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

Daco C. Harkes, Danny M. Groenewegen, and Eelco Visser
Delft University of Technology, The Netherlands
{d.c.harkes,d.m.groenewegen,e.visser}@tudelft.nl

Problem

Object-oriented programming languages allow specification of derived values through getters that contain the code that calculates the derived value. However, this implies calculate on (each) read. Changing to a cached implementation **requires code changes**.

Relational Databases provide views, materialized and non-materialized, for calculating derived values. However, views **limit expressiveness** by limiting recursive aggregation.

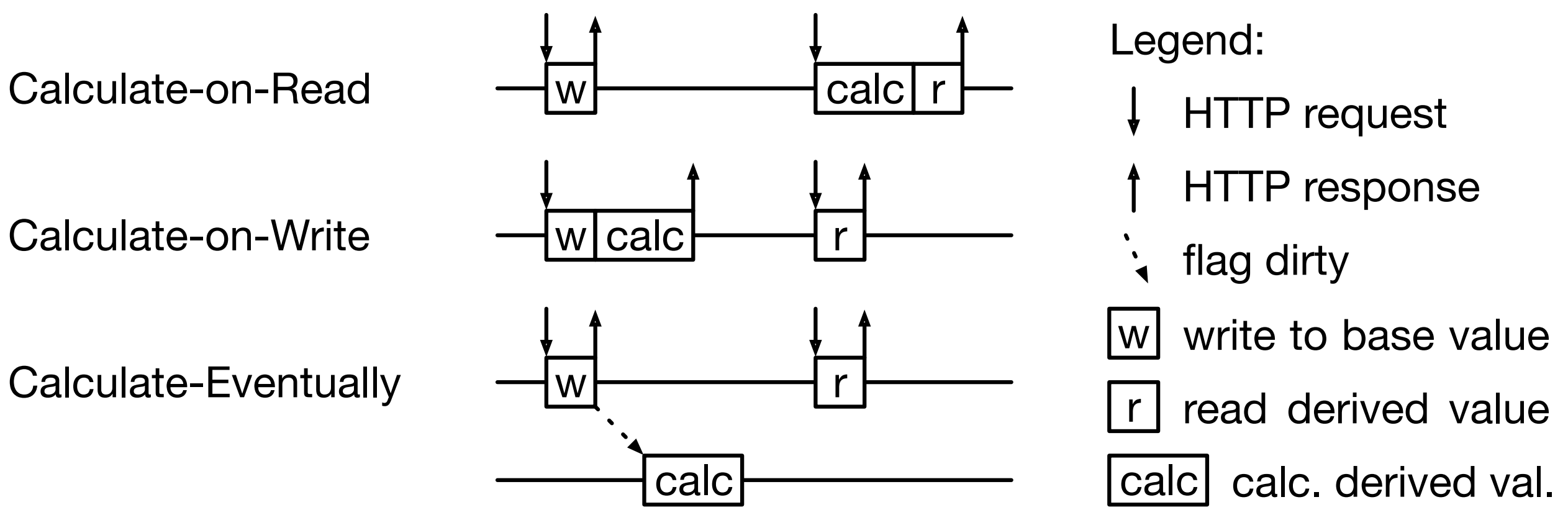
Solution

IceDust is a language which allows data modeling with **derived value attributes**, and provides multiple **calculation strategies** as compiler options. This provides separation of the functional specification from the calculation strategy.

An IceDust specification consists of **entities**, **attributes** (base values and derived values) of entities, and **bidirectional relations** between entities.

Calculation Strategies

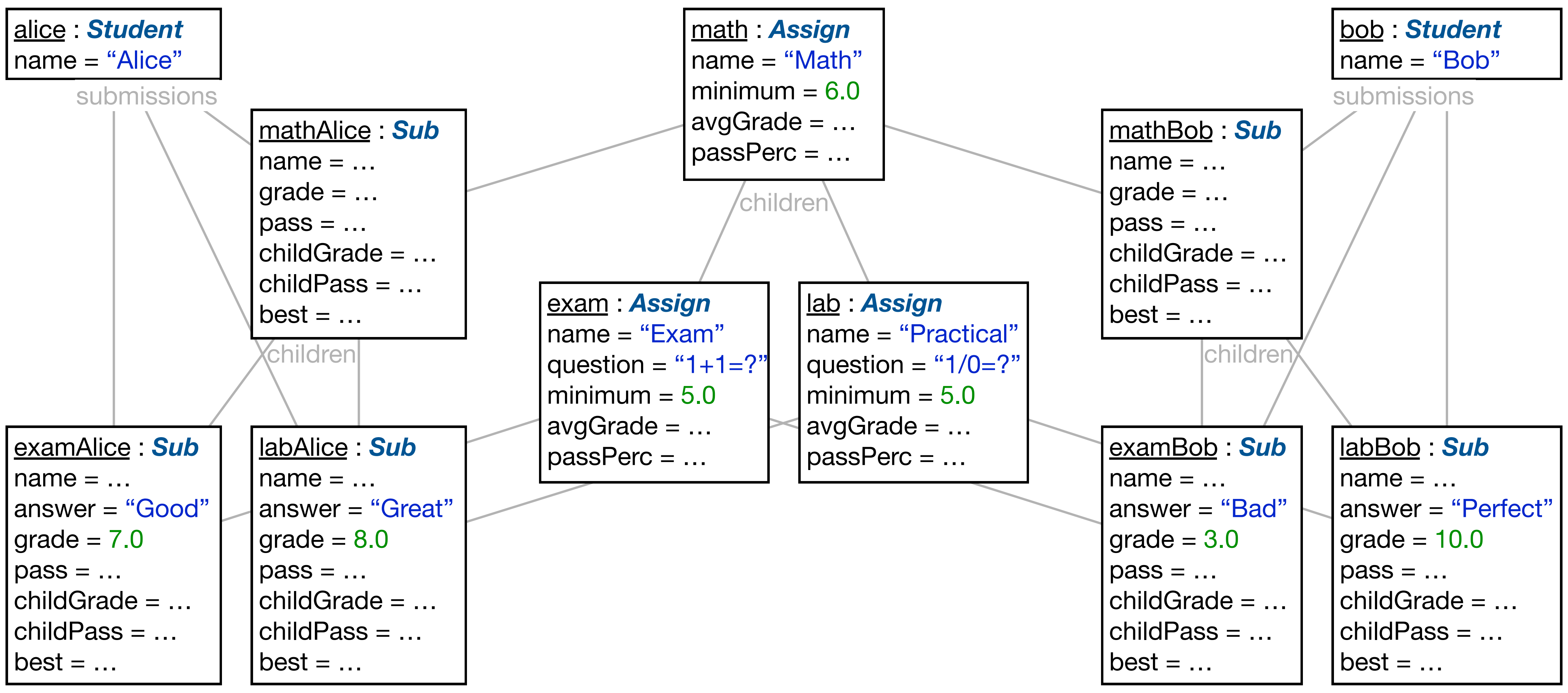
IIceDust provides three calculation strategies for calculating the values of attributes: **Calculate-on-Read**, **Calculate-on-Write** and **Calculate-Eventually**. The high level difference between these strategies is the moment that derived values are calculated:



Example IceDust Specification

```
entity Assignment {
  name       : String  Base Value Attribute
  question   : String?
  minimum    : Float?
  avgGrade   : Float?  = avg(submissions.grade)
  passPerc   : Float?  = sum(submissions.pass ? 1 : 0) / count(submissions)
}

entity Student {
  name       : String
}

entity Submission {
  name       : String  = assignment.name + " " + student.name  Derived Value Attribute
  answer     : String?

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

  best       : Boolean = grade == max(assignment.submissions.grade) <+ false
}

relation Assignment.parent     ? <-> * Assignment.children  Bidirectional Relation
relation Submission.parent     ? <-> * Submission.children
relation Submission.student    1 <-> * Student.submissions
relation Submission.assignment 1 <-> * Assignment.submissions
```

Example Data



alice : *Student*
name = "Alice"

math : *Assign*
name = "Math"
minimum = 6.0
avgGrade = …
passPerc = …

bob : *Student*
name = "Bob"

mathAlice : *Sub*
name = …
grade = …
pass = …
childGrade = …
childPass = …
best = …

mathBob : *Sub*
name = …
grade = …
pass = …
childGrade = …
childPass = …
best = …

exam : *Assign*
name = "Exam"
question = "1+1=?"
minimum = 5.0
avgGrade = …
passPerc = …

lab : *Assign*
name = "Practical"
question = "1/0=?"
minimum = 5.0
avgGrade = …
passPerc = …

examAlice : *Sub*
name = …
answer = "Good"
grade = 7.0
pass = …
childGrade = …
childPass = …
best = …

labAlice : *Sub*
name = …
answer = "Great"
grade = 8.0
pass = …
childGrade = …
childPass = …
best = …

examBob : *Sub*
name = …
answer = "Bad"
grade = 3.0
pass = …
childGrade = …
childPass = …
best = …

labBob : *Sub*
name = …
answer = "Perfect"
grade = 10.0
pass = …
childGrade = …
childPass = …
best = …

Dependency Analysis

IceDust specifications define the value of attributes in terms of other attributes. The Calculate-on-Write and Eventually-Consistent strategies require dependency and data flow information.

Dependency analysis is done with **path-based abstract interpretation**. For example the dependency paths of the pass attribute are:

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

IceDust does not have statements, and thus no control flow. As such the data flow paths are the inverses of the dependency paths. Paths can be inverted by inverting the bidirectional relations in the paths. For example the data flow paths to the pass attribute are:

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

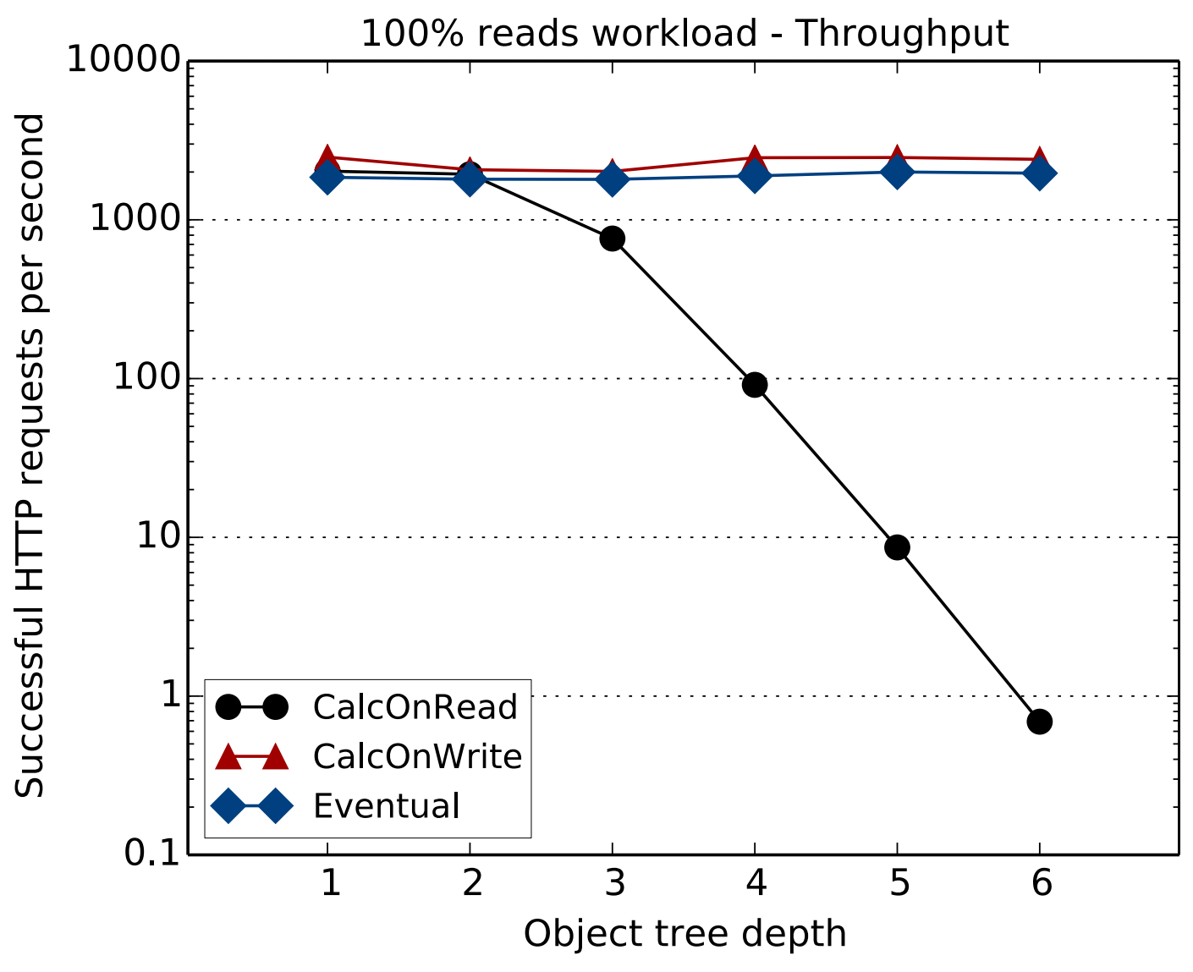
Benchmarks

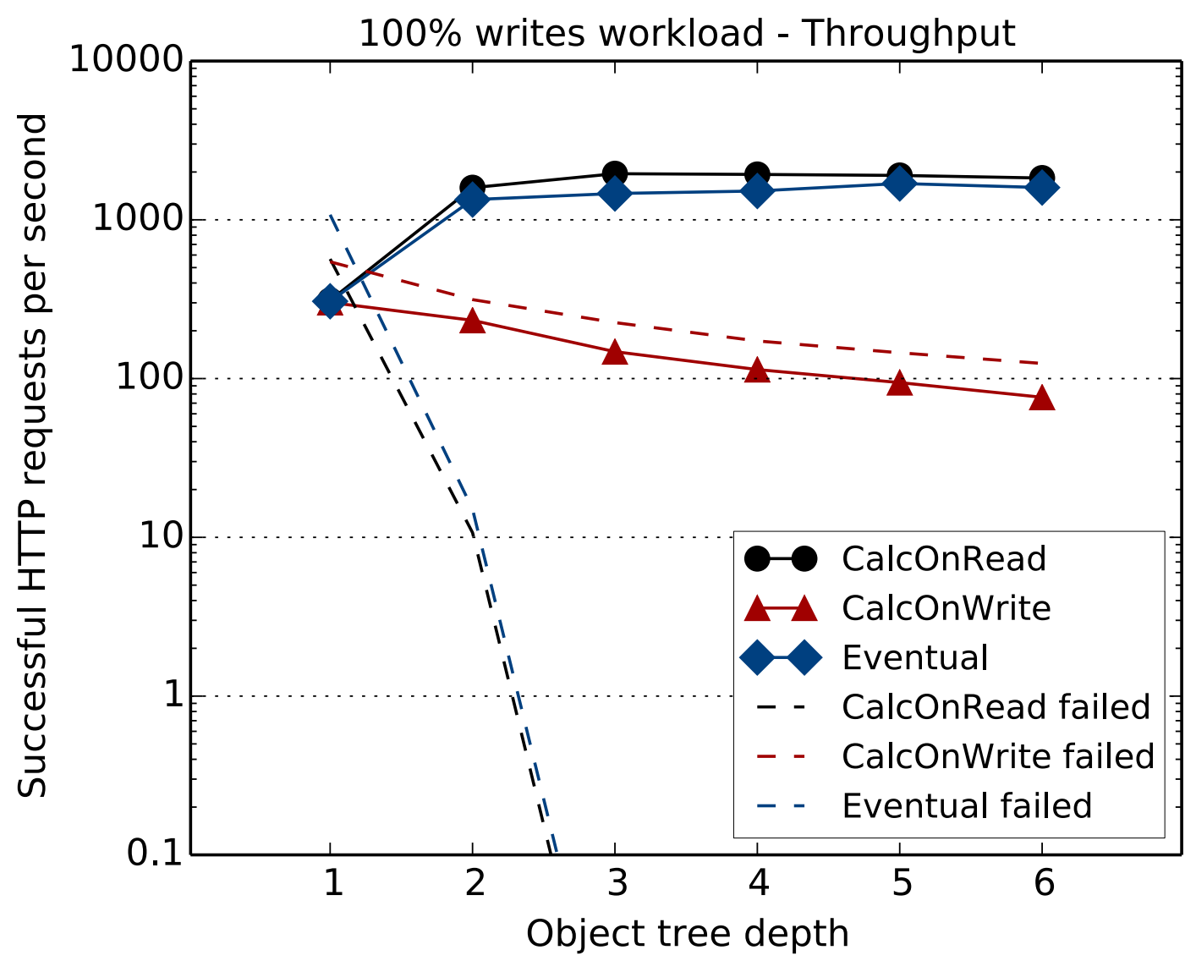
We benchmarked derived values that depend on up to 100000 base values in varying workloads. We calculate a recursive aggregate in a tree. (See benchmark specification below.) The trees have a branching factor of 10.

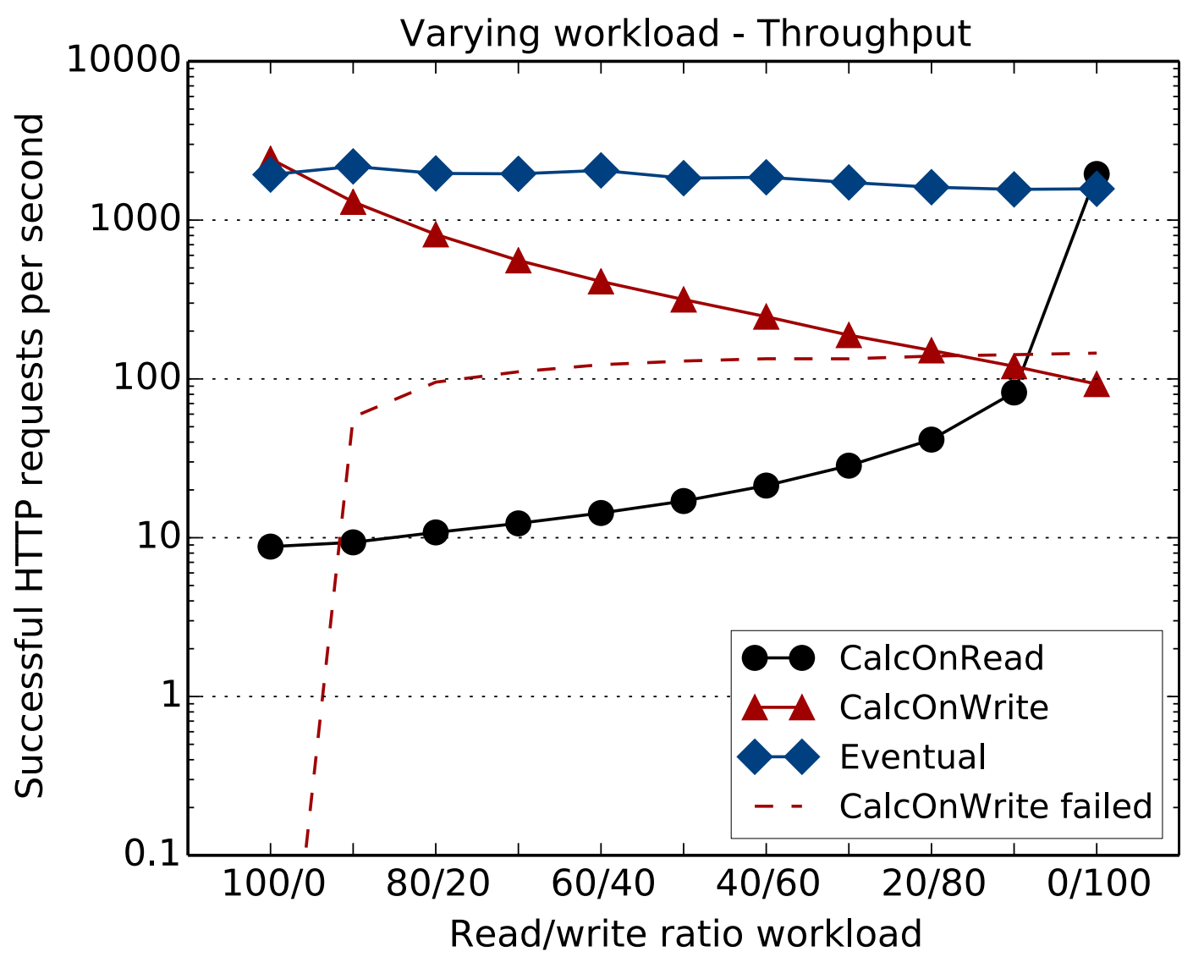
The first two benchmarks test the throughput on read only and write only workloads, with the derived value on top of the tree depending on a varying number of base values. Calc-on-Read performance on reads suffers when derived values depend on many base values. Calc-on-Write performance suffers on writes as many concurrent writes to the derived value cache cause failing database transactions.

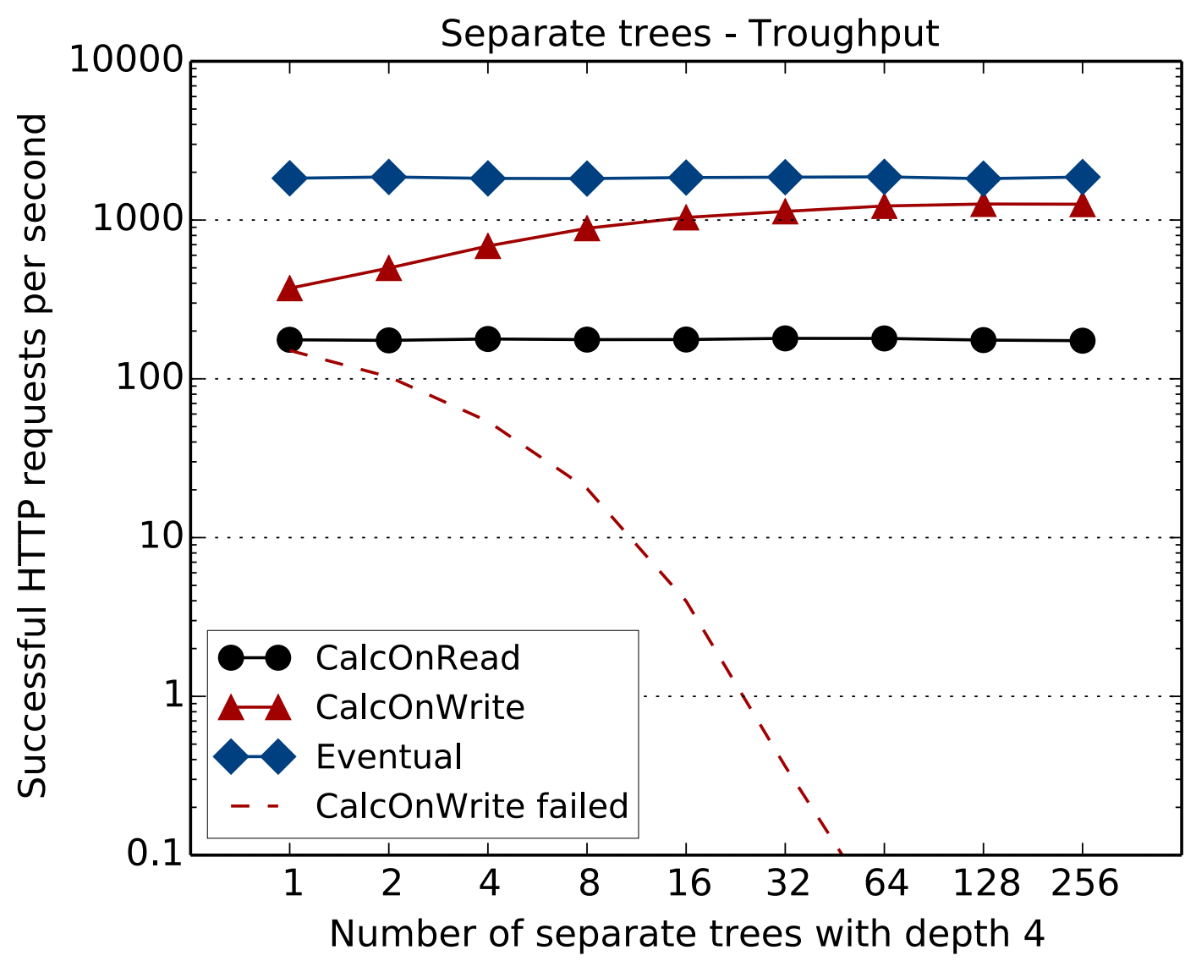
The third benchmark tests a varying workload with a tree of depth 5. Calc-on-Read and Calc-on-Write performance is bad on all workloads, while Calculate-Eventually keeps a steady performance.

The fourth benchmark varies the number of trees, so that there are multiple, completely separated derived values, with a 50/50 workload. Calc-on-Write performs good for completely separated calculations.


100% reads workload - Throughput


100% writes workload - Throughput


Varying workload - Throughput


Separate trees - Troughput

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

Calculation Strategy Implementations

The different calculation strategies require different code patterns to be generated by the compiler. Below is a snippet of meta-code that generates part of the Calc-on-Read and Calc-on-Write implementation.

```
for a : T m = e1 in E.attributes

  function calculate_a() : T {  return e1;  }

  // Calculate-on-Read
  function get_a() : T {  return calculate_a(); }

  // Calculate-on-Write
  a : T
  static a_dirty : Set<E>
  function get_a() : T {  return this.a;  }
  function update_a() {  a := calculate_a();  }

for E.a → path.a2 in DataFlow where a2.entity=E2

  // Calculate-on-Write
  function set_a(newV : T){
    if(a != newV){  a := newV; E2.a2_dirty.addAll(path);  }
  }

// Calculate-on-Write
static function update_derived_values() {
  // go through all dirty and update until all empty
}

// Eventually-Consistent
// Same as calc-on-write, but dirty flag to separate thread
```

Harkes, D. C., Groenewegen, D. M., Visser, E.: *IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs.* ECOOP (2016)
Harkes, D. C., Visser, E.: *Unifying and Generalizing Relations in Role-Based Data Modeling and Navigation.* SLE (2014)
Steimann, F.: *Content over container: object-oriented programming with multiplicities.* ONWARD! (2013)
Visser, E.: *WebDSL: A case study in domain-specific language engineering.* GTTSE (2007)

TU Delft