Wander Join and XDB: Online Aggregation via Random Walks

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

ABSTRACT

Joins are expensive, and online aggregation is an effective approach to explore the 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 needs strong assumptions (e.g., the tuples in a table are stored in random order). This paper proposes a new approach, the *wander join* algorithm, to the online aggregation problem by performing random walks over the underlying join graph. We also design an optimizer that chooses the optimal plan for conducting the random walks without having to collect any statistics *a priori*. Selection predicates and group-by clauses can be handled as well. We have developed an online engine called XDB by integrating wander join in the latest version of PostgreSQL. Extensive experiments using the TPC-H benchmark have shown the superior performance of wander join. The XDB implementation has demonstrated its practicality in a full-fledged database system.

1. INTRODUCTION

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

The research community has long realized the need for interactive data analysis and exploration, and in 1997, began a line of work known as "online aggregation" [7]. 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.

Unfortunately, despite of many nice research results and well cited papers on this topic, online aggregation has had limited practical impact — we are not aware of any full-fledged, publicly available database system that supports it. Central to this line of work is the ripple join algorithm [5]. 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 actually join. However, we observe that this fraction is often exceedingly low, especially for equality joins (a.k.a. natural joins) involving multiple tables, while *all* queries in the TPC-H benchmark (thus arguably most joins used in practice) are natural joins. (2) It demands that the tuples in each table be stored in a random order.

This paper proposes a different approach, which we call *wander join*, to the online aggregation problem. Our basic idea is to not blindly take samples from each table and just hope that they join, but to make the process much more focused by leveraging indexes. Specifically, wander join takes a randomly sampled tuple only from one of the tables. After that, it conducts a random walk using indexes on the underlying join graph 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. To summarize:

- We introduce *wander join* to achieve 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.
- 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*.

- We have conducted extensive experiments to compare wander join with ripple join [5] and its system implementation DBO [2, 10]. The results show that wander join has outperformed ripple join and DBO by orders of magnitude in speed for achieving the same accuracy for in-memory data.
- We have implemented XDB by integrating wander join in PostgreSQL. On the TPC-H benchmark with tens of GBs of data, XDB 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.

2. BACKGROUND

Online aggregation. The concept of online aggregation was first proposed in the classic work by Hellerstein et al. [7]. 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 to terminate the query execution whenever a good approximation quality is met.

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 [4].

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. [4,6] studied this problem in depth, and derived new formulas for computing the confidence intervals for such estimators, and later proposed the *ripple join* algorithm [5]. Ripple join repeatedly takes random samples from each table in a round-robin fashion, and keep all the sampled tuples in memory. Every time a new tuple is taken from one table, it is joined with all the tuples taken from other tables so far.

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 [12, 13]. In general, for a multi-table join $R_1 \bowtie \cdots \bowtie R_k$, the index ripple join algorithm only does random sampling on one of the tables, say R_1. Then for each tuple t sampled from R_1, it computes $t \bowtie R_2 \bowtie \cdots \bowtie R_k$, and all the joined results are returned as samples from the full join.

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

`SELECT g, AGG(expression) FROM` $R_1, R_2, \ldots, R_k$
`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 or inequality conditions between pairs of the tables, and `selection predicates` can also be applied to any number of the tables. For example, in the following query, the first three terms in the `WHERE` clause are join conditions while the others are selection predicates:

`SELECT SUM(l_extended_price` $*$ `(1 −` `l_discount))`
`FROM` *nation*, *customer*, *orders*, *lineitem*
`WHERE n_nationkey = c_nationkey AND c_custkey = o_custkey`
`AND o_orderkey = l_orderkey AND n_name =′ US′ AND l_flag =′ R′`

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

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

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

3. WANDER JOIN

3.1 Wander join on a simple example

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

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

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

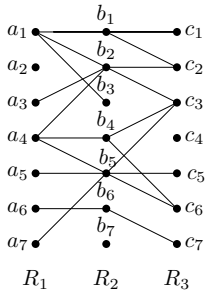


Figure 1: The 3-table join data graph: there is an edge between two tuples if they can join. Note that this represents a join query with general join conditions that are not necessarily natural/equi-join.

We model the join relationships among the tuples as a graph. More precisely, each tuple is modeled as a vertex and there is an edge between two tuples if they can join.

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

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

One problem an acute reader would immediately notice is that, different paths may have different probabilities. In the example above, the path $a_1 \to b_1 \to c_1$ has probability $\frac{1}{7}\cdot\frac{1}{3}\cdot\frac{1}{2}$, while $a_6 \to b_6 \to c_7$ has probability $\frac{1}{7}\cdot 1\cdot 1$. 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* [8] 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

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

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

Finally, we independently perform multiple random walks, and take the average of the estimators $v(\gamma_i)/p_i$. Since each $v(\gamma_i)/p_i$ is an unbiased estimator of the `SUM`, their average is still unbiased, and the variance of the estimator reduces as more paths are collected.

A subtle question is what to do when the random walk gets stuck, for example, when we reach vertex b_3 in Figure 1. In this case, we should not reject the sample, but return 0 as the estimate, which will be averaged together with all the successful random walks. This is because even though this is a failed random walk, it is still in the probability space. It should be treated as a value of 0 for the Horvitz-Thompson estimator to remain unbiased. Too many failed random walks will slow down the convergence of estimation, and we will deal with the issue in Section 4.

3.2 Wander join for acyclic queries

Although the algorithm above is described on a simple 3-table *chain join*, it can be extended to arbitrary joins easily. In general, we consider the *join query graph* (or *query graph* in short), where each table is modeled as a vertex, and there is an edge between two tables if there is a join condition between the two; see Figure 2 for examples.

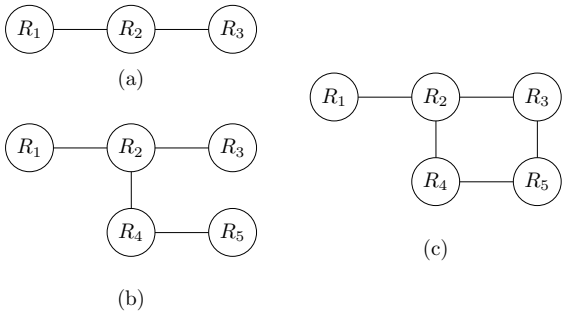


Figure 2: The join query graph for a (a) chain join; (b) acyclic join; (c) cyclic join.

When the join query graph is acyclic, wander join can be extended in a straightforward way. First, we need to fix a *walk order* such that each table in the walk order must be adjacent (in the query graph) to another one earlier in the order. For example, for the query graph in Figure 2(b), R_1, R_2, R_3, R_4, R_5 and R_2, R_3, R_4, R_5, R_1 are both valid walk orders, but R_1, R_3, R_4, R_5, R_2 is not since R_3 (resp. R_4) is not adjacent to R_1 (resp. R_1 or R_3) in the query graph. (Different walk orders may lead to very different performance, and we will discuss how to choose the best one in Section 4.)

Next, we simply perform the random walks as before, following the given order. The only difference is that a random walk may now consist of both "walks" and "jumps". For example, using the order R_1, R_2, R_3, R_4, R_5 on Figure 2(b), after we have reached a tuple in R_3, the next table to walk to is R_4, which is connected to the part already walked via R_2. So we need to jump back to the tuple we picked in R_2, and continue the random walk from there.

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

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

3.3 Wander join for cyclic queries

The algorithm for acyclic queries can also be extended to handle query graphs with cycles. Given a cyclic query graph, e.g., the one in Figure 2(c), we first find any spanning tree, such as the one in Figure 2(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 satisfies the join conditions on these edges. For example, after we have sampled a path $\gamma = (t_1, t_2, t_3, t_4, t_5)$ on Figure 2(b) (assuming the walk order R_1, R_2, R_3, R_4, R_5), then we need to verify that γ satisfies the non-spanning tree edge (R_3, R_5), i.e., t_3 should join with t_5. If they do not join, we consider γ as a failed random walk and return an estimator with value 0.

3.4 Estimators and confidence intervals

To derive estimators and confidence interval formulas for various aggregation functions, we establish an equivalence between wander join and sampling from a single table with selection predicates, which has been studied by Haas [4]. Imagine that we have a single table that stores all the paths in the join data graph, including the full paths, as well as the partial paths (like $a_1 \to b_3$). Wander join essentially samples from this imaginary table, though non-uniformly.

Suppose we have performed a total of n random walks $\gamma_1, \ldots, \gamma_n$. For each γ_i, let $v(i)$ be the value of the expression on γ_i to be aggregated, and set $u(i) = 1/p(\gamma_i)$ if γ_i is a successful walk, and 0 otherwise. With this definition of u and v, we can rewrite the estimator for `SUM` as $\frac{1}{n}\sum_{i=1}^{n} u(i)v(i)$. We observe that this has exactly the same form as the one in [4] for estimating the `SUM` for a single table with a selection predicate, except for two differences: (1) in [4], $u(i)$ is set to 1 if γ_i satisfies the selection predicate and 0 otherwise; and (2) [4] does uniform sampling over the table, while our sampling is non-uniform. However, by going through the analysis in [4], we realize that it holds for any definition of u and v, and for any sampling distribution. Thus, all the results in [4] carry over to our case, but with u and v defined in our way. The detailed formulas can be found in [11]; all of them can be computed easily in $O(n)$ time.

3.5 Selection predicates and Group By Clause

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

If the starting table of the random walk has an index on the attribute with a selection predicate, and the predicate is an equality or range condition, then we can directly sample a tuple that satisfies the condition from the index, using Olken's method [14]. Correspondingly, we replace $|R_{\lambda(1)}|$ in (3) by the number of tuples in $R_{\lambda(1)}$ that satisfy the condition, which can also be computed from the index. This removes the impact of the predicate on the performance of the random walk, thus it is preferable to start from such a table. More discussion will be devoted on this topic under walk plan optimization in Section 4.

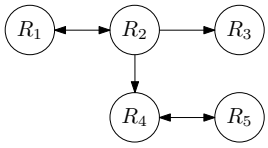
Wander join supports a Group By clause by maintaining multiple estimators simultaneously during the random walk process, one per group with respect to the grouping attribute(s). Each random walk is pushed to the group it belongs to and used to update the corresponding estimator.

4. WALK PLAN OPTIMIZER

Different orders in which to perform the random walk may lead to very different performance. 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), there must exist a table R_j earlier in the order such that there is a join condition between R_i and R_j. In addition, R_i should have an index on the attribute that appears in the join condition. Note that the join condition does not have to be equality. It can be for instance an inequality or even a range condition, such as $R_j.A \le R_i.B \le R_j.A + 100$, as long as R_i has an index on B that supports range queries (e.g., a B-tree).



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

Figure 3: A directed join query graph and all its walk plans.

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 2(b) might look like the one in Figure 3, and all possible walk plans are listed on the side. These plans can be enumerated by a simple backtracking algorithm. Note that there can be exponentially (in the number of tables) many walk plans. However, this is not a real concern because (1) there cannot be too many tables, and (2) more importantly, having many walk plans does not have a major impact on the plan optimizer, which we shall see later.

We can similarly generate all possible walk plans for cyclic queries, just that some edges will not be walked, and they will have to be checked after the random walk, as described in Section 3.3. We call them *non-tree* edges, since the part of the graph that is covered by the random walk form a tree. An example is given in Figure 4.

4.2 Walk plan optimization

The performance of a walk order depends on many factors. However, 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 $\frac{1}{W}\sum_{i=1}^{W} X_i$. We show that

$$\mathsf{Var}\left[\frac{1}{W}\sum_{i=1}^{W} X_i\right] = \mathsf{Var}[X_1]\mathsf{E}[T]/t.$$

Thus, for a given amount of time t, the variance of the final estimator is proportional to $\mathsf{Var}[X_1]\mathsf{E}[T]$. The next observation is that both $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$ can also be estimated

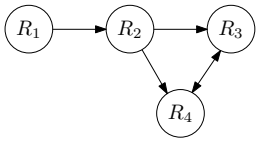


Figure 4: Walk plan for a cyclic query graph.

by the random walks themselves! In particular, $\mathsf{Var}[X_1]$ is just estimated as another aggregation function; for $\mathsf{E}[T]$, we simply count the number of index entries looked up, or the number of I/Os in external memory, in each random walk, and take the average.

Now, for each walk order, we perform a certain number of "trial" random walks and estimate $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$. Then we compute the product $\mathsf{Var}[X_1]\mathsf{E}[T]$ and pick the order with the minimum $\mathsf{Var}[X_1]\mathsf{E}[T]$. How to choose the number of trials is the classical sample size determination problem [1], which again depends on many factors such as the actual data distribution, the level of precision required, etc. We adopt the following strategy: We conduct random walks following each plan in a round-robin fashion, and stop until at least one plan has accumulated at least τ successful walks. Then we pick the plan with the minimum $\mathsf{Var}[X_1]\mathsf{E}[T]$ that has at least $\tau/2$ successful walks. This is actually motivated by association rule mining, where a rule must both be good and have a minimum support level. In our implementation, we use a default threshold of $\tau = 100$.

Finally, we observe that all the trial runs are not wasted. Since each random walk, no matter which plan it follows, returns an unbiased estimator. So we can include all the random walks, before and after the optimal one has been picked, in computing the final estimator. The confidence interval is also computed with all these random walks. This is unlike traditional query optimization, where the cost incurred by the optimizer itself is pure "overhead".

5. XDB: INTEGRATING WANDER JOIN INSIDE A DBMS ENGINE

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 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 the attributes used in the selection predicates. XDB is now open-sourced at `https://github.com/initialDLab/xdb`.

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 1000 milliseconds for up to 20000 milliseconds.

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 was also developed as part of the XDB system, where its visualization module was modified so that an online visualization of the (continuously updated) query results as well as the confidence intervals is enabled.

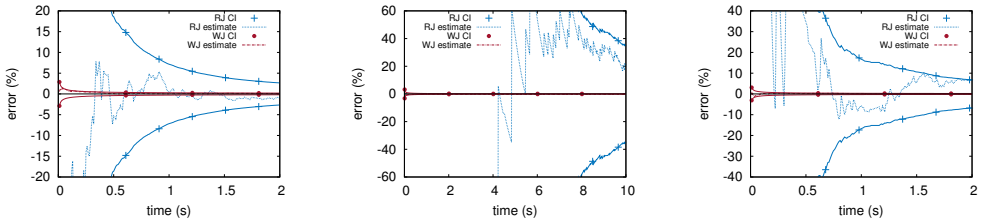
The only system implementation available for ripple join is the DBO system [2, 9, 10]. In fact, the algorithm implemented in DBO is much more complex than the basic ripple join in order to deal with limited memory, as described in these papers. We compared XDB with Turbo DBO, using the code at `http://faculty.ucmerced.edu/frusu/Projects/DBO/dbo.html`, as a system-to-system comparison. Note that due to the random order storage requirement, DBO was built from ground up. Currently it is still a prototype that supports online aggregation only (i.e., no support for other major features in a RDBMS engine, such as transactions, locking, etc.). On the other hand, XDB retains the full functionality of a RDBMS, with online aggregation 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 other features and functionality.

Note that the original DBO papers [9] 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 using XDB (which is a PostgreSQL with wander join implemented inside its kernel).

6. EXPERIMENTS

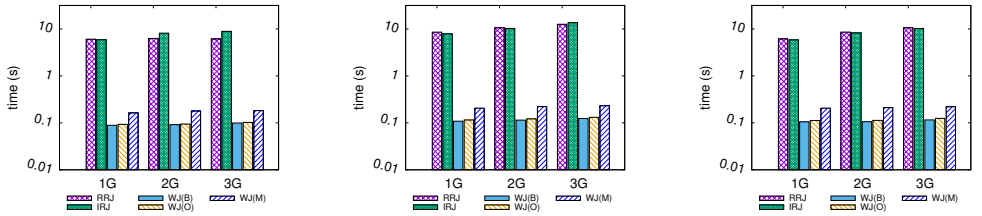
6.1 Experimental setup

We have evaluated the performance of wander join in comparison with ripple join and its variants, the DBO engine, under two settings: using a standalone implementation and a system implementation (XDB) respectively. In the stan-



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

Figure 5: Standalone implementation: Confidence intervals and estimates on barebone queries on 2GB TPC-H data set; confidence level is 95%.



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

Figure 6: Standalone implementation: Time to reach ±1% confidence interval and 95% confidence level on TPC-H data sets of different sizes.

dalone setting, we implemented both wander join and ripple join in C++. We ensure that the index structures fit in memory; in fact, all the indexes together take space that is a very small fraction of the total amount of data, because they are all secondary indexes, storing only pointers to the actual data records, which have many other attributes that are not indexed. Building these indexes is very efficient; in fact, they can be built with minimal overhead while loading data from the file system to the memory (which one has to do anyway). Similarly, for ripple join, we gave it enough memory so that all samples taken can be kept in memory.

The standalone implementation gives an ideal environment to both algorithms without any system overhead.

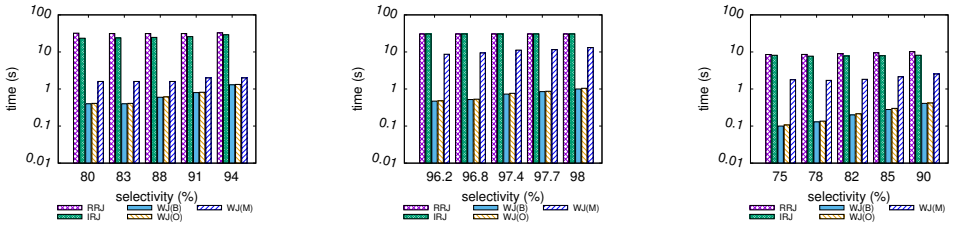
Data and queries. We used the TPC-H benchmark data and queries for the experiments, which were also used by the DBO work [2, 9, 10]. We used 5 tables, `nation`, `supplier`, `customer`, `orders`, and `lineitem`. We used the TPC-H data generator with the appropriate scaling factor to generate data sets of various sizes. We picked queries Q3 (3 tables), Q7 (6 tables; the `nation` table appears twice in the query) and Q10 (5 tables) in the TPC-H specification for testing.

Effect of data skewness. There are two types of skewness. Degree skewness refers to the skewness in the distribution of the number of tuples in one table that join another, while value skewness is the skewness of the distribution of the values being aggregated. The degree skewness will negatively impact the random walk process of wander join if a good walk order is not selected. This issue is addressed by our walk order optimization. Depending on how the cardinality of the join changes, it usually has no impact or even positive impact on the efficiency of wander join. In contrast, degree skewness often leads to worse performance for ripple join due to the increasing join sparsity for most tuples. On the other hand, value skewness has a negative impact on all online aggregation methods because the higher variance of aggregated values leads to a larger variance of the estimator. Unless prior knowledge of the value distribution is available, the effectiveness of (any) sampling methods will be affected.

6.2 Results on standalone implementation

We first run wander join (WJ) and ripple join (RJ) on a 2GB data set, i.e., the entire TPC-H database is 2GB, using the "barebone" joins of Q3, Q7, and Q10, where we drop all the selection predicates. Figure 5 plots 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 (obtained offline by running the exact joins to full completion). We can see that WJ converges much faster than RJ, due to the much more focused exploration strategy. Meanwhile, the estimates returned are indeed within the confidence interval almost all the time. For example, wander join converges to 1% confidence interval in less than 0.1 second whereas ripple join takes more than 4 seconds to reach 1% confidence interval. The full exact join on Q3, Q7, and Q10 in this case is 18 seconds, 28 seconds, and 19 seconds, respectively, using hash join.

Next, we ran the same queries on data sets of varying sizes. Now we include both the random order ripple join (RRJ) and the index-assisted ripple join (IRJ). For wander join, we also considered two other versions to see how the plan optimizer 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 6 we report the time spent by each algorithm to reach ±1% confidence interval with 95% confidence level on data sets of sizes 1GB, 2GB, and 3GB. We also report the time costs of the optimizer in Table 1. From the results, we can draw the following observations: (1) Wander join is in general faster than ripple join by two orders of magnitude to reach the same confidence interval. (2) The running time of ripple join increases with N, the data size, though mildly. (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

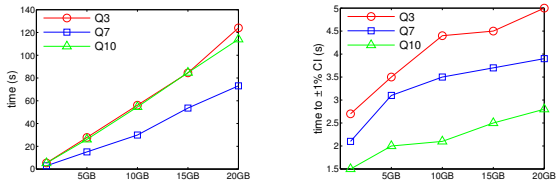


(a) Q3 (b) Q7 (c) Q10

Figure 7: Standalone implementation: Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H data set with multiple selection predicate of varying selectivity.

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

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



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

Figure 8: XDB: system implementation experimental results with sufficient memory – 32GB memory.

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

Finally, we put back the selection predicates to the queries. Figure 7 shows the time to reach ±1% confidence interval with 95% confidence level for the algorithms on the 2GB data set with all the predicates are put back. Here, we measure the overall selectivity of all the predicates as:

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

so higher means more selective.

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

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

6.3 Results on system implementation

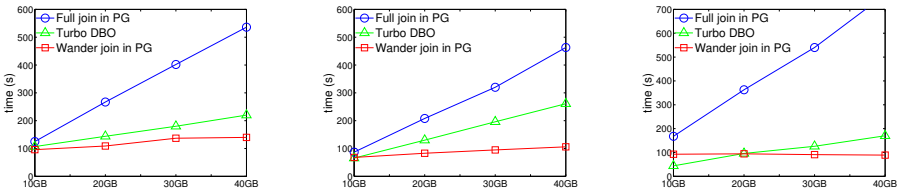
For the experimental evaluation on XDB, which is our PostgreSQL integration and implementation of wander join, we first tested how it performs when there is sufficient memory, and then tested the case when memory is severely limited. We compared against Turbo DBO in the latter case. Turbo DBO [2] is an improvement to the original DBO engine, that extends ripple join to data on external memory with many optimizations.

When there is sufficient memory. Due to the low-latency requirement for data analytical tasks and thanks to growing memory sizes, database systems are moving towards the "in-memory" computing paradigm. So we first would like to see how our system performs when there is sufficient memory. For this purpose, we used a machine with 32GB memory and data sets of sizes up to 20GB. We ran both online version of XDB and the built-in PostgreSQL full join in XDB on the same queries, both through the standard PostgreSQL SQL query interface.

Note that since we have built indexes on all the join attributes and there is sufficient memory, the PostgreSQL optimizer chose index join for all the join operators. We used Q3, Q7, and Q10 with all the selection predicates.

The results in Figure 8 clearly indicate a linear growth of the full join, which is as expected because the index join algorithm has running time linear in the table size. Also because all joins are primary key-foreign key joins, the intermediate results have roughly linear size. On the other hand, the data size has a mild impact on the performance of wander join. For example, the time to reach ±1% confidence interval for Q7 merely increases from 3 seconds to 4 seconds, when the data size increases from 5GB to 20GB in Figure 8(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, the cost per random walk step might grow slightly. Nevertheless, PostgreSQL with wander join reaching 1% CI has outperformed the PostgreSQL with full join by more than one order of magnitude when data size grows.

We have also run Turbo DBO in this case. However, it turned out that Turbo DBO spends even more time than PostgreSQL's full join, so we do not show its results. This



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

Figure 9: XDB: system implementation experimental results with limited memory – 4GB memory.

seems to contradict with the results in [10]. In fact, this is because DBO is intentionally designed for large data and small memory. In the experiments of [10], the machine used had only 2GB of memory. With such a small memory, PostgreSQL had to resort to sort-merge join or nested-loop join for each join operator, which is much less efficient than index join (for in-memory data). Meanwhile, DBO follows the framework of sort-merge join, so it is actually not surprising that it is not as good as index joins for in-memory data. In our next set of experiments where we limit the memory size, we do see that DBO performs better than the full join.

When memory is limited. In our last set of experiments, we used a machine with only 4GB memory, and ran the same set of experiments as above on data sets of sizes starting from 10GB and increasing to 40GB. The time for wander join inside PostgreSQL and Turbo DBO to reach ±5% confidence interval with 95% confidence level, as well as the time of the full join in PostgreSQL, are shown in Figure 9.

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 8 to more than 100 seconds in Figure 9, and that's 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 DBO clearly have a linear dependency on the data size, though at different rates. On the 10GB and 20GB data sets, wander join and DBO have similar performance, but eventually wander join would stand out on very large data sets.

Anyway, spending 100 seconds just to get a ±5% estimate does not really meet the requirement of interactive data analytics, so strictly speaking both wander join and DBO have failed in this case (when data has significantly exceeded the memory size). Fundamentally, online aggregation requires some form of randomness so as to have a statistically meaningful estimation, which is at odds with the sequential access nature of hard disks. This appears to be an inherent barrier for this line of work. However, as memory sizes grow larger and memory clouds get more popular (for example, using systems like RAMCloud [15] and FaRM [3]), with the SSDs as an additional storage layer, in the end we may not have to deal with this barrier at all.

7. CONCLUSION

We have open sourced the XDB engine at `https://github.com/InitialDLab/XDB`. In addition to the integration with the PostgreSQL kernel, we have also designed and implemented a front-end interface using Apache Zeppelin, which is able to show the query results in the form of table, line plot and other visualization representation in a continuous online fashion. For future work, an important open problem is to extend online aggregations to nested queries.

8. ACKNOWLEDGMENTS

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. The authors greatly appreciate the valuable feedback provided by the anonymous SIGMOD reviewers and Professor Jeffrey Naughton in preparing this manuscript.

9. REFERENCES

[1] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, 2001.
[2] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo charging estimate convergence in dbo. In *VLDB*, 2009.
[3] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.
[4] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, 1997.
[5] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.
[6] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *JCSS*, 52:550–569, 1996.
[7] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.
[8] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *JASA*, 47:663–685, 1952.
[9] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2007.
[10] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. *ACM TODS*, 33(4), Article 23, 2008.
[11] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, 2016.
[12] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *PODS*, 1990.
[13] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, 1990.
[14] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.
[15] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, 2011.