Optimizing Queries over Partitioned Tables in MPP Systems

Lyublena Antova
lantova@gopivotal.com

Amr El-Helw
aelhelw@gopivotal.com

Mohamed A. Soliman
msoliman@gopivotal.com

Zhongxian Gu
zgu@gopivotal.com

Michalis Petropoulos
mpetropoulos@gopivotal.com

Florian Waas
flw@datometry.com

ABSTRACT

Partitioning of tables based on value ranges provides a powerful mechanism to organize tables in database systems. In the context of data warehousing and large-scale data analysis partitioned tables are of particular interest as the nature of queries favors scanning large swaths of data. In this scenario, eliminating partitions from a query plan that contain data not relevant to answering a given query can represent substantial performance improvements. Dealing with partitioned tables in query optimization has attracted significant attention recently, yet, a number of challenges unique to Massively Parallel Processing (MPP) databases and their distributed nature remain unresolved.

In this paper, we present optimization techniques for queries over partitioned tables as implemented in Pivotal Greenplum Database. We present a concise and unified representation for partitioned tables and devise optimization techniques to generate query plans that can defer decisions on accessing certain partitions to query run-time. We demonstrate, the resulting query plans distinctly outperform conventional query plans in a variety of scenarios.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing; Distributed databases*

Keywords

query optimization; MPP systems; partitioning

1. INTRODUCTION

It is hardly necessary to motivate the need for systems for processing large amounts of data. Big data and data-driven analysis is ubiquitous in all business verticals today, including government agencies, financial corporations, telcos, insurance and retail. From running simple reports to executing complex analytics workloads to find insights in data, these organizations are heavily investing in big data solutions, and have motivated the birth of multiple startup companies, as well as change in direction of many traditional database vendors to enable scalable and efficient processing of large amounts of data.

SIGMOD 2014 Snowbird Utah USA

http://dx.doi.org/10.1145/2588555.2595640 .

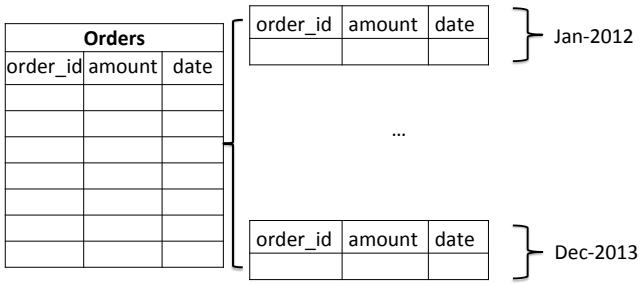


Figure 1: Table *orders* partitioned by *date*

Data partitioning is a well-known technique for achieving efficiency and scalability when processing large amounts of data. Partitioning comes in various shapes and forms. Many modern database systems partition the data in order to store and proccess it on different nodes to achieve parallelism [9, 11, 16]. Data can also be partitioned on a single machine, either horizontally by rows, or vertically by columns, to reduce the scans to acquire the data needed to asnwer a query. Some of the well-known benefits of data partitioning include: reduced scan time by scanning only the relevant parts, and improved maintenance capabilities: data can be loaded, indexed, or reformatted independently.

In this paper, we present techniques for optimizing complex queries to take advantage of partitioning and select only relevant partitions for scanning.

A common scenario that most of our customers use is to partition the data chronologically based on a date or timestamp field. Consider the example in Figure 1.

It shows an example table *orders* containing data for the past 2 years, which is partitioned into monthly partitions. The idea behind this partitioning scheme is that if a query specifies a range predicate on the partitioning key, it can avoid scanning redundant partitions that will not satisfy the predicate. One such query, which summarizes the orders from the last quarter is shown in Figure 2.

To evaluate this query, we need to scan only the last three partitions instead of all 24 that constitute the *orders* table. This technique is often referred to as *static partition elimination* and is implemented by nearly all systems supporting partitioning [9, 11, 16]. It consists of statically determining which partitions need to be scanned at query optimization time based on the predicate specified in the query.

```
SELECT avg(amount)
FROM orders
WHERE date BETWEEN '10-01-2013' AND '12-31-
2013'
```

Figure 2: Example query where non-relevant partitions can be eliminated statically

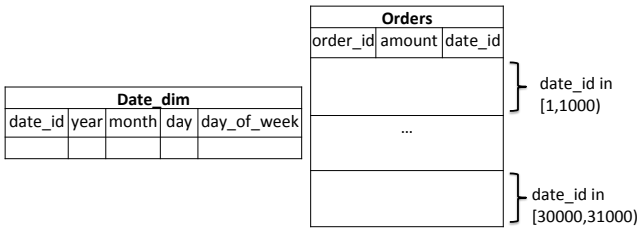


Figure 3: Fact table *orders* partitioned by *date_id*, foreign key in the dimension table *date_dim*

Often our customers model their data using a star schema design like the one presented in Figure 3. In this figure the fact tables *orders* has a foreign key to a separate dimension table *date_dim*. This model is a normalized form of the one in Figure 1 and allows for specifying additional properties of dates like "day of week". The fact table and dimension table can be partitioned independently. In our case the fact table is partitioned by the foreign key *date_id*. Queries on this schema often need to join the fact table with the dimension table, and static partition elimination is not always possible.

```
SELECT   avg(amount)
FROM     orders
WHERE    date_id IN
         (SELECT  date_id
         FROM     date_dim
         WHERE    year = 2013 AND
                  month BETWEEN 10 AND 12)
```

Figure 4: Example query where non-relevant partitions can only be eliminated at query time after evaluating part of the query

Figure 4 presents a rewriting of the original query from Figure 2, where now the *orders* table has to be joined with the dimension table to find the matching tuples. In this query the values for the partitioning keys of *orders* are not known beforehand, but are only determined at runtime after evaluating the subquery on the dimension table. This is known as *dynamic partition elimination* and most database systems either do not support it, or provide a very rudimentary support that works for simple queries and schema designs. See Section 5 for details. In addition to join-induced partition elimination, there are other cases where the partitioning key may not be known at optimization time, namely in the case of prepared statements with parameters. This case as well may require support for dynamic partition elimination, as parameter values are only provided at runtime.

In this paper we propose a novel approach for optimizing complex queries on partitioned tables. In particular, our contributions are:

- A model for representing partitioning and queries on partitioned tables with the following properties:
  - Abstract operators following the producer-consumer paradigm: a *PartitionSelector* determining which partitions need to be scanned (producer), and a *DynamicScan*, which consumes the partition ids produced by the PartitionSelector and performs the actual scan of those partitions.
  - Compactness of generated query plans: plan size independent of the total number of partitions in the tables, or the number of partitions that need to be scanned.
  - Support for both static and dynamic partition elimination in a unified way.
  - The approach is independent of the actual storage format of partitioned tables and can be applied to a variety of models for representing partitioning in the storage layer.
- Algorithms for partition elimination, which generate all *interesting* placements of PartitionSelector within a query plan.
- Support advanced partitioning schemes such as multi-level hierarchical partitioning.
- Performance experiments showing that the efficiency of plans generated by our optimizer over existing approaches. We are able to identify opportunities for partition elimination in complex queries, and achieve multi-fold improvement of the query runtime. The results from our in-lab experiments were also confirmed in early field trials of Pivotal's Greenplum Database, Pivotal's MPP datatabase, where some customers reported multi-fold improvement of query runtime due to advanced partition elimination techniques.

The rest of the paper is organized as follows. Section 2 presents our model for querying partitioned tables and describes algorithms for partition elimination. Section 3 presents the implementation of partition selection algorithms in Pivotal's MPP systems, and Section 4 demonstrates the efficiency of the approach. Finally, we review related work in Section 5, and conclude in Section 6.

2. OPTIMIZING QUERIES ON PARTITIONED TABLES

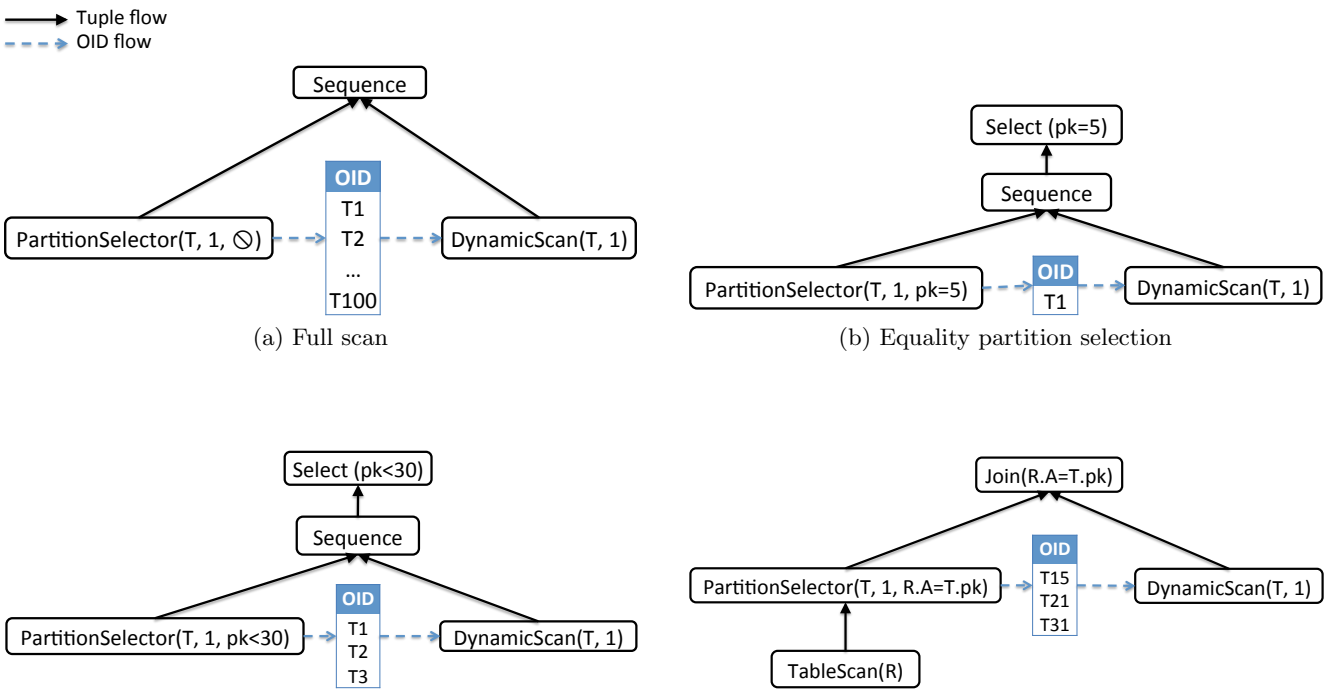
2.1 Definitions

A table T over schema $(A_1, ..., A_n)$ is a set of tuples $\{< a_1, ..., a_n >\}$. We say that a table T is *logically partitioned* on a key $pk \in \{A_1, ..., A_n\}$ into partitions $T_1, ..., T_m$ if there exists a partitioning function

$$f_T : p_k \mapsto \{T_1, ..., T_n, \perp\}$$

which is used to assign tuple t to a partition T_i or the invalid partition (denoted by $\perp$) based on the value of the partitioning key. The latter means that the tuple cannot be mapped to any partition. For a partitioned table T we will denote the set of its partitions as $\mathcal{P}(\mathcal{T})$.

Note that the partitions $T_1, ..., T_m$ need not be materialized on disk. For simplicity we assume that given a logical partition object id (OID) T_i the storage layer can locate and retrieve the tuples belonging to that partition.



(a) Full scan

(b) Equality partition selection

(c) Range partition selection

(d) Join partition selection

Figure 5: Implementing different scan patterns using various PartitionSelectors

In most systems the partitioning function f_T implements either a categorical, or range partitioning scheme. We also assume the existence of a function f_T^*:

$$f_T^* : \phi(pk) \mapsto \{..., T_{i_1}, ...\} \subseteq \mathcal{P}(\mathcal{T})$$

which for a given predicate $\phi(pk)$ on the partitioning key, returns a set of partition OIDs $\{T_{i_1}, .., T_{i_m}\}$ such that if a tuple $t \notin \{T_{i_1}, .., T_{i_m}\}$ then t does not satisfy ϕ. In other words, function f_T^* performs partition selection for a given predicate on the partitioning key, returning all partitions that *may* satisfy the given predicate ϕ. Note that such function always exists, as it can trivially return $\mathcal{P}(\mathcal{T})$. Also, for predicates of the form $pk = c$, the function f_T^* is the same as the partitioning function f_T applied to the value c. Ideally, the partition selection function should return the minimal set of partition OIDs for which the given predicate is satisfied, as it will be the basis for partition pruning described next. In Section 3 we discuss how f_T^* can be implemented in a commercial DBMS to handle complex predicates on the partitioning key.

2.2 Query Model for Partitioned Tables

To implement scans over partitioned tables, we extend the set of physical query operators by two new constructs: *PartitionSelector* and *DynamicScan*. Those two operators come in pairs and implement a producer-consumer model, where the PartitionSelector computes OIDs of partitions for the DynamicScan, which in turn is responsible for retrieving tuples belonging to those partitions. The PartitionSelector can communicate the partition OIDs to the DynamicScan via shared memory or any other communication channel known in the literature.

As a first example, consider a simple full table scan query over a partitioned table. Let T be a partitioned table with partitions $T_1, ..., T_{100}$, where partition T_i holds tuples for which $pk \in [(i-1) * 10 + 1, i * 10)$. Figure 5(a) shows the query plan implementing a full scan over T. Note the PartitionSelector and the DynamicScan on the left and right side of the plan, respectively. The PartitionSelector produces all child partition OIDs $T_1, ..., T_{100}$, shown in the table in the middle, and sends them for consumption to the DynamicScan, denoted by the dotted lines in the figure. At the root of the plan tree is a Sequence operator ensuring the PartitionSelector is executed before the DynamicScan.

By varying the shape of the PartitionSelectors and their placement in the query plan, we can implement more complex patterns, such as partition selection based on equality and range predicates, as well as dynamic partition elimination. Figures 5(b) and 5(c) show the query plans for equality and range selection over T, respectively. Here, the PartitionSelector is annotated with the partition-selecting predicate from the query, and the selected OIDs in the table in the middle of these figures only include the OIDs of partitions that may satisfy the selection predicate. Finally, Figure 5(d) shows a query plan that accomplishes dynamic partition elimination for a join query. Note that the PartitionSelector is on the opposite side of the DynamicScan and its predicate $R.A = T.pk$ refers to values from table R. When the outer (left) side of the join is executed, tuples from R will be streamed into the PartitionSelector, where the partition selection function will be applied to choose those partitions determined by the value of $R.A$. In this case, no Sequence operator is necessary, as the Join operator enforces the left-to-right order of execution of children.

Below we give informal definition of the three new operators PartitionSelector, DynamicScan, and Sequence.

PartitionSelector. An operator that is annotated with a partition table OID T, partScanId identifier, and an optional

predicate on the partitioning key of T. The PartitionSelector is an operator with side effects: based on the OID T and the given predicate, it computes all child partition OIDs which satisfy the predicate using the function f^*_T defined in Section 2.1. It then pushes them to the DynamicScan with the same partScanId. The partScanId identifier is used to identify the different (PartitionSelector, DynamicScan) pairs in case the query refers to multiple partitioned tables, or to multiple instances of the same partitioned table. If no predicate is specified, the PartitionSelector retrieves all child OIDs of T. A PartitionSelector may have at most one child. If it has a child, the output of the operator is the same as that of its child, otherwise no output is produced.

DynamicScan. An operator that is responsible for scanning tuples from a partitioned table T. It is annotated with the partitioned table OID and a partScanId. The DynamicScan operator consumes the partition OIDs provided by the PartitionSelector with the same partScanId and retrieves the tuples from those partitions.

Sequence. An operator that executes its children in order and returns the result of the last child.

Our query model can be generalized by abstracting DynamicScan as a table-returning function that produces a set of rows from partitioned table based on a (run-time) parameter. Such abstraction can be used to describe several proposals in the literature [9, 11] where partition selection and scanning are done by different operators. For example, an Index-Join implementation of this model performs partition selection by the outer child of the join which computes the keys of partitions to be scanned (the parameter values), while the inner child of the join performs partition scanning by looking up an index defined on partition key.

A key point that needs to be addressed when implementing this model in MPP databases is that partition selection and scanning operators could be running within processes/threads in different machines. We describe in Section 3 our approach to address such challenge.

2.3 Placement of PartitionSelectors

In this section we describe the algorithm which computes the placement of PartitionSelectors given a physical operator tree with DynamicScans. Often, there are multiple ways to place PartitionSelectors in the expression tree, but not all placements achieve optimal partition elimination as we will see next. We will use as a running example the query in Figure 6 which selects all sales records placed in the last quarter in California. The tables date_dim and sales_fact are partitioned on month and date_id, respectively.

```
SELECT *
FROM sales_fact s, date_dim d customer_dim c
WHERE d.month BETWEEN 10 AND 12 AND
c.state='CA' AND d.id=s.date_id AND c.id=s.cust_id;
```

Figure 6: Query selecting all sales records from the last quarter

The input to our algorithms is an expression tree, produced by the query optimizer, which has DynamicScan operators for scanning the partitioned tables, but no PartitionSelectors have been placed yet. A possible expression tree for the example query of Figure 6 is given in Figure 8(a). The goal of the algorithm is to find optimal placement of PartitionSelectors, where optimality is determined with respect to the minimum number of parts that need to be scanned. For this example, the result of the algorithm is shown in Figure 8(b). Note the two PartitionSelectors in the resulting plan. The lower one with ID 1 implements partition elimination for the DynamicScanof date_dim, while PartitionSelector 2 implements dynamic partition elimination for sales_fact using values from the selection on date_dim. Another possible PartitionSelector placement is to push PartitionSelector 2 on the inner side of the join. However, no partition elimination will be done for the latter query plan. Note also that the DynamicScan and corresponding PartitionSelector do not have to be immediate children of the same node: in Figure 8(b) they are separated by multiple levels in the plan tree. This shows the versatility of approach in optimizing complex queries on partitioned tables.

For simplicity, we present the recursive implementation of the algorithm. We show in Section 3.1 how the algorithms can be implemented to work on a compact representation of the plan space as opposed to a complete operator tree. Note also that the algorithms for PartitionSelector placement are orthogonal to data distribution, where base tables are hash distributed and placed onto different physical nodes of the cluster. Section 3 provides details on how partitioning and distribution can be combined into one system.

Algorithm 1: PlacePartSelectors

```
Input  : List inputPartSelectors, Expression expr
Output: Expression where all partition selection has been
        enforced
1 List partSelectorsOnTop;
2 List childPartSelectors;
3 expr.operator.ComputePartSelectors(inputPartSelectors,
  partSelectorsOnTop, childPartSelectors);
4 List newChildren;
5 foreach child in expr.children, childPartSelectorList in
  childPartSelectors do
6 |   Expression newChild = PlacePartSelectors(child,
  |   childPartSelectorList);
7 |   newChildren.Add(newChild);
8 end
9 return EnforcePartSelectors(partSelectorsOnTop,
  expr.Operator, newChildren);
```

PartSelectorSpec	
partScanID:	int
partKey:	ColRef
partPredicate:	Expression

Figure 7: PartSelectorSpec

The main function is shown in Algorithm 1. It accepts an input expression *expr* and a list of input *PartSelectorSpecs*, and returns an expression where all PartitionSelectors have been placed. Each PartSelectorSpec is a compact specification of the PartitionSelector operator which needs to be placed for each unresolved DynamicScan. Its structure is shown in Figure 7 and contains the partScanId identifying the DynamicScan, the partitioning key, and optionally a predicate on the partitioning key that can be used for partition elimination. Initially, the partPredicate is NULL, but as

we push PartitionSelectors through the different operators, it may get augmented, as shown in the following subsections. The input PartSelectorSpecs for the *PlacePartSelectors* function are initialized by traversing the tree and identifying all DynamicScans that need corresponding PartitionSelectors. This input list is first passed to the function *ComputePartSelectors*, the main driver of the computation (line 3). This function is overloaded for each operator type, and computes which PartitionSelectors need to be placed on top of the current operator (in the output list *partSelectorsOnTop*), and which need to be pushed down to the children (output list *childPartSelectors*). After the two lists have been computed in Line 4 of Algorithm 1, the algorithm recursively pushes the necessary PartitionSelectors to the child nodes (Lines 6 and 7), and places the rest of the PartitionSelectors on top (Line 9).

The following subsection show the implementation of *ComputePartSelectors* for several of the partition-eliminating operators, including Join (Section 2.3.3), Select (Section 2.3.2), and a default implementation for non-partition eliminating operators (Section 2.3.1). Listed below are the helper functions used in the algorithms presented next:

- **EnforcePartSelectors**: Given an operator and a list of PartitionSelectors which need to be enforced on top, constructs a new expression tree where the PartitionSelectors have been placed on top
- **Operator::HasPartScanId**: Checks whether the DynamicScan with the given part scan id is in scope of the expression tree rooted at the current operator
- **FindPredOnKey**: Given a scalar expression, extracts a predicate referring to the given column
- **Conj**: Construct a conjunction of the given predicates

2.3.1 Default PartitionSelector Placement

The default implementation of *ComputePartSelectors* is presented in Algorithm 2 and is used for operators do not have a partition-filtering predicate, such as GroupBy, Union, Project, etc. It simply pushes the PartSelectorSpecs to the subexpression which defines the DynamicScan with the given partScanId (Line 8), or places it on top, if the DynamicScan is not defined in the subtree rooted by the operator (Line 3).

Algorithm 2: Operator::ComputePartSelectors

```
   Input  : List inputPartSelectors, List
            partSelectorsOnTop, List childPartSelectors
   Output: Compute default partition selectors for operators
 1 foreach partSpec in inputPartSelectors do
 2     if !this.HasPartScanId(partSpec.partScanId) then
 3         partSelectorsOnTop.Add(partSpec);
 4     end
 5     else
 6         foreach child operator op do
 7             i = order of op among children;
 8             if op.HasPartScanId(partSpec.partScanId)
               then
 9                 childPartSelectors[i].Add(partSpec);
10             end
11         end
12     end
13 end
```

Algorithm 3: Select::ComputePartSelectors

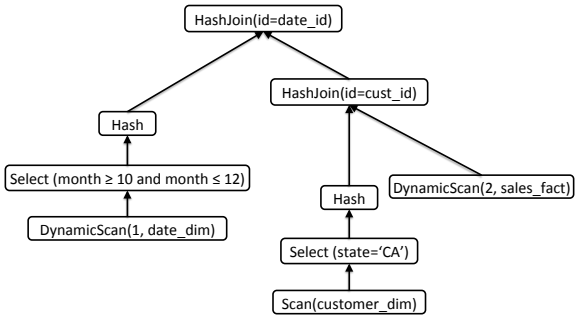
```
   Input  : List inputPartSelectors, List
            partSelectorsOnTop, List childPartSelectors
   Output: Compute partition selectors for Select operator
 1 foreach partSpec in inputPartSelectors do
 2     if !this.HasPartScanId(partSpec.partScanId) then
 3         partSelectorsOnTop.Add(partSpec);
 4     end
 5     else
 6         Expression partKeyPredicate =
           FindPredOnKey(partSpec.partKey,
           this.Predicate());
 7         if partKeyPredicate found then
 8             PartSelectorSpec newPartSpec = new
               PartSelectorSpec(partSpec.partScanId,
               partSpec.partKey, Conj(partKeyPredicate,
               partSpec.partPredicate);
 9             childPartSelectors[0].Add(newPartSpec);
10         end
11         else
12             childPartSelectors[0].Add(partSpec);
13         end
14     end
15 end
```

2.3.2 PartitionSelector Placement for Select

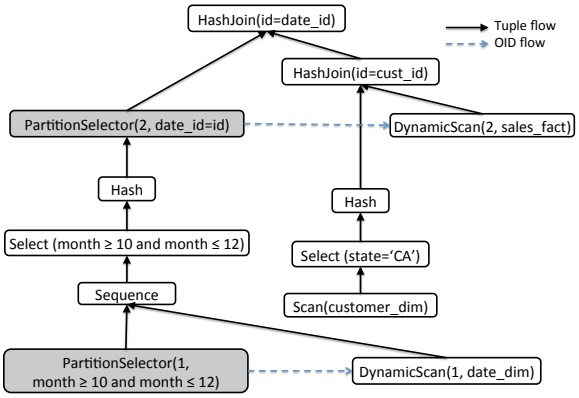
Algorithm 3 shows the implementation of *ComputePartSelectors* for the Select operator. For each input PartSelectorSpec, we check whether the corresponding part scan is defined in the subtree rooted by the Select operator (Line 2). If not, it is enforced on top of the Select operator. If it is defined below, we push the PartSelectorSpec to the child node, but before that we extract any partition-filtering predicates on the partitioning key, and add them to the PartSelectorSpec (Line 6-13) by first constructing a conjunction with any predicates passed from the top. An example run of the function for the example of Figure 8(a) is shown in Figure 8(c). DynamicScan with id 1 is defined in the subtree of the Select, so it is pushed further down to the child node. Also, since the Select's predicate involves the partitioning key *month*, the predicate is added to the PartSelectorSpec for the child node. The PartitionSelector for DynamicScan 2 needs to be resolved on top of the Select operator since it is not defined in the subtree.

2.3.3 PartitionSelector Placement for Join

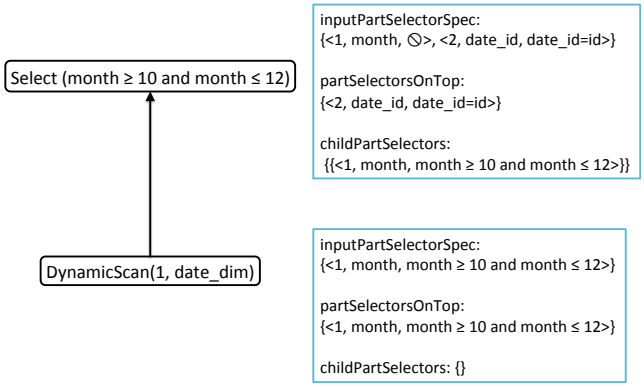
Algorithm 4 shows the implementation of *ComputePartSelectors* for the Join operator. Just like the default implementation of this method, and the one for Select, we start by checking whether a given part scan is in the scope of the subtree rooted at the Join operator (Line 2). If not, it needs to be enforced on top of the Join operator. Otherwise we probe to see if it is defined in the outer side. Recall that the DynamicScan is the consumer, and the PartitionSelector is the producer in our model. So



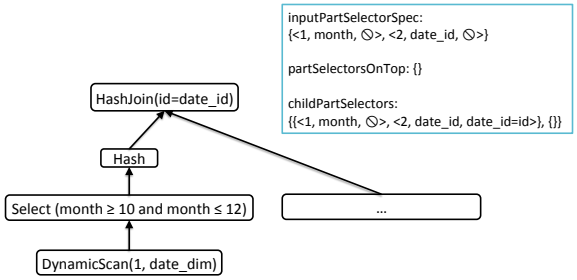
(a) Hash join query tree before PartitionSelector placement



(b) Hash join query tree after PartitionSelector placement



(c) Execution of *ComputePartSelectors* for Select



(d) Execution of *ComputePartSelectors* for HashJoin

Figure 8: Placing PartitionSelectors in an expression

if the DynamicScan is defined in the outer side of a Join operator, we cannot place the PartitionSelector on the inner side, as that destroys the order between the producer and the consumer. This check is implemented in Lines 7-8. If the DynamicScan is defined in the inner side, we check to see if the join predicate contains a condition on the partitioning key, which can be used for partition elimination. If the answer is yes, we push the PartSelectorSpec onto the outer side (Line 16), otherwise it needs to be resolved on the inner side, close to where the DynamicScan is defined.

An example run of the function for the example of Figure 8(a) is shown in Figure 8(d). DynamicScan with id 1 is defined in the outer side of the HashJoin, so the PartSelectorSpec is pushed further down to the outer child node. DynamicScan with id 2 is defined in the inner child, and the HashJoin's predicate restricts the partitioning key *date_id*. This means that we can use it to do partition pruning by pushing the PartSelectorSpec onto the outer side. The result of the function *ComputePartSelectors* for the join node is shown in the bottom part of the box in Figure 8(d). No PartitionSelectors are placed in the "on top" list, and none are pushed to the inner child of the hashJoin (noted by the {} in the childPartSelector list). Thus both PartitionSelectors for scan ids 1 and 2 will be placed on the join's outer side.

2.4 Multi-level Partitioned Tables

Most database systems also support hierarchical partitioning, in which a table is partitioned over multiple levels. Figure 9 depicts an example of such partitioning scheme. The *orders* table uses a 2-level partitioning. The first level uses the *date* column such that each partition contains the data for one month. These partitions are further split into subpartitions using the *region* column. A query on this table can specify particular date ranges, regions, or both. The system can use these conditions to avoid scanning partitions that will yield no results.

In order to support multi-level partitioning, the structures and algorithms explained earlier need to be extended. The PartitionSelector defined in Section 2.2 has to be extended so that it is annotated with a list of optional predicates instead

Algorithm 4: Join::ComputePartSelectors

```
   Input  : List inputPartSelectors, List
            partSelectorsOnTop, List childPartSelectors
   Output: Compute partition selectors for join
 1 foreach partSpec in inputPartSelectors do
 2     if !this.HasPartScanId(partSpec.partScanId) then
 3         partSelectorsOnTop.Add(partSpec);
 4     end
 5     else
 6         Expression partKeyPredicate =
           FindPredOnKey(partSpec.partKey,
           this.Predicate());
 7         bool definedInOuterChild =
           children[0].HasPartScanId(partSpec.partScanId);
 8         if definedInOuterChild then
 9             childPartSelectors[0].Add(partSpec);
10         end
11         else if partKeyPredicate not found then
12             childPartSelectors[1].Add(partSpec);
13         end
14         else
15             PartSelectorSpec newPartSpec = new
               PartSelectorSpec(partSpec.partScanId,
               partSpec.partKey, Conj(partKeyPredicate,
               partSpec.partPredicate);
16             childPartSelectors[0].Add(newPartSpec);
17         end
18     end
19 end
```

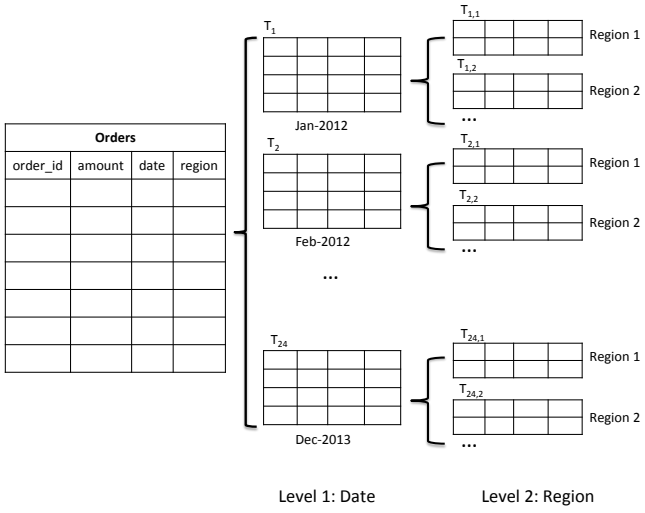


Figure 9: Multilevel partitioning by *date* and *region*

of a single optional predicate. Based on the OID T and the given predicates, it computes all leaf-level child partition OIDs that satisfy the predicates. For example, using the *orders* table in Figure 9, we show in Figure 10 a number of possible predicates given to the PartitionSelector, and the child partition OIDs it computes in each case.

The PartSelectorSpec from Figure 7 has to support a list of partKeys and a list of predicates as shown in Figure 11. The number of items in both lists have to be equal, and be the same as the number of partitioning levels for the table

partPredicate	**Partition OIDs**
date='Jan-2012'	$T_{1,1}, T_{1,2}, ..., T_{1,n}$
region='Region 1'	$T_{1,1}, T_{2,1}, ..., T_{24,1}$
date='Jan-2012' AND region='Region 1'	$T_{1,1}$
ϕ	*all leaf part OIDs*

Figure 10: Multi-level partition selection

PartSelectorSpec	
partScanID:	int
partKeys:	List<ColRef>
partPredicates:	List<Expression>

Figure 11: Extended PartSelectorSpec

on which this PartSelectorSpec is defined. Note that some elements of the partPredicates list may be empty, indicating the absence of a predicate on the corresponding partKey.

The only change to the algorithms is that the *FindPredOnKey* function used in Algorithms 3 and 4 must be extended to take a list of partKeys, and return a list of predicates corresponding to these keys. This function returns NULL if no partition-filtering predicates are found.

3. IMPLEMENTATION

Greenplum Database (GPDB) is Pivotal's massively parallel relational database system. HAWQ [12] is Pivotal's SQL engine for Hadoop. Both Pivotal products use Orca [15], a new query optimizer designed specifically to support large scale analytic processing in different computing architectures. We describe the implementation of partition selection inside Orca in Section 3.1. We then describe the runtime environment in Section 3.2.

3.1 Query Optimization

In an MPP system, data can be distributed to different hosts or physical machines. During query execution, the distribution of intermediate results can be enforced in multiple ways including *hash* distribution, where tuples are distributed to hosts based on some hash function, *replicated* distribution, where a full copy of a table is stored at each host and *singleton* distribution, where the whole distributed table is gathered from multiple hosts to a single host. Data distribution is orthogonal to partitioning. That is, a distributed table can also be partitioned on each host.

A query plan can enforce a particular data distribution through special Motion operators. During query execution, a Motion operator acts as the boundary between two active processes sending/receiving data and potentially running in different hosts. This constrains the shape of valid plans that can perform partition selection, since we rely on shared memory to accomplish the required communication between PartitionSelector and DynamicScan operators (cf. Section 2.2). Specifically, the optimizer must guarantee that a pair of communicating PartitionSelector and DynamicScan operators run within the same process. This means that no Motion operator can exist between PartitionSelector, DynamicScan and their lowest common ancestor.

Figure 12 shows examples of valid and invalid plans based on the previous constraint. In general, it is not straightforward to constrain the shape of arbitrarily complex plans,

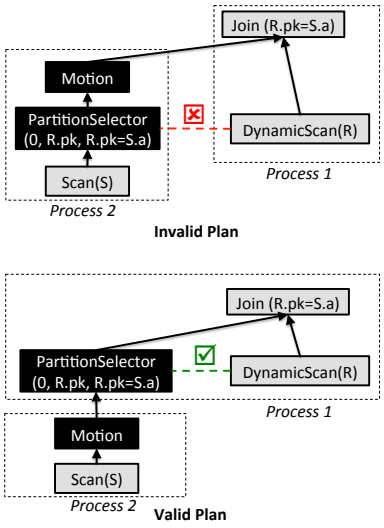


Figure 12: Interaction of Motion and PartSelector

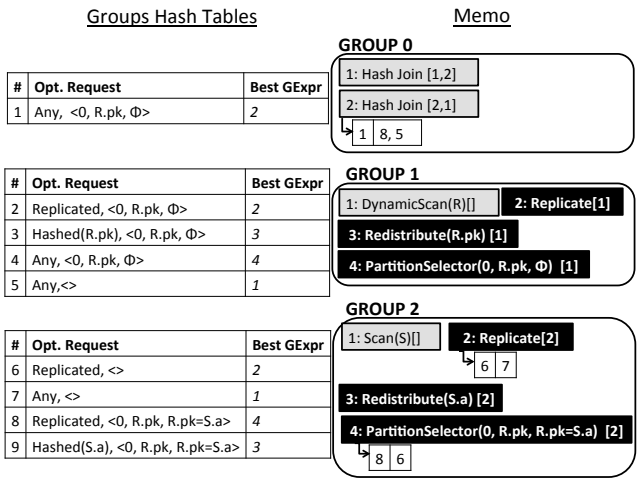


Figure 13: Partial Memo

where PartitionSelector and DynamicScan appear in distant sub-trees. We next show how the query optimizer handles this requirement in a principled way.

Orca is the new Pivotal query optimizer based on the Cascades optimization framework [6]. Orca compactly encodes optimizer's search space in the Memo structure, which is composed of a set of containers called *groups*, where each group contains logically equivalent expressions called *group expressions*. Each group expression is an operator that has other groups as its children. This recursive Memo structure allows compact encoding of a very large space of plan alternatives. Figure 13 shows an example Memo. The dependencies between operators are encoded using group references. For example, HashJoin[1,2] in Group 0 is a group expression that has two child groups 1 and 2.

We describe optimization of queries on partitioned tables in Orca using the following simple example:

```
SELECT * FROM R, S WHERE R.pk=S.a
```

where R and S are hash distributed tables, R is a partitioned table with part key R.pk and S is a non-partitioned table.

Orca has an extensible property enforcement framework that models requirements such as data distribution and partition selection as physical properties. A given plan may either satisfy a physical property on its own (e.g., a hash-distributed table delivers hash-distributed data), or an enforcer operator (e.g., Motion) needs to be plugged in the plan to deliver the required property. The PartitionSelector operator is the enforcer of partition selection property.

Figure 13 shows a partial Memo structure for the previous query. We distinguish enforcer operators using black boxes. Enforcer's child belongs to the same group containing the enforcer itself. For example, 'Redistribute(R.pk)[1]' is an enforcer that delivers Hashed(R.pk) distribution using a child operator in Group 1. Similarly, 'Replicate[2]' is an enforcer that delivers Replicated data distribution using a child operator in Group 2. Adding multiple enforcers to the same group allows considering different permutations of enforced properties, while discarding the invalid ones, as we show next.

Optimization starts by specifying required properties in an initial optimization request submitted to the Memo group that corresponds to the root of the query expression. An optimization request r submitted to Memo group g is computed by finding the plan satisfying r, rooted by an operator in g and has the least estimated cost. We call the root operator of such a plan the best group expression (GExpr) of request r.

For each incoming optimization request to g, each group expression in g creates corresponding requests to child groups based on operator's local requirements. Figure 13 shows *groups hash tables*, where each incoming optimization request is cached and is associated with its best GExpr. The small tables below some group expressions show the relevant mappings between incoming requests and child requests, as we describe next.

In our example query we assume the initial optimization request is *req. #1*: {Any, <0, R.pk, ϕ>} in Group 0, which specifies that results can have Any data distribution but must select partitions for table 0, which is R. When optimizing the HashJoin operators in Group 0 for this request, we need to consider additional local requirements to co-locate tuples to be joined based on the condition R.pk=S.a. This can be accomplished in multiple ways including: (1) requesting one child to be Replicated and the other child to have Any distribution, and (2) requesting S distribution to be Hashed(S.a) and R distribution to be Hashed(R.pk). For partition selection, HashJoin requests using the interesting partition selection condition R.pk=S.a in its left child, because of the implicit execution order of join children (left to right). These different optimization alternatives are enumerated and encoded using different optimization requests, as shown in Figure 13.

Valid plans are encoded in the Memo by maintaining, for each operator, a mapping of computed optimization requests to the corresponding child optimization requests. For example, in Figure 13 the small table below PartitionSelector in Group 2 indicates that the best plan rooted by PartitionSelector for *req. #8* requires a child plan in Group 2 that satisfies *req. #6*. This plan is given by Replicate operator, which in turn requires a child plan satisfying *req. #7* which is given by Scan(S).

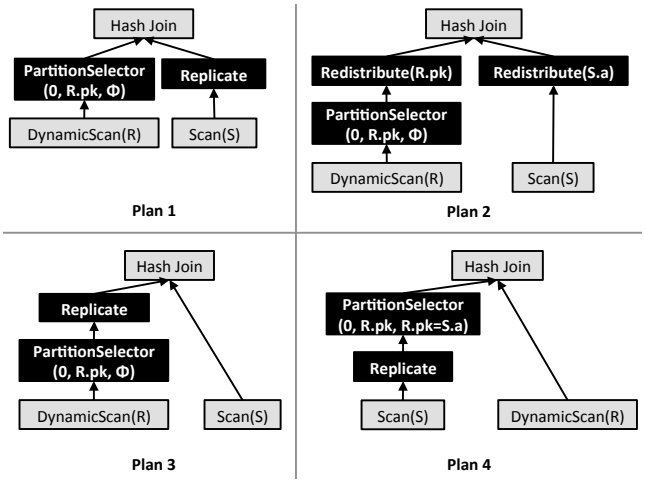


Figure 14: Partial plan space

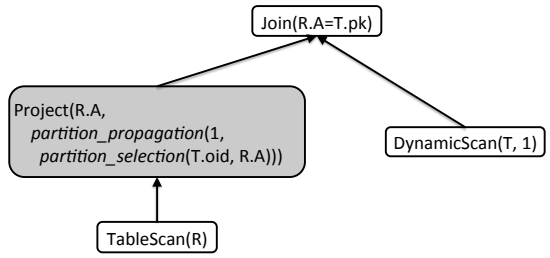
To illustrate, consider HashJoin[2,1] operator in Group 0. A pair of optimization requests are generated for the child groups: *req. #8*: {Replicated, <0, R.pk, R.pk=S.a>} for Group 2, and *req. #5*: {Any, <>} for Group 1. In order to satisfy *req. #8* using plans rooted by Replicate in Group 2, we cannot consider PartitionSelector as a child. The reason is that such a plan would make the communication between partition selection producer and consumer impossible, as we discussed earlier in the beginning of this section. On the other hand, when considering plans rooted by PartitionSelector in Group 2, we can consider Replicate as a child. This capability is implemented in Orca by allowing each operator to prohibit enforcing some properties on top of it. Whenever a plan alternative is considered, operator-specific logic is executed to guarantee that enforcers are plugged in the right order.

Figure 14 shows the partial plan space corresponding to the partial Memo in Figure 13. Plans 1, 2, and 3 are generated through optimization requests originating from HashJoin[1,2], while Plan 4 is generated through optimization requests originating from HashJoin[2,1]. The only plan that performs partition selection is Plan 4. This is possible at the cost of replicating S. Orca's cost model considers these different alternatives in order to pick the best plan.

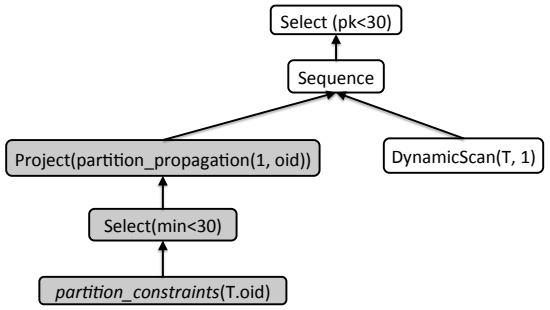
3.2 Runtime Environment

GPDB and HAWQ allow users to partition a table by specifying a range or categorical constraints for each partition. On disk, partitions are represented as separate physical tables, with associated check constraint to represent the corresponding constraint on the partitioning key. Note that each constraint can be written in the form $pk \in \cup_i (a_{i_1}, a_{i_k})$, where (a_{i_1}, a_{i_k}) is an open or closed interval, possibly open-ended. This representation also covers categorical partitioning, where the interval's start and end overlap.

To implement PartitionSelector operators, we use a combination of special-purpose built-in functions, and existing query operators to invoke these functions. The built-in functions retrieve metadata information about the partitioned table during query execution, and the query plan may have filters on top to select only those partitions which need to be scanned. Table 1 lists the built-in functions which implement partition selection functionality. The first three functions perform retrieval of the partitioning metadata in different forms, and the last one, *partition_propagation*, pushes a selected partition OID from the PartitionSelector to the DynamicScan with the given id.



(a) Implementation of equality-based partition selection



(b) Implementation of range-based partition selection

Figure 15: Implementation of partition selectors in runtime environments

Figures 15(a) and 15(b) show the implementation of the PartitionSelectors for equality and range-based partition selection from Figures 5(d) and 5(c), respectively. To implement equality-based partition selection (Figure 15(a)), we call the function *partition_selection* with the join parameter for the partitioning key ($R.A$), and pass the resulting OID to *partition_propagation*, which pushes it to the DynamicScan. To implement range-based partition selection (Figure 15(b)), we invoke the function *partition_constraints* which returns all child partitions together with their range constraints. The selection on top of the function call selects only those partitions whose range starts before the constant specified in the query, and the Project on top propagates the OIDs of tuples which pass the filter to the DynamicScan.

Note that in this approach "static" and "dynamic" partition selection are implemented in a uniform way. For example, to implement the "static" partition elimination from Figure 5(b), we need to invoke *partition_selection* with the constant from the query, whereas in Figure 15(a) we pass values from a table.

4. EXPERIMENTS

In this section, we present the experimental evaluation of our approach. The goal of our experiments is to show that: (1) our partitioning model does not introduce overhead in table scans, (2) our partition elimination algorithm is effective, and (3) our model guarantees the compactness and scalability of query plan sizes.

function	return type	description
partition_expansion(rootOid)	setof(OID)	set of all child partition OIDs for the given root OID
partition_selection(rootOid, value)	OID	OID of child partition containing the given value for the partitioning key
partition_constraints(rootOid)	(OID,min,minincl,max,maxincl)	set of child partition OIDs with their constraints
partition_propagation(partScanId, oid)	void	pushes the given partition oid to the DynamicScan with given id

Table 1: Built-in partition selection functions

#parts	Description	Overhead
42	each part represents 2 months	3%
84	partitioned monthly	3%
169	partitioned bi-weekly	1%
361	partitioned weekly	2%

Table 2: Partitioning *lineitem*

4.1 Setup

For our experiments, we use a cluster with four nodes connected with 10Gbps Ethernet. Each node has dual Intel Xeon eight-core Sandy Bridge processors at 2.7GHz, 64GB of RAM and twelve 600GB SAS drives in two RAID-5 groups. The operating system is Red Hat Enterprise Linux 5.5.

For partition elimination experiments, we use 256GB TPC-DS benchmarks [1] with partitioned tables. TPC-DS is an industry standard decision-supporting benchmark that consists of a set of complex business analytic queries. Its 25 tables, 429 columns and 99 query templates can well represent a modern decision-support system and is an excellent benchmark for testing query optimizers.

4.2 Overhead of Partitioning

In this section, we show the scalability of our approach with respect to the number of partitions. We accomplish that by using the simple query:

```
SELECT * FROM lineitem;
```

using the *lineitem* table from the TPC-H benchmark [2], with 7 years worth of data. We vary the number of partitions, using some common partitioning scenarios, as illustrated in Table 2. The table also shows the overhead introduced by each of these scenarios versus the case of using an unpartitioned *lineitem* table. We can see from the results that partitioning does not introduce significant overhead, and that the performance of the full scan query is stable regardless of the number of partitions. This makes partitioning a very scalable approach to improve performance.

4.3 Partition Elimination Effectiveness

In this section we use a subset of the TPC-DS benchmark [1] to demonstrate the effectiveness of our partition elimination approach. We compare Orca to the legacy query optimizer of GPDB (a.k.a. *Planner*). The workload used in this experiment contains queries from TPC-DS that reference partitioned tables: *store_sales, web_sales, catalog_sales, store_returns, web_returns, catalog_returns* and *inventory*. We ran the entire workload using both *Planner* and Orca. Table 3 shows a high-level classification of the workload based on partition elimination. In 80% of the queries, Orca's partition elimination algorithm is just is good as *Planner*. In 14% of the workload, Orca succesfully eliminates more partitions than *Planner*, resulting in reduced scanning cost. In 6% of the workload (the last 2 categories), Orca produced sub-optimal plans that do not eliminate partitions or eliminate fewer partitions than *Planner*. These sub-optimal plans are partly due to cardinality estimation errors or sub-optimal cost model parameters that need further tuning. We are actively investigating these issues and constantly improving Orca.

Figure 16 shows the number of scanned parts from each table, aggregated across the whole workload. As can be seen in the figure, the number of partitions scanned from each table using Orca is less than the number of partitions scanned using *Planner*. Orca's partition elimination approach eliminates up to 80% of the partitions, as is the case with the *web_returns* table.

Note that in the above experiment we do not report absolute running times, as Orca contains multiple improvements over the legacy *Planner*, which makes it hard to quantify the runtime benefits brought solely by partition selection across the two systems. A comprehensive performance study of Orca, including comparison with *Planner* and competitor systems is outside of the scope of this paper, and is presented in [15].

For the purposes of this paper, we conducted another experiment with the TPC-DS benchmark from above, in which we ran the same workload in Orca for two different configurations; in one case partition selection was enabled, and in the other case it was disabled, keeping all other parameters the same. Figure 17 shows the relative improvement for each query obtained as a result of enabling partition selection. The improvement is depicted as a percentage of the running time without partition selection, so that an improvement of 50% means the query ran in half the time, and the bigger the percentage, the higher the savings. One can see that across the board partition selection speeds up execution time in various degrees, both for short-running and long-running queries. More than half of the queries improved above 50%, and for over 25% of the queries the improvement was above 70%. Figure 17 also shows one large and two small outliers in the short-running, and the medium-to-long running query blocks, respectively, for which performance degraded when partition selection was turned on. We investigated these cases and found out that the outliers are caused by Orca picking a suboptimal plan for the partition-selection case, which we contribute to imperfect tuning of cost model parameters.

Category	Percentage
Orca eliminates parts, *Planner* does not	11%
Orca eliminates more parts than *Planner*	3%
Orca and *Planner* eliminate parts equally	80%
Orca eliminates fewer parts than *Planner*	3%
Orca does not eliminate parts, *Planner* does	3%

Table 3: Workload classification

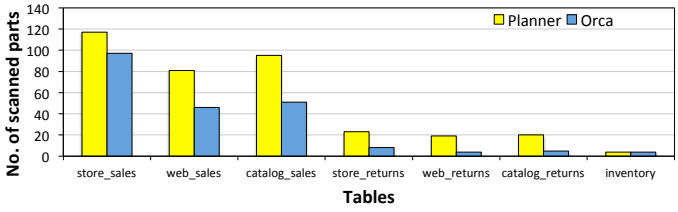


Figure 16: Partition elimination

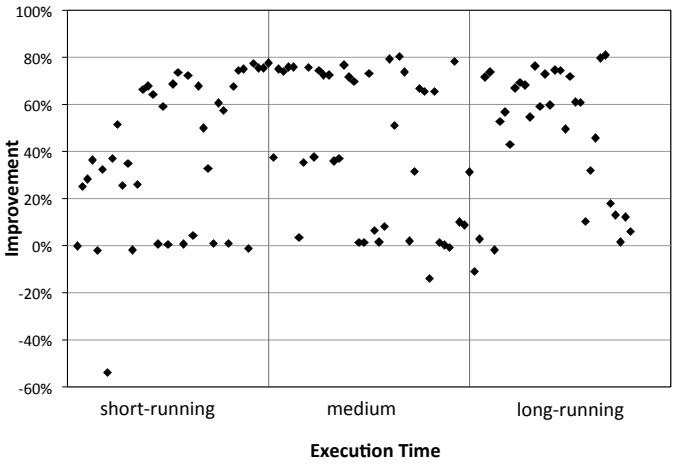


Figure 17: Relative improvement in execution time when partition selection is enabled

4.4 Plan Size Comparison

We performed experiments to show the scalability of our approach with respect to plan size. We compared the results of Orca against plans produced by *Planner*, where partitioning is based on PostgreSQL inheritance mechanism, and query plans explicitly list all partitions that need to be scanned. Note that the latter is also the approach presented in [7].

4.4.1 Queries with a Constant Partition-eliminating Predicate

In this experiment we start with a partitioned *lineitem* table from the TPC-H benchmark, and run a simple selection query, which limits the number of partitions that need to be scanned with a predicate of the form $l\_shipdate < X$. By varying X we produce queries that select 1%, 25%,50%,75%, and 100% of the partitions. Figure 18(a) shows that Orca's query plan size remains constant, while *Planner*'s plan grows linearly with the number of partitions that need to be scanned (and thus explicitly enumerated in the plan).

4.4.2 Queries with a Join Partition-eliminating Predicate

In this experiment, we consider synthetically generated tables $R(a, b)$, $S(a, b)$ partitioned on $R.b$ and $S.b$, respectively. We vary the number of partitions in each table, and execute the following join query:

```
select * from R, S where R.b=S.b and S.a<100;
```

The planner supports dynamic partition elimination, where the necessary partition OIDs are computed at runtime and stored in a parameter, which is then passed to the actual query plan for the join, which uses the parameter to determine whether a given partition needs to be scanned or not. The query plan however needs to list all partitions, as it is not known at optimization time which partitions will be eliminated. Thus the query plan size is a function of the number of partitions, as can be seen in Figure 18(b). As seen in the figure, the measured plan size for Orca also shows some dependence on the number of partitions. This is due to a limitation of the way metadata is replicated on the segment nodes, which requires some part of the metadata required by the partitioning functions in Section 3.2 to be embedded in the query plan structure and shipped together to the segment nodes. The actual query plan size is indeed independent of the number of partitions and has the shape of the plan shown in Figure 8(b).

4.4.3 DML Queries over Partitioned Tables

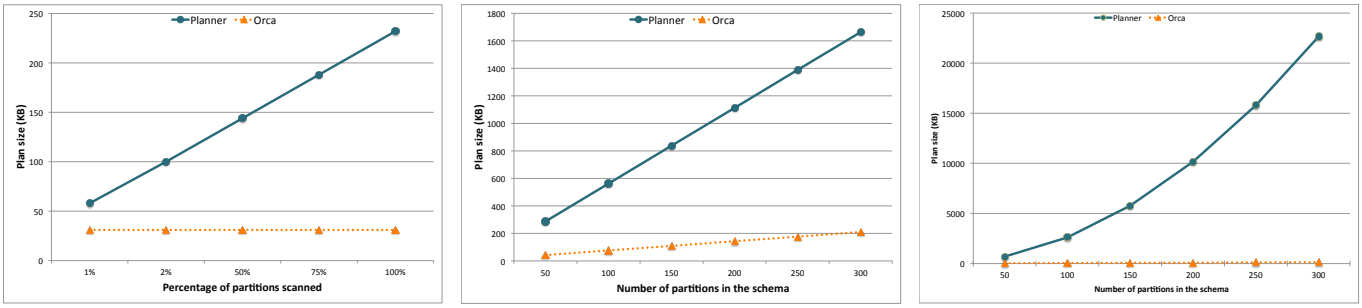
In this experiment, we use tables R and S from Section 4.4.2. We execute the following DML statement which updates table R with values from S:

```
update R set b=S.b from S where R.a=S.a;
```

Figure 18(c) shows that the plan size for *Planner*-produced plans grows quadratically with the number of partitions in the tables, as the plan needs to enumerate all join combinations between the individual parts. In Orca, this enumeration is not necessary, so the plan size remains nearly the same. Again, the small variations in Orca's plan size are due to inefficiencies of the metadata catalog as pointed out in the previous experiment, and not to the actual plan shape.

5. RELATED WORK

In this section, we review related efforts in exploiting possible query optimization opportunities when querying partitioned tables. Most DBMSs (IBM DB2 [9], Oracle [11], Microsoft SQL Server [16]) support both single-level and multi-level partitioning. In most of these systems, static partition elimination takes place during query optimization, where selection predicates are used to determine which parts to scan, and only these parts are referenced in the execution plan. Dynamic partition elimination works by binding the join variables at run time, and computing qualifying partitions for scanning. Additionally, Oracle [11] supports *partition-wise joins* for cases where two partitioned tables are joined on their corresponding partitioning key. In this case, the join is broken down to smaller joins, where every partition from the first table is only joined with the matching partitions from the other table. While some form of dynamic partition elimination is implemented in those systems, we could only find a handful of simple examples of single-level equality joins, with no description of how the query optimizer selects these plans, and whether the techniques work for complex queries or handle only simple join patterns. To the best of our knowledge, our paper is the first comprehensive study of how to implement dynamic partition elimination techniques



(a) Static partition elimination (b) Dynamic partition elimination (c) DML queries

Figure 18: Plan size comparison between *Planner* and Orca for different query patterns

for complex queries and integrate them in a general purpose query optimizer for an MPP system.

A recent research effort [7] added sophisticated optimization techniques for querying partitioned tables to the PostgreSQL optimizer. The authors assume a star/snowflake schema where the dimension and the fact tables are partitioned on the same column(s). Their approach works on partition-wise joins and eliminates joins that cannot produce results based on subsumption of the partitioning constraints. Although the approach works well for the assumed schema, it does not decouple the plan size from the number of partitions. This is a significant issue when dealing with thousands of partitions. In addition, it does not directly apply to massively parallel processing systems.

As Hadoop quickly becomes a popular ecosystem for big data analytics, many SQL on Hadoop solutions have been proposed. Open-source approaches, such as Cloudera's Impala [10], Facebook's Presto [5] and Hortonworks' Stinger [8] only support static partition elimination inherited from Hive [4]. Although dynamic partition elimination is proposed [13], it is not implemented yet, while Pivotal's HAWQ [12] utilizes Orca and supports both static and dynamic partition elimination in a unified way.

Finding good partitioning schemes is part of database physical design tuning and is outside the scope of this paper. However, various techniques to find a good partitioning scheme automatically have been proposed [3, 14].

6. SUMMARY

The distributed nature of MPP database systems poses unique challenges when dealing with partitioned tables. The key if finding an abstraction that meshes well with both the elementary building blocks of modern query optimizers and yet meets performance-oriented scalability requirements.

In this paper, we presented a methodology that represents partitioned tables and all relevant methods to access them as elements of an algebra. The resulting framework enables the query optimizer to explore a variety of plan alternatives. Most importantly, our design emphasizes dynamic plans in which desicions to access a partition—at potentially considerable I/O cost—are deferred to query execution in order to take advantage of specific characteristics of the underlying data, which are unknown at optimization time.

The system presented is implemented in Pivotal Greenplum Database as part of the Orca optimizer initiative and has proven highly effective in production deployments.

For future work, we plan to address a number of advanced subjects including indexing, better modeling of costs, and completeness of interoperability with other optimizations.

7. REFERENCES

[1] TPC-DS. `http://www.tpc.org/tpcds`, 2005.

[2] TPC-H. `http://www.tpc.org/tpch`, 2009.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD*, 2004.

[4] Apache. Hive. `http://hive.apache.org/`, 2013.

[5] L. Chan. Presto: Interacting with petabytes of data at Facebook. `http://prestodb.io`, 2013.

[6] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bul.*, 18(3), 1995.

[7] H. Herodotou, N. Borisov, and S. Babu. Query Optimization Techniques for Partitioned Tables. In *SIGMOD*, 2011.

[8] Hortonworks. Stinger, Interactive query for Apache Hive. `http://hortonworks.com/labs/stinger/`, 2013.

[9] IBM. DB2 Partitioned Tables. `http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.partition.doc/doc/c0021560.html`, 2007.

[10] M. Kornacker and J. Erickson. Cloudera Impala: Real-Time Queries in Apache Hadoop, for Real. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`, 2012.

[11] T. Morales. Oracle Database VLDB and Partitioning Guide 11g Release 1 (11.1). 2007.

[12] Pivotal. HD: HAWQ. `http://www.gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf`, 2013.

[13] L. J. Pullokkaran and L. Leverenz. MapJoin and Partition Pruning. `https://cwiki.apache.org/confluence/display/Hive/MapJoin+and+Partition+Pruning`, 2013.

[14] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating Physical Database Design in a Parallel Database. In *SIGMOD*, 2002.

[15] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A Modular Query Optimizer Architecture for Big Data. In *SIGMOD*, 2014.

[16] R. Talmage. Partitioned Table and Index Strategies Using SQL Server 2008. 2009.