Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions

Jiaxing Zhang[†] Hucheng Zhou[†] Rishan Chen[†‡] Xuepeng Fan[†&] Zhenyu Guo[†]
Haoxiang Lin[†] Jack Y. Li [†§] Wei Lin[*] Jingren Zhou[*] Lidong Zhou[†]
[†]*Microsoft Research Asia* [*]*Microsoft Bing* [‡]*Peking University*
[&]*Huazhong University of Science and Technology*[*] [§]*Georgia Institute of Technology*

ABSTRACT

Map/Reduce style data-parallel computation is characterized by the extensive use of user-defined functions for data processing and relies on data-shuffling stages to prepare data partitions for parallel computation. Instead of treating user-defined functions as "black boxes", we propose to analyze those functions to turn them into "gray boxes" that expose opportunities to optimize data shuffling. We identify useful functional properties for user-defined functions, and propose SUDO, an optimization framework that reasons about data-partition properties, functional properties, and data shuffling. We have assessed this optimization opportunity on over 10,000 data-parallel programs used in production SCOPE clusters, and designed a framework that is incorporated it into the production system. Experiments with real SCOPE programs on real production data have shown that this optimization can save up to 47% in terms of disk and network I/O for shuffling, and up to 48% in terms of cross-pod network traffic.

1 INTRODUCTION

Map/Reduce style data-parallel computation [15, 3, 23] is increasingly popular. A data-parallel computation job typically involves multiple parallel-computation phases that are defined by *user-defined functions* (or *UDF*s). The key to data-parallel computation is the ability to create *data partitions* with appropriate properties to facilitate independent parallel computation on separated machines in each phase. For example, before a reducer UDF can be applied in a reduce phase, data partitions must be *clustered* with respect to a reduce key so that all data entries with the same reduce key are mapped to and are contiguous in the same partition.

To achieve desirable data-partition properties, *data-shuffling* stages are often introduced to prepare data for parallel processing in future phases. A data-shuffling stage simply re-organizes and re-distributes data into appropriate data partitions. For example [45], before applying a reducer UDF, a data shuffling stage might need to perform a *local sort* on each partition, *re-partition* the data on each source machine for re-distribution to destination machines, and do a multi-way *merge* on re-distributed sorted data streams from source machines, all based on the reduce key. Data shuffling tends to incur expensive network and disk I/O because it involves all data [47, 26]. Our analysis of a one-month trace collected from one of our production systems running SCOPE [10] jobs shows that with tens of thousands of machines data shuffling accounts for 58.6% of the cross-pod traffic and amounts to over 200 petabytes in total. Data shuffling also accounts for 4.56% intra-pod traffic.

In this paper, we argue that reasoning about data-partition properties across phases opens up opportunities to reduce expensive data-shuffling. For example, if we know that data-partitions from previous computation phases already have desirable properties for the next phase, we are able to avoid unnecessary data-shuffling steps. The main obstacle to reasoning about data-partition properties across computation phases is the use of UDFs [24, 26]. When a UDF is considered a "black box", which is usually the case, we must assume conservatively that all data-partition properties are lost after applying the UDF. One of the key observations of this paper is that those "black boxes" can be turned into "gray boxes" by defining and analyzing a set of useful *functional properties* for UDFs in order to reason about data-partition properties across phases. For example, if a particular key is known to pass-through a UDF without any modification, the data-partition properties of that key will be preserved. Furthermore, we show how we can re-define the partitioning function in a data-shuffling phase to help preserve data-partition properties when UDFs are applied.

We make the following contributions in this paper. First, we define how a set of data-partition properties are related to data-shuffling in a simple and general UDF-centric data-parallel computation model. Second, we define how a set of functional properties for UDFs change the data-partition properties when the UDFs are applied. Third, we design a program analysis framework to identify functional properties for UDFs, and develop an optimization framework named SUDO to reason about data-partition properties, functional properties, and data shuffling. We further integrate SUDO into the production SCOPE optimization framework. Finally, we study the re-

[*]Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

al workload on the SCOPE production system to assess the potentials for this optimization and provided careful evaluations of a set of example showcase applications. The study uses 10,000 SCOPE programs collected from production clusters; SUDO is able to optimize 17.5% of the 2,278 eligible programs which involve more than one data-shuffling stages. Experiments with real SCOPE programs on real production data have shown savings of up to 47% in terms of disk and network I/O for shuffling, and up to 48% in terms of cross-pod network traffic.

The rest of the paper is organized as follows. Section 2 introduces the system model that SUDO operates on, defines data-partition properties, and shows how they relate to data-shuffling. Section 3 defines functional properties of UDFs, describes how they affect data-partition properties when UDFs are applied, and presents the SUDO optimization framework that reasons about data-partition properties. Section 4 presents further optimization opportunities that expand on the framework in the previous section, by considering re-defining partition functions. Implementation details are the subject of Section 5, followed by our evaluations in the context of the SCOPE production system in Section 6. Section 7 discusses the related work, and we conclude in Section 8.

2 SYSTEM MODEL

A typical data-parallel *job* performs one or more transformations on large datasets, which consist of a list of *records*; each with a list of *columns*. A transformation uses a *key* that consists of one or more columns. For parallel computation, a dataset is divided into *data partitions* that can be operated on independently in parallel by separated machines. A data-parallel job involves multiple parallel-computation *phases* whose computations are defined by user-defined functions (UDFs). Following Map/Reduce/Merge [12], our model contains three types of UDFs: *mappers*, *reducers*, and *mergers*. By choosing low-level "assembly language"-like computation model, our techniques can be applied broadly: programs written in any of the high-level data-parallel languages, such as SCOPE [10], HIVE [40], PigLatin [33], and DryadLINQ [46], can be compiled into jobs in our model.

cross-partition \ within-partition	none	contiguous	sorted
none	AdHoc	-	LSorted
partitioned	Disjoint	Clustered	PSorted
ranged	-	-	GSorted

Table 1: Data-partition properties defined in SUDO.

Certain *data-partition properties*, which are defined with respect to keys, are necessary before a computation phase can be applied. For example, a reducer or a merger requires that all the records with the same key are contiguous in the same partition. In general, data-partition properties specify behaviors within a partition and across partitions, as shown in Table 1. Within a partition, records are *contiguous* if all same-key records are stored together, or *sorted* if they are arranged by their keys. Across partitions, records are *partitioned* if same-key records are mapped to the same partition, or *ranged* if they are stored on partitions according to non-overlapping key ranges. Among the nine combinations (cells in Table 1), we focus specifically on the following six: AdHoc, LSorted, Disjoint, Clustered, PSorted, and GSorted. For example, GSorted means that the records are *sorted* within a partition and *ranged* across partitions. We do not include the rest because they are not important in practice based on our experiences; incorporating them into SUDO is straightforward.

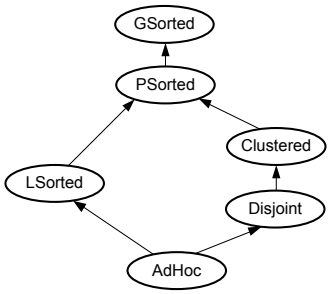

Figure 1: Data-partition property lattice in SUDO.

A data-partition property is *stronger* than another that it implies; for example, GSorted implies PSorted, which in turn implies Clustered. Such relationships are captured in the lattice shown in Figure 1, where a data-partition property is stronger than its lower data-partition properties. With this lattice, we can define *max* of a set of data-partition properties as the weakest one that implies all properties in that set. For example, *max* of Clustered and LSorted is PSorted. We define *min* analogously to max.

Data-shuffling stages are introduced to achieve appropriate data-partition properties by re-arranging data records without modifying them. A typical data shuffling stage consists of three steps: a *local-sort* step that sorts records in a partition with respect to a key, a *re-partition* step that re-distributes records to partitions via hash or range partitioning, and a multi-way *merge* step that clusters re-distributed records based on the key. Combinations of those steps are used to achieve certain properties; some requires taking all three steps, while others do not, depending on the data-partition properties before and after data shuffling. Figure 2 illustrates the relationship between data-partition properties and data-shuffling

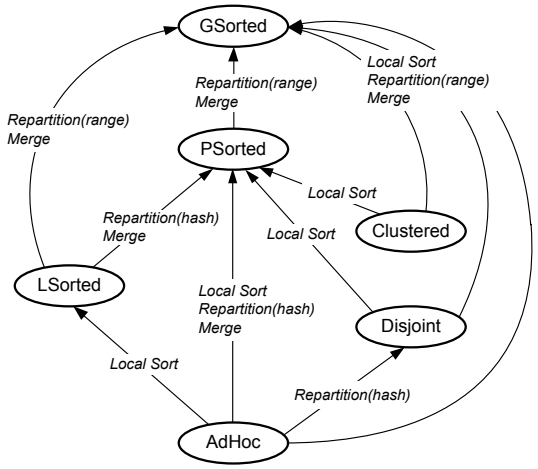

Figure 2: Transformation of data-partition properties using data shuffling.

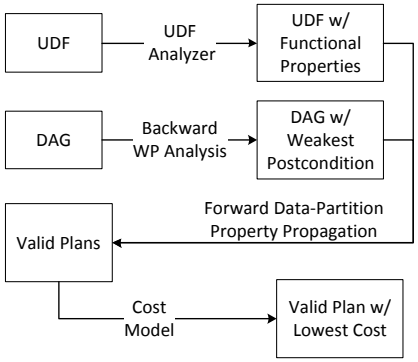

Figure 3: SUDO Optimization with Functional Properties.

steps. Note that because Clustered cannot be generated precisely through data-shuffling steps as the current implementation for the *merge* step uses merge-sort, SUDO always generates PSorted instead to satisfy Clustered.

In the rest of this paper, a data-parallel job is represented as a directed acyclic graph (DAG) with three types of vertices: *data vertices* that correspond to input/output data, each with an associated data-partition property; *compute vertices* that correspond to computation phasess, each with a type (mapper, reducer, or merger) and a UDF; and *shuffle vertices* that correspond to data-shuffling stages, each indicating the steps in that stage. The re-partitioning stages in shuffle vertices also specify whether hash or range partitioning is used. This DAG can be created manually or generated automatically by a compiler from a program in a high-level language, and allows us to flexibly define SUDO optimization scope. For example, we can use the same framework to analyze a pipeline of jobs or a segment of a job.

SUDO focuses on optimizing data shuffling by finding a *valid execution plan* with the lowest cost for a job J. The plan satisfies the following conditions: (i) the execution plan differs from J only at data-shuffling stages; (ii) for each computation phase, the input must have the necessary data-partition properties, i.e., data partitions must be Clustered for a reducer and PSorted for a merger; (iii) for a merger, all its input vertices must have the same data-partitioning (i.e., PSorted or GSorted on the same merge key); and finally, the execution plan preserves all data-partition properties of any output vertex.

Two SUDO optimizations are enabled by reasoning about how data-partition properties propagate through computation phases with UDFs. The first identifies unnecessary data-shuffling steps by using *functional properties* of UDFs to reason about data-partition properties, while the second further re-defines the partition function in a re-partitioning step of data shuffling to propagate certain data-partition properties for optimizations. We describe the first optimization in Section 3 and the second in Section 4.

3 FUNCTIONAL PROPERTIES

Data-shuffling stages in our model tend to be expensive as they often involve heavy disk and network I/O. They are added to satisfy data-partition properties for subsequent computation phases and to satisfy user requirements on output data. Although a preceding data-shuffling stage can result in certain data-partition properties, a computation phase with a UDF is not guaranteed to preserve those properties because traditionally, UDFs are considered "black boxes".

Our main observation for the first SUDO optimization is that, by defining appropriate functional properties, UDFs can be turned into "gray boxes" that expose how data-partition properties propagate across phases to facilitate the elimination of unnecessary data-shuffling steps. Figure 3 illustrates the overall flow of the SUDO optimization with functional properties. Given a job, we first extract and analyze all its UDFs to determine their functional properties. At the same time, we do a *backward WP analysis* to compute the *W*eakest *P*re-condition before each computation phase and the *W*eakest *P*ost-condition after each data-shuffling stage that maintains correctness as defined in Section 2. We then do a *forward data-partition property propagation* to generate valid execution plans with optimized data-shuffling stages, and then select the plan with the lowest cost according to a

cost model. The rest of this section elaborates on this optimization flow.

3.1 Defining Functional Properties

A functional property describes how an output column that is computed by a UDF depends on the UDF's input columns. Because SUDO focuses on optimizing the data-shuffling stages that are added to satisfy data-partition properties, we are particularly interested in functional properties that preserve or transform data-partition properties. Ideally, those functional properties should be simple enough to be identified easily through automatic program analysis. Here we limit our attention to deterministic functions that compute a single output column from a single input column in one single record. A UDF might exhibit one functional property on one output column and another on another column; we focus on columns that are used as a reduce key, merge key, or re-partition key, as well as those used to compute those keys. A list of interesting functional properties for SUDO follows.

A *pass-through* function f is an identity function where the output column is the same as the corresponding input column. By definition, a reducer/merger is a pass-through function for the reduce/merge key. A pass-through function preserves all data-partition properties.

Function f is *strictly-monotonic* if and only if, for any inputs x_1 and x_2, $x_1 < x_2$ always implies $f(x_1) < f(x_2)$ (*strictly-increasing*) or always implies $f(x_1) > f(x_2)$ (*strictly-decreasing*). Examples of strictly-monotonic functions include normalizing a score (e.g., $\texttt{score}' = \lg(\texttt{score})/\alpha$), converting time formats (e.g., `DateTime.ToFileTime()`), adding common prefix or postfix to a string (e.g., supplementing "http://" and "/index.html" to the head and tail of a site), and any linear transformation (e.g., $y = a \cdot x + b$ where $a \neq 0$). A strictly monotonic function also preserves all data-partition properties, although the output column might be in a reverse sort-order.

Function f is *monotonic* if and only if, for any inputs x_1 and x_2, $x_1 < x_2$ implies $f(x_1) \leq f(x_2)$ (*increasing*) or $f(x_1) \geq f(x_2)$ (*decreasing*). Examples of monotonic functions include time-unit conversion (e.g., $\texttt{minute} = \lfloor \texttt{second}/60 \rfloor$) and substring from the beginning (e.g., "abcd" $\rightarrow$ "ab" and "ac123" $\rightarrow$ "ac"). Monotonic functions preserves sort-order within a partition, but are not guaranteed to preserve partitioned or ranged properties across partitions because two different input keys can be mapped to the same output key.

Function f is *one-to-one* if and only if, for any inputs x_1 and x_2, $x_1 \neq x_2$ implies $f(x_1) \neq f(x_2)$. Examples of one-to-one UDFs include reversing urls (e.g., "www.acm.org" $\rightarrow$ "org.acm.www") and MD5 calculation (assuming no conflicts). One-to-one functions do not preserve sort-order, but do preserve contiguity within a partition and the partitioned property across partitions. As a result, it preserves data-partition properties such as Disjoint and Clustered but downgrades GSorted and PSorted to Clustered.

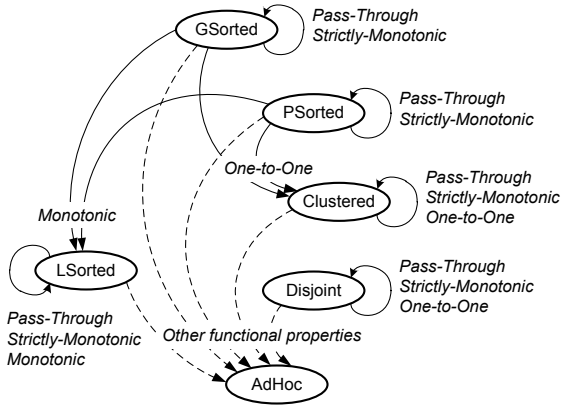

Figure 4: Data-partition property propagation through UDFs with various functional properties.

Figure 4 shows how data-partition properties propagate through UDFs with various functional properties, which are not chosen randomly. In fact, monotonic is sufficient and necessary for preserving LSorted; one-to-one is sufficient and necessary for preserving Clustered; and strictly-monotonic is sufficient and necessary for p-reserving GSorted. A proof is given in the Appendix.

3.2 Identifying Functional Properties

SUDO allows users to annotate UDFs with appropriate functional properties. It further uses program-analysis techniques to infer properties automatically whenever possible, in a bottom-up approach inspired by *bddbddb* [1]. Because functional properties focus on the dependency relationship between an output column and its relevant input columns, SUDO applies program slicing [42] to extract a UDF's core function for inferring its functional property with respect to each output column of interest. The analysis then starts from *facts* about the input columns, and applies *deduction rules* to the low-level instructions as well as third-party library calls to infer functional properties recursively until a fixed point is reached. The process returns the final functional properties associated with the UDFs upon termination.

Figure 5 shows examples of facts and rules. A fact represents the property of a function between a variable in a UDF and an input column that the variable is computed from. For example, for a variable y and an input column t, such that $y = f(t)$ for some function f, $\texttt{One2One}(y,t)$ states that f is a one-to-one function. Figure 5 (line 1) defines the basic fact that every input col-

```
1  PassThrough(t,t)
2
3  _(y,t) :- ASSIGN y x, _(x,t)
4  StInc(z,t) :- ADD z x y, StInc(x,t), Inc(y,t)
5  StInc(z,t) :- ADD z x y, Inc(x,t), StInc(y,t)
6
7  One2One(y,t) :- MD5 y x, One2One(x,t)
8
9  StInc(x,t) :- PassThrough(x,t)
10 One2One(x,t),Inc(x,t) :- StInc(x,t)
11 One2One(x,t),Dec(x,t) :- StDec(x,t)
12 Func(x,t) :- _(x,t)
13 Inc(x,_),Dec(x,_) :- Constant(x)
```

Figure 5: Examples of facts and deduction rules in a datalog format. `StInc`, `StDec`, `Inc`, and `Dec` stand for strictly-increasing, strictly-decreasing, increasing, and decreasing, respectively.

umn is considered a `PassThrough` function over itself.

Deduction rules infer the functional property of an instruction's output operand from the functional properties of its input operands. SUDO introduces a set of deduction rules; examples are shown in Figure 5 (lines 3-5). The first rule (line 3) states that, for `ASSIGN` instructions, the functional property of the input operand x can simply be propagated to the output operand y (the _ symbol means any functional property). The second and third rules state that, for `ADD` instructions, the output operand is strictly-increasing as long as one of the input operands is strictly-increasing, while the other is increasing.

Besides instructions, UDFs also call into functions in third-party libraries. SUDO either applies the deduction rules directly to the instructions in the library calls, or treats these function calls simply as "instructions" and provide deduction rules manually. SUDO has an accumulated knowledge base with manually provided deduction rules for commonly used library calls. Those manual annotations are particularly useful for cases where the automatic inference runs into its limitation. For example, the functional property for `MD5` (line 7) cannot be inferred automatically.

We also have rules that encode the relations among functional properties, where one function property might imply another. Examples of such rules are shown in Figure 5 (lines 9-13). We use `Func` as the weakest functional property that satisfies no specific constraints and also introduce `Constant` as a pseudo functional property for constant values, which is both *increasing* (`Inc`) and *decreasing* (`Dec`).

3.3 Backward WP Analysis

Based on the definition of a valid execution plan, we must figure out the requirement on each data-shuffling stage for validity. This is done through the following backward WP analysis in a reverse topological order of a given job. When visiting a new vertex, the backward WP analysis computes weakest precondition and postcondition as shown in Figure 6; the weakest postcondition associated with a data-shuffling stage is the data-partition property required on the result of that stage.

```
procedure OnVisitVertex(v)
  v.WPrecondition ← AdHoc
  if v.Type = UDF then
    if v.UDF = Reducer then
      v.WPrecondition ← Clustered
    else if v.UDF = Merger then
      v.WPrecondition ← PSorted
    end if
  else if v.IsOutputVertex then
    v.WPrecondition ← v.GetOutputDataProperty()
  else
    v.WPostcondition ←
      v.OutVertices.Select(ov => ov.WPrecondition).Max()
  end if
end procedure
```

Figure 6: Algorithm for backward WP analysis.

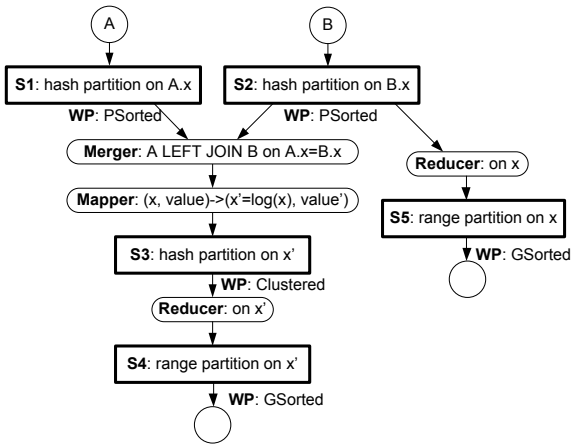

Figure 7: A sample data-parallel job.

Figure 7 depicts a sample compiled DAG from a SCOPE job: circles correspond to data vertices; rectangles correspond to data-shuffling stages, and rounded rectangles correspond to computation phases with UDFs. SUDO firstly analyzes the UDFs in the job, and annotates them with functional properties. In this case, both the merger and the reducers are pass-through functions, and the mapper is a strictly-monotonic function (using *log*). Then it runs backward WP analysis to get the weakest postcondition for all data shuffling stages: this results in GSorted for S4 and S5, Clustered for S3, PSorted for S1, and *max*(PSorted, Clustered) = PSorted for S2.

3.4 Forward Property Propagation

Once SUDO completes the backward WP analysis, it tries to find all valid execution plans according-

```
procedure ForwardExplorer (currGraph, traverseSuffix)
  while traverseSuffix.IsNotEmpty() do
    v ← traverseSuffix.RemoveFirst()
    if v.InVertices.Count = 0 then
      inputDP ← v.Input.GetDataProperty()
    else if v.InVertices.Count = 1 then
      inputDP ← v.InVertices[0].CurrentPostDP
    else
      if !v.ValidateMerge() then
        return
      end if
      inputDP ← v.InVertices[0].CurrentPostDP
    end if
    if v.Type = UDF then
      v.CurrentPostDP ←
       DPPropertyTransition(inputDP, funcProperty)
    else
      for prop in {p|p >= v.WPostcondition} do
        v.CurrentPostDP ← prop
        v.SSteps ← GetShufflingSteps(inputDP, prop)
        ForwardExplorer(currGraph, traverseSuffix)
      end for
    end if
  end while
  Plans.Add(currGraph)
end procedure
```

Figure 8: Algorithm for enumerating all valid plans.

ly through forward data-partition property propagation. This process tracks output data-partition property in a `CurrentPostDP` field for each vertex and discovers valid query plans along the way. The goal is to set the needed steps in each data-shuffling stage in order to generate valid execution plans. This is done through a recursive procedure `ForwardExplorer` (shown in Figure 8), which takes the current execution graph (`currGraph`) and the current suffix of the topologically ordered list of vertices (`traverseSuffix`). The procedure also uses three primitives: `DPPropertyTransition` takes an input data-partition property and a functional property, and outputs the resulting data-partition property based on the propagation graph in Figure 4; `GetShufflingSteps` takes an input data-partition property and an output data-partition property, and outputs the needed data-shuffling steps according to Figure 2; `ValidateMerge` checks whether input vertices of a merger phase all conform to the same data-partition property (PSorted or GSorted on the merge key).

Using the same example in Figure 7, we show how the algorithm can create a different execution plan: the `CurrentPostDP` is set to AdHoc for the input data vertices and PSorted for S1 and S2. Because the Merger is a pass-through function, its `CurrentPostDP` is set to PSorted. The `CurrentPostDP` is also set to PSorted after the Mapper because it is strictly-monotonic. Because PSorted implies Clustered, which is the weakest postcondition for S3, all steps of S3 can be removed

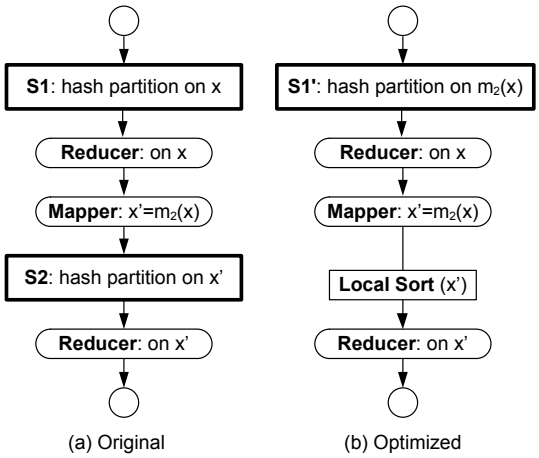

Figure 9: Chained shuffling optimization.

to produce an alternative valid plan.

For simplicity, this algorithm enumerates all valid execution plans. It is conceivable to add heuristics to prune the candidates if there are too many valid execution plans to evaluate. All valid executions are evaluated based on the cost model, where the one with the lowest cost will be chosen.

4 Re-Defining Partitioning Key

Not all UDFs have the desirable functional properties for preserving data-partition properties, especially when we conservatively assume the input data can be arbitrary. SUDO can further leverage the ability to re-define a partitioning key to apply some constraint to the input data so as to preserve certain data-partition properties (e.g., Disjoint) for optimizing data-shuffling. These mechanisms further increase the coverage of the optimization framework described in Section 3. This section describes how we can re-define partitioning keys to preserve data-partition properties.

Considering the simple case in Figure 9 with two data-shuffling stages S_1 and S_2, where S_1 does a hash re-partitioning on key x, S_2 does a hash re-partitioning on key x', and mapper m_2 is not one-to-one, we cannot eliminate steps in S_2 using the algorithm discussed in Section 3 because m_2 does not have the needed functional property. It is however possible to re-define the partitioning key in S_1 by taking into account m_2 and S_2, in order to eliminate some steps in S_2. For example, if m_2 maps each x to a single x', SUDO can apply hashing on $m_2(x)$ for S_1, rather than on x directly. Then, this hash re-partitioning ensures not only that all records with the same x are mapped to the same partition, but also that all records with the same x' after applying m_2 are mapped to the same partition. This can help eliminate

the re-partitioning step in S_2.

It is worth pointing out that there are side effects in this optimization, although it reduces the amount of total network I/O by eliminating a later re-partitioning step. The re-partitioning in S_1' is slightly more expensive as it has to invoke m_2 on all input records; the number of these records might far exceed the number in the later mapper phase with m_2 because of the reducer on x. To reduce this extra overhead, SUDO applies program slicing to get a simpler function, as done in the analysis of functional properties. Also, the new partitioning might lead to data skew that does not exist in the original plan because of the different partitioning key used. Again, a cost model is used to assess whether or not to apply this optimization.

We can easily generalize this type of optimizations to a chain of data-shuffling stages. $S_1, S_2, \ldots, S_N$ is a chain of data shuffling with hash re-partitioning, where before each S_i (except S_1) there is a mapper with UDF m_i ($i = 2 \ldots N$). To guarantee that a single re-partitioning causes all the keys in later phases to be partitioned appropriately, we can construct partition function in S_1 as a hash on $(m_N \ldots m_3 m_2(x))$ where x is the initial input key.

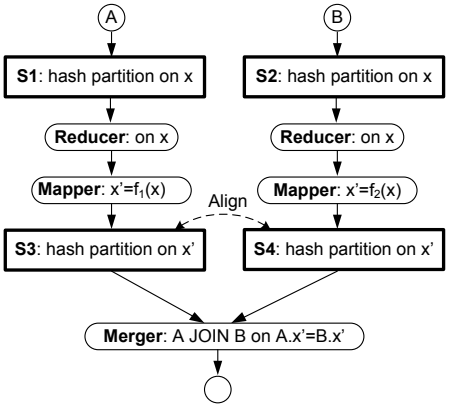

Figure 10: Joined shuffling optimization.

Two other data-shuffling patterns, *joined shuffling* and *forked shuffling* are more complicated. Joined-shuffling is widely used to implement a JOIN to correlate records in data-parallel programs. Figure 10 shows one such case. Shuffling stages S_3 and S_4 are required and inserted by the merger phase. This pattern can be considered as the merge of multiple chained data shuffling: in this example, one formed by S_1, f_1, and S_3, while the other by S_2, f_2, and S_4. Separately, SUDO can use a hash function on $f_1(x)$ for S_1 and a hash function on $f_2(x)$ for S_2 in order to remove re-partitioning in S_3 and S_4. Due to merger, the two chains must be aligned in that the same key will be mapped to the same partition. This is easily achievable as long as SUDO applies the same hashing to $f_1(x)$ and $f_2(x)$.

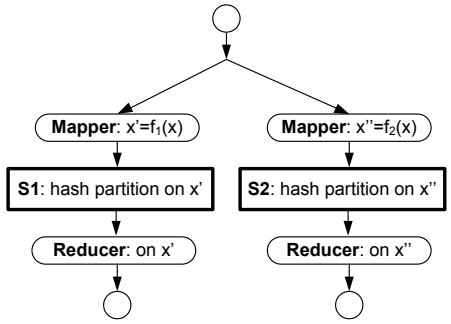

Figure 11: Forked shuffling optimization.

Figure 11 shows an example of forked shuffling, where an input is consumed by two separate threads of processing. In the figure, f_1 and f_2 are two mapper functions that map a record key from x to x' and x'', respectively. Two data-shuffling stages S_1 and S_2 perform hash re-partitioning on x' and x'', respectively. It is conceivable that SUDO can perform one data-shuffling by hashing on a function of x, to create disjoint data partitions both for $f_1(x)$ and for $f_2(x)$. That is, the data-shuffling must guarantee that, if $f_1(x_1) = f_2(x_2)$, then x_1 and x_2 are mapped to the same partition by S_0. For example, given $f_1(x) = \lfloor x/2 \rfloor$ and $f_2(x) = \lfloor x/3 \rfloor$, stage S_0 should use $\lfloor x/6 \rfloor$ as the re-partitioning key. Such a function might not always exist, as is the case with $f_1(x) = \lfloor x/2 \rfloor$ and $f_2(x) = \lfloor (x+1)/2 \rfloor$. Constructing the symbolic function automatically is challenging. We resort to recording known patterns and recognizing them through pattern matching on a sliced data-dependency graph. Currently, SUDO includes the following patterns: $f(x) = \lfloor x/a \rfloor$, $f(x) = x \ mod \ a$, where x is an integer.

SUDO creates new execution options with re-defined partitioning keys: they can be regarded as special mechanisms to preserve certain data-partition properties. Those options can be integrated into the framework described in Section 3 to create more valid execution plans, which will be evaluated using a cost model.

5 IMPLEMENTATION

We have implemented SUDO based on and integrated into the SCOPE compiler [10] and optimizer [47]. SCOPE is a SQL-like scripting language for data-parallel computation. The SCOPE optimizer uses a transformation-based optimizer to generate efficient execution plans. The optimizer leverages many existing work on relational query optimization and performs rich and non-trivial query rewritings that consider the input script in a holistic manner. The main contribution of the SUDO imple-

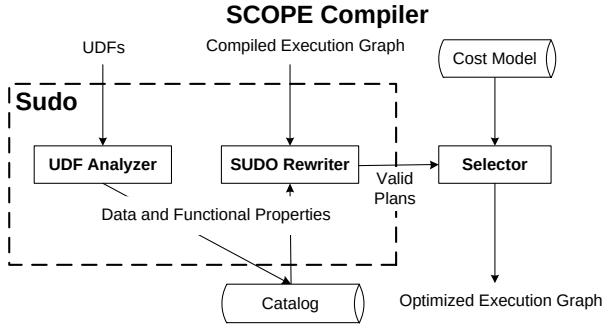

Figure 12: SUDO architecture.

mentation is to add the support of reasoning about functional properties, data-partition properties, and data shuffling into the current optimizer: without understanding UDFs, the system cannot derive any structural properties and thus potentially miss important optimization opportunities. With SUDO, functional properties between input and output columns are identified and integrated into the optimization framework. It enables efficient property derivation and allows the optimizer to optimize query plans with UDFs effectively.

Figure 12 depicts an architectural overview of SUDO, its components, and their interactions with the existing components in the SCOPE compiler and optimizer. The *UDF analyzer* and the *rewriter* are the two main SUDO modules. The UDF analyzer extracts the functional properties of the UDFs, while the *rewriter* generates optimized execution plans by leveraging the functional properties. The current implementation for the two modules contains 8,281 and 1,316 lines of C# code, respectively. The execution plans are further fed to the *selector*, which uses a cost model to find the best one among them.

5.1 UDF Analyzer

We have implemented the UDF analyzer at the high-level IR (HIR) of the Phoenix framework [35] shipped with Visual Studio 2010, together with the bddbddb engine [1]. Phoenix is a framework for building compiler-related tools for program analysis and optimizations. It allows external modules to be plugged in with full access to the internal Phoenix Intermediate Representation (IR). With Phoenix, the analyzer feeds the instructions and the library calls for a given UDF to the bddbddb engine, together with the deduction rules. The engine then applies the deduction process, as discussed in Section 3.2.

We describe in further detail the rules in SUDO. In the extracted IR excluding opcode `CALL`, we select top 8 unary operators and 7 binary operators in terms of frequency of use. These operators account for 99.99% of operator uses (again excluding `CALL`). We have a total of 28 rules for those operators. Coming up with those 28 rules requires some care. First, if done naively, we might have many more rules. We have reduced the number with the following two approaches: (i) some instruction type, such as `ASSIGN` and `BOX`, belong to the same *equivalent class* as they share the same set of deduction rules. (ii) binary operators (e.g., the `ADD` operator shown in Figure 5 (b)) usually requires more than one rule. We manually introduce several pseudo operators to convert some to others. For example, we add `NEGATE` and `RECIPROCAL` in order to convert `SUBSTRACT` and `DIVIDE` to `ADD` and `MULTIPLY` respectively, thereby reducing the total number of rules.

Second, *constraints* are often needed in the rules for precision. The constraints could be on some aspects of the operands, such as their types and value ranges. For example, the `CONVERT` operator is used to convert numbers between different types. Converting a number from a type with a smaller byte size to one with a larger size (e.g., from `int` to `double`) preserves its value; that conversion is considered a pass-through function. This is *not* the case for the opposite direction. SUDO extracts operand types and makes the rules type-sensitive with the type constraints embedded to handle these cases.

Finally, the UDFs contain loops and branches. The value of an operand may come from any one of its input operands defined in any of the branches. SUDO introduces the `INTERSECT` operator with the rule stating that the output operand has a certain property if both its input operands have the same functional property.

For the 157 system calls in the extracted IR of mscorlib and ScopeRuntime, we set 73 rules for 66 unary calls and 10 binary calls. The remaining system calls are marked `Func` (as in line 12 of Figure 5). For the 891 third-party calls (with no source code), we select top 10 most-frequently used and wrote 16 rules for them. Most of them are string related (e.g., `UrlReverse`, `PrefixAdd`, and `String.Concat`), while others are mostly math related.

5.2 SUDO Rewriter

SUDO rewriter generates all valid execution plans using the algorithms presented in Section 3, as well as the mechanisms described in Section 4. The implementation of the algorithm for enumerating all of the valid plans described in Section 3 is straightforward. In practice, the techniques in Section 4 plays an important role in uncovering optimization opportunities even after manual performance tuning and optimizations. The goal of SUDO rewriter is in principle similar to that of the SCOPE optimizer in that they are both enumerating all valid plans for cost estimation, although with key differences; for example, the SUDO rewriter works at the "physical" lev-

el, while the SCOPE optimizer starts with logical relational operators. For ease of implementation, the current SUDO rewriter simply takes as input the best physical execution plan from the SCOPE optimizer. The results from the SUDO rewriter are then assessed based on the internal cost model of the SCOPE optimizer. This simple integration might lead to sub-optimal results as two optimization phases are carried out separately. We are in the process of integrating SUDO into the SCOPE optimizer to reason about functional properties and structured data properties in a single uniform framework, and seamlessly generates and optimizes both serial and parallel query plans.

6 EVALUATION

In this section, we use real traces in a SCOPE production bed to assess the overall potential for SUDO optimizations and evaluate in details the effectiveness of those optimizations on representative jobs (pipelines).

6.1 Optimization Coverage Study

We have studied 10,099 jobs collected over a continuous period of time from a production bed with tens of thousands machines. The study aims at revealing the distribution of functional properties for the UDFs in those jobs and gauging the optimization opportunities for SUDO to leverage those properties.

Property	UDF funcs (#)	Ratio(%)
Pass-through	1,998,819	84.73
Strictly-increasing	147,820	6.27
Strictly-decreasing	0	0.00
Increasing	138	0.00
Decreasing	0	0.00
One-to-one	1,758	0.08
Func	210,544	8.92
Sum	2,359,079	100

Table 2: Statistics on functional properties.

We first look at the computation for each output column in each UDF to infer its functional property. Our analysis is limited to handling only computation where an output column in a row depends on a single input column in a single input row. Among the 236,457 UDFs and 3,000,393 output columns, 2,359,079 (78.63%) output columns satisfy this constraint. We then carry out our study on the 2,359,079 functions that produce those output columns.

For each functional property, Table 2 reports the number of functions that have this functional property and the percentage. For a function that satisfies multiple functional properties, only its strongest functional property is counted. For example, for a strictly-increasing function, it is counted only as strictly-increasing, but not as increasing. Pass-though functions clearly dominate and accounts for 84.73%, followed by strictly-increasing functions with 6.27%. A small fraction (0.08%) of the functions are one-to-one. About 8.92% of the functions do not have any of the functional properties SUDO cares about. Surprisingly, we do not find any (strictly-) decreasing functions.

We further run our optimizer to see how many jobs SUDO can optimize by leveraging the identified functional properties. This study focuses only on jobs with more than one data-shuffling stages. This eligible job set contains a total of 2,278 (22.6%) jobs. Among all the eligible jobs, SUDO is able to optimize 17.5% of them.

It is worth noting that the original SCOPE compiler supports user annotations of the pass-through functional property. In practice, 6% of the eligible jobs have been annotated, and 4.6% are actually optimized. Our automatic analysis engine is shown to have discovered significantly more optimization opportunities.

6.2 Optimization Effectiveness Study

To study the optimization effectiveness on individual jobs (pipelines), we select three important web search related SCOPE jobs. All the jobs are running on the same cluster as the one we collected the trace from for our UDF study. The number of machines used in each job depends on the size of the input data. Table 3 summarizes the optimization details for each case, while Table 4 gives the detailed performance comparisons between the original versions and the optimized ones. In this section, we describe these cases with their optimizations and then discuss the performance numbers together for ease of comparison.

6.2.1 Anchor Data Pipeline

Hyperlinks in web pages form a web graph. Anchor texts associated with hyperlinks, together with the web graph, are valuable for evaluating the quality of web pages and other search-relevance related metrics. One of the basic anchor-data pre-processing jobs is to put the anchors that point to the same page together (using a data-shuffling stage), and de-duplicate the anchors with the same text. The job further outputs the reversed url and the anchor text pairs, e.g., (“org.acm.www/sigs”, anchor text) instead of (“www.acm.org/sigs”, anchor text), as this reversed url format is the de-facto representation of a url as urls of the same domain are laid out contiguously in that format to enable simple domain-level aggregations.

The optimization opportunity comes when other jobs consume the output of this job. Figure 13 shows an example where the second job *Job2* tries to count the words in the anchor text for each url. Before optimization, Job2 has to insert a data shuffling stage (*S2*) to group the data by url. However, this operation is redundant because

Case	OptStages	FuncProperty	PreservedDP	RPF Stages	PreservedDP+	EndShuffSteps
§ 6.2.1	S2	One-to-One	Clustered	{∅}	Clustered	{∅}
§ 6.2.2	S2,S3	Increasing,Increasing	LSorted,LSorted	{S1}	PSorted,PSorted	{∅},{∅}
§ 6.2.3	S3,S4, S5,S6	None,None, Pass-through,Pass-through	AdHoc,AdHoc, GSorted,GSorted	{S1,S2}	Disjoint,Disjoint, GSorted,GSorted	{LS},{LS}, {∅},{∅}

Table 3: Optimization detail for the three cases. The columns represent the section reference, the shuffling stages that are optimized (*OptStages*), the functional properties of the UDFs before the shuffling stages, preserved data-partition property by the UDFs (*FuncProperty*), shuffling stages having their partition functions re-defined (*RPF Stages*), data-partition properties preserved with both *FuncProperty* and *RPF*, and the final steps for each in *OptStages*. *LS* stands for local sort.

url reversing is a one-to-one function, which preserves the Clustered property. SUDO recognizes this functional property and eliminates stage *S*2.

6.2.2 Trend Analysis Pipeline

Trend analysis is a way to understand how things change over time, which is important for many search related applications as well as being a stand-alone service. One kind of trend-analysis job collects the $\langle term, time, aspect \rangle$ tuples in the search-query logs, where *term* is the search keyword, *time* is when the query is submitted, and *aspect* is one of the search query's property (e.g., its market), and aggregates various aspects' occurrences at different time-scales such as hours, days, and months. For instance, we can know the top three market along the years for a specific brand using trend analysis.

Figure 14 shows a pipeline of trend-analysis jobs. Job1 pre-processes the input query logs and aggregates the entries within the same second for all aspects we need. Based on this result, Job2 calculates the $\langle term, time, market \rangle$ distribution over days, and Job3 calculates the $\langle term, time, intent \rangle$ distribution over weeks,

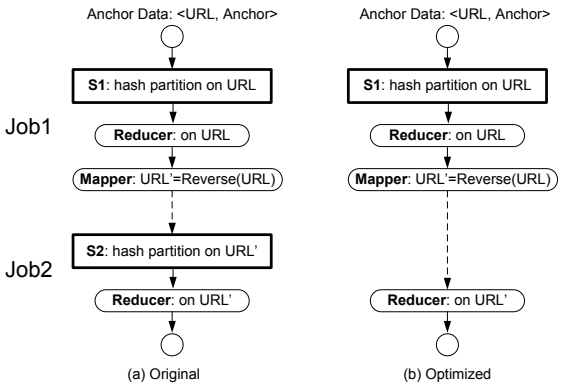

Figure 13: Optimization on the anchor data pipeline. The redundant shuffling stage in Job2 is eliminated.

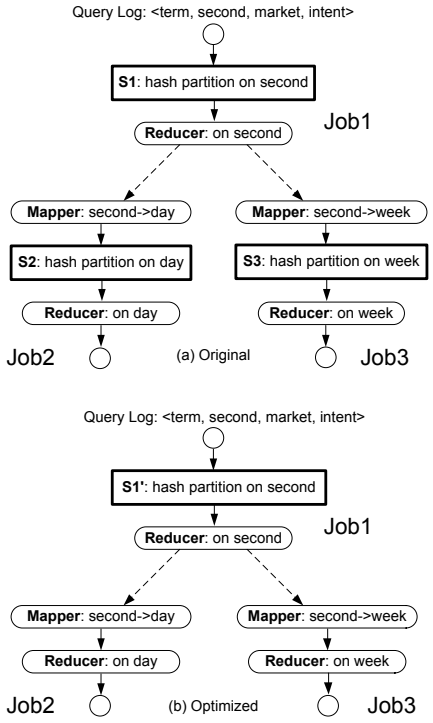

Figure 14: Optimization on the trend-analysis pipeline. The two data-shuffling stages on new time-scales are merged into the previous shuffling.

where intent is obtained from user-click-behavior of the queries and tells whether the search query is for information, business, or other purposes. Before optimization, each job requires a data-shuffling stage. SUDO merges the three shuffling stages into one and redefines the partition key in Job1. The function ensures that the seconds within the the same week are always on the same partition, which guarantees the Disjoint property even after the two mapper functions that converts seconds to days and weeks, respectively. Besides, because the time conversion function is increasing, LSorted is preserved and the local-sort operations in Job2 and Job3 are eliminat-

ed. Together, the optimization eliminates two shuffling stages, and ensures PSorted property before each reducer function in the three jobs.

6.2.3 Query-Anchor Relevance

Search queries and anchors are two term sets that can be bridged by urls. Looking at the joined data set is important to improve the search quality. If a query happends to result in a url that an anchor points to, then the query and the anchor are very likely to be relevant. For instance, if the word "China" appears frequently in query with result URL `example.org/a.html`, and the word "Emerging Market" appears in the anchor that points the same `example.org/a.html`, then "China" and "Emerging Market" are relevant. Furthermore, if these two words appear in `example.org` many times and their pointing-to urls also overlap a lot, they have a high relevance.

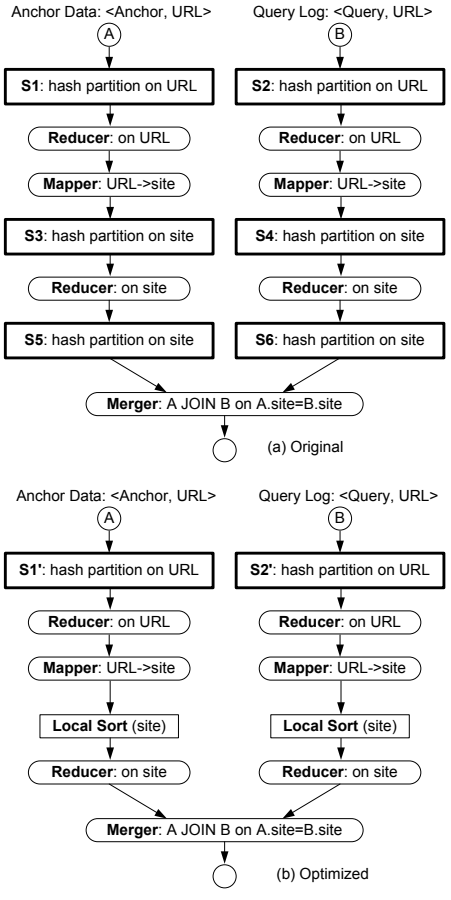

Figure 15: Optimization on anchor-query relevance analysis jobs. Four shuffling stages are saved.

Figure 15 shows a job that aggregates anchor data as well as query logs, correlates them via sites, and applies some learning models for relevance study (omitted in the figure). The job first groups anchor texts on url (around $S1$), and reduces that into $\langle$url, map$\langle$*term*,*frequency*$\rangle\rangle$, where *term* is a keyword inside anchor texts and *frequency* is the number of times that the term occurred in these anchors. Then it converts urls to sites using a mapper, and groups the records on the same sites ($S3$). Inside the reducer, it computes an aggregated frequency for each term based on the data for different urls (within the same site). The job applies almost the same algorithm to the query logs too. After that, it joins these two output datasets on site for further study with two further shuffling stages ($S5$ and $S6$). There are in total six shuffling stages in the job; SUDO applies its optimization and makes it two, as shown in the figure. The new shuffling stages partitions the input data according to the sites they belong to, so as to keep the Disjoint property along the data flow. However, because the desired data-partition property for reduce on site is not satisfied (the mapper function which converts url to site does not preserve any data-partition property), SUDO partially eliminates stage $S3$ and $S4$ with a local-sort operation only. Because the reducer on site does not change the site key, the last two data-shuffling stages can be eliminated as the PSorted property is preserved.

6.2.4 Optimization Effectiveness

Table 4 gives the performance comparisons between the original versions and the optimized ones. It shows significant read I/O reduction (47%, 35%, and 41%) for all three cases. In particular, cross-pod read data usually has a much larger reduction compared to intra-pod read data; this is because most data-shuffling stages involve a large number of machines (1,000 and 2,500) that cannot fit in a single pod except those in the first case (150).

The I/O reduction leads to the reduction of vertex numbers and vertex hours, where a vertex is a basic scheduling unit and a chain of UDFs can be continuously applied on one or more data partitions. The vertex numbers are reduced when the repartitioning step in the shuffling stages is removed, and the reduction ratio is basically proportional to the percentage of optimized shuffling stages (40%, 38%, and 82%). The reduction in vertex hour is relatively small and not proportional to the saved I/O; this is partially because the time complexity of the computation in the vertices is not necessarily linear to the input size ($O(n)$); for example, local sort takes $O(n*log(n))$, and the optimization introduces data-skew problems (described next).

Usually, significantly reduced I/O leads to much reduced end-to-end execution time (e.g., 47% vs. 40% and 35% vs. 45% for the first two cases). However, as discussed before, when redefining partition keys, SUDO

Case	Setting	In(GB)	Node#	E2E(min)	Vertex(hour)	Vertex#	TotalR(GB)	IPR(GB)	CPR(GB)
§ 6.2.1	original	241	150	25	35	2,974	902	102	800
	optimized	241	150	15	28	1,780	474	43	431
	reduction	-	-	40%	20%	40%	47%	59%	46%
§ 6.2.2	original	10,118	1,000	230	5,852	382,708	60,428	12,437	47,991
	optimized	10,118	1,000	127	3,350	237,751	39,080	11,658	27,422
	reduction	-	-	45%	42%	38%	35%	6%	43%
§ 6.2.3	original	241/341	2,500	96	856	352,406	14,651	2,970	11,681
	optimized	241/341	2,500	122	371	62,619	8,677	2,656	6,021
	reduction	-	-	**-27**%	57%	82%	41%	11%	48%

Table 4: Performance comparisons between unoptimized and optimized jobs. *In* indicates the input data size. *E2E* refers to the end-to-end time. *Vertex(hour)* represents the total vertex running-time across all machines. *TotalR*, *IPR*, and *CPR* report the sizes of total, intra-pod, and cross-pod read data for shuffling. The third case involves a join of two input sets, whose sizes are reported separately.

may bring in data-skew problems. We did observe a serious data-skew problem that causes the optimized job to be slower (-27%) than the original one in the third case, which was introduced by enforcing the records for the same sites on the same data partition before the first reduce function. The data skew problem leads to stragglers among partitions at the same computation phase and hurts the overall latency. In the future, we need more work on handling data-skew problems better and/or on the cost model for better execution plan selection.

6.3 Observations and Future Directions

During our study on the SCOPE jobs, we have also observed a set of interesting and somewhat "negative" issues that are worth further investigation. First, unlike in a typical MapReduce model, where reducers and mergers are constrained to be pass-through functions for the keys, SCOPE's programming model is different and allows reducers and mergers to change keys. Knowing that a reducer or merger in SCOPE is a pass-through function enables more optimizations in SCOPE, but we do not count them in our study because those are specific to SCOPE. In general, it is interesting to observe that a more constrained language often leads to better automatic optimizations. A good programming language must carefully balance flexibility and optimizability.

Second, we observed that for quite a few jobs, the I/O cost is dominated by the initial "extractor" phase to load the data initially. A careful investigation shows that some of those jobs were loading more data than necessary and could benefit greatly from notions like column groups in BigTable or any column-based organizations.

Third, for some jobs, we found that the first data-shuffling stage dominates in terms of shuffling cost, as the output size of the reduce phase after that stage is significantly smaller than the input. This was one of the reasons that caused us to look at pipeline opportunities because a first phase in a job might also be optimized if it takes outputs from another job.

Finally, the rule-based deduction can be further improved. To ensure soundness, our current implementation is conservative and can be improved by making the analysis *context-sensitive* and *path-sensitive*. By being context-sensitive, SUDO's analysis will be able to differentiate the cases where a function is invoked by different callers with different parameters; by being path-sensitive, SUDO's analysis takes branching conditions into account. SUDO can further incorporate the value-range information to handle operators, such as `MULTIPLY`, whose functional properties depend on the value ranges of the input operands. Currently, SUDO recognizes the sign of all constant numbers automatically, but requires that developers mark the value range of an input column if it is used as a partitioning key and involved in the deduction process with value-range sensitive operators. It would be ideal not having to depend on such annotations.

7 Related Work

SUDO proposes an optimization framework for data-parallel computation, as pioneered by MapReduce [15], followed by systems such as Hadoop [3] and Dryad [23]. The MapReduce model has been extended [12] with Merge to support joins and to support pipelining [13]. CIEL [30] is a universal execution engine for distributed data-flow programs. High-level and declarative programming languages, often with some SQL flavors, have been proposed. Examples include Sawzall [36], Pig [33], Hive [40], SCOPE [10], and DryadLINQ [46]. Those high-level programs are compiled into low-level execution engines such as MapReduce. Because SUDO works at the low-level Map/Reduce/Merge model, the optimizations can be applied in general to these systems: the extensive support of UDFs and heavy use of data-shuffling are common in all those proposals. The trend of embracing high-level programming languages also plays into our favor: SUDO can be integrated nicely into

the compiler/optimizer, can rely on high-level semantics, and often have the extra flexibility of re-defining partition functions, as they are being generated by a compiler/optimizer. In fact, many pieces of recent work (e.g., [43, 37, 11, 21, 20]) have introduced notions of query optimizations into MapReduce and its variants.

The idea of tracking data properties has been used extensively in the database community and dates back at least to System R [38], which tracks ordering information for intermediate query results. Simmen, *et al.* [39] showed how to infer sorting properties by exploiting functional dependencies. Wang, *et al.* [41] introduced a formalism on ordering and grouping properties; it can be used to reason about ordering and grouping properties and to avoid unnecessary sorting and grouping. Neumann and Moerkotte [32, 31] also described a combined platform to optimize sorting and grouping. Zhou, *et al.* [47] are the first to take into account partitioning, in addition to sorting and grouping, in the context of the SCOPE optimizer. Such optimization heavily relies on reasoning about data properties and applies complex query transformation to choose the best execution plan in a cost-based fashion. Agarwal, *et al.* [5] collects code and data properties by piggybacking on job execution. It adapts execution plans by feeding these contextual properties to a query optimizer. The existence of UDFs prevents the system from effectively reasoning relationship between the input and the output data properties [21]. As a result, the system may miss many important optimization opportunities and end up with suboptimal query plans. SUDO builds upon the idea of reasoning about data properties, defines a set of simple and practical data-partition properties, and takes UDFs and their functional properties into account.

The need to analyze UDFs, by means of different static analysis techniques such as dataflow analysis [6, 7, 28], abstract interpretation [14], and symbolic execution [19, 22, 25], has also been recognized. Ke *et al.* [26] focuses on data statistics and computational complexity of UDFs to cope with data skew, a problem that has been also studied extensively [16, 34, 17, 9, 8]. Manimal [24] extracts relational operations such as selection, projection and data compression from UDFs through static analysis, so that traditional database optimization techniques can be applied. Manimal use the ASM byte-code manipulation library [4] to process the compiled byte-code of UDFs. Given that previous papers [26, 24] care about different properties for UDFs for different optimization properties, it is an interesting future direction to see whether a coherent UDF framework can be established to enable all those optimizations. Scooby [44] analyzed the dataflow relations of SCOPE UDFs between input and output tables, such as column independence, column equality, and non-null of column's value, by extending the Clousot analysis infrastructure [27] so that the analysis can handle .NET methods. In comparison, SUDO extends the analysis to cover functional properties such as *monotone*, *strict monotone* and *one-to-one* of UDFs, other than column equality analysis, in order to optimize data shuffling. Yuan *et al.* [45] investigated the user-defined aggregations, especially the properties of commutative and associative-decomposable to enable partial aggregation. Such properties are explicitly annotated by programmers. Jockey [18] precomputes statistics using a simulator that captures the job's complex internal dependencies, accurately and efficiently predicting the remaining run time at different resource allocations and in different stages of the job to maximize the economic utility while minimizing its impact on the rest of the cluster. Steno *et al.* [29] can translate code for declarative LINQ [2] queries to type-specialized, inlined, and loop-based imperative code that is as efficient as hand-optimized code. It applies data analysis to eliminate chains of iterators and optimizes nested queries.

8 CONCLUSION

Extensive use of user-defined functions and expensive data-shuffling are two defining characteristics of data-parallel computation in the map/reduce style. By defining a framework that connects functional properties, data-partition properties, and data shuffling, SUDO opens up a set of new optimization opportunities that are proven effective using the workload in a large-scale production cluster. SUDO also reflects our belief that data-parallel computation could learn not only from database systems and distributed systems, but also from programming languages and program analysis. A systematic approach that combines those perspectives in a seamless framework is likely to bring tremendous benefits.

ACKNOWLEDGEMENT

We thank our shepherd, Srikanth Kandula, and the anonymous reviewers for their insightful comments. We are grateful to Chang Liu and Sean McDirmid for valuable feedback, and to Chao Wang, Zekan Qian, Bo Cao, and Lin Song for sharing production jobs and data.

REFERENCES

[1] bddbddb. `http://bddbddb.sourceforge.net/`.

[2] LINQ. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`.

[3] Hadoop. `http://lucene.apache.org/hadoop/`, June 2007.

[4] ASM. `http://asm.ow2.org/`, 2010.

[5] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *NSDI'12*, 2012.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[7] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[8] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in MapReduce clusters. In *EuroSys*, 2011.

[9] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *OSDI*, 2010.

[10] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[11] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a SQL implementation on the MapReduce framework. *PVLDB*, 4(12), 2011.

[12] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, 2010.

[14] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28, June 1996.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[16] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.

[17] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *SOCC*, 2011.

[18] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[19] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *RV*, 2009.

[20] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC'10*, 2010.

[21] H. Herodotou, F. Dong, and S. Babu. MapReduce programming and cost-based optimization? Crossing this chasm with Starfish. *PVLDB*, 4(12), 2011.

[22] W. E. Howden. Experiments with a symbolic evaluation system. In *AFIPS*, 1976.

[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[24] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *PVLDB*, 4(6), 2011.

[25] R. H. H. Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4), 1985.

[26] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS XIII*, 2011.

[27] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC*, 2008.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, 2011.

[30] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[31] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, 2004.

[32] T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *ICDE*, 2004.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[34] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[35] Phoenix. `http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx`, June 2008.

[36] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[37] G. Sagy, D. Keren, I. Sharfman, and A. Schuster. Distributed threshold querying of general functions by a difference of monotonic representation. *PVLDB*, 4(2), 2010.

[38] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[39] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, 1996.

[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2), 2009.

[41] X. Wang and M. Cherniack. Avoiding ordering and grouping in query processing. In *VLDB*, 2003.

[42] M. Weiser. Program slicing. In *ICSE*, 1981.

[43] S. Wu, S. Agarwal, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, 2011.

[44] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *SAS*, 2009.

[45] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, 2009.

[46] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[47] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, 2010.

APPENDIX:

A *partitioned dataset* with m partitions can be represented by a sequence $D = \langle S_1, S_2, \ldots, S_m \rangle$, where S_i is its i^{th} partition. Each partition S_i is represented by a sequence $(x_{i1}, x_{i2}, \ldots, x_{i,n_i})$, where x_{ij} is the key of the j^{th} item in the i^{th} partition. We say $x \in S_i$ if x is a key in the sequence of S_i. Keys in the sequences have orders. We can extend the order to between sequences. We say $S_i < S_j$ if $x < y$ for each $x \in S_i$ and $y \in S_j$. For a function f defined on keys, we can further define $f(D)$ a new partitioned dataset as $\langle f(S_1), f(S_2), \ldots, f(S_m) \rangle$, where $f(S_i) = (f(x_{i1}), f(x_{i2}), \ldots, f(x_{i,n_i}))$ for each $1 \leq i \leq m$.

Definition 1 (Clustered)**.** A partitioned dataset D is Clustered, if and only if, for any $x_{ij} = x_{i'j'}$, the following holds: (a) $i = i'$ (b) $x_{ik} = x_{ij}$ for all k with $j < k < j'$. That is, items with the same key must be stored contiguously in the same partition.

Definition 2 (LSorted)**.** A partitioned dataset D is LSorted if and only if, either $x_{ij} \leq x_{ij'}$ for any $1 \leq i \leq m$ and $j < j'$ (increasing), or $x_{ij} \geq x_{ij'}$ for any $1 \leq i \leq m$ and $j < j'$ (decreasing). That is, keys in the same partition must be sorted the same way in the sequence.

Definition 3 (GSorted)**.** A partitioned dataset D is GSorted if and only if the following holds: (a) D is LSorted and (b) either $S_i < S_{i'}$ for any $1 \leq i < i' \leq m$ (increasing), or $S_i > S_{i'}$ for any $1 \leq i < i' \leq m$ (decreasing). That is, keys are not only sorted within each partition, but also sorted across partitions.

Definition 4 (Data-Partition Property Preserving)**.** Function f preserves a data-partition property ϕ (Clustered, LSorted, or GSorted) if, for any D satisfying ϕ, $f(D)$ also satisfies ϕ. (Here we implicitly assume that f maps from the domain of keys to the same domain.)

Lemma 1. *(1) If f is strictly increasing and $S_i < S_{i'}$, then $f(S_i) < f(S_{i'})$. (2) If f is strictly decreasing and $S_i < S_{i'}$, then $f(S_i) > f(S_{i'})$.*

Proof. (1) For any $f(x) \in f(S_i)$ and $f(y) \in f(S_{i'})$, we have the corresponding $x \in S_i$ and $y \in S_{i'}$. $S_i < S_{i'}$ then implies $x < y$. Because f is strictly increasing, $f(x) < f(y)$ holds. Therefore, we have $f(S_i) < f(S_{i'})$. The proof for (2) is similar. □

Lemma 2. *If a partitioned dataset D is GSorted, then D is also Clustered and LSorted.*

Proof. By definition of GSorted (Definition 3), D is LSorted.

We now prove that D is also Clustered. Given two keys in D $x_{ij} = x_{i'j'}$, we want to prove the following hold: (a) $i = i'$ and (b) $x_{ik} = x_{ij}$ for all k with $j < k < j'$. First we prove (a) by contradiction. If $i \neq i'$, because D is GSorted, $S_i < S_{i'}$ or $S_{i'} < S_i$ holds. Therefore, $x_{ij} = x_{i'j'}$ cannot hold. Contradiction. Then we have $i = i'$. To prove (b), without loss of generality, we assume that partition i is locally increasingly sorted. For all k with $j < k < j'$, we have $x_{ij} \leq x_{ik} \leq x_{ij'}$. Since $x_{ij} = x_{i'j'} = x_{ij'}$, $x_{ik} = x_{ij}$ holds. D is Clustered. □

Theorem 1. *f preserves Clustered if and only if f is one-to-one.*

Proof. If f is one-to-one, for any D that is Clustered, we show that $f(D)$ is Clustered. That is, if $f(x_{ij}) = f(x_{i'j'})$, then the following must hold: (a) $i = i'$ and (b) $f(x_{ik}) = f(x_{ij})$ for any k with $j < k < j'$. Because f is one-to-one, $f(x_{ij}) = f(x_{i'j'})$ implies $x_{ij} = x_{i'j'}$. We have $i = i'$ because D is Clustered. For any $j < k < j'$, again because D is Clustered, $x_{ik} = x_{ij}$ holds. Because f is one-to-one, $x_{ik} = x_{ij}$ implies $f(x_{ik}) = f(x_{ij})$. Therefore, $f(D)$ is Clustered.

Next, we show that, if f preserves Clustered, then f is one-to-one. Prove by contradiction. Suppose there exists a function f that preserves Clustered, but is not one-to-one. There must exist two valid keys x and y, such that $x \neq y$ and $f(x) = f(y)$ hold. We construct a partitioned dataset $D = \langle (x), (y) \rangle$ that is Clustered; that is, the dataset has two partitions, with the first partition having only one key x and the second having only one key y. $f(D) = \langle (f(x)), (f(y)) \rangle$ is not Clustered because of $f(x) = f(y)$. Contradiction.

□

Theorem 2. *f preserves LSorted if and only if f is monotonic.*

Proof. First, we prove that, if f is monotonic, for any D that is LSorted, $f(D)$ is also LSorted. We consider the following four cases: for any two valid indexes j and j' in any partition i,

1) D is increasing and f is increasing: $j < j' \Rightarrow x_{ij} \leq x_{ij'} \Rightarrow f(x_{ij}) \leq f(x_{ij'})$.
2) D is increasing and f is decreasing: $j < j' \Rightarrow x_{ij} \leq x_{ij'} \Rightarrow f(x_{ij}) \geq f(x_{ij'})$.
3) D is decreasing and f is increasing: $j < j' \Rightarrow x_{ij} \geq x_{ij'} \Rightarrow f(x_{ij}) \geq f(x_{ij'})$.
4) D is decreasing and f is decreasing: $j < j' \Rightarrow x_{ij} \geq x_{ij'} \Rightarrow f(x_{ij}) \leq f(x_{ij'})$.

$f(D)$ is always LSorted.

Now we show that, if f preserves LSorted, then f is monotonic. Prove by contradiction. Suppose there exists a function f that preserves LSorted, but f is not monotonic. There must exist keys $x_1 < x_2$ and $x_3 < x_4$ hold, such that $f(x_1) < f(x_2)$ and $f(x_3) > f(x_4)$ hold. We construct an LSorted (increasing) D consisting of a single partition with $x_{1i} = x_1, x_{1i'} = x_2, x_{1j} = x_3, x_{1j'} = x_4$, where $i < i'$ and $j < j'$ hold. In $f(D)$, we have $f(x_{1i}) < f(x_{1i'})$ and $f(x_{1j}) > f(x_{1j'})$. Therefore, $f(D)$ is not LSorted. Contradiction. □

Theorem 3. *f preserves GSorted if and only if f is strictly-monotonic.*

Proof. We prove that, if f is strictly-monotonic, for any D that is GSorted, $f(D)$ is also GSorted. First, because f is monotonic, f preserves the LSorted property due to Theorem 2. Second, we show that f preserves the order among partitions. If D is increasing GSorted (i.e., for all $i < j$, $S_i < S_j$) and f is strictly increasing, then $f(S_i) < f(S_j)$ holds due to Lemma 1. A similar proof can be used to cover the three other cases depending

on whether GSorted increasing or decreasing and on whether f is strictly increasing or decreasing.

Next, we prove that, if f preserves GSorted, then f is strictly-monotonic. First, by Lemma 2, preserving GSorted implies preserving both Clustered and LSorted. Second, by Theorem 1 and Theorem 2, preserving Clustered implies f is one-to-one, and preserving LSorted implies f is monotonic. Therefore, preserving GSorted implies f is both one-to-one and monotonic, which implies that f is strictly-monotonic. □