The Design of Aspect-Oriented Pig Latin

Curtis Dyreson and Omar U. Florez

Utah State University
curtis.dyreson@usu.edu, omar.florez@aggiemail.usu.edu

Abstract. In this paper we apply the aspect-oriented programming (AOP) paradigm to Pig Latin, a dataflow language for cloud computing. Missing from Pig Latin is support for cross-cutting data 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 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.

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. Data *quality* [3],[17] *provenance* [5],[6] *accuracy* and *lineage* [4],[29], *time*, *security*, *reliability* and *performance* are potential cross-cutting concerns. Each concern may have an individual and distinct semantics.

In spite of many years of research on individual concerns, e.g., 30+ years of research in temporal databases, research and industrial database management systems (DBMSs) lack support for cross-cutting data concerns (though some DBMSs support individual concerns, e.g., security). DBMSs are large, complex systems and not designed to be easily configured or modified to support a cross-cutting concern. Developers currently have to rely on ad-hoc techniques to add concerns to a data collection.

To better support cross-cutting data concerns a new approach 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 add cross-cutting concerns to a program without having to reprogram. In

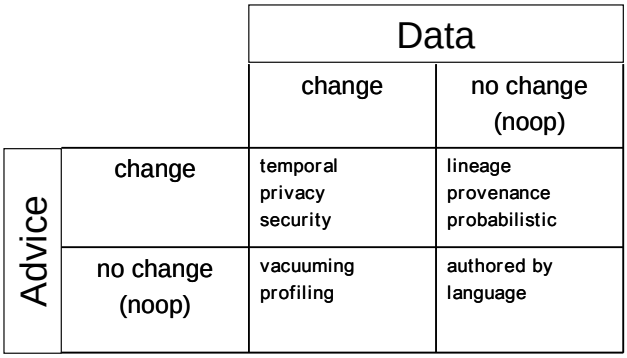

Figure 1 The space of cross-cutting concerns

AOP, an *aspect weaver* injects code, called *advice*, into a program at specified places, known as *point cuts*, to add new functionality to an existing program.

Previously we employed aspect-oriented techniques to create *aspect-oriented data* (AOD) for data stored in the relational model [10],[11]. 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. AOD 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-metadata) can also be modeled. We showed how to weave behavior into the relational algebra [11]. There has also been other research in using AOP in databases. Research has addressed using aspect-oriented techniques to program databases [24], using a relational database to support AOP [23], and applying AOP to XML schema [9].

Figure 1 gives a broad classification of the space of cross-cutting *data concerns* that can be addressed using an AOD 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 could also change the advice. In general, *change* or *no change* are the only possible data 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. This timestamps may (logically) delete data since the constructed times may be shorter. As a second example, consider data lineage. Lineage computes as metadata references to all of the data that contributes to a particular result, that is, it constructs advice 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.

In this paper we propose adapting AOP to Pig Latin [22] 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].

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.

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 **`Subscriber`** relation, is shown in Table 1. Each tuple in **`Subscriber`** records, in order, a name, city, and subscription amount.

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 furthermore gives a name and a type to each field within a tuple in the data. The statement also establishes 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 2. Next it applies query optimization rules to optimize the data flow (for instance the `GENERATE` 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

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

Subscriber
(Maya, Logan, $20)
(Jose, Logan, $15)
(Knut, Ogden, $20)

Table 1 Some data about subscribers to Magazine.com

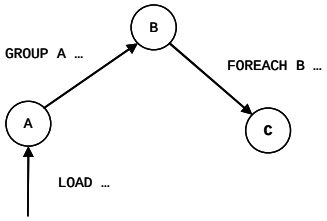

Figure 2 Dataflow in the simple program

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

optimized dataflow program evaluated using Hadoop, that is, the program is transformed to map-reduce constructs and executed in parallel.

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

2.2 Aspect-oriented Pig Latin Data

Pig Latin is a "NoSQL" language. A NoSQL language replaces SQL with a language that is better suited to programmers. 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 *does not support* cross-cutting data concerns. Users must resort to ad hoc techniques to implement, for instance, a temporal semantics for data. Snodgrass has pointed out the perils of relying on user good faith to correctly implement temporal semantics [27]. Usually Pig Latin programmers will not know which cross-cutting data concerns are present nor know how to program the semantics of an individual concern.

Pig Latin lacks many of the features found in other database languages. Pig Latin is a schema-less language. In the relational model of data, data is rigidly described by a fixed schema, but in Pig Latin, users load data from text files or back-end databases by sketching the types and number of each column in a relation in a query. Pig Latin also lacks data modification operators, the data is (assumed to be) created and maintained by other processes. Not surprisingly, Pig Latin also has no data constraint specifications. All

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

Data Cuts	
RF	**ID**
(1,	A)
(2,	B)
(2,	C)
(3,	D)

Security Advice		
Per	**Sec**	**MetaID**
(A,	Paid,	I)
(B,	Paid,	II)
(C,	Lapsed,	III)
(D,	Lapsed,	I)

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

Metadata Data Cuts	
RF	**MetaID**
(II,	X)
(III,	Y)
(I,	Z)

Temporal Meta Advice		
MetaId	**Start**	**End**
(X,	2007,	2008)
(Y,	2009,	now)
(Z,	2007,	now)

Table 4 Aspected subscribers

constraints are maintained by other processes. Finally, the Pig Latin data model supports sets and bags, as well as tuples; it is a non-first normal form (1NF) 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.

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* [8], or remain independent. Finally, since the advice is data, it too can be advised by meta-metadata. That is, metadata is to data as meta-metadata to metadata.

Several data cuts are concretely represented in Table 4 and Table 5 which extend and refine the subscriber database example of the previous section. We assume that each data tuple has an "**ID**" to allow it to be identified. This is the final field in each data tuple. For instance Maya's subscription is identified as `1`. Next, 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 **`RF`** 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 2007 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 can access this data.

Personal Info	Data Cuts	Security Advice	Lineage Advice
(Maya, maya@aol.com, 5)	(5, E)	(E, DBA)	(E, {5})
(Jose, jose@foo.com, 6)	(6, F)	(F, DBA)	(F, {6})
(Knut, knut@aol.com, 7)	(7, G)	(G, DBA)	(G, {7})

Table 5 Aspected personal information about subscribers

2.3 Weaving Behavior into Pig Latin Programs

An advice's behavior has to be woven into the evaluation of a Pig Latin program. The weaving technique that we adapt is to modify each transformation in a dataflow program is depicted in Figure 3. In Figure 3(a), the typical dataflow is depicted, data at a node *R* is transformed to that at node *X*. Figure 3(b) shows a Pig Latin *pattern* or *template* that will replace the transformation of Figure 3(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. In the next section we develop a specific template for each kind of Pig Latin transformation. But in general, the weaving is Pig Latin program modification whereby each statement in a program is replaced by a sequenced of statements constructed by instantiating a template for the transformation.

This strategy can be extended to additional levels of advice, e.g., meta-metadata. The pattern is repeatedly applied for each level as will be developed in the next section.

3 Aspect-Oriented Pig Latin

This section describes modifications of Pig Lain 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 transformation is redefined using (non-aspect-oriented) Pig Latin to illustrate that Pig Latin itself can be used to become aspect-oriented.

Each of the modifications is described in terms of a pattern or template. The template is applied to rewrite the corresponding transformation in a Pig Latin program.

We consider two broad categories of Pig Latin transformations: single vs. multiple tuple transformations. We first model single tuple transformations which involve only one tuple at a time and are generally simpler than the multiple tuple case.

3.1 Single Tuple

The single tuple transformations are `FILTER`, `GENERATE`, `SPLIT`, `SAMMPLE`, `LOAD`, `DUMP`, and `STORE`.

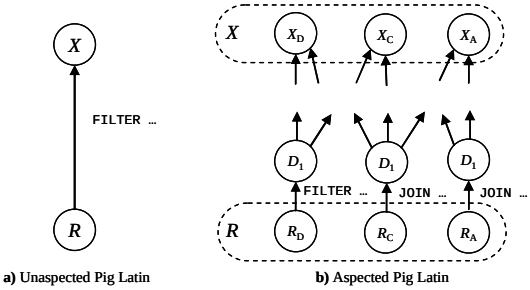

Figure 3 Weaving by template instantiation

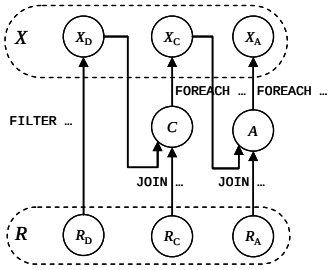

Figure 4 The pattern for $\texttt{FILTER}_{A0}$

Subscribers			
Name	**City**	**Amt**	**ID**
(Maya,	Logan,	$20,	1)
(Knut,	Ogden,	$20,	3)

Data Cuts
RF ID
(1, A)
(3, D)

Security Advice
Per Sec MetaID
(A, Paid, I)
(D, Lapsed, I)

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

Metadata Data Cuts
RF MetaID
(I, Z)

Temporal Meta Advice		
MetaId	**Start**	**End**
(Z,	2007,	now)

Table 6 Subscribers that paid $20 or more for their subscription

3.1.1 FILTER

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

X = FILTER R on P;

As the `FILTER` may remove some tuples, the data cuts and advice should be *synchronized* with the data, removing extraneous advice, following the `FILTER`. The template for $\texttt{FILTER}_{A0}$ is given below, with comments enclosed within '`/* */`'.

```
/* Filter the data */
X_D = FILTER R_D on P;

/* The code below is optional, repeated for each level of advice */
/* Remove extraneous cuts (cuts for tuples that were removed by
   filtering), remove by computing all the cuts still needed. */
C = JOIN R_C BY id, X_D BY id;
X_C = FOREACH C GENERATE C.id, C.ref;

/* Remove extraneous advice */
A = JOIN R_A BY ref, X_C BY ref;
X_A = FOREACH A GENERATE A.ref, A.data;
```

Figure 4 illustrates the basic pattern for the `FILTER` transformation. The pattern to the right of the `FILTER` transformation should be repeated for each level of advice. As an example, consider a query to filter subscribers below $20. The result is shown in Table 6. Maya and Knut are filtered, and synchronized with the advice to retain only their data cuts and advice tuples. Alternatively, the extraneous advice is harmless (except for occupying space) and can be left in place leading to the alternative, cheaper plan shown in Figure 5.

3.1.2 GENERATE

The `GENERATE` transformation projects only specified fields, $f_1, \ldots, f_n$, into the result.

```
X = FOREACH R GENERATE f1, ... , fn;
```

As all the tuples are retained, the data cuts and advice are unchanged, and so the template for $\texttt{GENERATE}_{A0}$ is simple.

```
/* GENERATE the data */
X_D = FOREACH R_D GENERATE f1, ... , fn;
/* Repeat rest of pattern for each level of advice */
X_C = R_C;
X_A = R_A;
```

The template is illustrated in Figure 6. 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 ..., c_n where each condition is a predicate involving field values.

```
SPLIT R INTO X IF c1, ..., Y IF cn;
```

A SAMPLE transformation 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 $\texttt{FILTER}_{A0}$. They apply the transformation to the data and then remove extraneous data cuts and advice, or alternatively, leave the cuts and advice unchanged since extraneous cuts and advice are harmless. We give the aspect-oriented split, $\texttt{SPLIT}_{A0}$, template below.

```
SPLIT R_D INTO X_D IF c1, ..., Y_D IF cn;

/* The rest is optional */
B = JOIN X_D BY X_D.id, R_C BY id;
C = JOIN B BY ref, R_A BY ref;
X_C = FOREACH C GENERATE C.id, C.ref;
X_A = FOREACH C GENERATE C.ref, C.advice;
...
D = JOIN Y_D BY Y_D.id, R_C BY id;
E = JOIN E BY ref, R_A BY ref;
Y_C = FOREACH C GENERATE C.id, C.ref;
Y_A = FOREACH C GENERATE C.ref, C.advice;
```

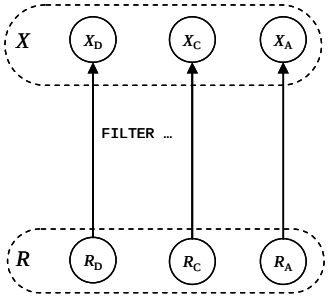

Figure 5 The alternative pattern for FILTER$_{AO}$

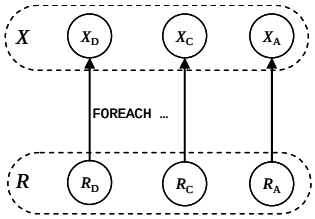

Figure 6 The pattern for GENERATE$_{AO}$

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) for each level of metadata rather than a single relation.

3.2 Multiple Tuple

Multiple tuple transformations involve more than one tuple and generally involve an advice-specific operation.

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 all are variants 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;

Advice constrains the join. If two tuples potentially join, their advice also needs to "join." For instance two tuples only join when their temporal advice overlaps (i.e., the tuples exist at the same time). The following template for an aspect-oriented join, JOIN$_{AO}$, regulates the join with advice.

```
/* Merge the data with the cuts and advice */
B = JOIN R_D BY R_D.id, R_C BY id;
C = JOIN B BY ref, R_A BY ref;
D = JOIN S_D BY S_D.id, S_C BY id;
E = JOIN D BY ref, S_A BY ref;

/* For each additional level of advice, join cuts and advice */
...

/* Join on the data conditions */
F = JOIN C BY f_R, D BY f_S;
```

```
/* Check the advice, for each level, repeat the following */
G = STREAM B THROUGH ADVICE-JOIN;

/* Generate the data, create a new cut */
X_D = FOREACH G GENERATE all data fields, G.id;

/* Generate the data cuts */
X_C = FOREACH G GENERATE G.id, G.ref;

/* Generate the advice */
X_A = FOREACH G GENERATE G.ref, G.advice;
```

The key transformation is streaming the data through an advice-specific join (`ADVICE-JOIN`). This user-defined function computes the "join" of the advice within 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, id's, and ref's 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 lists both tuples as the source.
  lineage-join((*rest*, *x*, *y*)) = {(*rest*, *x*), (*rest*, *y*)}
- Security advice – A partial order join is performed by keeping the most private group.
  security-join((*rest*, *x*, *y*)) = {(*rest*, **lca**(*x*, *y*))}

Finally, the `ADVICE-JOIN` manufactures a new data cut identifier and advice reference.

As an example, consider the `JOIN` of **`Subscribers`** with **`Personal Info`**. The result is shown in Table 7. The data relation contains three tuples. Note that the data cuts identifiers and advice references are composed values, manufactured from the underlying identifiers or references, respectively, they indicate which tuples were joined to produce a tuple in the join result. 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. The meta-metadata (**`Temporal Meta Advice`**) joins as is since every tuple overlaps the default interval "2007-now."

3.2.2 GROUP

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, ... , f_n;
```

Data		Data Cuts	Security Advice	Lineage Advice	Metadata Data Cuts	Temporal Meta Advice
Name …	ID	RF ID	Per Sec MetaID	Per Lin	RF MetaID	MetaId Start End
(Maya, …	1)	(1.5, A.E)	(A.E, DBA, I)	(A.E, {1,5})	(II, X)	(X, 2007, 2008)
(Jose, …	2)	(2.6, B.F)	(B.F, DBA,II)	(B.F, {2,6})	(III, Y)	(Y, 2009, now)
(Knut, …	3)	(2.6, C.F)	(C.F, DBA, III)	(C.F, {3,6})	(I, Z)	(Z, 2007, now)
		(3.7, D.G)	(D.G, DBA, I)	(D.G, {3,7})		

Table 7 Subscribers joined with Personal Info

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 following template regulates the grouping.

```
/* Merge the data with the cuts and advice */
B = JOIN R_D BY R_D.id, R_C BY id;
C = JOIN B BY ref, R_A BY ref;

/* Repeat merging for each level of advice */
...

/* Group on the data values */
D = GROUP C USING f_1, ... , f_n;

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

/* Flatten it and regroup using the data and new id */
F = FLATTEN E;
X_D = GROUP F USING f_1, ... , f_n, id;

/* Generate the data cuts */
X_C = FOREACH F GENERATE F.id, F.ref;

/* Generate the advice */
X_A = FOREACH F GENERATE F.ref, F.advice;
```

The template for a GROUP transformation is sketched in Figure 8. It uses an advice-specific 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 does not change the grouping.
  lineage-group(T) = noop
- Security advice – Each level in the hierarchy is its own group.
  security-group(T) = T

Cities	
City	ID
(Logan,	1)
(Ogden,	3)

Data Cuts	
RF	ID
(1,	A)
(1,	B)
(1,	C)
(3,	D)

Security Advice	
Per Sec	MetaID
(A, Paid,	I)
(B, Paid,	II)
(C, Lapsed,	III)
(D, Lapsed,	I)

Lineage Advice	
Per	Lin
(A,	1)
(B,	2)
(C,	3)
(D,	4)

Metadata Data Cuts	
RF	MetaID
(II,	X)
(III,	Y)
(I,	Z)

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

Table 8 Distinct cities

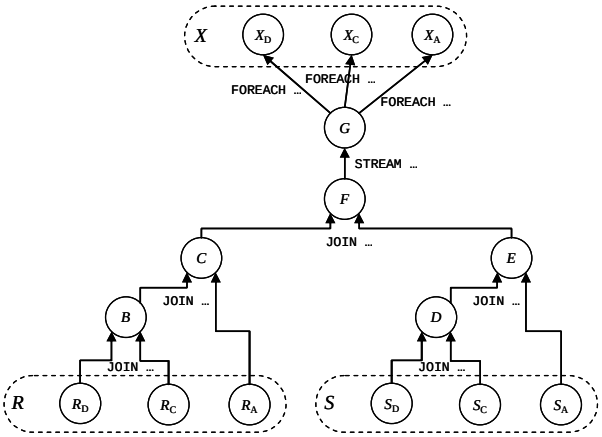

Figure 7 The pattern for JOIN$_{AO}$

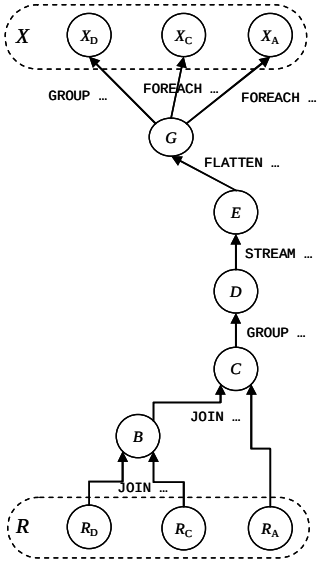

Figure 8 The template for GROUP$_{AO}$

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.

3.2.3 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 following template makes this change.

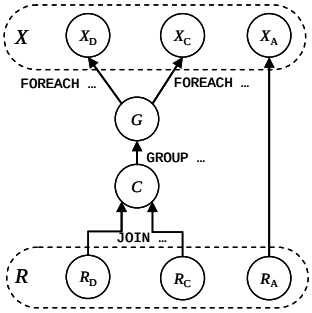

Figure 9 The pattern for DISTINCT$_{AO}$

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

The template is illustrated in Figure 9.

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 8.

3.2.4 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 The Example Revisited

We return to the example query of Section 2.1. Assume that we have a single security 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:int);
```

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

```
B = GROUP_A0 A BY dept;
C = FOREACH B GENERATE_A0 dept, COUNT(B.name);
DUMP_A0 C;
```

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

```
A_D = LOAD 'subscribers.data' USING PigStorage()
        AS (name: chararray, city: chararray,amount:int, id:chararray);
A_C = LOAD 'subscribers.cuts' USING PigStorage()
        AS (id: chararray, ref: chararray);
A_A = LOAD 'subscribers.advice' USING PigStorage()
        AS (ref: chararray, sec: int);

/* Merge the data with the cuts and advice */
B = JOIN A_D BY id, A_C BY id;
C = JOIN B BY ref, A_A BY ref;
/* Group on the data values */
D = GROUP C USING dept;
/* Stream through aspect-specific grouping */
E = STREAM D THROUGH SECURITY-GROUP;
/* Flatten it and regroup using the data and new id */
 F = FLATTEN E;
 B_D = GROUP F USING dept, id;
/* Generate the data cuts */
B_C = FOREACH F GENERATE id, ref;
/* Generate the advice */
B_A = FOREACH F GENERATE ref, sec;
/* Generate the result */
C_D = FOREACH B_D GENERATE dept, COUNT(B_D.name);
C_C = B_C;
C_A = B_A;
DUMP C_D;
```

3.4 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 Cartesian product. */
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 the new data cuts to remove extraneous 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.

3.5 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 $O(n \cdot m)$ 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 $\texttt{FILTER}_{A0}$, costs 1 FILTER + k*(2 JOINs and 2 GENERATEs), 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 $\texttt{FILTER}_{A0}$ is dominated by the cost of the k*2 JOINs, which concurs with the general rule of thumb, that the cost of joins dominates the query evaluation cost.

The analysis is summarized in Table 10. Most of the operations include only the additional cost of processing the cuts and advice, but five operations are much more expensive: $\texttt{FILTER}_{A0}$, $\texttt{JOIN}_{A0}$, $\texttt{GROUP}_{A0}$, $\texttt{SPLIT}_{A0}$, and $\texttt{SAMPLE}_{A0}$. We consider each in turn. $\texttt{FILTER}_{A0}$, increases the cost by eliminating 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 is harmless and does not need to be removed (other than for consistency), lowering the cost to $O(D)$. A similar speedup applies to $\texttt{SPLIT}_{A0}$ and $\texttt{SAMPLE}_{A0}$. $\texttt{JOIN}_{A0}$, is expensive because the data has to be joined to the advice and the data cuts for each join. This adds a factor of $O(kDC + kzAC)$ to the cost. This cost can be amortized over several joins by performing the join once and keeping the joined data in subsequent $\texttt{JOIN}_{A0}$ transformations. Finally $\texttt{GROUP}_{A0}$ 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.

Transform	Pig Latin	Aspect-oriented Pig Latin
FILTER	$O(D)$	$O(kDC + kzAC)$
GENERATE	$O(D)$	$O(D + kA + kC)$
DISTINCT	$O(D)$	$O(D + C)$
JOIN	$O(D^2)$	$O(D^2 + kDC + kzAC)$
GROUP	$O(D)$	$O(kDC + kzAC)$
UNION	$O(D^2)$	$O(D^2 + kC^2 + kzA^2)$
SPLIT/SAMPLE	$O(D)$	$O(kDC + kzAC)$
LOAD/STORE/DUMP	$O(D)$	$O(D + kC + kzA)$
PROGRAM ASPECTS	-	$O(kDC + kzAC)$

Table 10 Complexity Analysis

4 Related Work

There is a little previous research on support for manifold kinds of metadata in database management systems. Most closely related to this paper is our work on aspect-oriented relational algebra [10],[11], and query languages for metadata annotating XML [8],[16]. This paper in contrast focuses on Pig Latin, though we take the same approach as our relational algebra research by using data cuts and advice relations and the modeling of the many operations that relational and Pig Latin share, e.g., joins, selection, and projection. This paper makes three novel contributions. First we apply AOP to Pig Latin transformations that differ from relational algebra transformations, such as grouping, distinct, and sample, as well as to those that are familiar relational algebraic transformations. Second we show how to model meta-metadata as advice tagging advice.

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 [25], object-oriented [26], and XML [12]. 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 [15], 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.

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 [20]. The SPARCE system wraps or super-imposes a data model with a layer of metadata [21]. 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 [2],[19]. 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 [28]. The Dublin Core is a commonly used classification standard [7]. Commercial and research systems [1] to manage (descriptive) metadata collections have been developed. Methods to automatically extract content-related metadata have also been researched [14],[18]. The focus of the information retrieval research is on how to best use, manage, and collect metadata to describe data to improve search [13]. 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.

5 Conclusions

This paper proposes adapting a popular software engineering design technique to the management of data in a cloud computing context. Current cloud computing systems do a poor job of evaluating cross-cutting 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 data. A data aspect combines advice (metadata and special semantics) with a data cut (that pinpoints the data which the advice modifies). In our approach, the advice is woven around the data enriching its meaning and use. Though specific data aspects have individually been studied in detail, e.g., as a temporal database, data models and query languages that combine many, disparate aspects have received little attention.

This paper presents a design for aspect-oriented Pig Latin. Pig Latin is a dataflow language for analyzing data in the cloud. We proposed annotating or tagging 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.

6 Acknowledgements

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.

7 References

[1] M. Baldonado, C.-C. Chang, L. Gravano, A. Paepcke, "Metadata for Digital Libraries, Architecture and Design Rationale," in *ACM/IEEE JODL*, pp. 47-56, 1997.
[2] P.. Bernstein, "Applying Model Management to Classical Meta Data Problems," *CIDR* 2003, pp. 209-220.
[3] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya, "An Annotation Management System for Relational Databases," in *VLDB*, 2004, pp. 900–911.
[4] R. Bose and James Frew. *Lineage Retrieval for Scientific Data Processing: A Survey*. ACM Computing Surveys, 2005. 37(1): 1–28.
[5] P. Buneman, A. Chapman, and J. Cheney, "Provenance Management in Curated Databases," in *SIGMOD*, 2006. Chicago, pp. 539–550.
[6] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan, *Archiving Scientific Data*. ACM TODS, 2004. 29(1): 2–42.
[7] Dublin Core Metadata Initiative. At `dublincore.org`.
[8] C. Dyreson, M. Böhlen, and C. Jensen. "Capturing and Querying Multiple Aspects of Semi-structured Data," in *VLDB* 1999, pp: 290-301.
[9] C. Dyreson, R. T. Snodgrass, F. Currim, S. Currim and S. Joshi. *Weaving temporal and reliability aspects into a schema tapestry*. Data and Knowledge Engineering, 63(3), December 2007, pp. 752-773.
[10] C. Dyreson and Omar Florez. "Data Aspects in a Relational Database," in CIKM 2010, pp. 1373-1376.
[11] Curtis Dyreson. "Aspect-oriented Relational Algebra," in EDBT 2011, pp, 377-388.
[12] Dengfeng Gao and Richard T. Snodgrass. "Temporal Slicing in the Evaluation of XML Queries". In *VLDB*, pp. 632-643, 2003.
[13] Hector Garcia-Molina, Diane Hillmann, Carl Lagoze, Elizabeth Liddy, and Stuart Weibel, "How important is metadata?", In *ACM/IEEE-CS JODL*, pp. 369-369, 2002.
[14] Luis Gravano and Panagiotis G. Ipeirotis and Mehran Sahami, "QProber: A system for automatic classification of hidden-Web databases". ACM Transactions on Information Systems. 21(1): 1-41, 2003.
[15] C. S. Jensen, M. Soo, and R. T. Snodgrass, "Unification of Temporal Data Models," in *ICDE*, Vienna, Austria, 1993, pp. 262-271.
[16] Hao Jin and Curtis Dyreson, "Supporting Proscriptive Metadata in an XML DBMS," in *DEXA* Turin, Italy, September 2008, pp. 479-492.
[17] Anastasios Kementsietsidis, Floris Geerts, and Diego Milano, "MONDRIAN: Annotating and Querying Databases through Colors and Blocks," in *ICDE*, 2006, p. 82.
[18] Dongwon Lee and Yousub Hwang, *Extracting Semantic Metadata and its Visualization*. ACM Crossroads, 2001, 7(3): 19-27.
[19] Sergey Melnik, E. Rahm, E., and Phil A. Bernstein. "Rondo: A Programming Platform for Generic Model Management". In *ACM SIGMOD* 2003, pp. 193-204.
[20] George A. Mihaila, Louiqa Raschid, Maria-Esther Vidal. "Using Quality of Data Metadata for Source Selection and Ranking," in *WebDB* (Informal Proc.): 93-98. May 2000.
[21] Sudarshan Murthy, David Maier, Lois M. L. Delcambre, Shawn Bowers. "Superimposed Applications using SPARCE," in *ICDE*: 861. Boston, MA, USA, March 2004.
[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, "Pig Latin: a Not-so-Foreign Language for Data Processing," in *SIGMOD Conference*, 2008, pp. 1099-1110.
[23] Awais Rashid and N. Loughran, "Relational Data-base Support for Aspect-Oriented Programming," in *NetObjectDays Conference*. LNCS. Volume 2591, 2002, pp. 233-247.
[24] Awais Rashid, "Aspect-Oriented Programming for Database Systems," in *Aspect-Oriented Software Development*, 2004.
[25] Richard T. Snodgrass (Ed.). *The TSQL2 Temporal Query Language*. Kluwer, 1995. 629 pages.

[26] Richard T. Snodgrass, "Temporal Object-oriented Databases: a Critical Comparison," in *Modern Database Systems: the Object Model, Interoperability*, and Beyond. Addison Wesley, 1995. pp. 386-408.
[27] R. Snodgrass. Developing Time-Oriented Database Applications in SQL. Kaufmann. 1999. 540 pages.
[28] Adrienne Tannenbaum. *Metadata Solutions: Using Metamodels, Repositories, XML, and Enterprise Portals to Generate Information on Demand*. Addison-Wesley, 2001.
[29] Jennifer Widom, "Trio: A System for Integrated Management of Data Accuracy, and Lineage," in *CIDR*, 2005. pp. 262–276.