Efficiently Evaluating Complex Boolean Expressions

Yahoo! Research
Marcus Fontoura, Suhas Sadanadan, Jayavel Shanmugasundaram, Sergei Vassilvitski, Erik Vee, Srihari Venkatesan and Jason Zien

Agenda

- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Agenda

- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Simple example

- Display advertising
  - Ads: Boolean expressions (contracts)

```
age IN {young}
age IN {old} AND income IN {high, veryHigh}
income IN {high} AND browser NOT_IN {ie}
```

  - Publishers: assignments

```
age = old; income = high; browser = firefox
```

Simple example

- Display advertising
  - Ads: Boolean expressions (contracts)

    ~~`age IN {young}`~~

    `age IN {old} AND income IN {high, veryHigh}`

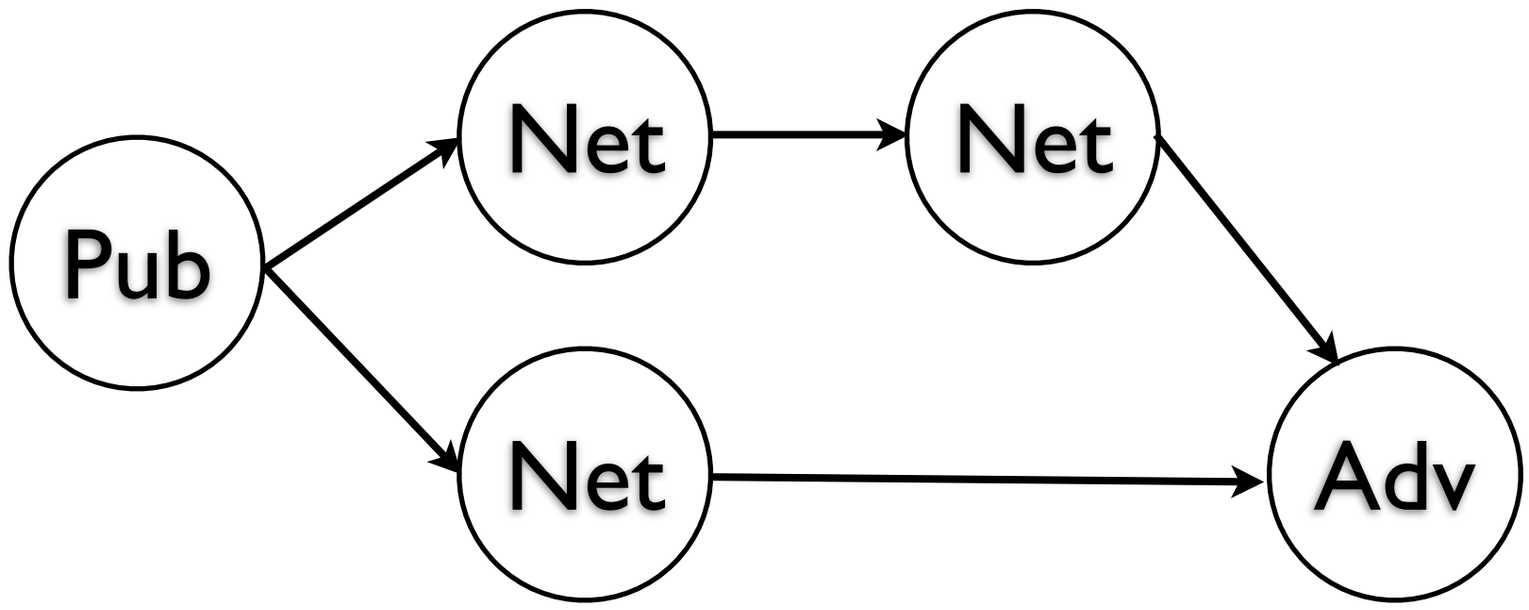
    `income IN {high} AND browser NOT_IN {ie}`

  - Publishers: assignments

    `age = old; income = high; browser = firefox`

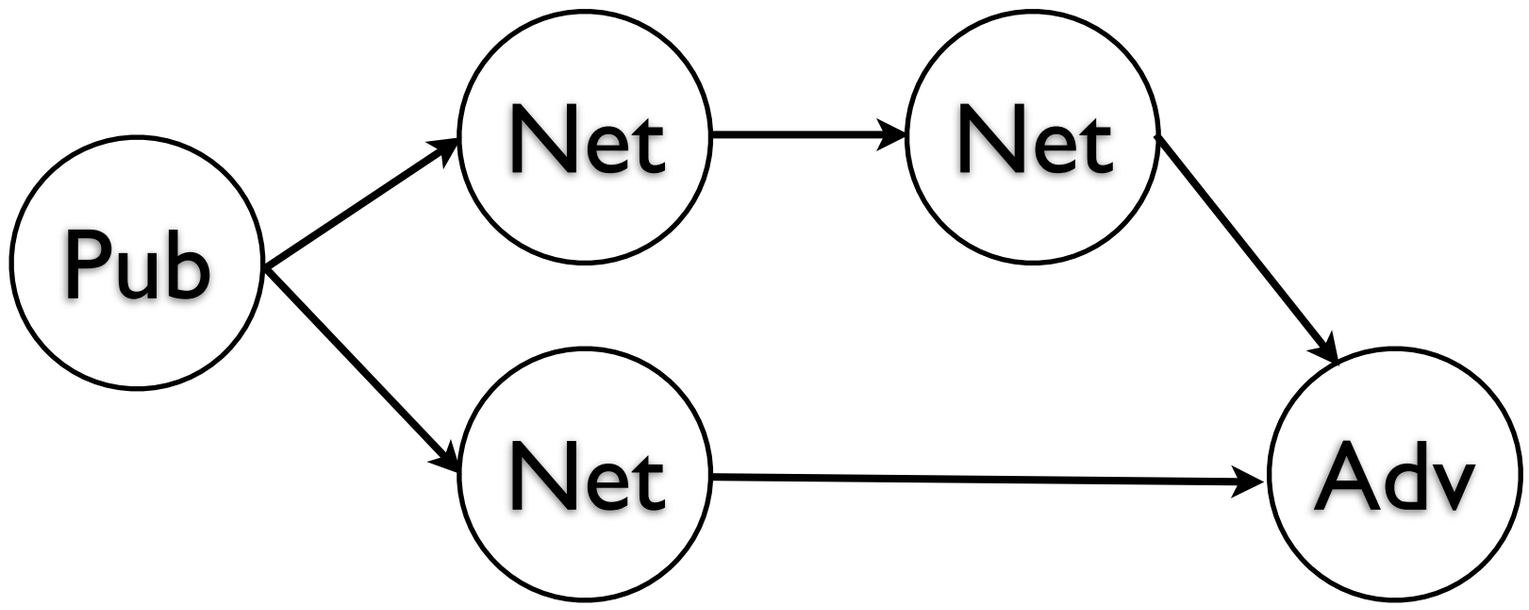
More complex example

- Display advertising exchange



More complex example

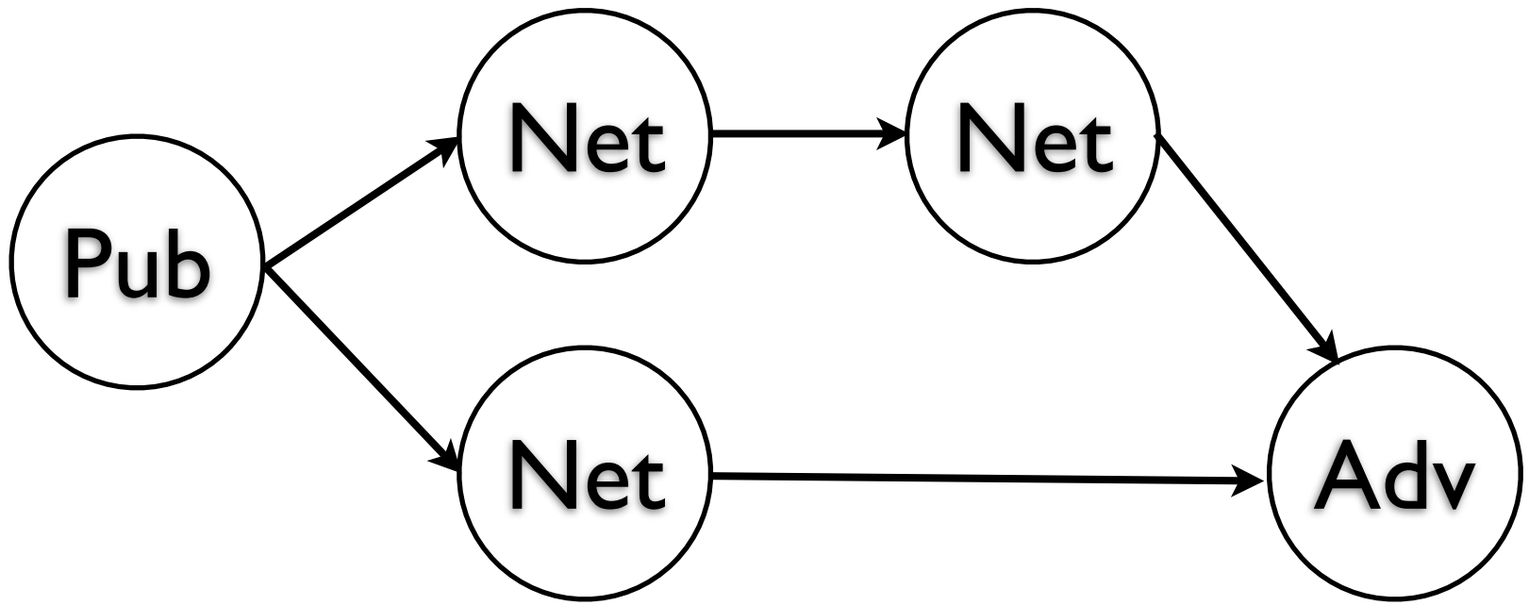
- Boolean expressions model the type of inventory sold by each node



More complex example

- Each Boolean expression can be a DNF/CNF

- Contracts for the publisher are "complex" expressions



Other examples

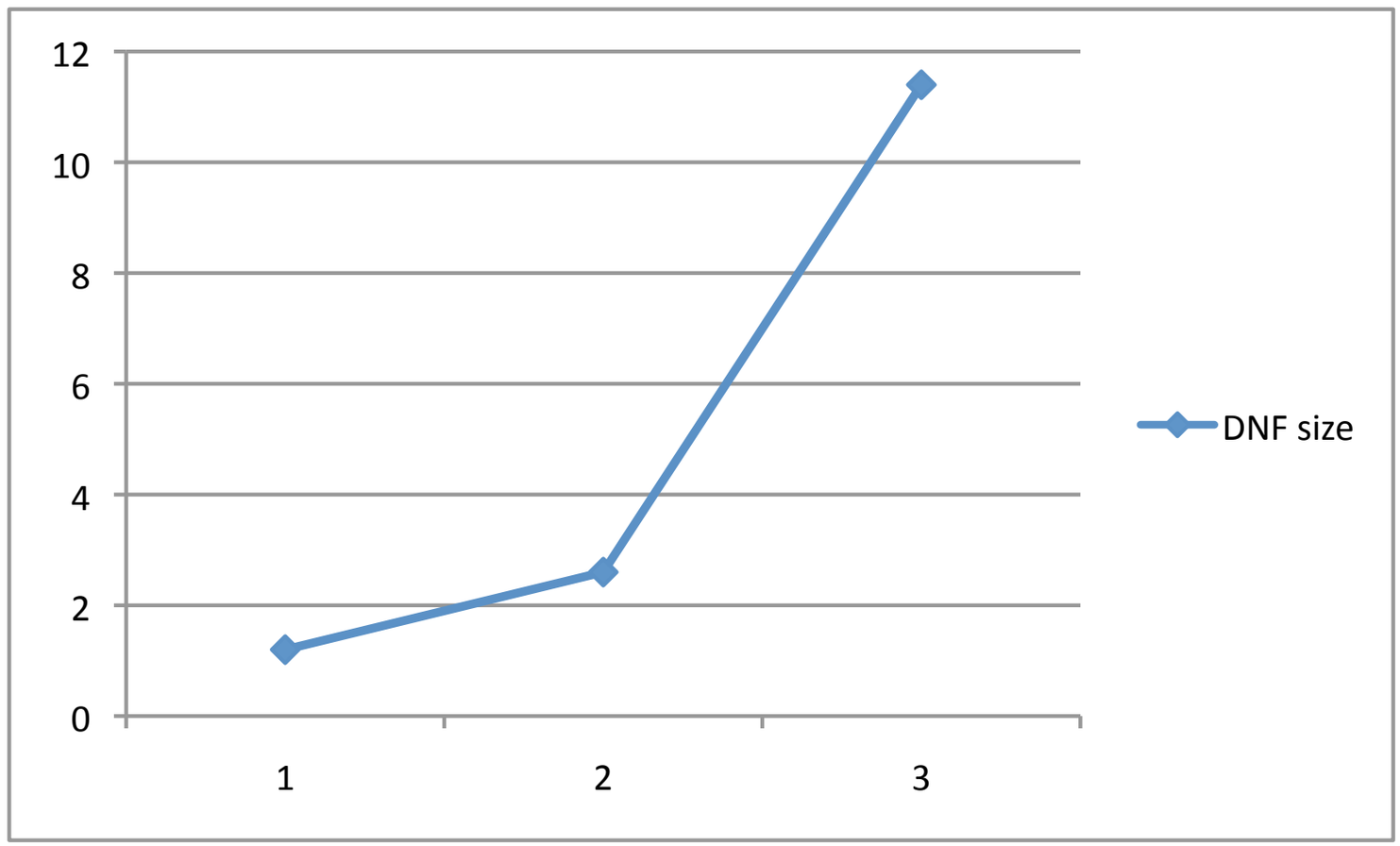
- Automatic targeting in display advertising
  - e.g. machine generated expressions to maximize click-through
- Information dissemination in social network graphs

State-of-the-art

- There are existing solutions for efficiently evaluating CNF and DNF expressions
  - Content-based publish-subscribe systems
- Normalizing complex Boolean expressions into DNF incurs in an exponential blow-up in size

DNF growth

- In KB, averaged over 20 DNFs of each size
- Data set is realistic



Normalization does not work

- In environments where performance requirements are strict
- Billions of queries per day

Problem definition

- Evaluate complex Boolean expressions (e.g. AND of DNFs)
  - Modeled as a tree of AND/OR nodes, where leafs are conjunctions of IN and NOT_IN operators
- Given an assignment, retrieve all valid expressions

Agenda

- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

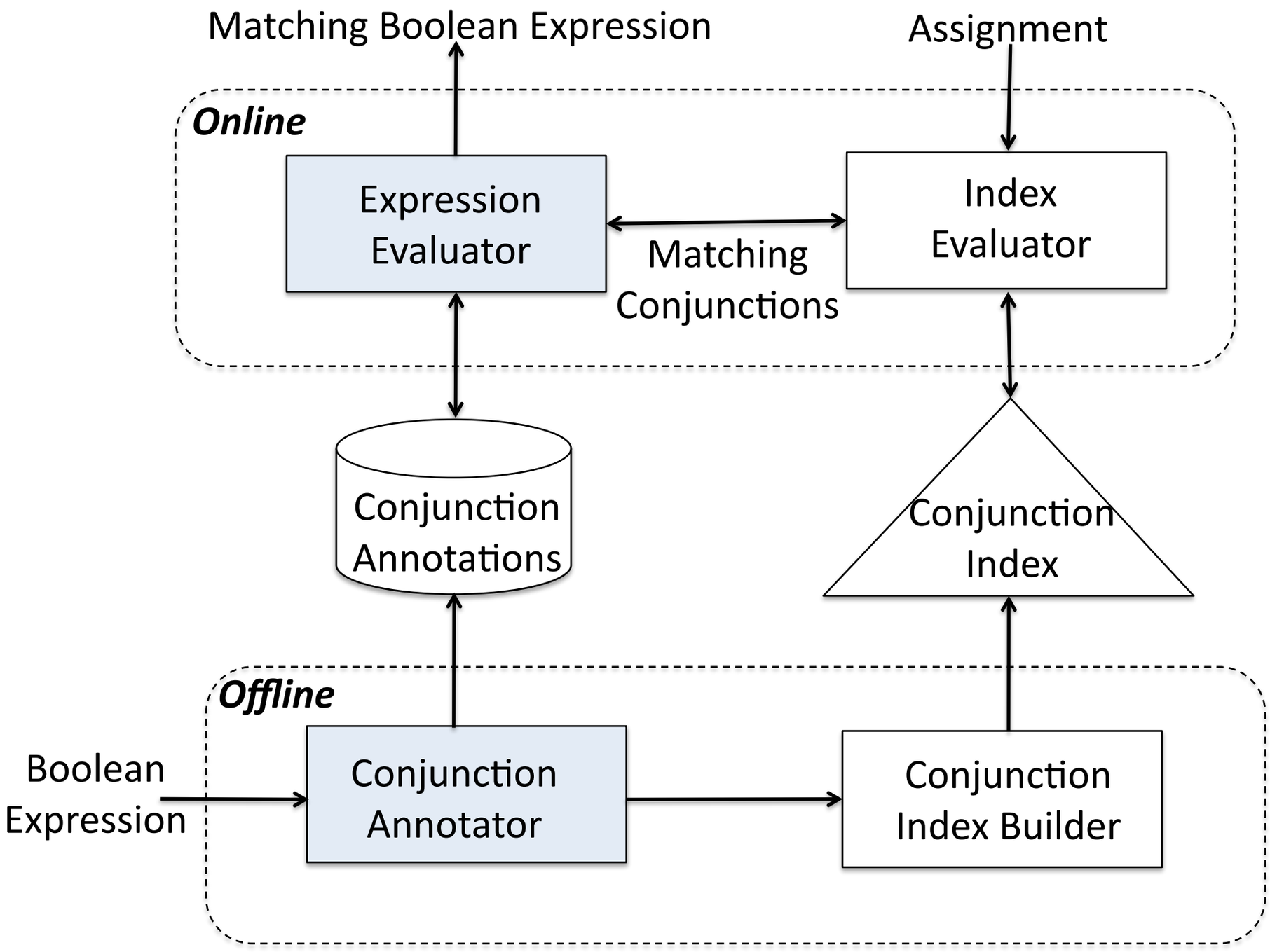
Agenda

- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Intuition

- (Offline) Annotate the conjunctions with their position on the complex Boolean expression tree
- Evaluate conjunctions (leafs) using a state-of-the-art algorithm
- Evaluate the trees bottom-up, using the retrieved conjunctions and their positions

Overall architecture



Background: conjunction evaluation

- Use an inverted index as the basic data structure
- Similar to processing conjunction queries in IR (but documents are queries)

Conjunction evaluation example

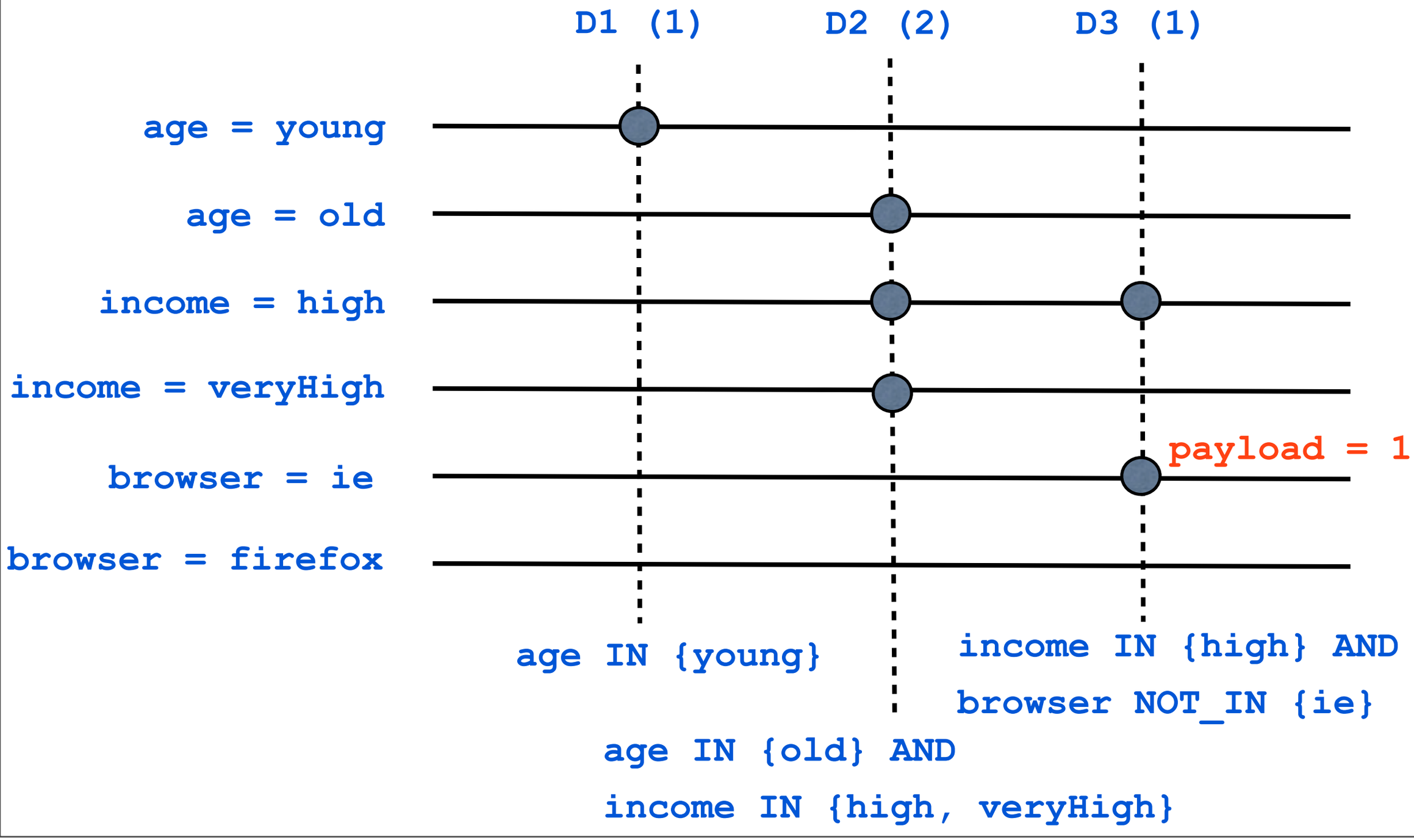
- Conjunctions (documents)

```
age IN {young}
age IN {old} AND income IN {high, veryHigh}
income IN {high} AND browser NOT_IN {ie}
```

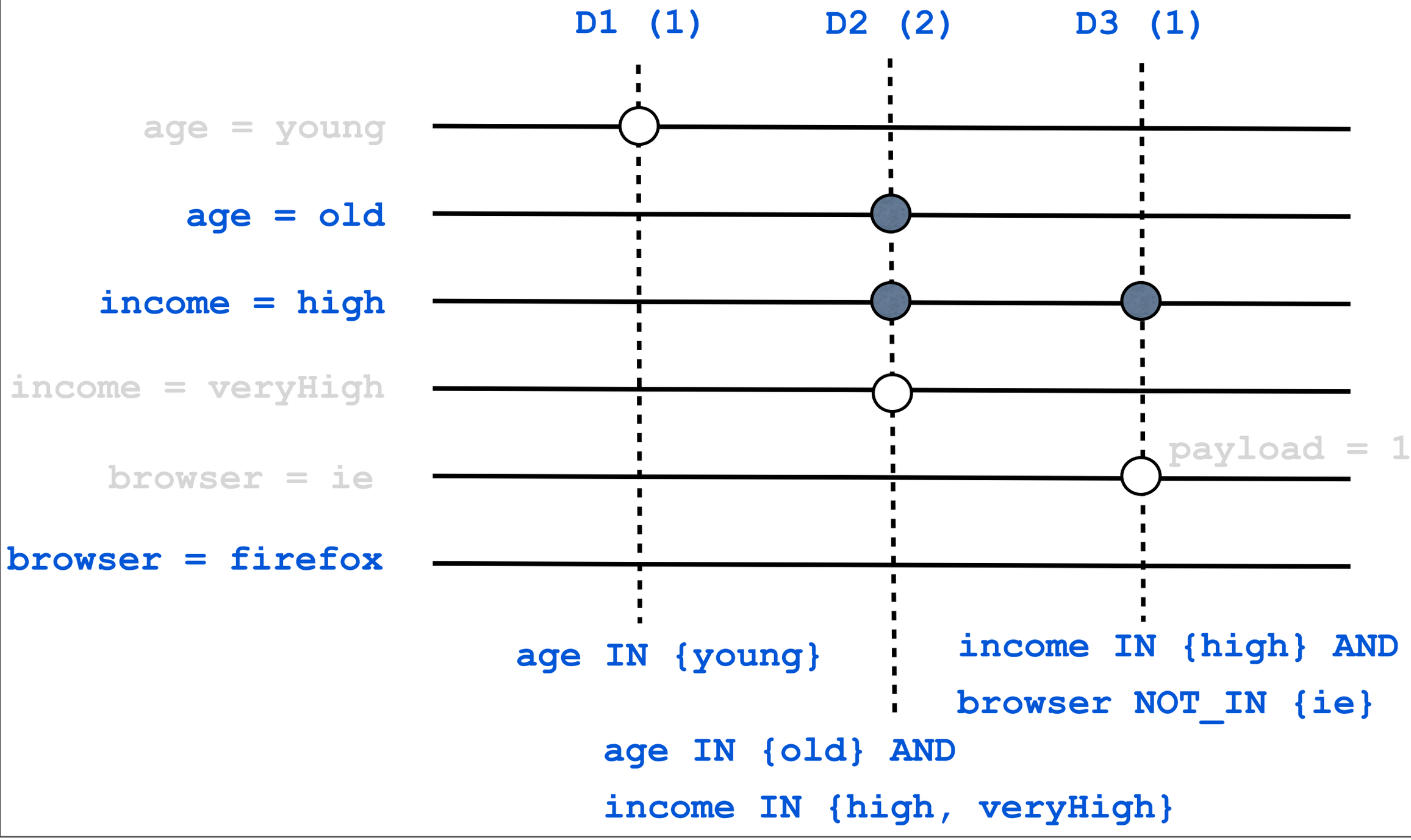
- Assignments (queries)

```
age = old; income = high; browser = firefox
```

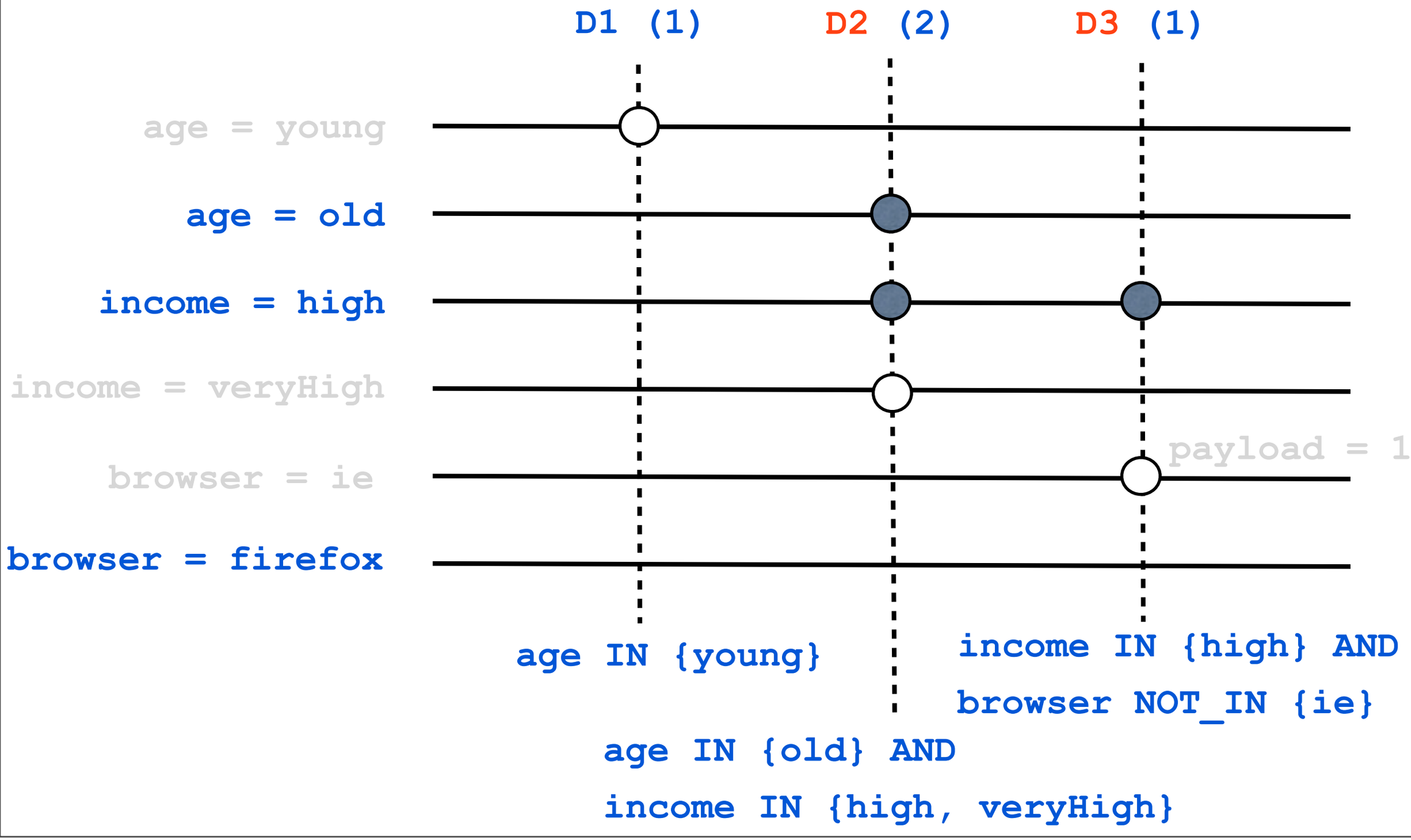
Conjunction evaluation example



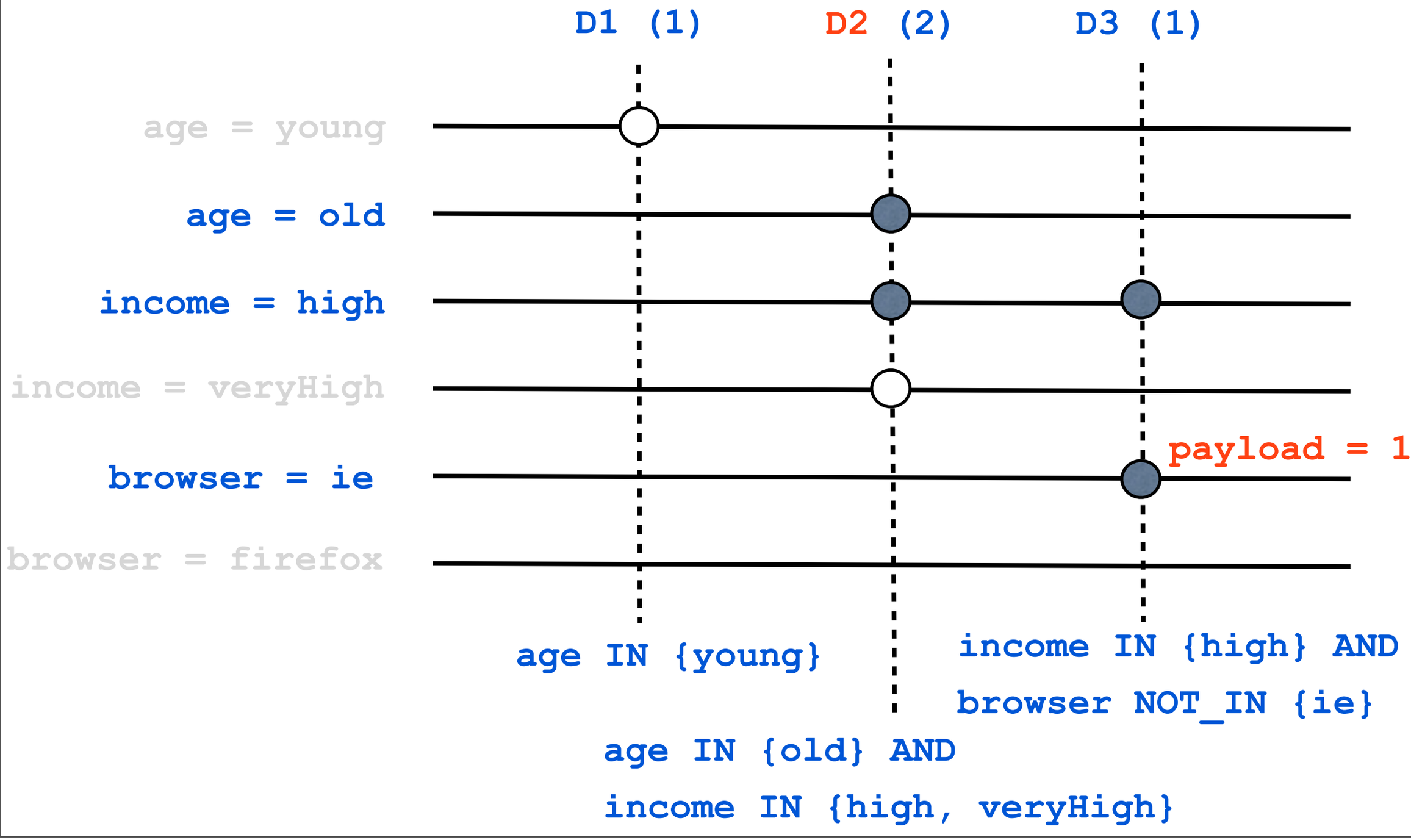
Conjunction evaluation example



Conjunction evaluation example



Conjunction evaluation another example



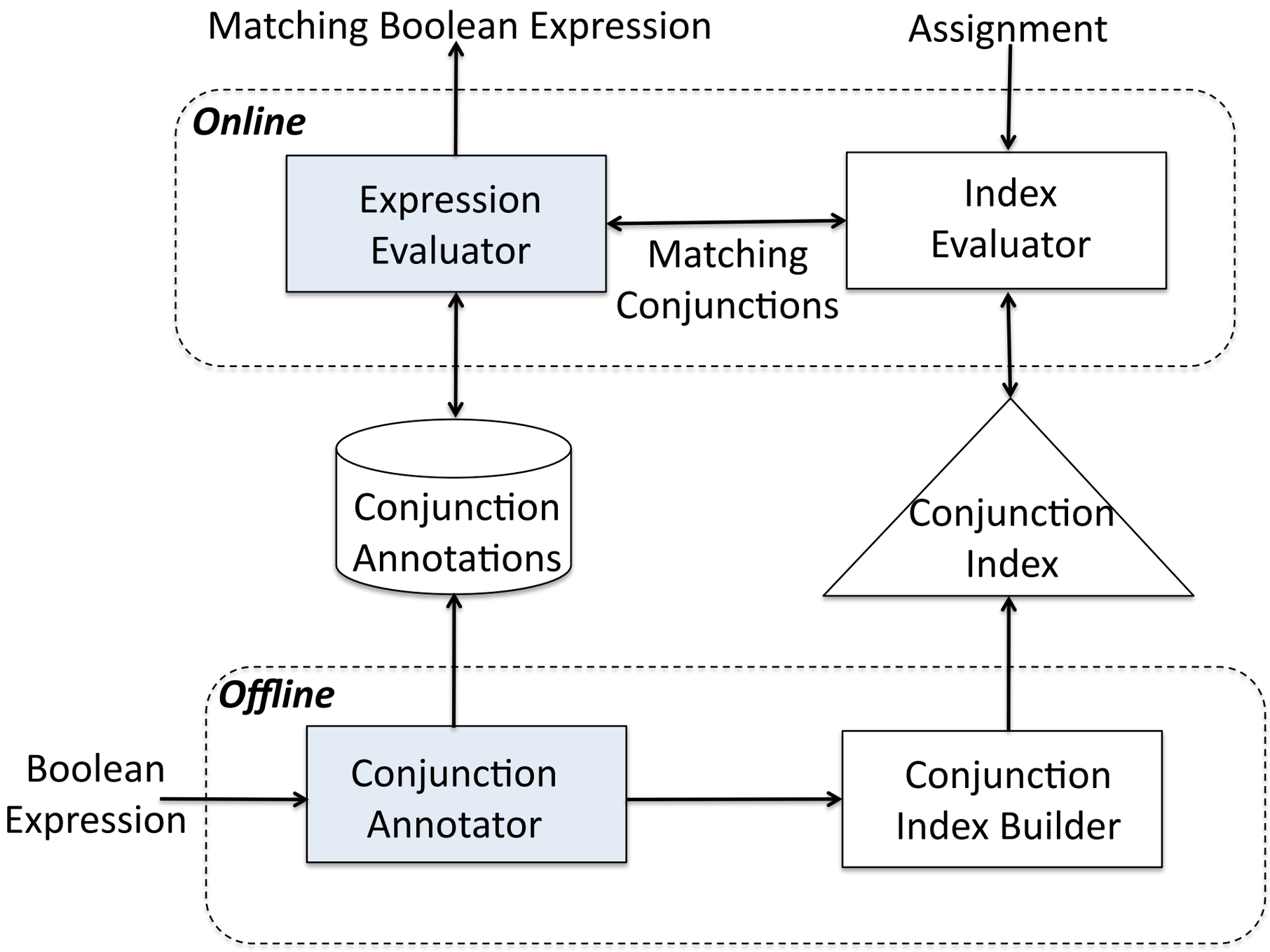
Agenda

- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Agenda

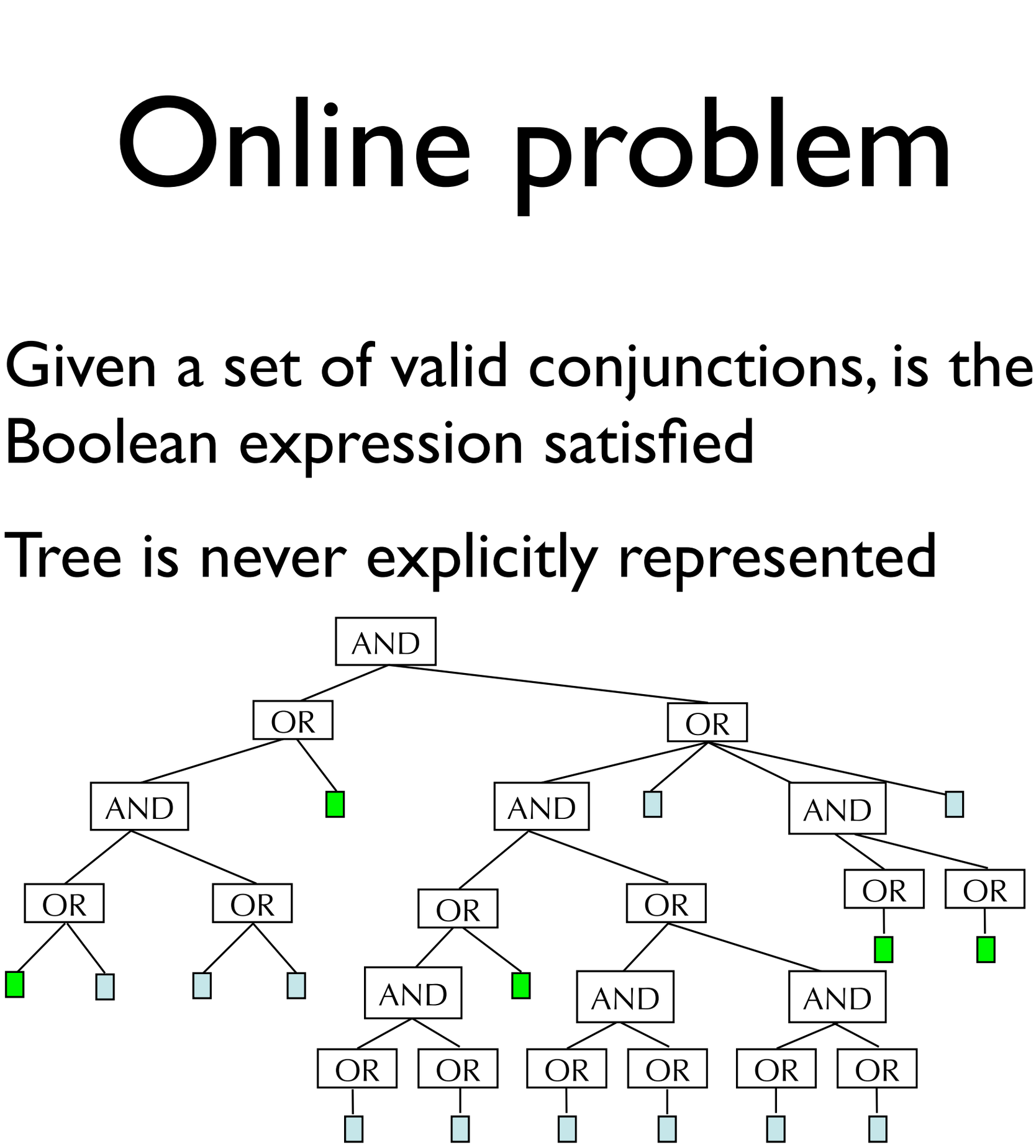
- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Overall architecture



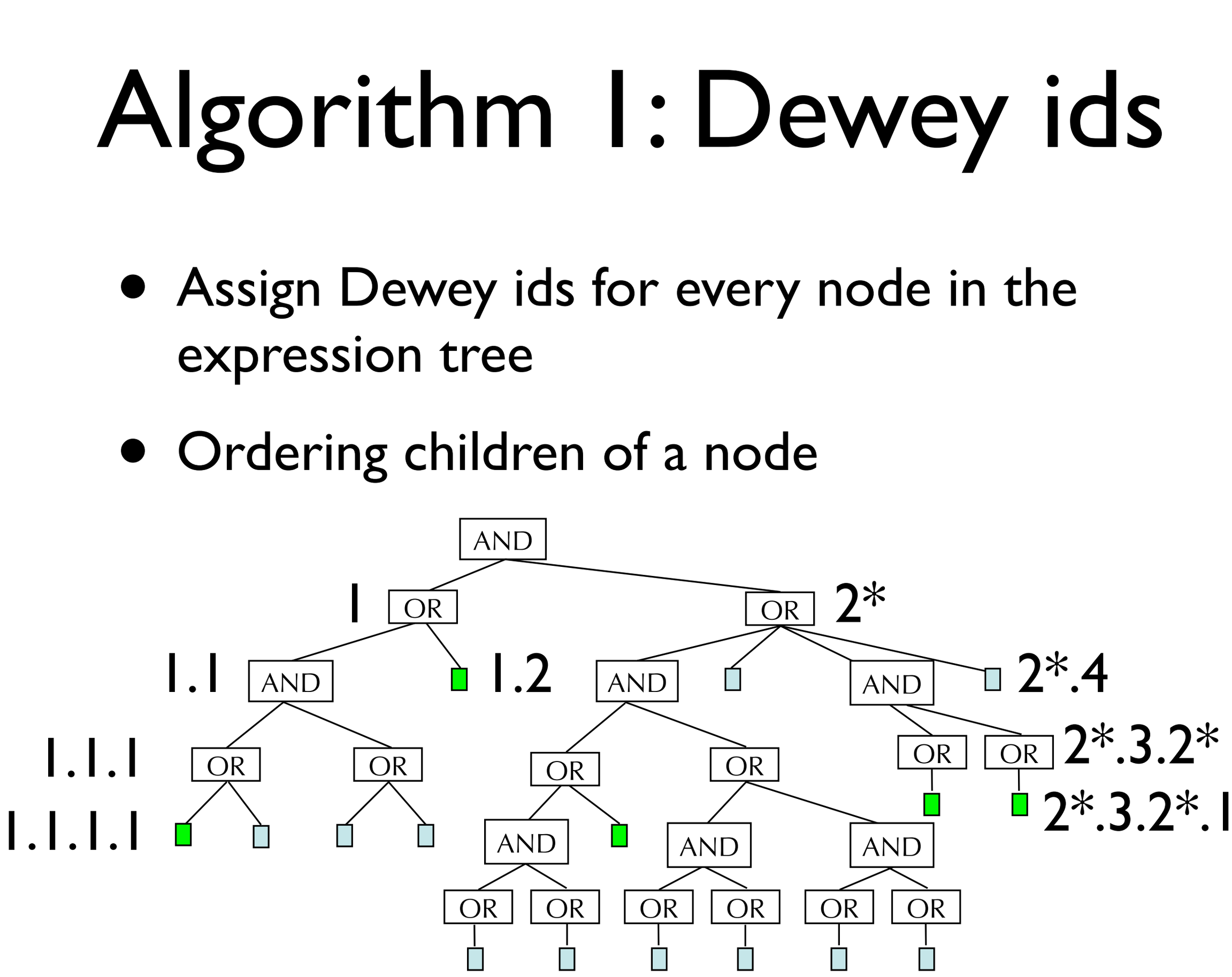
Online problem

- Given a set of valid conjunctions, is the Boolean expression satisfied
- Tree is never explicitly represented



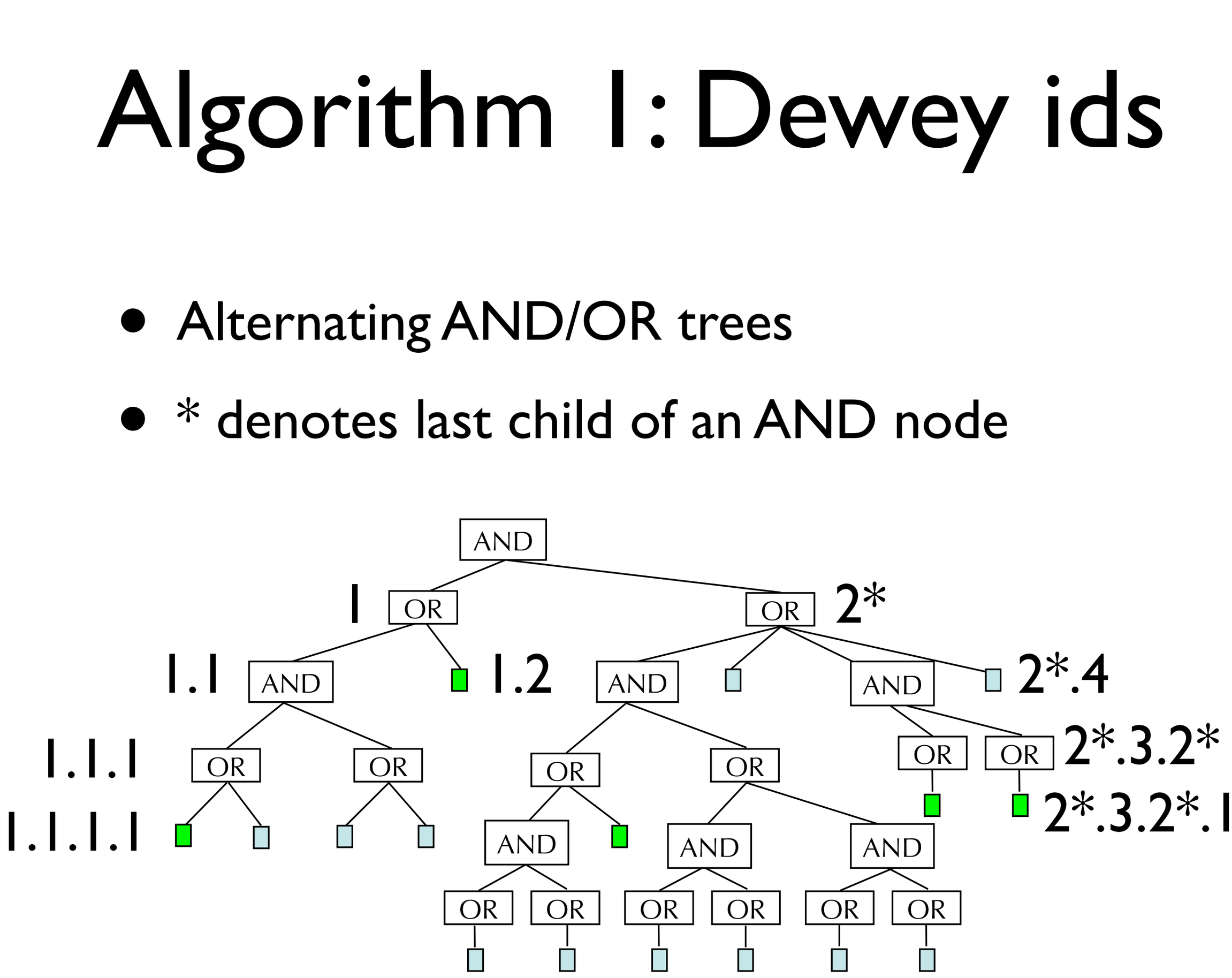
Algorithm 1: Dewey ids

- Assign Dewey ids for every node in the expression tree
- Ordering children of a node



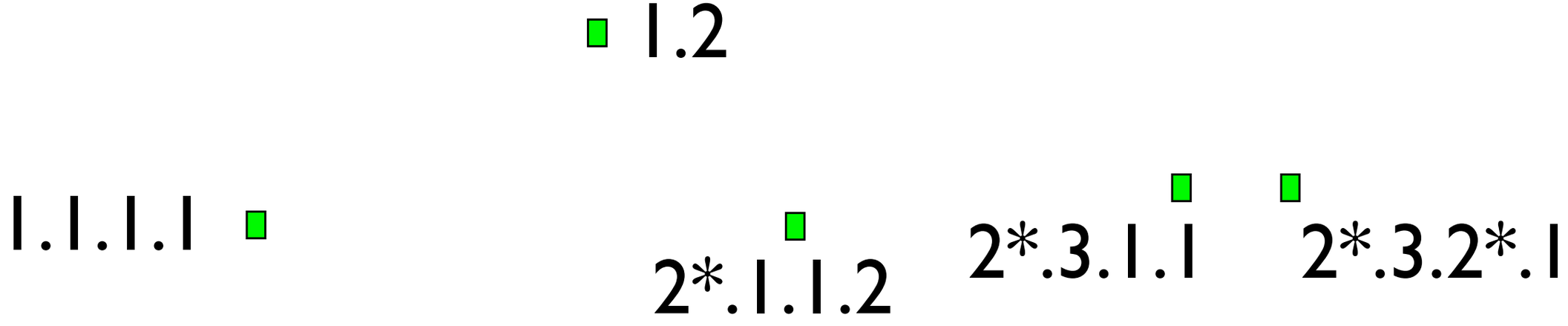
Algorithm 1: Dewey ids

- Alternating AND/OR trees
- * denotes last child of an AND node



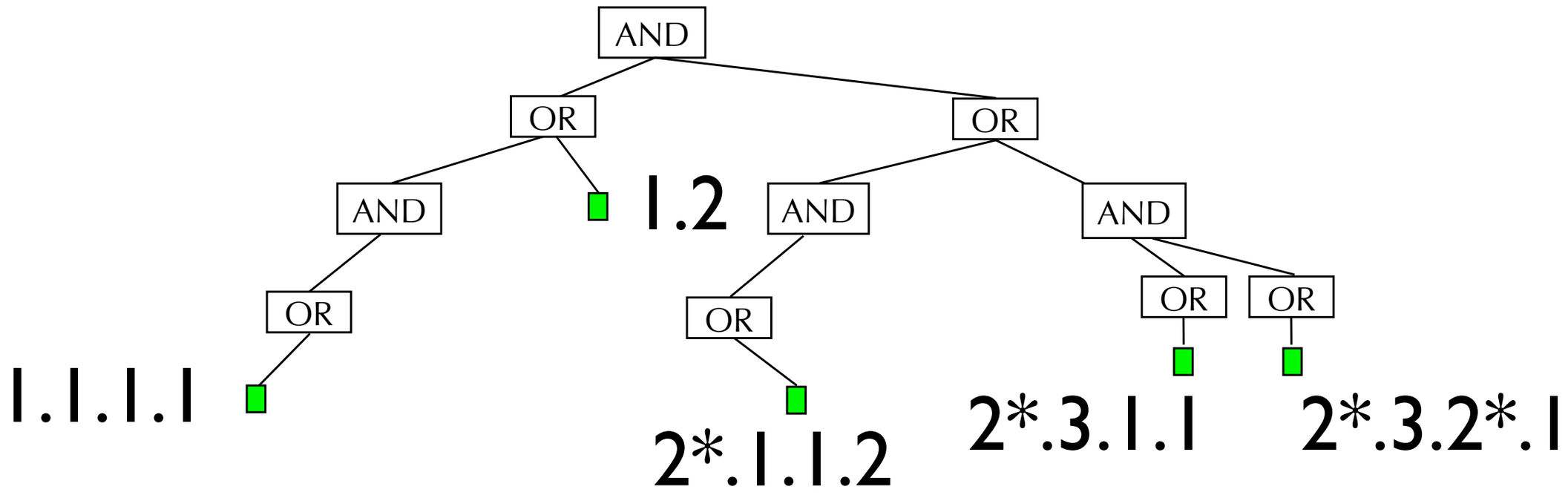
Algorithm 1: Dewey ids

- Index evaluator will return the leaf nodes, which are the matching conjunctions



Algorithm 1: Dewey ids

- Index evaluator will return the leaf nodes, which are the matching conjunctions

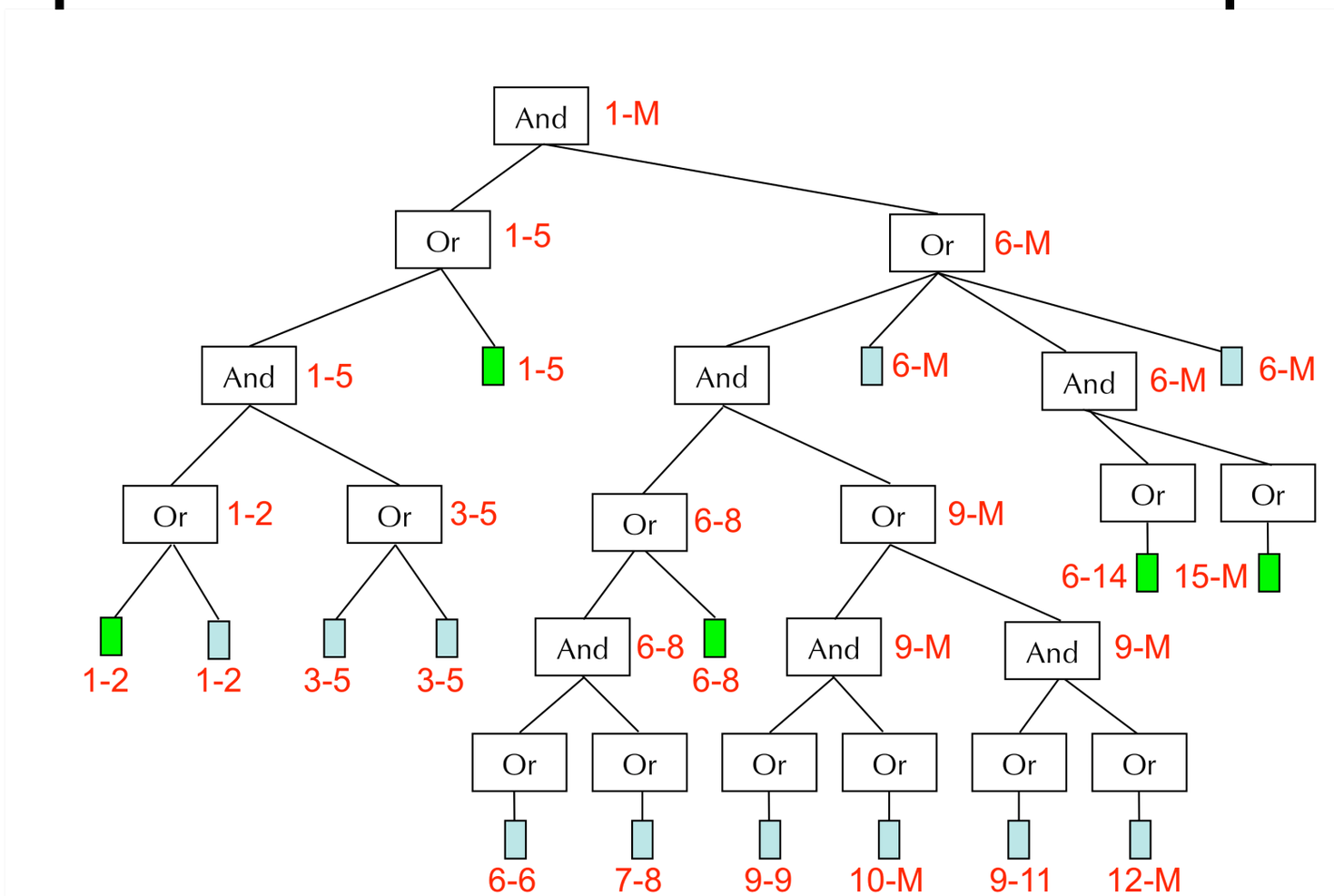


Algorithm 2: Interval ids

- We map each Boolean tree to a one dimensional interval [1,M]
- M is the maximum number of leaves
- Tree is satisfied if there is a subset of intervals that cover all integer points on [1,M] without overlap

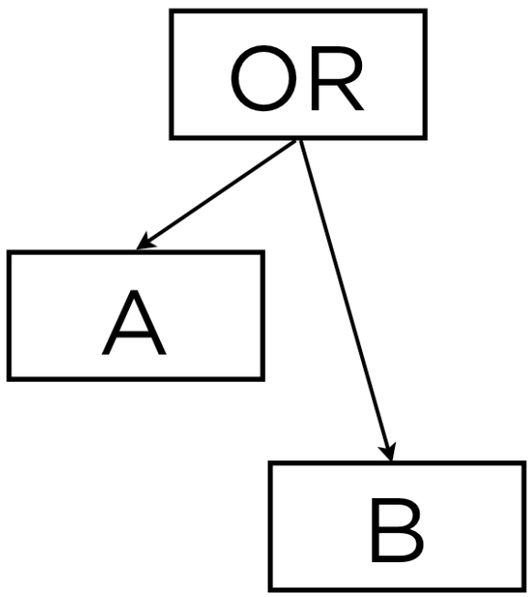
Algorithm 2: Interval ids

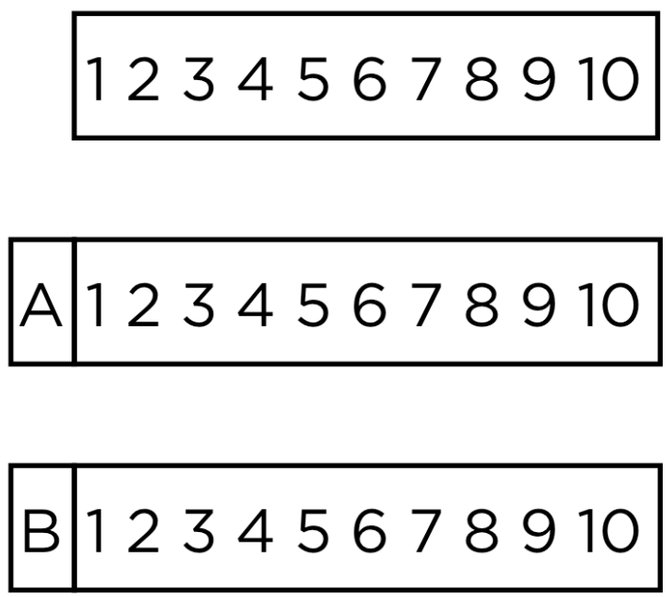
- Look at: [1-5] [6-14] [15-M] : all integer points covered without overlap



Assigning intervals

- Recursive procedure
- Children of an OR inherit the parent interval



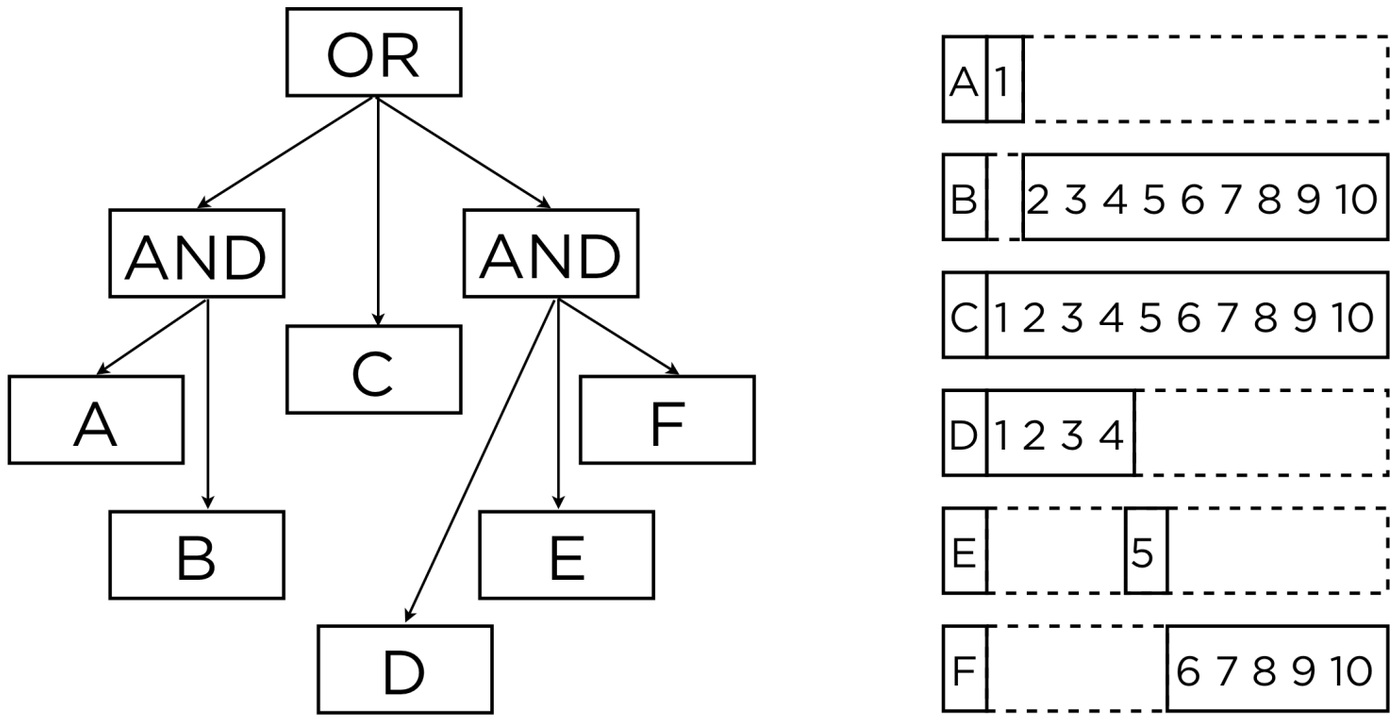


Assigning intervals

- Recursive procedure
- Children of an AND partition the interval

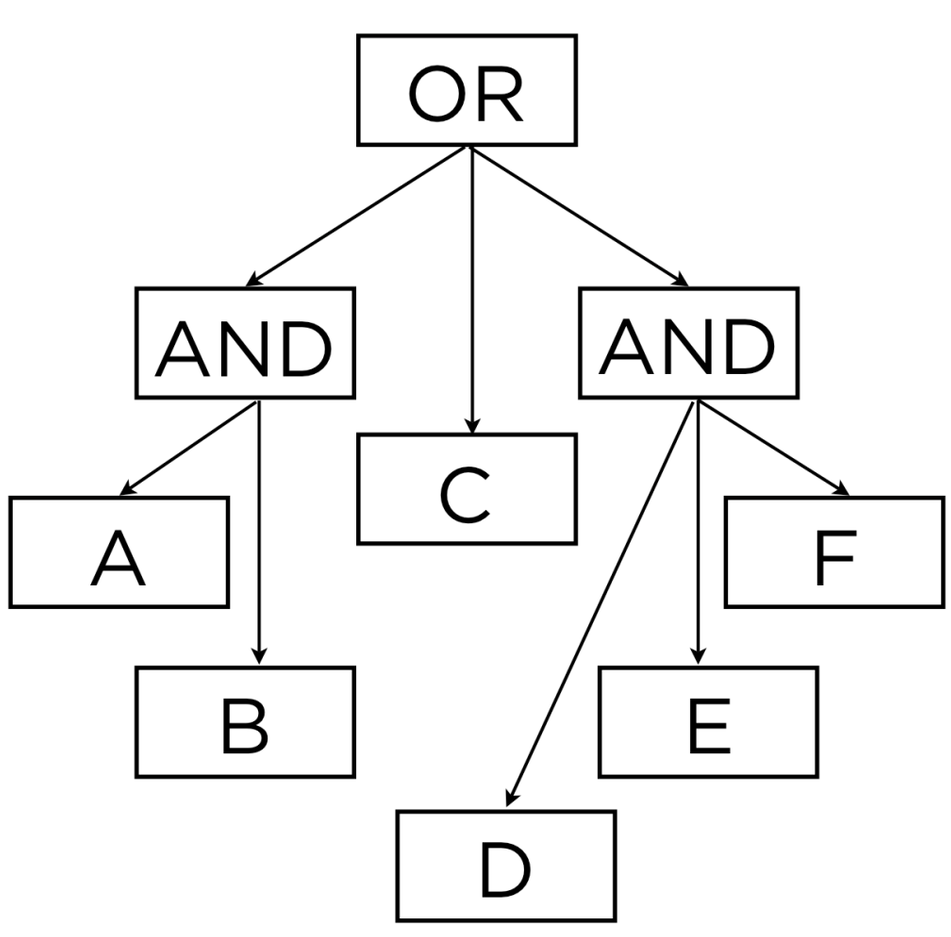
Slightly more complex example

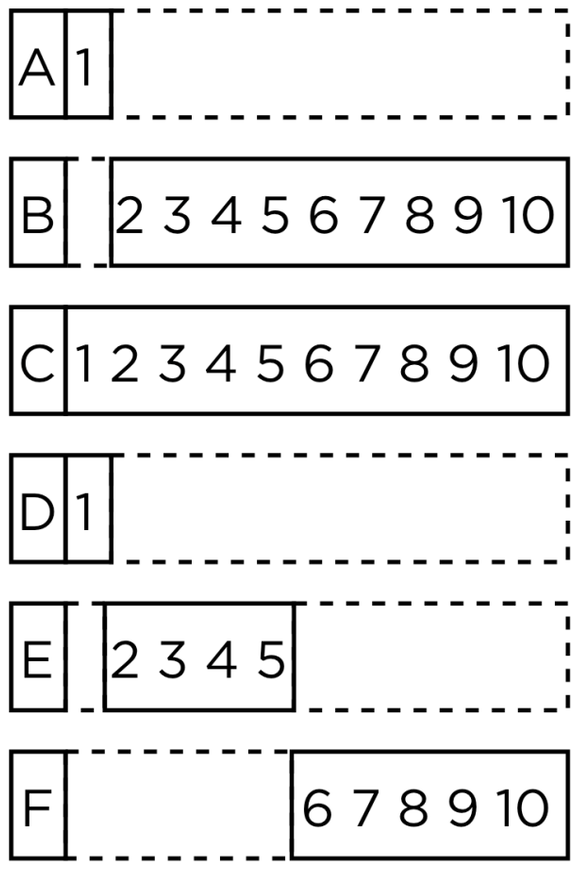
- B & D are not enough to satisfy, since intervals overlap
- D & E & F are OK, since intervals don't overlap



Caveats in labeling

- Bad labeling: A, E & F (or D & B) would lead us to satisfied contracts





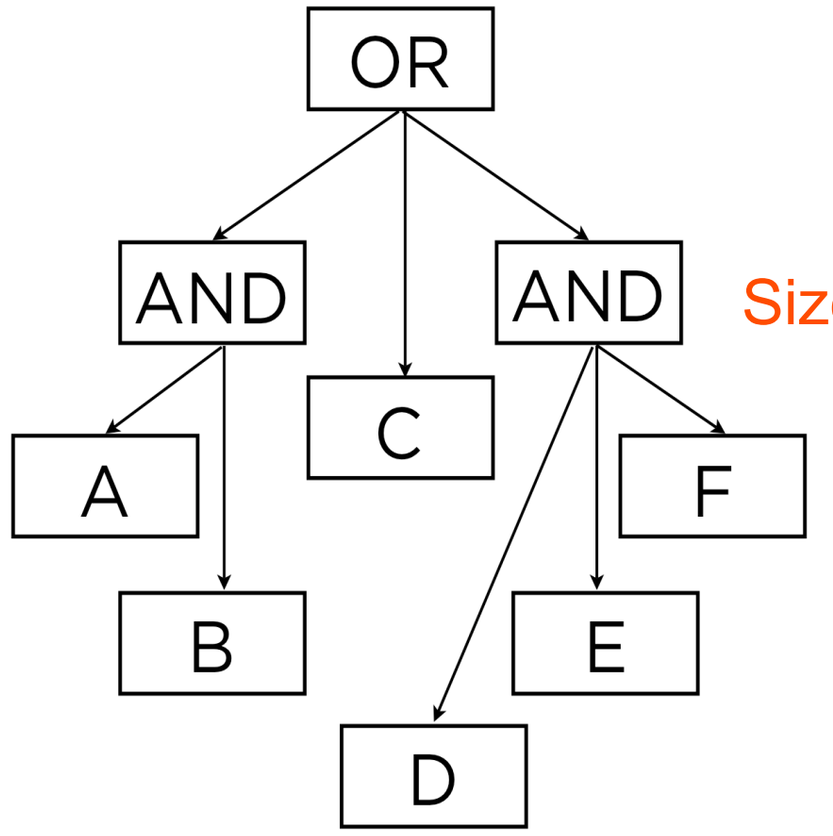
Assignment algorithm

- Recursive procedure
- Children of an OR inherit the parent interval
- Children of an AND partition the interval
- Extra technical condition
  - No two children of non left-most AND node children share the same starting

Assignment algorithm

- Let each leaf have size 1
- Let the size of an internal node = # of leaves in its subtree
- For each node n, let n.leftLeaves = # of leaves appearing before n in a pre-order traversal of the tree

Assignment algorithm

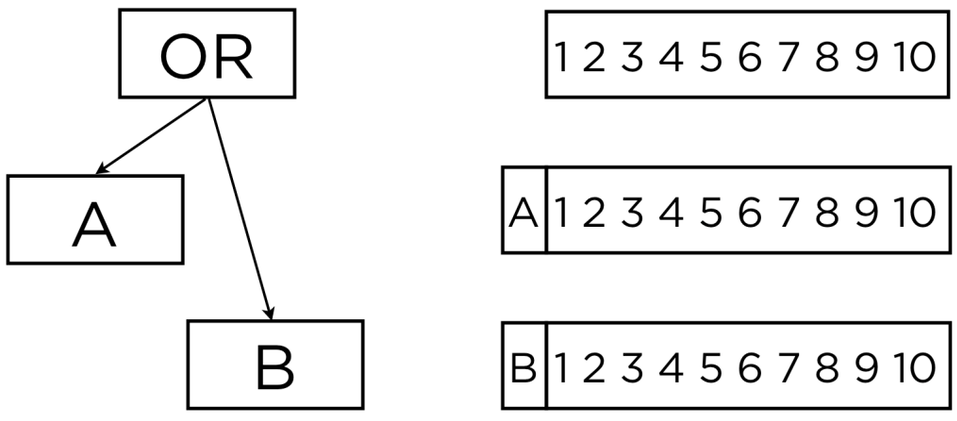


A.size=1, A.leftLeaves = 0

Size = 3, leaftLeaves = 3

Assignment algorithm

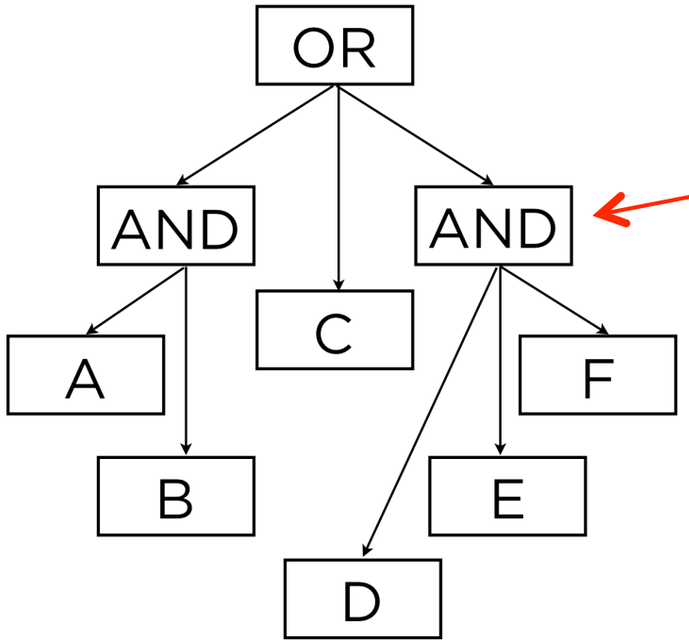
- Label the root node [1,M] : entire interval
- Label children recursively
- For an OR node:
  - Label of child = label of parent

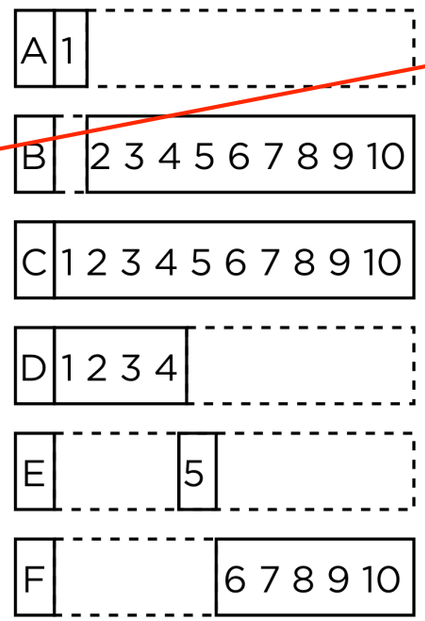


Assignment algorithm

- Parent AND node n with interval: [s, t]
- Label children with partition of parent node
  - First child c begins with s, ends with n.leftLeaves + c.size
  - Next child c': begins with c.end+1, ends with c'.begin + c'.size-1
  - …
  - Last child c'': … ends with t

Assignment algorithm





Size = 3, leaftLeaves = 3

Interval: 1…10

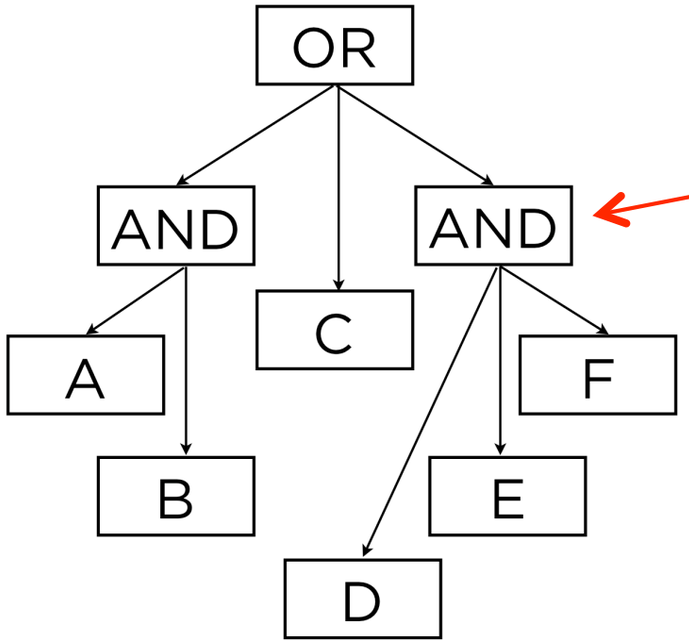
First child: [1, 3+1]

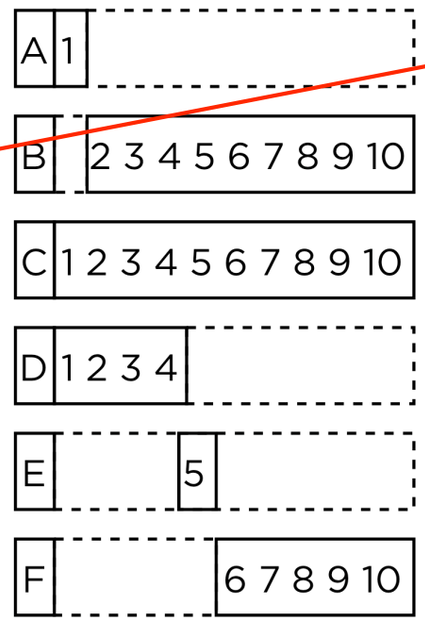
Second child[5, 5]

Last child [6, 10]

Assignment algorithm

- Theorem: this assignment algorithm produces a valid labeling





Size = 3, leaftLeaves = 3

Interval: 1…10

First child: [1, 3+1]

Second child[5, 5]

Last child [6, 10]

Evaluation

- Given a set of matched intervals, how do you find if there exists a set of non overlapping ones covering [1,M] ?
- Maintain a *matched* array
  - matched[i] = true iff there exist non overlapping segments that cover the interval [1,i]
- To evaluate:
  - Sort all segments by their starting position
  - Evaluate matched array
  - Given [s,t], if matched[s-1] = true, then set matched[t] = true

Example

- Suppose intervals returned are
  - [1,1], [1,4], [5,5], [6,10]
- Initialize matched by setting matched[0] = true, rest false
  - matched: 1 0 0 0 0 0 0 0 0 0 0

Example

- Suppose intervals returned are
  - [1,1], [1,4], [5,5], [6,10]
- Process [1,1]. Since matched[0] = true, set matched[1] = true
  - matched: 1 1 0 0 0 0 0 0 0 0 0

Example

- Suppose intervals returned are
  - [1,1], [1,4], [5,5], [6,10]
- Process [1,4]. Since matched[0] = true, set matched[4] = true
  - matched: 1 1 0 0 1 0 0 0 0 0 0

Example

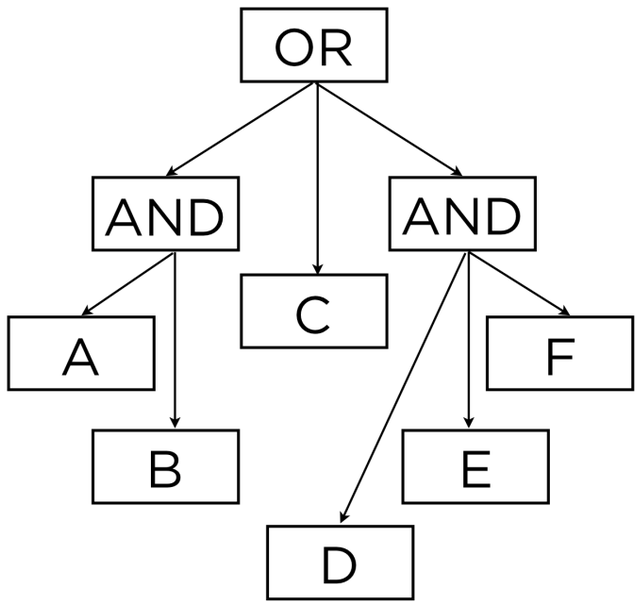
- Suppose intervals returned are
  - [1,1], [1,4], [5,5], [6,10]
- Process [5,5]. Since matched[4] = true, set matched[5] = true
  - matched: 1 1 0 0 1 1 0 0 0 0 0

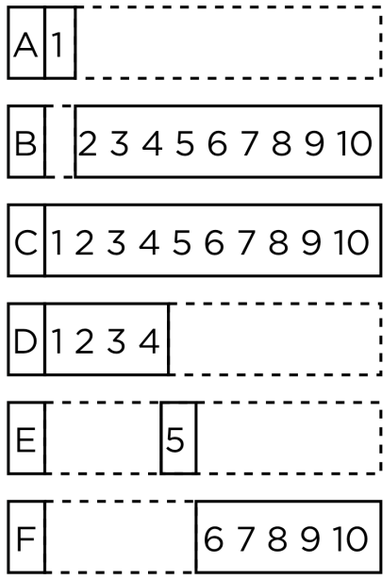
Example

- Suppose intervals returned are
  - [1,1], [1,4], [5,5], [6,10]
- Process [6,10]. Since matched[5] = true, set matched[10] = true.
  - matched: 1 1 0 0 1 1 0 0 0 0 1

Example

- Suppose intervals returned are
  - [1,1], [1,4], [5,5], [6,10]
- Final matched array: 1 1 0 0 1 1 0 0 0 0 1





Agenda

- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Agenda

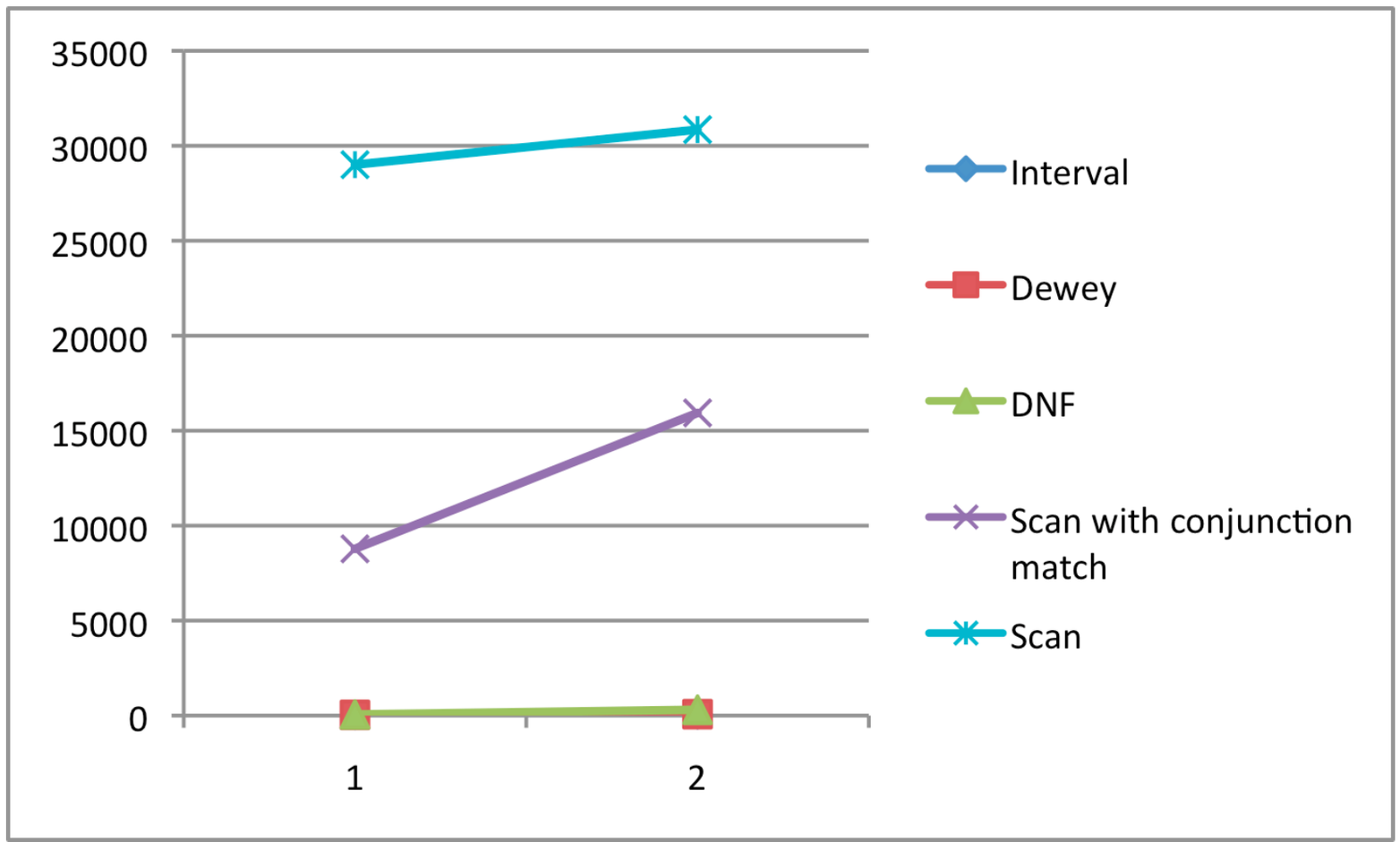
- Motivation and problem definition
- System architecture
- Algorithms
- Experiments

Data

- Generated a synthetic data set of expressions based on real logs
- Depth of the tree between 1 and 4
- Typical number of children of nodes between 1 and 4

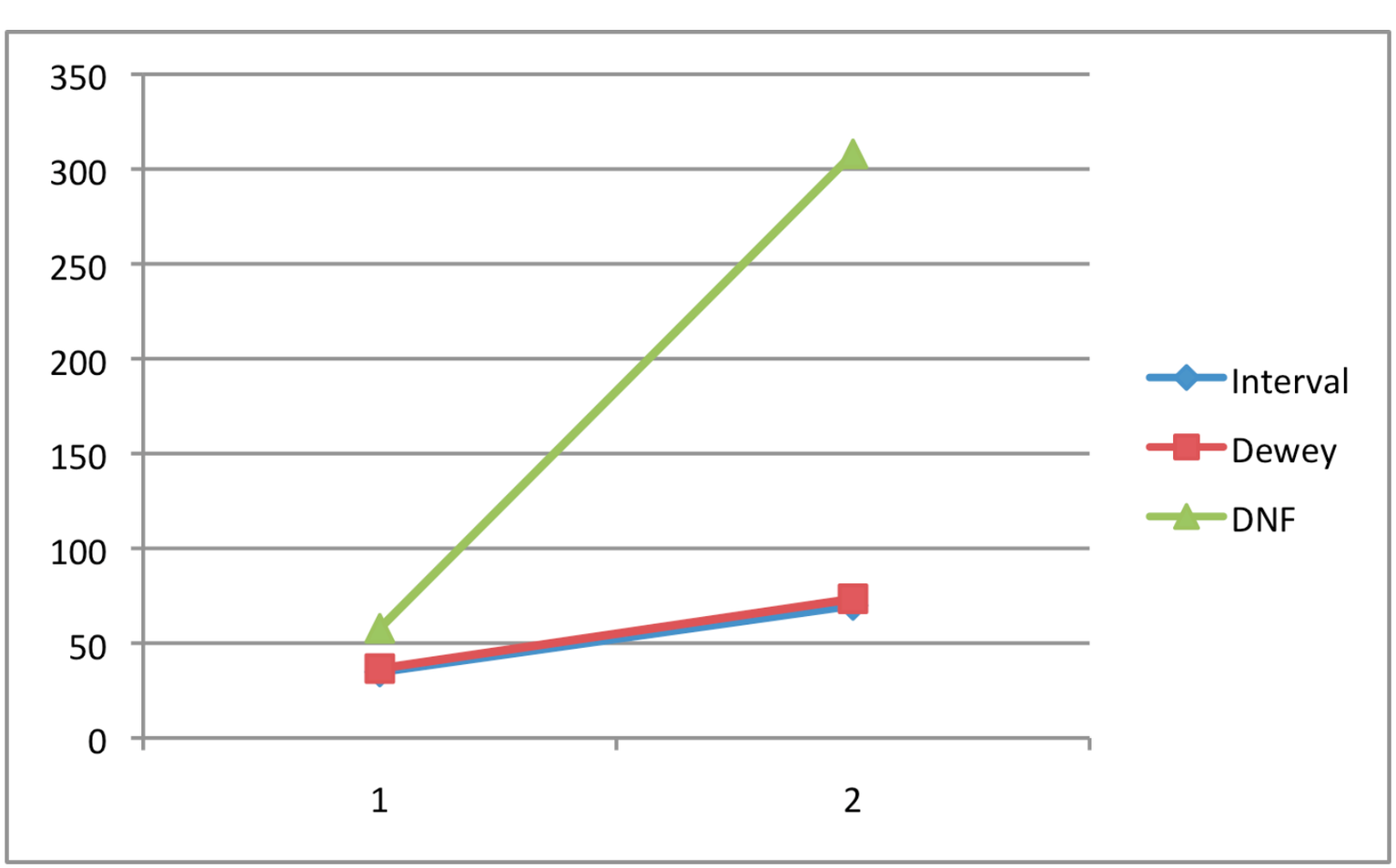
Performance of different methods

- Running time in ms (y axis) vs. tree depth (x axis). Scan does not scale wrt time



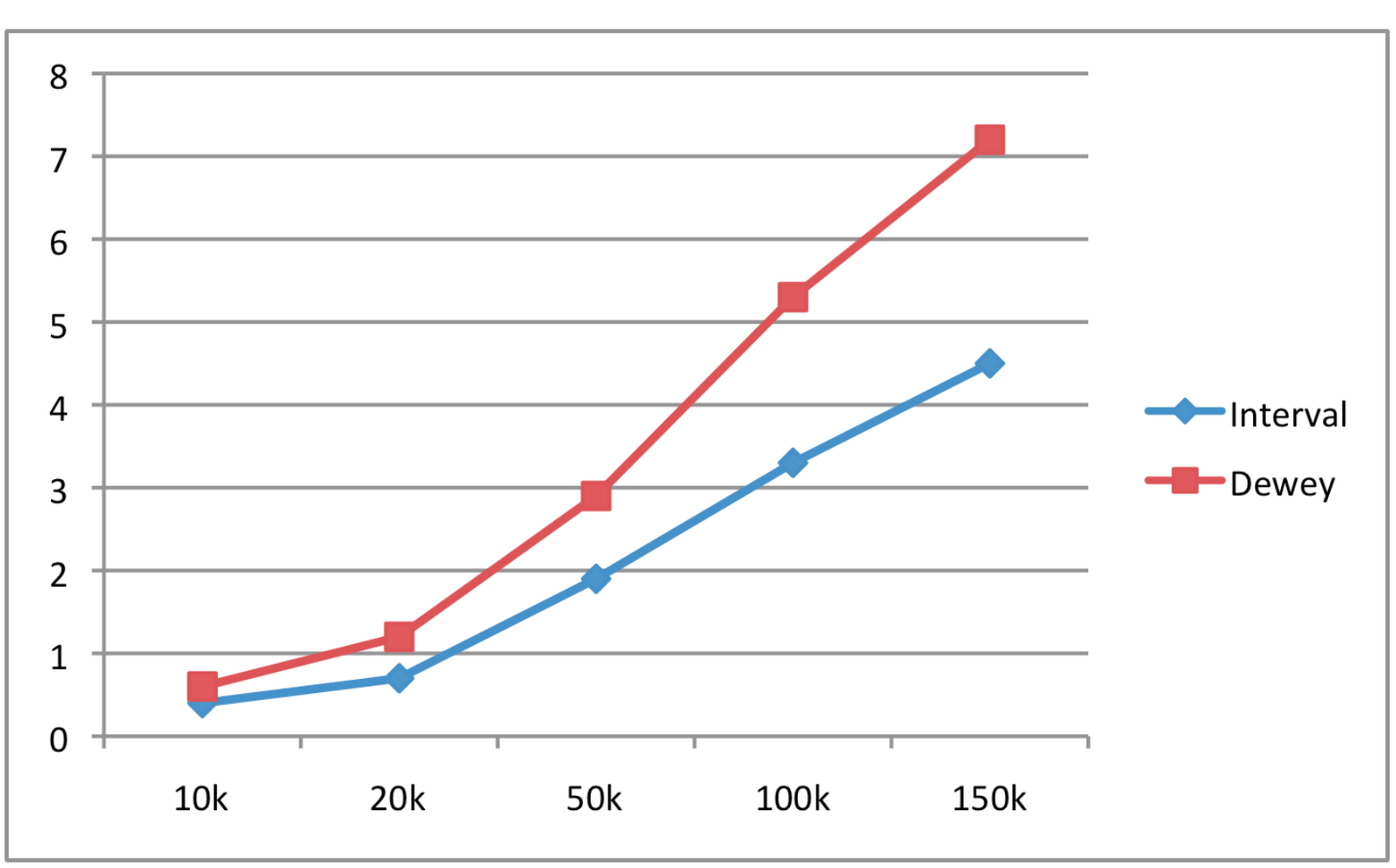
DNF performance

- Running time in ms (y axis) vs. tree depth (x axis)



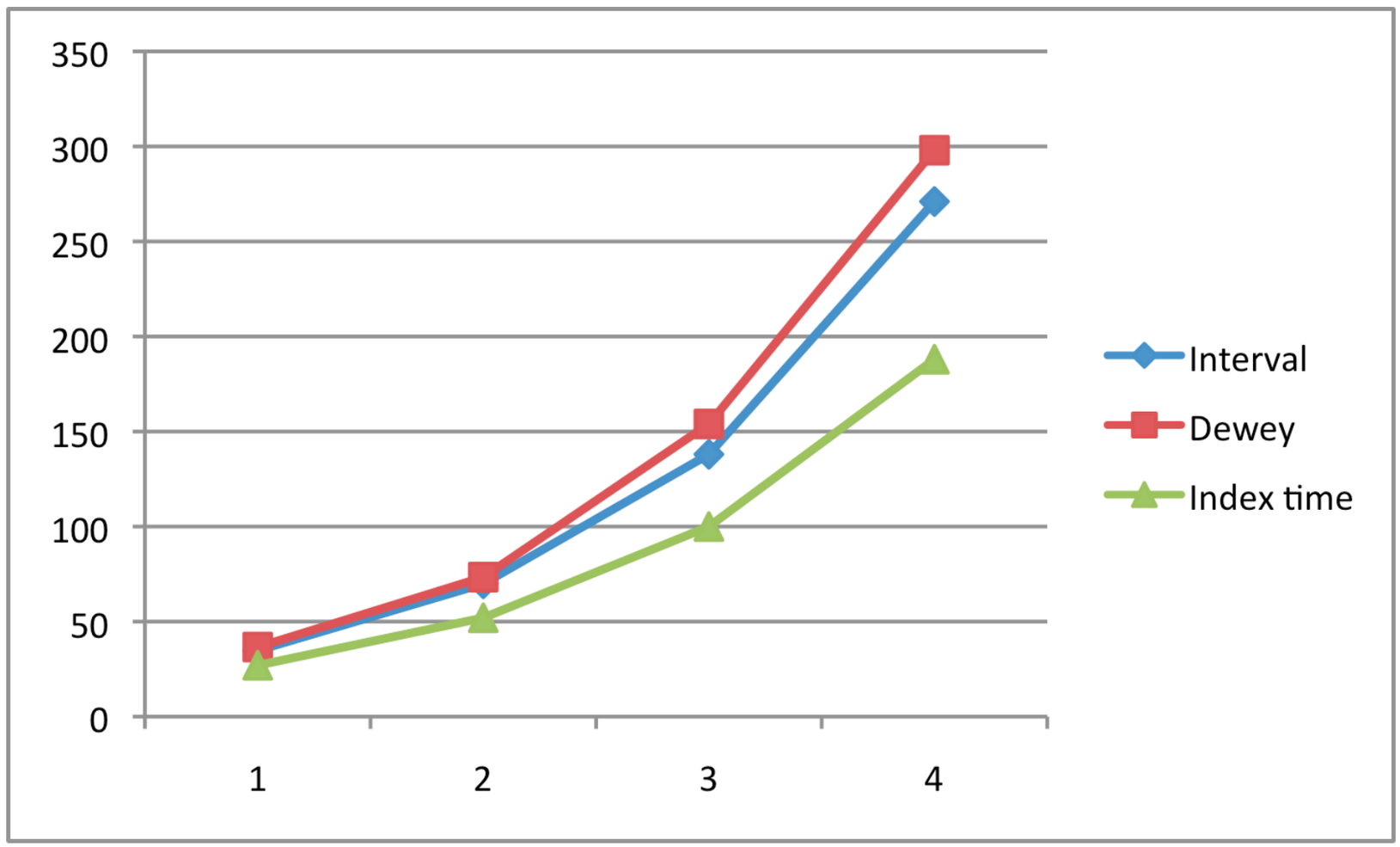
Interval and Dewey

- Running time of the tree evaluation in ms (y axis) vs. #boolean expressions in test



Conjunction matching time

- Running time of the tree evaluation in ms (y axis) vs. tree depth



Excluding conjunction matching