PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development

Jia Zou, R. Matthew Barnett, Tania Lorido-Botran*
Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian†, Binhang Yuan, Chris Jermaine
Department of Computer Science, Rice University
{jiazou,barnett,tl59,shangyu.luo,cm13,ss107,kt17,by8,cmj4}@rice.edu

ABSTRACT

This paper describes *PlinyCompute*, a system for development of high-performance, data-intensive, distributed computing tools and libraries. *In the large*, PlinyCompute presents the programmer with a very high-level, declarative interface, relying on automatic, relational-database style optimization to figure out how to stage distributed computations. However, *in the small*, PlinyCompute presents the capable systems programmer with a persistent object data model and API (the "PC object model") and associated memory management system that has been designed from the ground-up for high performance, distributed, data-intensive computing. This contrasts with most other Big Data systems, which are constructed on top of the Java Virtual Machine (JVM), and hence must at least partially cede performance-critical concerns such as memory management (including layout and de/allocation) and virtual method/-function dispatch to the JVM. This hybrid approach—declarative in the large, trusting the programmer's ability to utilize PC object model efficiently in the small—results in a system that is ideal for the development of reusable, data-intensive tools and libraries.

KEYWORDS

Distributed computing; Object model; Query compilation

ACM Reference Format:
Jia Zou, R. Matthew Barnett, Tania Lorido-Botran and Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, Chris Jermaine. 2018. PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18)*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196933

1 INTRODUCTION

Big Data systems such as Spark [58] and Flink [11, 21] have effectively solved what we call the "data munging" problem. That is, these systems do an excellent job supporting the rapid and robust development of problem-specific, distributed/parallel codes that extract actionable information from the structured or unstructured data. But while existing Big Data systems offer excellent support for data munging, there is a class of applications for which existing systems are used, but arguably are far less suitable: as a platform for the development of high-performance codes, especially reusable Big Data tools and libraries, by a capable systems programmer.

The desire to build new tools on top of existing Big Data systems is understandable. The developer of a distributed data processing tool must worry about data persistence, movement of data to/from secondary storage, data and task distribution, resource allocation, load balancing, fault tolerance, and many other factors. While classical high-performance computing (HPC) tools such as MPI [34] do not provide support for all of these concerns, existing Big Data systems address them quite well. As a result, many tools and libraries have been built on top of existing systems. For example, Spark supports machine learning (ML) libraries Mahout [4], Dl4j [1], and Spark `mllib` [44], linear algebra packages such as SystemML [15, 16, 31, 54] and Samsara [5], and graph analytics with GraphX [32] and GraphFrames [28]. Examples abound.

PlinyCompute: A platform for high-performance, Big Data computing. However, we assert that if one were to develop a system purely for developing high-performance Big Data codes by a capable systems programmer, it would not look like existing systems such as Spark and Flink, which have largely been built using high-level programming languages and managed runtimes such as the JVM. Virtual machines abstract away most details regarding memory management from the system designer, including memory deallocation, reuse, and movement, as well as pointers, object serialization and deserialization. Since managing and utilizing memory is one of the most important factors determining Big Data system performance, reliance on a managed environment can mean an order-of-magnitude increase in CPU cost for some computations [14]. This cost may be unacceptable for high-performance tool development by an expert.

This paper is concerned with the design and implementation of *PlinyCompute*, a system for development of high-performance, data-intensive, distributed computing codes, especially tools and libraries. PlinyCompute, or PC for short, is designed to fill the gap between HPC software such as OpenMP [27] and MPI [34], which provide little direct support for managing very large data sets, and dataflow platforms such as Spark and Flink, which may give up significant performance through their reliance on a managed runtime to handle memory management (including layout and de/allocation) and key computational considerations such as virtual function dispatch.

* Tania Lorido-Botran is currently affiliated with University of Deusto.
† Kia Teymourian is currently affiliated with Boston University.

Core design principle: Declarative in the large, high-performance in the small. PC is unique in that *in the large*, it presents the programmer with a high-level, declarative interface, relying on automatic, relational-database style optimization [22] to figure out how to stage distributed computations. PC's declarative interface is higher-level than competing systems, in that decisions such as choice of join ordering and join algorithms are totally under control of the system. This is particularly important for tool and library development because the same tool should run well regardless of the data it is applied to—the classical idea of *data independence* in database system design [53]. Because a tool developer does not need to choose a particular join algorithm to use (PC will do this at runtime, automatically) a library built on top of PC is typically very flexible. If the library is used on a small data set, PC will automatically choose an appropriate join algorithm. On a large data set, it will choose another. This means that the same library can be used on different data sets without a significant amount of tuning.

In contrast, *in the small*, PlinyCompute presents a capable programmer with a persistent object data model and API (the "PC object model") and associated memory management system designed from the ground-up for high performance. All data processed by PC are managed by the PC object model, which is exclusively responsible for PC data layout and within-page memory management. The PC object model is tightly coupled with PC's execution engine, and has been specifically designed for efficient distributed computing. All dynamic PC `Object` allocation is *in-place*, directly on a page, obviating the need for PC `Object` serialization and deserialization before data are transferred to/from storage or over a network. Further, PC gives a programmer fine-grained control of the system memory management and PC `Object` re-use policies.

This hybrid approach—declarative and yet trusting the programmer to utilize PC object model effectively—results in a system ideal for the development of data-oriented tools and libraries.

The system consists of four main components:

(1) The *PC object model* for building high-performance, persistent data structures.

(2) The *PC API and TCAP compiler.* In the large, PC codes are declarative and look a lot like classical relational calculus [24]. For example, to specify a join over five sets of objects, a PC programmer does not build a join directed acyclic graph (DAG) over the five inputs, as in a standard dataflow platform such as Spark. Rather, a programmer supplies two *lambda term construction functions*: one that constructs a lambda term describing the selection predicate over those five input sets, and a second that constructs a lambda term describing the relational projection over those five sets using the same API. These lambda terms are constructed using PC's built-in lambda abstraction families as well as higher-order composition functions. PC's TCAP compiler accepts such a specification, and compiles it into a functional, domain-specific language called *TCAP* that implements the join.

(3) The *execution engine*, which is a distributed query processing system for Big Data analytics. It consists of an optimizer for TCAP programs and a high-performance, distributed, vectorized TCAP processing engine. The TCAP processing engine has been designed to work closely with the PC object model to minimize memory-related costs during computation.

(4) Various *distributed services.* When PC runs on a distributed cluster it has a *master node* and one or more *worker nodes.* Running on the master node are the managers for the various distributed services provided by PC, primarily the *catalog manager* and the *distributed storage manager.* Also running on the master node is the software responsible for powering the distributed execution engine: the *TCAP optimizer* and the *distributed query scheduler.* When a user of PC requests to execute a graph of computations, the computations are compiled into a TCAP program on the user's process, then optimized by the TCAP optimizer and executed by the distributed query scheduler.

Our contributions. We describe the design and implementation of PlinyCompute. We also benchmark several library-style software written on top of PC. We begin with a small domain specific language for distributed linear algebra that we implemented on top of PC, called `lilLinAlg`. `lilLinAlg` was implemented in about six weeks by a capable developer who had no knowledge of PC at the outset, with the goal of demonstrating PC's suitability as a tool-development platform. We also benchmark the efficiency of manipulating complex objects, and also several standard machine learning code written on top of PC.

2 OBJECT MODEL OVERVIEW

There is growing evidence that the CPU costs associated with manipulating data, especially data (de-) serialization and memory (de-) allocation, dominate the time needed to complete typical Big Data processing tasks [48, 50, 51]. To avoid these potential pitfalls while at the same time giving the user a high degree of flexibility, PC requires programmers to store and manipulate data using the *PC object model*, which is an API for storing and manipulating objects, and has been co-designed with PC's memory management system and execution engine to provide maximum performance.

In PC's C++ binding[1], individual PC `Objects` correspond to C++ objects, and so the C++ compiler specifies the memory layout. However, where PC `Objects` are stored in RAM and on disk, and how they are allocated and deallocated, when and where they are moved, is tightly controlled by PC itself.

2.1 In-Place Allocation

The PC object model provides a fully object-oriented interface, and yet manages to avoid many of the costs associated with complex object manipulation by following the *page-as-a-heap* principle. All PC `Objects` are allocated and manipulated in-place, on a system- (or user-) allocated page. There is no distinction between the in-memory representation of data and the on-disk (or in-network) representation of data. Thus there is no (de-)serialization cost to move data to/from disk and network, and memory management costs are very low. Depending upon choices made by the programmer, "deallocating" a page of objects means simply unpinning the page and allowing it to be returned to the buffer pool, where it will

[1] Currently, PC supports only C++ development. While supporting other language bindings is desirable and part of our future plans, it will be challenging as PC makes heavy use of many features that are unique to C++, such as template meta-programming (Section 5.3).

be recycled and written over with a new set of objects. In computer systems design, this is referred to as *region-based allocation* [35, 55], and is often the fastest way to manage memory.

To illustrate the use of the PC object model from a user's perspective, imagine that we wish to perform a computation over a number of feature vectors. Using the PC object model's C++ binding, we might represent each data point using the DataPoint class:

```
class DataPoint : public Object {
public:
        Handle <Vector <double>> data;
};

makeObjectAllocatorBlock (1024 * 1024);
Handle <Vector <Handle <DataPoint>>> myVec =
     makeObject <Vector <Handle <DataPoint>>> ();
Handle <DataPoint> storeMe = makeObject <DataPoint> ();
storeMe->data = makeObject <Vector <double>> ();

for (int i = 0; i < 100; i++)
     storeMe->data->push_back (1.0 * i);

myVec->push_back (storeMe);
pcClient.createSet <DataPoint> ("Mydb", "Myset");
pcClient.sendData <DataPoint> ("Mydb", "Myset", myVec);
```

Here, the programmer starts out by creating a one megabyte *allocation block* where all new objects will be written, and then allocates data directly to that allocation block via a call to `makeObject()`. Each call to `makeObject()` returns a PC's pointer-like object, called a `Handle`. PC `Handle`s use offsets rather than absolute memory addresses, so they can be moved from process to process.

When the data are dispatched via `sendData()` as in the above example, the occupied portion of the allocation block is transferred in its entirety with no pre-processing and zero CPU cost, aside from the cycles required to perform the data transfer. This illustrates the principle of *zero cost data movement.*

2.2 Full Allocation Blocks

When an allocation block becomes full, PC cannot process a call to `makeObject()`, and an exception is thrown. In PC object model code running outside of PC (such as a driver program that is loading data into PC), the application programmer must handle the exception. For example, the user may send the data to PC (via a call to `sendData`, as above).

In code that is running within PC's execution engine, allocation blocks are created automatically on top of memory pages managed by the distributed storage manager. When a `makeObject()` in application code fails due to a page being full, the resulting exception is handled by the system. The precise way of exception handling depends upon the context. For example, during a distributed aggregation, PC associates the exception with the hash table used to aggregate data being full, and react by sending the page holding the hash table across the network, creating a new allocation block to hold a new hash table, and re-running the failed operation.

2.3 Automated Memory Management

C++ (and un-managed languages in general) can be more difficult for programmers to use compared to managed languages such as Java. Managed languages have the advantage of being garbage collected; memory leaks are rare when programming in a managed language, and many memory-related bugs such as double frees are non-existent. The PC object model mitigates many of these difficulties by handling many aspects of memory management. For example, object allocation and deallocation are handled by the PC object model. Continuing our example, the line of code `myVec = nullptr;` would automatically free all of the memory associated with the `Vector` of `DataPoint` objects, since PC `Objects` are reference counted.

This can be expensive, however, since the PC `Object` infrastructure must traverse a potentially large graph of `Handle` objects to perform the deallocation. Recognizing that low-level data manipulations dominate Big Data computation times [48, 50], PlinyCompute gives a programmer control over most aspects of memory management. If the programmer had used:

```
storeMe->data = makeObject <Vector <double>>
     (ObjectPolicy :: noRefCount);
```

then the memory associated with `storeMe->data` would not be reference counted, and hence not reclaimed when unreachable.[2] Use of `noRefCount` may mean lower memory utilization, but the benefit is nearly zero-cost memory management within the block. PC gives the developer the ability to manage the tradeoff. This illustrates another key principle behind the design of PlinyCompute: *Since PC is targeted towards tool and library development, PC assumes the programmer is capable. In the small, PC gives the programmer all of the tools s/he needs to make things fast.*

3 PLINYCOMPUTE'S LAMBDA CALCULUS

A PC programmer specifies a distributed query graph by providing a graph of high-level computations over sets of data—those data may either be of simple types, or they may be PC `Objects`.

The PC toolkit consists of a set of computations that can be used to compose a query graph: `SelectionComp` (equivalent to relational selection and projection), `JoinComp` (equivalent to a join of arbitrary arity and arbitrary predicate), `AggregateComp` (aggregation), `MultiSelectComp` (relational selection with a set-valued projection function) and a few others. Each of these is an abstract type descending from PC's `Computation` class.

Where PC differs from other systems is that a programmer customizes these computations by composing together various C++ codes using a domain-specific lambda calculus. For example, to implement a `SelectionComp` over PC `Objects` of type `DataPoint`, a programmer must implement the lambda term construction function `getSelection (Handle <DataPoint>)` which returns a lambda term describing how `DataPoint` objects should be processed.

Novice PC programmers sometimes incorrectly assume that the lambda term construction functions operate on the data themselves, and hence are called once for every data object in an input set—for example, that `getSelection()` would be repeatedly invoked to filter each `DataPoint` in an input set. This is incorrect, however. A programmer is not supplying a computation over input data; rather,

[2] In user code, a programmer utilizing the `noRefCount` policy must manually deallocate the allocation block when s/he is sure that there can be no remaining references into the block. In user code running on PC's execution engine, the engine itself is able to use its knowledge of how data are pushed through the pipelined execution engine (see Section 5) to determine when no user-created objects could possibly be reachable in a block, and then deallocate the block as a whole.

a programmer is supplying an expression in the lambda calculus that specifies *how to construct the computation.*

To construct statements in the lambda calculus, PC supplies a programmer with a set of built-in *lambda abstraction* families [45], as well as a set of *higher-order functions* [23] that take as input one or more lambda terms, and return a new lambda term. Those built-in lambda abstraction families include:

(1) makeLambdaFromMember(), which returns a lambda abstraction taking as input a Handle to a PC Object, and returns a function returning one of the pointed-to object's member variables;

(2) makeLambdaFromMethod(), which is similar, but returns a function calling a method on the pointed-to variable;

(3) makeLambda(), returning a function calling a native C++ lambda;

(4) makeLambdaFromSelf(), returning an identity function.

When writing a lambda term construction function, a PC programmer uses these families to create lambda abstractions that are customized to a particular task. The higher-order functions provided are used to compose lambda terms, and include functions corresponding to:

(1) The standard boolean comparison operations: ==, >, !=, etc.;

(2) The standard boolean operations: &&, ||, !, etc.;

(3) The standard arithmetic operations: +, -, *, etc.

For an example of all of this, consider performing a join over three sets of PC Objects stored in the PC cluster. Joins are specified in PC by implementing a JoinComp object. One of the methods that must be overridden to build a specific join is JoinComp :: getSelection() which returns a lambda term that specifies how to compute if a particular combination of input objects is accepted by the join. Consider the following getSelection() for a three-way join over objects of type Dept, Emp, and Sup:

```
Lambda <bool> getSelection (Handle <Dep> arg1,
    Handle <Emp> arg2, Handle <Sup> arg3) {
        return makeLambdaFromMember (arg1, deptName) ==
               makeLambdaFromMethod (arg2, getDeptName) &&
               makeLambdaFromMember (arg1, deptName) ==
               makeLambdaFromMethod (arg3, getDept);  }
```

This method creates a lambda term taking three arguments arg1, arg2, arg3. This lambda term describes a computation that checks to see if arg1->deptName is the same as the value returned from arg2->getDeptName(), and that arg1->deptName is the same as the value returned from arg3->getDept(). Note that the programmer does *not* specify an ordering for the joins, and does *not* specify specific join algorithms or variations. Rather, PC analyzes the lambda term returned by getSelection() and makes such decisions automatically.

In general, a programmer can choose to expose the details of a computation to PC, by making extensive use of PC's lambda calculus, or not. A programmer could, for example, hide the entire selection predicate within a native C++ lambda. The system relies on the willingness of the programmer to expose intent via the lambda term construction function.

4 EXECUTION ENGINE

PC's execution engine is tasked with optimizing and executing TCAP programs (pronounced "tee-cap"). PC's TCAP compiler calls the various user-supplied lambda term construction functions for each of the Computation objects in a user-supplied graph of computations, and compiles all of those lambda terms into a DAG of small, atomic computations—a TCAP program. In this section, we discuss how TCAP programs are executed by PC.

4.1 Vectorized or Compiled?

Volcano-style, record-by-record iteration [33] has fallen out of favor over the last decade, largely replaced by two competing paradigms for processing data in high-performance, data-intensive computing. The first is *vectorized* processing [8, 17, 36, 61], where a column of values are pushed through a simple computation in sequence, with few cache misses and no virtual function calls. The second is *code generation* [10, 18, 39, 46, 47], where a system analyzes the computation and then generates code—either C/C++ code, or byte code for a framework such as LLVM [40, 41].

While PlinyCompute certainly leverages ideas from both camps, we argue that the "vectorized vs. generated" argument is relevant mostly for relational systems with a data-oriented, domain-specific language (such as SQL). Classical vectorization requires an execution plan to be constructed consisting entirely of calls to a toolkit of vector-based operations shipped with the system. This is unrealistic when most/all computations are over user-defined types. Further, generating LLVM code for complex operations over user-defined types is unrealistic, akin to writing a full-fledged compiler.

PC uses a hybrid approach, where the PC execution engine is vectorized, but the code for the individual vectorized operations (called *pipeline stages*) is fully compiled. PC's C++ binding relies on template meta-programming (see Section 5.3) to convert the user-supplied lambda terms (see Section 4) into a DAG of pipeline stages over vectors of PC Objects or simple types. This DAG is decomposed into a set of pipelines. Input data are broken into lists of data vectors (called, appropriately, *vector lists*), and fed into the pipelines, where they are processed by each pipeline stage.

Using template meta-programming, pipeline stages are fully compiled. They have no internal virtual function calls (aside from any virtual function calls in the user's code), and the overhead associated with invoking the pipeline stage via a virtual functional call is amortized over all of the rows in an input vector list.

4.2 The TCAP IR

PC is designed to have a clear separation between the components of the system that (1) accept and pre-process user-supplied computations, (2) optimize those computations, and (3) execute those computations. Having a clear separation allows us to modify any of these three components, or to easily drop in a replacement component, such as a new optimizer.

To facilitate this, PC utilizes a text-based, human-readable intermediate representation (IR) for PC computations called *TCAP* to communicate between PC's input pre-processor (called the *TCAP compiler*) and the execution engine. The text of a TCAP program, plus the compiled native functions it refers to, totally describe a PC computation. While TCAP is unique to PC, the idea of using an IR with a text-based representation is obviously not unique—in fact, we took inspiration from systems such as the LLVM compiler framework that does the same thing.

The TCAP IR is a functional language. Each statement in a TCAP program operates over a set of vector lists (or multiple sets of vector lists in the case of a join), transforming the input set to a set of output vector lists. The various statements in a TCAP program form a DAG; statements are linked by input-output dependencies. During execution of a TCAP program, paths in the DAG are chained together to form pipelines. Optimization of TCAP as well as optimizing how TCAP programs are used to form pipelines is discussed in the Appendix of the paper.

The most common TCAP operation is the APPLY operation, which applies a pipeline stage encapsulating the repeated application of a user-defined function. Conceptually, this iterates though all of the vector lists in an input set. For each vector list, APPLY forms a new vector by applying the function to each row in a subset of the input vectors. There are many different TCAP operations (APPLY, FILTER, SCAN, AGG, HASHLEFT, JOIN, etc.). Many, such as JOIN and AGG, roughly correspond to relational operations.

To illustrate TCAP, we consider the following code:

```
Lambda <bool> getSelection (Handle <Dep> arg1,
    Handle <Emp> arg2, Handle <Sup> arg3) {
        return makeLambdaFromMember (arg1, deptName) ==
               makeLambdaFromMethod (arg2, getDeptName); }
```

PC's TCAP compiler will compile the lambda term resulting from a call to getSelection() into the following TCAP code:

```
WDNm_1(dep,emp,sup,nm1) <= APPLY(In(dep),
   In(dep,emp,sup), 'Join_2122', 'att_acc_1',
      [('type', 'attAccess'), ('attName', 'deptName')]);

WDNm_2(dep,emp,sup,nm1,nm2) <= APPLY(WDNm_1(emp),
   WDNm_1(dep,emp,sup,nm1), 'Join_2122',
      'method_call_2', [('type', 'methodCall_2'),
          ('methodName', 'getDeptName')]);

WBl_1(dep,emp,sup,bl) <= APPLY(WDNm_2(nm1,nm2),
   WDNm_2(dep,emp,sup), 'Join_2122', '==_3',
      [('type', 'equalityCheck')]);

Flt_1(dep,emp,sup) <= FILTER(WBl_1(bl),
   WBl_1(dep,emp,sup), 'Join_2122', []);
```

These four TCAP statements correspond to a pipeline of four stages, as shown above in Figure 1.

This particular TCAP code begins with an APPLY operation, which is a five-tuple, consisting of: (1) the vector list and constituent vector(s) for the APPLY to operate on, (2) the vector(s) from that vector list to copy from the input to the output, (3) the name of the computation that the operation was compiled from, (4) the name of the compiled code (pipeline stage) that the operation is to execute, plus (5) a key-value map that stores specific information about the operation that may be used later during optimization.

Specifically, in this case, the first APPLY in the TCAP computation describes a pipeline stage that takes as input a vector list called In, which is made of the constituent vectors, referred to using the names dep, emp, and sup. To produce the output vector list (called WDNm_1), the vectors dep, emp, and sup should be simply copied (via a shallow copy similar to pointer assignment) from In. In addition, the compiled code referred to by Join_2122.att_acc_1() will be

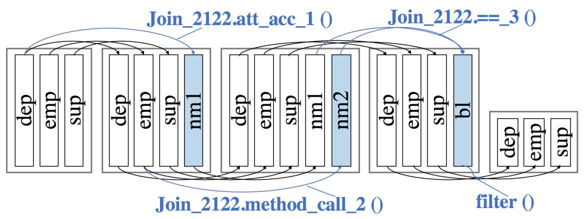

Figure 1: Execution of the TCAP program. The first two pipeline stages extract vectors from existing vectors of objects, beginning with a call to Join_2122.att_acc_1(), which extracts Dep.deptName from each item in the vector dep, producing a new vector called nm1. Then, Join_2122.method_call_2() produces a new vector called nm2. A bit vector bl is formed by checking the equality of those two vectors via a call to Join_2122.==_3(), and then all of the vectors are filtered.

executed via a vectorized application to the input vector dep. The result will then be put into a new vector called WDNm_1.nm1. The resulting vector list (consisting of the vectors shallow copied from the input as well as the new vector WDNm_1.nm1) is called WDNm_1.

Next, this TCAP program specifies that WDNm_1 is processed by APPLYing the method call getDeptName() on the attribute emp; this is done via application of the compiled code referred to by Join_2122.method_call_2. The vectors dep, emp, sup and nm1 are simply shallow-copied to the output vector list.

Then an equality check is performed to create WBl_1.bl (a vector of booleans) and the result is filtered based upon this column.

4.3 Template Meta-Programming

In PC, each vectorized pipeline stage (such as Join_2122.att_acc_1) is executed as fully-compiled native code, with no virtual function calls. In PC's C++ binding, this is accomplished by using the C++ language's extensive *template meta-programming* capabilities [37]. Templates are the C++ language's way of providing generic functionality. When a C++ template class or function is instantiated with a type, the C++ compiler actually generates optimized native code for that specific new type, at compile time. This is quite different from languages such as Java, that must typically rely on slow virtual function calls in order to implement generics.

To see how template meta-programming is used by PC, consider the TCAP operation from our example:

```
WBl_1(dep,emp,sup,bl) <=
   APPLY(WDNm_2(nm1,nm2),WDNm_2(dep,emp,sup),
      'Join_2122', '==_3', '');
```

Here, the pipeline stage Join_2122.==_3 that is specified by the APPLY operation actually refers to a function generated as a by-product of the PC's == operation in the line of code:

```
return makeLambdaFromMember (arg1, deptName) ==
    makeLambdaFromMethod (arg2, getDeptName);
```

The == operation (corresponding to a higher-order function that constructs a lambda term checking for equality in the output of two input lambda terms) is actually implemented as C++ template whose two type parameters LHSType and RHSType are inferred from the output types of the two input lambda terms. The == template returns an object of type EqualsLambda <LHSType, RHSType>, which itself has an operation returning a pointer to the pipeline stage Join_2122.==_3 referred to in the TCAP. As expected, this stage processes in an input vector list, creating a new vector of booleans, containing the truth values of the equality check of each LHSType from the left input vector and each RHSType from the right input vector. Using C++'s template meta-programming facilities, this pipeline stage consists of generated code containing the user-provided equality check operator over LHSType and RHSType; this operator is invoked for each of the pairs of objects in the input vectors. Thus, as the Join_2122.==_3 pipeline stage processes a vector list, there are no function calls that cannot themselves be in-lined by the compiler and optimized—unless, of course, the (potentially) user-defined equality operation over LHSType and RHSType objects itself contains a virtual function call.

5 THE OBJECT MODEL IN DETAIL

5.1 Generic and OO Programming

For decades, the dominant model used in data management was the flat relational model, which can achieve very good performance. The downside is that flat relations are very limiting to a programmer. Modern, object-based data analytics systems (such as Spark via its Resillient Distributed Dataset (RDD) interface [57]) offer far more flexibility, at the (possible) cost of significant performance degradation. PC attempts to utilize the principle of zero-cost data movement to facilitate this high level of flexibily, while approaching relational performance.

The PC object model provides a fully object-oriented interface, supporting the standard functionality expected in a modern, object-oriented type system, including generic programming (the PC object model supports generic Map and Vector types), pointers (or, more specifically, "pointer-like" objects called PC Handle objects), inheritance, and dynamic dispatch for runtime polymorphism.

For example, imagine that the goal is implementing a distributed linear algebra library on top of the PC object model, where huge matrices are "chunked" into smaller sub-matrices. A sub-matrix may be stored via PC's C++ binding using the following object:

```
class MatrixBlock : public Object {
public:
        int chunkRow, chunkColumn;
        int chunkWidth, chunkHeight;
        Vector <double> values;
};
```

Or, a sparse sub-matrix may be stored as:

```
class SparseMatrixBlock : public Object {
public:
        int chunkRow, chunkColumn;
        int chunkWidth, chunkHeight;
        Map <pair <int, int>, double> values;
};
```

In the sparse sub-matrix, the pair<int, int> indexes a non-zero entry in the chunk by its row and column. Because these data structures are allocated dynamically in library code, it is easy for the library developer to include logic that allows the size of containers such as values to be tuned either automatically, by the library, or by the library's user.

In order to guarantee zero-cost data movement, one rule that a PC programmer must follow is that any object that will be loaded into the distributed PC cluster must either be of a "simple" type (a simple type must contain no raw C-style pointers and no virtual functions, and a memmove must suffice to copy the object), or else it must descend from PC's Object class, which serves as the base for all complex object types. Complex objects are those that include containers (Vector, Map) or pointer-like Handle objects. Descending from PC's Object class ensures that the resulting class type has a set of virtual functions that allow it to be manipulated in and transferred across the distributed PC cluster, such as a virtual deep copy function.

5.2 PC Handles

To support linked data structures, dynamic allocation, and runtime polymorphism, it is necessary for a system to provide pointer-like functionality. This is provided by PC's built-in Handle type. A Handle to an object is returned from a dynamic allocation to the current allocation block, such as the following statement:

```
Handle<MatrixBlock>mySubMatrix=makeObject<MatrixBlock>();
```

Internally, PC Handle objects contain two pieces of data: an *offset pointer* that tells how far the physical address of the object being pointed to is from the physical location of the Handle, and a *type code* that stores the type of the object that is pointed to.

PC uses an offset pointer rather than a classical, C-style pointer in order to support zero-cost data movement. A Handle may begin its life allocated to one page, which may be stored on disk, then sent across a network to another process. The Handle pointer can function correctly at the new process. An actual C-style pointer cannot survive translation from one process to another, as the program will be mapped to a different location in memory.

5.3 Dynamic Dispatch

In PC, dynamic dispatch for virtual function calls is facilitated by the type code stored within each Handle object. Each type code begins with a bit that denotes whether or not the referenced type is a simple type (which, by definition, cannot have any virtual functions and for which a memmove suffices to perform a copy) or a type descended from PC's Object base class.

In the case of a PC Object or its descendants, the type code is a unique identifier for the PC-Object-descended type of the object that the Handle points to. In every major C++ compiler (GCC, clang, Intel, and Microsoft), virtual functions are implemented using a virtual function table, or vTable object, a pointer to which is located at the beginning of each C++ object having a virtual function. Unfortunately, the vTable pointer is a native, C-style pointer, and it does not automatically translate when an object is moved across processes. To handle this, in PC's C++ binding, whenever a Handle object is dereferenced, a lookup using the type code retrieves a process-specific pointer to that class' vTable object.

Obtaining a pointer to a class' vTable object is not straightforward. A user may run code on his/her machine that creates a PC Object, and then ships that PC Object into the distributed PC cluster. At the other end, it arrives at a PC worker process that has never seen that type of object before and hence does not have access to a vTable pointer for that class. PC addresses this issue by requiring that all classes deriving from PC's Object base class be registered with the catalog manager before they are loaded into the distributed storage system. This registration requires shipping a library file (a .so file in Linux/Unix) to the catalog manager. This library exposes a special getVTablePtr() function that returns a pointer to the vTable for the class contained in the .so file.

A vTable pointer lookup first goes to the PC process' vTable lookup table. When this lookup fails (because the process has not yet seen a vTable pointer for that class type) the request then goes to the catalog manager, which responds to the process with a copy of the appropriate .so file. This .so file is then dynamically loaded into the process' address space, the vTable pointer is located in the library, and returned to the requester.

In this way, PC provides something akin to the automated, dynamic loading of classes (via JVM .class files) that is provided by most Big Data systems.

5.4 Automatic Deep Object Copies

Since the fundamental goal of PC object model design is zero-cost data movement—an allocation block should be transferable across processes and machines immediately usable with no pre- or post-processing—one potential problem is off-block Handles, when there is a Handle located in one allocation block that points to a PC Object located in another allocation block. The Handle may be valid, but when the Handle's allocation block is moved to a new process where the target block is not located, the Handle cannot be dereferenced without a runtime error.

PC simply prevents this situation from ever happening. Whenever an assignment operation on Handle that is physically located in the active allocation block results in that Handle pointing outside of the block, a deep copy of the target of the assignment is automatically performed. This deep copy happens recursively, so any Handles in the copied object that point outside of the active allocation block have *their* targets deep-copied to the active block. For example, consider the following code:

```
makeObjectAllocatorBlock (1024 * 1024);
Handle <Vector <double>> data =
     makeObject <Vector <double>> ();

for (int i = 0; i < 1000; i++)
     data->push_back (i * 1.0);

makeObjectAllocatorBlock (1024 * 1024);
Handle <MatrixBlock> myMatrix =
     makeObject <MatrixBlock> ();
myMatrix->value = data; // deep copy of data happens
```

At the second makeObjectAllocatorBlock, the original allocation block, holding the list of doubles pointed to by data, becomes inactive. The submatrix myMatrix is allocated to the new active block. Hence, the assignment of data to myMatrix->value is cross-allocation block, and a deep copy automatically happens to ensure that the current block is zero-cost copy-able.

Such cross-block assignments require deep copies and are expensive, but they are rare, and a programmer who understands the cost can often avoid them, making sure to allocate data that must be kept together to the same block. Again, this is in-keeping with PC's design philosophy: trust the ability of the programmer to do the right thing, in the small.

6 EXPERIMENTS

6.1 Overview

In this section, we describe our experimental evaluation of PC. The aim is to answer the following questions:

(1) How useful is PC for the construction of high-performance Big Data tools and libraries?

(2) Can the PC object model be used to build computations that efficiently manipulate highly nested and complex objects?

(3) How well does PC compare to alternative systems for developing scalable ML algorithm implementations?

(4) How do optimizations such as zero-cost data movement and PC's unique pipeline processing contribute to performance gains?

In an attempt to answer each of these questions, we perform three different benchmarking tasks:

(1) We constructed a scalable, distributed linear algebra library called lilLinAlg on top of PC, and evaluated its performance for three computations: distributed Gram matrix construction, distributed least squares linear regression, and distributed nearest neighbor search.

(2) To test the utility of the PC object model, we first denormalized the TPC-H database [25] into an object-oriented representation, and then benchmarked two reasonably complex analytical computations.

(3) We also use PC as a platform to implement a machine learning library that consists of three ML algorithms: Latent Dirichlet Allocation (LDA), which is used for textual topic mining; Gaussian mixture model (GMM) learning, which is used to cluster data using a mixture of high-dimensional Normal distributions, and the simplest, k-means clustering.

Experimental Environment. All of the experiments reported in this paper were performed using a cluster of eleven Amazon EC2 m2.4xlarge machines. Each machine ran Ubuntu 16.04, except the linear algebra experiments on SciDB [19, 52] with version 14.8, which was supported on Ubuntu 12.04 [7]. Each machine had eight virtual cores, 200GB SSD disk, and 68 GB of RAM. In each PC cluster that we built, one of the eleven machines served as the master node and the rest ten machines served as worker nodes.

Since Apache Spark is one of the most widely-used Big Data system both for applications programming and for tool and library development, most (though not all) of our comparisons were with Spark (version 2.1.0) and HDFS. The configuration of the Spark cluster are carefully tuned for each experiment, details are listed in Section E in the Appendix. We do not clear the OS buffer cache, so HDFS data can be buffered/cached in the OS buffer cache.

6.2 Distributed Linear Algebra

Since PC is designed to support the construction of high-performance tools and libraries, our first benchmarking effort was aimed at determining whether PC is actually useful for that task. Thus, we asked a PhD student (who is also an expert programmer but at

	Gram Matrix			Linear Regression			Nearest Neighbor		
Dimensionality	10	100	1000	10	100	1000	10	100	1000
PC (`lilLinAlg`)	00:07	00:09	00:39	00:14	00:22	00:49	00:15	00:20	01:06
SystemML	00:04$*$	00:29	01:27	00:05$*$	00:30	01:22	00:04$*$	00:30	01:32
Spark `mllib`	00:20	00:54	17:31	00:35	01:01	17:42	01:20	04:49	14:30
SciDB	00:03	00:17	03:20	00:15	00:33	06:04	00:28	02:56	06:24
ScaLAPACK	-	-	00:15	-	-	00:11	-	-	-

Table 1: Linear algebra performance comparison. Times in MM:SS. A star ($*$) indicates running in local mode.

the outset knew nothing of PC) to use the system to build a small Matlab-like programming language and library for distributed matrix operations. We called this implementation `lilLinAlg`. This took six weeks.

Our goal was to determine the performance and functionality that an expert programmer (but PC novice) could deliver in a short time-frame, compared to a set of established distributed Big Data linear algebra implementations: SciDB [19, 52] (built from the ground up by a team consisting of MIT students and professional developers over the last nine years), Spark `mllib` [44] (the Big Data matrix implementation shipped with Spark), and SystemML [15, 16, 31] (a matrix and machine learning implementation developed over the last seven years by a team at IBM).

Implementation. In `lilLinAlg`, a distributed matrix is stored as a set of PC `Objects`, where each object in the set is a `MatrixBlock`, similar to the class described in Section 5.1. `lilLinAlg` uses the `MatrixBlock` object to implement a set of common distributed matrix computations, including transpose, inverse, add, subtract, multiply, transposeMultiply, scaleMultiply, minElement, maxElement, rowSum, columnSum, duplicateRow, duplicateCol, and many more. However, `lilLinAlg` programmers do not call these operations directly, rather, `lilLinAlg` implements its own Matlab-like DSL. Given a computation in the DSL, `lilLinAlg` first parses the computation into an abstract syntax tree (AST), and then uses the AST to build up a graph of PC `Computation` objects which is used to implement the distributed computation. For example, at a `multiply` node in the compiled AST, `lilLinAlg` will execute a PC code similar to the following:

```
Handle <Computation> query1 = makeObject<LAMultiplyJoin>();
query1->setInput (0, leftChild->evaluate(instance));
query1->setInput (1, rightChild->evaluate(instance));
Handle <Computation> query2 =
  makeObject<LAMultiplyAggregate>();
query2->setInput(query1);
```

Here, `LAMultiplyJoin` and `LAMultiplyAggregate` are both user-defined `Computation` classes that are derived from PC's `JoinComp` class and `AggregateComp` class, respectively. Both invoke the Eigen numerical processing library [2] to manipulate `MatrixBlock` objects.

`lilLinAlg`'s DSL looks a lot like Matlab and allows very short and easy-to-read codes. For example, a least squares linear regression over a large input matrix can be easily coded as:

```
X = load(myMatrix.data);
y = load(myResponses.data);
// '* denotes transpose-multiply; %*% denotes multiply
beta = (X '*␣X)^-1␣%*%␣(X␣'* y)
```

Experiments. Our experimental benchmark consisted of three different computations: a Gram matrix computation (given a matrix $\mathbf{X}$, compute $\mathbf{X}^T\mathbf{X}$), least squares linear regression (given a matrix of features $\mathbf{X}$ and responses $\mathbf{y}$, compute $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$), and nearest neighbor search in a Riemannian metric space [42] encoded by matrix $\mathbf{A}$ (that is, given a query vector $\mathbf{x}'$ and matrix $\mathbf{X}$, find the i-th row in the matrix that minimizes $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}') = (\mathbf{x}_i - \mathbf{x}')^T\mathbf{A}(\mathbf{x}_i - \mathbf{x}')$). For each computation we used 10^6 data points and data dimensionalities of ten, 10^2, and 10^3.

In addition to `lilLinAlg` and Spark `mllib`, we use SciDB 14.8 and SystemML V1.0 for all three experiments. We carefully tuned all platforms for the best performance. In addition to the various matrix block sizes, for both Spark-based systems, we tuned driver memory size, executor memory size, number of executors, etc. For SciDB, we tuned a number of parameters: max-memory-limit, smgr-cache-size, and many others. For `lilLinAlg`, the most critical performance parameter in this case was the page size.

Results and Discussion. Experimental results are given in Table 1. For every one of the higher-dimensional computations, the `lilLinAlg` implementation was the fastest. Often, it was considerably faster. For two of the three ten-dimensional computations, SystemML was the fastest. However, in all three of the ten-dimensional computations, SystemML chose *not* to distribute the underlying computation, as it was small enough to be efficiently extracted on a single machine.

We feel that overall, these results largely validate the hypothesis that PC is an excellent platform for the construction of Big Data tools and libraries. The only distributed linear algebra implementation to approach `lilLinAlgs` performance on the larger matrices was SystemML. However, SystemML was built over many years by a team of PhDs, and research papers have been written about the technology developed for the system, including one awarded a VLDB best paper award [29]. `lilLinAlg` was developed in six weeks by a single PhD student, and it is still faster (though to be fair, SystemML has a much broader set of capabilities than `lilLinAlg`). One may conjecture that had SystemML been built on a platform such as PC rather than on Spark, it might be faster.

Despite the demonstrated benefits of building `lilLinAlg` on top of PC, we point out that PC is a young system and so it is still missing some key functionality that would boost `lilLinAlg`'s performance even more. For example, PC cannot make use of pre-partitioning of the data stored in a set. If the `MatrixBlock` objects making up a distributed matrix could be pre-partitioned based upon the row/column at load time, the expensive join for an operation such as multiplication could avoid a runtime partitioning of the data, which requires shuffling each input matrix.

Comparison with ScaLAPACK. Finally, we were curious as to how these various tools compare with hand-coded ScaLAPACK solutions (Intel MKL, 2018 release). ScaLAPACK is an HPC tool for

Number `Customer` objects Kryo data size	2.4M 41.5GB	4.8M 83.1GB	9.6M 167.2GB	14.4M 251.1GB	19.2M 333.2	24M 416.2GB
	`Customers per Supplier`					
PlinyCompute: hot storage	00:11	00:19	00:35	00:51	01:08	01:21
Spark: hot HDFS	01:04	01:53	03:24	04:54	06:25	08:16
Spark: in-RAM deserialized RDD	00:16	00:29	00:56	01:21	02:18	03:56
	top-*k* Jaccard					
PlinyCompute: hot storage	00:03	00:03	00:04	00:05	00:05	00:06
Spark: hot HDFS	00:56	01:38	03:01	04:01	05:22	06:34
Spark: in-RAM deserialized RDD	00:08	00:12	00:21	00:32	01:11	02:38

Table 2: Performance comparison for large-scale OO computation. Times in MM:SS.

distributed matrix computations and has been under development for decades. Though it is unlikely to be used on the same sort of application as the other tools tested—least squares regression requires 148 lines of C code in ScaLAPACK plus loading of the compute grid, compared with three lines in `lilLinAlg`—it should nevertheless give a good idea of the upper bound on distributed linear algebra performance. For each of the 1000-dimensional problems for gram matrix and linear regression, we find that PC is within a factor of five of the HPC solution. Element-wise matrix-matrix multiply is not available on ScaLAPACK and we could not find an efficient solution to nearest neighbour search.

6.3 Big Object-Oriented Data

Programming with objects is attractive as a programming paradigm, but often costly in terms of performance, particularly for distributed computing. One answer is to simply disallow complex objects. The developers of Apache Spark, for example, have attempted to move away from object programming and towards a relational model of programming (with Datasets and Dataframes). Thus, the question we address in this particular set of experiments is: can the PC object model facilitate computations of heavily nested, complex objects?

Data Representation. We implement two different complex object computations on top of PC and on top of Spark. Both of these computations are over large data sets that store an instance of the TPC-H database [25]. But rather than storing the data set relationally, we denormalized the data into a set of nested objects. The simplest objects used in the denormalized TPC-H schema are `Part` objects, which do not look very different from the records in the `part` schema of the TPC-H database. In PC, these are defined as:

```
class Part : public Object {
private:
   int partID;
   String name;
   String mfgr;
  /* six more members and several public methods... */};
```

`Supplier` objects are defined similarly. `Lineitem` objects contain nested `Part` and `Supplier` objects. Then, `Order` objects have a nested list of `Lineitem` objects, and `Customer` objects have a nested list of all of the `Lineitem` objects:

```
class Customer : public Object {
private:
   Vector <Handle <Order>> orders;
   int custkey;
   String name;
  /* seven more members and several public methods... */};
```

Implementation and Experiments. Since the TPC-H data were denormalized into objects (making them non-relational), we wanted to create realistic computations that were specifically geared towards this representation. Our first computation is designed to require a lot of object manipulation, including insertion into complex containers. The second requires repeatedly computing the Jaccard similarity between sets, which is un-natural in a relational setting.

The first computation is *customers per supplier* where we compute, for each supplier, the list of `partIDs` that the supplier has sold to each of the supplier's customers. For each supplier, the result is an object that contains the supplier's name (as a `String`) and an object of type `Map <String, Handle <Vector <int>>>`. In this object, the `String` is the name of the customer, and the `Vector` stores the list of `partIDs` sold to that customer.

To run this computation on PC, we use two different PC `Computation` classes. The first, `CustomerMultiSelection`, transforms each `Customer` object to one or more `SupplierInfo` objects. Each `SupplierInfo` contains the name of a supplier and a `Handle` to a `Map` whose key is the name of the customer and whose value is the list of `partIDs` that the supplier has sold to the customer. The second `Computation`, called `CustomerSupplier- PartGroupBy`, groups all of those `SupplierInfo` objects according to the name of the supplier, computing, for each supplier, the map from customer name to `Vector` of `partIDs`.

On Spark we implemented an algorithmically equivalent, carefully-tuned code. Note that since the objects are all highly nested, it was not possible to develop a satisfactory Dataset or Dataframe implementation, and we used RDDs. Since Spark makes use of lazy evaluation, it is not possible to collect a timing for this computation unless we actually *do* something with the result. So in both the PC and the Spark computations we added a final count of the number of customers in each `Map` in each `SupplierInfo` object.

The second computation is the *top-k closest customer part sets* computation. For a given `Customer` object, we go through all of the associated `Order` objects and obtain the complete list of `partIDs` for each order. All duplicate `partIDs` are removed from this list, and then the Jaccard similarity between the resulting `partID` list and a special, query list are computed. This is done for all of the `Customer` objects, and the *k* `partID` lists with the closest similarity to the query list are returned.

In PC, the one query-specific `Computation` object that was implemented for the top-*k* closest customer part sets computation is the `TopJaccard` class, which is responsible for extracting a value to drive the top-*k* computation (in this case, the Jaccard similarity)

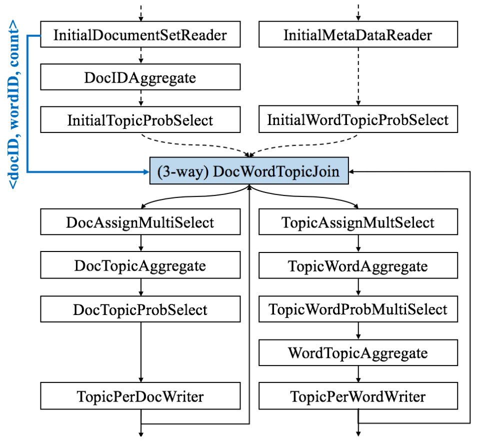

Figure 2: PC LDA's `Computation` objects and input-output dependencies. Computations connected by dash lines will only run once, during initialization.

as well as the object to be associated with that value (in this case, the `custkey` and the list of `partIDs` sold to that customer).

For both PC and Spark, and for both computations, we created TPC-H data set of various sizes: 2.4 million, 4.8 million, 9.6 million, 14.4 million, 19.2 million and 24 million `Customer` Objects respectively. For the "top-k closest customer part sets" computation, k was chosen to be $\frac{1024}{2.4 \times 10^6}$ times the size of the data.

For both Spark computations, we performed two runs at each data set size. For the first, we stored the data in HDFS, and measured the time to execute the query starting with a read from HDFS. For the second, we made sure that the data were de-serialized and stored in RAM by Spark. To do this, we applied a `distinct().count()` operation to an RDD storing `Customer` objects (thus ensuring full deserialization) before running each query. All data were serialized using Kryo, and parameters such as data partition size and parallelism are fully tuned to obtain optimal performance. For PC, we ran only one version of the computation where the various `Customer` objects are stored in PC's storage system.

Results and Discussion. Results are shown in Table 2. The difference in speed between the PC and the Spark implementations is significant. When Spark data are stored in a hot HDFS, the two computations are 6× to 66× faster in PC. This is an apples-to-apples comparison, because in both systems, the data are being fetched from system storage, where they can be buffered in OS buffer cache (by HDFS) or PC buffer pool, respectively.

If the Spark data are already fully deserialized and stored as an RDD in memory, then PC is still between 1.5× and 26× faster for both computations. Since in PC there is no distinction between serialized and deserialized data, there is no analogous case in PC.

Note that Spark had about the same performance for both computations. This is a bit surprising because the top-k computation seems, on the face of it, much easier than the "parts per supplier per customer" computation. One explanation could be not that Spark is surprisingly slow on the top-k, but that PC is relatively slow on the "parts per supplier per customer" computation. Profiling reveals that PC spends a lot of time on `String` operations (looking up particular customers in the `Map <String, Handle <Vector <int>>>`, for example). That is because PC `Strings` have the same representation in-RAM and on-disk, they are purposely designed to take little space—they do not cache hash values, for example (unlike Java `Strings`).

6.4 Machine Learning

Finally, we consider three common machine learning algorithms: LDA, GMM and k-means. These algorithms were chosen as they are among the most ubiquitous machine learning algorithms; virtually every widely-used machine learning library will contain all three.

Latent Dirichlet Allocation. The first algorithm we implemented was a Gibbs sampler for Latent Dirichlet Allocation, or LDA. LDA is a common text mining algorithm. While LDA implementations are common, we chose a particularly challenging form of LDA learning: a word-based, non-collapsed Gibbs sampler [20]. The LDA implementation is *non-collapsed* because it does not integrate out the word-probability-per-topic and topic-probability-per-document random variables. In general, collapsed implementations that do integrate out these values are more common, but such collapsed implementations cannot be made ergodic in a distributed setting (where ergodicity implies theoretical correctness in some sense). Our implementation is *word based* because the fundamental data objects it operates over are `(docID, wordID, count)` triples. This generally results in a more challenging implementation from the platform's point-of-view because it requires a many-to-one join between words and the topic-probability-per-document vectors. In our experiments, there are approximately 700 million such triples, and each vector is around 1KB. Hence, the many-to-one join between them results in 700GB of data. If the platform does not manage this carefully, performance will suffer.

The full PC LDA implementation requires fifteen different `Computation` objects, as shown in Figure 2. Each iteration requires a three-way `JoinComp`, three `MultiSelectionComps`, and three `AggregateComps`, among others. Our PC LDA computation makes use of the GSL library [3] to perform all necessary random sampling.

Spark `mllib`'s LDA implementation is based on expectation maximization and online variational Bayes. Therefore, we had a Spark expert carefully implement an algorithmically equivalent word-based, non-collapsed LDA Gibbs sampler. His implementation used both Spark's RDD and Dataset APIs as appropriate. The required statistical computations use the `breeze` package.

Gaussian Mixture Model. A Gaussian mixture model (GMM) is a generative, statistical model where a data set are modeled as having been produced by a set of Gaussians (multi-dimensional normal distributions). Learning a GMM using the expectation maximization (EM) algorithm is one of the classical ML algorithms. EM is particularly interesting for a distributed benchmark because in theory, the running time should be dominated by linear algebra operations (such as repeated vector-matrix multiplications).

Our PC implementation uses GSL [3] along with a single `AggregateComp` object, which contains inside of it the current version of the learned GMM model. As this `AggregateComp` is executed, a soft assignment of each data point to each Gaussian is performed, and updates to each of the Gaussians are accumulated.

PlinyCompute	Spark 1: vanilla	Spark 2: also with join hint	Spark 3: also with forced persist	Spark 4: also hand-coded multinomial
02:05	50:20	17:30	09:26	05:26

Table 3: LDA performance comparison. Times in MM:SS.

Dimensionality Number of points	100 10^7	300 10^6	500 10^6
PlinyCompute	00:30	00:38	1:38
Spark `mllib`	1:41	1:54	5:05
Tensorflow	00:44	00:18	00:36

Table 4: GMM performance comparison. Times in MM:SS.

Dimensionality Number of points	10 10^9	100 10^8	1000 10^7
PlinyCompute	00:37	00:09	00:06
Spark `mllib` RDD API	01:02	00:28	00:23
Spark `mllib` Dataset API	01:43	00:25	00:22
Tensorflow	00:19	00:11	00:12

Table 5: *k*-means performance comparison. Times in MM:SS.

It turns out that an algorithmically equivalent implementation exists in Spark `mllib`. There are only slight differences between the two; for example, PC computes uses the standard "log space" trick to compute the soft assignment and avoid underflow, whereas `mllib` uses thresholding.

***k*-Means.** *k*-means is a now-classic benchmark for Big Data ML. We specifically developed our PC *k*-means implementation to closely match the implementation in Spark's `mllib`.

Experiments. On the aforementioned eleven-node cluster, we ran all ML experiments using PC and Spark 2.1.0. For LDA, we created a semi-synthetic document database with 2.5 million documents from 20-Newsgroups data set by concatenating random pairs of newsgroup postings end-on-end. There are more than 739 million `(docID, wordID, count)` triples in the data set. We use a dictionary size of 20,000 words and a model size of 100 topics.

For GMM, we generated random data for three test cases: 10^7 data points with 100 dimensions, and 10^6 data points with 300 and 500 dimensions, respectively. For *k*-means, we generate random data for 10^9 data points with ten dimensions, 10^8 data points with 100 dimensions, and 10^7 data points with 1000 dimensions. Ten Gaussians are used for GMM, and 100 clusters for *k*-means.

All results are averaged over five iterations.

Results and Discussion. LDA Results (per iteration) are illustrated in Table 3. While Spark performed well, the amount of work required to arrive at a good solution was significant, representing about a week of tuning. First, among other things, our Spark expert had to force a broadcast join. Then, it was necessary to force Spark to persist the result of one of the joins for later use. Finally, it was necessary to hand-code a Multinomial sampler (avoiding the use of `breeze` for multinomial sampling) to obtain an implementation that was competitive with PC. This last bit of tuning (of course) can't be blamed on Spark, but the experience overall is illustrative: forcing a particular join and forcing a particular persist are *workload specific* optimizations. They may work for one workload but be a poor choice for another, and require a tool end-user to actually change library code to achieve performance. In contrast, like a database system, PC is fully declarative in-the-large. Decisions such as using a broadcast join instead of a full hash join as well as which intermediate results to materialize (and which to pipeline or discard) are fully automated.

The results for GMM are illustrated in Table 4. Here, PC achieved a 3× speedup compared with Spark `mllib`'s GMM implementation (using RDD APIs) for all cases. As illustrated in Table 5, for *k*-means, PC achieved a 2× to 4× speedup compared with the Spark `mllib` implementation.

Comparison with TensorFlow. We also implemented GMM and KMeans on top of TensorFlow version 1.5—unlike LDA, these computations map very nicely to the array-based abstraction offered by TensorFlow. The comparison is informative, but it is not necessarily fair. All data in Tensorflow are represented as arrays (tensors) while the PC and Spark/`mllib` implementations view the data as a large set of vectors. The TensorFlow representation likely leads to lower overhead and higher performance, though it is arguably less natural (this is not the implementation used in the `mllib` library implementation, for example). The PC and Spark implementations *could* also have utilized a small number of large matrices, though they did not. Another key difference is that all TensorFlow computations use 32-bit `floats` rather than 64-bit `doubles` like PC and Spark. Given TensorFlow's origins as a platform for deep learning, it has poor support for double-precision arithmetic. We find that the TensorFlow implementations are, on average, about as fast as the PC implementations.

6.5 Effect of Individual Optimizations

LDA is a good example of how PC's declarativity enables the system to automatically make implementation choices that require careful tuning on Spark. But even after optimizing the Spark implementation as well, PC is still faster. Which of the ideas presented in the paper help to achieve good performance? Or is the good performance simply related to using C++ rather than Java?

Benefit of Zero-Cost Data Movement. We give some evidence that (de-) serialization costs are significant by examining Spark. The (de-) serialization overhead in Spark falls into two categories: the overhead for Spark closures, and the overhead for RDDs. The former can be obtained by examining Spark event logs. We can determine RDD (de-) serialization overhead by running a Java profiler to sample the execution and estimate the amount of time spent in each method during workload execution.

In Figure 4 we show the fraction of time spent on for several of the benchmarks we ran. The overhead ranges from around 30% for LDA (which is a combined Dataset and RDD implementation) to up to 75% for Top-*k* Jaccard. This seems to show the benefit of zero-cost data movement; removing up to 75% of the execution time by removing (de)serialization costs seems very significant.

Benefit of Vectorization and Compilation. Measuring the benefit of vectorized execution in PC is relatively easy; we set the

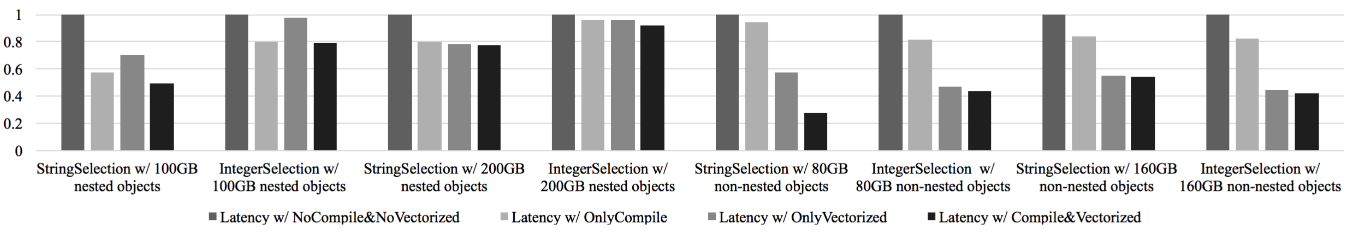

Figure 3: Normalized processing times when enabling/disabling vectorized and compiled pipeline processing.

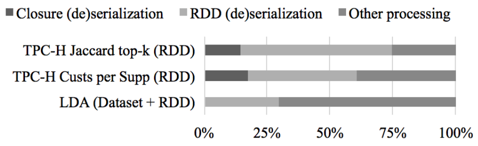

Figure 4: Spark (de)serialization overhead.

Workloads	Non-vectorized	Vectorized
LDA (Dataset + RDD)	12:34	02:05
TPC-H Custs per Supp (RDD)	00:12	00:11
TPC-H Jaccard top-k (RDD)	00:03	00:03

Table 6: Vectorized vs. not-vectorized PC processing times.

length of each vector to one, which results in one object at-a-time processing.

Measuring the benefit of running compiled code is more difficult because PC's execution engine is designed around compilation; we cannot simply turn it off. However, one of the benefits of compilation is that the compiler can inline function calls, avoiding the need for expensive virtual function invocations. In a simple, straight-line, TCAP computation without an aggregation or join, we can induce a virtual function call by using `makeLambdaFromMethod` in conjunction with a virtual method.[3]

This results in four combinations to try: one/not one for vector length, and virtual/not virtual for the method call. We implemented two simple computations over TPC-H `Customer` objects and execute each of these four combinations. The first computation looks for a certain `custkey` value (`custkey` is an integer) and the second looks for a particular `mktsegment` value (this is a string). We ran these computations over flat, relational-style `Customer` objects, as well as the much larger denormalized `Customer` objects described previously. The results are shown in Figure 3.

Here we see that both optimizations together can achieve up to 3.65× speedup. The gains were less pronounced for the denormalized `Customer` objects. These objects are 30× larger than the flat objects, meaning that scanning a set of such objects likely induces considerable memory traffic and cache misses, mitigating the effectiveness of compilation and vectorization.

Since it is easy to turn off vectorized processing for the more complex workloads, we tested this as well. Results are shown in Table 6. Vectorized processing is enormously helpful to LDA (which processes a small `(docID, wordID, count)` triples), but not very helpful when processing large denormalized TPC-H objects.

[3]This trick is less useful for more complicated computations containing joins and aggregations. When PC runs such a computation, it executes many different functions that are compiled to run on the specific input types.

Effect of C++. We also wanted to understand whether C++ itself results in a fast system. We ran a simple micro-benchmark on an Amazon EC2 `m2.4xlarge` machine, where we compared the various packages for statistical/scientific computing used throughout these experiments. We use a matrix multiplication to compare Java `breeze` (used in all Spark implementations) with Eigen (used by PC's `lilLinAlg`) and GSL (used in all of our PC ML implementations). We find that Java Breeze has similar performance with Eigen and 8× better performance than GSL. Thus, in one way, our C++ implementations were at a significant disadvantage. More details can be found in our technical report [60].

7 PRODUCTIVITY COMPARISON

PC may be faster, but is it significantly more difficult to develop for than a platform that uses a managed runtime?

PC gives a programmer more flexibility, and with that can come certain costs—less knowledgeable developers may find PC difficult to code for. But, by at least one metric—source lines of code (SLOC)—PC is *not* any more difficult as a development target than Spark. Table 7 shows the SLOC counts for the various implementations described here. We included code in Spark `mllib` and TensorFlow factorization package for GMM and k-means. There is not a significant difference between PC and Spark. While for LDA and GMM PC required 2× to 3× the code required for Spark, a lot of that was related to the fact that Scala has a nicer interface to numerical routines (via `breeze`).

Application	PC	Spark	Tensorflow
`lilLinAlg`	3505	3130 (Scala)	-
Custs per Supp	929	953 (Java)	-
top-k Jaccard	793	966 (Java)	-
LDA	1038	343 (Scala/ breeze)	-
GMM	932	474 (Scala/ `breeze`)	595 (Python)
k-means	695	670 (Scala)	843 (Python)

Table 7: Source lines of source code comparison.

8 CONCLUSIONS

We have presented PlinyCompute, which is unique in that *in the large*, it gives programmers a high-level declarative interface, and *in the small*, it gives experienced systems programmers access to a persistent object data model and API as well as a memory management system designed for high-performance, data-intensive operations. Through extensive benchmarking, we have shown that implementing complex object manipulation and library-style computations on top of PC can result in a speedup of 2× to more than 50× or more compared to equivalent implementations on Spark.

Acknowledgements. Work presented in this paper has been supported by the NSF under grant Nos. 1355998 and 1409543, and by the DARPA MUSE program, award No. FA8750-14-2-0270. Thanks to SIGMOD 2018's anonymous reviewers for their insightful feedback on earlier versions of this paper. Thanks to Leo Du, Joseph Hwang, Dimitrije Jankov, Jessica Yu and others for their contributions to PlinyCompute development.

REFERENCES

[1] [access date]. DL4J. https://deeplearning4j.org/. (Online). Oct 1, 2017.
[2] [access date]. Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page. (Online). Oct 1, 2017.
[3] [access date]. GSL - GNU Scientific Library. https://www.gnu.org/software/gsl/. (Online). Oct 1, 2017.
[4] [access date]. Mahout. http://mahout.apache.org. (Online). Oct 22, 2016.
[5] [access date]. Mahout Samsara. https://mahout.apache.org/users/environment/out-of-core-reference.html. (Online). Oct 22, 2016.
[6] [access date]. Project Tungsten Bringing Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. (Online). Oct 1, 2017.
[7] [access date]. SciDB's supported OS. http://www.paradigm4.com/HTMLmanual/14.8/scidb_ug/pr01s01.html. (Online). Nov 1, 2017.
[8] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1664–1665.
[9] Martín Abadi, Ashish Agarwal, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
[10] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1566–1569.
[11] Alexander Alexandrov et al. 2014. The Stratosphere platform for big data analytics. *VLDBJ* 23, 6 (2014), 939–964.
[12] Michael Armbrust, Reynold S Xin, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.
[13] MKABV Bittorf, Taras Bobrovytsky, et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*.
[14] Stephen M Blackburn, Robin Garner, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.
[15] Matthias Boehm, Michael W Dusenberry, et al. 2016. SystemML: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.
[16] Matthias Boehm, Shirish Tatikonda, et al. 2014. Hybrid parallelization strategies for large-scale machine learning in SystemML. *Proceedings of the VLDB Endowment* 7, 7 (2014), 553–564.
[17] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.
[18] Sebastian Breß, Bastian Köcher, et al. 2017. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *arXiv preprint arXiv:1709.00700* (2017).
[19] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 963–968.
[20] Zhuhua Cai, Zekai J Gao, et al. 2014. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 1371–1382.
[21] Paris Carbone, Asterios Katsifodimos, et al. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[22] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 34–43.
[23] Weidong Chen, Michael Kifer, and David S Warren. 1993. HiLog: A foundation for higher-order logic programming. *The Journal of Logic Programming* 15, 3 (1993), 187–230.
[24] Edgar F Codd. 1971. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. ACM, 35–68.
[25] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.
[26] Andrew Crotty, Alex Galakatos, et al. 2015. Tupleware:" Big" Data, Big Analytics, Small Clusters.. In *CIDR*.
[27] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.
[28] Ankur Dave, Alekh Jindal, et al. 2016. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.
[29] Ahmed Elgohary, Matthias Boehm, et al. 2016. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment* 9, 12 (2016), 960–971.
[30] Grégory M Essertel, Ruby Y Tahboub, et al. 2017. Flare: Native compilation for heterogeneous workloads in Apache Spark. *arXiv preprint arXiv:1703.08219* (2017).
[31] Amol Ghoting, Rajasekar Krishnamurthy, et al. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*. 231–242.
[32] Joseph E Gonzalez, Reynold S Xin, et al. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.
[33] Goetz Graefe. 1990. *Encapsulation of parallelism in the Volcano query processing system*. Vol. 19. ACM.
[34] William Gropp, Ewing Lusk, et al. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.
[35] Dan Grossman, Greg Morrisett, et al. 2002. Region-based memory management in Cyclone. *ACM Sigplan Notices* 37, 5 (2002), 282–293.
[36] Stratos Idreos, Fabian Groffen, et al. 2012. MonetDB: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering* 35, 1 (2012), 40–45.
[37] Nicolai M Josuttis. 2012. *The C++ standard library: a tutorial and reference*. Addison-Wesley.
[38] Peter J Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.. In *USENIX Winter*, Vol. 1994. 23–36.
[39] Yannis Klonatos, Christoph Koch, et al. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.
[40] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.
[41] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[42] Guy Lebanon. 2006. Metric learning for text documents. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 4 (2006), 497–508.
[43] Lu Lu et al. 2016. Lifetime-Based Memory Management for Distributed Data Processing Systems. *VLDB* 9, 12 (2016), 936–947.
[44] Xiangrui Meng, Joseph Bradley, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
[45] Dale Miller. 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation* 1, 4 (1991), 497–536.
[46] Fabian Nagel, Gavin Bierman, and Stratis D Viglas. 2014. Code generation for efficient query processing in managed runtimes. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1095–1106.
[47] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.
[48] Kay Ousterhout, Ryan Rasti, et al. 2015. Making Sense of Performance in Data Analytics Frameworks.. In *NSDI*, Vol. 15. 293–307.
[49] Shoumik Palkar, James J Thomas, et al. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.
[50] Juwei Shi, Yunjie Qiu, et al. 2015. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2110–2121.
[51] Sourav Sikdar, Kia Teymourian, and Chris Jermaine. 2017. An Experimental Comparison of Complex Object Implementations for Big Data Systems. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, 432–444.
[52] Michael Stonebraker, Paul Brown, et al. 2011. The architecture of SciDB. In *Scientific and Statistical Database Management*. Springer, 1–16.
[53] Michael Stonebraker, Lawrence A Rowe, et al. 1990. Third-generation database system manifesto. *ACM SIGMOD record* 19, 3 (1990), 31–44.
[54] Yuanyuan Tian, Shirish Tatikonda, and Berthold Reinwald. 2012. Scalable and numerically stable descriptive statistics in systemml. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 1351–1359.
[55] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.
[56] Yuan Yu et al. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.. In *OSDI*, Vol. 8. 1–14.
[57] Matei Zaharia et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2–15.

[58] Matei Zaharia, Mosharaf Chowdhury, et al. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud*. 1–10.
[59] Yili Zheng, Amir Kamil, et al. 2014. UPC++: a PGAS extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1105–1114.
[60] Jia Zou, R Matthew Barnett, et al. 2017. PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development. *arXiv preprint arXiv:1711.05573* (2017).
[61] Marcin Zukowski, Peter A Boncz, et al. 2005. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.

A RELATED WORK

Code Generation. DryadLinq [56] allows a user to express distributed data flow computations in a high-level language like C# and strongly typed .NET objects, and it compiles those computations into .NET assembler.

TupleWare [26] supports multiple high-level languages (any language with an LLVM compiler) and aims to optimize for UDFs by utilizing code generation to integrate UDF code with the engine code. Weld [49] is a recent system developed in Scala and Python. It proposes a common runtime for data analytics libraries by asking library developers to express their work using a new intermediate representation (IR) and compiles this IR into multi-threaded code using LLVM. Then, application developers can use unified APIs to call different libraries from Weld.

Besides those LLVM-based approaches, Flare [30] proposes a new backend for Apache Spark [12] that compiles SparkSQL queries with user defined scala functions into native code to be executed.

These systems (and also PC) are all focused on query execution via code generation for UDF-centric computations, to reduce virtual function call overhead. However, PC is unique in the sense that the goal of PC is to allow for efficient distributed programming with complex objects, so PC's code generation is closely integrated with other system features such as the object model, the TCAP IR and PC's lambda calculus.

Optimized Memory Management. Apache SparkSQL [12] serializes relational tables into byte arrays and stores the serialized bytes in a main-memory columnar storage. Spark Tungsten [6] optimizes the Spark execution backend by grouping execution data (such as hashed aggregation data) into byte arrays and data can be allocated off-heap via the sun.misc.Unsafe API, reducing GC overhead. Deca [43] is a memory management framework aiming at reducing GC overhead. It stores various Spark data types, e.g. UDF variables, user data and shuffle data into different off-heap containers so that objects in each container can have a similar lifetime and can be recycled together. All of these methods attempt to alleviate GC overhead; in contrast, PC simply does not use a managed runtime.

Relational Processing on Binary or Structured Objects. Apache Flink [11] uses reflection to analyze Java/Scala object types, and it maps each object type to one of a limited set of fundamental data types to provide comparators to efficiently compare binary representations and extract fixed-length binary key prefixes without deserializing the whole object.

Spark [6] has introduced Datasets/Dataframes to encode JVM objects into a relational, binary representation. Datasets/Dataframes enable relational-style processing through a relational query optimizer called Catalyst and also enables Java intermediate code generation to reduce virtual function call overhead.

Such techniques significantly boost performance by moving away from a flexible, object-oriented type of system to a more relational system. It is known that relational systems can be fast, but they limit the sort of applications that can easily be coded on top of the system. In contrast, PC attempts to offer a fully object-oriented interface.

Native Systems. Impala [13] is a C++-based SQL query engine that relies on Hadoop for scalability and flexibility in interface and schema. Impala compiles SQL into LLVM assembly code. However, Impala uses a relational data model, though it can read/write semi-structured data in storage formats such as Arvo, Parquet, RC and so on from/to external storage such HDFS, using standard serialization/deserialization methods.

Tensorflow [9] is a distributed computing framework mainly designed for deep learning. It mainly supports processing of numerical data with a very limited set of types. Tensorflow provides a much lower level API than PC's declarative interface, based on tensors, variables and sessions.

HPC systems propose distributed shared memory (DSM) [38] and a partitioned global address space (PGAS) [59] to enable remote object access without serialization and deserialization. Those systems, however, are not optimized for moving a large number of objects, and persisting data to disk still requires serialization and deserialization.

B WORKER ARCHITECTURE

Each worker node runs two processes: the *worker front-end process* and the *worker backend process*. Dual processes are used because the backend process executes potentially unsafe native user code. If user code happens to crash the worker backend process, the worker front-end process can re-fork the worker backend process. The worker front-end process interfaces with the master node, providing a local catalog manager and a local storage server (including a local buffer pool) and crucially, it acts as a proxy, forwarding requests to perform various computations to the worker backend process, where computations are actually run.

C OPTIMIZATION IN PC

In PC (as in any declarative data processing system), optimizing a user-specified computation gives rise to two distinct optimization problems: *logical* and *physical* optimization. Logical optimization executes transformations over a TCAP program that are guaranteed to result in a new TCAP program that must return the same result, but executes more efficiently. Physical optimization refers to the task of choosing the details of how to actually execute a TCAP computation over a given data set in a compute cluster.

C.1 Logical TCAP Optimization

Logical optimization of TCAP programs in PC is performed by a rule-based optimizer. The simple transformations currently available to the TCAP optimizer include some that resemble those available in a classical relational optimizer, as well as those that more closely resemble transformations used in a compiler for a high-level programming language. For example an example of the latter, imagine that a user supplies a `SelectionComp` with the following `getSelection()`:

```
Lambda <bool> getSelection (Handle <Emp> emp) {
   return makeLambdaFromMethod
      (emp, getSalary) > 50000 &&
      makeLambdaFromMethod (emp, getSalary) < 100000;
}
```

PC would compile this into the following (naive) TCAP:

```
JK2_1(emp,mt1) <= APPLY(In(emp), In(emp), 'Sel_43',
  'method_call_1',[('type', 'methodCall'),('methodName',
  'getSalary')]);

JK2_2(emp,bl1) <= APPLY(JK2_1(mt1), JK2_1(emp),
  'Sel_43','>_1',[('type','const_comparison'),
   ('op','>')]);

JK2_3(emp,bl1,mt2) <= APPLY(JK2_2(emp), JK2_2(emp,bl1),
  'Sel_v3', 'method_call_2',[('type','methodCall'),
  ('methodName', 'getSalary')]);

JK2_4(emp,bl1,bl2) <= APPLY(JK2_3(mt2), JK2_3(emp,bl1),
  'Sel_43','<_1',[('type','const_comparison'),
  ('op','<')]);

JK2_5(emp,bl3) <= APPLY(JK2_4(bl1,bl2), JK2_4(emp),
  'Sel_43', '&&_1',[('type','bool_and')]);

JK2_6(emp) <= FILTER(JK2_5(bl3), JK2_5(emp), 'Sel_43',[]);
```

This TCAP program first calls the method `getSalary()` on the input `Emp` objects to produce a new vector list `JK2_2`, storing the result of the method call in `JK2_1.mt1`. After comparing `JK2_1.mt1` to `50000`, the result of the method call is dropped. The method is then called once again on `JK2_2.emp` and the result compared with `100000` to produce `JK2_4`, at which point the two boolean vectors are "anded" and the result is filtered.

Obviously, there is a redundancy here as the method `getSalary()` will be called twice on each row in the input set. Hence, the second call should automatically be removed as being redundant. The TCAP optimization rule leading to its removal is:

(1) If two `APPLY` operations are both of type `methodCall` and both invoke the same `methodName`;
(2) And one `APPLY` operation is the ancestor of the other in the TCAP graph;
(3) And both `APPLY` operations operate over the same data object;
(4) Then the second `APPLY` operation can be removed, and the result of the first `APPLY` carried through the graph.

Our current TCAP optimizer also has rules that resemble classical relational optimizations, such as pushing down filter operations, and performing early projection of vectors from a vector list so that they need not be moved across machines during distributed joins and aggregations.

Ultimately, we plan to implement a full cost-based logical optimizer that can handle issues such as choosing the best join order. This will be challenging, because PC is geared towards computations over user defined functions for which good statistics are typically not available. We plan to investigate sampling and learning-based methods for optimizing over such computations.

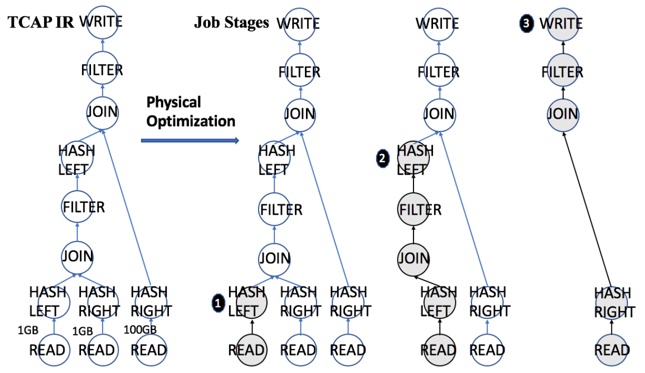

Figure 5: Decomposing a TCAP computation into job stages.

C.2 Physical TCAP Optimization

Currently, our physical TCAP optimizer makes two classes of decisions: (1) how to decompose the TCAP computation into individual pipelines (or *job stages*) and (2) which algorithms to run to realize the required distributed computations.

Consider Figure 5, which depicts the decomposition problem. The TCAP program in this figure is depicted as a graph; nodes are TCAP operations and the leaves of the graph represent source nodes.

Decomposition begins by checking the size of the set to be scanned at each source node. As a heuristic, the physical optimizer chooses source node with the smallest input size to construct the first job stage. The smallest input is chosen because the result of the job stage must typically be shuffled around the cluster; choosing a small set minimizes communication. The job stage is constructed by traversing TCAP operations from the source node until the first *pipeline breaker* is found. A pipeline breaker is a `JOIN` or `AGG` TCAP operation that requires the set of vector lists to be materialized and moved around the cluster for further processing. The resulting list of TCAP operations is used to construct a job stage that is run at each node in the cluster.

Once the first job stage is executed, the process of constructing additional job stages is repeated recursively, as shown in Figure 5. In this example, three pipelines are constructed and executed.

Unless a pipeline ends at a TCAP `WRITE` operation (which saves the result of a computation in an output set), a special output pipeline stage must be appended to the end of the job stage. For example, if the job stage ends at an `AGG`, a pipeline stage is added that builds a hash table on the grouping attribute and pre-aggregates the vector lists coming through the pipeline. Choosing which output pipeline stage to append to the end of the job stage partially determines what distributed algorithm will be run. For example, if the input set is small (smaller than available RAM) and the job stage ends in a `JOIN`, then the output pipeline stage appended to the job stage builds a single hash table to hold all of the rows in the input vector lists, organized via the hashed join key. After this job stage is run, then the resulting hash table can be broadcast to implement a broadcast join. If the input set is large and the job stage ends in a `JOIN`, then the output pipeline stage builds a *partitioned* hash table, where one partition will be sent to each of the nodes in the cluster. In this case, the other input to the join is also capped with an output pipeline stage that builds a partitioned hash table, and the partitioned hash tables constructed at each node are used to implement a full distributed hash join.

Using the input to a job stage as a proxy for the result size is a reasonable heuristic, but it is easy for this heuristic to perform poorly, as the various user-defined functions executed during the job stage can radically change the size of the set of vector lists produced. A better cost model, perhaps based on sampling, would make the physical optimizer much more robust. We also plan to add logic that uses the semantic information stored in the TCAP code to recognize when data are already partitioned on the join or aggregation key, and need not be shuffled.

D OBJECT MODEL TUNING

The PC object model is designed for minimal-cost data movement, the result being that there is often no serialization or deserialization cost when moving PC `Object`'s across processes. But memory management can still be costly. Deallocating and cleaning up complex objects (in particular, instances of container classes) can require significant CPU resources, which, depending upon the circumstance, may be un-necessary. In-keeping with the assertion that application programmers should be in control of performance-critical policies, it is possible to explicitly control how memory is reclaimed and re-used during PC computations. This is facilitated through a set of *allocation policies* that a programmer can choose from.

When the reference count for a PC `Object` located in a managed allocation block goes to zero, it is deallocated. The exact meaning of "deallocated" is controllable by the programmer, via a call to the `setAllocatorPolicy` on each computation object that is created (`JoinComp`, `SelectionComp`, etc.). Currently, PC ships with three allocator policies:

Lightweight re-use. This is the default policy. When a PC `Object` is deallocated, its space in the allocation block is made available for re-use by adding the space to a pool of similarly-sized, recycled memory chunks (all recycled chunks are organized into buckets, where a chunk of size n goes into bucket $\log_2(n)$). A request for RAM in a block is fulfilled by first scanning the recycled chunks in the appropriate bucket, then attempting to allocate new space on the end of the block, if that fails.

No re-use. The space containing deallocated PC `Objects` is not-reused. Hence, it is very similar to classical, region-based allocation—though PC `Objects` are reference-counted, and a destructor is called for each unreachable PC `Object`. Although this is the most efficient allocation policy, frequent allocations of temporary PC `Objects` will result in a lot of wasted space.

Recycling. This is layered on top of lightweight re-use. When the recycling allocator is used, any time a fixed-length PC `Object` is deallocated, it is added to a list of objects all having the same type. All calls to `makeObject` with the zero-argument constructor will pull an object off of the list of recyclable objects for the appropriate type. If an object is available for recycling, it is returned. If not, or if any other constructor other than the zero-argument constructor is called, then the lightweight re-use allocator is used to allocate space for the requested object.

Platform	num executors	executor mem	executor cores	driver mem
`lilLinAlg`	10	60GB	8	50GB
TPC-H	10	50GB	7	50GB
LDA	20	26.5GB$*$	4	55GB
GMM	80	70GB	1	55GB
k-means	10	60GB	8	50GB

Table 8: Spark Configurations for Different Experiments. A star ($*$) indicates additional 4GB off heap memory is used.

Note that variable-length objects are never recycled. There are just a few of these types in PC, and they are typically used internally to implement the built-in PC container types, and not by PC application programmers. For example, PC's variable-length `Array` class is used to implement the standard PC `Vector` container. These are not recycled because recycling allocations of such objects would need to match both on type and on size. Matching on both at once would be computationally expensive, and could also allow long lists of objects to build up, waiting to be re-used.

In addition to policies that can be set on a per-computation basis, it is also possible for a programmer to supply the following policies, on a per-`Object` bases, during PC `Object` allocation:

No reference counting. This PC `Object` is not reference counted, and it is not included in the total count of objects on an allocation block. If each PC `Object` on an allocation block is allocated in this way, this results in pure, region-based memory management, and is exceedingly lightweight.

Full reference counting. This is the default.

Unique ownership. The PC `Object` is not reference counted, but there can be one `Handle` object referencing the uniquely-owned object. When that `Handle` is destroyed, the object is deallocated.

E SPARK CONFIGURATION

The configuration of the Spark cluster such as number of executors, executor memory, number of cores for each executor, driver memory and so on were tuned for each experiment as shown in Table 8.

For TPC-H and LDA, of which total volume of data for input and processing exceeded available memory, we ran Spark in yarn client mode to avoid out-of-memory errors. For other experiments, we ran Spark in cluster mode to be consistent with PC.

In addition, input data for experiments using Dataset APIs were stored in Parquet format, and input data for experiments using RDD APIs were stored in Spark's object file format, and serialized using Kryo. Other Spark parameters such as parallelism, partition number, and so on were all carefully tuned for each experiment. More details are omitted due to space limitation.