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

ABSTRACT

This paper describes *PlinyCompute*, a system for development of high-performance, data-intensive, distributed computing tools and libraries. *In the large*, PC 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.

1 INTRODUCTION

Big Data systems such as Spark [58] and Flink [10, 23] 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 [35] 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` [45], linear algebra packages such as SystemML [17, 18, 31, 55] and Samsara [5], and graph analytics with GraphX [32] and GraphFrames [30]. 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 [16]. 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 softwares such as OpenMP [29] and MPI [35], which provide little direct support for managing very large datasets, 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 method/function dispatch to the JVM.

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 very high-level, declarative interface, relying on automatic, relational-database style optimization [24] 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 which join algorithms to run 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 [54]. A relatively naive library user cannot be expected to tune a library implementation of an algorithm to run well on his or her particular dataset, and yet with existing systems, this sort of tuning may require modification of the code itself, which is beyond the vast majority of end-users.

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

Under Review, 2017

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 in the small—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*, which is a toolkit 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 [26]. For example, to specify a join over five sets of objects, a PC programmer does not build a join 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. Logically, TCAP operates over sets of columns of PC `Objects`.

(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 and has been designed to work closely with the PC object model to minimize memory-related costs during computation.

(4) Various *distributed services*, which include a catalog manager serving system meta-data, a distributed storage manager.

Our contributions. We describe the design and implementation of PlinyCompute. Currently, PC consists of around 150,000 lines of C++ code, with a small amount of Prolog code, We benchmark several library-style softwares 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 performance of two analytics queries against TPCH data, and also several standard machine learning codes written on top of PC.

2 OVERVIEW OF PC

The PC software consists of (1) the PC object model, (2) the PC API and TCAP compiler (TCAP is a domain-specific language executed by PC), (3) the execution engine, and (4) various distributed services. In the next few sections of the paper, we discuss the first three software components in detail.

PC is a distributed system, and a PC runtime 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 a computation, the computation is compiled into TCAP on the user's process, then optimized by the master node's TCAP optimizer and executed by the master node's distributed query scheduler.

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.

3 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 [50, 52]. 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*. The PC object model 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, 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.

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 be recycled and written over with a new set of objects. In computer systems design, this is often referred to as *region-based allocation* [36, 56], and is often the fastest way to manage memory.

To illustrate 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;
};
```

To load such data into a PC runtime, we might write the following code:

```
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);
pcContext.createSet <DataPoint> ("Mydb", "Myset");
pcContext.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 ()`, 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*.

Object allocation and deallocation is handled by the PC object model. If the next line of code executed were: `myVec = nullptr;` then all of the memory associated with the `Vector` of `DataPoint` objects would be automatically freed, since PC `Object`s 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 [50, 52], 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.

This 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*.

Finally, we note that the PC object model is not used exclusively for application programming. The PC object model is used *internally*, integrated with PC's execution engine as well. For example, aggregation is implemented using PC's built-in `Map` class. Each thread maintains a `Map` object that the thread aggregates its local data to; those are merged into maps that are sent to various workers around the distributed PC runtime. All sends and receives of these `Map` objects happen without (de-)serialization.

4 PLINYCOMPUTE'S LAMBDA CALCULUS

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

The PC toolkit consists of a set of operations: `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 operations by writing code that composes together various C++ codes using a domain-specific lambda calculus. For example, to implement a `SelectionComp` over PC `Object`s 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 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, 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 [46], as well as a set of *higher-order functions* [25] 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 ()`, which returns a function calling a native C++ lambda;
(4) `makeLambdaFromSelf ()`, which returns the 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 `Object`s stored in the PC runtime. 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 terms describe 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. If the programmer chose to do this, PC would be unable to optimize the compute plan—the system relies on the willingness of the programmer to expose intent via the lambda term construction function.

5 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 computation, and compiles all of those lambda terms into a DAG of small, atomic computations—a TCAP program. A TCAP program is fully optimizable, using many standard techniques from relational query execution, as we will discuss in the Appendix of the paper. In this Section, we discuss how TCAP programs are executed by PC.

5.1 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 [7, 19, 37, 60], where a column of values are pushed through a simple computation in sequence, so as to play to the strength of a modern CPU, with few cache misses and no virtual function calls. The second is *code generation* [9, 20, 40, 48, 49], where a system analyzes the computation and then generates code—either C/C++ code, or byte code for a framework such as LLVM [41, 42].

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). The data manipulations directly specified by a SQL programmer are likely to be limited, consisting of comparisons between attributes, simple arithmetic, and logical operations. Applying classical vectorization to PC, which requires an execution plan to be constructed consisting entirely of calls to a toolkit of vector-based operations shipped with the system, is unrealistic when most/all computations are over user-defined types. Further, generating LLVM code for complex operations over user-defined types in a high-level language is akin to writing a full-fledged compiler.

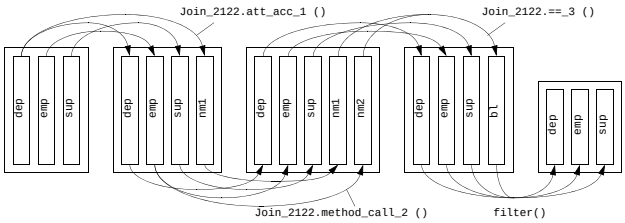

Figure 1: Execution of the first four stages of a pipeline constructed from the example TCAP program. The first two stages extract new vectors from existing vectors of objects, first via a call to `Join_2122.att_acc_1 ()`, which extracts `Dep.deptName` from each item in the vector `dep` of `Dep` objects, producing a new vector called `nm1`. Then, via a call to `Join_2122.method_call_2 ()`, which invokes `Dep :: getDeptName ()` on each of the `Emp` objects in the vector `emp`, producing 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.

PC uses a hybrid approach, where the PC execution engine is vectorized, but where the code for the individual vectorized operations (called *pipeline stages*) are fully compiled. In PC, a user does not specify the graph of pipeline stages directly. Rather, the programmer expresses her/his intent at a high-level, by building a graph of high-level operations. These operations are not associated with any specific physical implementation. For example, PC supplies an abstract *n*-way join called a `JoinComp`.

Operations are customized when a user supplies a set of *lambda term construction functions*. To customize an operation, a user writes member functions that return *lambda terms*, which are possibly complex terms in a lambda calculus that wrap C++ code. Those lambda terms express the programmer's intent, hence PC does not rely on analysis of C++ to build an optimized execution plan [11].

It is these lambda terms that are compiled into a DAG of pipeline stages over lists of PC `Objects` or simple types, which is then optimized (that is, operations are automatically re-ordered to form an optimal plan) using classical relational methods [24, 34, 38]. After optimization, the pipeline stages are fit together to produce a set of interconnected pipelines. Input data are broken into lists of data vectors (called, appropriately, *vector lists*), and fed into the various pipelines. Optimization of the DAG of pipeline stages is possible because the programmer expresses intent via the lambda calculus [14, 47]. Thus, PC's hybrid approach is vectorized, but it is *also* compiled—the opaque C++ user code is compiled into pipeline stages that are assmebled into an optimized plan.

5.2 TCAP and Vectorized Execution

As mentioned in Section 5, PC's vectorized execution engine repeatedly pushes so-called *vector lists*, which is a list of vectors, through a series of pipeline stages. Each pipeline stage takes as input a vector list, and produces a new vector list that consists of zero or more vectors from the input vector list, as well as one or more vectors appended at the end of the list. Pipeline stages are constructed in such a way that the overhead of a virtual function call can be amortized

on a vector list of objects, aside from any virtual function calls that may be present in the user's code. The TCAP language describes both the pipeline stages required to perform a PC computation, as well as the schema for each of the vector lists that will be produced during the PC computation, and how each of the pipeline stages adds or removes vectors from the vector lists that are pushed through the computation.

To see how this works through an example, consider a variant of the `getSelection ()`:

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

PC compiles 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_2212', '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_2212',
      '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_2212', '==_3',
      [('type', 'equalityCheck')]);

Flt_1(dep,emp,sup) <= FILTER(WBl_1(bl),
   WBl_1(dep,emp,sup), 'Join_2212', []);
```

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 the following. It describes a pipeline stage that takes as input a vector list called `Input`, 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 with pointer assignment) from `In`. In addition, the compiled code referred to by `Join_2212.att_acc_1` will be executed via a vectorized application to the input vector `dep`. The result will then be put into a new vector called `WDNm_1.name1`. The resulting vector list (consisting of the vectors shallow copied from the input as well as the new vector `WDNm_1.name1`) will be called `WDNm_1`.

Next, this TCAP program specifies that `WDNm_1` is processed by `APPLY`ing the method call `getDept Name()` on the attribute `emp`; this is done via application of the compiled code referred to by `Join _2212.method_call_2`. The vectors `dep`, `emp`, `sup` and `nm1` are simply shallow-copied to the output vector set.

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

Note that in each TCAP operation, the key-value map is only informational and does not affect its execution. However, this information can be vital during optimization.

5.3 Template Metaprogramming

In PC, each vectorized pipeline stage (such as `Join_2212.member_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 metaprogramming* capabilities [39]. Templates are the C++ language's way of providing generics 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 metaprogramming 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_2212', '==_3', '');
```

Here, the pipeline stage `Join_2212.==_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_2212.==_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 metaprogramming facilities, this pipeline stage is generated specifically for `LHSType` and `RHSType` and optimized by the compiler for use with those two types. As the `Join_2212.==_3` pipeline stage loops over the objects in the input vectors, there are no function calls that cannot themselves be inlined 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.

6 THE OBJECT MODEL IN DETAIL

6.1 Using PC Objects

Arguably, the choice of how individual data items are to be represented and manipulated in a data analytics or management system is one of the most controversial decisions that a system designer can make, both in terms of the programmability of the resulting system, and its performance. For decades, the dominant model used in data management was the flat relational model, which can achieve very good performance. Flatness generally means that there is typically 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.

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 [59]) offer far more flexibility, at the (possible) cost of significant performance degradation. PC attempts to combine this flexibility with excellent 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 system on top of the PC object model, where huge matrices are "chunked" into smaller sub-matrices. A sub-matrix may be stored via our current, 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 the chunk by its row and column.

But while the PC object model provides a rich, object-oriented programming model, it also provides the good performance characteristic of a flat relational model. The key principle underlying the PC object model is *zero-cost data movement*. That is, once a data object has been allocated and populated, moving the object to disk or across the network should be a simple matter of copying memory; there should be no CPU cost for serialization and deserialization.

At first glance, it would seem to be impossible to offer zero-cost data movement while allowing a programmer to create and manipulate such objects. Pointers and container classes generally lead to high memory (de)allocation costs and high object (de)serialization costs, resulting in high CPU cost. The PC object model avoids this by using a "page-as-a-heap" memory allocation model. The PC object model provides a call of the form:

```
makeObjectAllocationBlock (ptr, blockSize);
```

After such a call, *all subsequent PC Object allocations by the thread creating the object allocation block will be performed directly to the memory region starting at location* `ptr`. Typically, when it runs a computation, PC's execution engine will obtain a page from its memory pool to buffer output data, calling PC's `makeObjectAllocationBlock ()` function with a pointer to the page where output data are to be written. When an action taken by the execution engine or user-supplied code causes an out-of-memory execution, it means that the page is full. At that point, the execution engine can take appropriate computation-specific action, such as creating an object allocation block out of a new (empty) page, writing the full page out to disk, sending it across the network, etc. No serialization or deserialization or any sort of post-processing are needed, because all object allocations have taken place exclusively to the current allocation block.

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 runtime 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 runtime, such as a virtual deep copy function.

6.2 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.

6.3 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 runtime. 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. Objects of arbitrary type can be loaded into the distributed PC runtime and be processed using dynamically loaded native code, as long as the object type is registered first.

6.4 Allocation, Deallocation, and Assignment

There are three types of allocation blocks in PC, where an "allocation block" is a block of memory where PC `Objects` can be allocated.

Active allocation block. Each thread running in a PC process has exactly one *active* allocation block, that is currently receiving allocations (all calls to `makeObject` cause memory allocations to happen using that block). Such an allocation block is created via a call to `makeObjectAllocationBlock ()`. User code typically creates and manipulates objects in this block.

Inactive allocation block. Each thread also has one or more *inactive*, *managed* blocks. These are previously-active blocks of memory that contain one or more objects that are reachable from some `Handle` that is currently in RAM. When the number of reachable objects in such a block drops to zero, it is automatically deallocated. When a user (or the PC system software) calls `makeObjectAllocationBlock ()`, the newly created allocation block becomes the active block, and if the previously-active allocation block has any reachable objects on it, it becomes an inactive, managed block.

Unmanaged allocation block. Finally, there are zero or more *inactive*, *un-managed* blocks. These are blocks with reachable PC `Objects` that are *not* managed by the PC object model. These tend to be pages of objects that have been loaded into RAM from disk or across the network for processing during a distributed computation. Such blocks are paged in and out of the buffer pool in much the same way as a relational database would page data in and out. Rather than the PC object model being responsible for managing such blocks, PC's execution engine manages such blocks.

In PC, each managed allocation block (active or inactive) has an active object counter (the number of objects that are reachable from some `Handle` in RAM). Each object in each managed allocation block (active or inactive) is reference counted, or pre-pended with a count of the number of `Handle` objects that currently reference the object. Un-managed blocks (and objects inside of such blocks) are not reference-counted.

When the reference count on an object in a managed block goes to zero, it is automatically deallocated (at least, this is the default behavior; it is possible for a programmer to override this behavior for speed, if desired, as we describe in Section B.1). Once the number of reachable objects on an inactive, managed allocation block falls to zero, the block is automatically deallocated. 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 dangling `Handles`. Specifically: What happens 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 * 1204);
Handle <Vector <double>> data =
     makeObject <Vector <double>> ();
for (int i = 0; i < 1000; i++)
     data->push_back (i * 1.0);
makeObjectAllocatorBlock (1024 * 1204);
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.

6.5 The PC Object Model and Multiple Threads

PC does not need to lock reference counts during multi-threaded execution because a block can only be managed by one single thread. If a thread copies a `Handle` object referencing an object housed on another thread's managed block, the reference count will not be changed because from the copying thread's point-of-view, the allocation block is not managed. This can, in theory, result in a problematic case where one thread has a `Handle` to an object that has been deallocated on the other thread (since the reference

count on the home thread will not be updated to reflect the off-thread reference). But in practice, this is not problem. Parallel and distributed processing is transparent to PC application programmers, so most cross-thread references happen as the result of computations staged by the PC execution engine. The PC execution engine uses pages carefully so as to ensure that it is not possible for pages to be unpinned while references to them can still exist.

7 EXPERIMENTS

7.1 Overview

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

(1) How useful is PC for this task 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?

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 [27] into an object-oriented representation, and then benchmarked two reasonably complex analytical computations.

(3) We also implemented three iterative machine learning algorithms on top of PC: 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 that of eleven Amazon EC2 `m2.4xlarge` machines, running Ubuntu 16.04. Each machine had eight virtual cores, one 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 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. We do not clear the OS buffer cache, so HDFS data can be buffered/cached in the OS buffer cache.

7.2 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 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`.

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 [21, 53] (built from the ground up by a team consisting of MIT students and professional developers over the last nine years), Spark `mllib` [45] (the Big Data matrix implementation shipped with Spark), and SystemML [17, 18, 31] (a matrix and machine learning implementation developed over the last seven years by a team at IBM, built on top of Spark and Hadoop). The expert spent about six weeks in this effort.

Implementation. In `lilLinAlg`, a distributed matrix is stored as a set of PC `Objects`, where each object in the set is a `MatrixBlock`, as explained in Sec. 6.1, storing a contiguous rectangular sub-block of the matrix.

`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, column Sum, 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; these classes are chosen because distributed matrix multiplication is basically a join followed by an aggregation. Internally, the `LAMultiply Join` and `LAMultiplyAggregate` 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);
beta = (X '*␣X)^-1␣%*%␣(X␣'* y)
```

In the above DSL expression, `'*` represents a transpose-then-multiply computation, `^-1` represents an inverse computation, and `%*%` represents a multiply computation.

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 [43] 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 three different data dimensionalities:

	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	00:05∗	00:51	02:34	00:06∗	00:53	02:38	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

Table 1: Linear algebra benchmark. Format is MM:SS. A star (∗) indicates running in local mode.

ten, 10^2, and 10^3. This refers to the number of features in each data point. 10^6 data points were used.

In addition to `lilLinAlg`, for the Gram matrix and linear regression computations, SystemML V0.9 on Hadoop was used. For these two computations, Spark `mllib` along with Spark 1.6.1 was used. For nearest neighbor, SystemML V1.0 on Spark 2.1.0 was used, and for nearest neighbor, `mllib` along with Spark 2.1.0 was used. We use the same SciDB version—14.8—for all three experiments.

We spent considerable effort tuning all of the implementations. For `lilLinAlg`, this consisted mainly of efforts to choose the correct page size for holding the `MatrixBlock` objects. For the runs on the other three platforms, we also carefully tuned the systems for best performance. For example, we tuned Spark block size and repartition size for every experiment. In SystemML, we also carefully chose to use the parallel for loop, which boosted performance significantly. For fairness, for each of the distributed linear algebra tools, we do not count the time required to load data from client into the system (for example, for `lilLinAlg`, we do not count the time required to load data from text and into PC).

Results and Discussion. Experimental results are given in Table 1. This table shows that for every one of the higher-dimensional computations, the `lilLinAlg` implementation was the fastest. Often, it was considerably faster. Looking only at the nearest neighbor computation (where the latest version of Spark was used along with Spark's `mllib`) `lilLinAlg` was five times faster than `mllib` and thirteen times faster than SciDB.

For the ten-dimensional computations, there was some variability in the results. For two of the three 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. This demonstrates that for a small computation, the overhead of performing it in distributed fashion across multiple machines calls into question the viability of distribution in the first place.

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 `lilLinAlg`s performance on the larger matrices was SystemML. The newest SystemML version, on Spark, is only 50% slower than `lilLinAlg` for nearest neighbor search. 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 [17]. `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 significantly faster than it is now.

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. Thus, it is not unreasonable to suggest that as PC matures, it will be even faster.

7.3 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 datasets that store an instance of the TPC-H database [27]. But rather than storing the dataset 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... */};
```

`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 {
   Vector <Handle <Order>> orders;
   int custkey;
   String name;
   /* seven more members... */};
```

Implementation and Experiments. We then run two computations over the resulting set of `Customer` objects. The first computation is the *customers per supplier* computation where we compute, for each supplier, the complete list of `partkeys` that the supplier has sold to each of the supplier's customer's. For each supplier, the result is a an object that contains the supplier's name (as a `String`) and an

Number `Customer` objects	2.4M	4.8M	9.6M	14.4M	19.2M	24M
Kryo data size	41.5GB	83.1GB	167.2GB	251.1GB	333.2	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: PlinyCompute vs. Spark for large-scale OO computation. Times in MM:SS.

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 `partID`s 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 `partID`s that the supplier has sold to the customer. The second `Computation`, called `CustomerSupplierPartGroupBy`, 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 `partID`s.

On Spark we implemented an algorithmically equivalent carefully-tuned code. We used Spark version 2.1.0. Note that since the objects are all highly nested, it was not possible to develop a satisfactory Dataset or Dataframe implementation, and so our Spark implementation made use of Spark's RDD interface. All implementation was done in Java. 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 add a final count of the number of customers in each `Map` in each `SupplierInfo` object.

The second computation run over the denormalizd TPC-H schema is the *top-k closest customer part sets* computation. In this computation, for a given `Customer` object, we go through all of the associated `Order` objects and obtain the complete list of `partID`s for each order. All duplicate `partID`s 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 result of the computation is a list of *k* objects containing the Jaccard similarity, the integral `custkey`, and a `Vector <int>` that stores the complete list of unique `partID`s sold to that customer.

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

For both PC and for Spark, and for both computations, we created TPC-H dataset 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 $\frac{1024}{2.4\times10^6}$ times the size of the data.

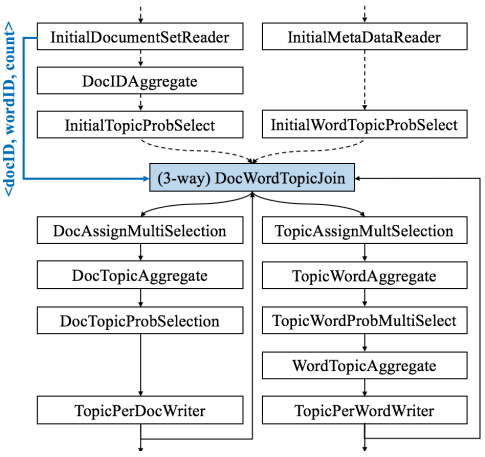

Figure 2: PC LDA's `Computation` objects and input-output dependencies. Computations connected by dash lines will only run once, during initialization. Computations connected by solid lines will run iteratively.

For both Spark computations, we performed two runs at each dataset size. For the first, we stored the data in HDFS, and measure 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 run only one version of the computation, where the various `Customer` objects are stored in PC's storage system.

Results and Discussion. Results are in Table 2. The difference in speed between the PC implementation and the Spark implementation 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. k was between 1,024 and 10,240, so (in theory) that should be a hard limit on the number of customer's whose data are moved off of each machine during a shuffle (since it is impossible for more than k customers processed on any machine to be in the top k). 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). Because PC `String`s 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`). This might explain why PC's speedup is less significant on that computation.

7.4 Machine Learning

Finally, we consider three common machine learning algorithms: LDA, GMM and k-means.

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 [22]. 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 in Figure 2. Each iteration requires a three-way `JoinComp`, three `MultiSelectionComp`s, and three `AggregateComp`s, 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 dataset 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.

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 dataset by concatenating random pairs of newsgroup postings end-on-end. There are more than 739 million `(docID, wordID, count)` triples in the dataset. 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 each test case, the same random data was used for comparing PC and Spark performance.

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.

Again, the same data is used on both PC and Spark platforms. Ten Gaussians are used for GMM, and ten clusters for k-means.

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`) 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. We will discuss the significance of this finding in the next subsection, where we discuss some of the issues surrounding Java vs. C++.

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

Table 3: PlinyCompute vs. Spark for LDA. Times in MM:SS, averaged over five iterations.

Dimensionality Number of points	100 10^7	300 10^6	500 10^6
PlinyCompute	00:30	00:38	1:42
Spark `mllib`	1:41	1:54	5:05

Table 4: PlinyCompute vs. Spark for GMM. Times in MM:SS, averaged over five iterations.

Dimensionality Number of points	100 10^9	300 10^8	500 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

Table 5: PlinyCompute vs. Spark for *k*-means. Times in MM:SS, averaged over five iterations.

Applications	PlinyCompute	Spark
`lilLinAlg`	3505	3130 (Scala)
`Customer`s per `Supplier`	929	953 (Java)
top-*k* Jaccard	793	966 (Java)
LDA	1038	343 (Scala/`breeze`)
GMM	932	474 (Scala/`breeze`)
k-means	695	670 (Scala)

Table 6: Lines of source code (LOC) comparison.

As illustrated in Table 5, for *k*-means, PC achieved a 2× to 4× speedup compared with the Spark `mllib` RDD implementation.

Experiments: Final Thoughts. The central hypothesis of this work was: "declarative in the large, high-performance in the small" can result in an excellent platform for tool and library development. We believe these experiments have shown that. The most convincing benchmark was likely the first one, where 1.5 person-months of engineering time resulted in the `lilLinAlg` tool that was faster than other competing systems with many years of development time. One of those systems (SciDB) was implemented in C++, while the others (Spark's `mllib` as well as `SystemML`) used Hadoop and Spark. Other benchmarks showed similar advantages to PC.

We close the benchmarks with two final questions. First: PC may be faster, but is it significantly more difficult to develop for than a platform that uses a managed runtime? PC certainly gives a programmer more flexibility, and with that can come certain costs—un-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 6 shows the SLOC counts for the various implementations described here, comparing to their Spark counterparts. If one believes that engineering effort is roughly proportional to SLOC written, there is not a significant difference between the two systems. 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`) than does GSL, which was used in those implementations.

Matrix Dimensions	1000×1000	10000×10000
GSL	1033 ms	26:18
Eigen	123 ms	3:57
`breeze`-native	179 ms	3:40

Table 7: Matrix multiplication benchmarked in `m2.4xlarge` instance by setting thread number to one for all packages. (for 1000×1000 matrix multiplication, processing times are recorded in milliseconds, and for 10000×10000 matrix multiplication, processing times are recorded in hh:mm:ss format.)

Our last question is: PC may be faster, but how much of that is related to "declarative in the large, high performance in the small?" Isn't C++ simply faster than Java, and might that explain a lot of the advantage realized by PC? We begin our answer to this question by pointing out that SciDB is written in C++, and not Java, so the C++ vs. Java question is not relevant to all of our findings. But even when the question is relevant, we assert that the most significant difference between Java on a modern JVM and C++ is that the latter gives the system developer more control over issues such as memory management, which the developer may use to produce a faster system (this is precisely what we have attempted to do with our development of the PC object model, for example). There is nothing inherent in C++ that makes it faster than Java if this extra flexibility is not used properly, especially in the age of JIT compilation and generational garbage collectors.

In fact, there is some good evidence that Spark and Java may have had some significant built-in *advantages* vs. our C++ implementations. Out of curiosity, we ran a simple micro-benchmark on an AWS `m2.4xlarge` machine, where we compared the various packages for statistical/scientific computing used throughout these experiments. In this benchmark, we run a single-thread 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). The results are shown in Table 7. Here we find that Java Breeze has slightly better performance than Eigen and *much* better performance than GSL. Thus, in one way, our C++ implementations were at a significant disadvantage.

Achieving excellent performance on complex, distributed computations is never a simple matter of "use C++, not Java". Many factors go into having a superior implementation, and those tend to even things out, as some of those factors go *against* PC. We argue that the reason that PC was consistently faster than its competitors are the design principles underlying the system—which was enabled by the choice of language.

REFERENCES

[1] [n. d.]. DL4J. https://deeplearning4j.org/. ([n. d.]). Accessed Sept 27, 2017.
[2] [n. d.]. Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page. ([n. d.]). Accessed Oct 22, 2016.
[3] [n. d.]. GSL - GNU Scientific Library. https://www.gnu.org/software/gsl/. ([n. d.]). Accessed Oct 22, 2016.
[4] [n. d.]. Mahout. http://mahout.apache.org. ([n. d.]). Accessed Oct 22, 2016.
[5] [n. d.]. Mahout Samsara. https://mahout.apache.org/users/environment/out-of-core-reference.html. ([n. d.]). Accessed Oct 22, 2016.

[6] [n. d.]. Project Tungsten: Bringing Spark Closer to Bare Metal. ([n. d.]). https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[7] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1664–1665.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[9] 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.

[10] Alexander Alexandrov et al. 2014. The Stratosphere platform for big data analytics. *VLDBJ* 23, 6 (2014), 939–964.

[11] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 47–61.

[12] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, 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] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 331–343.

[14] Hendrik Pieter Barendregt et al. 1984. *The lambda calculus*. Vol. 2. North-Holland Amsterdam.

[15] MKABV Bittorf, Taras Bobrovytsky, Casey Ching Alan Choi Justin Erickson, Martin Grund Daniel Hecht, Matthew Jacobs Ishaan Joshi Lenni Kuff, Dileep Kumar Alex Leblang, Nong Li Ippokratis Pandis Henry Robinson, David Rorke Silvius Rus, John Russell Dimitris Tsirogiannis Skye Wanderman, and Milne Michael Yoder. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*.

[16] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.

[17] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. SystemML: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[18] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid parallelization strategies for large-scale machine learning in SystemML. *Proceedings of the VLDB Endowment* 7, 7 (2014), 553–564.

[19] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[20] Sebastian Breß, Bastian Köcher, Henning Funke, Tilmann Rabl, and Volker Markl. 2017. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *arXiv preprint arXiv:1709.00700* (2017).

[21] 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.

[22] Zhuhua Cai, Zekai J Gao, Shangyu Luo, Luis L Perez, Zografoula Vagena, and Christopher Jermaine. 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.

[23] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 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).

[24] 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.

[25] 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.

[26] 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.

[27] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.

[28] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and Stanley B Zdonik. 2015. Tupleware:" Big" Data, Big Analytics, Small Clusters.. In *CIDR*.

[29] 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.

[30] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 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.

[31] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*. 231–242.

[32] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 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] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[35] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.

[36] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. *ACM Sigplan Notices* 37, 5 (2002), 282–293.

[37] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, 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.

[38] Matthias Jarke and Jurgen Koch. 1984. Query optimization in database systems. *ACM Computing surveys (CsUR)* 16, 2 (1984), 111–152.

[39] Nicolai M Josuttis. 2012. *The C++ standard library: a tutorial and reference*. Addison-Wesley.

[40] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.

[41] 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.

[42] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[43] Guy Lebanon. 2006. Metric learning for text documents. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 4 (2006), 497–508.

[44] Lu Lu and et al. 2016. Lifetime-Based Memory Management for Distributed Data Processing Systems. *VLDB* 9, 12 (2016), 936–947.

[45] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[46] 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.

[47] Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*. IEEE, 14–23.

[48] 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.

[49] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[50] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. 2015. Making Sense of Performance in Data Analytics Frameworks.. In *NSDI*, Vol. 15. 293–307.

[51] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[52] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2110–2121.

[53] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The architecture of SciDB. In *Scientific and Statistical Database Management*. Springer, 1–16.

[54] Michael Stonebraker, Lawrence A Rowe, Bruce G Lindsay, Jim Gray, Michael J Carey, Michael L Brodie, Philip A Bernstein, and David Beech. 1990. Third-generation database system manifesto. *ACM sIGMOD record* 19, 3 (1990), 31–44.
[55] 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.
[56] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.
[57] 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.
[58] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud*. 1–10.
[59] Matei Zaharia and et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2–15.
[60] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.

A RELATED WORK

Code Generation. DryadLinq [57] 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. LegoBase [40] switches the interface from declarative SQL to a high-level language (Scala) and uses a query engine written in Scala as a code generator to emit specialized and low-level C code for execution. TupleWare [28] 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 [51] 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. Since version 2.0, Spark [59] also exploits whole-stage code generation to generate JVM code. The goal is mainly to reduce type parsing and virtual function call overhead. PC uses a form of code generation (template metaprogramming) but the emphasis is quite different, in the sense that the goal is to allow for efficient distributed programming with complex objects.

Optimized Memory Management. Apache SparkSQL [12] serializes relational table 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 [44] 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 [10] 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 the Dataset/Dataframe to encode JVM objects into relational binary data 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 through Tungsten [6].

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 [15] is a C++-based SQL query engine that relies on Hadoop for scalability and flexibility in interface and schema. Impala compiles SQL into LLVM assembler. 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).

Google Spanner [13] is a distributed database system with a SQL query processor. It implements a dialect of SQL, which uses arrays and structures to support nested data as a first class citizen. To integrate with user applications, the relational data described needs to be translated to protocol buffers or user languages. PC takes a fundamentally different approach, as all code is object-oriented rather an a mix of SQL and other high (or medium) level languages.

Tensorflow [8] 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.

B *k*-MEANS EXAMPLE

Now that we have described PlinyCompute at a high level, we turn to an example of what PlinyCompute looks like to a programmer.

Imagine that a user wished to use PC to build a high-performance library implementation of a *k*-means algorithm. Once the programmer had defined the basic type over which the clustering is to be performed (such as the `DataPoint` class), a programmer would likely next define a simple class that allows the averaging of vectors:

```
class Avg : public Object {
        long cnt = 1;
        Handle <Vector <double>> data = nullptr;
        Avg &operator + (Avg &addMe)
           {/* add addMe into this */}
};
```

The programmer might next add a method to the `DataPoint` class that converts the `DataPoint` object to an `Avg` object:

```
Avg DataPoint :: fromMe () {
        Avg returnVal;
        returnVal.data = data;
        return returnVal;
}
```

And also add a method to the `DataPoint` class that accepts a set of centroids, computes the Euclidean distance to each, and returns the closest:

```
long DataPoint :: getClose (Vector <Vector <double>>
        &centroids) {...}
```

Next, a programmer using PC would define an `AggregateComp` class using PC's lambda calculus, since, after all, the *k*-means algorithm is essentially an aggregation:

```
class GetNewCentroids : public AggregateComp
    <Centroid, long, Avg, DataPoint> {

public:
   Vector <Vector <double>> centroids;

   Lambda <long> getKeyProjection (
       Handle <DataPoint> aggMe) override {
          return makeLambda (aggMe,
              [&] (Handle <DataPoint> &aggMe)
                 {return aggMe->
                     getClose (centroids);});
   }
   Lambda <Avg> getValueProjection (
       Handle <DataPoint> aggMe) override {
          return makeLambdaFromMethod
              (aggMe, fromMe);
   }
};
```

The declaration `AggregateComp <Centroid, long, Avg, DataPoint>` means that this computation aggregates `DataPoint` objects. For each data point, it will extract a key of type `long`, a value of type `Avg`, which will be aggregated into objects of type `Centroid`. To process each data point, the aggregation will use the lambdas constructed by `getKeyProjection` and `getValueProjection`. In this case, for example, `getKeyProjection` builds a lambda which simply invokes the native C++ lambda given in the code—this native C++ lambda returns the identity of the centroid closest to the data point.

To build a computation using this aggregation class, a programmer would need to specify the `Centroid` class (the result of this aggregation):

```
class Centroid : public Object {
        long centroidId;
        Avg data;
public:
        long &getKey () {return centroidId;}
        Avg &getVal () {return data;}
};
```

And then build up a computation using these pieces:

```
Handle <Computation> myReader =
    makeObject <ObjectReader <DataPoint>>
     ("myDB", "mySet");
Handle <Computation> myAgg = makeObject
    <GetNewCentroids> ();
myAgg->centroids = ... // initialize the model
myAgg->setInput (myReader);
Handle <Computation> myWriter =  makeObject <Writer
     <Centroid>> ("myDB", "myOutSet");
    myWriter->setInput (myAgg);
pcContext.executeComputations (myWriter);
```

After execution, the set of updated centroids would be stored in `myDB.mySet`. Performing this computation in a loop, where the centroids are repeatedly updated until convergence, completes the implementation.

C OPTIMIZING TCAP

Optimizability is one of the drivers for the decision to compile all computations expressed in PC's lambda calculus into TCAP. TCAP resembles relational algebra, and it is similarly amenable to rule- and cost-based optimization using a combination of methods from relational query optimization and classical compiler construction.

PC's optimizer is currently implemented in Prolog; a series of transformations are fired iteratively to improve the plan until the plan cannot be improved further. For an example of the sort of optimization present in PC, consider the task of removing redundant method calls. 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) < 10000;
}
```

PC would compile this into the following 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 `In::emp` to produce a new vector list `JK2_2`, storing the result of the method call in `JK2_1.mt1`. After comparing `JK2_2.bl1` 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. If `getSalary` simply accesses a data member, the additional call is costless. But in the general case a method call may run an arbitrary computation. Hence, the second call should automatically be removed as being redundant (by definition, all method calls evaluated during computation should be purely functional, and so they must return the same value when called a second time). 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.

In our example, the optimized TCAP program is:

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

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

JK2_4(emp,bl1,bl2) <= APPLY(JK2_3(mt1),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',[]);
```

Currently, the optimizations implemented in PC are rule-based (such as pushing down selections). We plan to work on cost-based optimization in the future, which is a challenging research problem because of lack of statistics over the arbitrary PC `Object`s.

C.1 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 `Object`s is not-reused. Hence, it is very similar to classical, region-based allocation—though PC `Object`s 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 `Object`s 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.

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.

D SPARK CONFIGURATION

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

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

Platform	num executors	executor mem	executor cores	driver mem
`lilLinAlg`	10	60GB	8	50GB
TPC-H	10	50GB	7	50GB
LDA	20	30.5GB*	4	55GB
GMM	80	70GB	1	55GB
k-means	10	60GB	8	50GB

Table 8: Workload-specific Spark Configurations for Different Experiments.