HyperPartitioning

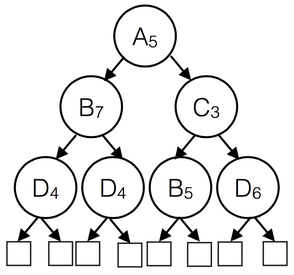
Anil Shanbhag (anils@mit.edu), Yi Lu (yil@mit.edu)

Introduction

- Data partitioning significantly improves the query performance in distributed database systems.
- Existing techniques focus on finding best data partitioning given a workload.
- Many modern analytics applications involve ad-hoc/ exploratory analysis.
- Workloads might change over time.
- Static workload-based partitioning not suitable.
- HyperPartitioning is a multi-attribute data partitioning approach which does not require an upfront workload and adapts to the user queries.

Our Approach

- Many data warehouses uses a block-based storage system like HDFS to store their data. Instead of creating the blocks based on size, we create blocks based on a partitioning tree.

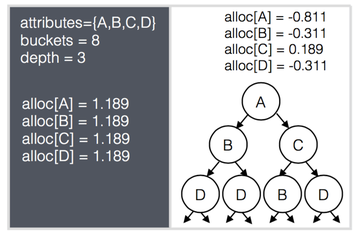


- During data upload time, we use the upfront partitioning algorithm to generate a balanced partitioning tree. The tree is created such that:
  1. Average number of ways of partitioning of each attribute is the same
  2. Each leaf node (~ block) has same size.
- At runtime, when a user submits a query, the optimizer is run. The optimizer tries to improve the partitioning by replacing / reorganizing the partitioning tree based on query predicates.
- The re-partitioning interleaves with query execution there-by sharing scan. Also, we never re-partition data not accessed by the query.

The source code is available at github.com/mitdbg/mdindex

Upfront Partitioning

- Goal: Partition the data so that the average partitioning on all the attributes is the same.
- Done at upload time.
- Allocation$_j$ (average partitioning of an attribute j) $= \Sigma\ n_{ij}\ c_{ij}$
  n_{ij} is number of ways node i partitioned on attr j
  c_{ij} is the fraction of data this applies to
- Create tree with num of leaves == num of blocks
- Breadth first traversal over tree. At each node, set attribute with highest allocation remaining at the node.



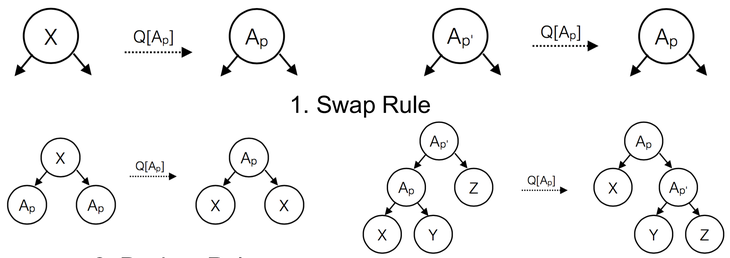
Adaptive Re-Partitioning

- Cost Model

$$\text{Cost}(T, q) = \sum_{b \in \text{lookup}(T,q)} n_b$$

$$\text{RepartitioningCost}(T, q) = \sum_{b \in B} c \cdot n_b$$

- Repartitioning happens when reduction in the total cost of the query workload is greater than re-partitioning cost.
- Alternatives found via transformations on the partitioning tree

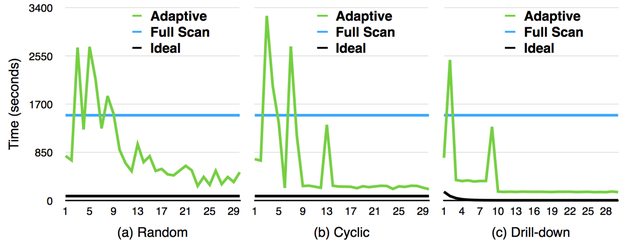


Exploring Alternatives

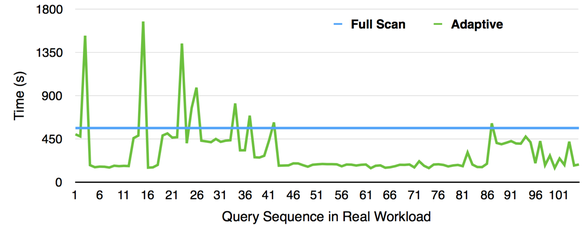
- Goal: Explore the alternative partitioning trees to check if inserting incoming predicate A_p into the partitioning tree is feasible.
- Use bottom-up approach.
- Find the best plan for left subtree and right subtree. This information is sufficient to apply all the rules.
- Only rule 1 does physical re-organisation. After inserting predicate, we use the sample to re-estimate the bucket counts. Use these to see if plan beneficial.
- If query has multiple predicates, insert one at a time. Find the best plan. Now in the best plan, try to insert predicates not inserted in best plan. Do till no change in plan.

Experimental Results

- Upfront data partitioning leads to 2x overhead.
- Optimization time is very small (1-5 seconds).
- First experiment is on system behaviour for different query patterns:



- When applied to a real-world workload:



Query runtimes for on a workload from a local startup.