PDQCollections: A Data-Parallel Programming Model and Library for Associative Containers

Maneesh Varshney Vishwa Goudar
Technical Report #130004
Computer Science Department
University of California, Los Angeles
April, 2013

ABSTRACT

Associative containers are content-addressable data structures, such as maps, ordered maps, multimaps, sets etc., that are wide employed in a variety of computational problems. In this paper, we explore a parallel programming paradigm for data-centric computations involving associative data. We present PDQCollections - a novel set of data structures, coupled with a computation model (which we refer to as the Split-Replicate-Merge model) - that can efficiently exploit parallelism in multi-core as well distributed environments, for in-memory as well as large datasets. The distinguishing characteristics of our programming model are the design of data structure that inherently encapsulate parallelism and the computation model that transforms the problem of parallelization to that of defining addition operator for value data types. The PDQ design offers fundamental benefits over traditional data structures: with memory-bound workloads, PDQ avoids locks and other forms of synchronization, and is able to significantly outperform lock-based data structures; with larger disk-bound workloads, PDQ does not use caches and avoids random disk access, and thus significantly outperform traditional disk-backed structures; and with distributed workloads, PDQ accesses remote data resources sequentially and only once, and is able to significantly outperform distributed data structures. We highlighted the distinguishing capabilities of PDQ library with several applications drawn from a variety of fields in Computer Science, including machine learning, data mining, graph processing, relational processing of structured data and incremental processing of log data.

1. INTRODUCTION

Associative containers, also commonly referred to as hashes, hash tables, dictionaries or maps, are a cornerstone of programming template libraries. They provide standardized, robust and easy-to-use mechanisms to efficiently store and access key-value mappings. In this paper, we explore data-centric programming problems where either the input or the output or both are associative containers. Such computations are commonly employed in the fields of document processing, data mining, data analytics and machine learning, statistical analysis, log analysis, natural language processing, indexing and so on. In particular, we seek a data-parallel programming framework for associative data, where the parallelism can scale from multi-core to distributed environments, the data can scale from in-memory to disk-backed to distributed storage and the programming paradigm is as close as possible to the natural sequential programming patterns.

The problems of data parallelism with associative containers are unlike from the index-based data structures such as the arrays and matrices. The most familiar parallel programming paradigm for the latter is the parallel for loop, as exemplified in OpenMP, Intel’s Thread Building Blocks (TBB) and Microsoft’s .NET Task Parallel Library (TPL). In this model, the index range of the for loop is partitioned and each partition of the range is assigned to a separate thread. However, this paradigm requires that the input to the computation (that has to be partitioned for parallelization) must be index-addressable. Secondly, as the outputs generated by each thread must be protected against concurrent modifications, each thread must write data in non-overlapping memory regions or use critical sections or concurrent-safe containers. Maps are content-addressable, and consequently, can neither serve as input or output in a parallel for computation. Furthermore, in the data-parallel programming context, the synchronization locks are content-addressable, and consequently, can neither serve as input or output in a parallel for computation. Finally, these libraries, are typically meant for shared-memory and memory bound workloads and offer little support or perform poorly with distributed systems and persistent data stores.

In our study of data-parallel programming paradigms, we did not discover any library or framework that can: (a) operate on associative containers, (b) execute in shared memory multi-threaded as well as distributed contexts, (c) support data size that scales from in-memory to disk-backed, and (d) have parallelization constructs that are as close as possible to the natural sequential and object-oriented style of programming. Towards this last point, we note that the widely acclaimed Map-Reduce model, owing to the functional nature of the programming framework, does not provide to the programmers the familiar Abstract Data Type of associative containers.

In this paper, we present PDQCollections \(^1\) – a novel set of data structures, coupled with a computation model, for exploiting data parallelism in associative data sets. PDQ-Collections is a comprehensive library of collection classes that implement the native associative container interfaces: map, multimap, ordered map, ordered multimap, set, sorted set and others. We have also proposed a computation model, which we refer to as the Split-Replicate-Merge (SRM) model, that transparently and efficiently supports and exploits par-

\(^1\)PDQ could stand for Processes Data Quickly
allelism in multi-core as well as distributed environments, over the data scales that range from memory-bound to disk-backed. We have shown an equivalence between the SRM and the map-reduce (MR) model by mutual reducibility, that is, any problem that can be solved by one model can be solved by another.

The distinguishing characteristic of our programming model is **encapsulating the parallelism within the data structure implementations** (the PDQ Collections classes) rather than the program code. Object oriented programming encourages encapsulating the distinguishing characteristics within the implementation of the objects, while providing familiar interfaces to the programmer. For example, Berkeley DB’s StoredMap abstracts the knowledge that the data is backed on disk, while providing the familiar Map interface; GUI libraries hide the platform specific characteristics behind the implementation, while providing the same widget interface; and the Remote Method Invocations hide the fact that the objects are remotely located. However, traditional approaches for data parallel programming have instead relied on modifying the computation code, either by extending the grammar of a language (e.g. the parallel for loop), or enforcing alternate paradigms (e.g. functional programming in Map Reduce).

By encapsulating parallelism within the data structures, the programming model is able to cleanly separate out the code for actual computation and the code for parallelization. Consider the analogy of Semaphore vs. Monitors: semaphores require introducing the logic of concurrency within the code, as opposed to monitors where all logic is captured separately within the monitor object. Similarly, with our programming model, the logic of parallelism is not interspersed within the code, rather it is expressed separately. As a concrete example, we shall discuss the Frequent Pattern (FP) growth algorithm [1] in Section 5.2, where the input data is processed to generate a FP tree data structure. The traditional methods of parallelism would require modifying the tree-generation code by embedding parallel constructs within the code. With our model, the programmer only needs to separately define how to “add” two trees. We believe, our design choice lead to a cleaner separation of functionality, resulting in a flexible and robust system architecture.

The SRM computation model employs the divide-and-conquer strategy for parallelization. We have proposed a novel shared-nothing strategy for merging associative containers, which obviates the need for any explicit or underlying synchronization, concurrent access or transactional processing. By avoiding locking overheads, PDQ containers significantly outperform the locking-based data structures, such as ConcurrentMaps. The same strategy, when applied to disk-bound workloads, is a cache-free design and ensures that data is always read and written to disk sequentially. By avoiding cache-miss penalties and random disk seek overheads, PDQ containers significantly outperform the traditional disk-backed data structures, such as BerkeleyDB.

The PDQ collection classes are versatile in managing data over a wide range of scale. Initially, the data is stored in-memory, but as the size of the container grows, the data is transparently and automatically spilled over to the disk, where it is stored in a format most efficient for further processing. The disk-backed form of the container objects have the capability to store data on multiple disks, if present, to improve the disk I/O speeds. They have a distributed implementation as well, where the data can be flexibly stored at common location (e.g. SAN, NFS) or in a distributed storage.

The rest of the paper is organized as follows: in Section 2 we provide a formal description of programming model and in Section 3 we discuss our implementation. We conduct a comparative design performance analysis study in Section 4 with concurrent data structures (for memory bound computation), BerkeleyDB’s StoredMap (for disk-backed computation) and Hazelcast (for distributed computation) to highlight the salient characteristics of our programming model that explains how PDQ significantly outperforms these traditional data structures. We illustrate several applications of our PDQ in different fields of Computer Science in Section 5 and conclude in Section 6.

2. DESIGN

In this section, we provide a formal description of programming model, including the data model, which formalizes the problems that can be parallelized with our model, the Split-Replicate-Merge computation model, which describes the strategy for parallelism, and the programming APIs. We also show an equivalence with the Map Reduce programming model.

2.1 Data Model: Mergeable Maps

This section formalizes the data-centric problems that can be parallelized using our model. We begin by defining an associative container as a mapping function from key space \( \mathbb{K} \) to value space \( \mathbb{V} \):

\[
A : \mathbb{K} \rightarrow \mathbb{V}
\]

Different variants of associative containers can be interpreted by adapting the semantics of the mapping function. For example, the container is a **multimap** if the value space \( \mathbb{V} \) is a set of collection of values, or a set if a Boolean space. Similarly, the container is an array if the key space \( \mathbb{K} \) is a range of natural numbers, or a matrix if a set of tuples. Other complex containers, such as graphs, can similarly be interpreted.

Next we define a computation as a function with input in the domain \( I \) (which may be an associative container) to produce as output an associative container \( A \):

\[
C_{seq} : I \rightarrow A
\]

We have explicitly formulated the computation to be a sequential function. For the sake of simplicity, we have also assumed only one input and one output for this computation. The extension to the general case is trivial, and will be discussed later.

Our objective is to develop a data-parallel programming paradigm for this computation that simultaneously satisfies the following requirements:

1. the computation can be parallelized and scaled across multiple cores and processors, in shared-memory concurrent as well as distributed processing contexts.
2. the parallel computation can handle input and output data that scales from memory-bound to disk-backed to distributed storage.
3. both forms of scaling (processing and data) can be achieved by using the unmodified sequential form of the computation.

We claim that the above requirements can be satisfied for the class of computation problems with inherent parallelism that can be formulated as follows:

If \( I_1 \subseteq I \) and \( I_2 \subseteq I \), such that \( I_1 \cap I_2 = \emptyset \), and \( C(I_1) \rightarrow A_1, C(I_2) \rightarrow A_2 \). Then the following must be equivalent:

\[
C(I_1 \cup I_2) \iff A_1 \oplus_{\text{merge}} A_2
\]

Where the merge operator \( \oplus_{\text{merge}} \) is defined as:

\[
(A_1 \oplus_{\text{merge}} A_2)(k) = \begin{cases} 
A_1(k) & \text{if } k \in A_1, k \notin A_2 \\
A_2(k) & \text{if } k \notin A_1, k \in A_2 \\
A_1(k) \oplus A_2(k) & \text{otherwise}
\end{cases}
\]

Here, \( \oplus \) is some user-defined operator. It is required, however, that this “add” operator is both commutative and associative.

Intuitively, the above formulation specifies that the inherent parallelism in the computation is such that if partial results were obtained by processing partial inputs, then the result of the combined inputs can be produced by merging the partial outputs.

We shall later prove that this formulation of parallelism is equivalent to the map-reduce programming model, that is, any problem that can be solved by map-reduce can be formulated in this manner. Let us now review some examples to illustrate this model of parallelism in action.

**Word Count:** Given a document, find the number of times each word appeared in the document. The input is a list of words, and the output is a simple map of string to integers. This problem can be parallelized since we can partition the input list into sublists, execute the word count program separately on each sublist, and merge the generated output maps by defining \( \oplus \) as integer addition operator.

**Inverted Index:** Given a map of keys to a list of words (the input), generate a map of words to the list of keys they belonged to (the output). Although we have not yet discussed how the maps are partitioned, for now we imagine that the key space is partitioned and the sub-maps are created with key-value mappings for each partition, and the program to invert indices is executed separately on each sub-map. If we define \( \oplus \) as the list concatenation operator, the merged output will indeed be the inverted index of the original map.

**Finding Average:** Find the average from the list of numbers. Since the average of averages is not the average, we need to define a custom data type with two fields, \( \text{sum} \) and \( \text{count} \). We also define the \( \oplus \) operator for this data type that adds the two fields. The output of the computation is a “singleton” container, which contains only one element. The computation, then, takes as input a sublist of numbers and computes the result into this data type. These partial results are ‘added’ together to produce the final result.

### 2.2 Computation Model: Split, Replicate, Merge

In this section we describe our parallel computation model, called the Split-Replicate-Merge model, to exploit parallelism for the class of problems identified in the previous section. As previously, for the sake of simplicity we assume the computation with single input and single output. The model executes as follows: the input is first \textit{split} into \( M \) partitions. The number of splits is at least equal to the number of processing units available, although it can be greater if it is desired that each partition should be of some manageable size (for example, if each partition has to be processed entirely within memory). Next, the output data structure is \textit{replicated} \( M \) times. Each replication is an initially empty data structure of same type as the output data container type. These splits and replicas are paired together, and \( M \) instances of computation are executed (one for each pair) concurrently. Finally, the replicas, which now contain the results of the computation from the respective splits, are \textit{merged} to produce the final output. The following pseudo code outlines this process:

\[
\text{PDQRun}(C_{\text{seq}}, \text{input}, \text{output}) \rightarrow \{ \text{split}_i, 1 \leq i \leq M \} \\
\text{REPLICATE(output)} \rightarrow \{ \text{rep}_i, 1 \leq i \leq M \} \\
\text{parallel for}(i: 1 \text{ to } M) \text{ do} \\
\quad C_{\text{seq}}(\text{split}_i, \text{rep}_i) \\
\text{end} \\
\text{output} \leftarrow \bigcup_{i=1}^{M} \text{rep}_i \quad \text{// Merge}
\]

**Listing 1: The Split-Replicate-Merge Computation Model**

Note that there is a single synchronization point in the entire model, which is the implied barrier at the end of the parallel for loop.

This model is easily extended to support multiple input and output parameters, as well as “shared” parameters. With multiple inputs, the implementation must ensure that each input generates equal number of splits, and the respective split from each input share the same keyspace. The multiple outputs are replicated, and later merged, independent of each other. The shared arguments are neither split nor replicated; the same copy of the object is provided to all the computation instances.

In section 3, we shall discuss efficient implementations for splitting and merging. We show that associative containers can be split in \( O(1) \) time, and the replicas can be merged in parallel as well. We shall also see how this computation model can be easily adapted to handle volumes of data and to operate in distributed execution mode.

### 2.3 Programming Model

Whereas the computation model described previously is meant to be transparent to the programmers, it is the programming model, described herein, that is visible to the programmers. This model describes how the computation must be coded for parallelization.

We describe the programming model by means of the Word Count program, outlined in Listing 2. Although the example shown is coded in Java SE 6, the model can be applied to any programming language that supports Object Polymorphism and function references, either as closures, function pointers or functors, and, of course, a multi-threaded environment.

```
1 @Parallel(name="wordcount")
2 public void wordcount(
3   @ArgIn java.io.FileReader input,
4   @ArgOut java.util.Map<String, Integer> output) { 
5     BufferedReader reader = new BufferedReader(input); 
6     String line; 
7     for ((line = reader.readLine()) != null) {
```
The first segment of the code (lines 1 through 14) is the actual computation. Notice that the method arguments are the native data types, and the method body is entirely sequential. The @ArgIn and @ArgOut Java annotations indicate that the arguments are split into the input and replication of output, respectively. The @Parallel annotation merely serves as a naming marker for the method, which we utilize to extract the method reference (java.lang.reflect.Method) using the Java reflection APIs. This is only a user-friendly way to name a method, other techniques such as function objects would have sufficed as well.

Next, we instantiate the objects for the computation - a FileReader as input, and a Map of String to Integer as output. Here, instead of using the Java native classes, we create a naming marker for the method, which we utilize to extract the method reference (java.lang.reflect.Method) using the Java reflection APIs. This is only a user-friendly way to name a method, other techniques such as function objects would have sufficed as well.

Observe this salient feature of our programming model: to parallelize a sequential program, the programmer only needs to replace the programming language’s native input/output container, or File I/O, classes with the counterpart classes from the PDQ library; while the actual computation code itself remains unmodified.

The parallel execution is invoked in line 22 where the reference to the computation method and the input/output objects are passed to the PDQ library. This library call executes the Split-Replicate-Merge computation on the provided arguments. The behavior of the computation model execution can be controlled by the configuration parameters, shown in line 17.

### 2.4 Equivalence with the Map Reduce Model

Map-Reduce is a widely acclaimed programming model for processing large data sets. This is a functional-style model that processes data in two stages of map and reduce phases. We show this model is equivalent to the SRM model by proving that the two models are reducible to each other.

Consider an arbitrary problem that can be parallelized with the map-reduce model. There exists, then, a map function that transforms the key-value mappings of the input into an intermediate list of mappings in a different domain:

\[ \text{MAP} : (k_1, v_1) \rightarrow \text{list}(k_2, v_2) \]

This intermediate output is processed to collect all values for a given key together, and the key space is sorted. This serves as the input to the reduce function, which generates the final result:

\[ \text{REDUCE} : (k_2, \text{list}(v_2)) \rightarrow \text{list}(k_3, v_3) \]

This problem can be formulated by the Split-Replicate-Merge model in two execution steps, as shown in Listing 3. In the first step, the input map is produced to process a SortedMultimap of intermediate values, which is further processed in the second step to generate the output result.

```java
@Parallel(name="step1")
void step1(Map<K1, V1> input, SortedMultimap<K2, V2> intermediate) {
  for (k1, v1 in input)
    List<K2, V2> list = MAP(k1, v1);
    for (k2, v2 in list)
      intermediate.add(k2, v2);
}

@Parallel(name="step2")
void step2(SortedMultimap<K2, V2> intermediate, SortedMultimap<K3, V3> output) {
  for (k2, list_v2 in intermediate)
    List<K3, V3> list = REDUCE(k2, list_v2);
    for (k3, v3 in list)
      output.add(k3, v3);
}

void main(Map<K1, V1> originalInput) {
  Map<K1, V1> input = new PDQMap<K1, V1>(originalInput);
  SortedMultimap<K2, V2> intermediate = new PDQSortedMultimap<K2, V2>();
  SortedMultimap<K3, V3> output = new PDQSortedMultimap<K3, V3>();
  PDQ.run(this, "step1", input, intermediate);
  PDQ.run(this, "step2", intermediate, output);
}
```

### Listing 3: Reducing SRM model to Map-Reduce

Conversely, consider an arbitrary problem that can formulated with our data and computation model, that is, the following method is defined:

\[ \text{SRM}(\text{Cseq fn }, \text{Map}<K1, V1> \text{ input }, \text{Map}<K2, V2> \text{ output}) \]

This problem can be reduced to the map-reduce model as follows:

```java
class Mapper {
  Map<K1, V1> input;
  void map(K1 k, V1 v) {
    input.put(k, v);
  }
}

class Reducer {
  void reduce(K2 k2, List<V2> list_v2) {
    V2 sum = fold(@+, list_v2);
    EMIT(k2, sum);
  }
}
```

### Listing 4: Reducing Map-Reduce model to SRM
In the mapper class, the key-values mappings are first collected into a Map. The output object is replicated, and the computation is invoked on these two objects. The mappings from the replica (which served as the output container for the chunk of input provided to this Mapper) are emitted to the reducer. At the reducer, the list of values for a key are aggregated together using the \( \oplus \) operator, which is then emitted for the given key.

We have thus shown that that two programming models can be emulated with each other and, consequently, are equivalent.

### 2.5 Related Work

Piccolo [2] is a data-centric programming model with a similar objective of parallel and distributed computing using associative containers. The system ensures execution locality by partitioning the input across machines with a user-specified partitioning function. While each process processing its partition locally to produce the corresponding output, it also has access to global state. In other words, it can attain a globally consistent view of the value corresponding to an output key. Piccolo achieves this by introducing a user-specified accumulation function akin to our add operator. When called by the user, this operator can “accumulate” the values for a specific key across all processes. This differentiates the PDQ programming model from Piccolo’s. While Piccolo offers fine-grained access to global state, it requires user control over the merge procedure. On the other hand, PDQ executes the merge operation transparent to the user. PDQ also reduces the merge overhead partitioning the output across a SuperMap. Finally, in contrast to Piccolo that only supports, memory-constrained outputs, PDQ transparently allows in-memory, out-of-core and distributed processing and output data structures.

### 3. Implementation

Our primary goal for implementation is efficient processing of data by maximizing the CPU utilization as well as by maximizing the disk I/O. Our implementation targets data sizes up to few terabytes, and optimized for multi-core systems and small to medium sized clusters. Concerns such as large scale cluster management, fault tolerance are not addressed. Our secondary goal is transparent parallelism, that is, the computation can be coded using programming language’s native data type and sequential style of programming, and does not require any modifications to language grammar. In the following, we describe our data structures that can transparently switch from memory bound, to disk-backed and distributed operational behavior. We have implemented our programming model as a library in Java SE 6.

#### 3.1 Data Structures

The PDQCollections library provides a suite of data structure implementations that are meant to be drop-in replacement for their counterpart native implementations. These include the typical container data types that implement the Map, Multimap, SortedMap, SortedMultimap, Set, SortedSet, List and other interfaces, as well as File I/O classes, that inherit the FileReader and FileInputStream classes.

Recall the computation model outlined in Listing 1, where we discussed that the input containers are partitioned into splits, and that the output containers are first replicated, and later these replicas are merged back. This behavior is enforced by requiring the input and output containers to implement the Splittable and Mergeable interfaces, respectively. The signatures of these two interfaces are shown below:

```java
interface Splittable {
    Iterator<Object> split();
}
interface Mergeable {
    Object replicate();
    void merge(Object[] partials);
}
```

The splits of an input container and the replicas of an output container are objects of the same abstract data type as the respective input and output containers. In addition, the replicated objects are in-memory data structures. The collections classes support both interfaces, while the file reader classes only implement the Splittable interface.

#### 3.1.1 PDQ Collections Classes

Recall that an associative container can be viewed as a mapping from key space to value space: \( A : K \rightarrow V \). If the execution environment has \( N \) threads, then the key space is exhaustively divided into \( N \) non-overlapping subspaces and a delegate associative container is created for each of these key subspaces. In other words, the PDQ container is a wrapper around \( N \) associative containers, each managing a portion of the keyspace, as defined below:

\[
A_{PDQ} : f_{PDQ}(k) = \begin{cases} 
    f_{assoc1}^1(k) & \text{if } k \in K_1 \\
    f_{assoc2}^2(k) & \text{if } k \in K_2 \\
    \vdots & \\
    f_{assocN}^N(k) & \text{if } k \in K_N 
\end{cases}
\]

where \( K_1, K_2 \ldots K_N \) are the \( N \) divisions of the key space, and \( f_{assoc1}, f_{assoc2}, \ldots f_{assoc} \) are the \( N \) delegates. As an example, in an execution environment with four threads, a PDQMap wraps four java.util.HashMap delegates. For an operation on a key \( k \) (for example, get(), put(), contains(), etc), the key subspace \( K_i \) is identified that contains this key, and the operation is applied to the associated delegate \( f_{assoc}^i \).

One simple strategy for partitioning the key space is by partitioning the hash code range of the keys, that is, a key \( k \) will belong to subspace \( hash(k) \mod N \). Although, it must be ensured that this hashing function is different from functions that may be used internally by the associative container (for example, HashMap).

There are several benefits to this wrapped delegates design. First, the Splittable interface is trivially implemented: each delegates is a split! Notice that our approach is not to split the container after it is populated, rather to create the splits beforehand and populate them incrementally. This is an easier as well as an essentially zero-cost operation.

Secondly, this design allows for merging of the partial results in parallel. Each PDQ Collection implements the Mergeable interface, where the replicas themselves are wrapped collections of delegates. That is, for an execution environment with \( N \) threads, the PDQ collection wraps \( N \) delegates, as shown below:

\[
A_{PDQ} := \{D_1, D_2, \ldots, D_N\}
\]

where \( D_i \) is the \( i^{th} \) delegate of the PDQ collection. This
collection object will generate \( N \) replications, which themselves wraps \( N \) containers, as shown below:

\[
R_i := \{ D_{1i}^{R_i}, D_{2i}^{R_i}, \ldots, D_{Ni}^{R_i} \}
\]

\[
R_2 := \{ D_{12}^{R_2}, D_{22}^{R_2}, \ldots, D_{N2}^{R_2} \}
\]

\[
\ldots
\]

\[
R_N := \{ D_{1N}^{R_N}, D_{2N}^{R_N}, \ldots, D_{NN}^{R_N} \}
\]

where \( R_i \) is the \( i \)th replication, and \( D_{ji}^{R_i} \) is its \( j \)th delegate. Now the merge operation can be applied in parallel by shuffling the replication delegates and grouping them according to the key subspace they manage. That is:

\[
\text{MERGE} \left( \mathcal{A}_{PDQ}, [R_1, R_2, \ldots, R_N] \right):
\]

\[
\begin{aligned}
\text{parallel for } (i: 1 \text{ to } N): \\
D_i &\leftarrow D_{1i}^{R_i} \uplus D_{2i}^{R_i} \ldots \uplus D_{Ni}^{R_i} \\
\end{aligned}
\]

Similar parallelization benefits are achieved when the container data is stored on disk, as we shall see in Section 3.3. In this case, the various activities such as sorting, serialization and writing to disk can be applied in parallel.

### 3.1.2 PDQ File I/O Classes

The PDQ library provides replacements for the java.io file I/O classes. In particular, the PDQFileReader and PDQFileStream classes support reading characters and bytes, respectively, from files. Both implement the Splittable interface, and, therefore, can act as inputs to the computation. Similarly, the PDQFileWriter and PDQFileOutputStream classes support writing characters and bytes, respectively, and implement the Mergeable interface.

The primary consideration in reading files is blocksize – the size of the chunk of input that is processed by a single thread. The input file is partitioned into chunks of size blocksize each, and these chunks serve as input splits to the computation. If the file contains variable length records, the splits coincide with record boundaries. Recall our implementation goal which states that the computation for each input split must happen within memory. It is, therefore, necessary to ensure that the blocksize must not exceed the limit where the output generated (for that split) cannot be entirely contained within memory. Since the exact limit is dependent on the processing environment, this parameter is user-defined.

The PDQ file output classes implement the Mergeable interface by generating replicas that are standard java.io output classes for files where the filename is appended with a part identifier (e.g., filename.part0000).

### 3.2 Thread Pool Execution Mode

This section describes the PDQ library’s shared-memory parallel execution mode of the computation. As the name implies, the concurrency is achieved by thread pools, and we make use of the java.util.concurrent.ThreadPoolExecutor service for the thread pool creation and management.

This execution mode is applicable when the output of the computation can be contained within memory. For a pool of \( N \) threads, the input is first split into \( N \) partitions. If the input is a file, then the blocksize is automatically computed to create \( N \) chunks. The output containers are replicated \( N \) times as well, and the \( N \) instances of computations are assigned to the thread pool. At this point, the main thread invokes the thread pool and waits until all threads within the pool complete, which can be viewed as an implied barrier for the main thread. Subsequently, the main thread shuffles the delegates of the replicas for merging, in accordance with the process described previously, and assigns these merge tasks to the thread pool. There is a second implied barrier, at the end of which the partial results are merged back into the output containers.

As the cost of splitting, replicating and shuffling/distributing delegates is essentially negligible, this execution mode effectively utilizes all the processing cores throughout the program execution.

### 3.3 Disk-backed Computation

The previous execution mode is extended to handle cases when the output of the computation is too large to fit in memory, thus necessitating the need to store the output data on to the disk. In the following description, we refer to the process of storing the output disk on disk as externalization.

The first difference from the memory-bound execution mode is that the number of the input splits, \( M \), can be greater than the number of processing cores, \( N \). In this case the computation unfolds in multiple iterations. In the first iteration, the first \( N \) splits are processed, in accordance with the thread-pool execution mode described previously; in the second iteration, the next \( N \) splits are processed similarly and the results from this iteration are merged back into the previously accumulated results. This process continues until all \( M \) splits are processed, at the end of which the accumulated result (obtained by merging partial results from each iteration) will be the final output of the computation.

At the same point, the accumulated results will be large enough to overflow the memory. Before reaching this state, we enforce externalization of memory contents to the disk, thus freeing up memory for further processing. At the end of each iteration, the library determines whether the memory contents must be externalized to the disk. In our implementation, we check the currently available memory in JVM (via the freeMemory() method of the java.lang.Runtime class) and decide to externalize if it falls below a threshold.

When externalizing the output container, each of the delegates of the output container are externalized in parallel.
The process involves sorting the key space of the delegate container, if it was not already sorted, and serializing the key-value pairs to a flat file. We use the Java’s native serialization framework for externalizing data. At this end of this externalization step there will be $N$ files on disk, which contains the memory contents of the $N$ delegates respectively, with each file written in parallel. The next externalization step will write data into a new set of $N$ files.

When all the $M$ splits are processed, these files are merged into a Mapfile. A mapfile is a file that stores a sorted list of key-value mappings along with an index (typically kept in memory) that maps a subset of the keys to the offset within the file where they are located. To retrieve the value for a key, the index is consulted to locate the offset of the closest key, and then the file is read sequentially from that offset until the said key is located. Since the externalized files are already sorted on disk, the $O(N)$ disk merge algorithm is used to merge them into the mapfiles. These mapfiles, subsequently, serve as the delegates for the output container.

Figure 1 illustrates this execution mode as a sequence diagram. The outer loop in the diagram represents the iterations, and the opt block represents the optional step of externalizing the delegates to disk.

Observe that the actual computation, at each iteration, is entirely memory bound. While the merged results may be externalized to the disk, the output data is always within memory while the computation is being executed. This eliminates any need for disk-based data structures, which are substantially more costly than the in-memory structures. This design also not employ any form of caching disk data within memory, and thus avoids caching relating scalability overheads. Finally, the data is always read and written sequentially from the files, which also eliminates random disk seek overheads.

If multiple hard drives are available, the disk I/O can be further improved by storing the externalized files onto these multiple disks. In the thread pool environment with $N$ threads, each externalization step generates $N$ files, and at the end of the computation, there are $N$ delegate mapfiles for the output container. These files can be spread across the hard drives, and given that the file I/O operations are parallelized in our implementation, it follows that the data can be read or written simultaneously from all the hard drives.

3.4 Distributed Mode

In this final section, we describe our implementation strategy for distributed execution. As noted previously, our design goal is catered towards small to medium sized clusters. The implementation follows a master-slave architecture, where the master executes the main application and requests the workers to participate in the computation. The main application jar file is added to the classpath of the workers to ensure the visibility of the classes that define the computation.

When the main program, executing at the master node, invokes the computation via the $PDQ.run(...) \text{ method}$ (see line 22 in Listing 2), the PDQ library contacts the worker daemon processes and prepares for the distributed execution. The master first notifies each worker their unique rank, which is a number between 1 and total number of workers. The workers use the rank to determine the portions of the input that they are responsible for processing. Subsequently, the master serializes and transmits over the wire the computation method and its input/output arguments. At this point, the workers have sufficient information to begin processing their share of the input. Each worker executes independently of others, and after processing its input, serializes and transmits the results back to the master. The master may choose to merge the results gathered from the workers either locally (if the data is small enough), or employ the slaves again for distributed merge.

A brief note on sharing the input and output data: in simple cases, the master and workers may exchange the entire data set over the wire. This simple approach does not require any cluster configuration, however, it is not the most efficient method of sharing data, for example, it adds the serialization and deserialization overhead. Alternatively, the data may be stored on shared file system, for example, Network File System (NFS), or distributed file system, for example, Hadoop Distributed File System (HDFS). In this case, only the reference to the file needs to be shared.

4. DESIGN ANALYSIS

In this section, we compare the PDQ programming model against representative data structures for the memory-bound, disk-bound and distributed computations. Our objective is to gain an insight into the fundamental nature of the PDQ computation model and illustrate the impact of our design choices towards overall performance of data-centric computations. We are not interested in raw performance comparisons between two systems, as those statistics vary widely based on parameters of implementation, programming style, efficiency of the libraries used (e.g., threading, serialization, networking). Instead, we wish to expose those salient characteristics of our model that makes the PDQ library consistently outperform traditional designs, irrespective of actual implementations under consideration.

The summary of our findings is as follows: with memory-bound workloads, the PDQ library avoids locks and any other form of synchronization, and thus significantly scales better with number of processing cores than the lock-based data structures, such as ConcurrentMap. For large disk-bound workloads, the PDQ library does not employ any form of caching and always accesses the disk with sequential read and write requests. In contrast, the disk-backed data structures, such as BerkeleyDB and KyotoCabinet, are two-tiered with in-memory cache for the on-disk B-tree, where the efficacy of the cache diminishes with smaller cache ratio. By avoiding the cache miss penalties and random disk seek overheads, the PDQ library significantly scales better with data size. Finally, with distributed workloads, the PDQ library accesses remote data resources sequentially and only once, and is thus able to significantly outperform distributed data structures.

Single-host experiments were conducted on a machine hosting a dual hyper-threaded 6-core Intel Xeon 2.30GHz processor with 32GB of memory and a 500GB of disk. The machine runs CentOS 6.3 with kernel version 2.6.32. For distributed experiments, we used a cluster of eight machines, each of which is comprised of two quad core Intel Xeon 2.33 GHz processors with no hyper-threading, 8GB memory and a 200GB hard drive, with CentOS release 5.8 and 2.6.18 kernel. The cluster nodes are connected with infiniband switching as well as 100 MBps ethernet. All machines have Java Hotspot 1.6 runtime available.
We begin our analysis with the thread pool execution mode (section 3.2). “Term frequency” is one of the common applications of maps, which counts the number of occurrences of each unique term in a document. An implementation, using the PDQ library, was discussed earlier in Listing 2, which we compare against the ConcurrentHashMap class from the java.util.concurrent package, as well as a critical section (using Java’s synchronized keyword) around a regular java.util.HashMap. Our dataset comprised of a collection of web documents containing a total of 300 million words, from a set of 5.2 million unique words [3]. We investigate how efficiently these three implementations exploit varying levels of concurrency.

We measure the performance as the ratio of parallel to sequential execution time (factor speedup), which is shown in Figure 2 for the three implementations, for varying number of threads ranging from 1 through 12. Notice that, regardless of the type of data structure selected, each of the k threads processed only 1/k of the input data. This implies that difference between the theoretic speedup factor of k and the observed speedup can be attributed to the overhead of maintaining consistency of the output data (although, there are other factors in play as well, such as garbage collection, memory allocation, context switches etc, but we believe their contribution is relatively small). From the figure, we observe that the concurrent and synchronized maps are constrained by their lock-based approach to data consistency. This locking overhead is significant and in fact lead to a runtime degradation in comparison to sequential data processing for this benchmark problem. In contrast, the divide-and-conquer approach of the PDQ library offers an alternative mechanism to achieve data consistency, which we have shown to outperform locking-based maps. Furthermore, the PDQMaps don’t exhibit diminishing speedup with increasing number of threads as was observed with concurrent and synchronized maps.

### 4.2 Disk-backed Data Structures

To analyze the merits of the PDQ library in processing large datasets (Section 3.3) we compared it with embedded disk-based database implementations from Berkeley DB Java Edition (BDB JE) and Kyoto Cabinet (KC). To focus exclusively on the disk I/O performance, we have limited the benchmarks to single-threaded execution. In any case, we have shown previously that locking-based data consistency (which is employed by BDB and KC) does not scale for data-centric computations.

“Reverse index” is another common application of multimaps, which creates a mapping from values to their keys in the original index. We used the dataset from DBpedia [4] which lists the URLs of English Wikipedia pages and the URLs of other Wikipedia web pages they point to. The reverse index is a mapping from a page URL to URLs that link to it. The input is a 21GB text file, where each line represents one outgoing link and has the format “(url of source page) ... (url of linked page)”. The input was read using the PDQFileReader class, which automatically partitions the input file, and the output was stored in a PDQMultimap object in case of the PDQ library, StoredMap object in case of BDB and the DB object for KC.

Figure 3 (next page) is a time series of the disk read and write activities (top plot), CPU utilization (middle plot) and rate at which the input data is processed, for both PDQMultimap (the left plots) and BDB StoredMap. In case of PDQMultimap, we can clearly observe the cycles of processing input (when the disk read rate and data processing rates peak) and externalizing the intermediate results to disk (when the disk write rate peaks while the CPU waits as no data processing occurs). Around the 300s mark, we observe this cycle ends and the external merge begins. During the external merge, data is read and written in rapid succession while CPU utilization observes a marked drop.

In contrast, the BDB StoredMap observes good performance only when the database is entirely contained in the memory cache. During the first 200 seconds, StoredMap utilizes the CPU well as data is read from the disk and processed at a high rate. Once the database is too large to fit in memory, we observe a spike in the disk write rate as the contents of the cache are flushed to disk. Subsequently, the disk read and data processing rates drop significantly and none of these performance markers improve over time. Once the database is too large to fit in memory, only portions of it can be stored in the cache at a time, and cache page swaps contribute substantially to disk I/O. This is verified by high percentage of time that CPU spends in “waiting” state. To summarize, once the size of output data exceeds the threshold of RAM, the StoredMap spends a lot of time bringing relevant data items from disk to cache and dumping stale cache entries to disk.

There is a second fundamental limitation of StoredMap which is not apparent from the previous plots. As the size of data on disk increases, the effectiveness of cache decreases (since a random entry is less likely to be found in cache) and consequently, the performance further degrades. This is verified in Figure 4 which shows that the execution time increases exponentially with the input data size. PDQMultimap, in contrast, offers linear scalability.

### 4.3 Distributed Data Structure

We compared the distributed execution mode (Section 3.4) of the PDQ library against a distributed implementation of the Map container type from Hazelcast [5]. The hazelcast map partitions the contents across the multiple JVMs based on the hashcode of the keys. The likelihood that a given key is present locally decreases in inverse proportion to the size of the cluster. Since the version of Hazelcast we used only supported in-memory data, we used the same “Term frequency” benchmark problem that we consid-
Figure 3: Performance Comparison of PDQ and Berkeley DB (BDB) for Disk I/O Rate (top), CPU Utilization (middle) and Processing Rate (bottom)

Figure 4: Scalability Performance of PDQMultiMap vs. Berkeley DB Map

Figure 5: Performance Comparison of PDQMap vs. Hazelcast

Notice that it is not just the amount of data transferred over the network that influences the performance, it is also the network access pattern. In case of Hazelcast map, remote data is accessed while the computation is being executed, which implies that the processor would remain idle while waiting for the remote data to be fetched. In contrast, the PDQMap transfers data only twice: first, the initial input workload is distributed by the master to the slaves, and secondly, by the slaves when they transmit their computed results back to the master. In between these two network accesses, the processors are fully occupied with computation and never wait for network activity. The figure 6 illustrates the breakdown of the execution times with PDQMap into four phases. As a summary, whereas distributed maps like...
Figure 6: Breakdown of execution times for PDQMap by operation

Hazelcast are latency-bound, the PDQMaps are throughput-bound. What makes this observation pressing is that it is easier to cater to network throughput bound operations. For example, the data can be compressed, faster network interconnects may be used in the cluster, network parameters may be tuned and so on.

5. APPLICATIONS

In this section, we illustrate several applications of our PDQ library in the different fields of Computer Science. The k-Nearest Neighbor Classification in Machine Learning and the FP Tree Growth Algorithm in Data Mining illustrate how the PDQ programming model transforms the problem of parallelization to that of “adding” data structures. The PageRank algorithm for Graph Processing illustrates the efficient operation of PDQ data structures for large datasets in distributed environments. The PDQ programming model has a broad scope of applicability, which we illustrate by examples in the relational processing of structured-data and incremental processing of log data.

5.1 Machine Learning: k-Nearest Neighbor based Classification

We consider the problem of handwriting recognition, and specifically, recognizing handwritten numerical digits. Our training dataset consists of 42,000 images, where each image is 28x28 pixels and represents one handwritten digit [6]. Each of these images is correctly labeled with the represented digit. The machine learning objective is stated as: given a new image, ascertain the digit it represents. We employ the k-Nearest Neighbors based Classification approach, which employs the following steps: (a) represent each sample item as a vector, (b) define distance between two vectors (we used the Euclidean distance), (b) determine the k vectors from the training samples that are closest to test vector, and (d) take the majority vote of labels of these k selected samples.

We defined a LimitedPriorityQueue that contains no more than k entries, that is, if a new entry is less than the greatest entry in queue, the former is inserted and the latter is removed. To execute the computation in parallel, we simply need to define the $\text{add} \left( \text{first}, \text{second} \right)$ operator for the LimitedPriorityQueue objects, as shown below:

```python
def add(first, second):
    result = first.copy()
    for d, label in second:
        result.insert(d, label)
    return result
```

We can now split the input training data, and apply the k-Nearest computation for each split. Each computation would produce a LimitedPriorityQueue object, which we “merge” as shown above, which is the final result. The parallel speedup on a 12-core machine, for varying number of processing threads, is shown in the Figure 7.

5.2 Data Mining: Frequent Itemset Discovery

Consider a retail store, where a customer purchases a set of items (a transaction). There are many such transactions, and we would like to determine which items are frequently bought together. The Frequent Pattern (FP) Tree growth algorithm is an efficient technique to discover the frequent itemsets. In the first phase of the algorithm, the input data is scanned and the FP tree is constructed, in the second phase the tree is mined for frequent item sets. We will focus on parallelization of first phase only.

Our strategy for parallelizing FP tree construction would then be to figure out how to “add” two trees. We omit the details, and only give a brief sketch on a recursive strategy for addition. The recursive function takes as input two tree nodes, one from each tree, and is initially invoked with the root nodes of the two trees. For each child of the node from the first tree, we check if it is present in the second tree. If present, we fuse these two nodes and recursively apply the function for these two nodes; and if absent, we simply copy the subtree rooted at the child node. An example is shown in Figure 8 (header table and node links are omitted for clarity). The left tree was constructed for transactions $\{A, B, E\}$, $\{A, C, D\}$ and $\{C, E\}$, while the transactions for the right tree were $\{A, D, E\}$, $\{E\}$ and $\{C, E\}$.

We applied our implementation on a 1.4GB dataset of 1.7 million transactions with 5.2 million unique items, and aver-
5.3 Graph Processing: Pagerank Algorithm

If we represent the web as a directed graph, with webpages represented as nodes and links as edges, then the PageRank algorithm [7] can be used to generate weights for each node, which measures the relative importance of that web page.

We applied the algorithm on the Wikilinks dataset [4], which is a graph of all English Wikipedia pages along with internal page links (21 GB text file). We processed this raw data and created a Multimap, which consumed about 8GB, when serialized on disk. This dataset is too large to be completely processed in-memory. With this example, we highlight the capability of our associative container implementations that can automatically and transparently transition their behavior from in-memory to disk-backed as well distributed execution without requiring any code changes to the program.

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>Time per Iteration (min)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20:38</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10:58</td>
<td>1.88X</td>
</tr>
<tr>
<td>4</td>
<td>5:04</td>
<td>4.06X</td>
</tr>
<tr>
<td>8</td>
<td>2:25</td>
<td>8.54X</td>
</tr>
</tbody>
</table>

Table 1: Speedup of PageRank algorithm on a Cluster

We executed the algorithm on a distributed cluster of machines, the size of which ranged from 1 through 8. Since the computation is mostly I/O-bound, we spawned only one thread per machine. The following table shows the speedup with respect to the cluster size.

5.4 Relational Processing of Structured Data

Let us consider a motivating example from the DBLP repository [8] of academic publications in Computer Science. We extracted two database tables: paper (paperId, authorId), which lists all published papers and their authors, and citations (paperId, citedpaperId), which lists citations for each paper. Equivalently, we can view them as multimaps of paperId to authorId, and paperId to citedpaperId, respectively. We chose multimaps, since each paper may have multiple authors, and may cite multiple papers.

Sometimes authors cite their own papers, or of their colleagues. Colleagues are the researchers that an author has co-authored some paper with, or equivalently:

\[ A \text{ is-colleague-of } B \Leftrightarrow \exists p, (p, A) \in \text{paper} \land (p, B) \in \text{paper} \]

We would like to find out, for each author, the percentage of papers they cited that were their own or of their colleagues (which should give us some insight into how clustered the research community is!). The following is an intuitive (but un-optimized) SQL statement to count the number of citations, for each author, that were their own or from colleagues:

\[
\text{SELECT author, count(paper)} \\
\text{FROM} \\
\text{(SELECT author, paper, coauthors, group_concat(coauthors) AS C)} \\
\text{FROM papers INNER JOIN citations AS paper} \\
\text{ON author = citedauthor)} \ AS Y1 \\
\text{INNER JOIN} \\
\text{(SELECT p1. uid AS paper, x1.coauthor AS coauthors)} \\
\text{FROM papers AS P1 INNER JOIN} \\
\text{(SELECT p1. authorId AS author, p2. authorId AS coauthor)} \\
\text{FROM papers AS P1 INNER JOIN papers AS P2} \\
\text{WHERE P1.paperId = P2.paperId)} \ AS X1 \\
\text{GROUP BY P1. authorId = X1.author)} \ AS Y2 \\
\text{GROUP BY author)}
\]

In context of associative containers, we can imagine a data-flow programming paradigm where the data can be transformed using some set of relational operators. If \( A : K \rightarrow V \) and \( B : K \rightarrow T \) are two associative containers, then we have defined for following operations (only a subset listed here, relevant for this problem):

\[
\begin{array}{ll}
\text{Operation} & \text{Definition} \\
\text{reverse}(A) : V \rightarrow K & v \rightarrow k \in \text{reverse}(A) \Leftrightarrow k \rightarrow v \in A \\
\text{join}(A, B) : K \rightarrow <V, T> & k \rightarrow \langle v, t \rangle \in \text{join}(A, B) \Leftrightarrow k \rightarrow v \in A \land k \rightarrow t \in B \\
\text{mix}(A, B) : V \rightarrow T & v \rightarrow t \in \text{mix}(A, B) \Leftrightarrow \exists k \in K, k \rightarrow v \in A \land k \rightarrow t \in B \\
\text{unique}(A) : K \rightarrow V & \text{remove duplicate values for each key}
\end{array}
\]

We omit the proof and only claim that each of these operations can be parallelized by the PDQ library both in multicore and distributed environments, for memory-bound as well as disk-bound workloads. We can now describe a counterpart sequence of data flow statements to compute the same result as the SQL statement, as shown in the following:

\[
\#T1 = (\text{Multimap from papers table}) \\
\#T2 = (\text{Multimap from citations table})
\]
5.5 Incremental Log Processing

In several contexts, logs are generated continuously (e.g. web traffic) and several statistics rely on cumulative processing of logs (e.g. top web pages visited). Processing the entire log dataset for these statistics, for every time a report is requested, may not be practical. Alternatively, statistics can be generated incrementally; however, in traditional programming approaches, custom code has to be developed for this purpose.

PDQ library offers a native and efficient mechanism for incremental processing of data. Recall that the our SRM computation model splits the inputs, creates replicas of output, applies computation to each split/replica pair, and finally merges the replicas back to the output container. So far we had assumed that the output container is initially empty. However, if the output container has some data to begin with, the replicas will be merged into the existing data. Since the PDQ data structures are natively Serializable, incremental processing is easily accomplished, as illustrated below:

```python
def first_report(input, output):
    PDQ.run(generate_report, input, output)
    saveFile(output, "timestamp_1")

def incremental_report(last_timestamp, input_since_last_timestamp):
    output = readFile("timestamp_" + last_timestamp) + last_timestamp
    PDQ.run(generate_report, input_since_last_timestamp, output)
    saveFile(output, "timestamp_" + (++last_timestamp))
```

After producing the first report, the output is saved to disk; while for the subsequent reports, the previously saved report is loaded from disk and used as output.

We used the WorldCup98 dataset [9] which is website request logs over 92 days, from 30 servers in four countries. Each log entry is 20B and the total data set size is 26GB. A report was generated each week, and the Figure 9 shows the execution time of incremental processing each week, compared with full processing of data.

6. CONCLUSION

We have presented PDQCollections, a library of associative containers that inherently encapsulate parallelism and a computation model that transforms the problem of parallelization to that of defining addition operator for value data types. We provided the Data Model, which describes what problems can be solved, the Split-Replicate-Merge Computation Model, our strategy for parallelism, the Programming APIs and shown the equivalence with the Map Reduce programming model. We have shown that the PDQ library can efficiently exploit parallelism in multi-core as well distributed environments, for in-memory as well as large datasets. We discussed our implementation that included the data structure design, the thread pool execution mode for shared-memory and memory-bound workloads, the disk backed execution mode to handle computations when output data is too large to fit in memory and finally, the distributed execution mode for small to medium sized clusters.

Our comparative study with concurrent data structures (for memory bound computation), BerkeleyDB’s StoredMap (for disk-backed computation) and Hazelcast (for distributed computation) highlighted the salient characteristics of our model that explained how it significantly outperforms these traditional data structures. We also illustrated the application of our library in several fields of Computer Science.

7. REFERENCES