, we sample from the tuples in the next table that can join the previous one uniformly at random, which can be done efficiently via a B-Tree index.

Different paths may have different probabilities to be sampled. It is obvious that sample mean is no longer an unbiased estimator. Fortunately, the *Horvitz-Thompson estimator* [9] can serve as an unbiased estimator. Suppose a path γ is sampled with probability $p(\gamma)$, whose value to be aggregated is $v(\gamma)$. Then $v(\gamma)/p(\gamma)$ is an unbiased estimator of the $\texttt{SUM}$ aggregator. If the random walk gets stuck before the path is complete, we should return 0 as the estimator. The probability of a path γ can be easily computed on-the-fly as the path is sampled. Let $\gamma = (t_1, t_2, t_3)$ and $d_k(t_i)$ be the number of tuples in table k that can join t_i. Then the probability of the path γ is

$$p(\gamma) = \frac{1}{|R_1|} \cdot \frac{1}{d_2(t_1)} \cdot \frac{1}{d_3(t_2)} \tag{2}$$

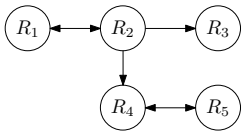
With a number of independent unbiased estimators computed from the sampled paths, we can get an unbiased estimator with lower variance by taking the average. We omit the details for the formulas of variance and confidence interval for different aggregate operators in this demo paper, but they can be derived following similar techniques from [5].

4. WANDER JOIN IN POSTGRESQL

We implemented wander join in PostgreSQL by extending its parser, plan generator, and query exeuctor. We added serveral keywords to the SQL parser like `ONLINE`, `WITHTIME`, `CONFIDENCE`, `REPORTINTERVAL`. Following is the online aggregation query for the example based on Q3 of TPC-H:

```
SELECT ONLINE SUM(l_extendedprice * (1 - l_discount))
FROM customer, orders, lineitem
WHERE c_mktsegment='BUILDING'
  AND c_custkey=o_custkey
  AND l_orderkey=o_orderkey
WITHTIME 20 CONFIDENCE 95 REPORTINTERVAL 1
```

The above query tells the engine that it is an online aggregation query, such that the engine should report the estimations and their associated confidence intervals, calculated with respect to 95% confidence level every 1 second for up to 20 seconds.



R_1, R_2, R_3, R_4, R_5
R_1, R_2, R_4, R_3, R_5
R_1, R_2, R_4, R_5, R_3
R_2, R_1, R_3, R_4, R_5
R_2, R_1, R_4, R_3, R_5
R_2, R_1, R_4, R_5, R_3
R_2, R_3, R_1, R_4, R_5
R_2, R_3, R_4, R_1, R_5
R_2, R_3, R_4, R_5, R_1
R_2, R_4, R_1, R_3, R_5
R_2, R_4, R_1, R_5, R_3
R_2, R_4, R_3, R_1, R_5
R_2, R_4, R_3, R_5, R_1
R_2, R_4, R_5, R_1, R_3
R_2, R_4, R_5, R_3, R_1

Figure 2: A directed join query graph and all its walk plans.

The online aggregation queries are passed to plan generator for wander join. The plan generator constructs the *join*

query graph (or *query graph* in short), where each vertex represents a table and there is a *directed* edge from one table to another when there is an index on the join attribute of the second table. Figure 2 shows a possible query graph and all its walk plans. A walk plan is a walk order of the tables. We first try to find a valid walk order that for each table R_i (except the first one in the order), there exists a table R_j earlier in the order such that there is a join condition between R_i and R_j. In addition, R_i has an index on the join attribute. If it is not possible to find such a valid walk order, we relaxes the requirement by decomposing the graph into serveral components so that each component has a valid sub-walk order. All the join conditions not used in the walk and all the selection conditions will be verified as soon as tuples from the related tables are sampled. If the walk does not satisfy the conditions, it is treated as a failed walk and return an estimator of 0.

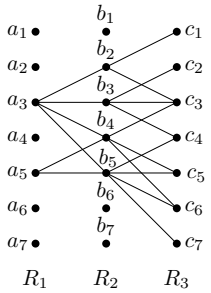


Figure 3: Structure of the join data graph has significant impact on different walk plans' performance.

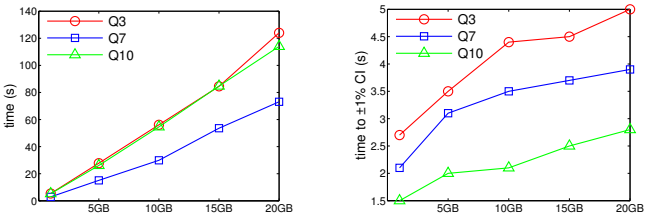
Selection conditions, structures of the join data graph and different walk orders may greatly influence the success rate of the walk, i.e. the probability that the walk does not get stuck, which in turn influences how fast the estimator converges. Considering the join data graph in Figure 3, if we perform the random walk by the order $R1$, $R2$, $R3$, then the success probability is only 2/7, but if we follow the order $R3$, $R2$, $R1$, it is 100%. Even if the success rate is the same, different non-uniformity also influences the the speed of convergence. Instead of dealing with all the issues, we observe that the performance of the random walk is measured by the variance of the final estimator after a given amount of time. To find an optimal plan, we simply generate all the promising plans. Then we run some trial samples according to each of the plan and pick the best one to continue with. Note the total number of plans could be exponential in the number of tables but it is not a real concern because 1) the number of tables is usually small (up to 8 tables in TPC-H queries) 2) the trial samples can also be combined into the final estimator so that they are not wasted.

5. EVALUATION AND DEMO PLAN

5.1 Evaluation

We compared our system against the full join in PostgreSQL and the ripple join implemented in Turbo DBO [4], an improvement to the original DBO engine that extends ripple join to data on external memory with many optimizations. We used 95% as the default confidence level for both DBO and our system.

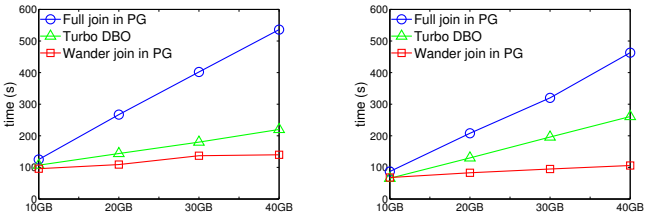
Due to the low-latency requirement for data analytical tasks and thanks to growing memory sizes, database systems are moving towards the "in-memory" computing paradigm.



(a) PostgreSQL full join (b) Wander join in PG

Figure 4: Experimental results with sufficient memory: 32GB memory.

We first tested our PostgreSQL implementation of wander join when there is sufficient memory. We used a machine with 32GB memory and data sets of sizes up to 20GB. We ran 3 queries based on Q3, Q7 and Q10 in TPC-H benchmark using wander join, full join in PostgreSQL and Turbo DBO. As shown in Figure 4, the time of full join linearly clearly grows. The data size has a mild impact on the performance of wander join because the use of B-tree indices leads to logarithmical increase in access cost when data size grows. Nevertheless, the time growth is much slower than full join and the time to 1% confidence interval (CI) outperforms full join in PostgreSQL by more than one order of magnitude. We also ran Turbo DBO in this case but it turned out to be *even slower than full join in PostgreSQL*, so we do not show its results. Partly it is because DBO is designed for large data running with very small memory.



(a) Q3 (b) Q7

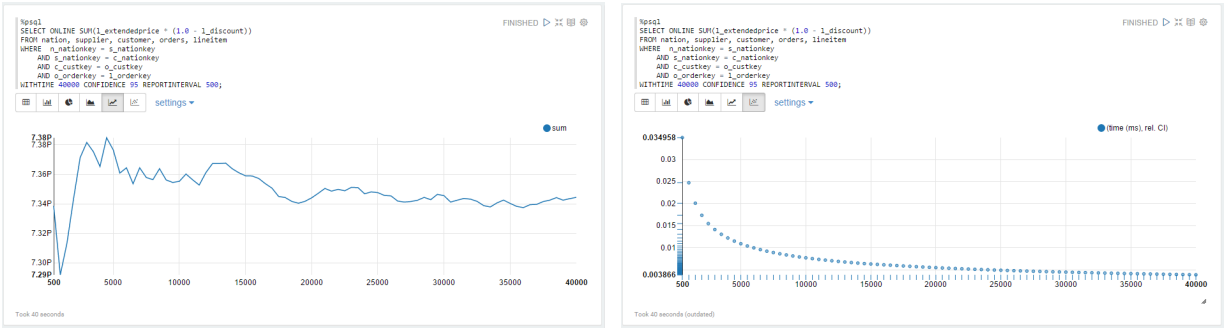
Figure 5: Experimental results with limited memory, 4GB memory (Wonder join in PG and Turbo DBO run to 1% confidence interval).

To compare our system and Turbo DBO in limited memory, we ran the queries again on a machine with only 4GB memory on data sized from 10GB to 40GB. As seen in Figure 5, our system outperforms both full join in PostgreSQL and Turbo DBO. Full join and Turbo DBO have a linear growth in time while our system's running time still grows mildly. Both DBO and our system run to 1% CI.

5.2 Demonstration Plan

We will have an end-to-end demonstration of online aggregation queries using wander join implemented in PostgreSQL over TPC-H dataset. We will also show the comparison of the performance of wander join in PostgreSQL, ripple join in DBO and full join in PostgreSQL.

Queries and dataset. The TPC-H benchmark used in our demonstration is a decision support benchmark. It features 8 tables: part, supplier, partsupp, nation, region, customer, orders, lineitem. The benchmark also features a set of business analysis queries (e.g. query of the value of goods shipped between two specific nations). We will prepare several online aggregation queries, e.g., Q3, Q7 and Q10 in TPC-H. Some queries will have multiple selection conditions and some will have groupby clause. Attendees can either use these ready-to-run queries or construct their own queries on



(a) Estimator converges over time.

(b) (Relative) Confidence interval drops over time

Figure 6: Our web interface front-end for the demonstration of wander join in PostgreSQL.

the fly to test the system. We will preload TPC-H datasets with different scale factors.

Web interface for thed demo. We have connected PostgreSQL to Apache Zeppelin, which can show the query results in the form of table, line plot and other visualization representation. Figure 6 shows our web interface. Attendees can launch both online aggregation queries and normal queries through a query input form (more forms can be created on the fly). When launching online aggregation queries to PostgreSQL, our system will use wander join to execute the query. Attendees can see how the estimator converges over time as in Figure 6(a), or plot the relative confidence interval against elapsed time as in Figure 6(b).

We will also prepare the corresponding ready-to-run DBO queries. Attendees can see a side-by-side comparison of performance among wander join in PostgreSQL, ripple join in Turbo DBO and full join in PostgreSQL.

6. RELATED WORK

The concept of online aggregation was first proposed in [8]. In particular, related to our problem, online aggregation over joins was first studied in [6], where the ripple join algorithm was designed. Extensions to ripple join were done over the years [4, 12, 13, 15], in particular, to support ripple join in DBO for large data on external memory.

In addition to these efforts, there is an increasing interest in building sampling-based approximate query processing systems, e.g., represented by systems like BlinkDB, Monte-Carlo DB, Analytical Bootstrap, DICE and others [1–3, 10, 11, 14, 16–18], but these systems do not support online aggregations over joins.

7. ACKNOWLEDGMENT

Feifei Li is supported in part by NSF grants 1443046 and 1251019. Feifei Li is also supported in part by NSFC grant 61428204 and a Google research award. Bin Wu and Ke Yi are supported by HKRGC under grants GRF-621413, GRF-16211614, and GRF-16200415.

8. REFERENCES

[1] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *SIGMOD*, pages 481–492, 2014.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[3] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and it's done: Interactive queries on very large data. In *PVLDB*, volume 5, 2012.

[4] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo charging estimate convergence in dbo. In *VLDB*, 2009.

[5] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *Proc. Ninth Intl. Conf. Scientific and Statistical Database Management*, 1997.

[6] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 287–298, 1999.

[7] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52:550–569, 1996.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1997.

[9] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47:663–685, 1952.

[10] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. The monte carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst.*, 36(3):18, 2011.

[11] P. Jayachandran, K. Tunga, N. Kamat, and A. Nandi. Combining user interaction, speculative query execution and sampling in the DICE system. *PVLDB*, 7(13):1697–1700, 2014.

[12] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2007.

[13] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. *ACM TODS*, 33(4), Article 23, 2008.

[14] A. Klein, R. Gemulla, P. Rösch, and W. Lehner. Derby/s: a DBMS for sample-based query answering. In *SIGMOD*, 2006.

[15] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2002.

[16] S. Nirkhiwale, A. Dobra, and C. M. Jermaine. A sampling algebra for aggregate estimation. *PVLDB*, 6(14):1798–1809, 2013.

[17] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. ABS: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, pages 1067–1070, 2014.

[18] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, pages 277–288, 2014.