The Join Game

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

ABSTRACT

Join is one of the most frequently used and costly operations in the database management systems. In most join queries, the majority of the process time is spent on scanning and attempting to join the parts of the relations that do not satisfy the join condition and do not generate any join results. Nevertheless, modern join operators usually use a static approach in scanning the relations and do not adapt to different underlying data distributions. In this paper, we introduce *bandit Join*, a join algorithm that quickly learns the parts of the relations that are likely to produce join results and utilizes those parts to produce k join results efficiently. We show that in many cases, bandit Join can outperform the nested loop algorithm.

CCS CONCEPTS

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

KEYWORDS

query processing, join algorithms, bandit problems, reinforcement learning

ACM Reference format:
Vahid Ghadakchi, Mian Xie, Arash Termehchy, Rashmi Jadhav, and Michael Burton. 2020. The Join Game. In *Proceedings of aiDM '20: International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Portland, OR, June 19, 2020 (aiDM '18),* 5 pages.
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The join operator is one of the most important, costly and frequently used operators over relational databases. It has been, however, a long-standing challenge to efficiently join large relations. This challenge is more prominent in interactive systems, 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 [7]. These methods, however, are not applicable for many use cases where 1) the precomputed data structures are not available and 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. Similarly, it may take a while or require too much main memory to sort large relations. Moreover, these methods fall short of satisfying user's desired response time for large relations.

To overcome the aforementioned challenges, researchers have proposed join algorithms that process subsets of input relations to provide the users with a sufficiently large subset of answers. More specifically, as opposed to reducing the total time of the join, a group of join algorithms aim at quickly outputting an initial subset of the joint tuples and gradually completing them [9]. This approach enables users to receive and inspect a subset of the answers in a short time, which is useful in many applications, such as interactive data exploration and analysis. A notable example of this approach is *Limit* or *Stop* [4] operators in the database systems, which limit the output tuples of a join operation in a hope to reduce the query response time [1]. These operators take k as an input and generate k results in a fraction of time needed to process the whole query. The time required to generate k results is defined as the *response time* and the goal of these systems is to minimize the response time of the queries.

While the state-of-the-art approaches are successful in lowering the response time, they either require 1) large amounts of memory [8] or 2) a preprocessed data structure or statistics of the underlying data [5]. To overcome these challenges, we propose *bandit join*, a join operator that efficiently generates k join results by adapting to the underlying data distribution. We model the join processing as an online learning problem where the goal is to learn and use parts of the input relations that generate high number of join results which would lead to fewer I/O scans and a faster join algorithm. In this paper, we focus on binary equi-joins, i.e., joins over two relations with equality predicates and leave the other types of joins as an interesting future direction.

2 THE ONLINE LEARNING FRAMEWORK

The binary join operator is preceded by two scan operators over two relations. Each scan operator reads blocks of tuples from its

aiDM '18, June 19, 2020, Portland, OR

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

corresponding relation on the disk. After reading a block, the scan operator outputs tuples of this block which is consumed by the join operator as an input. The join operator collects tuples from two relations and checks to see if the received tuples satisfy the join condition. If so, their corresponding joint tuple will be generated as an output, otherwise the tuples will be discarded. Ideally, the scan operators should send tuples that have a higher chance of joining with each other which depends on the values of their join attributes. This way, the number of I/O and disk accesses, that are the dominant overhead of performing a join, will be minimized.

In Bandit Join, the scan operator learns an effective strategy in emitting tuples that have a higher chance of generating a joint tuple. In this section, we present the learning framework of the Bandit Join.

2.1 Agents, Actions, and Rewards

Agents: Each scan operator is an agent in our learning framework. At each *iteration* of processing the join, each scan operator reads a tuple from its base table and sends it to the join operator. We use the word *iteration* and *round* interchangeably to refer to a single iteration of the join. In our setting for a binary join of two relations, one of the scan agents is learning to emit tuples that produce a join with higher likelihood. The second scan operator emits random tuples and follows a random strategy. If a pair of the input tuples from scan operators satisfy the join predicate and generate a new joint tuple, the join operator outputs their corresponding joint tuple.

Consider join of relations R and S denoted as $R \bowtie S$. R is defined as the outer relation and S is defined as the inner relation. By the abuse of notation, unless otherwise noted, we refer to the scan operators over R and S simply as agent or operator R and S, respectively. In our framework, agent R is the learning agent and agent S acts randomly and sends random tuples.

Actions: Each scan operator may perform one of the following *actions* at each iteration of processing the join: 1) `next:` reads the next tuple of the input relation on the disk sequentially and sends it to the join operator. 2) `go:` reads the tuple of the input relation at a given address and sends it to the join operator. 3) `end:` sends an empty tuple to the join operator when the scan operator reaches the end of the relation. 4) `reuse:` informs the join operator to use the last sent tuple. Note that the latter two actions are introduced to simplify the exposition of our framework. More actions can be defined in different settings (e.x. if we had access to an index).

Reward: An action performed by the learning scan operator is successful if it leads to generating a new joint tuple. In this case, the join operator will send a positive reward of value 1 to the learning scan operator. Otherwise, the reward will be 0. The *reward* of an agent in round T of the learning process is $u_T = \frac{1}{T}\sum_{t=1}^{T} r_t$, where r_t is the reward of the action of the agent in round t. For a fixed T, the larger the reward is, the more joint tuples the agent generate.

2.2 Strategies and Adaptation

The *history* of an agent at round t is a sequence of pairs (a_i, r_i), $0 \leq i < t$ where a_i and r_i are the action and reward of the agent at round i of the join, respectively. The *strategy* of each agent at round t is a mapping from its history to an action at time t.

An agent may follow a fixed strategy to perform the join. For example, modeling the nested loop join algorithm using our framework, the scan operator for the inner relation follows a fixed strategy of performing a *next* action in each round of the algorithm except for the round where it exhausts all the tuples in the relation. In this case, it performs an *end* followed by a *go* to the beginning of the relation in the subsequent round. Similarly, the scan operator for the outer relation performs a *reuse* action in all rounds of the join excepts for the ones where the other agent performs *end* in which the outer agent performs *next*.

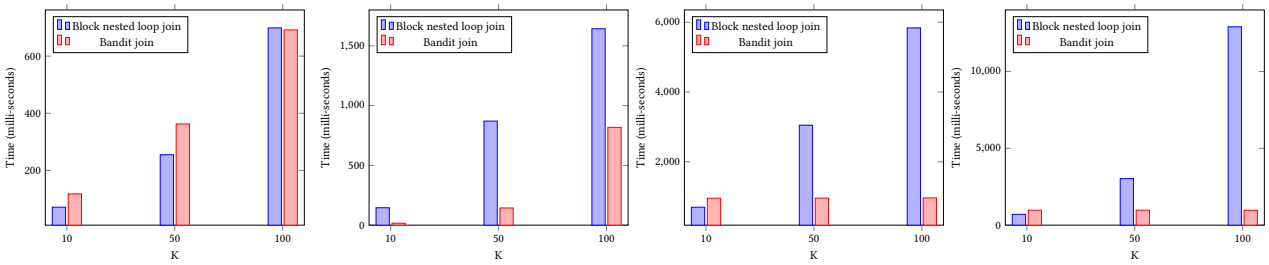
Nevertheless, if the underlying relations contain sufficiently many tuples, i.e., the join has sufficiently large number of rounds, an agent may achieve a higher reward in time T by adapting its strategy during the join. This agent can leverage its experience from the previous rounds of the join to modify its strategy and get a potentially greater reward for the next round(s). For example, using the history of the join, an agent may learn that tuple t_1 joins with significantly more tuples in the other relation than tuple t_2. Thus, if it sends t_1 to the join operator more often than t_2, it may generate answers faster than the case where t_1 is sent more frequently.

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. If an agent mostly explores possible sequence of actions, it may take a long time to generate k 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 THE OPTIMAL JOIN STRATEGY

Finding an optimal join strategy has three challenges. First, the agent does not know the optimal tuple with the greatest reward before processing the full join, therefore, it has to learn it while processing the join. Clearly, it should deliver a reasonably accurate estimate within relatively small number of rounds. Otherwise, this strategy may take as much time as the strategies that examine the join of every possible pairs of tuples, e.g., nested loop. Second, users would often like to receive some results fast, e.g., in interactive data exploration. 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. Third, as soon as all the joint tuples that can be produced using the optimal tuple are generated, this tuple will not deliver any more rewards. Thus, operators have to detect when the possible rewards generated from a tuple has been exhausted. Also, when this happens, the operators should proceed to find the next best tuple and continue the join.

We propose a learning algorithm that overcomes the mentioned challenges and is asymptotically effective. The proposed algorithm has two stages. First operator R explores a limited number of tuples to learn the tuple with maximum reward $r_{max} \in R$. Then, R will reuse r_{max} until there is no hope of producing any joint tuples



(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

using r_{max}. Next, R removes r_{max} from its set of available tuples and repeats the previous steps to learn the next best tuple in R with the largest reward and produces its joint tuples. The operator R will continue this process until the required number of join results is generated. We define each iteration to find an r_{max} a *super-round* of our algorithm.

To find r_{max} at each super-round, Bandit Join models relation scanning as an infinitely many-armed bandit problem [2, 3, 15]. These problems assume the set of available actions is too large to be fully explored. Thus, they aim at effectively estimating the most rewarding action(s) using a sufficiently small random sample of the set of actions. There are different algorithms to solve infinitely many-armed bandit problems. Bandit Join uses *m-run* algorithm [2] to find the r_{max} at each super-round. In this approach, to compute $R \bowtie S$, operator S performs a sequential scan and sends a tuple of S to the join operator in each round. Operator R first starts by sequentially scanning its relation but maintains a mapping from the scanned tuples' addresses to their total observed rewards in the main memory. As long as the current tuple $r \in R$ produces a join result, R keeps reusing this tuple. As soon as r fails to produce a result, R performs a `next` action and moves to the next tuple in the subsequent round. If there is a tuple that has m consecutive successes, operator R stops its scan and declares that tuple as the estimated r_{max}. Otherwise, it stops scanning after reading m distinct tuples. In this case, R picks the tuple with max reward from the set of seen tuples 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 tuple with a relatively small number of I/O accesses. This approach will result in an asymptotically optimal strategy by setting m equal to the square root of the total rounds of the algorithm, i.e., rounds of the join. We do not include the asymptotic analysis in this paper due to the lack of space.

4 EXPERIMENTS

We evaluate our method against block nested loop join [7]. We do not compare against sort-merge and hash join as they contradict our assumptions mentioned in the Introduction.

4.1 Experiment Setting

4.1.1 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` For each query, we report the time to obtain k join results.

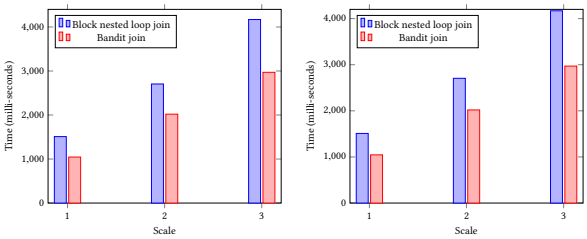
4.1.2 Implementation, Hardware Setup, and Operating System. We implement bandit join inside PostgreSQL 11.5 database management system, on a Linux server with Intel(R) Xeon(R) 2.30GHz cores and 500GB of memory. Multi-threading is turned off for the database server and the size of available cache and memory for database is set to minimum possible value (less than $128KB$). The queries are submitted to PostgreSQL using python and psycopg2 library.

4.2 Evaluation of Bandit Join against Block Nested Loop

As our baseline, we implement block nested loop join (improved version of PostgresSQL's nested loop). In block nested loop, instead of reading one tuple from R and joining it with S, we read R in groups. This reduces the I/O access of the nested loop join if tuples of R are not clustered on the disk. We set the group size to 32.

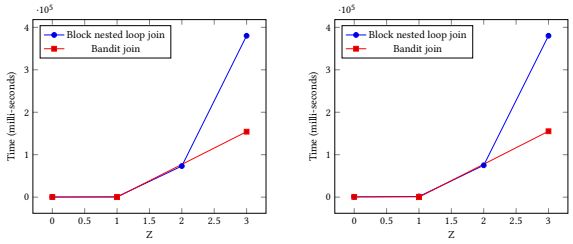
One of the parameters that impact the query processing time is the probability distribution of the attribute values and more specifically the skew in the data [6]. We evaluate the query runtime over data-sets with different skews by assuming a Zipfian distribution with parameter z over the data [6]. 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, (ex. 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. In this case, the m-run algorithm identifies r_{max} more effectively.

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



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

Figure 2: Impact of database size on response time



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

Figure 3: Impact of data skew on the response time

Figure 1a shows the running time of Q_{14} using block nested loop and bandit join over a dataset with zero skew (uniform distribution). Block nested loop outperforms bandit join for $k = 10$ and $k = 50$. The reason is that, with uniform distribution, it is difficult for bandit join to learn which tuple group can produce more joint results. However, as we increase k, bandit join has more time to learn and outperforms block nested loop. Figure 1b shows the same results for datasets with $z = 1$. This figure shows that for a slightly skewed data, the bandit join is doing much better than block nested loop. The reason is, for this distribution, the bandit join is able to learn a proper tuple group. On the other side, block nested loop has a high chance of getting unlucky and facing "useless" tuple groups, until it finds a group that actually generates sufficiently enough join results.

4.3 Scalability

Next, we evaluate the impact of database size on the performance of bandit join and block nested loop. Figure 2 show the response time of bandit join and nested loop join on three databases with different sizes. We see that as the database size grows, bandit join outperforms block nested loop 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, because it will skip a tuple/group after first failure, it still outperforms block nested loop in generating k results.

4.4 Skew Resilience

Roughly speaking, as the skew in data increases, the performance of bandit join becomes much better than block nested loop as shown in Figure 3a and 3b. However, there is an optimal point for performance of bandit join based on data skew. More specifically, if the z is too small or too large, Bandit Join may have slightly worse performance than a medium value for z. This characteristic is more obvious in response time of Q_{14} shown in the figure.

5 RELATED WORK

Recently, there has been a myriad of approaches to utilize machine learning techniques in designing different components of database managements systems. In [11], Kraska et al. propose learned database indexes that outperforms a B-Tree index in specific cases. In their follow-up work, the authors propose SageDB [11], a database system that its core parts are learned components. In [10], authors address the problem of operator selection for query optimization. Marcus and Papaemmanouil propose a method that models join order enumeration as an RL problem [12]. In [14], the authors present a learning system called SkinnerDB to find the optimal join order for a query without having previous training data. Ortiz et al. propose a method to learn states for query optimization [13]. Their objective is: 1) to learn a useful representation for each subquery; 2) to use the learned representation to find an optimal query plan using reinforcement learning. None of these methods consider the selection operator and our proposed method is an orthogonal approach to the mentioned techniques.

6 ONGOING WORK AND FUTURE CHALLENGES

We would like to extend the bandit join to support joins with more than two relations. In this setting, the reward of an agent is based on the final joint tuples rather than the immediate results of an intermediate node in the join tree. As another exciting ongoing work, we would like to extend single learning agent setting to the case where all the agents can learn their strategy.

The proposed bandit join algorithm in this paper shows that there is room for improving query processing using bandit problem framework. We believe bandit join introduces an exciting research path in query optimization where each query operator is an agent and the query processing is modeled as a multi-agent collaborative learning problem. In this setting, agents interact with each other with the common goal of achieving maximum efficiency in query processing.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. . BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys '13.*
[2] Donald A. Berry, Robert W. Chen, Alan Zame, David C. Heath, and Larry A. Shepp. . Bandit problems with infinitely many arms. *The Annals of Statistics '97.*
[3] Thomas Bonald and Alexandre Proutière. . Two-target Algorithms for Infinite-armed Bandits with Bernoulli Rewards. In *NIPS '13.*
[4] Michael J. Carey and Donald Kossmann. . Reducing the Braking Distance of an SQL Query Engine. In *VLDB '98.*
[5] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. . On Random Sampling over Joins. In *SIGMOD '99.*
[6] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. . Scalable and adaptive online joins. *VLDB '14.*
[7] Hector GarciaMolina, Jeff Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book.*
[8] Peter J. Haas and Joseph M. Hellerstein. . Ripple Joins for Online Aggregation. In *SIGMOD '99.*
[9] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. . Online Aggregation. In *SIGMOD '97.*

[10] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. . Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv '18*.
[11] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. . The case for learned index structures. In *ICDM '18*.
[12] Ryan Marcus and Olga Papaemmanouil. . Deep reinforcement learning for join order enumeration. In *aiDM '18*.
[13] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. . Learning state representations for query optimization with deep reinforcement learning. *arXiv '19*.
[14] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. . SkinnerDB: regret-bounded query evaluation via reinforcement learning. In *SIGMOD '19*.
[15] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. . Algorithms for Infinitely Many-armed Bandits. In *NIPS '08*.