IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs*

Daco C. Harkes[1], Danny M. Groenewegen[2], and Eelco Visser[3]

1 **Delft University of Technology**
`d.c.harkes@tudelft.nl`
2 **Delft University of Technology**
`d.m.groenewegen@tudelft.nl`
3 **Delft University of Technology**
`e.visser@tudelft.nl`

Abstract

Derived values are values calculated from base values. They can be expressed in object-oriented languages by means of getters calculating the derived value, and in relational or logic databases by means of (materialized) views. However, switching to a different calculation strategy (for example caching) in object-oriented programming requires invasive code changes, and the databases limit expressiveness by disallowing recursive aggregation.

In this paper, we present IceDust, a data modeling language for expressing derived attribute values without committing to a calculation strategy. IceDust provides three strategies for calculating derived values in persistent object graphs: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. We have developed a path-based abstract interpretation that provides static dependency analysis to generate code for these strategies. Benchmarks show that different strategies perform better in different scenarios. In addition we have conducted a case study that suggests that derived value calculations of systems used in practice can be expressed in IceDust.

1998 ACM Subject Classification D.3.2 Data-flow languages

Keywords and phrases Incremental Computing, Data Modeling, Domain Specific Language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.11

1 Introduction

Derived values are values calculated from base values (provided by users). When a base value changes, the derived values depending on it should change accordingly. Hence, the important events for interacting with derived values are writes to base values and reads of derived values. This specification of derived values leaves room for multiple strategies for calculating derived values. Derived values can be calculated when they are read or they can be cached and updated when the underlying base values change. The performance of these strategies depends on characteristics of the data model and usage scenarios. When neither of these calculation strategies provides acceptable performance, updates can be postponed, temporarily allowing reads to return outdated derived values.

Object-oriented programming languages express derived values through getters containing code that calculates a derived value, implying that the derived value is recalculated each time it is read. Switching to calculating the derived value when an underlying value changes,

* This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

or switching to eventually calculating the derived value, requires invasive code changes. By contrast, most relational databases allow easy switching between calculate-on-read and calculate-on-write as they support both materialized and non-materialized views for calculating derived values. However, relational databases only provide limited expressiveness for recursion, and do not support eventual calculation of derived values. Datalog provides more expressiveness than relational database views, but also limits recursion, and does not support eventual calculation of derived values.

This paper presents IceDust, a language supporting definition of attributes with derived values without committing to a calculation strategy. The compiler provides three different implementation strategies for calculating derived values: (1) Calculate-on-Read, which calculates the derived value every time it is read, (2) Calculate-on-Change, which maintains a cache incrementally by calculating the derived value every time an underlying value is changed, and (3) Calculate-Eventually, which schedules calculations of derived values, and thus sacrifices consistency temporarily. All these strategies allow unrestricted recursion, but do not provide termination guarantees. In particular, our contributions are:

- The IceDust language for data modeling with derived values (Section 2)
- A formal analysis of the dependencies in IceDust programs (Section 3)
- Three calculation strategies to satisfy different non-functional requirements (Section 4)
- Benchmarks showing the performance differences between the strategies (Section 5)
- A case study of migrating a custom eventual calculation system to IceDust (Section 6)

2 Declarative Data Modeling with Derived Values

This section discusses three issues of data modeling with derived values in object-oriented programming languages and show how data modeling in IceDust addresses these issues and leads to concise specifications. As running example we use (an aspect of) a learning management system in which students solve assignments. Figure 1 shows a Java implementation with classes `Assignment` and `Question`, where assignment represents a collection of questions and its progress is the average of the progress on the individual questions.

Bidirectional Relations. Object-oriented languages model bidirectional relations as properties in classes on both sides of the relation. Keeping these properties consistent requires code that has to be repeated for every bidirectional relation. Figure 1 includes five methods concerned with keeping the relation `Assignment-Question` consistent on updates: `setAssignment`, `addQuestion`, `removeQuestion`, `_addQ`, and `_remQ`. This pattern is identical for all one-to-many relations, but cannot be abstracted over in an object-oriented language. To avoid such boilerplate code, IceDust supports *bidirectional relations* as a language feature:

```
entity Assignment { }
entity Question { }
relation Assignment.questions * <-> 1 Question.assignment
```

These bidirectional relations are named on both sides of the relation, inspired by Object-Role Modeling [17]. The IceDust compiler keeps both sides of the association consistent without additional boilerplate code.

Native Multiplicities. Explicit collections and possible null values in object-oriented languages lead to boilerplate code to deal with the cardinalities of values returned by an expression. Operators in object-oriented languages are defined for operands with a cardinality of exactly one. Safely applying an operator to a nullable operand, requires a `null`-check.

```
public class Assignment {
  public Double getAverageProgress() {  return calculateAverageProgress();  }
  public Double calculateAverageProgress() {
    Stream<Double> progresss =
      questions.stream().map(q -> q.getProgress()).filter(p -> p != null);
    OptionalDouble average = progresss.mapToDouble(p -> p).average();
    return average.isPresent() ? average.getAsDouble() : null;
  }
  private Collection<Question> questions;
  public Collection<Question> getQuestions(){ return new HashSet<>(questions); }
  public void addQuestion(Question q) {  q.setAssignment(this);  }
  public void removeQuestion(Question q) {  q.setAssignment(null);  }
  protected void _addQ(Question q) {  questions.add(q);  }
  protected void _remQ(Question q) {  questions.remove(q);  }
}
public class Question {
  private Assignment assignment;
  public Assignment getAssignment() {  return assignment;  }
  public void setAssignment(Assignment a) {
    if(assignment != null) { assignment._remQ(this); }
    if(a != null) { a._addQ(this); }
    assignment = a;
  }
  private Double progress;
  public Double getProgress() {  return progress;  }
  public void setProgress(Double p) {  progress = p;  }
}
```

Figure 1 Object-oriented assignment system (implementation strategy: Calculate-on-Read).

Applying an operator to a collection of values, requires lifting it to a map. For example, accessing the progress of each individual question in Figure 1 is encoded as

```
questions.stream().map(q -> q.getProgress()).filter(p -> p != null)
```

IceDust adopts *native multiplicities* [18], delegating the handling of the cardinality of values returned by an expression to the language. For example, retrieving the progress for all questions is simply a projection:

```
questions.progress
```

Language constructs to get expressions of cardinality exactly one, such as `map`, `filter`, and `!= null`, are no longer required, as the type system knows how many values an expression returns (multiplicity denoted by `~`, where `*` is [0,n), `+` is [1,n), `?` is [0,1], and `1` is [1,1]):

```
mathAssignment                          // : Assignment ~ 1
mathAssignment.questions                // : Question   ~ *
mathAssignment.questions.progress       // : Float      ~ *
avg(mathAssignment.questions.progress) // : Float      ~ ?
```

Sometimes it is still necessary to reflect explicitly on the cardinality of a value. To that end one can use the `count` operator, for example, for counting the number of questions:

```
count(questions)
```

Reflection on the cardinality of values is also often used to select an alternative if no value is present. For specifying alternatives the choice operator (`<+`) can be used:

```
input <+ myDefault //if (count(input) > 0) input else myDefault
```

```
//Take all Code from Calculate–on–Read and add/change the following:
public class Assignment {
 private Double cachedAvgProgress;
 public Double getAverageProgress() {  return cachedAvgProgress;  }
 public void updateAvgProgress(){cachedAvgProgress = calculateAverageProgress();}
 protected void _addQ(Question q){ questions.add(q);    updateAvgProgress(); }
 protected void _remQ(Question q){ questions.remove(q); updateAvgProgress(); }
}
public class Question {
 public void setProgress(Double p){progress = p; assignment.updateAvgProgress();}
}
```

Figure 2 Object-oriented assignment system (implementation strategy: Calculate-on-Write).

Derived Value Attributes. Last but not least, object-oriented languages force early decisions on the implementation strategy for calculating derived values. In an object-oriented language, a derived value calculation can be expressed with a method that computes the value. However, this encodes a Calculate-on-Read implementation strategy. For cheap calculations or calculations that are done infrequently that may be fine. But for others, it may be necessary to cache the calculated value. Such an alternative computation strategy requires an invasive redefinition of the implementation. For example, Figure 2 implements a caching strategy for the `getAverageProgress` computation of Figure 1. Instead of computing the average on read, it is computed on writes of `progress` and `questions`. For this example, the impact of the change was relatively minor because in Figure 1 we had already factored `calculateAverageProgress` into a separate method. However, in real code the impact is typically non-trivial. In particular, because the introduction of a cached value requires taking into account all of its dependencies in order to trigger recomputation on any change that affects it. For example, `averageProgress` depends on `progress` and `questions`. Thus, `setProgress`, `_addQ`, and `_remQ` need to trigger recalculation of `averageProgress`.

IceDust provides *derived value attributes* for declarative specification of the value of attributes in terms of other attributes without committing to an implementation strategy:

```
entity Assignment {  avgProgress : Float? = avg(question.progress)  }
```

This separation of concerns enables focusing on specification of the logic of the derived value. The derived value expression specifies what the value of the attribute should be. Derived value attributes in IceDust support recursive definitions, including recursive aggregation (which is not supported in materialized views or stratified Datalog):

```
entity Assignment {  progress : Float? = avg(children.progress)  }
relation Assignment.parent ? <–> * Assignment.children
```

Language Definition. We have combined the ideas for improving data modeling by means of bidirectional relations, native multiplicities, and derived value attributes in the design of the experimental IceDust language. In order to embed IceDust data models in full fledged web applications the compiler generates code in the WebDSL programming language [36].

The design of IceDust was heavily influenced by previous work on relations as a first-class language construct. From Rumer [4] and RelJ [5] we adopt the restriction to binary, bidirectional relations. From the Relations language [18] we adopt the syntax of declarations and property access, integrating multiplicities in relations. Multiplicities derive from the work of Steimann [32, 33], which extends an object-oriented language with multiplicity annotations to support uniform treatment of values of different cardinality and avoids the boilerplate code

```
Program ::= model Entity* Relation*
Entity ::= entity E { Attribute* }
Relation ::= relation E.r m <-> m E.r
Attribute ::= a : T m
            | a : T m = e
            | a : T m = e (default)
T ∈ PrimitiveType ::= Boolean | Int | Float | Datetime | String
m ∈ Multiplicity ::= ? | 1 | * | * (ordered) | + | + (ordered)
e ∈ Expr ::= f ( e ) | e1 ⊕ e2 | ! e | e1 ? e2 : e3 | e . a | e . r | e as T | this | Literal
Literal ::= true | false | null | int | float | datetime | string
f ∈ AggrOp ::= min | max | avg | sum | concat | count | conj | disj
⊕ ∈ BinOp ::= + | - | * | / | % | && | || | > | >= | < | <= | == | =! | <+ | ++
```

Figure 3 Syntax of the IceDust data modeling language.

required to support different multiplicities. We adopt the integration of such multiplicities in the type system (dubbed native multiplicities) of the Relations language [18].

Figure 3 defines the grammar of IceDust. `E`, `a` and `r` are entity, attribute and relation names respectively. An IceDust program consists of entities and relations. Entities contain three kinds of attributes: 'normal' attributes (`a : T m`), derived value attributes (`a : T m = e`), and default value attributes (`a : T m = e (default)`). Users can set the value of 'normal' attributes and read the value later. Users cannot set the value of derived value attributes, but they can read the value calculated with expression `e`. Finally, users can set the value of default value attributes and read the value later, but they can also set the value to null (or not set it all) and read the value calculated by `e`. Attributes are limited to primitive types, as an entity type would create a unidirectional relation (which would give problems in the dependency analysis). A relation defines a bidirectional relation with a name and multiplicity on both sides. The domain of the expression language is primitive types (Boolean, Int, Float, Datetime and String) and objects. The language covers object graph navigation and calculations over the primitive types. Note the aggregation operations over primitive types to deal with multiplicities `*` and `+`. The expression language is expressive enough to specify derived values, and simple enough to allow multiple implementation strategies.

The type system of IceDust is mostly concerned with native multiplicities. A type in IceDust is a tuple of two lattice values (Figure 4). The primitive types, the declared entities, the `null` and error type form a lattice. Multiplicity and ordering form another lattice. During derived value calculation all values are read-only in IceDust. A value which is lower or equal in both lattices can be used in a place where a certain type, multiplicity, and ordering is expected. For example, a `Float?` can be supplied where a `Float*` is expected.

3 Dependency and Data Flow Analysis

IceDust specifications define the value of attributes in terms of other attributes. These definitions are declarative in the sense that they abstract from the implementation strategy used to calculate the values. In the next section we define three implementation strategies for the calculation of attribute values: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. The latter two strategies require dependency and data flow information. In this section we define the computation of dependencies between attributes by means of a path-based abstract interpretation of expressions. Since IceDust does not have statements, data

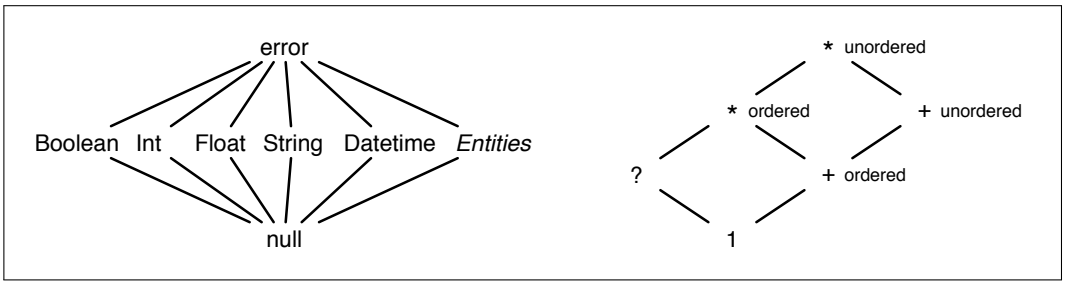

Figure 4 IceDust's type lattice (left), and multiplicity and ordering lattice (right).

flow coincides with control flow, and the data flow relation is the inverse of the dependency relation. The static dependency and data flow analysis is performed in three steps: (1) compute attribute dependencies by means of path-based abstract interpretation, (2) reverse the dependencies to construct the data flow relation, and (3) organize the data flow in a graph and extract strongly connected components with a topological ordering.

Example. To illustrate the analysis we use a more complex version of the learning management system (Figure 5). This example features a tree of assignments, and grade calculation logic for submissions by students to these assignments. `Assignments` are structured in a tree through the `parent-children` relation. A `Submission` represents the solution created for an assignment by a student. Leaf submissions are graded by assigning a grade to the `grade` attribute (overriding the default value), while the grades of non-leaf submissions depend (indirectly) on the grades of their child submissions:

```
grade      : Float?  = if(childPass) childGrade else null (default)
pass       : Boolean = grade >= (assignment.minimum<+0.0) && childPass <+ false
childGrade : Float?  = avg(children.grade)
childPass  : Boolean = conj(children.pass)
```

Note that students only receive a grade for a collection-submission if all of the child submissions are `pass`, and a submission is only a `pass` when its `grade` is above the `minimum` assignment grade and all its children pass. The `minimum` for an assignment is optional, without `minimum` the grade should be higher than or equal to 0.0, which is always true. Submissions are (one of) the `best` of an assignment when their `grade` equals the highest grade. Finally, every assignment has an average grade and pass percentage. This example is interesting for dependency analysis as it features mutually recursive definitions of `grade`, `pass`, `childGrade` and `childPass` through the `parent-child` relation of `Submission`.

Step 1: Dependencies. The dependencies of an attribute are all the attributes and relations that are needed to compute the derived value of that attribute. The dependencies are reachable from the entity of the attribute via a path. A dependency is denoted by $(Ent.Attr \leftarrow \pi)$, where $Ent.Attr$ is the attribute and π is the path to an attribute or relation.

Computing the dependencies requires extracting paths from expressions defining derived values. The *path-based abstract interpretation* relation (Figure 6) defines the dependency paths of an expression. We use the notation $Expr \searrow \{\pi\}\{\rho\}$, where $Expr$ is the expression that is abstractly interpreted, and $\{\pi\}$ and $\{\rho\}$ are the sets of paths defined by the abstract interpretation. The paths in $\{\pi\}$ are extensible, while the paths in $\{\rho\}$ are not. All paths start with `this` [This] or with object graph navigation [NavStart]. When navigating the object graph by means of `e.attrOrRel` all dependency paths $\{\pi\}$ in `e` are extended with

```
entity Assignment {
  name       : String
  question   : String
  minimum    : Float?
  avgGrade   : Float? = avg(submissions.grade)
  passPerc   : Float? = sum(submissions.passInt) / count(submissions) * 100.0
}
entity Student {
  name       : String
}
entity Submission {
  name       : String  = assignment.name + " " + student.name
  answer     : String?

  grade      : Float?  = if(childPass) childGrade else null (default)
  pass       : Boolean = grade >= (assignment.minimum<+0.0) && childPass <+ false
  childGrade : Float?  = avg(children.grade)
  childPass  : Boolean = conj(children.pass)

  passInt    : Int     = if(pass) 1 else 0
  best       : Boolean = grade == max(assignment.submissions.grade) <+ false
}
relation Assignment.parent     ? <-> * Assignment.children
relation Submission.parent     ? <-> * Submission.children
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
```

Figure 5 Example program for dependency analysis.

`attrOrRel` [Nav]. The `if` [If] only allows extension of paths in the second and third operand, so Π_1 is passed to $\{\rho\}$. Operators with multiple operands take the union of the paths of their operands [Op], unary operators pass on paths [Not,Cast,Aggr], and literals do not contain any paths at all [Literal]. Path-based abstract interpretation of the expression defining `pass`

```
grade >= (assignment.minimum <+ 0.0) && childPass <+ false
```

produces a set containing the following paths:

```
grade
assignment.minimum
childPass
```

The *dependencies* relation (Figure 6) defines the dependencies of an attribute, entity and program. We use the notation $Attr|Ent|Prog \searrow\searrow \{(Ent.Attr \leftarrow \pi)\}$, where $Attr|Ent|Prog$ is an attribute, entity or program, and $\{(Ent.Attr \leftarrow \pi)\}$ is a set of dependencies. When an attribute depends on a value at the end of a path, it also depends on the relations en route. So, the rule for attributes [Att] takes the transitive prefix of the paths of its expression. As paths are concatenated later, and a `this` keyword in the middle would produce an invalid path, the `this` is removed from paths. As an example, the dependencies of `pass` are:

```
(Submission.pass ← grade)
(Submission.pass ← assignment.minimum)
(Submission.pass ← assignment)
(Submission.pass ← childPass)
```

The dependencies in for the individual attributes together constitute the dependencies for a full program [Ent,Prog].

Path-based abstract interpretation $\boxed{Expr \searrow \{\pi\}\{\rho\}}$

$$\frac{}{\text{this} \searrow \{\text{this}\}\{\}}\ [\text{This}] \qquad \frac{e \in Literal}{e \searrow \{\}\{\}}\ [\text{Literal}]$$

$$\frac{}{attrOrRel \searrow \{attrOrRel\}\{\}}\ [\text{NavStart}] \qquad \frac{e \searrow \Pi\ \mathrm{P}}{!\ e \searrow \Pi\ \mathrm{P}}\ [\text{Not}]$$

$$\frac{e \searrow \Pi\ \mathrm{P}}{e\ .\ attrOrRel \searrow \{\pi\ .\ attrOrRel \mid \pi \in \Pi\}\ \mathrm{P}}\ [\text{Nav}] \qquad \frac{T \in PrimitiveType \quad e \searrow \Pi\ \mathrm{P}}{e \text{ as } T \searrow \Pi\ \mathrm{P}}\ [\text{Cast}]$$

$$\frac{\oplus \in BinOp \quad e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2}{e_1 \oplus e_2 \searrow \Pi_1 \cup \Pi_2 \quad \mathrm{P}_1 \cup \mathrm{P}_2}\ [\text{Op}] \qquad \frac{f \in AggrOp \quad e \searrow \Pi\ \mathrm{P}}{f(e) \searrow \Pi\ \mathrm{P}}\ [\text{Aggr}]$$

$$\frac{e_1 \searrow \Pi_1\ \mathrm{P}_1 \quad e_2 \searrow \Pi_2\ \mathrm{P}_2 \quad e_3 \searrow \Pi_3\ \mathrm{P}_3}{e_1\,?\,e_2 : e_3 \searrow \Pi_2 \cup \Pi_3 \quad \Pi_1 \cup \mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3}\ [\text{If}]$$

Dependencies $\boxed{Attr|Ent|Prog \searrow\!\searrow \{(Ent.Attr \leftarrow \pi)\}}$

$$\frac{e \searrow \Pi\ \mathrm{P} \quad E = \text{entity-of}(attr) \quad \Pi_2 = \bigcup \{\text{trans-pref}(\text{remove-this}(\pi)) \mid \pi \in \Pi \cup \mathrm{P}\}}{attr : T\ m\ \{= e, = e\ (\text{default})\} \searrow\!\searrow \{(E.attr \leftarrow \pi) \mid \pi \in \Pi_2\}}\ [\text{Att}]$$

$$\frac{}{\text{entity } t\ \{a^*\} \searrow\!\searrow \bigcup \{dep \mid a \searrow\!\searrow dep,\ a \in a^*\}}\ [\text{Ent}]$$

$$\frac{}{\text{model } E^*\ R^* \searrow\!\searrow \bigcup \{dep \mid E \searrow\!\searrow dep,\ E \in E^*\}}\ [\text{Prog}]$$

$\text{remove-this}(\text{this}\ .\ \pi) = \pi$
$\text{remove-this}(attrOrRel\ .\ \pi) = attrOrRel\ .\ \pi$
$\text{trans-pref}(\pi\ .\ attrOrRel) = \{\pi\ .\ attrOrRel\} \cup \text{trans-pref}(\pi)$
$\text{trans-pref}(attrOrRel) = \{attrOrRel\}$

Figure 6 Dependency analysis step 1: path-based abstract interpretation

Dependency inversion $\boxed{(Ent.Attr \leftarrow \pi) \nearrow (Ent.AttrOrRel \rightarrow \pi)}$

$$\frac{E_2 = \text{entity-of}(attrOrRel)}{(E\ .\ attr \leftarrow \pi\ .\ attrOrRel) \nearrow (E_2\ .\ attrOrRel \rightarrow \text{inv-path}(\pi)\ .\ attr)}\ [\text{InvDep}]$$

$\text{inv-path}(\pi\ .\ attrOrRel) = attrOrRel^{-1}\ .\ \text{inv-path}(\pi)$
$\text{inv-path}(attrOrRel) = attrOrRel^{-1}$
$\text{inv-path}(\text{null}) = \text{null}$

Data flow $\boxed{Prog \nearrow\!\nearrow \{(Ent.AttrOrRel \rightarrow \pi)\}}$

$$\frac{\text{model } E^*\ R^* \searrow\!\searrow Dep}{\text{model } E^*\ R^* \nearrow\!\nearrow \{df \mid dep \nearrow df,\ dep \in Dep\}}\ [\text{Prog}]$$

Figure 7 Dependency analysis step 2: data flow

Data flow graph $\boxed{Prog \nearrow\!\nearrow\!\nearrow (\{AttrOrRel\}, \{(AttrOrRel, Attr)\})}$

$$\frac{\text{model } E^*\ R^* \nearrow\!\nearrow DFlow \quad E = \{(x,y) \mid (Ent.x \rightarrow \pi.y) \in DFlow\} \quad V = \{x \mid (x,y) \in E\} \cup \{y \mid (x,y) \in E\}}{\text{model } E^*\ R^* \nearrow\!\nearrow\!\nearrow (V, E)}\ [\text{Prog}]$$

Figure 8 Dependency analysis step 3: data flow graph.

Step 2: Data Flow. The data flow of an attribute or relation is the set of all the attributes that depend on it to compute their derived value. The data flow relation is the inverse of the dependency relation. We write $(Ent.AttrOrRel \rightarrow \pi)$ to denote the data flow relation from the source, $Ent.AttrOrRel$, to the target, the end of the path π.

The *dependency inversion* relation, $(Ent.Attr \leftarrow \pi) \nearrow (Ent.AttrOrRel \rightarrow \pi)$, in Figure 7 defines the inverse of a dependency. A dependency is inversed by swapping source and target, and inverting the path π to get the path from target to source. The function `inv-path(`π`)` inverts the names in on path, and inverts their order. Name inversion is selecting the name on the opposing side of a relation; all relations in IceDust are bidirectional, and have names on both sides. All names in π can be inverted because they are relations. (π is the prefix of a full path, and only the last name of a path can be an attribute.) If the dependencies of the attribute `pass` are inversed the resulting data flow is:

```
(Submission.grade      → pass)
(Assignment.minimum    → submissions.pass)
(Submission.assignment → pass)
(Submission.childPass  → pass)
```

Step 3: Data Flow Graph. The data flow graph relation $Prog \nearrow\!\!\!\nearrow (V, E)$ in Figure 8, defines a data flow graph in terms of the data flow relation. The nodes in the graph are the attributes and the relations in an IceDust program. The edges (x, y) in the graph are $(AttrOrRel, Attr)$ from the data flow relation $(Ent.AttrOrRel \rightarrow \pi.Attr)$. Using Tarjan's algorithm [35] we find strongly connected components and a topological ordering for the data flow. The strongly connected components correspond to recursive dependencies.

The data flow graph for our example application is shown in Figure 9. The attributes `grade`, `pass`, `childGrade`, and `childPass` mutually depend on each other, a cycle in the graph (group 6). (The data flow is not cyclic: data flows up the submission tree.) The `minimum` precedes group 6 in the topological ordering, as `pass` in group 6 depends on it but minimum itself depends on nothing. On the other hand, the `passPerc`, `averageGrade`, and `best` depend on the results in group 6. The derived `name` for submissions is disconnected from the grade calculation, as the name of the submission does not have anything to do with the grade calculation. Relations only flow to attributes, and not vice versa. In IceDust, relations cannot be derived. This limits the expressiveness of IceDust, but also avoids 'dynamic dependencies', dependencies that are discovered while computing derived values.

Topological ordering can be used to statically schedule the computation of derived values. This is used in stratified Datalog, where a topological sort of the dependencies between rules is used to determine the order of computation [3, 13]. We will elaborate on computation scheduling, and on similarities with existing approaches, in later sections.

4 Implementation Strategies

The declarative specification of derived values in IceDust allows deferring the decision about implementation strategy from implementation to compilation time, and allows switching strategies to realize different non-functional requirements without invasive code changes. In this section we present three implementation strategies: Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually. For each of these we have a compilation scheme that specifies what code to generate for IceDust's concepts.

On a high level the difference between the generated code for the different implementation strategies is the point in time at which derived values are calculated. Figure 10 shows the

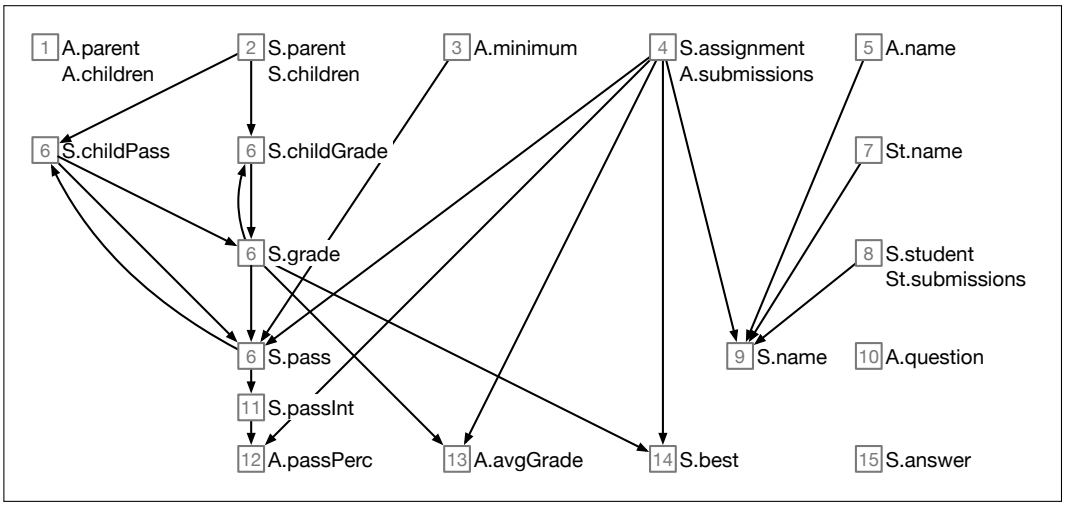

Figure 9 Step 3 example: strongly connected components and topological ordering in data flow.

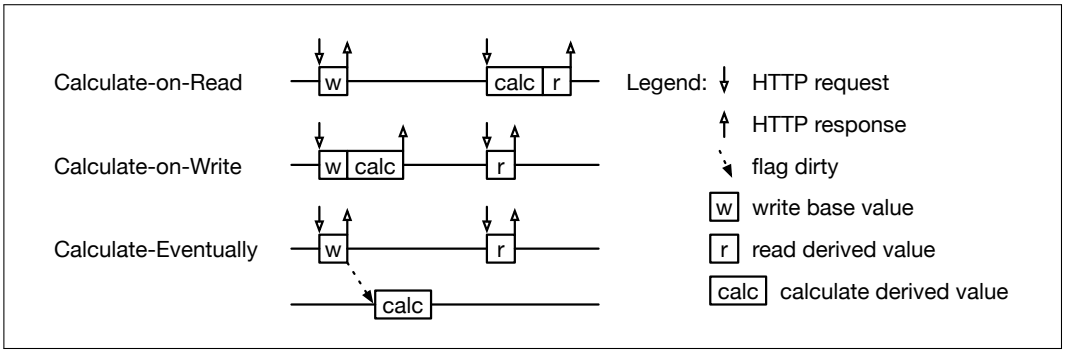

Figure 10 Thread activation diagrams for code generated by different implementation strategies.

differences by means of thread activation diagrams in response to incoming HTTP requests. The code generated by Calculate-on-Read calculates derived values when they are read. This means that writes to base values, on which derived values can depend, will be fast, but reads of derived values will be slow. The code generated by Calculate-on-Write calculates the derived values that depend on changed base values right away. Writes will be slow, but reads will be fast. The code generated by Calculate-Eventually schedules calculation of derived values on a separate thread. Writes and reads will be fast, but consistency is not guaranteed: possibly outdated derived values might be read.

Compiling to WebDSL IceDust is used to specify the data model and derived values for web applications. Our compiler compiles IceDust specifications to the WebDSL web programming language [36], which is a high-level target language for the implementation of data models. WebDSL persists its data in a relational database. This provides (1) data safety in case of a power outage, (2) enables large data sets, and (3) enables concurrent data access for concurrent HTTP requests. WebDSL's data modeling language is close to IceDust; it features entities and attributes (including Calculate-on-Read derived values):

```
entity Assignment {
  name     : String
  avgGrade : Float := avg([s.grade | s:Submission in subs where s.grade!=null])
}
```

Note the list comprehension syntax for applying a map to access the grade for each submission and filter on null values. WebDSL does not have bidirectional relations like IceDust, but it does have inverse properties:

```
entity Submission {  assignment  : Assignment (inverse = submissions)  }
entity Assignment {  submissions : Set<Submission>                      }
```

If a property is an inverse of another property, WebDSL keeps the values in the properties consistent. Our compiler targets these inverse properties for bidirectional relations, giving us the consistency of bidirectional relations for free.

With WebDSL already providing data persistence, large data sets, concurrency, Calculate-on-Read derived values, and inverse property consistency, our compilation schemes can focus on the essentials: default value behavior, multiplicities, bidirectional relations, and the Calculate-on-Write and Calculate-Eventually implementations for derived values.

The rest of this section describes the three implementation strategies in detail, using MorphJ[20]-style code generation templates for the compilation schemes. The templates use WebDSL (`black` with `purple keywords`) as target language and template-level control statements (*`blue italic`*) that iterate over entities, attributes, relations, and data flow edges (*`orange italic`*). We explain WebDSL code along the way, using callouts (for example: [1]) to refer to specific parts of generated WebDSL code.

Calculate on Read. Figure 11 defines the Calculate-on-Read compilation scheme. To translate IceDust with Calculate-on-Read to WebDSL we need to translate three IceDust features: (1) multiplicities, (2) default value attributes, and (3) bidirectional relations.

Multiplicities `?` and `1` are translated to WebDSL primitives, while multiplicities `*` or `+` are translated to lists. The getter for a normal attribute [2] (see Figure 11) is static for null-safety, it might be called on a null value, for example: `Assignment.get_passPerc(null)`. The getter is lifted to deal with a list of entities for which the attribute is referenced [10]. Attributes can only have multiplicity `?` or `1`, so there is no generation for multiplicity `*` or `+`. (List typed attributes would create overhead in WebDSL's mapping to the underlying database.)

Default value attributes are translated to two attributes [6,7] and one getter [9] in WebDSL. The first attribute [6] corresponds to the value possibly set by the user. The second attribute [7] corresponds to the default value expression. The getter [9] will return the user provided value, if any, and otherwise the default value. When only [6] is used to write values, and only [9] is used to read values the default value attribute will have IceDust's semantics. WebDSL features no `private` attributes and methods, so this behavior cannot be encapsulated.

Bidirectional relations are translated to properties and inverse properties (which are kept consistent by WebDSL). The right-hand side of the relation is translated to a normal WebDSL property [12,14,16], and the left-hand side is translated to a property with an inverse [11,13,15]. Unordered to-many relations are translated to sets, while the ordered relations are translated to lists. It would suffice to translate them all to lists, but WebDSL's relational database mapping has more overhead for lists than for sets. Relation navigation is overloaded on multiplicity: navigate from single entity via a to-one relation [17], or via a to-many relation [19], and navigate from multiple entities via a to-one relation [18], or via a to-many relation [20].

```
for E in Entities
  entity E {
    for a : T m in E.attributes
      a : T (default=null)[1]

      static function get_a(e : E) : T {  return if(e == null) null else e.a;  }[2]

    for a : T m = e1 in E.attributes
      a : T := calculate_a()[3]

      function calculate_a() : T {  return e1;  }[4]

      static function get_a(e : E) : T {  return if(e == null) null else e.a;  }[5]

    for a : T m = e1 (default) in E.attributes
      a : T (default=null)[6]
      a_default : T := calculate_a()[7]

      function calculate_a() : T {  return e1;  }[8]

      static function get_a(e : E) : T {
        return if(e == null) null else if(e.a == null) e.a_default else e.a;
      }[9]

    for a : T m {, = e1, = e1 (default)} in E.attributes
      static function get_a(entities : [E]) : [T] {
        return [E.get_a(e) | e : E in entities where E.get_a(e) != null];
      }[10]

    for relation E.l {1,?} <-> m2 E2.r in Relations
      l : E2 (inverse=r)[11]
    for relation E2.r m2 <-> {1,?} E.l in Relations
      l : E2[12]
    for relation E.l {*,+} (unordered) <-> m2 E2.r in Relations
      l : {E2} (inverse=r)[13]
    for relation E2.r m2 <-> {*,+} (unordered) E.l in Relations
      l : {E2}[14]
    for relation E.l {*,+} (ordered) <-> m2 E2.r in Relations
      l : [E2] (inverse=r)[15]
    for relation E2.r m2 <-> {*,+} (ordered) E2.r in Relations
      l : [E2][16]

    for relation E.l {1,?} <-> m2 E2.r
    and relation E2.r m2 <-> {1,?} E.l in Relations
      static function get_l(e : E) : E2 { return if(e == null) null else e.l; }[17]

      static function get_l(entities : [E]) : [E2]{
        return [E.get_l(e) | e : E in entities where E.get_l(e) != null];
      }[18]

    for relation E.l {+,*} <-> m2 E2.r
    and relation E2.r m2 <-> {+,*} E.l in Relations
      static function get_l(e : E) : [E2]{
        return if(e == null) null else [e2 | e2 : E2 in e.l];
      }[19]

      static function get_l(entities : [E]) : [E2]{
        return [e2 | e : E in entities, e2 : E2 in e.l];
      }[20]
  }
```

Figure 11 Compilation scheme for Calculate-on-Read implementation strategy.

Calculate on Read Properties. The compiled Calculate-on-Read programs have the following properties: (1) derived value reads are consistent, (2) transactions might fail, and (3) cyclic derived values cause a stack overflow exception at runtime.

Derived value consistency is based on database transactions. HTTP requests see all changes to base data from previous requests, and no changes to base data from concurrent requests. They compute the derived values, so these are consistent. The database performs optimistic locking, consequently transactions with concurrent edits to the same values are rejected. A cycle in the static dependency graph, such as group 6 in Figure 9, can admit a cyclic attribute value definition (for example a submission being a child of itself, and its grade being the average of its child grades). Such a cyclic derived value cannot be computed. The generated code will keep recursing into the getters until stack space is exhausted.

Calculate on Write. Figure 12 defines the Calculate-on-Write compilation scheme. The Calculate-on-Write compilation scheme builds on the Calculate-on-Read compilation scheme, only mentioning the new or changed WebDSL code. The general idea for Calculate-on-Write is caching all derived values, and incrementally maintaining the cached values on writes (like materialized views [14]). Updating a derived value can lead to having to update other derived values. This behavior is realized by dirty flagging (and updating) all dependent attributes on updating an attribute or relation (like push-based reactive programming [27]). To avoid unnecessary recomputation, updates are scheduled using the topological sort of the data flow graph (like stratified Datalog [3, 13]). So, to translate IceDust with Calculate-on-Write to WebDSL, we need to generate caches, dirty flagging, and recalculation.

Derived value caches store the derived values [22,25]. The properties containing the cached derived values are managed by code keeping track of dirty values [29,30,36], and code for updating dirty values [27,28].

Dirty flagging of derived values happens when underlying values are updated. WebDSL provides `extend function` hooks to intercept calls to setters. When a setter is called, all dependent values are dirty flagged by traversing the data flow `path`s [31–34]. Attributes and relations with multiplicity `?` and `1` only dirty flag when the value changes [31], while relations with multiplicity `*` and `+` dirty flag on additions and removals [32,33]. As relations have two names, dirty flagging is done for both names. Moreover, updating a relation can also implicitly remove another relation. For example, moving a submission to a different assignment

```
bobsSubmissionToMath.assignment := logicAssignment;
```

will trigger:

```
bobsSubmissionToMath.set_assignment(logicAssignment);
mathAssignment.remove_from_submissions(bobsSubmissionToMath);
logicAssignment.add_to_submissions(bobsSubmissionToMath);
```

Recalculation of derived values [35] is performed after user code is run, and before the flush to database. The computation is scheduled statically by means of the topological sort of the connected components in the data flow graph. Within a connected component, a `while` continues computing derived values until none of the derived values is dirty anymore.

Calculate on Write Properties. This compilation scheme yields programs with the following properties: (1) the derived value reads are consistent, (2) transactions might fail, (3) cyclic derived values can cause non-termination, (4) scheduling is optimal for acyclic dependency graphs, and (5) scheduling is naive for connected components inside the dependency graph.

Consistency of derived values is based on consistency of derived values within a single HTTP request, and database concurrent transaction semantics. For any changed attribute

```
//All code from Calculate-on-Read, except generated fields for attributes.
for E in Entities
  entity E {
    for a : T m in E.attributes
      a : T (default=null)[21]

    for a : T m = e1 in E.attributes
      a : T (default=calculate_a())[22]

      function update_a() {  a := calculate_a();  }[23]

    for a : T m = e1 (default) in E.attributes
      a : T (default=null)[24]
      a_default : T (default=calculate_a())[25]

      function update_a() {  a_default := calculate_a();  }[26]

    for a : T m {= e1, = e1 (default)} in E.attributes
      static function a_update_all() {
        for(e in E.get_and_empty_a_dirty()){  e.update_a();  }
      }[27]

      static function get_and_empty_a_dirty() : {E} {
        var values := E_a_dirty; E_a_dirty := Set<E>(); return values;
      }[28]

      static function a_has_dirty() : Bool {  return E_a_dirty.length != 0;  }[29]

      static function a_flag_dirty(entities : {E}){ E_a_dirty.addAll(entities);}[30]

    for E.a -> path.a2 in DataFlow where a.multiplicity in {?,1} and E2=a2.entity
      extend function set_a(newV : T){ if(a != newV){ E2.a2_flag_dirty(path); }}[31]

    for E.l -> path.a2 in DataFlow where l.multiplicity in {*,+} and E2=a2.entity
      extend function add_to_l      (n:T){ if(l != n){ E2.a2_flag_dirty(path); }}[32]

      extend function remove_from_l(n:T){ if(l != n){ E2.a2_flag_dirty(path); }}[33]

    for E.a -> path.a2 in DataFlow where a2 : T m = e1 (default) and E2=a2.entity
      extend function set_a_default(newValue : T) {
        if(a == null && a_default != newValue){  E2.a2_flag_dirty(path);  }
      }[34]
  }

// update_derivations gets called before flush to database
function update_derivations(){
  var not_empty : Bool;
  for ConnectedComponent cc in DataFlowGraph topologically sorted
    not_empty := true;
    while(not_empty){
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        E.a_update_all();
      not_empty := false;
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        not_empty := not_empty || E.a_has_dirty();
    }
}[35]

for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
  request var E_a_dirty : {E} := Set<E>()[36]
```

Figure 12 Compilation scheme for Calculate-on-Write implementation strategy.

or relation, all the values that depend on it are dirty flagged and recomputed. By induction, all values that depend transitively on a changed value get dirty flagged and recomputed. Computation only stops if all dirty flags are processed. As such, for a specific HTTP request, all derived values in memory are up to date when computation terminates. Flushing to the database only succeeds if previously read data remains unchanged, guaranteeing consistency. Failing transactions occur more often in Calculate-on-Write than in Calculate-on-Read, as both the updates to base values, and the updates to derived value caches can cause conflicts.

Cyclic derived values, such as the average submission grade depending on itself, can cause non-termination. If an updated value dirty flags itself (transitively), and its new value is different, the computation loops. A diverging value causes non-termination, while a converging value is a fix point calculation. Incremental Datalog implementations guarantee termination by disallowing recursive aggregation and negation: stratified negation [3] and stratified aggregation [26]. We allow recursive aggregation, but do not guarantee termination.

Derived values should only be recomputed after all values they depend upon are already updated. With acyclic data flow graphs, topological scheduling completely removes unnecessary recomputation. With cyclic data flow graphs, topological scheduling only partially removes unnecessary recomputation: the connected components are statically scheduled, but the derived values inside a connected component are updated without scheduling.

Calculate Eventually. Figure 13 defines the Calculate-Eventually compilation scheme. The Calculate-Eventually compilation scheme builds on the Calculate-on-Write compilation scheme, only stating additions and changes. The idea is to take the dirty flags from Calculate-on-Write, but pass these on to a separate, dedicated thread, allowing the HTTP request handlers to finish early. The writes to base values will still be synchronous, but the updates to derived values will be asynchronous. So, to translate to WebDSL we need to generate code that (1) dirty flags cross-thread, and (2) updates derived values in a separate thread.

Cross-thread dirty flagging communicates dirty flags from request handlers to the updater thread. WebDSL abstracts over concurrent handling of requests by running request handlers completely separated from each other. Communication between the threads handling HTTP requests, normally, is through the database. However, the database cannot notify the updater thread, so in memory communication is required. To communicate in memory between threads in WebDSL we need `native` Java code. For each derived value attribute we generate a `ConcurrentLinkedQueue` [45], and make this queue available in WebDSL by means of a static function in a `native class` [44]. As an HTTP request is handled, derived values get dirty flagged locally (as in Calculate-on-Write). After the changes are flushed to database, the local dirty flags are communicated cross-thread. Because an entity can be mapped from the relational database to an object in memory multiple times (once per request handler), the cross-thread dirty flagging needs to communicate an entity's unique identifier (UUID) [39].

The *derived value recalculation thread* is started with WebDSL's recurring tasks mechanism [42]. Every millisecond the thread is started, if not still running. The thread performs the same calculations as Calculate-on-Write, but uses the cross-thread dirty flags [43]. It loads entities with dirty flagged derived values into memory [37], then updates derived values, and finally propagates its own local dirty flags to the cross-thread dirty flags [39].

Calculate Eventually Properties. The Calculate-Eventually programs have the following properties: (1) derived values will eventually be up to date, (2) derived value reads are not glitch-free, (3) derived value calculation can starve under load, (4) after load subsides only relevant updates are calculated, and (5) cyclic values can cause non-termination.

Eventual calculation is guaranteed by the invariant that outdated derived values are

```
//All code for Calculate-on-Write, except for update_derivations.
for E in Entities
  entity E {
    for a : T m {= e1, = e1 (default)} in E.attributes
      static function get_and_empty_a_dirty_async() : {E} {
        var queue := DirtyQueues.get_E_a_queue();  var values : {E};
        while(!queue.isEmpty()){
          values.add(loadEntity(E, UUIDFromString(queue.poll() as String)) as E);
        }
        return values;
      }[37]

      static function a_has_dirty_async() : Bool {
        return !DirtyCollections.get_E_a_queue().isEmpty();
      }[38]

      static function a_flag_dirty_async() {
        var dirty := E.get_and_empty_a_dirty();
        DirtyCollections.get_E_a_queue().addAll([v.id.toString()|v : E in dirty]);
      }[39]

      static function a_update_all_async() {
        for(e in E.get_and_empty_a_dirty_async()){  e.update_a();   }
      }[40]
  }

//flag_dirty_async is called on every request after write to database
function flag_dirty_async() {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    E.a_flag_dirty_async();
}[41]

invoke update_derivations() every 1 milliseconds[42]
function update_derivations(){
  var not_empty : Bool;
  for ConnectedComponent cc in DataFlowGraph topologically sorted
    not_empty := true;
    while(not_empty){
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        E.a_update_all();
      flagDirtyAsync();
      not_empty := false;
      for a : T m {= e1, = e1 (default)} in cc where E = a.entity
        not_empty := not_empty || E.a_has_dirty();
    }
}[43]

native class derivations.DirtyQueues as DirtyQueues {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    static get_E_a_queue() : Queue[44]
}
```

```
public class DirtyQueues {
  for a : T m {= e1, = e1 (default)} in Attributes and E = a.entity
    private static Queue<String> E_a_queue =new ConcurrentLinkedQueue<String>();[45]
    public static Queue<String> get_E_a_queue(){  return E_a_queue;  }[46]
}
```

Figure 13 Compilation scheme for Calculate-Eventually implementation strategy.

always accompanied by a dirty flag. Dirty flags are only sent by request handling threads after their changes are flushed to the database, ensuring the updater thread never processes dirty flags without seeing the changes. During updates (the same) derived values might be dirty flagged again. To ensure new dirty flags are processed, the updater thread copies and empties the dirty flag queues before processing flags. New flags will be processed subsequently.

Glitch-freedom is not provided inside connected components, as there is no topological ordering on the instance level. Also, derived value calculation starvation happens when server load is high. However, successive changes to the same base values will not create extra dirty flags. So when the system has spare resources, it will just compute the derived values based on the latest base values, and ignore all the intermediate base values. And finally, like Calculate-on-Write, cyclic derived values can cause non-termination.

5 Evaluation

The declarative specification of derived values in IceDust allows switching implementation strategies to realize different non-functional requirements without invasive code changes. In this section we benchmark different generated implementations, to evaluate whether they are indeed able to satisfy different non-functional requirements. The benchmarks differ in (1) the read/write ratio, (2) the number of base values derived values depend upon, and (3) the number of fully unrelated derived values. The measured non-functional properties are (1) throughput of derived value reads and base value writes per second, (2) the number of failing writes per second, and (3) the response time for reading derived values and writing base values. In this section we will discuss the benchmark setup and results.

Benchmark Setup. In the case study (Section 6) we encounter derived values that depend on up to 60000 values transitively. The essence of the calculation is a tree-like structure with aggregations on every level. For the benchmark evaluation of the different implementation strategies we use a simplified model, a simple tree with an average on each level:

```
entity Node {  avgValue : Float? = avg(children.avgValue) (default)  }
relation Node.parent ? <-> * Node.children
```

The tree branches out with a factor of 10, up to 6 deep (size 1, 11, . . . , 111111).

The benchmarks consist of read and write requests. Read requests retrieve the average at the top node of the tree. Write requests update a value at a random leaf node of the tree. Benchmarks are warmed up for 10 seconds, and then measured for 15 minutes. The Siege[1] tool is used to execute the benchmark requests. It is configured to use 10 concurrent threads, which initial benchmarks indicated to be a reasonable concurrency level. If the concurrency level is too low, the computer is not using maximum resources; if it is too high, too many requests will queue up increasing response times but not improving throughput. The benchmarks were performed on an early 2013 Macbook Pro laptop with Intel Core i7 2,7Ghz, 4 cores (8 threads), and 16 GB memory. The Java servlet web application generated by WebDSL was deployed on OS X 10.11, Java 1.8.0_60, MySQL 5.6.27, and Tomcat 7.0.40.

Benchmark Results. The first two benchmarks determine the behavior in extreme workloads with only read or only write requests. Figure 14 shows that the performance for a workload of 100% decreases as the tree gets deeper. Beyond depth 4 the response takes longer than

[1] https://www.joedog.org/siege-home/

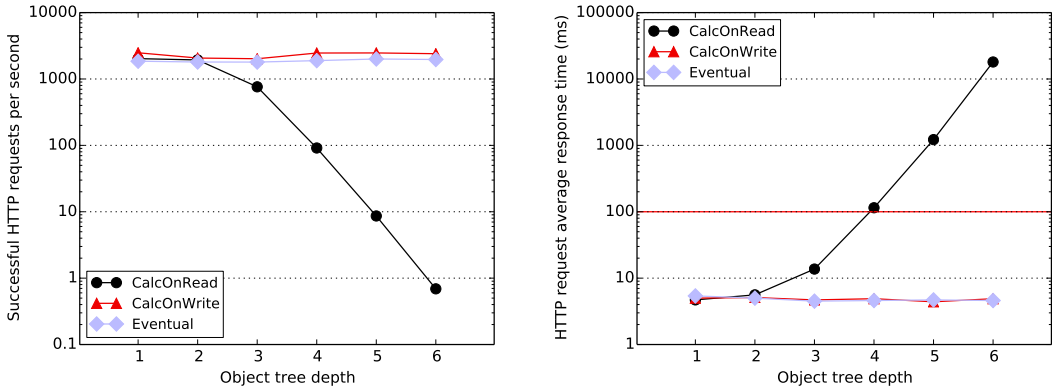

Figure 14 Read-only workload benchmark throughput (left) and latency (right).

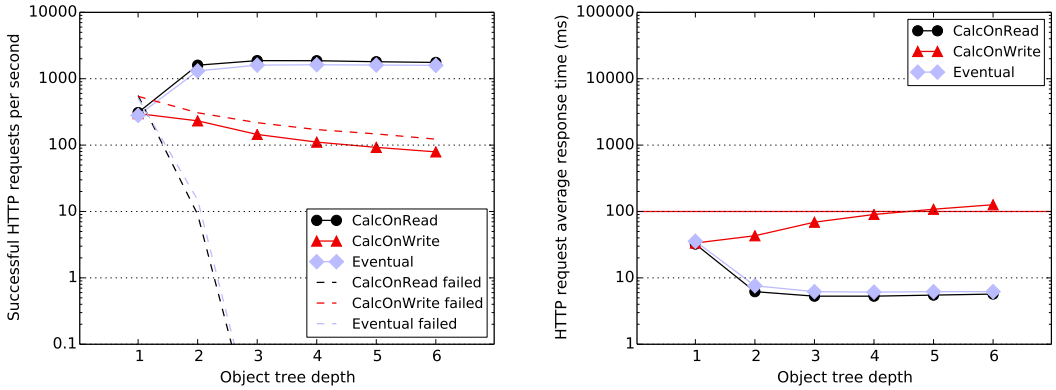

Figure 15 Write-only workload benchmark throughput (left) and latency (right).

0.1 second, and is noticeable for users. Calculate-on-Read response times increase linearly with the number of read objects, indicating that this is the limiting factor. The other implementation strategies stay at a steady high throughput with low latency, because they only retrieve a single node with a cached value on each request. The maximum throughput is around 1900 transactions per second, indicating the general overhead of the system.

The benchmark in Figure 15 shows the performance for write-only workloads. In addition to the throughput of successful requests, the failed requests are indicated by the dashed lines. When a database transaction fails due to conflicting writes, WebDSL retries it up to 3 times before failing the entire request. This improves usability for typical scenarios where a single transaction conflict may occur occasionally. Multiple subsequent failed transactions only occur in extreme situations where many concurrent requests conflict. For example, in this benchmark at tree depths 1 (1 object) and 2 (11 objects), all implementation strategies have repeated transaction failures resulting in failed requests. In general, the maximum throughput the system supports for concurrent edits on a single object is close to 300 edits per second (tree depth 1, all implementation strategies). Calculate-on-Write has many request failures (around 60%) at all tree depths, which makes the implementation unusable in practice for this use case. Calculate-on-Read and Calculate-Eventually have overall high throughput and low latency, except for the low tree depths where transaction failures occur.

Figure 16 shows the trade-off between Calculate-on-Read and Calculate-on-Write in mixed read/write ratio workloads. Calculate-on-Write suffers from many transaction failures, except for 100% reads. So, it is unusable if all base values aggregate into a single value, even

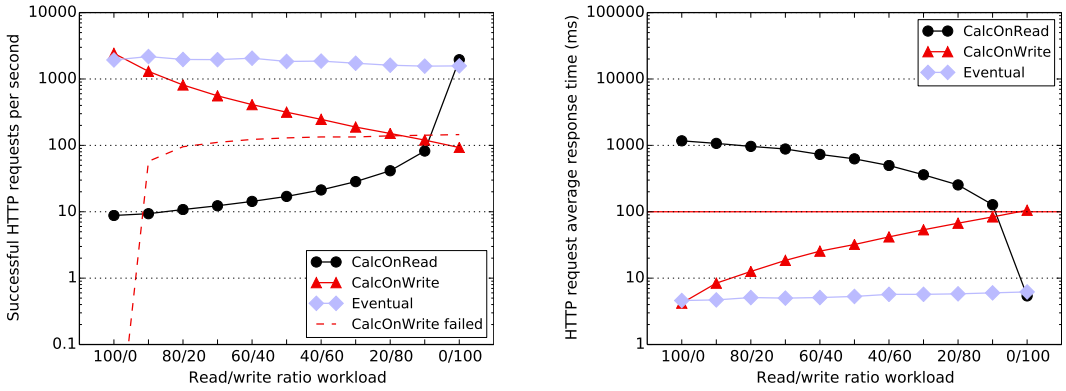

Figure 16 Varying workload benchmark throughput (left) and latency (right) with tree depth 5.

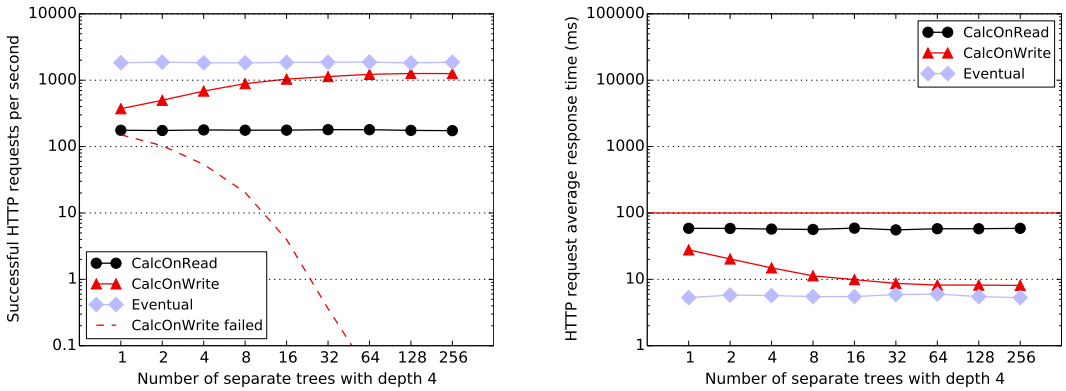

Figure 17 Separate trees benchmark throughput (left) and latency (right) with 50-50 workload.

with a small number of concurrent writes. Calculate-on-Read improves when the workload shifts from reads to writes. However, the average response time for anything except 100% writes is unacceptebly high. Calculate-Eventually has stable high throughput and low latency for all the different workload scenarios. If all base values in a system aggregate transitively into a single derived value, the only viable strategy is Calculate-Eventually.

In the final benchmark, shown in Figure 17, we investigate whether not aggregating all base values into the same derived value makes the Calculate-on-Write strategy viable. We compare the implementation strategies when there are up to 256 separate trees of depth 4. Calculate-on-Write performs better indeed in scenarios with more disconnected derived values. With 16 trees, the number of failed transactions drops below 0.5%. If consistency is desired, and derived values depend on roughly 1000 base values, the trade-off throughput-wise between Calculate-on-Read and Calculate-on-Write is at 2 separate trees. However, Calculate-on-Write still has many failing requests. Only at around 16 trees the number of failing transactions falls below 0.5%, and Calculate-on-Write becomes a viable option. When eventual calculation is acceptable, it is always the most performant solution.

Discussion. We could perform many more benchmarks (for example with other data structures than trees, or with workloads with more structure than a read/write ratio). However, the presented benchmarks show that each implementation strategy is useful in specific cases.

The form of non-functional requirements determines the form of verification required [10]. The verification for quantitative requirements is measurements, and for operational

requirements is review, test or formal verification [10]. Consistency and eventual calculation are operational requirements. We tested whether our implementations satisfy consistency and eventual calculation. In future work this can be improved with formal verification.

6 Case Study

We applied IceDust to the grading policies in a learning management system in which students can submit assignments that get graded semi-automatically. The system contains complex derived value calculations, contains a lot of data (hundreds of thousands of entities, with millions of derived values), and is subject to intense workloads on a small subset of the data. The complex derived value calculations were specified in IceDust's declarative derived value attributes, and the Calculate-Eventually strategy was used to generate an implementation. In this section we (1) reflect on the expressiveness of IceDust, based on the experiences from the case study, and (2) highlight parts of the resulting declarative specification that are quite different from the original imperative implementation.

First, let us introduce the learning management system in more detail. The system is a much more complicated version of the one introduced in Section 3. It features semi-automatic grading, programming assignments with test cases, and automatically graded multiple choice questions. Assignments are structured in a tree, and students get a weighted average for each node in the tree up to the top, which is their final grade for the course. The grading logic also includes deadlines, deadline extensions, late penalties, minimum grades, and alternative assignments. The courses, and their assignments, have statistics such as the percentage of students with a passing grade. For the largest course in the system, the statistics depend transitively on Â± 60000 individual submissions (Â± 250 students, with Â± 15 assignments per week, running for 12 weeks, and exams with multiple questions in the end).

Explicit Not Yet Calculated Values. Expressing the grading logic in IceDust forced us to look at previously implicit things. Students which did not attempt an assignment get a 0.0 on a scale of 1.0 to 10.0. The other students would get a 1.0, as grading would be triggered:

```
// only call calculateGrade if submission is attempted
function calculateGrade() {  grade := max(1.0, calculatedGrade);  }
```

The grading logic states that grades cannot be lower than 1.0, but if grading is not triggered the float default value is used. As the compiled code from IceDust detects everything that should be calculated, these grades would be changed from 0.0 to 1.0. To model assignments that are not attempted by students, `attempted` should be explicitly mentioned:

```
grade : Float = if(not attempted) 0.0 else max(1.0, calculatedGrade)
```

Explicit Stateful Calculations. The imperative code, also implicitly, kept old grades when grades where published and a newly calculated grade was lower:

```
if(assignment.statsPublic()) {  newgrade := max(oldgrade, newgrade);  }
```

In the new specification this requires an explicit self-reference:

```
grade : Float = if(public) max(grade, calculatedGrade) else calculatedGrade
```

Note that the IceDust specification only works for push-based implementations: Calculate-on-Write or Calculate-Eventually. Calculate-on-Read would throw a stack overflow exception. The sentence *'only update when grade is higher'* implies push-based calculation: the previous calculated grade needs to be cached for when a new grade becomes available. It is arguable

whether someone should express logic like this, as once grades are visible, it is not traceable anymore how a grade was calculated. This is on the border of what is expressible in IceDust.

Code Factorization Differences. In IceDust the value of an attribute is defined in a single place: the derived value expression. In imperative code, assignments to attributes can happen in multiple places, which means assignments can be distributed over `if`s:

```
if(assignment.passOne){
  passSub := disj([s.pass() | s:Submission in submissions]);
  grade   := max([s.grade() | s:Submission in submissions]);
} else{
  passSub := conj([s.pass() | s:Submission in submissions]);
  grade   := avg([s.grade() | s:Submission in submissions]);
}
```

In IceDust `if`s need to be distributed over derived value attributes:

```
grade   : Float?  = if(assignment.passOne) max(children.grade)
                    else                   avg(children.grade)

passSub : Boolean = if(assignment.passOne) disj(children.pass) // pass one
                    else                   conj(children.pass) // pass all
```

Whether the old or new specification is preferable is arguable. If the cases would be more complex than a single if it would lead to repeated code in IceDust.

Application Analysis. Finally, we made a more analytic observation: even though the data-flow graph of the specification in IceDust contains more than 100 nodes, it contains just a single connected component. Grades, weightedGrades (weighted averaging is used), pass, and child-pass are mutually recursive (like Figure 9). All other dependencies are acyclic. This system has derived value-wise just a single complex part: the grade calculation. Intuitively we already knew this, but now we can quantify this with properties of the data-flow graph.

7 Related Work

The related work is organized along language design and the three implementation techniques (Calculate-on-Read, Calculate-on-Write, and Calculate-Eventually).

Languages with Relations. There are multiple languages that feature relations as a language construct. We will cover closely related languages and highlight the differences.

Rumer [4] features first-class citizen relations with queries for navigation. IceDust relations are not first-class citizen, and navigation is through member access instead of queries. In Rumer multiplicities can be specified in constraints which are enforced at runtime, while in IceDust these are part of the type system. Rumer does not support derived value attributes, but queries can be used to specify Calculate-on-Read derived values. Finally, Rumer is an imperative in-memory language, while IceDust is declarative and persists its data.

RelJ [5] also features first-class citizen relations. In RelJ participants of relations do not have names, so navigation is positional (using `from` and `to`). RelJ features only multiplicity upper bounds, no lower bounds. These multiplicities are enforced at runtime: either by throwing exceptions, or by implicitly removing previous relations. RelJ does not feature derived value attributes, and RelJ is an imperative in-memory language like Rumer.

Relations [18] features multiplicities as part of the type system and derived value attributes like IceDust. Its derived values are, however, only Calculate-on-Read. Relations is declarative,

like IceDust, but its data is only in memory, not persistent. Relations in this language are first-class citizen like Rumer and RelJ, but feature navigation through member access. IceDust relations are not first-class citizen, but feature the same member access navigation.

Alloy [21] is a language for bounded model checking which features language constructs similar to IceDust: bidirectional relations, multiplicities, and derived values. Alloy is more expressive than IceDust: it features n-ary relations, and its derived values specify derived relations (as opposed to derived attribute values). However, Alloy's bounded model checker only works on small data sets, and primitive values (only integers in Alloy) should be avoided as they blow up the state space. IceDust, on the other hand, supports derived values over arbitrary primitive values (int, string, float, datetime, and boolean), and admits efficient implementation strategies applied to large data sets. To compute derived values in large data sets from an Alloy specification, Alloy would need an operational semantics not based on bounded model checking or SAT solving. An approach for an operational semantics for Alloy was proposed in [9], but this approach is not complete. As Alloy has much greater expressive power (first-order logic), we also expect such an Alloy operational semantics to not be efficient. Finally, another difference is that in IceDust the multiplicities are checked in the type system, while in Alloy these are only checked during bounded model checking.

Calculate on Read. We do not cover Calculate-on-Read extensively, as it is the default implementation for many formalisms. We cover only the object-oriented approaches.

Object-oriented languages lend themselves for various Calculate-on-Read optimization techniques. Wiedermann and Cook take imperative code with for loops and if statements and convert those to SQL queries [37]. Their approach is similar to our work in that it operates on persistent objects by means of an object-relational mapper. Also their approach for analyzing dependencies is similar: path-based abstract interpretation. They optimize imperative code that can be expressed as queries. Our approach, on the other hand, treats code that cannot be expressed as queries, recursive aggregation. The Java Query Language (JQL) adds queries to Java [38]. The rationale for queries is that these are more succinct to write, and more efficient than nested for loops. JQL has been incrementalized, we will cover this in the next subsection. This paper adds over these approaches the possibility to easily switch to an incremental or eventual calculation implementation strategy.

Incremental Computation (Calculate on Write). Incremental computation is present in many fields in computer science. We relate our Calculate-on-Write implementation scheme to existing incremental approaches.

Materialized views in relational databases can be incrementally maintained [14]. Recursion and stratified aggregation can be supported [15]. Stratified aggregation does not admit recursive aggregation. (See next paragraph for relaxations of stratified aggregation in logic databases.) Switching between implementation strategies in relational database also do not require invasive code changes: the definitions for materialized and non-materialized views are identical. Relational databases do, however, not support eventually-calculated views.

Logic Databases or Deductive Databases are a more expressive than relational databases. Logic Databases support stratified aggregation like relational databases [26]. Since stratified aggregation does not support recursive aggregation, more relaxed notions of aggregation have been introduced, such as Monotone Aggregation [29]. Monotone Aggregation has also been incrementalized [28]. A recent survey [13] states that at present, the Datalog community seems not to have converged on any of the proposed semantics for aggregation through recursion. This means that in practice recursive aggregation is often not supported. For example LogiQL [12], the language of LogicBlox, does not support recursive aggregation.

Functional reactive programming (FRP) [8], with for example REScala [30], Scala.React [23], or i3QL [25], provides incremental computation. Calculate-on-Read style code wrapped with FRP libraries behaves as Calculate-on-Write. FRP abstractions provide single-threaded, in-memory derived values. In contrast, IceDust provides concurrent, persistent derived values.

Spreadsheets provide incremental computation. The data structure in a spreadsheet is a 2d grid. IceDust's data structure is an object graph. Moreover our object graph is typed, while spreadsheets are free form. Spreadsheets do mostly have an implicit structure [19]. IceDust with Calculate-on-Write can be seen as a structured spreadsheet without a 2d grid.

Object-oriented programs can also be incrementalized. Incremental Updates for Materialized OQL views [11] proposes to generalize incremental view maintenance from relational databases to support the Object Query Language (OQL) as view definition language. MOVIE [2] develops this work further. They also provide an overview of relational incremental view maintenance implementations, with either a relational or an object-oriented surface syntax. These approaches, even though some have an object-oriented surface syntax, are part of the relational paradigm (with the limitations previously mentioned for materialized views).

The Java Query Language is incrementalized [39]. Their benchmarks show, like ours, that for different read-write ratio workloads the incremental or calculate-on-read solution offers better performance. Demand-Driven Incremental Object Queries [22] improves over JQL by using auxiliary indices for incrementality. Similar to [37] they transform imperative code to a relational calculus. But instead of performing relational queries like [37] they use the relational model to generate code that incrementally maintains the caches. Our approach uses path-based abstract path interpretation instead of a relational calculus to generate maintenance code. Both of the above approaches slightly differ in use cases from our approach: they target set membership of objects e.g. whether an object belongs to a set specified by a query, while our approach targets derived value attributes.

Graph queries can be incrementally evaluated in IncQuery [34]. IncQuery's data structure is a graph, like ours, but its goal is to pattern match. Our approach does not support pattern matching on graph structures, rather it computes derived attribute values.

Attribute grammars feature a declarative style of specifying derived values. Attribute grammars can also be incrementally computed [7]. As attribute grammars only support trees, one could look at reference attribute grammars to support full blown graphs [31]. Reference attribute grammars do support graph structures, but there is a clear distinction between the tree, and the derived graph edges. In our approach the graph is the basis. Fitting our data models onto attribute grammars would require extracting a spanning tree, and deriving the other edges. In this process we would lose the correspondence to the data flow graph, and derived edges would become dynamic dependencies, which would complicate scheduling.

Self-adjusting computation [1] does not cover a single programming paradigm as it features multiple languages (including SLf, for functional programming, and SLi, for imperative programming). Self-adjusting computation automatically transforms a Calculate-on-Read style program to a Calculate-on-Write style program. Our approach does not take Calculate-on-Read as basis, but instead provides a declarative language to express derived values.

Eventual Calculation. The code generated by the Calculate-Eventually implementation scheme makes derived values of attributes eventually consistent with base values. We cover existing work on eventual (or asynchronous) computation of derived values in this subsection.

Event and Actor programming, with for example Akka [16] or RX [24], provide an asynchronous update mechanism for calculating derived values. Updates to derived values are asynchronous, meaning that there is no consistent view of base values and derived values

at the same time. As such, these do not provide consistency, like the code produced by our Calculate-Eventually implementation strategy.

Eventual consistency for distributed data also features eventual calculation, but is unrelated. As a recent survey on Eventual Consistency states: "shared data is updated at different replicas, updates are transmitted asynchronously, and conflicts are resolved consistently" [6]. Our approach does not have different replicas of data, there is a single database. Our approach does not have asynchronous updates, the update is synchronous as a HTTP response is only sent after the transaction in the database is completed. And finally, our approach does not have conflicts during the calculation of derived values, as the base values define unambiguously what the derived values of attributes should be.

8 Conclusion

Data modeling with declarative derived value attributes in IceDust allows deferring the decision about implementation strategy from implementation to compilation time, and allows switching strategies without invasive code changes. We have demonstrated that these different strategies provide different non-functional properties, so that a specific strategy can be chosen to realize certain non-functional requirements. Finally, a case study indicated our approach is useful for expressing derived values of systems used in practice.

In future work, we would like to explore more implementation strategies, such as transitive dirty flagging on writes with recalculation on reads, or eventually calculated with flags indicating whether the values are up to date or not. We also would like to explore more flexibility in implementation strategies by allowing composition of different strategies, and live switching between strategies. A type system should restrict compositions to only sound ones: consistent values cannot depend on eventually calculated values, and calculate on write values cannot depend on calculate on read values. Finally, we would like to guarantee termination by specifying non-circular relations and runtime non-circularity checking.

References

1 Umut A. Acar. Self-adjusting computation: (an overview). In *PEPM*, pages 1–6, 2009. doi:10.1145/1480945.1480946.

2 M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. Movie: An incremental maintenance system for materialized object views. *DKE*, 47(2):131–166, 2003. doi:10.1016/S0169-023X(03)00048-X.

3 Krzysztof R Apt, Howard A Blair, and Adrian Walker. *Towards a theory of declarative knowledge*. IBM Thomas J. Watson Research Division, 1986.

4 Stephanie Balzer. *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH, Zürich, 2011.

5 Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286, 2005. doi:10.1007/11531142_12.

6 Sebastian Burckhardt. Principles of eventual consistency. *FTPL*, 1(1-2):1–150, 2014. doi:10.1561/2500000011.

7 Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, pages 105–116, 1981. doi:10.1145/567532.567544.

8 Conal M. Elliott. Push-pull functional reactive programming. In *haskell*, pages 25–36, 2009. doi:10.1145/1596638.1596643.

9 Theophilos Giannakopoulos, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Towards an operational semantics for alloy. In *FM*, pages 483–498, 2009. doi:10.1007/978-3-642-05089-3_31.

10 Martin Glinz. Rethinking the notion of non-functional requirements. In *WCSQ*, pages 55–64, 2005.

11 Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental updates for materialized oql views. In *DOOD*, pages 52–66, 1997. doi:10.1007/3-540-63792-3_8.

12 Todd J. Green. Logiql: A declarative language for enterprise applications. In *PODS*, pages 59–64, 2015. doi:10.1145/2745754.2745780.

13 Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *FTDB*, 5(2):105–195, 2013. doi:10.1561/1900000017.

14 Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *DEBU*, 18(2):3–18, 1995.

15 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993. doi:10.1145/170035.170066.

16 Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

17 Terry Halpin. Object-role modeling (orm/niam). In *Handbook on architectures of information systems*, pages 81–103. Springer, 2006. doi:10.1007/978-3-662-03526-9_4.

18 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In *SLE*, pages 241–260, 2014. doi:10.1007/978-3-319-11245-9_14.

19 Felienne Hermans, Martin Pinzger, and Arie van Deursen. Automatically extracting class diagrams from spreadsheets. In *ECOOP*, pages 52–75, 2010. doi:10.1007/978-3-642-14107-2_4.

20 Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with morphj. In *PLDI*, pages 79–89, 2008. doi:10.1145/1375581.1375592.

21 Daniel Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2):256–290, 2002. doi:10.1145/505145.505149.

22 Yanhong A Liu, Jon Brandvein, Scott D Stoller, and Bo Lin. Demand-driven incremental object queries. *arXiv preprint arXiv:1511.04583*, 2015.

23 Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731, 2013. doi:10.1007/978-3-642-39038-8_29.

24 Erik Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *CUFP*, page 11, 2010. doi:10.1145/1900160.1900173.

25 Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: language-integrated live data views. In *OOPSLA*, pages 417–432, 2014. doi:10.1145/2660193.2660242.

26 Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.

27 H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002. doi:10.1145/581690.581695.

28 Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *SLP*, pages 204–218, 1994.

29 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *PODS*, pages 114–126, 1992. doi:10.1145/137097.137852.

30 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. In *AOSD*, pages 25–36, 2014. doi:10.1145/2577080.2577083.

31 Emma Söderberg and Görel Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Lund University, 2012.

32 Friedrich Steimann. Content over container: object-oriented programming with multiplicities. In *OOPSLA*, pages 173–186, 2013. doi:10.1145/2509578.2509582.

33 Friedrich Steimann. None, one, many - what's the difference, anyhow? In *SNAPL*, pages 294–308, 2015. doi:10.4230/LIPIcs.SNAPL.2015.294.

34 Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *MoDELS*, pages 653–669, 2014. doi:10.1007/978-3-319-11653-2_40.

35 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAMCOMP*, 1(2):146–160, 1972. doi:10.1137/0201010.

36 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007. doi:10.1007/978-3-540-88643-3_7.

37 Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL*, pages 199–210, 2007. doi:10.1145/1190216.1190248.

38 Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *ECOOP*, pages 28–49, 2006. doi:10.1007/11785477_3.

39 Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the java query language. In *OOPSLA*, pages 1–18, 2008. doi:10.1145/1449764.1449766.