# YAMTL Solution to the TTC 2018 Social Media Case

Artur Boronat
Department of Informatics, University of Leicester, UK
aboronat@le.ac.uk

**Abstract**

Software models raise the level of abstraction of software artefacts involved in the design, implementation and testing phases of software systems. Such models may be used to automate many of the tasks involved in them, where queries play an important role. Moreover, some of those models may be inferred automatically from existing software artefacts, e.g., by means of reverse engineering, yielding potentially very large models (VLMs). Technology to analyse VLMs efficiently enables the application of model-driven software development in industry and is the subject of study in the TTC 2018 Social Media Case. YAMTL is both a model transformation (MT) language that is available as an internal DSL of Xtend and a companion MT engine that can be used from any JVM application and that supports incremental execution of MT. In this paper, we present the YAMTL solution to the social media case and discuss its performance, scalability and memory usage w.r.t. the reference solution. The YAMTL solution was deemed to be the *most scalable solution* at the TTC 2018.

## 1 Introduction

The goal of the TTC 2018 Social Media Case [Hin18] aims at performing two analyses of a social media network, represented as an instance of a static object model of a social network, in order to find the three most influential posts and comments according to certain criteria. Both analyses should be implemented as model queries and should be applied over a specific network. The networks to be analysed are subject to updates and an important challenge of the case is to refresh the conclusions from each analysis in the most efficient manner after each network update.

In this paper, we present a solution to the case with YAMTL [Bor18], a MT language available from within Xtend [Fou18] and its companion MT engine that can execute such model transformations in an incremental way. Therefore, the analyses to be performed are implemented using model transformations and YAMTL's machinery for tracking model dependencies, for detecting the impact of an udpate and for re-executing only the affected part of the model transformation are reused in order to implement model queries. The relevant excerpts of YAMTL for defining MT are explained in section 2 and the YAMTL solution is discussed in section 3. Finally, the solution is discussed in section 4 using the performance results obtained with the case benchmark framework along its evaluation criteria [Hin18].

## 2 YAMTL Outline

YAMTL is a MT language provided as an internal DSL of Xtend [Fou18] that augments it with MT modules, which can be used to specify declarative MTs. In this subsection, the excerpt of YAMTL that is relevant to



In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 11th Transformation Tool Contest, Toulouse, France, 29-06-2018, published at http://ceur-ws.org

the solution is presented. In particular, MT modules used to implement queries consist of helpers and of rules, whose computation logic is performed in the source pattern, at matching time. Hence, the interpretation of a query is performed by the YAMTL pattern matcher and rules are scheduled but not fully executed.

The declaration of a YAMTL transformation module starts by creating a specialization of the class `YAMTLModule`, as shown in Appendix A.2. Within its constructor, the `header()` of the transformation defines its signature, declaring its input and output models, and the `ruleStore()` contains the declaration of rules.

Each rule has an input pattern for matching variables and an output pattern for creating objects. An input pattern consists of `in` elements together with a global rule filter condition, which is true if not specified. Each of the `in` elements is declared with a variable name, a type and a local filter condition, which is true if not specified. A rule is applied whenever a match for each `in` variable is found such that both the corresponding local filter conditions and the global filter condition are satisfied. In each filter condition, the expression `'variable'.fetch as Type` fetches the value of `'variable'` from the execution environment. This is normally used to access matched objects in a filter expression. An output pattern consists of `out` elements, each of which is declared with a variable name, a type and an action block. Filter conditions and action blocks are specified as non-pure lambda expressions in Xtend. In the YAMTL solution presented in this paper, output pattern elements will be declared as *no-op* as we are only interested in the query side of a transformation rule.

When loading a YAMTL transformation definition, YAMTL initializes transformation rules by inserting their abstract representation in the rule store. Once the rule store is initialized, rules are type checked. If no error arises, the pattern matcher finds all rule matches and schedules them. During this phase, the input pattern elements are ordered by the size of their type extent and YAMTL finds a match for a rule by first mapping matched `in` elements to objects that satisfy the corresponding filter condition. When a total match is found[1], the satisfaction of that match is finally asserted by the rule filter condition.

A rule is scheduled, and traced, as a transformation step, which consists of a labelled pair of two matches, the match for the input pattern of the rule, which enables its application, and the match for the output pattern of the rule, which is only initialized when the transformation step is executed. This last step is skipped when scheduling transformation rules only, for executing queries, using the execution phase `MATCH_ONLY`.

The MT engine has been extended with an incremental execution mode, which consists of two phases: the *initial phase*, the transformation is executed in batch mode but, additionally, tracks feature calls in objects of the source model involved in transformation steps as *dependencies*; and the *propagation phase*, the transformation is executed incrementally for a given source update and only those transformation steps affected by the update are (re-)executed. This means that components of YAMTL's execution model have been extended but the syntax used to define model transformations is preserved. Hence, a YAMTL batch model transformation can be executed in incremental mode, without any additional user specification overhead.

In YAMTL, model udpates are represented using the Eclipse Modeling Framework change model[SBPM09]. Given a model update $\delta_s$ between a source model $M_s$, already synchronized with a target model $M_t$ via a model transformation $t : M_s \xrightarrow{*} M_t$ (where $\xrightarrow{*}$ denotes a sequence of transformation steps), and an updated source model $M'_s$, YAMTL propagates the model update $\delta_s$ along $t$. The effect of this forward propagation is the application of an update $\delta_t$ on the target model $M_t$.

## 3 The YAMTL Solution

The solutions to both task one and task two exploit a common functional requirement − both of them require the selection of the three best submissions and both queries only remember these three elements during query evaluation. For each computationally-relevant statement in a query that uses universal quantification we define a model transformation rule, where the input pattern consists of a single `in` element, in which the variable corresponds to the universal variable and the filter condition corresponds to the predicate preceded by the quantifier. The score used to rank each submission is computed in the filter.

The YAMTL solutions employ two auxiliary data structures, one for storing the `threeBestCandidates`, when their score is greater than zero, and a list of `candidatesWithNilScore`. When a `submission` is processed, by evaluating the filter condition of the corresponding rule, if its score is zero, it is added to `candidatesWithNilScore`. Otherwise, if its `score` is greater than zero, it is added to `threeBestCandidates` using the method `Util.addIfIsThreeBest(list, submission, score)`, shown in Appendix A.1, which inserts the candidate in the list if its score is greater than the score of one of the elements in the list. The insertion

[1] YAMTL supports more complex source patterns with: several `in` elements, which may be functionally dependent on previously matched `in` elements; and derived `in` elements [Bor18].

process starts from the end of the list (i.e., from the current best candidate with the smallest score) and proceeds towards the head of the list if the score of the element to be inserted is greater than the current best candidate score. When the submission reaches the first position of the list, it is inserted. When the score of a current best candidate (in either the first or the second position) cannot be improved, the submission is inserted after it. When there is a tie, submission timestamps are used to resolve it (using the most recent submission as the better candidate). If the insertion causes the list to grow, the last element is trimmed so that only the three best candidates are kept at any one time.

A query is then evaluated by executing the corresponding model transformation in `MATCH_ONLY` mode. The result is obtained with the expression `(xform as TaskXModule).getBestThree().map[id].join('|')`, which completes the list of best candidates with submissions with nil score if its length is smaller than three by using the method `Util.getBestThree(weightedList, nilScoreSubmissionList)`, shown in Appendix A.1. When the list of best candidates is shorter than three, this method sorts the list of submissions with nil score, using their timestamps, before selecting as many submissions as needed to complete the list of `threeBestCandidates`. This means that sorting submissions with nil score is skipped quite frequently.

The integration of the solution into the benchmark framework is achieved by extending the harness class `Solution` with three methods that need to be overriden: the constructor, where the transformation is configured; `Initial()`, which performs the initial transformation; and `Update(delta)`, which recomputes the query for a given model udpate. In the batch variant of the solution, the initial step is achieved by invoking the method `xform.execute()` and the update step is achieved by resetting the internal caches and the lists `threeBestCandidates` and `candidatesWithNilScore`, by applying the source update with `xform.applyDelta('sn', deltaName)`, and by applying the transformation from scratch with `xform.execute()`. The benchmark harnesses for tasks one and two can be found in B.1.1 and B.1.2, respectively. In the incremental variant of the solution, the initial step is performed as in the batch variant but the update step is achieved by propagating the delta identified by `deltaName` using the expression `xform.propagateDelta('sn', deltaName)`, which applies the delta to the source model and then re-executes the model query only for the parts of the model that have been modified. The benchmark harnesses for tasks one and two can be found in B.2.1 and B.2.2, respectively.

In the following subsections, the discussion focusses on task-specific query details that are not common to both tasks. The complete code of the implementation of the queries in YAMTL can be found in Appendix A.2 for task one and in in Appendix A.3 for task two. It is important to note that the query solutions using YAMTL modules are the same for both variants (batch and incremental).

### 3.1 Task 1: Most Controversial Posts

The goal in the first task is to obtain the most controversial posts, that is those which spark off more comments. The query is implemented as a transformation rule, whose input pattern is formed by an `in` element that will match posts that contain comments as indicated in the filter. In particular, the method `post.getAllComments()` fetches all contained `Comment`s within the matched `post`. If the list of contained comments has elements, the score of the post is computed by adding, for each comment, the sum of 10 plus the amount of users that liked the comment. Then the post is recorded with the method `Util.addIfIsThreeBest(list, submission, score)`, which checks whether the post becomes a best candidate or not and stores it in `bestThreeCandidates` accordingly. When there are no contained comments in the matched post, the post is inserted into `candidatesWithNilScore`. The implementation of the query is shown in Listing 1.

### 3.2 Task 2: Most Influential Comments

The goal in the second task is to obtain the most influential comments by using a metric over the groups of users that liked a particular comment. Groups of users who liked a comment are defined as the equivalence classes induced by the friendship relation. This amounts to defining strongly connected components in the graph where the nodes are users who liked a comment and the bidirectional edges are friendship links. The search of strongly connected components has been implemented using depth-first search, an idea originally introduced in [HT73], as shown in Appendix A.3. The expression `val fc = new FriendComponentUtil(comment.likedBy)` computes the connected components of the graph whose nodes are the set of users who liked the comment, i.e., `comment.likedBy`. The expression `fc.components.map[c | c.size * c.size].sum` then computes the sum of squares of the size of each component in the graph. Similarly to the query of task one, when the graph has nodes the score that is computed is associated with the comment and inserted into the list `threeBestCandidates` if its

```
new Rule('CountPosts')
  .in('post', SN.post).filter[
    val post = 'post'.fetch as Post
    var int score = 0
    var matches = false
    val commentList = post.getAllComments()
    if (commentList?.size > 0) {
      score = commentList.map[c | 10 + c.likedBy.size].sum
      threeBestCandidates.addIfIsThreeBest(post, score)
      matches = true
    } else candidatesWithNilScore.add(post)
    matches
  ].build()
  .out('postAux', SN.post, []).build()
.build()
```

Listing 1: Solution to Task One (Most Controversial Posts).

```
new Rule('UserComponentsByComment')
  .in('comment', SN.comment).filter[
    val comment = 'comment'.fetch as Comment
    var score = 0
    var matches = false
    if (comment.likedBy.size > 0) {
      val fc = new FriendComponentUtil(comment.likedBy)
      score = fc.components.map[c | c.size * c.size].sum
      threeBestCandidates.addIfIsThreeBest(comment, score)
      matches = true
    } else candidatesWithNilScore.add(comment)
    matches
  ].build()
  .out('commentAux', SN.comment, []).build()
.build()
```

Listing 2: Solution to Task Two (Most Influential Comments)

score improves the score of any of the current best candidates using the method `Util.addIfIsThreeBest(list, comment, score)`. When the graph has no elements the comment is inserted into `candidatesWithNilScore`. The implementation of the query is shown in Listing 2.

## 4 Evaluation and Conclusions

According to the evaluation criteria proposed [Hin18], completeness has been addressed by considering all of the proposed models (of different sizes) and their model updates. In addition, the solutions pass the correctness criteria defined in the benchmark framework test suite. Below we discuss the rest of the proposed evaluation criteria.

### 4.1 Understandability and Conciseness

YAMTL is a MT language that does not provide implicit support for incremental query evaluation at present. However, given that queries can be defined as rule filter conditions and that model transformations can be executed incrementally, YAMTL provides all the ingredients that are required to solve the proposed case. This means that the solution introduces some noise by (1) enabling universal quantification using transformation rules, whose output pattern is immaterial, and (2) by storing the best candidates for each task in auxiliary data structures. On the one hand, comprehending the query, involves understanding the declaration of model-to-model transformation rules in YAMTL, which amounts to understanding how a conditional transformation rule is matched. On the other hand, the declaration of transformation rules is somewhat more verbose than other MT languages available as external DSLs (e.g., ATL).

### 4.2 Performance and Scalability

In this section, the experimental results obtained from running the benchmark accompanying the case for the YAMTL solution are discussed and compared with the results obtained for the reference solution, NMF [Hin15]. Each solution was run in two execution modes: batch and incremental, which we denote with the suffixes *-batch* and *-incr* respectively. The results are plotted per solution (and execution mode) using three graphs (both for time and for memory usage), as shown in Appendix C: one for the results obtained in the transformation of the

initial model (initial phase); one for the results obtained in the re-execution of the query after the twenty source updates (update phase); and a final one combining the results from the previous two phases (combined phase). The results were obtained from ten runs of the benchmark. For the initial phase, for each model size the median of the ten results was selected as the representative one. For the update phase, the median of the times reported per update was selected and then the medians of each update were aggregated per model size. The benchmark was run on a MacBookPro11,5 Core i7 2.5 GHz with 16 GB of RAM with .NET Core 2.1.301 and JRE (build 1.8.0_181-b13) and Java HotSpot (build 25.181-b13, mixed mode).

In the initial phase, *NMF-batch* shows better performance for small models. However, as the size increases the distance in used time between *NMF-batch* and *YAMTL-batch* shortens. *YAMTL-batch* outperforms *NMF-batch* for models of size larger than 256 in task one, and for models of size larger than 32 in task two. On the other hand, *YAMTL-incr* also outperforms *NMF-batch* for models of size larger than 128 in task two. *YAMTL-incr* is several orders of magnitude more efficient than *NMF-incr*, probably due to the amount of caching performed during query evaluation. YAMTL's solution uses a MT to implement the query and only caches those transformation steps that are computationally relevant.

In the update phase, *NMF-batch* is faster than *YAMTL-batch* for small sizes but this time only up to size 8 for task one and up to size 2 for task two. To put results into perspective, warm up times used by the HotSpot VM and by .NET Core CLR are not considered by the benchmark and these may affect the first results in each execution. In the incremental variants, however, the opposite phenomenon is observed, *YAMTL-incr* outperforms *NMF-incr* for small sizes, up to 256 in task one and up to 128 in task two, and then *NMF-incr* takes over. The bottleneck in YAMTL is currently caused by the application of some deltas by the method `ChangeDescription.apply()` of the EMF change model API, a third-party dependency of YAMTL.

When combining reported times for the initialization and update phases, the most scalable approach is *YAMTL-incr*, which is also the most efficient from models of size 8 in task one and from models of size 4 in task two, followed by *YAMTL-batch*. For smaller sizes, *NMF-batch*, is more efficient.

Regarding memory usage, *YAMTL-batch* is the thriftiest of all solutions, in the three phases, closely followed by *YAMTL-incr*. *NMF-batch* shows better scalability than NFM-incr due to the absence of caching mechanisms used in incremental query re-evaluation.

**Acknowledgements**

The author would like to thank both the organizers and the participants of the tool transformation contest (TTC) 2018 for the discussion on incremental techniques.

## References

[Bor18] Artur Boronat. Expressive and efficient model transformation with an internal dsl of xtend. In *Proceedings of the 21th ACM/IEEE International Conference on MoDELS*, pages 78–88. ACM, 2018.

[Fou18] The Eclipse Foundation. Xtend (official web page), 2018. `http://www.eclipse.org/xtend/`.

[Hin15] Georg Hinkel. Change propagation in an internal model transformation language. In *ICMT*, volume 9152, pages 3–17. LNCS, 2015.

[Hin18] Georg Hinkel. The TTC 2018 Social Media Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, June 2018.

[HT73] John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

# A Auxiliary Code

## A.1 Common Utility Functions Used in Task 1 and Task 2

```
class Util {
  def static getBestThree(List<Pair<Submission,Integer>> weightedList, List<Submission> completionList) {
    val list = new ArrayList<Submission>(weightedList.map[key])
    if (list.size<3) {
      val numberOfMissing = 3 - list.size
      val orderedList = completionList.sortInplace([c1,c2|
        - c1.timestamp.compareTo(c2.timestamp)
      ])
      list.addAll(orderedList.take(numberOfMissing))
    }
    list
  }

  // start with last
  def static void addIfIsThreeBest(List<Pair<Submission,Integer>> list, Submission submission, int score) {
    addIfIsThreeBest(list, submission, score, list.size-1)
  }
  def static private void addIfIsThreeBest(List<Pair<Submission,Integer>> list, Submission submission, int
     score, int index) {
    if (index == -1) {
      list.add(0, submission -> score)
      if (list.size > 3)
        list.remove(list.last)
    } else {
      val element = list.get(index)
      if (element.key==submission) {
        list.remove(index)
        addIfIsThreeBest(list, submission, score, index-1)
      } else {
        if (score == element.value) {
          if (submission.timestamp.after(element.key.timestamp)) {
            // advance a position to check if there is another
            // submission with the same value and smaller timestamp
            if ((index==0) || (index>=1 && list.get(index-1).value == score))
              addIfIsThreeBest(list, submission, score, index-1)
            else
              list.add(index, submission -> score)
              if (list.size > 3)
                list.remove(list.last)
          } else {
            if (index<2) {
              list.add(index+1, submission -> score)
              if (list.size > 3)
                list.remove(list.last)
            }
          }
        } else if (score > element.value) {
          // advance a position to check if there is another
          // submission with a smaller value
          addIfIsThreeBest(list, submission, score, index-1)
        } else {
          if (index<2) {
            list.add(index+1, submission -> score)
            if (list.size > 3)
              list.remove(list.last)
          }
        }

      }
    }
  }

  def static sum(Iterable<Integer> list) {
    list.reduce[v1, v2 | v1+v2]
  }
}
```

## A.2 Task1Module: solution to task one

```
class Task1Module extends YAMTLModule {
  val SN = SocialNetworkPackage.eINSTANCE

  @Accessors
  val List<Pair<Submission,Integer>> threeBestCandidates = newArrayList

  @Accessors
  val List<Submission> candidatesWithNilScore = newArrayList

  new () {
    header().in('sn', SN).out('out', SN)

    ruleStore( newArrayList(
      new Rule('CountPosts')
        .in('post', SN.post)
          .filter[
            val post = 'post'.fetch as Post
            var int score = 0
            var matches = false
            val commentList = post.getAllComments()
            if (commentList?.size > 0) {
              score = commentList.map[c | 10 + c.likedBy.size].sum
              threeBestCandidates.addIfIsThreeBest(post, score)
              matches = true
            } else {
              candidatesWithNilScore.add(post)
            }
            matches
          ]
          .build()
        .out('postAux', SN.post, []).build()
        .build()
    ))
  }

  // HELPERS
  def getBestThree() {
    threeBestCandidates.getBestThree(candidatesWithNilScore)
  }
  def static List<Comment> getAllComments(Submission submission) {
    if (submission.comments?.size>0) {
      val commentList = newArrayList
      commentList.addAll(submission.comments)
      for (comment: submission.comments) {
        val list = comment.getAllComments
        if (list!==null) commentList.addAll(list)
      }
      commentList
    }
  }
}
```

### A.3 Task2Module: solution to task two

```
class Task2Module extends YAMTLModule {
  val SN = SocialNetworkPackage.eINSTANCE
  @Accessors
  val List<Pair<Submission,Integer>> threeBestCandidates = newArrayList
  @Accessors
  val List<Submission> candidatesWithNilScore = newArrayList
  new () {
    header().in('sn', SN).out('out', SN)
    ruleStore( newArrayList(
      new Rule('UserComponentsByComment')
        .in('comment', SN.comment)
          .filter[
            val comment = 'comment'.fetch as Comment
            var score = 0
            var matches = false

            if (comment.likedBy.size > 0) {
              val fc = new FriendComponentUtil(comment.likedBy)
              score = fc.components.map[c | c.size * c.size].sum
              threeBestCandidates.addIfIsThreeBest(comment, score)
              matches = true
            } else {
              candidatesWithNilScore.add(comment)
            }
            matches
          ]
          .build()
        .out('commentAux', SN.comment, []).build()
        .build()
    ))
  }
  // HELPER
  def getBestThree() {
    threeBestCandidates.getBestThree(candidatesWithNilScore)
  }
}
```

Computation of Strongly Connected Components in Task 2:

```
class FriendComponentUtil {
  // to know which users liked the comment
  Set<User> userSet
  // maps a user to its component
  Map<User,Set<User>> userToComponent = newHashMap
  // set of resulting components
  @Accessors
  Set<Set<User>> components = newHashSet
  // computes connected components
  new(List<User> userList) {
    // to know which users liked the comment
    userSet = userList.toSet
    for (u : userList) {
      var comp = userToComponent.get(u)
      if (comp === null) { // not visited
        comp = newHashSet(u)
        components.add(comp)
        userToComponent.put(u, comp)
        u.computeComponent(comp)
      }
    }
  }
  // explores friends recursively until all the relevant ones are visited
  def private void computeComponent(User u, Set<User> comp) {
    for (f: u.friends) {
      if (userSet.contains(f)) {
        val isAdded = comp.add(f)
        if (isAdded) { // not visited
          userToComponent.put(f, comp)
          f.computeComponent(comp)
        }
      }
    }
  }
}
```

# B Configuration of Batch and Incremental Variants

## B.1 Variant Incremental

### B.1.1 Class SolutionQ1: Benchmark Integration

```
class SolutionQ1 extends Solution {

  new() {
    xform = new Q1_yamtl
    xform.stageUpperBound = 1
    xform.extentTypeModifier = ExtentTypeModifier.LIST
    xform.selectedExecutionPhases = ExecutionPhase.MATCH_ONLY
    xform.fromRoots = false
    xform.executionMode = ExecutionMode.INCREMENTAL
  }

  override String Initial() {
    xform.execute()
    (xform as Q1_yamtl).controversialPosts.map[key.id].join('|')
  }

  override String Update(String deltaName) {
    xform.propagateDelta('sn', deltaName)
    (xform as Q1_yamtl).controversialPosts.map[key.id].join('|')
  }
}
```

### B.1.2 Class SolutionQ2: Benchmark Integration

```
class SolutionQ2 extends Solution {

  new() {
    xform = new Q2_yamtl
    xform.stageUpperBound = 1
    xform.extentTypeModifier = ExtentTypeModifier.LIST
    xform.selectedExecutionPhases = ExecutionPhase.MATCH_ONLY
    xform.fromRoots = false
    xform.executionMode = ExecutionMode.INCREMENTAL
  }

  override String Initial() {
    xform.execute()
    (xform as Q2_yamtl).influentialComments.map[key.id].join('|')
  }

  override String Update(String deltaName) {
    xform.propagateDelta('sn', deltaName)
    (xform as Q2_yamtl).influentialComments.map[key.id].join('|')
  }

}
```

## B.2 Variant Batch

### B.2.1 Class SolutionQ1: Benchmark Integration

```
class SolutionQ1 extends Solution {

  new() {
    xform = new Q1_yamtl
    xform.stageUpperBound = 1
    xform.extentTypeModifier = ExtentTypeModifier.LIST
    xform.selectedExecutionPhases = ExecutionPhase.MATCH_ONLY
    xform.fromRoots = false
  }

  override String Initial() {
    xform.execute()
    (xform as Q1_yamtl).controversialPosts.map[key.id].join('|')
  }

  override String Update(String deltaName) {
    (xform as Q1_yamtl).controversialPosts.clear()
    (xform as Q1_yamtl).postWithoutScoreList.clear()
    xform.reset()
    // apply source delta only
    xform.applyDelta('sn', deltaName)
    // transform updated source model
    xform.execute()
    (xform as Q1_yamtl).controversialPosts.map[key.id].join('|')
  }
}
```

### B.2.2 Class SolutionQ2: Benchmark Integration

```
class SolutionQ2 extends Solution {

  new() {
    xform = new Q2_yamtl
    xform.stageUpperBound = 1
    xform.extentTypeModifier = ExtentTypeModifier.LIST
    xform.selectedExecutionPhases = ExecutionPhase.MATCH_ONLY
    xform.fromRoots = false
  }

  override String Initial() {
    xform.execute()
    (xform as Q2_yamtl).influentialComments.map[key.id].join('|')
  }

  override String Update(String deltaName) {
    (xform as Q2_yamtl).influentialComments.clear()
    (xform as Q2_yamtl).commentWithoutScoreList.clear()
    xform.reset()
    // apply source delta only
    xform.applyDelta('sn', deltaName)
    // transform updated source model
    xform.execute()
    (xform as Q2_yamtl).influentialComments.map[key.id].join('|')
  }
}
```

# C Experimental Results

## C.1 Task 1 Results

### C.1.1 Time

Model sizes are plotted along the X axis (in logarithmic scale) and time is plotted along the Y axis (in logarithmic scale).

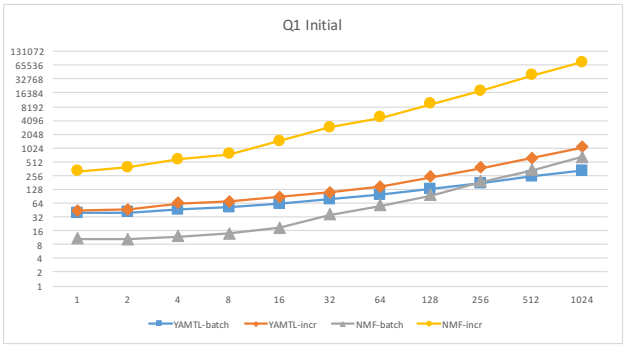


Figure 1: Initial transformation time (ms, Y axis) per model size (X axis)

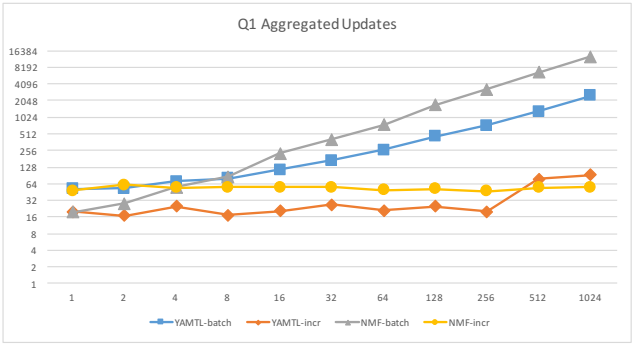


Figure 2: Aggregated update time (ms, Y axis) per model size (X axis)

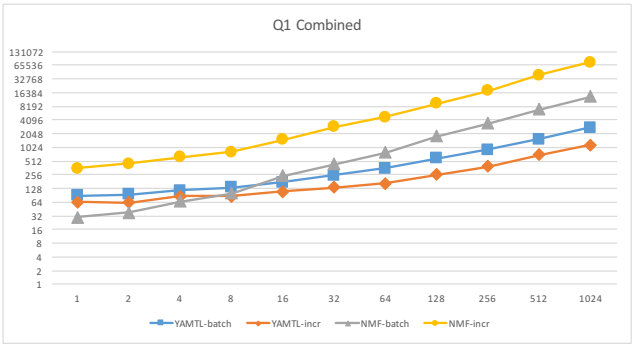


Figure 3: Combined time (ms, Y axis) per model size (X axis)

### C.1.2 Memory

Model sizes are plotted along the X axis (in logarithmic scale) and memory usage is plotted along the Y axis (in logarithmic scale).

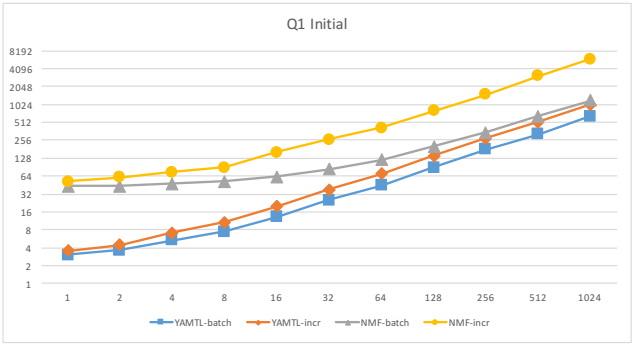


Figure 4: Initial transformation memory (MB, Y axis) per model size (X axis)

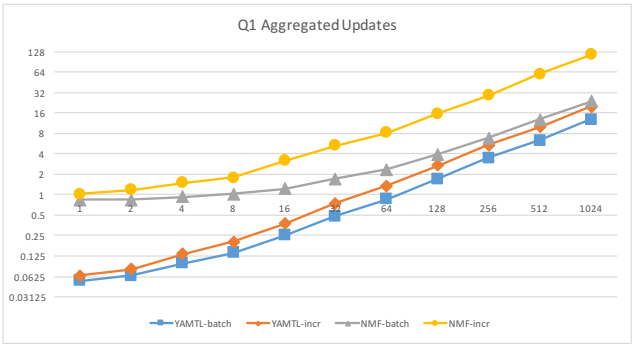


Figure 5: Aggregated update memory (GB, Y axis) per model size (X axis)

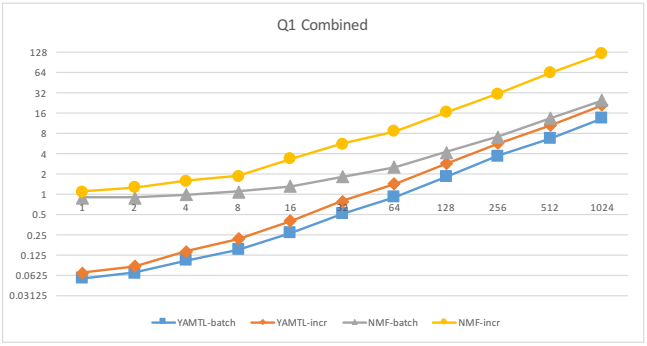


Figure 6: Combined memory (GB, Y axis) per model size (X axis)

## C.2 Task 2 Results

### C.2.1 Time

Model sizes are plotted along the X axis (in logarithmic scale) and time is plotted along the Y axis (in logarithmic scale).

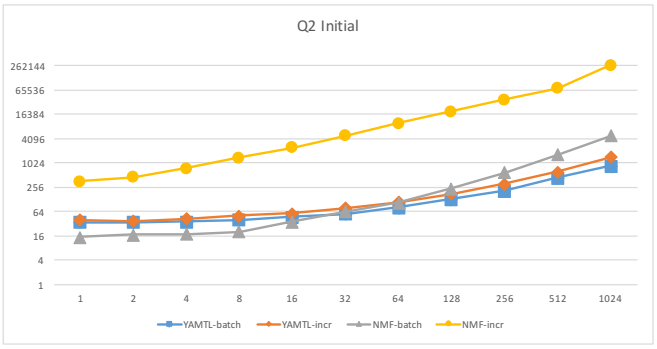


Figure 7: Initial transformation time (ms, Y axis) per model size (X axis)

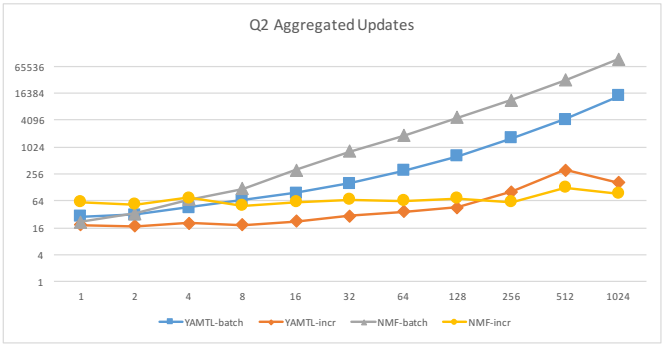


Figure 8: Aggregated update time (ms, Y axis) per model size (X axis)

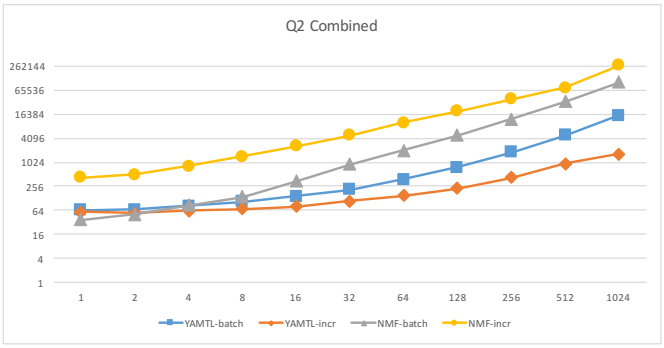


Figure 9: Combined time (ms, Y axis) per model size (X axis)

### C.2.2 Memory

Model sizes are plotted along the X axis (in logarithmic scale) and memory usage is plotted along the Y axis (in logarithmic scale).

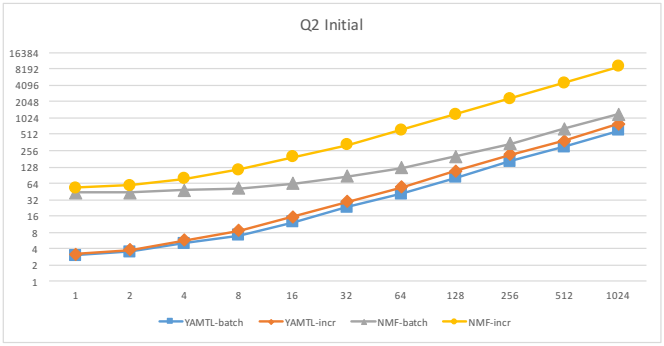


Figure 10: Initial transformation memory (MB, Y axis) per model size (X axis)

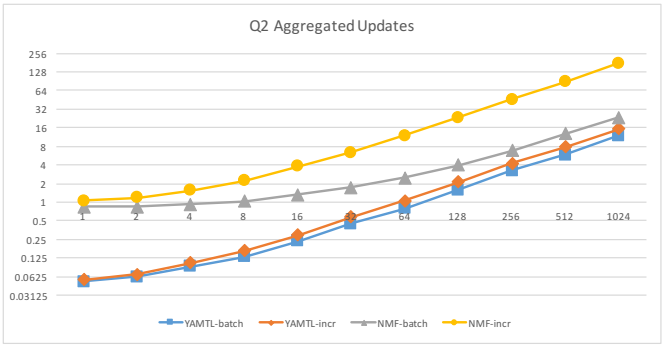


Figure 11: Aggregated update memory (GB, Y axis) per model size (X axis)

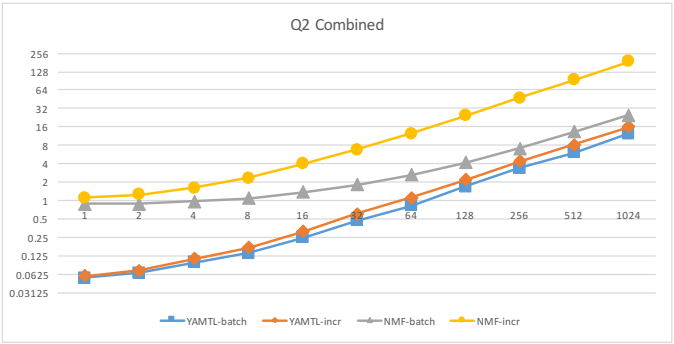


Figure 12: Combined memory (GB, Y axis) per model size (X axis)