

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

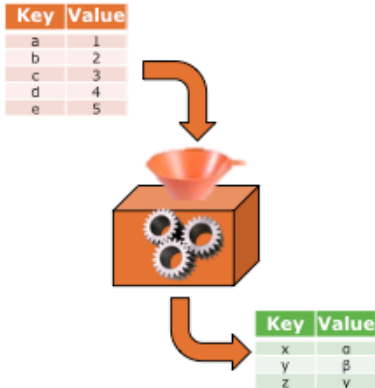
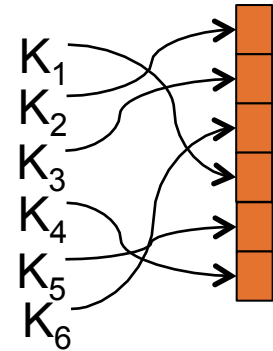
**Maneesh Varshney
Vishwa Goudar
Christophe Beaumont**

Jan, 2013

* PDQ could stand for Pretty Darn Quick or Processes Data Quickly

Associative Containers

- Also known as *maps*, *dictionaries*, *hash*
- Standardized, robust, easy-to-use mechanisms to store key-value mappings
- Java *Collections*, C++ *Standard Template Library*, C# *Collection Classes*



We consider **data-centric** programming problems, where the input or output or both are associative containers

- Commonly found in document processing, data mining and analytics, 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

- Parallelism can scale from multi-core to distributed environments.
- Data can scale from in-memory to disk-backed to distributed storage.
- Programming paradigm is as close as possible to the natural sequential programming patterns.

parallel for Loop Considered Unsuitable

- Most prominent data parallel programming paradigm
- As seen in OpenMP, Intel's Thread Building Block (TBB), C# .NET's Task Parallel Library (TPL)
- Why is it unsuitable for associative containers?

Input Side

- ❖ The index range of the for loop range is partitioned, and each segment is assigned to a thread within a pool of threads

```
parallel for (i: 1 to input.Length) do  
    ... = f(input[i])
```

- ❖ The input can only be arrays or matrices (i.e. memory addressable)
- ❖ Not suitable for maps which are content-addressable

Output Side

- ❖ Output generated by each thread must be protected against concurrent modification

```
parallel for (i: 1 to N) do  
    Begin Critical; output.add(...); End Critical  
    _____ or _____  
    output.concurrentSafeAdd(...)
```

- ❖ By using critical sections or concurrent safe data structures. Requires careful programming, and imposes excessive locking overhead.

- These libraries, by nature, are meant for shared-memory and memory bound workloads and offer little support or perform poorly with distributed systems and persistent data stores.

Our Approach: Smart Data Structures

- Object oriented programming encourages encapsulating the distinguishing characteristics within the implementation of the objects, while providing familiar interfaces to the programmer
 - 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
 - Remote Method Invocations hide the fact that the objects are remotely located
- These design patterns lead to a cleaner separation of functionality, resulting in a flexible and robust system architecture
- 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**)
- The distinguishing characteristic of our programming model is
 - encapsulating the parallelism***
 - within the data structure implementations***
 - rather than the program code***
- The PDQ library provides *implementations* for the associative container *interfaces*

Our Approach: Conquering the Divide-and-Conquer

- Any data parallel programming paradigm would:
 - *Divide* the problem into smaller segments
 - Process each segment concurrently
 - Combine the outputs (*conquer*) to produce final result
- We have defined a novel strategy for combining associative containers:

To combine maps A and B :

if mapping $k \rightarrow v$ exists in one map only \Rightarrow add the mapping to output

if mapping $k \rightarrow v_1 \in A$ and $k \rightarrow v_2 \in B \Rightarrow$ add mapping $k \rightarrow v_1 \oplus v_2$ to output
(where \oplus is some programmer defined operator)

- Most significantly, **the effort required in parallelizing a computation is simply defining how to add the value types**
- We show this to be a very powerful strategy that can:
 - Efficiently exploit parallelism in multi-core and distributed environments
 - Efficiently process data that is too large to fit in memory or a single hard drive
 - Shown to be significantly efficient than locking-based data structures (e.g. Java's ConcurrentHashMap) and caching-based on-disk data structures (e.g. BerkeleyDB)
 - More on this with examples later...

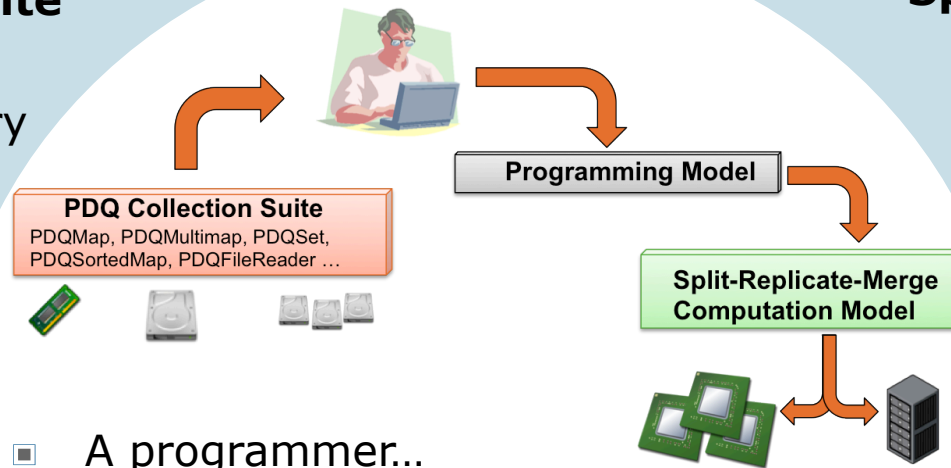
We present PDQCollections...

- ❑ a suite of smart data structures, and
- ❑ a computation model for exploiting data parallelism (which we call Split-Replicate-Merge)

PDQ Collections Suite

Comprehensive library of collection classes that fully implement native associative container interfaces

Versatile in managing data over a wide range of scales



- A programmer...
 - instantiates the *PDQ Collection* objects, and
 - programs the computation using the *Programming API*
- The PDQ library...
 - transparently embeds the program in the *Computation Model*, and
 - executes the computation model in a parallel or distributed environment

Split-Replicate-Merge Computation Model

Supports highly efficient processing of PDQ collection objects.

Transparently supports and exploits parallelism in multi-core & distributed environments.

Which problems can be solved?

Formally:

Our data-parallel programming model is applicable to the class of problems where:

For a computation C_{seq} and input I ,

If $I_1 \subset I$ and $I_2 \subset I$, such that $I_1 \cap I_2 = \emptyset$
and

$$C_{seq}(I_1) \rightarrow A_1$$

$$C_{seq}(I_2) \rightarrow A_2$$

Then, the following must be equivalent :

$$C_{seq}(I_1 \cup I_2) \Leftrightarrow A_1 \underset{\text{merge}}{\cup} A_2$$

Where the merge operator is defined as :

$$(A_1 \underset{\text{merge}}{\cup} A_2)(k) := \begin{cases} A_1(k), & \text{if } k \in A_1 \wedge k \notin A_2 \\ A_2(k), & \text{if } k \notin A_1 \wedge k \in A_2 \\ A_1(k) \oplus A_2(k), & \text{otherwise} \end{cases}$$

for some user defined operator \oplus

Intuitively:

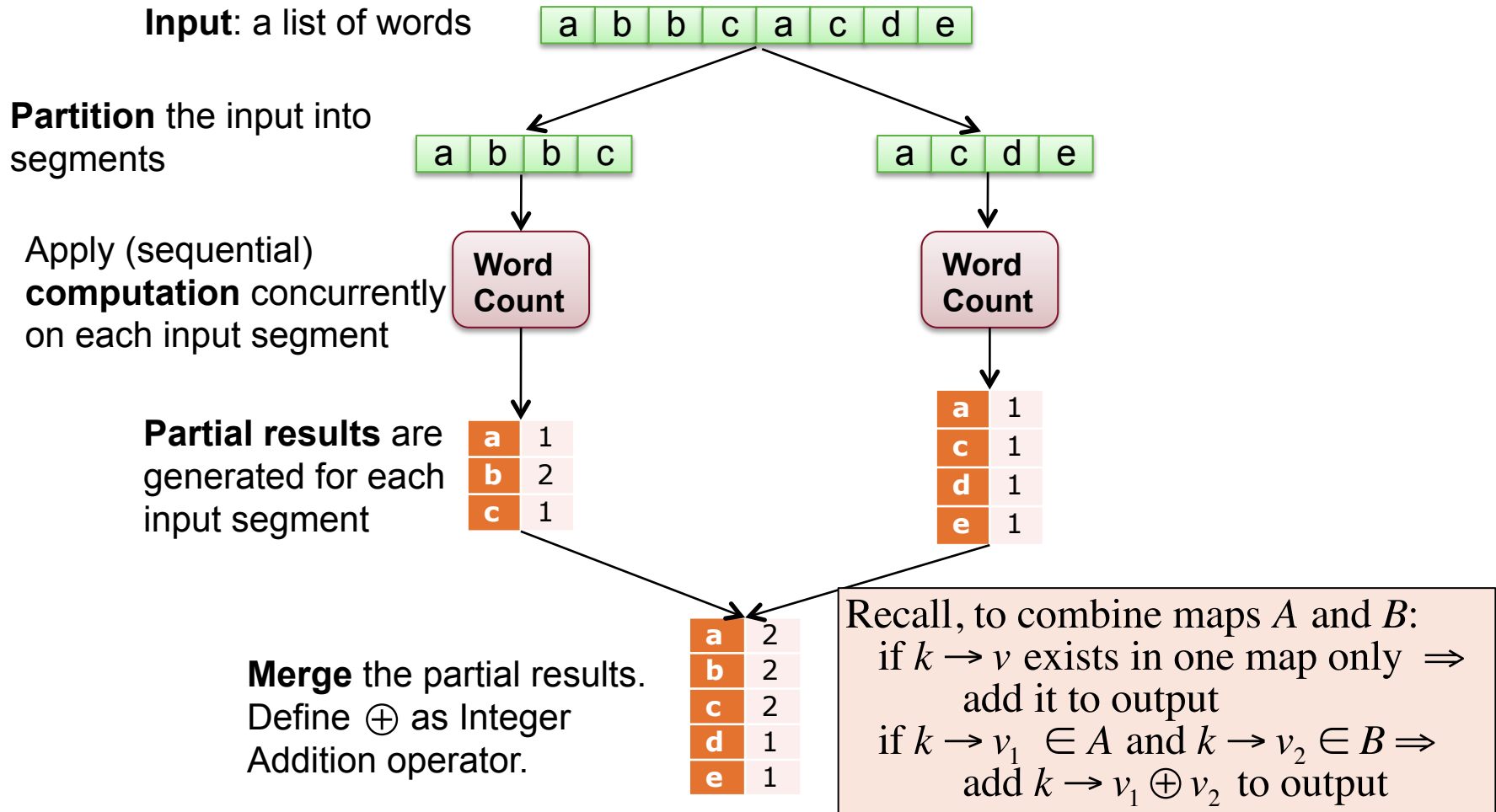
the inherent parallelism in the computation is such that if partial results were obtained by processing partial inputs, then the result of the combined inputs can be produced by merging the partial outputs.

Practically:

This data model is equivalent to the Map-Reduce model. That is, any problem solved by MR can be formulated in this manner.

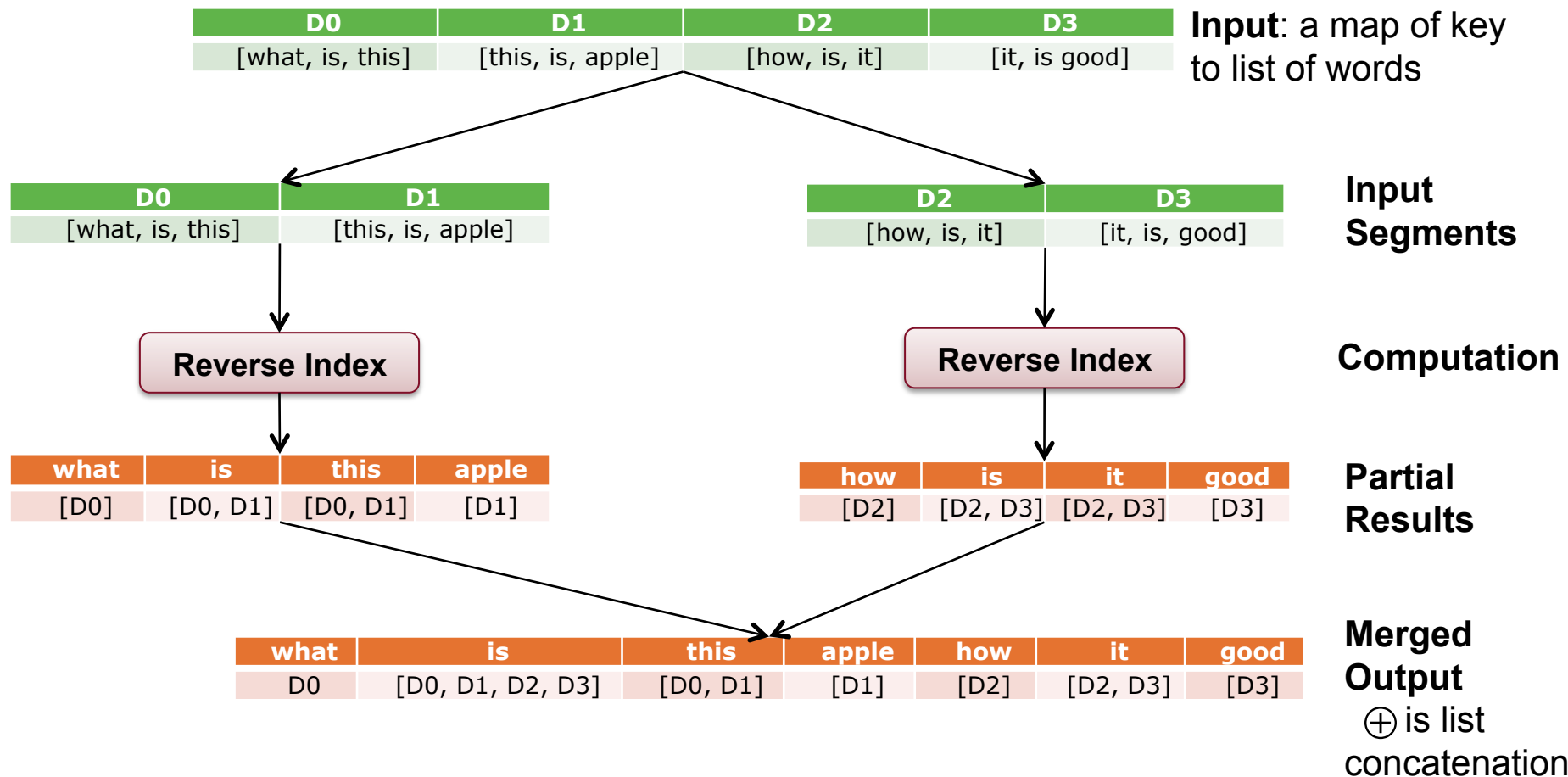
How does it work? Word Count Example

Given a list of words, find the number of times each word appears.



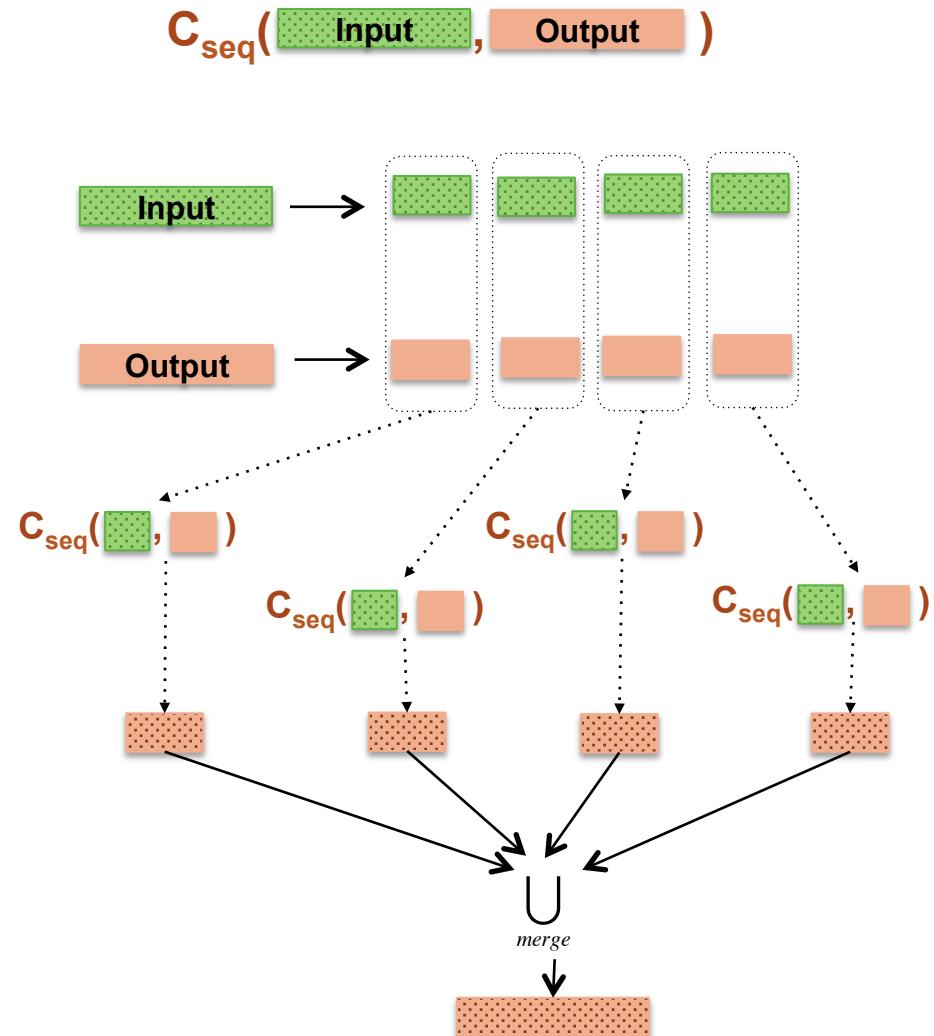
How does it work? Reverse Index Example

Given a map of keys to a list of words (the input), generate a map of words to the list of keys that they belonged (the output).



Split-Replicate-Merge Computation Model (the strategy for parallelization)

- The model is provided:
 - The computation method, input data container, output data (empty) container
- Input is partitioned into N **splits**
- Output is **replicated** N times
 - Replica is a data structure of same type as the output container
 - Initially empty
- Splits and replicas are paired, and N instances of computations are executed, one for each pair
- Each instance of computation generates partial results
- The partial results are **merged** into the output container



Further Details...

Please refer to the paper for further details on the following topics:

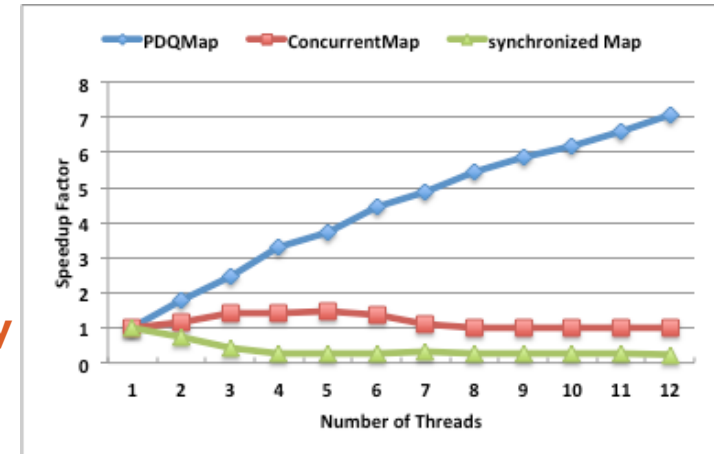
- A formal description of programming model, including
 - Data Model, which describes what problems can be solved
 - The Split-Replicate-Merge Computation Model, which describes the strategy for parallelism
 - The Programming Model and APIs
 - Equivalence with the Map Reduce programming model
- Our implementation (in Java SE6)
 - Defining the *Splittable* and *Mergeable* interfaces
 - Delegate design pattern based **implementation of PDQ container objects, which allows $O(1)$ splitting and parallel merging**
 - Thread pool execution mode: our implementation strategy for shared-memory and memory-bound workloads
 - Disk backed execution mode, an extension of the above to handle computations when output data is too large to fit in memory
 - Distributed execution mode for small to medium sized clusters
- Design Performance Analysis
 - Comparative study with concurrent data structures (for memory bound computation), *BerkeleyDB's* `StoredMap` (for disk-backed computation) and *Hazelcast* (for distributed computation)
 - Highlight the salient characteristics of our programming model that explains how PDQ significantly outperforms these traditional data structures

Design Performance Analysis

The PDQ programming model is compared against representative data structures for memory-bound and disk-bound computations

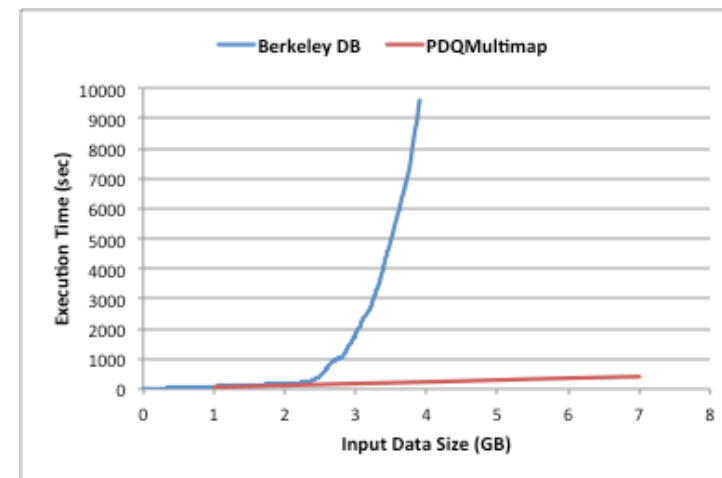
Memory Bound Computation

- Compared with synchronized blocks and `java.util.concurrent.ConcurrentHashMap`
- 'Word Count' problem. Input is a list of 300 million tokens drawn from a set of 5.2 million unique items
- In contrast with these alternatives, PDQ computation does not use locks
- **By avoiding locking overheads, PDQ significantly outperforms locking-based data structures**



Disk Bound Computation

- Compared with BerkeleyDB JE's StoredMap
- 'Reverse Index' problem. 7GB data on disk, single threaded execution
- Disk-backed structures are two-tiered: in-memory cache and on-disk B-tree. The cache miss penalty increases as the cache ratio gets smaller
- PDQ library reads and writes (sequentially) to disk only once, and does not require caches
- **By avoiding caching and random disk accesses, PDQ significantly outperforms traditional disk-backed data structures**




Applications of PDQ Library

- In the following we illustrate several applications of our PDQ library to highlight some of its distinguishing capabilities:
 - The *k*-Nearest Neighbor Classification problem in Machine Learning and the *FP Tree Growth Algorithm* in Data Mining illustrate how the PDQ programming model translates 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
 - relational processing of structured data
 - incremental processing of data

Application in Machine Learning: k-Nearest Neighbor based Classification

Handwritten Digit Recognition

- Each sample (e.g. ) is a 28x28 pixel image, each pixel is 1 byte
- That is, each image is a vector $V = \{v_1, v_2 \dots v_{784}\}$, $0 \leq v_i \leq 255$
- 42,000 training samples [images along with correct *labels*]

Machine Learning: given a test vector V_{test} determine the digit it represents.

- K-Nearest Neighbors based Classification approach:
 - Define *distance* between two vectors (we used Euclidean distance)
 - Find the k vectors from the training samples that are *closest* to V_{test}
 - Take the majority vote of labels of these k selected samples
- We defined a `LimitedPriorityQueue` that contains no more than k entries (if a new entry is less than the greatest entry in queue, the former is inserted and the latter is removed)
- The computation is then simply (using Python-like pseudo code):

```
for s in TrainingSamples:
    d = distance(Vtest , s.vector)
    queue.insert(d, s.label)
take_majority([label for d, label in queue])
```

k-Nearest Neighbor based Classification (contd...)

Parallelization with PDQ library

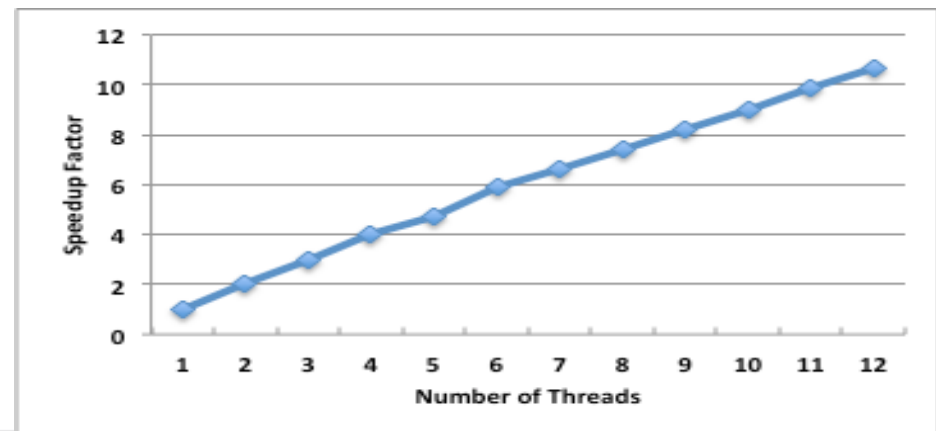
- We need to define the “add” operator for the LimitedPriorityQueue

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

- Prepare input (line 7)
 - PDQFileReader class automatically partitions a file
- Prepare output (lines 8)
 - PDQMap class automatically splits and merges intermediate results. Also, supply the add function.
- PDQ computation (line 9)
 - Supply the method, inputs and outputs
 - The library will execute the Split-Replicate-Merge model
- Notice that the actual computation (lines 1-6) is still sequentially written !

```
1 def computation(training_samples, map):  
2     queue = LimitedPriorityQueue(k)  
3     for s in training_samples:  
4         d = distance(Vtest, s.vector)  
5         queue.insert(d, s.label)  
6         map.put('default', queue)  
7 training_samples = PDQFileReader(filename)  
8 output = PDQMap(add)  
9 PDQ.run(computation, training_samples, output)  
10 queue = output.get("default")  
11 take_majority([label for d, label in queue])
```

- **Speedup of 10x achieved on a 12-core machine!**



Application in Data Mining: FP Tree Growth Algorithm

Frequent item set discovery

- In a store, a customer purchases a set of items (a transaction)
- There are many such transactions
- We wish to determine which items are frequently bought together

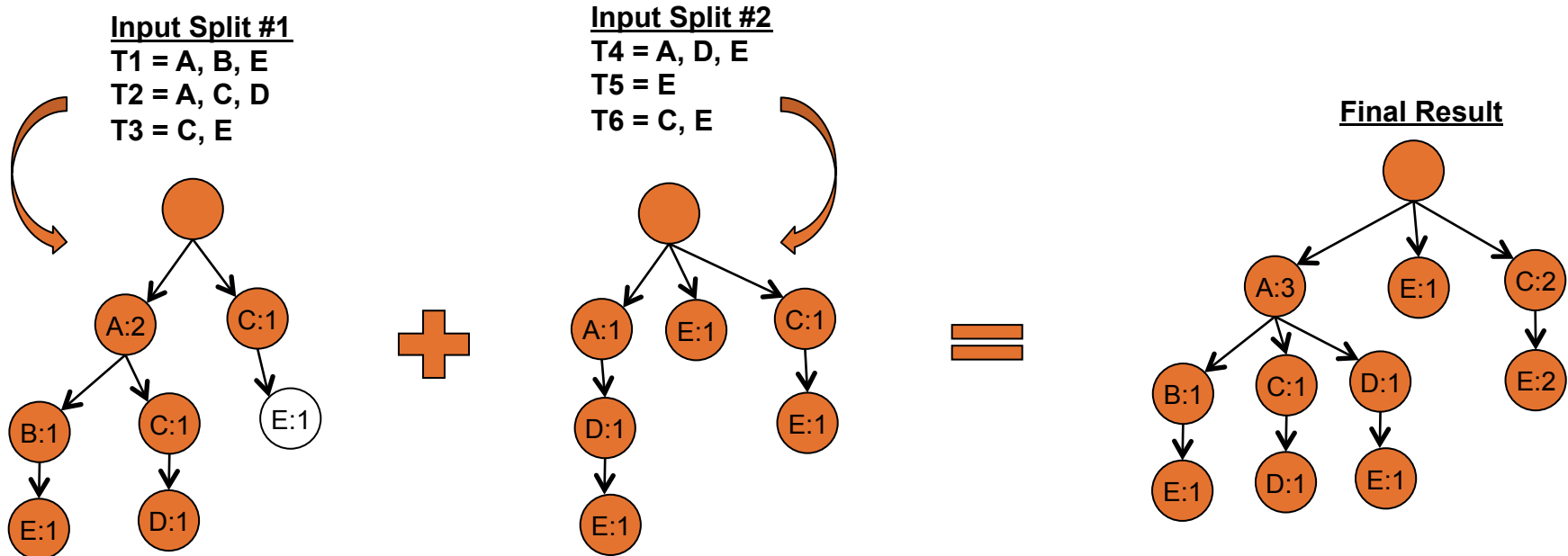
Frequent Pattern (FP) Tree Growth Algorithm

- In first phase, an FP Tree is constructed
 - Go over the data and count number of times each item is purchased
 - Remove those items whose count is below a threshold (called *support*)
 - For each transaction: sort items in descending order of frequency and add to the FP Tree
 - Please refer to existing literature for the algorithm to build the FP tree
- In second phase, the tree is mined for frequent item sets
 - Since each unique itemset is mined independently (and hence trivially parallelizable), we 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.

FP Tree Growth Algorithm (contd...)

Parallelization with PDQ Library

- Two trees can be easily added recursively
- An example shown below (header table and node links omitted for clarity)



- Our dataset: 5.2 million unique items, 1.7 million transactions, average 177 items per transaction, file size of 1.4 GB
- Speedup of 5x achieved on a 8-core machine!**

Application in Graph Processing: Pagerank Algorithm

Pagerank Algorithm

- Represents the web as directed graph: webpages as nodes, links as edges
- The algorithm generates weights for each node, called *rank*, which measures the relative importance of that web page
 - All nodes start with equal rank
 - In each iteration, each node transfers its rank equally amongst the nodes that it links to (please refer to literature for complete description)
 - Compute until ranks converge or a fixed number of iterations
- Our dataset
 - Wikilinks: graph of all English Wikipedia pages and internal page links (**21 GB**)
 - Processed raw data and created a `Multimap` (**8GB** when serialized on disk)
 - Large data, cannot be completely processed in-memory
- A highlight of PDQ library is that our associative container data structures 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

Pagerank Algorithm (contd...)

Parallelization with PDQ Library

■ Preparation

- Define the graph as a Map of String to GraphNode
- GraphNode is a structure with fields: rank (double) and outlinks (string list)
- Define the "add" operator for two GraphNodes: add ranks, concatenate outlinks

■ Pagerank algorithm computation

- Computation for one iteration is shown (omits damping factor and other details for brevity)
- prevIter is an input split, while currIter is an output replica
- **PDQ library ensures that splits and replicas are memory-resident**
- Hence, the map operations (get(), has_key()) are fast. For this reason, PDQ significantly outperforms working directly with disk-backed data structures (e.g. BerkeleyDB), which would have required random disk seeks.

```
def pagerank_iteration(prevIter, currIter):  
    for key, node in prevIter:  
        # copy the graph structure to next iteration  
        if not currIter.has_key(key):  
            currIter[key] = GraphNode()  
        currIter[key].outlinks = node.outlinks  
        # this is weight transferred to each outlink  
        transfer = node.rank / len(node.outlinks)  
        # actually transfer the weight to each outlink  
        for neighbor in node.outlinks:  
            if not currIter.has_key(neighbor):  
                currIter[neighbor] = GraphNode()  
            currIter[neighbor].rank += transfer
```

■ Executed the algorithm on a cluster of machines ranging from 1 to 8

- Why super-linear? Each JVM handles less data, hence spends less time in garbage collection

Cluster Size	Exec time per Iteration (min)	Speedup
1	20:38	
2	10:58	1.88x
4	5:04	4.06x
8	2:25	8.54x

Application in Relational Processing of Structured Data

A motivating example

- DBLP is a repository of academic publications in computer science
- We extracted two tables:
 - paper (paperId, authorId) Equivalently, a `Multimap` of paperId to authorId
 - citations (paperId, citedpaperId) Equivalently, a `Multimap` of paperId to citedPaperId(Each paper may have multiple authors, and may cite multiple papers)
- Sometimes authors cite their own papers, or of their colleagues (defined as people they have co-authored some paper with, or equivalently:
$$A \text{ is-colleague-of } B \Leftrightarrow \exists p, (p,A) \in \text{paper} \wedge (p,B) \in \text{paper}$$
- We would like to find out, for each author, what percentage of papers they cited were their own or of their colleagues
 - This should give us some insight into how clustered the research community is!
- We have defined following relational operations on maps (and multimaps)
 - (Only a subset of operations listed here, relevant for this problem)Let $A : K \rightarrow V$ and $B : K \rightarrow T$ are associative containers
$$\begin{array}{ll} \text{reverse}(A) : V \rightarrow K & v \rightarrow k \in \text{reverse}(A) \Leftrightarrow k \rightarrow v \in A \\ \text{join}(A,B) : K \rightarrow \langle V,T \rangle & k \rightarrow \langle v,t \rangle \in \text{join}(A,B) \Leftrightarrow k \rightarrow v \in A \wedge k \rightarrow t \in B \\ \text{mix}(A,B) : V \rightarrow T & v \rightarrow t \in \text{mix}(A,B) \Leftrightarrow \exists k \text{ s.t. } k \rightarrow v \in A \wedge k \rightarrow t \in B \\ \text{unique}(A) : K \rightarrow V & \text{remove duplicate values for each key} \end{array}$$
- Each of these operations can be parallelized by the PDQ library

Application in Relational Processing of Structured Data (contd...)

- The following shows an intuitive (but unoptimized) SQL statement and a counterpart PDQ data flow statements to count number of citations, for each author, that were their own or from colleagues

```
SELECT author, count(paper)
FROM
  (SELECT author, paper, coauthors,
    group_concat(coauthors) AS C
  FROM
    # for each author, the papers they cite
    (SELECT paper.uid AS author,
      citation.citedpaperId AS citedpaper
    FROM paper INNER JOIN cites
    ON author = citedpaper) AS Y1
  INNER JOIN
    # for each paper, its coauthor set
    (SELECT P1.pid AS paper, X1.coauthor AS coauthors
    FROM paper AS P1 INNER JOIN
    # for each author, list of coauthors
    (SELECT P1.authorId AS author,
      P2.authorId AS coauthor
    FROM paper AS P1 INNER JOIN paper AS P2
    WHERE P1.paperId = P2.paperId
    GROUP BY P1.uid, P2.uid) AS X1
    ON P1.authorId = X1.author
    GROUP BY P1.pid, X1.coauthor) AS Y2
  ON Y1.citedpapers = Y2.paper
  GROUP BY author, paper) AS Z1
WHERE
  (SELECT find_in_set(author, C)) > 0
GROUP BY author
```

```
T1 = (Multimap from papers table)
T2 = (Multimap from citations table)

T3 = reverse(T1) [papers written by each author]

T4 = mix(T1, T1) [coauthors for each author]
T5 = unique(T4) [unique coauthors for each author]

T6 = mix(T3, T5) [coauthor set for each paper]
T7 = unique(T6)

T8 = mix(T2, T1) [for each paper, authors that cite it]

T9 = join(T8, T7) [for each paper, authors that cite it
  and the coauthor set of this paper]

PDQ.run(compute, T9, result)

def compute(T9, result):
  for key, (citing_authors, coauthors) in T9:
    for citing_author in citing_authors:
      if citing_author in coauthors:
        result[citing_author] ++
```

Application in Log Processing: Incremental Processing of New Data

Incremental Log Processing

- In several contexts, logs are generated continuously (e.g. web traffic)
- 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

Incremental Processing with PDQ Library

- **PDQ library offers a native and efficient mechanism for incremental processing of data**
- Recall, the Split-Replicate-Merge 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 are merged into the existing data
- Since the PDQ data structures are natively `Serializable` (in our Java based implementation), incremental processing is easily accomplished, as shown next

Incremental Processing of New Data (contd...)

- After producing the first report, the output is saved to disk

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

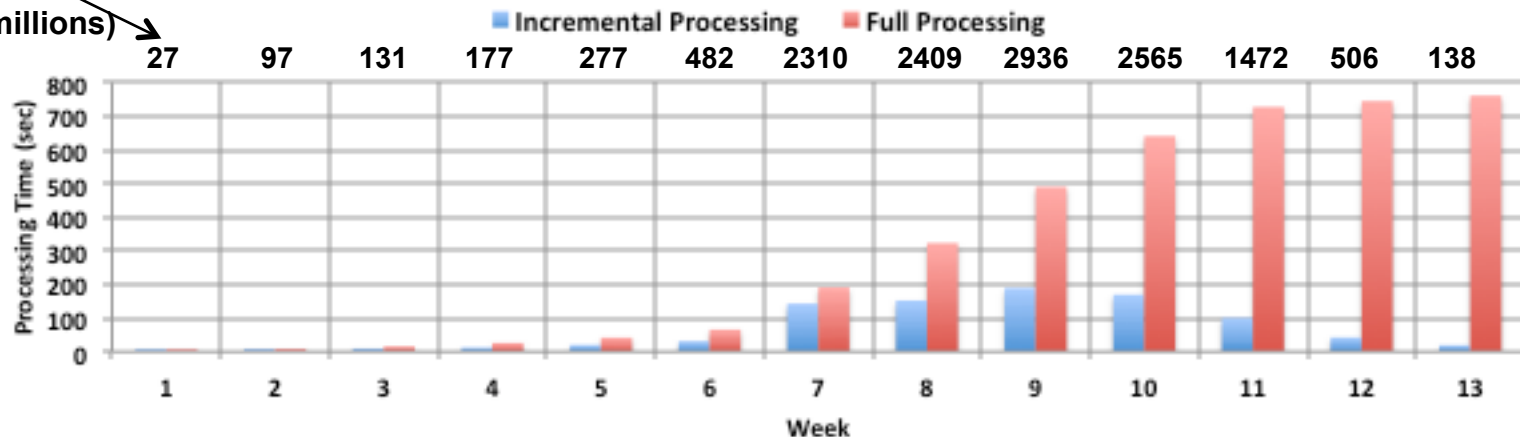
- For the subsequent reports, the previously saved report is loaded from disk and used as output

```
def incremental_report(last_timestamp, input_since_last_timestamp):  
    output = read_from_file("timestamp_" + last_timestamp)  
    PDQ.run(generate_report, input_since_last_timestamp, output)  
    save_to_file(output, "timestamp_" + (last_timestamp + 1))
```

- WorldCup98 Dataset

- Webpage request logs over 92 days, from 30 servers in four countries
- Each log entry is 20B, total data set 26GB
- Generated report each week

New log entries
this week (millions) →



Conclusions

- We have presented PDQCollections:
 - A library of associative containers that inherently encapsulate parallelism
 - A computation model that transforms the problem of parallelization to that of defining addition operator for value data types
- The PDQ library can efficiently exploit parallelism in multi-core as well distributed environments, for in-memory as well as large datasets
- The PDQ design offers some fundamental benefits over traditional data structures:
 - When processing memory-bound workloads, PDQ avoids locks and other forms of synchronization, and is able to significantly outperform lock-based data structures
 - When processing larger disk-bound workloads, PDQ does not use caches and avoids random disk access, and is able to significantly outperform traditional disk-backed structures
 - When processing distributed workloads, PDQ accesses remote data resources sequentially and only once, and is able to significantly outperform distributed data structures
- Our implementation can automatically and transparently transition its behavior from in-memory to disk-backed as well as distributed execution
- We highlighted the distinguishing capabilities of PDQ library with several applications drawn from a variety of fields in Computer Science