Aspect-Oriented Relational Algebra

Curtis E. Dyreson
Utah State University
Logan, Utah, USA
+1 435 797 0742
Curtis.Dyreson@usu.edu

ABSTRACT
In this paper we apply the aspect-oriented programming (AOP) paradigm to the relational algebra. AOP is a way to add support for cross-cutting concerns to existing code without directly modifying that code. Data, like code, also has cross-cutting concerns such as versioning, privacy, and reliability. AOP techniques can be used to weave metadata around an application's data. The metadata imbues the data with additional semantics that must be observed in constraint and query processing. In this paper we show how to modify the relational algebra to process data woven together with metadata. We also analyze the overhead on evaluating an aspect-enhanced query.

Categories and Subject Descriptors
H.2.3 [**Database Management**]: Relational algebra, metadata.

General Terms
Management, Languages.

Keywords
Aspect-oriented, relational algebra, cross-cutting concerns.

1. INTRODUCTION
A *cross-cutting concern* is a universal program behavior, one that is potentially needed in many disparate parts of a program, but is often developed and modeled separately. Common cross-cutting concerns in programming include object versioning, event logging, and memory management: such functionality is used to enhance, instrument, or debug an application, making it more robust, portable, and reliable. Cross-cutting concerns can be quickly and easily added to an application using a new programming paradigm called *aspect-oriented programming* (AOP). In AOP each concern is modeled as an *aspect*. An aspect couples *advice*, which is code from the implementation of a cross-cutting concern, with a *point cut*, which specifies where and when in the execution of the application the advice is *woven* or placed.

Figure 1 gives an overview of AOP. In the figure an *aspect weaver*, e.g., AspectJ, weaves a program (`Program.java`) with two cross-cutting concerns, one that implements object persistence (`Persist.java`) and another that implements event logging (`Log.java`). Each aspect combines advice from a concern with a point cut. The aspect weaver injects the advice into the program's behavior, at the model-level (e.g., in the UML), the code-level (e.g., in Java), or at a low-level (e.g., in the JVM). AOP supports the separation of concerns in program design and development, modularizes implementation of cross-cutting concerns, and promotes code reuse. AOP promotes *aspect independence* which is the principle that a cross-cutting concern is independent of an application and can be added *ex post facto* to enhance an application without reprogramming the application, e.g., in Figure 1 without modifying `Program.java`.

Data, like code, also has important cross-cutting concerns. Data can be annotated with descriptions of *where* it came from, *who* inserted or changed it, and *what* its quality is [4],[19]. The *provenance* of the data, what manipulations were performed on it to get it to this point, can also be recorded [6],[7]. Similarly, the *accuracy* and *lineage* of the data can be captured [5],[30]. *Security* and *privacy* introduce additional cross-cutting concerns, such as who has access to the data and to whom information has been released. *Reliability* and *performance* requirements are also potential cross-cutting concerns.

But current DBMSs offer little support for cross-cutting data concerns, though research has addressed using AO techniques to program databases [25], and using a relational database to support AOP [24]. In this paper we propose adapting the AOP paradigm to relational data, creating aspect-oriented relations and queries. (We do not consider aspect-oriented schemas in this paper [10].) The AOP paradigm modifies *dynamic* program behavior, so has to be adapted to include data, which is (largely) *static*.

Figure 2 gives an overview of the process of creating aspect-oriented relations and queries. In the figure a temporal concern (`temporal.rel`) and a privacy concern (`privacy.rel`) are woven into the data (`data.rel`) and to a query (`query.sql`). The advice in each concern is a relation of *metadata*, that is, the advice is data about data. The advice could describe who has access to the data, how the data was measured, and when the data is current, among other things. For a temporal cross-cutting concern, the advice will be a relation of timestamps, where each timestamp describes (part of) the data's lifetime. For a privacy concern it could be a relation of privacy groups or privileges. The advice is bound to data (or a query) at a *data cut*, which specifies where the advice should be woven. The data cut together with the advice forms a *data aspect*. A *data aspect weaver* weaves the advice into the data yielding an aspect-oriented relation or around a query yielding an aspect-oriented query.

The three notions that we borrow from AOP for aspect-oriented data are

1) that aspects can be developed independent of applications,

EDBT 2011, March 22-24, 2011, Uppsala, Sweden.

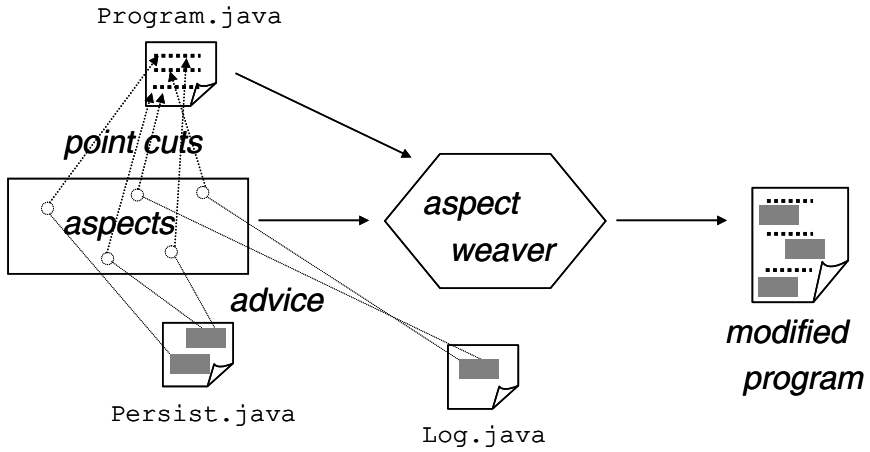

Figure 1 An overview of aspect-oriented programming

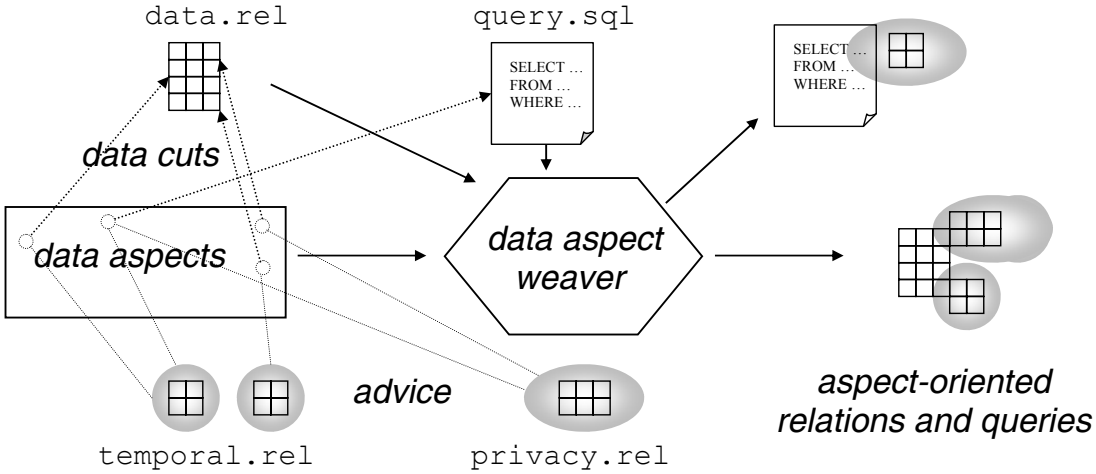

Figure 2 Creating aspect-oriented relations and queries

2) that aspects can be woven into already existing applications, and
3) that there needs to be some system (DBMS) support for implementing aspects.

Previously we described how to model data aspects in a relational database [11]. In this paper we focus on the relational algebra. The main contribution of this paper is an extension of each relational algebraic operator to support queries on the data aspects. We introduce new operations to filter, regulate, and synchronize the data aspects. We also briefly describe how constraints in the relational model can be augmented.

This paper is organized as follows. The next section develops a motivating example. After that, aspects are described in detail. The paper then focuses on query weaving by developing a relational algebra for aspect-oriented relations and queries. The last three sections cover performance, related work and future work.

2. MOTIVATING EXAMPLE

Assume that a company records information about its employees in a relational database. The database is designed well and has an extensive set of integrity constraints. Many applications, such as a payroll application, have been built to query, update, and manage the database. A portion of the database, the **`Employees`** relation, is shown in Table 1. Each tuple in **`Employees`** records a name (**`Name`**), department where employed (**`Dept`**), salary (**`Sal`**), and a key (**`ID`**) (note that we adopt a very simple key only for expository purposes, any key will suffice).

2.1 Cross-cutting Concerns

Companies continually evolve, over time new database requirements arise. A new tax law makes it mandatory to retain the salary history of each employee, so that auditors can ascertain the pay-roll at any given time. Another requirement comes from applica-tion coders who need to test their code with the "live" system, mixing live data with "test" data, that is, data that should be used only for testing. A third requirement emerges out of a

Employees			
Name	**Dept**	**Sal**	**ID**
Joe	Shoes	40K	1
Joe	Admin	100K	2
Sue	Shoes	50K	3
Fred	Admin	90K	4

Table 1 Some data about employees

new privacy policy adopted by the company. The policy establishes a hierarchy (complete partial order) of privacy groups, with the most privi-leged, super user at the top and the general public at the base. Data for each group will be available only to users of that group (or to users in groups above that group in the hierarchy).

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 the new data and functionality without changing a line of application code, database query code, or integrity constraint code.

2.2 Aspect-oriented Relations

In an aspect-oriented approach, the database designers "tag" data in the database with advice, creating aspects. Figure 3 shows the employee data in Table 1 together with several aspects. There are three kinds of aspects in the figure: temporal aspects, test aspects, and privacy aspects. Each aspect binds advice to data. A temporal aspect binds a timestamp(s) to data to signify when an employee is employed, i.e., the valid time lifetime of the employee. Test advice is a number identifying a test suite, and a test aspect identifies the test suite to which the data belongs. Finally, privacy advice is the name of a group established by the privacy policy.

Aspects are developed and applied independently, but several aspects can be simultaneously woven to data to act in concert. For instance, in Figure 3 the privacy and temporal advice woven to the fact that Joe worked in Admin earning 100K means that he was employed from 2007 to now and that only people in at least the administrators group can see this fact. We will call the combined aspects a *perspective* [9]. Alternatively, the aspects can be treated independently, in which case each aspect forms its own perspective.

2.3 Aspect-oriented Queries

Aspects can also be applied to intensional data, i.e., queries. Consider a query to retrieve the salary of each employee named Joe. In the relational algebra this query can be expressed as follows:

$\pi_{\mathbf{Sal}}(\sigma_{\mathbf{Name} = \texttt{'Joe'}}(\mathbf{Employees}))$.

The query can be aspected to retrieve different aspects of Joe's salary. For instance, if the query were "aspected" with the temporal advice '(2002, 2002)', then its evaluation would produce the salaries of employees named Joe who worked during 2002. If it were instead aspected with the test advice 'test suite 20', then data in that test suite would be used to compute a result (test data is excluded by default).

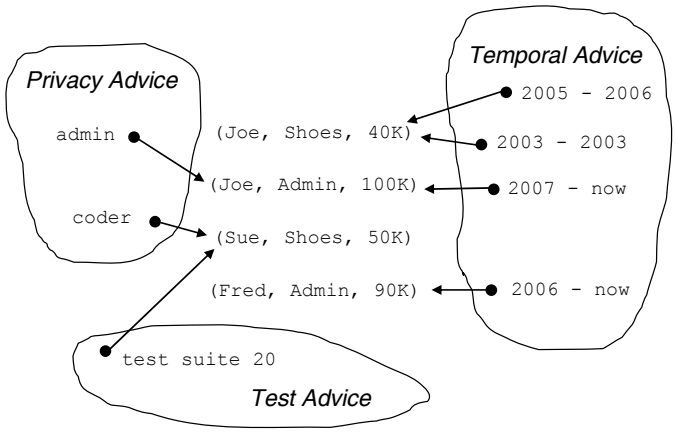

Figure 3 Adding advice to the data

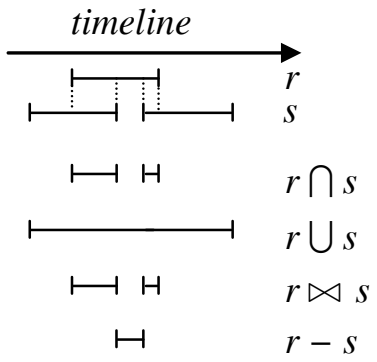

Figure 4 Time periods in sequenced semantics

While describing data using tags is common at social networking sites, such as `flickr.com`, an aspect-oriented approach has to go further because each kind of advice imposes a *semantics* for using the data in a query or view, to satisfy a constraint, or when the data is modified in an update, insertion, or deletion. For instance, temporal advice should impose *sequenced semantics* [28]. In sequenced semantics an operation is logically applied at every snapshot. Figure 4 sketches the time period of the result for four binary operations in the relational algebra using sequenced semantics; *r* and *s* are the time periods of the operands, e.g., tuples can join only on times that they both exist in the database.

Constraints, like queries, can also be aspected. For instance, suppose that the key of the **`Employees`** relation is a foreign key in some other relation. Then the valid time advice of any tuple that references Joe's employment in Shoes has to contain (at least) the lifetime of Joe's employment in Shoes; otherwise some snap-shot of the database violates a temporal referential integrity constraint.

Temporal Advice	
TTBeg	**TTEnd**
2003	2003
2005	2006
2007	now
2006	now

Table 2 A relation of temporal advice

Finally, we do not discuss in this paper GUI tools that would help designers quickly and easily tag data and queries with advice, or tools to help designers specify or implement an aspect's semantics. Though we believe such tools are critical to aspect-oriented relations and queries, they are beyond the scope of this paper.

3. WHAT IS AN ASPECT?

In AOP an aspect enhances the *dynamic*, run-time behavior of a program; hence the advice in an aspect is implemented as a body of code. A data aspect on the other hand enhances *static* data. So a data aspect is, in part, a body of data, or more precisely in the relational model, a relation. As an example Table 2 shows the temporal advice for facts in the **Employees** relation of Table 1. Each tuple in the **Temporal Advice** relation is a time period from **TTBeg** to **TTEnd**. Multiple time periods may advise a single employee. Note that no data cuts have been specified (yet), so the advice has not been woven to the data.

Some data will not be explicitly woven to advice by a data aspect or will be aspected by only a few kinds of advice, in which case default advice will be used for each cross-cutting concern. The default advice for the example relation in Table 1 is that the data's lifetime is assumed to encompass only the time *now*, the data is "live" data not inserted or modified as a result of testing, and the data is public without any special privacy restrictions.

3.1 Tuple Data Cuts

In AOP, the code in the aspect is woven into a program's code at point cuts. A point cut is a place or event in a program's execution. Possible point cuts include entry to a function, exit from a loop, entry to an assignment statement, and lookup of a variable's value. In adapting AOP to the relational model, far fewer data cuts are discernable, in fact, only three are readily apparent: a value, a tuple, and a relation.

In this paper, we focus on *tuple data cuts*, that is, we assume advice will be woven to a tuple. Let's consider a single data relation, R_D, consisting of a set of tuples $t_1, \ldots, t_n$. Assume further that the advice is a relation R_A consisting of tuples $m_1, \ldots, m_n$. A data cut for a tuple, t_i, weaves tuples $m_j \ldots m_k$ to t_i. Figure 3 illustrates many data cuts. For example in the figure, the advice tuples, (2003, 2003) and (2005, 2006), are woven to a tuple in the data, (Joe, Shoes, 40K), by a data cut. The advice represents the valid time lifetime of Joe's employment in Shoes.

Many data cuts are concretely represented in Table 3 and Table 4 which extend and refine the company database example of the previous section. Table 3 shows the aspected **Employees** relation (combining Table 1 with Table 2), and Table 4 extends the database with an aspected **Departments** relation. An advice tuple shaded in grey denotes default advice. The **Data Cuts** relation weaves advice, identified by the **RF** column, to data. An employee can be aspected in three ways.

1. By temporal aspects that record the valid time lifetime of each employee.

2. By test aspects that mark tuples which are used only for testing purposes.

3. By privacy aspects that record the privacy group for a tuple, established by the privacy policy. The groups are in a complete partial order from the lowest level (public) to the top secret level (top). The admin and coder groups are at the same level in the partial order, below top and above public.

Each tuple in the advice has an **RF** column, which identifies the perspective to which the tuple belongs. A single perspective can have several tuples of the same kind of advice, and different perspectives can be woven to the same data tuple. For instance, perspectives W and Z are woven to **Departments** tuple 6.

3.2 Advice Behavior

The advice for a tuple becomes *active* whenever a tuple is used in a query or view, used to satisfy a constraint, or modified in an update. Advice can play (at least) three roles when it becomes active as described below.

First, advice can *regulate* an operation. Query operations, such as the unary and binary operations of the relational algebra, (logically) produce data. The advice can regulate this production, turning it off for some tuples. For example, suppose we evaluate a query to project the names of all employees. Such a query is traditionally interpreted (in a temporal database) to involve only employees currently employed, i.e., those whose time of employment overlaps *now*. When the projection is applied to the **Employees** relation of Table 1, the advice for each tuple regulates whether it is included in the result. In Joe's case, Figure 3 shows that the lifetime of his employment in Shoes ends prior to the current time as neither timestamp contains the current time (assumed to be 2008). So the name Joe is not included in the result of the projection (though Joe may be included from the projection applied to other tuples).

Second, advice can *mutate* data before or after an operation. The mutation modifies the tuple, either changing attribute values, or appending new values. For example, suppose that we want to append the time of Joe's employment in Shoes to the projection of employee names, i.e., convert the advice to data. An aspect can play a mutator role and compose the temporal advice for each tuple in the projection with its advice after it is projected. A mutator role for an aspect would also be useful in translating along *attribute data cuts*, e.g., when values in a column of data advised by an English aspect are compared to data advised by a Spanish aspect, the English can be translated to Spanish with the help of a dictionary.

Third, an aspect can *construct* data and advice. The construction can be done prior to or after an operation. As an example, suppose that some tuples in Employees do not have temporal advice. Over what times are these tuples current? One possibility is that they are only current *now*. To enforce this in query semantics, prior to an operation on the Employees relation the temporal aspect could construct new advice for each unadvised employee. As a second example, suppose we want to track the *lineage* of tuples produced by a query, where the lineage is defined as the

Employees			
Name	Dept	Sal	ID
Joe	Shoes	40K	1
Joe	Admin	100K	2
Sue	Shoes	50K	3
Fred	Admin	90K	4

Data Cuts	
ID	RF
1	A
2	B
3	C
4	D

Temporal Advice		
RF	TTBeg	TTEnd
A	2003	2003
A	2005	2006
B	2007	now
D	2006	now
C	now	now

Test Advice	
RF	Suite
C	20
A	0
B	0
D	0

Privacy Advice	
RF	Group
B	admin
C	coder
A	public
D	public

Table 3 Aspected Employees

Departments		
Loc	Dept	ID
E104	Shoes	5
A2	Admin	6
F77	Shoes	7

Data Cuts	
ID	RF
5	W
6	X
7	Y
6	Z

Temporal Advice		
RF	TTBeg	TTEnd
W	2000	2003
X	2002	now
Y	2007	now
Z	now	now

Test Advice	
RF	Suite
W	15
Z	20
X	0
Y	0

Privacy Advice	
RF	Group
W	coder
X	public
Y	public
Z	public

Table 4 Aspected Departments

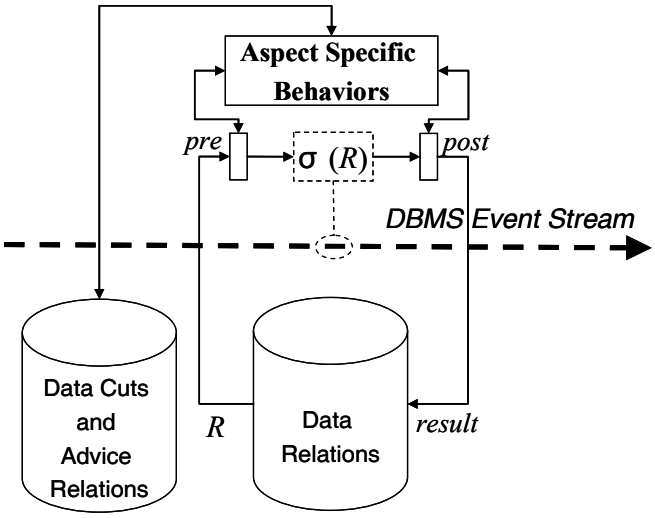

Figure 5 An architecture for the query weaver

data tuples involved in producing a query result. Lineage data aspects can be dynamically constructed to track the lineage.

3.3 Weaving Behavior into Queries

An advice's behavior has to be woven into the evaluation of a query, and around other events in a DBMS. This requires support from the DBMS, specifically, opening the DBMS's event stream to weaving. Figure 5 shows an architecture for a data aspect weaver. In the figure the DBMS event stream is depicted as a dashed arrow representing a sequence of discrete events. One particular event, a restriction operation applied to relation *R*, is highlighted. The operation has to be woven together with aspect-specific behaviors as described in Section 4, but in general, each aspect has a special behavior when involved in a join, union, intersection, filter (evaluating a query aspect), or difference operation. Prior to the operation, the aspect can mutate or construct *R* (it depends on the semantics of the advice), as represented by the *pre* edge in the figure. After the operation *R* can be regulated by the aspect, as represented by the *post* edge in the figure. The advice and data cuts relations are heavily involved in the evaluation of a query operation like restriction.

4. ASPECT-ORIENTED QUERIES

This section develops an algebra for aspect-oriented relations. We model a complete set of relational algebra operators, showing how each is redefined to support data aspects. Each operation is redefined using the (non-aspect-oriented) relational algebra to illustrate that current DBMS software can be used to become aspect-oriented. We focus on the regulatory role of aspects. Only data aspects are initially considered; in Section 4.2 query aspects are introduced.

4.1 Aspect-oriented Relational Algebra

In the following definitions, without loss of generality, we assume that a data relation, R_D, has a single advice relation, R_A, and as single data cuts relation, R_C.

[Aspect-Oriented (AO) Restriction] Select the tuples that meet a condition, P. As the restriction may remove some tuples, the data cuts and advice should be *synchronized* with the data, removing extraneous advice, following the selection:

$$\sigma_P([R_D, R_C, R_A]) = \textbf{synch}([\sigma_P(R_D), R_C, R_A]).$$

The synchronization operation removes the advice that no longer advises any tuple in the data.

[Synchronize] Synchronize the advice with the data by removing extraneous advice tuples, i.e., those that do not advise any data:

$$\textbf{synch}([R_D, R_C, R_A]) = [R_D, R_C \ltimes R_D, R_A \ltimes (R_C \ltimes R_D)].$$

As an example, consider a query to select employees in Shoes. The result is shown in Table 5. Joe and Sue are in Shoes so their tuples are selected, and synchronized with the advice to retain only their data cuts and advice tuples.

Employees			
Name	**Dept**	**Sal**	**ID**
Joe	Shoes	40K	1
Sue	Shoes	50K	3

Data Cuts	
ID	**RF**
1	A
3	C

Temporal Advice		
RF	**TTBeg**	**TTEnd**
A	2003	2003
A	2005	2006
C	now	now

Test Advice	
RF	**Test**
C	20
A	0

Privacy Advice	
RF	**Group**
C	coder
A	public

Table 5 Employees in Shoes and their advice

Projection is straightforward if it does not require duplicate elimination (i.e., when a key is projected).

[AO Projection without duplication elimination] Project using the list of attributes B, which has a key. The advice is unchanged.

$$\pi_B([R_D, R_C, R_A]) = [\pi_B(R_D), R_C, R_A]$$

[AO Projection with duplicate elimination] Project using the list of attributes B. After projecting, duplicates should be eliminated, and their advice combined.

$$\pi_B([R_D, R_C, R_A]) = \textbf{dup-elim}([\pi_B(R_D), \pi_{B,\mathbf{ID}}(R_C), R_A])$$

Duplicate elimination is more complex with aspects since data cuts on duplicate tuples must be transferred to tuples that remain.

[AO Duplicate-elimination] Duplicate elimination changes the data cuts for some advice, while removing duplicates:

$$\textbf{dup-elim}([R_D, R_C, R_A]) = [R_D, R_C - D, D_A]$$

where

$$D = \pi_{T.B,T.\mathbf{ID}}(\sigma_{S.B=T.B \wedge S.\mathbf{ID}<T.\mathbf{ID}}(C))$$

is a relation of duplicate data tuples,

$$C = \rho_S(R_D \bowtie R_C) \times \rho_T(R_D \bowtie R_C)$$

and

$$D_A = \pi_{R_C.\mathbf{ID},A}(R_C \bowtie_{R_C.B=D.B} (D \bowtie R_A)) \cup R_A \rtimes (R_C - D)$$

is the advice for the duplicates, with affiliations switched to the remaining tuples.

As an example, consider the projection of employee names. The result is shown in Table 6. The tuple with **ID** 2 has been eliminated as a duplicate, and the advice for that tuple has switched to refer to the tuple with **ID** 1. Note that there appears to be a privacy policy violation in the projection. The privacy advice for the tuple with **ID** 2 is admin in Table 3 but in the projection, the privacy advice for Joe is public as well as admin. Since the temporal advice for the tuple with **ID** 2 is part of the new advice for Joe, in effect Joe's time in Admin becomes visible to all as a result of the projection. But this apparent violation is because the query aspect was not applied (see Section 4.2, in effect, this query was asked from the perspective of the top privacy group).

[AO Joins and **Cartesian Product]** Let $\otimes$ be $\bowtie$, $\bowtie_\theta$, $\rtimes$, or $\times$. When tuples are composed, their advice must be as well. For every combination of data, manufacture a combined perspective:

$$[R_D, R_C, R_A] \otimes [S_D, S_C, S_A] = \textbf{synch}(\textbf{regulate}([R_D \otimes S_D, (R_C \times S_C) \rtimes J_A, J_A]))$$

Names	
Name	**ID**
Joe	1
Sue	3
Fred	4

Data Cuts	
ID	**RF**
1	A
3	C
4	D

Temporal Advice		
RF	**TTBeg**	**TTEnd**
A	2003	2003
A	2005	2006
A	2007	now
D	2006	now
C	now	now

Test Advice	
RF	**Test**
C	20
A	0
D	0

Privacy Advice	
RF	**Group**
A	public
A	admin
C	coder
D	public

Table 6 Employee names and their advice

where

$$J_A = \textbf{advice-join}(R_A, S_A)$$

is a join of the advice.

The **regulate** operation ensures that each data tuple has advice.

[Regulate] This operation removes from the data those tuples that do not have any advice:

$$\textbf{regulate}([R_D, R_C, R_A]) = [R_D \rtimes R_C, R_C, R_A)].$$

[**Advice join**] Advice join (**advice-join**) computes the "join" of pairs of advice tuples using an aspect-specific join. Example advice-specific joins are listed below.

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

$$\textbf{temporal-join}(\{(i, t, u)\}, \{(j, v, w)\}) = \begin{cases} \{(i, j, \textbf{max}(t, v), \textbf{min}(u, w))\} \\ \quad \text{if } (t, u) \text{ overlaps } (v, w) \\ \{\} \quad \text{otherwise} \end{cases}$$

- Test advice – Test suite x joins test suite y if both are the same test or either is live data (assumes test data is mixed with live data in a test).

$$\textbf{test-join}(\{(i, x)\}, \{(j, y)\}) = \begin{cases} \{(i, j, x)\} & \text{if } x = y \text{ or } y = 0 \\ \{(i, j, y)\} & \text{if } x = 0 \\ \{\} & \text{otherwise (if } x \neq y) \end{cases}$$

- Privacy advice – A partial order join is performed by keeping the most private group.

$$\textbf{privacy-join}(\{(i, x)\}, \{(j, y)\}) = \begin{cases} \{(i, j, \textbf{lca}(x, y))\} \\ \quad \text{if } \textbf{lca}(x, y) = x \text{ or } \textbf{lca}(x, y) = y \\ \{\} \quad \text{otherwise} \end{cases}$$

As an example, consider the natural join of **Employees** with **Departments**. The result is shown in Table 7. The data relation contains four tuples. Note that the **ID** column has a composed identifier, which also serves to indicate which tuples were joined from Table 3 and Table 4 to produce a tuple in the join result. Some tuples have been "regulated," for instance, tuples 1 and 7 would join if only the data were considered, but by inspecting their temporal advice we can determine that the tuples were never in the database at the same valid time, and hence their combin-

EmployeesInDepartments

Name	Dept	Sal	Loc	ID	
Joe	Shoes	40K	E104	1	5
Joe	Admin	100K	A2	2	6
Sue	Shoes	50K	F77	3	7
Fred	Admin	90K	A2	4	6

Data Cuts

ID		RF
1	5	A.W
2	6	B.X
3	7	D.X
4	6	C.Y

Temporal Advice

RF		TTBeg	TTEnd
A	W	2003	2003
B	X	2007	now
D	X	2007	now
C	Y	now	now

Test Advice

RF		Suite
A	W	15
B	X	0
C	Y	0
D	X	0

Privacy Advice

RF		Group
A	W	coder
B	X	admin
C	Y	coder
D	X	public

Table 7 Joining employees with departments

ation should not be in the join result. Similarly tuples 3 and 5 are from different test suites, so are not in the join result.

[AO Intersection] The aspects regulate the intersection by keeping only tuples whose advice also intersects. The intersection can be computed using a semi-join, which was previously defined.

$$[R_D, R_C, R_A] \cap [S_D, S_C, S_A] = [R_D, R_C, R_A] \ltimes [S_D, S_C, S_A]$$

As an example, consider the intersection of the **Names** relation in Table 6 with the **Aliases** relation in Table 8. The result is the **AliasNames** relation in Table 9. Joe is in the result, but Sue is not since the intersection of her test advice produces no result.

[AO Union] In a union operation, the advice for each pair of unioned tuples must also be computed, and if duplicate tuples are present in the union, then their advice should be combined.

$$[R_D, R_C, R_A] \cup [S_D, S_C, S_A] = \textbf{dup-elim}([R_D \cup S_D, R_C \cup S_C, \textbf{advice-union}(R_A, S_A)])$$

The union utilizes an advice union.

[**Advice union**] Advice union (**advice-union**) computes the union of advice relations. The union utilizes an advice-specific union; examples are listed below for each kind of advice.

- Temporal advice – Computes the temporal union as a single period if the two time periods overlap, otherwise, each time period is kept.

$$\textbf{temporal-union}(\{(i, t, u)\}, \{(j, v, w)\}) = \begin{cases} \{(i, \textbf{min}(t, v), \textbf{max}(u, w))\} & \text{if } (t, u) \text{ overlaps } (v, w) \\ \{(i, t, v), (j, v, w)\} & \text{otherwise} \end{cases}$$

- Test advice – Test suite x always unions with test suite y, i.e., all test advice is retained.

$$\textbf{test-union}(\{(i, x)\}, \{(j, y)\}) = \begin{cases} \{(i, x)\} & \text{if } x = y \\ \{(i, x), (j, y)\} & \text{otherwise} \end{cases}$$

- Privacy advice – The idea is to keep the least private group of the pair, by determining if one group is an ancestor in the partial order, otherwise, both groups should be retained.

Aliases

Name	ID
Joe	8
Sue	9

Data Cuts

ID	RF
8	G
9	H

Temporal Advice

RF	TTBeg	TTEnd
G	2008	now
H	2000	now

Test Advice

RF	Suite
G	15
H	15

Privacy Advice

RF	Group
G	top
H	public

Table 8 Aliases relation

AliasNames

Name	ID	
Joe	1	8

Data Cuts

ID		RF	
1	8	A	G

Temporal Advice

RF		TTBeg	TTEnd
A	G	2008	now

Test Advice

RF		Suite
A	G	15

Privacy Advice

RF		Group
A	G	top

Table 9 Intersection results

AliasOrNames

Name	ID
Joe	1
Sue	3
Fred	4

Data Cuts

ID	RF
1	A
3	C
4	D

Temporal Advice

RF	TTBeg	TTEnd
A	2003	2003
A	2005	2006
A	2007	now
C	2000	now
D	2006	now

Test Advice

RF	Test
C	20
C	15
A	0
D	0

Privacy Advice

RF	Group
A	public
A	admin
C	coder
D	public

Table 10 Union results

$$\textbf{privacy-union}(\{(i, x)\}, \{(j, y)\}) = \begin{cases} \{(i, x)\} & \text{if } \textbf{lca}(x, y) = y \text{ or } x = y \\ \{(i, y)\} & \text{if } \textbf{lca}(x, y) = x \\ \{(i, x), (j, y)\} & \text{otherwise} \end{cases}$$

As an example, consider the union of the **Names** relation in Table 6 with the **Aliases** relation in Table 8. The result is the **AliasOrNames** relation in Table 10. The union adds only advice to the result, in particular increasing the valid time of tuple 3 and adding a new test advice tuple.

The difference operation is the most complicated operation because advice tuples may potentially be "trimmed." For instance, suppose we take the difference of Jennifer's employment in Shoes from 2005-2006 with her employment in Shoes from 2004-2005. Jennifer's advice tuple should be trimmed in the result to represent that she was employed from 2006-2006.

[AO Difference] An aspect-oriented difference operation removes the intersection of the two relations:

$$[R_D, R_C, R_A] - [S_D, S_C, S_A] = \textbf{regulate}([R_D \cup I_D, R_C \cup I_C, \textbf{advice-diff}(R_A, I_A])$$

where

$$[I_D, I_C, I_A] = \textbf{dup-elim}([R_D, R_C, R_A] \cap [S_D, S_C, S_A])$$

NamesNotAliases		
Name	ID	
Joe	1	8
Sue	3	9
Fred	4	@

Data Cuts			
ID		RF	
1	8	A	G
3	9	C	H
4	@	D	@

Temporal Advice			
RF		TTBeg	TTEnd
A	G	2003	2003
A	G	2005	2006
A	G	2007	2007
D	@	2006	now
C	H	now	now

Test Advice		
RF		Suite
C	H	20
A	G	0
D	@	0

Privacy Advice		
RF		Group
A	G	public
A	G	admin
C	H	coder
D	@	public

Table 11 Difference results

is the intersection of the two relations. It may seem odd that we are increasing the "data" tuples, but the tuples will be regulated by the difference applied to the advice.

[**Advice difference**] Advice difference (**advice-diff**) computes the "difference" in advice tuples as follows. First, compute the portion of each advice tuple which is different than an advice tuple in the intersection.

$PossiblePortions$ = **advice-diff** (R_A, I_A)

The aspect-specific difference, **advice-diff**, is described in detail below. Next, find the intersection tuples that overlap some possible portion:

$OverlapPortions$ = **advice-semijoin**($PossiblePortions$, I_A)

where **advice-semijoin** is an advice-specific semi-join, similar to the advice join described earlier in this paper. Finally, just keep the possible portions that do not overlap.

advice-diff(R_A, I_A) = $PossiblePortions - OverlapPortions$

Below are some advice-specific difference operations.

- Temporal advice – Computes the temporal difference, which is the portion that does not overlap.

$$\textbf{temporal-difference}(\{(i,t,u)\},\{(j,v,w)\}) = \begin{cases} \{(i,j,\textbf{min}(t,v),\textbf{min}(u,w)-1), \\ \quad (i,\textbf{max}(t,v)+1,\textbf{max}(u,w))\} \\ \qquad \text{if } (t,u) \text{ overlaps } (v,w) \\ \{(i,j,t,u)\} \quad \text{otherwise} \end{cases}$$

- Test advice – Test suite x is different from test suite y as follows.

$$\textbf{test-difference}(\{(i,x)\},\{(j,y)\}) = \begin{cases} \{\} & \text{if } x = y \\ \{(i,j,x)\} & \text{otherwise} \end{cases}$$

- Privacy advice – A partial order difference determines if the subtractor is more private; if so then we keep the privacy advice.

$$\textbf{privacy-difference}(\{(i,x)\},\{(j,y)\}) = \begin{cases} \{\} & \text{if } \textbf{lca}(x,y) = x \\ \{(i,j,x)\} & \text{otherwise} \end{cases}$$

As an example, consider taking the difference of the **Names** relation of Table 6 and the **Aliases** relation of Table 8. The result is shown in Table 11. First the intersection is computed (Table 9). Only Joe is in the intersection. Next Joe's advice is trimmed, removing the intersecting advice from **Names**; in this case, only his valid time is trimmed reducing his time in Admin to just 2007.

4.2 Query Aspects

A query (or parts thereof) can also be aspected. A query aspect represents a constraint on the relations that participate in the query.

[**Filter**] Let a query, Q, involve a relation, [R_D, R_C, R_A], and be aspected by a perspective, Q_A. Then the **filter** operation constrains R_D to tuples that have advice consistent with the query perspective prior to evaluating the query as follows.

$$\textbf{filter}(Q, Q_A, [R_D, R_C, R_A]) = Q(\textbf{synch}(\textbf{regulate}([R_D, R_C \ltimes \textbf{ao-filter}(R_A, Q_A), R_A])))$$

Each advice tuple passes through an advice-specific filter operation, which leaves the tuple unchanged, trims the tuple, or regulates it.

[**Advice filter**] Advice filter (**advice-filter**) filters advice tuples. The filtering is aspect-specific. Example filters are listed below.

- Temporal advice – Computes the temporal overlap, if any, as follows.

$$\textbf{temporal-filter}(\{(i,t,u)\},\{(v,w)\}) = \begin{cases} \{(i,\textbf{max}(t,v),\textbf{min}(u,w))\} \\ \qquad \text{if } (t,u) \text{ overlaps } (v,w) \\ \{\} \quad \text{otherwise} \end{cases}$$

- Test advice – Test suite x is filtered by test suite y as follows.

$$\textbf{test-filter}(\{(i,x)\},\{(y)\}) = \begin{cases} \{(i,x)\} & \text{if } x = y \text{ or } x = 0 \\ \{\} & \text{otherwise (if } x \neq y) \end{cases}$$

- Privacy advice – Privacy group x is filtered by privacy group y as follows.

$$\textbf{privacy-filter}(\{(i,x)\},\{(y)\}) = \begin{cases} \{(i,x)\} & \text{if } \textbf{lca}(x,y) = x \\ \{\} & \text{otherwise} \end{cases}$$

For example, suppose the following query is evaluated from the default aspect perspective, P (time is `now`, test suite `0`, and privacy group `public`).

$$\pi_{\textbf{Sal}}(\sigma_{\textbf{Name} = \text{'Joe'}}(\textbf{Employees}))$$

The query is translated to the corresponding aspect-oriented query, where **Employees** is [$\mathbf{E}_D$, $\mathbf{E}_C$, $\mathbf{E}_A$], and with selection and projection pushed as far as possible into the query.

$$\textbf{dup-elim}(\textbf{synch}(\textbf{regulate}([\pi_{B,\textbf{ID}}(\sigma_{\textbf{Name}=\text{'Joe'}}(\mathbf{E}_D)), \mathbf{E}_C \ltimes \textbf{ao-filter}(\mathbf{E}_A, P), \mathbf{E}_A])))$$

Salary	
Sal	ID
40K	1

Data Cuts	
ID	RF
1	A

Temporal Advice		
RF	TTBeg	TTEnd
A	2003	2003
A	2005	2006

Test Advice	
RF	Test
A	0

Privacy Advice	
RF	Group
A	public

Table 12 Joe's public salary history

Evaluating the query yields the empty result. Both of Joe's tuples are filtered because Joe worked in Shoes only prior to now, and Joe's current employment in Admin is visible only to the privacy group `admin`.

Query aspects are powerful and we assume that the default perspective is managed by the DBMS rather than by a user. A user acquires a default perspective when they first log in (so only via a log in procedure can a user be a part of any privacy group other than `public`). The user can subsequently change various aspects of their default perspective by setting session environment variables, for instance, the user should be able to change the temporal component of their default query aspect. Suppose for instance that the user changes their default temporal aspect to be (`2000`, `now`), and re-evaluates the query given above. The result is shown in Table 12.

Independent of the default query perspective, a query can also have data aspects, in which case the aspects are composed with the perspective to filter the data prior to evaluating the query. For instance, suppose the query is part of a test suite. Then the query would be aspected with test advice, e.g., (`test suite 20`). When evaluated, the query would produce an empty result since Joe's tuple is not part of the test suite.

4.3 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 operations like **synch**, are implemented in the relational algebra. 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.

An aspect-oriented selection incurs a **synch** operation, which involves three semi-joins raising the cost from $O(D)$ without aspects to $O(D) + O(\mathbf{max}(A,C)*D)$ where $O(\mathbf{max}(A,C)*D)$ is the cost of the largest semi-join in the **synch**.

The complexity of a projection that does not eliminate duplicates remains unchanged, but duplicate elimination adds a cost of $O((C*D)^2) + O(C*A)$ to compute the duplicate data tuples and the advice relation.

Joins and Cartesian product involve additional **synch** and **regulate** operations and aspect-specific behaviors for pairs of advice tuples, which we assume can be computed in constant time (like all of the behaviors given in this paper). The cost of a join increases from $O(D^2)$ to $O(D^2) + O(\mathbf{max}(A,C)*D) + O((C^2*A^2)*D)$, where the latter term is the cost of a **regulate**.

Finally, union and difference also potentially involve duplicate elimination adding a term of $O((C*D)^2) + O(C*A)$ to their complexity.

4.4 Query Optimization

There are several possibilities for query optimization in the aspect-oriented relational algebra. Standard optimization rules remain largely unaffected, for instance, restrictions can be pushed into binary operations, such as joins, and the order of restrictions can be permuted. Several query optimization rules (a non-exhaustive list) for aspect-oriented operations are listed below.

- Synchronizations can be collapsed.

  $\mathbf{synch}(\mathbf{synch}([R_D, R_C, R_A])) = \mathbf{synch}([R_D, R_C, R_A])$

- Synchronizations can be delayed.

  $F(\mathbf{synch}([R_D, R_C, R_A])) = \mathbf{synch}(F([R_D, R_C, R_A]))$

- Synchronizations can be eliminated (extraneous advice plays no role in query evaluation).

  $\mathbf{synch}(F([R_D, R_C, R_A])) = F([R_D, R_C, R_A])$

- Regulation can be collapsed.

  $\mathbf{regulate}(\mathbf{regulate}([R_D, R_C, R_A])) = \mathbf{regulate}([R_D, R_C, R_A])$

- Regulation can be pushed into duplicate elimination.

  $\mathbf{regulate}(\mathbf{dup\text{-}elim}([R_D, R_C, R_A])) = \mathbf{dup\text{-}elim}(\mathbf{regulate}([R_D, R_C, R_A]))$

4.5 Aspect-oriented Constraints

Constraints can also be aspected to specify the range of advice for which the constraint holds. All constraints, by default, apply only to the default perspective of the data. Consider a primary key constraint. When the constraint is evaluated after a data insertion or modification, it ensures that each tuple in a data relation can be uniquely identified. For example, suppose that the key of the **`Departments`** relation in Table 4 is the **`Dept`** attribute. At first glance, this does not appear to be a valid key as two tuples have the same **`Dept`** value (`Shoes`). But the key constraint is aspected, by default, to apply only to live data, at the current time, and with no privacy. Prior to evaluating the constraint, the **`Departments`** relation is filtered to remove tuples not in the default perspective, and then the constraint is evaluated.

By aspecting a constraint, it can be changed to apply to a different data set. For example, suppose that the key of the **`Departments`** relation in Table 4 is extended to include the **`Loc`** attribute, and that the database designers decide to add the constraint that the new key is applicable at all times. The new key would be aspected with the temporal advice (`beginning`, `forever`), asserting that it applies to, from a temporal perspective, all of the data.

Finally, each constraint can have aspect-specific behavior. For instance when specifying a temporal key, the temporal advice can add a restriction that no two tuples can have the same key value(s) at the same time, allowing the possibility that the key could be the same at different times.

	0%	33%	66%	100%
index select	2	25	25	25
select	1550	1573	1573	1573
join	37913	55343	181368	314226
projection	1550	21814	41949	89505

Table 13 Logical block I/O

In summary, to the evaluation of a constraint is added 1) a filter that applies a constraint aspect to remove tuples from consideration prior to the evaluation of the constraint, and 2) an aspect-specific check that is performed after evaluation of the constraint to further check the data.

4.6 Experiment to Measure Cost

We implemented (part of) an SQL-to-SQL translator that takes an SQL query (written in the MySQL dialect of SQL) and translates it to an aspected SQL query, i.e., a layered approach to implementation of aspected-oriented relations and queries. We used the translator to translate four simple queries, listed below, on a scaled-down version of the Internet Movie Database. The database has only two tables, an Actor table with information about actors and a Casting table, which relates an actor to a movie in which the actor was cast. Both tables are approximately 0.5GB in size, and there are indexes for the keys of each table.

1) Select the actor with id 12 (point query using an index).
2) Select the actor with name 'Bruce Willis' (no index).
3) Join the actor and casting tables on the actor id.
4) Project the name of each actor.

We experimented with the aspected queries as follows. We varied the percentage of aspected data, from 0% to 100%. The 0% case is the "unaspected" case, i.e., the queries are the original SQL queries. In the 100% case, every tuple was aspected with a single aspect. In the 100% case, the advice and data cuts added 0.8GB. We also created indexes for the data cuts and advice tables.

We then evaluated the four queries and measured the logical block I/O (which is a much better measure of the actual cost of the query than time since it factors out caching effects, a query that logically reads half as many blocks as another query often runs in the same time since the time is driven by the number of physical vs. logical reads, in other words, we are using the "worst-case" measure for showing the cost of aspect-oriented data). The raw results are given in Table 13, and a chart of the results in Figure 6. The "slowdown" is the ratio of the block I/O in the aspected case vs. the unaspected case. So a slowdown of 25 represents twenty-five times more block I/O.

The slowdown is reasonable for the selection and the join, but the cost blows out for projection. The culprit is duplicate elimination in the projection. MySQL lacks a difference operator, so a difference has to be implemented using a subquery, and the subquery involves the Cartesian product of two joins. The indexed-based selection is also expensive, but mostly because a point query using an index is extremely fast; observe that the raw cost of an aspected-indexed select is still very low.

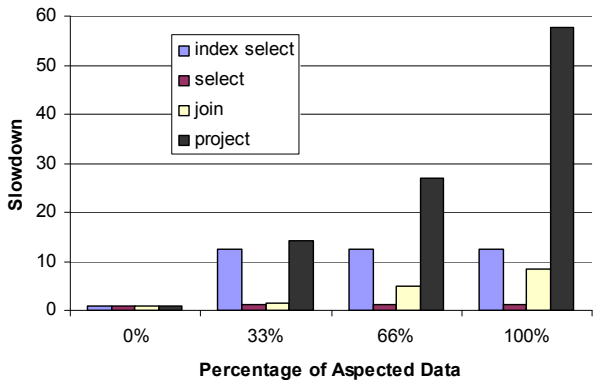

Figure 6 Slowdown

5. 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 the AUCQL language for querying different kinds of metadata in a semi-structured data model [9], which was later developed into a query language, MetaXQuery, for XML data [16],[17],[18]. This paper in contrast focuses on the relational model.

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 [26], object-oriented [27], 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 [22]. The SPARCE system wraps or superimposes a data model with a layer of metadata [23]. 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 [3],[21]. Our approach differs from these systems by focusing on the relational data model and relational algebra extensions 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 [29]. The Dublin Core is a commonly

used classification standard [8]. Commercial [1] and research systems [2] to manage (descriptive) metadata collections have been developed. Methods to automatically extract content-related metadata have also been researched [14],[20]. 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 with metadata.

6. CONCLUSIONS AND FUTURE WORK

This paper proposes adapting a popular software engineering design technique to database management. Aspect-oriented programming is used to add cross-cutting concerns to an already existing application, without having to reprogram the application. Cross-cutting concerns are also present in data management. We presented a design for aspect-oriented relations and queries, which allow a database to be re-engineered with cross-cutting data concerns. We proposed annotating or tagging 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 relational algebra queries could be woven with aspect-specific behaviors to correctly implement a cross-cutting concern.

We have several ideas for future work. The first is to clarify the mutator and constructor roles for aspects, in particular, designing new query syntax to specify mutators for converting advice to data. Second we have not yet considered how aspects impact grouping for aggregates (e.g., grouping and aggregation in temporal relational databases have been extensively researched [31]). Third, we have yet to consider data modification. And, finally, we believe that GUI tools are critical to making aspects easy to use.

REFERENCES

[1] Amaxus CMS, "An Overview of the Amaxus XML Content Mangement System". Available at `www.boxuk.com`.

[2] Michelle Baldonado, Chen-Chuan K. Chang, Luis Gravano, Andreas Paepcke, "Metadata for Digital Libraries, Architecture and Design Rationale," in *ACM/IEEE JODL*, pp. 47-56, 1997.

[3] Phil A. Bernstein, "Applying Model Management to Classical Meta Data Problems," in *CIDR* 2003, pp. 209-220.

[4] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya, "An Annotation Management System for Relational Databases," in *VLDB*, 2004, pp. 900–911.

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

[6] Peter Buneman, Adriane Chapman, and James Cheney, "Provenance Management in Curated Databases," in *SIGMOD*, 2006. Chicago, IL, pp. 539–550.

[7] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan, *Archiving Scientific Data*. ACM Transactions on Database Systems, 2004. 29(1): 2–42.

[8] Dublin Core Metadata Initiative. "Dublin Core Metadata Element Set, Version 1.1: Reference Description", DCMI Recommendation, June 2003. At `dublincore.org`.

[9] Curtis E. Dyreson, Michael H. Böhlen, and Christian S. Jensen. "Capturing and Querying Multiple Aspects of Semi-structured Data," in *VLDB* 1999, pp: 290-301.

[10] Curtis Dyreson, Richard T. Snodgrass, Faiz Currim, Sabah Currim and Shailesh Joshi. *Weaving temporal and reliability aspects into a schema tapestry*. Data and Knowledge Engineering, 63(3), December 2007, pp. 752-773.

[11] Curtis Dyreson and Omar Florez. "Data Aspects in a Relational Database," in CIKM 2010, to appear (short paper, 4 pages).

[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] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass, "Unification of Temporal Data Models," in *ICDE*, Vienna, Austria, Apr. 1993, pp. 262-271.

[16] Hao Jin and Curtis E. Dyreson, "Grouping in MetaXQuery", *Proc. WISE* 2004, LNCS 3306 , Nov. 2004, pp. 688-693.

[17] Hao Jin and Curtis E. Dyreson, "Sanitizing Using Metadata in MetaXQuery," in *ACM SAC*, pp. 1732-1736. Santa Fe, NM, March 2005.

[18] Hao Jin and Curtis Dyreson, "Supporting Proscriptive Metadata in an XML DBMS," in *DEXA* Turin, Italy, September 2008, pp. 479-492.

[19] Anastasios Kementsietsidis, Floris Geerts, and Diego Milano, "MONDRIAN: Annotating and Querying Databases through Colors and Blocks," in *ICDE*, 2006, p. 82.

[20] Dongwon Lee and Yousub Hwang, *Extracting Semantic Metadata and its Visualization*. ACM Crossroads, 2001, 7(3): 19-27.

[21] Sergey Melnik, E. Rahm, E., and Phil A. Bernstein. "Rondo: A Programming Platform for Generic Model Management". In *ACM SIGMOD* 2003, pp. 193-204.

[22] 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.

[23] Sudarshan Murthy, David Maier, Lois M. L. Delcambre, Shawn Bowers. "Superimposed Applications using SPARCE," in *ICDE*: 861. Boston, MA, USA, March 2004.

[24] Awais Rashid and N. Loughran, "Relational Data-base Support for Aspect-Oriented Programming," in *NetObjectDays Conference*. LNCS. Volume 2591, 2002, pp. 233-247.

[25] Awais Rashid, "Aspect-Oriented Programming for Database Systems," in *Aspect-Oriented Software Development*, 2004.

[26] Richard T. Snodgrass (Ed.). *The TSQL2 Temporal Query Language*. Kluwer, 1995. 629 pages.

[27] Richard T. Snodgrass, "Temporal Object-oriented Databases: a Critical Comparison," in *Modern Database Systems: the Object Model, Interoperability*, and Beyond. Addison Wesley Pub. Co., 1995. pp. 386-408.

[28] Richard T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann. 1999. 540 pages.

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

[30] Jennifer Widom, "Trio: A System for Integrated Management of Data Accuracy, and Lineage," in *CIDR*, 2005. pp. 262–276.

[31] Donghui Zhang, Alexander Markowetz, Vassilis Tsortas, Dimitrios Gunopulos, and Bernhard Seeger. *On Computing Temporal Aggregates with Range Predicates*. ACM TODS, 2008.