Supporting Data Aspects in Pig Latin

Curtis E. Dyreson, Omar U. Florez, Akshay Thakre, and Vishal Sharma
Department of Computer Science
Utah State University
Logan, Utah, USA
curtis.dyreson@usu.edu,{omar.florez,akshay.thakre,vishal.sharma}@aggiemail.usu.edu

ABSTRACT

In this paper we apply the aspect-oriented programming (AOP) paradigm to Pig Latin, a dataflow language for cloud computing, used primarily for the analysis of massive data sets. Missing from Pig Latin is support for cross-cutting data concerns. Data, like code, has cross-cutting concerns such as versioning, privacy, and reliability. AOP techniques can be used to weave metadata around Pig data. The metadata imbues the data with additional semantics that must be observed in the evaluation of Pig Latin programs. In this paper we show how to modify Pig Latin to process data woven together with metadata. The data weaver is a layer that maps a Pig Latin program to an augmented Pig Latin program using Pig Latin templates or patterns. We also show how to model additional levels of advice, i.e., meta-metadata.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Metadata, cloud computing

General Terms

Management, Languages

Keywords

Aspect-oriented, Pig, cross-cutting concerns

1. INTRODUCTION

No matter whether data is stored in a database, flat file, spreadsheet, or as persistent objects, data has cross-cutting concerns. A *cross-cutting data concern* is a data need that is *universal* (potentially applicable to an entire database) and *widespread* (can be used to enhance many different databases). Many data collections have cross-cutting data concerns, and as a collection evolves, new concerns may arise. For instance, a new privacy policy is implemented to hide certain information in a Facebook page. A *privacy* cross-cutting concern could be added to the relevant Facebook data to hide it from the general public.

There are many data needs that are universal and widespread. Heretofore, these needs have not been seen as cross-cutting concerns. Data *security* and *privacy* policies govern every interaction with a datum, and have been researched for many years [5, 11, 16]. Security and privacy are of special concern in cloud computing since data is stored and processed in the cloud on potentially untrustworthy computers [7, 31, 39]. Data quality is another potential cross-cutting concern [2]. Data warehouses aggregate data from a variety of data sources of varying quality and queries that mix low and high quality data should provide a measure of quality along with a result. Data *provenance* [9, 10] and *lineage* [3, 8] track the data and/or processes that produce a query result, which aids in debugging and understanding complex queries. *Time* is another potential cross-cutting concern, both the time of the transaction that creates a datum and the time that it is valid in the real-world need to be tracked for many applications [30]. Each of these potential cross-cutting concerns has an individual, distinct semantics.

Currently, there does not exist a general framework to support cross-cutting data concerns (though systems often support individual concerns, e.g., security). Data management systems are large and complex, and are not designed to be easily configured or modified to support cross-cutting data concerns. Developers currently have to rely on ad-hoc techniques to add concerns to a data collection, or use a database management system that already supports a particular concern. To support cross-cutting data concerns a new paradigm is needed, one that looks to fields outside of databases for useful techniques and insights. Aspect-oriented programming (AOP) provides a framework that can be adapted to our needs. AOP was developed to extend existing programs with new functionality without having to reprogram.

1.1 Harnessing Aspects for Data Cross-cutting

Previously we used aspect-oriented techniques to create aspect-oriented data (AOD) for data stored in the relational model [12, 14]. AOD "tags" data with metadata from a cross-cutting data concern to create a data aspect. The aspect becomes active whenever the data is used. A data aspect weaver weaves behavior for the cross-cutting concern into the evaluation of a query, constraint, or object management operation. We showed how to weave behavior into the relational algebra [12]. There has also been other research in using AOP in databases. Research has addressed using aspect-oriented techniques to program databases [32], using a relational database to support AOP [33], and applying AOP to XML schema [15].

Figure 1 gives a broad classification of the space of cross-cutting data concerns using an AOP approach. In general, a data aspect has access to two things: *data* and *advice*, which is the metadata that annotates the data. A data aspect becomes active when the data is used in an operation in the sense that the aspect can *change* (insert, update, or modify) the data or make *no change*. The aspect

		Data	
		change	no-change (noop)
Advice	change	temporal privacy security quality	lineage provenance probabilistic
	no-change (noop)	vacuuming profiling	authored by language

Figure 1: The space of cross-cutting concerns

could also change the advice. In general, "change" or "no change" are the only possible effects (ignoring side effects like computation time involved). In Figure 1 the concerns are partitioned into four categories based on whether the advice and/or data changes. For example, a temporal cross-cutting data concern constructs new timestamps during some query operations, such as a join operation. The new timestamps become advice for some data, e.g., a tuple in the join result. These timestamps may (logically) delete data since the constructed times may be shorter. As a second example, consider data lineage. Lineage keeps track of all of the data that contributes to a particular result, that is, it constructs advice (metadata) for data, but the constructed advice does not change the data. As a third example, a profiling cross-cutting concern generates statistics (new data) about the data usage, but the advice itself does not change.

AOP can successfully model the kinds of cross-cutting concerns already researched in databases (e.g., time, provenance) and new kinds not yet researched. For instance, *versioned security* where a magazine subscriber has access to articles at the time the subscription was current even after the subscription has ended. Versioned security can be modeled as a temporal aspect tagging a security aspect in our framework, i.e., as meta-metadata. Recursively higher levels of advice (meta-meta-metadata) can also be modeled.

1.2 Pig Latin

In this paper we propose adapting AOP to Pig Latin [19, 29] to support cross-cutting data concerns. Pig Latin is a dataflow language and cloud computing platform for the analysis of massive datasets. Developed by researchers at Yahoo, Pig Latin is one of the first, and is (in our opinion) the best, of the emerging cloud computing languages for data analysis. Though relatively new, Pig Latin already has a strong user and development community.[1] Pig Latin is a typical "NoSQL" language. A NoSQL language replaces SQL, the *de facto* query language for databases, with a language that is better suited to programmers. As a dataflow language, Pig Latin is more amenable than declarative languages, like SQL, to aspect-oriented techniques. A Pig Latin program is a sequence of statements. Each statement represents a transformation of some data.

Pig Latin currently has ***no support*** for cross-cutting data concerns. Users must resort to ad hoc techniques to support, for instance, temporal semantics for data. Snodgrass has pointed out the perils of relying on user good faith to correctly implement temporal semantics [36]. Often users will not know which cross-cutting concerns are present nor the semantics of each individual concern.

Pig Latin lacks many of the features found in other database languages, such as SQL. In the relational model, data is rigidly described by a fixed schema, Pig Latin, on the other hand, is schemaless. Pig Latin users dynamically load data into a query from text files or back-end databases. Pig Latin also lacks data modification operators, such as `INSERT`, `DELETE`, and `UPDATE`. Pig Latin data is created and maintained by other processes. Not surprisingly, Pig Latin also has no data constraint specifications. All constraints are maintained by other processes. Finally, the Pig Latin data model supports sets and bags, as well as tuples, i.e., it is a non-first normal form data model. So while sharing some commonalities with other database query languages, Pig Latin is different, over and above the cloud computing framework (Hadoop) that supports its back-end.

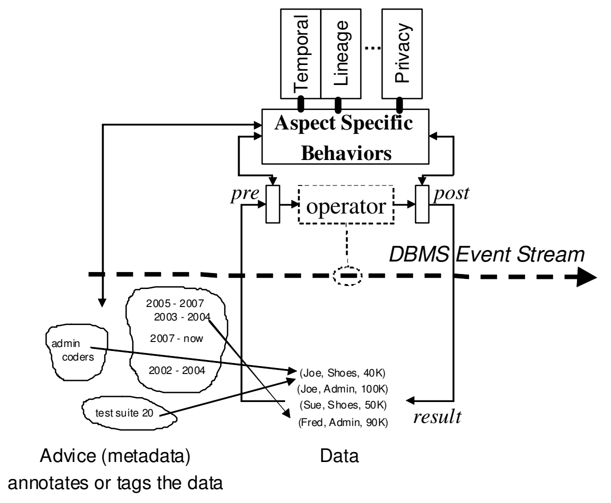

Figure 2: Opening Pig Latin programs to AOD

Figure 2 illustrates the role that a data aspect weaver has to fulfill in the evaluation of a Pig Latin program. In the figure, "Data" is annotated or tagged with "Advice," which is metadata from a cross-cutting concern such as privacy. Over time, a stream of Pig Latin "operators," e.g., a join, are evaluated. When the operator is executed, the advice becomes active. Associated with each kind of advice are "pre," "post," and "intra" advice operators that kick in before, during, and after evaluation of the operation. These operators are specified in a code module which is plugged into the Pig Latin evaluation engine. For example, suppose that Pig Latin evaluates a join. For each pair of tuples in the join with temporal advice, a pre-join operator is called in the temporal module prior to joining, the intra-join operator is called as part of the join, and a post-join operator is called after the join. For temporal advice the pre- and post-joins are no-ops (no action is taken) and the intra-join is temporal intersection. The operations are specific to each kind of cross-cutting concern. All of the advice-specific operators are specified in code that is called at the appropriate time during program evaluation. The plug-in module for each kind of advice consists of these operations.

Previously we described how to model data aspects in a relational database [14], and in the relational algebra [12]. This paper shares a common motivation with our previous work, but in this paper we focus on Pig Latin, which has a different data model and query language. The main contribution of this paper is an extension of each Pig Latin transformation to support data aspects.

This paper is organized as follows. The next section develops a motivating example. After that, data aspects are developed in greater detail. The paper then presents aspect-oriented Pig Latin. The final sections cover related work and summarize the paper.

[1] `http://hadoop.apache.org/pig/`

Subscribers
(Name, City, Amt, Id)
(Maya, Logan, $20, 1)
(Jose, Logan, $15, 2)
(Knut, Ogden, $20, 3)

Table 1: Some data about subscribers to Magazine.com

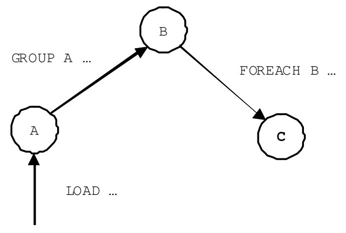

Figure 3: Dataflow in the simple program

2. MOTIVATION

Assume that Magazine.com stores data about its subscribers in a collection of Pig relations. A Pig relation is a bag of tuples, similar to a table in an SQL database. Each tuple is an ordered list of fields. Each field is a piece of data. Unlike an SQL table, not all tuples have to have the same number of fields. Moreover, Pig relations can have values that are themselves tuples, bags, or maps, something that is not allowed in a relational database. A portion of the data, the **Subscribers** relation, is shown in Table 1. Each tuple in Subscriber records, in order, a name (**Name**), city (**City**), subscription amount (**Amt**), and a tuple identifier (**Id**).

2.1 Pig Latin

Magazine.com would like to count the subscribers per city. The following Pig Latin program computes the desired count.

```
A = LOAD 'subscribers' USING PigStorage()
      AS (name: chararray, city: chararray,
          amount: int);
B = GROUP A BY city;
C = FOREACH B GENERATE city, COUNT(B.name);
DUMP C;
```

The program has four statements. The first statement loads the data, and gives a name and a type to each field within a tuple. The statement also establishes the **Subscribers** relation as the data node `A`. A grouping transformation is applied to the data in node `A` to produce node `B`. The data is grouped into bags by value as shown in Table 2. The data in node `B` is then processed to generate the name and count for each city as shown in Table 3. The final statement, `DUMP`, displays the data accumulated at node `C`.

This program has a very simple dataflow, with only three nodes. To evaluate the program, Pig Latin first constructs a representation of the dataflow as illustrated in Figure 3. Next it applies query optimization rules to optimize the data flow (for instance the `FOREACH` transformation could be combined with the `GROUP` transformation to generate only the needed fields while grouping). Only when the `DUMP` statement is parsed is the optimized dataflow program evaluated using Hadoop, that is, the program is transformed to map-reduce constructs and executed in parallel.

2.2 Cross-cutting Data Concerns

On-line magazines earn revenue by restricting content to paid subscribers. *Security* enforces the restriction. Each subscriber should

B
(Logan, {(Maya, Logan, $20), (Jose, Logan, $15)})
(Ogden, {(Knut, Ogden, $20)})

Table 2: Subscribers grouped by city

C
(Logan, 2)
(Ogden, 1)

Table 3: The count of subscribers

be able to see their own data, but not that of others. Subscribers complain that once their subscription ends, they are no longer able to see the content to which they once subscribed, but they should be able to do so. Magazine.com decides to support both security and *versioned security*, whereby subscribers still have access to content as of the time when they subscribed. To help the programmers implement the system, Magazine.com also decides that it is important to support *lineage* in query evaluation. Lineage keeps track of which facts were used to produce a result, thereby helping programmers understand how the query produced a particular result.

To accommodate the new requirements, which are all cross-cutting concerns, the designers need to add new data and functionality to their existing database and its applications. Ideally, the designers will be able to add without changing a line of existing Pig Latin programs.

2.3 Aspect-oriented Pig Latin Data

In an aspect-oriented approach, the database designers "tag" data in the database with advice, creating aspects. The tagging could be at different levels, i.e., in the Pig Latin data model, the tagged data could be an *attribute value*, a *tuple*, or a *relation*. We focus on *tuple-* and *relation-tagging* in this paper. The advice that tags a tuple is assumed to pertain to all of the attribute values within that tuple, and for a relation, the advice applies to all of the tuples in the relation. Relation-tagging is useful for establishing default advice for each tuple in the relation.

Though aspects are developed independently, more than one kind of advice can tag a tuple or relation, for instance a tuple could be tagged with both lineage and security advice. The advice can be combined into a single *perspective* [13], or remain independent. Finally, since the advice is data, it too can be advised by *meta-metadata*, i.e., metadata is to data as meta-metadata is to metadata.

Several data cuts are concretely represented in Table 4 and Table 5 which extend and refine the Magazine.com database example of the previous section. Each advice tuple is prefixed with a "perspective id" (**Per**), which is the first field in each advice tuple. The first tuple of **Security Advice** has a **Per** of `A`. A data cut is a pairing of a tuple id with an advice id. Table 4 shows the aspected **Subscribers** relation. The Data Cuts relation weaves advice to data identified by the **Id** column. This tagging scheme is repeated for the meta-metadata (**Temporal Meta Advice** and **Metadata Data Cuts**). Each subscriber is tagged by a security aspect that records the security on the tuple and a lineage aspect that denotes how the tuple was constructed. Initially, the lineage is just the identifier of the tuple itself. The security is a partial order from the lowest levels (`Paid` and `Lapsed`) to the top level (`DBA`). Only paid subscribers have access to the content. The meta-metadata records when the security advice is current. Jose was a paid subscriber from `2007` to `2008` at which time his subscription lapsed. If the data is rolled back to its state current in `2007`, Jose should have access to the content of the site. Said differently, `Jose` paid for the `2007` to `2008` content and therefore should have access to that data by setting his content perspective to some time in that range. An advice tuple shaded in grey denotes default advice, that is, data advised by relation-tagging. The default temporal advice starts in `2006` when the site began.

Table 5 extends the database with an aspected **Personal Info** relation that records personal information about each subscriber. By default, only the `DBA` has access to this data.

2.4 Aspect-Oriented Pig Latin Programs

Advice is involved whenever data is used in a query. For instance, suppose that two tuples are to be joined. Sequenced temporal semantics permits the tuples to be joined *only at the times they both existed*. For instance, the tuples with **Id** `1` and `2` in Table 4 can be joined only at times `2007-now` since tuple `2` was not in the database in `2006`.

An advice's behavior is woven into the evaluation of a Pig Latin program as shown in Figure 4. In Figure 4(a), the typical dataflow as depicted. Data at a node R is transformed to that at node X. Figure 4(b) shows a Pig Latin pattern or template that will replace the transformation of Figure 4(a). In the aspected case, the relation at node R consists of three components: a data relation, R_D, an advice relation, R_A, and a data cuts relation, R_C. Without loss of generality we focus on a single kind of advice, more generally there would be several advice relations. Each of these relations must be transformed to create the three components of the result data node: X_D, X_A, and relation, X_C.

Another way to conceptualize the weaving is to imagine a Pig Latin program evaluated simultaneously at two levels: the *data level* and the *advice level* (the cuts attach the advice to the data). At the data level, the Pig Latin program proceeds as written by evaluating each transformation on the data. The weaving needs to add transformations at the advice level to the dataflow. There are three basic patterns for the dataflow at the advice level.

1. Single tuple - Pig Latin transformations that process a single relation tuple-by-tuple have no special operations for advice. Security, privacy, data quality measures, etc. already annotate and describe the data. The advice sticks to the data through the transformation. For example a transformation to project the values in a column retains the advice for each value.

2. Pair of tuples - Two Pig Latin relations can be related by processing pairs of tuples, one chosen from each relation. For a pair of tuples, the advice that annotates each tuple must be processed together. For instance, a join operation will need to join the advice for a pair of tuples while joining the pair.

3. Multiple tuples - Some Pig Latin transformations relate many tuples in a single relation, for instance, when grouping a relation, the advice for all of the tuples in the group must be processed.

The weaving also has to *synchronize* the advice and data levels after each transformation since the advice level can impact the data level (and vice-versa). For example, in the temporal sequenced join discussed previously, tuples at the data level join only if they also join at the advice level.

In the next section we develop a specific template for each kind of Pig Latin transformation. But in general, the weaving is a Pig Latin program modification whereby each statement in a program is replaced by a sequence of statements constructed by instantiating a template for the transformation. This strategy can be extended

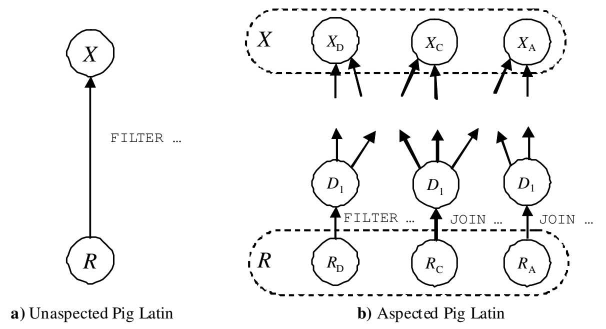

Figure 4: Dataflow in the simple program

to additional levels of advice, e.g., meta-metadata. The pattern is repeatedly applied for each level.

3. ASPECT-ORIENTED PIG LATIN

This section describes changes to Pig Latin to support aspect-oriented data. Recall that each kind of aspect (e.g., security) enforces a semantics on the use of the data. All uses must obey that semantics. We model the bulk of Pig Latin transformations, showing how each is redefined to support data aspects. Each modification is described in terms of a pattern or template. The template is applied to rewrite the corresponding transformation in a Pig Latin program at the data and advice levels. Each transformation is redefined using (non-aspect-oriented) Pig Latin to illustrate that Pig itself can be used to become aspect-oriented. A key optimization to the basic strategy, which we call *advice inlining*, is presented in Section 3.5. Only data aspects are initially considered; in Section 3.6 program aspects are introduced.

We consider three broad categories of Pig Latin transformations: single, paired, and multiple tuple transformations. We first model single tuple transformations which involve only one tuple at a time and are generally simpler than the other cases.

3.1 Single Tuple

The single tuple transformations are `FILTER`, `FOREACH`, `SPLIT`, `SAMPLE`, `LOAD`, `DUMP`, and `STORE`. We discuss the `FILTER` in detail and only breifly present the other transformations.

3.1.1 FILTER

We first describe `FILTER` and then present an detailed example of the transformation using aspected data.

The `FILTER` transformation selects tuples from a data node that meet some condition, P.

X = `FILTER` R `ON` P;

The aspected-oriented transformation, `TFILTER`$_{AO}$, first applies a `FILTER` at the data level. As the `FILTER` may remove some tuples, the data cuts and advice should then be synchronized with the data, removing extraneous advice, an operation that we call `TRIM`$_{AO}$.

Figure 5 illustrates the basic pattern for `FILTER`. The `TRIM`$_{AO}$ pattern to the right of the figure should be repeated for each level of advice. The Pig Latin code template for `FILTER`$_{AO}$ is given below, with comments enclosed within '`/* */`'.

```
/* Filter the data */
```
X_D = R_D `ON` P;

Subscribers
(Name, City, Amt, Id)
(Maya, Logan, \$20, 1)
(Jose, Logan, \$15, 2)
(Knut, Ogden, \$20, 3)

Data Cuts
(Id, Per)
(1, A)
(2, B)
(2, C)
(3, D)

Security Advice
(Per, Sec)
(A, Paid)
(B, Paid)
(C, Lapsed)
(D, Lapsed)

Lineage Advice
(Per, Lin)
(A, {1})
(B, {2})
(C, {2})
(D, {3})

Metadata Data Cuts
(Per, MetaPer)
(B, X)
(C, Y)
(A, Z)
(D, Z)

Temporal Meta Advice
(MetaPer, Start, End)
(X, 2007, 2008)
(Y, 2009, now)
(Z, 2006, now)

Table 4: Aspected `Subscribers`

Personal Info
(Name, City, Amt, Id)
(Maya, maya@aol.com, 5)
(Jose, jose@aol.com, 6)
(Knut, knut@aol.com, 7)

Data Cuts
(Id, Per)
(5, E)
(6, F)
(7, G)

Security Advice
(Per, Sec)
(E, DBA)
(F, DBA)
(G, DBA)

Lineage Advice
(Per, Lin)
(E, {5})
(F, {6})
(G, {7})

Table 5: Aspected personal information about subscribers

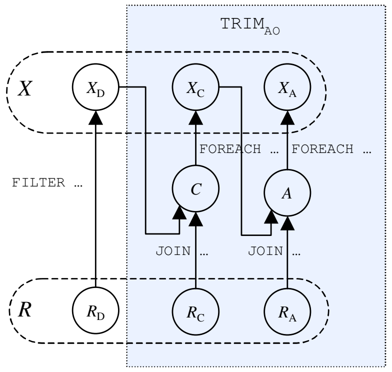

Figure 5: The template for FILTER$_{AO}$

```
/* TRIM the advice level */
/* Remove extraneous cuts*/
$C =$ JOIN $R_C$ BY id, $X_D$ BY id;
$X_C =$ FOREACH $C$ GENERATE $C$.id, $C$.per;

/* Remove extraneous advice */
$A =$ JOIN $R_A$ BY per, $X_C$ BY per;
$X_A =$ FOREACH $A$ GENERATE $A$.per, $A$.metadata;
```

As an example, consider a query to filter subscribers below \$20. At the data level, `Maya` and `Knut` pass the filter, but Jose is filtered. The result is shown in Table 6. In the table, extraneous, inert advice and data cuts that could be trimmed are highlighted in gray. TRIM$_{AO}$ will clean up this extra advice and synchronize the advice level to the data level, but the extraneous advice is harmless (except for occupying space) and can be left in place leading to the alternative, cheaper plan shown in Figure 6.

3.1.2 FOREACH

The FOREACH projects only specified fields, $f_1, \ldots, f_n$, into the result.

```
$X =$ FOREACH $R$ GENERATE $f_1,\ldots,f_n$;
```

As all the tuples are retained, the data cuts and advice are unchanged, and so the template for FOREACH$_{AO}$ is simple.

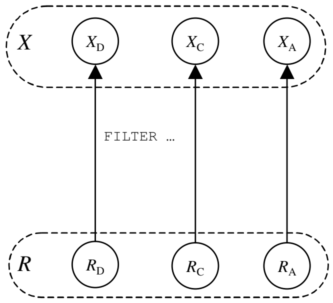

Figure 6: An alternative, cheaper template for FILTER$_{AO}$

```
/* GENERATE the data */
$X_D =$ FOREACH $R_D$ GENERATE $f_1,\ldots,f_n$;

/* Repeat rest of pattern for each level of advice */
$X_C = R_C$;
$X_A = R_A$;
```

The template is illustrated in Figure 7. As an example, consider generating subscriber cities. Each city is generated along with all of the advice and meta advice.

3.1.3 Split and Sample

A SPLIT transformation partitions a relation into n relations for parallel processing. The split is based on conditions $c_1, \ldots, c_n$ where each condition is a predicate involving field values.

```
SPLIT $R$ INTO $X_1$ IF $c_1,\ldots,X_n$ IF $c_n$;
```

A SAMPLE transformation is chooses a random sampling of a relation. It is used to estimate results. The *sample_size* is a percentage of the size of the relation, e.g., 0.01 would represent 1%.

```
$X =$ SAMPLE $R$ sample_size;
```

The aspect-oriented versions of these transformations are similar to FILTER$_{AO}$. They apply the transformation at the data level and then remove extraneous data cuts and advice using TRIM$_{AO}$, or alternatively, leave the cuts and advice unchanged since extraneous cuts and advice are harmless.

Subscribers	Data Cuts	Security Advice	Lineage Advice	Metadata Data Cuts	Temporal Meta Advice
(Name, City, Amt, Id)	(Id, Per)	(Per, Sec)	(Per, Lin)	(Per, MetaPer)	(MetaPer, Start, End)
(Maya, Logan, $20, 1)	(1, A)	(A, Paid)	(A, {1})	(B, X)	(X, 2007, 2008)
(Knut, Ogden, $20, 3)	(2, B)	(B, Paid)	(B, {2})	(C, Y)	(Y, 2009, now)
	(2, C)	(C, Lapsed)	(C, {2})	(A, Z)	(Z, 2006, now)
	(3, D)	(D, Lapsed)	(D, {3})	(D, Z)	

Table 6: `Subscribers` that paid $20 or more for their subscription, the cells shaded gray can be trimmed

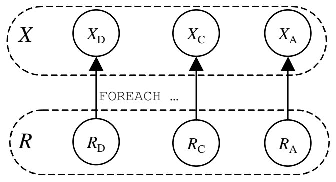

Figure 7: The template for FOREACH$_{AO}$

3.1.4 Load, Store, and Dump

The LOAD transformation loads a relation from disk into memory, STORE stores an in-memory relation to disk, and DUMP displays a relation. The aspect-oriented versions of these transformations must be trivially augmented to deal with three relations (the data, the data cuts, and the advice) rather than a single relation.

3.1.5 User-defined Functions and MapReduce

User-defined functions (UDFs) implemented in Java can be added to a Pig Latin dataflow. As of Pig version 0.10.0, a MAPREDUCE transformation can run a Map-Reduce job, also coded in Java. In both cases, each tuple in a relation is streamed through an arbitrary program, and output tuples are collected. Our current design only supports data aspects in Pig Latin, not in Java. Hence users must rewrite the UDF and Map-Reduce jobs to handle aspected data. For Aspect-oriented Pig Latin, we currently give the user the option to join the data relation to the cuts and advice relations prior to streaming the tuples, or to simply stream the data relation (potentially violating the semantics of the data concerns involved). We discuss alternative strategies in future work.

3.2 Pair of Tuple Transformations

Paired tuple transformations involve a pair of tuples and generally invoke an advice-specific operation to process the advice at the advice level.

3.2.1 Joins

Pig Latin has several kinds of joins: cross, replicated, inner, outer, skewed, and merge. Additionally Pig Latin has a COGROUP transformation that groups tuples that would join. Semantically they are all a variant of an equi-join, where two tables are joined on the values of one or more fields being equivalent.

X = JOIN R BY f_R, S BY f_S;

For an aspect-oriented join, JOIN$_{AO}$, advice constrains the join. If two tuples potentially join at the data level, their advice also needs to "join" at the advice level (the meaning of a join depends on the kind of advice). For instance two tuples only join when their temporal advice overlaps (i.e., the tuples exist at the same time).

Figure 8 shows the template for JOIN$_{AO}$. First each relation involved in the join is merged by joining the data to the data cuts and then to advice (the joins must be repeated for each metadata level).

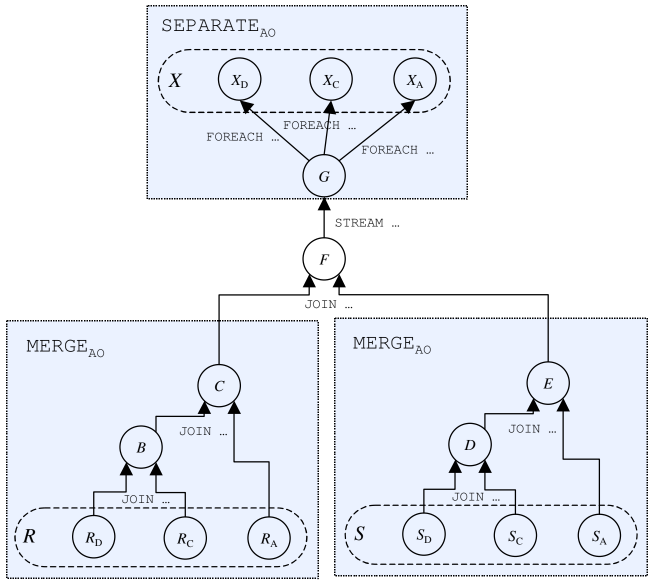

Figure 8: A JOIN$_{AO}$ first merges both relations, then joins, and finally separates

We call this operation MERGE$_{AO}$. Next, the two merged relations are joined, effectively joining at the data level. Then, the result is streamed through an advice-specific join operation (described in detail below). Finally, the merged relations are separated into data, cuts, and advice using the FOREACH transformation. We call the separation step, SEPARATE$_{AO}$.

At the advice level, advice is streamed through an advice-specific join (ADVICE-JOIN). This user-defined function "joins" the advice for a tuple using an advice-specific technique. Example advice-specific joins are listed below. These examples assume that the last four (or two) fields in a tuple are the advice pairs to be tested, and that the data, ids, and perspectives are called "*rest*."

- Temporal advice — Computes the temporal join for pairs of time periods, i.e., the time when the periods overlap.

  temporal-join((*rest*, t, u, v, w)) =
  {(*rest*, **max**(t, v,), **min**(u, w))}

- Lineage advice — Lineage x always joins with lineage y and manufactures new advice that is the union of the previous lineage.

  lineage-join((*rest*, x, y)) = {(*rest*, $x \bigcup y$))}

- Security advice — A partial order join is performed by keeping the most private group.

security-join((*rest*, x, y)) = {(*rest*, x), (*rest*, **lca**(x,y))}

The ADVICE-JOIN also manufactures a new data cut identifier and advice reference.

As an example, consider the JOIN$_{AO}$ of **Subscribers** with **Personal Info**. First, the two relations are individually merged. Next, the join is performed at the data level, resulting in the relation shown in Table 7. At this point, the data level has been joined, but the advice level has not. After the advice-specific joins are performed and the relation is separated, Table 8 results. In this example, no tuples were removed from the join due to incompatible or mismatching advice, but some of the advice has been trimmed, For instance the security advice joins only at the level of the DBA which is the least common ancestor of each pair of security advice values. The meta-metadata (**Temporal Meta Advice**) joins only on the interesection of the pair of time intervals. In the final result, the data cuts identifiers (perspectives) are composed values, manufactured from the underlying identifiers (perspectives).

3.2.2 *Union*

In a UNION transformation, the tuples in each relation are put into a single bag. The union does not eliminate duplicates.

X = UNION R, S;

The aspect-oriented union, UNION$_{AO}$, similarly combines the advice and data cuts (assuming that the ids and references are disjoint).

```
$X_D$ = UNION $R_D$, $S_D$;
/* Repeat for each level of advice */
$X_C$ = UNION $R_C$, $S_C$;
$X_A$ = UNION $R_A$, $S_A$;
```

3.3 Multiple Tuple

Multiple tuple transformations involve groups or collections of tuples. The advice for a group needs to be processed as a group.

3.3.1 *Grouping*

Grouping is important when computing aggregates. Pig Latin has a GROUP transformation that groups tuples on fields, $f_1, \ldots, f_n$.

X = GROUP R USING $f_1, \ldots, f_n$;

Advice constrains the grouping. Two tuples potentially group only if their advice also groups. For instance two tuples are in the same group only when their temporal advice overlaps (i.e., the tuples exist at the same time). The template for GROUP$_{AO}$ is sketched in Figure 9. First, the data is merged with the cuts and advice, using the MERGE$_{AO}$ pattern. Next, at the data level, the data is grouped. Then, the data is streamed through an advice-specific grouping operator, ADVICE-GROUP, to compute the groups for the advice. The semantics of this operator depends on the kind of advice. The input to this operator is a set of advice values (the advice for all of the group members). The output is a refined set of advice.

- Temporal advice — Compute membership constant periods, that is those intervals of time for which group membership does not change.

  temporal-group(T)
  $$= \{(t,u) \mid (t,\_) \in T \wedge (\_,u) \in T \wedge \neg(\exists(w,\_) \in T) \vee \exists(\_,w) \in T[t < w < u])\}$$

- Lineage advice — Lineage forms a set of the ids of all of the tuples in the group.

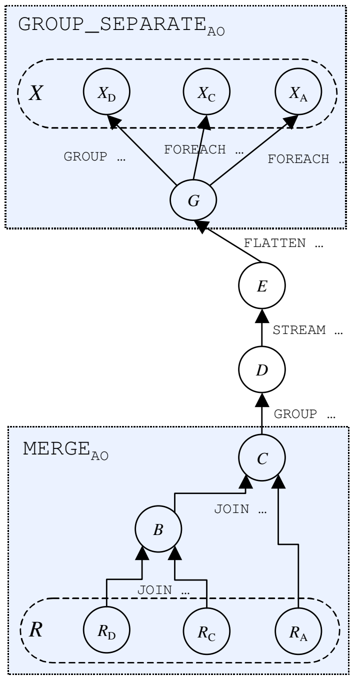

Figure 9: The template for GROUP$_{AO}$

  lineage-group(T) = $\{t.\texttt{id} \mid t \in T\}$

- Security advice — Each level in the hierarchy is its own group.

  security-group(T) = T

Finally, the grouped data must be separated into cuts and advice by a GROUP_SEPARATE$_{AO}$ pattern.

As an example, consider grouping the **Subscribers** relation using the City field. To simplify this example we assume a single kind of advice: security advice. The result is shown in Table 9. Each city will end up in a separate group. But because the cities have different advice, they will be further split into more groups. As part of the aspect-specific grouping, new advice corresponding to each group is manufactured.

3.3.2 *Distinct*

The DISTINCT transformation eliminates duplicate tuples from a relation.

X = DISTINCT R;

For aspect-oriented distinct, DISTINCT$_{AO}$, when duplicates of a tuple are eliminated, the data cuts to the duplicates must be changed to attach to the tuple that was not eliminated. The duplicate elimination does not *coalesce*, that is, it does not eliminate or reduce overlapping or redundant advice.

The template for DISTINCT$_{AO}$ is shown in Figure 10. The cuts are first merged with the data (the advice does not have to be since it the distinct is applied at the data level, not the advice level). Next, the tuples are grouped on all of the data fields, yielding distinct groups of tuples. Finally, the cuts and advice are separated and each data tuple gets a new identified. Since the pattern involves some complexities, we give a code template below.

Data	**Data Cuts**	**Security Advice**	**Lineage Advice**	**Metadata Data Cuts**	Temporal **Meta Advice**
Name, . . ., Id, . . ., Id	**Id, Per, Id, Per**	**Per, Sec, Per, Sec**	**Per, Lin, Per, Lin**	**Per, M, Per, M**	**M, Start, End, M, Start, End**
Maya, ..., 1, ..., 5	1, A, 5, E	A, Paid, E, DBA	A, {1}, E, {5}	A, Z, E, Z	Z, 2006, now, Z, 2006, now
Jose, ..., 2, ..., 6	2, B, 6, F	B, Paid, F, DBA	B, {2}, F, {6}	B, Y, F, Z	Y, 2009, now, Z, 2006, now
Jose, ..., 2, ..., 6	2, C, 6, F	C, Lapsed, F, DBA	C, {2}, F, {6}	C, X, F, Z	X, 2007, 2008, Z, 2006, now
Knut, ..., 3, ..., 7	3, D, 7, G	D, Lapsed, G, DBA	D, {3}, G, {7}	D, Z, G, Z	Z, 2006, now, Z, 2006, now

Table 7: Subscribers joined with Personal Info prior to advice-specific joins

Data	**Data Cuts**	**Security Advice**	**Lineage Advice**	**Metadata Data Cuts**	**Temporal Meta Advice**
(Name, . . ., Id)	**(Id, Per)**	**(Per, Sec)**	**(Per, Lin)**	**(Per, MetaPer**	**(MetaPer, Start, End)**
(Maya, ..., 1.5)	(1.5, A.E)	(A.E, DBA)	(A.E, {1,5})	(A.E, Z)	(X, 2007, 2008)
(Jose, ..., 2.6)	(2.6, B.F)	(B.F, DBA)	(B.F, {2,6})	(B.F, Y)	(Y, 2009, now)
(Jose, ..., 2.6)	(2.6, C.F)	(C.F, DBA)	(C.F, {2,6})	(C.F, X)	(X, 2007, 2008)
(Knut, ..., 3.7)	(3.7, D.G)	(D.G, DBA)	(D.G, {3,7})	(D.G, Z)	(Z, 2006, now)

Table 8: Subscribers joined with Personal Info after advice-specific join and SEPARATE$_{AO}$

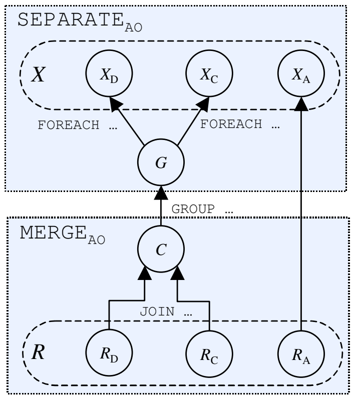

Figure 10: The template for DISTINCT$_{AO}$

```
/* Merge the data with the data cuts */
$C$ = JOIN $R_D$ BY id, $R_C$ BY id;

/* Group the duplicates */
$G$ = GROUP $C$ on all data fields;

/* Generate the data with a minimum cut Id */
$X_D$ = FOREACH $G$ GENERATE data fields, min($R_D$.id);

/* Generate the data cuts */
$X_C$ = FOREACH $G$ GENERATE , min($R_D$.id), $R_C$.ref;

/* Advice is not changed */
$X_A = R_A$;
```

As an example, consider computing distinct cities. First the subscriber names are generated yielding three tuples. Next the DISTINCT transformation is applied, yielding two tuples in the result (Logan and Ogden). The advice for the Logan tuples remains as three distinct perspectives. The result is shown in Table 10.

3.4 The Example Revisited

We return to the example query of Section 2.1. Assume that we have a single temporal aspect. The aspect-oriented version of the program is given below.

```
A = LOAD$_{AO}$ 'subscribers' USING PigStorage()
    AS (name:chararray, city:chararray,
      amount:int, id:chararray);
B = GROUP$_{AO}$ A BY dept;
C = FOREACH B GENERATE$_{AO}$ dept, COUNT(B.name);
DUMP$_{AO}$ C;
```

The aspect-oriented behavior is woven into the program using the templates described in this section, yielding the following Pig Latin program.

```
AD = LOAD 'subscribers.data' USING PigStorage()
        AS (name:chararray, city:chararray,
          amount:int, id:chararray);
AC = LOAD 'subscribers.cuts' USING PigStorage()
        AS (id:chararray, per:chararray);
AA = LOAD 'subscribers.advice' USING PigStorage()
        AS (per:chararray, start:int, end:int);

/* Merge the data with the cuts and advice */
B = JOIN AD BY id, AC BY id;
C = JOIN B BY per, AA BY per;

/* Group on the data values */
D = GROUP C USING dept;

/* Stream through aspect-specific grouping */
E = STREAM D THROUGH TEMPORAL-GROUP;

/* Flatten it and regroup using the data and id */
F = FLATTEN E;
BD = GROUP F USING dept, id;

/* Generate the data cuts */
CC = FOREACH F GENERATE id, per;

/* Generate the advice */
CA = FOREACH F GENERATE per, start, end;

/* Generate the result */
CD = FOREACH BD GENERATE dept, COUNT(BD.name);
DUMP CD;
```

Subscribers	Data Cuts	Security Advice
(Logan, 11 {(Maya, Logan, $20, 1)}) (Jose, Logan, $15, 2)})	(11, J)	(J, Paid)
(Logan, 12, {(Jose, Logan, $15, 2)})	(12, H)	(H, Lapsed)
(Ogden, 13, {(Knut, Ogden, $20, 3)})	(13, I)	(I, Paid)

Table 9: Grouped subscribers

Cities	Data Cuts	Security Advice	Lineage Advice	Metadata Data Cuts	Temporal Meta Advice
(City, Id)	(Id, Per)	(Per, Sec)	(Per, Lin)	(Per, MetaPer)	(MetaPer, Start, End)
(Logan, 1)	(1, A)	(A, Paid)	(A, 1)	(A, X)	(X, 2007, 2008)
(Ogden, 3)	(1, B)	(B, Paid)	(B, 2)	(B, Y)	(Y, 2009, now)
	(1, C)	(C, Lapsed)	(C, 3)	(C, Z)	(Z, 2006, now)
	(3, D)	(D, Lapsed)	(D, 4)	(D, Z)	

Table 10: Distinct cities

3.5 Optimizing by Advice In-lining

Maintaining separate data, cuts, and advice relations throughout the evaluation of a Pig Latin program can incur many invocations of TRIM$_{AO}$, MERGE$_{AO}$, and SEPARATE$_{AO}$. Since they involve JOINs, which are expensive, a more optimal strategy is to *in-line* the advice at the start of a query, and separate at the end. The in-lining removes data cuts by attaching the advice directly to the data. The following template can be used to in-line advice for an aspected relation. Let R_D be the data relation, R_C by the cuts relation, and $R_{A_1}, \ldots, R_{A_n}$ be n advice relations. The template should be applied for each level of advice.

```
$I_1 =$ JOIN $R_{A_1}$ by per, $R_{A_2}$ by per;
$I_2 =$ JOIN $I_1$ by per, $R_{A_3}$ by per;
...
$I_n =$ JOIN $I_{n-1}$ by per, $R_{A_n}$ by per;
$I =$ JOIN $I_n$ by per, $R_C$ by per;
$R_I =$ COGROUP $R_D$ by id, $I$ by id;
```

The first n steps join each kind of advice to form a combined perspective. Next, the perspective is joined to the data cuts. Finally, the data is co-grouped with individual perspectives (all the perspectives that would join with the advice).

An example of advice in-lining in given in Table 11.

The benefit of in-lining is that in the templates presented in Sections 3.1 (Figure 5) to 3.3 (Figure 10), only the operations **not** shaded in gray need to be performed on the in-lined data and advice. The cost of in-lining is incurred once for the query. Prior to a DUMP or STORE the data must be separated as well.

3.6 Program Aspects

A Pig Latin program can also be aspected. A program aspect represents a constraint on the relations that are evaluated. Suppose that a program involves a relation, $[R_D, R_C, R_A]$, and is aspected by a perspective consisting solely of advice, P_A. Then the program aspect transformation constrains R_D to tuples that have advice consistent with the perspective prior to evaluating the program. It does so as follows.

```
/* First relate all of the advice, CROSS is
  the Cartesian product transformation.  */
$B =$ CROSS $R_A$, $P_A$;

/* Generate new advice */
$X_A =$ STREAM $B$ THROUGH ADVICE-PROGRAM-ASPECT;

/* Use new advice to remove extraneous data
  cuts */
$C =$ COGROUP $R_C$ BY ref, $X_A$ BY ref;
$X_C =$ FOREACH $C$ GENERATE $C$.id, $C$.ref;

/* Use new data cuts to remove extran. tuples */
$D =$ COGROUP $R_D$ BY id, $X_C$ BY id;
$X_D =$ FOREACH $D$ GENERATE $D$.data, $D$.id;
```

Each advice tuple is passed through the advice-specific stream, which leaves the tuple unchanged, trims the tuple, or removes it.

3.7 Complexity Analysis

The increased modeling power of aspect-oriented data comes with an increased cost. In this section we analyze the worst-case time complexity, assuming that all of the aspect-oriented transformations are implemented in Pig Latin (some advice-specific behaviors are implemented as user-defined functions). Let D be the size of each data relation, C be the size of a data cuts relation, and A be the size of an advice relation. Typically A will be much smaller than D, and if there is a lot of default advice, C will also be much smaller than D. Finally, let z be the number of different kinds of advice, e.g., z is three for the examples in this paper, and let k be the number of levels of advice, e.g., in our examples, k is 2. We assume that all binary operations, i.e., the various kinds of join, cost $O(n * m)$ where n and m are the size of the operands, and unary operations, i.e., filtering, sampling, generating, and grouping, cost $O(n)$. Though this assumption overestimates the join cost, which in practice (e.g., in a good hash join) can be nearly linear, the assumption is appropriate for our complexity analysis.

To determine the cost of each aspect-oriented transformation, we summed the cost of every Pig Latin transformation in a template. For example a FILTER$_{AO}$, costs 1 FILTER $+k\dot{(}2$JOINs and 2 FOREACHs), which yields a total cost of $O(D) + O(kDC + kzAC) + O(kC + kzA)$, or $O(kDC + kzAC)$ by simplifying the equation. The analysis states that the cost of FILTER$_{AO}$ is dominated by the cost of the $k\dot{2}$ JOINs, which concurs with the general rule of thumb, that the cost of joins dominates query cost.

The analysis is summarized in Table 12. Most of the opera-

Subscribers	Advice
Name, City, Amt, Id	Id, Per, Sec, Lin, {Per, MetaPer, Start, End}
`(Maya, Logan, $20, 1)`	`{(1, A, Paid, {1}, {(A, Z, 2006, now)})}`
`(Jose, Logan, $15, 2)`	`{(2, B, Paid, {2}, {(B, X, 2007, 2008)}),`
	`(2, C, Lapsed, {2}, {(C, Y, 2009, now)})}`
`(Knut, Ogden, $20, 3)`	`{(3, D, Lapsed, {3}, I, {(D, Z, 2006, now)})}`

Table 11: Advice in-lined `Subscribers`

Transform	Pig Latin	Aspect-oriented Pig Latin
`FILTER`	$O(D)$	$O(D+ \texttt{TRIM}_{AO})$
`FOREACH`	$O(D)$	$O(D + kA + kC)$
`DISTINCT`	$O(D)$	$O(D+ \texttt{MERGE}_{AO} + \texttt{SEPARATE}_{AO})$
`JOIN`	$O(D^2)$	$O(D^2+ \texttt{MERGE}_{AO} + \texttt{SEPARATE}_{AO})$
`GROUP`	$O(D)$	$O(kD+ \texttt{MERGE}_{AO} + \texttt{SEPARATE}_{AO})$
`UNION`	$O(D^2)$	$O(D^2 + kC^2 + kzA^2)$
`SPLIT/SAMPLE`	$O(D)$	$O(k(D+ \texttt{TRIM}_{AO}))$
`LOAD/STORE`	$O(D)$	$O(D + kz(C + A))$
$\texttt{TRIM}_{AO}$	-	$O(D(z(CA^k)))$
$\texttt{MERGE}_{AO}$	-	$O(D(z(CA^k)))$
$\texttt{SEPARATE}_{AO}$	-	$O(D + z(kCA))$
Program aspects	-	$O(kDC + kzAC)$

Table 12: Complexity Analysis

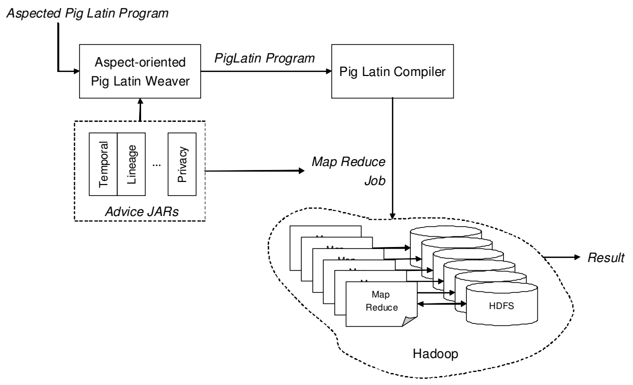

Figure 11: An overview of the implementation architecture

tions include only the additional cost of processing the cuts and advice, but five operations are much more expensive: $\texttt{FILTER}_{AO}$, $\texttt{JOIN}_{AO}$, $\texttt{GROUP}_{AO}$, $\texttt{SPLIT}_{AO}$, and $\texttt{SAMPLE}_{AO}$. We consider each in turn. $\texttt{FILTER}_{AO}$, increases the cost by trimming extraneous cuts and advice (those that have been filtered from the relation). But as we pointed out in Section 3.1.1 the extraneous cuts and advice are harmless and do not need to be removed (other than for consistency), lowering the cost to $O(D)$. A similar speedup applies to $\texttt{SPLIT}_{AO}$ and $\texttt{SAMPLE}_{AO}$. $\texttt{JOIN}_{AO}$, is expensive because the data has to be joined to the advice and the data cuts for each join. This adds the cost of a merge and separate phase. Finally $\texttt{GROUP}_{AO}$ is inherently more expensive than `GROUP`, because data must be grouped by both data and metadata, increasing both the number of groups and the cost of computing each group.

Advice in-lining can lower the cost of many of the operations since the relations do not have to be merged, separated, or trimmed. The cost of in-lining is an additional merge and separate phase.

4. IMPLEMENTATION

Apache Pig is an open source, Java implementation of Pig Latin.[2] We modified the source code to implement Aspect-oriented Pig Latin. Our modifications and an experimental reproducibility package can be found at the project's website.[3] The modified architecture is given in Figure 11. A Pig Latin program is input to the Aspect-oriented Pig Latin Weaver. The weaver translates an aspected Pig Latin to a Pig Latin program by weaving the aspects into the code using the transformations discussed in Section 3. JARs that contain the advice-specific operations are also part of the transformation. That program is then compiled into a Map-Reduce job, which can be run on Hadoop (or on a local machine). The input data resides on the Hadoop Distributed File System (HDFS). As the Map-Reduce job evaluates it also calls advice-specific behavior in the JARs.

We implemented the data weaver by rewriting the Pig Latin parser (which is specified using ANTLR[4]). We also had to modify some run-time libraries to pass schema information within Hadoop. Normally a Map-Reduce job does not know the schema of the data, but since we need to sometimes find the advice data in a tuple each tuple needs to know the schema of the relation to which it belongs. So we modified the system to retain schema information.

The cost of weaving is trivial; the weaver takes a fraction of a second since most Pig Latin programs are tens of lines in length. To get an idea of the cost of evaluating a data analysis program that enforces metadata semantics, we designed an experiment to measure the cost of single-tuple and tuple-pair operations. We chose $\texttt{FILTER}_{AO}$ as a representative single-tuple operation, and $\texttt{JOIN}_{AO}$ for the tuple-pair operation. We generated three datasets of 5 million, 10 million, and 15 million tuples, respectively. We then aspected the tuples with "best-case" advice (each tuple is aspected by the same advice, so the advice is a single tuple) and "worst-case" advice (each tuple has different advice, so there are 5 million advice tuples for 5 million data tuples). We used only one kind of advice: temporal. For both cases the cuts relation has the same number of tuples as the data relation (so 5 million data cuts for 5 million data tuples). To mitigate the impact of parallel evaluation, we ran Hadoop on a single Linux machine with two Intel 686 2.66 GHZ chips, 3.5 GB of RAM, and mirrored 500 GB disks (RAID level 1). We tested using Java 1.6, Pig 0.10.0, and Hadoop 1.0.1. We ran each test five times and took the average cost. Times were measured using the "real" system time captured by Linux's time command. We measured the cost of loading and storing each dataset together with the operation. We then subtracted from the cost, the I/O time. Aspect-oriented I/O costs are approximately triple the non-aspected case in the experiment since advice and cuts relations must also be read and stored. In a long program with several transformations the I/O cost would be spread out over several operations.

Table 13 gives the result of the experiment. The times are given

[2]`http://pig.apache.org`
[3]`http://cs.usu.edu/~cdyreson/AOD`

[4]`http://www.antlr.org`

Transform	Pig Latin	Aspect-oriented Pig Latin		Advice In-lined
		best	worst	
LOAD/STORE				
5 million	22s (4.4)	48s (9.7)	67s (13.5)	-
10 million	37s (3.7)	88s (8.8)	114s (11.4)	-
15 million	52s (3.5)	112s (7.4)	178s (11.8)	-
FILTER - Filters on the condition "true"				
5 million	5s (1.0)	120s (24.0)	132s (26.4)	5s (1.0)
10 million	10s (1.0)	220s (22.0)	250s (25.0)	11s (1.1)
15 million	13s (0.9)	310s (20.6)	371s (24.7)	15s (1.0)
JOIN - Equi-join on data fields				
5 million	112s (22.4)	236s (47.2)	353s (70.6)	150s (30.0)
10 million	230s (23.0)	437s (43.7)	587s (58.7)	271s (27.1)
15 million	335s (22.3)	678s (45.2)	825s (55.0)	370s (24.6)

Table 13: Evaluation Experiment

in seconds and in parentheses, the throughput, which is measured as the number of seconds per million tuples (lower is better). While aspect-oriented Pig Latin programs cost more, they also do more; they enforce metadata semantics in data processing. Un-aspected Pig Latin programs do not observe such semantics. The big increase in cost is for $\texttt{FILTER}_{AO}$. This is because $\texttt{FILTER}_{AO}$ adds two joins to the filtering. But as we observed in Section 3.1.1, the joins can be deferred since they only remove "inert" data.

The table also shows the cost for the advice in-lining optimization on the "best" data case. For $\texttt{FILTER}_{AO}$, advice in-lining makes the tuples slightly larger, but performs the same filtering as the non-aspected case. For $\texttt{JOIN}_{AO}$, the non-aspected JOIN is coupled with a STREAM transformation to join advice. The advice in-lining operations do not show the cost of the in-lining.

5. RELATED WORK

There is a little previous research on support for manifold kinds of metadata in database management systems, though descriptive metadata has been studied in detail (c.f., [6, 10, 20]). Most closely related to this paper is the AUCQL language for querying different kinds of metadata in a semi-structured data model [13], which was later developed into a query language, MetaXQuery, for XML data [23, 24]. This paper in contrast focuses on Pig Latin.

The database research community has researched models and support for specific kinds of metadata, or in our terminology, specific kinds of aspects. One of the most important and most widely researched kinds is temporal. Temporal extensions of every data model exist, for instance, relational [35], object-oriented [34], and XML [17]. This paper generalizes the work in relational temporal databases by proposing an infrastructure that supports many kinds of advice, not just temporal advice. More specifically we extend tuple-timestamped models [22], whereby the temporal metadata modifies the entire tuple. Other tuple-level, relational model extensions to support security, privacy, probabilities, uncertainty, and reliability have been researched, but no general framework or infrastructure exists which can support all the disparate varieties, other than our own work [12, 14]. This paper makes two important novel contributions. First, we develop aspect-oriented programming for Pig Latin, which shares some operations in common with relational algebra, but has different transformations, such as grouping, distinct, sample, split, load, and store. Additionally, Pig Latin has a non-1NF data model, though in this paper we focused only on the 1NF aspects. The second contribution is extending the framework to meta-metadata as advice tagging advice.

There are several systems that have aspect-like support for combining different kinds of metadata. Mihaila et al. suggest annotating data with quality and reliability metadata and discuss how to query the data and metadata in combination [27]. The SPARCE system wraps or super-imposes a data model with a layer of metadata [28]. The metadata is active during queries to direct and constrain the search for desired information. Systems that provide mappings between metadata (schema) models are also becoming popular [4, 26]. Our approach differs from these systems by focusing on Pig Latin to support AOP, and by building a framework whereby the behavior of individual data aspects can be specified as "plug-in" components.

The information retrieval community has been very active in researching descriptive metadata, in particular metadata that is used to classify knowledge [37]. The Dublin Core is a commonly used classification standard [38]. Commercial and research systems [1] to manage (descriptive) metadata collections have been developed, as well as methods to automatically extract content-related metadata [21, 25]. The focus of the information retrieval research is on how to best use, manage, and collect metadata to describe data to improve search [18]. In contrast, our focus is on modeling data aspects which *impose* a semantics on the *use* of the data, i.e., they go beyond the simple, descriptive tagging of data.

Finally, our goal in this paper, consistent with AOP and unlike many of the above approaches, is to maximally reuse existing languages and systems. Hence we focus on using Pig Latin itself to support aspect-oriented data by weaving the support for cross-cutting concerns, expressed in Pig Latin, into Pig Latin programs.

6. CONCLUSIONS

Cloud computing data analysis platforms like Hadoop do a poor job of supporting cross-cutting data concerns. Data has a wide variety of cross-cutting concerns: time, security, reliability, privacy, quality, summaries, rankings, and uncertainty. In this paper we adapted techniques from aspect-oriented programming (AOP) to Pig Latin. Pig Latin is a dataflow language for analyzing data in the cloud. We proposed annotating Pig data using data aspects. A data aspect binds advice (metadata) to data. The advice also has semantics that must be observed when the data is used in a query. We showed how to weave Pig Latin into a Pig Latin program to support cross-cutting concern data concerns.

In future we plan to address optimization, and management of data cuts, in particular mechanisms for tagging data with advice. Pig Latin does not have any data management role, so this would be prior to analysis by Pig Latin. We also plan to investigate new optimization rules for Aspect-oriented Pig Latin. We anticipate that several optimizations are possible, such as keeping a relation joined to its advice throughout evaluation of a program rather than splitting it into data, cuts, and advice relations after each transformation. We also need better support for UDFs. Each UDF must currently be written to correctly implement aspects. This places a great burden on UDF coders. To avoid requiring a user to modify each UDF, we need to instead partition the data into sets of tuples that have the same advice, and evaluate the UDF for only the given set. To the best of our knowledge there is no research yet on such partitioning for cloud computing. An alternative solution is to re-engineer a programming language like Java to support data aspects. Consider for instance an IF statement that compares two values tagged with temporal metadata. For some time periods the IF condition may be true, yet false for others, so each branch might need to be evaluated (for different time periods), resulting in a very different IF than currently exists in Java. This is also an open problem to the best of our knowledge.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1144404 entitled “III: EAGER: Aspect-oriented Data Weaving.” Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] M. Q. W. Baldonado, K. C.-C. Chang, L. Gravano, and A. Paepcke. Metadata for Digital Libraries: Architecture and Design Rationale. In *ACM DL*, pages 47–56, 1997.

[2] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006.

[3] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with Uncertainty and Lineage. *VLDB J.*, 17(2):243–264, 2008.

[4] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[5] E. Bertino, G. Ghinita, and A. Kamra. Access Control for Databases: Concepts and Systems. *Foundations and Trends in Databases*, 3(1-2):1–148, 2011.

[6] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB*, pages 900–911, 2004.

[7] K. Birman, G. Chockler, and R. van Renesse. Toward a Cloud Computing Research Agenda. *SIGACT News*, 40(2):68–80, June 2009.

[8] R. Bose and J. Frew. Lineage Retrieval for Scientific Data Processing: A Survey. *ACM Computing Surveys*, 37(1):1–28, 2005.

[9] P. Buneman and W. C. Tan. Provenance in Databases. In *SIGMOD Conference*, pages 1171–1173, 2007.

[10] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. DBNotes: A Post-It System for Relational Databases Based on Provenance. In *SIGMOD Conference*, pages 942–944, 2005.

[11] D. E. Denning and P. J. Denning. Data Security. *ACM Comput. Surv.*, 11(3):227–249, Sept. 1979.

[12] C. E. Dyreson. Aspect-Oriented Relational Algebra. In *EDBT*, pages 377–388, 2011.

[13] C. E. Dyreson, M. H. Böhlen, and C. S. Jensen. Capturing and Querying Multiple Aspects of Semistructured Data. In *VLDB*, pages 290–301, 1999.

[14] C. E. Dyreson and O. U. Florez. Data Aspects in a Relational Database. In *CIKM*, pages 1373–1376, 2010.

[15] C. E. Dyreson, R. T. Snodgrass, F. Currim, S. Currim, and S. Joshi. Weaving Temporal and Reliability Aspects into a Schema Tapestry. *Data Knowl. Eng.*, 63(3):752–773, 2007.

[16] B. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving Data Publishing: A Survey of Recent Developments. *ACM Comput. Surv.*, 42(4):14:1–14:53, June 2010.

[17] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *VLDB*, pages 632–643, 2003.

[18] H. Garcia-Molina, D. Hillmann, C. Lagoze, E. D. Liddy, and S. Weibel. How Important is Metadata? In *JCDL*, page 369, 2002.

[19] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a Highlevel Dataflow System on top of MapReduce: The Pig Experience. *PVLDB*, 2(2):1414–1425, 2009.

[20] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. In *ICDE*, page 82, 2006.

[21] L. Gravano, P. G. Ipeirotis, and M. Sahami. QProber: A System for Automatic Classification of Hidden-Web Databases. *ACM Trans. Inf. Syst.*, 21(1):1–41, 2003.

[22] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unification of Temporal Data Models. In *ICDE*, pages 262–271, 1993.

[23] H. Jin and C. E. Dyreson. Sanitizing using Metadata in MetaXQuery. In *SAC*, pages 1732–1736, 2005.

[24] H. Jin and C. E. Dyreson. Supporting Proscriptive Metadata in an XML DBMS. In *DEXA*, pages 479–492, 2008.

[25] D. Lee and Y. Hwang. Extracting Semantic Metadata and its Visualization. *ACM Crossroads*, 7(3):19–27, Mar. 2001.

[26] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD Conference*, pages 193–204, 2003.

[27] G. A. Mihaila, L. Raschid, and M.-E. Vidal. Using Quality of Data Metadata for Source Selection and Ranking. In *WebDB (Informal Proceedings)*, pages 93–98, 2000.

[28] S. Murthy, D. Maier, L. M. L. Delcambre, and S. Bowers. Superimposed Applications using SPARCE. In *ICDE*, page 861, 2004.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[30] G. Özsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.

[31] S. Pearson and A. Benameur. Privacy, Security and Trust Issues Arising from Cloud Computing. In *IEEE Cloud Computing (CloudCom)*, pages 693–702, Dec. 2010.

[32] A. Rashid. *Aspect-Oriented Database Systems*. Springer, 2003.

[33] A. Rashid and N. Loughran. Relational Database Support for Aspect-Oriented Programming. In *NetObjectDays*, pages 233–247, 2002.

[34] R. T. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In *Modern Database Systems*, pages 386–408. 1995.

[35] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[36] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[37] A. Tannenbaum, editor. *Metadata Solutions: Using Metamodels, Repositories, XML, and Enterprise Portals to Generate Information on Demand*. Addison-Wesley, 2001.

[38] H. Wagner and S. Weibel. The Dublin Core Metadata Registry: Requirements, Implementation, and Experience. *J. Digit. Inf.*, 6(2), 2005.

[39] M. Zhou, R. Zhang, W. Xie, W. Qian, and A. Zhou. Security and Privacy in Cloud Computing: A Survey. In *International Conference on Semantics Knowledge and Grid (SKG)*, pages 105 –112, Nov. 2010.