Wander Join and XDB: Online Aggregation via Random Walks

FEIFEI LI, University of Utah, USA
BIN WU and KE YI, Hong Kong University of Science and Technology, China
ZHUOYUE ZHAO, University of Utah, USA

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 article 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. To demonstrate the usefulness of wander join, we have designed and implemented XDB (approXimate DB) by integrating wander join into various systems including PostgreSQL, Spark, and a stand-alone plug-in version using PL/SQL. The design and implementation of XDB has demonstrated wander join's practicality in a full-fledged database system. Extensive experiments using the TPC-H benchmark have demonstrated the superior performance of wander join over ripple join.

CCS Concepts: • **Information systems → Database query processing**; **Join algorithms**; **Online analytical processing**; • **Theory of computation → Random walks and Markov chains**; **Database query processing and optimization (theory)**;

Additional Key Words and Phrases: Join, online aggregation, random walk

ACM Reference format:
Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander Join and XDB: Online Aggregation via Random Walks. *ACM Trans. Database Syst.* 44, 1, Article 2 (January 2019), 41 pages.
https://doi.org/10.1145/3284551

This is an extended version of the paper entitled Wander Join: Online Aggregation via Random Walks, published in SIGMOD'16, ISBN 978-1-4503-3531-7/16/06. DOI: http://dx.doi.org/10.1145/2882903.2915235.
Feifei Li and Zhuoyue Zhao are supported in part by NSF grants 1251019, 1302663, 1443046 and NSFC grant 61428204. Bin Wu and Ke Yi are supported by HKRGC under grants GRF-621413, GRF-16211614, and GRF-16200415.
Authors' addresses: F. Li, University of Utah, School of Computing, 50 S. Central Campus Drive, Salt Lake City, Utah, USA, 84112; email: lifeifei@cs.utah.edu; B. Wu, Hong Kong University of Science and Technology, Department of Computer Science and Engineering, Clear Water Bay, Hong Kong, China; email: bwuac@cse.ust.hk; K. Yi, Hong Kong University of Science and Technology, Department of Computer Science and Engineering, Clear Water Bay, Hong Kong, China; email: yike@cse.ust.hk; Z. Zhao, University of Utah, School of Computing, 50 S. Central Campus Drive, Salt Lake City, Utah, USA, 84112; email: zyzhao@cs.utah.edu.

0362-5915/2019/01-ART2 $15.00
https://doi.org/10.1145/3284551

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 eight 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" [23]. 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 save a lot of computing resources.

Unfortunately, despite the 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 [22] in the year 2000 reportedly had an implementation as an internal project at Informix, 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 [19]. 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 requires drastic changes to the database engine, which is perhaps one of the main reasons why ripple join has not seen wide adoption in real systems.

This article 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 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. This results in a significant performance improvement. Moreover, wander join can be implemented without modifying the core database engine, including the storage format and transaction processing units, which is another important advantage of wander join over previous approaches.

To summarize, we have made the following contributions:

(1) 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.

(2) 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.

(3) We introduce a number of optimizations that are designed to further improve the performance of wander join with respect to selection predicates and the Group-By clause.

(4) We have developed the XDB system (approXimate DB) by implementing wander join inside the kernel of PostgreSQL. We also extend the system integration of XDB to Spark, and a plug-in version using PL/SQL is also introduced so that no kernel updates are needed to implement XDB. The details are presented in Section 5. On the TPC-H benchmark with tens of GBs of data, The PostgreSQL version of XDB with 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.

(5) We have conducted extensive experiments to compare wander join with ripple join [19] and XDB with ripple join's system implementation TurboDBO [11, 30]. The experimental setup and results are described in Section 6. The results show that wander join and XDB has outperformed ripple join and TurboDBO by orders of magnitude in speed for achieving the same accuracy for in-memory data. When data exceeds main memory size, XDB and TurboDBO initially have similar performance, but XDB eventually outperforms TurboDBO on very large datasets.

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 8. The article is concluded in Section 9 with remarks on a few directions for future work.

1.1 Comparison with the Conference Version

Compared with the conference version, this article contains the following improvements and extensions:

(1) In the conference version, we used aggregate B^+-trees for randomly sampling a neighbor. We have replaced it by a standard B^+-tree, so that we can now implement wander join without modifying the storage engine and transaction processing at all (Section 3.2).

(2) We give improved methods for dealing with `GROUPBY` clauses, which make sure that all groups, even smaller ones, are estimated well (Sections 3.4 and 3.5).

(3) We give a more detailed derivation of the estimators and confidence interval formulas for various aggregation functions (Section 3.5).

(4) We introduce two optimization techniques to further improve wander join's practical performance (Section 3.6).

(5) We describe how the trial runs used by the optimizer can also be included into the overall estimator to further reduce its variance (Section 4.2.2).

(6) We design and implement the XDB engine by integrating wander join into the kernel of the latest version of PostgreSQL. The design choices and impacts to the PostgreSQL kernel are described in Section 5.1. It is also open sourced at https://github.com/InitialDLab/XDB and https://github.com/InitialDLab/zeponline.

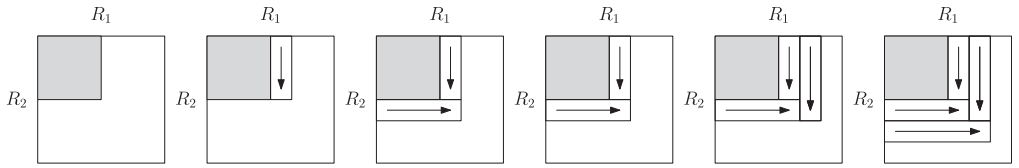


Fig. 1. Illustration of the ripple join algorithm on two tables R_1 and R_2.

(7) We have also extended XDB to Spark, a popular main memory based massively parallel data analytics engine designed to scale (Section 5.2).
(8) Finally, we have shown how wander join can also be implemented outside a database engine in PL/SQL to provide a plug-in version of XDB that is able to work with any popular database system without the need of updating its kernel (Section 5.3).
(9) We have re-run all the experiments in Sections 6 and 7 to incorporate the improvements and extensions introduced above.

2 BACKGROUND, PROBLEM FORMULATION, AND RIPPLE JOIN

2.1 Online Aggregation

The concept of online aggregation was first proposed in the classic work by Hellerstein et al. in the late 1990's [23]. 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 [18].

2.2 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. [18, 20] studied this problem in depth, and derived new formulas for computing the confidence intervals for such estimators, and later proposed the *ripple join* algorithm [19]. Ripple join repeatedly takes random samples from each table in a round-robin fashion, and keeps 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 [37, 38]. 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.

2.3 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 involve any attributes of the tables. The join conditions consist of equality, inequality, or range conditions between pairs of the tables, and selection predicates can also be applied to any subset of the tables. While we allow the join conditions and selection predicates to be unknown until query time, we do require that the attributes involved in the join conditions be given in advance. As we will see later, our solution depends on the availability of indexes on these join attributes. In practice, most join conditions are between a primary key and a foreign key, and such constraints are part of the database schema which are known *a priori*.

At any point in time during query processing, the algorithm should output an estimator $\tilde{Y}$ for AGG(expression) together with ε, α such that

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

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 three 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, and so forth. The natural join returns all combinations of tuples from the three 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 SUM(D).

Our algorithm is a Monte Carlo algorithm based on random sampling. 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 three-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

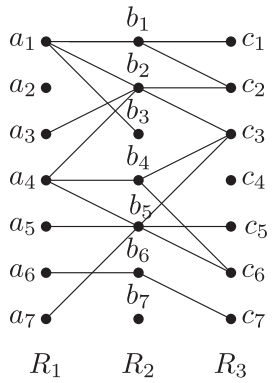


Fig. 2. The three-table join data graph: there is an edge between two tuples if they can join.

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" toward 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* [24] 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 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

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

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 $v(\gamma_i)/p(\gamma_i)$. Since each $v(\gamma_i)/p(\gamma_i)$ is an unbiased estimator of the 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 3.5.

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. This holds for all the aggregation functions defined in Section 2.3. Too many failed random walks will slow down the convergence of estimation, and we will deal with the issue in Section 4.

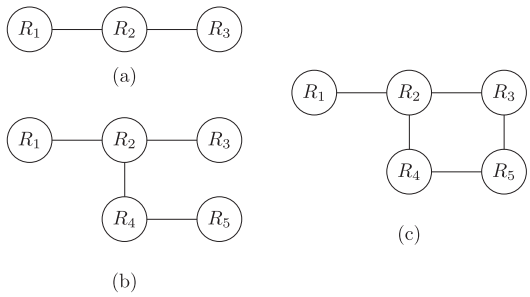


Fig. 3. The join query graph for (a) a chain join; (b) an acyclic join; (c) a cyclic join.

3.2 Wander Join for Acyclic Queries

We are now ready to formally describe the wander join algorithm on a join query on k tables. We model a join query as a *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.

In this section, we first consider the case when the join query graph is acyclic. First, we 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 (R_4, respectively) is not adjacent to R_1 (R_1 or R_3, respectively) in the query graph. Different walk orders may lead to very different performances, and we will discuss how to choose the best one in Section 4.

Next, we perform the random walks following the given order, in a similar way as on the three table join in Section 3.1. There are two differences, though. The first 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 in 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.

Secondly, on the simple three-table join example, we have assumed that we can randomly sample a neighbor of a tuple. In the conference version of the article, we used an aggregate B-tree for this purpose, i.e., at each internal node v of the B-tree, we add an additional field that keeps track of the total number of tuples stored below v. This way, we can always sample a neighbor of a tuple uniformly. However, in real systems, aggregate B-trees are seldom used, due to its high overhead when performing updates: When a tuple is to be inserted or deleted, the extra fields at all its ancestor nodes have to be updated. This incurs high cost and complicates concurrency control.

Our new observation is that the HT estimator still works even if each step of the random walk γ is not uniform, as long as the its probability $p(\gamma)$ can be computed. 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

$$p(\gamma) = \frac{1}{|R_{\lambda(1)}|} \prod_{i=2}^{k} \Pr[t_{\eta(i)} \to t_{\lambda(i)}], \tag{3}$$

where $\Pr[t_{\eta(i)} \to t_{\lambda(i)}]$ denotes the probability that we walk from $t_{\eta(i)}$ to $t_{\lambda(i)}$. We observe that this probability can be computed as we walk down the B-tree.

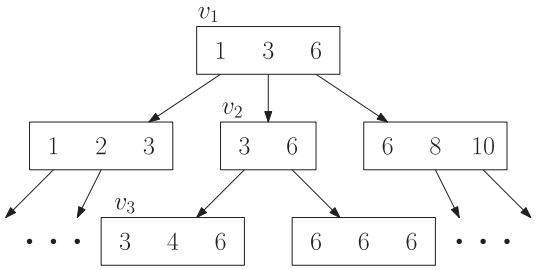


Fig. 4. Walking down a B-tree while computing the probability $\Pr[t_{\eta(i)} \rightarrow t_{\lambda(i)}]$.

Consider the example in Figure 4, which is a standard B-tree built on the join attribute in $R_{\lambda(i)}$. For generality, here we allow duplicates in the attribute values (so the B-tree serves as a secondary index in this case), and each routing element equals to the smallest value in its subtree. Suppose the join condition is such that all tuples with attribute values between 3 and 5 would join with $t_{\eta(i)}$ (recall that we support equality, inequality, and range join conditions). Starting from the root node v_1, we see that among its three children, two of them can potentially contain joining tuples (the first and the second child). So we pick each of these two children with probability 1/2. Suppose we have picked the second child, v_2. Then we see that it has only one child (i.e., v_3) that can contain joining tuples, so we pick v_3 with probability 1. Finally, after reaching a leaf node (v_3 in this example), we use binary search to find the range of joining tuples (3, 4 in this example), and pick one uniformly at random (with probability 1/2 for each of 3 and 4 in this example). The corresponding probability is thus $\Pr[t_{\eta(i)} \rightarrow t_{\lambda(i)}] = \frac{1}{2} \cdot 1 \cdot \frac{1}{2}$. Note that different joining tuples may be picked with different probabilities.

By replacing the aggregate B-tree with a standard B-tree, we gain the benefit of not having to modify the storage engine and transaction processing units in the database system. This stands in contrast with earlier online aggregation algorithms, which often require drastic changes to the underlying database engine.

3.3 Wander Join for Cyclic Queries

Wander join can be 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 $\gamma = (t_1, t_2, t_3, t_4, t_5)$ in 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.

Note that the failure probability of random walks on cyclic queries are generally higher than that on acyclic queries, resulting in a slower convergence rate. However, the costs to evaluate cyclic queries completely are also much higher. In fact, the relative speedup of wander join over full evaluation is even higher for cyclic queries, as shown in our experimental results in Section 7.1.3.

3.4 Selection Predicates and Group-By

Wander join can deal with selection predicates in the query easily: In the random walk process, whenever we reach a tuple t on 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 the same B-tree walking algorithm as described in Section 3.2. Correspondingly, we replace $\frac{1}{|R_{\lambda(1)}|}$ in Equation (3) by the probability that the tuple is sampled from the B-tree. In this case, the predicate will not fail any random walk, thus it is preferable to start from a table with selection predicates. More discussion will be devoted on this topic under walk plan optimization in Section 4.

If there is a `GROUP BY X` clause in the query, we will treat the query as multiple queries, each with a selection predicate $X = x$ for each distinct value x of attribute X. We will always start the random walks from the table containing the attribute X, and perform random walks from each of the groups in a round-robin fashion. For each group, we maintain an estimator and a confidence interval (see the next section on how to compute confidence intervals). After each group has received a number of random walks, we stop using round-robin. Instead, we always choose the group that has the largest confidence interval to start the next random walk. This simple greedy strategy will try to make sure that all the groups are well estimated.

3.5 Estimators and Confidence Intervals

As mentioned above, for each random walk γ, $v(\gamma)/p(\gamma)$ is an unbiased estimator of the `SUM` $= \sum_\gamma v(\gamma)$, where γ ranges over all the join results. Thus, we can perform multiple independent random walks and take the average of the $\gamma(v)/p(v)$'s to improve the accuracy. However, two questions remain: (1) How about other aggregates such as `COUNT` and `AVG`? (2) What can we say about the accuracy of the final estimator after, say, n random walks have been performed?

For ripple join, these questions are highly nontrivial, as it returns non-independent samples with complicated correlations that have been carefully taken care of. As wander join takes independent random walks, thus all the individual estimators are independent, the situation is much easier. Interestingly, we observe that the two questions above for wander join (with or without selection predicates) reduce to the case of sampling from a single table with a selection predicate, which has been studied by Haas [18].

3.5.1 Sampling from a Single Table. Let us first restate the problem studied by Haas [18]. We are given a table of N tuples, where each tuple t is associated with value $v(t)$, as well as an indicator variable $u(t)$ that is 1 if t meets the predicate and 0 otherwise. He expresses any aggregation function in the form of an average:

$$\mathtt{AGG} = \frac{1}{N}\sum_t u(t)v(t). \tag{4}$$

For example, setting $v(t) = N$ gives the `COUNT`; setting $v(t)$ to be N times the actual value of t gives the `SUM`.

Suppose we have sampled n tuples randomly (with replacement): $t_1, \ldots, t_n$. For any function f, h, introduce the following notation:

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

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

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

Then Haas [18] derived the estimators for various aggregation functions, as well as estimators for their variances, as follows:

$$\texttt{SUM}: \tilde{Y}_n = T_n(uv), \tilde{\sigma}_n^2 = T_{n,2}(uv); \tag{5}$$

$$\texttt{COUNT}: \tilde{Y}_n = T_n(u), \tilde{\sigma}_n^2 = T_{n,2}(u); \tag{6}$$

$$\texttt{AVG}: \tilde{Y}_n = T_n(uv)/T_n(u), \tilde{\sigma}_n^2 = \frac{1}{T_n^2(u)}\left(T_{n,2}(uv) - 2R_{n,2}T_{n,1,1}(uv,u) + R_{n,2}^2 T_{n,2}(u)\right), \tag{7}$$

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

Note that all of these estimators can be easily computed incrementally in $O(1)$ time per sample, because they all boil down to maintaining $T_n(f)$, $T_{n,q}(f)$, and $T_{n,q,r}(f,h)$, each of which is nothing but a sum over n terms, each corresponding to one sample.

3.5.2 The Reduction. To draw a reduction from wander join to the singe table sampling problem, we start with the following two observations after going through the derivation in [18]: (1) His results hold for any definition of u and v. (2) His results hold even if each t_i is sampled non-uniformly, as long as $\mathsf{E}[u(t_i)v(t_i)] = \texttt{AGG}$, where AGG is as defined in Equation (4), but the t_i's still need to be independent.

Based on these observations, we reduce the computation of estimators in wander join to that of sampling from a single table, formally stated as follows.

Lemma 1. *For each path γ sampled by wander join, define $v(\gamma) = 1$ if* AGG *is* COUNT*, and the actual value of the expression on γ to be aggregated if* AGG *is* SUM*. Define $u(\gamma) = 1/p(\gamma)$ if γ is a successful path, and 0 otherwise. Then the estimators for wander join are formulas (5)–(7).*

Proof. Imagine that we have a single table that stores all the paths in the join data graph (Figure 2), including both successful paths, as well as failed paths. Wander join can be equivalently seen as sampling from this imaginary table, though non-uniformly. Note that by the definitions of u and v in the lemma, we still have Equation (4). Suppose we have sampled a total of n random paths $\gamma_1, \ldots, \gamma_n$ by random walks. From the HT estimator, we have $\mathsf{E}[u(\gamma_i)v(\gamma_i)] = \texttt{AGG}$ for each γ_i, where AGG is either SUM or COUNT. Thus, all the results in [18] carry over to our case. This also includes other aggregation functions that are derived from SUM and COUNT, such as AVG, VARIANCE, and STDEV. □

Finally, we compute $\tilde{\sigma}_n^2$ according to Equations (5)–(7). Then the half-width of the confidence interval can be computed as (for a confidence level threshold α)

$$\varepsilon_n = \frac{z_\alpha \tilde{\sigma}_n}{\sqrt{n}}, \tag{8}$$

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

3.6 Optimization Techniques

In this section, we introduce two optimization techniques that can further improve the performance of wander join in practice.

3.6.1 Leaf Scanning. The first optimization technique concerns the last table in the walk order. The observation is that, if we have already walked a long way to reach the last table $R_{\lambda(k)}$, it is quite wasteful to just return one estimator. Instead of sampling one tuple from $R_{\lambda(k)}$ from those that can join with $t_{\eta(k)}$, we can visit all of them. Note that these tuples are stored consecutively in the B-tree index anyway. Note that if there is a selection predicate on $R_{\lambda(k)}$, we only consider

those that satisfy the predicate. This way, we are essentially getting multiple estimators at the cost of one walk. We call this technique *leaf scanning*.

There are a couple of points that one has to be careful with. First, these paths are not independently chosen, as they all share the same tuples in all but the last table. So we cannot directly apply the estimators and confidence interval formulas in Section 3.5. To get around this issue, we consider them as one "combined" path. More precisely, suppose these paths are $\gamma_1, \ldots, \gamma_\ell$. We set the value of the combined path as $v(\gamma) = \sum_{i=1}^{\ell} v(\gamma_i)$; correspondingly, we also remove the $\Pr[t_{\eta(k)} \to t_{\lambda(k)}]$ factor when computing $p(\gamma)$ in Equation (3). A further optimization is that, if the aggregation function is `COUNT` or any `AGG(expression)` where `expression` does not depend on the tuple in the last table, we can avoid actually retrieving the tuples in $R_{\lambda(k)}$; all we need is the number of tuples in $R_{\lambda(k)}$ that can join with $t_{\eta(k)}$. When the index has some aggregate information, this can be obtained more cheaply.

The second consideration is that the cost associated with retrieving all the tuples in $R_{\lambda(k)}$ that can join with $t_{\eta(k)}$ may not be bounded. Since these tuples might be correlated, it may not be beneficial to retrieve all of them (in the extreme case where their values are all the same, randomly picking one is as good as retrieving all of them). Furthermore, the cost depends on whether the index is a primary or a secondary index. If the index is a secondary index and the attribute to be aggregated is *not* the attribute on which the index is built upon, we need another level of redirection to retrieve the actual tuple to get that attribute. Therefore, we use the following implementation in practice. On the last table, we start the B-tree walking algorithm as described in Section 3.2. When the algorithm reaches a leaf node of the B-tree index and information about the attribute being aggregated is stored in that leaf node, we scan all tuples stored in the leaf. Otherwise, we revert to the original algorithm and only sample one tuple from the last table.

3.6.2 Selective Predicates. The second optimization technique deals with highly selective predicates. Suppose we walk from tuple $t_{\eta(i)}$ to table $R_{\lambda(i)}$, which has a selection predicate, and the selectivity is ρ (i.e., a fraction of ρ of the tuples on R satisfy the predicate). If ρ is very small, this may fail many random walks.

The optimization technique actually uses a similar idea as above. When performing the B-tree walking algorithm on $R_{\lambda(i)}$ and reaching a leaf node, we scan all the tuples stored in that leaf node to filter out all those that satisfy the predicate, and only randomly pick one of these tuples to continue the walk. Note that the calculation of $\Pr[t_{\eta(i)} \to t_{\lambda(i)}]$ should also be modified accordingly, i.e., in the last step, the probability should be 1 over the actual number of tuples in that leaf node that join with $t_{\eta(i)}$ *and* satisfy the predicate on $R_{\lambda(i)}$.

3.7 The Costs of Indexing

Our random walk based approach crucially depends on the availability of indexes. For example, for the three-table chain join in Equation (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. Similarly, when the query has a selective predicate, an index on the selection attribute can also make the random sampling more effective (see Section 3.4). On the other hand, there are costs associated with building and maintaining all these indexes. In this subsection, we discuss these costs and possible remedies to mitigate these costs.

The first cost is storage. Note that wander join only needs secondary B-tree indexes, in which we store a value and a pointer to each record in the base table. Assuming that we use 32-bit integers for both the value and the pointer, this is 8 bytes per record. In the TPC-H benchmark data, the

average size of a record is 118 bytes, due to the fact many tables are *wide*, which is true for many real-world database designs. Among all the 61 columns, only 35 are used in the join and (equality or inequality) selection conditions in all the 22 queries specified in the TPC-H benchmark. Thus, even if we build indexes on all these columns, the extra space cost is only about 1.4 times of the raw data. Thus, we would argue that space is not a severe cost.

The major cost of indexing is actually the maintenance cost, especially in a concurrent environment where multiple updates could take place at the same time, and the index has to be locked to avoid write conflicts. Thus, wander join is indeed not designed for such update-heavy workloads. Instead, it is better suited for an OLAP engine, which only sees batch updates that take place in offline time (e.g., at night). If one wishes to implement wander join in an OLTP engine, then we would not recommend using the traditional B-tree, but some more recent indexing schemes, such as the *fractal tree index* [5] (already implemented in MySQL and MongoDB), adaptive and holistic indexing [16, 17, 21, 26, 45] with transaction and concurrency control support, which support updates much more efficiently.

3.8 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 on 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 6. 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, $i = 1, \ldots, k-1$, 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 three to four tables, with the largest one having eight 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.

There are a few other aspects where we can compare wander join with ripple join:

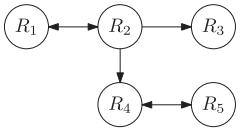
(1) *Computational costs:* There is also a major difference in terms of computational costs. Computing the confidence intervals in ripple join requires a fairly complex algorithm with worst-case running time $O(kn^k)$ [18], due to the non-independent nature of the sampling. On the other hand, wander join returns independent samples, so computing confidence intervals is very easy, as described in Section 3.5. 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.
(2) *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.
(3) *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 this section, we present a walk plan optimizer that chooses the best walk plan without the need to collect statistics.

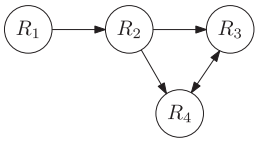
4.1 Walk Plan Generation

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.



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

Fig. 5. A directed join query graph and all its walk plans.



R_1, R_2, R_3, R_4. non-tree edge: (R_2, R_4)
R_1, R_2, R_4, R_3. non-tree edge: (R_2, R_3)

Fig. 6. Walk plan generation for a cyclic query graph.

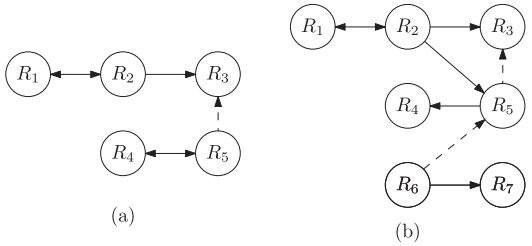


Fig. 7. Decomposition of the join query graph into directed spanning trees. Dashed edges are non-tree edges.

4.1.1 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 5, 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 forms a tree. An example is given in Figure 6.

4.1.2 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 7 (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 7 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 7(a) and (R_5, R_6) in Figure 7(b). Note that (R_3, R_5) in Figure 7(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 involved. Note that in the extreme case when there are no indexes, thus each component contains only one table, our algorithm essentially degenerates into ripple join.

4.1.3 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 (eight 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 the following three steps.

(1) For each vertex v, 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 7(b), only $T(R_1) = \{R_1, R_2, R_3, R_4, R_5\}$ and $T(R_6) = \{R_3, R_4, R_5, R_6, R_7\}$ 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.
(2) 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 7(b), and they both cover R_3, R_4, R_5.
(3) 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

[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.

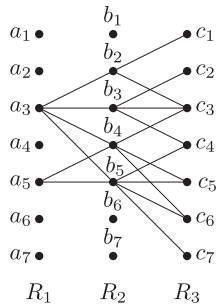


Fig. 8. Structure of the join data graph has a significant impact on the performance of different walk plans.

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 7(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 $T(R_1)$. Also, we give a proof that this algorithm produces a consistent assignment.

Lemma 2. *The algorithm produces a consistent assignment.*

Proof. We will prove by contradiction. 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 directly connected to $T(v) \setminus M$, which contradicts with the earlier statement that $u \in T(v) \cap M$. □

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 8, 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%.

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.

4.2.1 A Self Reduction. 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 though 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* [40]:

$$\begin{aligned}
&\mathrm{Var}\left[\frac{1}{W}\sum_{i=1}^{W} X_i\right] \\
&= \mathrm{E}\left[\mathrm{Var}\left[\frac{1}{W}\sum_{i=1}^{W} X_i \,\middle|\, W\right]\right] + \mathrm{Var}\left[\mathrm{E}\left[\frac{1}{W}\sum_{i=1}^{W} X_i \,\middle|\, W\right]\right] \\
&= \mathrm{E}\left[\frac{1}{W^2}\sum_{i=1}^{W} \mathrm{Var}[X_i]\right] + \mathrm{Var}\left[\frac{1}{W}\sum_{i=1}^{W} \mathrm{E}[X_i]\right] \\
&= \mathrm{E}[\mathrm{Var}[X_1]/W] + \mathrm{Var}[\mathrm{E}[X_1]] \quad //\mathrm{Var}[X_i] = \mathrm{Var}[X_j], \mathrm{E}[X_i] = \mathrm{E}[X_j] \text{ for any } i, j \\
&= \mathrm{Var}[X_1]\mathrm{E}[1/W] + 0 \\
&= \mathrm{Var}[X_1]\mathrm{E}[T/t] \\
&= \mathrm{Var}[X_1]\mathrm{E}[T]/t.
\end{aligned}$$

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

The next observation is that both $\mathrm{Var}[X_1]$ and $\mathrm{E}[T]$ can also be estimated by the random walks themselves! In particular, $\mathrm{Var}[X_1]$ is just the variance of the estimator, i.e., σ_n^2 in Section 3.5, while $\mathrm{E}[T]$ is just the average running time of performing a random walk following the walk plan. Thus, the problem of estimating the quality of a walk plan reduces to another instance of the online aggregation problem.

Thus, our optimizer will perform a certain number of "trial" random walks and estimate $\mathrm{Var}[X_1]$ and $\mathrm{E}[T]$ for each walk plan. Then we compute the product $\mathrm{Var}[X_1]\mathrm{E}[T]$ and pick the order with the minimum $\mathrm{Var}[X_1]\mathrm{E}[T]$. 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, and so forth. However, in our case, we do not have to pick the very best plan: If two plans have similar values of $\mathrm{Var}[X_1]\mathrm{E}[T]$, 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 $\mathrm{Var}[X_1]\mathrm{E}[T]$ 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 $\mathrm{Var}[X_1]\mathrm{E}[T]$ 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 $\tau = 100$.

4.2.2 Incorporating Trial Random Walks into the Overall Estimator. Finally, another interesting observation is that all the trial random walks are not wasted, since each random walk, no matter which plan it follows, returns an unbiased estimator. So the trial random walks issued by the optimizer can also be incorporated into the overall estimator, further reducing its variance. This is unlike traditional query optimization, where the cost incurred by the optimizer itself is pure "overhead."

However, some care has to be taken when deciding which trial random walks should be included or not, as some plans may return estimators with very high variances, such that including them may actually hurt the overall quality. Suppose there are m walk plans. For each plan, the optimizer has conducted x trial runs, and the trial runs of these m plans have variances $\sigma_1^2, \sigma_2^2, \ldots, \sigma_m^2$,

respectively, which have been estimated as in Section 3.5. Note that the time costs of the walk plans, i.e., the $E[T]$'s, do not play a role here. We sort these plans so that $\sigma_1^2 \le \cdots \le \sigma_m^2$.

Suppose the ℓ-th plan is the optimal plan picked by the optimizer. Note that ℓ may not be 1 since the plan with the smallest variance may have a higher time cost. Suppose we have performed y random walks in the actual execution of wander join following the optimal plan with variance σ_ℓ^2. Averaging these y random walks yields a variance of σ_ℓ^2/y. If we also include the trial random walks, then we may further reduce this variance. The first simple observation is that, if including the trial runs from the j-th plan can reduce the variance, then including the trial runs from the i-th plan must also reduce the variance, for any $i < j$. Thus, the problem reduces to picking the best i, such that the overall variance

$$\text{Var} = \frac{x(\sigma_1^2 + \cdots + \sigma_i^2) + y\sigma_\ell^2}{(ix + y)^2} \tag{9}$$

is minimized.

However, minimizing Equation (9) naively requires evaluating it for each i, which takes $O(m)$ time. This is too costly as we need to solve this minimization problem every time y increases. Below we derive a much more efficient method.

Introducing $\bar{\sigma}_i^2 = (\sigma_1^2 + \cdots + \sigma_i^2)/i$ and $z = ix$, we rewrite Equation (9) as

$$\text{Var} = \frac{z\bar{\sigma}_i^2 + y\sigma_\ell^2}{(z + y)^2}. \tag{10}$$

Instead of finding the best i to minimize Equation (10), we find the best z assuming that $\bar{\sigma}_i^2$ is fixed. Taking the derivative of Equation (10) with respect to z, we obtain

$$\begin{aligned}\frac{d\text{Var}}{dz} &= \frac{(z + y)^2\bar{\sigma}_i^2 - (z\bar{\sigma}_i^2 + y\sigma_\ell^2) \cdot 2(z + y)}{(z + y)^4} = \frac{(z + y)\bar{\sigma}_i^2 - 2(z\bar{\sigma}_i^2 + y\sigma_\ell^2)}{(z + y)^3} \\ &= \frac{-z\bar{\sigma}_i^2 + y\bar{\sigma}_i^2 - 2y\sigma_\ell^2}{(z + x)^3}.\end{aligned} \tag{11}$$

So, the optimal z that minimizes Var is when Equation (11) equals 0, i.e.,

$$z_{opt} = \left(\frac{2\sigma_\ell^2}{\bar{\sigma}_i^2} - 1\right) y.$$

However, this z_{opt} does not really solve the original minimization problem (9), since it assumes a fixed $\bar{\sigma}_i^2$. When $z \neq ix$, $\bar{\sigma}_i^2$ also changes. Nevertheless, the key observation is that at least it tells us whether a particular i is too small or too large. Specifically, if $z_{opt} < ix$, we should reduce i; if $z_{opt} > ix$, we should increase i. Then, we can use binary search to find the optimal i. In fact, we just need to do the binary search when we start with $y = 1$. Later on, whenever y increases, note that z_{opt} can only increase. Thus, we just need to wait until $z_{opt} > ix$, at which point we gradually increase i until $z_{opt} < ix$. This way, for most cases, we just need a simple inequality check when the optimal i does not change. Occasionally we need some more calculation when i needs to be increased, but there are only at most m such steps in the whole process.

5 XDB: INTEGRATING WANDER JOIN WITH DIFFERENT SYSTEMS

Wander Join can be easily integrated into existing database engines. To demonstrate this point, we have developed XDB (approXimate DB) by integrating wander join in various systems. In particular, we have designed and developed XDB in three versions:

(1) A tight integration with a traditional relational DBMS kernel; in particular, we used the latest version of PostgreSQL.
(2) An extension to Spark, a popular main memory based massively parallel data analytics engine designed to scale in a cluster setting.
(3) A plug-in version using PL/SQL so that users can realize XDB on top of any popular database systems without the need of updating its kernel.

5.1 Integration and Implementation in PostgreSQL

First, we have integrated 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. XDB is now open-sourced at https://github.com/initialDLab/XDB and https://github.com/InitialDLab/zeponline.

XDB extends PostgreSQL's parser, query optimizer, and query executor to support keywords like `CONFIDENCE`, `ONLINE`, `WITHINTIME`, and `REPORTINTERVAL`. 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 20000 CONFIDENCE 95 REPORTINTERVAL 1000.
```

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 1,000ms for up to 20,000ms.

Online aggregation queries are passed to an optimizer specific to wander join. The optimizer builds the join query graph and generates valid walk paths from the join query graph. The optimizer also replaces aggregation operators with online aggregation estimators and relative confidence interval operators. If the query contains an `INITSAMPLE` clause, which allows the engine to execute a number of trial runs using multiple paths to find the best walk order, all the valid walk paths are retained in the query plan. The query executor later iterates through all the walk paths, performs a number of trial runs as specified by the query, and computes a rejection rate estimation and a variance estimation. It then orders the walk plans by the rejection rate and breaks tie (rejection rates differed within 5%) by the variance estimation.

The executor extracts samples from primary or secondary B-tree indexes one by one given a walk path. The B-tree indexes are augmented with counts of subtrees in their internal nodes. The executor uses the counts to find the degrees of the tuples in the join data graph and extract samples. Selection predicates are immediately applied when the related tuples are sampled, instead of waiting until the walk is complete. Once a walk completes, the executor maintains a few aggregations of the samples and probabilities for the estimators. The executor returns the current estimators and relative confidence intervals periodically. Finally, it returns an empty tuple when the time budget is used up, which informs PostgreSQL that no more tuples are available.

A Zeppelin frontend is also developed as part of the XDB system, where its visualization module has been modified so that an online visualization of the (continuously updated) query results as well as the confidence intervals is enabled. Figure 9 shows a running query in the online version of Zeppelin with the PostgreSQL version of XDB running in the backend (note that execution time

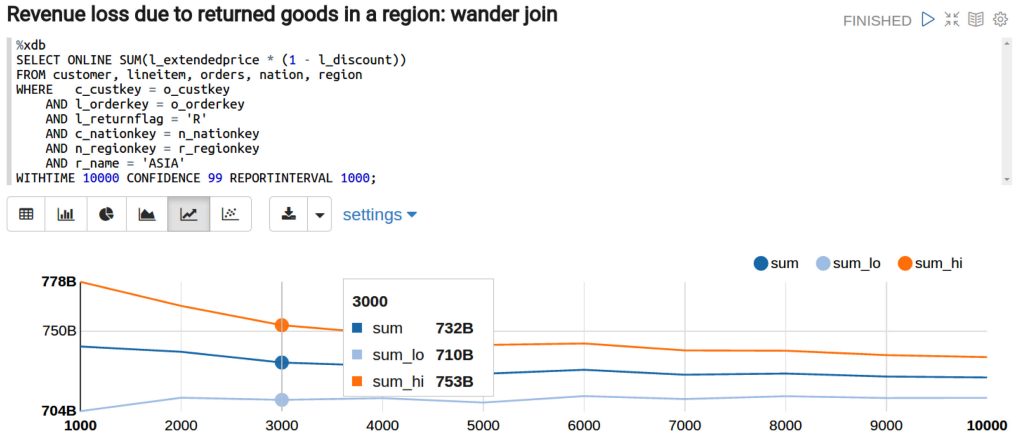


Fig. 9. XDB with the modified online Zeppelin: execution time in milliseconds.

is in milliseconds). The output shows three curves, which represent the continuous estimations by the estimators $\tilde{Y}$, $\tilde{Y} + \varepsilon$, and $\tilde{Y} - \varepsilon$, respectively. The system guarantees that the final exact aggregate value Y satisfies

$$\Pr[|Y - \tilde{Y}| \leq \varepsilon] \geq \alpha,$$

where α is the user-specified confidence level ($\alpha = 0.99$ in this query example) and ε is the confidence interval that gets continuously updated every second (for up to 10s) in this query example.

The only system implementation available for ripple join is the TurboDBO system [11, 29, 30]. In fact, the algorithm implemented in TurboDBO is much more complex than the basic ripple join in order to deal with limited memory, as described in these papers. We compared XDB with TurboDBO, 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, TurboDBO was built from the 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, XDB 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 [29] 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 XDB (which is a PostgreSQL with wander join implemented inside its kernel).

5.2 Integration and Implementation in Spark

Another major advantage of wander join is that it is an "embarrassingly parallel" algorithm. Since all the random walks are independent, it is straightforward to run all of them in parallel. This stands in contrast with ripple join, which is not so easy to made parallel. For the original ripple join algorithm, it is possible to make it run on a tightly coupled parallel database [39], but it is nontrivial and works only for hash joins. TurboDBO [11] is a centralized algorithm. A disadvantage of wander join, as we will later demonstrate in our experimental study (in Sections 7.1.2 and 7.1.5), is that its performance drops quite a lot when there is insufficient main memory (though it is still better than TurboDBO).

Therefore, a main memory based parallel/distributed database engine would provide an ideal environment for wander join to unleash its full potential. To substantiate this idea, we have

implemented wander join in Spark, a popular massively parallel data analytics engine designed to scale to thousands of machines.

The basic idea to implement wander join in a parallel system is to combine the random walks into large batches, where each batch consists of b independent random walks. The batch size mostly depends on how frequent the user wants the result to be reported; otherwise it should be as large as possible to best exploit the potential parallelism.

We load the tables into different RDD's, the partitioned dataset abstraction in Spark. The tuples in a partition are packed into a single object. Then we build two-level indexes over the RDD's. A sorted array index w.r.t. the join keys is built in each partition as a local index. Then, we collect the range of keys of the partitions as a global index. We implement each batch as a Spark job. In each step of a random walk, we use the global index to figure out which partitions could contain joining tuples in the next table. Then we sample partitions that the tuples are supposed to be shuffled to according to a multinomial distribution with the number of tuples in the matching partitions as weights. After the shuffle, we sample a matching tuple for each shuffled result. In the end, we run a reduce job to collect the statistics for the aggregation estimator and relative confidence interval calculation.

For group-by queries, we run the first batch on the original query to get an initial result. In subsequent batches, we use the relative confidence interval of the previous batch as weights to distribute samples to all the groups. More samples are retrieved from the groups with higher variance than from those with smaller variance. Thus, the relative confidence intervals of all the groups will shrink to roughly the same.

5.3 A Plug-in Design through PL/SQL

Thanks to its nonsurgical nature, wander join can be implemented *almost* completely outside a database engine. In this section, we describe our efforts in implementing wander join in PL/SQL as a stored procedure in System X. The obvious benefit of a PL/SQL implementation is that we can provide online aggregation simply as an add-on package, which can be used on any database system that supports PL/SQL. Unfortunately, we cannot completely achieve this goal, due to a couple of primitive operations that are currently not in the SQL standard. In fact, we feel that these primitives are so basic that they should be included in the SQL standard some day. We get around these unsupported primitives by building auxiliary tables in an offline stage. With these auxiliary tables, we can implement wander join in pure PL/SQL, but the downside is that these auxiliary tables will have to rebuilt whenever the underlying data changes.

The basic idea in implementing wander join in PL/SQL is the same as that in Spark, namely, we will perform the random walks in batches of size b. The key observation is that each step of the b random walks can be performed by a join, while the full random walks can be actually done by a single SQL statement using nested joins.

Throughout this section, we will use the three tables shown in Tables 1, 2, and 3 in the TPC-H benchmark dataset as running examples. Note that only a subset of the columns of the tables are shown here; the full tables consist of many more columns. The underlined column names denote the primary keys.

5.3.1 Primitive Operations and Workarounds.

Primitive 1: Sampling with a Predicate. The first primitive we need is just sampling a given number of tuples (with replacement) from a table with a selection predicate. We note that some database systems do provide a `SAMPLE` clause that may follow a `SELECT` statement. However, the implementation is that the database will evaluate the `SELECT` statement first, and then flip a coin for each result to decide whether it should be returned to the user. Such an implementation is

Table 1. customer

c_custkey	*c_nationkey*	*c_mktsegment*
1	1	BUILDING
2	4	AUTOMOBILE
3	3	AUTOMOBILE
4	20	MACHINERY
5	18	HOUSEHOLD
6	3	BUILDING

Table 2. orders

o_orderkey	*o_custkey*	*o_orderdate*
1	6	1993-10-14
2	6	1993-12-24
4	3	1997-11-01
6	4	1995-11-02
7	1	1994-03-06
8	2	1992-09-09
9	6	1992-05-24

Table 3. lineitem

l_linenumber	*l_orderkey*	*l_extenedprice*	*l_discount*
1	9	4,225.50	0.04
2	9	63,476.30	0.00
3	8	12,754.04	0.05
4	1	21,526.68	0.07
5	1	14,145.45	0.09
6	1	61,156.44	0.04
7	2	22,548.97	0.01
8	7	19,092.48	0.06
9	6	28,906.25	0.10
10	4	3,765.35	0.08

very inefficient when the table is large and/or the selection predicate is selective. Recall that when there is a B-tree index on the selection predicate, a much more efficient method is to walk down the B-tree as described in Section 3.2. This algorithm may not return a uniform sample but this is not a problem for wander join.

Since this B-tree walking algorithm is not implemented in System X, we introduce the following workaround. Suppose we want to sample from the `customer` table with a selection predicate on `c_mktsegment`. We sort the `customer` table by `c_mktsegment` and assign consecutive numbers $1, 2, \ldots$, which we call *ranks*, to all the tuples (please see Table 4). Then, we build an auxiliary table `mktsegment_to_rank` that maps the `c_mktsegment` to the corresponding ranges of ranks (see Table 5). These are done in an offline stage (using PL/SQL).

If an online query requires a sample of size b from the `customer` table with the predicate *c_mktsegment* = 'BUILDING', then we can use the following SQL statement:

```
SELECT customer.*
  FROM customer,
     (SELECT round(dbms_random.value(c_low_rank, c_high_rank)) AS c_sample
      FROM dual,
        (SELECT c_low_rank, c_high_rank
         FROM mktsegment_to_rank
         WHERE c_mktsegment = 'BUILDING')
      CONNECT BY LEVEL <= b)
  WHERE c_rank = c_sample.
```

Table 4. customer with Ranks

c_rank	*c_custkey*	*c_nationkey*	*c_mktsegment*
1	2	4	AUTOMOBILE
2	3	3	AUTOMOBILE
3	1	1	BUILDING
4	6	3	BUILDING
5	5	18	HOUSEHOLD
6	4	20	MACHINERY

Table 5. mktsegment_to_rank

c_mktsegment	*c_low_rank*	*c_high_rank*
AUTOMOBILE	1	2
BUILDING	3	4
HOUSEHOLD	5	5
MACHINERY	6	6

If there are multiple columns on which we would like to support such a sampling-with-predicate primitive, then we add one such rank column and a corresponding column_to_rank auxiliary table.

Primitive 2: Random Join. Recall that a standard join operator $R_1 \bowtie R_2$ returns, for each tuple $t_1 \in R_1$, all tuples $t_2 \in R_2$ such that t_2 joins with t_1. Instead of finding all such t_2's, we define a *(left) random join* operator, which returns only one $t_2 \in R_2$, selected randomly from all tuples in R_2 that join with t_1, for each $t_1 \in R_1$. If no tuple in R_2 join with t_1, no result is returned for t_1. This can also be easily achieved by the B-tree walking algorithm (Section 3.2). In the absence of such a primitive in System X, we provide the following workaround.

Suppose we have already obtained b tuples from the customer table, represented by their *c_rank*'s. These ranks are stored in a temporary table called tmp_customer_ranks. To facilitate a random join from tmp_customer_ranks to orders, we sort the tuples in the orders table by *o_custkey*, and assign ranks to them by this order. Then, we build an auxiliary table, called customer_to_order, which maps each customer to the range of ranks of his/her orders. Please see Table 6. Then, the following SQL statement will compute the random join from tmp_customer_ranks to orders. For simplicity, this query only retrieves the rank of the randomly selected order for each customer. If other attributes of these orders are needed, we just need to perform another join with the original orders table by the primary key *o_orderkey*.

```
SELECT round(dbms_random.value(o_low_rank, o_high_rank)) AS o_sample
FROM tmp_customer_ranks, customer_to_orders
WHERE tmp_customer_ranks.c_rank = customer_to_orders.c_rank.
```

Similarly, we build a table orders_to_linetime to facilitate the random join from orders to lineitem (Table 7), and also add a rank column to the lineitem (Table 8). The ranks of lineitem table are according to the *l_orderkey* of these line items. Again, if a table joins with more than one table, multiple table_to_table auxiliary tables and corresponding ranks will be added. For example, lineitem also has a *l_partkey* column, which is a foreign key referencing the *p_partkey* column in the part table. So, we also add a part_to_linetime table and another rank column in

Table 6. customer_to_orders

c_rank	*c_custkey*	*o_low_rank*	*o_high_rank*
1	2	2	2
2	3	3	3
3	1	1	1
4	6	5	7
5	4	4	4

Table 7. orders_to_lineitem

o_rank	*o_orderkey*	*l_low_rank*	*l_high_rank*
1	7	7	7
2	8	8	8
3	4	5	5
4	6	6	6
5	1	1	3
6	2	4	4
7	9	9	10

Table 8. lineitem with Ranks

l_rank	*l_linenumber*	*l_orderkey*	*l_extenedprice*	*l_discount*
1	4	1	21,526.68	0.07
2	5	1	14,145.45	0.09
3	6	1	61,156.44	0.04
4	7	2	22,548.97	0.01
5	10	4	3,765.35	0.08
6	9	6	28,906.25	0.10
7	8	7	19,092.48	0.06
8	3	8	12,754.04	0.05
9	1	9	4,225.50	0.04
10	2	9	63,476.30	0.00

the lineitem table according to the *l_partkey* of the line items. In our implementation, we retrieve all the foreign key constraints from the system catalog and build all the auxiliary tables automatically using PL/SQL. Necessary indexes are also built on the auxiliary tables, so that a random join can be performed more efficiently. In fact, System X's own optimizer will automatically decide the best physical plan for a random join at runtime: when b is small (relative to the index size), System X will do an index lookup for each tuple; when b is relatively large, a full index scan is often used.

5.3.2 Wander Join in PL/SQL. Equipped with the two primitives, we are now ready to give a full example of wander join in PL/SQL. This example uses the following query from the TPC-H benchmark:

```
SELECT SUM(l_extendedprice * (1 - l_discount))
FROM customer, orders, lineitem
WHERE l_orderkey = o_orderkey
   AND c_custkey = o_custkey
   AND l_shipdate > '1995-01-02'
   AND c_mktsegment = 'BUILDING'.
```

One batch of b random walks of wander join can be implemented with the following SQL statement:

```
SELECT SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount) * d) / b
FROM lineitem,
  (SELECT round(dbms_random.value(l_low_rank,l_high_rank) AS l_sample,
               d * (l_high_rank - l_low_rank + 1) AS d
```

```
FROM orders_to_lineitem,
  (SELECT round(dbms_random.value(o_low_rank, o_high_rank)) AS o_sample,
            d * (o_high_rank - o_low_rank + 1) AS d
   FROM customer_to_orders,
    (SELECT round(dbms_random.value(c_low_rank, c_high_rank)) AS c_sample
              c_high_rank - c_low_rank + 1 AS d
     FROM dual,
       (SELECT c_low_rank, c_high_rank
        FROM mktsegment_to_rank
        WHERE c_mktsegment = 'BUILDING')
     CONNECT BY LEVEL <= b)
   WHERE c_sample = customer_to_orders.c_rank)
 WHERE o_sample = orders_to_lineitem.o_rank)
WHERE l_sample = lineitem.l_rank AND lineitem.l_shipdate > '1995-01-02'.
```

The code should be self-explanatory, though there are a few points worth highlighting: (1) The sampling probability of each random walk is calculated as the d column in the intermediate results of the subqueries. It gets multiplied by the length of the range of the ranks in every step, so that in the end, $1/d$ is exactly the final sampling probability. (2) Selection predicates can be incorporated. In particular, the predicate on *c_mktsegment* is on the starting table, so we make use of the `mktsegment_to_rank` table so as to focus on sampling from those tuples satisfying the predicate. The other predicate on *l_shipdate* is checked in the end, and the random walk is failed if it does not satisfy this predicate. Note that a failed random walk returns 0, so we do not need any special handling, but just divide the whole sum by b in the final estimator. (3) This example does not include the confidence interval, but it can also be easily computed, since it is nothing but another aggregation function.

We have implemented the whole wander join algorithm as a stored procedure. It takes in a SQL query as input (as a string), as well as the batch size b as a parameter. Then it generates the SQL statement such as the one above automatically. Then it uses a PL/SQL loop to iteratively execute every batch until the user terminates the query.

Unfortunately, we have not been able to implement the walk plan optimizer in PL/SQL, due to the large overhead associated with executing a nested SQL statement in System X. Recall that our optimizer issues a small number (around 100) of random walks for each plan to estimate its quality. However, using such a small batch size is just not economical (running one batch of random walks takes at least 2s in System X regardless of the batch size), and doing this for all plans would just be too slow. Therefore, the walk plan optimizer is perhaps the only component in our algorithm that has to be implemented inside the database engine. In our PL/SQL implementation, we simply use the order in the `FROM` clause provided by the user.

6 EXPERIMENTS ON STAND-ALONE IMPLEMENTATION

We have implemented wander join in a variety of settings. In this section, we report the experimental results on a stand-alone implementation of wander join, so as to see its "pure" algorithmic performance in comparison with ripple join without any system overhead. In later sections, we evaluate its performance in full-fledged database systems, including PostgreSQL, Spark, and a major commercial database system (referred to as System X[2]).

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

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

Fig. 10. Q3 in experiments.

6.1 Stand-Alone Implementation

We have implemented both wander join and ripple join in C++. Since ripple join is only designed for memory-resident data, our stand-alone implementation of wander join is also optimized for main memory. In particular, we have used main memory index structures for both algorithms, i.e., hash tables (using `std :: unordered_map`) and binary search trees (BST) (using `std :: ordered_map`). On the other hand, in our PostgreSQL implementation (Section 5.1), we use the B-tree index provided by PostgreSQL without any modification.

Specifically, all raw data tuples in each table are stored in the primary key order in an array. For each join key, a hash table index is built, so that sampling a neighbor takes $O(1)$ time. For each key having a selection predicate, we build a BST index, so that randomly sampling a tuple in the table $O(\log N)$ 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 tuples, which have many other attributes that are not indexed.

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. 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 has to check tuples one by one until reaching a tuple that satisfies 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, BST indexes are built on all the selection predicates.

6.2 Data and Queries

We used the TPC-H benchmark data and queries for the experiments, which were also used by the DBO/TurboDBO work [11, 29, 30]. We used five tables, `nation`, `supplier`, `customer`, `orders`, and `lineitem`. We used the TPC-H data generator with the appropriate scaling factor to generate datasets of various sizes. We picked three queries Q3 (three tables), Q7 (six tables; the `nation` table appears twice in the query), and Q10 (four tables) in the TPC-H specification and only kept join and aggregation parts of them as our test queries. We included the SQL statements of these queries in Figures 10–12.

6.3 Experimental Results

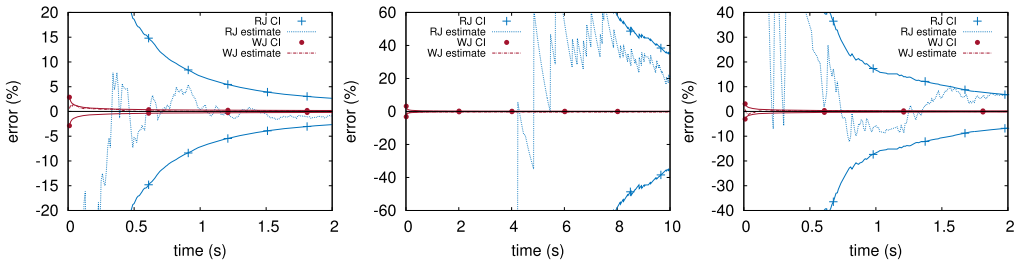
6.3.1 Queries without Selection Predicates. We first run wander join and ripple join on a 2GB dataset, i.e., the entire TPC-H database is 2GB, using the "barebone" joins of Q3, Q7, and Q10, where we drop all the selection predicates and group-by clauses. In Figure 13 we plot how the *confidence interval* (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

```
SELECT SUM(l_extendedprice * (1 - l_discount))
FROM supplier, lineitem, orders, customer, nation n1, nation n2
WHERE s_suppkey = l_suppkey
AND o_orderkey = l_orderkey
AND c_custkey = o_custkey
AND s_nationkey = n1.n_nationkey
AND c_nationkey = n2.n_nationkey
AND n1.n_name = 'CHINA'
```

Fig. 11. Q7 in experiments.

```
SELECT SUM(l_extendedprice * (1 - l_discount))
FROM customer, lineitem, orders, nation
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND l_returnflag = 'R'
AND c_nationkey = n_nationkey
```

Fig. 12. Q10 in experiments.

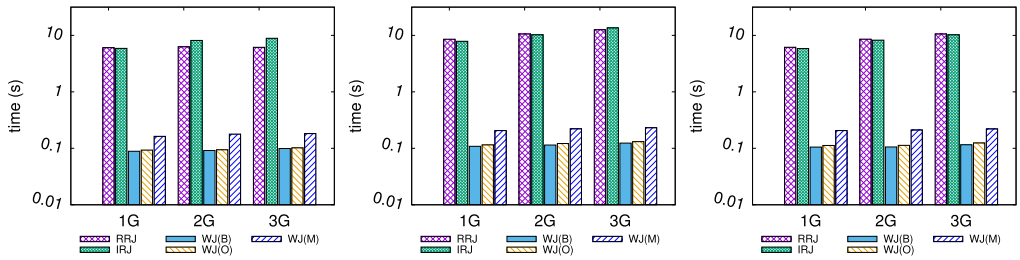


(a) Q3 (b) Q7 (c) Q10

Fig. 13. Stand-alone implementation: Confidence intervals and estimates on barebone queries on 2GB TPC-H dataset; confidence level is 95%.

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 0.1s whereas ripple join takes more than 4s to reach 1% confidence interval. The full exact join on Q3, Q7, and Q10 in this case is 18s, 28s, and 19s, respectively, using hash join.

Next, we ran the same queries on datasets 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 14 we report the time spent by each algorithm to reach ±1% confidence interval with 95% confidence level on datasets of sizes 1GB, 2GB, and 3GB. We also report the time costs of the optimizer in Table 9. From the results, we can draw the following observations:



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

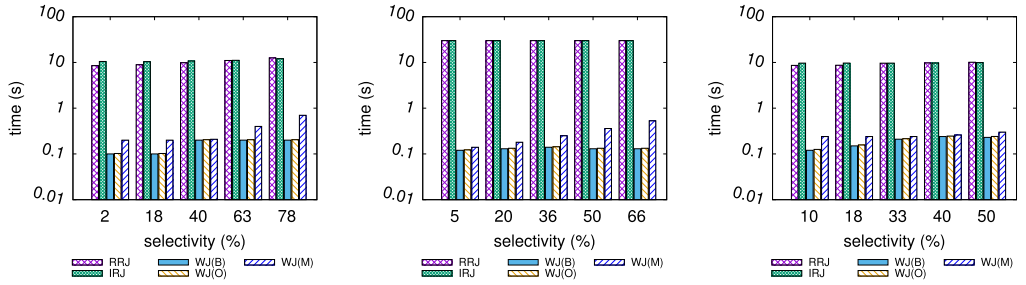
Fig. 14. Stand-alone implementation: Time to reach ±1% confidence interval and 95% confidence level on TPC-H data sets of different sizes.

Table 9. Stand-Alone Implementation: Time Cost of Walk Plan Optimization (Execution Time to Reach ±1% Confidence Interval and 95% Confidence Level on TPC-H Data Sets of Different Sizes)

	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

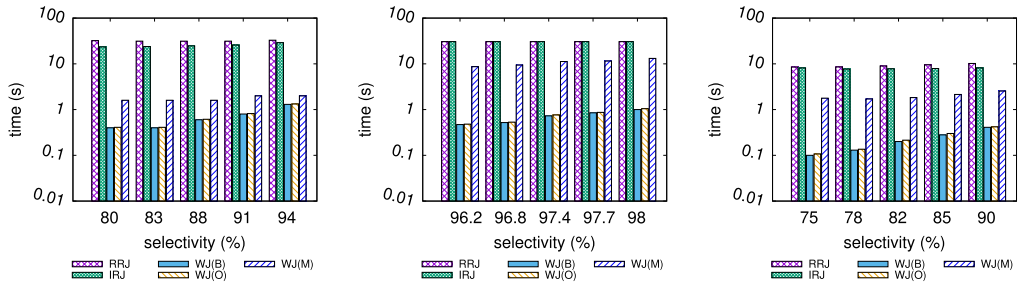
(1) Wander join is in general faster than ripple join by two orders of magnitude to reach the same confidence interval.
(2) The running time of ripple join increases with N, the data size, though mildly. Recall from Section 3.8 that ripple join expects to get $n^k(\frac{d}{N})^{k-1}$ sampled join results 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.

6.3.2 Queries with Selection Predicates. Next, we put back the selection predicates to the queries. Figure 15 shows the time to reach ±1% confidence interval with 95% confidence level for the algorithms on the 2GB dataset, with one selection predicate of varying selectivity, while



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

Fig. 15. Stand-alone implementation: Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H dataset with one selection predicate of varying selectivity.



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

Fig. 16. Stand-alone implementation: Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H dataset with multiple selection predicate of varying selectivity.

Figure 16 shows the results when all the predicates are put back. Here, we measure the overall selectivity of all the predicates as

$$1 - (\text{join size with predicates})/(\text{barebone join size}), \tag{12}$$

so higher means more selective.

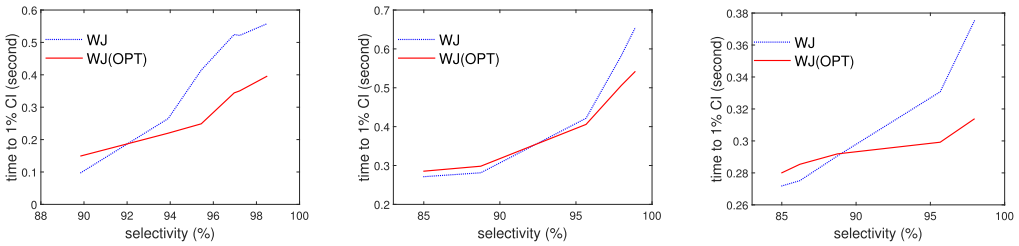
From the results, we see that one selection predicate has little impact on the performance of wander join, because most likely the optimizer will elect to start the walk from that table. Multiple highly selective predicates do affect the performance of wander join, but even in the worst case, wander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

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 will not be incorporated into the overall estimator. However, the good plans can accumulate $\tau = 100$ successful random walks very quickly, so we do not waste too much time anyway.

6.3.3 Results on the Optimization Techniques. Finally, we looked more closely into the improvements offered by the two optimization techniques introduced in Section 3.6. Both techniques are described assuming a B-tree index, where we scan all tuples in a leaf block in the B-tree. Since we use hash tables and BSTs as our index structures, which have no notion of a "leaf block," we simply set a virtual block size of 256, and scan up to so many tuples in the index.

Table 10. Performance Improvement of the Leaf Scanning Technique

Query	Zipf.	Max(%)	Min(%)	Avg.(%)
Q3	0	32	20	25
	0.5	25	18	23
	1	23	16	21
	1.5	24	15	20
	2.0	18	8	13
Q10	0	48	26	35
	0.5	41	33	27
	1	36	23	29
	1.5	32	23	27
	2	23	14	18



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

Fig. 17. Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H dataset with multiple selection predicate of varying selectivity.

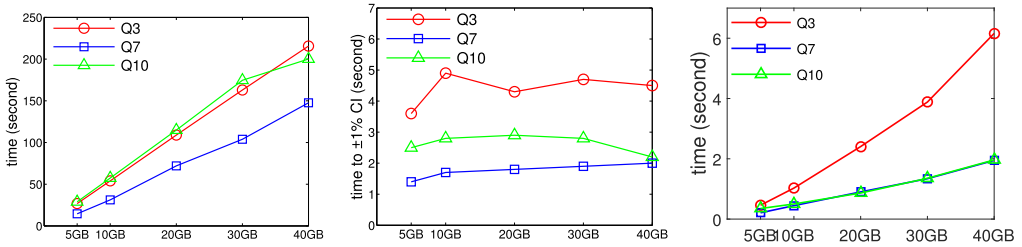
Recall that the idea of the first optimization technique, leaf scanning, is to return multiple estimators with one random walk, which might be correlated. Therefore, its performance will depend on the actual data distribution. To this end, we have tuned the parameter in the TPC-H data generator to generate datasets with different levels of skewness. Specifically, we controlled the Zipf parameter in the data distribution for the `lineitem` table, which is set to be the last table in the walk order. We generated datasets multiple times, and measured the performance improvement, i.e., the speedup in terms of the time to reach 1% confidence interval with 95% confidence level. The results are shown in Table 10. We see from the table that, indeed, the performance improvement depends on how skewed the data is: the improvement is smaller when the data is more skewed. Generally speaking, the leaf scanning technique provides around 20% performance improvement.

The second optimization technique described in Section 3.6 is aimed at highly selective queries. Therefore, we tested it with queries with varying selectivity. The results on the algorithm with and without using this technique are shown in Figure 17. From the results, we see that this technique is indeed more effective when the selectivity of the predicates is higher.

7 EXPERIMENTAL RESULTS WITH SYSTEM IMPLEMENTATION (XDB)

7.1 XDB with PostgreSQL: Experimental Results

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 TurboDBO in the latter case. TurboDBO [11] is an



(a) PostgreSQL full join (b) Wander join in PostgreSQL (c) MonetDB full join

Fig. 18. System implementation experimental results with sufficient memory: 128GB memory.

improvement to the original DBO engine, that extends ripple join to data on external memory with many optimizations.

7.1.1 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 toward 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 128GB memory and datasets of sizes up to 40GB. We ran both wander join (implemented inside PostgreSQL) and the built-in PostgreSQL full join on the same queries.

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 18 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 3s to 4s, when the data size increases from 5GB to 40GB in Figure 18(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 datasets, the CPU cache may not be as effective as on smaller datasets. These system reasons might have explained the small performance drop of wander join on larger datasets. 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 TurboDBO in this case. However, it turned out that TurboDBO spends even more time than PostgreSQL's full join, so we do not show its results. This seems to contradict with the results in [30]. In fact, this is because TurboDBO is intentionally designed for large data and small memory. In the experiments of [30], 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, TurboDBO 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 TurboDBO performs better than the full join.

In-memory column-oriented databases such as MonetDB are specifically optimized for OLAP workloads. It is known to be able to outperform row-store RDBMS by orders of magnitude on

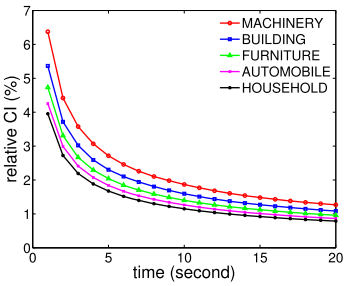
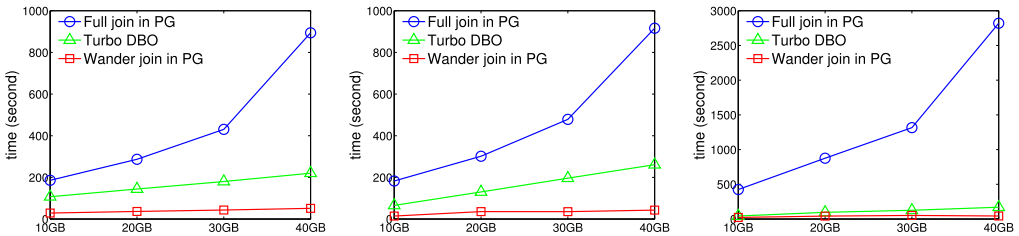


Fig. 19. Wander join in PostgreSQL with sufficient memory: Q10 with "GROUP BY *c_mktsegment*" on 40GB data.



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

Fig. 20. System implementation experimental results with limited memory, 4GB memory.

certain OLAP workloads but could also perform worse than a row-store RDBMS on OLTP workloads. There is simply not a single simple strategy that works for all workloads [13, 14]. Hence, it is not really fair to compare wander join in PostgreSQL, which has to retrieve an entire row at each step of random walk, to full join in an optimized column store. Nevertheless, in order to show that wander join can still be useful even in that case, we ran MonetDB on the same queries and datasets.

As shown in Figure 18(c), full join in MonetDB outperforms PostgreSQL by up to two orders of magnitude, and also outperforms wander join in all data sizes but TPC-H scale factor 40, due to its highly optimized column store and query engine. However, the running time of full join in MonetDB scales linearly with respect to data size. On the other hand, wander join's running time only increases logarithmically with respect to data size. Thus, wander join performs better than a well-optimized full join when the data size is large enough. In fact, the usefulness of wander join is orthogonal to the physical database design because it can (1) improve the analytical workload processing capability of a traditional row store; (2) be useful even for a highly optimized column store as long as data size continues growing.

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 19. Since we use a greedy strategy that tries to minimize the variance for the group that currently has the largest confidence interval, this results in a fairly balanced result in the convergence of the estimators for all the groups.

7.1.2 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 TurboDBO to reach ±5% confidence interval with 95% confidence level, as well as the time of the full join in PostgreSQL, are shown in Figure 20.

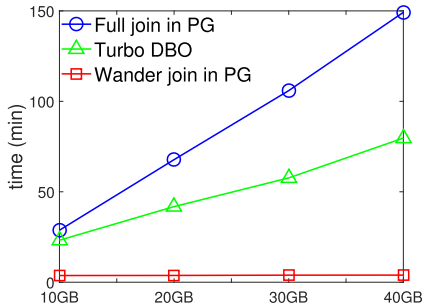


Fig. 21. System implementation experimental results on cyclic query with sufficient memory: 128G.

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 18 to more than 50s in Figure 20, and that is after we have relaxed the target confidence interval from ±1% to ±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 TurboDBO clearly have a linear dependency on the data size, though at different rates. On the 10GB dataset, wander join and TurboDBO have similar performance, but eventually wander join would stand out on very large datasets.

Anyway, spending 50s just to get a ±5% estimate does not really meet the requirement of interactive data analytics, so strictly speaking both wander join and TurboDBO have failed in this case (when data has significantly exceeded the memory size). However, as memory sizes grow larger and memory clouds get more popular (e.g., using systems like RAMCloud [43] and FaRM [12]), with the SSDs as an additional storage layer, in the end we may not have to deal with this barrier at all. What is more, as shown in Figure 20, wander join (and TurboDBO) 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.

It is also interesting to contrast TurboDBO and wander join in the way they handle disk-resident data. TurboDBO stores the tuples in each table in a random order on disk, and reads them sequentially at query time. Thus, it has good I/O efficiency but poor success probability of joining the tuples together. On the other hand, wander join stores the tuples in B-trees, and probes the B-tree at query time. It has low I/O efficiency but is better at finding join results. How to combine the advantages of the two approaches to yield better results remains a challenging problem.

7.1.3 Cyclic Queries. Up to now, only line queries from TPCH are executed and analyzed. Aiming to validate our idea discussed in Section 3.3, we chose one cyclic query(Q5) from TPCH to study wander join (implemented inside PostgreSQL), TurboDBO (with sufficient memory), and the built-in PostgreSQL full join's performance on this special query. And the experimental setting used is the same with Section 7.1.1. Then, we measured each method's performance by the time to 1% CI and plotted them in Figure 21.

Table 11. PostgreSQL with Wander Join: Time Cost of Walk Plan Optimization and Total Execution Time (and the Actual Error Achieved)

		Sufficient memory					Limited memory				
	SF[1]	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	11.92	11.90	0.002	15.52	0.18	121.35	111.59	1.80	478.99	3.62
	20	11.34	11.32	0.90	13.81	0.03	166.78	155.57	0.35	399.10	1.87
	30	11.52	11.50	0.41	13.79	0.08	194.80	183.67	1.02	568.52	2.38
	40	11.39	11.37	0.71	13.30	0.40	194.90	183.73	4.41	624.51	3.53
Q7	10	72.23	72.00	0.48	72.14	0.65	767.49	749.21	0.79	1873.37	1.53
	20	60.87	60.63	0.27	62.81	0.05	1,506.90	1,482.04	0.12	2,032.00	0.81
	30	60.72	60.44	0.07	63.29	0.06	1,634.60	1,602.61	2.18	2,140.00	1.49
	40	64.35	64.10	0.39	65.79	0.06	2,001.17	1,968.47	0.77	2,388.67	1.54
Q10	10	13.95	13.69	0.44	17.89	0.87	196.75	189.77	2.89	632.42	5.01
	20	14.77	14.49	0.35	23.48	0.51	236.16	228.85	0.66	640.89	0.51
	30	13.28	12.98	0.005	24.34	0.04	159.58	152.55	0.31	570.00	3.21
	40	10.1	9.8	0.23	43.67	0.20	125.74	119.26	1.40	1,014.744	5.74

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 (%).

Due to the higher failure probability of random walks on cyclic queries, wander join consumed much more time to converge to 1% CI than line queries. However, wander join can still perform very well and it reached 1% CI almost within a same time span (≈4min). In contrast, full join's running time grew linearly as the scale of data increased. Also, TurboDBO's performance was influenced by data size. As we mentioned the evaluation cost on cyclic queries would be higher, TurboDBO would hardly reach such strict CI limit in early levelwise steps. So, different from Figure 20, we observed that the gap between wander join and TurboDBO would be larger in Figure 21.

7.1.4 Effectiveness of Walk Plan Optimization. In the stand-alone 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 11. Table 11 presents the results on the effectiveness and overhead of walk plan optimization for wander join in PostgreSQL. Table 12 presents our PostgreSQL implementation of wander join against both TurboDBO and a commercial database system (denoted as System X). For results in both Tables 11 and 12, we investigated both sufficient memory and limit memory scenarios. In Table 11, 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 stand-alone 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 +

Table 12. Accuracy Achieved in 1/10 of System X's Running Time for Computing the Full Join

		Sufficient memory					Limited memory				
	SF[1]	System X[2]	TurboDBO		PG+WJ[5]		System X	TurboDBO		PG+WJ	
			CI[3]	AE[4]	CI	AE		CI	AE	CI	AE
Q3	10	32.24	–	–	1.24	0.47	107.27	–	–	8.39	1.65
	20	74.29	–	–	0.75	0.02	249.94	–	–	5.87	7.52
	30	65.17	–	–	0.84	0.31	428.39	–	–	5.03	5.13
	40	90.23	–	–	0.70	0.03	707.04	48.50	30.60	4.28	0.58
Q7	10	33.62	–	–	0.68	0.17	103.3	–	–	5.81	5.09
	20	73.03	–	–	0.46	0.16	205.7	–	–	6.65	4.05
	30	57.82	–	–	0.54	0.001	326.35	–	–	5.17	1.02
	40	77.92	–	–	0.48	0.29	445.86	–	–	4.89	2.73
Q10	10	40.43	–	–	0.81	0.32	146.57	47.71	23.24	6.15	0.25
	20	98.96	82.06	21.93	0.52	0.35	326.67	35.62	14.60	5.72	1.20
	30	109.19	138.29	66.50	0.47	0.10	697.06	26.43	6.69	4.28	5.00
	40	138.87	97.68	11.99	0.37	0.11	829.97	11.31	1.32	3.91	3.73

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.

plan execution) is not much more expensive than the best plan execution itself, and is still much better than the plan used by PostgreSQL.

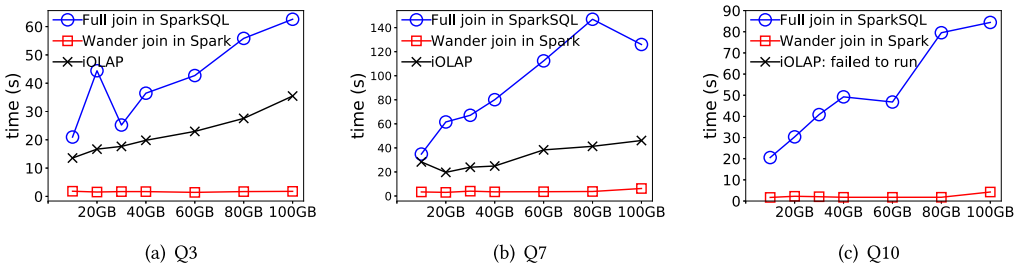
In summary, we see that in all cases, the optimizer can pick a plan that is much better than the plan generated from the input query and used by PostgreSQL. And generally speaking, query optimizer in a database engine tries to optimize the full join, not online aggregation. That's the value of having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

7.1.5 Comparing with a Commercial-Level Database System. Finally, to gauge how our PostgreSQL (PG) implementation of wander join performs in comparison to a major commercial database system (referred to as System X), we ran the queries (in full) on System X, and then see how much accuracy our PG (with wander join) and TurboDBO 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 12. 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, TurboDBO did not return any results in the time given, which is consistent with previously reported results, that TurboDBO usually starts to return results after a few minutes [11, 30].

7.2 Spark Version of XDB: Experimental Results

We compare our wander join implementaion in Spark against the full join of SparkSQL and iO-LAP [56] on a 16-worker cluster. Each worker has an 8-core CPU at 3.5GHz with 64GB RAM. For



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

Fig. 22. Spark implementation experimental results.

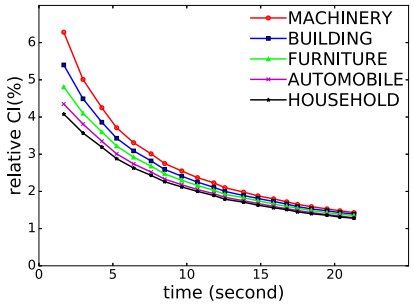


Fig. 23. Wander join performance on group-by query in Spark.

wander join in Spark and full join of SparkSQL, we installed Spark 1.6.2 and Hadoop 2.6.4. For iOLAP, we use the open-sourced implementation on Github (https://github.com/amplab/iolap), which is based on Spark 1.4.3. We did not run our implementation of wander join and full join on SparkSQL using the same Spark version because the SparkSQL as well iOLAP cannot run Q10 with Spark 1.4.3. We tested Q3 and Q7 on both versions and found no significant difference in performance of wander join in Spark and full join in SparkSQL. Hence, we conclude that different versions of Spark do not invalidate the following experimental results.

We first compare the performance of wander join with the full join in SparkSQL on TPC-H datasets of size up to 100GB. We run both algorithms on the same three queries as in the PG experiments. The times reported here are the medians of five independent runs. Figure 22 shows the time to reach 1% confidence interval with 95% confidence using our Spark wander join implementation and iOLAP, as well as the time of the full join in SparkSQL. As expected, wander join takes only 1–6s to reach 1% confidence interval on Spark, a constant time regardless of the data size. In contrast, both iOLAP and the full join take increasing amounts of time to complete. The time for iOLAP to reach 1% confidence interval is about an order of magnitude larger than wander join in Spark. The time for full join to complete is not linear to the data size because SparkSQL uses a larger number of partitions when the total data size increases, which could improve the performance a little bit. Nevertheless, the full join is still much more expensive than wander join on Spark. The performance gap between iOLAP/full join and wander join gets larger as the data size increases.

Next, we show the exprimental result of running a group-by query using wander join on Spark (Figure 23). We run a query that computes the revenue cost of returned items in each customer market segment on the 100GB TPC-H data. In the first batch, the relative confidence intervals vary among the five groups. In the later batches, the relative confidence interval converges to roughly

Table 13. Time Cost of Building Auxiliary Tables

Scale factor (GB)	Loading time (min)	Time to build auxiliary tables (min)
10	67	11.5
20	210	149
30	450	140
40	632	337

Table 14. Accuracy Achieved by Wander Join (PL/SQL Implementation) in 1/10 of System X's Time for Computing the Full Query

Query	Scale factor (GB)	Full query time (s)	AE (%)	CI (%)
Q3	10	16.22	1.17	2.60
	20	45.36	0.92	2.73
	30	47.24	0.21	3.13
	40	75.05	3.46	4.66
Q7	10	22.73	4.46	6.40
	20	31.53	3.98	6.44
	30	34.87	1.44	11.05
	40	38.83	2.37	11.31
Q10	10	21.77	1.62	2.78
	20	50.83	2.02	3.29
	30	54.61	0.59	2.60
	40	73.98	1.66	3.72

the same as we put more samples in those groups with higher relative confidence interval. Wander join achieves $< 3\%$ relative confidence interval among all groups in about 8s, while the full join takes more than 60s to complete.

7.3 Plug-In Version of XDB: Experimental Results

One of the first concerns in our PL/SQL implementation of wander join is the cost to build all the auxiliary tables. In fact, it is quite reasonable. In Table 13, we report the time to build all the auxiliary tables (including building indexes on them), in comparison with the time to load the raw TPC-H data into System X. We see that the former is just a fraction of the latter. The main reason is that building the auxiliary tables only relies on a small subset of the columns, while the full TPC-H data consists of many other columns, some even with texts.

Then, similar to the experiments in Section 7.1.5, we looked at the accuracy achieved by wander join in 1/10 of System X's time to execute the full query, over various queries and different data sizes. As before, in Table 14, we report both the actual error (AE) and the confidence interval (CI) at 95% confidence level. Note that both the full query and wander join (in PL/SQL) are run in System X.

Comparing with the results in Table 12, we see that the PL/SQL version of wander join performs worse than its PostgreSQL counterpart. This is expected, since our PostgreSQL implementation is inside the system engine, with direct access to the B-tree indexes, so that we can perform the sampling more efficiently. On the other hand, in the PL/SQL implementation, every batch of

random walks is executed as a complex nested SQL query, which needs to pass System X's entire query processing pipeline. This involves a lot of system overhead. Nevertheless, we believe that the results in Table 14 are quite encouraging, so that users can still enjoy the benefits of online aggregation even if they do not want to change their operational database to PostgreSQL.

8 RELATED WORK

The concept of online aggregation was first proposed in [23], and since then has generated much follow-up work, including the efforts in extending it to distributed and parallel environments [44, 46, 47, 51, 55] and multiple queries [52]; a complete survey of these works is out of the scope of this article. In particular, related to our problem, online aggregation over joins was first studied in [19], where the ripple join algorithm was designed. Extensions to ripple join were done over the years [11, 29, 30, 39], in particular, to support ripple join in DBO/TurboDBO for large data on external memory. Note that we have already reviewed the core ideas in online aggregation and ripple join in Section 2. PR-Join [9] performs ripple joins on individual hash functions to get more join results in early stages of join and thus can potentially achieve a higher convergence rate in confidence interval. It is only designed for two-table joins rather than the more complex multi-way joins we consider in this work.

Wander join can be considered as a generalization to *join synopsis* [1], which also performs walks in the join graph to obtain samples for approximate query answering. However, their algorithm works only for a special type of joins in which the property there is a bijection between the join results and tuples in a particular table, called the *source relation*. Thus, a random sample can be obtained by only sampling the source relation, and then simply finding the matching tuples from the remaining tables. In our terminology, their walk is only random in the first step; the remaining steps are all deterministic (because there is only one choice). In fact, wander join exactly degenerates to their algorithm when performed on this special type of joins. CS2 [54] extends join synopsis to the joins where the bijection requirement does not strictly hold. In this case, it takes a sample from the source relation and then takes *all* the joinable tuples from the rest of the relations. Again, there is only randomness in the source relation while the rest is deterministic.

Online aggregation is closely related to another line of work known as *query sampling* [8, 25, 42, 49]. 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 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 works on using sampling for approximate query processing, selectivity estimation, and query optimization [2, 3, 7, 15, 31–33, 35, 41, 48, 50, 53, 57, 58]. In particular, 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. iOLAP, Sampling+Seek, QuickR and others [2–4, 10, 27, 28, 34, 36, 41, 56–58]), but these systems do not support online aggregations over joins (or are limited to star schema or more restrictive types of joins).

9 FUTURE DIRECTIONS

This work has presented some promising results on wander join, a new approach to online aggregation for joins. It shows significantly better performance than state-of-the-art algorithms in different systems and settings. Yet, it has a lot of potential to be further exploited. Here we list a few directions for future work:

(1) 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.
(2) 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.
(3) Wander join does not have control on the distribution of its samples. Instead, it solely depends on the index structure. In extremely skewed cases, where there is one single extreme value, it might fail to sample that tuple, leading to inaccurate estimates. A way to remedy that is to build a small index that contains only the extreme values and always incorporate the extreme values into the estimates. However, that imposes more overhead in index building and maintenance and requires further investigation.
(4) Wander join currently only handles SPJA queries without nesting. In the case of nested queries, which are more common in realistic workloads, it is still unclear how to use wander join for that. One possible usage could be using wander join to estimate SPJA subquery that appears in a selection predicate and bound the errors in a similar fashion to bounding tuple uncertainty in iOLAP [56]. It would be interesting to see if wander join can improve the state-of-the-art in such cases.

ACKNOWLEDGMENTS

The authors greatly appreciate the valuable feedback provided by the anonymous TODS reviewers and Professor Christian S. Jensen in preparing this manuscript.

REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *ACM SIGMOD International Conference on Management of Data*.
[2] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael I. Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. 2014. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*. 481–492.
[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*. 29–42.
[4] Sameer Agarwal, Aurojit Panda, Barzan Mozafari, Anand P. Iyer, Samuel Madden, and Ion Stoica. 2012. Blink and it's done: Interactive queries on very large data. In *Proceedings of the VLDB Endowment*, Vol. 5.
[5] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. C. Kuszmaul, and J. Nelson. 2007. Cache-oblivious streaming B-trees. In *ACM Symposium on Parallelism in Algorithms and Architectures*.
[6] George Casella and Roger L. Berger. 2001. *Statistical Inference*. Duxbury Press.
[7] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1998. Random sampling for histogram construction: How much is enough? In *ACM SIGMOD International Conference on Management of Data*.
[8] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On random sampling over joins. In *ACM SIGMOD International Conference on Management of Data*.
[9] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2010. PR-join: A non-blocking join achieving higher early result rate with statistical guarantees. In *ACM SIGMOD International Conference on Management of Data*.
[10] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating aggregates with distribution precision guarantee. In *ACM SIGMOD International Conference on Management of Data*.

[11] Alin Dobra, Chris Jermaine, Florin Rusu, and Fei Xu. 2009. Turbo charging estimate convergence in DBO. In *International Conference on Very Large Data Bases*.

[12] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *NSDI*. 401–414.

[13] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree - Are hybrid physical designs important? In *ACM SIGMOD International Conference on Management of Data*.

[14] David Gembalczyk, Felix Martin Schuhknecht, and Jens Dittrich. 2017. An experimental analysis of different key-value stores and relational databases. In *Datenbanksysteme für Business, Technologie und Web (BTW'17), 17. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)*.

[15] Phillip B. Gibbons and Yossi Matias. 1998. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD International Conference on Management of Data*.

[16] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. 2012. Concurrency control for adaptive indexing. *PVLDB* 5, 7 (2012), 656–667.

[17] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, Stefan Manegold, and Bernhard Seeger. 2014. Transactional support for adaptive indexing. *VLDB J.* 23, 2 (2014), 303–328.

[18] P. J. Haas. 1997. Large-sample and deterministic confidence intervals for online aggregation. In *9th International Conference on Scientific and Statistical Database Management*.

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

[20] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Comput. System Sci.* 52 (1996), 550–569.

[21] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB* 5, 6 (2012), 502–513.

[22] Joseph M. Hellerstein, Ron Avnur, and Vijayshankar Raman. 2000. Informix under CONTROL: Online query processing. *Data Min. Knowl. Discov.* 4, 4 (2000), 281–314.

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

[24] D. G. Horvitz and D. J. Thompson. 1952. A generalization of sampling without replacement from a finite universe. *J. Amer. Statist. Assoc.* 47 (1952), 663–685.

[25] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent range sampling. In *ACM Symposium on Principles of Database Systems*.

[26] Stratos Idreos, Stefan Manegold, and Goetz Graefe. 2012. Adaptive indexing in modern database kernels. In *EDBT*. 566–569.

[27] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Chris Jermaine, and Peter J. Haas. 2011. The Monte Carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst.* 36, 3 (2011), 18.

[28] Prasanth Jayachandran, Karthik Tunga, Niranjan Kamat, and Arnab Nandi. 2014. Combining user interaction, speculative query execution and sampling in the DICE system. *PVLDB* 7, 13 (2014), 1697–1700.

[29] Christopher Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2007. Scalable approximate query processing with the DBO engine. In *ACM SIGMOD International Conference on Management of Data*.

[30] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2008. Scalable approximate query processing with the DBO engine. *ACM Trans. Database Syst.* 33, 4 (2008), Article 23.

[31] Ruoming Jin, Leonid Glimcher, Chris Jermaine, and Gagan Agrawal. 2006. New sampling-based estimators for OLAP queries. In *ICDE*. 18.

[32] Shantanu Joshi and Christopher M. Jermaine. 2008. Robust stratified sampling plans for low selectivity queries. In *ICDE*. 199–208.

[33] Shantanu Joshi and Christopher M. Jermaine. 2009. Sampling-based estimators for subset-based queries. *VLDB J.* 18, 1 (2009), 181–202.

[34] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily approximating complex AdHoc queries in BigData clusters. In *ACM SIGMOD International Conference on Management of Data*.

[35] Albert Kim, Eric Blais, Aditya G. Parameswaran, Piotr Indyk, Samuel Madden, and Ronitt Rubinfeld. 2015. Rapid sampling for visualizations with ordering guarantees. *PVLDB* 8, 5 (2015), 521–532.

[36] Anja Klein, Rainer Gemulla, Philipp Rösch, and Wolfgang Lehner. 2006. Derby/S: A DBMS for sample-based query answering. In *SIGMOD*.

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

[38] R. J. Lipton, J. F. Naughton, and D. A. Schneider. 1990. Practical selectivity estimation through adaptive sampling. In *ACM SIGMOD International Conference on Management of Data*.
[39] Gang Luo, Curt J. Ellmann, Peter J. Haas, and Jeffrey F. Naughton. 2002. A scalable hash ripple join algorithm. In *ACM SIGMOD International Conference on Management of Data*.
[40] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.
[41] Supriya Nirkhiwale, Alin Dobra, and Christopher M. Jermaine. 2013. A sampling algebra for aggregate estimation. *PVLDB* 6, 14 (2013), 1798–1809.
[42] F. Olken. 1993. *Random Sampling from Databases*. Ph.D. dissertation. University of California at Berkeley.
[43] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The case for RAMCloud. *Commun. ACM* 54, 7 (2011), 121–130.
[44] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. 2011. Online aggregation for large MapReduce jobs. In *Proceedings of the VLDB Endowment*, Vol. 4.
[45] Eleni Petraki, Stratos Idreos, and Stefan Manegold. 2015. Holistic indexing in main-memory column-stores. In *SIGMOD*. 1153–1166.
[46] Chengjie Qin and Florin Rusu. 2013. Parallel online aggregation in action. In *SSDBM*. 46:1–46:4.
[47] Chengjie Qin and Florin Rusu. 2014. PF-OLA: A high-performance framework for parallel online aggregation. *Distrib. Parallel Databases* 32, 3 (2014), 337–375.
[48] David Vengerov, Andre Cavalheiro Menck, and Mohamed Zait. 2015. Join size estimation subject to filter conditions. In *International Conference on Very Large Data Bases*.
[49] Lu Wang, Robert Christensen, Feifei Li, and Ke Yi. 2016. Spatial online sampling and aggregation. In *International Conference on Very Large Data Bases*.
[50] Mingxi Wu and Chris Jermaine. 2006. Outlier detection by sampling with accuracy guarantees. In *SIGKDD*. 767–772.
[51] Sai Wu, Shouxu Jiang, Beng Chin Ooi, and Kian-Lee Tan. 2009. Distributed online aggregation. *PVLDB* 2, 1 (2009), 443–454.
[52] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. 2010. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*. 651–662.
[53] Fei Xu, Christopher M. Jermaine, and Alin Dobra. 2008. Confidence bounds for sampling-based group by estimates. *ACM Trans. Database Syst.* 33, 3 (2008).
[54] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: A new database synopsis for query estimation. In *ACM SIGMOD International Conference on Management of Data*.
[55] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. 2015. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*. 913–918.
[56] Kai Zeng, Sameer Agarwal, and Ion Stoica. 2016. iOLAP: Managing uncertainty for efficient incremental OLAP. In *ACM SIGMOD International Conference on Management of Data*.
[57] Kai Zeng, Shi Gao, Jiaqi Gu, Barzan Mozafari, and Carlo Zaniolo. 2014. ABS: A system for scalable approximate queries with accuracy guarantees. In *SIGMOD*. 1067–1070.
[58] Kai Zeng, Shi Gao, Barzan Mozafari, and Carlo Zaniolo. 2014. The analytical bootstrap: A new method for fast error estimation in approximate query processing. In *SIGMOD*. 277–288.

Received January 2017; revised June 2018; accepted October 2018