Qd-tree: Learning Data Layouts for Big Data Analytics

Maria-Sofia Georgaki
National and Kapodistrian University of Athens
Athens, Greece
msgeorgaki@di.uoa.gr

ABSTRACT
Due to the huge size of data produced by modern applications and the need to process and analyze them, partitioning has become necessary. Query Data Routing Trees(Qd-trees) is a new framework that helps partition data by taking into account the metric of the numbers of blocks skipped, which relates to I/O costs and the efficiency of queries. Given a specific set of queries, a Qd-tree is constructed with the purpose of optimizing the aforementioned metric. A Qd-tree is used in both partitioning data as well as enhancing queries to take advantage of the completed partitioning. One way such a tree can be built, is by using a greedy approach. The other way proposed is to construct the trees using Deep Reinforcement Learning, which leads to gradually better trees using Reinforcement Learning while taking advantage of Deep Learning to address high dimensionality

PAPER OVERVIEW

The original paper presents a novel approach to optimize I/O costs using a known query set to build a tree which can then be used to partition data and accelerate query run times. The goal is to be able to skip as many blocks as possible when querying for data. In most cases, the minimum size of a block, b, needs to be set to cater to the underlying mechanics of the database. For example the minimum size of a block in SQL Server is 1 million tuples.

All pushed down unary predicates can be used in order to cut the space of the data and help in their partitioning. Each node of the tree can be cut only once and represents a subspace of the actual data-set. This can be achieved by the following node fields description

1. Categorical map
2. Range hyper-cube
3. Cut - if the node is not a leaf

The categorical map is simply a map that describes whether a value could be included in the subspace under the node. For example colour:[red:1, green:0, blue:1] means that red and blue can be under the node, however green is certainly not.In a similar way, the range hyper-cube can describe the ranges of values that can potentially be included in node's subspace. e.g quantity:[3, 8]. The cut is one of the unary predicates and helps form the sub-spaces of the children nodes that will follow.

Two ways to help build a Qd-tree were proposed. The first one is a greedy algorithm that chooses the next cut to be made according to the current best reward. However, as with every greedy algorithm, it leads to local optimum solutions and may miss the global optimum solution. On the other hand a way to address this fault in the greedy algorithm is to used Dynamic Programming to be able to keep track of all the states, actions and rewards. However, the high dimensional and complexity of the problem make it impossible to use Dynamic Programming. Which leads to the second algorithm proposed, which makes use of Deep Reinforcement Learning.

First of all the problem must be formed as a Markov Decision Process. The state can be any subspace of the data and the action space is the set of cuts at our disposal. When an action/cut is selected, two new states are produced which will then be explored. The agent used by the authors is Woodblock, which consists of two neural networks with shared weights and an underlying policy method for updating those weights.The first network is the policy network and the second one is the value network. Both of them are responsible for choosing the next action to be taken, by emitting a probability distribution for the action space and choosing an action by estimating the cumulative reward for taken the action.

The actual rewards are calculated in the end of each episode - construction of a tree by executing the query workload on a sample data set of size s in order for the calculation to be quick and efficient. Each node is assigned a reward after running the queries against the tree nodes: its number of records, if it is a leaf or the number of records of it's left multiplied by two if it is not a leaf.These rewards are then normalized by dividing with the number of the queries in the workload multiplied with all of the records under the node. This means that the meaningful rewards the agent receives as feedback are sparse and delayed. The underlying policy is Proximal Policy Optimization which is suited for such rewards.

The stopping condition for each tree construction is not based on the number of steps taken nor the depth of the tree, as it could potentially hinder the construction of better trees. The required minimal size of a block is used instead. If a cut results in sub-spaces with less than b records, then the node is not cut further and a leaf is formed. The construction as a whole stops when no more cuts can be made.

The process of building the tree and routing the records, happens offline. on the other hand the query routing happens online during execution.

Some framework extensions that could further increase block skipping were proposed.

Advanced Cuts : Include cuts made on columns e.q orderdate < shipdate. This can be a done by having an additional map to describe the subspace in the same way that the categorical map is used. It can include both equality and range operators and they are treated in the same way.

Data Overlap : Apply a relaxed cutting condition instead of the minimum size of a block. In this case a partition can have less than b blocks when the tree is constructed. Then all such blocks where |block|<b are identified and their records are copied to neighbouring blocks. In essence space which is relatively cheap is traded for faster execution times.

Data Replication: Again favoring time against space. Build a starting tree and then identify the queries that have the worst block skipping under the tree. Then a second tree is constructed by using these queries with the data being fully replicated in the new tree's blocks.

The authors conducted a variety of experiments with a combination of setups. Setups include a Single-Node Spark, a commercial DBMS and a Distributed 4 node Spark cluster. The data-sets used were TPC-H, with the schema denormalized and two real world data-sets which contained error logs from commercial applications.From the results it can be gathered that Qd-trees outperform other competitors and existing and wildly used partitioning techniques.

IMPLEMENTATION

The implementation for this project, targets the commercial DBMS experiments using TPC-H dataset and includes only the deep learning approach of the algorithm. Python3.8 was used, as well as Postgres as a database. The data-sets used were smaller in size compared to those of the authors of the paper(85 GB). Moreover the extended cuts extension was implemented as it was used in the authors' experiments as well. Please take note that not all operators are supported, meaning that not all templates the authors used are eligible for query generation. More information can be found in the readme file. In some cases the neurocuts source code was used as a guide https://github.com/neurocuts/neurocuts.

Records

The dbgen algorithm was adapted to have a new struct which was used for the denormalized lineitem object which includes all other values of the schema. The size of the produce file for scale factor one is around 8GBs. All values of all other tables are included in the lineitem object. There is an init.sql query in order to create the new table.

Queries

Not all templates where used with qgen, however the queries were adapted to the new linitem table and the new schema.

Query Parsing

The package sqlparse was used. Some of the keywords in some templates were not recognized by the package, which lead to the exclusion of those templates. The cuts are constructed by using the output of the sqlparse functions. The cuts have the following fields : attr1, op, attr2, cut_type

Tree Nodes and Tree Construction: Each node consists of three dictionaries. One for range values, one for the categorical map and one for the extended cuts map. As with any binary tree, each node also points to it's two children. The field block_id is used as a node id, which increments along the construction of the tree. Each node is also described by the binary encoded field which keeps track if the node is on left or right part of the cuts of the tree up to that point. The positions of this map are in fact aligned with the cut number that has been applied to the tree. The tree is constructed by using a python deque in order to be able to build it level by level, left to right, as described by the authors.

Record Manipulation and Routing

The records are read and manipulated by using pandas. When comparing records with the tree cuts, numpy functions are used so as to have vectorized commands whenever possible. They are directly stored in the database by using pandas _tosql function and an SQLAlchemy engine.

Query Routing

Each query is split into its cuts which are then compared to the node metadata each time. If there is an intersection, then the block_id of the leaf is appended to the block_id in clause of the query.

Deep Learning Environment

rllib was used for the deep learning algorithm, which is a library specifically for developing reinforcement learning algorithms. It is used in conjunction with the gym package which is used to describe the observation space and the action space. The observation space is in fact the state of the environment and the input provided to the models. In my implementation it consists of the available actions, the action mask and the state of the current node. The action space includes all the available actions, each action is sampled from this space. In this implementation, a custom model is used in order to mask the actions that have already been made, in order to not be able to choose them for future cuts. Reward calculation is as described by the authors. A MultiAgent environment is used, in order to be able to tackle each subspace as an independent problem. Sparse and delayed rewards are used, only when a tree is completed. After the end of each episode the newly constructed tree is save to a JSON file, so as to read it later to route queries and records.

Loading and Using the Qd-Tree

The Qd-tree can be loaded after its construction and be used separately from the deep learning environment. First of all, all the queries needed to query the partitions will be outputted in a specified file. Then it is up to the user to run them in order for the record routing to work correctly. All the records are routed and have their block id appended to the data-frame while in memory. Then they are saved in the corresponding partition. Similarly, the queries are parsed and they are evaluated against leaf node metadata.In this case, each intersection is added to the block_id clause which is printed for the user. Unfortunately these need to be added to the queries by hand.

Configuration
A JSON file can be found, which contains most of the configurable values for the agent and the experiments. It includes paths for the query directory, the data-frame which will be used etc. It can be customised according to the needs of the execution.

Supported Operators
The supported operators are =,!=, <>, <=, >=, <, >.

CHANGES AND ADDITIONS
The description of the nodes is the same, however the dictionaries are sparse. If a column does not exits in the dictionary, it means that there is a probability that the node's subspace contains the value wanted. Moreover a custom model is used with the agent, in order to mask actions already taken in previous steps. The algorithm without the mask would still favor other cuts as each cut repeated would probably lead to no skipping and it would be less likely for the agent to choose the same action again. As mentioned before not all operators are supported.

EXPERIMENTS
The experiments include only the TPC_H dataset, using Postgres and were done with trees that were created after running the agent to completion with the current settings. The data-sets used were generated with dbgen. Query execution time was measured with and without the partitioning. All experiments were run on a laptop with 32GB of RAM and an i7-1185G5 processor. Please note that in one of the last experiments a bug was found that may influence test results because the routing stops midway because it leads nodes to non existing partitions.The rate achieved is not as high as the authors'. It was close to 4000 records per second.

	Experiments Times	
Data size	*Tree constr time*	*Routing time*
8GB	252 secs	30mins
16GB	418 secs	1 hour

Table 1. Timed Experiments

OBSERVATIONS
The construction time of a tree can be trivial when compared to the overall gain in query execution time. Moreover the execution could be stopped whenever the user wants and the last tree will still have been saved for future use. The agent can run in memory, however it is quite demanding, taking into account the sample size as well. A solution would be for it to be loaded in chunks each time from disk, but I believe it defeats the purpose of the algorithm. In computers used in productions environments RAM would not be a problem.

CONCLUSION
Taking into account queries in order to build a learned index for records and queries alike is a quite interesting idea. With the growing number of data produced by applications, it is necessary to find better and better solutions that can efficiently tackle problems that handle a large amount of data. Time has become more important than the price for hardware, more disk space can be added at a low cost and processing resources are wildly available. Qd-trees provide well rounded solution to the problem of block skipping. The only drawback I believe, is the need to update and reconstruct the tree when necessary, for example when a new set of queries is added to an existing workload and the new queries have very different filters. If a new tree was to be built, then all of the records would have to be saved again, which would be time consuming for big data-set and may result in a need for downtime of a service etc.