Dual-Agent Progressive Join

Vahid Ghadakchi
School of EECS
Oregon State University
ghadakcv@oregonstate.edu

Mian Xie
School of EECS
Oregon State University
xiemia@oregonstate.edu

Arash Termehchy
School of EECS
Oregon State University
termehca@oregonstate.edu

Rashmi Jadhav
School of EECS
Oregon State University
jadhavr@oregonstate.edu

Michael Burton
School of EECS
Oregon State University
burtomic@oregonstate.edu

Bakhtiyar Doskenov
School of EECS
Oregon State University
doskenob@oregonstate.edu

ABSTRACT

It is crucial to provide real-time performance in many query workloads, such as interactive and exploratory data analysis. In these settings, users need to view a subset of query results or a progressive presentation of the entire results quickly. Nevertheless, it is challenging to deliver such results over large dataset for common relational binary operators, such as join. Join algorithms usually spend a long time on scanning and attempting to join parts of relations that may not generate any result. Current solutions to this problem usually require lengthy and repeating preprocessings, which are costly in general settings and may not be possible to do in some cases, such as interactive workloads or evolving datasets. Also, they may support restricted types of joins. In this paper, we outline a novel approach for achieving efficient progressive join processing in which the scan operator of the join learns online and during query execution the portions of its underlying relation that might satisfy the join condition and use them in the join. We further improve this method by an algorithm in which both scan operators collaboratively learn an efficient join execution strategy. We also show that this approach generalizes traditional and non-learning methods of join. Our empirical studies using standard benchmarks indicate that this approach outperforms similar methods considerably.

PVLDB Reference Format:
Vahid Ghadakchi, Mian Xie, Arash Termehchy, Rashmi Jadhav, Michael Burton, and Bakhtiyar Doskenov. Dual-Agent Progressive Join. PVLDB, 14(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

It is crucial to provide real-time performance for queries over large data in many settings, such as interactive and exploratory data analysis or online query processing [4, 5, 13, 14, 17]. The recent pandemic showed an urgent need for analyzing the ginormous data quickly to model and forecast public health concerns in real-time [10, 20]. For example, it is vital that epidemiologists interactively test various queries about how a novel virus spreads over numerous and continually evolving case reports efficiently to recommend effective public policies as fast as possible. In these settings, it is desirable to provide subsets of query results quickly to reduce the time that users spend on data exploration and interaction. For instance, in an interactive environment, an epidemiologist may want to view subsets of answers to her query fast to design her next query according to these answers quickly. In these settings, users might also want to view results of their queries progressively without any long (initial) delays [13, 15, 17, 25]. This enables users to view and investigate the results without long waiting periods and as the results are being produced. This capability reduces users' waiting time and speeds up their data analysis tasks. For example, it is important to present the query results without long delays to the epidemiologist, so she can investigate them quickly as they become available. Using this method, users can stop the query execution quickly and as soon as they have enough information.

It is, however, challenging to find quickly subsets or the entire results of many binary relational operators, such as *join*, over relations with many tuples. To execute these operators, the relational data system has to check numerous if not all pairs of tuples from both input relations to find the ones that satisfy the operator's predicate. In many join queries, a substantial majority of these pairs do not satisfy the join predicate. As the system often does not know the pairs of tuples that satisfy the predicate, it may spend a long time to check numerous pairs without returning any or very few results. Also, the information of input relations often stored on secondary storage and accessing them takes lengthy I/O accesses. Hence, returning subsets or the entire join results might be very time-consuming and involve long delays. Thus, users may have to wait for a long time to view query results. This has become more challenging due to the rapid growth and frequent evolution of available datasets.

To address this problem, researchers have proposed algorithms that progressively read tuples from base relations and maintain them in main memory to improve I/O efficiency of the query execution, e.g., *Ripple Join* [13]. These methods, however, quickly run out of main memory over large relations before generating sufficiently many results [15]. Some methods build auxiliary data structures, e.g., indexes, over both relations to locate tuples that generate results quickly [17, 22]. Most users, however, cannot afford to wait

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

for the time-consuming preprocessing steps of building these data structures, which are often repeated whenever the data evolves. The indexes needed for a query workload are usually determined by experts or pre-trained models [1, 6]. Most normal users, such as epidemiologists, do not have the expertise to guide index building effectively. There might not also be sufficient training data to learn an accurate model of the query workload to guide these preprocessing steps. Moreover, current guiding models usually assume that the distribution of query workload is fixed over time [1, 6]. Users, however, increasingly produce non-stationary workloads particularly in interactive settings [18]. Also, indexes often take substantial storage and update overheads. Users might not have enough resources to build and maintain indexes for many queries.

Sort- and hash-based join algorithms produce the entire results of a join query efficiently but they need to reorganize the input relations in time-consuming steps before generating any result [11]. Researchers have also extended these algorithms for progressive query computation [15, 25]. There methods, however, support only a limited types of queries, e.g., joins with equality conditions [11]. Due to the popularity of statistical learning and inference over large data, join predicates increasingly contain complex conditions, such as user-defined functions or inequality conditions. They might also need a significant amount of main memory to generate results efficiently, which might not be available, e.g., progressive sort-based keeps considerable portions of both relations in main memory [15].

Toward realizing the vision of delivering real-time insight over large data, we propose a novel approach to binary join processing in which scan operators of the join collectively *learn online and during query execution* to return results quickly. The query plan for joining two relations is typically modeled as a tree with two scan operators as leaves and a join operator as its root. During query execution, each scan reads and sends tuples to the join operator and is informed of the success of its sent data in producing results by the join operator. Every scan operator has a strategy of reading tuples and sending them to the join operator in the query. Based on this feedback, the scan operators quickly learn the portions of data that are most likely to generate joint tuples and improves the efficiency of its strategy. Hence, they read and send portions of the data that generate more results earlier in query execution and avoid or postpone sending fragments of the data that might not contribute toward producing joint tuples. This method produces subsets of join results quickly. Since the scan operators learn the most promising portions of relations online and by performing the join, our method does not involve long (initial) delays to produce results. This approach does not require any upfront data preparation, offline training data, or precomputed data structures and their corresponding delays, overheads, and tuning. It can be used for both stationary and non-stationary query workloads. Each scan treats its join operator and its condition as a black box, therefore, it can learn efficient execution strategies joins other than the ones with equality conditions. This method is naturally implemented within the pipelining framework of query processing in current relational data systems.

Nevertheless, it is challenging to accurately learn efficient query processing strategies over large data and produce query results quickly at the same time. Each scan operator should establish a trade-off between producing fresh results based on its current information and strategy, i.e., *exploitation*, and searching for more efficient strategies, i.e., *exploration*. Moreover, as each relation may contain numerous tuples, each scan operator may have to explore many possible strategies to find the efficient ones. This becomes more challenging by the usual restrictions on accessing data on the secondary storage. For example, in the absence of any index, a scan operator has to examine tuples (blocks) on disk sequentially. Current online learning algorithms, however, often assume that each learner has a random access to all its alternatives. In this paper, we leverage tools from the area of online learning and devise algorithms that address the aforementioned challenges efficiently with theoretical guarantees. Our contributions are as follows.

- We provide a novel framework for processing a join query as collaborative online learning of its scan operators. We show that this approach generalizes the current methods of join processing. Therefore, a system might use a combination of traditional algorithms and learning operators to execute queries (Section 2).
- We show that due to the enormous search space, limited types of access to data, i.e., sequential scan, and limited sparse reward signals, scan operators cannot use popular online learning methods to learn efficient strategies quickly (Section 3). We propose a novel algorithm to learn efficient strategies in this setting and show that it performs significantly fewer I/O accesses than the comparable baselines statistically speaking. We propose a method for join processing in which one scan operator learns its strategy and the other one uses the traditional method of sequential scan (Section 4).
- We show that it will improve the time of producing subsets of join results, if both scan operators of join learn and adapt their strategies online. The resulting join algorithms, however, can perform some redundant I/O accesses. We propose techniques so that the scan operators can coordinate their learning and reduce these redundant accesses. Hence, they collaboratively learn efficient query processing strategies quickly (Section 5).
- We report the results of our extensive empirical study using a standard benchmark. Our empirical studies indicate that the proposed methods significantly outperform comparable current techniques for delivering both subsets and progressive delivery of complete query results in most cases (Section 6). We also show that in some cases learning is not by definition helpful, e.g., where all tuples in both relation generate the same number of results. We show that in these cases, our method is almost as efficient as comparable techniques.

One might extend our framework and methods to some other types of binary relational operators, such as intersection. However, due to popularity of joins, in this paper, we focus on and report empirical studies for join queries. The source code and other information about the proposed algorithms and reported empirical studies can be found at https//github.com/OSU-IDEA-Lab/Join-Game/tree/co-learn-updated.

2 FRAMEWORK

We present the framework that models processing of a join query method as online collaborative learning of its operators. Our framework extends to other binary operators, such as intersection. Nonetheless, we restrict our attention in this paper to binary joins and leave detailed explanation and instantiating our framework for other types of queries as future work.

2.1 Agents, Actions, & Reward

2.1.1 Agents and Actions. Relational data Systems usually model a query as a tree of logical operators called *logical query plan* [11]. In a join query, the leaf nodes are scan operators that read information form the secondary storage in blocks (pages). The tuples are pipelined from scans to the root operator, i.e., join, as the join operator calls its children's API to get fresh tuples. 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. As motivated in the preceding section, in this paper, we assume that both relations to be joined are large and none of them are small enough to fit into available main memory.

2.1.2 Reward. The *reward* of an action in each round of the query execution is the total number of joint tuples produced during that round. The join operator shares this reward with its child scan operators immediately after attempting to join their recently sent blocks. As we assume that the relations are stored on the secondary storage, each agent has to perform some I/O access(es) to read block(s) from the secondary storage. Since the dominating cost of performing joins is the time to perform I/O access, ideally, each scan operator may receive some reward in each round so that its effort of reading a block from disk produces some answers and does not go to waste. This increases the I/O efficiency of query execution. The *history* of a scan operator O at round t denoted as $H^O(t)$, is the sequence of pairs (a_i, r_i), $0 \leq i \leq t-1$, where a_i and r_i are the action and the reward of the operator O at round i of the join.

2.2 Strategy & Learning

2.2.1 Strategies. The *strategy* or policy of an operator O at round t is a mapping from $H^O(t)$ to the set of its available actions. A strategy is essentially the execution algorithm of its (logical) operator. An operator might follow a *fixed strategy* that do not change in the course of query execution. Current query operators often follow fixed strategies. For example, the scan operator for the outer relation in the (block-based) nested loop join plan follows a fixed strategy of sending the next block of the relation stored on the disk whenever requested by its parent join operator. The scan operator for the inner relation follows a similar fixed strategy.

An operator may use an *adaptive strategy* and choose actions in each round based on the actions' rewards and performance in their previous rounds using the history up to the current round. In particular, if the underlying relations contain sufficiently many blocks, i.e., a join has sufficiently many rounds, a scan operator may achieve a higher long-term reward by modifying its strategy during the query processing. For example, scan operators for the outer and inner relations in the nested loop join may use their history to estimate and send the blocks that are more likely to produce new joint tuples instead of the ones that may not lead to any results. Using the history of the join, a scan operator may learn that block b_1 in its base relation joins with significantly more tuples of the other relation than block b_2, i.e., b_1 is more rewarding than b_2. Thus, if it reads and sends tuples from b_1 to the join operator more often than b_2, it may cause the query to produce more joint tuples by performing the same number of I/O accesses. As the query execution progresses, the operators may learn and improve their estimate and the effectiveness of their strategies using the performance of their actions in preceding rounds. Our framework generalizes current approaches to query processing as operators that follow fixed strategies. Hence, it supports using both traditional and adaptive strategies for query operators.

2.2.2 Learning Strategies. Since the rewards of actions are not known at the start of the query processing, an operator has to learn these rewards while executing the query. Thus, the operator may initially explore a subset of available actions to find the reasonably rewarding actions quickly. Subsequently, the operator may *exploit* this knowledge and use the most rewarding actions according to its investigations so far to increase the short-term overall reward or *explore* actions that have *not* been tried to find actions of higher rewards with the goal of improving the total reward in the long-run. For example, consider the join of relations R and S where the scan over S uses a fixed strategy of sending a randomly chosen block in each round. The scan over R modifies its strategy in each round based on the information in the preceding rounds to read and send blocks that generate most joint tuples to the join operator. The scan over R may initially send a few randomly chosen blocks to its parent join operator to both produce some joint tuples and estimate the average joint tuples generated from each block in the subset, i.e., its reward. In the subsequent rounds, the scan operator on R may send the blocks with highest rewards so far, i.e., exploit, or pick other blocks that have not been tried before with the hope of finding ones with higher estimated reward than current ones, i.e., explore. Arguably, the most important challenge in online learning is to find a balance between exploration and exploitation.

2.2.3 Overall Objective Function. A query processing algorithm should return the results of a query as fast as possible, therefore, each operator should maximize its total reward using fewest possible rounds. As the datasets are often very large, it is desirable to deliver the results progressively where users see and investigate earlier tuples quickly while the system executes the query and delivers the rest of the results [4, 13, 14]. The user may stop the query execution as soon as she receives a sufficient amount of information, e.g., sufficiently many tuples, or let it run until completion. This is particularly useful in interactive or exploratory workloads where users need to know at least an estimate of the results very fast. Moreover, many analyses over large data may be satisfied by a sufficiently large subset of query results [4, 13, 14]. To quantify the overall objective of query processing, One may select a metric, such as *discounted (weighted/geometric) average of delays*, that is biased toward faster generation of early results. It measures the users' waiting time for receiving both a sample of and the full query

result. More precisely, the discounted weighted average of delays is defined as $\sum_{i=1}^{l} \gamma^i t_i$ where $0 < \gamma < 1$, t_i is the time to generate the ith result, and l is set to the number of desired results. It reflects the benefits of progressive computation over large data more precisely than some other metrics, such as *completion time*. The faster a query operator learns an efficient strategy during query execution, the nigher the value of this objective function is.

2.3 Why Dual-Agent Framework?

One may implement the learning of the processing strategy for entire query in a single place, e.g., the root join operator, instead of both scans and join operators. However, each operator has access to a distinct subset of data, e.g., each scan operator reads a different relation, and consequently a separate subset of available actions. Thus, it is natural to place learning of an effective strategy to apply those actions in its operator. Learning an effective strategy becomes substantially more difficult as the number of possible action grows [23]. Our framework simplifies learning effective strategies by distributing the set of possible actions and placing relevant actions of each operator in a distinct learning agent. Moreover, in our dual-agent approach, each operator communicates with the other operator(s) by sending data and receiving rewards and treats their internal logic, e.g., join condition, and implementation as black boxes. Thus, as opposed to the centralized approach, the implementations of learning operators may be reused across different queries, e.g., a learning scan can be used for binary operators other than join, such as intersection. Additionally, it may not be possible to learn an effective strategy for some operators online. For example, some aggregation functions require to see all tuples of a relation before delivering a precise result, which may make it difficult to compute via online learning. By distributing learning across various operators, our framework enables the data system to identify the parts of the query for which (online) learning of an efficient execution strategy is possible. This approach may also simplify using multi-threading and parallelism in query processing as done in traditional relational query processing.

3 CHALLENGES OF SCALABLE DUAL-AGENT JOIN PROCESSING

3.1 Initial Design

To illustrate the promises and clarify the challenges of our approach, we present an initial algorithm to implement our framework.

3.1.1 Multi-armed Bandit Algorithms. As explained in Section 2, the online strategy learning algorithm of scan operators should strike a balance between the time spent on exploring and learning the rewards of blocks and exploiting this knowledge to generate join results during query execution. Multi-armed Bandits (MAB) is a classic online learning problem over a set of actions with unknown reward that formalizes the aforementioned trade-off between exploration and exploitation [23]. It has been extensively used to implement the ideas of online learning in various domains. Consider a set of actions, i.e., arms, with unknown (and different) expected rewards. Every time the action is performed, we observe a random sample of its reward. The objective in MAB problem is to find the optimal sequence of actions in order to maximize the total expected reward. There are MAB algorithms that achieve near-optimal total expected reward [23]. A popular example of such algorithms is the *UCB-1* algorithm. In a nutshell, these algorithms initially try every action once, observe its reward, and provide preliminary estimations of the expected reward of each action. In the subsequent rounds, they pick more often the actions that have *sufficiently large estimated expected rewards* or *have not been tried enough* and update their estimated rewards. After sufficiently many trials, they fully commit to the action with the largest estimated expected reward. This approach delivers considerable reward while searching for the most rewarding action and strikes a near-optimal balance between exploration and exploitation statistically.

3.1.2 Adaptation of Popular MAB Algorithms. One may use the aforementioned algorithms to learn strategies for scan operators in a join. We denote the scan over R and S in the join of R and S (with an arbitrary join condition) as R-scan and S-scan, respectively. To simplify the exposition, we first assume that S-scan has a fixed strategy of reading a randomly chosen block from S and sending it to the join operator, i.e., *random strategy*. We will present learning for both scan operators in Section 3.2.3. It is shown that in the absence of any order, e.g., sorted relation, sequential scan, i.e., heap-scan, simulate random sampling effectively [13]. Thus, we assume that S-scan implements the random strategy using the sequential scan over S. On the other hand, R-scan aims at learning the rewards of blocks in R accurately and quickly and joining blocks of R with S in a decreasing order of their rewards. This way, the join operator progressively produces results efficiently and significantly reduces the users' total waiting time to view query results as explained in Section 2. As R-scan does *not* know the reward of blocks of R prior to executing the join, its learning should take place (online) during join processing.

To leverage the algorithms explained in the preceding section, R-scan first sequentially scans the entire R once. For each block B_R read from R, we sequentially read one block from S and join these blocks to observe the reward of B_R. The algorithm keeps track of rewards of each block of R in a *reward table* that is maintained in main memory. Each entry in the reward table maintains the address of every block in R on disk and its current accumulated reward. After this step, R-scan randomly accesses the block deemed the most promising by UCB-1 and joins it with the next block of S and generates its results. This may require multiple sequential scans of S. To avoid generating duplicate results, the algorithm keeps track of the blocks of R and S that have been joined in a table in main memory where each of its entries stores the addresses of all block of S joined so far with a block of R. If the selected block of R and current block of S have been joined, the algorithm tries the next block of S.

MAB algorithms assume that whenever an action is tried, it delivers some stochastic reward, i.e., each action has infinite reward. However, the reward of each block in our setting is limited as after a block of R is joined with every block of S it will deliver zero reward. Using only the estimated most rewarding block of R, the join may not produce sufficiently many results. Thus, R-scan should detect an exhausted block and proceed to find the next most rewarding block. In our adaptation of UCB-1, after discovering the most rewarding block in R, R-scan performs a full join between that block and S by

doing a full sequential scan of S. It then removes that block from its reward table and repeats the steps of UCB-1 over the remaining blocks of R to learn and use the subsequent most rewarding blocks in R. The join terminates after generating sufficiently many or entire results.

3.2 Shortcomings of the Initial Design

3.2.1 Very Large Set of Actions. Each scan operator may face numerous possible actions at each step of query processing over large relations. It may have to find the most rewarding block from hundreds of thousands of blocks in its underlying relation. Popular MAB algorithms need to inspect every action several times before finding the most rewarding one. The scan algorithm described in Section 3.1.2 uses the reward table that contains the addresses of every block in its underlying relation to access its chosen block at each round. Given the considerable I/O overhead associated with accessing each block, it will take long time in learning the most rewarding block. Furthermore, the reward table for a large relation may not fit in the available main memory. There are variations of MAB algorithms that take into account the overheads of accessing actions [8]. They learn both the unknown rewards and overhead for each action. They avoid trying actions with significant overheads and find the one with the desired balance between its reward and overhead. Every block has almost a fixed access cost in our scan algorithm and thus these MAB algorithms cannot reduce the search space by eliminating more costly actions. Moreover, these algorithms are designed for a relatively small number of actions.

3.2.2 Limited Rewards & Redundant Learning. Due to limited reward of each block, each scan operator should address another trade-off in addition to the usual exploration and exploitation in MAB settings. MAB algorithms usually assume that each action produces a stochastic reward whenever tried, therefore, they allow for lengthy exploration, e.g., each action may be tried several times. However, if the reward of each block is limited, it may be worth stopping exploration altogether as soon as the algorithm finds a sufficiently rewarding block. It may then stick to that block and exhaust its reward before trying other blocks. There are some MAB algorithm designed for cases where the rewards are limited [8]. They generally encourage a greedy approach based on the value of observed reward to minimizes the time for exploration, e.g., if the value of the observed reward for an action is larger than a certain threshold, stick to and exhaust that action. However, these algorithms are *not* designed for a large set of actions and do not address the challenges mentioned in the preceding section. In the algorithm described in Section 3.1.2, the scan operator has to learn the subsequent most rewarding action afresh. This may significantly prolong learning and execution. It is exacerbated by the sheer number of actions for each scan. To address this problem, one may reuse the estimated rewards in the preceding rounds for actions that are still available to the algorithm. However, it is not clear how many more times these actions should be tried to learn an accurate estimate of their rewards.

3.2.3 Lack of Collaboration in Dual-Agent Learning. In the method proposed in the preceding section, only one scan learns the optimal order for joining its blocks. There may be several highly rewarding blocks in S learning of which enables the system to generate a significant fraction of results quickly. This may happen particularly when each tuple of R joins with many tuples in S and vice versa, i.e., m to n joins. S-scan may leverage a similar method of learning and exploiting its learned information as the one used by R-scan. However, MAB algorithms require random sample of stochastic reward of each action in each round to deliver accurate estimates [23]. Hence, R-scan should provide a randomly selected block of R in each round so that S-scan observes a random reward sample for the block of S it tries in that round. Because R-scan reads and sends more promising blocks more often according to its learning algorithm, it *cannot* provide random samples of R blocks for S-scan. It is not clear how to devise a learning algorithm for S-scan that effectively finds most rewarding blocks in the absence of random sample of rewards.

4 SINGLE SCAN LEARNING

In this section, we investigate learning for only one of the scan operator in the join of R and S, namely R-scan, and propose learning methods that address challenges discussed in Section 3. We assume that S-scan has a fixed random strategy as explained in Section 3.1.2.

Our algorithm constitutes of a series of *super-rounds*. In each super-round, the algorithm learns and selects the most rewarding block from all blocks in R that have not yet been selected and uses the selected block to generate join results. It also produces results while learning the most rewarding block. The algorithm continues to the next super-round until it generates a given number of tuples or the complete results based on users' input. We first explain and analyze the methods for learning each most rewarding block in the first super-round. Then, in Section 4.2, we show how to arrange these super-rounds to find and use rewarding blocks and generate sufficiently many results progressively.

4.1 Learning Most Rewarding Block

4.1.1 M-Run. Following the analysis in Section 3, as R and S may contain numerous tuples, R-scan should ideally learn accurate estimates of the most rewarding block of R by accessing a relatively small fraction of R and S. Moreover, we would like R-scan to learn accurate estimates without many costly random accesses. To address the aforementioned challenges, we use two insights. First, researchers have proposed learning algorithms, called *many-armed MAB algorithms*, that effectively estimate actions with relatively high reward over infinitely many actions by exploring a sufficiently large random subset of them [2, 3, 27]. We use this approach to quickly estimate the block with highest reward in R as it requires to access only a random subset of blocks in R. Second, as explained in Section 3.1.2, since we do not assume any specific order of tuples in R based on attributes involved in the join, e.g., sorted order, we may safely assume that the sequential scan of R delivers random samples of its blocks. Thus, one does not have to perform random accesses to pick a random sample of blocks. We also show in this and next sections that using this approach, R-scan may estimate the most rewarding block of R without performing any additional random access.

To estimate the most rewarding block in R, we use *M-Run* learning method [2]. We first introduce an exploration technique over

R called N-*Failure*. We define each *round* as the join of a block of R and a block S. Using N-Failure, initially, R-scan and S-scan (sequentially) read a new block form their relations and send these blocks to the join operator to join. In each subsequent round, S-scan reads a fresh block from S. R-scan, on the other hand, does not read any block and keeps sending its current block in main memory to the join operator if this block has produced at least one joint tuple during the last N rounds. Otherwise, it reads the next block of R using sequential scan and sends it to the join operator. The reward of each block of R is the number of joint tuples it produces during its N-Failure exploration. For each block of R with non-zero reward, R-scan maintains its address to reward mapping in a reward table stored in the main memory.

R-scan performs N-Failure for every block of R until it either scans M blocks of R or finds a block of R that produces at least one join tuple in each of last M rounds. At this point, R-scan and S-scan *fully exploit* the most rewarding block in R by performing a full join of this block and entire S. That is, R-scan picks the block with the highest reward from the reward table, reads it from disk using random access, and sends it to the join operator. S-scan resets its sequential scan form the beginning of S, reads S block by block and sends each block to the join operator. As the most rewarding block of R may have already been joined with a sequence of blocks in S during its N-Failure turn, R-scan keeps track of this range of block numbers for each block in the reward table. If the reward signal is sparse, e.g., the join is very selective, in some super-rounds, the rewards of all examined blocks may be zero. In this case, R-scan picks its last scanned block that is already in main memory, i.e., Mth block, for the full join. It stores the information of this block in reward table to track all blocks that have been fully exploited.

4.1.2 Performance Analysis. Since we would like the learning to scale for relation R with numerous blocks, it is reasonable to set M to a sufficiently small value otherwise the learning of the most rewarding block may take many rounds and significantly many I/O accesses. On the other hand, small values of M may not deliver sufficiently precise estimate of the desired tuple.

We theoretically analyze the algorithm in the preceding section to find optimal values of N and M and compare it to similar methods. We investigate the *failure proportion* of the proposed method, which is the fraction of rounds during which the join does not produce any result. More precisely, due to the stochastic nature of our algorithm, we analyze its expected failure proportion. To simplify our analysis, we assume that each block in R and S contains only a single tuple. We denote the probability of success, i.e., generating a result, for a block i in R as p_i. We assume that probabilities of success of different blocks are independent and drawn from a distribution F. We also assume that F *is a uniform distribution in the interval* $[a, b]$ $0 \leq a \leq b \leq 1$. The results generalize to arbitrary distribution in the range $[0, 1]$. Using other distributions introduces substantial complexity without creating significantly different insight. We refer the interested reader to [2]. In our empirical study in Section 6, we evaluate cases in which expected rewards of different blocks are drawn from distributions with various degrees of skewness.

The following proposition establishes a lower bound on the expected failure proportion in each super-round. We denote the total number of blocks of S as $|S|$.

Proposition 4.1. *The expected failure proportion in each super-round of the join has a lower bound of* $(1-b)+(b-a)\sqrt{\frac{2}{|S|}}$.

Proof. The maximum number of join operations in each super-round is $|S|$. Each success of a block R is equivalent to success of its joined block from S. Hence, using Theorem 7 in [2], we have the stated lower bound. □

Proposition 4.1 holds for every learning algorithm in R-scan as far as S-scan uses a random strategy. To get a clear understanding of the result of the Proposition 4.1, let $b = 1$ and $a = 0$. Proposition 4.1 indicates that a lower bound on the expected failure proportion of every learning method is $\sqrt{\frac{2}{|S|}}$. This lower bound comes from the inherent difficulty of learning the most rewarding block while processing the join and the restriction on the access method of R-scan to its actions, i.e., sequential scan.

Next, we ask whether the learning algorithm explained in the preceding section will achieve an expected failure proportion close to the aforementioned lower bound. Interestingly, the following proposition shows that it is enough to use a random sample of R with a modest size to learn a block that achieves an expected failure proportion close to the lower bound of Proposition 4.1.

Proposition 4.2. *If R-scan uses M-run with $N = 1$ and $M = \sqrt{|S|(b-a)}$, the expected failure proportion of a super-round of the join is less than or equal to* $(1-b)+2\sqrt{\frac{(b-a)}{|S|}}$ *asymptotically.*

Proof. The proof directly follows from Theorem 8 in [2]. □

Again, to get a better understanding of the result of Proposition 4.2, let $b = 1$ and $a = 0$. In this case, the expected failure is at most $\frac{2}{\sqrt{|S|}}$, ignoring additive and multiplier constants, for $M = \sqrt{|S|}$. Consider a join algorithm that checks every block of S for each block of R to generate join results, such as block-based nested loop (BNL) [11] or Ripple Join [13]. They have the expected failure proportion of $1 - \frac{a+b}{2}$, e.g., 0.5 for $b = 1$ and $a = 0$, in each super-round. For a sufficiently large n, our method has about $\frac{b-a}{2}$, e.g., 0.5 for $b = 1$ and $a = 0$, less expected failure proportion in each super-round than these methods. This is indeed interesting as it learns its strategy online by checking a rather small subset of large relations. This analysis also shows that the relative efficiency of this algorithm grows with the difference between b and a. For instance, if $b = a$, as all blocks produce the same number of results, learning will not be useful no matter what method is used. We show in Section 6 that in cases the different blocks in R have almost the same reward, our approach produces subsets of or the entire join results progressively as efficiently as comparable methods.

If the number of blocks in R and S are comparable, this value for M indicates that R has to read a relatively small portion of R to find the most rewarding block in each super-round. Since binary join is symmetric, we can assign operator R to the larger relation of the two to reduce the time it spends on learning. In this case, however, the failure proportion may get relatively high. Thus, it may be more reasonable to pick R to be the smaller relation to achieve more precise estimates. If $|R| \leq \sqrt{|S|}$, then R-scan may use all blocks of R to find the most rewarding block. Nonetheless,

in this case, R contains significantly fewer blocks than S. Thus, considering all available actions to R-scan and S-scan, one finds the most rewarding block by scanning relatively small number of blocks in R and S. Another important factor on deciding the relation whose scan should follow a learning strategy is the cardinality of the join. If the join between R and S is 1 to m in which each tuple of R joins with multiple tuples of S, blocks of R may vary significantly more in terms of reward than the ones of S. Hence, it is more efficient for R-scan to use a learning strategy rather than S-scan.

Since the values of a and b are not known before performing the join, the results of Proposition 4.2 cannot be directly used to find the precise value of M. In our empirical study, we use values of $O(\sqrt{|S|})$ to reduce the range of possible values in training the hyper-parameter M. This analysis indicates that with a sufficiently small value for N one may achieve a low expected failure proportion in each round. In practice and for other distributions of rewards, one may choose slightly larger values than 1 for N not to miss promising blocks. We have trained N for a range of relatively small numbers, e.g., fewer than 15, in our empirical studies.

4.1.3 Granularity of Actions. In this paper, we assume that there is not enough main memory available to pick a larger unit of action than a block. Given enough available main memory, instead of using each block as an action, one may use a sufficiently large sequence of blocks, i.e., *partition*, as actions. This is particularly useful where the amount of reward from one block may not be sufficiently large to determine the most rewarding block(s). For example, in a relatively selective join, each block of R may produce a very small number of joint tuples. Due to the typical estimation errors during learning, an online learning algorithm may declare several blocks with different actual rewards to have the equal reward. Using larger partitions, the algorithm may also produce more results during learning and N-failure exploration. As our focus is on investigating the potential of using online learning in query processing, especially when the resources are limited, we use a block as our unit of action. We assume that each block fits at least a single tuple [11].

4.1.4 Other Learning Approaches. Yizao Wang et al. have proposed MAB algorithms for many-armed bandit with rewards between 0 and 1 that first samples M actions and then uses an extension of UCB-1 to find the most rewarding action among them [27]. To be used in our setting, their algorithm requires R-scan to keep the order of $\sqrt{|S|}$ blocks in the main memory, which may not be possible due to limited available memory. Thomas Bonald et al. have proposed another algorithm for many-armed bandit problem that achieves the lower bound of $\sqrt{2|S|}$ [3]. Although this algorithm provides a lower asymptotic expected failure proportion for a single super-round, it has a couple of issues to be used in our setting. In this method, R-scan may access different sets of blocks in each super-round. As we explained in Section 3.2.2, this makes it difficult to reuse learning in previous super-rounds and prolongs learning and query execution. We will show in Section 4.2 that M-run enables super-rounds to share estimates so each new super-round explores only a new block.

Another approach is to start exploitation after finishing exploration. R-scan first sequentially reads every block of R. For each read block B_R of R, the join operator may ask for sufficiently many blocks from S-scan to estimate the reward of B_R accurately. S-scan may re-scan and send the same set of blocks in S to the join operator for each block of R. That is, it may sequentially scan the first l blocks of S in each round of the algorithm. After fully scanning R, we have a reliable estimate of the reward of every block of R in the reward table, i.e., end of *exploration phase*. At this point, the algorithm may join blocks of R in their decreasing estimated reward with every block in S from staring from the one in position l using a block-based nested loop approach.

It is shown that this method requires many trials to deliver a reliable estimate of the reward of each block of R [23]. It needs to join each block of R in the exploration phase with $O(\sqrt[3]{|S|^2}\log|S|)$. Hence, most combinations of R and S blocks may have been tested during exploration and not many blocks in S left to exploit the estimates computed during exploration phase. Furthermore, the algorithm still needs to scan the entire R to estimate the expected reward of each block in R. The exploration phase in this approach may take a long time during which the method may produce joint tuples. It also has the overhead of maintaining a large reward table as the extension of MAB algorithms explained in Section 3.1.2.

4.2 Subsequent Super-Rounds

After each super-round, R-scan excludes the most rewarding block from its list of available actions. In each subsequent super-rounds, R-scan resumes its sequential scan from the last position that it was stopped. It reads and sends the next unread block of R to the join operator, explores its reward using N-failure technique, and adds its reward and address to the reward table in main-memory if its reward is non-zero. S-scan will continue its sequential scan of S in each step of this N-failure exploration. M-run accesses blocks sequentially and evaluates each accessed block reward using N-failure exploration technique only once. Because R-scan has the reward information of a new set of M blocks, after reading only one new block, it has enough information to estimate a new most rewarding block with the performance guarantees of Section 4.1.2 by selecting the block with most reward in the reward table. Thus, in each super-round after the first one, R-scan estimates the new most rewarding block by reading only one new block from R. This resolves the challenges explained in Section 3.2.2 for learning afresh to estimate the most rewarding blocks in each super-round. It then accesses the selected block using its address in the reward table as explained in Section 4.1.1.

If the goal is to efficiently produce a given number of joint tuples, the algorithm will stop as soon as the desired number of tuples are produced [4]. If the user would like to find as many joint tuples as possible in a certain number of rounds, i.e., time limit, the algorithm will stop as soon as it reaches the given number of rounds. Otherwise, it continues until all joint tuples are produced. Towards the end of the join processing, R-scan reaches a point where it has fewer than M available blocks. Since their rewards of the ones with non-zero reward are already computed, R-scan picks and sends them to join operator to be joined with the entire S in decreasing order of their reward. Blocks with zero estimated reward are not selected in any super-round. If there is a need for more output results, R-scan sequentially scan the R and sends them to the join operator one by one to be joined with the entire S. It skips the blocks in the

reward table as they have been already joined with S. S-scan keeps sequentially scanning S for each sent block of R.

If there is enough main memory available, operators may reduce the rounds of the join and subsequently the running time of the join by learning a list of rewarding tuples instead of only one in a super-round. In this method, operator R and S will follow the learning phase of k consecutive super-rounds without performing their subsequent join phases. At the end of this learning stage, operators will learn the k most rewarding blocks in R. Then, the operators perform a single join phase for all the learned k blocks by joining them with all blocks in S in one sequential scan of S. This methods reduce the number of scans of S almost by a factor of k.

5 COLLABORATIVE SCANS

As explained in Section 3.2.3, in m to n joins of R and S, some blocks of S may have substantial reward. Hence, it may reduce the response time of the join if R-scan and S-scan both use learning strategies. In this setting, each scan operator should both *learn the reward of its blocks* and *provide randomly sampled blocks* for the other scan so they both learn the rewards of their own blocks. This may double the number of I/O accesses or data processing effort by each scan during learning and slow down join processing significantly.

One approach to address this problem is to interleave and combine learning and sampling such that the set of explored blocks of one scan is also a random set of blocks from its base relation. That is, each scan may view each block as both an action of itself and also a sample from the environment for the other scan in the join. Each explored action in M-Run is in fact a random sample of available actions. Thus, if every scan operator uses M-Run learning method, in each exploration, it sends to the join operator both one action of its available actions, i.e., blocks, and a random sample of blocks of its underlying relation. Hence, each scan can both learn the reward of its own actions and at the same time enable the other scan to measure the rewards of its current actions by observing a random sample from its environment.

More precisely, in each super-round, one scan performs an N-Failure exploration and the other one performs a sequential scan. They will switch strategies for the subsequent super-round. Each scan that performs N-Failure collects rewards and other relevant information of its explored blocks as explained in Section 4.1.1. After reaching to its Mth explore block, each scan decides its most rewarding block and fully joins with with the other relation. In the subsequent super-rounds, the scan operators follow the algorithm described in Section 4.2. The join will stop after generating the desired number of tuples. The join operator detects and avoids reporting duplicate results using techniques described in Section 4.1.1.

Because each scan switches turns between M-Run and random strategies, it may sequentially read some blocks during its random strategy and may *not* resume its next N-Failure exploration immediately after the block that it explored in its last N-Failure exploration. Hence, it may skip doing N-Failure for some blocks. To ensure that each scan explores every block, it has to *go back* to the position after its last N-Failure. Nonetheless, the implementations of sequential scan in current relational data system implementations usually starts from the beginning of the relation. We have observed this in particular in PostgreSQL, which we have used to implement our algorithms. Thus, to resume the sequential scan from the last block for which the scan has done N-Failure, it has to restart the scan from the beginning and read and skip potentially many block, especially towards the middle and end of the algorithm. To save I/O access time, in our current implementation, our scans do not go back on disk and continue their sequential scan when they switch turns. Therefore, our current implementation of the case where both scan operators learn may *not* produce the full join results.

Instead of the aforementioned zig-zag between scans, one may use an algorithm with separate iterations. During the first one, R-scan uses a learning and S-scan uses a random strategy. After exploring and exploiting all blocks of R, in the second iteration S-scan follows a learning strategy and R-scan a random one. In this method, the second iteration is not useful as all the results have been already produced by the first one. One may follow such an approach during the zig-zag between scans, e.g., exploring sufficiently many blocks of R and then switching to S. This approach may wait a long time to discover and use rewarding blocks in S, particularly after exploring the first M blocks. Hence, it may not offer significant improvement over the method in which only one scan operator follows a learning strategy.

6 EXPERIMENT

6.1 Experimental Setting

6.1.1 Methods. We evaluate our proposed methods and the comparable methods explained in Section 1 that do not require lengthy preprocessing to create auxiliary data structure, e.g., index, and are not limited to certain types of joins, e.g., only equijoin. Thus, we do not compare our method to index-based ones, e.g., [17], and progressive versions of sort- or hash-based methods [15]. We compare our methods to *Ripple-Join* [13] and block-based nested loop join (BNL) [11] that satisfy the aforementioned properties. Ripple-Join quickly runs out of main memory in almost all settings before generating answers as explained in Section 1. Previous work has found similar results about Ripple-Join [15].

6.1.2 Workload. We use TPC-H benchmark, *www.tpc.org/tpc*, to generate the queries and databases for our experiments. TPC-H is a benchmark for decision support queries that usually contain many join operators and is widely used to evaluate join processing techniques. We use the $scale = 1$ to generate a TPC-H database of size 1 GB. We did not use larger values as it takes a very long time, e.g., more than a day, for the BNL to process some queries over larger versions of TPC-H database. Table 1 exhibits the details of the TPC-H relations used in our experiments.

Table 1: Information on TPC-H Relations(s=1)

Relation	Primary Key	Size (# Tuples)
customer	custkey	150,000
supplier	suppkey	10,000
part	partkey	200,000
partsupp	(partkey, suppkey)	800,000
orders	orderkey	1,500,000
lineitem	(linenumber, orderkey)	6,000,000

We have used all TPC-H queries with binary joins. To increase the number of queries in our study, we have also extracted binary joins from TPC-H queries with multiple joins. As explained in Section 5, collaborative learning of both scans (co-learning) may *not* be useful to execute 1-N joins in which each tuple of the right relation joins with at most one tuple of the left one. Thus, we will use this method only for M-N joins, i.e., each tuple of the right (left) relation may join with multiple tuples from the left (right) one. We have categorized our queries as 1-N and M-N joins. The original TPC-H query numbers and their corresponding (extracted) binary joins are as follows.

(1) 1-N joins
  (a) $Q_{14} : part \bowtie_{partkey} lineitem$
  (b) $Q_{12} : orders \bowtie_{orderkey} lineitem$
  (c) $Q_{10} : customer \bowtie_{custkey} orders$
  (d) $Q_{15} : supplier \bowtie_{suppkey} lineitem$
(2) M-N joins
  (a) $Q_{9} : partsupp \bowtie_{partkey} lineitem$
  (b) $Q_{11} : orders \bowtie_{o\_orderdate=l\_shipdate} lineitem$

We will refer to these binary joins by their query number in TPC-H workload for the rest of this section.

One of the parameters that impacts the join processing time is the distribution of the join attribute values; more specifically their skewness. It has been recognized that data skew is prevalent in databases and real world data distributions are often non-uniform [9, 21]. For instance, different customers usually place significantly different number of orders and different orders may contain various number of line items. We evaluate the query run-time over different versions of TPC-H database with different degrees of skewness in the join attributes. The Zipfian distribution (Zipf) has been used to evaluate the performance of query execution methods on skewed data distributions [7, 9, 21]. Zipf distribution has been frequently used in other domains, e.g. linguistics and Web search and mining, to model skewness in data [12, 28]. Our experiments use Zipf distributions with z values ranging between $[0, 0.5, 1, 1.5, 2]$ 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. Figure 1 exhibits the frequencies of the top 3 most occurring join attribute values ($orderkey$ in relation $lineitem$) with increasing skew parameter z. We have noticed that the data distribution $z = 0.5$ is nearly uniform. Moreover, the results of our experiments using $z = 0.5$ are very similar to the ones with $z = 0$. Hence, we do not report the results of experiments with $z = 0.5$ in the this section. When $z = 2$, the most frequent value occurs more than 50% of the total number of tuples in the relation(number of tuples in $lineitem = 6 * 10^6$). The proportion of the second and third frequent tuples decrease sharply.

We run each query 5 times and report the average time across these 5 times. As the order of tuples on disk may influence the results, we have randomly shuffled the order of blocks and tuples on disk for each relation before each run. The reported times include both learning and execution times.

6.1.3 Platform. We have implemented our proposed methods and BNL inside PostgreSQL 11.5 database management system. PostgreSQL supports and implements a tuple-based nested loop algorithm for join, which is significantly less efficient than BNL [11].

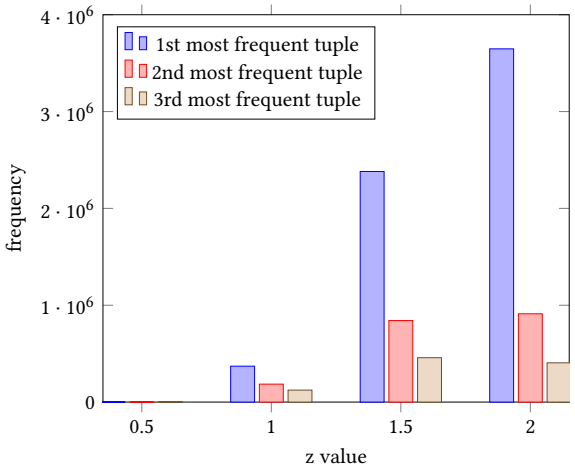


Figure 1: Frequencies of the top 3 most frequent values in a join attribute with Zipf distribution for different values of z

Hence, we do not use the tuple-based nested loop algorithm of PostgreSQL in our experiments. We have performed our empirical study on a Linux server with Intel(R) Xeon(R) 2.30GHz cores and 500GB of memory. The size of available cache and memory of the PostgreSQL server is set to the minimum possible value of $128KB$. This allows the database to only cache a few blocks of the relations and mitigates the impact of caching on the reported results.

6.1.4 Hyper-Parameters. Our proposed methods have two hyper-parameters that should be trained. The value of N in ***N*-failure** exploration determines the termination of exploring phase of current block. We have used query $Q_{12} : orders \bowtie_{orderkey} lineitem$ to train the value of N. Following the algorithm described in Section 4, we have evaluated a range of relatively small values for N, i.e., $N = [1, 10, 50, 100]$. We have chosen it to be 10. We have used 10-failure exploration strategy in all reported experiments.

Learning length specifies the number of blocks to be explored before entering into exploitation phase. Generally, greater learning length means that more blocks would be learned and it is more likely to find the promising block at the time of exploitation. However, longer learning also costs certain amount of time as well as I/O. There is a trade off between learning length and expected time to finish the query. We train the value of learning length using the same query as the one used to train N. The theoretical results in Section 4.1.2 provides us a reasonable estimate of the desired learning lengths, i.e., $O(\sqrt{|S|})$. After trying various learning lengths such as $\frac{1}{10} \times \sqrt{|S|}$, $\sqrt{|S|}$, $2 \times \sqrt{|S|}$, $4 \times \sqrt{|S|}$, $8 \times \sqrt{|S|}$, $20 \times \sqrt{|S|}$, $50 \times \sqrt{|S|}$, we pick $20 \times \sqrt{|S|}$ as learning length. If this learning length is increased, it wouldn't result in any obvious improvement of performance since it consumes more time doing I/O calls. Also, if we make it any smaller, the performance decreases sharply.

6.2 Experimental Results

We first report our results on using online learning for a single scan and both scans and compare them to BNL for generating different

samples of the results in Sections 6.2.1 and 6.2.2. Then, we will report our results on producing the complete join in Section 6.2.3.

6.2.1 Single Scan Learning. In single scan learning, only one out of the two participating relations learn to locate the more rewarding blocks as explained in Section 4. Considering a 1-N join query, it is reasonable to scan the relation whose every tuple may join with multiple tuples from the other relation, e.g., with the primary key as join attribute, to adopt a learning strategy. This is because each tuple in the other relation joins with at most one tuple and there may not be a considerable difference in rewards of different blocks of this relation. Thus, we report the results of the cases where only the scan over the *1* side of the 1-N join uses a learning strategy. To fairly compare our algorithm to BNL, we use the relation with learning scan as the outer relation in BNL. To simplify our description, we refer to this relation as outer relation in both BNL and our algorithm.

For an M-N join query, however, it is not clear which scans should use a learning strategy since the join attributes in both relations contain duplicate values. Thus, we report the result of learning scan over each relation for M-N joins using the algorithm proposed in Section 4 and report the average of the two cases. We also run BNL with each relation as outer in separate runs and report their average.

Figure 2 demonstrates the response time comparison of the performance of the single scan learning (denoted as *sl*) approach with BNL. This response time is averaged over all six queries in our chosen workload. We evaluate the queries over different output sample sizes, denoted as K. Since the queries Q_9, Q_{10}, Q_{11}, Q_{12}, Q_{14}, Q_{15} deal with relations of different sizes, their full joins generate different total number of tuples. Thus, we use a different maximum values for K per query based on their total output sizes. More precisely, we use K up to 1000 for Q_{12}, 5000 for Q_9, Q_{10}, Q_{14}, and 50000 for Q_{11} and Q_{15}, respectively.

Figure 2 shows three subgraphs that correspond to the 3 different ranges of K values, i.e., $[20-1000]$, $[20-5000]$, $[20-50000]$. The subgraphs with the ranges $K = [20-1000]$, $K = [20-5000]$, and $K = [20-50000]$ show the average results for queries Q_{12}, Q_{14}, Q_9, Q_{10}, Q_{14}, and Q_{11}, Q_{15}, respectively. Each graph compares the performance of both the algorithms as the z value increases from 0 to 2. As aforementioned, $z = 0$ and $z = 0.5$ have very similar distribution, therefore, we disregard plotting the curve for $z = 0.5$, leading to the plots containing $z = [0, 1, 1.5, 2]$. The solid curve represents single learning, whereas the dashed curve represents BNL. Each color is representative of a different z value for both algorithms.

When $z = 0$, the data distribution is uniform, i.e., no skew in the data. This means that all the blocks of outer relation may have nearly similar rewards, i.e., they generate similar number of tuples. Hence, learning may not offer a considerable advantage over BNL. Single scan learning, however, is able to generate sufficiently many tuples to compensate for the time it spends on exploration and learning. Thus, the performance of BNL and single scan learning looks very similar at $z = 0$. When $z \geq 1$, single scan learning outperforms BNL significantly as it is able to spot the most rewarding blocks during join processing. Also, the more skewed the data is, the more improvement single scan learning shows compared to BNL. We observe this trend in all three subgraphs except the experiment with $K = 5000$ and $z = 2$. For this exceptional setting, single scan learning does not perform as expected and even worse than BNL. This is mainly due to the performance of our method on the M-N join query Q_9. The join attribute $partkey$ is very skewed in both relations' $partkey$ attributes and at $z = 2$. Due to the significant skewness of the join attribute distribution on both relations, a substantial majority of the results are generated from very few blocks in the outer relation and most other blocks may not generate any results. Hence, our method may not be able to spot these few highly rewarding blocks relatively quickly and by scanning only a subset of blocks in the outer relation. Of course, for the same reason, it may also take a long time for BNL to generate results for this query.

Using discounted weighted average, the relative performance of single scan learning and BNL are similar to their reported performance according to response time. Hence, due to the lack of sufficient space, we do not report them. It is also more intuitive to observe the performance of methods and compare them by their response times.

6.2.2 Collaborative Scans. We compare the response times of the collaborative scan algorithm and BNL using the two M-N queries mentioned earlier, Q_9 and Q_{11}. Each reported response time of BNL is the average of the response times of two different sets of runs with each relation as the outer one. To compare the efficiency of the collaborative scan method with the one of single scan learning, we also report the response times of the single scan learning method over these queries. We report two runs of the single scan method for each query in each of which one of the scans uses the proposed learning strategy.

Figure 3 and Figure 4 show the response times of BNL, single scan learning runs (denoted as *sl_left* and *sl_right*), and collaborative scans (denoted as *cl*) over Q_9 and Q_{11}, respectively. The response times of the collaborative scans runs are generally between single scan learning on either relations. In each setting, one of the relations may contain the most rewarding blocks, therefore, one of the single scan learning runs delivers the most efficient results. Nonetheless, it is *not* clear which one of scan operators should use a learning strategy to deliver the most efficient results without actually running the query. The collaborative scans approach provides a middle-ground between the two possible configurations for single scan and is generally more efficient than the slowest configurations. This is because the collaborative scan approach has access to candidate blocks in both relations.

All these learning algorithms are at least as efficient as BNL where the data is uniform and more efficient than BNL in almost all other settings. The differences between the learning algorithms and BNL become generally more significant as the skewness parameter z increases. We again observe the same exception in Q_9 at $z = 2$ and as we discussed in the preceding section. It is very hard to learn rewarding blocks on this query as the rewarding blocks are very rare due to highly skewed join attributes on both relations.

All three learning algorithms are at least as efficient as BNL or outperform it over Q_{11}. The performances of all learning algorithms on Q_{11} are close and it is difficult to certainly state which one outperforms the rest. The join attributes in this query, i.e., $orderdate$ and $shipdate$, are highly selective. This makes it rather difficult

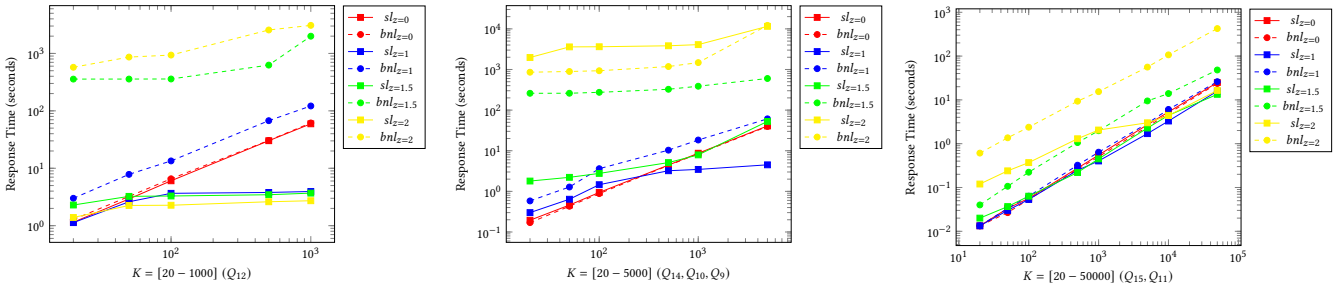


Figure 2: Response times of generating join samples using Single Scan Learning (sl) and BNL over different sets of comparable queries.

to spot the top rewarding blocks, Nonetheless, all three learning methods find sufficiently rewarding blocks in this setting.

Our empirical results using weighted discounted average provide a similar outcome for different methods over our workload. Thus, for the same reasons mentioned in Section 6.2.1, we do not report them in this section.

6.2.3 Full Join Evaluation. In this set of experiments, we aim at evaluating the waiting time needed to generate the complete join results using our proposed algorithms and BNL. It takes days to generate the entire results for many of our queries using BNL using the TPC-H data with $s = 1$. Hence, we have reduced the scale of the generated dataset by using $s = 0.1$ to shrink the size to $100MB$ and the number of tuples to $\frac{1}{10}$ of $s = 1$ which is showed in Table 1.

We compare the response times of BNL and single learning by averaging weighted time of all the queries. As explained in Section 5, the collaborative scans method is appropriate in the cases where users need a sample of the join results. Hence, we do not report the results of the collaborative scans algorithm in this section. We measure the performance of each method for every single query using discounted weighted time average as explained in Section 2.2.3 as it reflects and evaluates the nature of progressive query processing and measures users' waiting times. The discounted weighted average is defined as: $\sum_{i=1}^{l} \gamma^i t_i$ where $0 < \gamma < 1$ and l is set to the number of total tuples in the join results. We set the value of the discount factor γ to 0.99. We report the average of the aforementioned results for all queries in our workload.

Figure 5 shows the discounted weighted average of generating the complete results for BNL and single scan learning methods for all queries in our workload. Our proposed single scan learning method outperforms BNL significantly. The more skewed the data is, the larger improvement the learning approach offers. For uniform distributions $z \leq 0.5$, the weighted discounted average of both algorithms are quite close. When $z = 1$, we observe obvious difference and single learning starts to outperform BNL. When $z \geq 1.5$, the improvement increases much sharply. This is consistent with the results of our empirical studies for generating join samples in preceding sections.

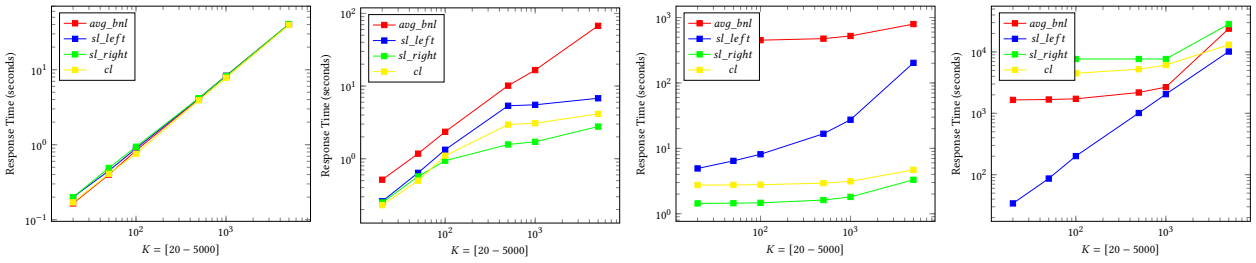
Lower values for γ in discounted weighted average put more emphasis on faster generation of early generated tuples than the larger values. They are biased in the favor of our method and show our method to outperform BNL substantially.

7 RELATED WORK

Adaptive Query Processing. Researchers have proposed adaptive query optimization and planning techniques to deal with the lack of information about the query workload [9, 16]. As opposed to our framework, this line of research is about finding most efficient algorithm(s) from a fixed set of traditional query execution algorithm, e.g., traditional join algorithms, to run operators in a query plan. For instance, researchers in [9] use a especial new operator in the query plan that chooses the best algorithm between using nested loop, sort-merge, or hash-join to execute its child join operator. These methods might also reorganize order of executing operators in the query plan. In our approach, however, each query operator aims at learning an efficient algorithm of accessing tuples of underlying relations and generating results. Also, Our methods deal with a significantly larger set of actions than the ones in adaptive query processing techniques.

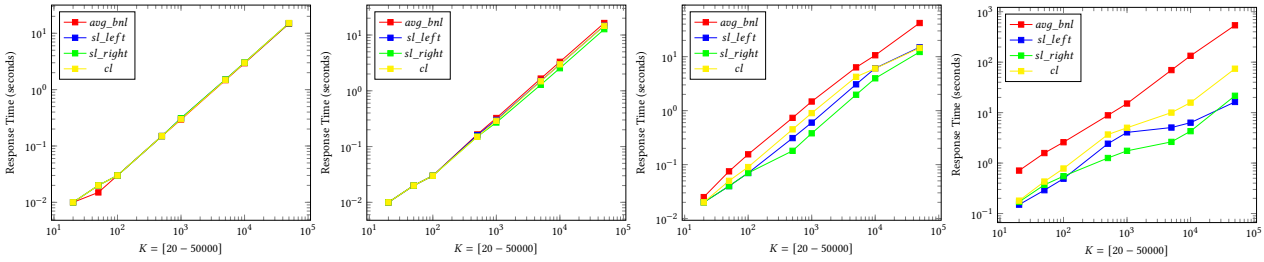
Offline Learning for Query Processing. The authors in [26] learn the distribution of the data to increase the sampling efficiency for queries with *count* aggregation function with restrictions that are evaluated quickly for each tuple, i.e., selection over a single table. This approach supports a restricted class of queries. For example, it does *not* handle joins as checking the join condition for a tuple in one relation may need a full scan of the other one, which requires a significant amount of time. It also heavily relies on learning an accurate predictive model for some attributes based on the content of others. Such dependencies may not be always present. It first learns a model for the query and then uses it to sample tuples. This separation of exploration and exploitation is shown to learn significantly less effective models in substantially longer time than the online methods that combine exploration and exploitation [23]. Our approach handles a significantly larger class of queries and uses collaborative online learning to find efficient strategies. Researchers have also leveraged a precomputed probabilistic graphical model over each relation to compute samples of their join [22]. It takes a time linear to the size of a relation to learn such model over the relation. This approach will face the challenges explained in Section 1 for methods that require precomputed data structures. Also, as opposed to these methods, our approach can compute the full join results progressively.

Reinforcement Learning for Operator Ordering & Query Optimization. Some systems learn efficient order of executing joins in a query during or over the course several execution of



(a) z = 0 (b) z = 1 (c) z = 1.5 (d) z = 2

Figure 3: Response times of generating join samples using Collaborative Scans (cl), Single Scan Learning (sl_left and sl_rigth), and BNL over Q_9.



(a) z = 0 (b) z = 1 (c) z = 1.5 (d) z = 2

Figure 4: Response times of generating join samples using Collaborative Scans (cl), Single Scan Learning (sl_right and sl_left), and BNL over Q_{11}.

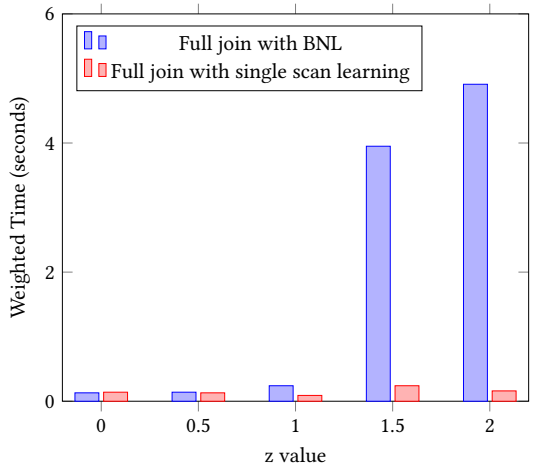


Figure 5: The discounted weighted averages for producing the full join results using single scan learning and BNL over all queries.

the query using reinforcement learning techniques [16, 19, 24]. As opposed to our approach, these system do not learn query processing algorithms. They also face significantly fewer actions than our methods. Their actions can be also accessed randomly.

8 CONCLUSION AND FUTURE WORK

Many users would like to investigate (subsets of) join results progressively over large and evolving data quickly. Current methods rely on preparing and maintaining auxiliary data structures or processing or reorganizing input relations. We proposed a novel online learning approach to return (subsets of) joins progressively and efficiently over large dataset without any preprocessing. In our approach, the scan operators in a binary join learn the portions of the tuples in each relation that produce the most join results during query execution quickly. Our empirical study indicates that our proposed methods outperform similar current methods in many settings significantly and deliver similar performance in the rest.

We believe that our approach introduces an exciting path in query processing by treating each query operator as a online learning agent and modeling query processing as multi-agent collaborative learning. Our proposed algorithm may be used to execute joins over base relations in a query plan with multiple joins. One may also use this method to perform joins in addition to the ones over base relations in a query with multiple joins. In this setting, each join operator will aim at optimizing its learning and efficiency of generating its own results. Nevertheless, to generate final query results quickly, each scan or join operator may define the reward of each blocks or tuples in its underlying relation based on its contribution to generating the results of the entire query. On the other hand, such a global reward signal may be too sparse, therefore, they may need to combine it with their local reward to learn promising tuples. We plan to devise methods that share global and local learning in queries that join more than two relations.

REFERENCES

[1] 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.

[2] 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.

[3] Thomas Bonald and Alexandre Proutière. 2013. Two-Target Algorithms for Infinite-Armed Bandits with Bernoulli Rewards. In *NeurIPS*. 2184–2192.

[4] Michael J. Carey and Donald Kossmann. 1997. On Saying "Enough Already!" in SQL. In *SIGMOD*, Joan Peckham (Ed.). 219–230.

[5] 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.

[6] 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.

[7] Stavros Christodoulakis. 1983. Estimating record selectivities. *Information Systems* 8, 2 (1983), 105–115. https://doi.org/10.1016/0306-4379(83)90035-2

[8] Wenkui Ding, Tao Qin, Xu-Dong Zhang, and Tie-Yan Liu. 2013. Multi-Armed Bandit with Budget Constraint and Variable Costs. In *AAAI*.

[9] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and Adaptive Online Joins. *Proc. VLDB Endow.* 7, 6 (2014), 441–452.

[10] Centers for Disease Control and Prevention. 2021. CDC Stands Up New Disease Forecasting Center. https://www.cdc.gov/media/releases/2021/p0818-disease-forecasting-center.html

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

[12] Sharad Goel, Andrei Broder, Evgeniy Gabrilovich, and Bo Pang. 2010. Anatomy of the Long Tail: Ordinary People with Extraordinary Tastes. In *WSDM*. 201–210.

[13] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). 287–298.

[14] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD*, Joan Peckham (Ed.). 171–182.

[15] Chris Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. 2005. A Disk-Based Join With Probabilistic Guarantees. In *SIGMOD*. 563–574.

[16] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *CoRR* abs/1802.09180 (2018). arXiv:1802.09180

[17] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*. 615–629.

[18] Ben McCamish, Vahid Ghadakchi, Arash Termehchy, Behrouz Touri, and Liang Huang. 2018. The Data Interaction Game. In *SIGMOD*. 83–98.

[19] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. *CoRR* abs/1803.08604 (2018). arXiv:1803.08604

[20] DJ Patil. 2020. 6 lessons learned to get ready for the next wave of COVID. https://medium.com/@dpatil/6-lessons-learned-to-get-ready-for-the-next-wave-of-covid-ee595766d4cb

[21] Silvio Salza and Mario Terranova. 1989. Evaluating the Size of Queries on Relational Databases with non Uniform Distribution and Stochastic Dependence. In *SIGMOD*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). 8–14.

[22] Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. 2021. *PGMJoins: Random Join Sampling with Graphical Models*. 1610–1622.

[23] Aleksandrs Slivkins. 2019. Introduction to Multi-Armed Bandits. *Found. Trends Mach. Learn.* 12, 1-2 (2019), 1–286.

[24] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. 1153–1170.

[25] Tolga Urhan and Michael J. Franklin. 2000. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE DATA ENGINEERING BULLETIN* 23 (2000), 2000.

[26] Brett Walenz, Stavros Sintos, Sudeepa Roy, and Jun Yang. 2019. Learning to Sample: Counting with Complex Queries. *Proc. VLDB Endow.* 13, 3 (2019), 390–402.

[27] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. 2008. Algorithms for Infinitely Many-Armed Bandits. In *NeurIPS*. 1729–1736.

[28] GK Zipf. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, Massachusetts, USA.