Multi-Agent Join

Vahid Ghadakchi
Oregon State University
ghadakcv@oregonstate.edu

Mian Xie
Oregon State University
xiemia@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

Bakhtiyar Doskenov
Oregon State University
doskenob@oregonstate.edu

Bharghav Srikhakollu
Oregon State University
srikhakb@oregonstate.edu

Summit Haque
Oregon State University
haquesu@oregonstate.edu

Huazheng Wang
Oregon State University
huazhengwang@gmail.com

Abstract

Real-time performance is crucial for interactive and exploratory data analysis, where users require quick access to subsets or progressive presentations of query results. Delivering real-time results over large data for common relational binary operators like join is challenging, as join algorithms often spend considerable time scanning and attempting to join parts of relations that may not produce any results. Existing solutions often involve repetitive preprocessings, which are costly and may not be feasible for interactive workloads or evolving datasets. Additionally, these solutions may support only restricted types of joins. This paper presents a novel approach for achieving efficient progressive join processing. The scan operator of the join learns online during query execution, identifying portions of its underlying relation that satisfy the join condition. Additionally, an algorithm is introduced where both scan operators collaboratively learn to optimize join execution.

1 Introduction

Providing real-time performance for relational data queries is crucial in various applications Carey and Kossmann (1997, 1998); Hellerstein et al. (1997); Haas and Hellerstein (1999); Li et al. (2016). Users often require fast answers for each query in a sequence to expedite data analysis tasks. However, with the substantial and rapidly growing volume of data, achieving highly efficient query processing is challenging Dimitriadou et al. (2014); Idreos et al. (2015). This challenge is particularly pronounced when dealing with queries over multiple relations, such as *join* or *intersection*, as algorithms may need to explore a vast space of possible tuple combinations, resulting in significant computational overhead for queries with conditions like join predicates.

Existing algorithms for returning subsets of query results quickly often demand substantial computational resources, significant preprocessing time, or are limited to specific query types Hellerstein et al. (1997); Haas and Hellerstein (1999); Jermaine et al. (2005); Li et al. (2016). For example, *Nested loop ripple Join* can run out of main memory for large relations Haas and Hellerstein (1999); Jermaine et al. (2005), while methods utilizing auxiliary data structures or indexes are time-consuming and require user expertise Li et al. (2016); Chaudhuri and Narasayya (2007); Aken et al. (2021). Sort- and hash-based approaches are constrained to specific query types and may need substantial

Machine Learning for Systems Workshops at 37th NeurIPS 2023.

memory Jermaine et al. (2005). With the rising complexity of join predicates, including user-defined functions, and the need for efficient processing of large datasets, there is a growing demand for more resource-efficient algorithms.

We propose a novel approach to join processing for delivering real-time insights over large data. In this approach scan operators collectively learn during *query execution*, prioritizing promising data portions and efficiently producing subsets of join results without upfront data preparation. This method eliminates the need for offline training or precomputed data structures, accommodating both stationary and non-stationary query workloads while minimizing initial delays in result generation. The proposed algorithm currently works on join queries with two relations but further advancement is possible to scale it to multiple joins making all the scan operators learn in similar way.

2 Framework

We present the framework that models processing of a join query method as online collaborative learning of its operators.

Agents and Actions. Relational data Systems usually model a query as a tree of logical operators called *logical query plan* Garcia-Molina et al. (2009). In a join query, the leaf nodes are scan operators that read information from the secondary storage in blocks (pages). Each *iteration* (or *round*) of processing the query starts with the call from the join operator. In our framework, each scan operator in the query plan is a learning *agent*. The *actions* of an agent as the set of its available activities in each round. The set of actions of each scan operator is the set of blocks in its base relation. Figure 1 shows the query plan for a join between two relations.

Reward. The *reward* of an action of an operator is the total number of output tuples produced if this action is chosen by the operator. For example, in Figure 1, the reward of reading a tuple by a scan operator is the number of joint tuples produced involving this tuple. After an operator selects an action, the operator sends the intermediate results produced by this action to its parent operators and eventually the root of the plan. The root operator recursively shares the reward of this action with its child operators, e.g., in Figure 1 the join operator sends reward information to scan operators.

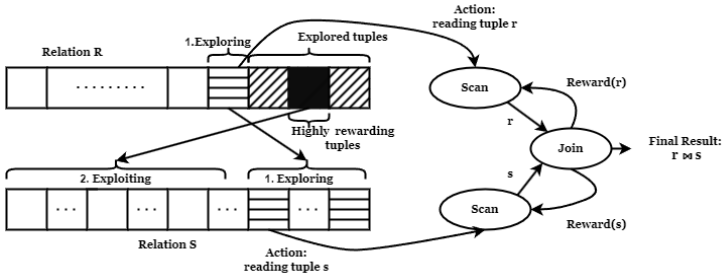

Figure 1: Framework for learning join strategy online

Strategies. The *strategy* of an operator is a mapping from its past actions and rewards to its available actions. Some operators have *fixed strategies* that don't change during query execution, while others use *adaptive strategies*, selecting actions based on past performance and rewards. Adaptive strategies become more beneficial when dealing with queries involving many rounds.

Learning strategies. Operators learn action rewards during query execution by *exploring* initially and then alternating between *exploration* of new actions and *exploitation* of known rewarding ones to maximize both short-term and long-term rewards. An important challenge in online learning is to find a balance between exploration and exploitation. Exploring for too long may hinder learning by investing excessive time in a single block, while exploring too briefly risks missing beneficial blocks.

Example 1. Let us review a simple approach to learning query processing. Consider a query that joins relations R and S in Figure 1. To identify rewarding tuples, the scan operators on R and S which we can call R-scan and S-scan respectively may first sequentially scan small subsets of these relations and send them to the join operator. The join operator joins tuples in the explored subsets of R and S and informs the scans of their tuples' rewards. Using this sample of join attempts, the R-scan may learn of highly rewarding tuples in the relation. R-scan exploits this knowledge by sending highly rewarding tuples of R to the join operator to join with the remaining tuples of S where the S-scan sequentially reads tuples in S. The R-scan iteratively explores and then exploits other rewarding tuples in R.

Challenges and overheads. To learn the rewards of explored tuples accurately, the early join successes must be an unbiased sample. It may also take a while to learn the reward of each tuple. The algorithms does not exploit its knowledge quickly, i.e., only after the complete join of the explored subsets. Furthermore, many join attempts between highly rewarding tuples of R and tuples of S might still not produce any result.

3 Single scan learning

In this section, we focus on learning for the R-scan operator in the join of R and S in Figure 1, assuming a fixed strategy of reading randomly chosen tuples for S-scan. It is shown that in the absence of any order, e.g., sorted relation, sequential scan, i.e., heap-scan, simulate random sampling effectively Haas and Hellerstein (1999). R-scan aims to quickly and accurately learn the rewards of tuples in R, joining them with S in decreasing order of rewards to efficiently produce results and minimize user waiting time as explained in Section 2. Learning occurs during join processing, as R-scan lacks prior knowledge of tuple rewards. The algorithm operates in super-rounds, where it learns and selects the most rewarding unselected tuples from R to generate join results, continuing until a specified number of tuples or complete results are achieved based on user input. In Appendix A the Algorithm 1 is presented to clarify what occurs in each super-round.

Learning most rewarding tuple. Utilizing the *many-armed bandit (MAB)* algorithm M*-Run* Berry et al. (1997), our algorithm employs N-Failure exploration technique. During each *round*, S-scan reads a fresh tuple from S, while R-scan, if its current tuple produced joint tuples in the last N rounds, continues sending it to the join operator; otherwise, it reads the next tuple of R. The reward for each R tuple is the number of joint tuples during N-Failure exploration, stored in a reward table in main memory. R-scan performs N-Failure for each R tuple until scanning M tuples or finding a tuple producing at least one join tuple in each of the last M rounds. At this point, R-scan and S-scan fully *exploit* the most rewarding R tuple by performing a full join. R-scan selects the highest reward tuple from the reward table, reads it from disk using random access, and sends it to the join operator. S-scan resets its *sequential scan* from the beginning of S, reading and sending each tuple to the join operator. To handle sparse rewards, R-scan may pick the last scanned tuple in main memory for the full join in some *super-rounds*, updating the reward table to track fully exploited tuples.

Subsequent super-rounds. After each super-round, R-scan excludes the most rewarding tuple from its available actions, resuming sequential scanning from the last stopped position. It reads and sends the next unread tuple of R to the join operator, explores its reward with the N-Failure technique, and updates the reward table in main memory if the reward is non-zero. S-scan continues sequential scanning of S during this N-Failure exploration. In each super-round after the first, R-scan estimates the new most rewarding tuple by reading only one new tuple from R since it has enough information after reading one tuple to select the most rewarding tuple from the reward table. It then accesses the selected tuple using its address in the reward table.

4 Collaborative scans

In m-to-n joins of R and S, employing learning strategies on both R-scan and S-scan may optimize response time but doubles I/O accesses, significantly slowing join processing. To address this, interleaving learning and sampling is proposed, treating explored tuples as both scan actions and random samples from the base relation. Using the M-Run learning method, each exploration involves sending both available actions and random samples to the join operator. In each super-round, one scan performs N-Failure exploration, while the other performs sequential scan, switching strategies for subsequent rounds. The join stops after reaching the desired tuple count. Due to switching, a scan may not immediately resume N-Failure after the last explored tuple, potentially skipping exploration.

5 Preliminary empirical results

Methods. We evaluate our proposed methods and the comparable methods explained in Section 1. We compare our methods to *Nested Loop* join (NL) Garcia-Molina et al. (2009) and *Sort-Merge-Shrink* (SMS) join Jermaine et al. (2005). SMS is a sort-based approach to generate a subset of results of equi-joins progressively and quickly. Empirical studies indicate that SMS outperforms hash-based algorithms Jermaine et al. (2005). Although SMS is not fully comparable to ours for not supporting non-equijoins, to better understand the performance of our methods, we compare them to SMS.

Data. We use TPC-H benchmark *www.tpc.org/tpc*, to generate the queries and databases for our experiments. We generate a TPC-H database of size 50 GB. The join processing time is influenced by the skewness of join attribute values. We evaluate the query run-time over different versions of TPC-H database with different degrees of skewness in the join attributes. Our experiments use Zipf distributions with z values ranging between $[0, 1, 1.5]$ wherein $z = 0$ will result in uniform

distribution and for $z \geq 0.5$, the distribution becomes more skewed as the value of z grows. We run each query 3 times by randomly shuffling orders of tuples and report the average. The reported times include both learning and execution.

Queries. We have used the following M-N joins, Q_9:$partsupp \bowtie_{partkey} lineitem$ and Q_{11}: $orders \bowtie_{o\_orderdate=l\_shipdate} lineitem$ and the following 1-N join queries, Q_{10} : $customer \bowtie_{custkey} orders$, Q_{12} : $orders \bowtie_{orderkey} lineitem$, Q_{15} : $supplier \bowtie_{suppkey} lineitem$ from TPC-H workload. To evaluate the performance of our methods over joins with complex conditions, we modify the join predicates of these queries to resemble *similarity joins*. We modify the join predicate of TPC-H queries and use *Levenshtein distance* instead of equality to implement approximate match between join attributes and perform *non-equal joins*.

Platform. We have performed the empirical study inside PostgreSQL 11.5 database management system on a Linux server with Intel(R) Xeon(R) 2.30GHz cores and 500GB of memory. We have used query Q_{12} : $orders \bowtie_{orderkey} lineitem$ to train the hyper-parameters of the proposed algorithms, e.g,. N.

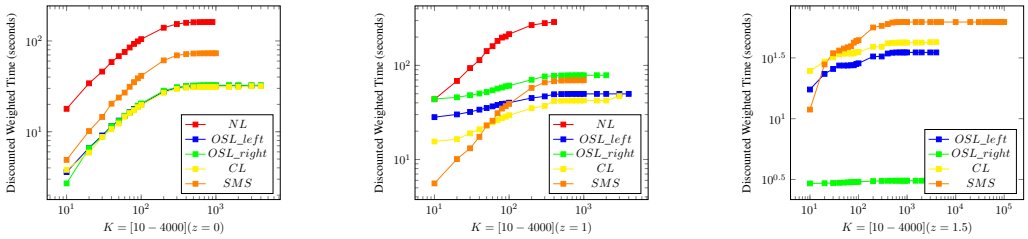

Figure 2: Discounted weighted average times for CL, OSL_right, OSL_left, SMS, and NL over Q_9.

In Figure 2 we compare collaborative learning (CL), single scan learning (OSL_left and OSL_right), NL, and SMS using an M-N equi-join query Q_9 using log-log graphs for three different z values. There we plotted the *discounted weighted average times* of different approaches for different *output sample size (K)*. Here, discounted weighted average time measures the users' waiting time for receiving samples of results and sample size means the required amount of joint tuples to be generated from the query. Here, for all three different z values, at least one approach among OSL_left, OSL_right, and CL is performing better than both SMS and NL. NL does not produce any results over the dataset with $z = 1.5$. The reported results for SMS use significantly larger memory (4GB) than other methods (0.5GB). The differences between the learning algorithms with NL and SMS become generally more significant as the skewness parameter z increases. In each setting, one of the relations may contain more tuples with high rewards than the other. Therefore, using OSL for one of the scan operators may sometimes become more efficient than using CL. Nonetheless, it is not clear which scan operators should use a learning strategy to deliver the most efficient results before running the query. It is an interesting question and requires more analysis to figure out what characteristics should be present for making this decision intuitively. In Figure 4 of Appendix B we presented another comparison among all the approaches using another M-N equi-join query Q_{11}.

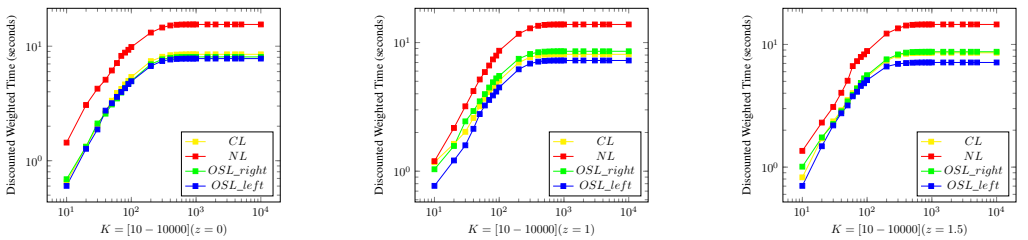

Figure 3: Discounted weighted average times for CL, OSL_left, OSL_right, and NL over Q_{10}

Figure 3 reports the results of CL, OSL (OSL_left and OSL_right), and NL over non-equijoin query created from Q_{10}, using log-log graphs. We observe almost a similar trend to the ones of equi-join queries where learning-based methods substantially outperform NL. SMS is not present here as it does not support non-equijoin queries. Figure 5 of Appendix B reports the results over non-equijoin query created from $Q15$.

References

Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

Donald A. Berry, Robert W. Chen, Alan Zame, David C. Heath, and Larry A. Shepp. 1997. Bandit problems with infinitely many arms. *The Annals of Statistics* 25, 5 (1997), 2103 – 2116. `https://doi.org/10.1214/aos/1069362389`

Michael J. Carey and Donald Kossmann. 1997. On Saying "Enough Already!" in SQL. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997*, Joan Peckham (Ed.). ACM Press, Tucson, Arizona, USA, 219–230. `https://doi.org/10.1145/253260.253302`

Michael J. Carey and Donald Kossmann. 1998. Reducing the Braking Distance of an SQL Query Engine. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, New York City, New York, USA, 158–169.

Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, University of Vienna, Austria, 3–14.

Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: an automatic query steering framework for interactive data exploration. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 517–528. `https://doi.org/10.1145/2588555.2610523`

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education, New Jersey, USA.

Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). ACM Press, Philadelphia, Pennsylvania, USA, 287–298. `https://doi.org/10.1145/304182.304208`

Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997*, Joan Peckham (Ed.). ACM Press, Tucson, Arizona, USA, 171–182. `https://doi.org/10.1145/253260.253291`

Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. 2015. Overview of Data Exploration Techniques. In *SIGMOD*.

Chris Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. 2005. A Disk-Based Join With Probabilistic Guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, Baltimore, Maryland, USA, 563–574. `https://doi.org/10.1145/1066157.1066222`

Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, San Francisco, CA, USA, 615–629. `https://doi.org/10.1145/2882903.2915235`

Appendix A Single scan learning algorithm

Algorithm 1: OSL algorithm to generate k tuples for $R \bowtie S$

```
rewardTable ← ∅
while Size(result) < k do
    // start a super round
    while Size(rewardTable) ≤ M do
        // Read sequentially the next tuple in R
        r ← R-Scan.nextTuple()
        // Estimate reward of r using N-Failure
        failure ← 0
        success ← 0
        // Read tuples sequentially from S
        s ← S-Scan.nextTuple()
        while failure ≤ N do
            if r ⋈ s ≠ ∅ then
                success ++
                Append(result, r ⋈ s)
            else
                failure ++
            if Size(result) ≥ k then
                return
            s ← S-Scan.nextTuple()
        // Add information of r to reward table
        rewardTable ← rewardTable ∪(r.address, success)
    // Find most rewarding tuple
    maxTuple ← ArgMax(rewardTable)
    rewardTable ← rewardTable \ maxTuple
    // Exploit the most rewarding tuple
    r ← R-Scan.getTuple(maxTuple.address)
    Append(result, t ⋈ S)
```

Algorithm 1 describes the general structure of the Single Scan Learning algorithm *Online Sequential Learning (OSL)* for R-scan. To simplify the exposition, we show operations of both R-scan and S-scan. OSL constitutes a series of *super-rounds*. In each super-round, our method first *explores and learns* the most rewarding tuple amongst all (remaining) tuples in R. It produces join results while learning. It then *exploits* the information learned in the super-round and performs a full join of the selected tuple with S. Our method continues to the next super-rounds until it generates sufficiently many results based on the user's preference.

Appendix B More empirical results

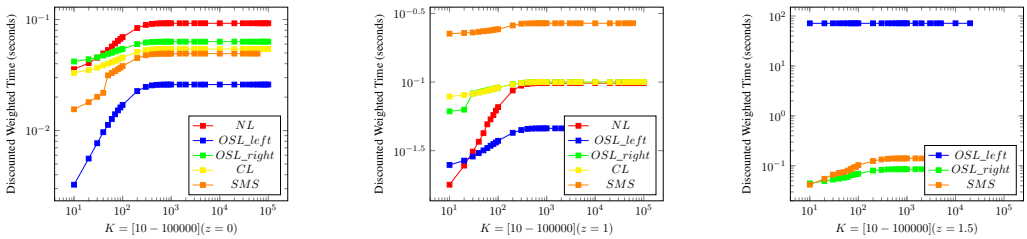

Figure 4: Discounted weighted average times for CL, OSL_right, OSL_left, SMS, and NL over Q_{11}.

Figure 4 compare the discounted weighted average times of the collaborative learning (CL), single scan learning (OSL_left and OSL_right), NL, and SMS for different output sample sizes (K) using the M-N equi-join query Q_{11} using log-log graph. Here NL again does not produce any results over the dataset with $z = 1.5$. The reported results for SMS use significantly larger memory (4GB) than other methods (0.5GB). The differences between the learning algorithms with NL and SMS become generally more significant as the skewness parameter z increases.

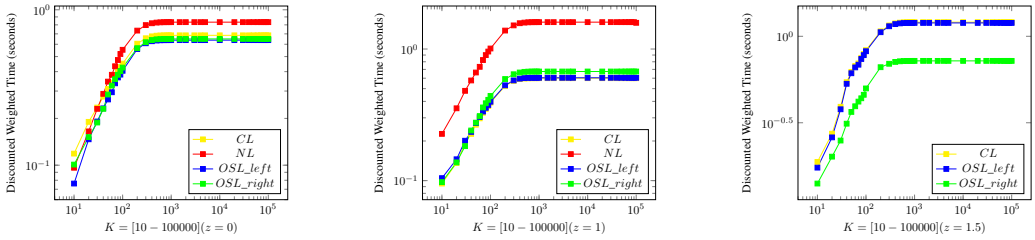

Figure 5: Discounted weighted average times for CL, OSL_right, OSL_left and NL over Q_{15}.

Figure 5 reports the results of CL, OSL (OSL_left and OSL_right), and NL over non-equijoin query created from $Q15$ using log-log graphs. We observe almost a similar trend to the ones of equi-join queries where learning-based methods substantially outperform NL. This difference is more significant for values of $z > 0$ in the results of Q_{15} as due to the skewness in the data learning-based methods may find more highly rewarding partitions. NL has not produced any answers for Q_{15} and $z = 1.5$ after about 15 minutes.