

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

- Background and Motivation
- Project Structure
- Installation and Setup
- Generalization
- Results
- Visualizations
- Analysis
- Future Work

Query Optimization and Challenges

- **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:**

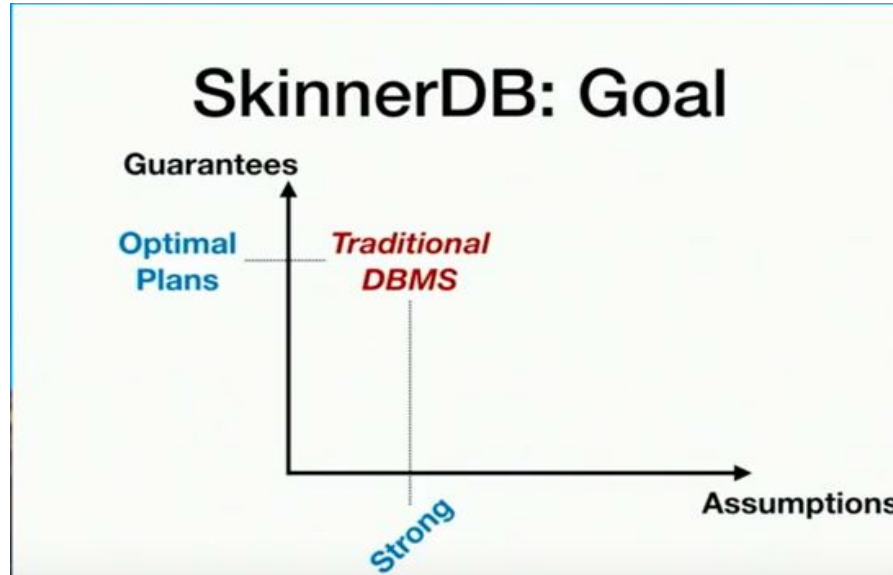
- Improve the learning framework with Reinforcement Learning, using no data statistics.

Background & Motivation

- Traditional DBMS makes strong assumptions about queries and data while making query plans

Background & Motivation

- Traditional DBMS makes strong assumptions about queries and data while making query plans



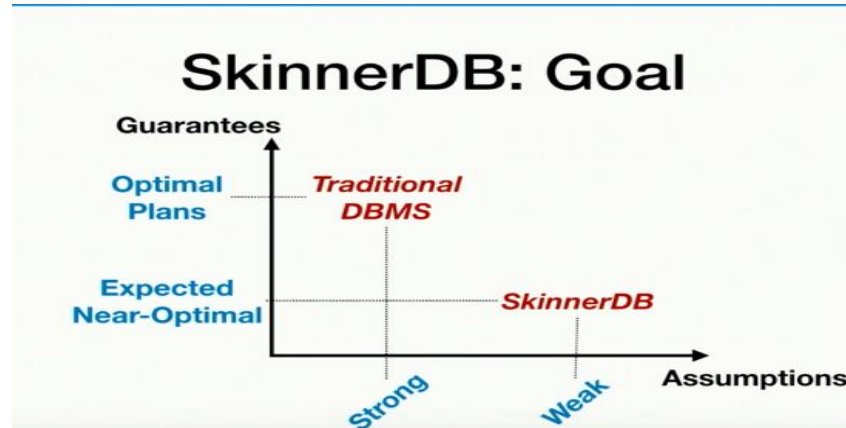
[6]

Background & Motivation

- Traditional DBMS makes strong assumptions about queries and data while making query plans
- SkinnerDB does not make any assumptions on the data and query, but instead tries to give probabilistic guarantees on expected vs optimal execution time for a given query

Background & Motivation

- Traditional DBMS makes strong assumptions about queries and data while making query plans
- SkinnerDB does not make any assumptions on the data and query, but instead tries to give probabilistic guarantees on expected vs optimal execution time for a given query



[6]

Background & Motivation

- Traditional DBMS makes strong assumptions about queries and data while making query plans
- SkinnerDB does not make any assumptions on the data and query, but instead tries to give probabilistic guarantees on expected vs optimal execution time for a given query
- Focus on join-ordering queries as they tend to be one of the most performance impactors.

Background & Motivation

- Traditional DBMS makes strong assumptions about queries and data while making query plans
- SkinnerDB does not make any assumptions on the data and query, but instead tries to give probabilistic guarantees on expected vs optimal execution time for a given query
- Focus on join-ordering queries as they tend to be one of the most performance impactors
- Use Reinforcement Learning to 'learn join-orders'

SkinnerDB and ML for query optimization

- Don't learn from past queries to learn future queries; but learn during the execution of current query to optimize remaining execution time (avoid generalization)

SkinnerDB and ML for query optimization

- Don't learn from past queries to learn future queries; but learn during the execution of current query to optimize remaining execution time (avoid generalization)
- Notion of 'intra-query' learning instead of 'inter-query' learning

SkinnerDB and ML for query optimization

- Don't learn from past queries to learn future queries; but learn during the execution of current query to optimize remaining execution time (avoid generalization)
- Notion of 'intra-query' learning instead of 'inter-query' learning
- Intra-Query Learning pays off for difficult queries

How is this achieved?

- Divide query execution into 'micro-episodes' (~10ms)

How is this achieved?

- Divide query execution into 'micro-episodes' (~10ms)
- In each micro-episode, select a join-order (random initially)

How is this achieved?

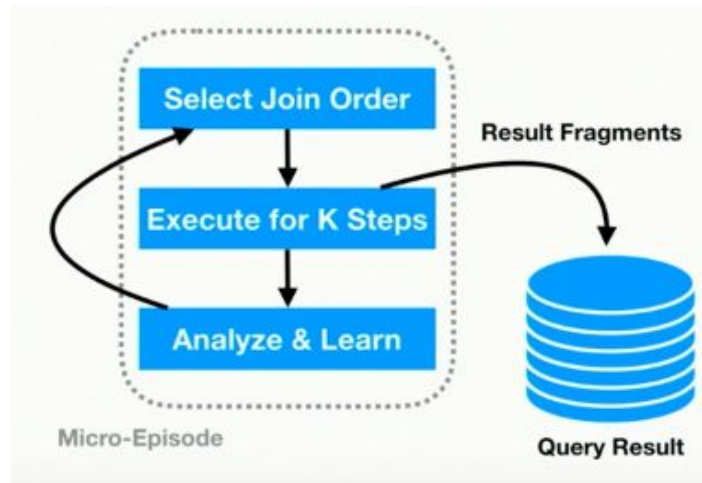
- Divide query execution into 'micro-episodes' (~10ms)
- In each micro-episode, select a join-order (random initially)
- Execute the join-order for fixed steps (result fragment)

How is this achieved?

- Divide query execution into 'micro-episodes' ($\sim 10\text{ms}$).
- In each micro-episode, select a join-order (random initially).
- Execute the join-order for fixed steps (result fragment)
- Measure evaluation progress and learn better join-order for the following episodes

How is this achieved?

- Divide query execution into 'micro-episodes' (~10ms).
- In each micro-episode, select a join-order (random initially).
- Execute the join-order for fixed steps (result fragment)
- Measure evaluation progress and learn better join-order for the following episodes



SkinnerDB: State of Art (2019)

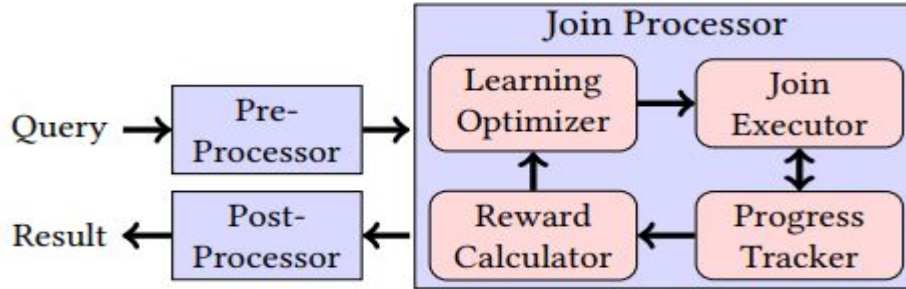


Fig 1: Architecture of SkinnerDB_[1]

- **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.

What's the problem with this?

- 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

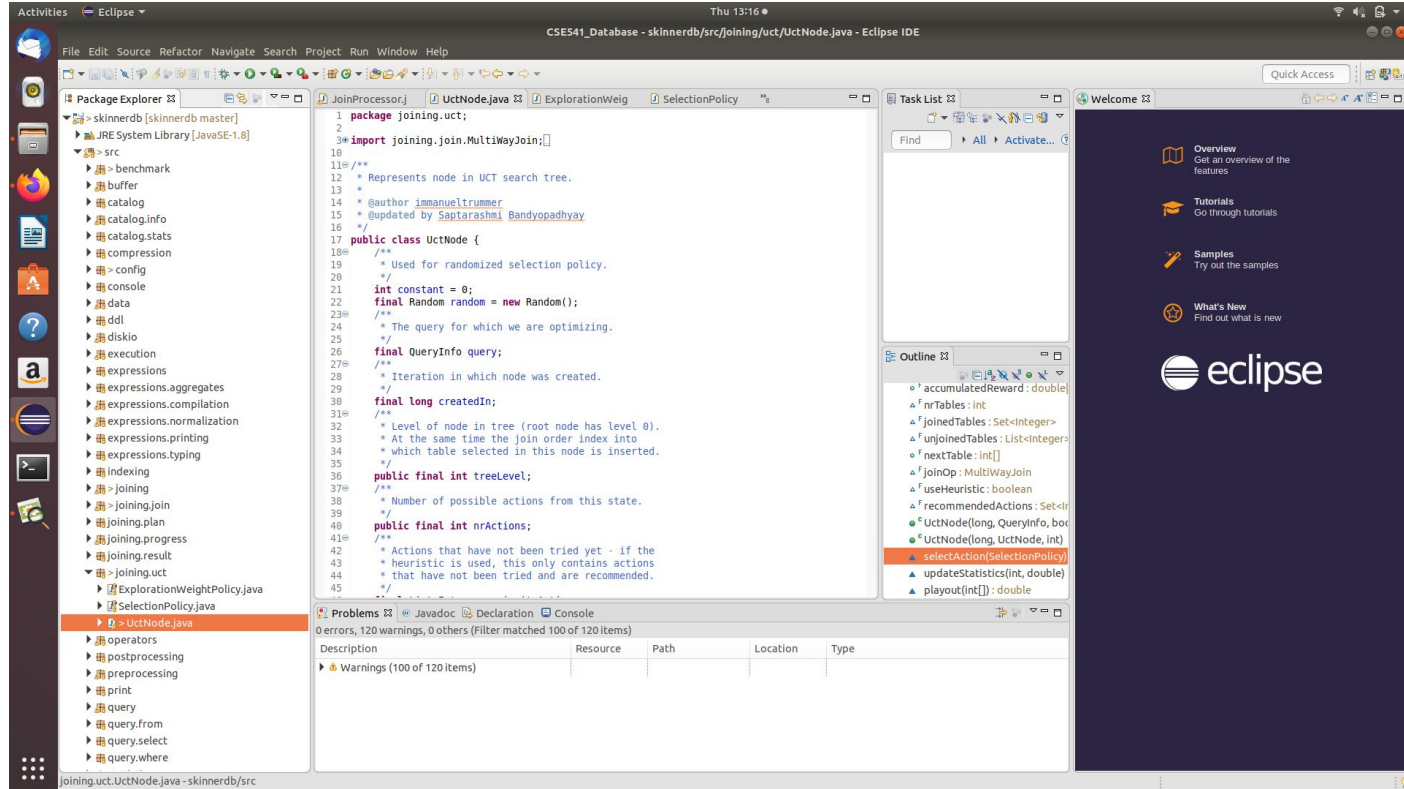
Our Contributions

- Lesser Execution Time and Higher Rewards with Different Schemes for Dynamic and Elastic Time Slices
- Identification of the Best Selection Policy and Exploration Weight Policy by empirical investigation of 5 selection policies and 4 exploration weight policies
- Generalization of our Research Contributions

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



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-Order Benchmark [Leis 2015]

- Subset of IMDB Database
- Designed specially for evaluating query optimizers on join-orders
- 3.36 GB, 21 CSV files

Join-Order Benchmark

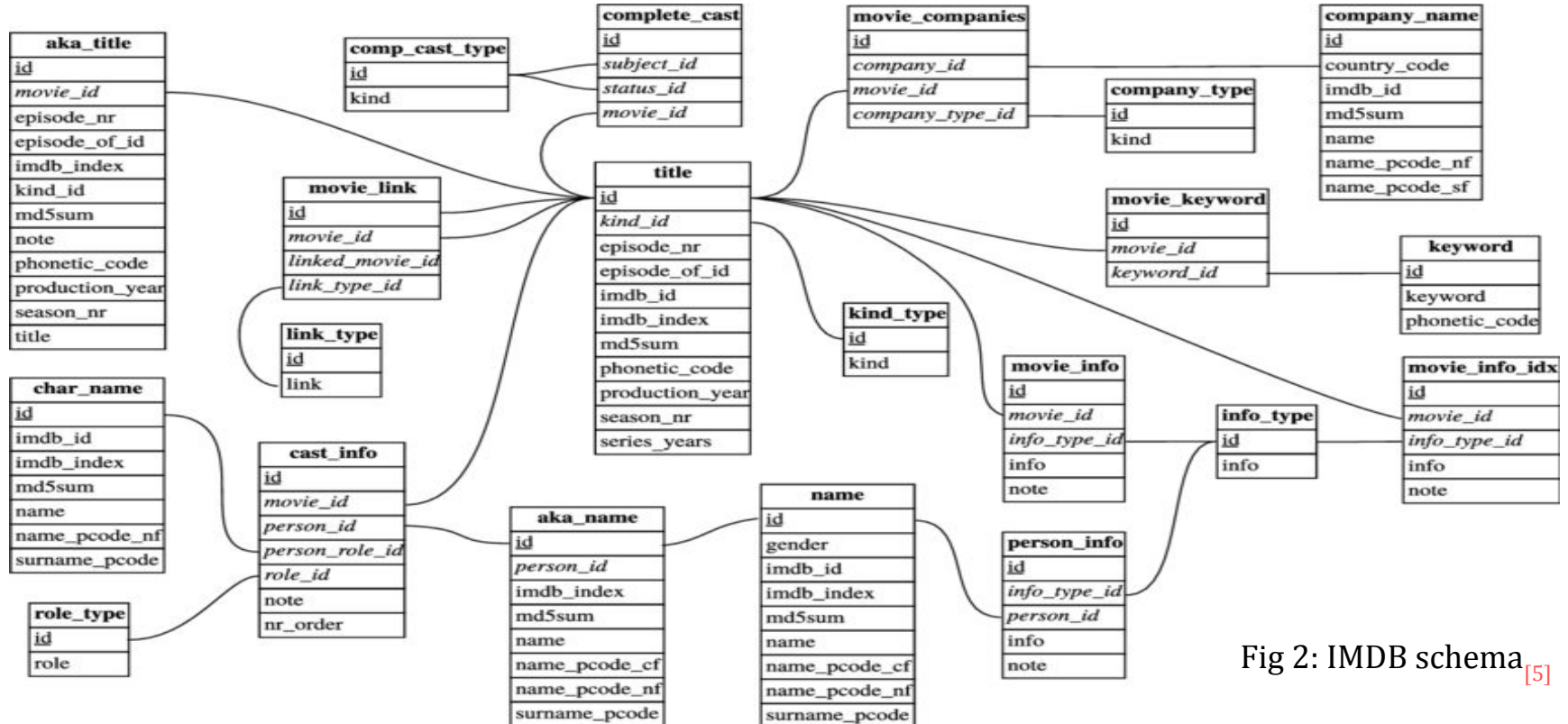


Fig 2: IMDB schema ^[5]

Join Benchmark Output

Query	MIIs	PreMIIs	PostMIIs	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	6.69269084373089E-05	1.27769581431916E-04	1.0005325023401900
01c.sql	547	520	0	1461	3041	0	645	1516	9	7	3	7	0.07333685140912460	0.5029891287489510	2.0202916138364100
01d.sql	1977	1946	1	1198	2550	0	561	1124	8	6	4	6	0.07460383344751540	0.44733247657954900	1.8950764316296400
02a.sql	216	38	0	210877	520674	0	1458342059	183259	46	16	7834	1042	0.010648301578590600	0.42390422566528200	2.001003802888050
02b.sql	348	41	0	293624	754031	0	1458102399	330713	71	25	5228	1509	0.0057340458570207700	0.4456090918005670	2.0089137816293200
02c.sql	33	24	0	650	1519	0	1339447	624	6	4	0	4	1.74563921456988E-04	6.66234888691912E-04	1.0013954632415800
02d.sql	1128	52	0	305341	683551	0	1458557330	225280	45	15	68316	1368	0.062444342781625700	0.4392351627485530	2.0003915280681700
03a.sql	877	842	0	17079	49103	0	111107977	19067	20	8	206	99	0.007365222522336600	0.1749451889448860	1.002471329451920
03b.sql	361	336	0	5560	11845	0	3376600	5982	20	9	5	24	0.046951031149041100	0.2704678362573100	1.0019353952543300
03c.sql	1466	1399	0	30536	75128	0	111140227	30260	27	9	7250	151	0.05138149545868360	0.21831200071212900	1.0008490555942700
04a.sql	813	756	0	20254	54985	0	135114789	20908	34	13	740	110	0.009857892566159480	0.10816578896739200	1.0018042942544600
04b.sql	279	230	0	3968	8550	0	3408781	4128	21	11	6	18	0.044827740956140400	0.2936619121667720	1.87274388965510
04c.sql	2609	2258	0	39664	91921	0	158723980	32464	33	14	4700	184	0.02799892557650860	0.14742458925837900	1.0013386623103600
05a.sql	824	785	0	1142	2502	0	248	1242	8	6	0	6	0.002873480067502660	0.01327882152647670	1.0255740646789900
05b.sql	978	961	0	501	1002	0	1	501	5	3	0	3	0.029459486064293100	0.08815232722143870	1.1760498887202200
05c.sql	3568	3535	0	4710	10424	0	3144	4792	22	11	669	21	0.10698569388289400	0.539820757578000	2.0181471410556700
06a.sql	2177	2025	0	7528	18024	0	1544257	7950	37	19	6	37	0.01836721654555220	0.27142238930790600	1.4977246620793500
06b.sql	465	437	0	1817	4035	0	361091	1976	11	7	12	9	0.04018437960359370	0.2834229632629030	1.1370417006859600
06c.sql	693	676	0	1748	4095	0	78390	1808	11	8	2	9	0.27342299775104200	1.9054868953300800	
06d.sql	2628	2272	1	23416	61905	0	5149805	23907	38	15	88	124	0.0031158880572232400	0.2754197500973350	1.0001677533060700
06e.sql	5009	4149	1	280774	526894	0	1822177	246800	73	26	6	1054	6.39008490361331E-04	0.2714220646364890	1.6751170384840200
06f.sql	13626	901	9	1625085	3271790	0	463221764	826390	44	14	785477	6544	0.12018606124670600	0.1620000000000004	1.0003376722652800
07a.sql	4243	1929	0	344088	798373	0	28204909	345237	161	110	32	1597	0.001568338151418980	0.20378182540216300	2.675926692024660
07b.sql	2904	2658	0	5002	11274	0	147348	6885	26	23	16	23	0.012903637095697100	0.15102168180699200	1.360371389797080
07c.sql	28921	3597	1	3706441	9899166	0	581823248	4380205	304	208	68185	19799	0.005707162332195300	0.27777776665312500	1.0044454902364300
08a.sql	4874	4721	0	28508	59609	0	62018	29741	47	37	62	120	0.025328817806536000	0.34873161266116900	1.641293014461550
08b.sql	4641	4552	0	1714	3695	0	12493	1741	10	8	6	8	0.0418525343964672	0.31417369353885800	1.660070868098250
08c.sql	110806	136	18	21987201	55152726	0	5778175223	18412936	208	143	2487611	110306	0.022598579642344600	0.37288960421605200	1.6779898930040100

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:

$$R = 0.5 * P + 0.5 * N_r / B,$$

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 $1 - \epsilon$ % 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]

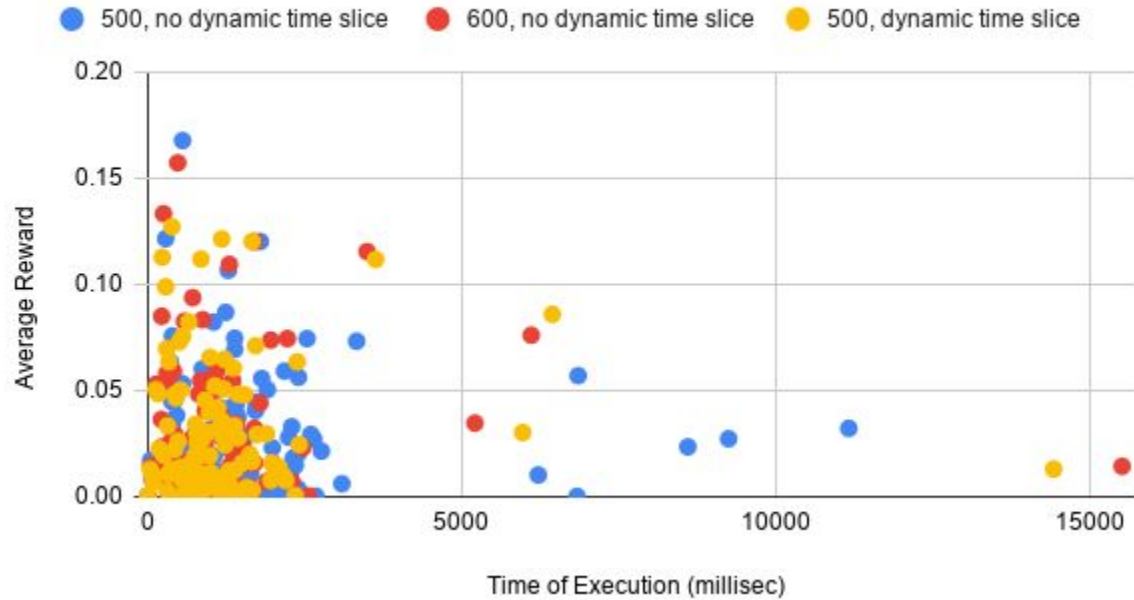
Selected join order: [2, 3, 1, 4, 0]
Obtained reward: 0.0044000140800450555
Table offsets: [0, 0, 729, 1379864, 388]
Table cardinalities: [1, 1, 28889, 1380035, 2528312]

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

Dynamic Time Slices

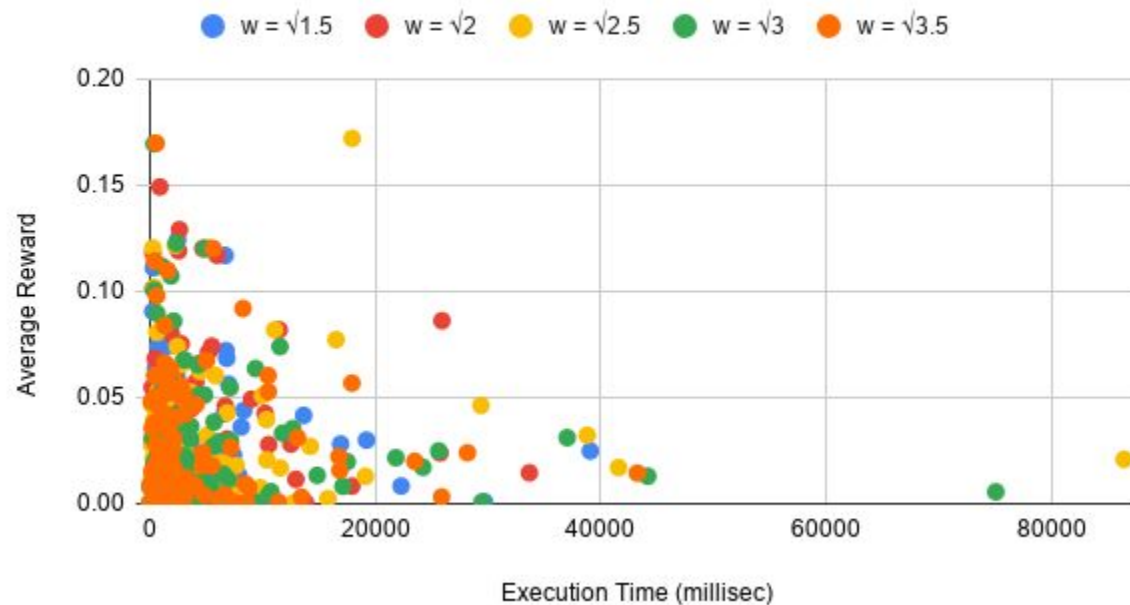


Analysis for Dynamic Time Slices

- 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

Average reward for different weight factor

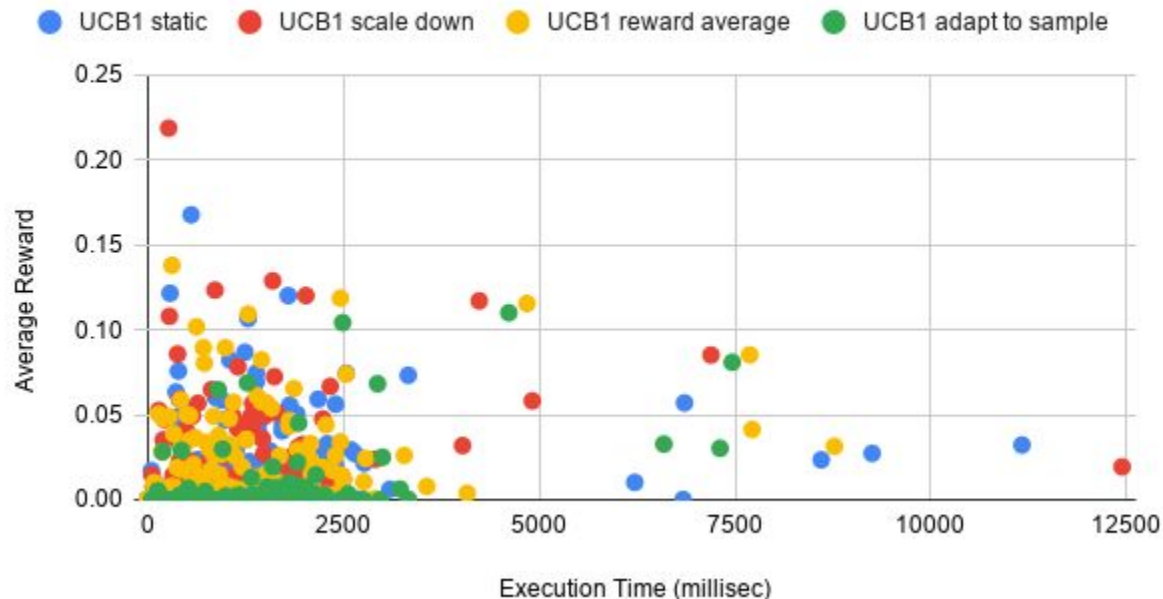


Analysis for Exploration Weights

- 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

Selection Policy - UCB1

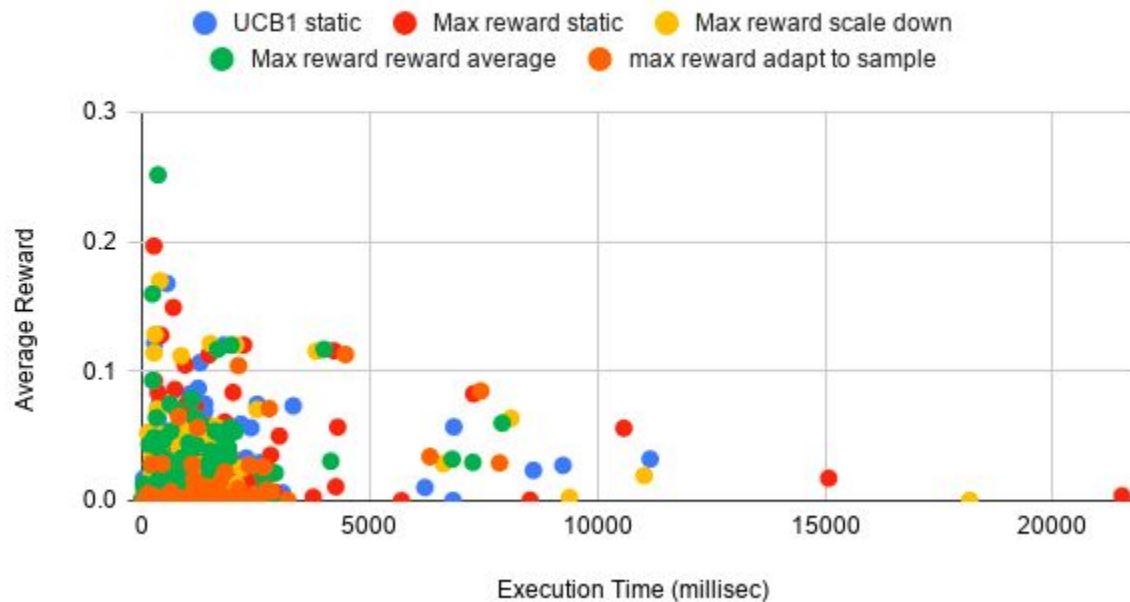


Analysis for UCB1 Default Selection Policy

- 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

Selection Policy - Max Reward

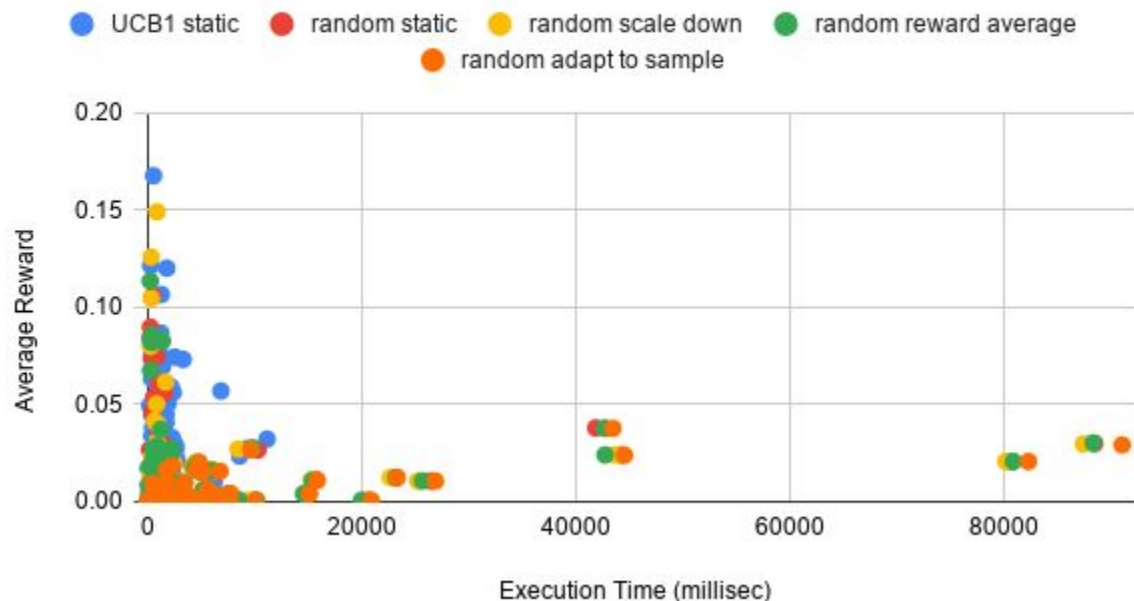


Analysis for Max Reward Selection Policy

- 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

Selection Policy - Random

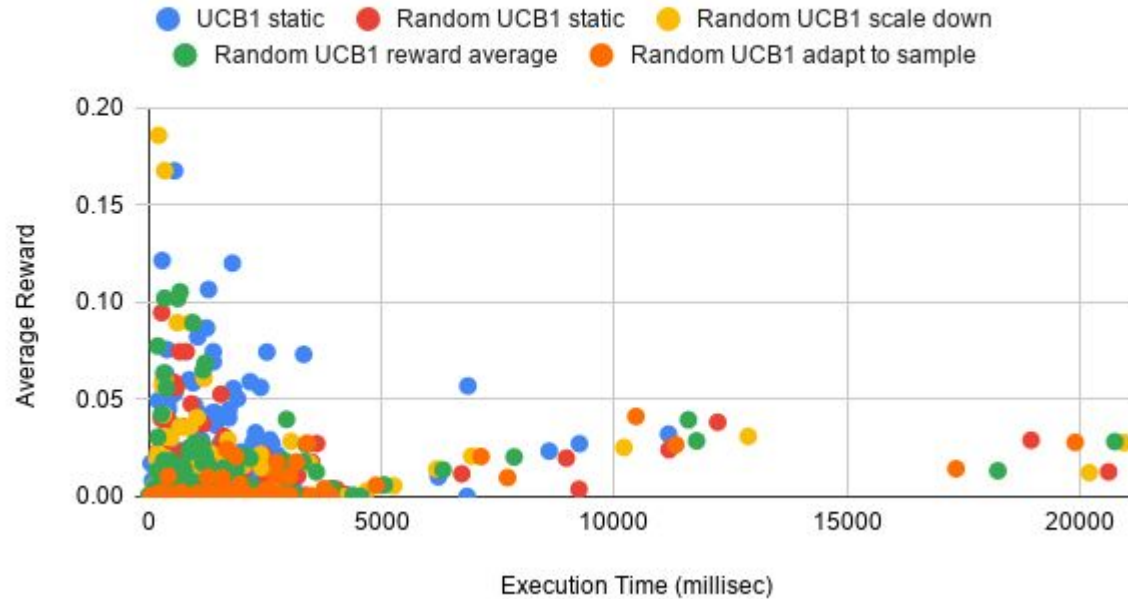


Analysis for Random Default Selection Policy

- 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

Selection Policy - Random UCB1

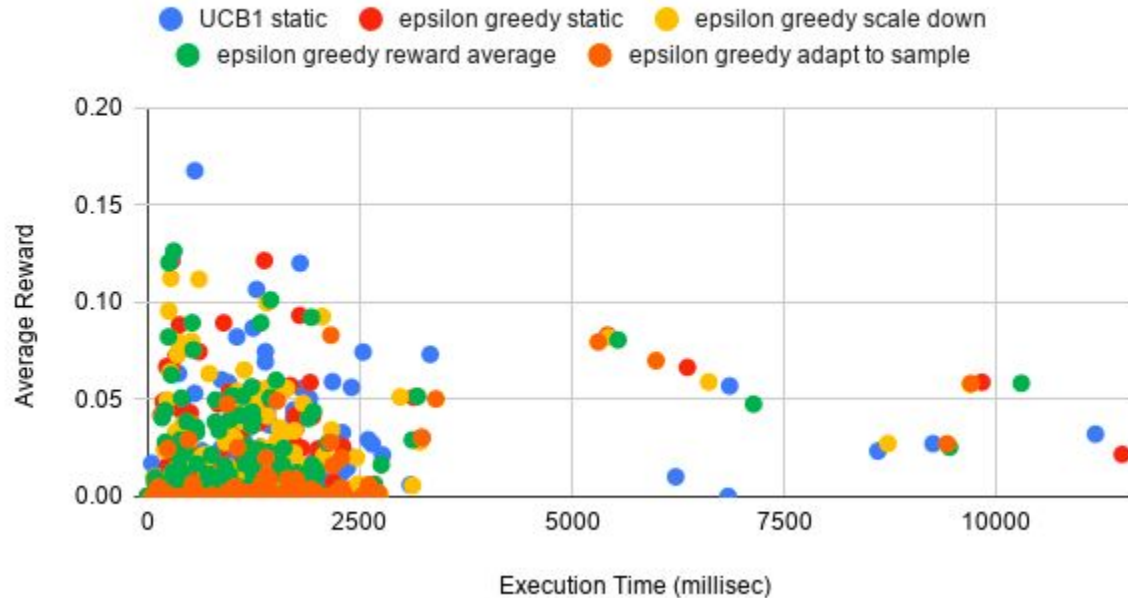


Analysis for RANDOM_UCB1 Default Selection Policy

- 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

Selection Policy - Epsilon Greedy



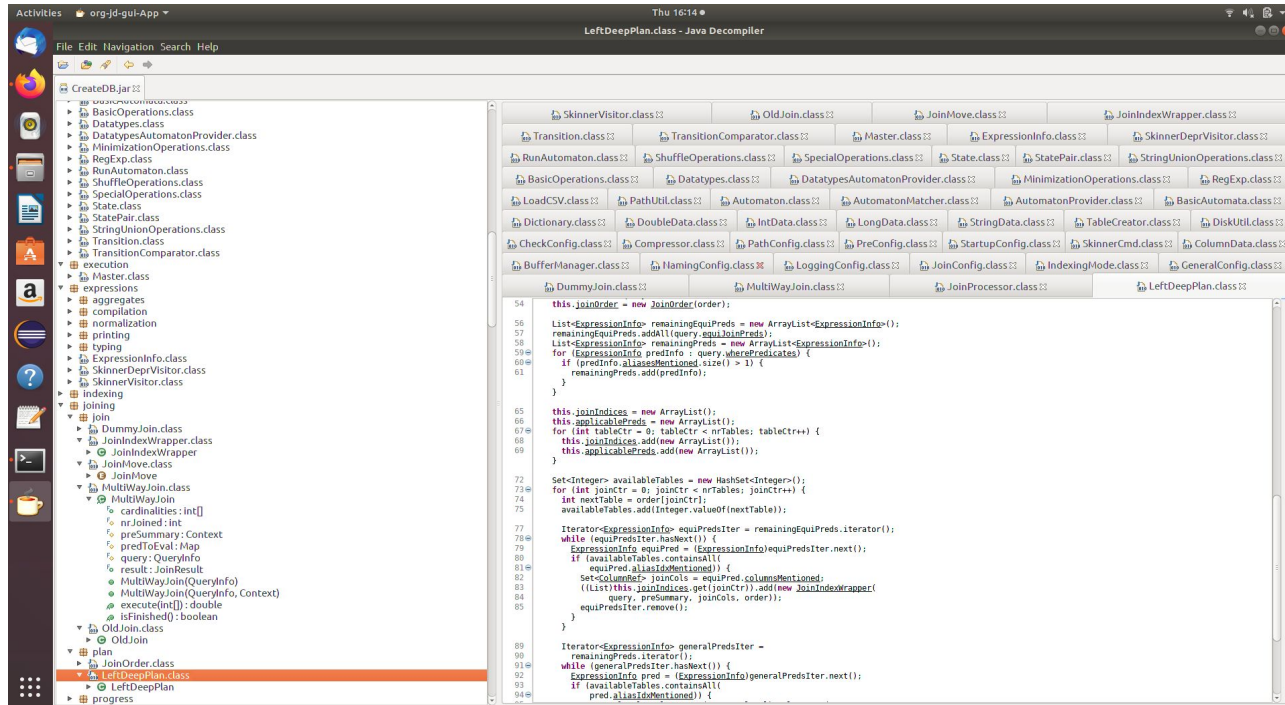
Analysis for Epsilon Greedy Default Selection Policy

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

Reverse Engineering for New Dataset Creation

- CreatedB.jar is used to generate binaries from the text database (in csv format)
- The source code of Create DB.jar was unavailable
- To create binaries of new datasets in SkinnerDB format, we decompiled the .java files with JD decompiler from the .class files
- Decompiling is challenging as the source code is incomplete and there are 30 packages each having multiple source codes


Reverse Engineering Approach



Permuted Dataset Approach

- Idea of dataset permutation was adapted (standard ML approach)
- Specifically, column in the database tables were shuffled
- This shuffling does not affect the integrity constraints on the tables
- Two different shuffling approaches were followed:
 - Random Shuffle
 - Simple Sort

1	2	3
4	5	6
7	8	9



7	8	6
1	5	9
4	2	3

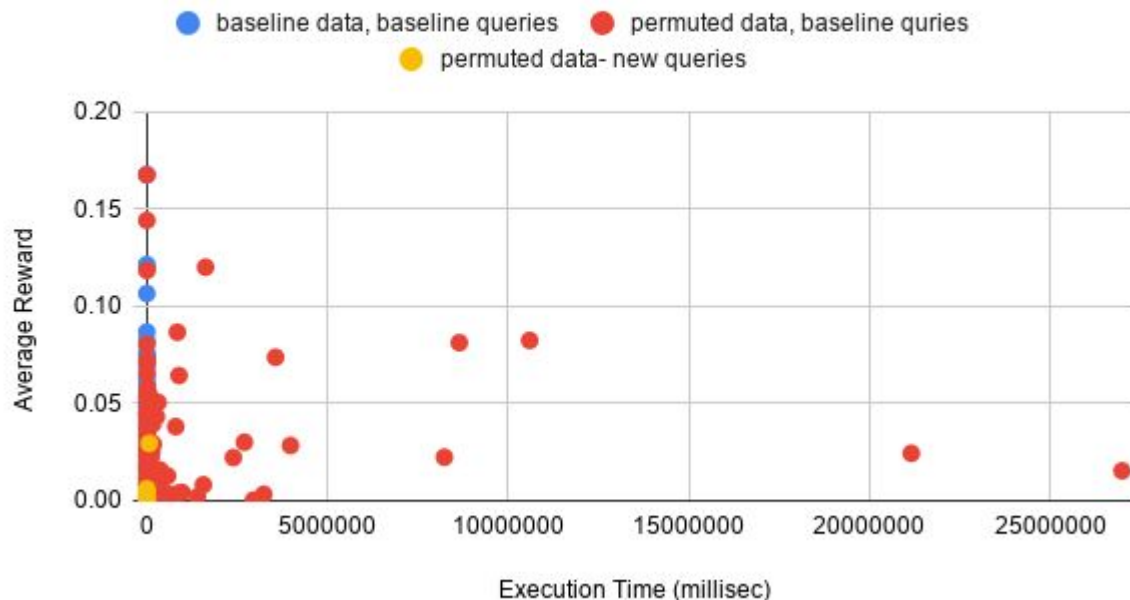
Illustration of the Permutation Approach
(Random Shuffle)

Generalization of the Best Strategy

- We have demonstrated that the Best policy combination of Max Reward as Selection Policy and Reward Average as Exploration Weight Policy illustrates a similar trend on permuted dataset.
- We have also tested generalization with new query sets over the existing 113 queries.
- The Modified Approach is obviously not generalizable for a completely new dataset due to scalability differences between the 2 datasets.

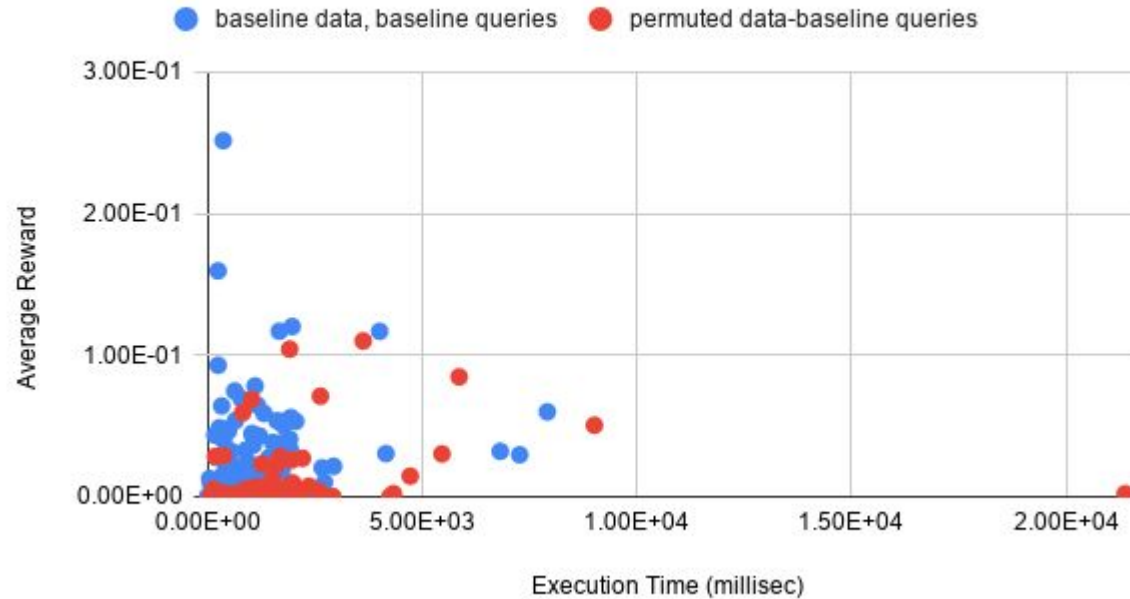
Generalization of SkinnerDB Implementation

Generalization of original SkinnerDB implementation



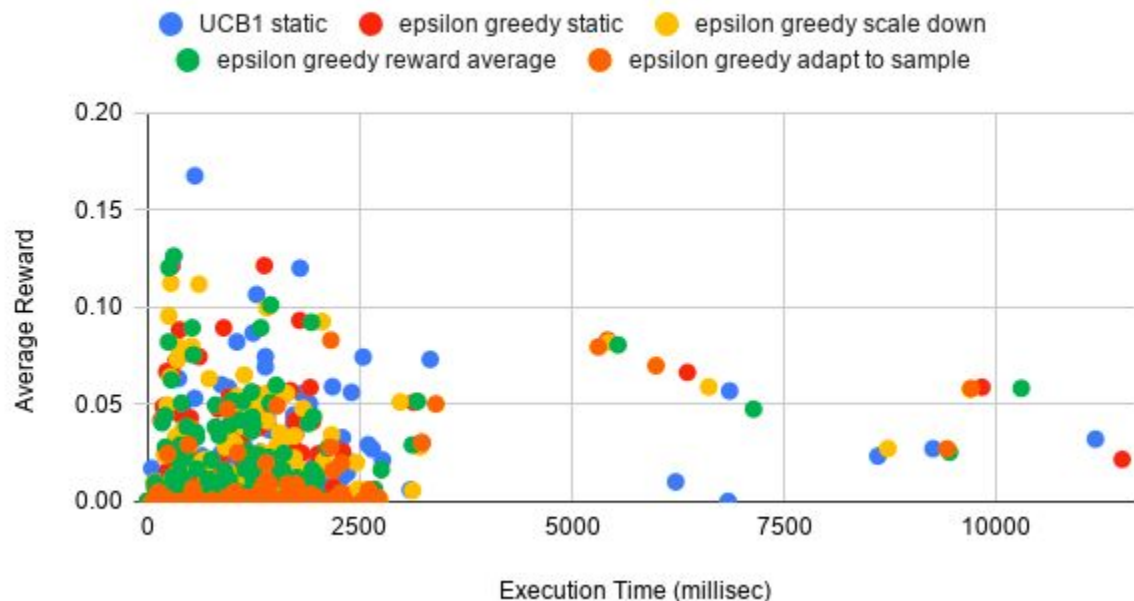
Generalization of Our Best Policy Combination

Generalization of maxreward-reward average Policy



Generalization for Another Policy Combination

Selection Policy - Epsilon Greedy

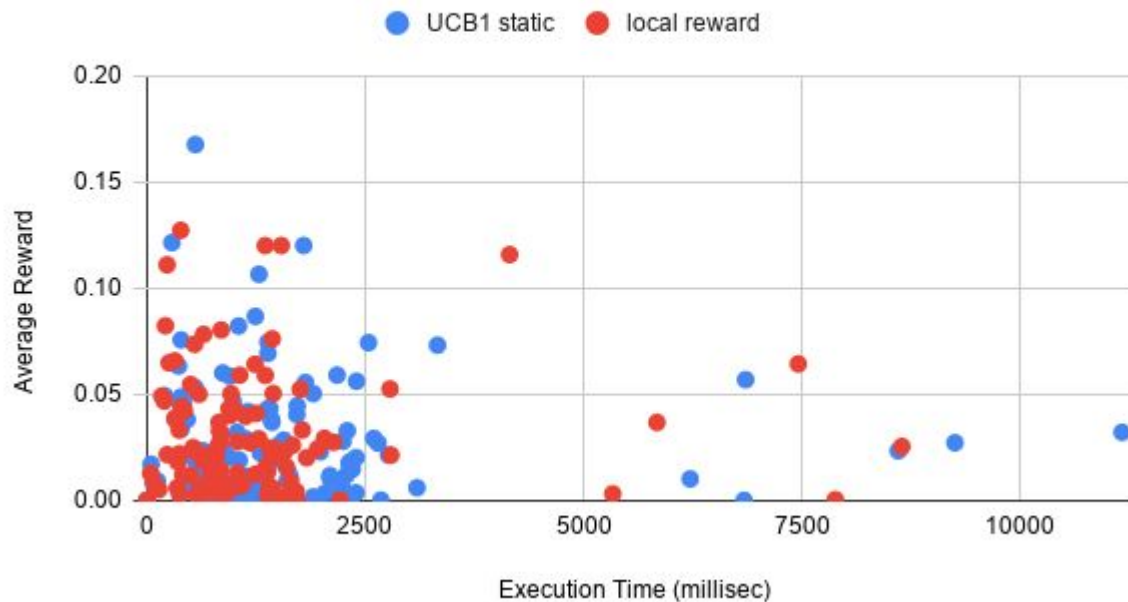


Results of Generalization on Permuted Dataset

- Both the shuffling strategies worked the same way
- No significant changes to either execution time or max reward was observed
- Conclusion: This approach is generalizable (atleast to this benchmark)
- Reason: Because of Intra-Query Learning (not dependent to past queries)

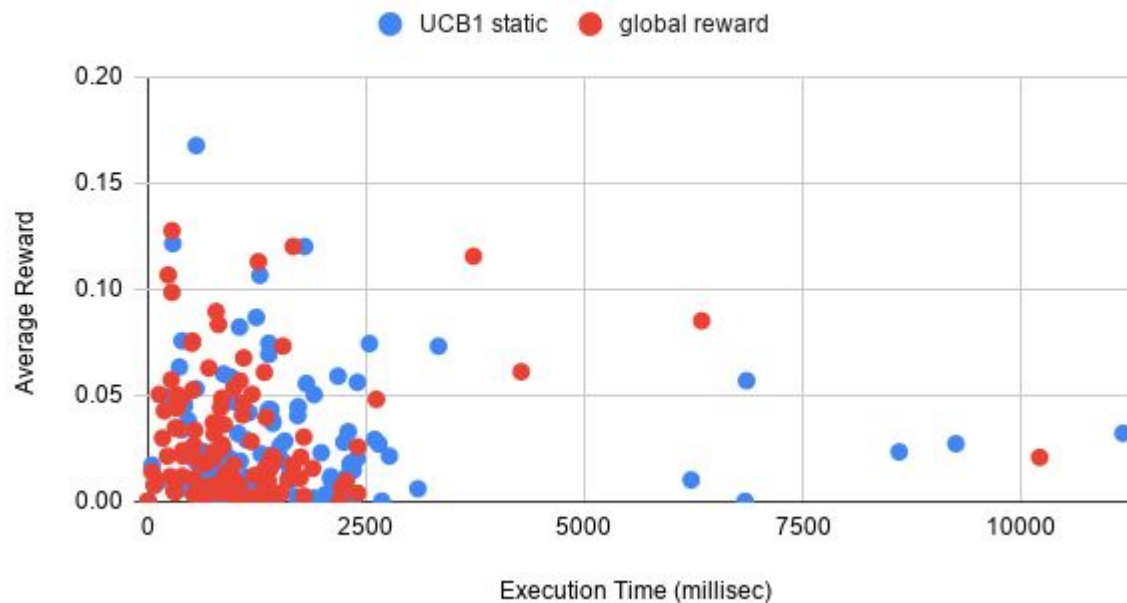
Elastic Time Slices Scaled By Local Reward

Elastic time slice with local reward



Elastic Time Slices Scaled By Global Reward

Elastic time slice with global reward

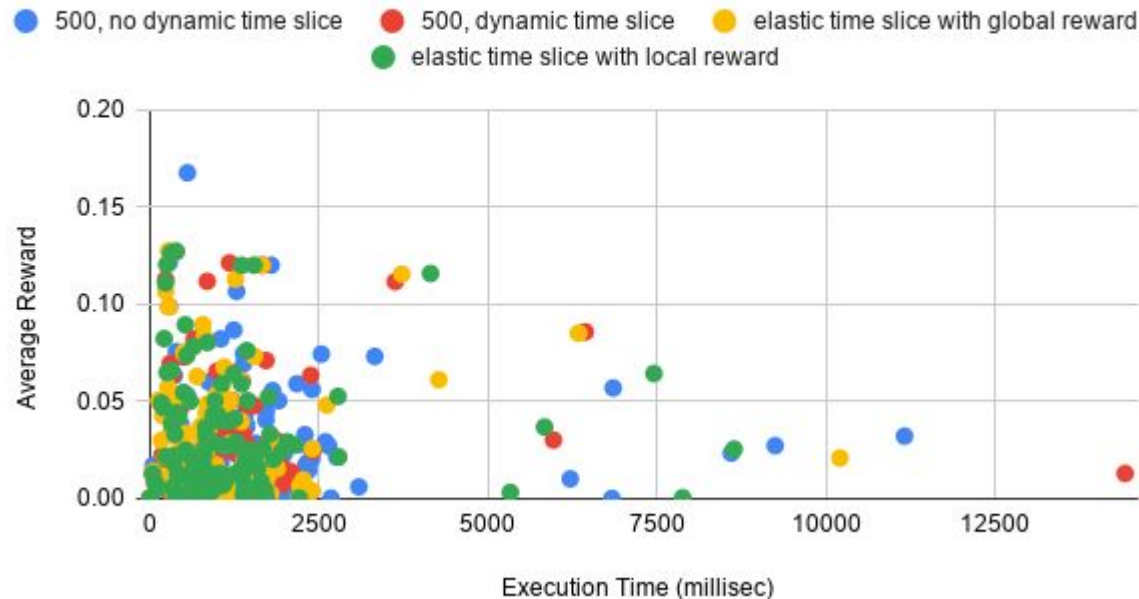


Elastic Time Slice with Local and Global Reward

- Local Reward: Between 0 to 1, and proportional to progress
- Global Reward: Cumulative reward from the UCT search tree
- The reward has been normalized by a constant of 5 and type-caster to integer
- We observed that elastic time slice with local and global reward perform better than static reward function.

Comparative Analysis of the Time Slices Approaches

Dynamic and Elastic Time slices



Query Execution on FIFA17 Dataset

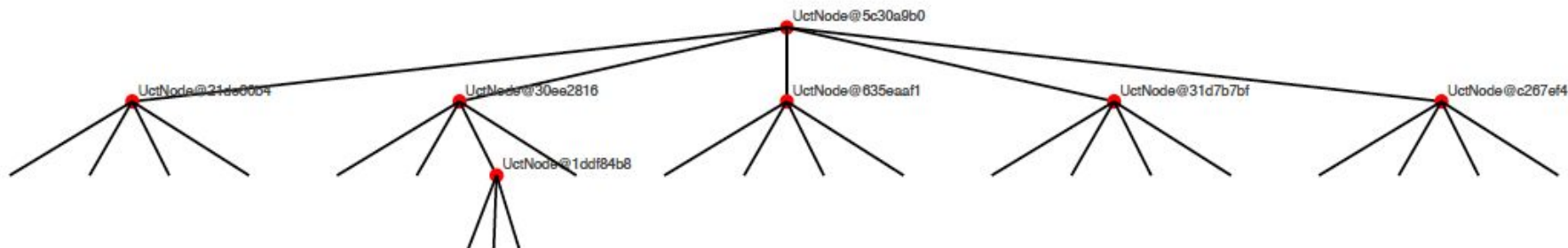
- The FIFA17 Dataset is much smaller (10 MB) than the IMDB dataset (3GB) with 4 tables: *FullData*, *ClubNames*, *NationalNames* and *PlayerNames* [7]
- We created 10 queries for the above dataset to run the SkinnerDB.jar file
- Execution time is much less than IMDB dataset (max of 161 ms. vs. max of 1857 ms. for IMDB dataset)

UCT Tree for a query

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



UCT Tree Explanation for each node

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

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

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>
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.

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>
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.

References

6. <https://www.youtube.com/watch?v=QRYVnKaZ9fw> (*SIGMOD'19*)
7. <https://www.kaggle.com/artimous/complete-fifa-2017-player-dataset-global>

THANK YOU