Partition Balancing in Distributed Nested Data-Parallel Systems

Abstract

Map-reduce frameworks such as Apache Hadoop and Spark provide the abstraction of a large, flat array that is processed in parallel across many machines. While this simple programming model has enabled the broad adoption of data-parallel distributed frameworks, these systems cannot express irregular parallel computation, and their performance is impacted by data imbalance across nodes. In this paper, we present a distributed framework for implementing *nested data parallel* (NDP) computation. Unlike previous NDP systems described by Blelloch and Bergstrom et al. that relied on the use of a *flattening* transformation at compile-time, we present a cost model that is used to select between multiple partitioning strategies at run-time. Through this, we provide a user-tunable means for trading node-to-node imbalance versus communication when executing distributed NDP programs.

1. Introduction

The use of map-reduce as a flat data-parallel (FDP) programming model for distributed systems has grown rapidly since it was introduced in Google's seminal 2004 paper [?]. The development of the open-source Apache Hadoop system enabled the use of this programming model outside of Google. Modern map-reduce systems such as Apache Spark [?] have refined the programming model further by reducing the dependency of the framework on disk via in-memory caching. While this refinement has enabled the use of map-reduce for iterative workloads, the programming model remains confined to computation that can be expressed via flat data-parallel operations. Specifically, Spark presents users with the abstraction of a resilient distributed dataset (RDD), which appears as a flat array that is distributed across compute nodes in a cluster [?].

To expand the algorithms that could be expressed as a data-parallel computation, Blelloch introduced the nested data-parallel (NDP) model [?]. In this programming model, users are provided the abstraction of an array whose elements are a nested level of arrays and data-parallel operators are applied on the nested arrays. A primary complication in the implementation of the NDP model is that the nested arrays frequently do not have uniform size. Several approaches have been suggested for balancing work in NDP programs, including the compile-time vectorization of NDP programs for execution on single-instruction multiple-data (SIMD) machines [? ?] and flattening for multiple-instruction multiple-data (MIMD) machines [?]. Additionally, dynamic work-stealing approaches have been implemented [?].

In this paper, we introduce a distributed programming framework for NDP algorithms. Our implementation is built on top of Apache Spark. We track the structure of the nested arrays at runtime and choose between two strategies (*uniform* and *segmented*) for partitioning data across machines based on the estimated cost of each strategy. In the segmented partitioning strategy, all values in a single nested segment are co-located on a single compute node, and most computation can proceed without communication. The uniform strategy provides perfect load balance across all nodes, but many operations will need to communicate to execute. To choose between these strategies at runtime, we provide a cost model that evaluates the performance tradeoff of communication overhead versus node-to-node imbalance. Since the partitioning strategy is chosen at runtime, we can leverage knowledge of the nested array structure.

This work introduces the following contributions:

1. We introduce a dynamically optimized distributed method for implementing NDP algorithms that is amenable to "cloud computing" platforms.
2. We demonstrate that the current best-known `scan` algorithm has $O(\frac{n \log p}{p})$ performance on bulk-synchronous systems, and provide a "block-parallel" relaxation of the `scan` operator that has $O(\frac{n}{p})$ performance.
3. We demonstrate a model-based approach for balancing partition load in distributed systems.

2. Background

Understanding the system we describe in the rest of the paper requires an understanding of the NDP processing model, as well as the design of current MapReduce-style distributed computing frameworks. In this section, we compare and contrast the FDP and NDP programming models, and discuss the system design of the Spark MapReduce framework. While the problem of implementing scalable distributed FDP frameworks is well studied [? ?], this paper builds upon those systems to focus on the implementation of scalable distributed NDP platforms.

2.1 Data-Parallel Programming Models

In data-parallel frameworks, processing proceeds in parallel across all of the elements in a dataset. We can further subdivide data-parallel programming models into flat and nested data parallelism:

Flat data-parallelism: In a FDP system, the parallel collection contains single elements which are then processed in parallel. On a collection containing elements of type T, a few basic operations include:

- `map`, which applies a function $F : T \mapsto U$ to all elements of the collection, and returns a new collection containing elements of type U

- reduce, which successively applies a function $F : T, T \mapsto T$ to pairs of elements from the collection until all elements have been reduced down into a single scalar of type T

FDP may also be known as *regular* data parallelism, as the data parallel computation proceeds over the single items in the dataset (all parallel computation occurs on single elements, i.e., "collections" of size 1). Because there are no dependencies between elements in the dataset, FDP can implemented efficiently on a broad range of computing platforms, including SIMD and MIMD architectures, graphics processing units (GPUs) [**?**], and distributed systems [**?**].

Nested data-parallelism: An NDP system extends the collection structure of an FDP system. Instead of a flat collection, the collection is divided into *segments* or *nests*, which are sub-groupings of elements. Operations may then execute across the whole collection, or across the segments inside of the collection. A full introduction to NDP can be found in the text by Blelloch [**?**]. Blelloch justifies the utility of the NDP model by noting that NDP reduces the asymptotic runtime of some graph and linear algebra algorithms by a factor of $\log n$. A good demonstration of the utility of the nested vector model comes from implementing a matrix-vector multiply. Here, we assume that we have a matrix A which has been packed into a nested vector by splitting each row i into it's own segment, and a dense vector v. We can implement the matrix-vector multiply by:

```
def matrix-vector-multiply(A: SegmentedArray[Int],
                                       v: Array[Int]
  ): Array[Int] = {
  A.map-with-index((point: Int, idx: Int) => {
    point * v(idx)
  }).segmented-reduce(_ + _)
}
```

Here, we use the `map-with-index` function to multiply each point in the matrix by the corresponding point in the vector. This can execute in parallel per element. The `segmented-reduce` function then sums up all of the elements from each row. The naïve implementation of a `segmented-reduce` is parallel per segment, but an optimized implementation is parallel per element in the whole collection.

2.2 Data-Parallel Distributed Computing

This work builds upon the infrastructure provided by the Apache Spark distributed data-parallel computing framework [**? ?**]. Spark was designed for "cloud computing" platforms, where machines may be unreliable, and where network performance may preclude the use of traditional distributed message passing systems such as the Message Passing Interface (MPI). Unlike Apache Hadoop, where data is shuffled to/from disk between all processing stages [**? ?**], Spark has an in-memory processing model. The in-memory processing model leads to large ($100\times$) performance for iterative jobs implemented using Spark instead of Hadoop [**?**].

Spark's programming model is implemented on top of the *resilient distributed dataset* (RDD) abstraction [**?**]. The RDD abstraction presents a view of an array which is chopped into *partitions* that are distributed over the computers within the Spark cluster. Programs enqueue data-parallel transformations on the Spark master, which are then interpreted into a directed-acyclic graph (DAG) for execution; this approach is similar to the DAG scheduling approach pioneered in Dryad [**?**]. Spark executes the DAG whenever a disk shuffle is required, or if a newly enqueued operation would create a cycle in the DAG (i.e., an iterative computation is scheduled).

To execute a stage, the Spark master serializes the user defined function (UDF) which is being applied, transmits this function to all worker nodes, and then applies the computation. While the general application programming interface (API) that Spark provides is data-parallel across all elements, the API transformations are implemented by applying the transformation iteratively across all elements of a partition—Spark also exposes this parallel-by-partition interface via the `mapPartitions` call. If data must be moved between partitions after an execution stage, a disk shuffle will occur. Disk shuffles are analogous to an execution stage with interleaved computation and all-to-all communication followed by a barrier before the next stage. This process is similar to how the results of `map` phases are moved to reducers in both MapReduce and Hadoop [**?**]—the main distinction is that Spark allows successive `map` phases, and that shuffles do not occur after all stages.

3. Implementation

We have implemented an NDP model on top of the FDP computing model of the Apache Spark system. Our implementation is approximately 3k lines of Scala [**?**] code, and adds the following NDP operations to Spark's FDP operations:

- Block-parallel `scan`: We provide an $O(\frac{n}{p})$ operation which is a slight relaxation of the `scan` primitive [**?**]. We discuss the full details of this primitive in §**??**.
- `combine`, `shuffle`, and `transpose`: These operations allow for the creation of novel nested structures from a current nested collection. `combine` allows for an n-to-1 remapping of values to new index positions, while shuffle allows for a 1-to-1 reordering of indices. The `transpose` operator is important for linear algebra.
- Efficient indexed `get`: Spark does not currently support indices. We add indices to track the nested collection structure. By adding indices, we are able to provide $O(1)$ lookup by only performing the lookup on the single partition known to contain the value.
- Segmented operations:
  - `segmented-scan`: The `segmented-scan` operator applies a `scan` across all segments. In our API, we provide the correctness guarantees of a block parallel `scan`. For the `segmented` partitioning strategy introduced in §**??**, we tighten this guarantee to match the canonical `scan` operator.
  - `segmented-reduce`: We provide $O(\frac{n}{p})$ `segmented-reduce`. This operation calculates a scalar value per segment by applying a reduction operator associatively across the segment.
- Multi-collection operations:
  - p-operations: p-operations denote a class of operations that apply an operator pairwise to values from two collections that exist at the same index. For two numerical nested collections, the `p-+` operator would be used to implement matrix addition.
  - `merge`: The `merge` operator allows for the creation of a nested dataset from one or more flat datasets, and is instrumental in building up nested collections.

Our API provides the abstraction of a `NestedRDD`, which is an RDD (see Zaharia et al [**?**] or §**??**) where each value is keyed by its position in the nested collection. As mentioned above, the `NestedRDD` also maintains an index which tracks the size of all the segments in the RDD. Although a naïve implementation of the `NestedRDD` can be implemented using RDDs and the Scala collection classes, this implementation is inefficient:

- As Scala is built upon the Java Virtual Machine (JVM), the use of collections would increase the number of objects created, and would thusly increase garbage collection (GC) overhead.
- Scala's collection classes provide a wide variety of data-parallel methods but are known for having generally poor performance when compared to their Java equivalents.
- The use of collection classes on top of RDDs provides insufficient control over the placement and partitioning of data.

To improve upon the naïve implementation, we introduce two different methods for dynamically dividing datasets across machines in a Spark cluster. We choose between these two partitioning strategies on the basis of a simple model for calculating imbalance. We introduce the model we use for deciding between the two partitioning strategies in §**??**; this model uses the current structure of the segments in the array to select a partitioning strategy. The partitioning strategies themselves are explained in §**??**. In addition to the optimized implementations, we do make the naïve `NestedRDD` implementation available to enable correctness checking of the more efficient partitioning methods.

Since the structure of the segments is necessary for choosing a partitioning strategy, we must track the structure at alloverset times. We define nested collections as being either sparse or dense:

- In a *dense* nested collection, for all segments S^k, with segment size s_k, there exists a value $S_i^k \forall i \in \{0, \ldots, s_k - 1\}$.
- In a *sparse* nested collection, we may have gaps. Therefore, it is possible for us to have a value S_i^k where $i \geq s_k$, and it is also possible for $\{S_{\hat{i}}^k\} = \emptyset$ for a given $\hat{i} < \max_{i \in \{S_i^k\} \neq \emptyset} i$.

When a nested collection is dense, we must recompute the segment structure after `combine` and `merge` operations. While `transpose` causes the segment structure to change, we can calculate the new structure from the previous known segment structure. For sparse nested collections, the segment structure must also be computed after a p-operation, and we cannot use the prior segment structure to calculate the new structure after a `transpose` operation. We can fully recompute the segment structure with the following algorithm:

```
def recompute-segment-structure(newRdd: NestedRDD[T]
  ): (SegmentStructure, Boolean) = {
  // get the index for each element
  // returns rdd with tuple of (segment number,
  //                             index in segment)
  val idxRdd: RDD[(Int, Int)] =
    newRdd.map(v => v.getIndex)

  // get the number of keys per segment
  val nestSizes: Map[Int, Int] = idxRdd.map(kv => kv._1)
    .count()

  // what is the highest index in each segment?
  val maxIndex: Map[Int, Int] =
    idxRdd.reduceByKey(_ max _)

  // return the number of keys in each nest and whether
  // the collection is dense
  (SegmentStructure(nestSizes),
   nestSizes.join(maxIndex)
     .forall(kv => kv._1 - 1 == kv._2))
}
```

Recomputing the segment structure is facilitated by storing the index with every value, and by forbidding duplicate indices in the collection.

Our implementation has no impact on the fault tolerance built into the Spark framework. Spark provides fault tolerance by tracking the *lineage* of all partitions in the system [**?**]. If a partition is lost, Spark repeats the computation needed to reproduce the partition from the last checkpointed state. As our API is implemented on top of Spark's partitioning system, users of our system benefit from Spark's built-in fault tolerance mechanisms.

3.1 Modeling Imbalance

Extensive work has been done to characterize and model the performance of parallel systems from both theoretical [**?**] and practical [**? ?**] angles. In practice, we are able to use a simpler model for modeling imbalance. Specifically, we can conceptualize a simple model as follows:

- Our cluster is constructed out of n processing elements $e_i, i \in \{0, \ldots, n-1\}$, and our dataset is partitioned into n partitions $P^i, i \in \{0, \ldots, n-1\}$, with size s_i.
- We perform c processing stages. We assume that each stage performs an equal amount of computation.
- During each processing stage, each processing element e_i computes on the data in partition P^i in s_i time, and then performs a barrier. Therefore, we allocate time $\max_i s_i$ to each processing stage.
- Between each processing stage, we pay a barrier/communication latency of t_b. This parameter will vary system to system, and depends on the communication network performance. For simplicity, we normalize this parameter to the cost of performing a processing stage.

With this model, the computational cost of a job is:

$$C = c \max_i s_i + (c-1) t_b$$

As we will describe in the following section on partitioning strategies (§**??**), we seek to decide between two strategies: `uniform`, which minimizes imbalance, and `segmented`, which minimizes computation. In the `segmented` setup, the maximum partition size is the size of the longest segment (s_l), while the `uniform` setup equally balances v values across p partitions such that $s_i = \frac{n}{p}, \forall i \in \{0, \ldots, p-1\}$. The `uniform` approach will perform up to two stages of computation, while the `segmented` approach will never perform more than one stage. Thus:

$$\begin{aligned} T_u &\overset{?}{=} T_s \\ s_l &\overset{?}{=} 2\frac{n}{p} + t_b \\ \frac{s_l p}{2n} &\overset{?}{=} t_b \end{aligned}$$

We chose the `segmented` strategy whenever the imbalance caused by the largest partition is smaller than the overhead of executing two stages with a barrier.

In practice, because the amount of data we are shuffling is small, the cost of $s_t \sim 2\frac{n}{p} \gg t_b$, so t_b can be dropped. Additionally, the equation above will not exactly predict imbalance; if all executors are c-core machines, executor e_i will process co partitions, where o is an oversubscription factor (typically 3, discussed in §**??**. The combination of oversubscription and bucketing cores into executors will typically load balance.

3.2 Partitioning Strategies

Given the performance model introduced in §**??**, we introduce two separate strategies for data partitioning:

- `segmented`: In this strategy, each segment/nest of the nested array is allocated a single partition of the RDD. Unless the size of each segment is identical, the size of each partition may vary.
- `uniform`: In this strategy, each partition of the RDD is allocated the same number of values; therefore, the partition sizes are uniform. In this strategy, a single segment may be split across one or more partitions.

If the number of segments equals the number of partitions, and all segments have identical length, both strategies are identical and reduce to the `segmented` strategy.

We face the following tradeoffs when picking a strategy for partitioning a dataset:

- *How well will load be balanced?* The `uniform` strategy is highly desirable because we are able to perfectly allocate the keys to partitions, and load is balanced. However, in practice, load balance can be improved by oversubscribing partitions to processing elements (see *Tuning Guide* in [**?**]; typically, an oversubscription factor of 3 is used). If there is oversubscription, we may be able to co-schedule small and large partitions on the same processing element. While we may have imbalance between each partition, we may be able to achieve balance between processing elements.
- *How much overhead does the strategy incur?* While the `segmented` strategy may have increased imbalance, it has low overhead because we can guarantee that all segment-based operations complete without requiring communication between partitions. This is desirable because the bulk synchronous model common to MapReduce frameworks implies that all communication requires a barrier.

The additional overhead of the `uniform` partitioning model is limited to the `segmented-scan` operation. To implement a `segmented-scan`, we fall back on the block-parallel `scan` implementation that we introduce in §**??**. There are several additional complications due to the segmented nature of the `scan`; specifically, we must track whether the results generated by previous partitions are related to this segment. This requires us to track additional information during the *pre-scan* phase, and necessitates filtering when performing the *update* phase.

While it may appear that the `segmented-reduce` operation will also incur additional overhead, this overhead is negligible. For the `segmented` partitioning strategy, each partition performs a reduction across its elements, and then sends the results to the master. The `uniform` strategy performs a keyed reduction—we only reduce together values that are from the same segment. Thus, if a partition contains values from s segments, it will compute s reduced values (one per segment) after the initial partition reduction. We label these values with the segment number, collect the values to the driver, and reduce together values with like labels. If we ensure that the reduction operator is associative and commutative, this approach will be correct. Overhead is limited, as generally $n_p \gg s_p$, where n_p is the number of values in partition p, and s_p is the number of segments with values in partition p.

The structured nature of a nested array provides a significant advantage over unstructured data-parallel models, as it enforces a rigid, known, and efficiently quantifiable order across the data. For unordered datasets, a hash function must be used to partition data in parallel across nodes [**? ?**]. Even for ordered datasets, finding an even partitioning is expensive; in Spark, sorting requires the calculation of histogram describing the distribution of keys across the ordering [**?**]. Since the key distribution is described by the array structure, and a nested array structure can be described by an array of lengths, we can statically track the whole ordering on the master node, instead of needing to recalculate the key distribution before repartitioning.

3.3 Block-Parallel Scans

The `scan`[1] operator is challenging to implement in parallel, as the scan operation on array element a_n depends on the values of all elements $a_0 \ldots a_{n-1}$. For clarity, we reproduce the definition of a `scan` given by Blelloch [**?**] here:

Definition 1. The `scan` operator takes an associative operator $\oplus$, and an ordered set of n elements:

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns the ordered set:

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

Blelloch proposed a parallel algorithm for computing vector scans [**?**]; this algorithm has recently been implemented for GPUs [**?**]. This algorithm is composed of two basic steps:

1. First, perform an "up sweep" by performing a parallel tree-reduction across the vector.
2. "Down sweep" by shifting in the identity value and rotating the tree elements.

A detailed description of this algorithm, as well as a proof of correctness is presented by both Blelloch [**?**] and Harris [**?**]. This algorithm completes in $O(\frac{n}{p} + \log p)$ time when evaluating a vector of n elements on p processors. For $n \gg p$, the runtime is approximated as $O(\frac{n}{p})$. Despite the efficient runtime, the *up sweep, down sweep* algorithm has several drawbacks:

- For a vector of type T, the $\oplus$ operator is restricted to $T, T \mapsto T$. This is an unnecessary restriction if we loosen definition **??** to include an additional input variable b_0, which is the identity value. If b_0 has type U, this allows $\oplus$ to become $U, T \mapsto U$. We include this in our updated definition **??**.[2]
- The *up sweep, down sweep* algorithm only has $O(\frac{n}{p})$ runtime when executing on systems with a parallel random access memory (PRAM) compute model. As described in §**??**, MapReduce computing systems provide a block synchronous model. For this model, the runtime degrades to $O(\frac{n \log p}{p})$, as we have $\frac{n}{p}$ runtime for each of the $\log p$ stages in both the *up sweep* and *down sweep* stages.

To address the issues we have just presented, we introduce a block-parallel scan:

Definition 2. The block-parallel `scan` operator takes an associative *scan* operator $\oplus$ ($U, T \mapsto U$), and an associative *update* operator $\otimes$ ($U, U \mapsto U$), an identity value b_0, and an ordered set of n elements:

$$A = [a_0, a_1, \ldots, a_{n-1}]$$

The ordered set A is partitioned into p ordered partitions:

[1] The `scan` operator is also known as the *all-prefix sum* [**?**].

[2] Note that an identity value is required for the `scan` operations implemented by both Blelloch [**?**] and Harris [**?**]. We are simply adding the identity value to the formal definition of a `scan` .

$$
\begin{aligned}
P^0 &= [a_0, a_1, \ldots, a_{\frac{n}{p}-1}] \\
P^1 &= [a_{\frac{n}{p}}, a_{\frac{n}{p}+1}, \ldots, a_{2\frac{n}{p}-1}] \\
&\ldots \\
P^{p-1} &= [a_{(p-1)\frac{n}{p}}, a_{(p-1)\frac{n}{p}+1}, \ldots, a_{n-1}] \\
A &= P^0 \cup P^1 \cup \ldots \cup P^{p-1}
\end{aligned}
$$

For notational convenience, we assume that $n \bmod p = 0$.

For all $P^k, k \in \{0, \ldots, p-1\}$, we compute intermediate *block pre-scans* S^k where:

$$
\begin{aligned}
S_0^k &= b_0 \\
S_i^k &= S_{i-1}^k \oplus P_{i-1}^k, \forall i \in 1, \ldots, p
\end{aligned}
$$

Note that all sets S_k have size $p+1$, not p.

For all $S^k, k \in \{0, \ldots, p-1\}$, we then collect the S_p^k terms into set C, where $C_k = S_p^k$. We then perform the *update* phase across all S^k:

$$
u^k = \begin{cases} b_0 & \text{if } k = 0 \\ C_0 \otimes \ldots \otimes C_{k-1}, & \text{otherwise} \end{cases}
$$

$$
U_i^k = u_k \otimes S_i^k, \forall i \in \{1, \ldots, p\}
$$

We return the ordered set:

$$
\begin{aligned}
&S = U_0 \cup U_1 \cup \ldots \cup U_{p-1} \\
&S = [b_0 \oplus a_0, \ldots, b_0 \oplus a_0 \oplus \ldots \oplus a_{\frac{n}{p}-1}, \ldots, \\
&(b_0 \oplus a_0 \oplus \ldots \oplus a_{\frac{n}{p}-1}) \otimes (b_0 \oplus a_{\frac{n}{p}} \oplus \ldots \oplus a_{2\frac{n}{p}-1}) \otimes \ldots \\
&\otimes (b_0 \oplus a_{(p-1)\frac{n}{p}} \oplus \ldots \oplus a_{n-1})]
\end{aligned}
$$

It is not possible to generally prove that the scan operator is equal to the block-parallel scan operator for all general $\oplus$ operators. However, it is straightforward to prove equivalence for some common operators. For example, given $T = U =$ integer, and $\oplus = + = \otimes$, it follows that $b_0 = 0$. In this case, all of the b_0 terms drop out. Since $\oplus = \otimes$, the block-parallel scan operation is demonstrated to match the scan operation.

Both the per-block *pre-scan* and the *update* operations from definition **??** have runtime $O(\frac{n}{p})$. On the computing platforms we are targeting, the cost of collecting a small amount of data from each partition is negligible. Therefore, the block-parallel scan has runtime $O(\frac{n}{p})$. We are able to use the block-parallel scan algorithm to perform efficient parallel scans across all datasets, and to accelerate segmented scans when using the uniform partitioning strategy introduced in §**??**.

4. Performance

5. Discussion

In this paper, we have advocated for a more efficient scan implementation, and introduced a dynamic system for picking partitioning strategies. In this section, we explain our reasoning for not including a traditional scan implementation, explain our vision for future extensions of this work, and provide readers with a link to our implementation.

5.1 Relaxed Accuracy Models For Scans

In §**??**, we introduced the block-parallel scan, which is a relaxed implementation of the scan described by Blelloch [**?**] that has improved performance when executing on a bulk-synchronous MapReduce platform. We have decided not to include the less-performant scan implementation for the following reasons:

- The parallel *sweep up, sweep down* algorithm proposed by Blelloch can only work for transformations with fixed type (i.e., $\oplus : T, T \mapsto T$), which is more restrictive than a sequential scan implementation.
- While it is not possible to generally prove equivalence between the block-parallel scan and the traditional scan, we believe that the two approaches are sufficiently similar for most purposes, and we note that the two implementations can be proven equivalent for several common cases (e.g., +-scan and *-scan).

With the degraded performance ($O(\frac{n}{p})$ to $O(\frac{n \log p}{n})$ of the scan implementation on bulk-synchronous architectures, scans across the full dataset would not be highly useful due to poor performance with the growth of $\log p$. Applying the traditional scan implementation to segmented-scans may still remain desirable, as the $\log p$ growth is mitigated by p being the maximum number of partitions that a single segment is split across. However, if a traditional segmented-scan is necessary, users can force the partitioning-strategy in use to the segmented strategy—this point is addressed in §**??**.

5.2 Future Work

A major avenue for future work surrounds improving the balancing decision framework described in §**??** and §**??**. There are two prominent areas where the implementation of this model could be improved:

- *The current model does not include setup costs:* To switch between the uniform and segmented partitioning strategies, a cluster-wide shuffle must occur. Currently, we do not model this cost.
- *The current model does not look at the computation DAG:* Spark creates a DAG for all scheduled computation. We do not use this DAG during strategy selection, but it could be used for determining how long setup costs would be amortized over, as well as identifying whether the full stage penalty should be applied to the uniform strategy.

We see the greatest benefit as coming from the use of the DAG during the partition selection phase. This is partially because viewing the DAG would allow us to determine the true impact of setup costs (if we perform a shuffle, but then leave data in place for a long time, the cost of the shuffle is decreased), but is mostly because the DAG would allow us to minimize the penalty afforded to uniformly partitioned datasets. In §**??**, we note that the runtime of uniformly partitioned stages is $2\frac{n}{p} + t_b$; this is true for segmented-scans. Other operators have predicted runtime of $\frac{n}{p}$. If we encounter a workload that doesn't contain many scans, the cost of computing on uniformly partitioned data will drop towards $\frac{n}{p}$, which implies that our cost model is inaccurate for workloads with few scans.

We have avoided this implementation at the current time, because it is not straightforward to implement on top of the Spark API. A dynamic partitioning strategy selecting algorithm that operated on the Spark DAG would need to have the capability to modify the Spark DAG by inserting stages whenever it determined that we needed to switch partitioning methods and shuffle the dataset. Our current implementation avoids this by implementing our API at a much higher level on top of the Spark API.

5.3 Availability

Our implementation is released as open-source software under the Apache 2 license, and is available through the Apache Maven Central repository [**?**]. A link to the implementation in our publicly open source-control system is provided in the non-anonymized supplementary material submitted with the paper.

6. Conclusion