Cost-Based Optimization of Logical Partitions for a Query Workload in a Hadoop Data Warehouse

Shu Peng[1], Jun Gu[1], X. Sean Wang[1], Weixiong Rao[2], Min Yang[1], and Yu Cao[3]

[1] Shanghai Key Laboratory of Intelligent Information Processing, School of Computer Science, Fudan University, Shanghai, China
{pengshu,gujun,xywangCS,m_yang}@fudan.edu.cn
[2] School of Software Engineering, Tongji University, Shanghai, China
rweixiong@gmail.com
[3] EMC Labs, Tsinghua Science Park, Beijing, China
yu.cao@emc.com

Abstract. Recently, Hadoop has become a common programming framework for big data analysis on a cluster of commodity machines. To optimize queries on a large amount of data managed by the Hadoop Distributed File System (HDFS), it is particularly important to optimize the reading of the data. Previous works either designed file formats to cluster data belonging to the same column, or proposed to place correlated data onto the same physical nodes. In query-workload aware situation, a possible optimization strategy is to place data that may not be used by the same query into different logical partitions so that not every partition is needed for a query, while physically distribute the data in each partition evenly across the compute nodes. This paper proposes a condition-based partitioning scheme to implement this optimization strategy. Experiments show that the proposed scheme not only reduces the I/O cost, but also maintains the workload of the compute nodes balanced across the cluster.

Keywords: Hadoop, Data Analysis, Data Partition, Query Workload, Cost-based Optimization.

1 Introduction

Over the past few years, Hadoop [1] has been widely used for data mining and data analysis due to its two advantages. First, the Hadoop parallel programming framework allows users to comfortably write MapReduce [4] jobs on a big cluster of commodity machines. Second, with the help of Hadoop distributed file system (HDFS) to distribute input files across the clustered machines, the MapReduce jobs can then process local file chunks with very low communication cost.

The main processing mode in using HDFS to evaluate queries is to scan the entire data. For an efficient use of the cluster of machines, the HDFS splits large data files into data blocks and then distributes the blocks randomly and evenly across the machines. This policy can balance the workload of the machines when the scan is done. When all the data needs to be scanned, this is indeed a very effective method.

Optimization over the above basic scanning method has been introduced in the literature, especially for data in a relational table form. RCFile [7] was designed as a data format that stores the column-wise compressed data in order to reduce the I/O cost. However, often a query does not need to use all the data in a data file. For example, a select-project query may only need to access a small portion of the rows and a few of the columns in a relation. A general approach in the literature (e.g. CoHadoop [5]) is to put the data needed by the same query onto the same physical machines to reduce data shuffling. However, this approach may lead to unbalanced workload across the cluster, reducing the efficiency of the whole cluster.

In this paper, we propose a novel approach, namely a condition based partition scheme (CPS). The main idea of CPS is to analyze the query workload and then to place correlated data into the same logical partitions, and each logical partition is instead physically stored in clustered machines. Here, "correlated data" means the data that are used by the same query. In detail, by analysing the queries, we divide correlated rows of input tables into the logical partitions. When the raw table data is uploaded to HDFS, the HDFS policy then still randomly and evenly distributes the data across the cluster, without incurring unbalanced workload across the cluster. When the queries are processed, only those logical partitions needed by the queries are accessed. In this way, we can avoid the full scan of the whole input data and access only the needed partitions, and yet do not introduce any imbalance in processing tasks among the cluster nodes. Consequently, the CPS greatly optimizes the query processing time.

In summary, in this paper, we make the following contributions:

1. We introduce a logical partitioning approach to avoid full scan of data, yet to keep the physical workload balanced across the cluster.
2. We design a method that does the logical partitioning correctly and automatically in an optimized manner, given a query workload.
3. We design a prototype to implement (based on Hive [2]) the logically partitioning scheme so that only the related partitions are scanned without changing Hadoop internals.
4. We conduct experiments to compare the implemented prototype with the state of art approaches, using the benchmark data from TPC-H [3]. The experimental results show that our prototype implementation outperforms RCFile by 20% to 40% on query processing time.

The rest of the paper is organized as follows. First we introduce the background and motivation of our work in Section 2 and Section 3, respectively. Next, we present the design of our partitioning scheme in Section 4.After that, we evaluate the performance of the developed prototype in Section 5, and review the related works in Section 6. Finally Section 7 concludes the paper.

2 Background

To provide a technical background of this paper, we give an overview of two systems, namely Hadoop and Hive, and an important concept used in this paper: logical partition.

2.1 Hadoop and Hive

The Hadoop system mainly contains two components, namely HDFS and MapReduce. Before processing the input files, Hadoop needs to distribute the files randomly and evenly across a cluster of commodity machines. The files are then maintained on HDFS with data blocks of 64 MB by default. Next, to initiate a data query task, Hadoop starts map tasks (mappers) and reduce tasks (reducers) concurrently on the clustered machines. The mappers sequentially read the input files from HDFS with each data block as a unit and shuffle the intermediate results to the reducers. When the jobs are finished, the reducers write the output back to HDFS.

To support queries, Hive provided an open-source data warehousing solution atop Hadoop. To process the queries written by an SQL alike declarative language, Hive compiles the queries into MapReduce jobs that are executed by Hadoop. Besides, Hive provides commands and third party application interfaces, such that users can execute queries in a flexible way.

2.2 Logical Partition

Hive supports the concept of logical partition. That is, for a given table, the data belonging to the same logical partition is placed into one directory on HDFS. However, the files are physically distributed onto multiple nodes in a cluster. For example, Hive uploads the table raw files onto HDFS by executing MapReduce jobs, and then splits the raw files into N partitions. Next, the N partitions are evenly placed onto M datanodes of a Hadoop cluster.

3 Motivation

For a given table T in data warehouse, in the simplest case, we assume that only one selection query refers to T. We can directly select all rows from T satisfying the query condition into a partition, and let all remaining rows into another partition. Intuitively, by treating the query condition as a sub-range R in the universe U, the partitioning scheme is to separate the universe U into the sub-range R and the opposite $U \setminus R$.

In the general case, there instead could exist multiple queries referring to the table T. As an example of the general case, three following queries refer to the table *Lineitem* in the TPC-H [3] benchmark (we will use the queries throughout this paper).

```
Q1: SELECT l_returnflag, count(1),  sum(l_extendedprice*(1-l_discount))
    FROM Lineitem
    WHERE l_shipdate>‘1997-09-02’
    GROUP BY l_returnflag;

Q2: SELECT sum(l_extendedprice*l_discount) as revenue
    FROM Lineitem
    WHERE l_shipdate >= ‘1994-01-01’ AND l_shipdate < ‘1995-01-01’ AND
          l_discount >= 0.05 AND l_discount <= 0.07;

Q3: SELECT l_shipmode, count(1)
    FROM Lineitem
    WHERE l_shipmode = ‘MAIL’ OR l_shipmode = ‘SHIP’
    GROUP BY l_shipmode;
```

Given the general case, the key of the proposed partitioning scheme is to generate a best partition plan with the minimal query processing time. Therefore, we may independently consider each of the queries, and then follow the above simplest case (with only one query) to partition the input table T. However, the approach could lead to too many partitions and MapReduce jobs during processing queries and still cannot guarantee the minimal query processing time.

To this end, we need to carefully find a best way to partition the table T for the minimal query processing time. Here, we will define a *partition plan* that is used to partition T (Section 4.1). Given multiple queries, each of the queries might involve an associated plan to partition the table T, and the whole queries involve the various combinations of such plans (Sections 4.2). Consequently, based on a cost model, we find the best one with the least cost among all possible candidates for the minimal processing time (Section 4.3).

4 Design of CPS Partitioning Scheme

4.1 Definition of Partition Plan

For a given table T with schema T^s, we consider a set of queries over T. Such queries involve the number n of query conditions θ_i with $1 \leq i \leq n$. Based on such n conditions, we then define a partition plan $P = \{p_1, p_2, ..., p_n\}$ if and only if the conditions θ_i associated with p_i satisfies the requirement $\theta_1 \vee \theta_2 \vee ... \vee \theta_n = True$. In general, partition p_i is determined by an associated condition θ_i, such that the data of p_i satisfies the condition $\theta_i = True$. When p_i are ready, to process a query having multiple conditions, we will consider only those partitions corresponding to such conditions for query evaluation.

Based on the three queries Q_1, Q_2 and Q_3, we give an example of a valid partition plan in Table 1. Just for simplicity, we only consider the predicate conditions referring to the attribute *l_shipdate*. As shown in this table, each partition has an unique ID, and the data in each of the partitions is determined by selecting the rows from *Lineitem* satisfying the associated conditions.

Table 1. Example of Partition Plan for *Lineitem*

UID	Associated Conditions
0	*l_shipdate* < '1994-01-01'
1	*l_shipdate* ≥ '1994-01-01' and *l_shipdate* < '1995-01-01'
2	*l_shipdate* ≥ '1995-01-01' and *l_shipdate* ≤ '1997-09-02'
3	*l_shipdate* > '1997-09-02'

4.2 Generate Candidate Plans

We note that the query conditions could involve disjunctive and (or) conjunctive forms of individual predicates. Thus, we need to transform the disjunctive and (or) conjunctive combinations of such predicates. The purpose of such transformations is to simply the query combinations and meanwhile should not falsely miss any query results needed by the queries (and thus should correctly answer the queries without falsely missing any results).

With equivalent transformations in logical theory, we can transform query conditions into their the major conjunctive normal form (CNF), and each element in CNF refers to one attribute. Based on all such elements, we then use the algorithm in Section 4.2 to generate the candidates of partition plans.

Before transforming the predict conditions in queries, we first formally define the *candidate partition plan* as follows. For the table schema T^s, the candidate partition plan about an attribute a contains all selective conditions involving a and their corresponding negative conditions. To avoid overlapping conditions, a candidate plan satisfies $\forall\theta_i,\ \theta_j \in \Theta,\ \theta_i \wedge \theta_j = \emptyset$.

Next, we consider the following typical data warehouse query, and would like to find out all the candidate plans.

$$SELECT\ \mathcal{A}\ FROM\ T\ WHERE\ \mathcal{P}$$

where $\mathcal{A}$ denotes an aggregate function over attributes of table T, and $\mathcal{P}$ denotes a set of predicates in the query conditions. Denote $\pi(\mathcal{P})$ to be the attributes appearing in the conditions. For each query in given workload, we can extract the attributes set and union all the sets to figure out the attributes appearing in the queries. Then, for every attribute, we combine the query conditions refer to it. Finally, we figure out a candidate plan for each attribute as shown in Table 2.

4.3 Cost Model of Partition Plan

In this section, we give a cost model to measure the cost of each plan. Consider table $T = (a_1, a_2, ..., a_k)$ where a_i with $1 \leq i \leq k$ is the i-th attribute of T and $M(T)$ is the size of T. Suppose we have a candidate plan $P = \{p_1, p_2, ..., p_n\}$ determined by attribute a and the corresponding condition set $\phi_T^a = \{\theta_1, \theta_2, ..., \theta_n\}$. Denote $r(\theta)$ is the selective ratio of T, when filtering T according to θ. Then, the size of the i-th partition of table T is

$$S(T, i) = r(\theta_i) \times M(T) \tag{1}$$

Table 2. Candidate Partition Plans of Demo Workload

Attribute	UID	Corresponding Condition
l_shipdate	0	l_shipdate > '1997-09-02'
	1	l_shipdate <= '1997-09-02' AND l_shipdate >= '1995-01-01'
	2	l_shipdate >= '1994-01-01' AND l_shipdate < '1995-01-01'
	3	l_shipdate < '1994-01-01'
l_discount	0	l_discount < 0.05
	1	l_discount >= 0.05 AND l_discount <= 0.07
	2	l_discount > 0.07
l_shipmode	0	l_shipmode = 'MAIL'
	1	l_shipmode = 'SHIP'
	2	l_shipmode <> 'MAIL' AND l_shipmode <> 'SHIP'

Next, we use the function $f(T, i, q)$ to determine whether or not the i-th partition is hit by q, where $\pi(q)$ is the set of attributes in *where* clause of q and c_q is the where clause of q:

$$f(T, i, q) = \begin{cases} 1, & a \notin \pi(q) \\ 1, & c_q \cap \theta_i \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

In addition, if aggregation operations existing in query, there incurs two MapReduce jobs to figure out query result, the first one executes selection operation and the other one processes its result for aggregation. We accumulates the input size and use function $g(q)$ to determine whether q having aggregations:

$$g(q) = \begin{cases} 1 + r(q), & \text{if } q \text{ contains aggregation functions} \\ 1, & \text{otherwise} \end{cases} \tag{3}$$

Then, given the query set $Q(T)$ contains all the queries refer to T in current workload, the cost of fetching input data of T is

$$C(T) = \sum_{\forall q \in Q} \sum_{1 \leq i \leq n} S(T, i) \times g(q) \times f(T, i, q) \tag{4}$$

With the above cost model, we then can calculate the cost of each candidate plan, and the one with the least cost is chosen as the final plan.

5 Experiments

We evaluate the performance of CPS on three aspects: (i) the usefulness of our cost model, (ii) the advantages of CPS over previous works.

We conduct our experiments on a Hadoop cluster with one master node and 5 slave nodes. Each node has the same hardware and software configuration with 64-bit Linux and 750 GB hard disk. We implement CPS service atop Hadoop

1.2.1 and Hive 0.11.0. Each mapper/reducer task uses 1024 MB memory and the block size is 64 MB by default. We use dbgen in TPC-H to generate the synthetic dataset and the scale factor is 30.

5.1 Usefulness of the Proposed Cost Model

In section 4.1, we generate 3 candidate partition plans for table *Lineitem*. Next, for each of these plans, we first use the cost model to compute the theoretical cost, and next compute the empirical results based on our experiments. Instead of precisely computing the theoretical cost, we alternatively use the size of the data used by the plans. Denote M to be the total size of an input table, and $r \cdot M$ then indicates the size of the data used by a plan to process the query.

In Table 3, for each of the candidate plans with respect to the rows *l_shipdate*, *l_discount* and *l_shipmode*, the 2nd - 4th columns indicate the data size used by each query from Q_1 to Q_3 (because of aggregation, the ratio of input size to table size maybe larger than 1), and the 5th column sums the data size. The rightmost column instead gives the real query time of executing all three queries. Though the result given by the proposed cost modal does not precisely indicate the absolute cost of each candidate plan, this table does verify that the empirical value (i.e., the real query time) is roughly consistent with the theoretical value computed by the cost model (i.e., the total cost in the 5-th column).

Table 3. Comparison with Theory Cost Values

Key Attribute	Q1	Q2	Q3	Total Value	Execution Time(s)
l_shipdate	$0.33 \cdot M$	$0.15 \cdot M$	$1.39 \cdot M$	$1.87 \cdot M$	101
l_discount	$1.17 \cdot M$	$0.07 \cdot M$	$1.39 \cdot M$	$2.63 \cdot M$	125
l_shipmode	$1.17 \cdot M$	$1 \cdot M$	$0.78 \cdot M$	$2.95 \cdot M$	141

5.2 Comparison between CPS and Previous Works

In this experiment, we compare CPS with two alternative approaches (a special file format RCFile and physically clustering). In detail, we first use RCFile as the typical example of optimize the design of file format. Second, we purposely move the data blocks belonging to the same (logical) partitions onto the same physical nodes, such that we emulate the physical clustering approach (CoHadoop). Finally, we combine the two techniques (the RCFile and the CPS partitioning scheme) together in order to provide the utmost performance. In this experiment, we extract 12 selection queries[1] about table *Lineitem* in TPC-H and processing queries on those tables. Based on these approaches, we conduct the experiment by 10 times and then measure the average processing time given in Figure 1(a).

[1] We choose all the selections about *Lineitem*, including the selections in *Join* query by ignoring the operations on other tables.

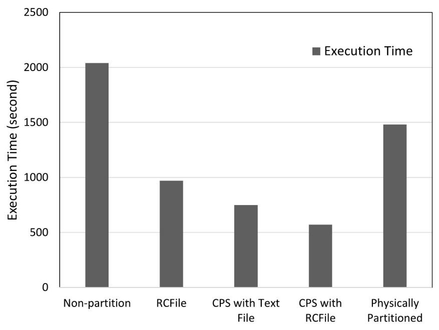

Fig. 1. Comparison of Different Optimization Policy

As shown in this figure, the case without any partition obviously uses the longest time. Next, compared with RCFile, the CPS partition scheme saves about 20% time. The physical clustering method has the longer execution time than RCFile does. Instead, the combination of the two techniques (i.e., RCFile and CPS) can achieve the best result by saving around 40% time. This experiment clearly verifies the advantages of CPS over the previous works, and in particular, the combination of CPS and RCFile achieves the best result among all approaches.

6 Related Work

First, CoHadoop extended Hadoop by placing data blocks (and their replicas) of correlated files onto the same data nodes. In this way, the correlated files can be accessed locally inside the same machine, and the network traffic is reduced. Unfortunately, this approach could lead to unbalanced workloads in clustered machines, in case that the correlated data is associated with very high (or very low) volume size. Different from CoHadoop, some previous works specially designed file formats to optimize Hadoop query processing time. The typical examples of such works include *CFile*, [6] and *RCFile*. *CFile* supported a column-wise file format based on the entire file level, while *RCFile* was an in-block column-wise storage. The column-store allowed faster query processing time by less I/O cost. However, these data formats do not cluster semantically related data together.

7 Conclusion

In this paper, we proposed a new partition scheme, CPS, to optimize query processing, based on the Hive data warehouse and the Hadoop distributed file system. Our experimental results have successfully verified that the CPS scheme can improve the performance of query processing over the state of the art methods. As future work, we will extend our partition scheme to support more complex query conditions, such as those having multiple attributes conditions, and the query operators like *Join* and *Order By*.

References

1. Hadoop, `http://hadoop.apache.org/`
2. Hive, `http://hive.apache.org/`
3. TPC-H, Benchmark Specification, `http://www.tpc.org/tpch/`
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
5. Eltabakh, M.Y., Tian, Y., Zcan, F., Gemulla, R., Krettek, A., McPherson, J.: CoHadoop: flexible data placement and its exploitation in Hadoop. Proc. VLDB Endow. 4(9), 575–585 (2011)
6. Lin, Y., Agrawal, D., Chen, C., Ooi, B.C., Wu, S.: Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In: SIGMOD Conference, pp. 961–972. ACM, New York (2011)
7. He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., Xu, Z.: RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: ICDE Conference, pp. 1199–1208 (2011)