CSE 541: Final Project Report

Reinforcement Learning Based Query Evaluation Using Dynamic Time Slices

Saptarashmi Bandyopadhyay
szb754@psu.edu

Kirti Jagtap
ktj35@psu.edu

Bodhisatwa Chatterjee
bxc583@psu.edu

Instructor: **Dr.Wang-Chien Lee**

Pennsylvania State University, University Park, PA

1 Introduction

Traditional database systems makes strong assumptions about the query and the data while selecting query plans. If those assumptions hold true, then we end up with an optimal query plan. However, if the prior assumptions fail then we select a query plan with is sub-optimal and often degrading in performance. For example, for queries containing join operations, the query traditional optimizers enumerates possible join-orders and uses a pre-defined cost model to select the best join-order from this list. If the cost model and the cardinality estimation are accurate, then we get an optimal query plan. However, the calculation of cardinality is based on several simplifying assumptions like uniformity of data, which are not always necessarily true.

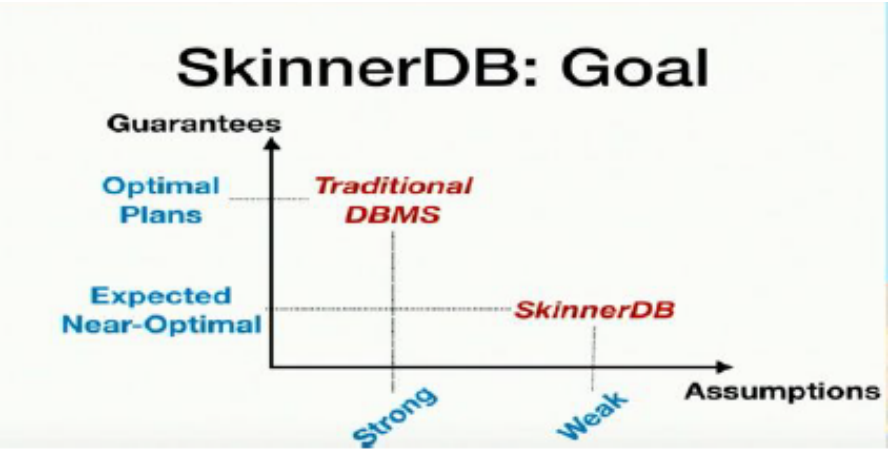

Figure 1: Query Plans in Traditional Database vs SkinnerDB [8]

The goal of SkinnerDB [9], which is the state-of-art query optimizer, is to avoid these strong assumptions about both query and data and give probabilistic guarantees on the expected query execution time. SkinnerDB provides near-optimal query plans for a given query and provides an estimate on expected vs optimal query execution time. In SkinnerDB, the major focus is *Join-Ordering* for queries containing multiple joins. The reason for this is that selecting the ordering of joins is often the major performance bottleneck for query execution. For example, let us consider that we have three relations in our database: **customers**(customer_id, customer_name, customer_location), **products**(product_id, product_location) and **orders**(order_id order_product_id, order_customer_id, order_active). Let's say that a user queries the list of all customers' names along with the location of each product which they have ordered. In this case, if we join **products** and **customers** first, we have to perform the entire cross-product between the tables as there are no join predicates which relate to these tables. Instead, if we choose to join **customers** and **orders** first, then the sub-join of orders joined with customers will contain one entry for every order by customer.

This is much smaller than the earlier cross-product and would save a lot of computations.

However, we have identified that there are multiples issues in this system. First of all, SkinnerDB assumes that any query can be divided into multiple fixed time slices. This might not be possible for all cases. Also, sometimes it happens that a query might finish executing before the allocated time slice has elapsed; in this case we are wasting the remaining portion of the time slice instead of aggregating the past result fragments. Also, it might not make sense always to execute each of the selected join-order for the same number of fixed steps. Furthermore, as it is often the case with all reinforcement learning applications, the subsequent states are dependent on the choice of initial state and the initial join-order selected by SkinnerDB is random, which might not always be the best case. The task of combining result fragments from the previous join-orders with the current join-order strategy is not necessarily a trivial task.

Based on all these limitations, we have extended SkinnerDB to address those challenges. Our major contributions to the research work on Skinner DB are as follows:

- We have introduced the notion of Dynamic Time Slices and Elastic Time Slices and have done a quantitive comparision between them
- We have introduced a new scheme for identification of the best selection policy
- We have modified the original work on exploration weights and created a scheme for identification of the Best Exploration Weight Policy
- Generalization of our results on Permuted Dataset
- Generalization of our results on new query workload

We have demonstrated that our proposed changes to the existing research on Reinforcement Learning based Query Optimization significantly reduces the query execution time and leads to higher rewards. We identified the best selection policy to be Max Reward Policy and the best exploration weight policy to be Reward Average Policy. A combination of the best selection and the best exploration weight policy is leading to the least execution time for majority of the queries in the query workload. We have permuted the columns of the IMDB dataset, so as not to violate the integrity constraints for the tables in the schema. Then we have regenerated the Join-Order Benchmark in Skinner DB format for the permuted dataset, after decompiling and analyzing the source executable that generates the Skinner DB format files as the original source code was unavailable. It has been observed that on repeating the same experiments on the permuted dataset, a similar trend is observed for the execution time and rewards obtained for queries in the workloads. We have also changed the queries by writing new SQL queries and modifying existing queries and the execution time and rewards for the new query workload is similar to the original query workload. This demonstrations that our modifications to the Skinner DB project can be generalized.

The organization of this report is as follows:

- We have presented an extensive literature survey for the works on query optimization and join-ordering in general.
- We have introduced the architecture of SkinnerDB and explained its working in detail
- We then present the new contributions which we are making to SkinnerDB
- We present the summary of 41 experiments which we have performed based on our implementations
- We show how our results can be generalized in terms of different datasets

2 Related Work

In this section, we will extend upon the literature survey presented in our progress report, where we explained about several traditional query optimization techniques and how they were limited because of their static nature. In this progress report, we will focus specifically on the join-ordering problem and how approaches has evolved over time and where does our work stand.

2.1 Prior works on Join-Order Optimization

There have been numerous attempts to solve this problem of join-ordering in database research.Earlier approaches used the concept of dynamic programming to solve this problem [9]. An access path for each relation was defined and a cost was assigned to it. Based on the paths, several possible plans for evaluating a particular query was generated and the selection of the plan was done by calculating the cost of the path. The recurrent nature of the access path gave the opportunity to exploit the optimal substructure property in the join-tree and hence the scope of dynamic programming came into view. Attempts were made to improve the performance and several other techinques like Kruskal Algorithm were also used to select the most efficient join-tree[5]. Another traditional strategy for join-order enumeration was to define cost for each relation and then combining a pair of relations. The combination of the relations with the lowest cost was done based on greedy method[7] and PostgreSQL[8] uses this technique for join-order enumeration.

One of the successful attempt was to determine the join structure of the query prior to the query optimization and then the tables were ordered with the help of graph theory [2]. The major drawback of this method was that it is not always possible to determine the structure of the join-ordering before the actual join and also ordering of tables is an included overhead. On a similar domain, the join-ordering problem was approached by genetic algorithms and ant-colony algorithm, where the main idea is to find the best path of join by probabilistic simulations with fairly short execution time. This method creates a problem when the sample size of queries and relations are not large enough for to be simulated probabilistically [3].

The concept of using Deep Reinforcement Learning to solve the problem of join-order enumeration was first proposed in [7] as **ReJoin Framework**. The problem with the classical strategies which involved dynamic programming and greedy method to find out the lowest cost relation pairs [5], [**?**] was that they were static in nature; meaning that they didn't learn from the past experiences and did not take any feedback bases on prior computations. Therefore, this idea of join-order enumeration was formulated as a reinforcement learning problem, in which an agent has access to set of relations, which represent the state vectors and each action represents the joining of the relations and subsequent actions result in a new relation state with a reward associated with it. The objective is to maximize this reward and hence find out an optimal set of join-actions. Basically, each state can be visualized as binary join sub-tree and each action can be thought of as putting the two sub-trees together to get a new tree. Also, in reinforcement learning an agent can act as long as the environment reaches a terminal state. Here, the terminal state is the complete binary join-tree and at that point a reward is assigned. There are no rewards for intermediate sub-trees and in this way, the drawback of not learning from experience in handled in ReJoin. This brings us to main focus of our proposal SkinnerDB.

3 Skinner DB

SkinnerDB [9] is the current state-of-art in terms of query evaluation by reinforcement learning approaches. It is different from all prior approaches since it does not maintain apriori data statistics or cost model. It works on *Intra-Query* Learning, rather than *Inter-Query* Learning, which means that we don't learn from past queries, but rather from the current query to optimize its remaining execution time.

The major takeaways from this model is that data statistics of previous queries and cost of cardinality models are not used. The best join orders for each given query is selected and is executed in equal time slices, which gives us tuples as result. The result tuples obtained from each time slice are merged to obtain the final result.

3.1 Working Mechanism

SkinnerDb divides the execution of query into small steps called *micro-episodes*. This term is used interchangeably with *time-slices* in our work. Each micro-episode is a fixed time interval of 10 ms. Initially, a random join-order is selected. Then that join-order is executed for a fixed steps of K times. During this time, fragments of the results for the query is generated and stored. After this execution, the progress reward is obtained for the prior join-order and a better join-order is selected based on that.

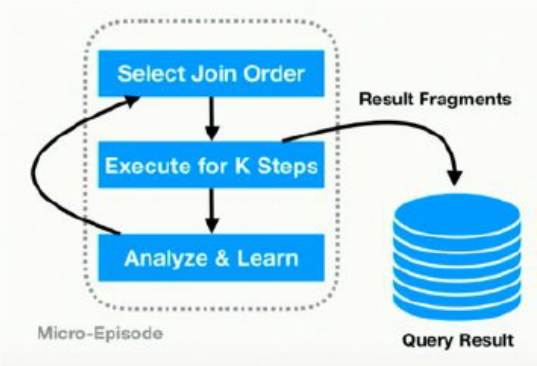

Figure 2: Each micro-episode of SkinnerDB [9]

3.2 Architecture

The internal components of SkinnerDb is shown in the figure below. The query is first pre-preprocessed inorder to create hash tables on all columns subject to equality predicates. This step is done is parallel to reduce the extra overhead. Then the learning optimizer is used to select a specific join order which is tried out at the beginning of each time slice. This specific join order is selected based on statistics on the quality of join orders that were collected during the current query execution. Those selected join orders are then moved forward to the join executor. The join executor executes the join order until a particular time-interval, which is called the timeout, is attained. Then the result tuples are combined into a result set and an additional check for duplicate results generated by different join orders is also done consequently. In SkinnerDB, the join executor is either a generic SQL processor or a specialized execution engine, which enhances the performance.

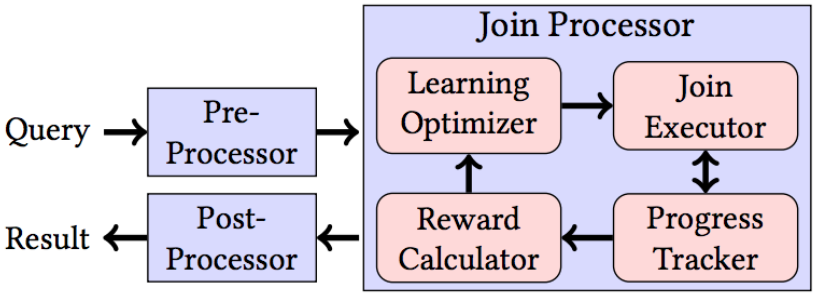

Figure 3: The Components of SkinnerDB [9]

3.3 Important Concepts

This section highlights the important definitions that have been used in our investigative project. The concepts are defined as follows:

1. **Progress**: The progress is SkinnerDB is defined as the percentage of join order completed during execution of a current query which effectively denoted the portion of input data processed on a certain state.

2. **Reward**: In SkinnerDB, both long-term and short-term reward are calculated according to the following expression:

$$R = \frac{1}{2}(P + \frac{N_r}{B}) \tag{1}$$

   In the above equation, N_r is the number of processed tuples, B is the number of fixed time slices and P is the progress. As we can see, the reward is inversely proportional to the time-slice and directly proportional to the number of processed tuples.

3. **Regret**: In SkinnerDB, regret is defined as the difference between actual and optimal execution time.

4. **Total Execution Time**: It is simply the sum of query execution time, pre-processing time and the post processing time.

4 Design Ideas

We have introduced the concept of dynamic time slices as clearly explained in Section 4.1 in order to improve the query execution time for the selected join orders. We have also carried out investigations on finding the best default selection policy and exploration policy as described in Sections 4.3 and 4.5, to ensure that the total query execution time is minimized and the average join order is maximized. The Skinner DB project had used UCB1 default selection policy and Static default exploration weight policy. Repeated experimentation has been conducted by changing the exploration weight, as elaborated in Section 4.4 to find the best exploration weight for the current query workloads and database schema.

4.1 Dynamic Time Slices

For every iteration of the learning process, Skinner DB executes the selected join order in a fixed time slice (say x). In the coding implementation, we noticed that they have considered 500 as the number of fixed time steps to execute the selected join order. The underlying implication is that whichever join order provides higher short term reward will be executed for 500 time steps. Also, if the selected join order does not change across multiple iterations of learning, the fragmented execution of 500 time steps adds additional overhead. For example, if a selected join order required 1002 time steps to be executed, with the current strategy, 1500 time steps have to be utilized for every iteration of the U.C.T. algorithm. To avoid the wastage of time slices, we have introduced the concept of dynamic time slices.

The number of time slices increases by a fixed number (say y) for every iteration of the learning algorithm. In our experiments of dynamic time slices, we have considered $y = 5$. This means that initially the the selected join order is evaluated in 500 time steps. For the next iteration of the learning algorithm, it executes in 505 time steps. Thus the total execution is completed in 1005 time steps instead of 1500 as described in the previous paragraph, thereby reducing the wastage of time slots by $\frac{500-3}{500} * 100\% = 99.4\%$. It has been observed in the Section 7 that the idea of of dynamic time slices have drastically improved the query execution time by reducing the query evaluation overheads.

4.2 Elastic Time Slices

In the final phase of the project, we have worked on modulating the change in the fixed number of time slices to the short term reward and the long term reward of the best selected join order. This is because, the rewards are not same across every iteration of reinforcement learning although the execution time steps are fixed in SkinnerDB. Join orders which provide higher reward finish the execution quickly, while those with lower join order do not. For example, in 1 iteration of learning, the selected join reward had a reward of 0.3 which is executed in 431 time steps, while in another the case, the selected join order may have a reward of 0.18 and is executed in 497 time steps. If the execution of join order is iterated for a fixed time slice of 500, time is wasted for the first join order with higher reward and lower execution time. We have devising a formulation to dynamically change the value of the fixed time slices based on the short term reward, scaled to a constant of 5. The additional time slices have been converted to integer as it is observed that the rewards are floating point values.

The scheme for scaling the time slices with local reward is:

$$new\_timeslice = prev\_timeslice + integer(5 * local\_reward) \tag{2}$$

The scheme for scaling the time slices with global reward is:

$$new\_timeslice = prev\_timeslice + integer(5 * global\_reward) \tag{3}$$

4.3 Selection Policy

The selection policy essentially describes the strategy to select the actions (Tables being joined) for each search tree node in the UCT search tree. We have experimented by selecting each of the following 5 policies as default

selection strategies to study the impact in terms of increasing average reward and decreasing the execution time. SkinnerDB uses UCB1 as the default selection policy. The 5 selection policies supported by Skinner DB that we have experimented as the default policy are described as follows:

- UCB1: The UCB1 (Upper Confidence Bound-Variant 1) approach uses the UCT formula for calculating reward for every node in the UCT search tree [7]. The UCT formula is described in Section 4.4. The node with the highest reward computed with UCB-1 approach is selected for traversing the search tree.
- MAX REWARD: Actions are selected to ensure maximal reward in the long term.
- EPSILON (ϵ) GREEDY: The best action is selected after exploration for (1- ϵ)% of time and random action for ϵ % of time. It ensures a trade-off between exploration and exploitation as described in [6].
- RANDOM: The selected actions follow an uniform random distribution. There is no dependence with the obtained average reward.
- RANDOM UCB1: In this approach, the initial join order is selected randomly instead of starting from the root node in the UCT search tree. After that, the UCB1 strategy is adopted.

4.4 Exploration Weights

One of the key factors in selecting a node is to maintain a balance between exploration and exploitation. The UCT (Upper Confidence Bound applied to trees) formula for balancing exploration and exploitation is given below.

Each node in the search tree is selected based on the following formula and the node giving highest value.

$$r_c + w\sqrt{\frac{\log v_p}{v_c}} \tag{4}$$

where,

r_c is the average reward for c
v_c number of visits for child node
v_p number of visits for parent node
w is a weight factor

In the above expression,
r_c (the first part of the formula) represent the exploitation part of the algorithm

$w\sqrt{\frac{\log v_p}{v_c}}$(the second part of formula) represent exploration.

The sum of these two parts represent the upper bound of the confidence bound on the max reward possible passing through the corresponding node.

As regret is defined as the difference between the sum of obtained rewards to the sum of rewards for optimal choices[9]. The paper claims that taking $w = \sqrt{2}$ is sufficient to obtain bounds on expected regret but also suggests to try different values to optimize performance for specific domains. We tried different values for w as a part of the experiment and the results are given in section 6.4

4.5 Exploration Weight Policy

The exploration weight policy in the UCT algorithm defines how the exploration weights in the UCT algorithm are updated over time. We have experimented by selecting each of the following 4 exploration weight policies with the previously described 5 selection policies in Section 4.3 to determine the best default exploration weight policy and selection strategy that increases average reward and decreases the execution time. SkinnerDB uses STATIC as the default exploration weight policy. The 4 exploration weight policies supported by Skinner DB that we have experimented with are described as follows:

- STATIC: The exploration weight (w_e) is not updated over time.
- REWARD AVERAGE: w_e is updated on the short term average reward.

- SCALE DOWN: w_e is scaled down with an exploration factor of 10 times over the number of iterations in the learning algorithm.
- ADAPT TO SAMPLE: w_e is selected based on the initial reward sample.

5 Installation and Set-Up

The code repository of SkinnerDB project from the CornellDB group is available on GitHub.

Steps for installation and setup of the project in eclipse:

1. Clone and add the project in Git prospective of eclipse.
2. Import the project in Java prospective of eclipse using local git repository created in step 1
3. Run as 'Java Application' using 'SkinnerCmd.java'
4. The project has two arguments, which are set in Run Configurations:
   (a) Program Argument: Pass the path of folder where database is stored.
   For example, '/home/git/skinnerimdb/'

   (b) Virtual Machine arguments: The github library mentions the two settings for VM argument which work best for the benchmarking platform provided in the project. We had to set a virtual heap space of 16 GB due to the memory intensive nature of the large scale database system:
      i. Garbage collector: -XX:+UseConcMarkSweepGC
      ii. Heap space: -Xmx16G
5. As the code in the above GitHub library is under development, JAVA SE-1.8 is the compatible version for the project.

Below is the screenshot of the code repository in eclipse after successful setup.

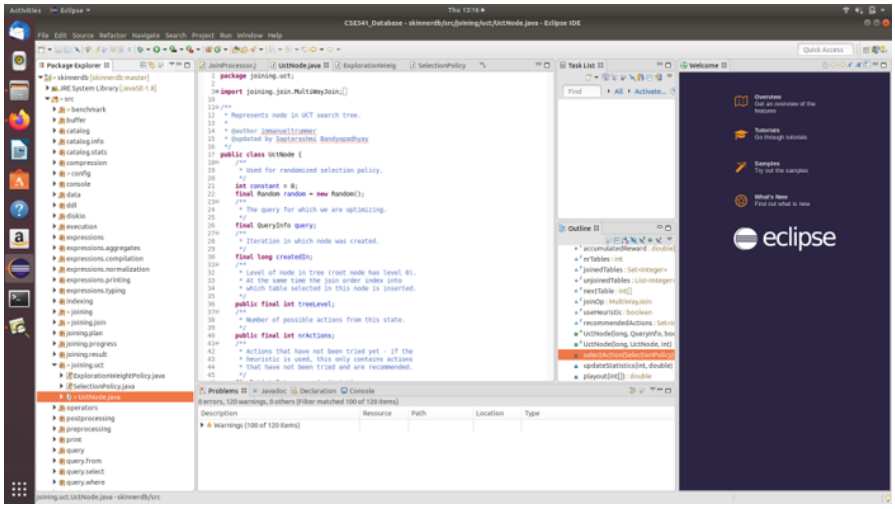

Figure 4: Code Organization in Eclipse I.D.E

6 Experiments

The Join-Order Benchmark, which is a subset of IMDB database has been used from the Skinner DB repository which has already been translated to the Skinner DB format. The IMDB database comprises of 21 relational schemas and more than 100 attributes, and provides us a detailed insight in operating the database system over large amounts of data. We also considered 113 SQL queries in the workload which are SPJ (Selection, Projection, Join) queries.

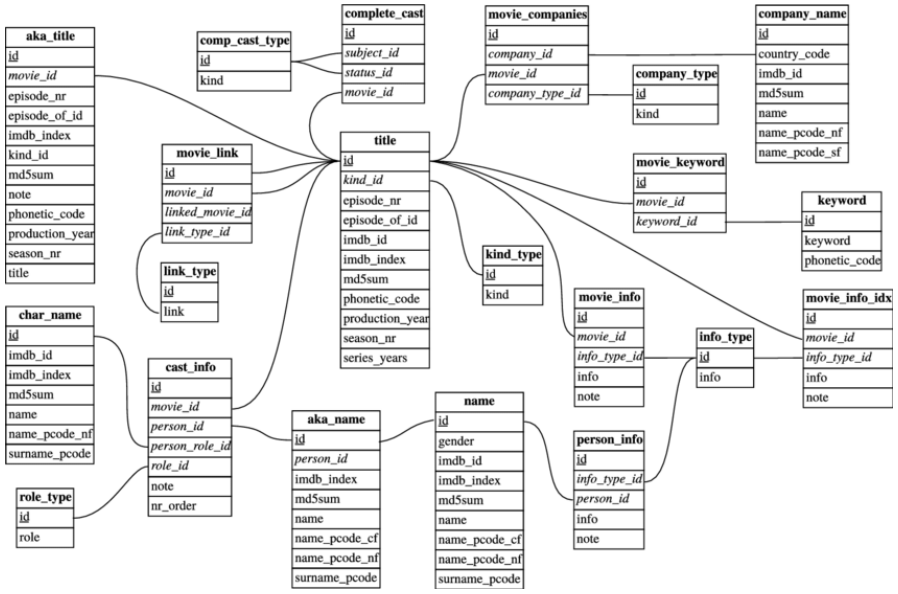

Figure 5: Schema of Join-Order Benchmark [4]

We have conducted 26 experiments with dynamic time slices, to select the default selection and exploration weight policy and to identify the best exploration weight for our research. The parameters have been described clearly in the subsequent Sections.

6.1 Dynamic Time Slices

We selected the number of time slots for which the fixed number of time slices will increase for each iteration of the learning algorithm to be 5. In another experiment, We also increased the number of time slices to 600. The results have been compared with the baseline system of 500 fixed time slices and no dynamic time slices. The experiments were repeated a number of times in order to obtain a reliable estimate.

6.2 Elastic Time Slices

The idea of increasing the timeslots by a fixed number (5, in our case) does not often reflect the impact of the particular dataset and the query workload on the query execution. Instead we have incorporated the idea of scaling the timeslot based on the local reward, while traversing each node of the UCT search and the global reward after complete traversal of the tree. This allows us to reduce the wastage of time in query execution. It is to be noted that the local reward is between 0 to 1 and the global reward may be a floating point value. So we have scaled them by a constant (5, as we did in dynamic time slices) and then type-casted to integer in the JAVA code.

6.3 Identifying the Best Default Exploration and Weight Policy

We conducted 20 experiments of query optimization by considering all combinations of the 5 selection policies described in Section 4.3 and 4 exploration weight policies, described in Section 4.5 as default policies. One of the experiments is the base case followed by Skinner DB where UCB1 is the default selection policy and STATIC is the default exploration weight policy. All the results have been compared to the base case. The experiments were repeated a number of times in order to obtain a reliable estimate.

6.4 Identifying the Best Exploration Weight

We tried four different exploration weights close to the weight used by the author ($w = \sqrt{2}$).

The below results were obtained using $w = \sqrt{1.5}, w = \sqrt{2}, w = \sqrt{2.5}, w = \sqrt{3}, w = \sqrt{3.5}$ in section 7.6. We run this experiment with default selection and exploration policy. The experiment was run number of times for each weight.

6.5 Reverse Engineering of source code to use new dataset

After executing our basic experiments, we have tried to use a new dataset to see if our conclusions can be generalized. However, the major problem was that the source code to create the database in SkinnerDB format, CreateDB.jar, does not have any associated source code being provided. We therefore decompiled the source executable file to understand the underlying functionality for creating the new database, as demonstrated in Figure 6. The JD-Compiler (version1.6.3) has been used for this purpose that decompiles the executable files to the code files. However, this reverse engineering approach by decompilation is very challenging as decompilation tools are not perfect and there are many errors like incomplete code that we had to assume, in order understand the functionality. The CreateDB.jar creates the converted database files, schema and a dictionary by taking the intial database, stored as csv files.

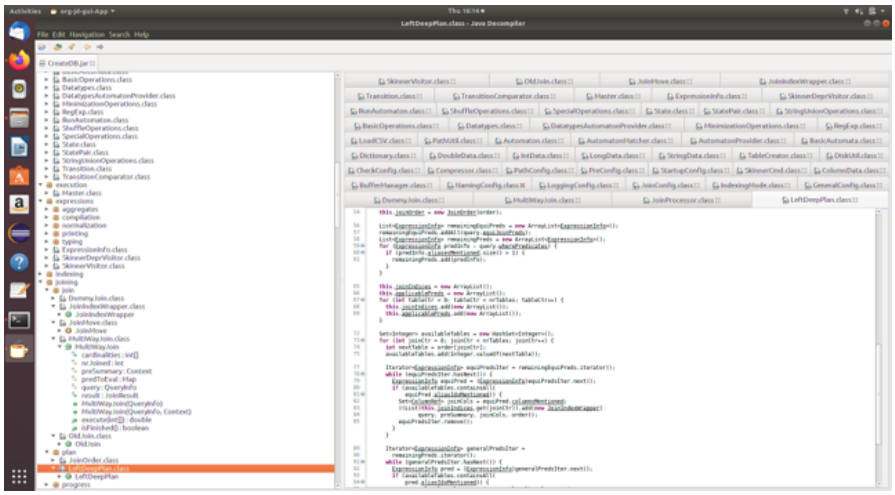

Figure 6: Decompiled code from executable to analyze dataset creation

6.6 Generalizing SkinnerDB

So far all the ideas which we have presented are based on the original implementation of SkinnerDB and the default dataset used by the system in SkinnerDB format. To make sure that all our discussions and the results which we obtain are still valid for general discussions, we need to either modify the dataset or use a new dataset altogether. This is a specially challenging problem as obtaining the raw csv/text files is not enough; it needs to be compiled into binaries which the SkinnerDB uses. In this section, we will show how our dataset is modified and what are the techniques that we followed to achieve that. We will also show how we generated a completely new dataset and modified it to work with our system.

Our goal is to use new databases in SkinnerDB format, to generalize the results from our ideas. Given the nascent nature of Reinforcement Learning based query optimization, generalization in our results will be significant in the domain of query optimization. However, it is extremely challenging as the source code to create new database in the Skinner DB format is not provided, and there is significant reverse engineering effort by decompiling the provided CreateDB.jar file.

6.6.1 Permuting the given dataset

The simplest modification which can be done is simply to change the column values in all the tables in the dataset. Changing column values will result in creating of new records and will ensure that the learning is done a different dataset. The reason of changing columns is that we are not violating the integrity of the data types present in the columns. This process is illustrated in the fig 7 and 8.

6.6.2 Modifying Queries to change workload

Another simple modification to generalize the results of the system is to simply change the queries on which the system is tested. We have hand-modified 10 selected queries to check if the query plans are modified or not.

	A	B	C	D
1	1	2	619801	6
2	2	50	1450538	10
3	3	50	257907	6
4	4	50	260637	6
5	5	50	766538	6

Figure 7: Original csv file(*movies_link.csv*)

	A	B	C	D
1	4	50	1450538	6
2	3	2	619801	6
3	2	50	766538	6
4	5	50	260637	10
5	1	50	257907	6

Figure 8: Permuted csv file(*movies_link.csv*)

6.6.3 Experimentation with a New Dataset

The FIFA17 Dataset is much smaller (10 MB) than the Join-Order Benchmark (3.36 GB) with 4 tables: FullData, ClubNames, NationalNames and PlayerNames [1]. We had to write 10 queries by ourselves which are different from the original query workloads.

7 Results

For each of the 26 experiments, conducted we generated a system log file highlighting the selected join queries in every iteration of the learning algorithm for each query in the workload. We also generated an output execution log at the end of the experiment, which contained statistics such as execution time and average reward.

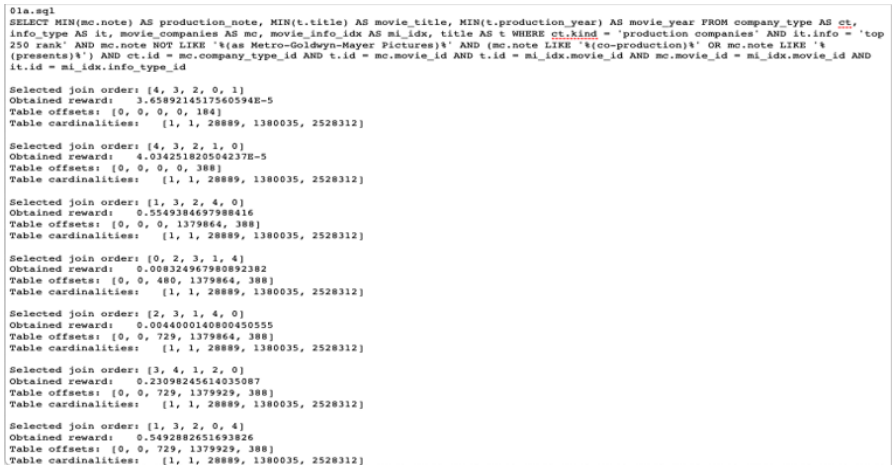

Figure 9: Selection of best join order for 1a.sql query

A representative image of the system log file for the query 1a.sql while experimenting on exploration weights has been shown in Figure 9. Each step shows the selected join order. The best join order is obtained, when the learning algorithm terminates after re-constructing the query from the intermediate join orders.

Query	Millis	PreMillis	PostMillis	Tuples	Iterations	Lookups	NrIndexEntries	nrUniqueLookups	NrUctNodes	NrPlans	JoinCard	NrSamples	AvgReward	MaxReward	TotalWork
01a.sql	320	294	0	1764	3433	0	1410	1533	9	7	142	7	0.11410326281897900	0.5549384697988420	2.025311170755360
01b.sql	890	840	0	881	1932	0	834	893	6	4	3	4	6.692699843730889E-05	1.27769581431916E-04	1.000532502340190
01c.sql	547	520	0	1461	3041	0	645	1516	9	7	3	7	0.07333685140912460	0.5029891287489510	2.020291613836410
01d.sql	1977	1946	1	1198	2550	0	561	1124	8	6	4	6	0.07460383344751540	0.4473324765795490	1.895076431629640
02a.sql	216	38	0	210877	520674	0	1458342059	183259	46	16	7834	1042	0.01064830157859060	0.4239042256652820	2.001003802888050
02b.sql	348	41	0	293624	754031	0	1458102399	330713	71	25	5228	1509	0.00573404585702077	0.4456090918905670	2.008913781629320
02c.sql	33	24	0	650	1519	0	1339447	624	6	4	0	4	1.74563921456986E-04	6.66234888691912E-04	1.001395463241580
02d.sql	1128	52	0	305341	683551	0	1458557330	225280	45	15	68316	1368	0.05244342781625700	0.4392351627485530	2.000391528068170
03a.sql	877	842	0	17079	49103	0	111107977	19057	20	8	206	99	0.00736222522336600	0.1749451889448860	1.002471329451920
03b.sql	361	336	0	5560	11845	0	3376600	5982	20	9	5	24	0.04695103114904110	0.2704678362573100	1.001935395254330
03c.sql	1466	1399	0	30536	75128	0	111140227	30260	27	9	7250	151	0.05138149545868360	0.2183120071212900	1.000849055942700
04a.sql	813	756	0	20254	54985	0	135114789	20908	34	13	740	110	0.00985789266159480	0.1081657889639200	1.001804294254460
04b.sql	279	230	0	3968	8550	0	3408781	4128	21	11	6	18	0.04482774095614040	0.2936619121667720	1.872743886965510
04c.sql	2609	2258	0	39664	91921	0	158723980	32464	33	14	4700	184	0.02799892557650860	0.1474245892583790	1.001338662310360
05a.sql	824	785	0	1142	2502	0	248	1242	8	6	0	6	0.00287348006750266	0.0132788215264767	1.025574064789900
05b.sql	978	961	0	501	1002	0	1	501	5	3	0	3	0.02945948606429310	0.0881523272143870	1.176049888720220
05c.sql	3568	3535	0	4710	10424	0	3144	4792	22	11	669	21	0.10698569388289400	0.5398207577578000	2.018147141055670
06a.sql	2177	2025	0	7528	18024	0	1544257	7950	37	19	6	37	0.01836721654555220	0.2714223893079060	1.497724662079350
06b.sql	465	437	0	1817	4035	0	361091	1976	11	7	12	9	0.04018437960359370	0.2834229932629030	1.137041700685960
06c.sql	693	676	0	1748	4095	0	78390	1808	11	8	2	9	0.08488166454977350	0.2734229977510420	1.905486895330080
06d.sql	2628	2272	1	23416	61905	0	5149805	23907	38	15	88	124	0.00311588805722324	0.2754197500973350	1.000167753306070
06e.sql	5009	4149	1	260774	526894	0	1822177	246800	73	26	6	1054	6.39008490361331E-04	0.2714220646364890	1.675117038484020
06f.sql	13626	901	9	1625085	3271790	0	463221764	826390	44	14	785477	6544	0.12018606124670600	0.162000000000004	1.000337672652800
07a.sql	4243	1929	0	344088	798373	0	28204909	345237	161	110	32	1597	0.00156833815141898	0.2037818254021630	2.675926692024660
07b.sql	2904	2658	0	5062	11274	0	147348	6885	26	23	16	23	0.01290367099697100	0.1510216818069200	1.360371389797060
07c.sql	28921	3597	1	3706441	9899166	0	561823248	4382205	304	208	68185	19799	0.00570716233219530	0.2777777665312500	1.004445902364300
08a.sql	4874	4721	0	28508	59609	0	62018	29741	47	37	62	120	0.02532881780638000	0.3487316126611690	1.641293014461550
08b.sql	4641	4552	0	1714	3695	0	12493	1741	10	8	6	8	0.04185253439648672	0.3141736935885800	1.660070868098250
08c.sql	110806	136	18	21987201	55152726	0	5778175223	18412936	208	143	2487611	110306	0.02259867964344600	0.3728896421605200	1.677998993040100

Figure 10: Output Execution log file for 1 experiment

A representative image of the output execution log file for the same experiment over all the queries have been shown in Figure 10. It demonstrates the actual query execution time, pre-processing time, post-processing time, average reward, join cardinalities among other statistics for each of the 113 queries in the workload.

In the graphs presented below for each of the parameters experimented upon, the data points refers to the average reward in the Y axis and the execution time in the X axis for every query. This is because the SkinnerDB paper mentions that the average reward is a measure of the execution time.

7.1 Visualization of the UCT search tree

For each query we have obtained the visualization of UCT (Upper Confidence bounds applied to Trees) search tree obtained during join order learning.

SQL QUERY

```
SELECT MIN(mc.note) AS production_note, MIN(t.title) AS movie_title, MIN(t.
    production_year) AS movie_year
FROM company_type AS ct, info_type AS it, movie_companies AS mc, movie_info_idx
    AS mi_idx, title AS t
WHERE ct.kind = 'production companies' AND it.info = 'top 250 rank' AND mc.note
    NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%' AND (mc.note LIKE '%(co-
    production)%' OR mc.note LIKE '%(presents)%') AND ct.id = mc.company_type_id
    AND t.id = mc.movie_id AND t.id = mi_idx.movie_id AND mc.movie_id = mi_idx.
    movie_id AND it.id = mi_idx.info_type_id;
```

```
Tables: company_type, info_type, movie_companies, movie_info_idx, title
```

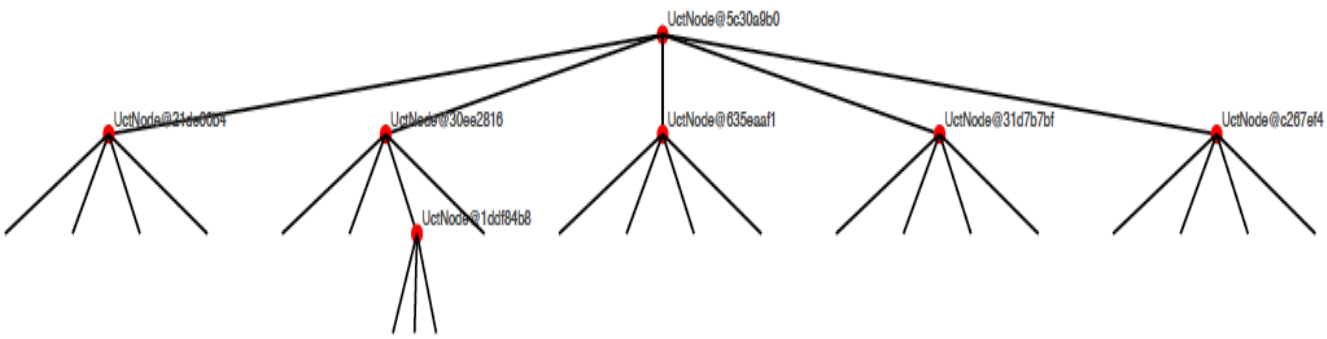

Figure 11: UCT search Tree for the above query

```
UctNode@5c30a9b0
Selected join order: [2, 0, 4, 3, 1]
Obtained reward: 0.004067985247405259

UctNode@21de60b4
Selected join order: [0, 2, 3, 1, 4]
Obtained reward: 0.0015707842453883075

UctNode@30ee2816
Selected join order: [4, 3, 1, 2, 0]
Obtained reward: 3.698119535880065E-5

UctNode@1ddf84b8
Selected join order: [1, 3, 2, 4, 0]
Obtained reward: 0.5549384664081669

UctNode@635eaaf1
Selected join order: [3, 2, 4, 1, 0]
Obtained reward: 0.2964434908714527

UctNode@31d7b7bf
Selected join order: [2, 3, 1, 0, 4]
Obtained reward: 0.004375831407967514

UctNode@c267ef4
Selected join order: [1, 3, 4, 2, 0]
Obtained reward: 0.538
```

7.2 Dynamic Time Slices

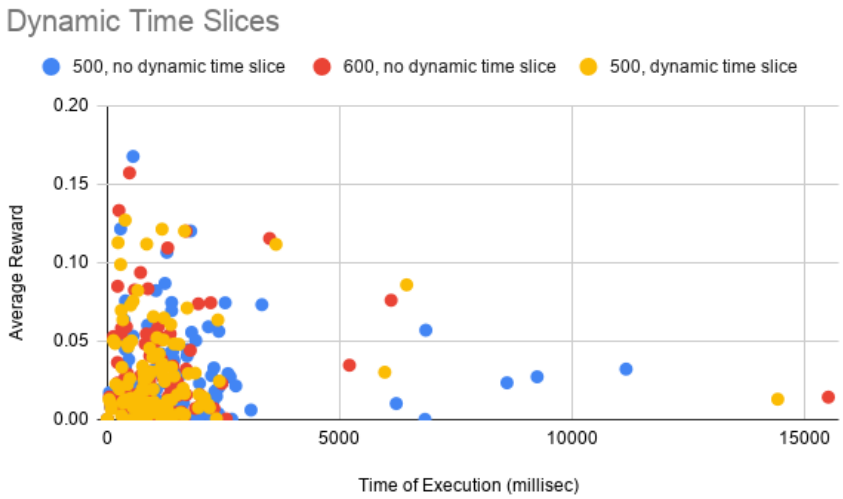

Figure 12: Average Reward w.r.t Total Execution Time for Dynamic Time slices and Increased Time Slices

It has been demonstrated in Figure 14 that the concept of dynamic time slices by increasing the fixed time slices by 5 time slots for every iteration, decreases the execution time over the base case. The data points of dynamic time slices are more clustered towards lower execution time, which highlights the efficiency of the approach, even when

the time slices are increased by a constant.

The example presented in Section 4.1 is thereby validated that fixed time slices waste more resources over dynamic time slices. The average reward is slightly lower than the base case as the average reward is not only dependent on execution time, and the queries finish quickly, thereby not adding any unnecessary reward.

Increasing the fixed number of time slices to 600 does not have much impact on the result.

7.3 Elastic Time Slices

In this section, we will present our results based on the elastic time slices and how they can be varied with scaled global and local reward.

7.3.1 Elastic Time Slices with scaled local reward

It has been demonstrated in Figure 13 that the concept of elastic time slices by increasing the fixed time slices with the integral value of the scaled local reward for every iteration, decreases the execution time over the base case, just like previous cases. The data points of elastic time slices are more clustered towards lower execution time, which highlights the efficiency of the approach, even when the time slices are increased by the scaled local reward. Few data points are outliers compared to the original approach, where none of the data points are executed on elastic time slices.

Figure 13: Average Reward w.r.t Total Execution Time for Elastic Time Slices with Local Reward

The example presented in Section 4.2 is thereby validated that fixed time slices waste more resources over elastic time slices. The average reward is slightly lower than the base case as the average reward is not only dependent on execution time, and the queries finish quickly, thereby not adding any unnecessary reward.

7.3.2 Elastic Time Slices with scaled global reward

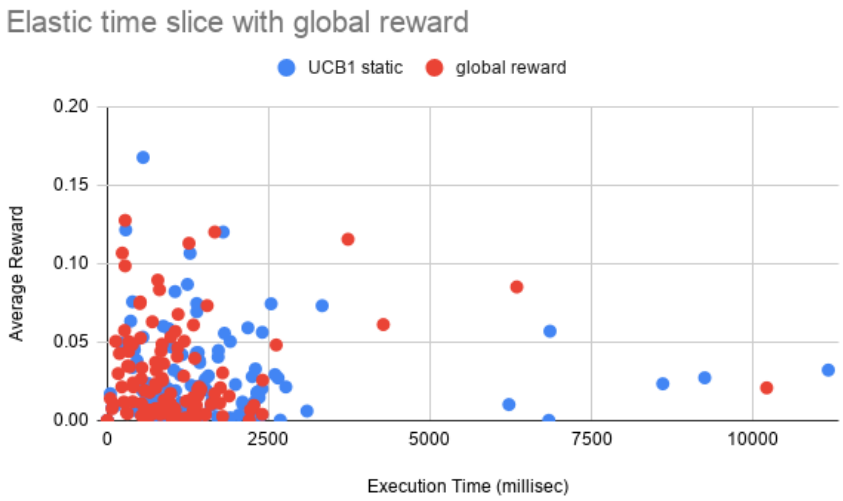

Figure 14: Average Reward w.r.t Total Execution Time for Elastic Time Slices with Global Reward

It has been demonstrated in Figure **??** that the concept of elastic time slices by increasing the fixed time slices by scaled global reward, decreases the execution time over the base case. The data points of elastic time slices are also more clustered towards lower execution time, which highlights the efficiency of the approach, even when the time slices are increased by a constant. There are fewer outliers in comparison to the previous sections.

7.4 Comparison of the time slice schemes

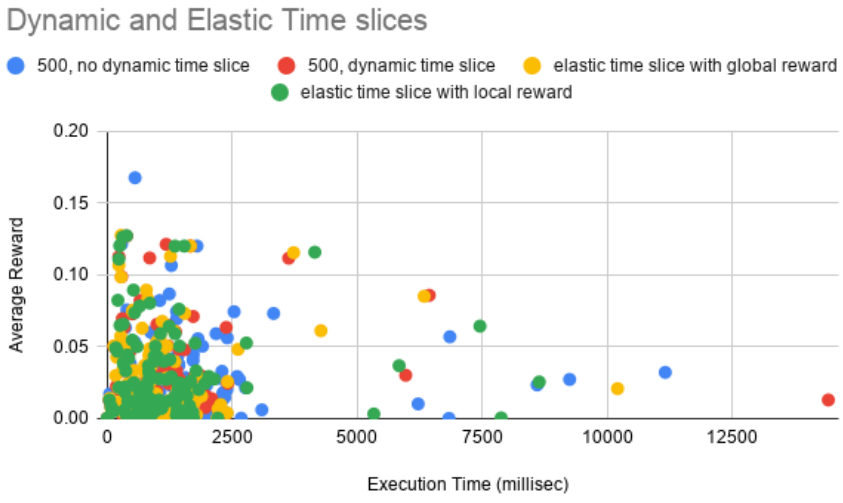

Figure 15: Average Reward w.r.t Total Execution Time for Dynamic Time slices and Elastic Time Slices

The results from dynamic and elastic time slices are compared in Figure 15. It is observed that the data points are most clustered for the elastic time slice approach scaled by local reward. There are few outliers for the elastic time slice strategy with global reward. All the dynamic and elastic time slices have lower execution time and higher reward than the baseline system where these schemes were not implemented.

7.5 Default Selection and Exploration Policy

All the graphs, presented below, have been labeled as (selection policy; exploration weight policy). All possible combinations of selection and exploration weight policy as default strategies have been presented.

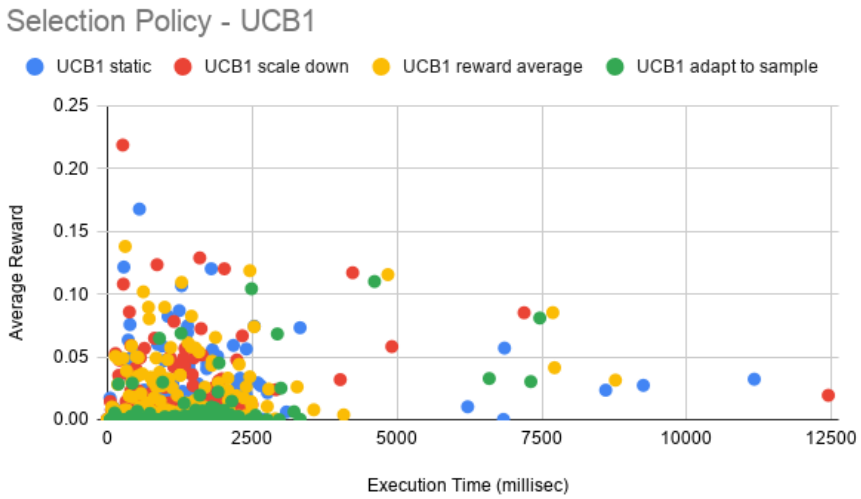

Figure 16: Average Reward w.r.t Total Execution Time for UCB1 as default selection policy and all the 4 exploration weight policies

Without considering some outlier queries, it can be observed in Figure 7.5 that base case of UCB1 selection policy with static exploration weight policies gives the best results. Most of the queries are executed within 2500 time units with higher average rewards. This distribution is however dependent on how the query workload is constructed.

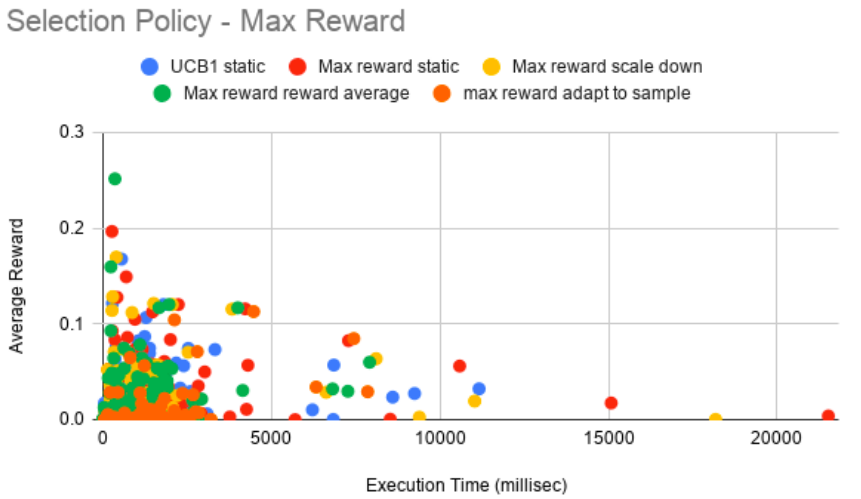

Figure 17: Average Reward w.r.t Total Execution Time for Max Reward as default selection policy and all the 4 exploration weight policies

Figure 17 shows that selection of the Max Reward as default selection strategy and Reward average as default exploration weight policy over the base case implemented by Skinner DB. The highest reward of 0.28 is obtained for max reward strategy. It is understood that max reward selection policy with reward average weight exploration policy will help us maximize the rewards in the best case. The query execution times are more clustered for this scenario, compared to Figure .

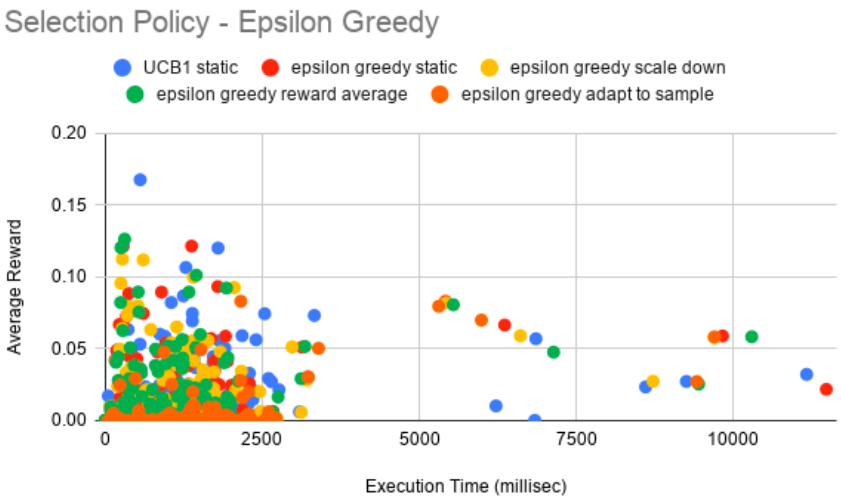

Figure 18: Average Reward w.r.t Total Execution Time for Epsilon Greedy as default selection policy and all the 4 exploration weight policies

We have considered $\epsilon = 0.01$ for our experiments. This ensures that exploitation of the previous strategies from the U.C.T. search tree is favored more over exploitation. The clustering over execution time is significantly reduced in Figure . However, there is no significant gain in either the average reward or the execution time as the balance between exploration and exploitation is hampered.

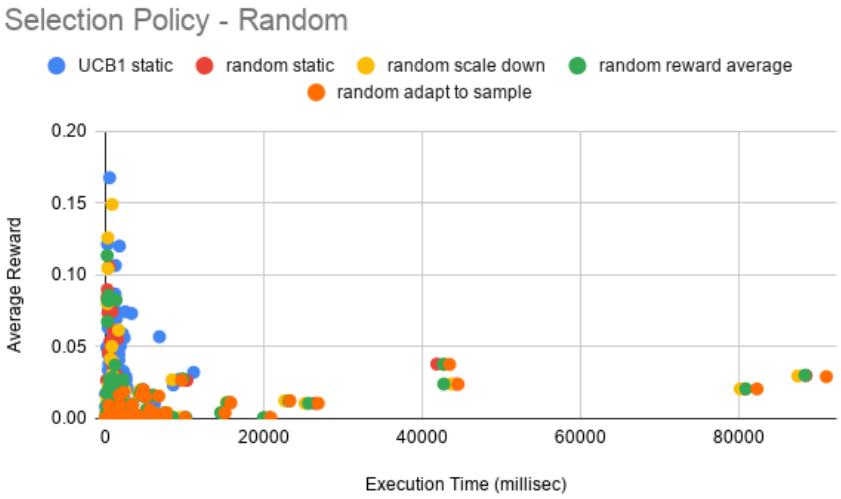

Figure 19: Average Reward w.r.t Total Execution Time for Random as default selection policy and all the 4 exploration weight policies

Our results in Random Strategy uniquely follow the uniform distribution as shown in Figure 7.5 which is not observed in any other result. This results in lesser execution time and higher rewards for najority of the queries. However, no strategy overcomes the maximum reward in the base case of UCB1 default selection policy and Static as the default exploration weight policy.

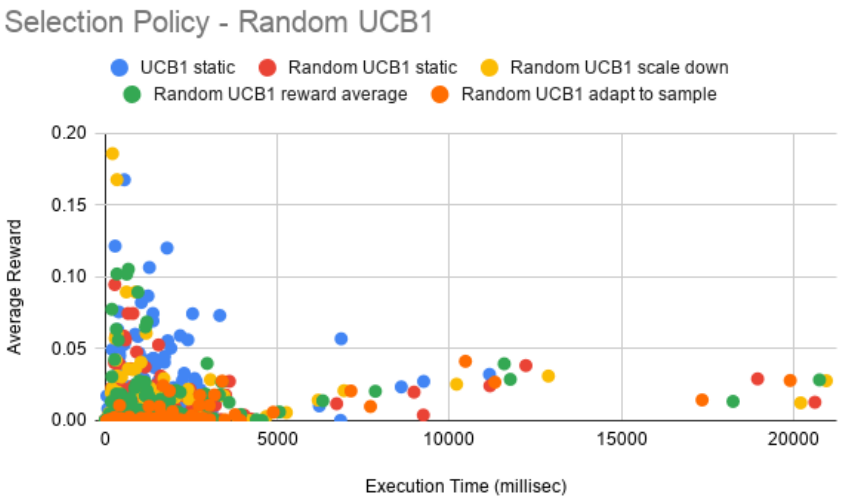

Figure 20: Average Reward w.r.t Total Execution Time for Random UCB1 as default selection policy and all the 4 exploration weight policies

The uniform distribution in Figure is violated in Figure once the UCT formula is applied to select actions. However more queries are executed with higher execution time than noticed before due to application of both the Random and the UCB1 strategies, leading to additional overhead.

7.6 Exploration Weights

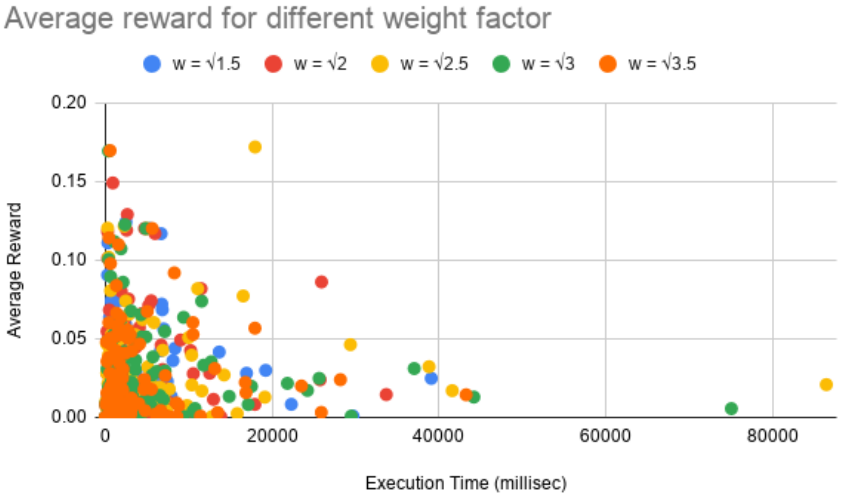

Figure 21: Average Reward w.r.t Total Execution Time for Different Exploration Weights

The above graph shows results for different iterations of the experiment with different queries. The blue, red, yellow, green and orange plot points represent the data for $w = \sqrt{1.5}, w = \sqrt{2}, w = \sqrt{2.5}, w = \sqrt{3}$ and $w = \sqrt{3.5}$ respectively.

We can see from the above graph that the results start improving as w approaches $\sqrt{2}$ and start degrading after a certain point.

From the above experiment we concluded that the value $w = \sqrt{2}$ and $w = \sqrt{2.5}$ works best for this experiment to obtain a bound on the expected regret.

7.7 Generalization of original Skinner DB implementation

The fig 22 shows the result of running the original SkinnerDB implementation on the permutated dataset and on the new query workloads. As we can observe from the graph, the clustering of rewards on permuted dataset is similar, but there are several outliers. However, changing the workload does not seem to create much difference.

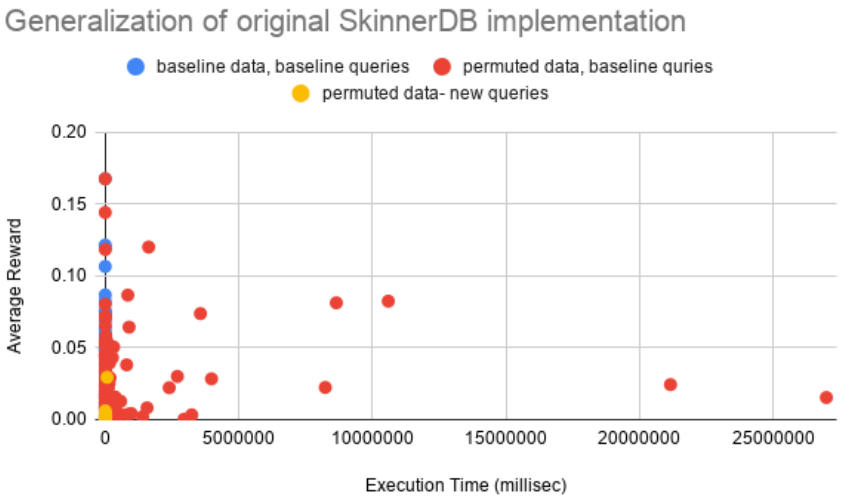

Figure 22: Generalization of original SkinnerDB implementation

7.8 Generalization of the best policy combination

The 23 shows the result of running SkinnerDB on the combination of the policies which we identified as the best: maxreward and reward average. We can notice a slight improvement in the performance of our implementation compared with the baseline performance on the original data.

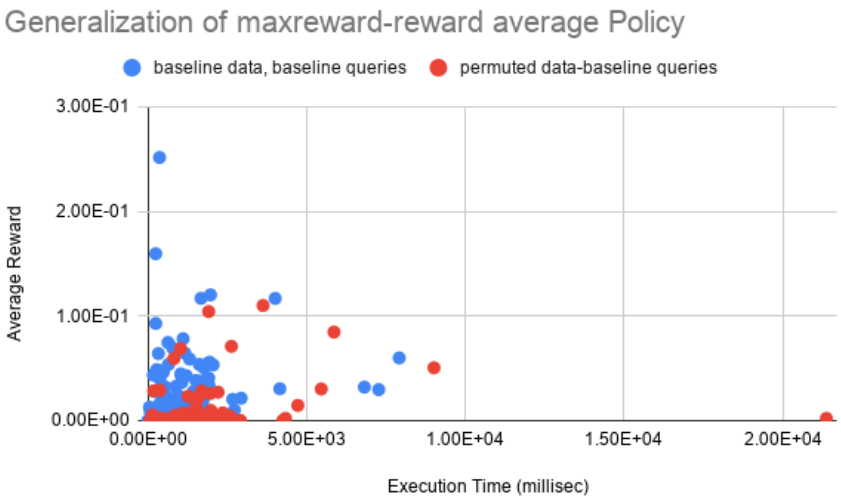

Figure 23: Generalization of the best policy combination

7.9 Generalization on Max Reward, Adapt to Sample Policy Combination

The fig 24 shows the result of running the system on the combination of maxreward with sample-to-adapt policy. As we can see, we the performance degrades and we incur lots of outliers, which means that combining maxreward with sample-to-adapt is not a good strategy.

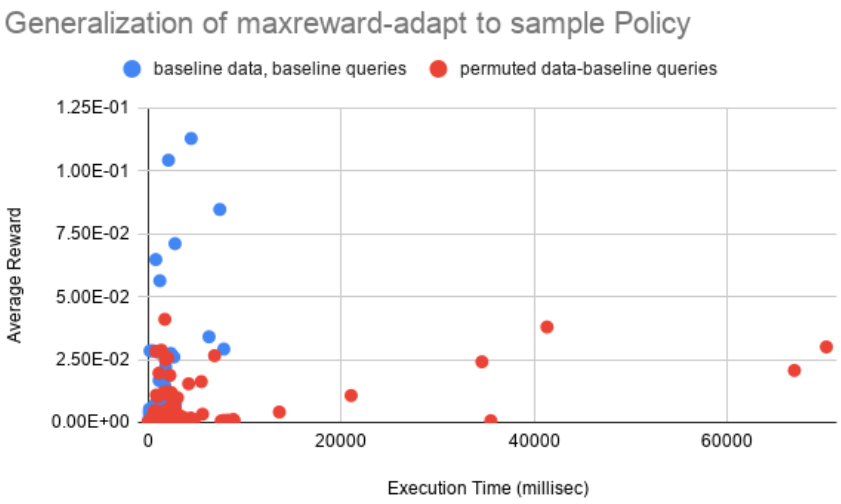

Figure 24: Generalization of maxreward-adapt to sample policy

7.10 Results on using FIFA17 Dataset on SkinnerDB implementation

The execution time is much less than Join-Order Benchmark (JOB) dataset (max of 161 ms. vs. max of 1857 ms. respectively). The scalability of the dataset cannot factored in finding the obtained joined order, as we have observed in our literature survey. Thus is expected as the FIFA17 Dataset is much smaller than the JOB dataset and it is a completely new dataset with a different schema.

8 Conclusion and Future Work

We have completed our ideas of dynamic and elastic time slices as we had proposed in the Project Proposal. We also have found the combination of the best selection and exploration weight policy. Our strategies improve the execution time for the selected data and workloads. They can also be generalized for permuted dataset and different workload.

The current formulation of reward is based on the progress of the join order as well as on the processed tuples and the fixed number of time slice in each step. However, if a query is executed faster, it does not mean that more joins are performed, which may lead to precise results. The data access may not be efficient due to poor design of the schema. Indexing may impact the execution times of the queries. All such factors affecting the query execution have to be investigated for incorporation in the reward function.

References

[1] Fifa17 database. https://www.kaggle.com/artimous/complete-fifa-2017-player-dataset-global, 2017. Accessed: 2017-02-17.

[2] Gray, J. E. Optimizing table join ordering using graph theory prior to query optimization, May 26 1998. US Patent 5,758,335.

[3] Kadkhodaei, H., and Mahmoudi, F. A combination method for join ordering problem in relational databases using genetic algorithm and ant colony. In *2011 IEEE International Conference on Granular Computing* (2011), IEEE, pp. 312–317.

[4] Leis, V., Radke, B., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., and Neumann, T. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* (09 2017).

[5] Marcus, R., and Papaemmanouil, O. Deep reinforcement learning for join order enumeration. *CoRR abs/1803.00055* (2018).

[6] McCaffrey, J. Epsilon greedy algorithm in reinforcement learning. https://jamesmccaffrey.wordpress.com/2017/11/30/the-epsilon-greedy-algorithm/, 2019. Accessed: 2017-11-30.

[7] Salloum, Z. Monte carlo tree search in reinforcement learning. https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f, 2019. Accessed: 2019-02-17.

[8] Trummer, I. Skinnerdb presentation in sigmod'19. In *2019 SIGMOD Proceedings* (2019), SIGMOD.

[9] Trummer, I., Wang, J., Maram, D., Moseley, S., Jo, S., and Antonakakis, J. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *CoRR abs/1901.05152* (2019).