IceDust Calculation Strategy Composition Performance in Web Applications

Daco C. Harkes[1] and Eelco Visser[1]

[1]Delft University of Technology {d.c.harkes,e.visser}@tudelft.nl

1 Introduction

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.

IceDust is a data modeling language for expressing derived attribute values without committing to a calculation strategy. IceDust provides four strategies for calculating derived values in persistent object graphs: `on-demand`, `incremental`, `eventual`, and `on-demand incremental`. The first three were introduced at ECOOP last year [2]. The new strategy is inspired by Adapton [1]: it flags caches dirty transitively on writes, and only recomputes caches on reads (Figure 1). At this ECOOP we present IceDust 2 [3], which enables calculation strategy composition. For every field with a derived value a strategy can be selected. The IceDust 2 type system restricts composition to only sound composition, ensuring derived values are up to date when read (except for `eventual` calculation). Moreover, we have extended IceDust with `inline` attributes. The expressions of these attributes are inlined on their use site (and thus cannot be recursive). This allows us to control the granularity of caching in IceDust.

The four strategies and inlining provide us with many possible options for calculating derived values in IceDust applications. In this extended abstract we benchmark various options for a single application under peak load.

2 Application, architecture, and load

We use IceDust for building web applications. We benchmark a learning management system in which students receive grades for programming assignments. Student final grades for a course are expressed as derived values over the individual programming grades, and the course statistics are expressed as derived values over the student final grades.

This web application has a traditional architecture. Data is stored in a relational database, HTTP requests are handled concurrently, and every request has its own object-relational mapping. The database (MySQL) uses optimistic locking, so concurrent overlapping edits can fail.

The learning management system is under peak load during exams. Around 300 students submit Scala programs which are automatically graded (on another server), and the teacher can see live grade statistics during the exam.

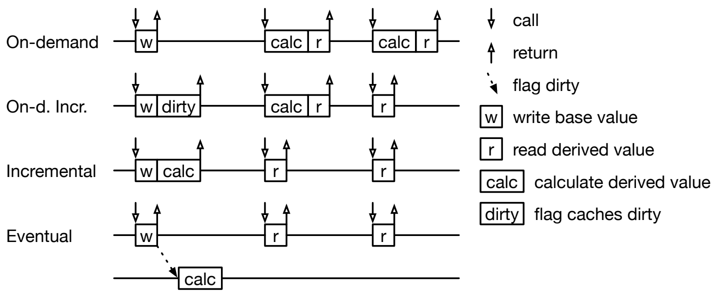

Figure 1: Thread activation diagrams of calculation strategies

```
entity Node1 {
  avgValue : Float? =
    avg(children.avgValue)                (on—demand)
}
relation Node1.children * <—> ? Node2.parent
entity Node2 {
  avgValue : Float? =
    avg(children.avgValue) (default)      (incremental)
}
relation Node2.children * <—> ? Node2.parent2
```

Figure 2: Synthetic benchmark program

3 Benchmarks

The essence of the grade calculation in our learning management system is a tree-like structure with aggregations on every level. So, first we do a micro-benchmark with recursive aggregation under strategy composition. We create a tree with depth 5 and a branching factor of 10. At the leaf nodes we assign values, and at all non-leaf nodes we compute the average of the children. To benchmark strategy composition we use a different strategy at the top node, `Node1`, than for the other nodes, `Node2`s (Figure 2).

During the benchmark we vary the number of subsequent reads and writes and measure how many HTTP request we can service per second. The benchmarks consist of 30 seconds start up, and 10 times 10 seconds of load. The figures show the average of the 10 runs, with bars showing the slowest and fastest run.

Figure 3 shows the results of our micro benchmark. Strategies `on-demand`, `incremental`, and `eventual` perform identical to our previous work [2]: `on-demand` performs horrible on reads, `incremental` performs horrible on concurrent writes, and `eventual` performs well under any load. The new strategy `on-demand incremental` outperforms both `on-demand` and `incremental` on most workloads. This is because it has the same concurrent conflicts as `incremental`, but it has less objects to load from database each request. Interestingly enough `on-demand incremental` does not perform well when

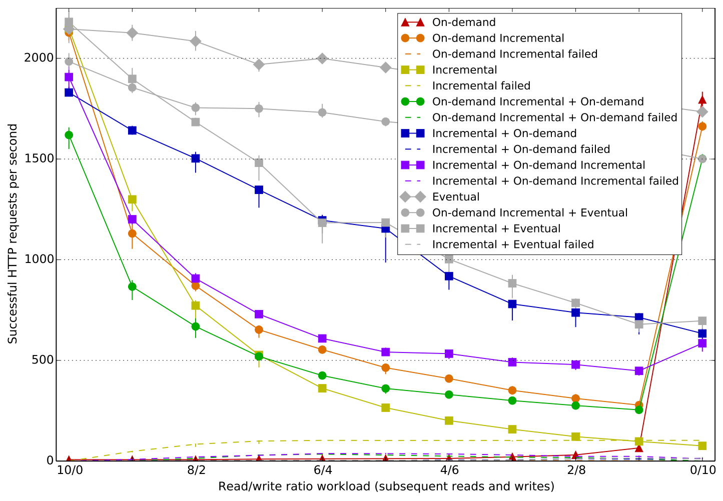

Figure 3: Throughput for micro benchmark with 11111 nodes

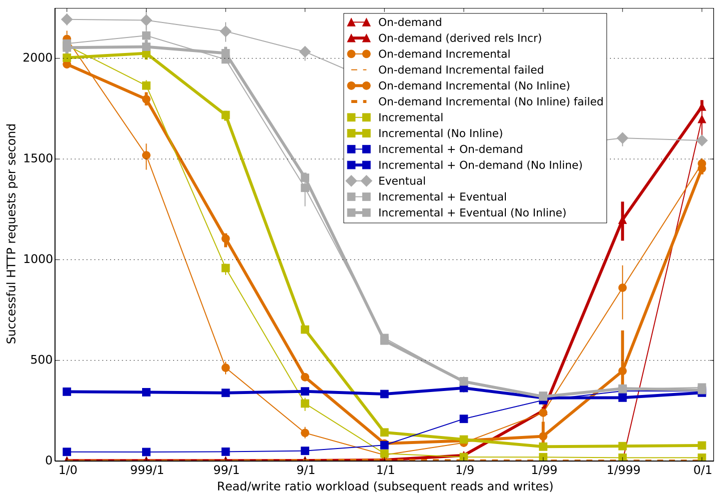

Figure 4: Throughput for full program with real data

used with `on-demand` for the top node (green). A read of the top node forces updates of the cache of its 10 children, which creates a larger transaction footprint, and thus more conflicts. If we use `incremental` for the tree and `on-demand incremental` (purple), or `on-demand` (blue) on top, we do get better performance. Especially blue performs well, as concurrency conflicts are minimized.

Next, we check whether these performance characteristics carry over to our full learning management system, and analyze the influence of changing the caching granularity. The IceDust specification of the systems derived values is 300+ lines, 14 entities, and 68 derived values. Out of these 68 derived values, 61 can be `inline`, while 7 are recursive and cannot be inlined. The data set is 200 students, a total of 300 assignments in the course (including non-graded practise assignments), and 13000 student submissions. During the exam 700 Scala programming submissions were edited. We benchmarked writing to these 700 submissions, while reading only the exam average grade (not the top node of the assignment tree, which would be the average course grade).

Figure 4 shows the benchmark results. In general the performance of the various strategies in the micro benchmark carries over to the full program: `on-demand` performs only on writes, `incremental` performs only on reads, `on-demand incremental` performs well on only reads and writes but not on mixed workloads, and the composition of `incremental` and `on-demand` (blue) performs quite well on any workload. (Note that the aggregation for course statistics is now over 200 students, rather than 10 children, explaining the lower performance on reads for `incremental + on-demand`.) It is clearly visible that inlining has a negative effect on performance (for all non-`eventual` strategies). This can be explained by the fact that inlining makes the granularity of caching larger, which in turn means that each HTTP request reads many more (possibly written to) fields, increasing concurrency conflicts. This is a classic trade-off between disk (and memory) space and performance.

4 Discussion

We do not expect these results to carry over to other applications, architectures, or load patterns. We have observed two limiting factors in the performance of IceDust-based web applications: (1) the number of objects that has to be loaded from database into memory per request, and (2) the footprint of concurrent transactions (write-write conflicts or read-write conflicts). In both benchmarks all values get (recursively) aggregated into a single derived value, but if derived values in applications depend on less values the number of conflicts is drastically reduced [2]. If the architecture does not support concurrency (or the load is not concurrent) conflicts will not happen at all. Moreover, if the application architecture does not include object-relational mapping but has all data in memory, then number of objects read and written does not matter.

IceDust has become a toolbox of building blocks, to build incremental applications. Which building blocks to use depends on the derived values in the application, the application architecture, and the load pattern on the application. In future work we would like to benchmark various applications, architectures, and load patterns to learn more about which strategies (and compositions) work in which scenarios.

References

[1] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *PLDI*, page 18, 2014.

[2] Daco Harkes, Danny M. Groenewegen, and Eelco Visser. Icedust: Incremental and eventual computation of derived values in persistent object graphs. In *ECOOP*, 2016.

[3] Daco Harkes and Eelco Visser. Icedust 2: Derived bidirectional relations and calculation strategy composition. In *ECOOP*, 2017.