Indexing HDFS Data in PDW: Splitting the data from the index

Vinitha Reddy Gankidi
University of Wisconsin-Madison
vinitha@cs.wisc.edu

Nikhil Teletia
Microsoft Jim Gray Systems Lab
nikht@microsoft.com

Jignesh M. Patel
University of Wisconsin-Madison
jignesh@cs.wisc.edu

Alan Halverson
Microsoft Jim Gray Systems Lab
alanhal@microsoft.com

David J. DeWitt
Microsoft Jim Gray Systems Lab
dewitt@microsoft.com

Abstract

There is a growing interest in making relational DBMSs work synergistically with MapReduce systems. However, there are interesting technical challenges associated with figuring out the right balance between the use and co-deployment of these systems. This paper focuses on one specific aspect of this balance, namely how to leverage the superior indexing and query processing power of a relational DBMS for data that is often more cost-effectively stored in Hadoop/HDFS. We present a method to use conventional B+-tree indices in an RDBMS for data stored in HDFS and demonstrate that our approach is especially effective for highly selective queries.

1. Introduction

The debate between relational DBMS and MapReduce systems for big data management has now converged in the traditional RDBMs vendor space to produce hybrid systems that aim to balance performance and cost. In such systems, a database is split across a parallel RDBMS and a MapReduce [3] system. Data in the MapReduce/Hadoop system is stored in HDFS and is made "visible" to the RDBMS as an external table. Several large database vendors employ this mechanism [1, 4, 5, 6]. Queries can now be issued to the RDBMS against data that is split across these two data systems. During query execution, a portion of the query may be executed as MapReduce jobs in Hadoop, while another portion of the query is executed inside the RDBMS.

This hybrid model has evolved naturally since relational DBMSs tend to provide far higher performance on queries compared to MapReduce-based systems (and support a larger set of SQL), but tend to be far more expensive when measured on the $/TB measure (which is often dominated by the installing, licensing, and annual service/maintenance fees). Deploying large databases purely in a parallel relational DBMS is cost prohibitive for some customers, especially since in these large databases all the data is not uniformly "hot." A sizable part of the database is cold data that is queried less frequently than the hot data. A natural partitioning is to store the hot data in the parallel RDBMS, and store the cold data in HDFS.

Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08

This hybrid model works well in practice when the workload consists of long-running analytical queries on large tables (which can be executed in a cost-effective way by keeping the table in HDFS and processing queries using MapReduce jobs), or for queries on the small amounts of hot data that is kept in the RDBMS. For that later category, fast query times are often the norm, as the RDBMS engines have sophisticated query optimization and query processing techniques that have been designed, refined, and solidified over the last three decades.

However, a key limitation of these hybrid systems is that even simple "lookup" (rifle-shot and range) queries on the large data files/tables in HDFS have long latencies. Real customer workloads are getting increasingly complicated in practice, and it is not unusual to have workloads that demand "interactive" query response times on these lookup queries. In fact a number of customers of Microsoft's product in this space (Polybase) have this requirement, which motivated the work that is described here.

The focus of this paper is on designing, evaluating and implementing methods to improve the speed of lookup queries on data that is stored in HDFS in a hybrid data processing system.

Our approach to solve the problem described above is to leverage the indexing capability in the RDBMS by building an index on the external HDFS data using a B+-tree that is stored inside the RDBMS. This *Split-Index* method leverages the robust and efficient indexing code in the RDBMS without forcing a dramatic increase in the space that is required to store or cache the entire (large) HDFS table/file inside the RDBMS.

A natural follow-up question is how to keep the index synchronized with the data that is stored in HDFS. On this aspect, we recognize that data that is stored in HDFS cannot be updated in place. Rather, HDFS only allows adding new data or deleting old data (by adding or deleting files in HDFS). Thus, we can solve this index-update problem (which can be thought of as a special case of the materialized view update problem) using an incremental approach by which we record that the index is out-of-date, and lazily rebuild it. Queries posed against the index before the rebuild process is completed can be answered using a *Hybrid-Scan* method that carefully executes parts of the query using the index in the RDBMS, and the remaining part of the query is executed as a MapReduce job on just the changed data in HDFS. Thus, lookup queries continue to be executed efficiently even with HDFS data that is undergoing modification.

While in the empirical evaluation in this paper we focus only on single table queries, we note that the Split-Index method also provides an opportunity to speed up some analytical join queries (as opposed to the simpler lookup queries) on data that sits in HDFS. If the query only looks at attributes that are in the index,

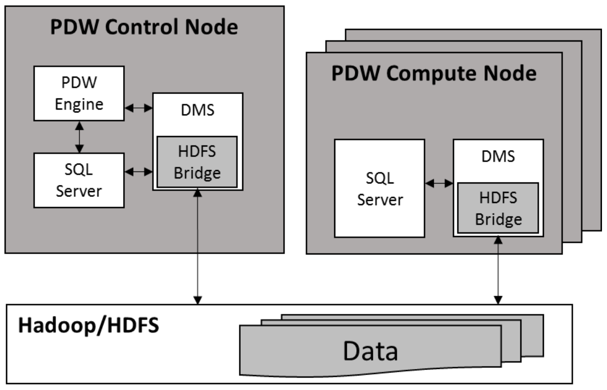

Figure 1: The Polybase Architecture

then even complex join queries can be answered completely inside the RDBMS. In other cases, the index can be used as a pre-filter to reduce the amount of work that is carried out as MapReduce jobs. Thus, the Split-Index method also serves as a versatile mechanism to cache the "hot" attributes of "cold" HDFS data inside the RDBMS, where it can generally be processed much faster. Furthermore, this Split-Index method can also be used as a mechanism to adapt to changing workloads and system configurations; thus, for example, if there is a new hot attribute in a HDFS-resident table, or more nodes are added to the RDBMS, then one can simply index more attributes of the HDFS file in the RDBMS. In other words, one can gradually materialize the most commonly accessed components of data in HDFS inside the RDBMS and maximize the return-on-investment (ROI) on the RDBMS deployment.

The remainder of this paper is organized as follows. In Section 2, we present some background information related to Polybase – the system that we use in this paper. In Section 3, we present the Split-Index approach. A method to incrementally update the Split-Index is presented in Section 4, and Section 5 presents the Hybrid-Scan approach. Our experimental results are presented in Section 6, and Section 7 contains our concluding remarks.

2. Polybase Background

Polybase employs a "split query processing" [1, 2] paradigm to achieve scalable query processing across structured data in relational tables and unstructured data in HDFS. Polybase leverages the capabilities of SQL Server PDW, especially, its cost-based parallel query optimizer and execution engine. While using MapReduce provides a degree of query-level fault tolerance that PDW lacks, it suffers from fundamental limitations that make it inefficient when executing trees of relational operators. Polybase relies on the cost-based query optimizer to determine when it is advantageous to push SQL operations on HDFS-resident data to the Hadoop cluster for execution.

2.1 Polybase Architecture

Polybase [1] extends the PDW architecture to allow for querying data that is stored in HDFS. As shown in Figure 1, PDW has a *control node* that manages a number of *compute nodes.* The PDW Engine in the control node provides an external interface and query requests flow through it. The control node is responsible for query parsing, optimization, creating a distributed execution plan, issuing plan steps to the compute nodes, tracking the execution steps of the plan, and assembling the individual pieces of final results into a single result set returned to the user. The compute nodes are used for data storage and query processing.

The control and compute nodes each run an instance of SQL Server and an instance of the *Data Movement Service* (*DMS*). DMS instances are responsible for repartitioning the rows of a table among the SQL Server instances on the PDW compute nodes. The *HDFS Bridge* component hosted inside each DMS is responsible for all communications with HDFS. It enables DMS instances to also import/export data from/to HDFS clusters.

2.2 External Tables

Polybase, like Greenplum, Oracle, Asterdata and Vertica, uses an external table mechanism for HDFS-resident data [4, 5, 6]. The first step in declaring an external table is to create an external data source for the Hadoop cluster on which the file resides. The external data source contains information about the Hadoop NameNode and the JobTracker for the cluster. The JobTracker is used to submit a Map job when the PDW optimizer elects to push selected query computation to Hadoop. The next step is to create an external file format, which contains information about the format of the HDFS files. Polybase supports both delimited text file, and RCFile.

The following example illustrates how an external table is created. The location clause is a path to either a single file or a directory containing multiple files that constitute the external table.

```
CREATE EXTERNAL TABLE hdfsLineItem
  (l_orderkey BIGINT NOT NULL,
   l_partkey BIGINT NOT NULL,
   ...)
WITH (LOCATION='/tpch1gb/lineitem.tbl',
DATA_SOURCE = VLDB_HDP_Cluster,
FILE_FORMAT = TEXT_DELIMITED)
```

2.3 HDFS-Import

When compiling a SQL query that references an external table stored in HDFS, the PDW Engine Service contacts the Hadoop NameNode for information about the HDFS file/directory. This information, combined with the number of DMS instances in the PDW cluster, is used to calculate the split (offset and length) of the input file(s) that each DMS instance should read from HDFS. This information is passed to DMS in the *HDFS Shuffle* operation of the DSQL (*distributed SQL*) plan along with other information that is needed to read the file. This additional information includes the file's path, the location of the appropriate NameNode, and the name of the RecordReader that the HDFS Bridge should use.

The system attempts to evenly balance the number of bytes read by each DMS instance. Once the DMS instances obtain split information from the NameNode, each instance can independently read the portion of the file that it is assigned, directly communicating with the appropriate Hadoop *Data Nodes* without any centralized control.

Once an instance of the HDFS Bridge has been instantiated by the DMS process, the DMS workers inform the HDFS Bridge to create a RecordReader instance for the specified split (offset and length) of an input file. The RecordReader then processes the rows in the input split and returns only the required attributes (we refer to this operation as *HDFS-Import*). The query is run on this imported data in PDW.

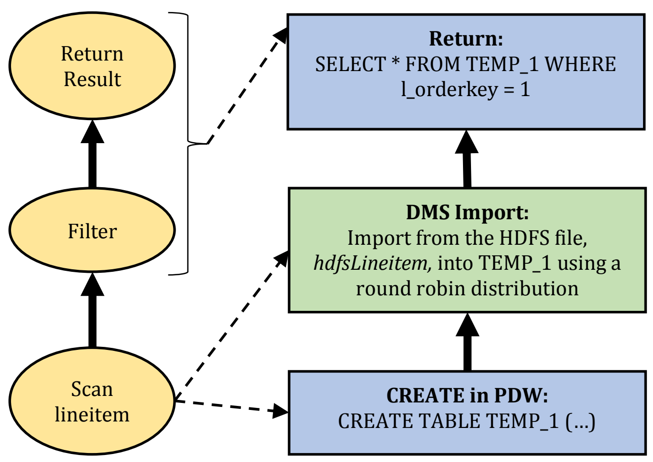
Return Result
Filter
Scan lineitem
Return:
SELECT * FROM TEMP_1 WHERE l_orderkey = 1
DMS Import:
Import from the HDFS file, hdfsLineitem, into TEMP_1 using a round robin distribution
CREATE in PDW:
CREATE TABLE TEMP_1 (...)

Figure 2: Query plan and the corresponding DSQL plan for the Direct-Import path

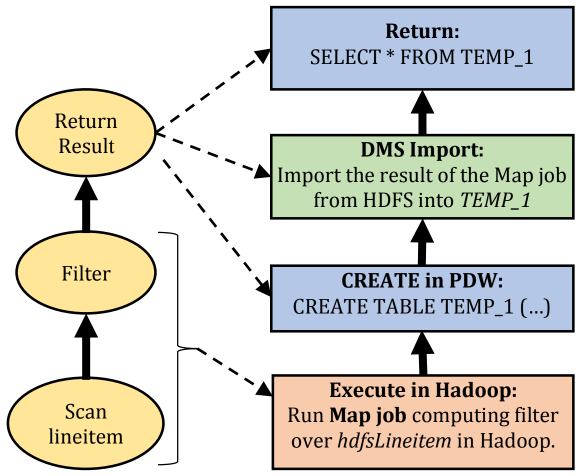
Return Result
Filter
Scan lineitem
Return:
SELECT * FROM TEMP_1
DMS Import:
Import the result of the Map job from HDFS into TEMP_1
CREATE in PDW:
CREATE TABLE TEMP_1 (...)
Execute in Hadoop:
Run Map job computing filter over hdfsLineitem in Hadoop.

Figure 3: Query plan and the corresponding DSQL plan for the Push-Down path

The execution path where the external data used inside a query is imported during the query execution is called *Direct-Import*.

2.4 Push-Down to Hadoop

When a query involves HDFS resident data, the optimizer can submit a Map job to the Hadoop cluster and then import only the result of the Map job. We call this approach the *Push-Down* execution path. The optimizer makes a cost-based decision to decide whether to use the Push-Down path or the Direct-Import path. A simple example where the Push-Down path is cost effective is a query with a selective predicate. Using the Push-Down path, the *Filter* (i.e. selection) operator can be computed on the Hadoop cluster, which results in smaller amount of data being imported into PDW. Currently a few operators (Project and Filter) are supported in the Polybase Push-Down path.

We illustrate the Push-Down and the Direct-Import paths using the following query:

```
SELECT *
FROM hdfsLineItem
WHERE l_orderkey = 1
```

The logical query plan for the above query is simple – it has two logical operations: a Scan operator followed by a Filter operator.

Figures 2 and 3 show the DSQL plans for the Direct-Import and the Push-Down paths, respectively. As shown in Figure 2, in the Direct-Import path, the entire customer table is first imported into PDW. While in the case of the Push-Down path, a Map job containing the Filter operation is submitted to the external Hadoop cluster. This Map job scans the customer table and materialized only those rows that have *c_nationkey* value equal to 1. After the Map job finishes execution, the result of the Map job is imported into PDW, and the remainder of the query is executed inside PDW.

Using the Push-Down approach, Polybase can delegate some computation responsibility to the external Hadoop cluster, thereby reducing the amount of data that is imported from the external cluster. If the data pipe between PDW and the Hadoop cluster has limited bandwidth, then the Push-Down approach can have significant performance advantage over the Direct-Import approach. The Polybase paper [1] discusses the crossover point between these two approaches in detail.

3. The Polybase Split-Index

In this section, we introduce the Polybase Split-Index approach, which is used to speed up the query execution time of lookup queries. Section 3.1 describes how the index is created and Section 3.2 describes how the index is used during query execution.

3.1 Index Creation

Consider a TPC-H *lineitem* table stored in HDFS. Creating a PDW Split-Index named *lineitem_index* on the *l_orderkey* attribute of the *lineitem* table entails the following steps:

1. Create a PDW table *lineitem_index* containing five columns: *l_orderkey* (the indexed attribute), *filename* (the name of the HDFS file that holds the record/tuple), *offset* (the offset of the record from the beginning of the file), *length* (the length of the record) and *blockNumber* (a computed column). The pair (filename, offset) acts as a *Record Identifier* (RID) for the row. The RID and length attributes are used to read the required number of bytes of the qualifying records during query execution. The blockNumber is a computed column (blockNumber = offset/16MB), on which the rows of the index table are hash partitioned. By distributing the *lineitem_index* table on the blockNumber, we ensure that all rows in a 16MB chunk are stored together in the Split-Index.

2. To populate the *linitem_index* table, the HDFS-Import operation (see Section 2.3) in invoked with a special "buildIndex" flag set to true. Next, the DMS workers running on each compute node of the PDW appliance instantiate an HDFS Bridge instance and pass the special flag to the RecordReader instance. When this flag is set, the RecordReader calculates the *physical pointer* (RID, length, blockNumber) to the record in addition to the indexed attributes. As rows are produced, the HDFS-Import operation inserts them into the hash-partitioned PDW table *lineItem_index,* and for each partition creates a clustered B-tree index on the *l_orderkey* attribute.

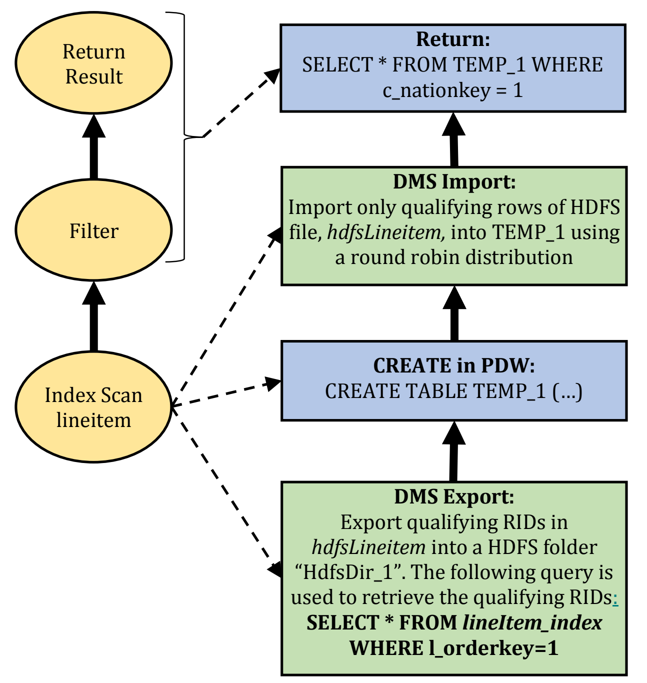
Return Result
Filter
Index Scan lineitem
Return:
SELECT * FROM TEMP_1 WHERE c_nationkey = 1
DMS Import:
Import only qualifying rows of HDFS file, hdfsLineitem, into TEMP_1 using a round robin distribution
CREATE in PDW:
CREATE TABLE TEMP_1 (...)
DMS Export:
Export qualifying RIDs in hdfsLineitem into a HDFS folder "HdfsDir_1". The following query is used to retrieve the qualifying RIDs:
SELECT * FROM lineItem_index WHERE l_orderkey=1

Figure 4: Query plan and the corresponding DSQL plan for the Index-Scan path

3. The list of all the HDFS files on which the Split-Index is created is stored as an extended property of the PDW Split-Index table. This information is used to update the index incrementally to keep it up-to-date with changes to the "master-data" in HDFS.

In Polybase, we can create an index on any number of columns, and on any PDW supported data type.

3.2 Index-based Query Execution

To reduce the latency of lookup queries, we enhance Polybase to use the Split-Index that is created in PDW. Note that this index can be used only when a query includes a predicate that involves the indexed attributes.

In the existing implementation of the HDFS-Import operation, the HDFS Bridge is given a range [offset, offset + length] to retrieve records from an HDFS file. This operation is called the *Range-Scan* operation. To execute the query using the Split-Index mechanism, we enhanced the HDFS Bridge to support an *Index-Scan* operation (in addition to the Range-Scan operation). With the Index-Scan operation, instead of giving the HDFS Bridge a range, it is provided a collection of RIDs to retrieve records from the HDFS file(s). The HDFS Bridge retrieves records by iterating over the list of RIDs.

Figure 4 shows the DSQL plan for the Index-Scan path using the following query:

```
SELECT *
FROM hdfsLineitem
WHERE l_orderkey = 1
```

The execution of this query involves the following steps:

1. Once the query is submitted to PDW, the PDW Engine (in the control node) creates a DSQL plan that uses the *lineItem_index* table in PDW.
2. Each compute node executes the query: '`SELECT * FROM lineItem_index WHERE l_orderkey=1`' against its portion of the Split-Index in PDW.
3. The results of the above query (i.e. a set of qualifying RIDs) are exported to a HDFS directory via the HDFS Bridge running in the DMS instances of the compute nodes.
4. A temporary table is created in PDW to store the result of the HDFS-Import-using-Index operation that is described next.
5. An *HDFS-Import-using-Index* operation is executed on each compute node. During the execution of this operation, a reference to the exported/materialized files containing the qualifying RIDs is passed to the HDFS Bridge instances. The HDFS Bridge performs a pointer-based join between these materialized RIDs and the *lineitem* files in HDFS.
6. The required attributes of the qualifying records are then imported into the temporary table in PDW that was created in Step 4 above.

There are multiple ways in which qualified RIDs can be passed to the HDFS Bridge. We used HDFS files to transfer the qualified RIDs to the HDFS Bridge. Other potential options include using shared memory or passing a list of RIDs along with the function call.

4. Incremental Index Update

Compared to data in a traditional database system, rows in Hadoop cannot be updated in place. Rather, new rows are added by adding new HDFS files to the directory, or existing rows are deleted by dropping one or more HDFS files from the directory. Given this pattern of data creation and updates, at any given point of time the data in the external table can be classified into three categories: 1) **Existing** data that is covered by the existing Split-Index in PDW, 2) **New** data that is not indexed, and 3) **Deleted** data that has been dropped in HDFS but is still represented in the Split-Index.

The Split-Index is an offline index, which means that when the data is updated, the index is not updated at the same time and requires that the administrator/user issue an explicit update statement. When an update statement is issued, an *incremental index update* method is invoked, which has the following two phases: i) Detecting updates (new or deleted data), and ii) Incrementally updating the existing index.

Recall that with each PDW Split-Index table we also store the list of HDFS files that are covered by that index (cf. Section 3.1). We use this index metadata to find new and deleted files.

For new files, the index update first requests the HDFS Bridge to scan the new files with the "buildIndex" flag set to true. The HDFS Bridge then returns the indexed attribute values and the physical pointer to the new records in these files. These records are then inserted into the existing PDW Split-Index table, and the index metadata is updated to add these new files.

For deleted files, we update the index using the following SQL statement:

```
DELETE FROM <index>
WHERE filename IN (<Deleted Files>)
```

These deleted files are also removed from the index metadata.

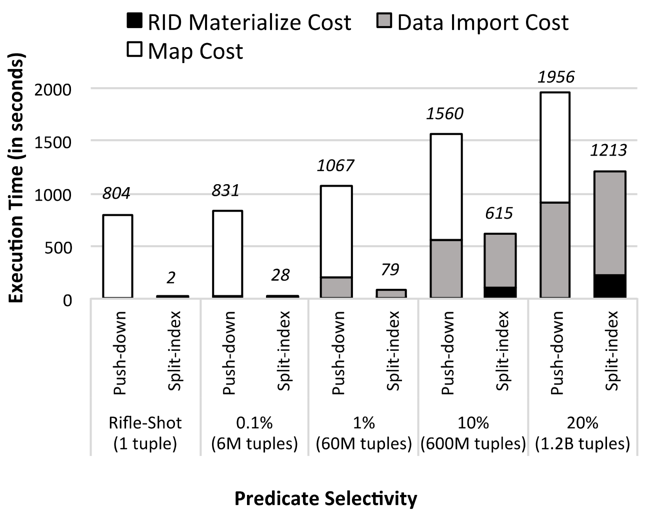

Figure 5: Split-Index execution path performance for lookup queries using the traditional Push-Down and the proposed Split-Index method for the 1TB *lineitem* table

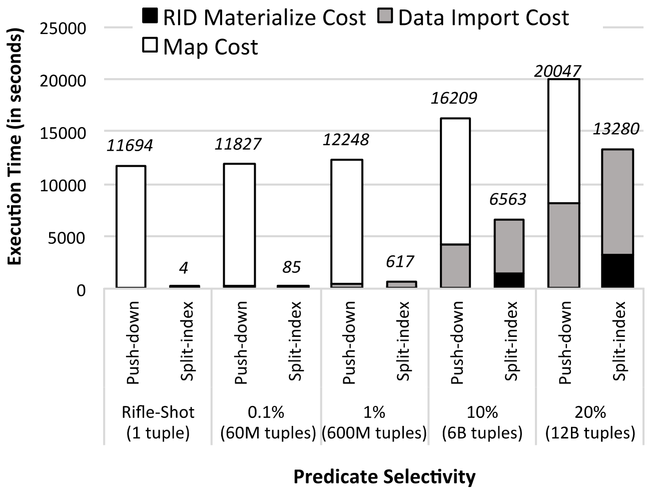

Figure 6: Split-Index execution path performance for lookup queries using the traditional Push-Down and the proposed Split-Index method for the 10TB *lineitem* table

5. Hybrid-Scan

As discussed above, a Split-Index in PDW can potentially be stale w.r.t. the data in HDFS. When a Split-Index is stale, we exploit the absence of in-place updates in HDFS, and use a *Hybrid-Scan* operation. This operation uses the stale index for 'existing' data (i.e. it invokes the Index-Scan operation), and it invokes the Range-Scan method (cf. Section 3.2) to scan the 'new', unindexed, data. 'Deleted' data is removed from the scan by adding the clause `"filename NOT IN (<Deleted Files>)"` to the query that computes the qualifying RIDs. We refer to the query execution path that uses the Hybrid-Scan operation as the *Hybrid-Scan path*. The optimizer also has the option of using either the Direct-Import path or the Push-Down path.

6. Experiments

In this section we present our experimental results. The key goals of our experiments are to:

1. Measure the performance gain of the Split-Index and the Hybrid-Scan paths over the Push-Down path for highly-selective lookup queries.
2. Understand the cost of index creation and index maintenance.
3. Understand the sensitivity of the Index-Scan operation to the data access pattern.
4. Understand the space vs. time tradeoffs when the data is entirely in PDW or HDFS with/without indexes built on the data.

6.1 Setup

We used a cluster with 9 PDW nodes and 29 Hadoop nodes. Each node has dual 2.13 GHz Intel Xeon L5630 quad-core processors, 96GB of main memory, and ten 300GB 10K RPM SAS disk drives. One drive is used to hold the operating system (OS), another to hold the OS swap space, and the remaining eight are used to store the actual data (HDFS files, or permanent and temporary table space for SQL Server). All the PDW nodes are on a single rack and all the Hadoop nodes are spread across the remaining two racks. Within a rack, nodes are connected using a 10 Gb/sec Ethernet link to a Cisco 2350 switch. Between the racks these switches communicate at 10 Gb/sec.

The PDW and Hadoop services on their corresponding nodes were run in a single Windows Hypervisor VM configured with Windows Server 2012, 88GB RAM, SQL Server 2012, a prototype version of PDW V2 AU1, and HDP 2.0 (Windows Hadoop 2.2.0). On each node 80 GB of memory was allocated for PDW or Hadoop services.

All experiments were run using a TPC-H *lineitem* table of different scale factors. The *lineitem* table was loaded into the Hadoop cluster as uncompressed text files. All the numbers reported in this section are cold cache numbers, which means neither PDW nor HDFS data was in memory cache.

6.2 Index Build Cost

In this section, we describe the cost associated with building the Split-Index. It took 3,265 and 32,714 seconds to build indexes over the 1000 and 10000 scale factor TPCH *lineitem* tables on the *l_orderkey* attribute respectively. The corresponding lineitem tables sizes are 1 TB and 10 TB, respectively. The sizes of the indexes are ~10% of the original data size (~80GB and ~800GB).

6.3 Lookup Queries

In this section, we compare the performance of the Split-Index against the Push-Down path, using the following query:

```
SELECT * FROM lineitem
WHERE l_orderkey <= [Variable]
```

For the above query, the optimizer has three execution paths. We will briefly explain these three execution paths and the costs associated with each path.

1. **The Direct-Import execution path**: In this path, the *lineitem* table is imported entirely into PDW incurring a "Data Import cost." Then, the remainder of the query is executed inside PDW.

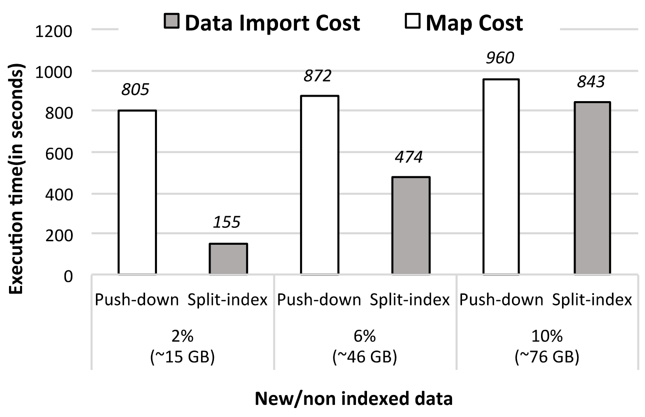

Figure 7: Rifle-shot query evaluation with the Hybrid-Scan method for a 1TB scale *lineitem* table

2. **The Push-Down execution path**: In this path, a Map job is submitted to the Hadoop cluster. The Map job filters the rows and stores the output of that job in HDFS. After the Map job has completed, the result of the Map job is imported into PDW, and the final result is returned back to the user. Note that currently PDW doesn't support returning the result directly from HDFS. This path has a "Map cost" and a "Data Import cost" associated with it.

3. **Index-Scan execution path**: In this path, the qualifying RIDs are first materialized into a HDFS directory by running a predicate query on the *lineitem_index* table. After the materialization, the HDFS Bridge imports the data only for the materialized RIDs. This path has an associated "RID Materialization cost" and a "Data Import cost."

Note that we do not show the result for the Direct-Import path as its performance is dominated by the "Data Import Cost" of importing the entire *lineitem* table, and adding these numbers obscures the comparison between the Split-Index and the Push-Down paths.

The results for the lookup queries are shown in Figures 5 and 6. For this experiment, we vary the selectivity of the predicate on the *l_ordeykey* attribute. The rifle-shot query only retrieves tuples with a specific *l_orderkey* value, whereas the range queries retrieve tuples within a range of *l_orderkey* values. The figure shows the time for the two paths: the *Push-Down* path and the *Split-Index* path. The Push-Down path starts a Map job on the Hadoop cluster to evaluate the predicate, and then imports the selected records into PDW to evaluate the remainder of the query. The Split-Index path, first evaluates the predicate using the B+tree in PDW, and materializes a list of "record-ids". The records in this list are then "imported" into PDW, where the remainder of the query is processed.

Figures 5 and 6 also show the breakdown of these costs for both methods. As can be seen in the figures, the Split-Index method is many orders-of-magnitude faster for highly selective queries. The Split-Index method outperforms the Push-Down method because the Split-Index method scans and imports only those tuples that satisfies the query predicate, while the Push-Down method scans the entire *lineitem* file during the Map job execution phase. Consequently, even though the imported data into PDW is the same for both methods, the amount of *lineitem* data that is scanned is far smaller for the Split-Index method, resulting in significantly better query execution time. For the other queries (besides the Rifle-Shot query), the Split-Index approach is faster than the Push-Down approach for similar reasons, but as the Data Import cost starts becoming a bigger component of the total query execution time, the performance gap between the two methods reduces.

6.4 Hybrid-Scan

We have also implemented the *Hybrid-Scan* (cf. Section 5) method that we described above. Figure 7 shows the results for the rifle-shot query from Figure 1 when varying the amount of new data that is added to the 1TB *lineitem* table. As can be seen in Figure 7, the Hybrid-Scan method outperforms the traditional Push-Down method when there is up to 10% new data, since the Hybrid-Scan method benefits from the efficient use of the Split-Index on existing data. In addition to the amount of new data, the crossover point of the Hybrid-Scan method is affected by the query selectivity, and the data access pattern. The impact of the data access pattern is described below in Section 6.6.

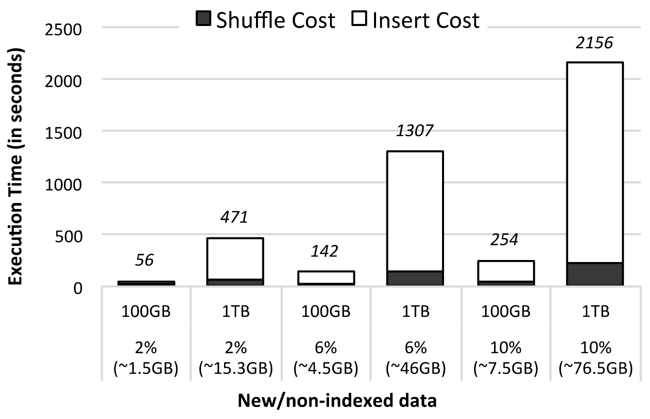

Figure 8: Index update cost when varying the amount of new data added to the 100GB scale and the 1TB scale *lineitem* tables.

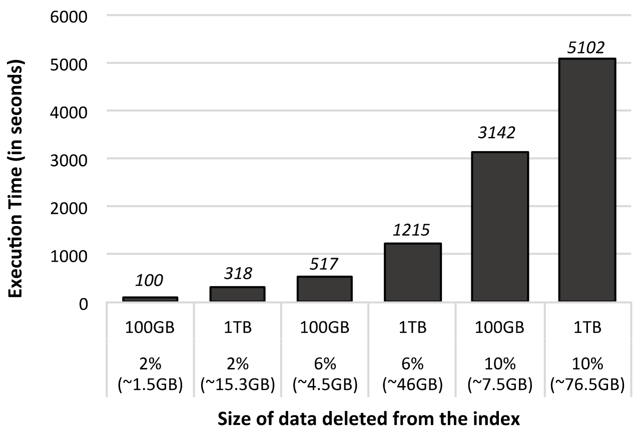

Figure 9: Index update cost when varying the amount of data is deleted from the 100GB and the 1TB lineitem tables.

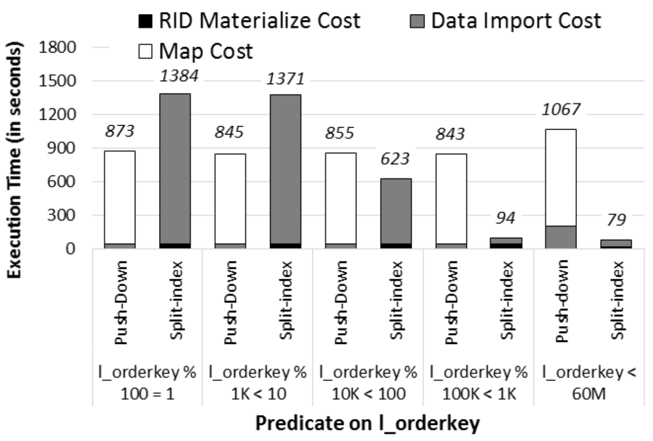

Figure 10: Sensitivity of the Index-Scan operation to the data access pattern for the 1TB lineitem table when the selectivity factor is fixed at 1%.

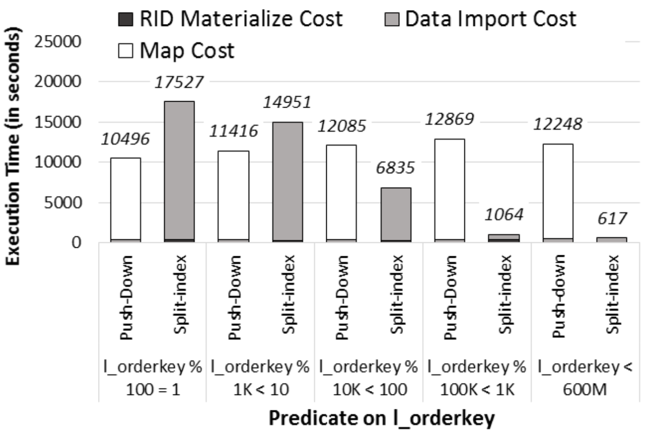

Figure 11: Sensitivity of the Index-Scan operation to the data access pattern for the 10TB *lineitem* table when the selectivity factor is fixed at 1%.

6.5 Index Maintenance

In this section, we describe the cost associated with incrementally updating the index. Figures 8 and 9 show the results when updating the index while varying the amount of data that is added or deleted to/from the 100GB and 1TB *lineitem* tables. (For this experiment, we use a 100 scale factor, i.e. 100GB, *lineitem* table.)

As shown in Figures 8 and 9, when we add 10% of 'New' data to the 1TB *lineitem* table, it takes 2,156 seconds to incrementally update the index, which is roughly 66% of the initial index build cost. In addition, it takes 5102 seconds to incrementally update the index when 10% of the data is deleted from the original 1TB *lineitem* table.

As can also be seen in Figures 8 and 9, the cost of incremental index update is proportional to the size of the *lineitem* table. During these incremental updates, transactional logging dominates the overall execution cost. An interesting option, which we plan to consider as part of future work, is to use a cost-based optimizer to drop and recreate the index if a large portion of the data has changed.

6.6 Data Access Pattern

In this section, we look at the sensitivity of the Index-Scan operation to the data access pattern. The results for this experiment are shown in Figures 10 and 11. For this experiment, we fix the selectivity factor at 1% and vary the access pattern by changing the predicate in the following query:

```
SELECT * FROM lineitem
WHERE l_orderkey <= [Variable]
```

The predicates that are used in each query are shown using labels on the x-axis of Figures 10 and 11. From these two figures we observe that the Split-Index method benefits as the data access pattern becomes more sequential, since as the data access pattern becomes more sequential, the number of unique file seeks that are required is reduced. The Lineitem table generated by the TPCH DBGen tool is clustered on the *l_orderkey* column. Because of this clustered layout, the predicate "*l_orderkey % 100K < 1K*" accesses the rows in sequential order, while the predicate *"l_orderkey%100 = 1"* accesses the rows in random seek order.

When the rows are accessed sequentially, as shown in the last two access patterns in Figures 10 and 11, we see significant benefit for the Split-Index method over the Push-Down method. Some of these performance benefits are the artifacts of HDFS caching and HDFS read ahead that is done on the client side. HDFS client reads the data in 4KB chunk (this parameter can be update using the io.file.buffer.size configuration setting, but we used the default value), and when the read pattern in sequential, there are a large number of hits (for rows) in the cache. In contrast, when there are random seeks, the cache hit rate is very low.

Please note that we clear the HDFS cache before running any individual query, the above mentioned caching occurs during the execution of the query.

6.7 Space vs. Time Tradeoffs

In this experiment, we analyze the space versus time tradeoff between keeping the data in HDFS and indexing the data in PDW. The goal of this experiment is to quantify the space versus performance tradeoff when all the data is stored in PDW as opposed to storing the data in HDFS and selectively building indices (i.e. a Split-Index) on increasingly larger number of columns in PDW. One way of looking at these experiments is to think of the Split-Index as a materialized (index) view on the data in HDFS, and we increase the amount of index data that is materialized in PDW. To keep this experiment manageable, we only consider the case where one index is built. (An interesting direction for future work is to consider the more general problem of physical schema optimization within the context of the Split-Index approach that is proposed in this paper.)

For this experiment, we consider the modified TPCH Query 6, which forecasts revenue change:

```
SELECT SUM(l_extendedprice*l_discount)
AS REVENUE
FROM lineitem
WHERE l_shipdate >= '1994-01-01'
  AND l_shipdate <
   dateadd(mm, 1, cast('1994-01-01' as date))
  AND l_discount BETWEEN .06 - 0.01 AND
               .06 + 0.01 AND l_quantity < 24
```

Table 1: Various approaches that were considered for the execution of the modified TPCH Query 6

Approach	Description
A1	*lineitem* in PDW. No Index.
A2	*lineitem* in PDW. One clustered B+ Tree PDW index on *l_shipdate.*
A3	*lineitem* in PDW. One clustered B+ Tree PDW index on (*l_shipdate, l_discount, l_quantity* and *l_extendedprice).*
A4	*lineitem* in HDFS. No Split-Index, and use the Push-Down approach.
A5	*lineitem* in HDFS. One Split-Index on *l_shipdate*, and use the Index-Scan approach.
A6	*lineitem* in HDFS. One Split-Index on (*l_shipdate, l_discount)*, and use the Index-Scan approach.
A7	*lineitem* in HDFS. One Split-Index on (*l_shipdate, l_discount, l_quantity)*, and use the Index-Scan approach.
A8	*lineitem* in HDFS. One Split-Index on (*l_shipdate, l_discount, l_quantity, l_extendedprice)*, and use the Index-Scan approach.

To reduce the selectivity factor of the query, we modified the *l_shipdate* predicate to consider dates within a month instead of a year (as in the original TPCH Q6). As the Split-Index method is sensitive to the data access pattern, we also sorted the 1TB *lineitem* table in HDFS by the *l_shipdate* attribute, to make accesses to the HDFS data more efficient.

The different configurations that we considered are shown in Table 1. There are two categories of approaches. In the first category of approach, which consists of the approaches A1, A2 and A3, the data is stored entirely in PDW, and we use larger amounts of disk space in PDW by varying the number of columns on which the PDW index is built. In the second category, i.e. the approaches A4 – A8, the data is resident in HDFS, and we vary

the number of columns on which the Split-Index is built.

The first category of approaches consumes increasing larger amount of space in PDW, while the second category of approaches takes smaller amounts of space in PDW. The first category of approaches is expected to have a higher performance, while the second category of approaches potentially has a lower cost. Collectively, these methods help gauge the return on investment (ROI) from using a more sophisticated data processing engine (PDW) while storing large data sets in the relatively cheaper HDFS storage.

The results for this experiment are show in Figure 12. As expected the PDW-only approaches (A1 – A3) have far higher performance compared to the other methods.

Figure 12 shows that the approaches A1, A2 and A3 have low execution times but have a high PDW disk footprint. In these approaches, the *lineitem* table is resident in PDW. Approaches A6, A7 and A8 have low execution times with a moderate PDW

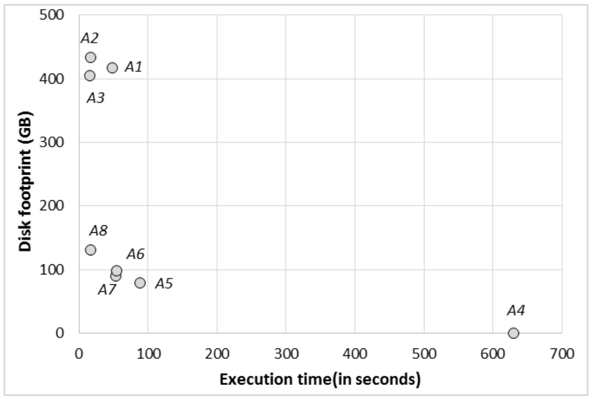

Figure 12: Execution time and PDW disk footprint for the different approaches that are described in Table 1.

disk footprint. These approaches employ the Split-Index method proposed in this paper. The Push-Down approach (A4) has zero disk footprint in PDW, but has a very high execution time. Approach A8 has similar execution times as A2 and A3, but the disk footprint of A8 is one third that of A2 or A3. This shows the effectiveness of the Split-Index in balancing the query execution time and the PDW disk footprint.

An astute reader may have observed that even though the approach A3 has a clustered index on four columns, the PDW disk space used by the approach A3 (~405 GB) is less than the PDW disk space that is used by approach A1 (~417 GB). The reason for this anomaly is page compression. Adding the clustered index on the columns *l_shipdate, l_discount, l_quantity* and *l_extendedprice,* results in a data layout that is sorted by these attributes, and that ordering of data turns out to provide better page compression for this dataset.

7. Conclusions and Future Work

In this paper we have implement a traditional database technique, i.e. “indexing,” to speed up queries on HDFS resident data. Our empirical results show that for lookup queries, our proposed Split-Index method improves query performance by two to three orders of magnitude. We also show how our proposed approach has advantage over existing methods (though to a smaller degree) for selective range queries. In addition, we exploit the absence of in-place updates in HDFS to incrementally update the Split-Index, and to enable a Hybrid-Scan method that can work with data that is only partially indexed by a Split-Index.

As part of future work, we want to expand our approach to other file formats such as the RCFile and the ORCFile. The ORCFile has block-level indexing and exploring synergies with that indexing method can lead to interesting designs. We also want to evaluate the impact of using our methods in cloud settings where the network bandwidth is far more limited. We think our approach will likely perform even better when the network bandwidth between PDW compute nodes and the Hadoop nodes is limited. We also want to implement an index advisor that automatically identifies the columns that can be cached in PDW given a query workload.

8. Acknowledgements

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab.

9. REFERENCES

[1] David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, Jim Gramling: Split query processing in polybase. SIGMOD Conference 2013: 1255-1266

[2] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, Erik Paulson: Efficient processing of data warehousing queries in a split execution environment. SIGMOD Conference 2011: 1165-1176

[3] Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150

[4] Oracle White paper. High Performance Connectors for Load and Access of Data from Hadoop to Oracle Database, June 2012. http://www.oracle.com/technetwork/bdc/hadoop-loader/connectors-hdfs-wp-1674035.pdf

[5] http://www.greenplum.com/sites/default/files/EMC_Greenplum_Hadoop_DB_TB_0.pdf

[6] Aster SQL-H: http://www.asterdata.com/sqlh/

[7] Mohamed Y. Eltabakh, Fatma Özcan, Yannis Sismanis, Peter J. Haas, Hamid Pirahesh, Jan Vondrák: Eagle-eyed elephant: split-oriented indexing in Hadoop. EDBT 2013: 89-100

[8] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, Jörg Schad: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). PVLDB 3(1): 518-529 (2010)

[9] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, Jörg Schad: Only Aggressive Elephants are Fast Elephants. PVLDB 5(11): 1591-1602 (2012)

[10] http://www.greenplum.com/sites/default/files/EMC_Greenplum_Hadoop_DB_TB_0.pdf

[11] http://www.asterdata.com/sqlh/