SELMA DQM: Self-Driving Materialized Views

Ashwini Gonibedu Dathatri
Data and Knowledge Engineering
Otto von Guericke University
Magdeburg,Germany

Disha Kishore Setlur
Digital Engineering
Otto von Guericke University
Magdeburg,Germany

Manoj Milind Borkar
Data and Knowledge Engineering
Otto von Guericke University
Magdeburg,Germany

Pooja Ajit
Digital Engineering
Otto von Guericke University
Magdeburg,Germany

Richhiey Thomas
Data and Knowledge Engineering
Otto von Guericke University
Magdeburg,Germany

Sharanya Hunasamaranhalli Thotadarya
Data and Knowledge Engineering
Otto von Guericke University
Magdeburg,Germany

***Abstract*—Large amount of data is stored and retrieved everyday from database tools in organizations. Maintaining stable query performance is a challenging task. This paper explores techniques to improve query performance. Usage of intermediate results of queries can reduce the computational cost and improve DB performance. The problem of Materialized View Selection is addressed here. There can be static or dynamic approaches to the this problem. In our study we consider a dynamic approach, based on deep reinforcement learning. We implement a prototype using the Apache Hive, SQL-on-Hadoop database and we experiment using the Join Order Benchmark. Hive improves the computational cost and query performance by rewriting the queries in accordance with selected materialized views. With the usage of Deep Reinforcement Learning we develop a system called Selma DQM, wherein the incremental value of materializing certain views is evaluated. This is based on selection and eviction policies. This system aims to be an improvement to the heuristic approach and works on an evolving workload. The system is trained using existing queries.**

I. Introduction

Billions of people in the world today are generating data. A study by IDC titled 'Data Age 2025' says "Worldwide data creation will grow to an enormous 163 zettabytes (ZB) by 2025"[1]. As we know, fetching this data efficiently, for a large number of queries is a demanding task for databases. Due to the enormous volume of data, and the fact that some parts of the data tend to be queried more often than others, queries tend to overlap and produce intermediate results[8]. Having smart databases that would reuse these intermediate results across queries and not having to pre-compute them everytime, helps to improve their efficiency. This gives rise to the idea of efficient Materialized View Selection.

Views are tables created during query execution. Ideally, these are formed by combining two or more base tables, that are not physically stored in the shape of the view. They contain the intermediate results of join queries, for example, or a subset of rows and/or columns of a table, or even summary results using aggregation functions[12]. On an everyday basis, increase in the frequency of similar query execution causes the same views to be fetched repeatedly. This results in poor query performance and a high time consumption [6].

In order to optimize this query performance, materialized views are created. Materialized views are tuples obtained from these views and stored physically in the cache. Fetching these results is much more efficient than having to repeatedly pre-compute the same results. But creating materialized views for all queries results in poor run time, and is not cost effective. Instead, with prior knowledge about the incoming queries, a subset of views that need to be materialized are first established. This is called the Materialized View Selection Problem (MVS).

There have been studies which extensively cover the scope of View Recommendation Systems. These are systems that suggest the views that need to be materialized. The decision is based on factors such a database schema, the historical query workload, and possibly a cost model. Hence this kind of system helps to overcome the View Selection Problem [11] [5] [3] [10] [1] [16]. One technique from these View Recommendation Systems is retrospective in nature, where it is assumed that future queries and the data are similar to that of the past. This assumption does not hold good for incoming queries that are generated in a different manner than the original queries. Hence the retrospective approach fails to work on ad hoc queries and evolving workloads. [8]. These limitations are overcome by dynamic strategies which adapt to changing environments. Reactive dynamic strategy analyses: i) The views that need to be materialized, keeping into account the storage constraints, ii) New queries that should be added in the workflow, and iii) The materialized views that should be evicted, to meet the overall expected cost for future workloads. However, the existing dynamic view selection work is heuristic based, when selecting which queries to add to the workflow, and also for the eviction task. This choice can be brittle in nature with respect to changing workloads.

[1]Source: https://blog.seagate.com/business/enormous-growth-in-data-is-coming-how-to-prepare-for-it-and-prosper-from-it

Thus progressing from a non dynamic strategy for static workload to a dynamic strategy for an evolving workload based on heuristics, we now arrive at a smart approach that is non heuristic in nature, and makes intelligent decisions to create, retain and evict materialized views[12]. Such system was proposed by Liang et al.

In our case, we design and evaluate a similar system, which observes if the materialized view creation has a net positive or negative impact on subsequent query execution. This decision is based on a frequency distribution using the Markov Decision Process. The frequency here is the number of times a particular query is accessed over time. We assign this net impact to a performance metric. This is a learning problem because the higher the performance metric, implies that the system decides to use this materialized view for future, and the lower the performance metric, the system must decide to avoid these counterproductive materialized views. Thus by considering actual run times of queries, future query plans are optimized. To achieve this goal, we need a framework that can automatically and continuously update its model based on past observations.

"Learning by doing" is the fundamental concept of Reinforcement Learning (RL)[14] [13]. It takes a set of actions using a policy/function and receives a feedback from a so-called environment. Through its policy, RL agents learn over time which actions to take to maximize their long-term reward. Connecting this to the Materialized View Selection problem, here RL agents can be adopted to learn which new views are valuable to materialize considering the current system state. Agents can also adapt to changing workloads based on the observed performance. This predictive model simply contributes to minimizing the overall query latency. To handle the storage constraints, Eviction and Recreation Policies are implemented. The eviction policy eliminates the materialized views that are least recently used, while the recreation policies are known to create and retain materialized views that are used on a regular basis.

Liang et al. consider an RL implementation for the MVS problem, where selection policies are effectively trained with an RL Algorithm. Authors implement a system called as DQM, using SparkSQL on two different workloads including Join Order Benchmark and the TPC-DS. However, this implementation is not open source, and in order to study the problem better, a re-implementation is needed. Hence we propose to implement and study a similar design. In contrast to Liang et al., we consider a different approach to query featurization, which might help the agent to learn better and hence take better decisions. [8]

To enhance query featurization we design a model in Deep Q-Materialization (DQM) system consisting of three main components: (1) A query parser that loads the queries into a candidate matrix, (2) a Dopamine agent that performs selection of these candidates for materialization, and (3) an eviction policy that selects the views to be deleted. This DQM system is integrated with Hive DB, and includes mechanisms to re-use observations from the training, such that itself can run faster. Since we are based on the goal of providing an early re-implementation of the study which deals with opportunistic view materialization [8], we consider the Join Order Benchmark and select Hive DB since it supports automatic query rewrite. This adaptive policy interacts with the Hive Environment across distributed systems[7].

To summarize this paper, we provide the following contributions:

1) We discuss how Self Driving Materialized View Selection(SELMA) works using a Markov Decision Process(MDP). We provide a novel design.
2) We implement a reinforcement learning algorithm based on Deep Q-Neural Network model to refine the MDP objective.
3) We evict views based on an Eviction Policy that handles the storage constraints, with a less-recently-used policy.
4) We present the experimental setup of our system which helps in understanding as to how efficient the framework works with the proposed use case.

The paper structure is as follows:
Section 2 gives an overview about the Background Research. Section 3 presents our Design and the System architecture in detail. In Section 4, we discuss the experimental evaluation of our system including the Experimental setup. Finally, Section 5 presents the Conclusion and the future work.

II. BACKGROUND

In this section we introduce and understand the prior approaches implemented to materialize the views, and discuss the concept of policies and Deep Reinforcement Learning.

A. *Existing Approaches for Materialized View Selection*

Hive and other large scale data management systems, handle automatic query rewrites. Thus, when materialized views are explicitly used, they are also automatically synced in the database, as a byproduct of query execution (i.e. views are updated and are kept consistent with the actual data) and without any human intervention. By reusing the intermediate results computed from frequent queries, the overall query performance is improved and cost is reduced [8]. An heuristic-based dynamic approach in the related field of work is the History-aware Query Optimization with Materialized Intermediate Views(HAWC) [11]. This heuristic based approach makes use of history pool and a view pool. It is an optimizer that uses two strategies: Query Optimization and Integer Optimization. The former uses history information to identify those query plans that produce intermediate results, such that by creating materialized views for these queries, it can potentially reduce the execution time of future queries [11]. The latter handles the update of the view pool with those views which are

the most cost effective in terms of execution. From the history pool, if the benefits of materializing the intermediate results and reusing it for future query plans outweighs the cost of running an expensive query plan, then these optimal strategies, are implemented more often. It makes use of a history pool containing a set of materialized views subject to disk space limit, which increases the chances of future queries reusing them. But these existing heuristic approaches have limitations. We cannot be sure if this approach would work for changing workloads.

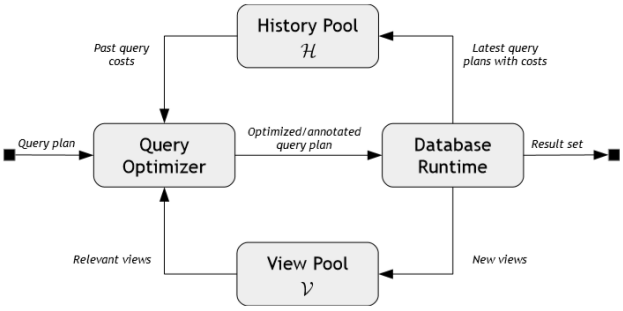

Fig. 1. Architecture of HAWC query optimization framework [11]

Another study on RECYCLER [9] gives priority to the most expensive views with respect to the creation cost. However, this approach also has a limitation. If the evicted view needs to be used, it becomes a tedious task to compute it. This can happen repeatedly since high creation cost views prevent other views from being stored. Adapting online approaches like DQM that observe the query latencies as the feedback will hopefully improve the query performance cost, since it models the trade-off between the gain from materializing common less creation cost views, with regards to the loss from evicting high cost less used views.
The study about Opportunistic View materialization [11] focuses on DRL that learns adaptive view materialization and eviction policies.

B. Deep Reinforcement Learning

RL is a cyclic process where an agent interacts with an environment without any prior knowledge, gains information, and performs actions over time. Each action the agent performs changes the existing state of the system. The goal of the agent is to learn a function that allows it to optimize its decision to choose an action.

In the context of our DQN system, the agent is trained to decide if a view should be materialized or not in order to reduce the computational cost and execution time of a join order query. This decision making process of the agent fundamentally is based on the Markov Decision Process. It is trained based on a policy that maps its action to the previous state, the state being the frequency of querying a particular join.

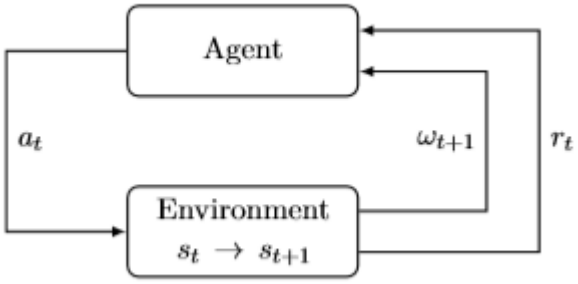

Fig. 2. Agent-Environment Interaction in DRL [14]

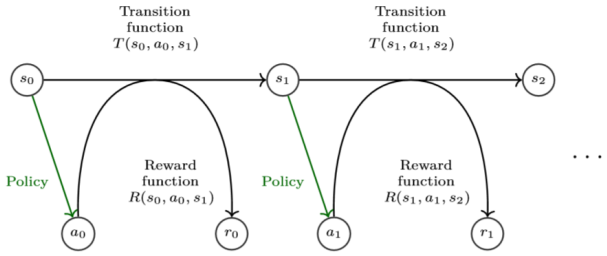

Fig. 3. Markov Decision Process: At each step, the agent takes an action that changes its state in the environment and provides a reward. [14]

This policy quantifies the efficiency of the model. When the agent exhibits a good behaviour, it is rewarded with a positive value else a negative value is rewarded. The positive value rewarded is based on the agents decision to materialize a join depending on the computational cost and the optimized time. The focus here is on maximizing the cumulative reward value. This policy component known to predict the goodness of an action/state pair is called the value function. The agent does not have any prior knowledge about the joins. This data is presented sequentially to the agent, which enables the learning behaviour progressively. This is called the online setting. Online setting is particularly advantageous in this scenario as it can identify the joins that occur more frequently and hence train better.

Deep reinforcement learning is an extension of RL, that uses neural networks for function approximation. In our work we focus on this approach.

In the next section we present the design of our solution.

III. DESIGN

In this section we present the design of our solution, that seeks to address our core research question: How well do DRL agents, perform in their training over a given workload?

Due to time and resource constraints, we scope our study to this research question, and leave aspects such as hyper-

parameter tuning and the re-implementation of a baseline as future work.

A. *System Architecture*

In this section, we discuss in detail about the current scope of our system design.

1) ***Agent****:* The agent plays the most crucial role in our system. It is responsible for taking decisions on whether a candidate view needs to be materialized or not. It runs in an episodic manner.

The agent works its way by taking actions within the environment that it is provided with. In our case, our agents play around in a custom environment which interacts with a database. The agent can take one of the two actions in this environment, either to materialize a candidate view, or to not materialize it. The input to the agent at each step consists of a concatenation of the current observation space and the current input candidate view. Using this input, the agent decides whether to materialize this candidate view or not. The agent takes these actions to maximize the long term expected reward from creating views in the environment. This is done using a reward function - assigns a reward for each action it takes. The reward function that our agent uses to learn in this environment is as follows:

$Reward = 100 * \sum_{i=1}^{k} baselines(i)/runtime(i)$

where baselines is the base cost of the query and the runtime is the current cost of the query (with the views selected in the current episode).

Challenges : The agent faces a challenging scenario. Once it has trained on this value function to an extent, the question is if the agent has to explore more in anticipation of being accurate over a large amount of data, or just exploit what it has already learned. Another challenge encountered is the Credit Assignment Problem. The agent usually receives rewards at the end of the episode. But it is important to determine which action fetches the reward. Each episode would check for multiple joins. Determining the joins that led to view materialization and helped in gaining the reward is demanding.

We overcome the challenge of Credit Assignment Problem by training 3 standard DRL agents, each having different features:

- ***DQN***- The output of this agent is in terms of Q-Values representing the corresponding action for the same state. This Q value is an optimal selection of action using a policy called Q function. It uses Experience Replay buffer wherein it stores and shuffles across actions based on its history.

- ***DQN-Rainbow***- The output of this agent is in terms of a categorical probability distribution instead of Q-Values. It uses Prioritized Experience Replay buffer wherein it stores and shuffles across prioritized actions that helped in maximizing the reward.

- ***DQN-IQ***- The output of this agent is in terms of Quantiles of the probability distribution. In our adopted implementation there is no usage of the prioritized experience replay buffer.

2) ***Environment****:* The environment provides a space in which the agent can take actions and observe rewards to improve itself. It specifies the structure for the observation space, the valid actions for the environment, and establishes communication with the database. Let us assume our database contains N tables. Within the environment, each view is represented as a one-hot feature vector containing $(n * (n-1))/2$ columns, where each column corresponds to a specific join between two tables in the database, without preserving order. The observation space is a similar array derived by adding the created candidate view in a step to a zero array. Thus, the observation space contains all of the created materialized views in the environment. The input to the agent at every step of the environment is a concatenation of the current candidate and the observation space, and this is represented by the state action featurization.

View Featurization -

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

Observation Space -

[0. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0.]

State Action Featurization -

[[0. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0.],

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]

The environment works in an episodic manner. Every episode runs for a certain number of steps, and in each step a candidate is provided to the agents to decide whether to materialize it or not. The candidates for the agent are picked by a weighted probability distribution over a set of queries. The candidates to be provided to the environment are stored in a queue. If the queue is empty, the environment picks a query and en-queues all its candidates. If not, the candidate is taken from the queue and sent to the agent. At the end of every step, the agent is provided with a reward.

- **Selection of the candidates:**
  We implement DQM on the MapReduce execution engine. 14 SQL Queries from Join Order Benchmark workload were selected. This is fed to the SQL query parser which will parse this into a vector format consisting of candidates of join order 2 as shown in the figure(4).

The output vector is a combination of all the possible candidates for the given input queries. From the matrix, a number of queries are selected by defining a weighted probability

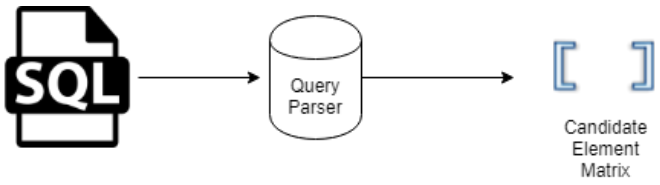

Fig. 4. Query Parser

distribution over them for each episode. Each episode is divided into an user-defined number of steps. The candidates are then fed into the environment. The environment interacts with the agents. The agent trains the model based on the selected candidates. The agent performs either of the two actions: to create a materialized view or not to create a materialized view. The Environment setup is discussed in detail in the later stages of this section. Figure(5) shows the game design in an episodic manner and Figure(6) shows the algorithm for the environment.

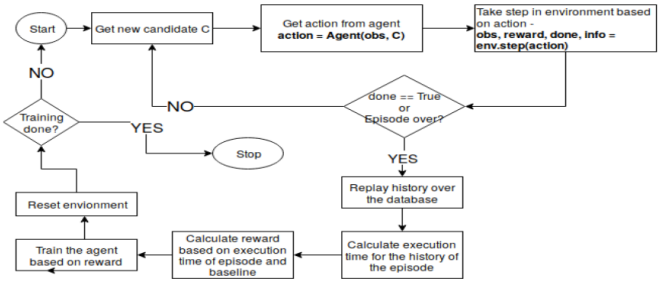

Fig. 5. Reinforcement Learning Game Design

Game Design :

1) The agent receives the candidates from the environment.
2) One of the two actions: Materialize a view or do not materialize a view is generated by the agent based on a policy.
3) For every action, a corresponding reward is assigned and the observation space is updated.
4) This is an iterative process which is checked based on the completion of the episode.
5) The query name, the candidate name, actions, and the step number is stored in the history buffer of the observation space.
6) If the episode is completed, the environment picks up the output from the agent and is communicated to the database which replays the history.
7) The respective execution time for replaying the history of the episode and the reward obtained based on the execution time are calculated.
8) The agents are then trained based on the reward and the environment is reset. And upon completion of training, the episode is then stopped.

Algorithm

```
1.  def step(action):
2.      take_action(action)
3.      If action == CREATE_VIEW :
4.          obs = update_obs_space(action)
5.          reward = 1
6.      Else
7.          reward = 0
7.      Store actions and other information in env history
8.      If episode over :
9.          replay history on hive DB
10.         calculate costs & reward
11.     Return obs, reward, ..
```

Fig. 6. Algorithm : Observation Space in the Environment

IV. Evaluation

A. *Experimental Setup*

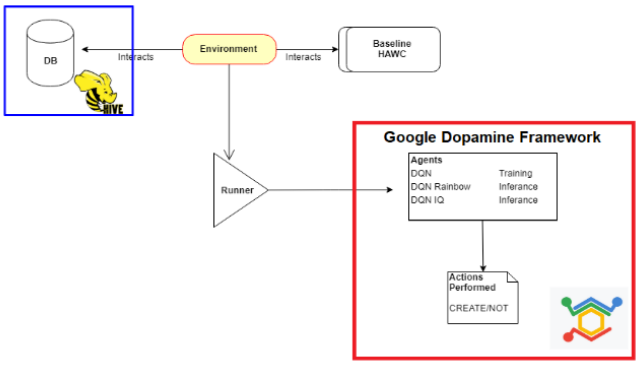

Fig. 7. SELMA Experimental Setup

The view creation action is issued and handled by the agent. The Database system interacts with the Environment and takes a call to create materialized views or not depending on the reward issued by the Google Dopamine agent.

*1) **Database**:* A database is a collection of tables and derived relations (materialized views) . Let Q be the set of read-only queries. These queries are fed to the Database in the order they arrive. The state of the DB changes accordingly depending upon the view creation and view deletion. Following this, the database system automatically re-writes queries for the views that are already materialized.

With respect to this feature of the DB, it is very important to make sure that we choose a DB which allows automatic query re-write. In accordance to this, we selected Hive as our potential DB. Hive stores materialized views using custom storage handlers. These handlers have explicit features such as LLAP acceleration which automatically produces full and

partial rewriting of queries for a large set which consists of joins, projections, filters and aggregation operations.[2] Hive also offers different kinds of file formats, namely JSON, Text, Avro, ORC, Sequence and Parquet. The file format of our interest is Parquet File because it uses Columnar Storage and helps in efficient data retrieval for our study use case. Parquet files also help in file compression[2].

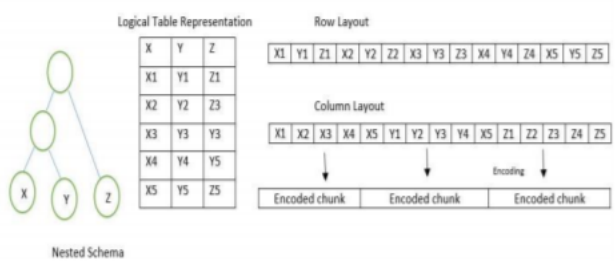

Fig. 8. Parquet File Structure. [2]

Hive runs on Hadoop - an open source distributed processing framework which implements map-reduce as it's execution engine. This is very effective to store and process large datasets on the hardware [15].

*2) **Dopamine and OpenAI Gym** :* Dopamine is an open source research framework which provides implementations of RL agents that are compact and reliable in nature. [4]. It was designed to depict the release of the chemical Dopamine in the human brain, which is a natural reward system in the biological system. The illustration of the Dopamine's design is depicted in the Figure 8, which includes the complete life cycle of an example experiment.

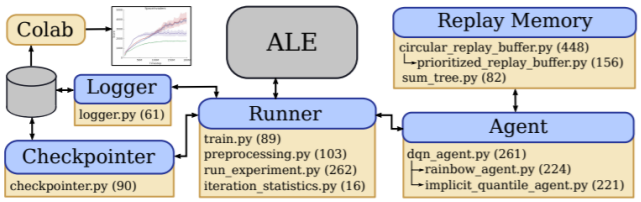

Fig. 9. Working design of Dopamine. [4]

OpenAI gym is a framework that helps build test environments that works on RL agent's to write algorithms and also to test them by sharing interfaces. This helps in enabling easy development and comparison of the RL algorithm. We use OpenAI Gym to build our database environment for the agent.

B. Experiment Results

We train three DRL agents on 14 queries from the Join Order Benchmark database. The Join Order Benchmark is a dataset containing join queries on the IMDB dataset, thus giving us join candidates for each query. Each of these queries have multiple candidates and the total number of candidates for the selected queries is 15. We consider only joins with table *title* for ease of implementation.

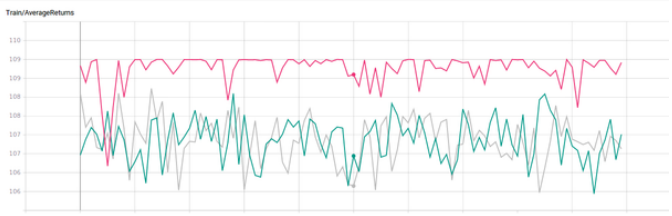

Fig. 10. Average rewards during training of agents. Red line indicates the DQN agent, green line indicates IQ-DQN and grey line indicates Rainbow DQN

Fig. 11. Average rewards during evaluation of agents. Red line indicates the DQN agent, green line indicates IQ-DQN and grey line indicates Rainbow DQN

Each agent is trained for 100 iterations with the above setup. Within each iteration, the agent plays 20 episodes within the environment. The maximum steps for an episode is set to 10. We run training and evaluation phases after every 200 steps for the agent, on each iteration. For the DQN agent, we use RMSProp optimizer with learning rate of 0.00025 and decay of 0.95. For IQ-DQN and Rainbow-DQN, we use Adam optimizer with an initial learning rate of 0.00005 and 0.09 respectively.

The experimentation results show that, DQN performs the best for the given problem in training and evaluation phases. In evaluation, we can see that the DQN behavior converges towards the end. In training, the rewards oscillate a little but seem to converge towards the end, as well. However for the IQ-DQN and the Rainbow-DQN, agents seem to be a little more challenged in the learning task and display erratic behavior throughout the training and test phases. This can be due to hyper-parameter issues, code configurations or it can also be a performance intrinsic to agent characteristics (e.g. DQN uses expected values, whereas the alternatives attempt to learn a probability distribution of these values). However more studies are needed to understand the causes of this different behavior.

IQ-DQN also shows spikes in rewards towards the end of its training phase. Thus, it seems that DQN outperforms the other two agents at the View Materialization learning problem. These results cannot be considered conclusive as the agents needed to be trained for more iterations, with more repetitions, and also with more queries and candidates. However, the

[2]Source: https://cwiki.apache.org/confluence/display/Hive/Materialized+views

experimental setup and execution of the experiment validates the use of Reinforcement Learning for the Materialized View selection problem. Usage of larger part of the dataset could push these findings further.

With these experiments we address our core research question, and we validate the goodness of our early implementation.

V. Conclusions and Future Work

In this paper, we studied the problem of optimizing the query workload using Deep Reinforcement Learning , where the agents learn the best way to select and materialize the views based on a reward system to improve the Database performance in terms of computational cost and query runtime. The experiments conducted provides affirmation on the path taken to solve the research problems defined for the scope of this project, however these results do not justify the true potential of the work in this field, as the experiments were limited to a small number of queries, candidates and iterations and a weak baseline. The future work with respect to the experiments can include implementation/use of a state-of-the art baseline, evaluating alternatives to tune the reward function, an evaluation of the eviction policy and full degree use of the complete dataset with the potential to make use of join order of larger magnitudes, or views of different shapes.

Exploration of various workloads and applications of Machine learning to improve the database internals will continue to be exponential especially in the field of materialization. We expect that our early implementation can help future students to understand this area better.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. 2000, pp. 496–505. URL: http://www.vldb.org/conf/2000/P496.pdf.

[2] Shahzad Ahmed et al. "Modern Data Formats for Big Bioinformatics Data Analytics". In: *CoRR* abs/1707.05364 (2017). arXiv: 1707.05364. URL: http://arxiv.org/abs/1707.05364.

[3] Nicolas Bruno and Surajit Chaudhuri. "Automatic Physical Database Tuning: A Relaxation-based Approach". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. 2005, pp. 227–238. DOI: 10.1145/1066157.1066184. URL: https://doi.org/10.1145/1066157.1066184.

[4] Pablo Samuel Castro et al. "Dopamine: A Research Framework for Deep Reinforcement Learning". In: *CoRR* abs/1812.06110 (2018). arXiv: 1812.06110. URL: http://arxiv.org/abs/1812.06110.

[5] Benoît Dageville et al. "Automatic SQL Tuning in Oracle 10g". In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 2004, pp. 1098–1109. DOI: 10.1016/B978-012088469-8.50096-6. URL: http://www.vldb.org/conf/2004/IND4P2.PDF.

[6] Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In: *IEEE Data Eng. Bull.* 18.2 (1995), pp. 3–18. URL: http://sites.computer.org/debull/95JUN-CD.pdf.

[7] Viktor Leis et al. "Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark". In: *The VLDB Journal* 27.5 (Oct. 2018), pp. 643–668. ISSN: 1066-8888. DOI: 10.1007/s00778-017-0480-7. URL: https://doi.org/10.1007/s00778-017-0480-7.

[8] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. "Opportunistic View Materialization with Deep Reinforcement Learning". In: *CoRR* abs/1903.01363 (2019). arXiv: 1903.01363. URL: http://arxiv.org/abs/1903.01363.

[9] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. "Recycling in pipelined query evaluation". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 2013, pp. 338–349. DOI: 10.1109/ICDE.2013.6544837. URL: https://doi.org/10.1109/ICDE.2013.6544837.

[10] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. "Efficient Use of the Query Optimizer for Automated Physical Design". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 1093–1104. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325974.

[11] Luis Leopoldo Perez and Christopher M. Jermaine. "History-aware query optimization with materialized intermediate views". In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 2014, pp. 520–531. DOI: 10.1109/ICDE.2014.6816678. URL: https://doi.org/10.1109/ICDE.2014.6816678.

[12] A. N. M. Bazlur Rashid, M. S. Islam, and A. S. M. Latiful Hoque. "Dynamic Materialized View Selection Approach for Improving Query Performance". In: *Computer Networks and Information Technologies*. Ed. by Vinu V. Das, Janahanlal Stephen, and Yogesh Chaba. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 202–211. ISBN: 978-3-642-19542-6.

[13] R. S. Sutton, A. G. Barto, and R. J. Williams. "Reinforcement learning is direct adaptive optimal control".

In: *IEEE Control Systems Magazine* 12.2 (Apr. 1992), pp. 19–22. ISSN: 1066-033X. DOI: 10.1109/37.126844.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN: 0262193981. URL: http://www.worldcat.org/oclc/37293240.

[15] Ashish Thusoo et al. "Hive: A Warehousing Solution over a Map-reduce Framework". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1626–1629. ISSN: 2150-8097. DOI: 10.14778/1687553.1687609. URL: https://doi.org/10.14778/1687553.1687609.

[16] D. C. Zilio et al. "Recommending materialized views and indexes with the IBM DB2 design advisor". In: *International Conference on Autonomic Computing, 2004. Proceedings.* May 2004, pp. 180–187. DOI: 10.1109/ICAC.2004.1301362.