Bandit Join: Preliminary Results

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

ABSTRACT

Join is arguably the most costly and frequently used operation in relational query processing. Join algorithms usually spend the majority of their time on scanning and attempting to join the parts of the base relations that do not satisfy the join condition and do not generate any results. This causes slow response time, particularly, in interactive and exploratory environments where users would like real-time performance. In this paper, we outline our vision on using online learning and adaptation to execute joins efficiently. In our approach, scan operators that precede a join, learn which parts of the relations are more likely to join during the query execution and produce more results faster by doing fewer I/O accesses. Our empirical studies using standard benchmarks indicate that this approach outperforms similar methods considerably.

CCS CONCEPTS

• **Information systems → Database query processing**.

KEYWORDS

query processing, join algorithms, online learning

ACM Reference Format:
Vahid Ghadakchi, Mian Xie, and Arash Termehchy. 2020. Bandit Join: Preliminary Results. In *Third Workshop in Exploiting AI Techniques for Data Management (aiDM'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3401071.3401655

1 INTRODUCTION

It has been a long-standing challenge to efficiently join large relations. This difficulty is more prominent in interactive and exploratory environments, where the users expect real-time performance. The inherent difficulty of processing join queries is due to the need to inspect all information in the participating relations and find tuples that satisfy the join condition. Traditionally, database systems improve the efficiency of join by precomputing certain data structures, e.g., indexes, or sorting the relations, e.g., sort-merge join. These methods, however, are not applicable for many cases where: 1) The precomputed data structures are not available; 2) Preprocessing the data is costly. For instance, indexes may not be available over some join attributes in the database or it may take a long time to build one. Furthermore, if the index is large and stored on the disk, updating the index can be costly. Similarly, it may take a while or require too much main memory to sort large relations. These methods usually fall short of satisfying user's desired response time for large relations.

To overcome these challenges, researchers have proposed join algorithms that process subsets of input relations to provide the users with a sufficiently large subset of answers [4, 5, 8, 11, 13]. These approaches enable users to receive and inspect a subset of the answers in a short time or efficiently estimate downstream aggregation functions, which are useful in many applications, such as interactive data exploration and analysis. While the state-of-the-art approaches are successful in lowering the response time, they either require large amounts of memory [8], a preprocessed data structure or statistics of the underlying data [5, 13].

To address these limitations, we propose a novel approach for join computation. We use online learning to find the parts of the base relations that are most likely to produce new join results during the query execution. This way, the database system is able to generate a sample of the join results with fewer I/O accesses than the current approaches. As a concrete instantiation of our approach, we outline the high-level description of *bandit join* for binary joins. Bandit join learns the promising data blocks for one of the base relations during the join computation. It is challenging to learn an effective model online over large relations as they may contain numerous blocks. We show how to address this challenge by extending a multi-armed bandit algorithm [1]. Also, bandit join processes joins with a small memory footprint. Our empirical study using a standard benchmark indicates that bandit join outperforms similar methods considerably.

2 FRAMEWORK

A join operator is preceded by two scan operators over two relations. In each iteration of the join computation, each scan operator reads blocks from its corresponding relation on the secondary storage and sends them to the join operator. The join operator checks if the received tuples satisfy the join condition. Ideally, the scan operators should send the tuples that have a higher chance of joining to reduce the number of I/O accesses. In bandit join, a scan operator learns an effective strategy of pinpointing blocks that have higher chance of generating fresh joint tuples while processing the join. Next, we explain our framework.

Agents & Actions: Each scan operator is an *agent* in our learning framework. With some abuse of notation, we refer to the scan operators over relations R and S as agent/operator R and S in the join of relations R and S. At each *iteration* (or *round*) of processing the join, each scan operator reads one or more blocks from its base relation and sends them to the join operator. There is usually a limit on the number of blocks each agent can read and send to the join

aiDM'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8029-4/20/06…$15.00
https://doi.org/10.1145/3401071.3401655

operator from its relation due to the limited amount of available main memory to the join operator. To simplify our explanation, in this paper, we assume that at each round, each agent reads only one block. Thus, we define the set of *actions* available to each agent in each round as the set of blocks that it can read from its base relation. In the setting considered in this paper and in the absence of auxiliary data structures, such as indexes, an agent can access every block of its relation using sequential scanning. We note that more actions can be defined in other settings (e.g., if there are indexes on relation), which we leave for the future work. As we assume that the relations are stored on the secondary storage, each agent has to perform some I/O access(es) to read a block from the secondary storage. Since the dominating cost of performing joins is the time to perform I/O access, agents should ideally process the join and deliver the desired results using the smallest possible number of I/O accesses. We will discuss how this overhead may impact the strategies of each agent in the succeeding paragraph. Thus, one may view the join processing is as a collaborative effort, i.e., game, of two aforementioned agents with the goal of generating desired results using the least number of I/O accesses.

Reward & History: The reward of an action in each round of the join execution is the number of new joint tuples produced at that round. The join operator sends this reward to each agent, i.e., scan operator, at the end of each round. The goal of both agents is to maximize the *long-term reward* of the game. The exact formula for the long-term reward depends on the underlying application. We consider the reward up to round t to be the total number of distinct joint tuples produced up to that round. Other reward functions, such as discounted geometric sum, are possible. The *history* of the game at round t for agent with relation R, $H^R(t)$, is the sequence of pairs (a_i, r_i), $0 \leq i \leq t-1$, where a_i and r_i are the action performed by the agent R and its reward at round i of the game, respectively.

Join Strategies: Based on the performance of sent blocks in the previous rounds of the game, an agent may decide to send a certain block in the current round. More precisely, the *strategy* of agent R at round t is a mapping from $H^R(t)$ to the set of its available actions. Ideally, an agent should adopt a strategy that maximizes its reward, i.e., send a block that leads to the most number of fresh joint tuples.

Fixed Strategies & Adaptive Strategies: An agent may follow a fixed strategy to perform the join. For example, modeling the (block-based) nested loop join algorithm using our framework, the scan operator for the inner relation follows a fixed strategy of sending the next block (or tuple) in each round of the algorithm except for the round where it exhausts all the tuples in the relation. In this case, it sends the block at the beginning of its relation. Similarly, the scan operator for the outer relation sends a fixed block during a single scan of the inner relation and then moves to the next available block on its relation as soon as the scan of the inner relation is done. Nevertheless, if the underlying relations contain sufficiently many blocks, i.e., the join has sufficiently large number of rounds, an agent may achieve a higher long-term reward by adapting and modifying its strategy during the join. For instance, this agent can leverage its experience from the previous rounds of the join to modify its strategy and get a potentially greater reward in the next round(s). Using the history of the join, an agent may learn that block b_1 joins with significantly more tuples in the other relation than block b_2. Thus, if it sends tuples from b_1 to the join operator more often than b_2, it may generate k answers by reading fewer blocks of the other relation which would reduce the I/O cost of scanning the other relation's blocks.

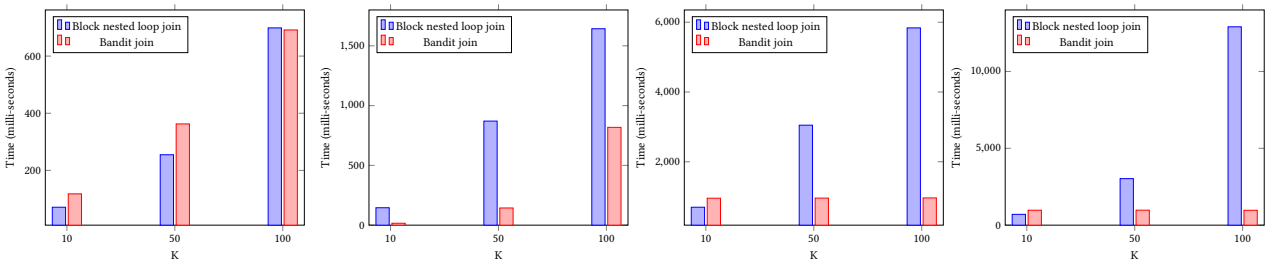
Exploration vs Exploitation: Since the success rate of tuples are *not* known at the start of the join, the agent has to learn them while performing the join. Such a learning method may first explore various actions or sequences of actions and then exploit this knowledge to choose promising actions in the later rounds of the join. The key element in this approach is to balance exploration and exploitation [1]. If an agent mostly explores possible sequence of actions, to gain more knowledge, it may end up scanning many blocks of the relation before generating any join results. On the other hand, if the agent mostly exploits the knowledge gained from the previous rounds of the join and performs a limited amount of exploration, it may not find the optimal strategy which in turn leads to a less efficient join.

3 LEARNING AND EXECUTING AN EFFECTIVE JOIN STRATEGY

Learning an effective join strategy has three challenges: 1) Prior to executing the join, the agent does not know which blocks or tuples would result in the highest rewards, therefore, learning should take place while processing the join. Clearly, the learning algorithm should find a reasonably accurate strategy within relatively small number of rounds. Otherwise, the learning may take as much time as join algorithms that examine every possible pair of blocks or tuples, e.g., nested loop join. 2) Users would often like to receive partial results in a short time. Since the learning phase may take a while, it may take a considerable amount of time for the operators to generate some results. Thus, the operators should combine learning a good strategy and producing output tuples in order to satisfy users' need. 3) Once all join results of an optimal block(s) or (group of) tuple(s) are generated, this block or tuple will deliver zero rewards. Thus, the agent should detect an exhausted optimal block, and proceed to find the next best block.

We propose bandit join, an online learning and joining algorithm that overcomes the mentioned challenges. Bandit join consists of *super-rounds*. At each super-round, agent R explores a limited number of blocks from the outer relation to learn the block with the maximum reward $r_{max} \in R$. Then, R reuses r_{max} until all possible results of r_{max} are generated and the inner relation S is exhausted. Next, R removes r_{max} from its set of available blocks. At this point, the algorithm has completed one *super-round*. The algorithm starts a new super-round to learn and use the next best block in R with the largest reward. This process continues until the required number of join results is generated. In our algorithm, agent S follows a fixed strategy of sequentially scanning blocks from its base relation S and sending them to the join operator. We plan to extend our algorithm to enable relation S to also adapt and find its best performing blocks.

In our framework, considering that the relations are not sorted by their join attribute, we assume reading the next block sequentially is equivalent to randomly sampling the joint attribute. This assumption has been used frequently in join computation with success, e.g., [8]. Therefore, one may also view the strategy of agent S as picking



(a) Response time of Q14 ($z = 0$) **(b) Response time of Q14 ($z = 1$)** **(c) Response time of Q12 ($z = 0$)** **(d) Response time of Q12 $z = 1$**

Figure 1: Response time of bandit join compared to block nested loop join for different values of k

random blocks from relation S. That is, the reward distribution for the blocks of R is fixed. Thus, agent R may use multi-armed bandit (MAB) algorithms to learn the best performing blocks online [3]. Nevertheless, MAB algorithm usually need to try each action multiple times in (semi-)random orders to find the most effective one. This will be very time-consuming in our setting as R may have numerous blocks to explore where accessing each of them require spending many I/O accesses without having any index. Thus, to find r_{max} at each super-round, bandit join models relation scanning as an infinitely many-armed bandit problem [1, 2, 17]. These problems assume the set of available arms/actions is too large to be fully explored. Thus, they aim at effectively estimating the most rewarding action(s) using a sufficiently small sample of the set of actions. Each action has a reward with an unknown probabilistic distribution that can be sampled during the exploration phase.

There are different algorithms to solve infinitely many-armed bandit problems. Bandit join uses *m-run* algorithm [1] to find the r_{max} at each super-round with $m = |S|$. Operator R first starts by sequentially scanning its relation and maintains a mapping from the scanned blocks' addresses to their total observed rewards in the main memory. As long as the current block $r \in R$ produces a join result, R keeps reusing this block. As soon as r fails to produce a result, R reads the next block in the subsequent round. If there is a block that has m consecutive successes, operator R stops its scan and declares that block as the estimated r_{max}. Otherwise, it stops scanning after reading m distinct blocks. In this case, R picks the block with maximum reward from the set of seen blocks whose positions and rewards are maintained in the memory. At this point, the *learning phase* of one super-round is finished and R reuses r_{max} to generate all of its joining tuples. For the next super-round, instead of re-running the strategies, the operators leverage the information gained from the previous super-round(s) to learn the next most rewarding block with a relatively small number of I/O accesses. A high level pseudo code of the bandit join is presented in Algorithm 1.

It is proved that the *m-run* algorithm achieves a success proportion to the asymptotically optimal one, by setting m equal to the square root of the total number of trials [1], which in our setting is the number of blocks in relation S. For R to deliver a reward close to asymptotically optimal one, the reward distribution of its underlying relation, i.e., R, should not be too skewed, e.g., one block of R joins with every tuple in S and the rest do *not* join with any [1]. Our empirical studies using standard benchmarks indicate that this method learns reasonably effective blocks even when the reward distributions of R is relatively skewed. For formal results on optimality and assumptions of the m-run algorithm, we refer the reader to [1].

Algorithm 1: Bandit_Join(R, S, join_condition, k)

```
reward ← create an empty dictionary
while Size(result) < k and HasNext(R) do
    while Size(reward) ≤ m do
        r ← NextBlock(R)
        if not HasNext(S) then
            Reset(S)
        s ← NextBlock(S)
        while Join(r, s) ≠ ∅ do
            Append(result, Join(r, s))
            if Size(result) ≥ k then
                return
            Increment(reward[r])
            s ← NextBlock(S)
    r ← ArgMax(reward)
    RemoveFromDictionary(reward, r)
    // join tuple r with the rest of relation S
    Append(result, Join(r, S))
```

4 EXPERIMENTS

We evaluate our method against block nested loop join (BNLJ) [7].

Dataset and Queries: We use TPC-H [1] to generate the queries and databases for our experiments. We experiment with 3 different database scales $s \in \{1, 2, 3\}$ with 1.5, 3, and 4.5 million tuples respectively. We use TPC-H queries, Q_{12} and Q_{14}. Note that, we only process and evaluate the join part of these queries to avoid the overhead introduced by the other operators.

Q_{12} : `SELECT * FROM order JOIN lineitem ON o_key = l_orderkey`

Q_{14}: `SELECT * FROM part JOIN lineitem ON p_key = l_partkey`

We run each query 10 times and report the average time to obtain k join results. The reported time includes learning and execution times.

Experiment Setup and Implementation: We implement bandit join inside PostgreSQL 11.5 database management system, 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 will allow the database to only cache a few blocks of the relations and will mitigate the impact of caching on the reported results. Multi-threading is turned off and its impact on bandit join will

[1] available at: http://www.tpc.org/tpch/

be examined in the follow-up works. We set the group size of the bandit join to 32 tuples. This number can be adjusted to match the exact number of the tuples in a block based on the size of each row in the relation.

4.1 Comparison with The Baseline

As our baseline, we implement BNLJ (improved version of PostgresSQL's nested loop). One of the parameters that impact the join processing time is the probability distribution of the join attribute values and more specifically the skew in the data [6]. We evaluate the query run-time over data-sets with different skews by assuming a Zipfian distribution with parameter z over the data [6]. Setting $z = 0$ will result in uniform distribution. As we increase the value of z, the distribution becomes more skewed. The skew in the value of join attributes will impact the join selectivity. If the skew is equal to zero (e.g., when we have a uniform distribution), the join selectivity of different tuples will be similar to each other and the range of join selectivities will be small. However, if the skew is high, the join selectivity of different tuples will have a high variance and a large range. Thus, the m-run algorithm identifies r_{max} more effectively.

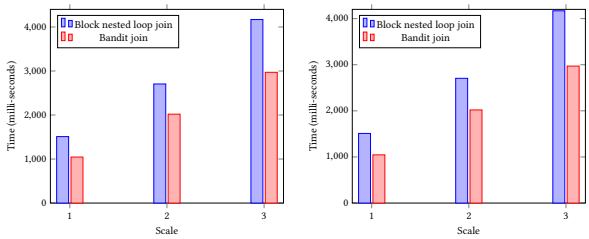
Figure 1a shows the running time of Q_{14} using BNLJ and bandit join over a dataset with zero skew (uniform distribution). Block nested loop outperforms bandit join for $k = 10$ and $k = 50$ because, with uniform distribution, it is difficult for bandit join to learn the best tuple group in a few rounds. However, as we increase k, bandit join has more time to learn and outperforms BNLJ. Figure 1b shows the results for datasets with slight skew ($z = 1$). This figure shows that for a slightly skewed data, the bandit join can learn an approximately good tuple group. But BNLJ is more likely to stick with a "useless" tuple group. Figures 1c and 1d show the same results for Q_{12}. These results conform with the previous ones except when $k = 50$ and $z = 0$, bandit join does a better job. One reason for this is that Q_{14} has a higher join selectivity than Q_{12} in this experiment. The average join selectivity of Q_{14} in this experiment is 30.06 which means that BNLJ can produce $k = 50$ in a few rounds without the need to learn the best tuple group.

4.2 Scalability

Next, we evaluate the impact of dataset size on the performance of bandit join and BNLJ with $k = 100$ and $z = 1$. Figure 2 shows the response time of bandit join and BNLJ on three datasets with different sizes. We see that as the dataset size grows, bandit join outperforms BNLJ with a larger margin. Note that Q_{12} is a primary key to primary key join. In this setting, m-run algorithm can not learn the optimal tuple/group. However, bandit join skips a tuple group at the first failure. Thus, if we read a tuple group that has a very low join selectivity and is very unlikely to produce a result, bandit join will skip that tuple group after one try but BNLJ will stick with it until it exhausts the inner relation.

5 RELATED WORK

There has been several approaches to utilize machine learning techniques in designing some components of database managements systems, such as query optimization [10, 14–16] and indexing [12]. The authors in [12] outline potential offline learning techniques to sort a relation. Our approach differs with these methods in viewing query operators as potentially adaptive agents that communicate



(a) Response time of Q_{14} **(b) Response time of Q_{12}**

Figure 2: Impact of dataset size on the response time.

with other operators/agents and collaboratively learn query evaluation strategies online. It also focuses on join evaluation. As it uses online learning during query execution, it produces results faster than the offline methods, which have to spend significant time to learn a model without using it.

6 ONGOING WORK

We believe bandit join introduces an exciting research path in query processing by treating each query operator as a learning agent and modeling query processing as a multi-agent collaboration, i.e., game. In this setting, agents interact with the common goal of achieving maximum efficiency in query processing. We plan to extend our work for the case where both scan operators learn and also for queries with more operators as well as more varieties of operators.

ACKNOWLEDGMENTS

We thank Raymond Liu for helpful discussions and simulations studies.

REFERENCES

[1] 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.*
[2] Thomas Bonald and Alexandre Proutière. 2013. Two-target Algorithms for Infinite-armed Bandits with Bernoulli Rewards. In *NIPS.*
[3] Sébastien Bubeck and Nicolò Cesa-Bianchi. 2012. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. In *Foundations and Trends® in Machine Learning* (foundations and trends® in machine learning ed.). 1–122.
[4] Michael J. Carey and Donald Kossmann. 1998. Reducing the Braking Distance of an SQL Query Engine. In *VLDB.*
[5] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On Random Sampling over Joins. In *SIGMOD.*
[6] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *VLDB.*
[7] Hector GarciaMolina, Jeff Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book.*
[8] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD.*
[9] Chris Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. 2005. A Disk-Based Join With Probabilistic Guarantees. In *International Conference on Management of Data*, Fatma Özcan (Ed.). 563–574.
[10] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv.*
[11] Albert Kim, Liqi Xu, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya Parameswaran. 2016. Optimally leveraging density and locality to support limit queries. *arXiv preprint arXiv:1611.04705* (2016).
[12] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *ICDM.*
[13] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *SIGMOD.*
[14] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *aiDM.*
[15] Jennifer et. al. Ortiz. 2019. Learning state representations for query optimization with deep reinforcement learning. *arXiv.*
[16] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: regret-bounded query evaluation via reinforcement learning. In *SIGMOD.*
[17] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. 2018. Algorithms for Infinitely Many-armed Bandits. In *NIPS.*