Oct 31, 2019

Reinforcement Learning Based Query Evaluation Using Dynamic Time Slices

Saptarashmi Bandyopadhyay (szb754@psu.edu)
Bodhisatwa Chatterjee (bxc583@psu.edu)
Kirti Jagtap (ktj35@psu.edu)

Presentation Outline

- Introduction
- Project Structure
- Installation and Setup
- Results
- Analysis
- Ongoing Work

Introduction to Query Optimization

- **Objective:**
  - Efficient Join Ordering by estimating the cost for query plans
- **Challenges:**
  - Dependent on statistics from previous results and queries for future queries
  - Generation of poor query execution plan
- **Opportunities:**
  - Improve the learning framework with Reinforcement Learning, using no data statistics.

SkinnerDB: State of Art (2019)

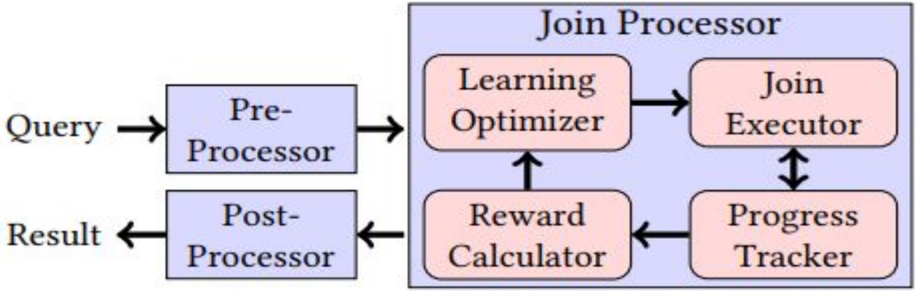


Fig 1: Architecture of SkinnerDB[1]

- **Major Takeaways:**
  - Data statistics and cost of cardinality models are not used.
  - The best join orders selected for a query are executed in equal time slices.
  - Result tuples obtained from each time slice have been merged to obtain the final result.

Issues in RL based Query Optimization

- Division of a query into many time slices may not be possible
- Identification of the initial join strategy
- Identification of the improved join strategy at intermediate steps
- Integration of the results obtained from the previous strategy with that obtained from the current strategy

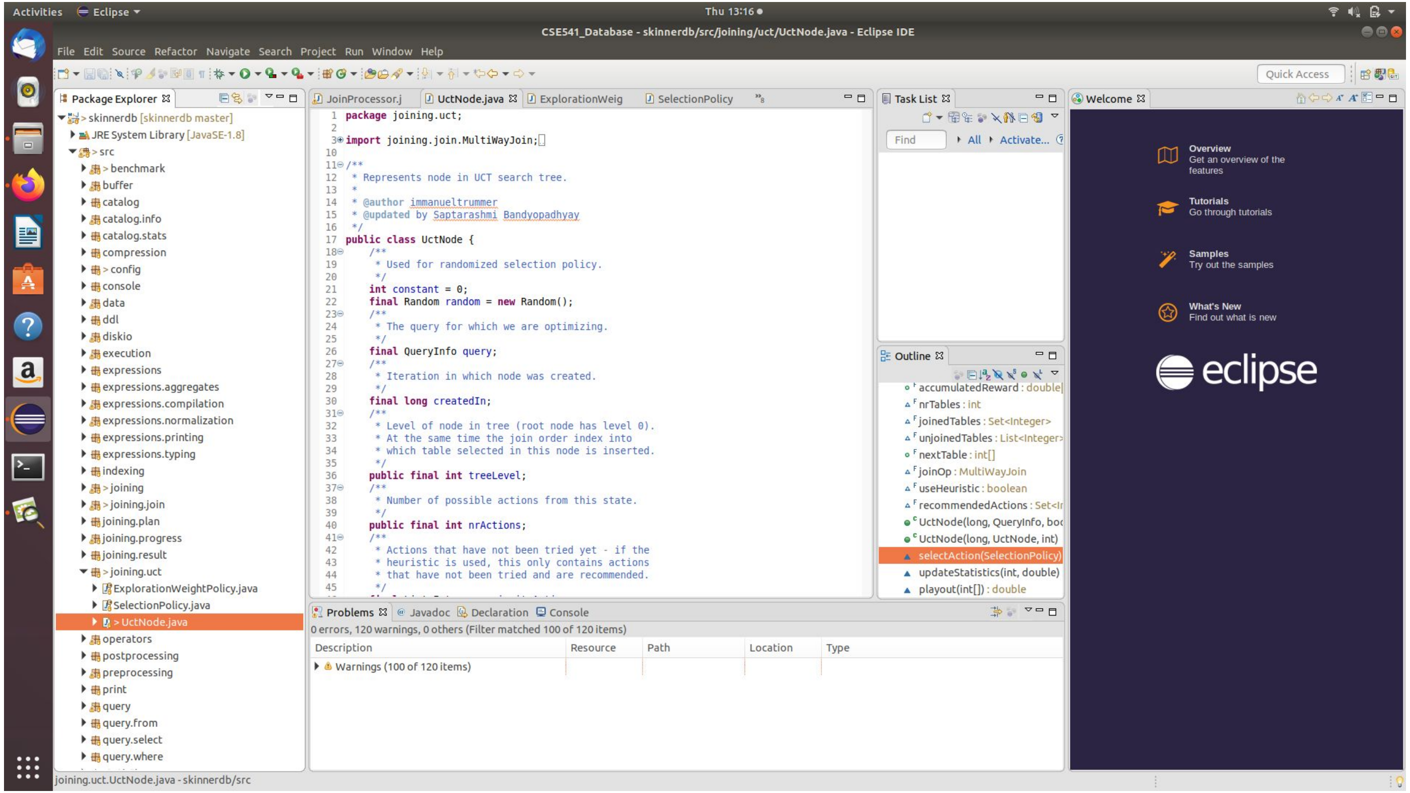
Installation and Setup

- The github project of SkinnerDB from the CornellDB group was imported to **Eclipse I.D.E.** (v4.11.0).

- **JAVA SE-1.8** had to be used for installation as there were problems with higher versions.

- The necessary libraries have been extracted to the generated **Skinner-<Experiment_no>.jar**.

Code Repository in Eclipse IDE



Database Used - IMDB

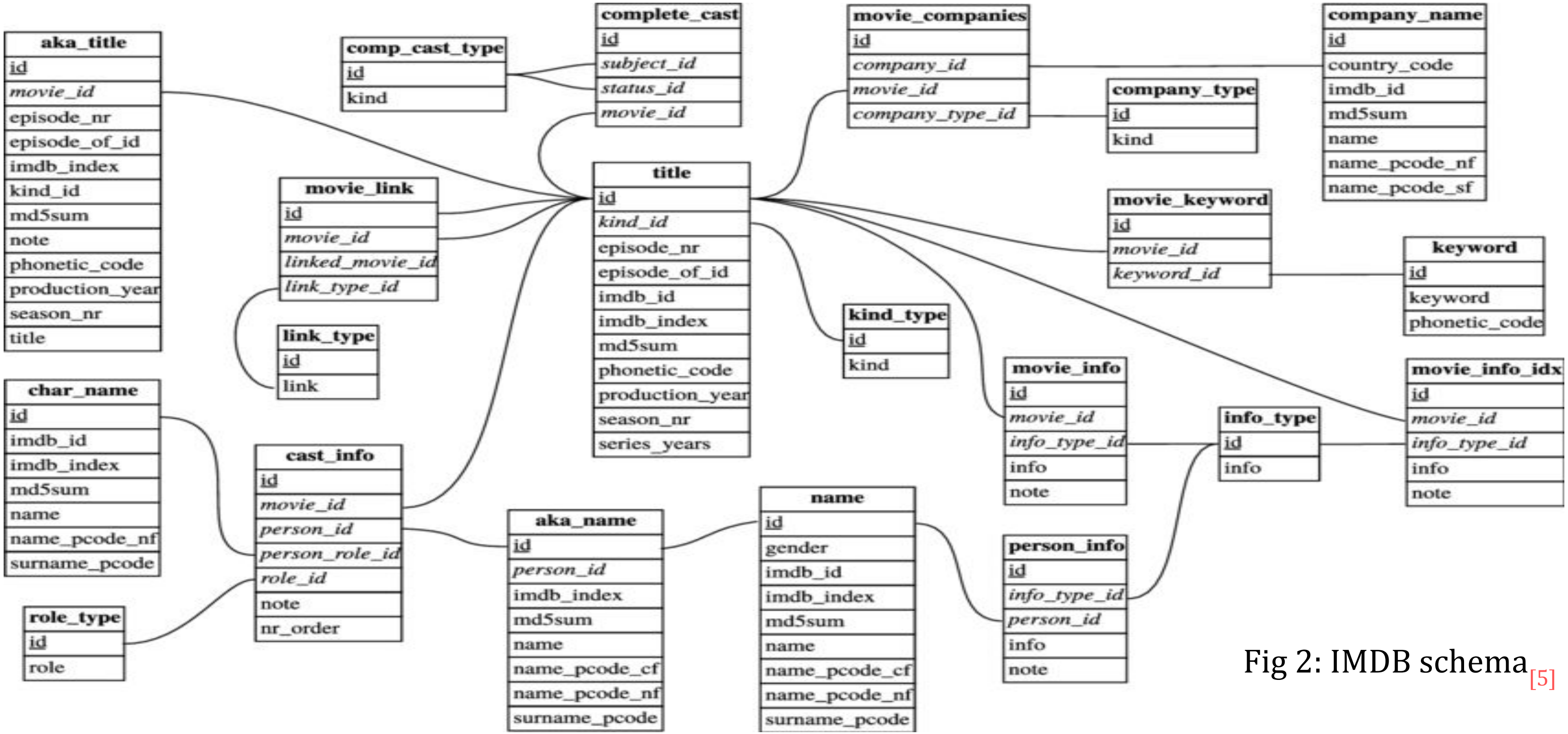


Fig 2: IMDB schema[5]

Input and Output

Input:

1) Database: IMDB

2) Queries: 113 queries

Output:

1) The UCT(**U**pper **C**onfidence bounds applied to **T**rees) search tree used during join order learning

2) Join-order Benchmark

Important Packages

The main packages in the repository:

1) *joining.join*, *joining.plan*, *joining.progress*, *joining.result*, *joining.uct*: For join ordering

2) *execution*, *console*: for high-level execution process.

3) *statistics*, *visualization*, *benchmark*: for join-order benchmark and UCT search tree visualization

*There are several other packages like *buffer*, *catalog*, *data*, *diskio*, *expressions*, *indexing*, *post-processing*, *preprocessing*, *query*, etc

Join Benchmark Output

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	6.69269984373089E-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	0.005734045857020770	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	0.05244434278162570	0.4392351627485530	2.000391528068170
03a.sql	877	842	0	17079	49103	0	111107977	19067	20	8	206	99	0.00736522252233660	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	0.2183120007121290	1.000849055594270
04a.sql	813	756	0	20254	54985	0	135114789	20908	34	13	740	110	0.00985789256615948	0.1081657889673920	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	1.025574064678990
05b.sql	978	961	0	501	1002	0	1	501	5	3	0	3	0.02945948606429310	0.0881523272214387	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	0.01836721654565522	0.2714223893079060	1.497724662079350
06b.sql	465	437	0	1817	4035	0	361091	1976	11	7	12	9	0.04018437960359370	0.2834229632629030	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	0.003115888057223240	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	0.1620000000000004	1.000337672265280
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	5002	11274	0	147348	6885	26	23	16	23	0.01290363709569710	0.1510216818069920	1.360371389797060
07c.sql	28921	3597	1	3706441	9899166	0	561823248	4380205	304	208	68185	19799	0.00570716233219530	0.2777777666531250	1.004445490236430
08a.sql	4874	4721	0	28508	59609	0	62018	29741	47	37	62	120	0.02532881780653600	0.3487316126611690	1.641293014461550
08b.sql	4641	4552	0	1714	3695	0	12493	1741	10	8	6	8	0.0418525343964672	0.3141736935388580	1.660070868098250
08c.sql	110806	136	18	21987201	55152726	0	5778175223	18412936	208	143	2487611	110306	0.02259867964234460	0.3728896042160520	1.677989893004010

Important Concepts

- **Progress** (P): percentage of join order completed (portion of input data processed on a certain state)

- **Reward** (R): Both the short term and long-term rewards are calculated as follows:

$$\mathbf{R = 0.5*P + 0.5*N_r /B},$$

where N_r is the number of processed tuples

B is the number of fixed time slices

Important Concepts (contd.)

- **Regret**: Regret is the difference between actual and optimal execution time.[1]
- **Total Execution Time**: (query execution + preprocessing time + post processing time)

Selection Policy

5 different selection policies are supported by SkinnerDB:

- **UCB1**: uses the UCT formula for calculating reward
- **MAX REWARD**: selects actions having maximal reward
- **EPSILON GREEDY**: best action is selected after exploration for 1- ε% of time and random action for ε % of time
- **RANDOM**: actions are selected as an uniform random distribution
- **RANDOM UCB1**: Initial join order is selected randomly as the root, then UCB1 strategy is adopted

Exploration Weight Policy

4 different exploration weight policies are supported by SkinnerDB:

- **STATIC**: exploration weight (w_e) is not updated
- **REWARD AVERAGE**: (w_e) is updated based on the average reward
- **SCALE DOWN**: (w_e) is scaled down over the number of iterations
- **ADAPT TO SAMPLE**: (w_e) is selected based on the initial reward sample

Control flow for Time Slices

- For every iteration during R.L., the best join order is executed for a fixed number of time slices in *OldJoin.java* and *MultiWayJoin.java*
  - They execute joins in small time slices, using for each slice a newly specified join order.
  - The result tuples are collected from different time slices and merged to give the final result.
- *MultiWayJoin.java* is called by *UctNode.java* that traverses each samples each node from the UCT search tree and returns the reward.

Dynamic Time Slices

- Instead of a fixed time slice of 500 steps for executing all join orders, we have increased the time slice by 5
- Implemented in UctNode.java for every iteration of the learning process, that selects the best possible Join Order.
- The increment by 5 happens in OldWayJoin.java which actually executes the partial join order.
- The fixed number of time slices have also been extended to 600 (without any dynamic time slices for now) to analyze the impact.

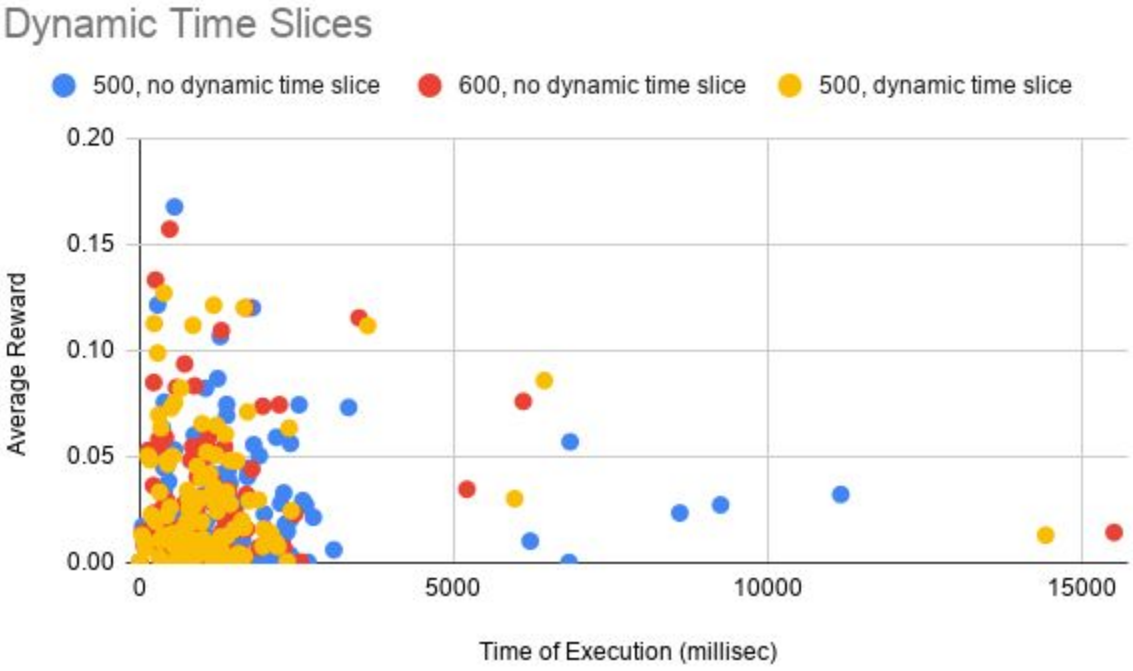
Experiments

- 27 experiments have been run
- Default Selection and Exploration policies
  - All combinations of 5 selection policies with 4 exploration weight policies
- Dynamic time slice of 5
- Changing fixed time slices to 600
- Changing exploration weights

Selected Join Orders for 1 query for 1 experiment

```
01a.sql
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

Selected join order: [4, 3, 2, 0, 1]
Obtained reward:    3.6589214517560594E-5
Table offsets: [0, 0, 0, 0, 184]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]

Selected join order: [4, 3, 2, 1, 0]
Obtained reward:    4.034251820504237E-5
Table offsets: [0, 0, 0, 0, 388]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]

Selected join order: [1, 3, 2, 4, 0]
Obtained reward:    0.5549384697988416
Table offsets: [0, 0, 0, 1379864, 388]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]

Selected join order: [0, 2, 3, 1, 4]
Obtained reward:    0.008324967980892382
Table offsets: [0, 0, 480, 1379864, 388]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]

Selected join order: [2, 3, 1, 4, 0]
Obtained reward:    0.004400014080045055
Table offsets: [0, 0, 729, 1379864, 388]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]

Selected join order: [3, 4, 1, 2, 0]
Obtained reward:    0.23098245614035087
Table offsets: [0, 0, 729, 1379929, 388]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]

Selected join order: [1, 3, 2, 0, 4]
Obtained reward:    0.5492882651693826
Table offsets: [0, 0, 729, 1379929, 388]
Table cardinalities:    [1, 1, 28889, 1380035, 2528312]
```

Dynamic Time Slices



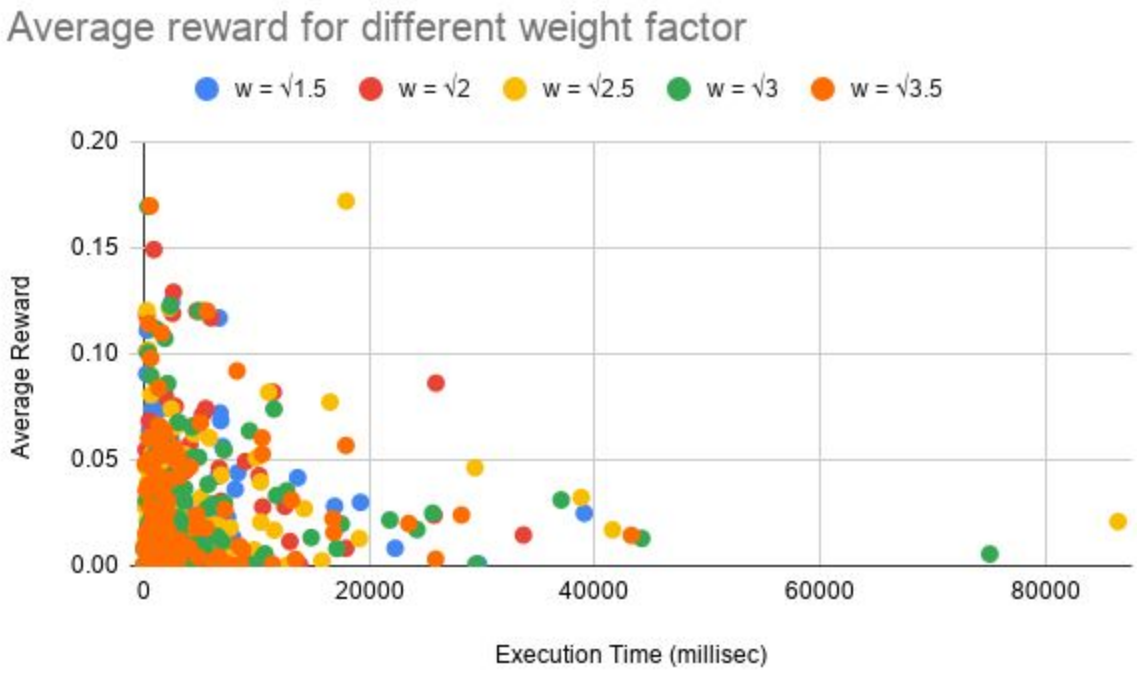
Analysis for Dynamic Time Slices

- Dynamic time slice decreases total execution time since static time slice wastes time

- Average reward increases with dynamic time slices

- Changing the fixed number of time slices does not have much impact on the result

Exploration Weights

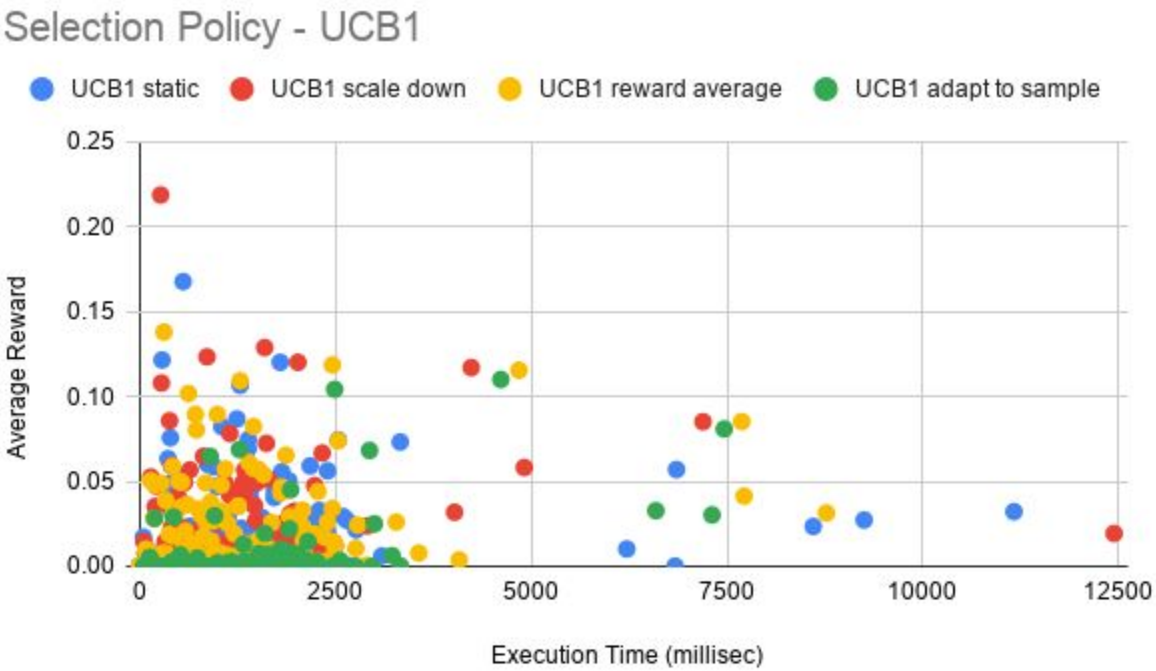


Analysis for Exploration Weights

- Execution time increases with increase in average weight factor for w=sqrt(2)

- With less weight factor there is increase in execution time.

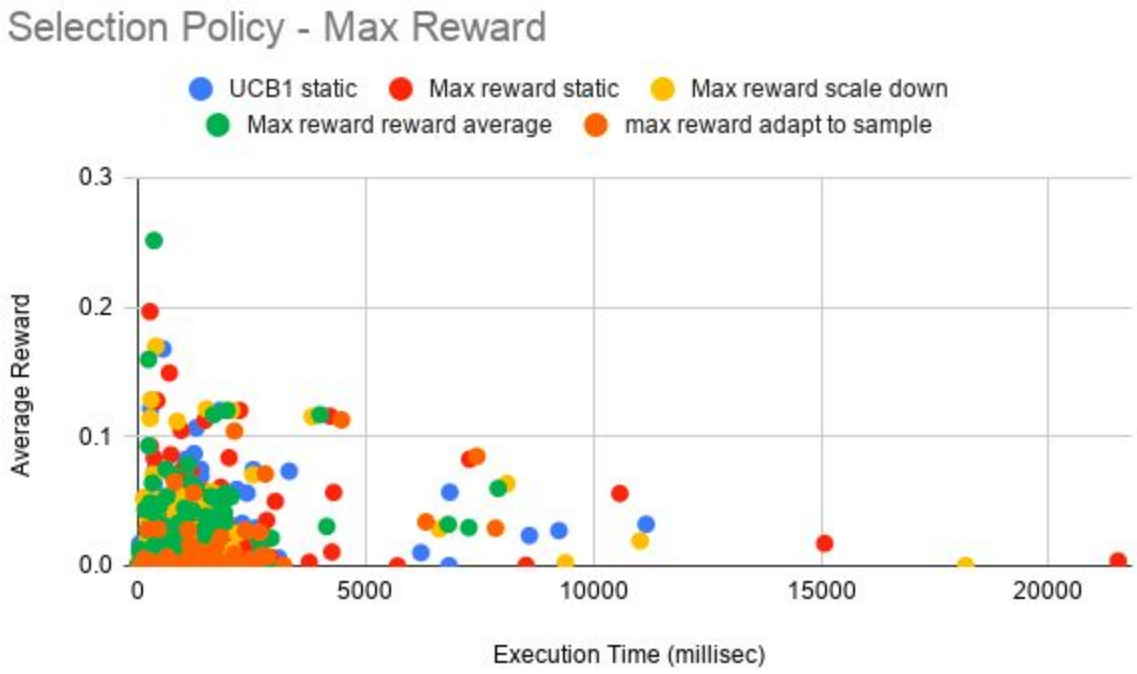
UCB1 as Default Selection Policy



Analysis for UCB1 Default Selection Policy

- Base case of UCB1 static average gives the best results
- Most of the queries are executed within 2500 time units with higher average rewards
- This is due to the workload selection of queries

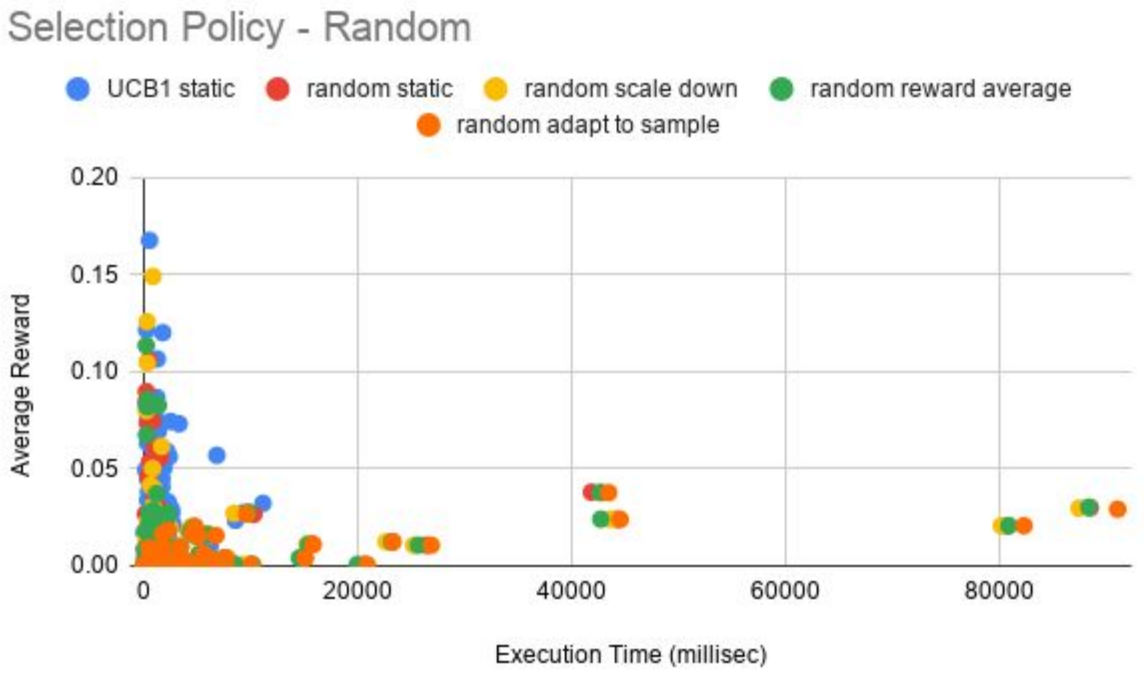
Max Reward as Default Selection Policy



Analysis for Max Reward Selection Policy

- 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.

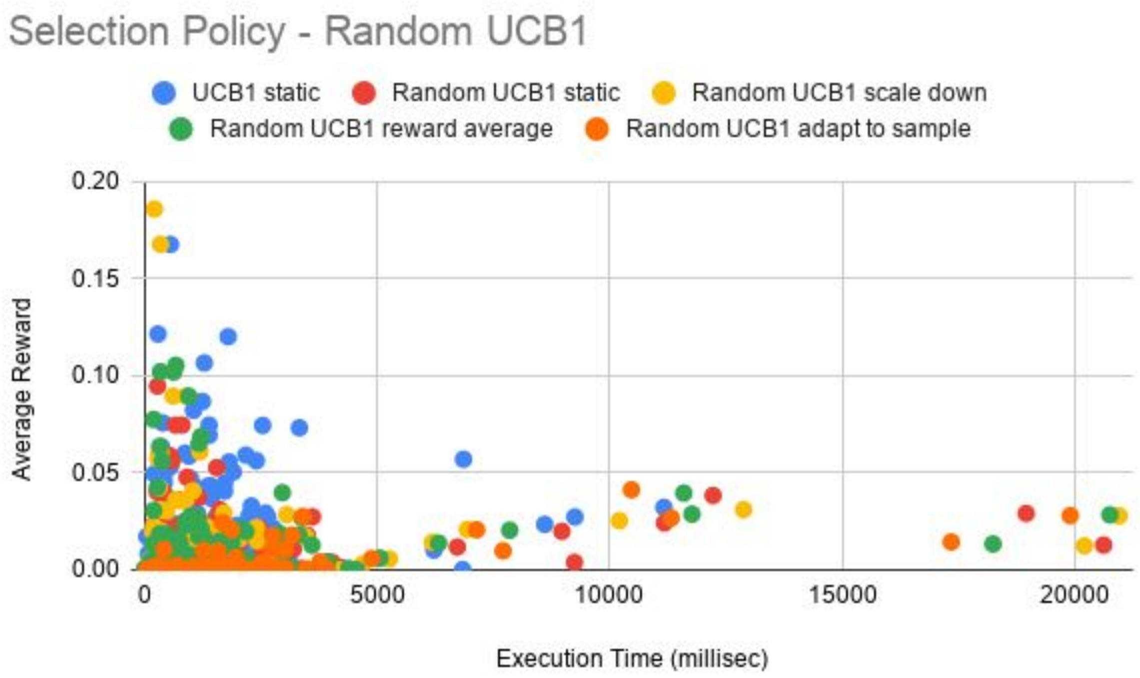
Random as Default Selection Policy



Analysis for Random Default Selection Policy

- Interestingly average reward is distributed more uniformly for Random selection strategy
- Reward average weight exploration strategy performs the best.
- Overall the maximum among the average rewards is close enough to UCB1.

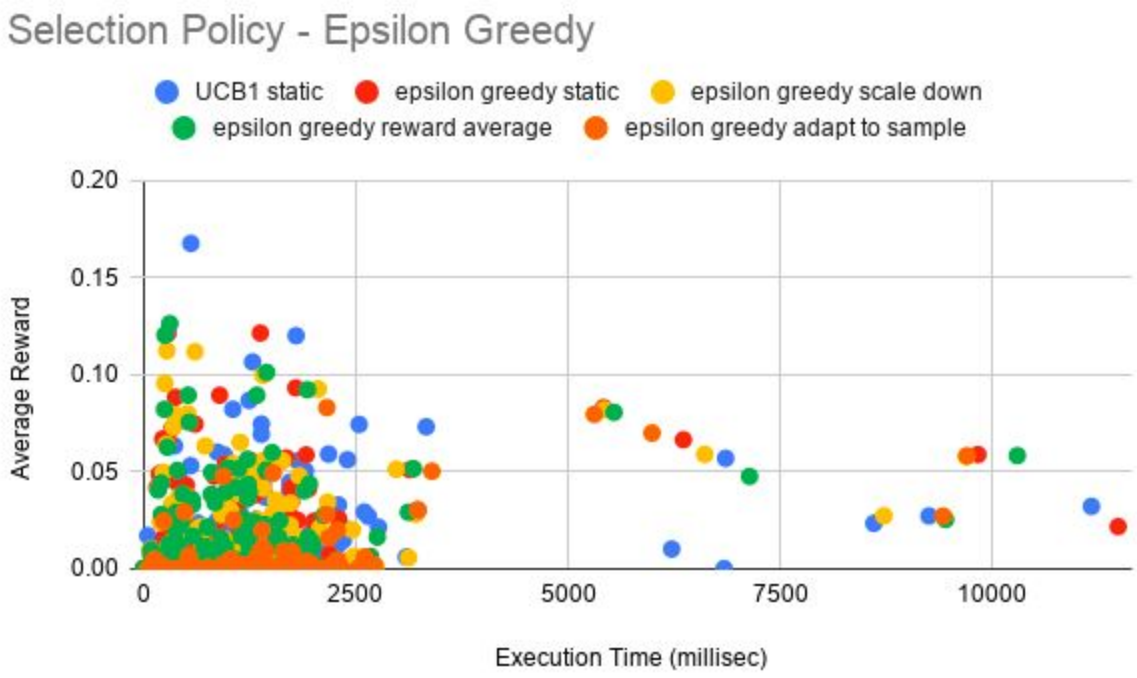
Random_UCB1 as Default Selection Policy



Analysis for RANDOM_UCB1 Default Selection Policy

- Random reward average gives best results over the base case.
- The cluster of query execution within 2500 time units are observed here again with higher average rewards
- This is due to the selection of the workload of queries

Epsilon Greedy as Default Selection Policy



Analysis for Epsilon Greedy Default Selection Policy

- For selection for a particular action, in Epsilon-Greedy Strategy, the best action is selected for (1-Epsilon)% of time and a random action is selected for (Epsilon)% for time.

- For our experiments, this strategy work excellently as in 90% of cases the actions selected leads to the optimal join plan.

- Epsilon can be adjusted to decrease the exploration for selecting the action.

Ongoing Work

- Clustering of indexes according for distinguishing features for successful joins. Might result in faster joins if a distinctive features of a plan are identified.

- Looking up encryption schemes for columns-based database. Current system has no support for checking whether a particular query is allowed to access all columns, so Fine-Grain Access Control is investigated with respect to SkinnerDB.

References

1. *Trummer, Immanuel, et al. "SkinnerDB: regret-bounded query evaluation via reinforcement learning." Proceedings of the 2019 International Conference on Management of Data. ACM, 2019.* [*https://github.com/cornelldbgroup/skinnerdb*](https://github.com/cornelldbgroup/skinnerdb)
2. *Krishnan, Sanjay, et al. "Learning to optimize join queries with deep reinforcement learning." arXiv preprint arXiv:1808.03196 (2018).*
3. *Ortiz, Jennifer, et al. "Learning state representations for query optimization with deep reinforcement learning." arXiv preprint arXiv:1803.08604 (2018).*
4. *Marcus, Ryan, and Olga Papaemmanouil. "Towards a hands-free query optimizer through deep learning." arXiv preprint arXiv:1809.10212 (2018).*
5. *Leis, Viktor, et al. "Query optimization through the looking glass, and what we found running the join order benchmark." The VLDB Journal—The International Journal on Very Large Data Bases 27.5 (2018): 643-668.*

THANK YOU