Wander Join: Online Aggregation via Random Walks

Feifei Li [1*], Bin Wu [2], Ke Yi [2†], Zhuoyue Zhao [3]
[1]University of Utah, Salt Lake City, USA
[2]Hong Kong University of Science and Technology, Hong Kong, China
[3] Shanghai Jiao Tong University, Shanghai, China
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 users 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 even needs unrealistic assumptions (e.g., tuples in a table are stored in random order). This paper proposes a new approach, the *wander join* algorithm, to the online aggregation problem by performing random walks over the underlying join graph. We also design an optimizer that chooses the optimal plan for conducting the random walks without having to collect any statistics *a priori*. Compared with ripple join, wander join is particularly efficient for equality joins involving multiple tables, but also supports θ-joins. Selection predicates and group-by clauses can be handled as well. Extensive experiments using the TPC-H benchmark have demonstrated the superior performance of wander join over ripple join. In particular, we have integrated and tested wander join in the latest version of PostgreSQL, demonstrating its practicality in a full-fledged database system.

Keywords

Joins, online aggregation, random walks

1. INTRODUCTION

Joins are often considered as the most central operation in relational databases, as well as the most costly one. For many of today's data-driven analytical tasks, users often need to pose ad hoc complex join queries involving multiple relational tables over gigabytes or even terabytes of data. The TPC-H benchmark, which is the industrial standard for decision-support data analytics, specifies 22 queries, 17 of which are joins, the most complex one involving 8 tables. For such complex join queries, even a leading commercial database system could take hours to process. This, unfortunately, is at odds with the low-latency requirement that users demand for interactive data analytics.

The research community has long realized the need for interactive data analysis and exploration, and in 1997, initialized a line of work known as "online aggregation" [19]. The observation is that such analytical queries do not really need a 100% accurate answer. It would be more desirable if the database could first quickly return an approximate answer with some form of quality guarantee (usually in the form of confidence intervals), while improving the accuracy as more time is spent. Then the user can stop the query processing as soon as the quality is acceptable. This will significantly improve the responsiveness of the system, and at the same time saves a lot of computing resources.

Unfortunately, despite of many nice research results and well cited papers on this topic, online aggregation has had limited practical impact—we are not aware of any full-fledged, publicly available database system that supports it. The "CONTROL" project at Informix [18] in year 2000 reportedly had implemented ripple join as an internal project, prior to its acquisition by IBM. But no open source or commercial implementation of the "CONTROL" project exists today. Central to this line of work is the ripple join algorithm [15]. Its basic idea is to repeatedly take samples from each table, and only perform the join on the sampled tuples. The result is then scaled up to serve as an estimation of the whole join. However, the ripple join algorithm (including its many variants) has two critical weaknesses: (1) Its performance crucially depends on the fraction of the randomly selected tuples that could actually join. However, we observe that this fraction is often exceedingly low, especially for equality joins (a.k.a. natural joins) involving multiple tables, while *all* queries in the TPC-H benchmark (thus arguably most joins used in practice) are natural joins. (2) It demands that the tuples in each table be stored in a random order.

This paper proposes a different approach, which we call *wander join*, to the online aggregation problem. Our basic idea is to not blindly take samples from each table and just hope that they could join, but make the process much more focused. Specifically, wander join takes a randomly sampled tuple only from one of the tables. After that, it conducts a random walk starting from that tuple. In every step of the random walk, only the "neighbors" of the already sampled

*Feifei Li was supported in part by NSF grants 1251019, 1302663, and 1443046. Feifei Li was also supported in part by NSFC grant 61428204.

†Bin Wu and Ke Yi are supported by HKRGC under grants GRF-621413, GRF-16211614, and GRF-16200415.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2882903.2915235

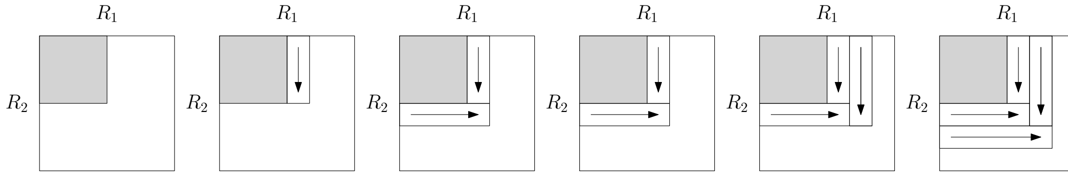


Figure 1: Illustration of the ripple join algorithm [15] on two tables R_1 and R_2.

tuples are considered, i.e., tuples in the unexplored tables that can actually join with them. Compared with the "blind search" of ripple join, this is more like a guided exploration, where we only look at portions of the data that can potentially lead to an actual join result. To summarize, we have made the following contributions:.

- We introduce a new approach called *wander join* to online aggregation for joins. The key idea is to model a join over k tables as a join graph, and then perform random walks in this graph. We show how the random walks lead to unbiased estimators for various aggregation functions, and give corresponding confidence interval formulas. We also show how this approach can handle selection and group-by clauses. These are presented in Section 3.
- It turns out that for the same join, there can be different ways to perform the random walks, which we call *walk plans*. We design an optimizer that chooses the optimal walk plan, without the need to collect any statistics of the data *a priori*. This is described in Section 4.
- We have conducted extensive experiments to compare wander join with ripple join [15] and its system implementation DBO [9,26]. The experimental setup and results are described in Section 5. The results show that wander join has outperformed ripple join and DBO by orders of magnitude in speed for achieving the same accuracy for in-memory data. When data exceeds main memory size, wander join and DBO initially have similar performance, but wander join eventually outperforms DBO on very large data sets.
- We have implemented wander join in PostgreSQL. On the TPC-H benchmark with tens of GBs of data, wander join is able to achieve 1% error with 95% confidence for most queries in a few seconds, whereas PostgreSQL may take minutes to return the exact results for the same queries.

Furthermore, we review the background of online aggregation, formulate the problem of online aggregation over joins, and summarize the ripple join algorithm in Section 2. Additional related work is surveyed in Section 6. The paper is concluded in Section 7 with remarks on a few directions for future work.

2. BACKGROUND, PROBLEM FORMULATION, AND RIPPLE JOIN

Online aggregation. The concept of online aggregation was first proposed in the classic work by Hellerstein et al. in the late 90's [19]. The idea is to provide approximate answers with error guarantees (in the form of confidence intervals) continuously during the query execution process, where the approximation quality improves gradually over time. Rather than having a user wait for the exact answer, which may take an unknown amount of time, this allows the user to explore the efficiency-accuracy tradeoff, and terminate the query execution whenever s/he is satisfied with the approximation quality.

For queries over one table, e.g., `SELECT SUM(quantity) FROM` R `WHERE discount` > 0.1, online aggregation is quite easy. The idea is to simply take samples from table R repeatedly, and compute the average of the sampled tuples (more precisely, on the value of the attribute on which the aggregation function is applied), which is then appropriately scaled up to get an unbiased estimator for the `SUM`. Standard statistical formulas can be used to estimate the confidence interval, which shrinks as more samples are taken [14].

Online aggregation for joins. For join queries, the problem becomes much harder. When we sample tuples from each table and join the sampled tuples, we get a sample of the join results. The sample mean can still serve as an unbiased estimator of the full join (after appropriate scaling), but these samples are *not* independently chosen from the full join results, even though the joining tuples are sampled from each table independently. Haas et al. [14, 16] studied this problem in depth, and derived new formulas for computing the confidence intervals for such estimators, and later proposed the *ripple join* algorithm [15]. Ripple join repeatedly takes random samples from each table in a round-robin fashion, and keep all the sampled tuples in memory. Every time a new tuple is taken from one table, it is joined with all the tuples taken from other tables so far. Figure 1 illustrates how the algorithm works on two tables, which intuitively explains why it is called "ripple" join.

There have been many variants and extensions to the basic ripple join algorithm. First, if an index is available on one of the tables, say R_2, then for a randomly sampled tuple from R_1, we can find all the tuples in R_2 that join with it. Note that no random sampling is done on R_2. This variant is also known as *index ripple join*, which was actually noted before ripple join itself was invented [32, 33]. In general, for a multi-table join $R_1 \bowtie \cdots \bowtie R_k$, the index ripple join algorithm only does random sampling on one of the tables, say R_1. Then for each tuple t sampled from R_1, it computes $t \bowtie R_2 \bowtie \cdots \bowtie R_k$, and all the joined results are returned as samples from the full join.

Problem formulation. The type of queries we aim to support is the same as in prior work on ripple join, i.e., a SQL query 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 such as `SUM`, `AVE`, `COUNT`, `VARIANCE`, and `expression` can in-

volve any attributes of the tables. The `join conditions` consist of equality or inequality conditions between pairs of the tables, and `selection predicates` can also 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 $\texttt{AGG}(\texttt{expression})$ together with a confidence interval, i.e.,

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

Here, ε is called the *half-width* of the confidence interval 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 estimate obtainable within the limit, together with a confidence interval.

3. WANDER JOIN

3.1 Wander join on a simple example

For concreteness, we first illustrate how wander join works on the natural join between 3 tables R_1, R_2, R_3:

$$R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D), \qquad (1)$$

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 $\texttt{SUM}(D)$.

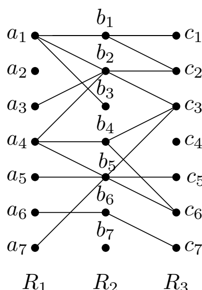


Figure 2: The 3-table join data graph: there is an edge between two tuples if they can join.

We model the join relationships among the tuples as a graph. More precisely, each tuple is modeled as a vertex and there is an edge between two tuples if they can join. For this natural join, it means that the two tuples have the same value on their common attribute. We call the resulting graph the *join data graph* (this is to be contrasted with the *join query graph* introduced later). For example, the join data graph for the 3-table natural join (1) may look like the one in Figure 2. This way, each join result becomes a path from some vertex in R_1 to some vertex in R_3, and sampling from the join boils down to sampling a path. Note that this graph is completely *conceptual*: we do not need to actually construct the graph to do path sampling.

A path can be randomly sampled by first picking a vertex in R_1 uniformly at random, and then "randomly walking" towards R_3. Specifically, in every step of the random walk, if the current vertex has d neighbors in the next table (which can be found efficiently by the index), we pick one uniformly at random to walk to.

One problem an acute reader would immediately notice is that, different paths may have different probabilities to be sampled. In the example above, the path $a_1 \to b_1 \to c_1$ has probability $\frac{1}{7} \cdot \frac{1}{3} \cdot \frac{1}{2}$ to be sampled, while $a_6 \to b_6 \to c_7$ has probability $\frac{1}{7} \cdot 1 \cdot 1$ to be sampled. If the value of the D attribute on c_7 is very large, then obviously this would tilt the balance, leading to an overestimate. Ideally, each path should be sampled with equal probability so as to ensure unbiasedness. However, it is well known that random walks in general do not yield a uniform distribution.

Fortunately, a technique known in the statistics literature as the *Horvitz-Thompson estimator* [20] can be used to remove the bias easily. Suppose path γ is sampled with probability $p(\gamma)$, and the expression on γ to be aggregated is $v(\gamma)$, then $v(\gamma)/p(\gamma)$ is an unbiased estimator of $\sum_\gamma v(\gamma)$, which is exactly the $\texttt{SUM}$ aggregate we aim to estimate. This can be easily proved by the definition of expectation, and is also very intuitive: We just penalize the paths that are sampled with higher probability proportionally. Also note that $p(\gamma)$ can be computed easily on-the-fly as the path is sampled. Suppose $\gamma = (t_1, t_2, t_3)$, where t_i is the tuple sampled from R_i, then we have

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

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

Finally, we independently perform multiple random walks, and take the average of the estimators $v(\gamma_i)/p_i$. Since each $v(\gamma_i)/p_i$ is an unbiased estimator of the $\texttt{SUM}$, their average is still unbiased, and the variance of the estimator reduces as more paths are collected. Other aggregation functions and how to compute confidence intervals will be discussed in Section 3.4.

A subtle question is what to do when the random walk gets stuck, for example, when we reach vertex b_3 in Figure 2. In this case, we should not reject the sample, but return 0 as the estimate, which will be averaged together with all the successful random walks. This is because even though this is a failed random walk, it is still in the probability space. It should be treated as a value of 0 for the Horvitz-Thompson estimator to remain unbiased. Too many failed random walks will slow down the convergence of estimation, and we will deal with the issue in Section 4.

3.2 Wander join for acyclic queries

Although the algorithm above is described on a simple 3-table *chain join*, it can be extended to arbitrary joins easily. In general, we consider the *join query graph* (or *query graph* in short), where each table is modeled as a vertex, and there is an edge between two tables if there is a join condition between the two. Figure 3 shows some possible join query graphs.

When the join query graph is acyclic, wander join can be extended in a straightforward way. First, we need to 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. For example, for the query graph in Figure 3(b), R_1, R_2, R_3, R_4, R_5 and R_2, R_3, R_4, R_5, R_1 are both valid walk orders, but R_1, R_3, R_4, R_5, R_2 is not since R_3 (resp. R_4) is not adjacent to R_1 (resp. R_1 or R_3) in the query graph. (Different walk orders may lead to very dif-

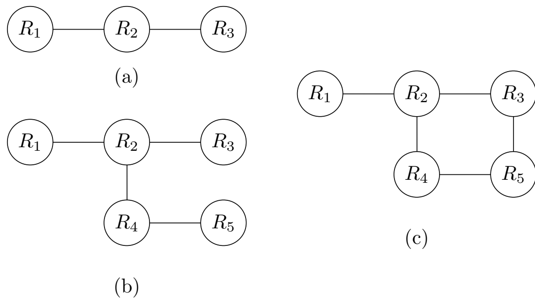


Figure 3: The join query graph for a (a) chain join; (b) acyclic join; (c) cyclic join.

ferent performances, and we will discuss how to choose the best one in Section 4.)

Next, we simply perform the random walks as before, following the given order. The only difference is that a random walk may now consist of both "walks" and "jumps". For example, using the order R_1, R_2, R_3, R_4, R_5 on Figure 3(b), after we have reached a tuple in R_3, the next table to walk to is R_4, which is connected to the part already walked via R_2. So we need to jump back to the tuple we picked in R_2, and continue the random walk from there.

Finally, we need to generalize Equation (2). Let $d_j(t)$ be the number of tuples in R_j that can join with t, where t is a tuple from another table that has a join condition with R_j. Suppose the walk order is $R_{\lambda(1)}, R_{\lambda(2)}, \ldots, R_{\lambda(k)}$, and let $R_{\eta(i)}$ be the table adjacent to $R_{\lambda(i)}$ in the query graph but appearing earlier in the order. Note that for an acyclic query graph and a valid walk order, $R_{\eta(i)}$ is uniquely defined. Then for the path $\gamma = (t_{\lambda(1)}, \ldots, t_{\lambda(k)})$, where $t_{\lambda(i)} \in R_{\lambda(i)}$, the sampling probability of the path γ is

$$p(\gamma) = \frac{1}{|R_{\lambda(1)}|} \prod_{i=2}^{k} \frac{1}{d_{\lambda(i)}(t_{\eta(i)})}. \quad (3)$$

3.3 Wander join for cyclic queries

The algorithm for acyclic queries can also be extended to handle query graph with cycles. Given a cyclic query graph, e.g., the one in Figure 3(c), we first find any spanning tree of it, such as the one in Figure 3(b). Then we just perform the random walks on this spanning tree as before. After we have sampled a path γ on the spanning tree, we need to put back the non-spanning tree edges, e.g., (R_3, R_5), and check that γ should satisfy the join conditions on these edges. For example, after we have sampled a path $\gamma = (t_1, t_2, t_3, t_4, t_5)$ on Figure 3(b) (assuming the walk order R_1, R_2, R_3, R_4, R_5), then we need to verify that γ should satisfy the non-spanning tree edge (R_3, R_5), i.e., t_3 should join with t_5. If they do not join, we consider γ as a failed random walk and return an estimator with value 0.

3.4 Estimators and confidence intervals

To derive estimators and confidence interval formulas for various aggregation functions, we establish an equivalence between wander join and sampling from a single table with selection predicates, which has been studied by Haas [14]. Imagine that we have a single table that stores all the paths in the join data graph, including both full paths, as well as partial paths (like $a_1 \to b_3$). Wander join essentially samples from this imaginary table, though non-uniformly.

Suppose we have performed a total of n random walks $\gamma_1, \ldots, \gamma_n$. For each γ_i, let $v(i)$ be the value of the expression on γ_i to be aggregated, and set $u(i) = 1/p(\gamma_i)$ if γ_i is a successful walk, and 0 otherwise. With this definition of u and v, we can rewrite the estimator for `SUM` as $\frac{1}{n}\sum_{i=1}^{n} u(i)v(i)$. We observe that this has exactly the same form as the one in [14] for estimating the `SUM` for a single table with a selection predicate, except for two differences: (1) in [14], $u(i)$ is set to 1 if γ_i satisfies the selection predicate and 0 otherwise; and (2) [14] does uniform sampling over the table, while our sampling is non-uniform. However, by going through the analysis in [14], we realize that it holds for any definition of u and v, and for any sampling distribution. Thus, all the results in [14] carry over to our case, but with u and v defined in our way. We give the estimators and confidence intervals for various estimators in Appendix A; here we just point out that any of them can be computed easily in $O(n)$ time.

3.5 Selection predicates and group-by

Wander join can deal with arbitrary selection predicates in the query easily: in the random walk process, whenever we reach a tuple t for which there is a selection predicate, we check if it satisfies the predicate, and fail the random walk immediately if not.

If the starting table of the random walk has an index on the attribute with a selection predicate, and the predicate is an equality or range condition, then we can directly sample a tuple that satisfies the condition from the index, using Olken's method [38]. Correspondingly, we replace $|R_{\lambda(1)}|$ in (3) by the number of tuples in $R_{\lambda(1)}$ that satisfy the condition, which can also be computed from the index. This removes the impact of the predicate on the performance of the random walk, thus it is preferable to start from such a table. More discussion will be devoted on this topic under walk plan optimization in Section 4.

If there is a `GROUP BY` clause in the query, the algorithm remains the same, except that each random walk path will end up in one of the groups and an estimator (and its confidence interval) is maintained for each group separately. However, this simple strategy will not work well when different groups have very different selectivities: popular groups tend to get hit by more random walks, while small groups may have few hits, leading to estimates with large confidence intervals. For queries on a single table (i.e., non-joins), there is a powerful technique to address this issue, known as *stratified sampling* [2,28]. But how to extend this technique to handling joins remains an open problem.

3.6 Justification for using indexes

Our random walk based approach crucially depends on the availability of indexes. For example, for the 3-table chain join in (1), R_2 needs to have an index on its B attribute, and R_3 needs to have an index on its C attribute. In general, a valid walk order depends on which indexes over join attributes are available. Insufficient indexing will limit the freedom of choices of random walk orders, which will be discussed in detail in Section 4.

However, we would argue that having plenty of indexes is actually a reasonable assumption: (1) Indexes can speed up general query processing tremendously. Without indexes, any query will require at least one full scan of the entire table, so indexes should have been built for any table that is queried often. (2) The main concern of not having an index

is the maintenance cost, i.e., the cost and overhead (such as locking) to update the index when new data records are inserted or deleted from the base table. But note that complex analytical (OLAP) queries, for which online aggregation is most useful, usually work in a data warehousing environment, which only sees batch updates that take place in offline time (e.g., at night). Even for online updates, new indexes are now available, such as the *fractal tree index* [5] (already implemented in MySQL and MongoDB), and recent work on adaptive and holistic indexing [12, 13, 17, 22, 41] with transaction and concurrency control support, that support such updates much more efficiently than traditional database indexes. (3) The ripple join algorithm also requires an index if tuples are to be maintained in a random order [15]. (4) When an index is not available, we could first build an index over the join attribute on the fly; building a secondary, unclustered index is usually cheaper than evaluating the join in full, say, with sort-merge joins.

3.7 Comparison with ripple join

It is interesting to note that ripple join and wander join take two "dual" approaches. Ripple join takes uniform but non-independent samples from the join, while random walks return independent but non-uniform samples. It is difficult to make a rigorous analytical comparison between the two approaches: Both sampling methods yield slower convergence compared with ideal (i.e., independent and uniform) sampling. The impact of the former depends the amount of correlation, while the latter on the degree of non-uniformity, both of which depend on actual data characteristics and the query. Thus, an empirical comparison is necessary, which will be conducted in Section 5. Here we give a simple analytical comparison in terms of *sampling efficiency*, i.e., how many samples from the join can be returned after n sampling steps, while assuming that non-independence and non-uniformity have the same impact on converting the samples to the final estimate. This comparison, although crude with many simplifying assumptions, still gives us an intuition why wander join can be much better than ripple join.

Consider a chain join between k tables, each having N tuples. Assume that, for each table $R_i, i = 1, \dots, k-1$, every tuple $t \in R_i$ joins with d tuples in R_{i+1}. Suppose that ripple join has taken n tuples randomly from each table, and correspondingly wander join has performed n random walks (successful or not).

Consider ripple join first. The probability for k randomly sampled tuples, one from each table, to join is $(\frac{d}{N})^{k-1}$. If n tuples are sampled from each table, then we would expect $n^k(\frac{d}{N})^{k-1}$ join results. Note that if the join attribute is the primary key in table R_{i+1}, we have $d_i = 1$. As a matter of fact, *all* join queries in the TPC-H benchmark, thus arguably most joins used in practice, are primary key-foreign key joins. Suppose $N = 10^6, k = 3, d = 1$, then we would need to take $n = (\frac{N}{d})^{\frac{k-1}{k}} = 10,000$ samples from each table until we get the first join result. Making things worse, this number grows with N and k.

Now let us consider wander join. In fact, under the assumption that each tuple joins with d tuples in the next table, the random walk will always be successful. In general, the efficiency of the random walks depends on the fraction of tuples in a table that have at least one joining tuple in the next table. We argue that this should not be too small. Indeed, for primary key-foreign key joins, each foreign key should have a match in the primary key table, so this fraction is 1. But if we walk from the primary key to the foreign key, this may be less than one. In general, this fraction is not too small, since if it is small, computing the join in full will be very efficient anyway, so users would not need online aggregation at all. Now we assume that this fraction is at least $1/2$ for each table. Then the success rate of a random walk is $\geq 1/2^{k-1}$, i.e., we expect to get at least $n/2^{k-1}$ samples from the join after n random walks have been performed. This leads to the most important property of our random walk based approach, that its efficiency does not depend on N, which means that it works on data of *any* scale, at least theoretically. Meanwhile, it does become worse exponentially in k. However, k is usually small; the join queries in the TPC-H benchmark on average involve 3 to 4 tables, with the largest one having 8 tables. But regardless of the value of k, wander join is better than ripple join as long as $n/2^{k-1} \geq n^k/N^{k-1}$ (assuming $d = 1$), i.e., $n/N \leq 1/2$. Note that $n/N > 1/2$ means we are sampling more than half of the database. When this happens and the confidence interval still has not reached the user's requirement, online aggregation essentially has already failed.

Computational costs. There is also a major difference in terms of computational costs. Computing the confidence intervals in ripple join requires a fairly complex algorithm with worst-case running time $O(kn^k)$ [14], due to the non-independent nature of the sampling. On the other hand, wander join returns independent samples, so computing confidence intervals is very easy, as described in Section 3.4. In fact, it should be clear that the whole algorithm, including performing random walks, computing estimators and confidence intervals, takes only $O(kn)$ time, assuming hash tables are used as indexes. If B-trees are used, there will be an extra log factor.

Run to completion. Another minor thing is that ripple join, when it completes, computes the full join exactly. Wander join can also be made to have this feature, by doing the random walks "without replacement". This will introduce additional overhead for the algorithm. A more practical solution is to simply run wander join and a traditional full join algorithm in parallel, and terminate wander join when the full join completes. Since wander join operates in the "read-only" mode on the data and indexes, it has little interference with the full join algorithm.

Worst case. Note that the fundamental lower bounds shown by Chaudhuri et al. [8] for sampling over joins apply to wander join as well. In particular, both ripple join and wander join perform badly on the hard cases constructed by Chaudhuri et al. [8] for sampling over joins. But in practice, under certain reasonable assumptions on the data (as described above and as evident from our experiments), wander join outperforms ripple join significantly.

4. WALK PLAN OPTIMIZER

Different orders to perform the random walk may lead to very different performances. This is akin to choosing the best physical plan for executing a query. So we term different ways to perform the random walks as *walk plans*. A relational database optimizer usually needs statistics to be collected from the tables *a priori*, so as to estimate various intermediate result sizes for multi-table join optimization. In

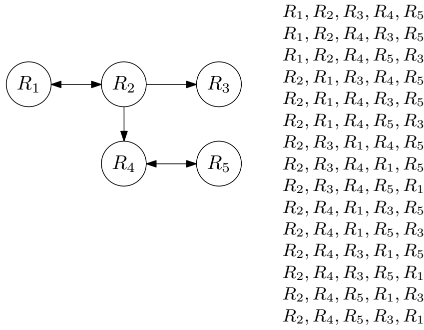


Figure 4: A directed join query graph and all its walk plans.

this section we present a walk plan optimizer that chooses the best walk plan without the need to collect statistics.

4.1 Walk plan generation

We first generate all possible walk plans. Recall that the constraint we have for a valid walk order is that for each table R_i (except the first one in the order), there must exist 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 should have an index on the attribute that appears in the join condition. Note that the join condition does not have to be equality. It can be for instance an inequality or even a range condition, such as $R_j.A \leq R_i.B \leq R_j.A + 100$, as long as R_i has an index on B that supports range queries (e.g., a B-tree).

When there is at least one valid walk order. Under the constraint above, there may or may not be a valid walk order. We first consider the case when at least one walk order exists. In this case, each walk order corresponds to a walk plan.

To generate all possible walk orders, we first add directions to each edge in the join query graph. Specifically, for an edge between R_i and R_j, if R_i has an index on its attribute in the join condition, we have a directed edge from R_j to R_i; similarly if R_j has an index on its attribute in the join condition, we have a directed edge from R_i to R_j. For example, after adding directions, the query graph in Figure 3(b) might look like the one in Figure 4, and all possible walk plans are listed on the side. These plans can be enumerated by a simple backtracking algorithm. Note that there can be exponentially (in the number of tables) many walk plans. However, this is not a real concern because (1) there cannot be too many tables, and (2) more importantly, having many walk plans does not have a major impact on the plan optimizer, which we shall see later.

We can similarly generate all possible walk plans for cyclic queries, just that some edges will not be walked, and they will have to be checked after the random walk, as described in Section 3.3. We call them *non-tree* edges, since the part of the graph that is covered by the random walk form a tree. An example is given in Figure 5.

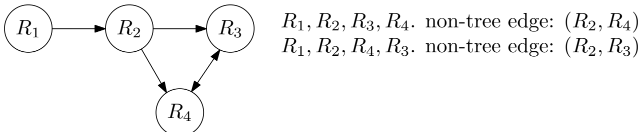


Figure 5: Walk plan generation for a cyclic query graph.

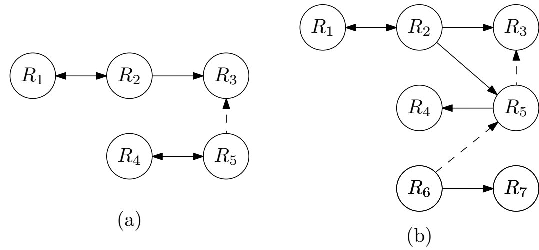


Figure 6: Decomposition of the join query graph into directed spanning trees. Dashed edges are non-tree edges.

When there is no valid walk order. The situation gets more complex when there is no valid walk order, like for the two query graphs in Figure 6 (dashed edges are also part of the query graph). First, one can easily verify that the sufficient and necessary condition for a query graph to admit at least one valid walk order is that it has a directed spanning tree[1]. When there are not enough indexes, this condition may not hold, in which case we will have to decompose the query graph into multiple components such that each component has a directed spanning tree. Figure 6 shows how the two query graphs can be decomposed, where each component is connected by solid edges.

After we have found directed spanning tree decomposition, we generate walk orders for each component, as described above. A walk plan now is any combination of the walk orders, one for each component. Then, we will run ripple join on the component level and wander join within each component. More precisely, we perform random walks for the components in a round-robin fashion, and keep all successful paths in memory. For each newly sampled path, it is joined with all the paths from other tables, i.e., checking that the join conditions on the edges between these components are met. For example, we check (R_3, R_5) in Figure 6(a) and (R_5, R_6) in 6(b). Note that (R_3, R_5) in 6(a) is checked by wander join for the component $\{R_1, R_2, R_3, R_4, R_5\}$. For every combination of the paths, one from each table, we use the HT estimator as in Section 3, except that $p(\gamma)$ is replaced by the product of the $p(\gamma_i)$'s for all that paths γ_i's involved.

Directed spanning tree decomposition. It remains to describe how to find a directed spanning tree decomposition. We would like to minimize the number of components, because each additional component pushes one more join condition from wander join to ripple join, which reduces the sampling efficiency. In the worst scenario, each vertex is in a component by itself, then the whole algorithm degrades to ripple join.

Finding the smallest directed spanning tree decomposition, unfortunately, is NP-hard (by a simple reduction from set cover). However, since the graph is usually very small (8 in the largest TPC-H benchmark query), we simply use exhaustive search to find the optimal decomposition.

For a given query graph $G = (V, E)$, the algorithm proceeds in 3 steps. In step 1, for each vertex v, we find the set of all vertices reachable from v, denoted as $T(v)$. Then, we remove $T(v)$ if it is dominated (i.e., completely contained) in another $T(v')$. For example, for the query graph in Figure 6(b), only $T(R_1) = \{R_1, R_2, R_3, R_4, R_5\}$ and

[1]A *directed tree* is a tree in which every edge points away from the root. A *directed spanning tree* of a graph G is a subgraph of G with all vertices of G, and is a directed tree.

$T(R_6) = \{R_3, R_4, R_5, R_6, R_7\}$ remain, since other $T(v)$'s are dominated by either $T(R_1)$ or $T(R_6)$. Denote the remaining set of vertices as U.

In step 2, we find the smallest subset of vertices C such that $\bigcup_{v \in C} T(v)$ covers all vertices, by exhaustively checking all subsets C of U. This gives the smallest cover, not a decomposition, since some vertices may be covered by more than one $T(v)$. For example, $T(R_1)$ and $T(R_6)$ are the optimal cover for the query graph in Figure 6(b), and they both cover R_3, R_4, R_5.

In step 3, we convert the cover into a decomposition. Denote the set of multiply covered vertices as M, and let $G_M = (M, E_V)$ be the induced subgraph of G on M. We will assign each $u \in M$ to one of its covering $T(v)$'s. However, the assignment cannot be arbitrary. It has to be *consistent*, i.e., after the assignment, all vertices assigned to $T(v)$ must form a single connected component. To do so, we first find the strongly-connected components of G_M, contract each to a "super vertex" (containing all vertices in this strongly-connected component). Then we do a topological sort of the super vertices; inside each super vertex, the vertices are ordered arbitrarily. Finally, we assign each $u \in M$ to one of its covering $T(v)$'s by this order: if u has one or more predecessors in G_M that have already been assigned, we assign u to the same $T(v)$ as one of its predecessors; otherwise u can be assigned to any of its covering $T(v)$'s. For the query graph in Figure 6(b), the topological order for M is R_5, R_3, R_4 or R_5, R_4, R_3, and in this example, we have assigned all of them to $T(R_1)$. In Appendix B, we give a proof that this algorithm produces a consistent assignment.

4.2 Walk plan optimization

We pick the best walk plan by choosing the best walk order for each component in the directed spanning tree decomposition. Below, we simply assume that the entire query graph is one component.

The performance of a walk order depends on many factors. First, it depends on the structure of the join data graph. Considering the data graph in Figure 7, if we perform the random walk by the order R_1, R_2, R_3, then the success probability is only 2/7, but if we follow the order R_3, R_2, R_1, it is 100%.

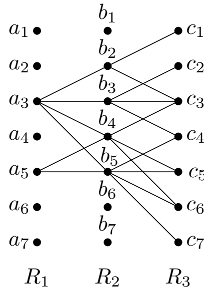


Figure 7: Structure of the join data graph has a significant impact on the performance of different walk plans.

Second, as mentioned, 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. Thirdly, for a cyclic query graph, which edges serve as the non-tree edges also affects the success probability. And finally, even if the success probability of the random walks is the same, different walk orders may result in different non-uniformity, which in turn affects how fast the variance of the estimator shrinks.

Instead of dealing with all these issues, we observe that ultimately, the performance of the random walk is measured by the variance of the final estimator after a given amount of time, say t. Let X_i be the estimator from the i-th random walk (e.g., $u(i)v(i)$ for `SUM` if the walk is successful and 0 otherwise), and let T be the running time of one random walk, successful or not. Suppose a total of W random walks have been performed within time t. Then the final estimator is $\frac{1}{W}\sum_{i=1}^{W} X_i$, and we would like to minimize its variance.

Note that W is also a random variable, we cannot just break it up as in standard variance analysis. Instead, we should do a conditioning on W, and use the *law of total variance* [36]:

$$
\begin{aligned}
&\mathsf{Var}\left[\frac{1}{W}\sum_{i=1}^{W} X_i\right] \\
=&\mathsf{E}\left[\mathsf{Var}\left[\frac{1}{W}\sum_{i=1}^{W} X_i \,\middle|\, W\right]\right] + \mathsf{Var}\left[\mathsf{E}\left[\frac{1}{W}\sum_{i=1}^{W} X_i \,\middle|\, W\right]\right] \\
=&\mathsf{E}[\mathsf{Var}[X_1]/W] + \mathsf{Var}[\mathsf{E}[X_1]] \quad //\mathsf{Var}[X_i] = \mathsf{Var}[X_j] \text{ and} \\
&\qquad\qquad //\mathsf{E}[X_i] = \mathsf{E}[X_j] \text{ for any } i, j \\
=&\mathsf{Var}[X_1]\mathsf{E}[1/W] + 0 \\
=&\mathsf{Var}[X_1]\mathsf{E}[T/t] \\
=&\mathsf{Var}[X_1]\mathsf{E}[T]/t.
\end{aligned}
$$

Thus, for a given amount of time t, the variance of the final estimator is proportional to $\mathsf{Var}[X_1]\mathsf{E}[T]$.

The next observation is that both $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$ can also be estimated by the random walks themselves! In particular, $\mathsf{Var}[X_1]$ is just estimated as in Section 3.4 and Appendix A; for $\mathsf{E}[T]$, we simply count the number of index entries looked up, or the number of I/Os in external memory, in each random walk and take the average.

Now, for each walk order, we perform a certain number of "trial" random walks and estimate $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$. Then we compute the product $\mathsf{Var}[X_1]\mathsf{E}[T]$ and pick the order with the minimum $\mathsf{Var}[X_1]\mathsf{E}[T]$. How to choose the number of trials is the classical sample size determination problem [6], which again depends on many factors such as the actual data distribution, the level of precision required, etc. However, in our case, we do not have to pick the very best plan: If two plans have similar values of $\mathsf{Var}[X_1]\mathsf{E}[T]$, their performances are close, so it does not matter which one is picked anyway. Nevertheless, we do have to make sure that, at least for the plan that is picked, its estimate for $\mathsf{Var}[X_1]\mathsf{E}[T]$ is reliable; for plans that are not picked, there is no need to determine exactly how bad they are. Thus, we adopt the following strategy: We conduct random walks following each plan in a round-robin fashion, and stop until at least one plan has accumulated at least τ successful walks. Then we pick the plan with the minimum $\mathsf{Var}[X_1]\mathsf{E}[T]$ that has at least $\tau/2$ successful walks. This is actually motivated by association rule mining, where a rule must both be good and have a minimum support level. In our implementation, we use a default threshold of $\tau = 100$.

Finally, we observe that all the trial runs are not wasted. Since each random walk, no matter which plan it follows, returns an unbiased estimator. So we can include all the random walks, before and after the optimal one has been picked, in computing the final estimator. The confidence

interval is also computed with all these random walks. This is unlike traditional query optimization, where the cost incurred by the optimizer itself is pure “overhead”.

5. EXPERIMENTS

5.1 Experimental setup

We have evaluated the performance of wander join in comparison with ripple join and its variants, the DBO engine, under two settings, which are described in detail below.

Standalone implementation. We first implemented both wander join and ripple join in C++. For wander join, data in each table is stored in primary key order in an array (using `std::vector`); for each join key, a hash table index is built (using `std::unordered_map`); for each key having a selection predicate, a binary search tree (BST) is built as the index (using `std::ordered_map`). Note that using these index structures, each step of the random walk takes $O(1)$ time and sampling from a table with a selection predicate takes $O(\log N)$ time. We ensure that all the index structures fit in memory; in fact, all the indexes combined together take space that is a very small fraction of the total amount of data, because they are all secondary indexes, storing only pointers to the actual data records, which have many other attributes that are not indexed. Furthermore, building these indexes is very efficient; in fact, they can be built with very minimum overhead while loading data from the file system to the memory (which one has to do anyway).

Similarly, for ripple join, we give it enough memory so that all samples taken can be kept in memory. For all samples taken from each table, we keep them in a hash table (also using `std::unordered_map`). Ripple join can take random samples in two ways. If the table is stored in a random order (in an array), we can simply retrieve the tuples in order. Alternatively, if an index is available, we can use the index to take a sample. The first one takes $O(1)$ time to sample a tuple and is also very cache-efficient. However, when there is a selection predicate, then the algorithm still has to read all tuples, but only keep those that satisfy the predicate. In this case, the second implementation is better (when the index is built on the selection predicate), though it takes $O(\log N)$ time to take a sample. We have implemented both versions; for the index-assisted version, indexes (BSTs) are built on all the selection predicates.

The idea for the standalone implementation is to give an ideal environment to both algorithms without any system overhead, so as to have a “barebone” comparison between the two algorithms.

System implementation. To see how the algorithms actually perform in a real database system, we have implemented wander join in the latest version of PostgreSQL (version 9.4; in particular, 9.4.2). Our implementation covers the entire pipeline from SQL parsing to plan optimization to physical execution. We build secondary B-tree indexes on all the join attributes and selection predicates.

The only system implementation available for ripple join is the DBO system [9, 25, 26]. In fact, the algorithm implemented in DBO is much more complex than the basic ripple join in order to deal with limited memory, as described in these papers. We compared wander join in our PostgreSQL implementation with Turbo DBO, using the code at http://faculty.ucmerced.edu/frusu/Projects/DBO/dbo.html, as a system-to-system comparison. Note that due to the random order storage requirement, DBO was built from ground up. Currently it is still a prototype that supports online aggregation only (i.e., no support for other major features in a RDBMS engine, such as transaction, locking, etc.). On the other hand, our integration of wander join into PostgreSQL retains the full functionality of a RDBMS, with online aggregation just as an added feature. Thus, this comparison can only be to our disadvantage due to the system overhead inside a full-fledged DBMS for supporting many other features and functionality.

Note that the original DBO papers [25] compared the DBO engine against the PostgreSQL database by running the same queries in both systems. We did exactly the same in our experiments, but simply using the PostgreSQL version with wander join implemented inside its kernel.

Data and queries. We used the TPC-H benchmark data and queries for the experiments, which were also used by the DBO work [9, 25, 26]. We used 5 tables, `nation`, `supplier`, `customer`, `orders`, and `lineitem`. We used the TPC-H data generator with the appropriate scaling factor to generate data sets of various sizes. We picked queries Q3 (3 tables), Q7 (6 tables; the `nation` table appears twice in the query) and Q10 (5 tables) in the TPC-H specification as our test queries.

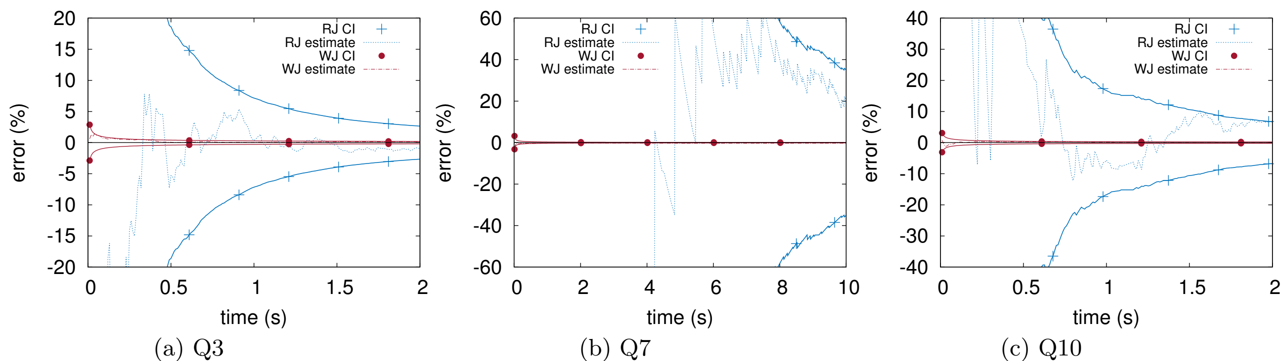
5.2 Results on standalone implementation

We first run wander join and ripple join on a 2GB data set, i.e., the entire TPC-H database is 2GB, using the “barebone” joins of Q3, Q7, and Q10, where we drop all the selection predicates and group-by clauses. In Figure 8 we plot how the *confidence interval* (CI) shrinks over time, with the confidence level set at 95%, as well as the *estimates* returned by the algorithms. They are shown as a percentage error compared with the true answer (which were obtained offline by running the exact joins to full completion). We can see that wander join (WJ) converges much faster than ripple join (RJ), due to the much more focused search strategy. Meanwhile, the estimates returned are indeed within the confidence interval almost all the time. For example, wander join converges to 1% confidence interval in less than 0.1 second whereas ripple join takes more than 4 seconds to reach 1% confidence interval. The full exact join on Q3, Q7, and Q10 in this case is 18 seconds, 28 seconds, and 19 seconds, respectively, using hash join.

Next, we ran the same queries on data sets of varying sizes. Now we include both the random order ripple join (RRJ) and the index-assisted ripple join (IRJ). For wander join, we also consider two other versions to see how the plan optimizer has worked. WJ(B) is the version where the optimal plan is used (i.e., we run the algorithm with every plan and report the best result); WJ(M) is the version where we use the median plan (i.e., we run all plans and report the median result). WJ(O) is the version where we use the optimizer to automatically choose the plan, and the time spent by the optimizer is included. In Figure 9 we report the time spent by each algorithm to reach $\pm 1\%$ confidence interval with 95% confidence level on data sets of sizes 1GB, 2GB, and 3GB. We also report the time costs of the optimizer in Table 1. From the results, we can draw the following observations:

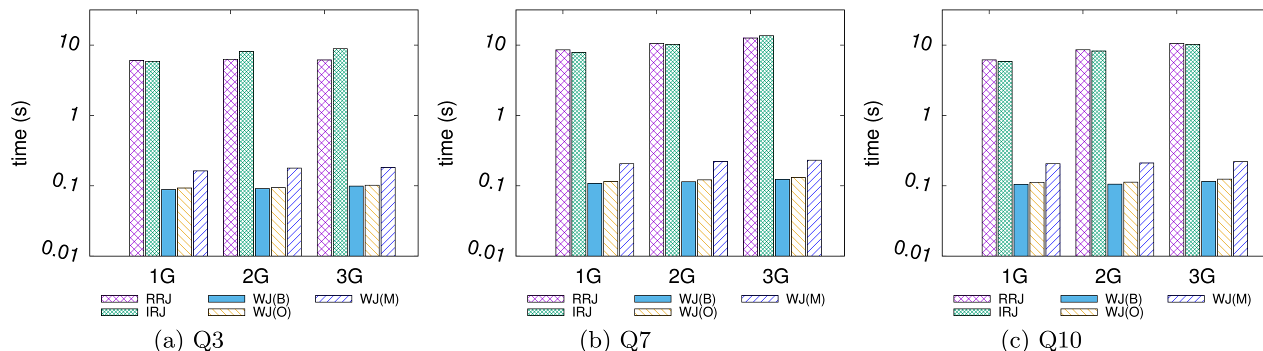
(1) Wander join is in general faster than ripple join by two orders of magnitude to reach the same confidence interval.

(2) The running time of ripple join increases with N, the



(a) Q3 (b) Q7 (c) Q10

Figure 8: Standalone implementation: Confidence intervals and estimates on barebone queries on 2GB TPC-H data set; confidence level is 95%.



(a) Q3 (b) Q7 (c) Q10

Figure 9: Standalone implementation: Time to reach $\pm 1\%$ confidence interval and 95% confidence level on TPC-H data sets of different sizes.

data size, though mildly. Recall from Section 3.7 that ripple join expects to get $n^k(\frac{d}{N})^{k-1}$ sampled join results after n tuples have been retrieved from each of the k tables. Thus, to obtain a given sample size s from the join, it needs $n = s^{1/k}(\frac{N}{d})^{(k-1)/k}$ samples from each table. This partially explains the slightly-less-than-linear growth of its running time as a function of N.

(3) The running time of wander join is not affected by N. This also agrees with our analysis: When hash tables are used, its efficiency is independent of N altogether.

(4) The optimizer has very low overhead, and is very effective. In fact, from the figures, we see that WJ(B) and WJ(O) have almost the same running time, meaning that the optimizer spends almost no time and indeed has found either the best plan or a very good plan that is almost as good as the best plan. Recall that all the trial runs used in the optimizer for selecting a good plan are not wasted; they also contribute to building the estimators. For barebone queries, many plans actually have similar performance, as seen by the running time of WJ(M), so even the trial runs are of good quality.

Finally, we put back the selection predicates to the queries. Figure 10 shows the time to reach $\pm 1\%$ confidence interval with 95% confidence level for the algorithms on the 2GB data set, with one selection predicate of varying selectivity, while Figure 11 shows the results when all the predicates are put back. Here, we measure the overall selectivity of all the predicates as:

$$1 - (\text{join size with predicates})/(\text{barebone join size}), \quad (4)$$

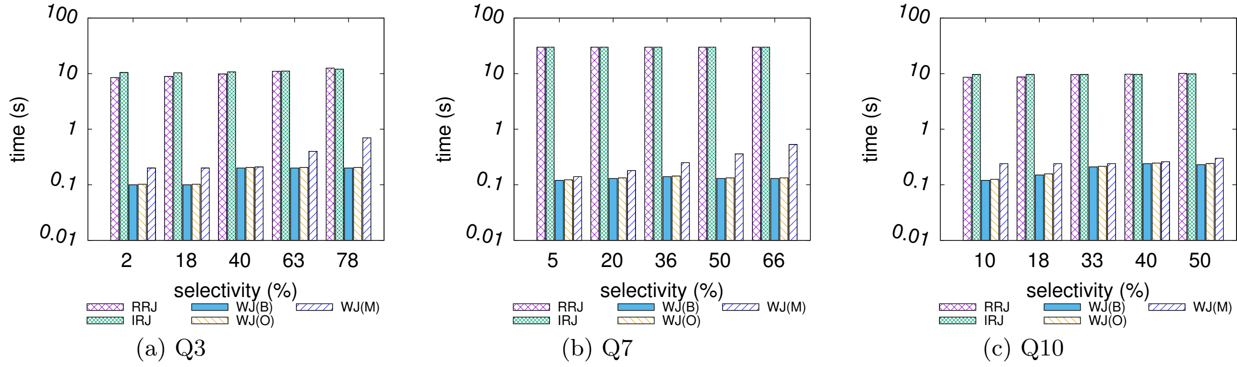
so higher means more selective.

From the results, we see that one selection predicate has little impact on the performance of wander join, because most likely its optimizer will elect to start the walk from that table. Multiple highly selective predicates do affect the performance of wander join, but even in the worst case, wander join maintains a gap with ripple join of more than an order of magnitude.

These experiments also demonstrate the importance of the plan optimizer: With multiple highly selective predicates, a mediocre plan can be much worse than the optimal one, and the plan optimizer almost always picks the optimal or a close-to-optimal plan with nearly no overhead. Note that in this case we do have poor plans, so some trial random walks may contribute little to the estimation. However, the good plans can accumulate $\tau = 100$ successful random walks very quickly, so we do not waste too much time anyway.

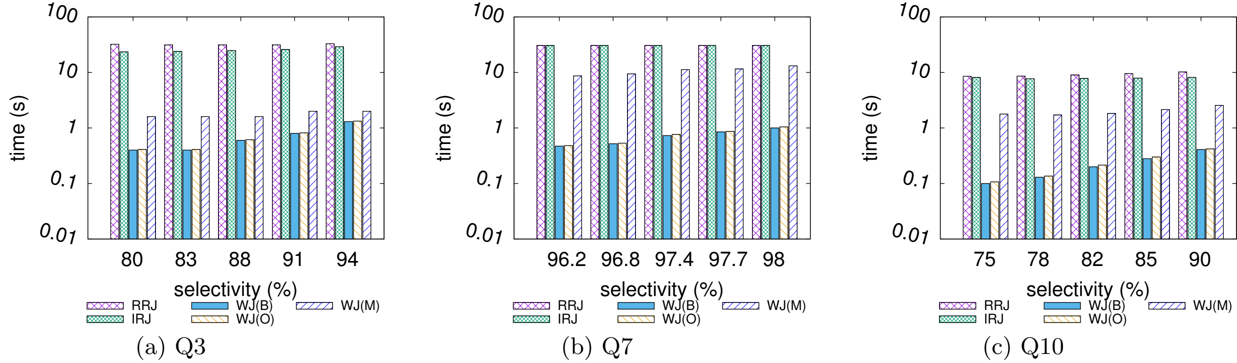
	size (GB)	optimization (ms)	execution (ms)
Q3	1	2.8	88.7
	2	2.8	91.3
	3	2.9	101.9
Q7	1	6.4	106.1
	2	6.4	112.1
	3	6.6	123.7
Q10	1	7.0	105
	2	7.3	105.6
	3	8.8	116

Table 1: Standalone implementation: Time cost of walk plan optimization (execution time to reach $\pm 1\%$ confidence interval and 95% confidence level on TPC-H data sets of different sizes).



(a) Q3 (b) Q7 (c) Q10

Figure 10: Standalone implementation: Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H data set with one selection predicate of varying selectivity.



(a) Q3 (b) Q7 (c) Q10

Figure 11: Standalone implementation: Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H data set with multiple selection predicate of varying selectivity.

5.3 Results on system implementation

For the experimental evaluation on our PostgreSQL implementation of wander join, we first tested how it performs when there is sufficient memory, and then tested the case when memory is severely limited. We compared against Turbo DBO in the latter case. Turbo DBO [9] is an improvement to the original DBO engine, that extends ripple join to data on external memory with many optimizations.

When there is sufficient memory. 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. So we first would like to see how our system performs when there is sufficient memory. For this purpose, we used a machine with 32GB memory and data sets of sizes up to 20GB. We ran both wander join (implemented inside PostgreSQL) and the built-in PostgreSQL full join on the same queries, both through the standard PostgreSQL SQL query interface. We extended PostgreSQL's parser, plan generator, and query executor to support keywords like `CONFIDENCE`, `WITHINTIME`, `REPORTINTERVAL`, and `ONLINE`. We also integrated the plan optimizer of wander join into the query optimizer of PostgreSQL. For example, an example based on Q3 of TPC-H benchmark is:

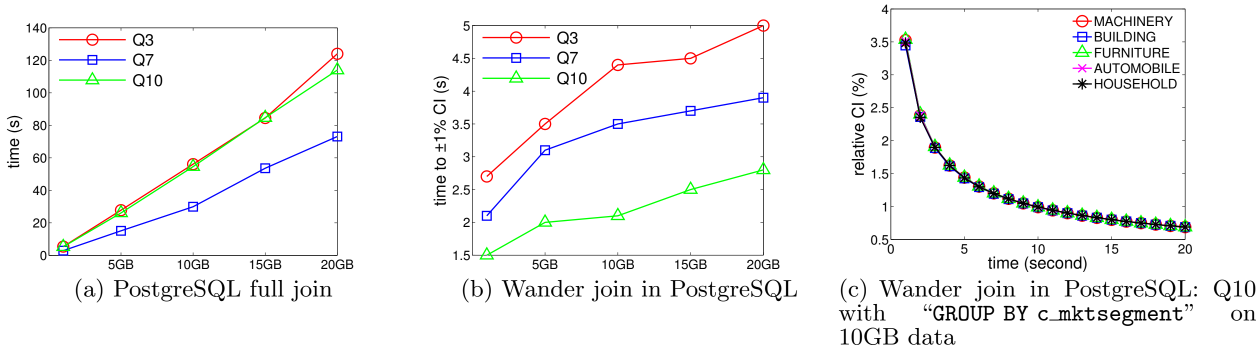
```
SELECT ONLINE
SUM(l_extendedprice * (1 - l_discount)), COUNT(*)
FROM customer, orders, lineitem
WHERE c_mktsegment='BUILDING' AND c_custkey=o_custkey
AND l_orderkey=o_orderkey
WITHINTIME 20 CONFIDENCE 95 REPORTINTERVAL 1
```

This 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, for both `SUM` and `COUNT` every 1 second for up to 20 seconds.

Note that since we have built indexes on all the join attributes and there is sufficient memory, the PostgreSQL optimizer had chosen index join for all the join operators to take advantage of the indexes. We used Q3, Q7, and Q10 with all the selection predicates, but without the group-by clause.

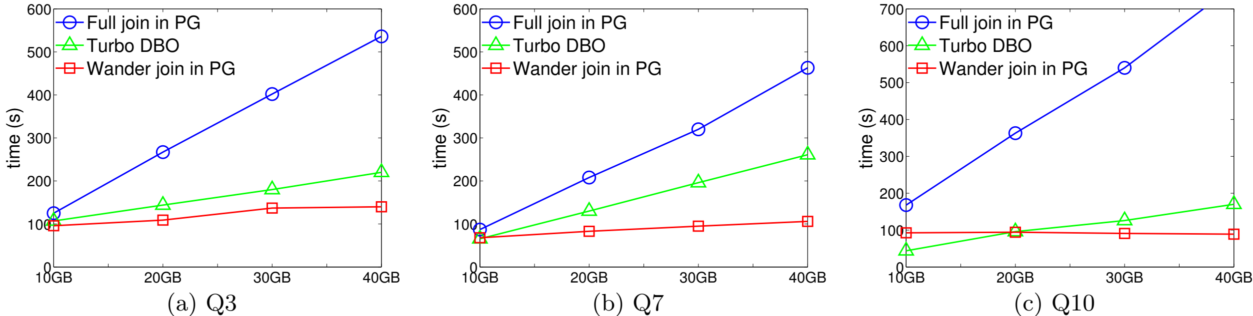
The results in Figure 12 clearly indicate a linear growth of the full join, which is as expected because the index join algorithm has running time linear in the table size. Also because all joins are primary key-foreign key joins, the intermediate results have roughly linear size. On the other hand, the data size has a mild impact on the performance of wander join. For example, the time to reach ±1% confidence interval for Q7 merely increases from 3 seconds to 4 seconds, when the data size increases from 5GB to 20GB in Figure 12(b). By our analysis and the internal memory experimental results, the total number of random walk steps should be independent of the data size. However, because we use B-tree indexes, whose access cost grows logarithmically as data gets larger, so the cost per random walk step might grow slightly. In addition, on larger data sets, the CPU cache may not be as effective as on smaller data sets. These system reasons might have explained the small performance drop of wander join on larger data sets. Nevertheless, PostgreSQL with wander join reaching 1% CI has outperformed the PostgreSQL with full join by more than one order of magnitude when data size grows.

We have also run Turbo DBO in this case. However, it



(a) PostgreSQL full join (b) Wander join in PostgreSQL (c) Wander join in PostgreSQL: Q10 with "GROUP BY c_mktsegment" on 10GB data

Figure 12: System implementation experimental results with sufficient memory: 32GB memory.



(a) Q3 (b) Q7 (c) Q10

Figure 13: System implementation experimental results with limited memory, 4GB memory.

turned out that Turbo DBO spends even more time than PostgreSQL's full join, so we do not show its results. This seems to contradict with the results in [26]. In fact, this is because DBO is intentionally designed for large data and small memory. In the experiments of [26], the machine used had only 2GB of memory. With such a small memory, PostgreSQL had to resort to sort-merge join or nested-loop join for each join operator, which is much less efficient than index join (for in-memory data). Meanwhile, DBO follows the framework of sort-merge join, so it is actually not surprising that it is not as good as index joins for in-memory data. In our next set of experiments where we limit the memory size, we do see that DBO performs better than the full join.

We also tested wander join with Q10 with a "`GROUP BY c_mktsegment`" clause. The confidence intervals as time goes on for each group are plotted in Figure 12(c). In the data set, data is evenly distributed among the groups defined by `c_mktsegment`, so the confidence intervals of all the groups reduce at the rate.

When memory is limited. In our last set of experiments, we used a machine with only 4GB memory, and ran the same set of experiments as above on data sets of sizes starting from 10GB and increasing to 40GB. The time for wander join inside PostgreSQL and Turbo DBO to reach $\pm 5\%$ confidence interval with 95% confidence level, as well as the time of the full join in PostgreSQL, are shown in Figure 13.

From the results, we see that a small memory has a significant impact on the performance of wander join. The running time increases from a few seconds in Figure 12 to more than 100 seconds in Figure 13, and that's after we have relaxed the target confidence interval from $\pm 1\%$ to $\pm 5\%$. The reason is obviously due to the random access nature of the random walks, which now has a high cost due to excessive page swapping. Nevertheless, this is a "one-time" cost, in the sense that each random walk step is now much more expensive, but the number of steps is still not affected. After the one-time, sudden increase when data size exceeds main memory, the total cost remains almost flat afterward. In other words, the cost of wander join in this case is still independent of the data size, albeit to a small increase in the index accessing cost (which grows logarithmically with the data size if B-tree is used). Hence, wander join still enjoys excellent scalability as data size continues to grow.

On the other hand, both the full join and DBO clearly have a linear dependency on the data size, though at different rates. On the 10GB and 20GB data sets, wander join and DBO have similar performance, but eventually wander join would stand out on very large data sets.

Anyway, spending 100 seconds just to get a $\pm 5\%$ estimate does not really meet the requirement of interactive data analytics, so strictly speaking both wander join and DBO have failed in this case (when data has significantly exceeded the memory size). Fundamentally, online aggregation requires some form of randomness so as to have a statistically meaningful estimation, which is at odds with the sequential access nature of hard disks. This appears to be an inherent barrier for this line of work. However, as memory sizes grow larger and memory clouds get more popular (for example, using systems like RAMCloud [39] and FaRM [10]), with the SSDs as an additional storage layer, in the end we may not have to deal with this barrier at all. What's more, as shown in Figure 13, wander join (and DBO) still shows much better latency (for an acceptable confidence interval like 5%) than the full join, and the gap only becomes larger as data size continues to grow. So it is still very useful to have online aggregation over joins as a valuable tool available for data analysts.

Effectiveness of walk plan optimization. In the standalone implementation, we have observed that the walk plan optimizer has low overhead and can generate walk plans

much better than the median plan. Similarly, we conducted experiments with our PostgreSQL implementation of wander join to see the effectiveness and the overhead of the walk plan optimizer, with either sufficient memory or limited memory. The results are shown in Table 2 (in Appendix C). In Table 2, instead of reporting the time of a median plan, we used the plan as constructed from the input query and used by PostgreSQL. From the results, we see that with sufficient memory, the results are similar to those on the standalone implementation, namely, there is very little overhead in the walk plan optimization. With limited memory, the optimizer tends to spend more time, due to system overhead and the page faults incurred by the round-robin exploration. But the total time (walk plan optimization + plan execution) is not much more expensive than the best plan execution itself, and is still much better than the plan used by PostgreSQL.

In summary, we see that in all cases, the optimizer can pick a plan that is much better than the plan generated from the input query and used by PostgreSQL. And generally speaking, query optimizer in a database engine tries to optimize the full join, not online aggregation. That's the value of having our own walk plan optimizer for wander join, and our walk plan optimization is both very effective and very efficient.

Comparing with a commercial-level database system. Finally, to gauge how our PostgreSQL (PG) implementation of wander join performs in comparison to a commercial-level database system, we ran the queries (in full) on System X[2], and then see how much accuracy our PG (with wander join) and DBO can achieve with 1/10 of the System X's full query time for the same query. System X uses the same machine and builds the same indexes as PG with wander join does.

We ran these experiments on both sufficient memory and limited memory for TPC-H data of different size (from 10GB to 40GB), using Q3, Q7, and Q10. The results are reported in Table 3 (in Appendix C). These results clearly demonstrate the benefits of wander join in getting high-quality approximate results in just a fraction of the time needed to get the accurate result, even when compared to state-of-the-art commercial-level database systems. Note that in many cases, DBO did not return any results in the time given, which is consistent with previously reported results, that DBO usually starts to return results after a few minutes [9, 26].

6. RELATED WORK

The concept of online aggregation was first proposed in [19], and since then has generated much follow-up work, including the efforts in extending it to distributed and parallel environments [40, 42, 43, 47, 50] and multiple queries [48]; a complete survey of these work is out of the scope of this paper. In particular, related to our problem, online aggregation over joins was first studied in [15], where the ripple join algorithm was designed. Extensions to ripple join were done over the years [9, 25, 26, 35], in particular, to support ripple join in DBO for large data on external memory. Note that we have already reviewed the core ideas in online aggregation and ripple join in Section 2.

Online aggregation is closely related to another line of work known as *query sampling* [8, 21, 38, 45]. In online aggregation, the user is only interested in obtaining an aggregate, such as `SUM` or `AVE`, on a particular attribute of all the query results. However, a single aggregate may not be expressive enough to represent sufficient properties of the data, so the user may require a random sample, taken uniformly and independently, from the complete set of query results that satisfy the input query conditions. Note that query sampling immediately solves the online aggregation problem, as the aggregate can be easily computed from the samples. But this may be an overkill. In fact, both 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. Nevertheless, query sampling has received separate attention, as a uniform and independent sample can serve more purposes than just computing an aggregate, including many advanced data mining tasks; in some cases, the user may just want the sample itself.

In addition to these efforts, there are also extensive work on using sampling for approximate query processing, selectivity estimation, and query optimization [1, 2, 7, 11, 27–30, 37, 44, 46, 49, 51, 52]. In partiluar, 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, 23, 24, 31, 37, 51, 52], but these systems do not support online aggregations over joins.

7. FUTURE DIRECTIONS

This work has presented some promising results on wander join, a new approach to online aggregation for joins. Yet, it has a lot of potential to be further exploited. Here we list a few directions for future work:

- Wander join is an "embarrassingly parallel" algorithm, and it should be very easy to implement it on a multi-core machine or a cluster. In particular, we are working on integrating wander join with SparkSQL [4]. Because wander join works much better for in-memory data than data from external memory on hard disks, Spark's massively parallel, in-memory computing framework provides an ideal platform for wander join. Recent efforts on extending traditional online aggregation techniques to Spark SQL in systems like G-OLA have already shown promising results [50].
- Because wander join can estimate `COUNT` very quickly, we can run wander join on any sub-join and estimate the intermediate join size. This in turn provides important statistics to a traditional cost-based query optimizer. It would be interesting to see if this can lead to improved query plan selection for full join computation.
- When the query has a group-by clause and the groups are highly unbalanced, some groups might be under-sampled. This problem can be in general solved by *stratified sampling* [1, 2, 34, 50, 52]. If the group-by attributes are from a single table, wander join can easily handle by simply starting the random walks from that table, but the problem is more complicated when the group-by involves attributes from different tables, which deserves further investigation.

[2] Legal restrictions prevent us from revealing the actual vendor name.

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 *Proceedings of the VLDB Endowment*, volume 5, 2012.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2015.

[5] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.

[6] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, 2001.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: how much is enough? In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.

[8] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1999.

[9] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo charging estimate convergence in dbo. In *Proc. International Conference on Very Large Data Bases*, 2009.

[10] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[11] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.

[12] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

[13] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.

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

[15] 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.

[16] 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.

[17] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[18] J. M. Hellerstein, R. Avnur, and V. Raman. Informix under control: Online query processing. *Data Min. Knowl. Discov.*, 4(4):281–314, 2000.

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

[20] 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.

[21] X. Hu, M. Qiao, and Y. Tao. Independent range sampling. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.

[22] S. Idreos, S. Manegold, and G. Graefe. Adaptive indexing in modern database kernels. In *EDBT*, pages 566–569, 2012.

[23] 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.

[24] 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.

[25] 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.

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

[27] R. Jin, L. Glimcher, C. Jermaine, and G. Agrawal. New sampling-based estimators for OLAP queries. In *ICDE*, page 18, 2006.

[28] S. Joshi and C. M. Jermaine. Robust stratified sampling plans for low selectivity queries. In *ICDE*, pages 199–208, 2008.

[29] S. Joshi and C. M. Jermaine. Sampling-based estimators for subset-based queries. *VLDB J.*, 18(1):181–202, 2009.

[30] A. Kim, E. Blais, A. G. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *PVLDB*, 8(5):521–532, 2015.

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

[32] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *Proc. ACM Symposium on Principles of Database Systems*, 1990.

[33] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1990.

[34] S. Lohr. *Sampling: Design and analysis*. Thomson, 2009.

[35] 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.

[36] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

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

[38] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[39] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, 2011.

[40] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. In *Proceedings of the VLDB Endowment*, volume 4, 2011.

[41] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, pages 1153–1166, 2015.

[42] C. Qin and F. Rusu. Parallel online aggregation in action. In *SSDBM*, pages 46:1–46:4, 2013.

[43] C. Qin and F. Rusu. PF-OLA: a high-performance framework for parallel online aggregation. *Distributed and Parallel Databases*, 32(3):337–375, 2014.

[44] D. Vengerov, A. C. Menck, and M. Zait. Join size

estimation subject to filter conditions. In *Proc. International Conference on Very Large Data Bases*, 2015.

[45] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. In *Proc. International Conference on Very Large Data Bases*, 2016.

[46] M. Wu and C. Jermaine. Outlier detection by sampling with accuracy guarantees. In *SIGKDD*, pages 767–772, 2006.

[47] S. Wu, S. Jiang, B. C. Ooi, and K. Tan. Distributed online aggregation. *PVLDB*, 2(1):443–454, 2009.

[48] S. Wu, B. C. Ooi, and K. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, pages 651–662, 2010.

[49] F. Xu, C. M. Jermaine, and A. Dobra. Confidence bounds for sampling-based group by estimates. *ACM Trans. Database Syst.*, 33(3), 2008.

[50] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, pages 913–918, 2015.

[51] 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.

[52] 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.

APPENDIX

A. ESTIMATORS AND CONFIDENCE INTERVALS

For any function $f, h : \mathbb{N} \to \mathbb{R}$, we introduce the following notation:

$$T_n(f) = \frac{1}{n}\sum_{i=1}^{n} f(i),$$

$$T_{n,q}(f) = \frac{1}{n-1}\sum_{i=1}^{n}(f(i) - T_n(f))^q,$$

$$T_{n,q,r}(f,h) = \frac{1}{n-1}\sum_{i=1}^{n}(f(i) - T_n(f))^q (h(i) - T_n(h))^r.$$

The estimators for a few common aggregation functions, as well as estimators for their variances, are given by:

$\texttt{SUM} : \tilde{Y}_n = T_n(uv), \tilde{\sigma}_n^2 = T_{n,2}(uv);$

$\texttt{COUNT} : \tilde{Y}_n = T_n(u), \tilde{\sigma}_n^2 = T_{n,2}(u);$

$\texttt{AVE} : \tilde{Y}_n = T_n(uv)/T_n(u), \tilde{\sigma}_n^2 =$

$$\frac{1}{T_n^2(u)}\left(T_{n,2}(uv) - 2R_{n,2}T_{n,1,1}(uv,u) + R_{n,2}^2 T_{n,2}(u)\right),$$

where $R_{n,2} = T_n(uv)/T_n(u)$.

Please see [14] for formulas for `VARIANCE` and `STDEV`.

Finally, after we have obtained an estimate $\tilde{\sigma}_{n(t)}^2$ for the variance of an estimator $\tilde{Y}$, the half-width of the confidence interval is computed as (for a confidence level threshold α)

$$\varepsilon_n = \frac{z_\alpha \tilde{\sigma}_n}{\sqrt{n}}, \tag{5}$$

where z_α is the $\frac{\alpha+1}{2}$-quantile of the normal distribution with mean 0 and variance 1.

B. ADDITIONAL PROOFS

Lemma 1 *The algorithm produces a consistent assignment.*

PROOF. Suppose that after the assignment, some $T(v)$ is disconnected. Then there must be a $u \in T(v) \cap M$ such that all its predecessors in $T(v)$ have been assigned to other $T(v')$'s, but u remains in $T(v)$. If any of u's predecessors is assigned before u, then the algorithm cannot have assigned u to $T(v)$. If all of u's predecessors are assigned after u, then they must be in the same strongly-connected component as u, and u does not have other predecessors in M. This means that u is directed connected to $T(v) \setminus M$. □

C. ADDITIONAL TABLES

Table 2 (see next page) presents the results on the effectiveness and overhead of walk plan optimization for wander join in PostgreSQL. Table 3 (see next page) presents our PostgreSQL implementation of wander join against both DBO and a commercial database system (denoted as System X). For results in both Tables 2 and 3, we investigated both sufficient memory and limit memory scenarios. The results of which have been discussed in the end of Section 5.

	SF[1]	sufficient memory					limited memory				
		total time[2]	optimized plan[3]		PG plan[4]		total time	optimized plan		PG plan	
			time	AE[5]	time	AE		time	AE	time	AE
Q3	10	9.24	9.03	0.26	20.2	0.09	330	323	0.95	556	2.91
	20	11.27	11.05	0.39	21.4	0.34	390	382	0.43	673	0.05
	30	11.28	11.03	0.23	21.5	0.37	429	422	1.22	702	0.49
	40	11.38	11.04	0.30	21.5	0.27	455	447	3.63	706	0.56
Q7	10	9.18	9.02	0.19	51.6	0.03	126	90	1.23	937	0.45
	20	9.32	9.22	0.42	56.4	0.07	188	131	3.90	1359	1.72
	30	8.47	8.17	0.28	60.5	0.11	215	145	1.67	1613	0.53
	40	8.56	8.27	1.16	60.6	0.30	230	157	1.00	1742	0.71
Q10	10	3.23	3.05	1.19	3.12	0.11	71	61	0.31	95	0.43
	20	3.33	3.17	0.39	4.34	0.02	96	79	0.51	118	1.03
	30	3.35	3.06	0.45	5.54	0.84	107	90	2.22	125	1.17
	40	4.22	4.07	0.06	7.87	0.76	111	93	2.48	134	0.97

Table 2: PostgreSQL with wander join: time cost of walk plan optimization and total execution time (and the actual error achieved).

[1] SF: scale factor (GB).
[2] total time: the total wall clock time for walk plan optimization and plan execution to reach the target confidence interval (CI) with 95% confidence level. The target CI is 1% for sufficient memory and 5% for limited memory.
[3] optimized plan: time taken and actual error achieved to reach the target CI by directly using the best plan selected by wander join's query optimizer (i.e., the plan execution time from the total time).
[4] PG plan: time taken and actual error achieved to reach the target CI by using the plan constructed from the input query and used by PostgreSQL.
[5] AE: actual error (%).

	SF[1]	sufficient memory					limited memory				
		System X[2]	DBO		PG+WJ[5]		System X	DBO		PG+WJ	
			CI[3]	AE[4]	CI	AE		CI	AE	CI	AE
Q3	10	32.24	–	–	1.18	0.09	107.27	–	–	15.9	7.8
	20	74.29	–	–	0.78	0.43	249.94	–	–	11.1	4.3
	30	65.17	–	–	0.84	0.40	428.39	–	–	9.6	4.5
	40	90.23	–	–	0.76	0.26	707.04	48.50	30.60	8.1	4.7
Q7	10	33.62	–	–	1.15	0.24	103.3	–	–	15.1	4.1
	20	73.03	–	–	0.70	0.35	205.7	–	–	11.2	3.4
	30	57.82	–	–	0.79	0.35	326.35	–	–	9.6	1.7
	40	77.92	–	–	0.69	0.05	445.86	–	–	8	0.3
Q10	10	40.43	–	–	0.75	0.01	146.57	47.71	23.24	13.7	1.2
	20	98.96	82.06	21.93	0.47	0.02	326.67	35.62	14.60	8.7	2.1
	30	109.19	138.29	66.50	0.46	0.05	697.06	26.43	6.69	6.9	1.3
	40	138.87	97.68	11.99	0.42	0.06	829.97	11.31	1.32	5.2	0.5

Table 3: Accuracy achieved in 1/10 **of System X's running time for computing the full join.**

[1] SF: scale factor (GB).
[2] System X: full join time on System X (seconds).
[3] CI: half width of the confidence interval (%).
[4] AE: actual error (%).
[5] PG+WJ: Our version of PostgreSQL with Wander Join implemented inside the PostgreSQL engine.
–: no result reported in the time given.