△▽
:-dyna.

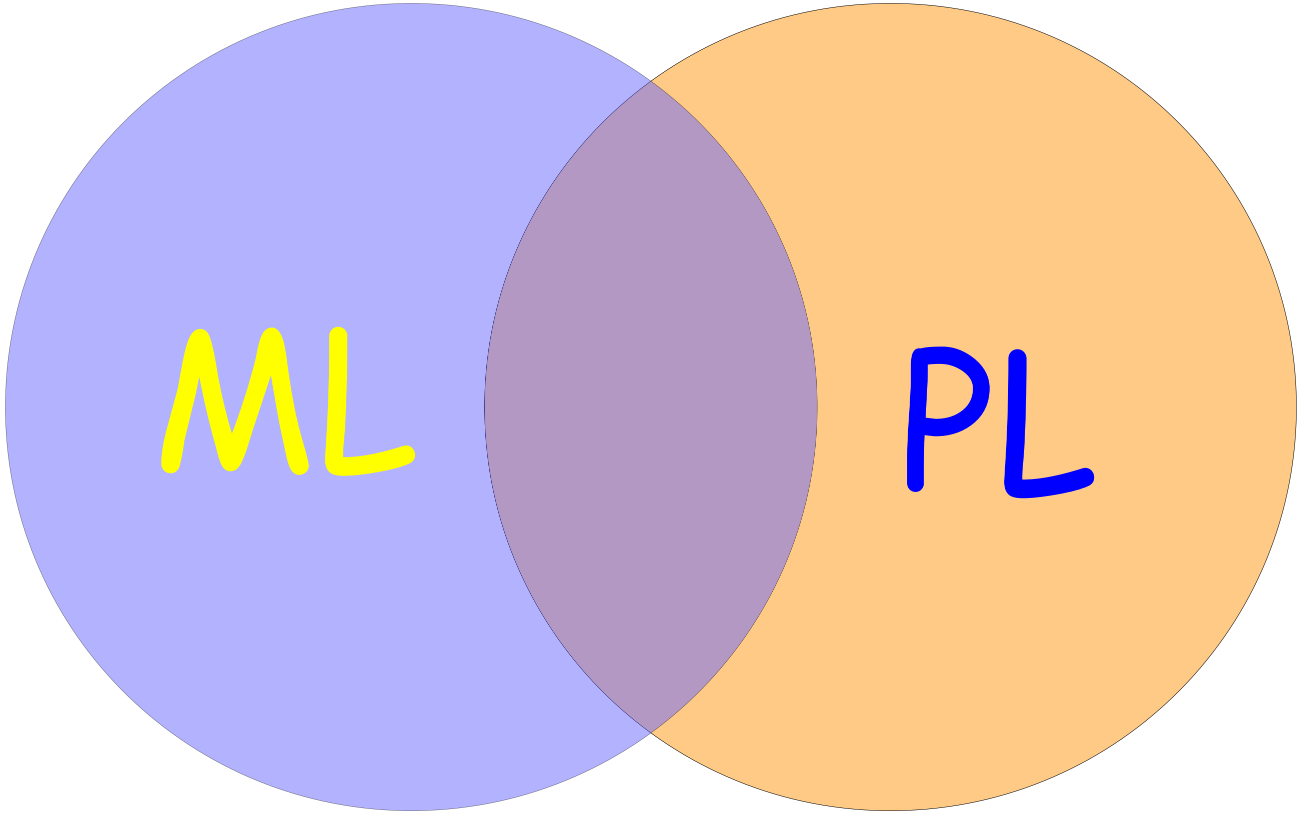
Tim Vieira, Matthew Francis-Landau,
Nathaniel Wesley Filardo, Farzad Khorasani,
and Jason Eisner

Dyna:

Toward a Self-Optimizing Declarative Language for Machine Learning Applications

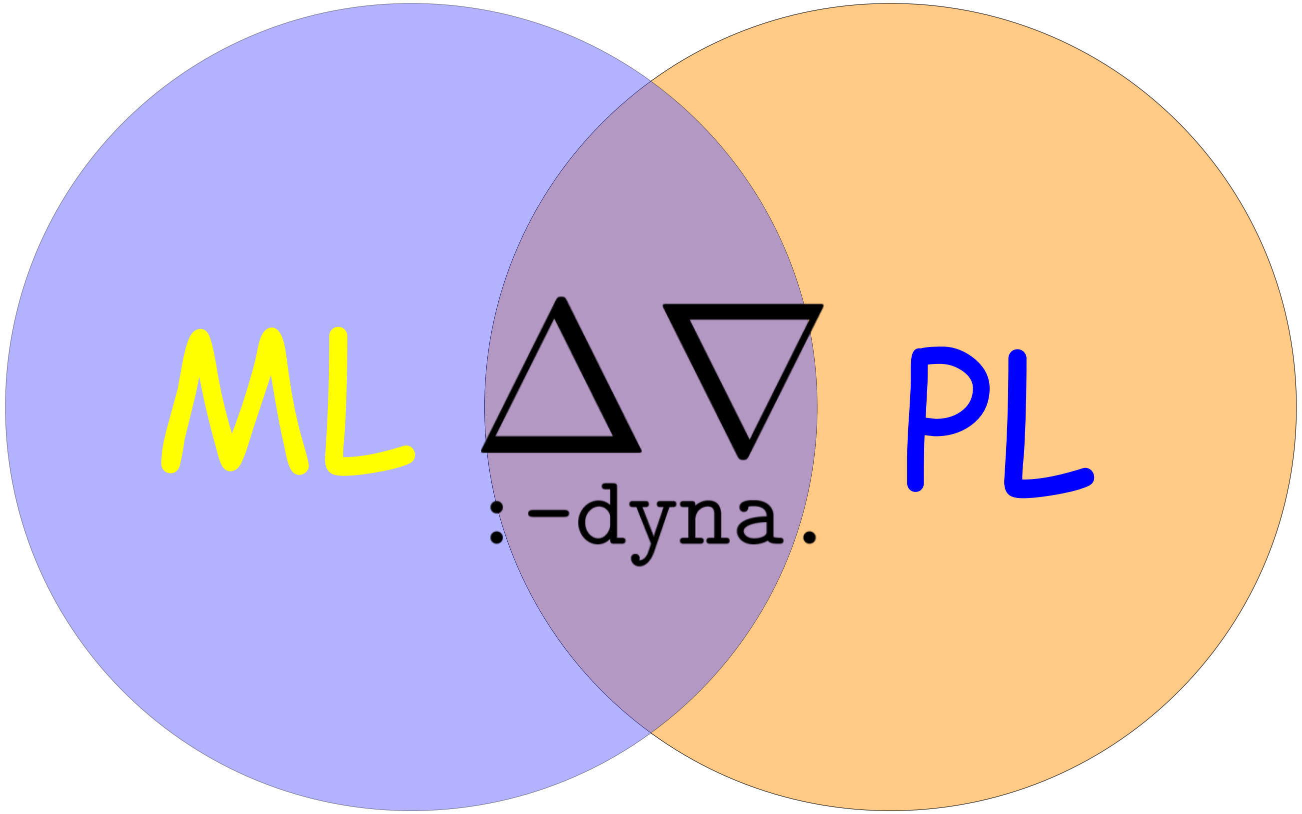
Dyna:

Toward a Self-Optimizing Declarative Language for Machine Learning Applications



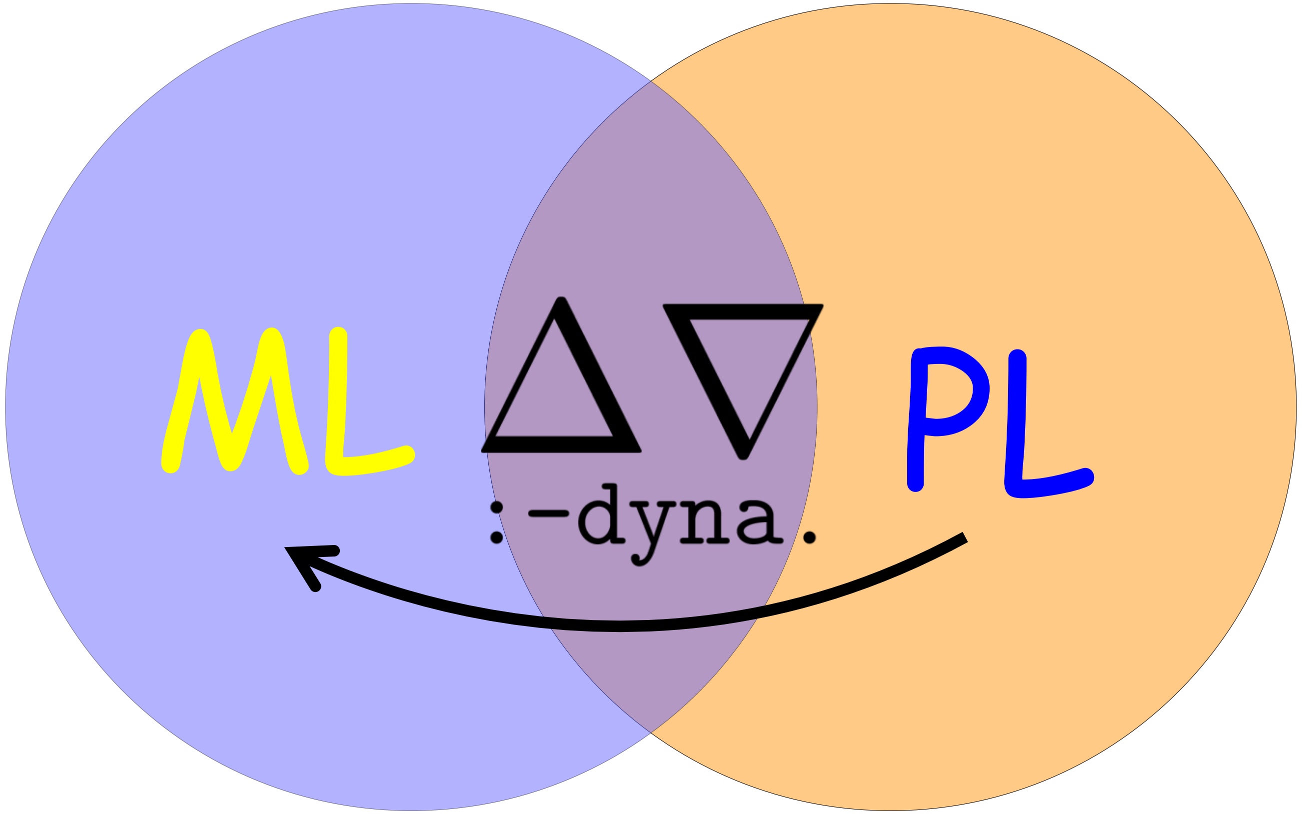
Dyna:

Toward a Self-Optimizing Declarative Language for Machine Learning Applications



Dyna:

Toward a Self-Optimizing Declarative Language for Machine Learning Applications

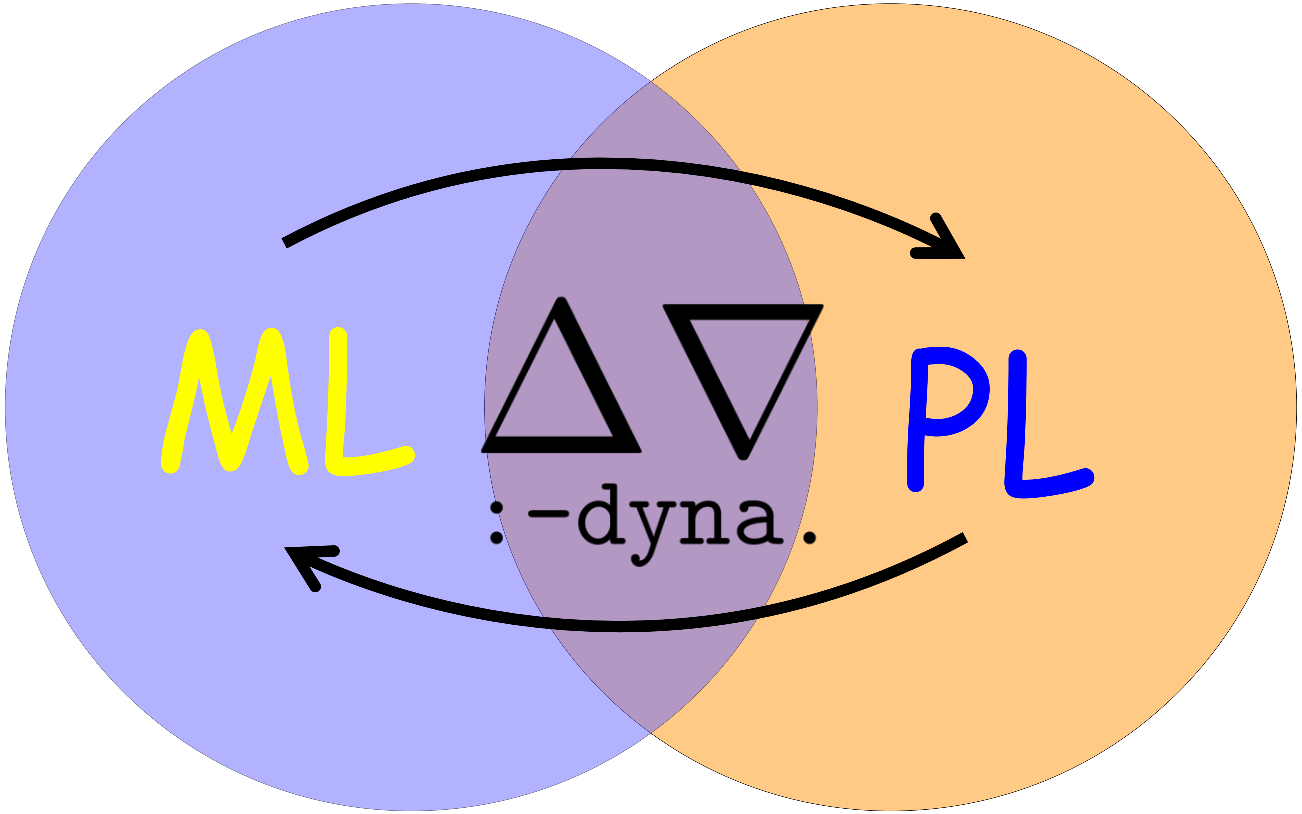


Faster to implement

Dyna:

Toward a Self-Optimizing Declarative Language for Machine Learning Applications

Faster execution

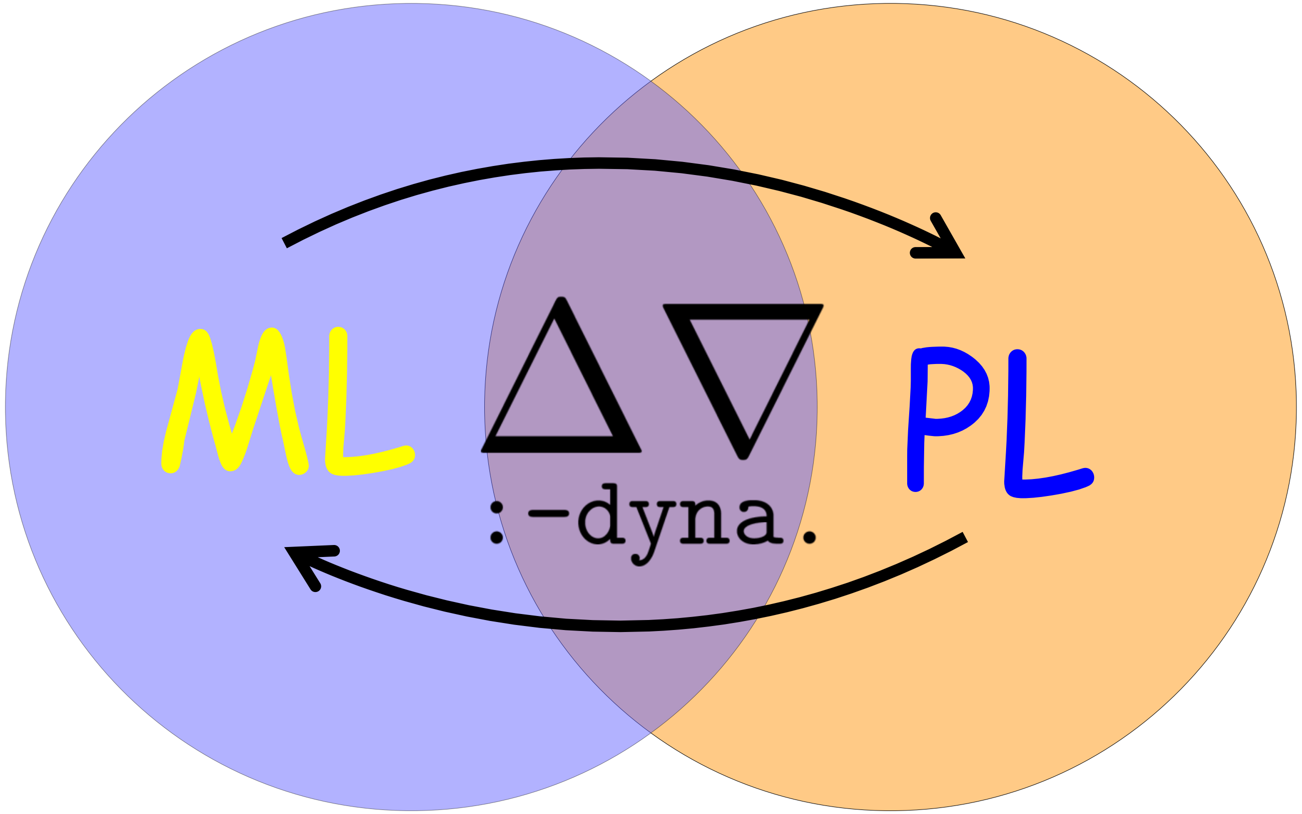


Faster to implement

Dyna:

Toward a Self-Optimizing Declarative Language for Machine Learning Applications

Faster execution



Faster to implement

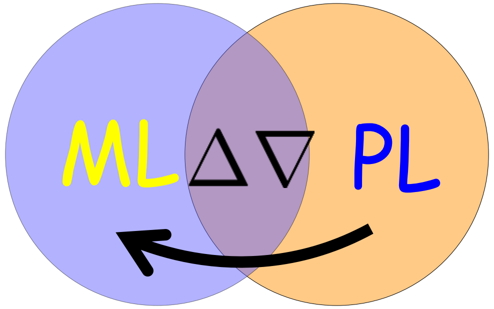
Outline

Outline

- Why Declarative Programming?

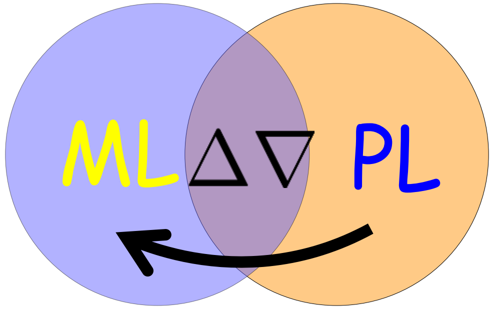
Outline

- Why Declarative Programming?
- Quick introduction to the Dyna language

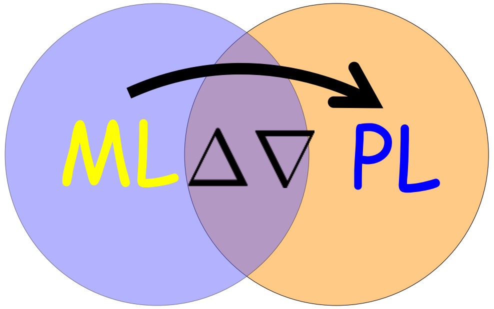


Outline

- Why Declarative Programming?
- Quick introduction to the Dyna language



- Automatic optimization of Dyna programs



Declarative Programming

Declarative Programming

A programming paradigm where the programmer specifies what to compute and leaves how to compute it to a solver.

Declarative Programming

A programming paradigm where the programmer specifies <u>what</u> to compute and leaves <u>how</u> to compute it to a solver.

- Examples: SQL, Prolog/Datalog, Mathematica, Regex, TensorFlow/Theano

Declarative Programming

A programming paradigm where the programmer specifies <u>what</u> to compute and leaves <u>how</u> to compute it to a solver.

- Examples: SQL, Prolog/Datalog, Mathematica, Regex, TensorFlow/Theano
- Solver seeks an efficient strategy (e.g., SQL query planning)

Why declarative programming?

Why declarative programming?

- Many ML algorithms have a concise declarative program

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
  - Loop orders

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
    - Loop orders
    - Data structures (e.g., hash map, dense array, linked list)

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
  - Loop orders
  - Data structures (e.g., hash map, dense array, linked list)
  - Global execution strategy (e.g., depth vs. breadth-first search)

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
    - Loop orders
    - Data structures (e.g., hash map, dense array, linked list)
    - Global execution strategy (e.g., depth vs. breadth-first search)
    - Parallelization opportunities

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
  - Loop orders
  - Data structures (e.g., hash map, dense array, linked list)
  - Global execution strategy   (e.g., depth vs. breadth-first search)
  - Parallelization opportunities
- Manually experimenting with all possibilities is time consuming

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
  - Loop orders
  - Data structures (e.g., hash map, dense array, linked list)
  - Global execution strategy (e.g., depth vs. breadth-first search)
  - Parallelization opportunities
- Manually experimenting with all possibilities is time consuming
  - Programmers usually only implement one

Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
    - Loop orders
    - Data structures (e.g., hash map, dense array, linked list)
    - Global execution strategy   (e.g., depth vs. breadth-first search)
    - Parallelization opportunities
- Manually experimenting with all possibilities is time consuming
    - Programmers usually only implement one
- Researchers don't have time to optimize the efficiency of their code
    - We can do better with automatic optimization

Why not optimize Python/Java/C++ etc.?

Why not optimize Python/Java/C++ etc.?

- Less flexibility
  - Choices of loop orders / data structures already decided by the human programmer

Why not optimize Python/Java/C++ etc.?

- Less flexibility
    - Choices of loop orders / data structures already decided by the human programmer
- Semantics of the program are not invariant to
    - Changing execution and loop order
        - Eager vs. lazy evaluation, top-down vs bottom-up evaluation.

Why not optimize Python/Java/C++ etc.?

- Less flexibility
  - Choices of loop orders / data structures already decided by the human programmer
- Semantics of the program are not invariant to
  - Changing execution and loop order
    - Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
  - Data structures

Why not optimize Python/Java/C++ etc.?

- Less flexibility
    - Choices of loop orders / data structures already decided by the human programmer
- Semantics of the program are not invariant to
    - Changing execution and loop order
        - Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
    - Data structures
    - Concurrency

Why not optimize Python/Java/C++ etc.?

- Less flexibility
  - Choices of loop orders / data structures already decided by the human programmer
- Semantics of the program are not invariant to
  - Changing execution and loop order
    - Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
  - Data structures
  - Concurrency
- Difficult to reliably discover long range interactions in a program

What is Dyna?

What is Dyna?

- Declarative language

What is Dyna?

- Declarative language
- Based on weighted logic programming

What is Dyna?

- Declarative language
- Based on weighted logic programming
- Prolog / Datalog like syntax
  - Uses pattern matching to define computation graphs

What is Dyna?

- Declarative language
- Based on weighted logic programming
- Prolog / Datalog like syntax
    - Uses pattern matching to define computation graphs
- Reactive

What is Dyna?

- Declarative language
- Based on weighted logic programming
- Prolog / Datalog like syntax
  - Uses pattern matching to define computation graphs
- Reactive
- Dyna programs are close to their mathematical description
  - Similar to functional programs

Dyna Day 1

Dyna Day 1

```
a = b * c.
```

`a` will be kept up to date if `b` or `c` changes. (Reactive)

Dyna Day 1

`a = b * c.`

a will be kept up to date if **b** or **c** changes. (Reactive)

```
b += x.
b += y.
```

equivalent to ***b = x+y.*** *(almost)*

b is a sum of <u>two</u> variables. Also kept up to date.

Dyna Day 1

```
a = b * c.
```

`a` will be kept up to date if **`b`** or **`c`** changes. (Reactive)

```
b += x.
b += y.
```

equivalent to ***`b = x+y.`*** *(almost)*

`b` is a sum of two variables. Also kept up to date.

```
c += z(1).
c += z(2).
c += z(3).
```

Dyna Day 1

`a = b * c.`

a will be kept up to date if **b** or **c** changes. (Reactive)

`b += x.`
`b += y.`

equivalent to ***b = x+y.*** *(almost)*

b is a sum of two variables. Also kept up to date.

`c += z(1).`
~~`c += z(2).`~~
~~`c += z(3).`~~

`c += z(N).`

a "patterns"
the capitalized N
matches anything

c is a sum of **all**
defined z(…) values.

Dyna Day 1

`a = b * c.`

a will be kept up to date if **b** or **c** changes. (Reactive)

`b += x.`
`b += y.` *equivalent to* ***b = x+y.*** *(almost)*

b is a sum of two variables. Also kept up to date.

`c += z(1).`
~~`c += z(2).`~~
~~`c += z(3).`~~

~~`c += z("four").`~~
~~`c += z(foo(bar,5)).`~~

`c += z(N).`

a "patterns"
the capitalized N
matches anything

c is a sum of **all**
defined z(…) values.

More interesting use of “patterns”

More interesting use of “patterns”

`a(I) = b(I) * c(I).`

- pointwise multiplication

More interesting use of “patterns”

`a(I) = b(I) * c(I).`

- pointwise multiplication

`a += b(I) * c(I).`

- dot product; could be sparse

$$\left(a = \sum_i b_i * c_i\right)$$

More interesting use of “patterns”

`a(I) = b(I) * c(I).`

- pointwise multiplication

`a += b(I) * c(I).`

- dot product; could be sparse

$$\left(a = \sum_i b_i * c_i\right)$$

More interesting use of “patterns”

`a(I) = b(I) * c(I).`

- pointwise multiplication

`a += b(I) * c(I).`

- dot product; could be sparse

$$\left(a = \sum_i b_i * c_i\right)$$

`a(I,K) += b(I,J) * c(J,K).`

- matrix multiplication; could be sparse
- **`J`** is free on the right-hand side, so we sum over it

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k}\right)$$

More interesting use of “patterns”

```
a(I) = b(I) * c(I).
```

- pointwise multiplication

```
a += b(I) * c(I).
```

- dot product; could be sparse

$$\left(a = \sum_i b_i * c_i\right)$$

```
a(I,K) += b(I,J) * c(J,K).
```

- matrix multiplication; could be sparse
- `J` is free on the right-hand side, so we sum over it

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k}\right)$$

More interesting use of “patterns”

```
a(I) = b(I) * c(I).
```

- pointwise multiplication

```
a += b(I) * c(I).
```

- dot product; could be sparse

$$\left(a = \sum_i b_i * c_i\right)$$

```
a(I,K) += b(I,J) * c(J,K).
```

- matrix multiplication; could be sparse
- **J** is free on the right-hand side, so we sum over it

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k}\right)$$

Dyna vs. Prolog

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K) :- b(I,J) , c(J,K).
```

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K) :- b(I,J) , c(J,K).
```

Dyna has “Horn equations”:

```
a(I,K) += b(I,J) * c(J,K).
```

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K)  :- b(I,J)  ,  c(J,K).
```

Dyna has "Horn equations":

```
a(I,K)  += b(I,J)  *  c(J,K).
```

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K) :- b(I,J) , c(J,K).
```

Dyna has "Horn equations":

```
a(I,K) += b(I,J) * c(J,K).
```

prove a **value** for it

e.g., a real number,

but could be any term

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K) :- b(I,J) , c(J,K).
```

Dyna has "Horn equations":

```
a(I,K) += b(I,J) * c(J,K).
```

prove a **value** for it

e.g., a real number,

but could be any term

definition from other values

b*c only has value when b and c do

if no values enter into +=, then a gets no value

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K) :- b(I,J) , c(J,K).
```

Dyna has "Horn equations":

```
a(I,K) += b(I,J) * c(J,K).
```

prove a **value** for it

e.g., a real number,
but could be any term

definition from other values

b*c only has value when b and c do
if no values enter into +=, then a gets no value

Like Prolog:
Allow nested terms
Syntactic sugar for lists, etc.
Turing-complete

Dyna vs. Prolog

Prolog has Horn clauses:

```
a(I,K) :- b(I,J) , c(J,K).
```

Dyna has "Horn equations":

```
a(I,K) += b(I,J) * c(J,K).
```

prove a **value** for it
e.g., a real number,
but could be any term

definition from other values
b*c only has value when b and c do
if no values enter into +=, then a gets no value

Like Prolog:
Allow nested terms
Syntactic sugar for lists, etc.
Turing-complete

Unlike Prolog:
Terms can have values
Terms are evaluated in place
Not just backtracking!

Shortest path

Shortest path

```
distance(X)        min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length          := distance(end).
```

Shortest path

```
distance(X)      min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length        := distance(end).

edge("a", "b") = 10.
edge("b", "c") = 2.
edge("c", "d") = 7.
edge("d", "b") = 1.
start = "a".
end = "d".
```

Shortest path

```
distance(X)      min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length        := distance(end).
```

$$\mathrm{distance}(x) = \min_{y \in \mathrm{edges}(x,\cdot)} \mathrm{edge}(x, y) + \mathrm{distance}(y)$$

$$\mathrm{distance}(\mathrm{Start}) = 0$$

$$\mathrm{Path\ length} = \mathrm{distance}(\mathrm{End})$$

Shortest path

```
distance(X)     min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length        := di    ance(end).
```

$$\min_{y\in\mathrm{edges}(x,\cdot)} \mathrm{edge}(x,y)+\mathrm{distance}(y)$$

$$\mathrm{tance}(\mathrm{End})$$

Variables not present in the head of an expression are aggregated over like with the dot product example.

Shortest path

```
distance(X)     min= edge(X, Y) + distance(Y).
distance(start) min= 0
path_length       := dis    nce(end).
```

$$\text{distance}(x) = \min_{y \in \text{edges}(x,\cdot)} \text{ed}$$

$$\text{distance}(\text{Start}) = 0$$

$$\text{Path length} = \text{distance}(\text{Enc}$$

Here the “`min=`“ aggregator only keeps the minimal value that we have computed

Shortest path

Note: Aggregation was already present in our mathematical definition.

```
n= edge(X, Y) + distance(Y).
min= 0.
:= distance(end).
```

$$\text{distance}(x) = \min_{y \in \text{edges}(x,\cdot)} \text{edge}(x, y) + \text{distance}(y)$$

$$\text{distance}(\text{Start}) = 0$$

$$\text{Path length} = \text{distance}(\text{End})$$

Shortest path

```
distance(X)      min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length        := distance(end).
```

After this converges we can *query* the state of the Dyna program.

Shortest path

```
distance(X)     min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length       := distance(end).
```

After this converges we can *query* the state of the Dyna program.

```
? path_length
```

Length at the end

Shortest path

```
distance(X)     min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length       := distance(end).
```

After this converges we can *query* the state of the Dyna program.

```
? path_length
? distance("c")
```

The distance of some other vertex

Shortest path

```
distance(X)      min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length        := distance(end).
```

After this converges we can *query* the state of the Dyna program.

```
? path_length
? distance("c")
? distance(X)
```

All of the vertices

Shortest path

```
distance(X)     min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length       := distance(end).
```

After this converges we can *query* the state of the Dyna program.

```
? path_length
? distance("c")
? distance(X)
? distance(X) > 7
```

All the vertices more than 7 away

Shortest path

```
distance(X)     min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length       := distance(end).
```

After this converges we can *query* the state of the Dyna program.

```
? path_length
? distance("c")
? distance(X)
? distance(X) > 7
? edge("a",X)
```

All of the edges leaving "a"

Aggregators

Aggregators

- Associative/commutative:
  - b **+=** a(X). % number
  - c **max=** a(X).
  - q **|=** p(X). % boolean
  - r **&=** p(X).

Aggregators

- Associative/commutative:
  - b **+=** a(X). % number
  - c **max=** a(X).
  - q **|=** p(X). % boolean
  - r **&=** p(X).

- Require uniqueness:
  - d **=** b+c.

Aggregators

- Associative/commutative:
  - b += a(X). % number
  - c max= a(X).
  - q |= p(X). % boolean
  - r &= p(X).

- Require uniqueness:
  - d = b+c.

- Last one wins:
  - fly(X) := true if bird(X).
  - fly(X) := false if penguin(X).
  - fly(bigbird) := false.

Aggregators

- Associative/commutative:
  - b += a(X). % number
  - c max= a(X).
  - q |= p(X). % boolean
  - r &= p(X).
- Require uniqueness:
  - d = b+c.
- Last one wins:
  - fly(X) := true if bird(X).
  - fly(X) := false if penguin(X).
  - fly(bigbird) := false.
- Choose any value:
  - e **?=** b.
  - e **?=** c.

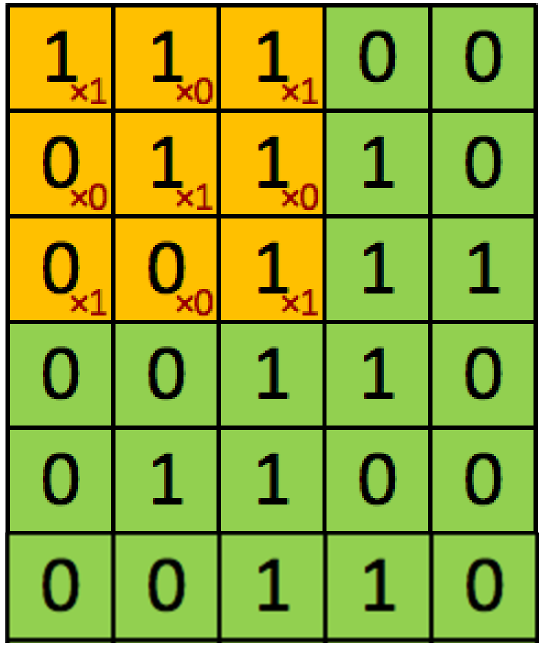
Aggregators

- Associative/commutative:
  - b += a(X). % number
  - c max= a(X).
  - q |= p(X). % boolean
  - r &= p(X).
- Require uniqueness:
  - d = b+c.
- Last one wins:
  - fly(X) := true if bird(X).
  - fly(X) := false if penguin(X).
  - fly(bigbird) := false.
- Choose any value:
  - e **?=** b.
  - e **?=** c.
- User definable aggregators
  - a(X) **smiles=** b(X, Z).

Aggregators

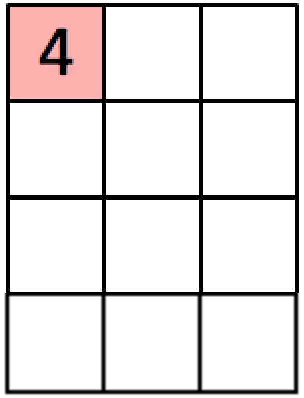
- Associative/commutative:
  - b += a(X). % number
  - c **max=** a(X).
  - q |= p(X). % boolean
  - r **&=** p(X).
- Require uniqueness:
  - d = b+c.
- Last one wins:
  - fly(X) := true if bird(X).
  - fly(X) := false if penguin(X).
  - fly(bigbird) := false.
- Choose any value:
  - e **?=** b.
  - e **?=** c.
- User definable aggregators
  - a(X) **smiles=** b(X, Z).
  - (Just define all of the operation of an commutative semigroup)

Neural Convolutional Layer



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0
0	0	1	1	0

Input image

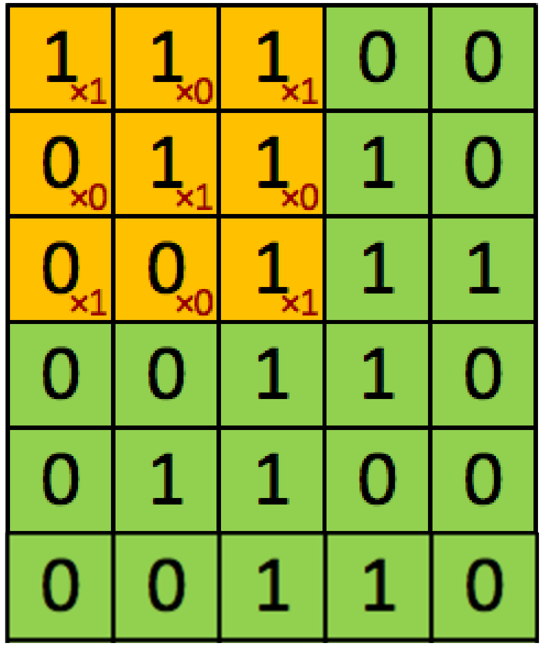


4		

Convolution output

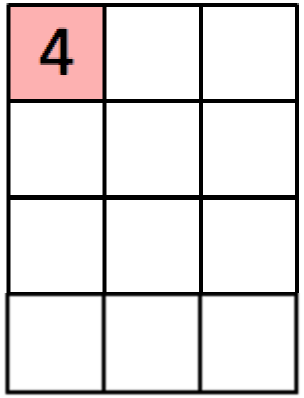
http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Neural Convolutional Layer



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0
0	0	1	1	0

Input image



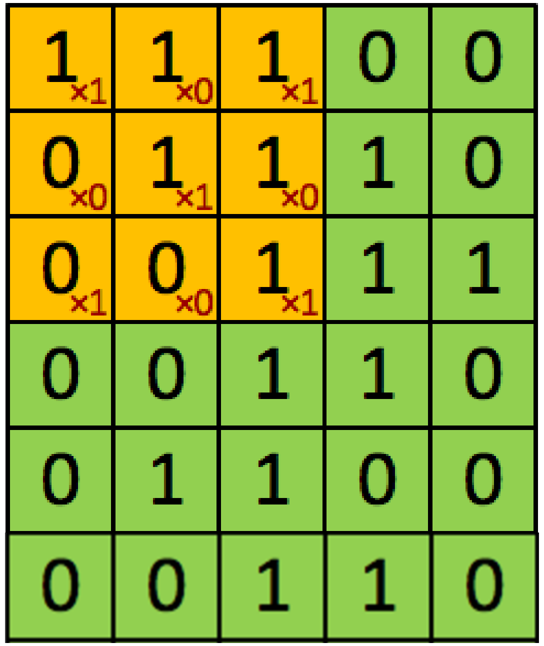
4		

Convolution output

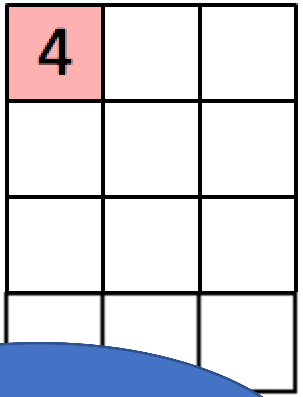
$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Neural Convolutional Layer



Input image



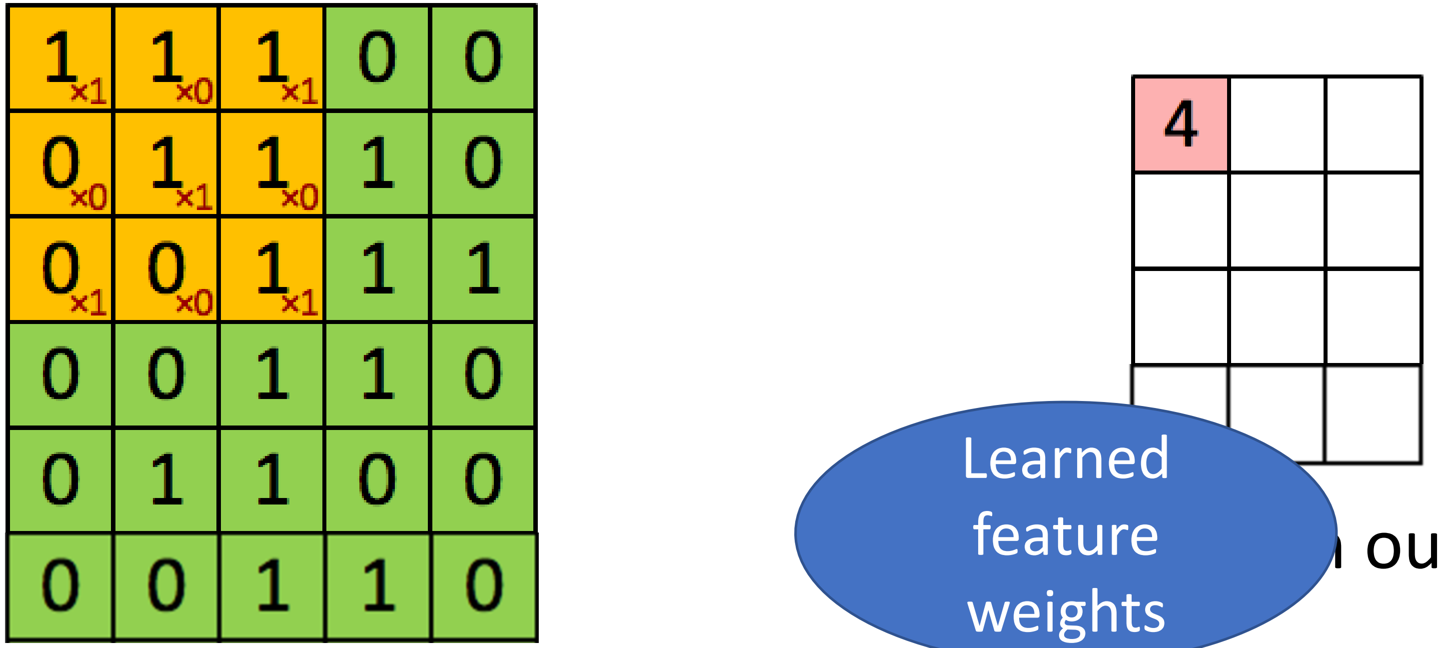
Input image

tput

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Neural Convolutional Layer



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0
0	0	1	1	0

Input image

4		

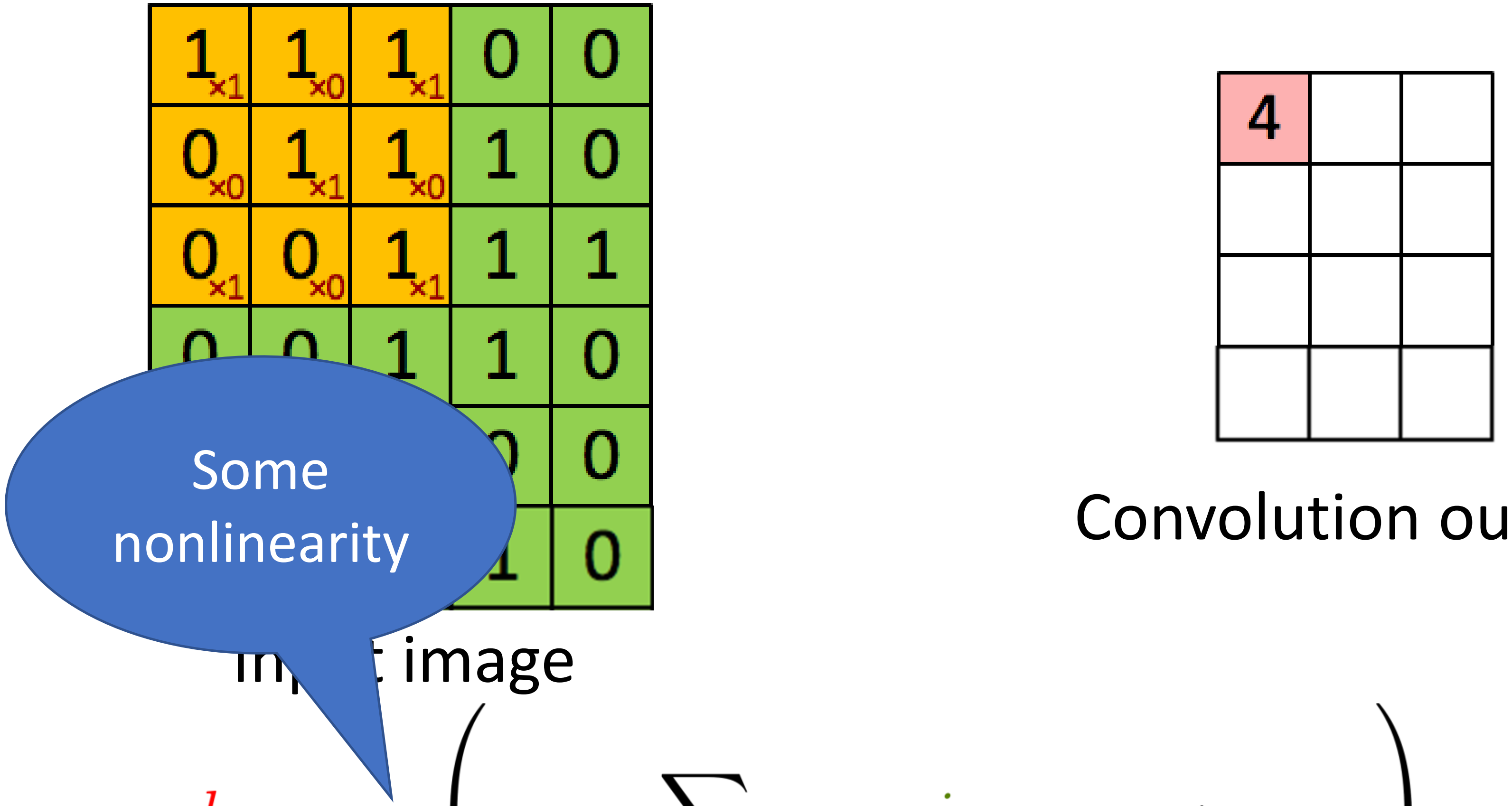
output

Learned feature weights

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Neural Convolutional Layer



In t image

Convolution output

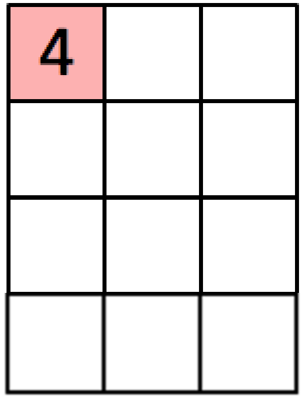
$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Neural Convolutional Layer

$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
[illegible]	1	1	0	0
[illegible]	0	1	1	0

Input image



4		

Convolution output

Convolution output

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Neural Convolutional layer

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

```
          (activation(I, J)).
(X) := 1 / (1 + exp(-X)).
```

Neural Convolutional layer

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

```
          σσ(X) := 1 / (1 + exp(-X)).
= σ(activation(I, J)).
activation(I, J) += input(I + M, J + N) * weight(M, N).
   weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY:-1..1.
```

Neural Convolutional layer

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

```
          σσ(X) := 1 / (1 + exp(-X)).
= σ(activation(I, J)).
activation(I, J) += input(I + M, J + N) * weight(M, N).
   weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY:-1..1.
```

Neural Convolutional layer

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

```
          σσ(X) := 1 / (1 + exp(-X)).
= σ(activation(I, J))
activation(I, J) += input(I + M, J + N) * weight(M, N).
   weight(DX,DY) := random(-1,1) for DX:-1..1, DY:-1..1.
```

Summation became an aggregator

Neural Convolutional layer

$$h_{i,} \quad \Bigg(\quad {}_{m+i,n+j} * w_{m,n} \Bigg)$$

We can easily define rules over $\mathbb{R} \to \mathbb{R}$

```
          σσ(X) :=    / (1 + exp(-X)).
= σ(activation(I,  J)).
activation(I, J) += input(I + M, J + N) * weight(M, N).
   weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY:-1..1.
```

Neural Convolutional layer

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

```
          σσ(X) := 1 / (1 + exp(-X)).
= σ(activation(I, J)).
activation(I, J) += input(I + M, J + N) * weight(M, N).
   weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY:-1..1.
```

Our ranges over m and n are reflected in the shape of weight

Neural Convolutional layer

$$h_{i,j} = \sigma\left(\sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} * w_{m,n}\right)$$

```
          σσ(X) := 1 / (1 + exp(-X)).
= σ(activation(I, J)).
activation(I, J) := input(I + M, J + N) * weight(M, N).
   weight(DX,DY) := (*  ..1, DY:-1..1.
```

Here keys are integers but we can also support more complicated structures

More Neural

```
σ(X) = 1 / (1 + exp(-X)).
out(J) = σ(activation(J)).
activation(J) += out(I) * edge(I,J).
```

More Neural

```
σ(X) = 1 / (1 + exp(-X)).
out(J) = σ(activation(J)).
activation(J) += out(I) * edge(I,J).
```

All weights have been
rolled into the edges
connecting neurons

More Neural

```
σ(X) = 1 / (1 + exp(-X)).
out(J) = σ(activation(J)).
activation(J) += out(I) * edge(I,J).
```

The output of a neuron is our nonlinearity applied to the sum of its inputs

More Neural

```
σ(X) = 1 / (1 + exp(-X)).
out(J) = σ(activation(J)).
activation(J) += out(I) * edge(I,J).
```

Note: nowhere in this program do we specify the form of our variables `I, J`

More Neural

```
σ(X) = 1 / (1 + exp(-X)).
out(J) = σ(activation(J)).
activation(J) += out(I) * edge(I,J).
```

Instead we can specify the structure of keys inside the definition of edge.

```
edge(input(X,Y),hidden(X+DX,Y+DY)) = weight(DX,DY).
weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY:-1..1.
```

More Neural

```
σ(X) = 1 / (1 + exp(-X)).
out(J) = σ(activation(J)).
activation(J) += out(I) * edge(I,J).
```

```
edge(input(X,Y),hidden(X+DX,Y+DY)) = weight(DX,DY).
weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY -1..1.
```

The weight do not depend on the absolute location of the input, so this is a convolution again.

Dyna Makes Algorithms Easy

Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject

Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
- Iterative algorithms – loopy belief propagation, numerical optimization

Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
- Iterative algorithms – loopy belief propagation, numerical optimization
- Neural networks – computation graphs passing dense matrices

Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
- Iterative algorithms – loopy belief propagation, numerical optimization
- Neural networks – computation graphs passing dense matrices
- Branch-and-bound, Davis–Putnam–Logemann–Loveland (DPLL)

Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likeliho or reject
- Iterative algorithms – loopy belief propag optimization
- Neural networks – computation graphs passin
- Branch-and-bound, Davis–Putnam–Logeman

All use graphs to propagate changes

Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likeliho … or reject
- Iterative algorithms – loopy belief propag
  optimization
- Neural networks – computation graphs passin
- Branch-and-bound, Davis–Putnam–Logeman

All use graphs to propagate changes

- Implementations and more in:
  - Dyna: Extending Datalog for modern AI. (Eisner & Filardo 2011)
  - Dyna: A non-probabilistic language for probabilistic AI. (Eisner 2009)

How much can a declarative language save us?

Implementing shortest path

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
               + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
              + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Java (Procedural)

```
queue = new FifoQueue<Pair<String, Float>>();
distances = new HashMap<String, Float>();
edges = new HashMap<Pair<String, String>,Float>();
// load edges
queue.push("start");
while(!queue.empty()) {
  d = queue.pop();
  for(e : edge) {
    if(e.first().second().equals(d.first())) {
      if(distance.get(e.first()) <
         d.second() + e.second()) {
        distance.put(e.first(),
                     d.second() + e.second());
        queue.push(e.first());
      }
    }
  }
}
path_length = distances.get("end");
```

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
               + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Java (Procedural)

```
queue = new PriorityQueue<String, Float>();
distances = new HashMap<String, Float>();
edges = new HashMap<Pair<String, String>,Float>();
// load edges
queue.push("start", 0);
while(!queue.empty()) {
  d = queue.pop();
  for(e : edge) {
    if(e.first().second().equals(d.first())) {
      n = d.second() + e.second();
      if(distance.get(e.first()) < n) {
        distance.put(e.first(), n);
        queue.push(e.first(), n);
      }
    }
  }
}
path_length = distances.get("end");
```

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
              + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Java (Procedural)

```
queue = new PriorityQueue<String, Float>();
distances = new HashMap<String, Float>();
edges = new HashMap<String,Map<String,Float>>();
// load edges
queue.push("start", 0);
while(!queue.empty()) {
  d = queue.pop();
  for(e : edge.get(d.first())) {
    n = d.second() + e.second();
    if(distance.get(e.first()) < n) {
      distance.put(e.first(), n);
      queue.push(e.first(), n);
    }
  }
}
path_length = distances.get("end");
```

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
              + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Java (Procedural)

```
placeIndex = new HashMap<String,Integer>();
queue = new PriorityQueue<Integer, Float>();
distances = new float[num_places];
edges = new float[num_places][num_places];
// load edges
queue.push(placesIndex.get("start"), 0);
while(!queue.empty()) {
  d = queue.pop();
  l = edges[d.first()];
  for(j = 0; j < l.length; j++) {
    n = d.second() + l[j];
    if(distances[j] < n) {
      distances[j] = n;
      queue.push(j, n);
    }
  }
}
path_length = distances[placeIndex.get("end")];
```

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
               + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Java (Procedural)

```
placeIndex = new HashMap<String,Integer>();
edges = new float[num_places][num_places];
// load edges
float distance(from) {
  if(from == placeIndex.get("start")) {
    return 0;
  }
  l = edges[from];
  r = infinity;
  for(j = 0; j < l.length; j++) {
    n = distance(j) + l[j];
    if(n < r)
      r = n;
  }
  return r;
}
path_length = distance(placeIndex.get("end"));
```

Implementing shortest path

Dyna (Declarative)

```
distance(X) min= edge(X, Y)
                 + distance(Y).
distance(start) min= 0.
path_length = distance(end).
```

Java (Procedural)

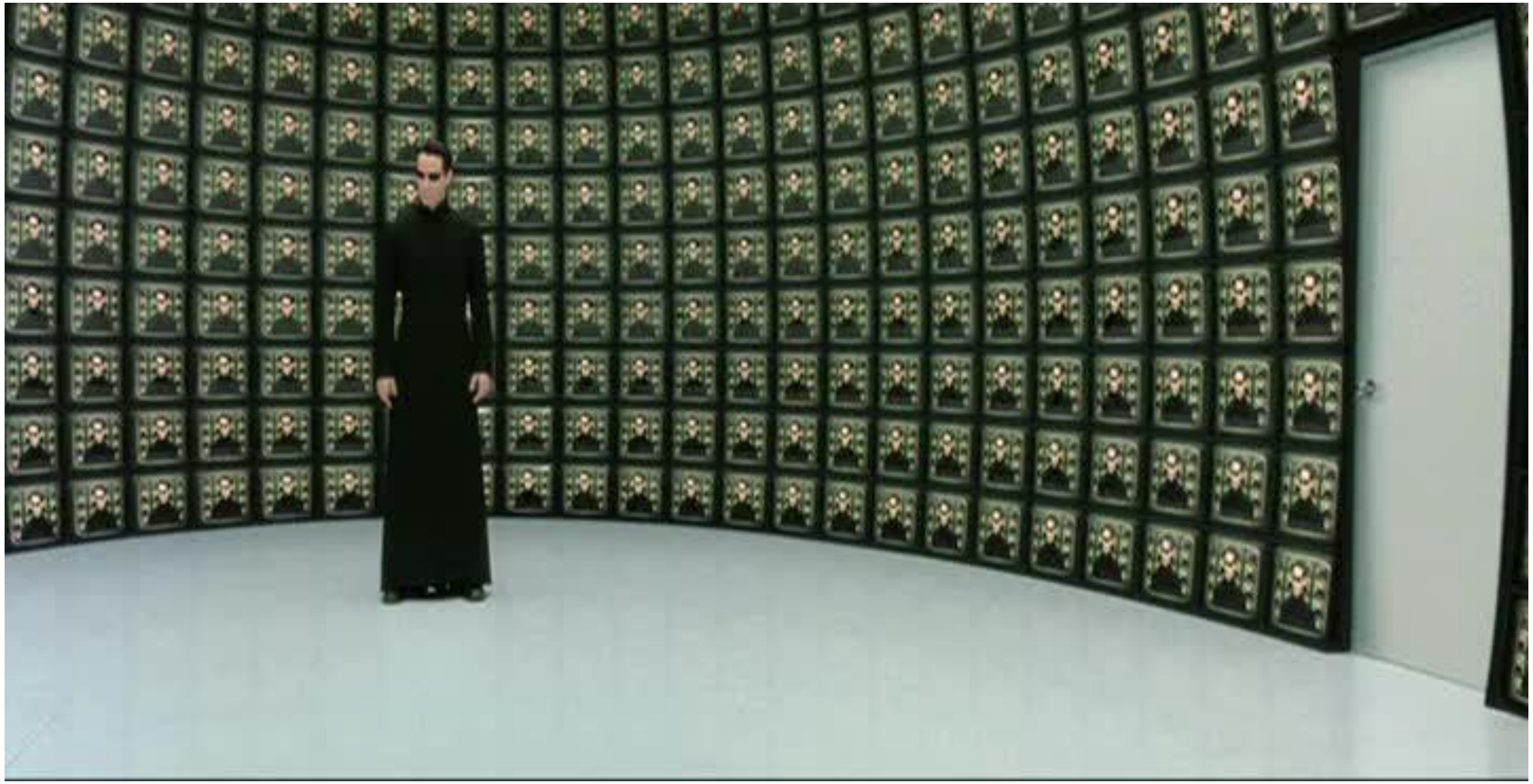
```
placeIndex = ne                            ();
edges = new
// load
float
  i
}
path_
```

A single Dyna program can represent hundreds of possible implementations.

Other implementations (not shown here) include A* and bidirectional search, and choice of data structures to support dynamic graphs

Given all of these implementations,

Given all of these implementations, the problem is choice



The Matrix Reloaded (2003)

If you are Neo, you have two choices

If you are Neo, you have two choices

- Take the Architect’s deal
    - Restart the Matrix
    - Let all the humans in Zion die
        - But restart Zion with 16 females and 7 males (fight another day)

If you are Neo, you have two choices

- Take the Architect’s deal
  - Restart the Matrix
  - Let all the humans in Zion die
    - But restart Zion with 16 females and 7 males (fight another day)
  - Already tried this <u>5 times</u>

If you are Neo, you have two choices

- Take the Architect’s deal
  - Restart the Matrix
  - Let all the humans in Zion die
    - But restart Zion with 16 females and 7 males (fight another day)
  - Already tried this 5 times
  - Current argmax

If you are Neo, you have two choices

- Take the Architect’s deal
    - Restart the Matrix
    - Let all the humans in Zion die
        - But restart Zion with 16 females and 7 males (fight another day)
    - Already tried this <u>5 times</u>
    - Current argmax
- Follow “an emotion specifically designed to overwhelm logic & reason”
    - Save Trinity

If you are Neo, you have two choices

- Take the Architect’s deal
    - Restart the Matrix
    - Let all the humans in Zion die
        - But restart Zion with 16 females and 7 males (fight another day)
    - Already tried this <u>5 times</u>
    - Current argmax
- Follow “an emotion specifically designed to overwhelm logic & reason”
    - Save Trinity
    - YOLO, figure this out as we go (unknown reward)

This raises the next question …

This raises the next question …

can machines love

This raises the next question …

can machines love

or at least make irrational choices

This raises the next question …

can machines love

or at least make irrational choices

$$action = \text{argmax } \pi(\cdot | \ldots)$$

The “Rational” Choice

This raises the next question …
can machines love
or at least make irrational choices

$$action = \text{argmax}\ \pi(\cdot|\ldots)$$

The "Rational" Choice

$$action \sim \pi(\cdot|\ldots)$$

The "Irrational" Choice

(Randomly sample)

This raises the next question …

can machines love

or at least make irrational choices

$$action = \text{argmax } \pi(\cdot|\ldots)$$

The “Rational” Choice

Exploitation

$$action \sim \pi(\cdot|\ldots)$$

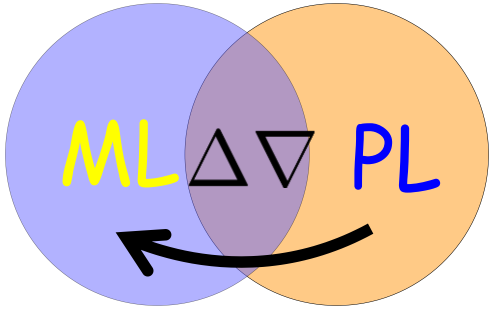
The “Irrational” Choice

(Randomly sample)

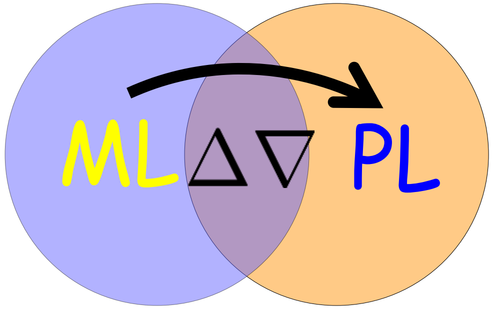
Exploration

Outline

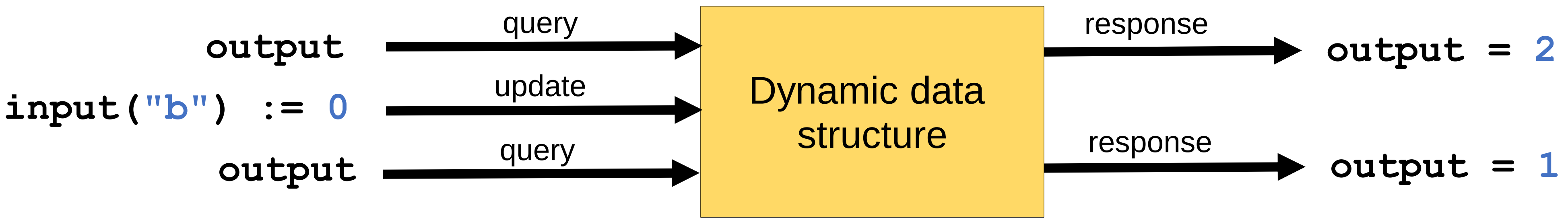
- Why Declarative Programming?
- Quick introduction to the Dyna language



- **Automatic optimization of Dyna programs**



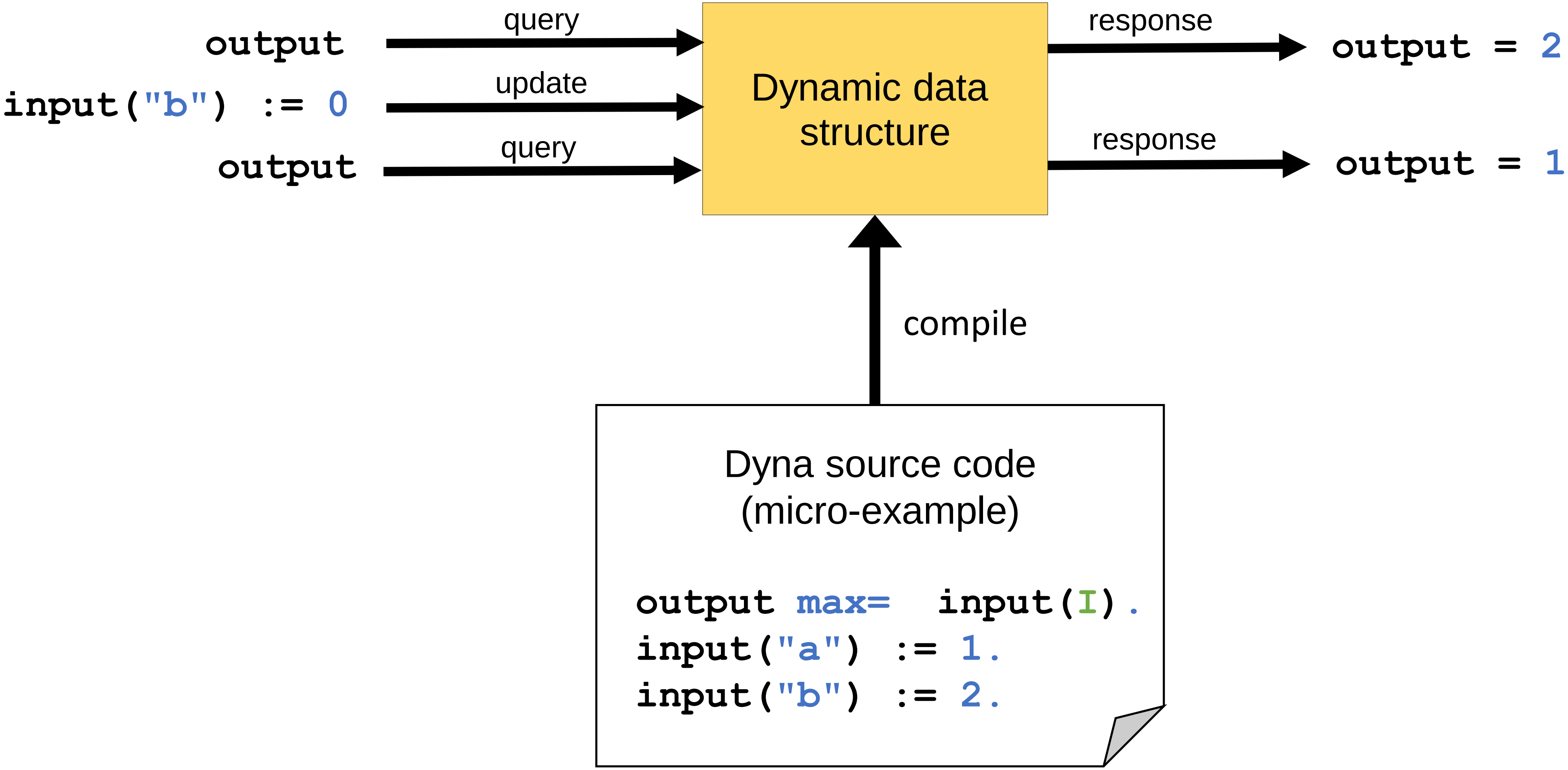
Dyna



Dyna source code
(micro-example)

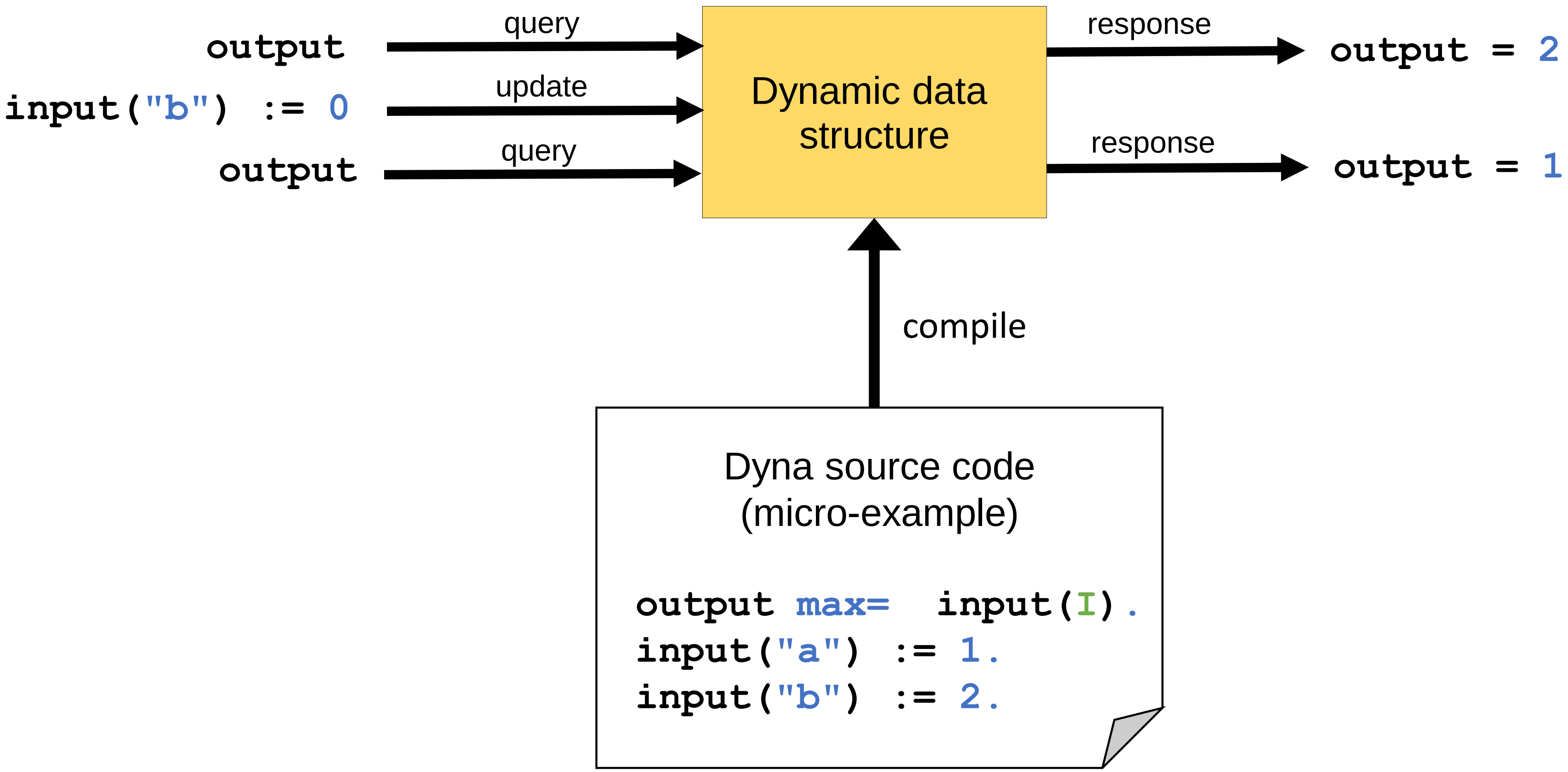
```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

Dyna



```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

Dyna



output

query

input("b") := 0

update

output

query

Dynamic data structure

response

output = 2

response

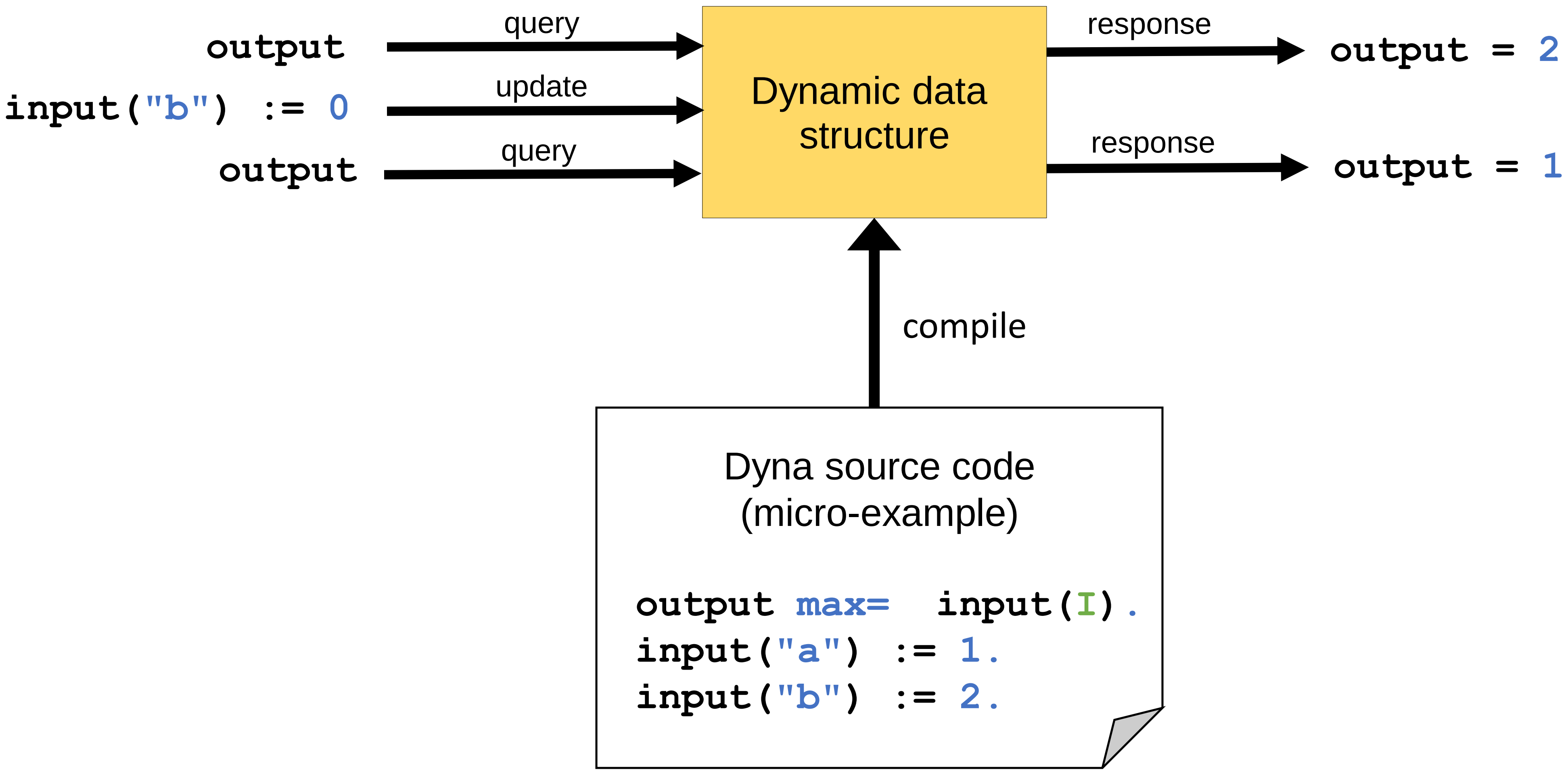
output = 1

compile

Dyna source code (micro-example)

```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

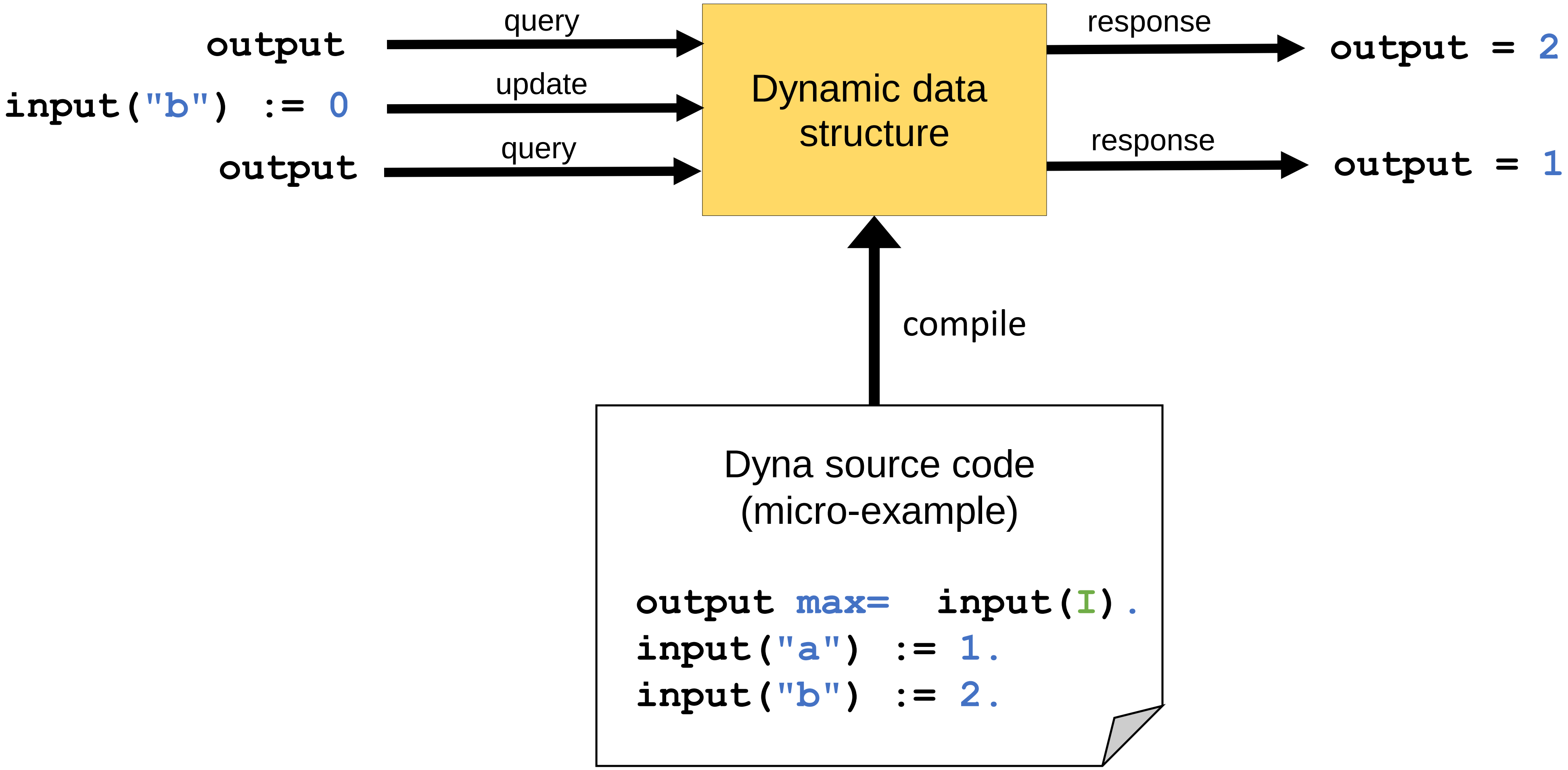
Dyna



Dyna source code
(micro-example)

```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

Dyna



`output` query

`input("b") := 0` update

`output` query

Dynamic data structure

response `output = 2`

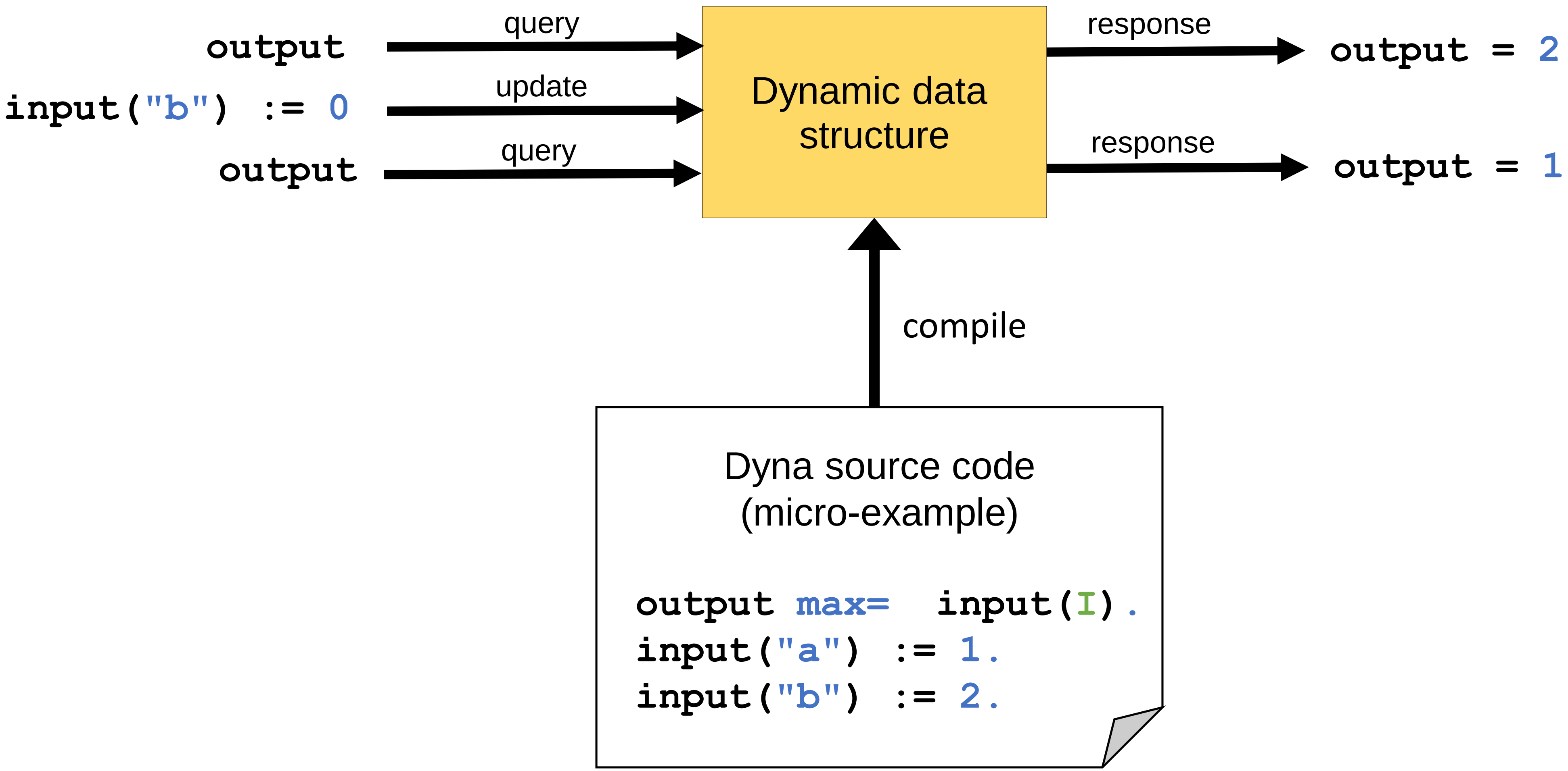
response `output = 1`

compile

Dyna source code (micro-example)

```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

Dyna



`output` query

`input("b") := 0` update

`output` query

Dynamic data structure

response `output = 2`

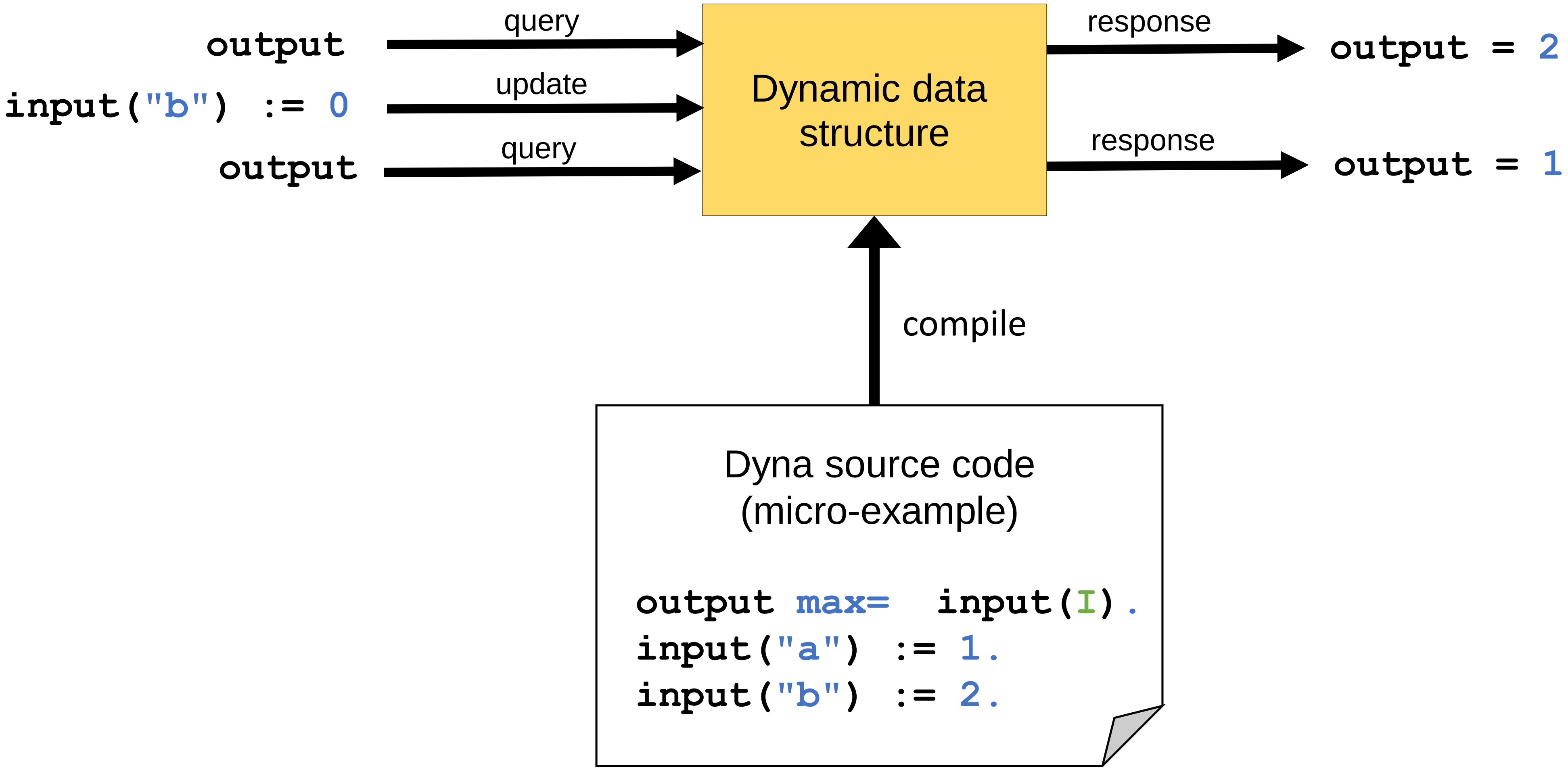
response `output = 1`

compile

Dyna source code (micro-example)

```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

Dyna



`output` query

`input("b") := 0` update

`output` query

Dynamic data structure

response `output = 2`

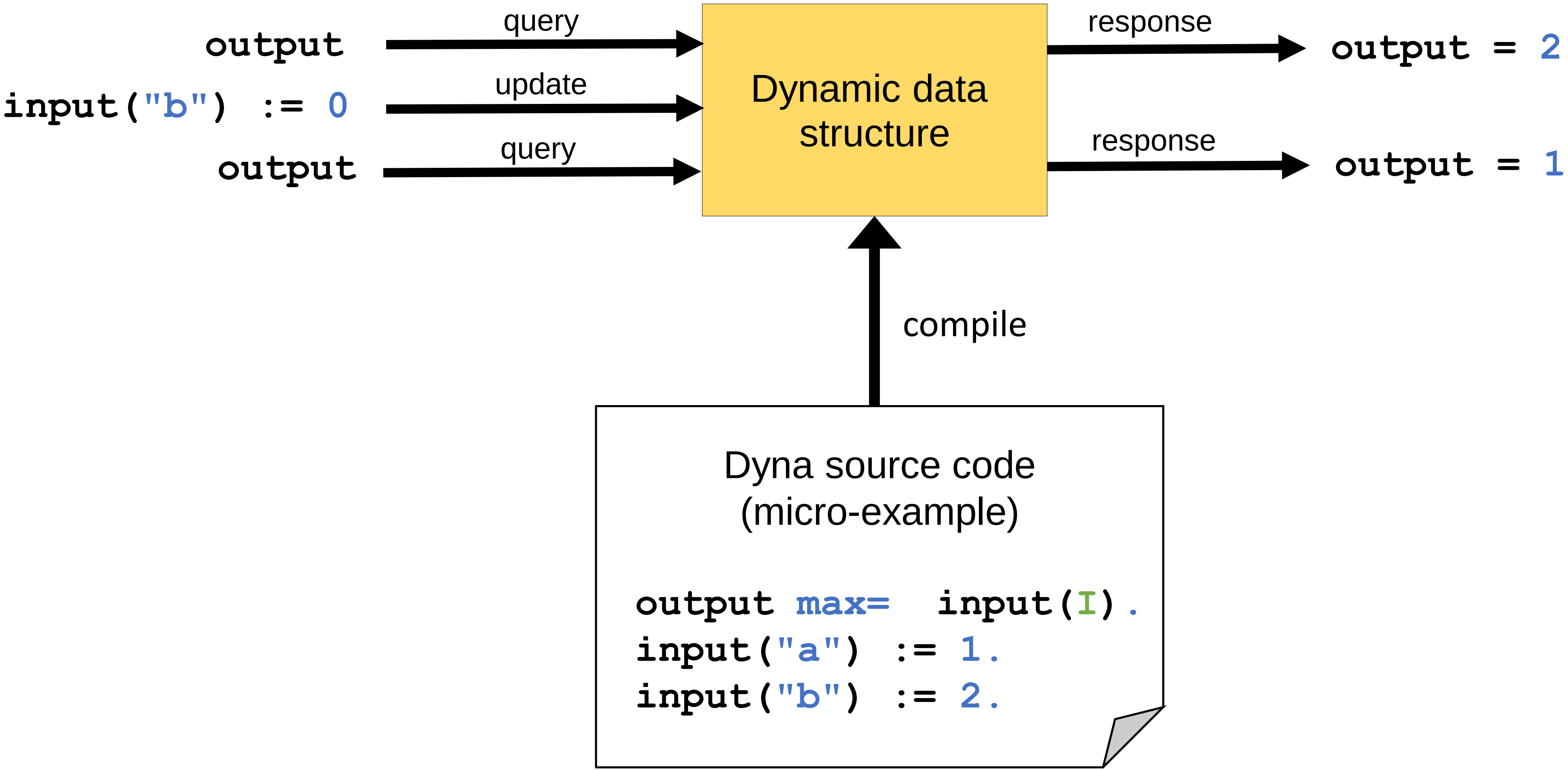
response `output = 1`

compile

Dyna source code (micro-example)

```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

Dyna



`output` query

`input("b") := 0` update

`output` query

Dynamic data structure

response `output = 2`

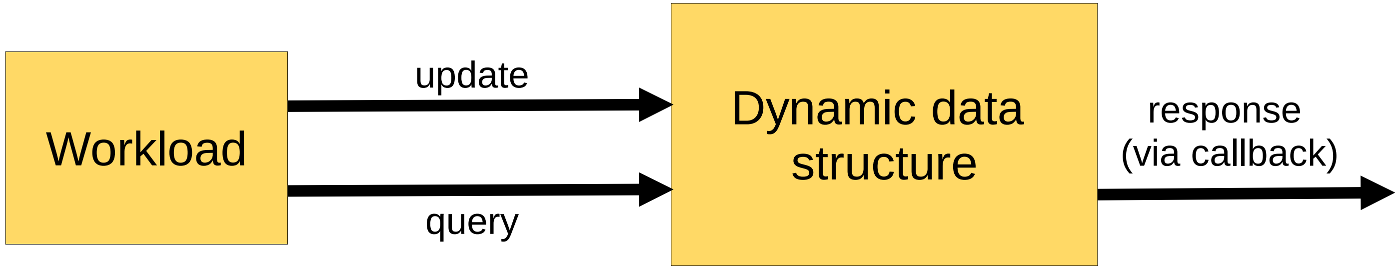
response `output = 1`

compile

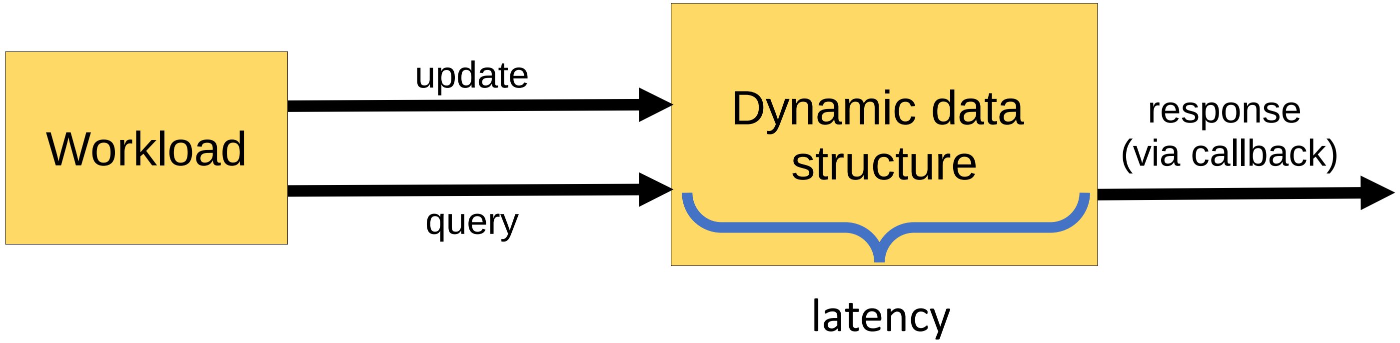
Dyna source code (micro-example)

```
output max=  input(I).
input("a") := 1.
input("b") := 2.
```

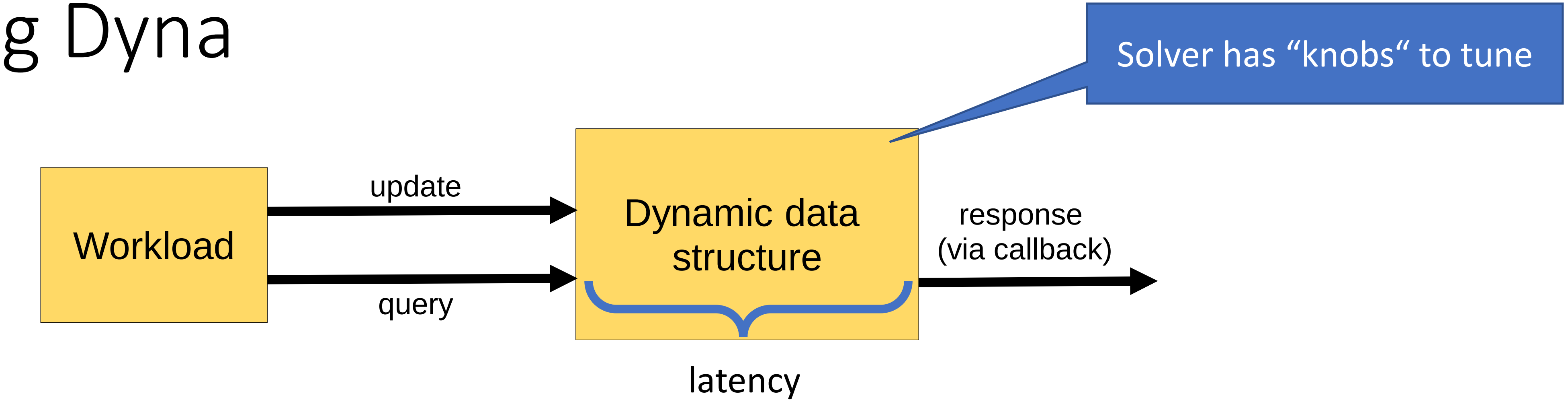
Tuning Dyna


Workload
update
query
Dynamic data structure
response (via callback)

Tuning Dyna


Workload
update
query
Dynamic data structure
response
(via callback)
latency

Tuning Dyna



Solver has “knobs“ to tune

Workload

update

query

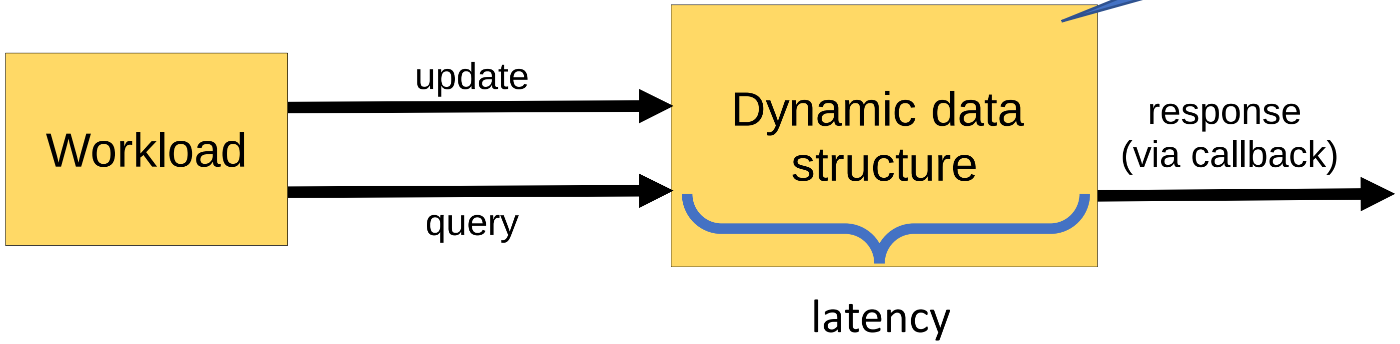
Dynamic data structure

response (via callback)

latency

Tuning Dyna

Solver has “knobs“ to tune



Workload

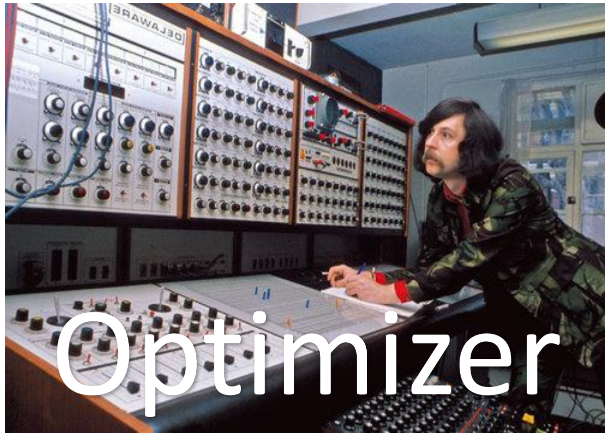
update

query

Dynamic data structure

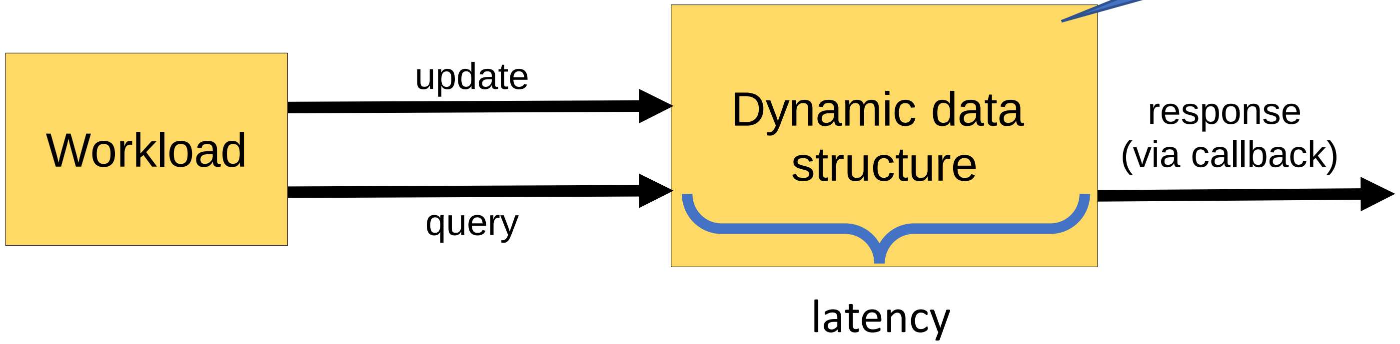
latency

response (via callback)



Tuning Dyna

Solver has “knobs“ to tune


Workload

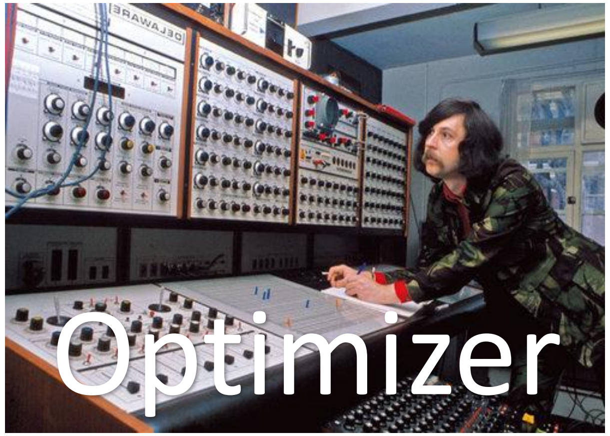
update

query

Dynamic data structure

latency

response (via callback)



Example knob: eager or lazy updates?
e.g., **dynamic max data structure**

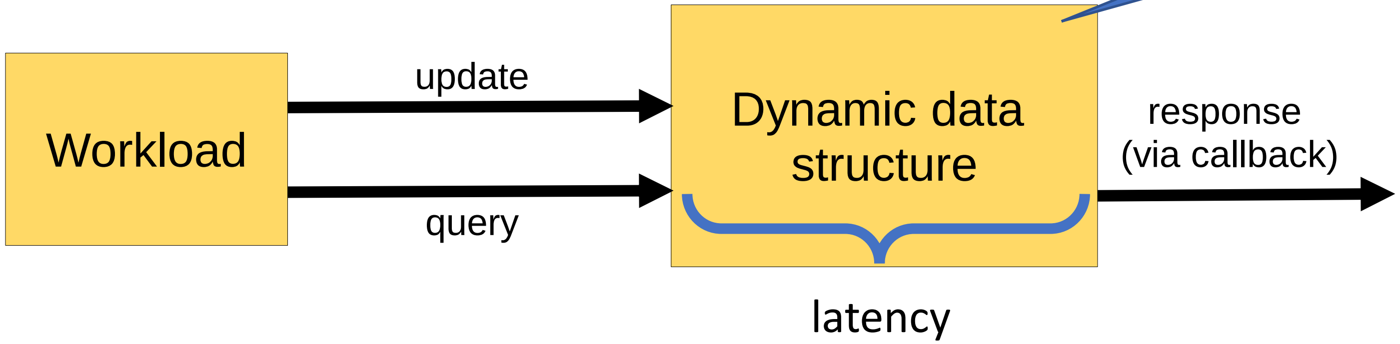
```
% Dyna:
output max= input(I).
```

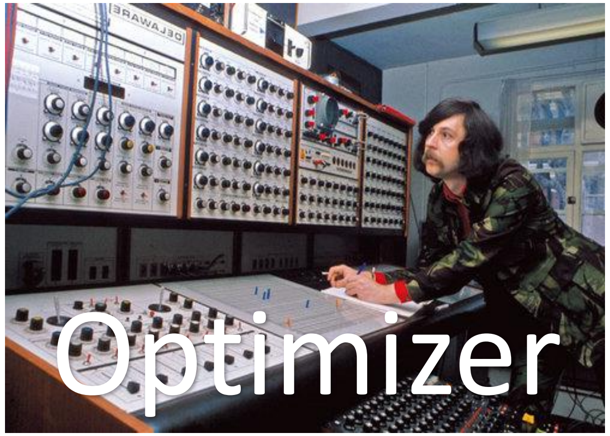
Max-heap:

- *O(log n)* per update
- *O(n)* per batch update (“heapify”)

Tuning Dyna

Solver has “knobs“ to tune





$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \,\mathrm{latency}(i)\right]$$

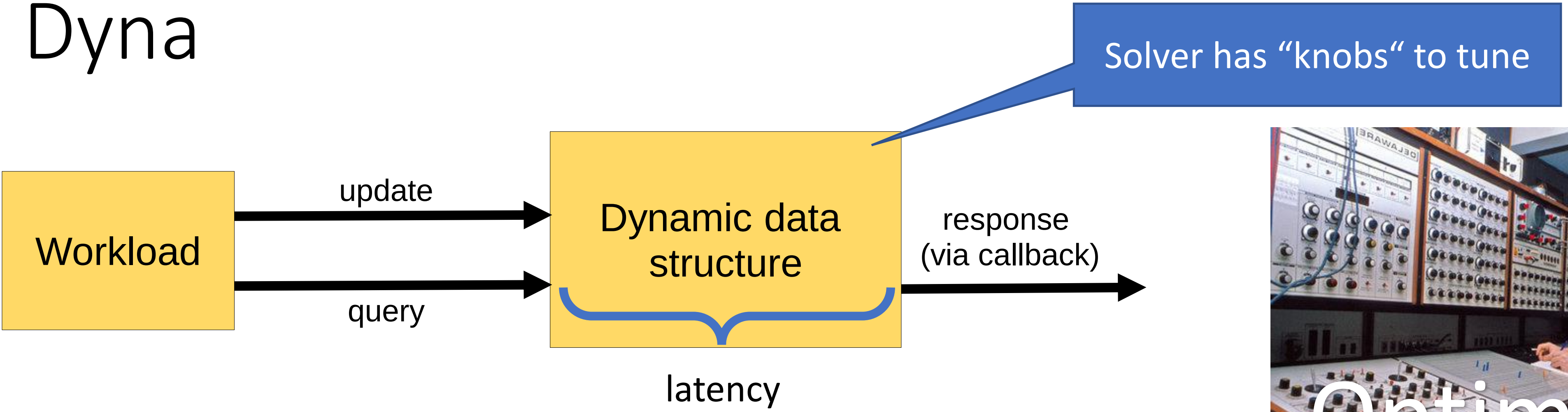
Example knob: eager or lazy updates?
e.g., **dynamic max data structure**

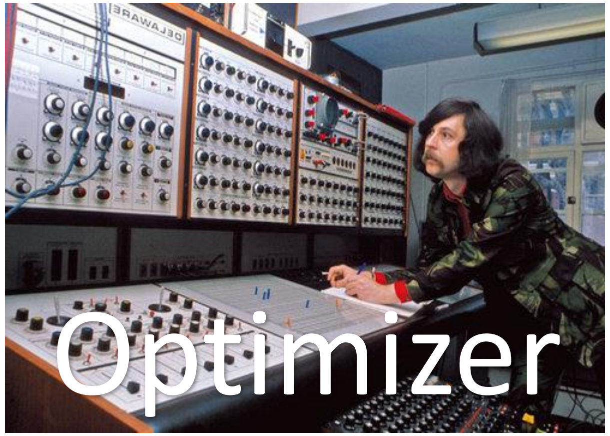
```
% Dyna:
output max= input(I).
```

Max-heap:

- *O(log n)* per update
- *O(n)* per batch update (“heapify”)

Tuning Dyna





Example knob: eager or lazy updates?
e.g., **dynamic max data structure**

```
% Dyna:
output max= input(I).
```

Max-heap:

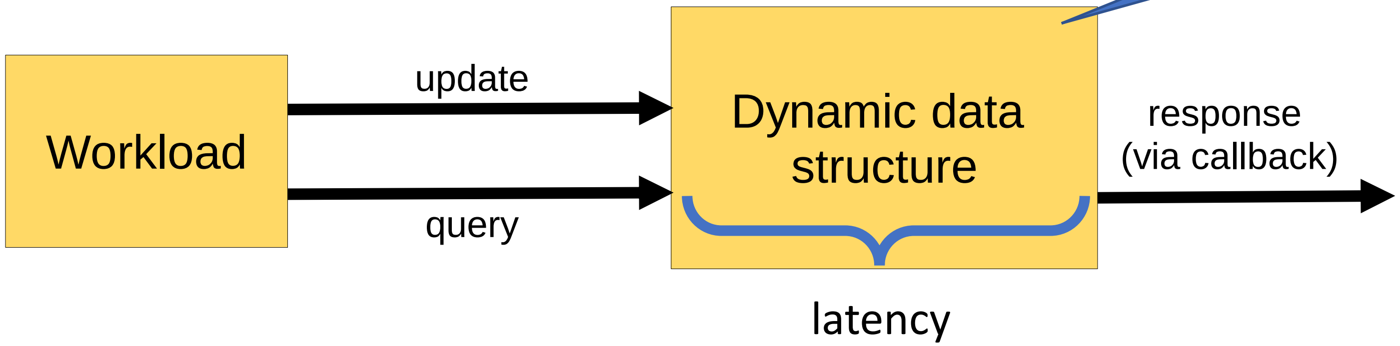
- *O(log n)* per update
- *O(n)* per batch update (“heapify”)

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \,\mathrm{latency}(i)\right]$$

knob settings

Tuning Dyna

Solver has “knobs“ to tune



Workload

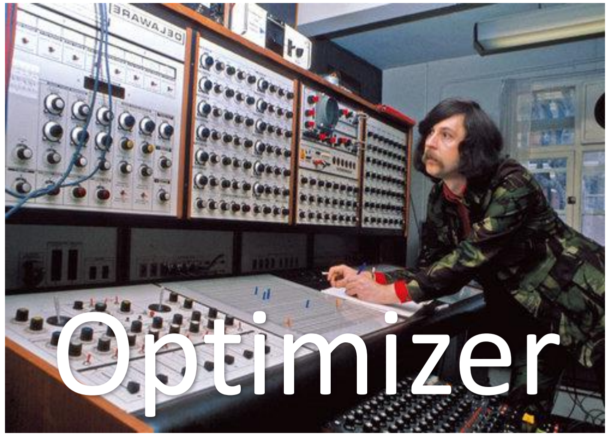
update

query

Dynamic data structure

response (via callback)

latency



Total cost knob setting
Average latency on workload

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^{i} \lambda_i \,\mathrm{latency}(i)\right]$$

knob settings

Example knob: eager or lazy updates?
e.g., **dynamic max data structure**

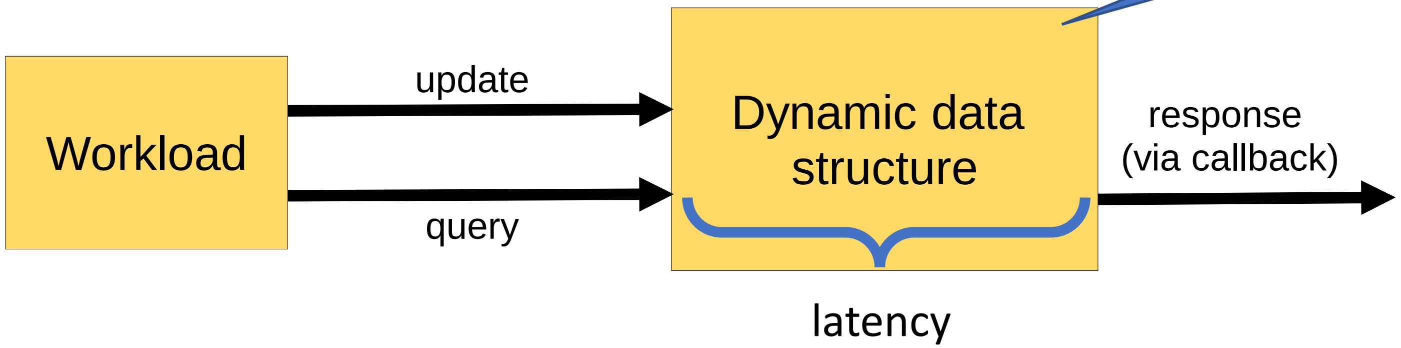
```
% Dyna:
output max= input(I).
```

Max-heap:

- *O(log n)* per update
- *O(n)* per batch update (“heapify”)

Tuning Dyna

Solver has “knobs“ to tune



Workload

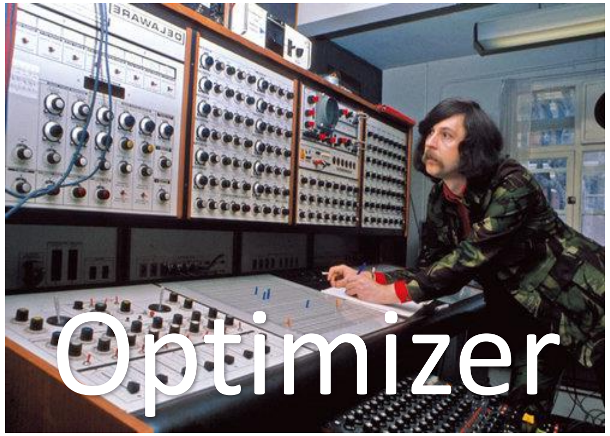
update

query

Dynamic data structure

response (via callback)

latency



Total cost knob setting
Average latency on workload

Encourage earlier jobs to finish first

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \, \text{latency}(i)\right]$$

knob settings

Example knob: eager or lazy updates?
e.g., **dynamic max data structure**

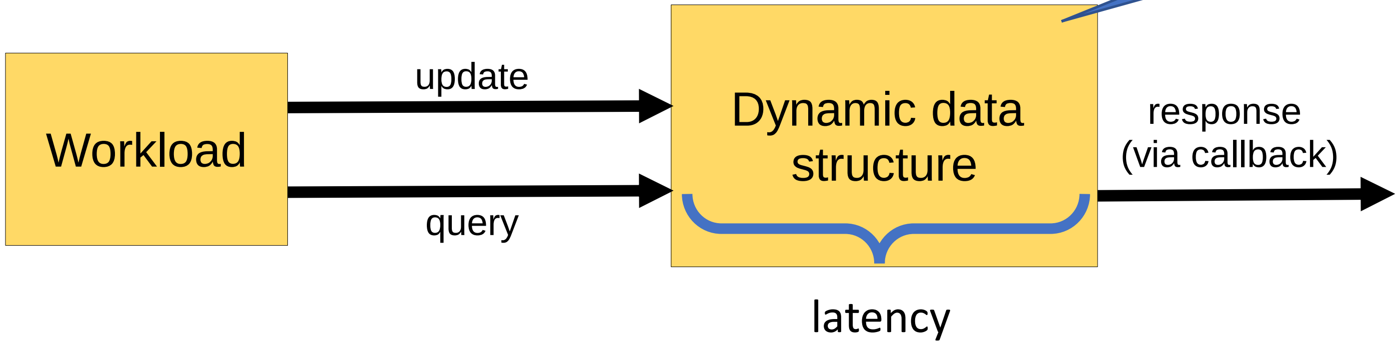
```
% Dyna:
output max= input(I).
```

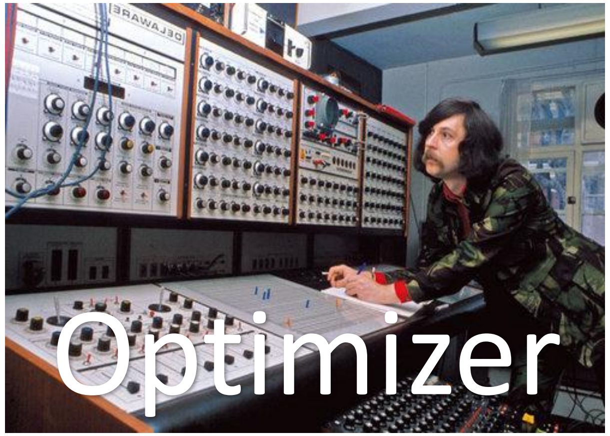
Max-heap:

- *O(log n)* per update
- *O(n)* per batch update (“heapify”)

Tuning Dyna

Solver has “knobs“ to tune





Total cost knob setting
Average latency on workload

Encourage earlier jobs to finish first

urgency

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^{i} \lambda_i \,\mathrm{latency}(i)\right]$$

knob settings

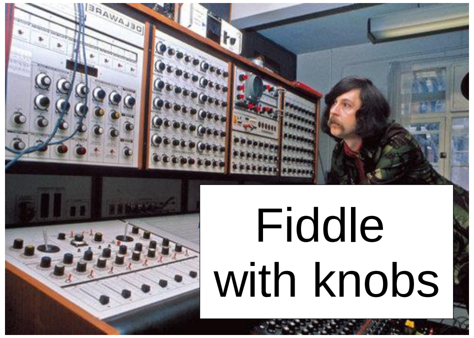
Example knob: eager or lazy updates?
e.g., **dynamic max data structure**

```
% Dyna:
output max= input(I).
```

Max-heap:

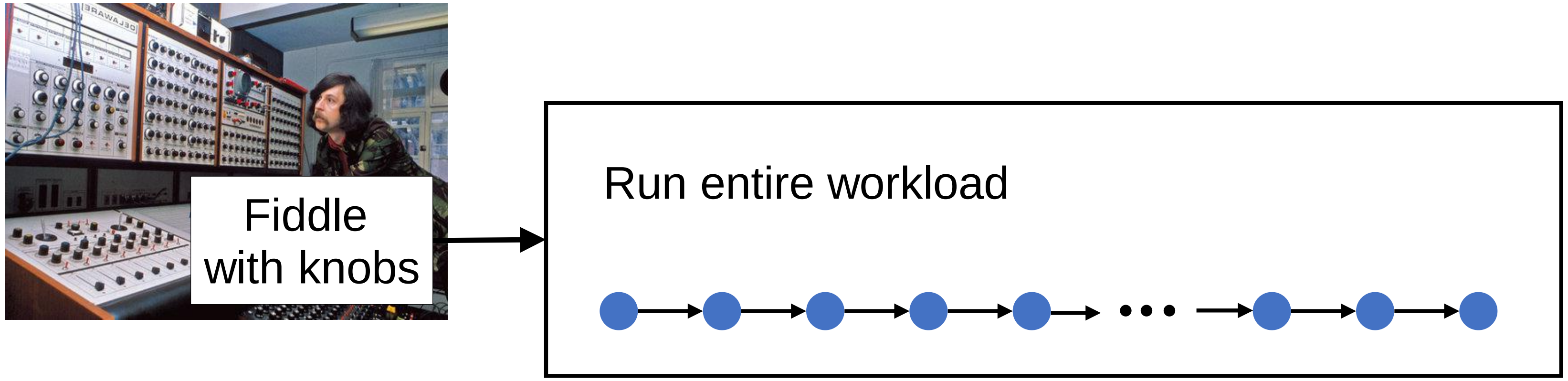
- *O(log n)* per update
- *O(n)* per batch update (“heapify”)

Off-line training



Fiddle
with knobs

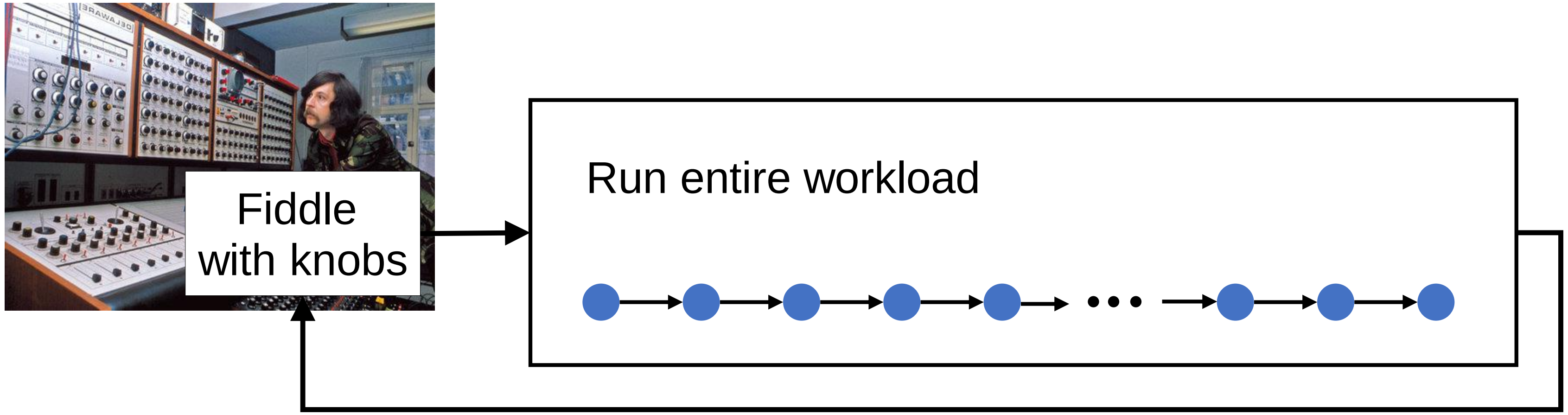
Off-line training



Fiddle with knobs

Run entire workload

Off-line training

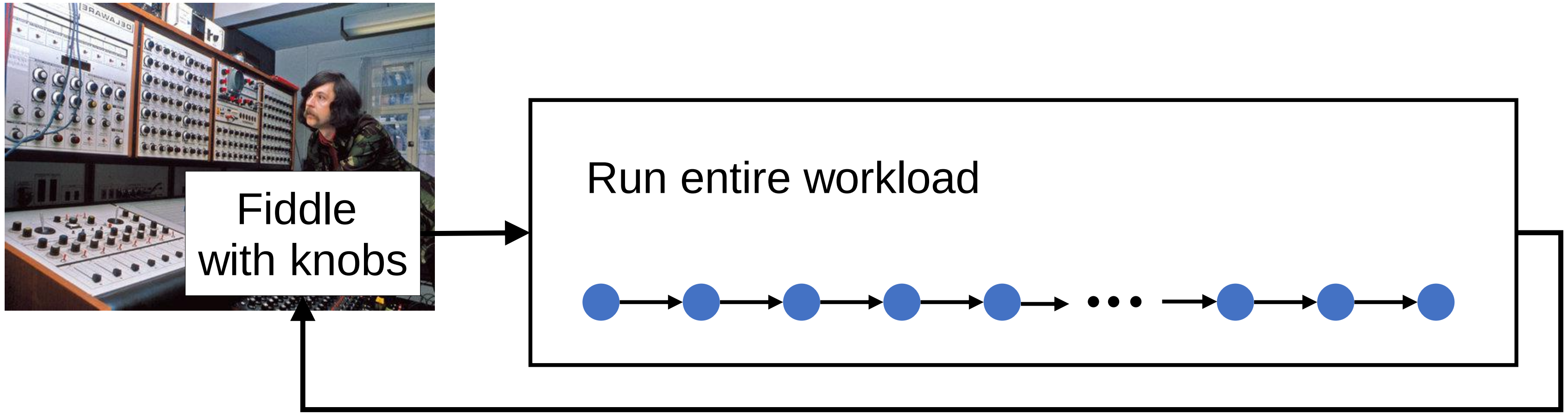


Fiddle with knobs

Run entire workload

Feedback $\rho(\pi)$

Off-line training



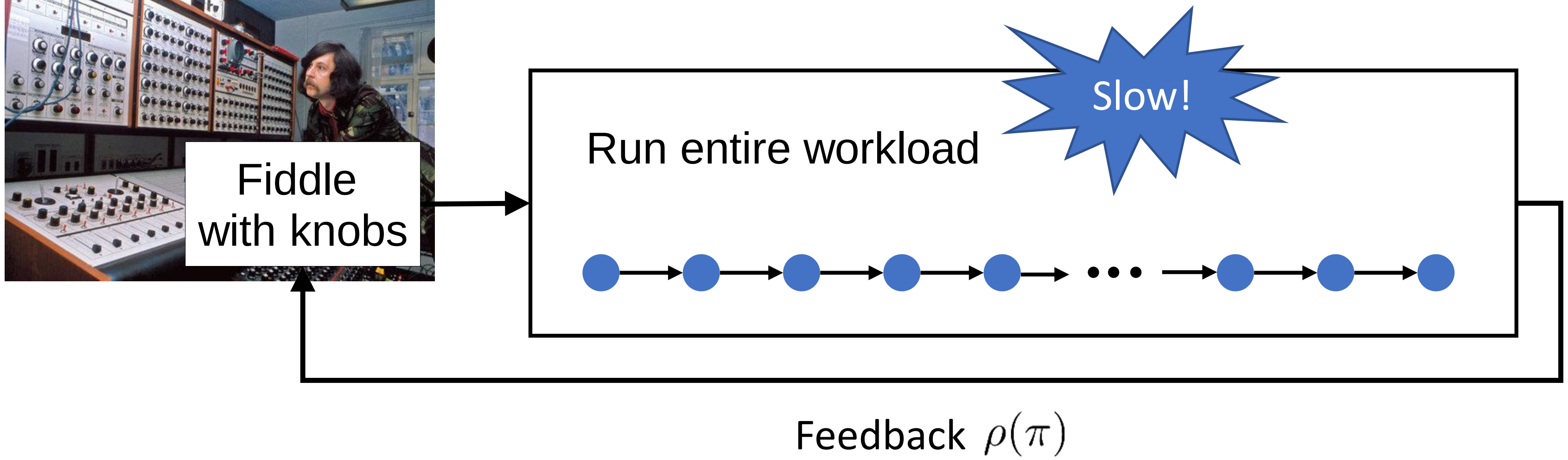
Fiddle with knobs

Run entire workload

Feedback $\rho(\pi)$

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)

Off-line training



Fiddle with knobs

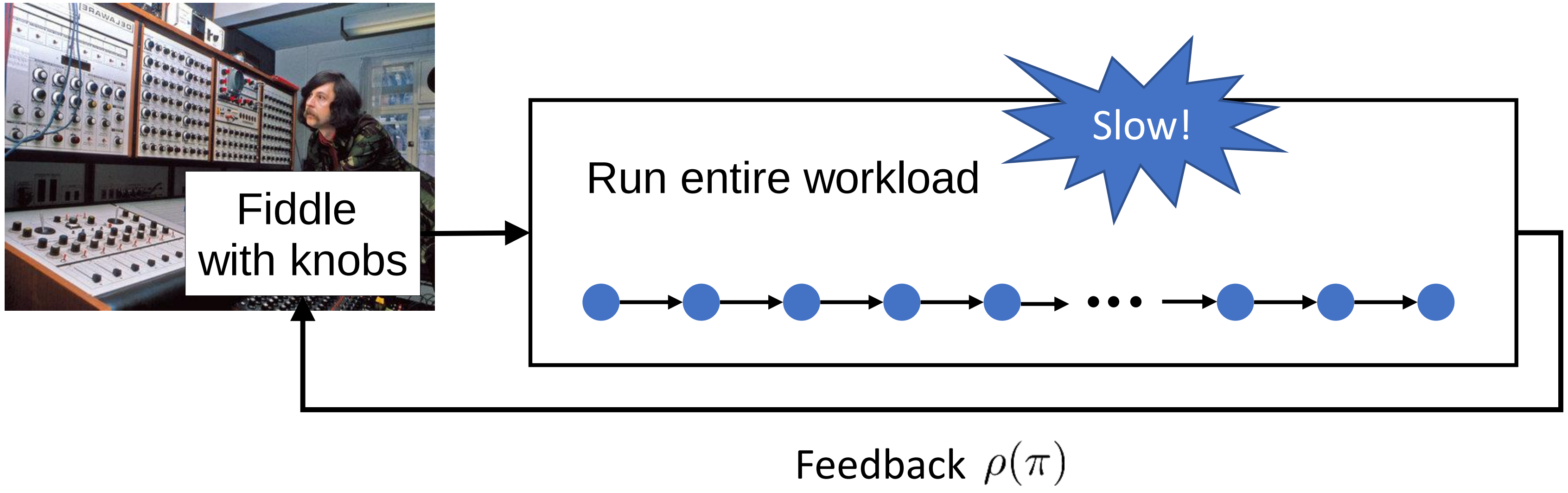
Run entire workload

Slow!

Feedback $\rho(\pi)$

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)

Off-line training



Fiddle with knobs

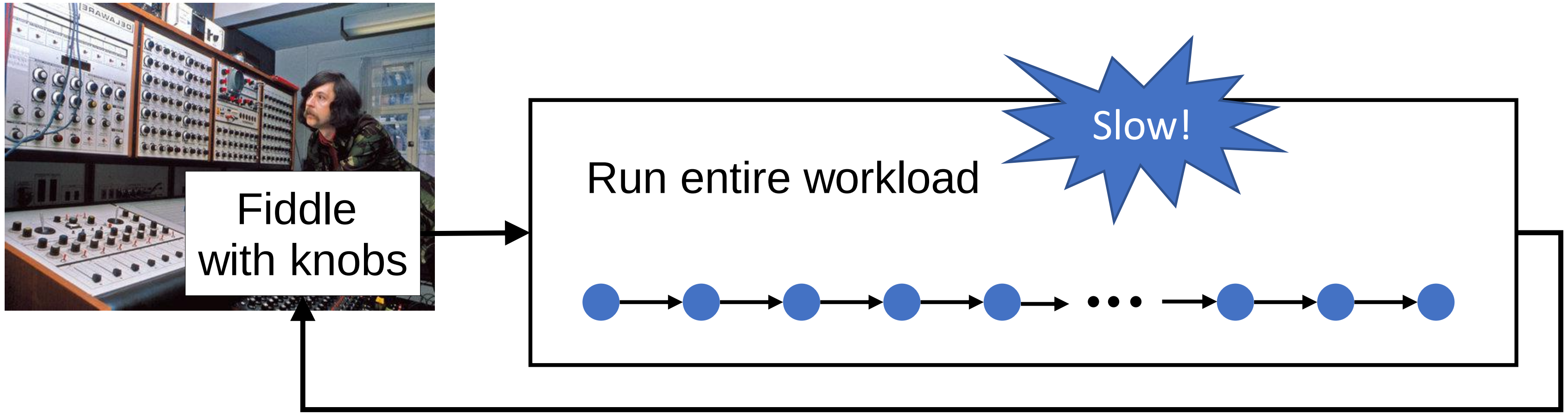
Run entire workload

Slow!

Feedback $\rho(\pi)$

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
- The inefficiency: this loop **explores** one policy per run.

Off-line training



Fiddle with knobs

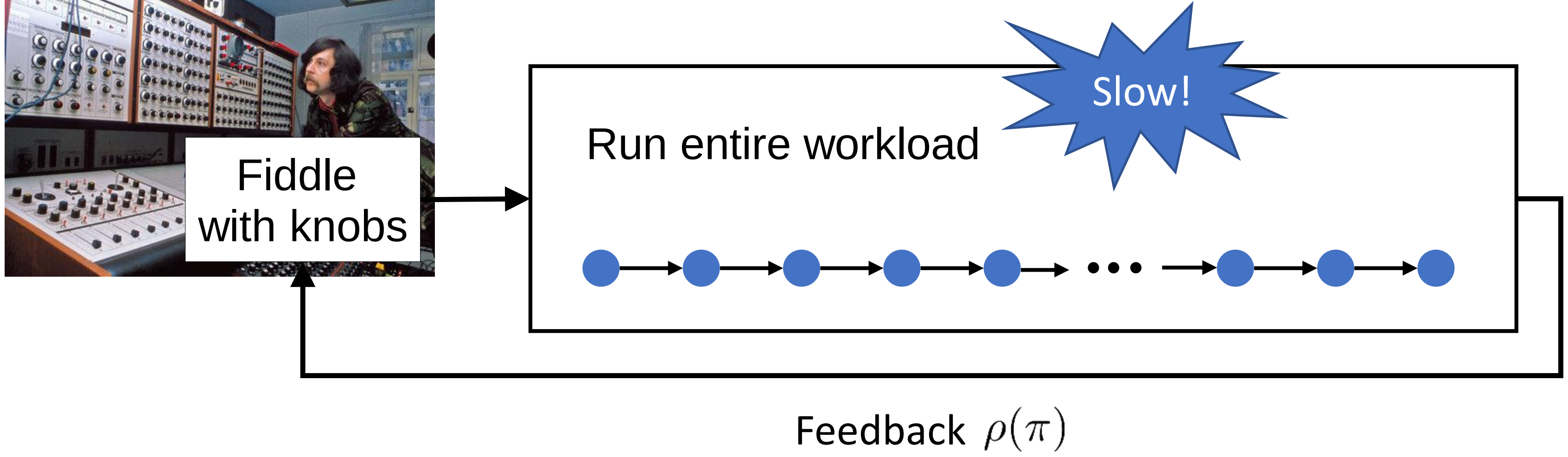
Run entire workload

Slow!

Feedback $\rho(\pi)$

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
- The inefficiency: this loop **explores** one policy per run.
- How do we tighten the loop to **get feedback more often**?

Off-line training



Open up the solver

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
- The inefficiency: this loop **explores** one policy per run.
- How do we tighten the loop to **get feedback more often**?

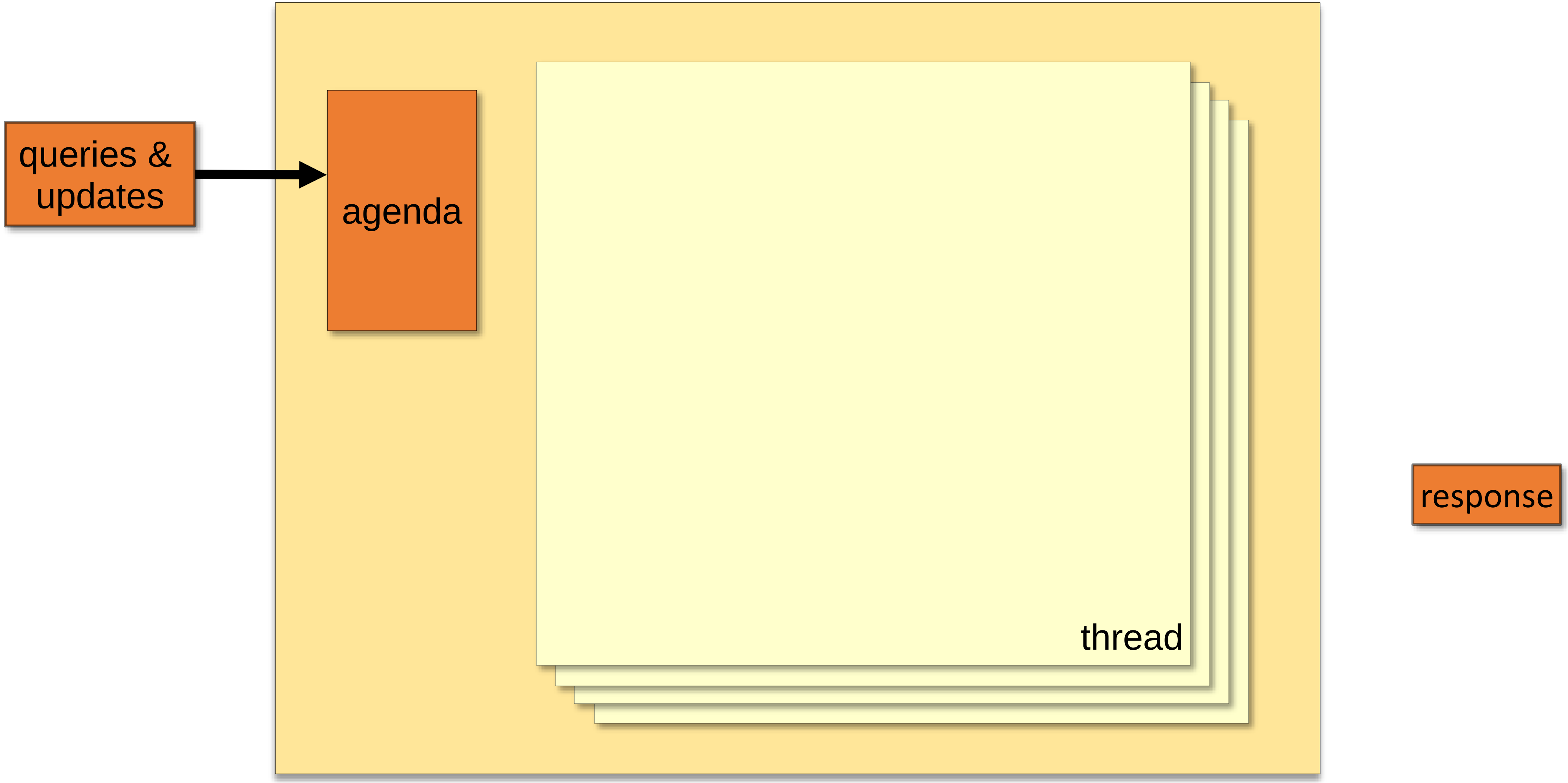
Inside the solver

queries & updates

response

Inside the solver

Inside the solver



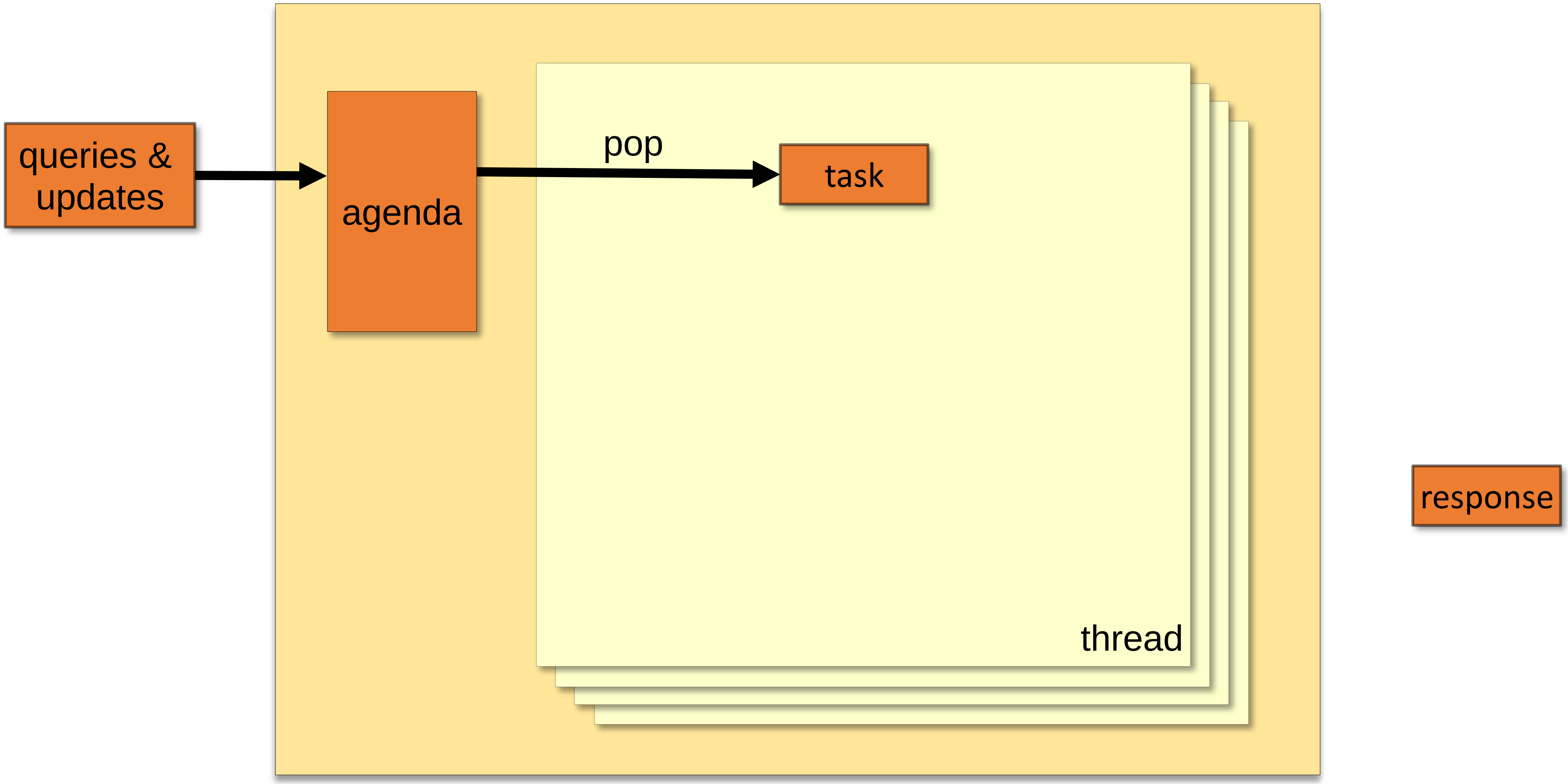
queries & updates

agenda

thread

response

Inside the solver



queries & updates

agenda

pop

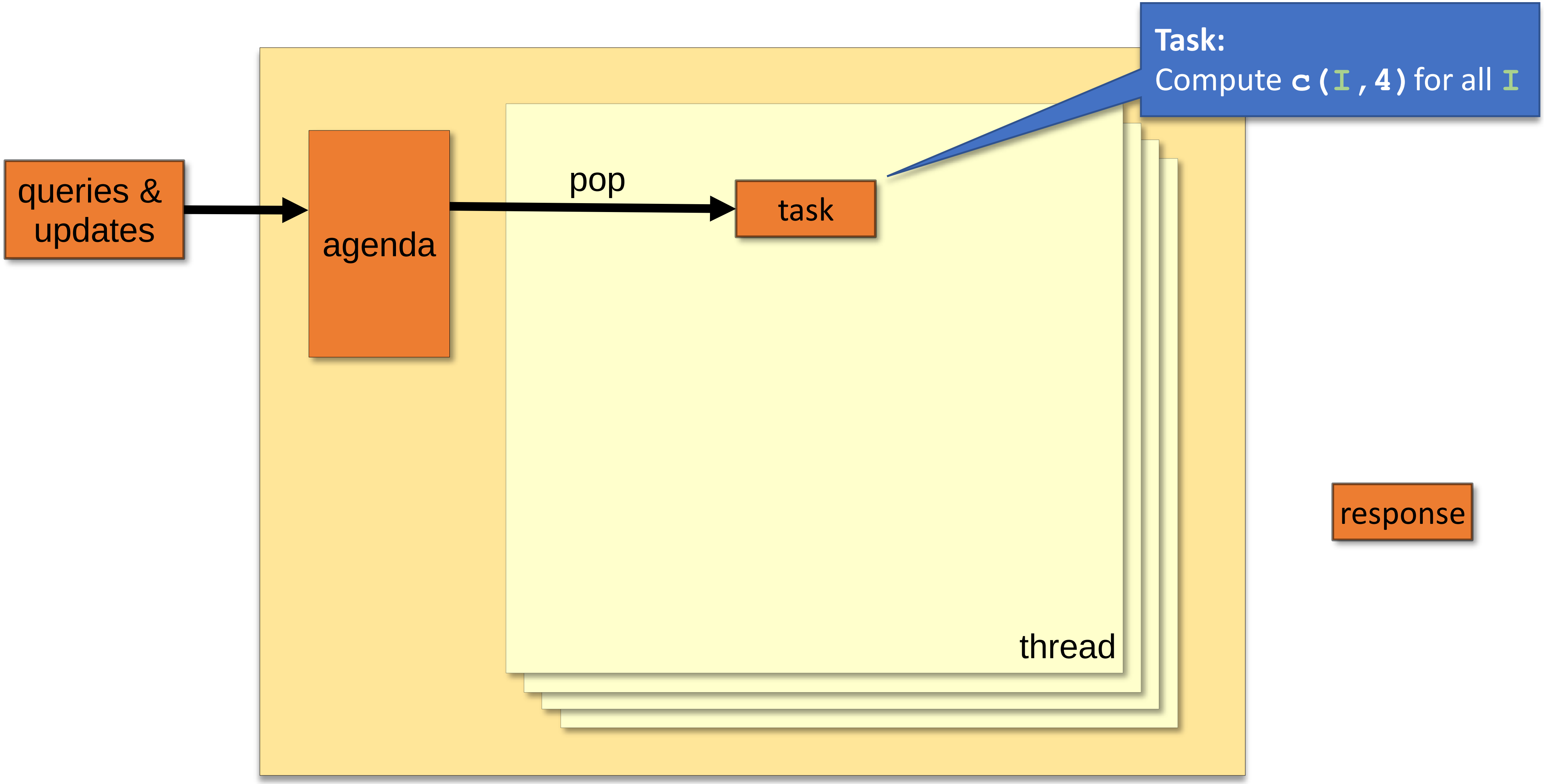
task

thread

response

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

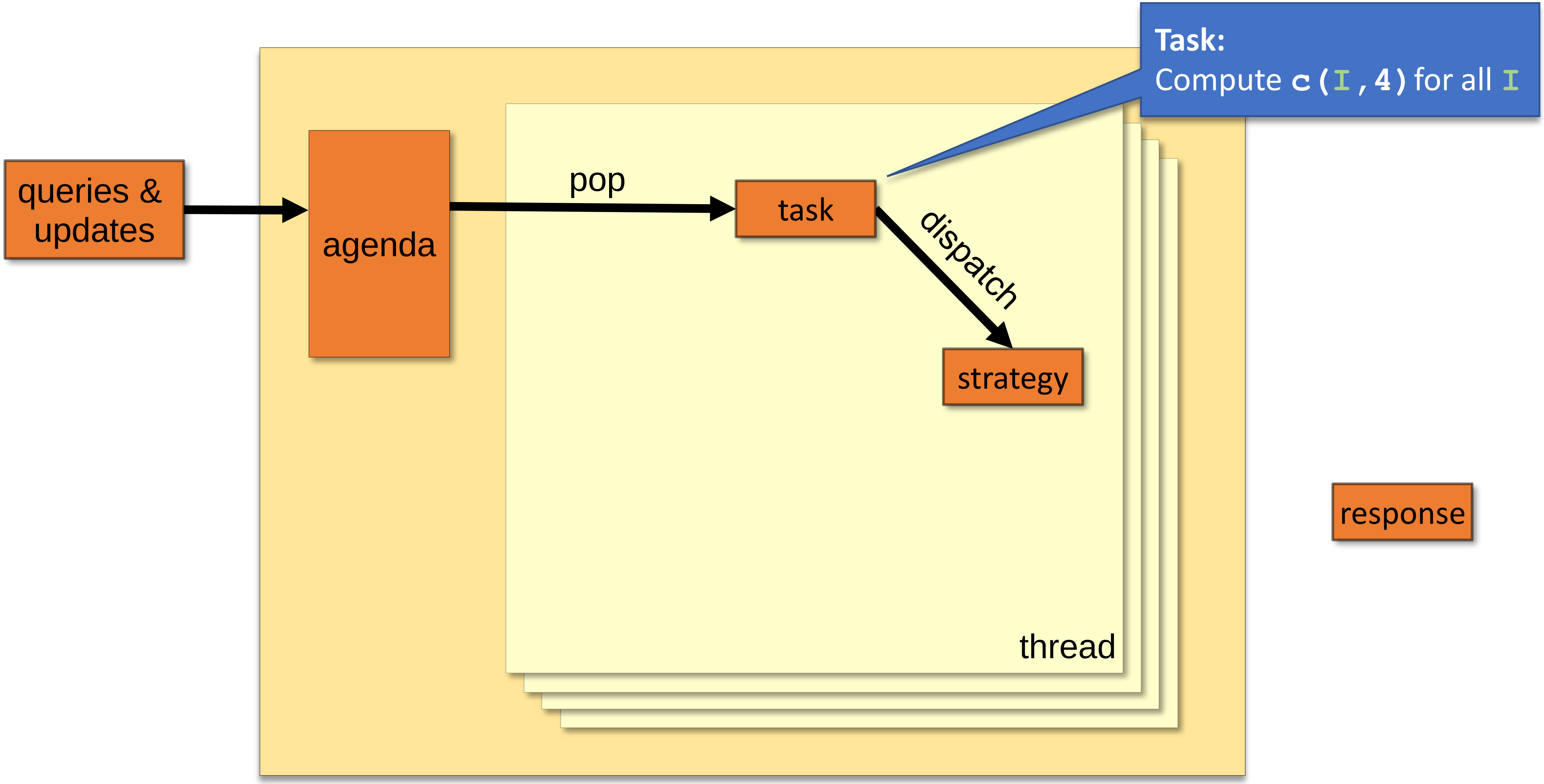


Task:
Compute `c(I,4)` for all `I`

response

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```



Task:
Compute `c(I,4)` for all `I`

queries & updates

agenda

pop

task

dispatch

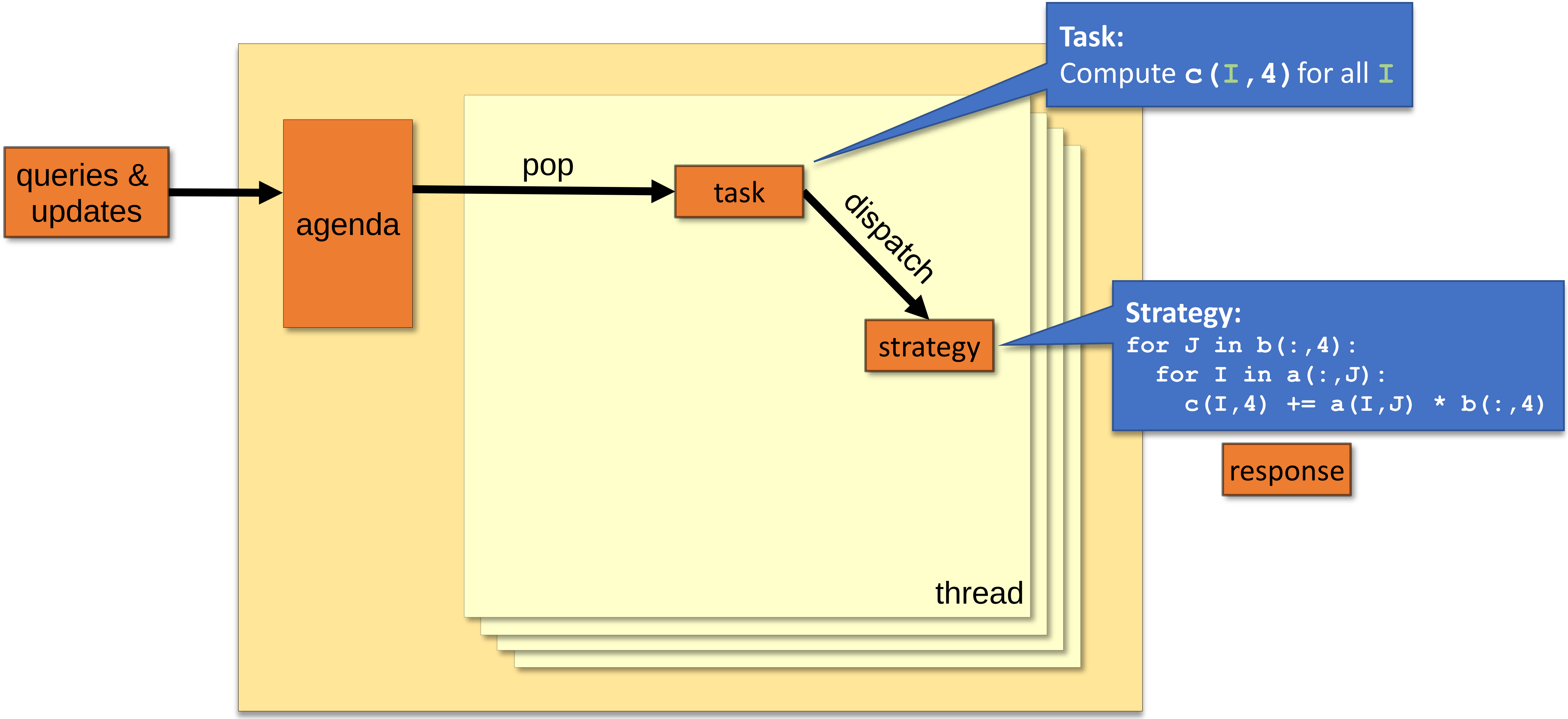
strategy

thread

response

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```



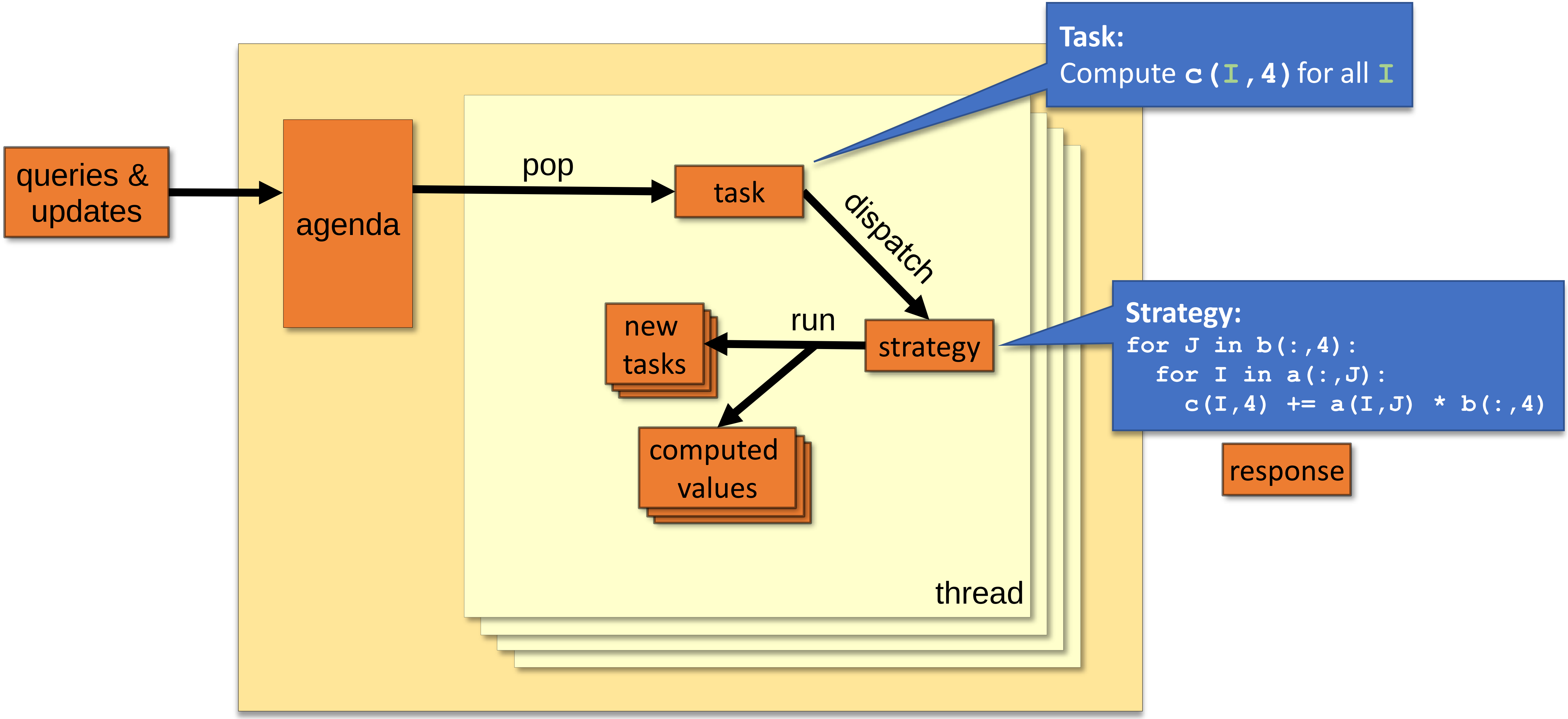
Task:
Compute `c(I,4)` for all `I`

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```



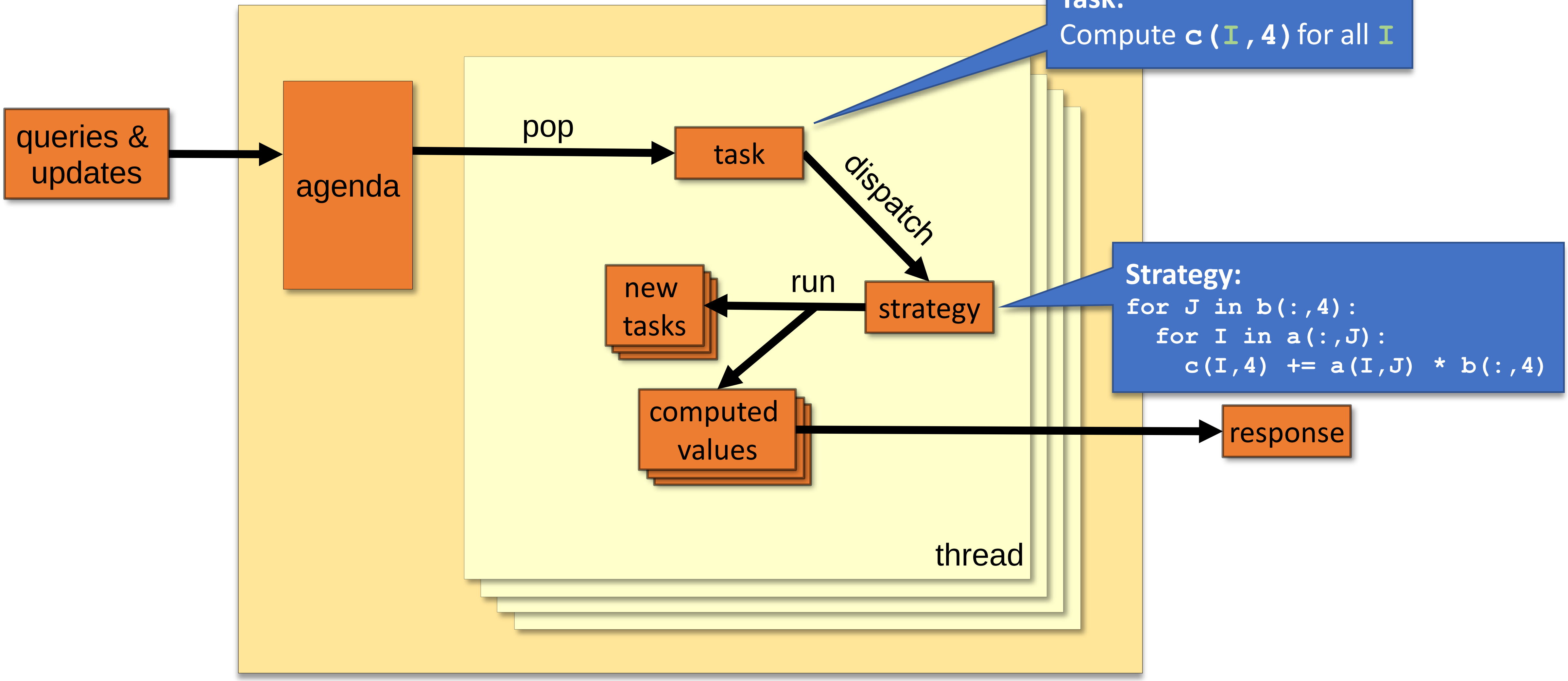
Task:
Compute `c(I,4)` for all `I`

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```



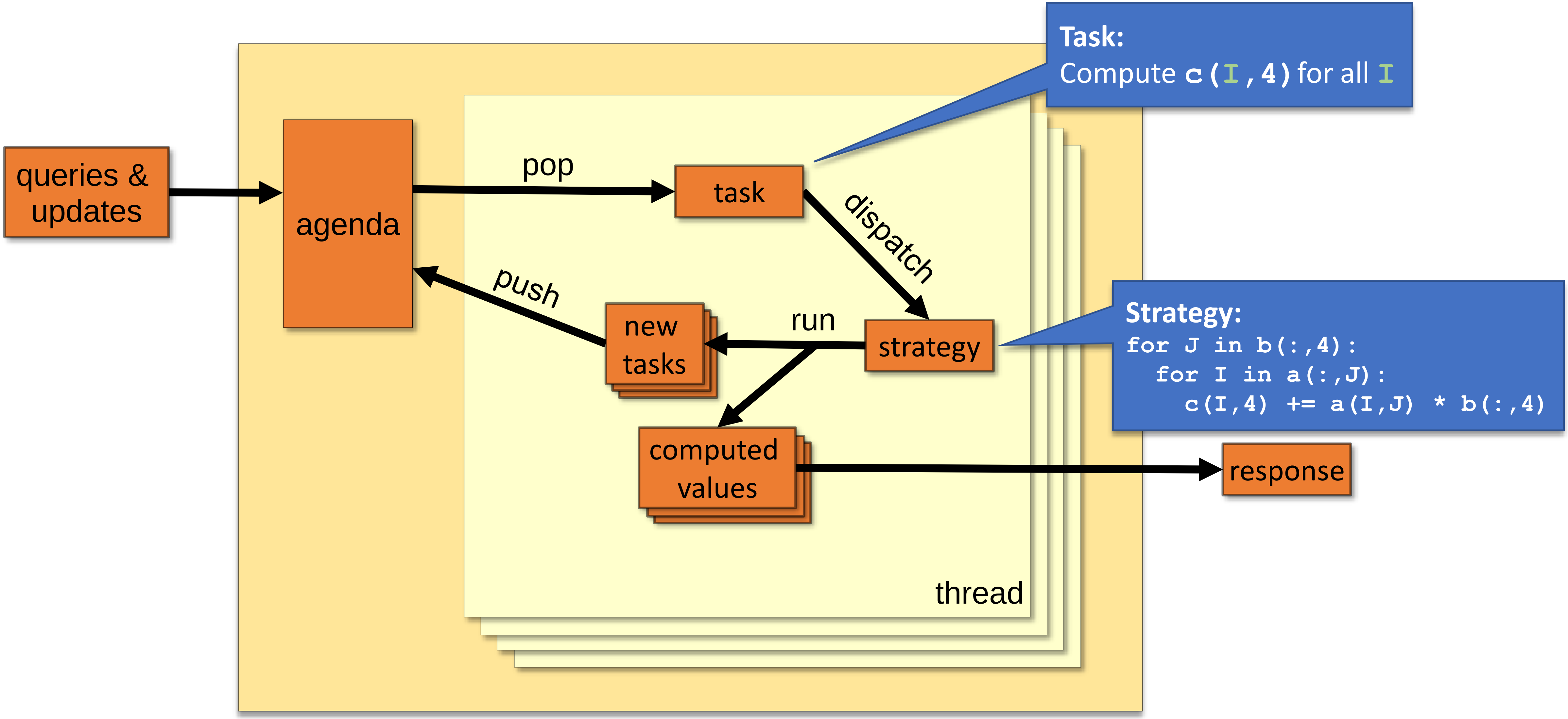
Task:
Compute `c(I,4)` for all `I`

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```



queries & updates

agenda

pop

task

dispatch

strategy

run

new tasks

computed values

push

thread

response

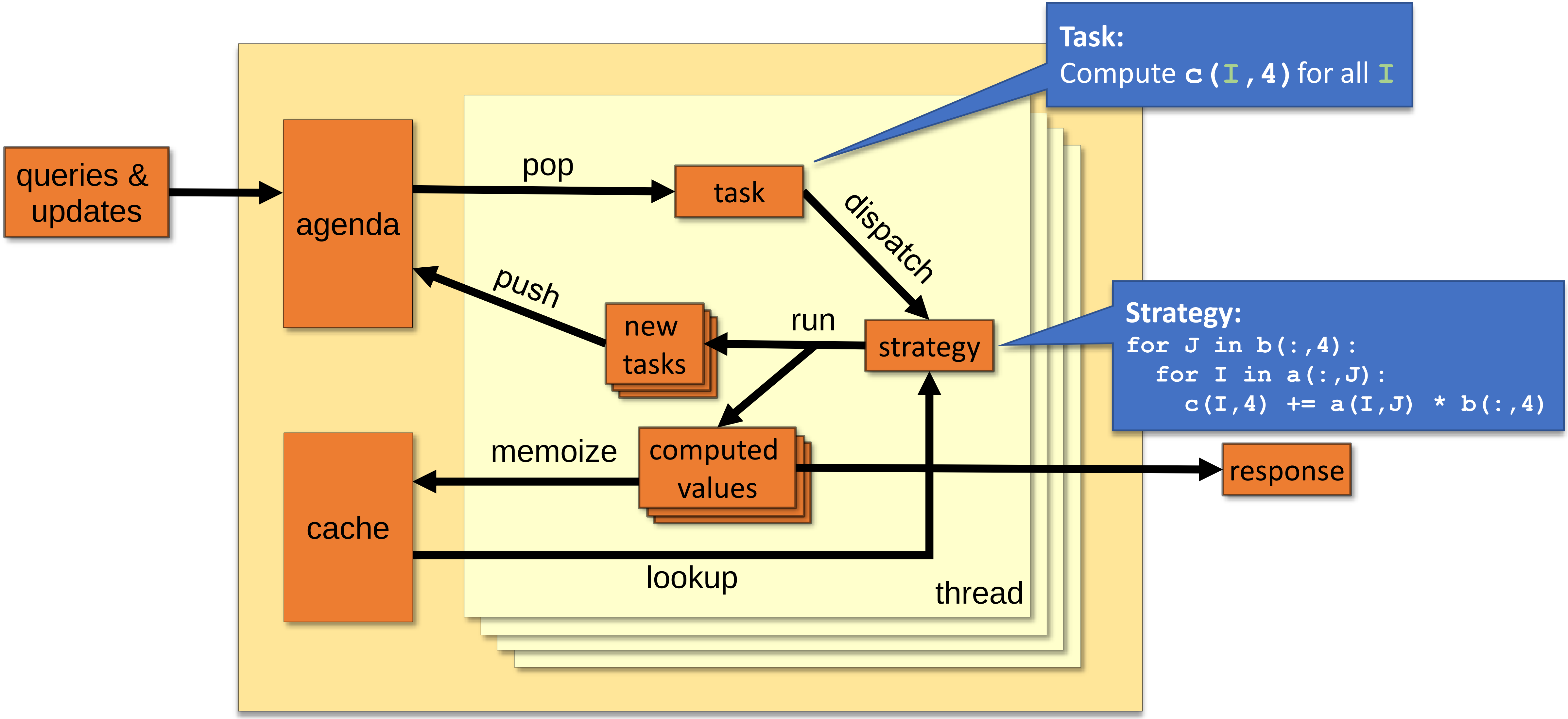
Task:
Compute `c(I,4)` for all `I`

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```



Task:
Compute `c(I,4)` for all `I`

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```

Inside the solver

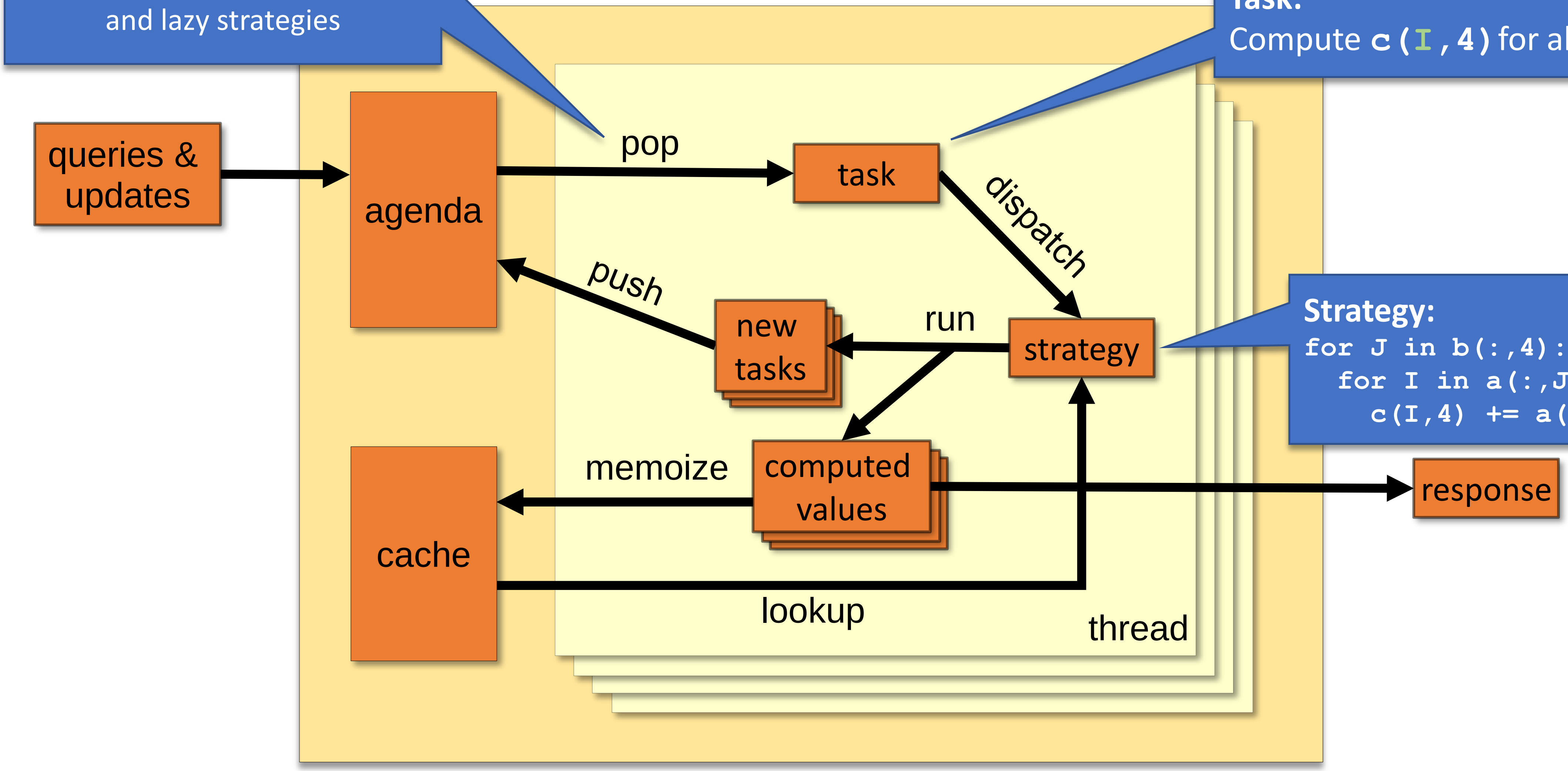
```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

Task:
Compute `c(I,4)` for all `I`

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```



Inside the solver

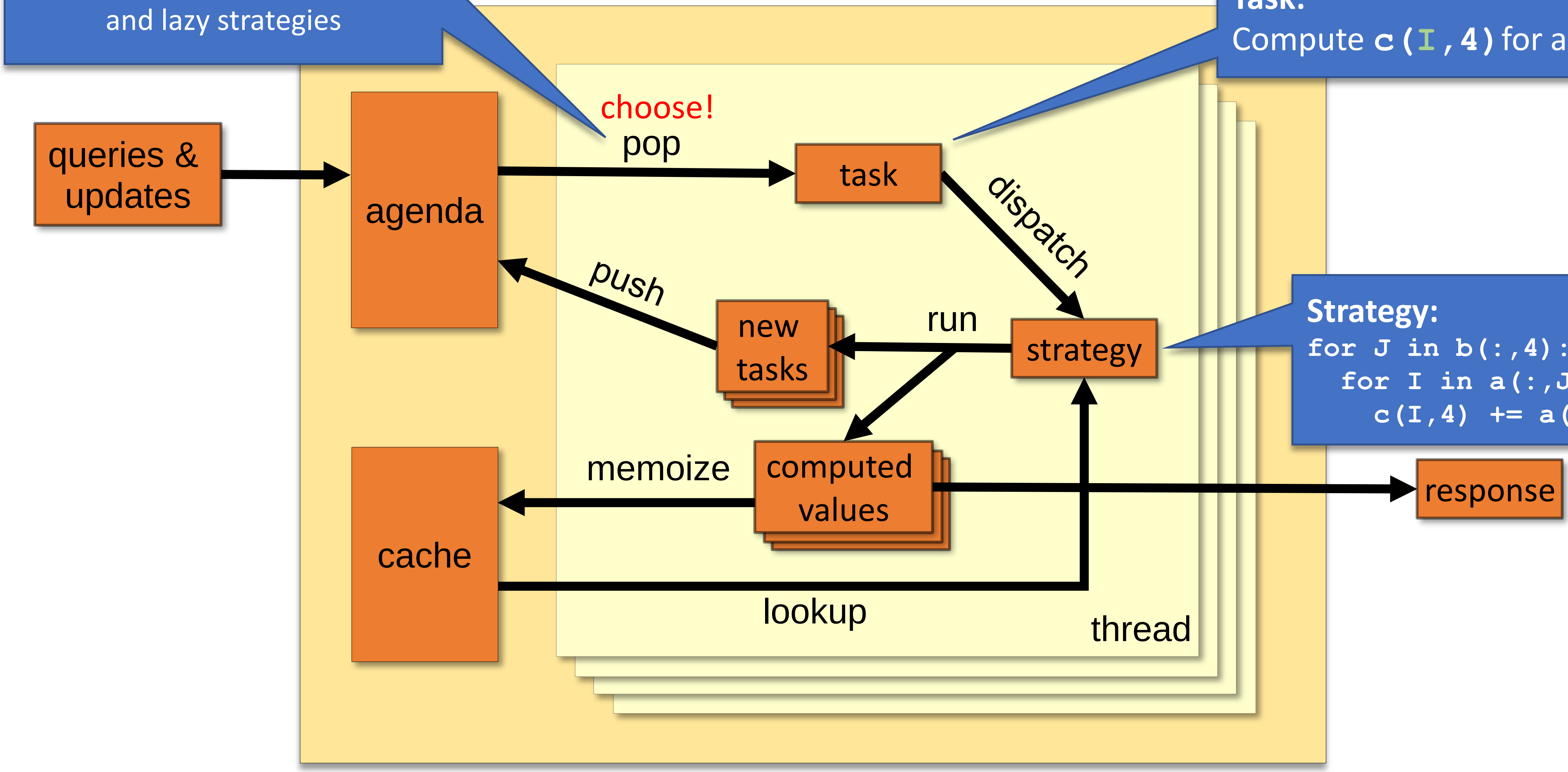
```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

Task:
Compute `c(I,4)` for all `I`

Strategy:

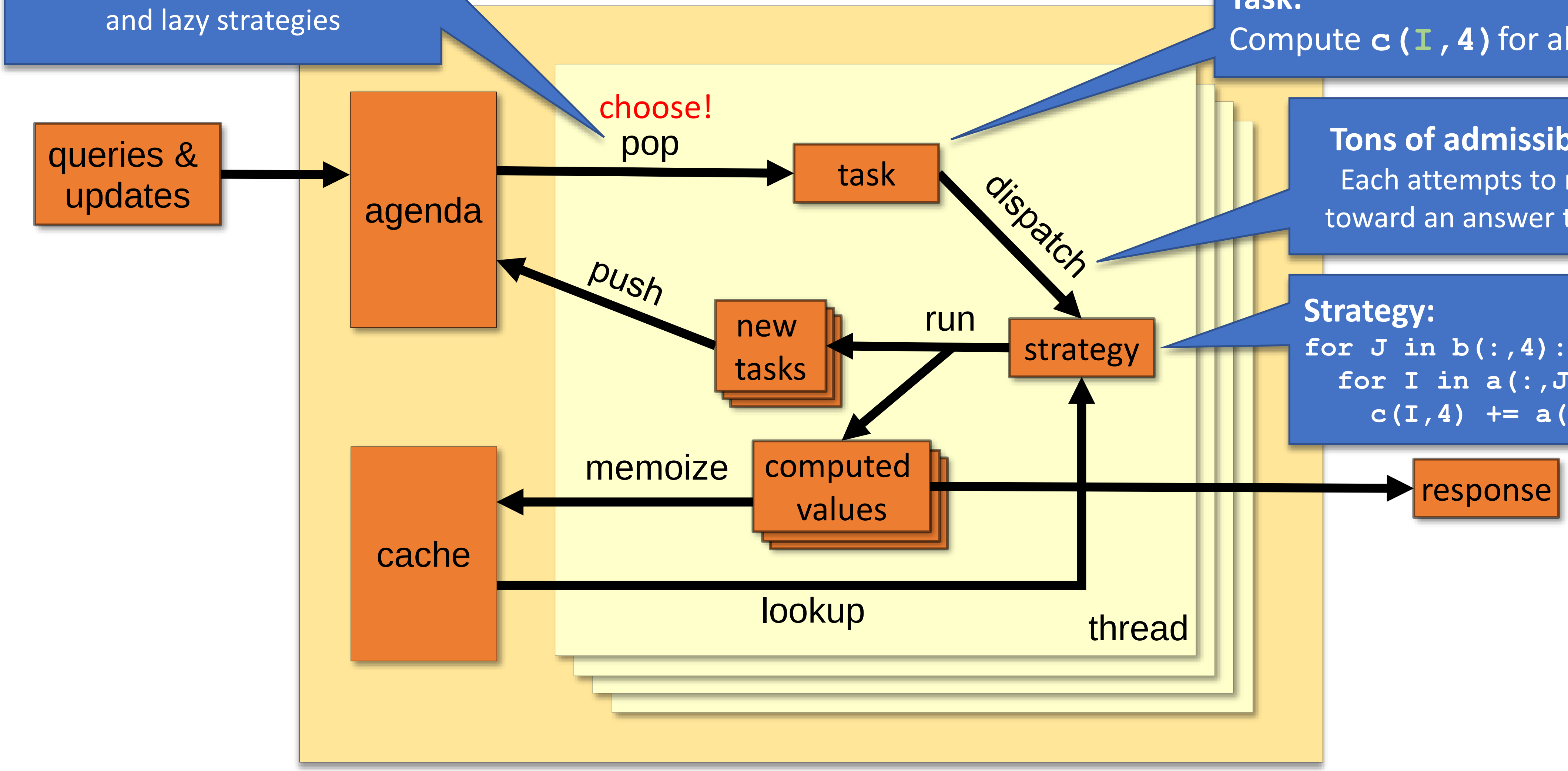
```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```



Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies



Task:
Compute `c(I,4)` for all `I`

Tons of admissible strategies.
Each attempts to make progress toward an answer to open queries

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

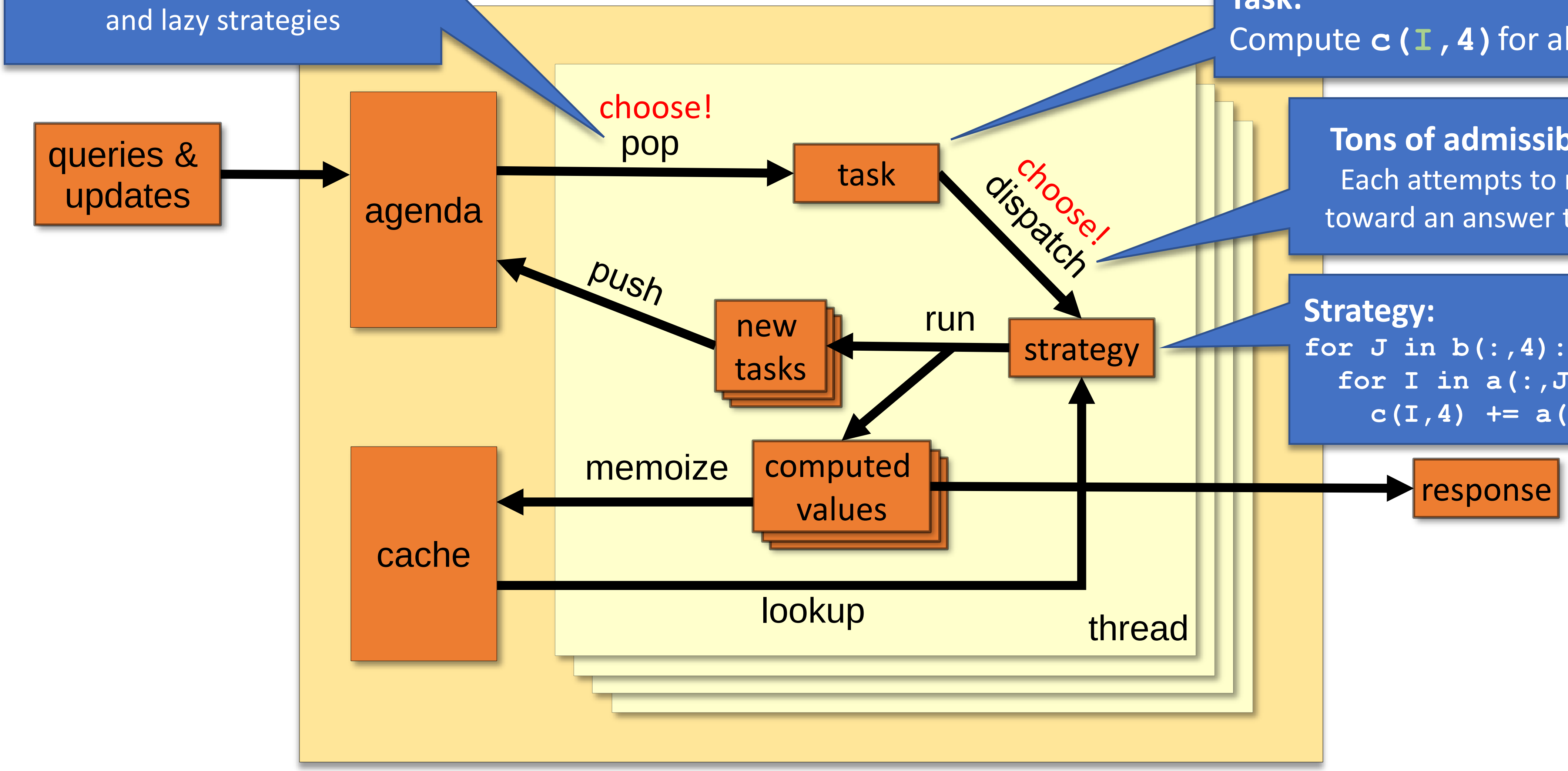
Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

Task:
Compute `c(I,4)` for all `I`

Tons of admissible strategies.
Each attempts to make progress toward an answer to open queries

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```



queries & updates

agenda

choose!
pop

task

choose!
dispatch

strategy

run

new tasks

push

computed values

memoize

cache

lookup

response

thread

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

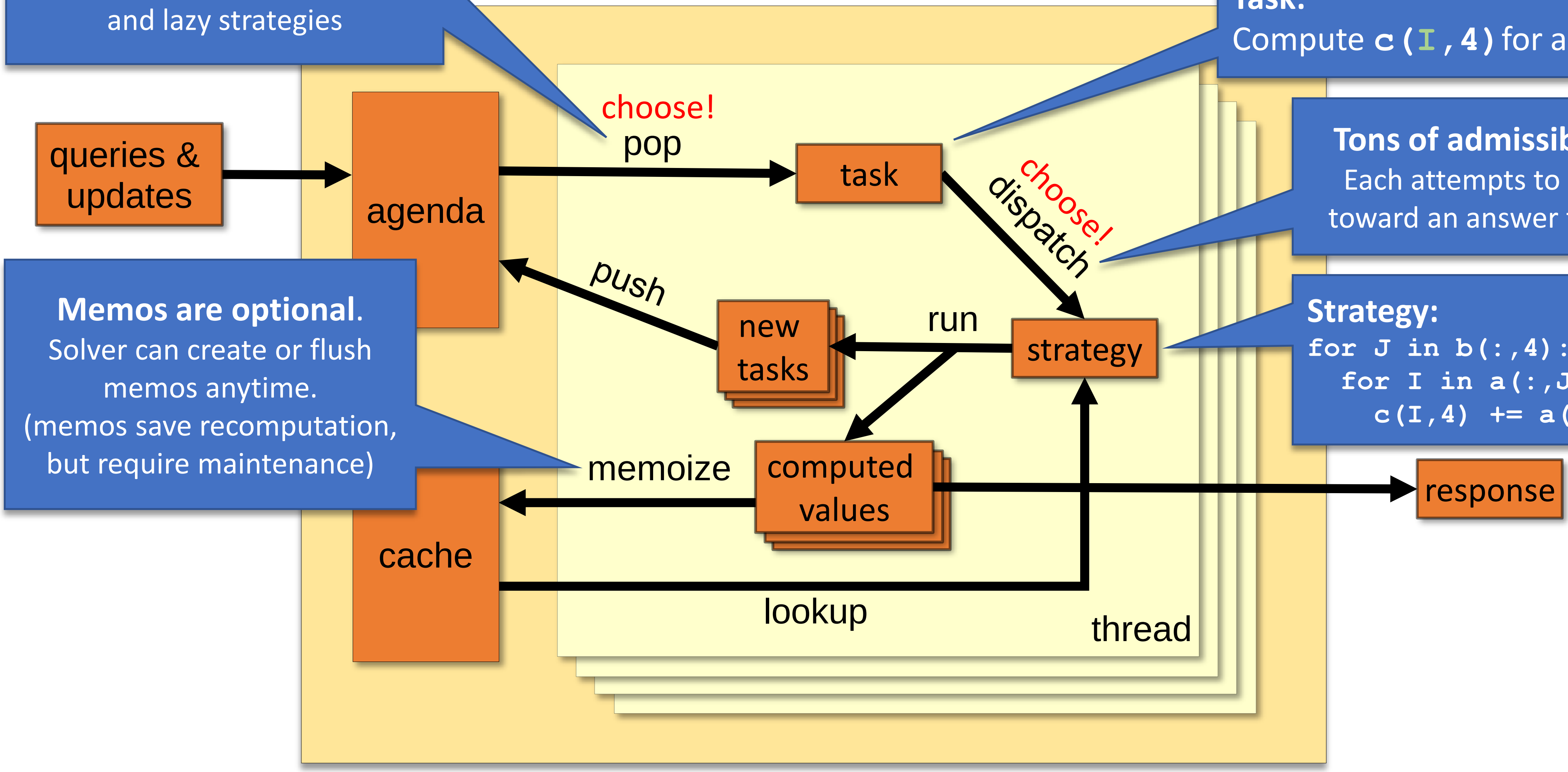
Task:
Compute `c(I,4)` for all `I`

Tons of admissible strategies.
Each attempts to make progress toward an answer to open queries

Memos are optional.
Solver can create or flush memos anytime.
(memos save recomputation, but require maintenance)

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```



queries & updates

agenda

choose!
pop

task

choose!
dispatch

strategy

run

new tasks

push

computed values

memoize

cache

lookup

thread

response

Inside the solver

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

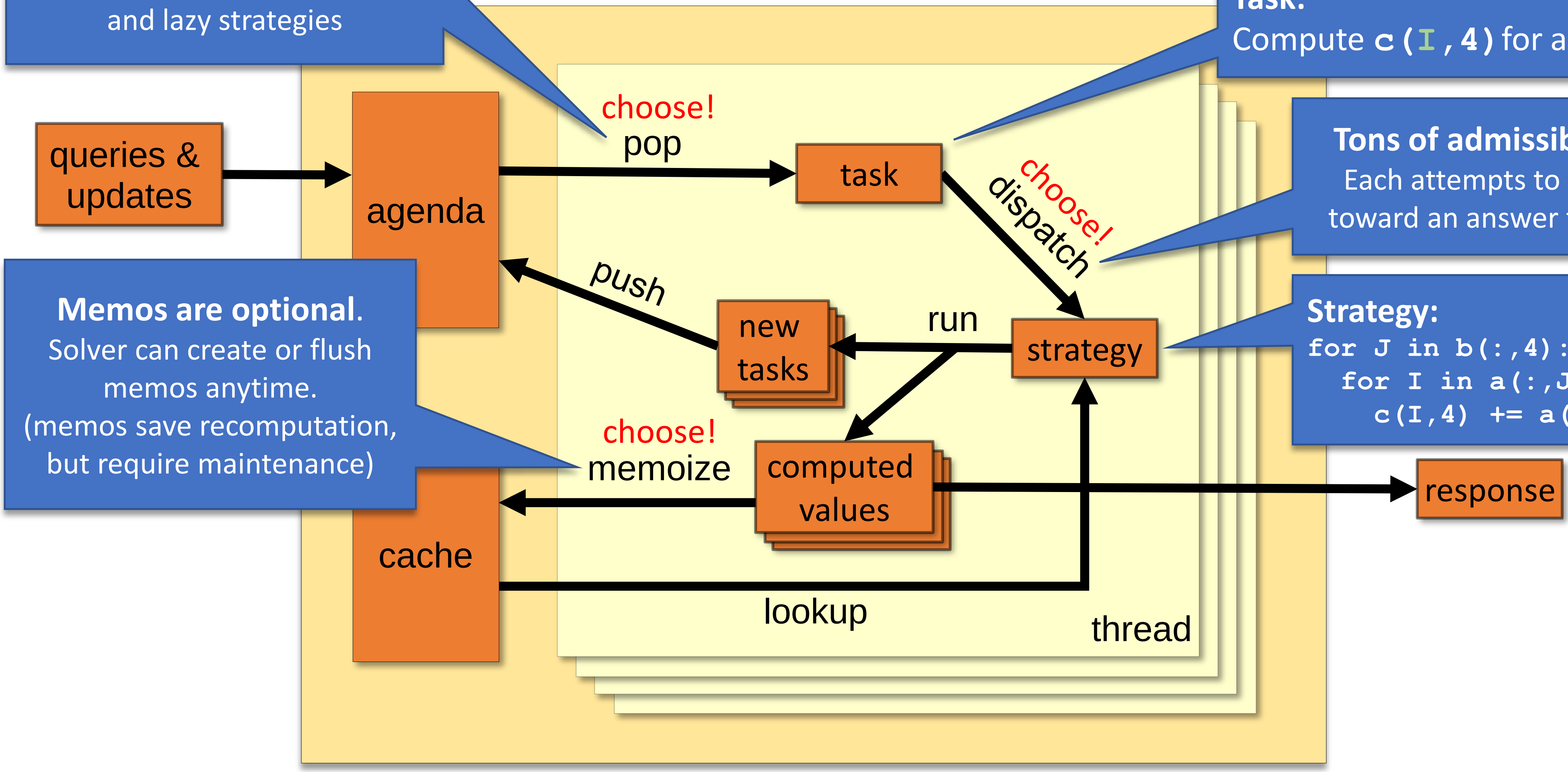
Task:
Compute `c(I,4)` for all `I`

Tons of admissible strategies.
Each attempts to make progress toward an answer to open queries

Memos are optional.
Solver can create or flush memos anytime.
(memos save recomputation, but require maintenance)

Strategy:

```
for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)
```



queries & updates

agenda

choose!
pop

task

choose!
dispatch

strategy

run

new tasks

push

computed values

choose!
memoize

cache

lookup

response

thread

Strategy options

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
    - Join strategies

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
    - Join strategies
    - Forward vs. backward chaining (update-driven vs. query-driven)
    - Dynamically identify unnecessary computation

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
    - Join strategies
    - Forward vs. backward chaining (update-driven vs. query-driven)
    - Dynamically identify unnecessary computation
        - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
    - Join strategies
    - Forward vs. backward chaining (update-driven vs. query-driven)
    - Dynamically identify unnecessary computation
        - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
    - Static or dynamic batching (consolidating similar tasks, including GPU)

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
    - Join strategies
    - Forward vs. backward chaining (update-driven vs. query-driven)
    - Dynamically identify unnecessary computation
        - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
    - Static or dynamic batching (consolidating similar tasks, including GPU)
    - Inlining depth (consolidating caller-callee)

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
  - Static or dynamic batching (consolidating similar tasks, including GPU)
  - Inlining depth (consolidating caller-callee)
- **Storage**

Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
  - Static or dynamic batching (consolidating similar tasks, including GPU)
  - Inlining depth (consolidating caller-callee)
- **Storage**
  - Memoization policy; choose low-level data structures

The Dyna solver

The Dyna solver

- Lots of challenges in defining this giant space while preserving correctness

The Dyna solver

- Lots of challenges in defining this giant space while preserving correctness
- Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.

The Dyna solver

- Lots of challenges in defining this giant space while preserving correctness
- Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
- Further reading
    - Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
    - Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)

The Dyna solver

- Lots of challenges in defining this giant space while preserving correctness
- Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
- Further reading
    - Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
    - Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)
- Lots of progress to come!

The Dyna solver

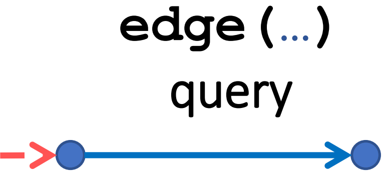
- Lots of challenges in defining this giant space while preserving correctness
- Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
- Further reading
  - Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
  - Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)
- Lots of progress to come!
  - **Nathaniel Wesley Filardo** is tying up many loose ends in his thesis (September 2017)

The Dyna solver

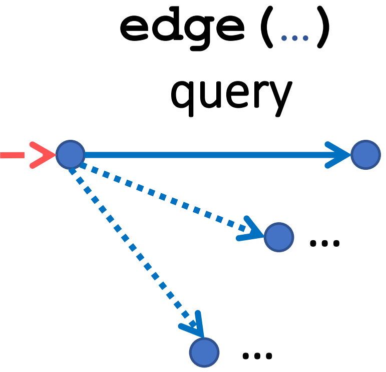
- Lots of challenges in defining this giant space while preserving correctness
- Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
- Further reading
  - Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
  - Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)
- Lots of progress to come!
  - **Nathaniel Wesley Filardo** is tying up many loose ends in his thesis (September 2017)
  - **He's on the job market!**

Sequential choices at runtime (some stochastic)

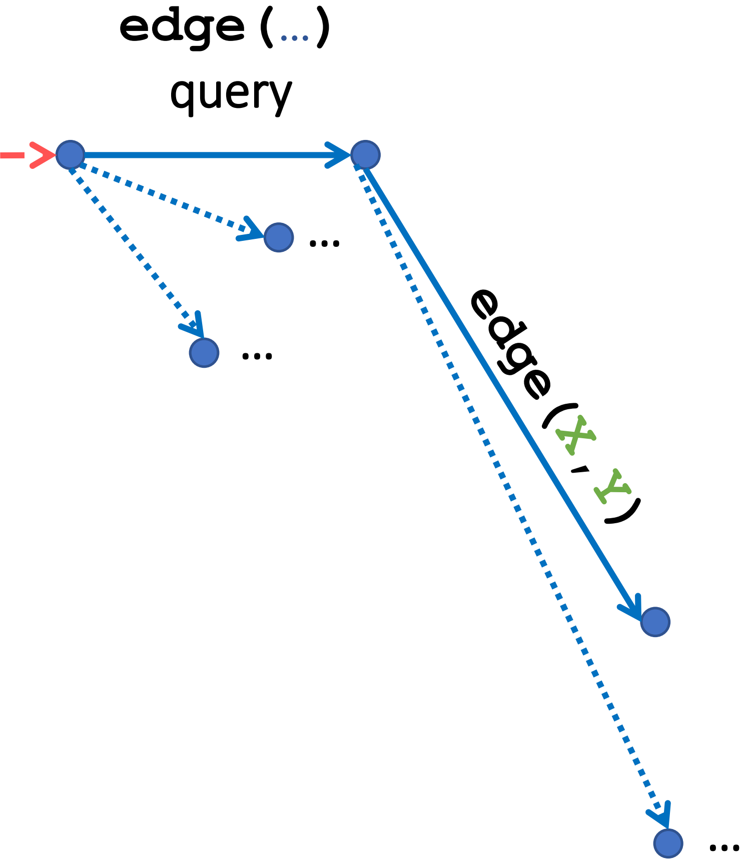
Sequential choices at runtime (some stochastic)



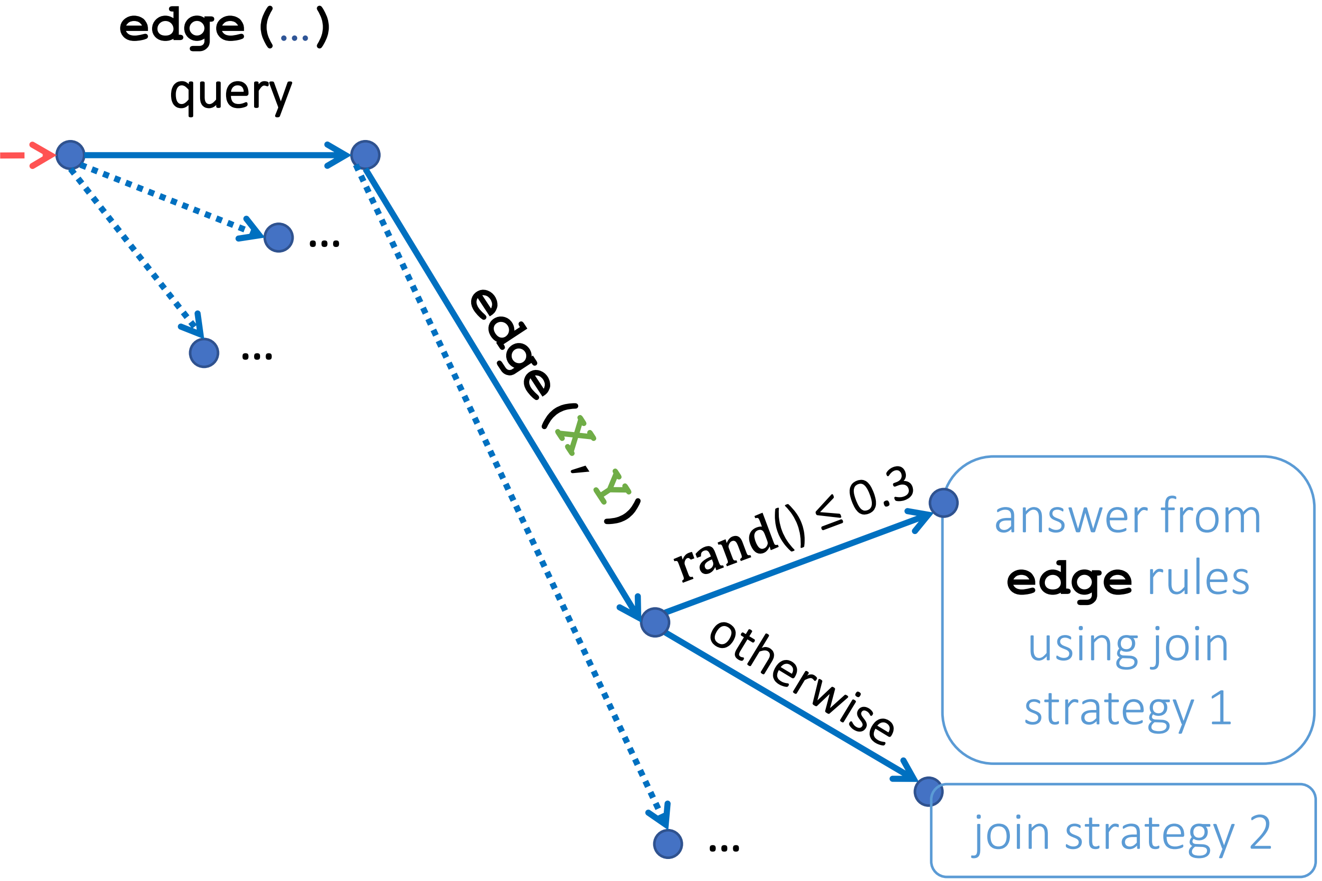
Sequential choices at runtime (some stochastic)



Sequential choices at runtime (some stochastic)



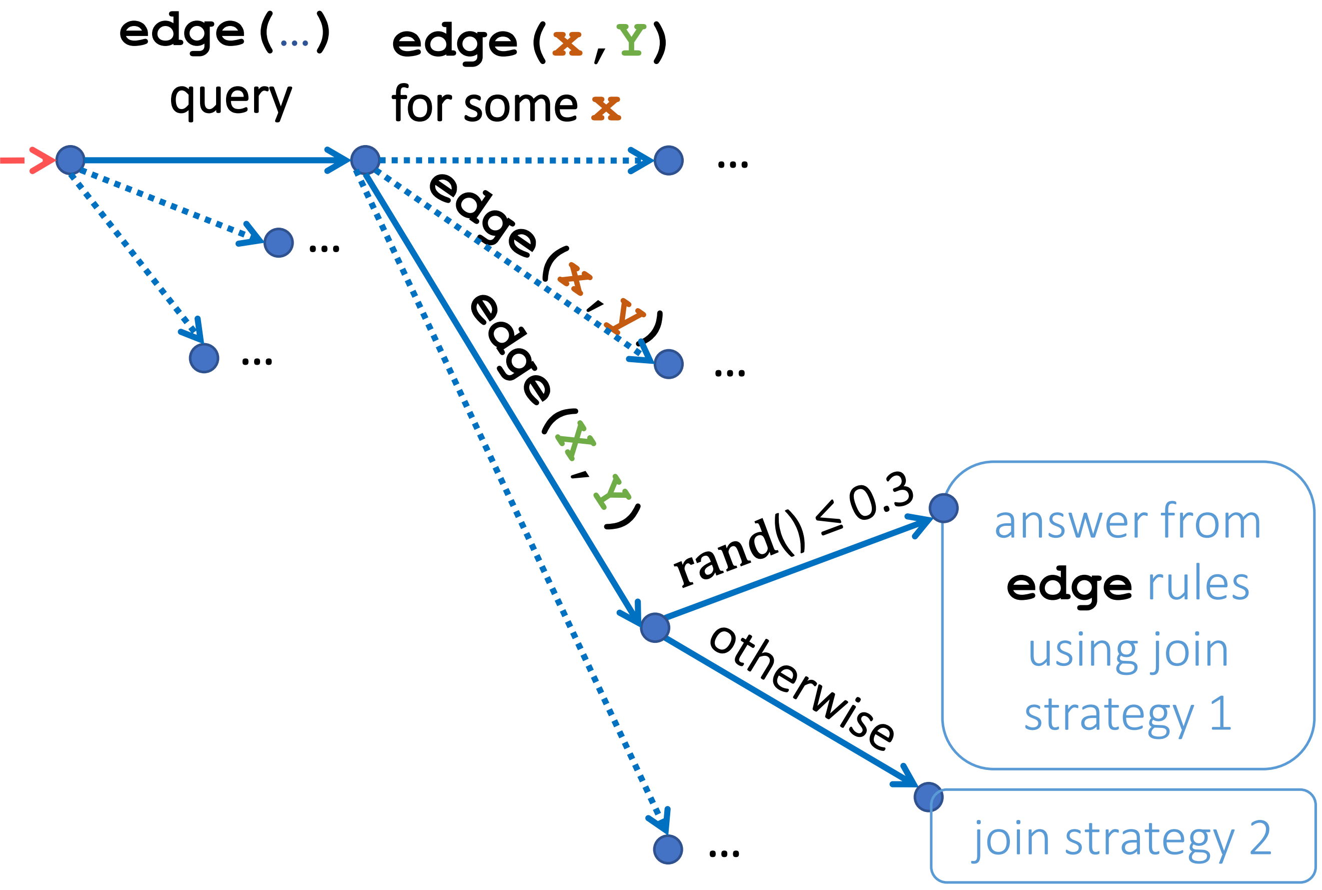
Sequential choices at runtime (some stochastic)



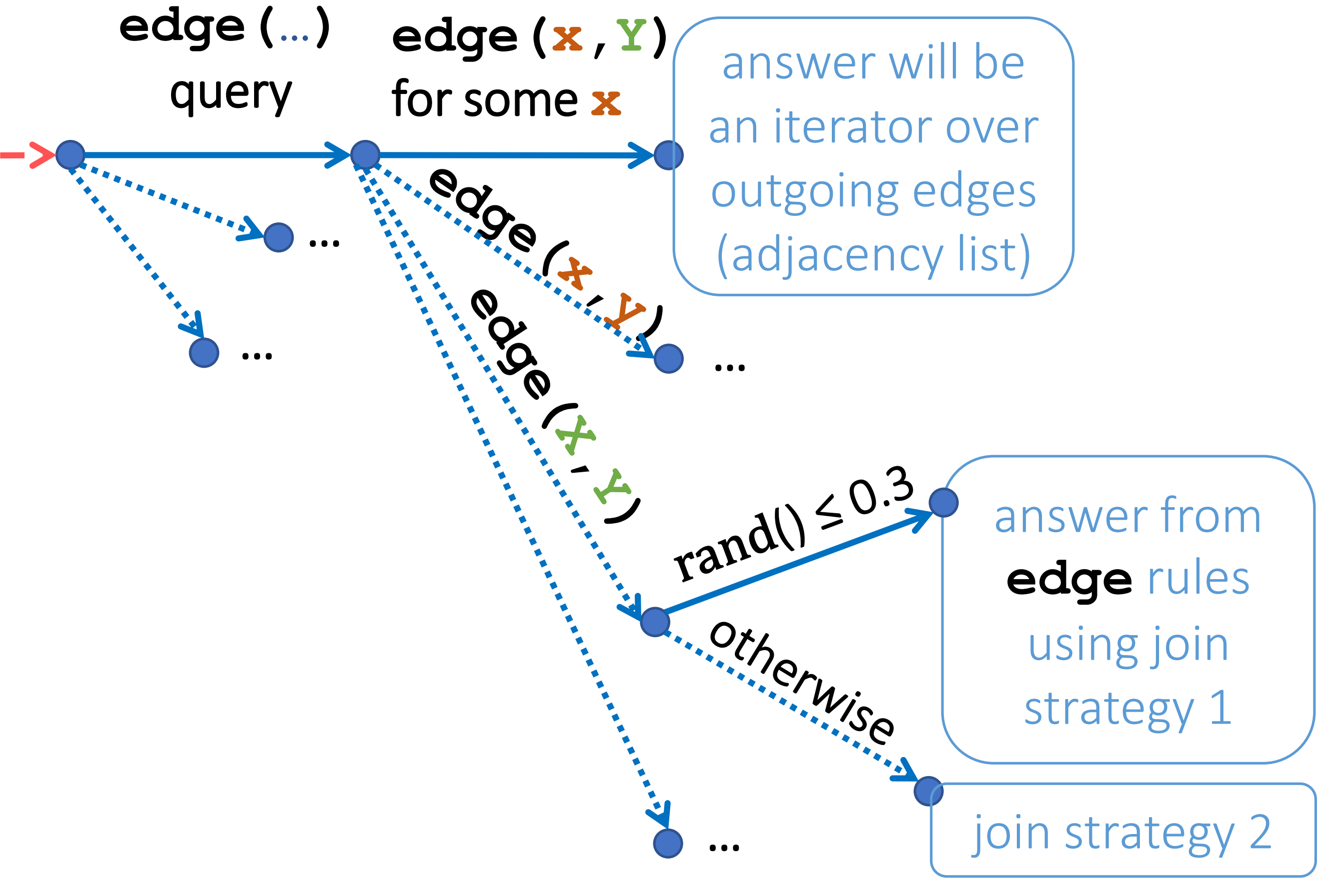
answer from **edge** rules using join strategy 1

join strategy 2

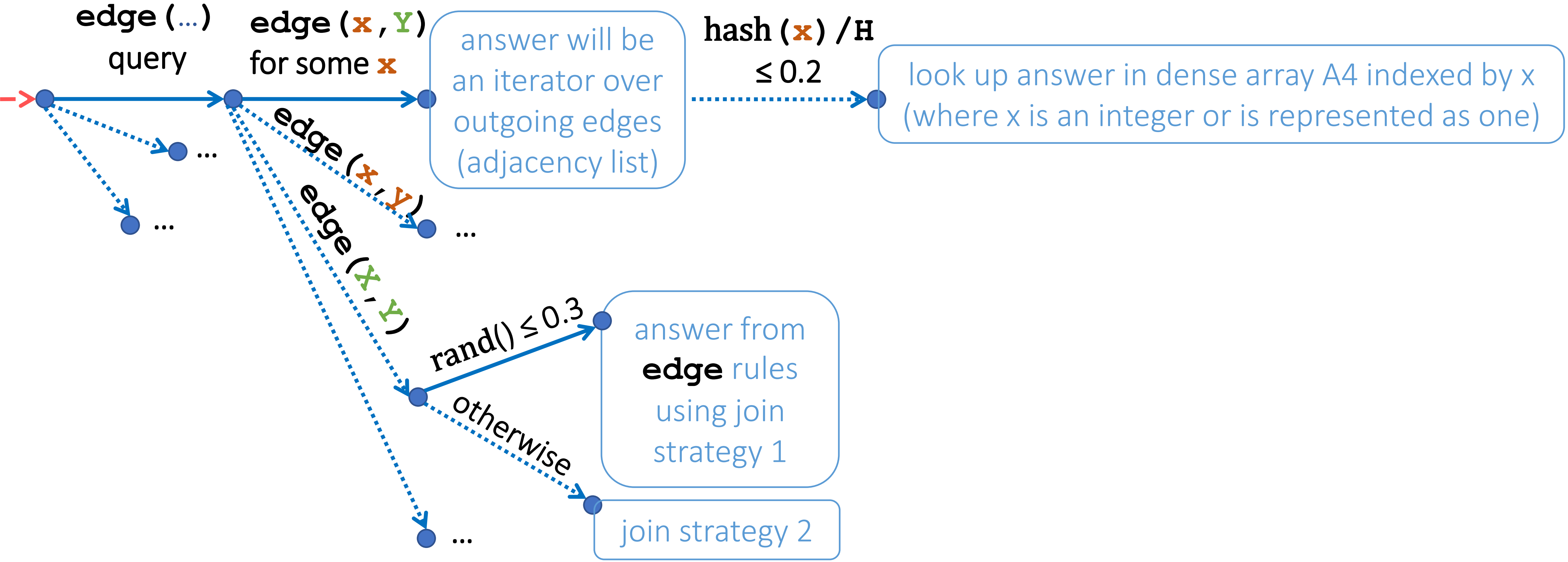
Sequential choices at runtime (some stochastic)



Sequential choices at runtime (some stochastic)



Sequential choices at runtime (some stochastic)



edge(...) query

edge(x,Y) for some x

answer will be an iterator over outgoing edges (adjacency list)

hash(x)/H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

edge(x,Y)

edge(X,Y)

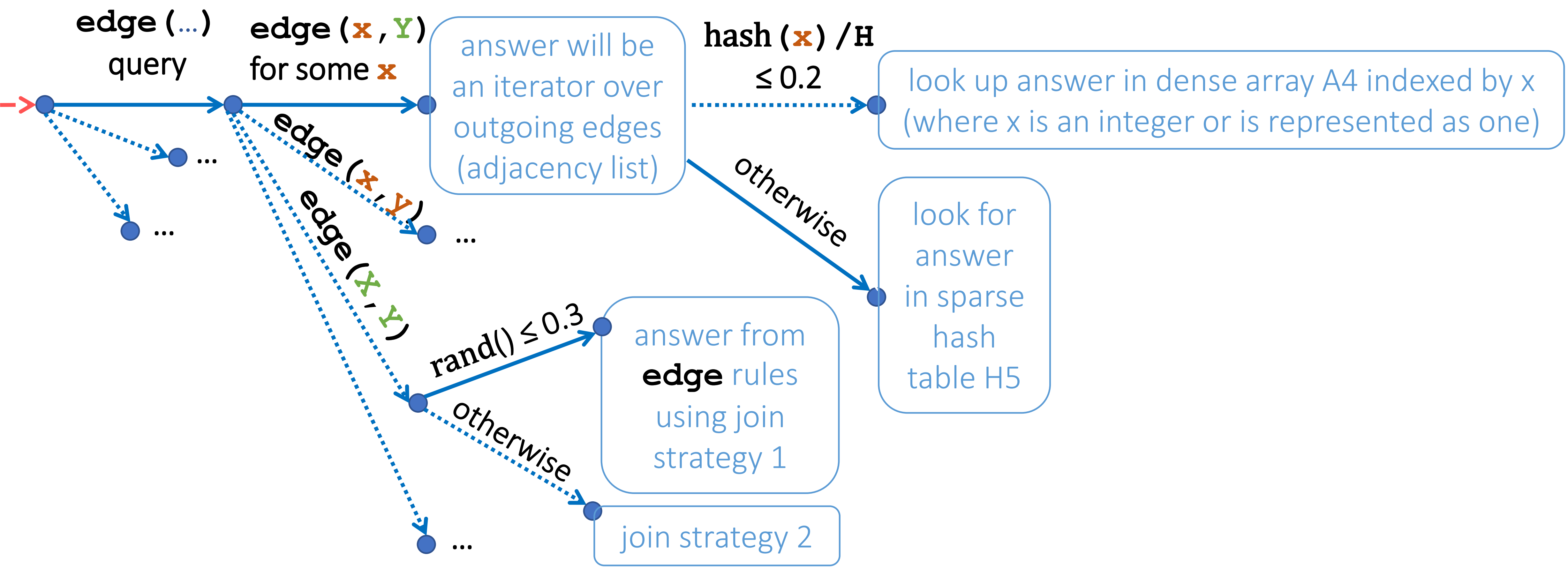
rand() ≤ 0.3

answer from **edge** rules using join strategy 1

otherwise

join strategy 2

Sequential choices at runtime (some stochastic)



edge(…) query

edge(x,Y) for some **x**

answer will be an iterator over outgoing edges (adjacency list)

hash(x)/H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

otherwise

look for answer in sparse hash table H5

edge(x,y)

…

edge(X,Y)

rand() ≤ 0.3

answer from **edge** rules using join strategy 1

otherwise

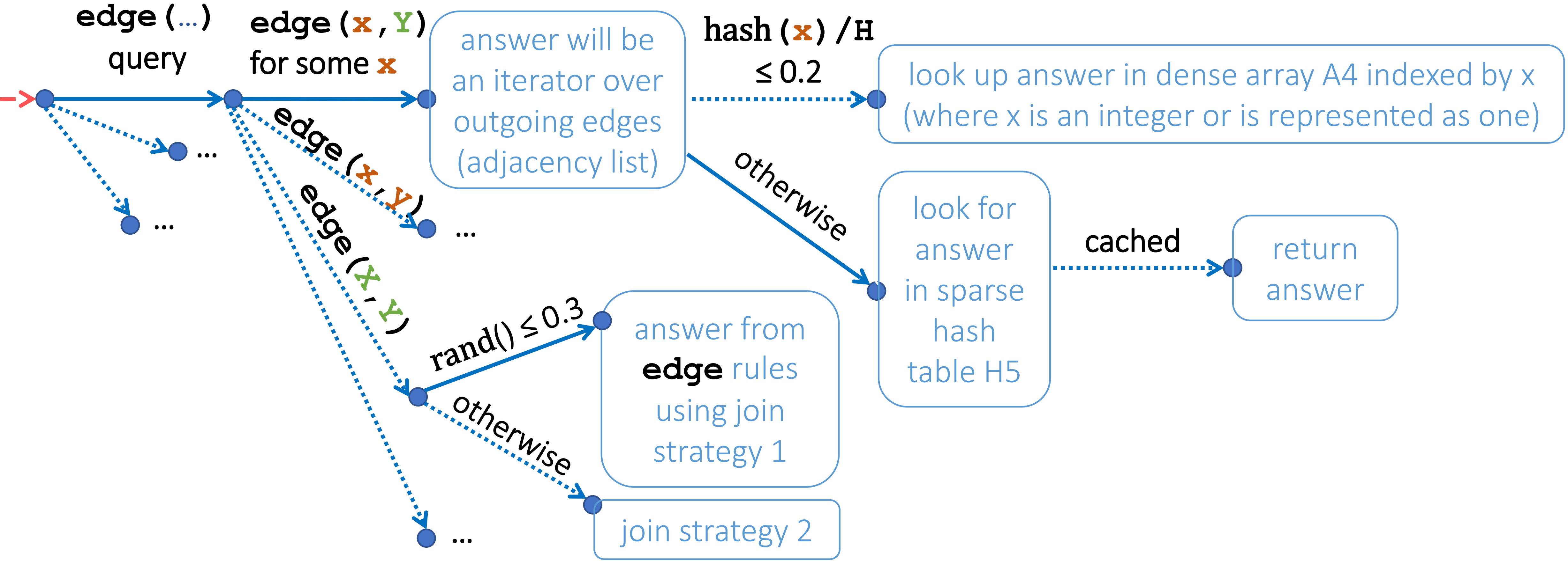
join strategy 2

…

…

…

Sequential choices at runtime (some stochastic)



edge(…)
query

edge(x,Y)
for some **x**

answer will be an iterator over outgoing edges (adjacency list)

hash(x)/H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

otherwise

look for answer in sparse hash table H5

cached

return answer

edge(x,Y)

edge(X,Y)

rand() ≤ 0.3

answer from **edge** rules using join strategy 1

otherwise

join strategy 2

…

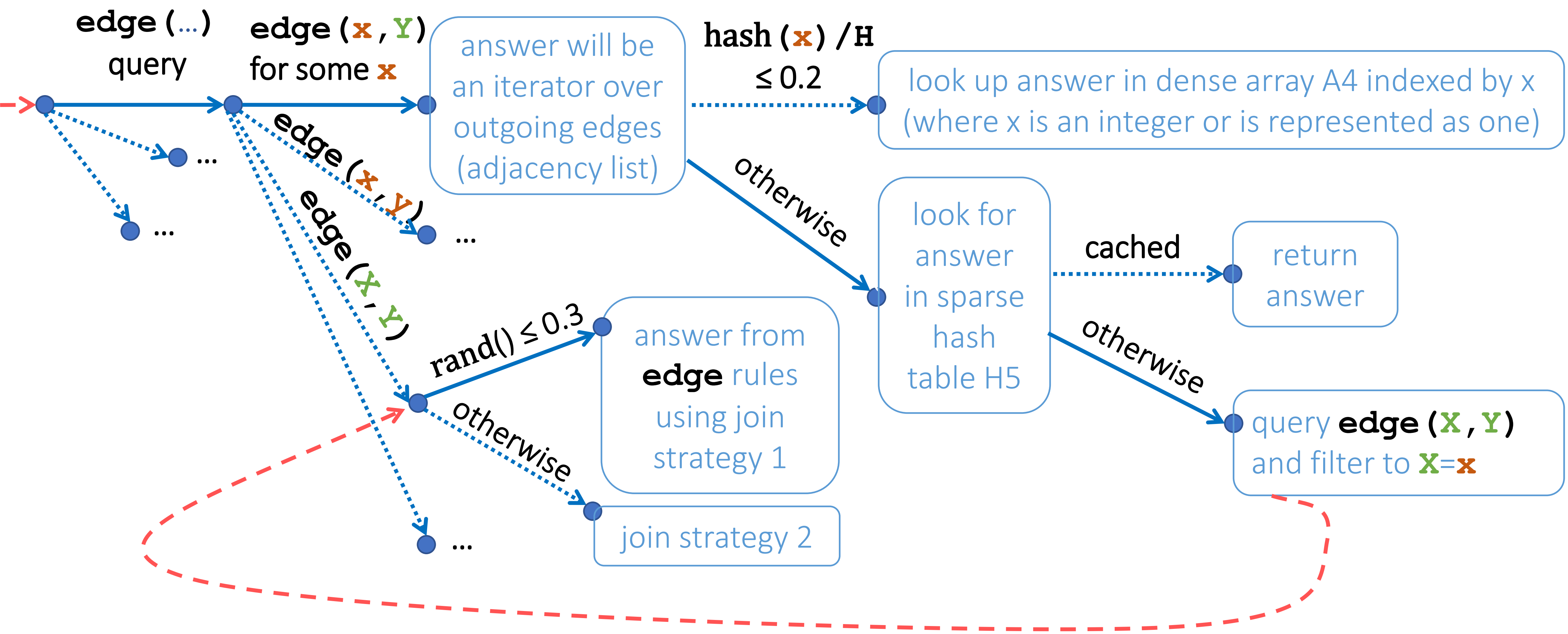
…

…

…

…

Sequential choices at runtime (some stochastic)



edge(...) query

edge(x,Y) for some **x**

answer will be an iterator over outgoing edges (adjacency list)

hash(x)/H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

otherwise

look for answer in sparse hash table H5

cached

return answer

otherwise

query **edge(X,Y)** and filter to **X=x**

edge(x,Y)

...

...

...

edge(X,Y)

rand() ≤ 0.3

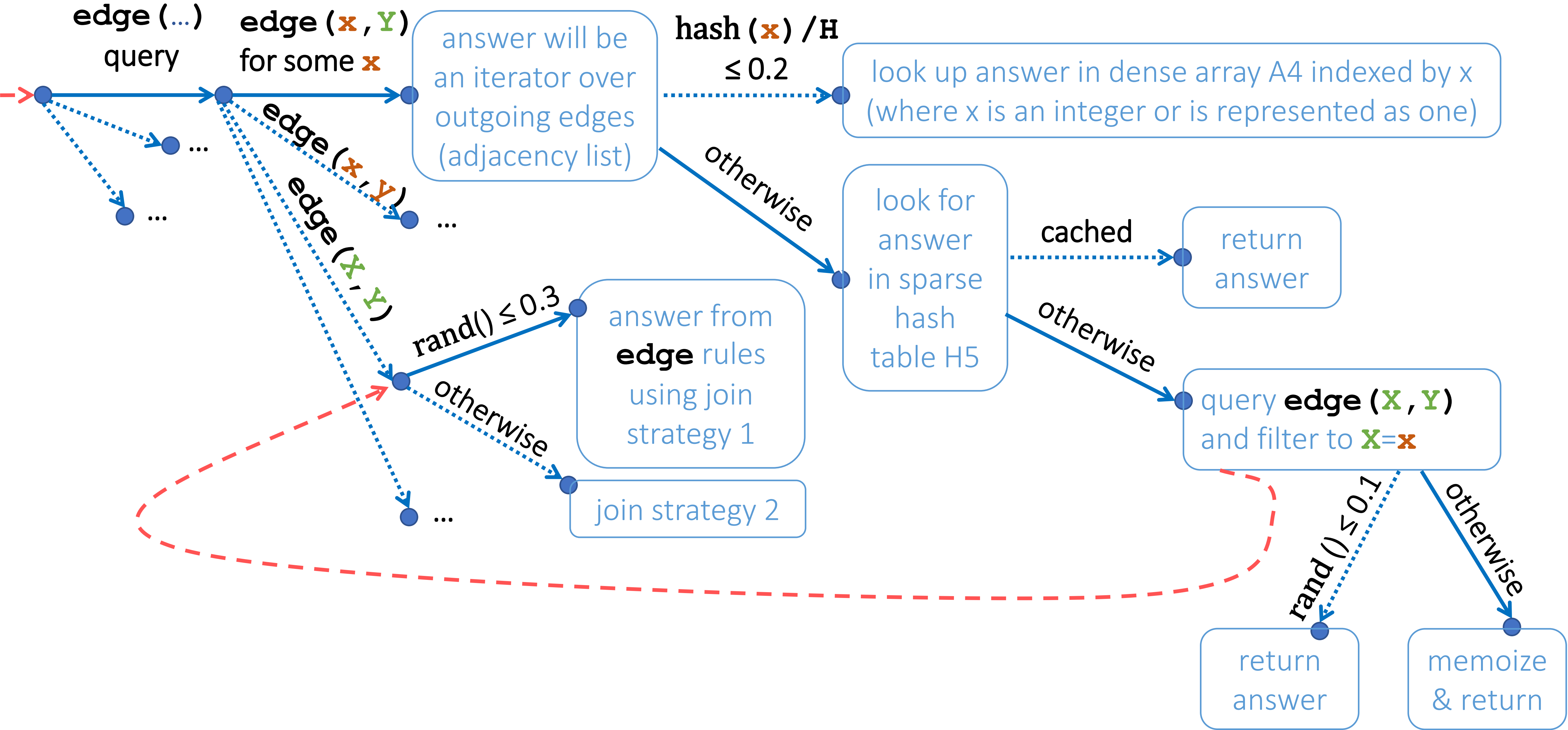
answer from **edge** rules using join strategy 1

otherwise

join strategy 2

...

Sequential choices at runtime (some stochastic)



edge(…) query

edge(**x**,**Y**) for some **x**

answer will be an iterator over outgoing edges (adjacency list)

hash(**x**)/H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

otherwise

look for answer in sparse hash table H5

cached

return answer

otherwise

query **edge**(**X**,**Y**) and filter to **X**=**x**

rand() ≤ 0.1

return answer

otherwise

memoize & return

edge(**x**,**y**)

edge(**X**,**y**)

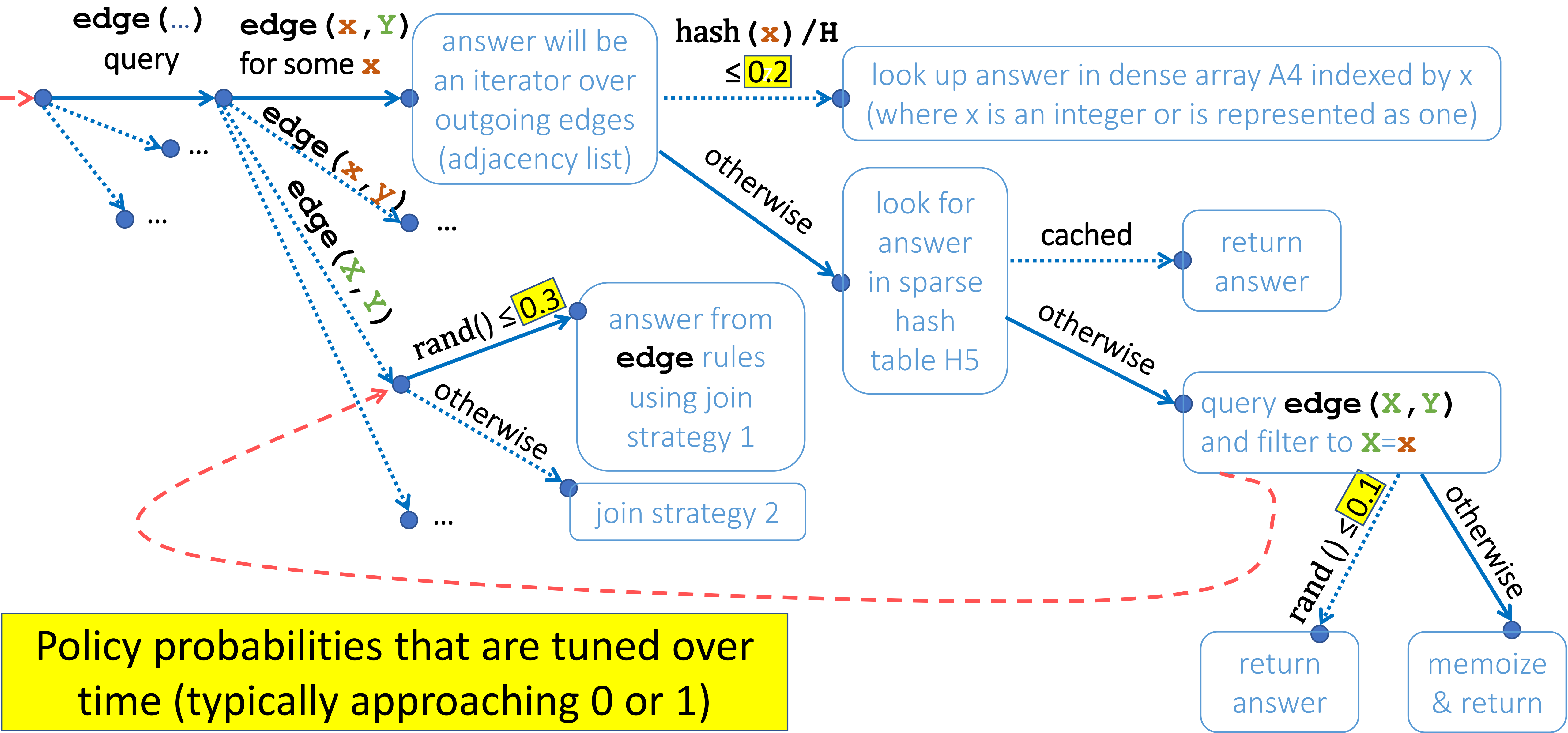
rand() ≤ 0.3

answer from **edge** rules using join strategy 1

otherwise

join strategy 2

Sequential choices at runtime (some stochastic)



edge(...) query

edge(x,Y) for some x

answer will be an iterator over outgoing edges (adjacency list)

hash(x)/H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

otherwise

look for answer in sparse hash table H5

cached

return answer

otherwise

query **edge**(X,Y) and filter to X=x

rand() ≤ 0.1

return answer

otherwise

memoize & return

edge(x,Y)

edge(X,Y)

rand() ≤ 0.3

answer from **edge** rules using join strategy 1

otherwise

join strategy 2

...

Policy probabilities that are tuned over time (typically approaching 0 or 1)

Policy probabilities can be sensitive to context of task

Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

- **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?

Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

- **Task characteristics**
    - What type of task?
    - What are the task parameters?
    - Who requested the task?
- **Dataflow**
    - What depends on this task (children)?
    - What does this task depend on (parents)?

Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

- **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?
- **Dataflow**
  - What depends on this task (children)?
  - What does this task depend on (parents)?
- **Agenda characteristics**
  - Are there a lot of open queries?
  - What's on the agenda? How long has it been there?

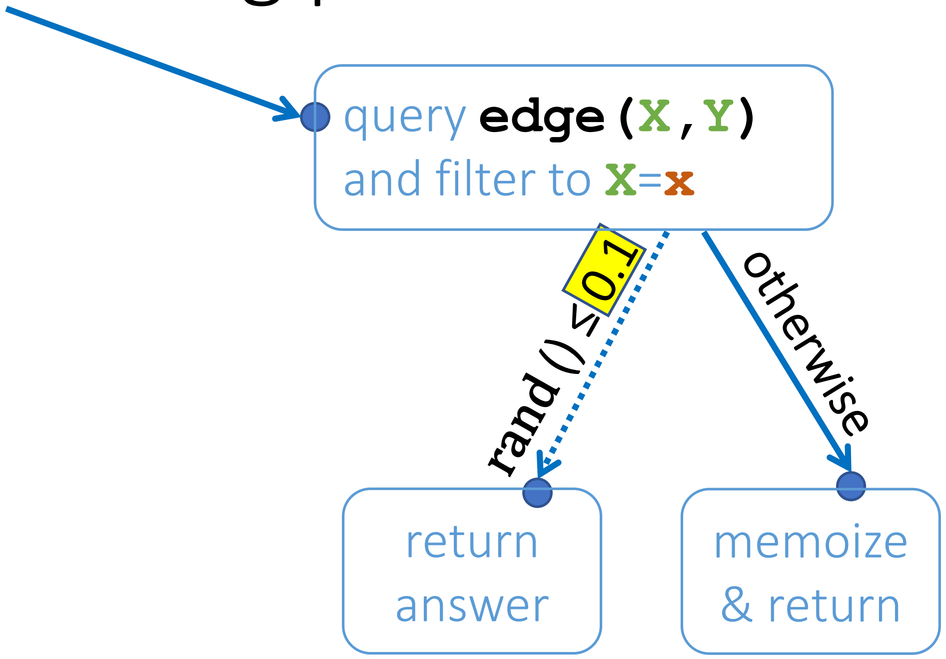
Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

- **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?
- **Dataflow**
  - What depends on this task (children)?
  - What does this task depend on (parents)?
- **Agenda characteristics**
  - Are there a lot of open queries?
  - What’s on the agenda? How long has it been there?
- **Cache characteristics**
  - Statistics: number, hit rate, frequency, & recency of memos (broken down by type)

Tuning probabilities

Tuning probabilities

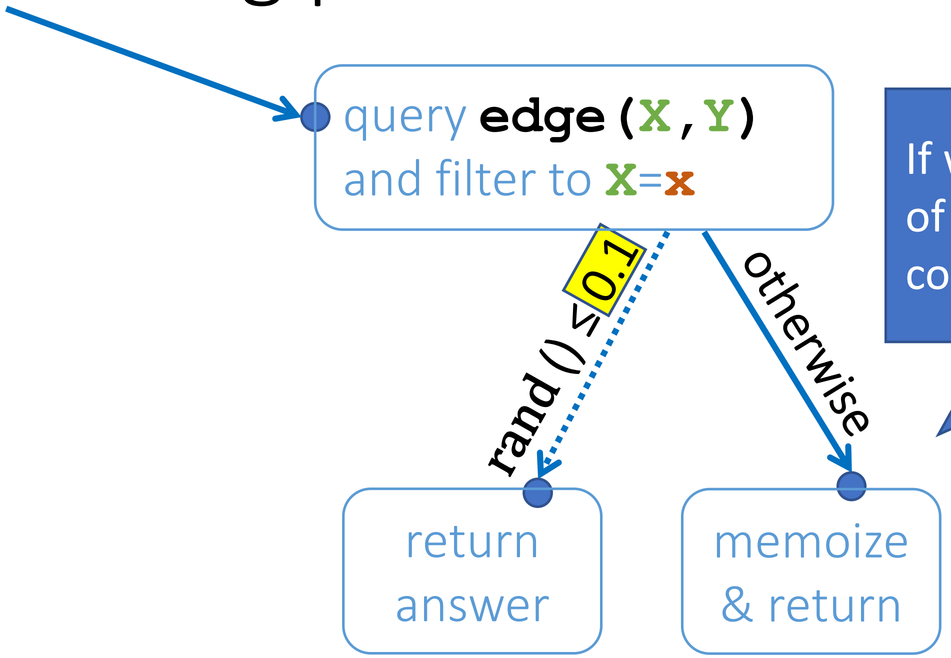


query **`edge(X,Y)`** and filter to `X`=`x`

return answer

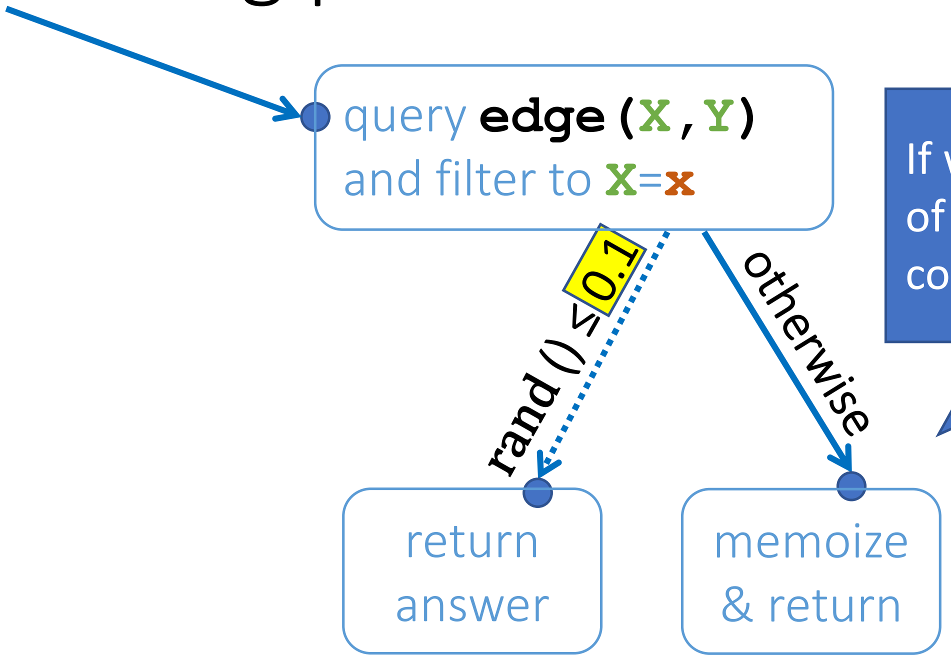
memoize & return

Tuning probabilities



If we knew the *long-term* cost of each action in context, we could update the policy now!

Tuning probabilities



query **`edge(X,Y)`** and filter to **`X`**=**`x`**

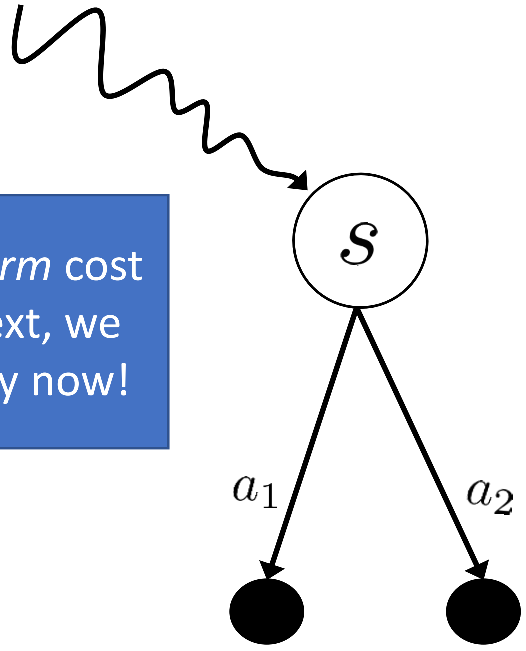
rand() ≤ 0.1

otherwise

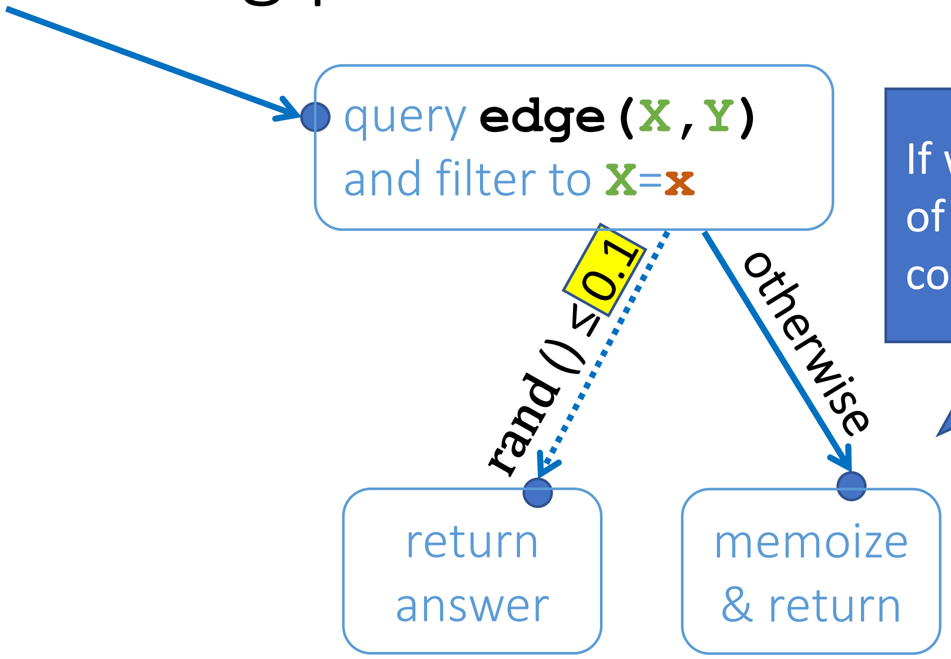
return answer

memoize & return

If we knew the *long-term* cost of each action in context, we could update the policy now!



Tuning probabilities



query **edge**(**X**,**Y**)
and filter to **X**=**x**

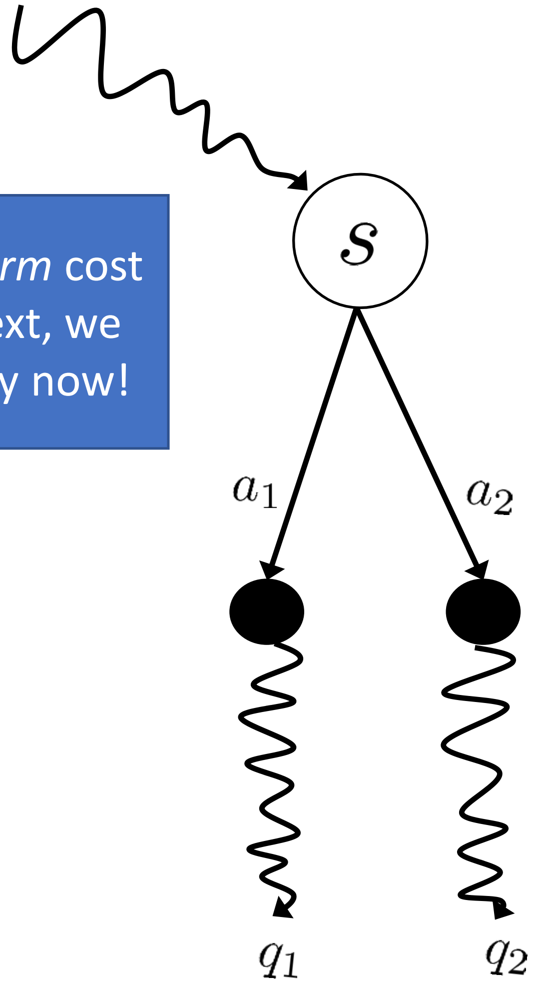
rand() ≤ 0.1

otherwise

return answer

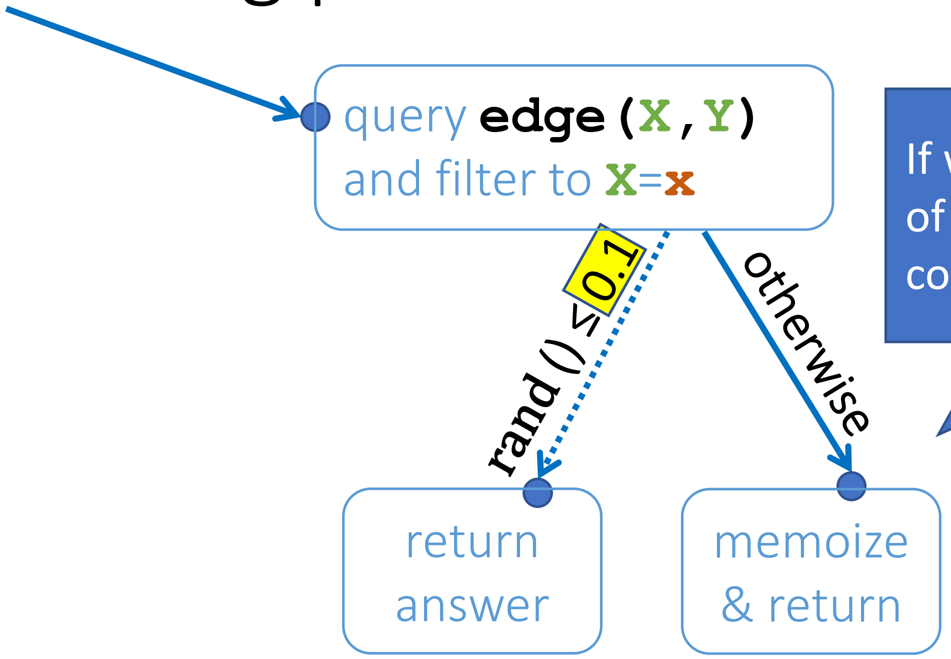
memoize & return

If we knew the *long-term* cost of each action in context, we could update the policy now!



hypothetically, fork state
and run finish workload

Tuning probabilities



query **`edge(X,Y)`** and filter to **`X`**=**`x`**

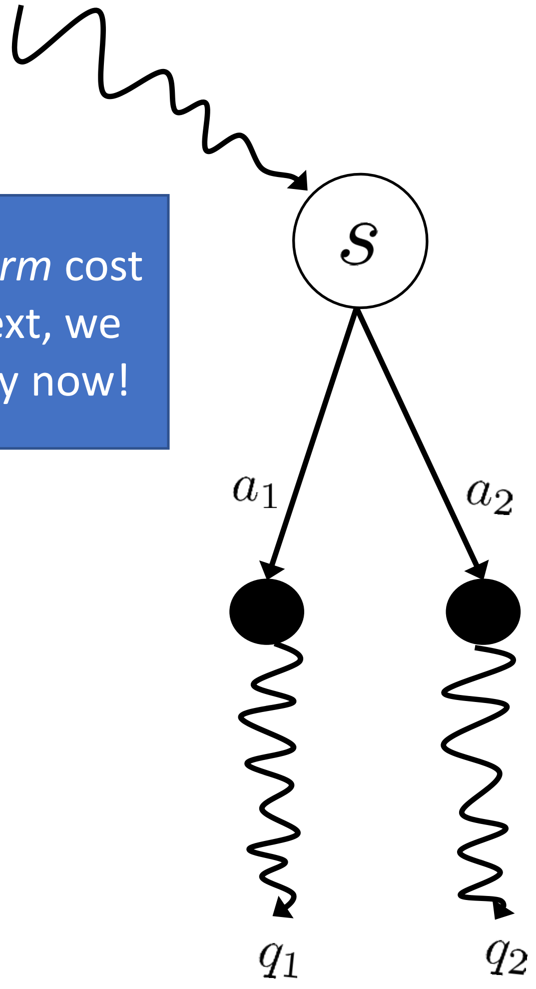
`rand()` ≤ 0.1

otherwise

return answer

memoize & return

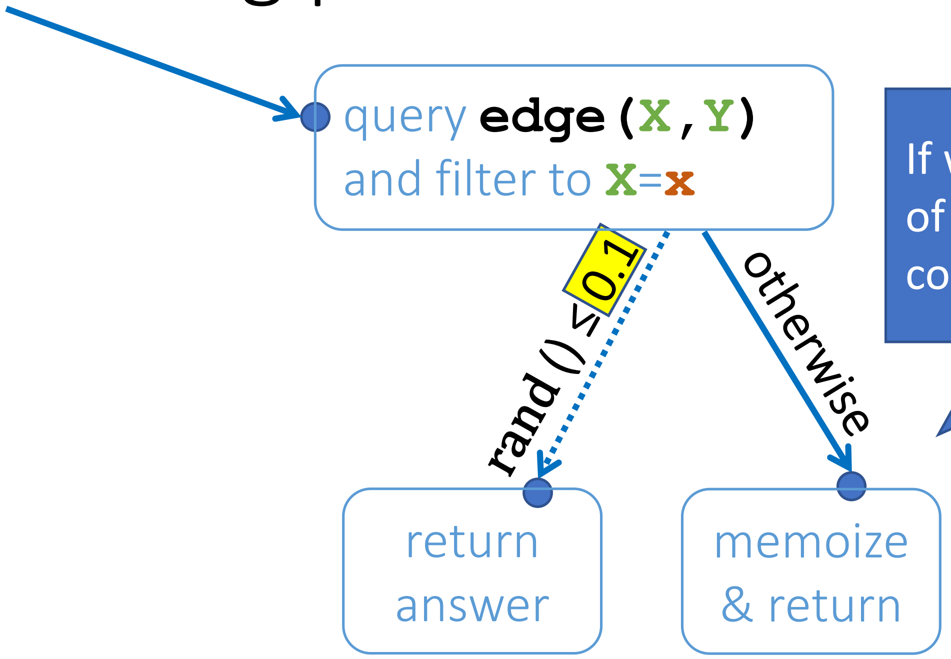
If we knew the *long-term* cost of each action in context, we could update the policy now!



hypothetically, fork state and run finish workload

Slow!

Tuning probabilities



query **edge(X,Y)** and filter to **X=x**

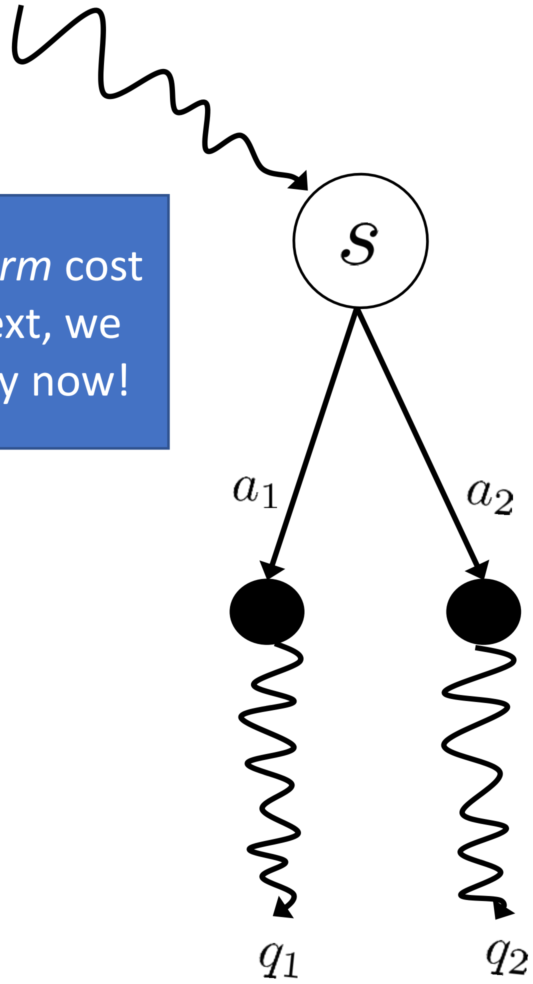
rand() ≤ 0.1

otherwise

return answer

memoize & return

If we knew the *long-term* cost of each action in context, we could update the policy now!



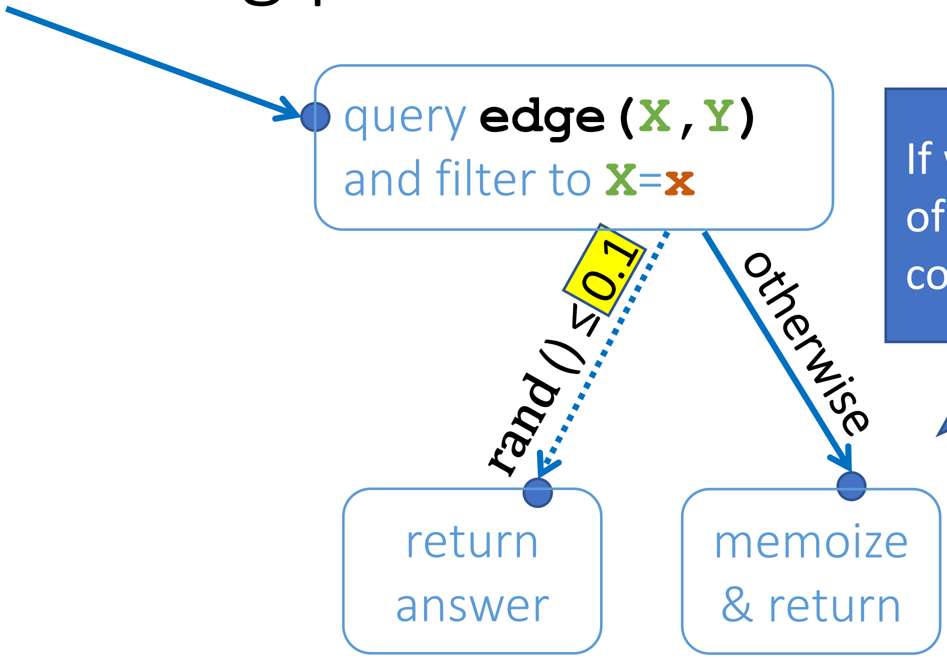
hypothetically, fork state and run finish workload

Slow!

Use ML to predict future costs!

$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i)$$

Tuning probabilities


query **edge(X,Y)** and filter to **X**=**x**

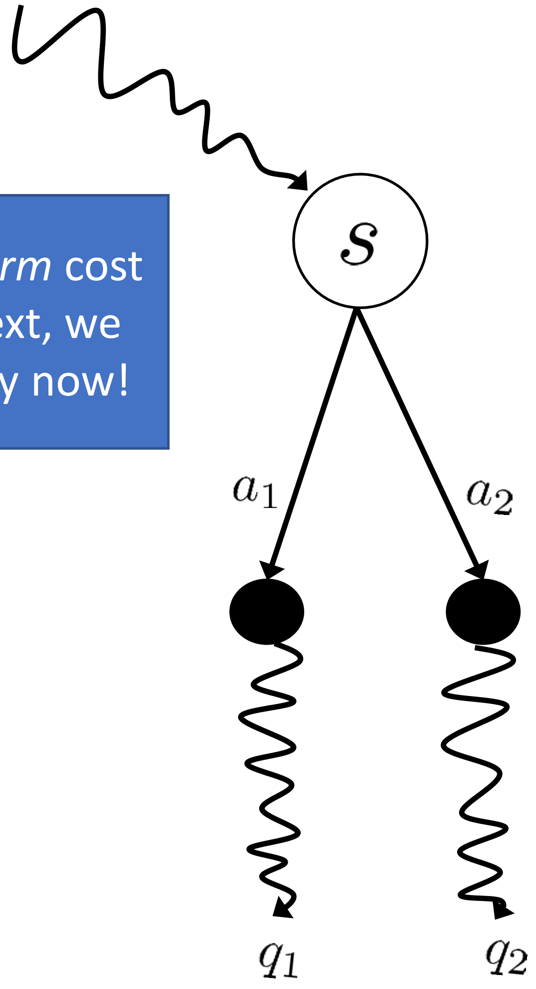
rand() ≤ 0.1

otherwise

return answer

memoize & return

If we knew the *long-term* cost of each action in context, we could update the policy now!



hypothetically, fork state and run finish workload

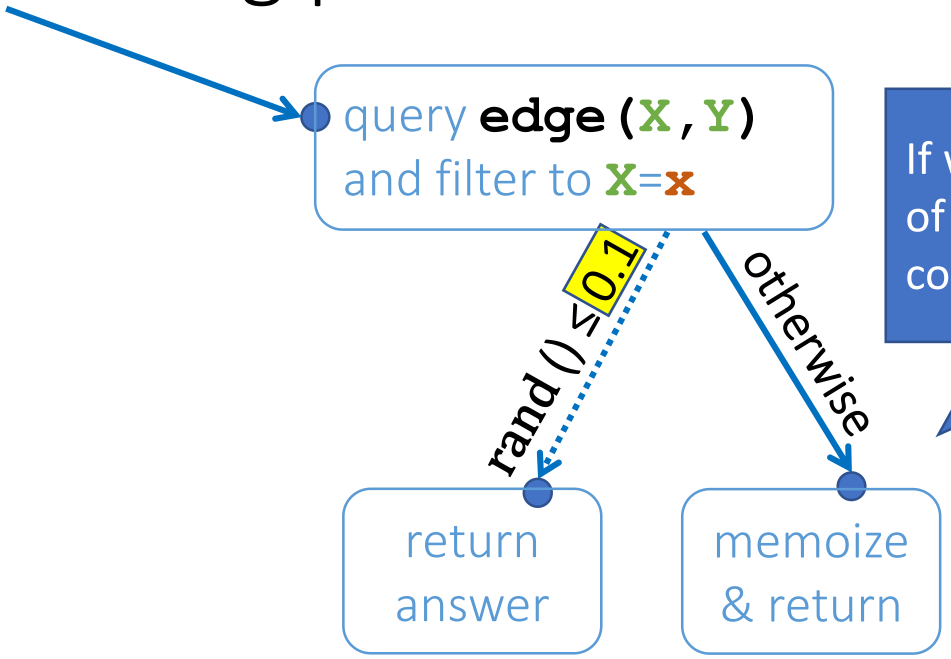
Slow!

Use ML to predict future costs!

Generalize from past experience to new situations

$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i)$$

Tuning probabilities



query **edge(X,Y)** and filter to **X**=**x**

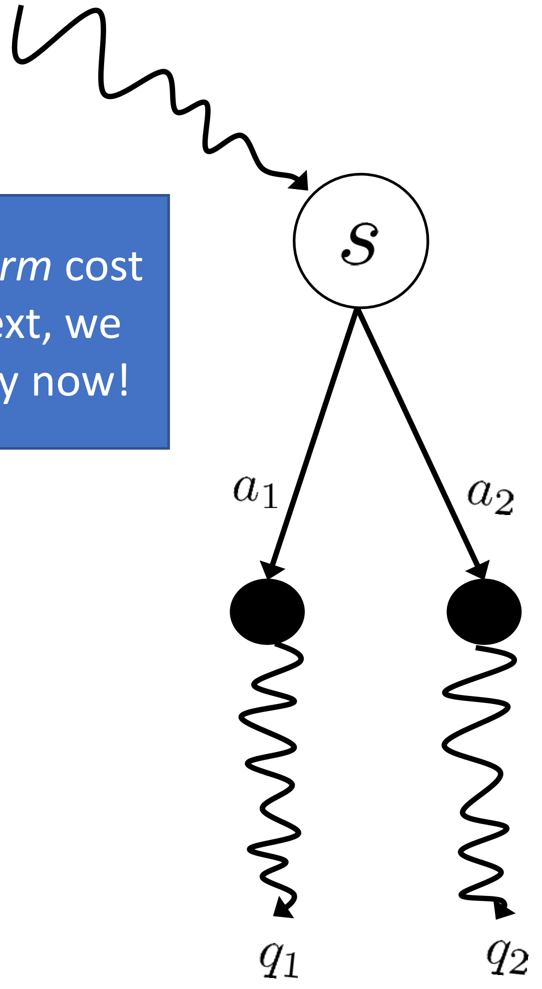
rand() ≤ 0.1

otherwise

return answer

memoize & return

If we knew the *long-term* cost of each action in context, we could update the policy now!



hypothetically, fork state and run finish workload

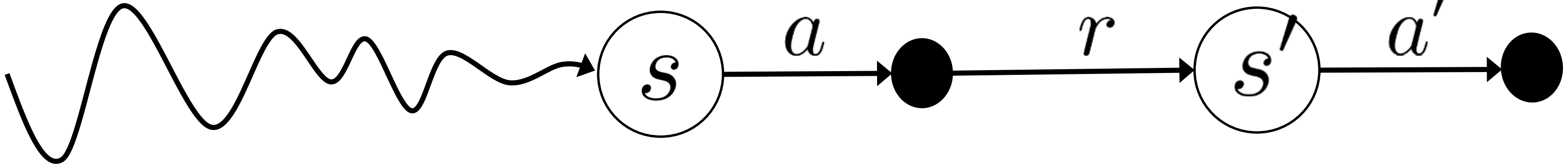
Slow!

Use ML to predict future costs!

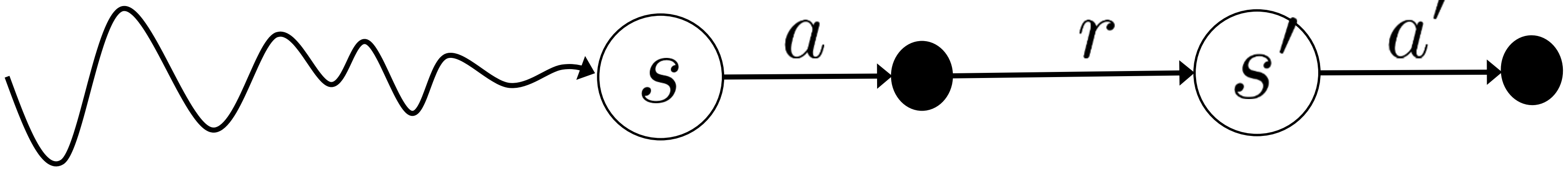
Generalize from past experience to new situations

$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i) = w^\top \Phi(s, a_i)$$

Temporal difference learning

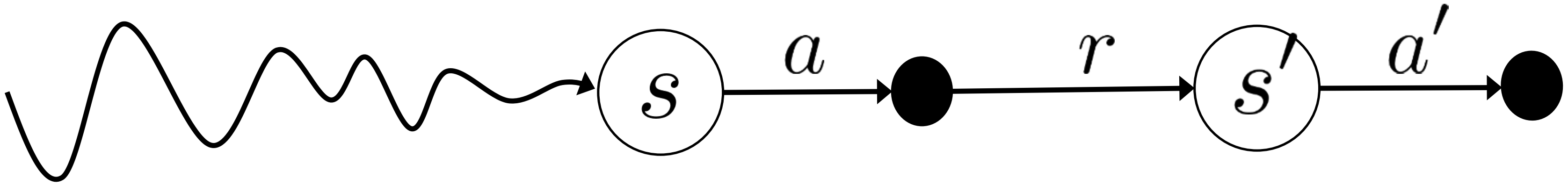


Temporal difference learning



$$\widehat{q}(s,a) \approx \mathbb{E}\left[r + \widehat{q}(s',a')\right]$$

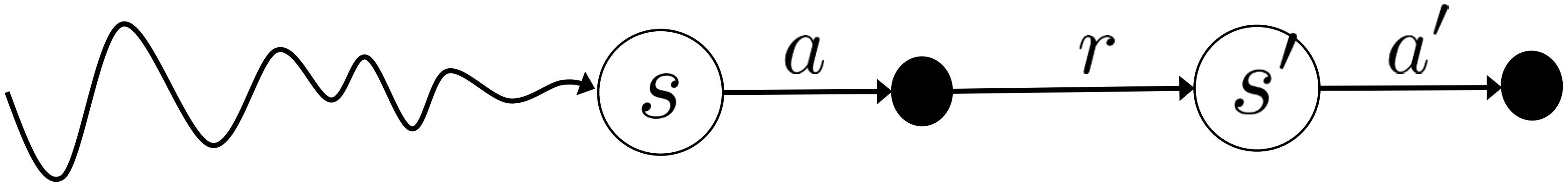
Temporal difference learning



Approximate dynamic programming

$$\widehat{q}(s,a) \approx \mathbb{E}\left[r + \widehat{q}(s',a')\right]$$

Temporal difference learning

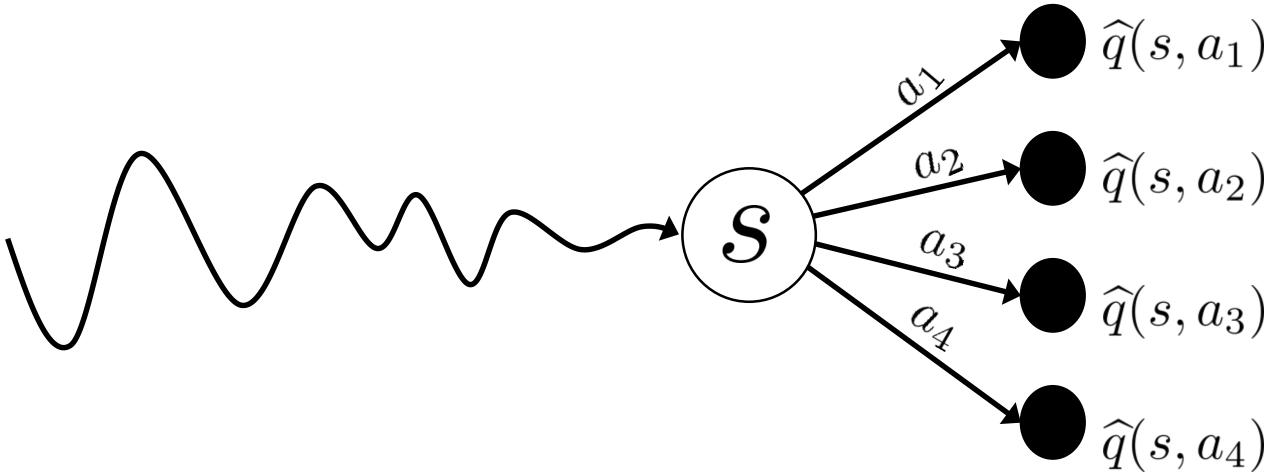


Make estimator agree with itself

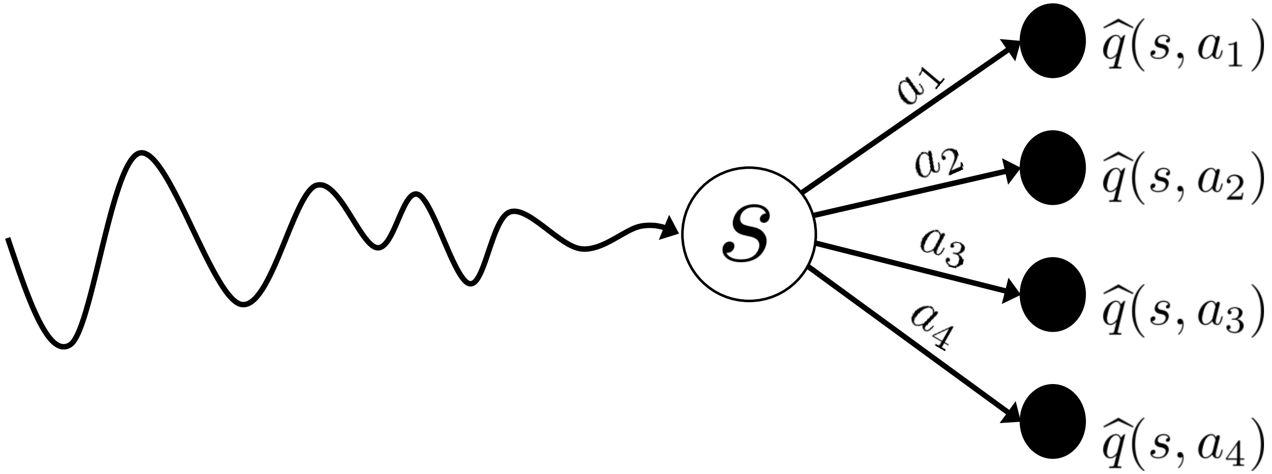
Approximate dynamic programming

$$\widehat{q}(s,a) \approx \mathbb{E}\left[r + \widehat{q}(s',a')\right]$$

Actor-critic policy gradient

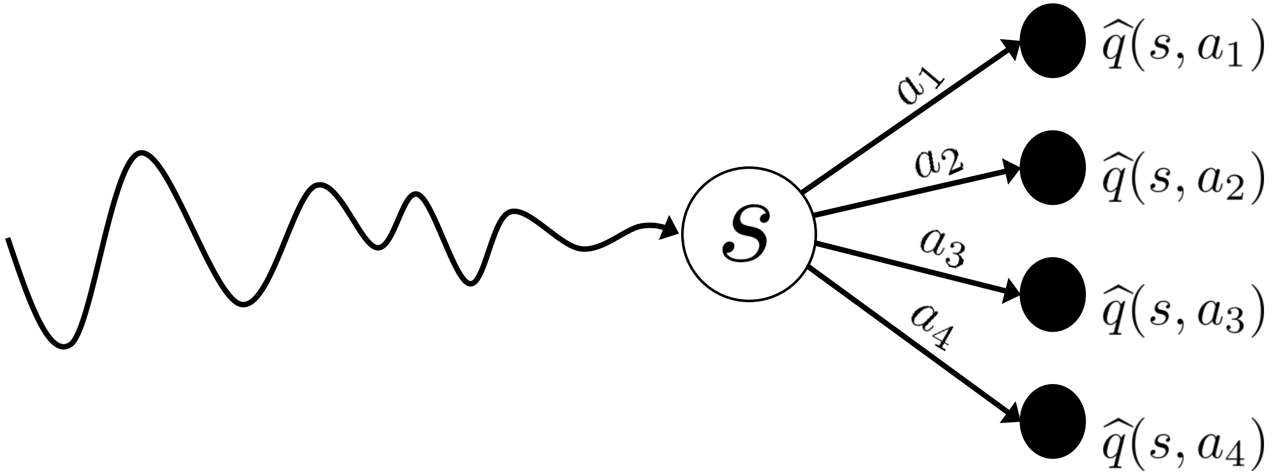


Actor-critic policy gradient



$$\pi(a|s) \propto \exp\big(\theta^\top f(s, a)\big)$$

Actor-critic policy gradient

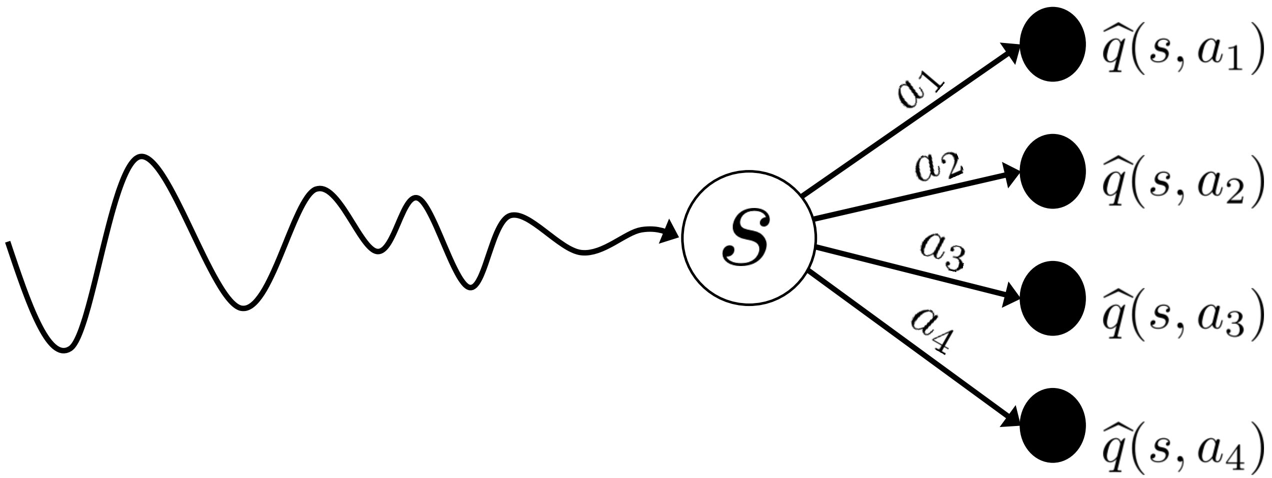


Says increase the probability of lower cost actions.

$$\pi(a|s) \propto \exp\left(\theta^\top f(s,a)\right)$$

$$\theta \leftarrow \theta - \alpha_t \cdot \widehat{q}(s,a) \nabla \log \pi(a|s)$$

Actor-critic policy gradient



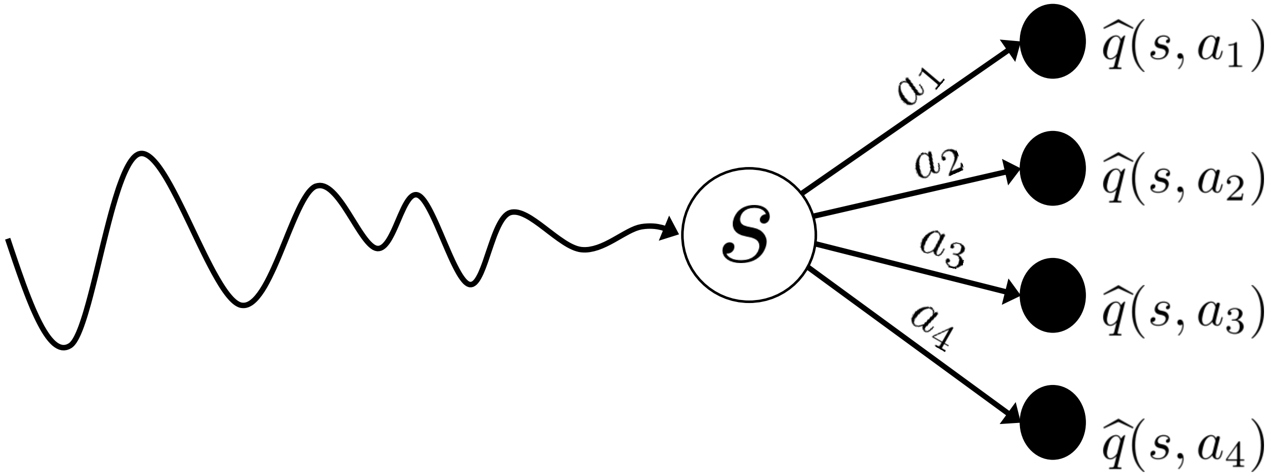
$$\pi(a|s) \propto \exp\left(\theta^\top f(s,a)\right)$$

$$\theta \leftarrow \theta - \alpha_t \cdot \widehat{q}(s,a) \nabla \log \pi(a|s)$$

Says increase the probability of lower cost actions.

$$\pi(\cdot|s) \approx \underset{a'}{\operatorname{argmin}}\ \widehat{q}(s,a')$$

Actor-critic policy gradient



Says increase the probability of lower cost actions.

$$\pi(a|s) \propto \exp\left(\theta^\top f(s,a)\right)$$

$$\pi(\cdot|s) \approx \underset{a'}{\operatorname{argmin}}\ \widehat{q}(s,a')$$

$$\theta \leftarrow \theta - \alpha_t \cdot \widehat{q}(s,a) \nabla \log \pi(a|s)$$

update q

$$w = w - \beta_t \cdot (\widehat{q}(s,a) - (r + \widehat{q}(s',a'))) \cdot \Phi(s,a)$$

Summary

Summary

- Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.

Summary

- Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.
- Tuning such as solver is a sequential decision making problem that can be tuned with online reinforcement learning techniques.

Summary

- Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.
- Tuning such as solver is a sequential decision making problem that can be tuned with online reinforcement learning techniques.
- Dyna
  - Has been designed from the ground up be as flexible as possible.
  - A powerful language for specifying machine learning applications.

Summary

- Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.
- Tuning such as solver is a sequential decision making problem that can be tuned with online reinforcement learning techniques.
- Dyna
  - Has been designed from the ground up be as flexible as possible.
  - A powerful language for specifying machine learning applications.

Check out the paper! Lots of great technical details in the paper that didn't have time to get into.

State of the language

State of the language

- We have to two language prototypes

State of the language

- We have to two language prototypes
  - Dyna 1 prototype (2005) was used in Jason’s lab, fueling a dozen NLP papers!

State of the language

- We have to two language prototypes
    - Dyna 1 prototype (2005) was used in Jason’s lab, fueling a dozen NLP papers!
    - Dyna 2 prototype (2013) was used for teaching an NLP course to linguists with no programming background.

State of the language

- We have to two language prototypes
  - Dyna 1 prototype (2005) was used in Jason’s lab, fueling a dozen NLP papers!
  - Dyna 2 prototype (2013) was used for teaching an NLP course to linguists with no programming background.
- Both were inefficient because they used too many one-size-fits-all strategies.

Thanks!

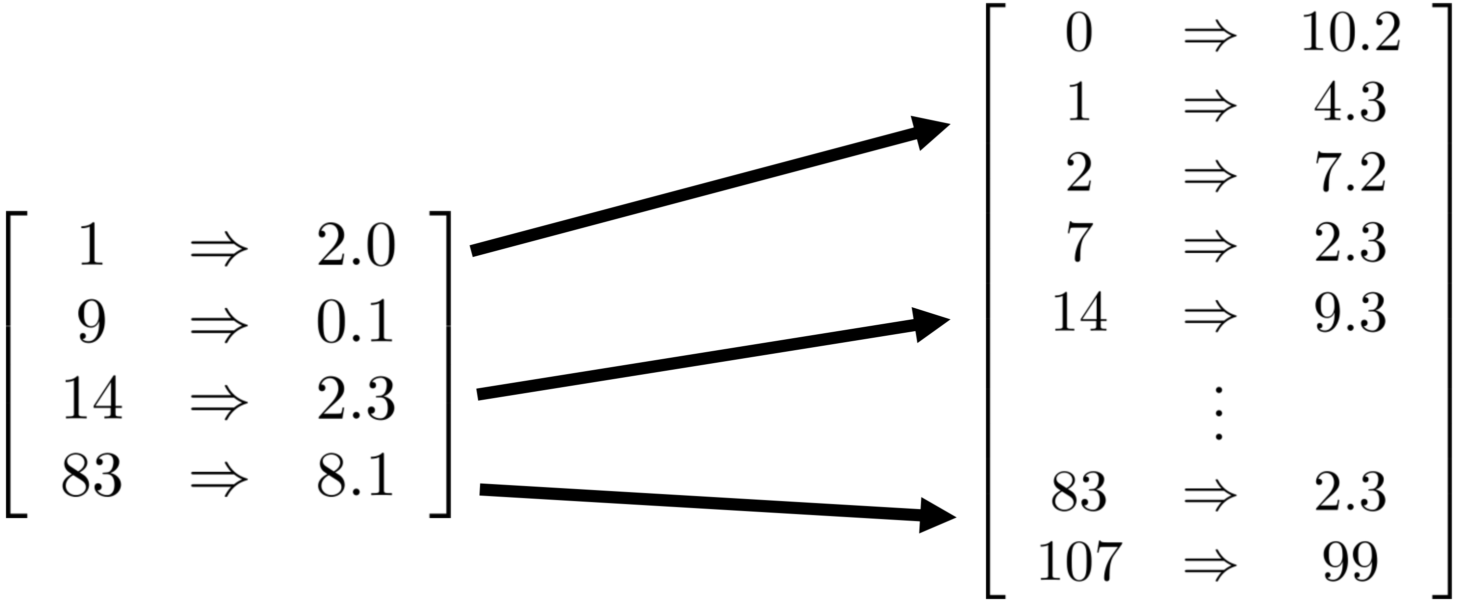
Questions? Comments?

Hire Wes Filardo!
http://cs.jhu.edu/~nwf

http://dyna.org

@xtimv
@matthewfl

Loop order, sparse vector product example



$$\begin{bmatrix} 1 & \Rightarrow & 2.0 \\ 9 & \Rightarrow & 0.1 \\ 14 & \Rightarrow & 2.3 \\ 83 & \Rightarrow & 8.1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & \Rightarrow & 10.2 \\ 1 & \Rightarrow & 4.3 \\ 2 & \Rightarrow & 7.2 \\ 7 & \Rightarrow & 2.3 \\ 14 & \Rightarrow & 9.3 \\ & \vdots & \\ 83 & \Rightarrow & 2.3 \\ 107 & \Rightarrow & 99 \end{bmatrix}$$

Join strategies

- Outer loop on `b(J,4)`, inner loop `a(I,J)`
- Outer loop on `a(I,J)`, inner loop `b(J,K)`, filter `K==4`
- Sort `a` and `b` on `J` and use Baeza-Yates intersection
- What if we can't loop over `b`? e.g.,
  `b(I,K) = X*Y.`
- In natural language parsing with the CKY algorithm, an unexpected loop order turned out to be 7-9x faster than the loop order presented in most textbooks because of cache locality (Dunlop et al. 2010).

```
% matrix multiplication
c(I,K) += a(I,J) * b(J,K).
```

Memoization and data structures

- Do we store results for future use?
  (tradeoff: memos must be kept up-to-date!)
- What data structure?

Batching, answering bigger queries/updates

- Batch pending `c` queries.
- Preemptively compute a broader query, `c(I,K)`
  Use clever mat-mul alg. (sparse or dense?)

Inlining

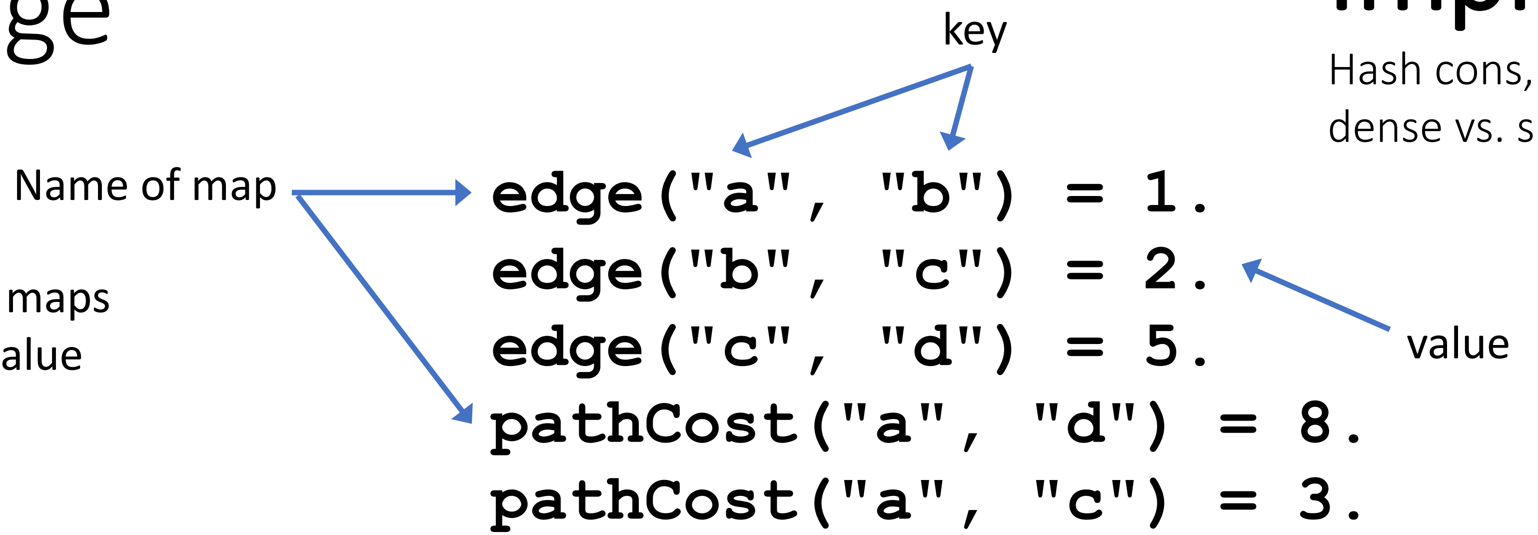
- Inline a and/or b queries.
- Bypass agenda and route directly to consumer, e.g.,
  `d(I) max= c(I,K).`
- Reduce overhead of method calls
- Reduce overhead of task indirection through agenda

Mixed policies

- Mixed task strategies selection
  - Policy will encounter similar tasks frequently
- Mixed storage strategies
  - e.g., use a hash table, prefix trie, dense array, …
  - Problem: random choice of strategy might not be consistent
    - E.g. might write to A and read from B (because of randomness)

Storage

Named maps
key -> value



```
edge("a", "b") = 1.
edge("b", "c") = 2.
edge("c", "d") = 5.
pathCost("a", "d") = 8.
pathCost("a", "c") = 3.
```

Implementations?

Hash cons, trie (different orders on keys), dense vs. sparse, sorted (what to sort on)

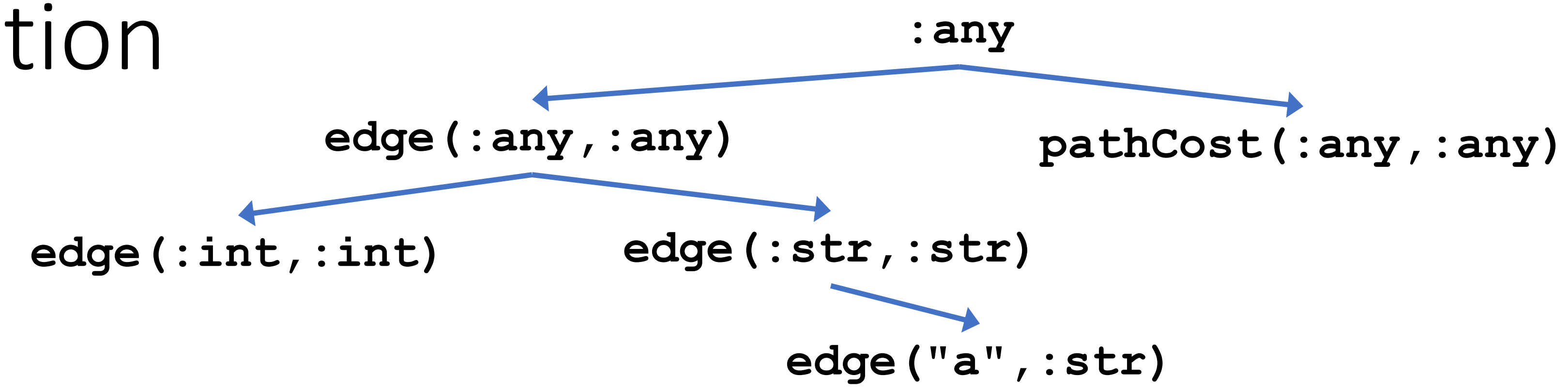
Queries

What's the weight of edge a->b	`? edge("a","b")`
Outgoing edges from a	`? edge("a",Y)`
Incoming edges to b	`? edge(X,"b")`
List all edges	`? edge(X,Y)`
List all self loops	`? edge(X,X)`
Edges with weight >= 10	`? edge(X,Y) >= 10`

Efficiency depends on

- Frequency of different queries
- Overhead of read/writes
- Sizes
- Lower-level thresholds

Mixed storage solution



`:any`

`edge(:any,:any)`

`pathCost(:any,:any)`

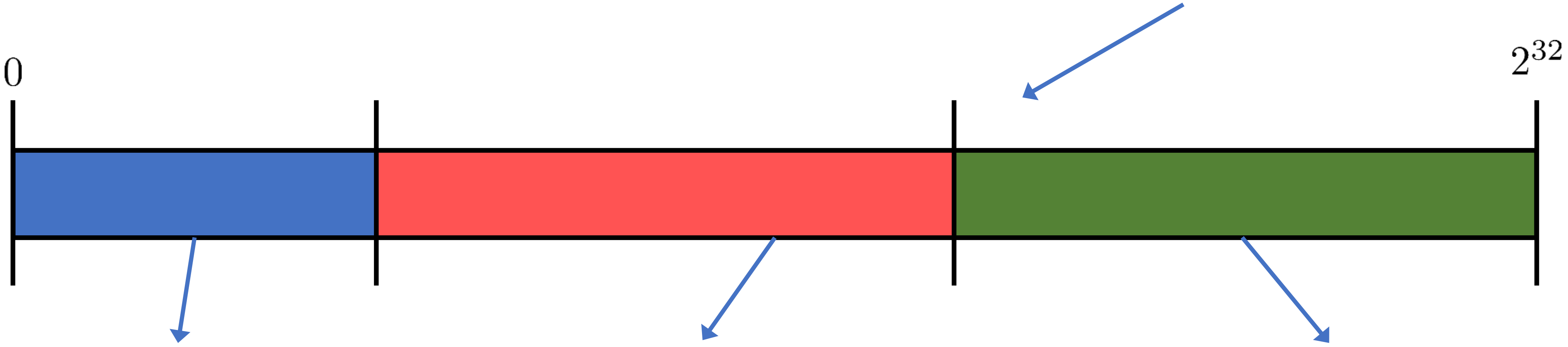
`edge(:int,:int)`

`edge(:str,:str)`

`edge("a",:str)`

hash(`edge("a", "b")`) / 2^32 ~ Uniform(0,1)

Small, temporary
data duplication

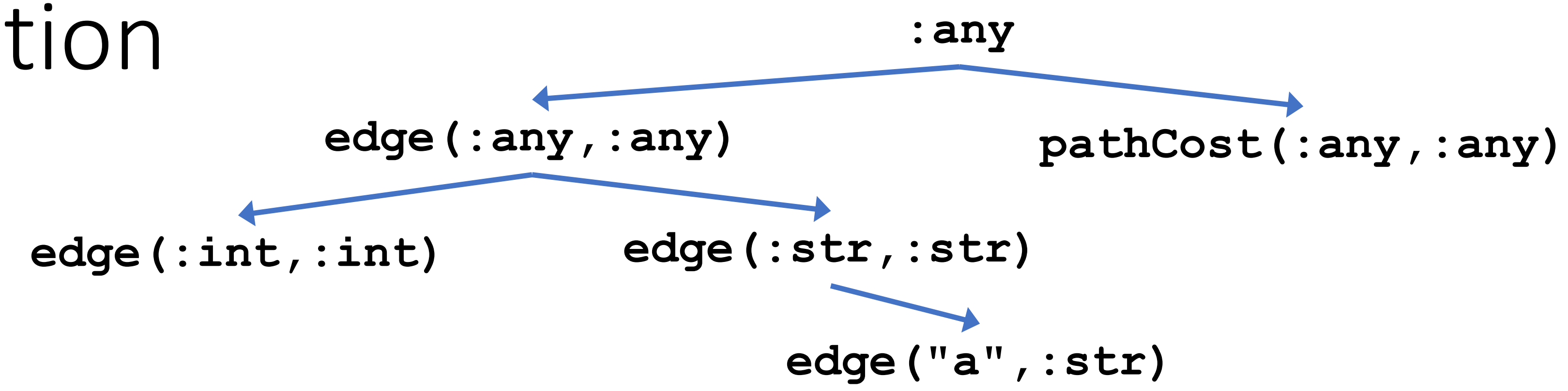


Hash table

Trie 1->2

Trie 2->1

Mixed storage solution



`:any`

`edge(:any,:any)`

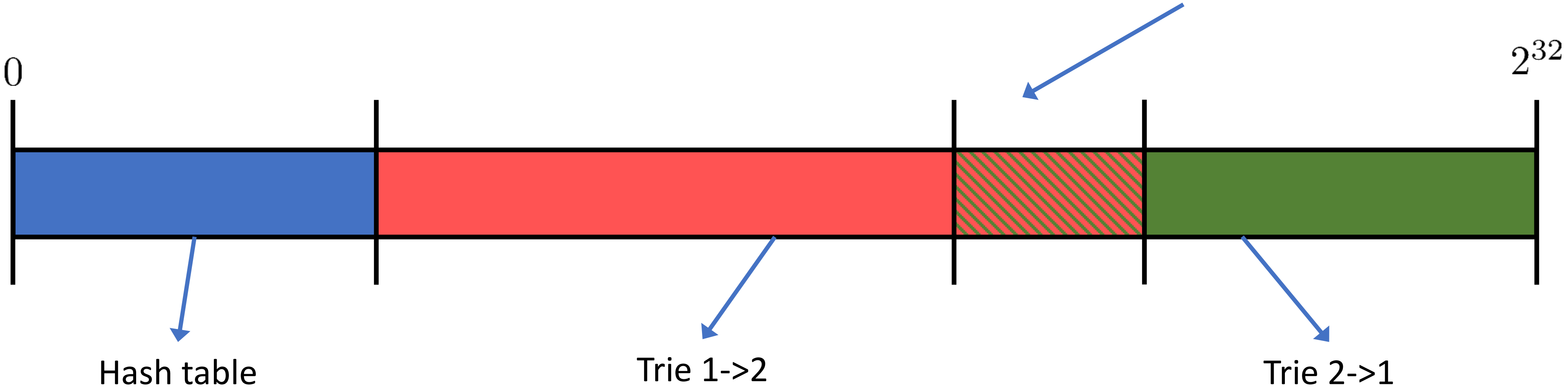
`pathCost(:any,:any)`

`edge(:int,:int)`

`edge(:str,:str)`

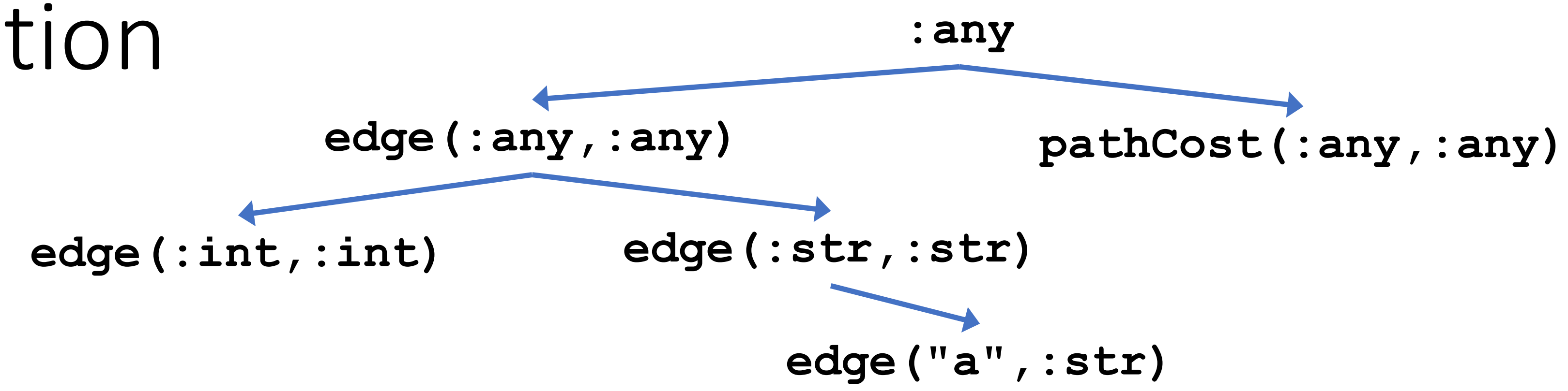
`edge("a",:str)`

hash(**`edge("a", "b")`**) / 2^32 ~ Uniform(0,1)



Small, temporary data duplication

Mixed storage solution


`:any`

`edge(:any,:any)`

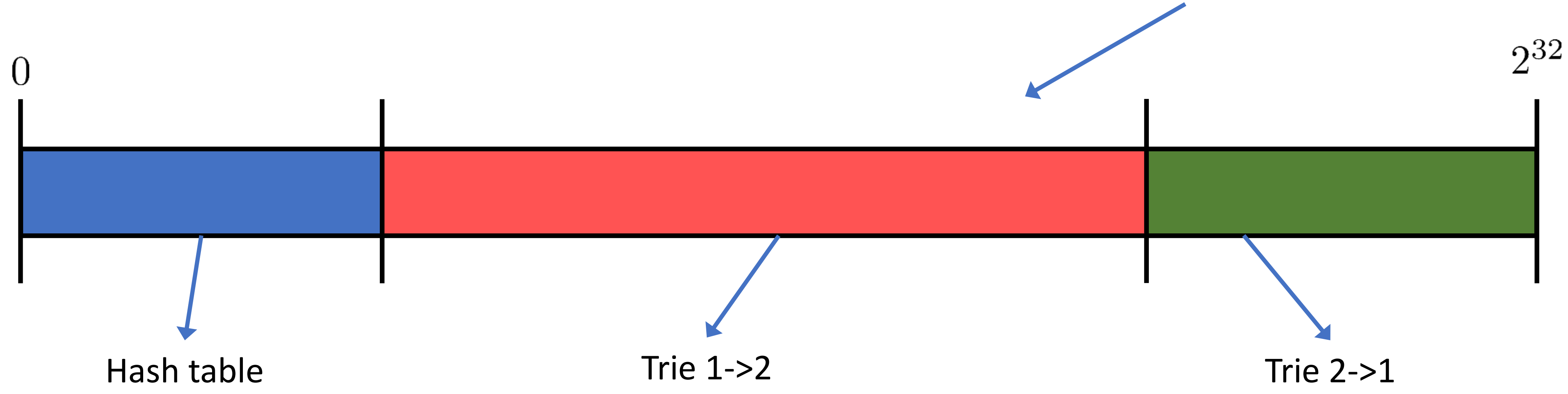
`pathCost(:any,:any)`

`edge(:int,:int)`

`edge(:str,:str)`

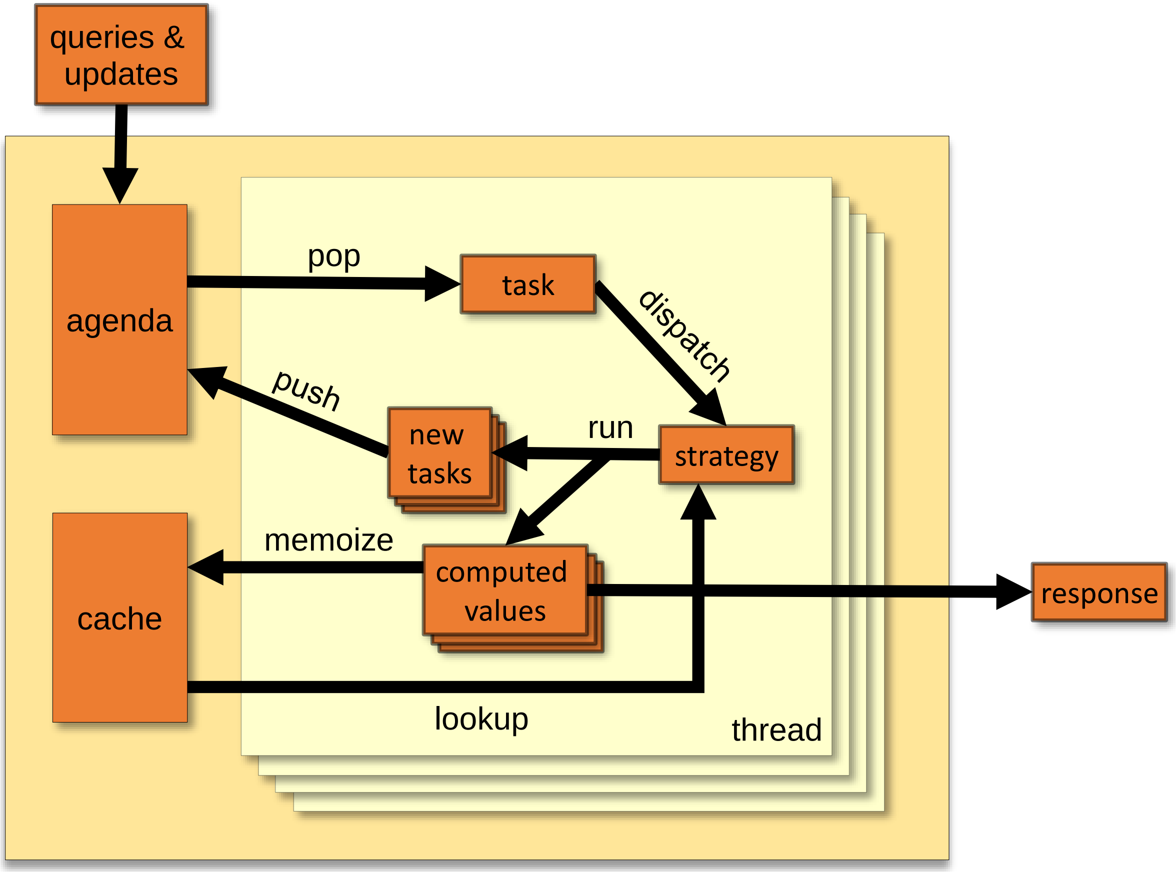
`edge("a",:str)`

hash(`edge("a", "b")`) / 2^32 ~ Uniform(0,1)

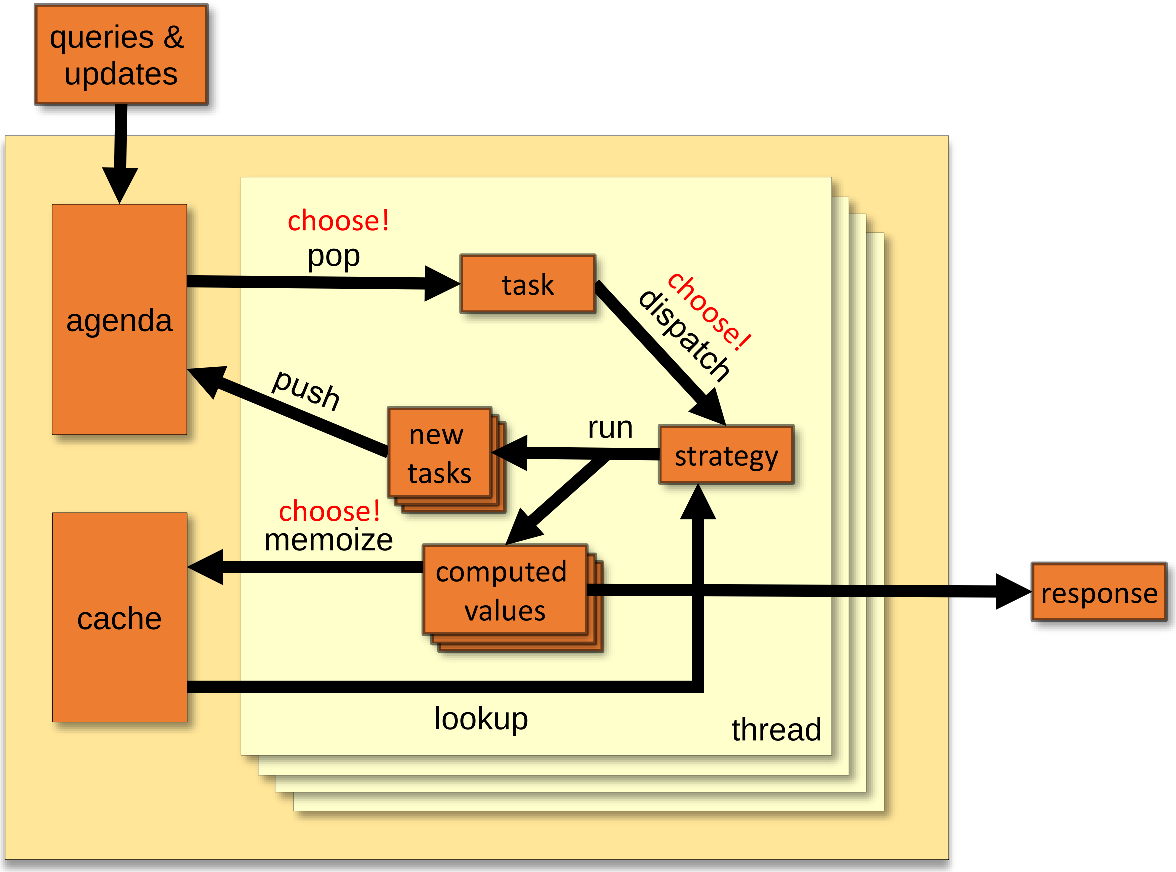


Small, temporary data duplication

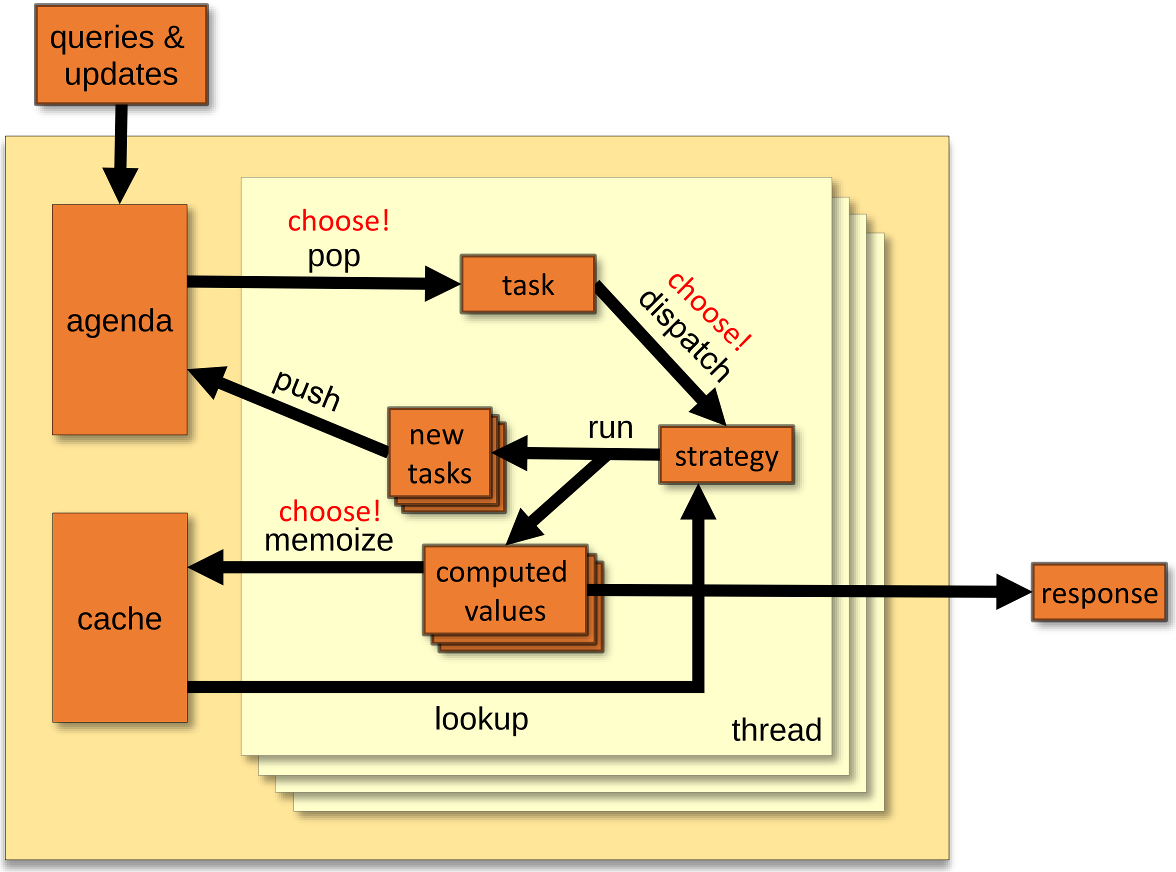
Learning to choose a good strategy



Learning to choose a good strategy

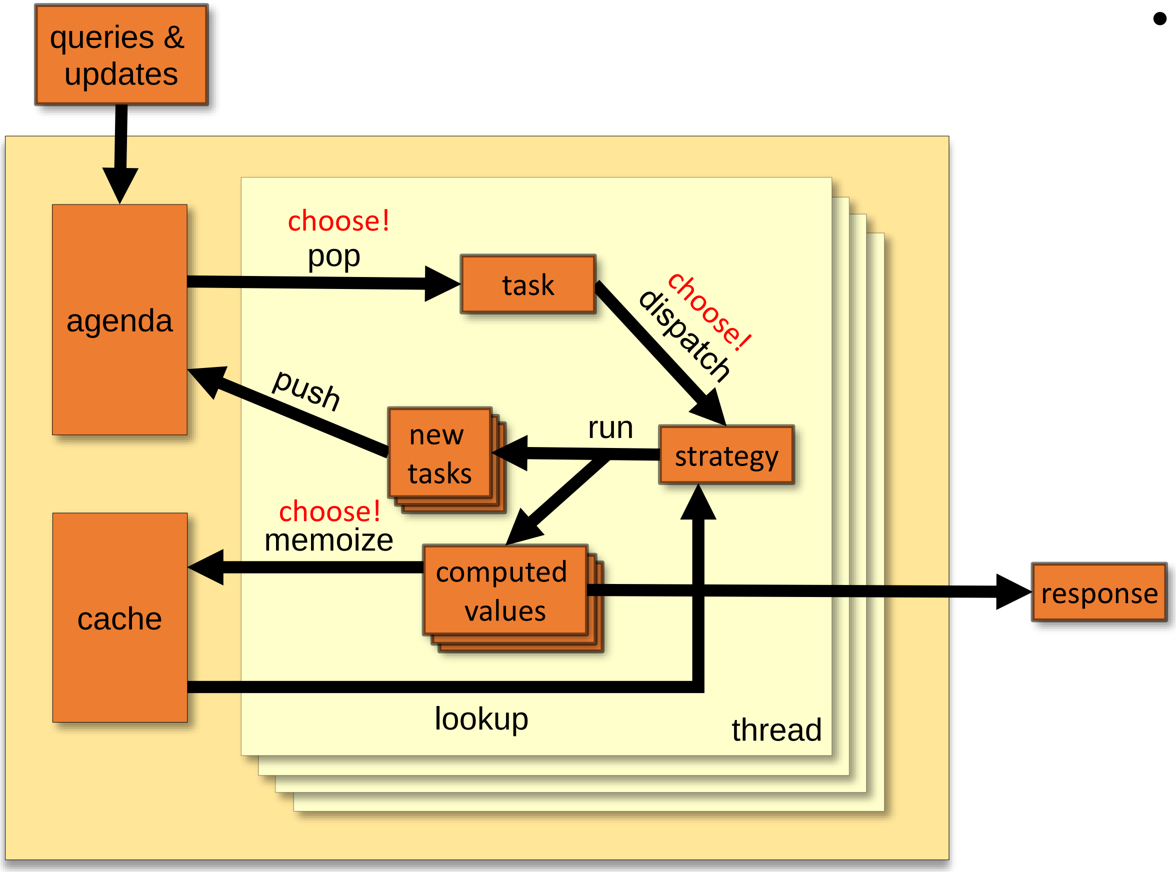


Learning to choose a good strategy



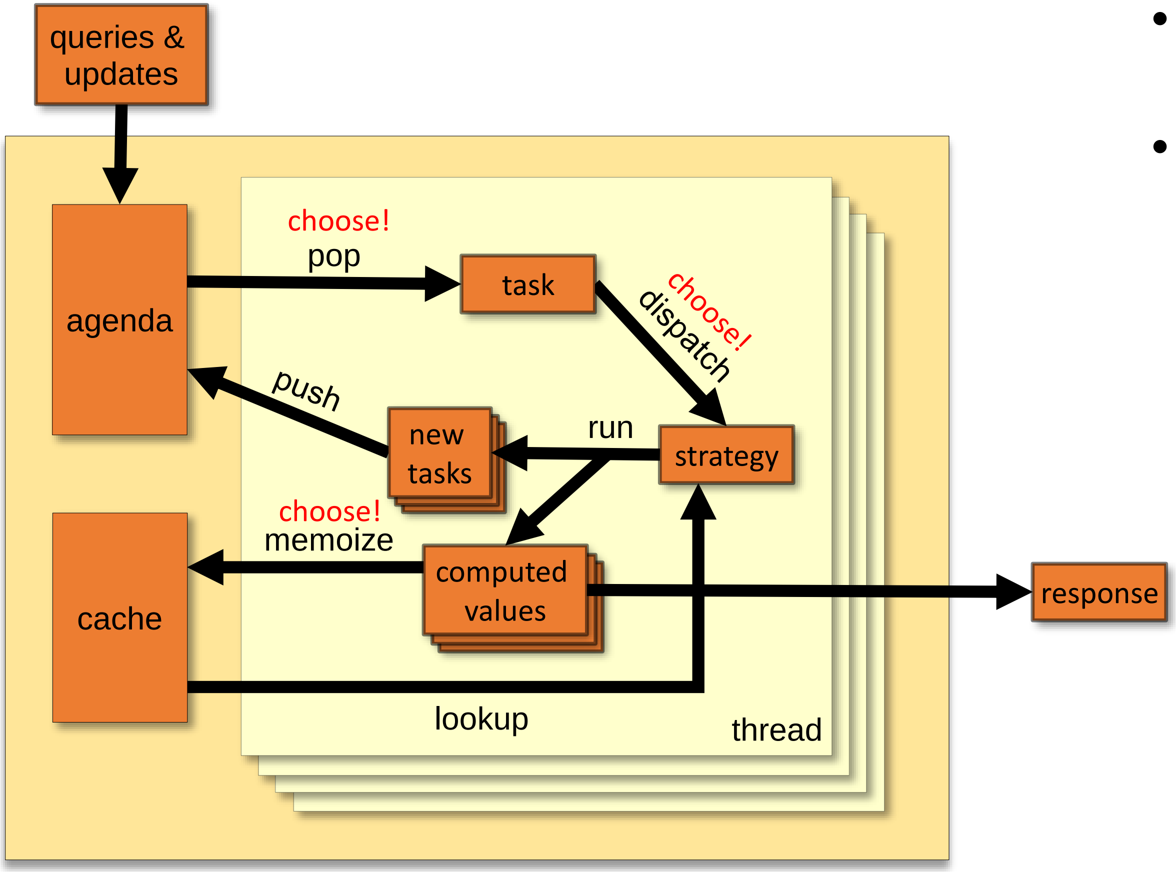
1. **Bandit**: Each time we execute a task (e.g., compute `c(I,j)` for an argument `j`):

Learning to choose a good strategy



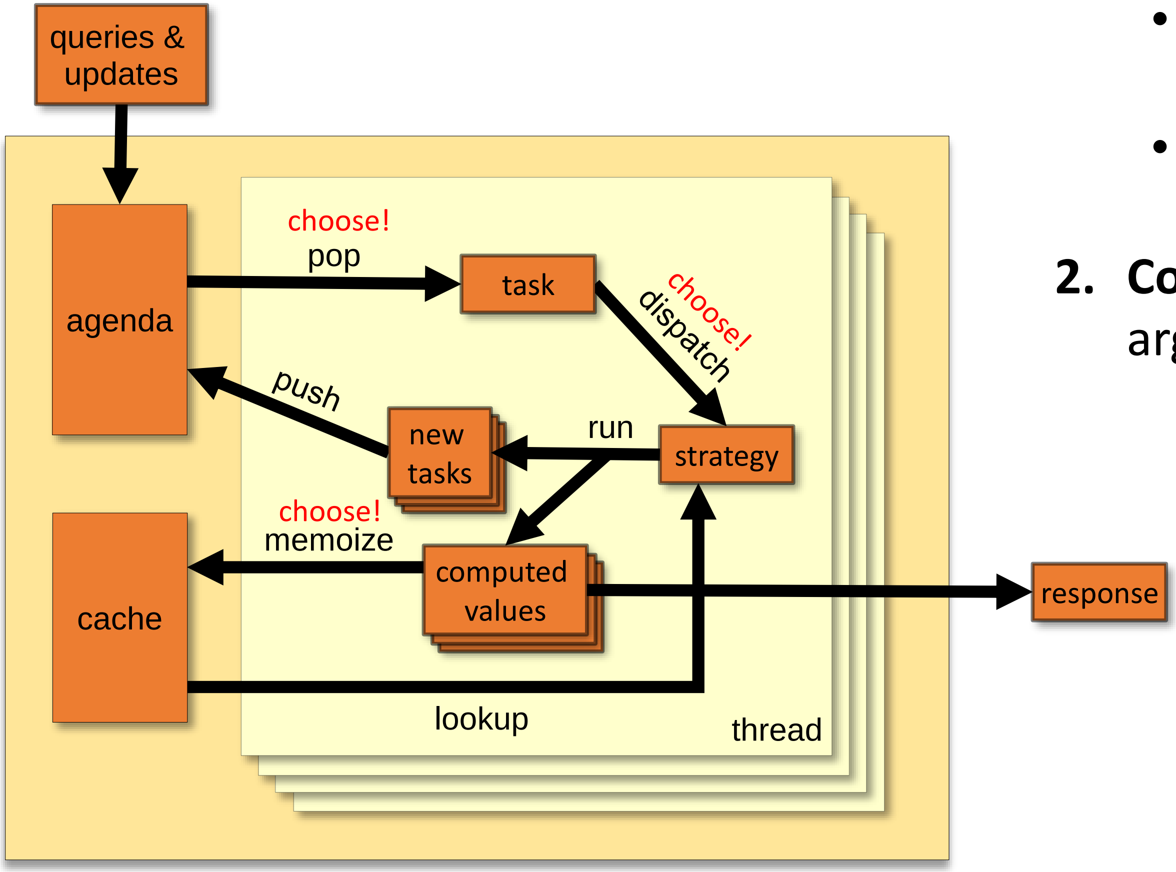
1. **Bandit**: Each time we execute a task (e.g., compute `c(I,j)` for an argument `j`):
    - Randomly try one of the available strategies (explore) according to some probability distribution

Learning to choose a good strategy



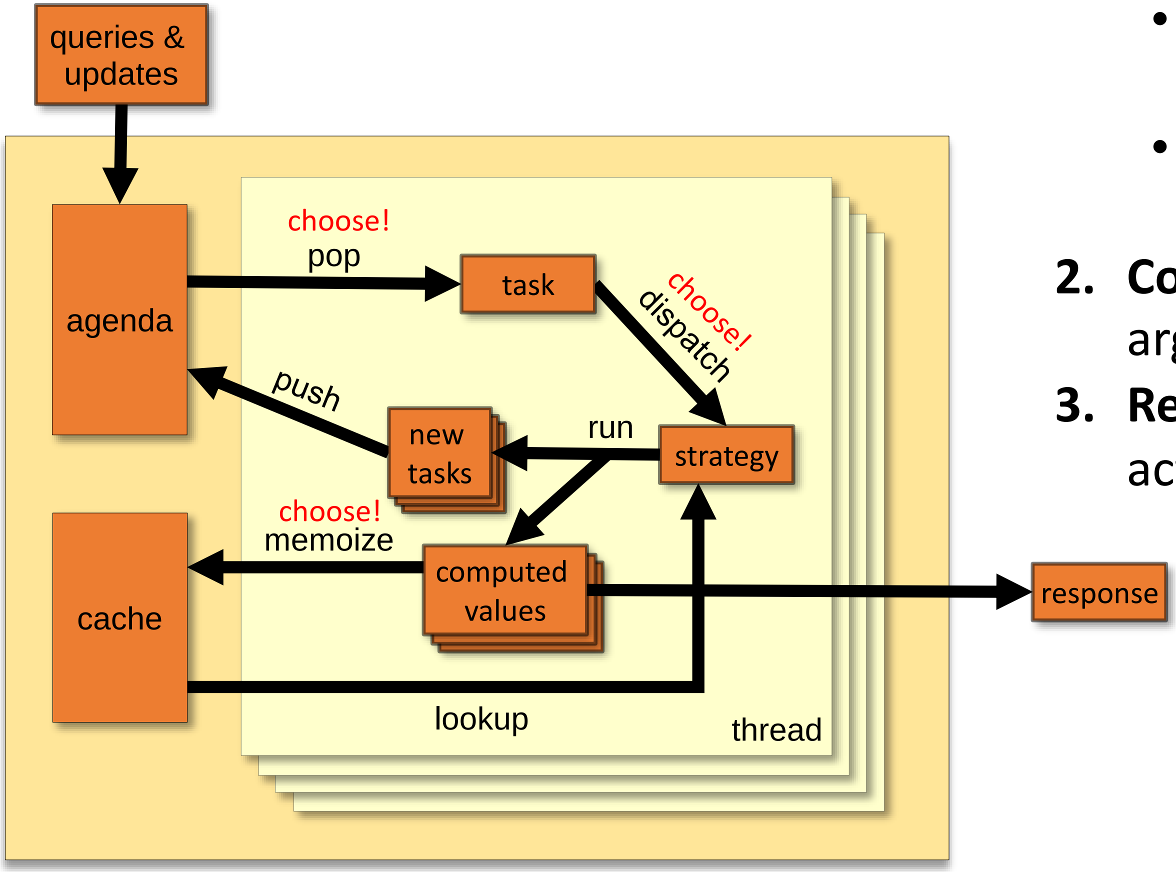
1. **Bandit**: Each time we execute a task (e.g., compute `c(I,j)` for an argument `j`):
   - Randomly try one of the available strategies (explore) according to some probability distribution
   - Bias this distribution in favor of strategies with lower measured cost (exploit).

Learning to choose a good strategy



1. **Bandit**: Each time we execute a task (e.g., compute `c(I,j)` for an argument `j`):
    - Randomly try one of the available strategies (explore) according to some probability distribution
    - Bias this distribution in favor of strategies with lower measured cost (exploit).
2. **Contextual bandit**: Allows distribution to depend on task arguments (e.g., `j`) and solver state.

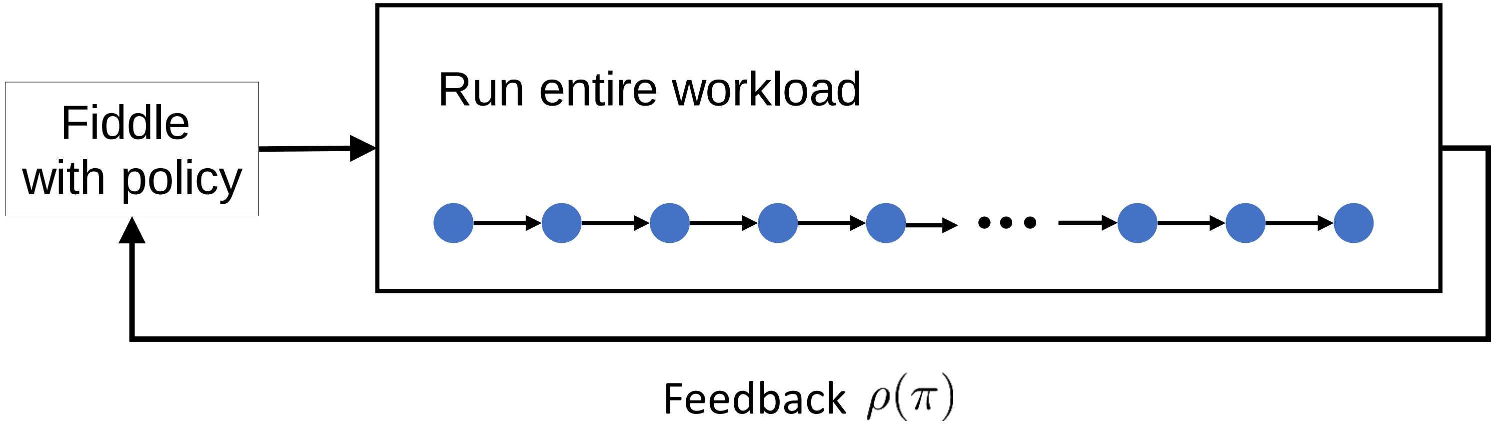
Learning to choose a good strategy



1. **Bandit**: Each time we execute a task (e.g., compute `c(I,j)` for an argument `j`):
   - Randomly try one of the available strategies (explore) according to some probability distribution
   - Bias this distribution in favor of strategies with lower measured cost (exploit).
2. **Contextual bandit**: Allows distribution to depend on task arguments (e.g., `j`) and solver state.
3. **Reinforcement learning**: Accounts for *delayed* costs of actions.

Back to online training…

Back to online training…



Fiddle with policy

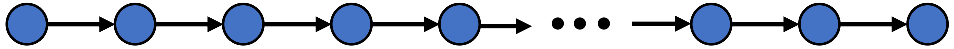
Run entire workload

Feedback $\rho(\pi)$

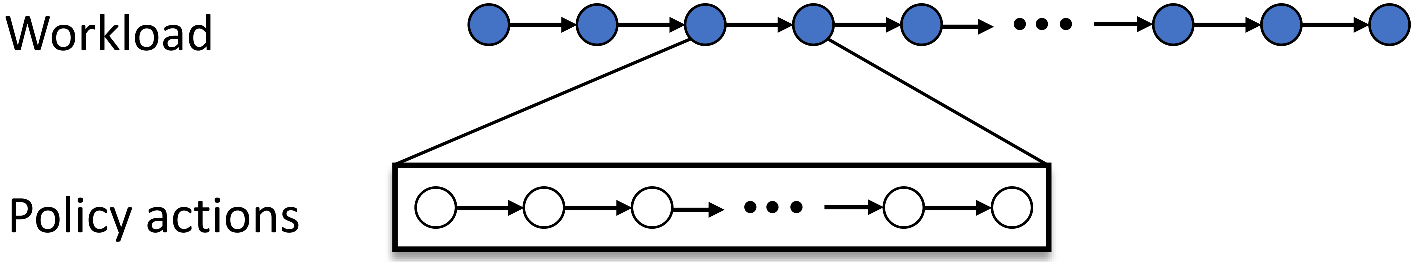
Solver Actions

Solver Actions

Workload



Solver Actions



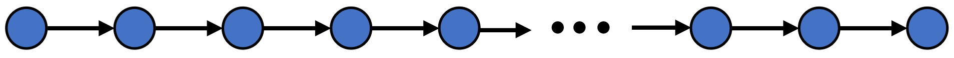
Rewrite the objective function

TODO inline in parens
use underbrace to give it a name

Add Jason integral picture?

Rewrite the objective function

Workload *i*

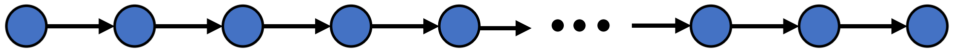


TODO inline in parens
use underbrace to give it a name

Add Jason integral picture?

Rewrite the objective function

Workload *i*



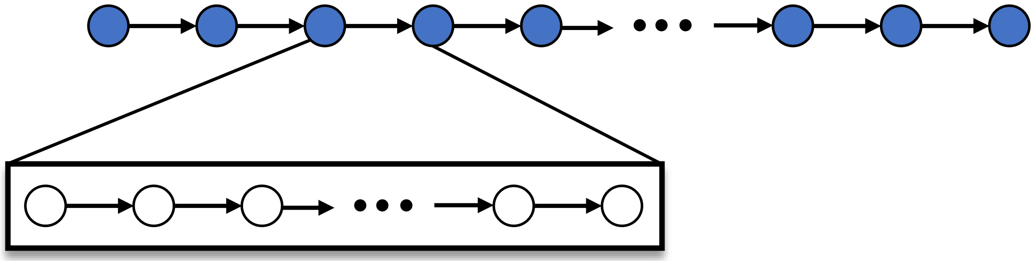
$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \,\mathrm{latency}(i)\right]$$

TODO inline in parens
use underbrace to give it a name

Add Jason integral picture?

Rewrite the objective function

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \, \text{latency}(i)\right]$$



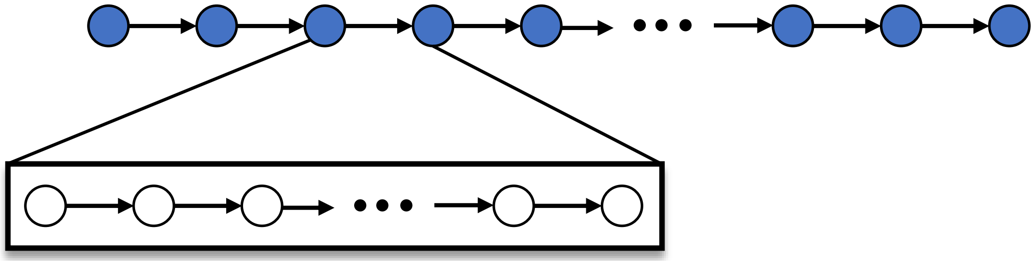
Workload *i*

Actions *t*

TODO inline in parens
use underbrace to give it a name

Add Jason integral picture?

Rewrite the objective function



Workload *i*

Actions *t*

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \,\mathrm{latency}(i)\right]$$

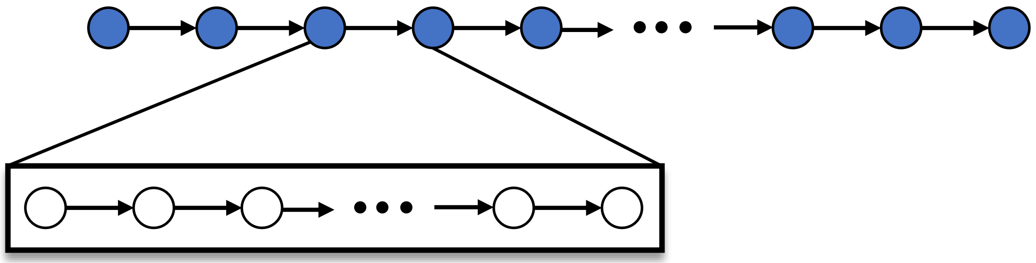
$$= \mathbb{E}\left[\sum_{t=1}^{\infty} \mathrm{load}(t) \cdot (\mathrm{clock}(t+1) - \mathrm{clock}(t))\right]$$

Rewrite in terms of the policy's time scale (used in RL)

TODO inline in parens
use underbrace to give it a name

Add Jason integral picture?

Rewrite the objective function



Workload *i*

Actions *t*

$$\rho(\pi) = \mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^i \lambda_i \,\mathrm{latency}(i)\right]$$

$$= \mathbb{E}\left[\sum_{t=1}^{\infty} \mathrm{load}(t) \cdot (\mathrm{clock}(t+1) - \mathrm{clock}(t))\right]$$

Rewrite in terms of the
policy's time scale (used in RL)

Each step tries to
decrease the load

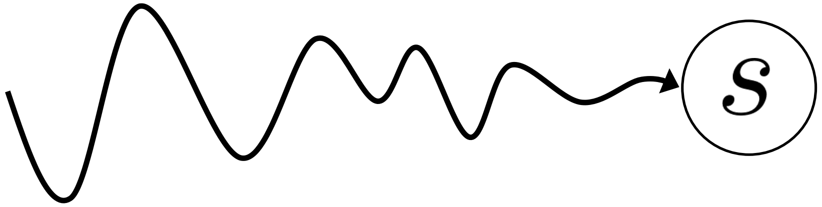
$$\mathrm{load}(t) = \sum_{i \in \mathcal{O}(t)} \gamma^i \lambda_i$$

TODO inline in parens
use underbrace to give it a name

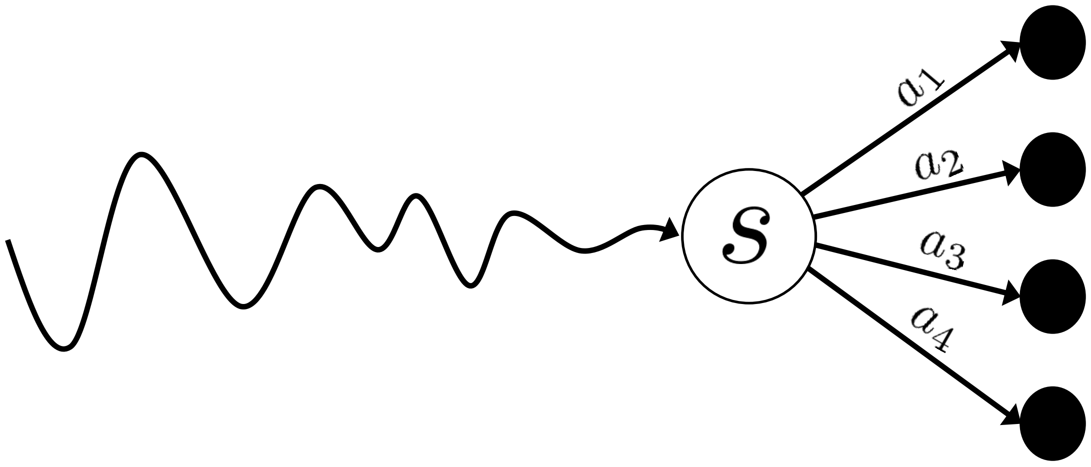
Add Jason integral picture?

Life of π

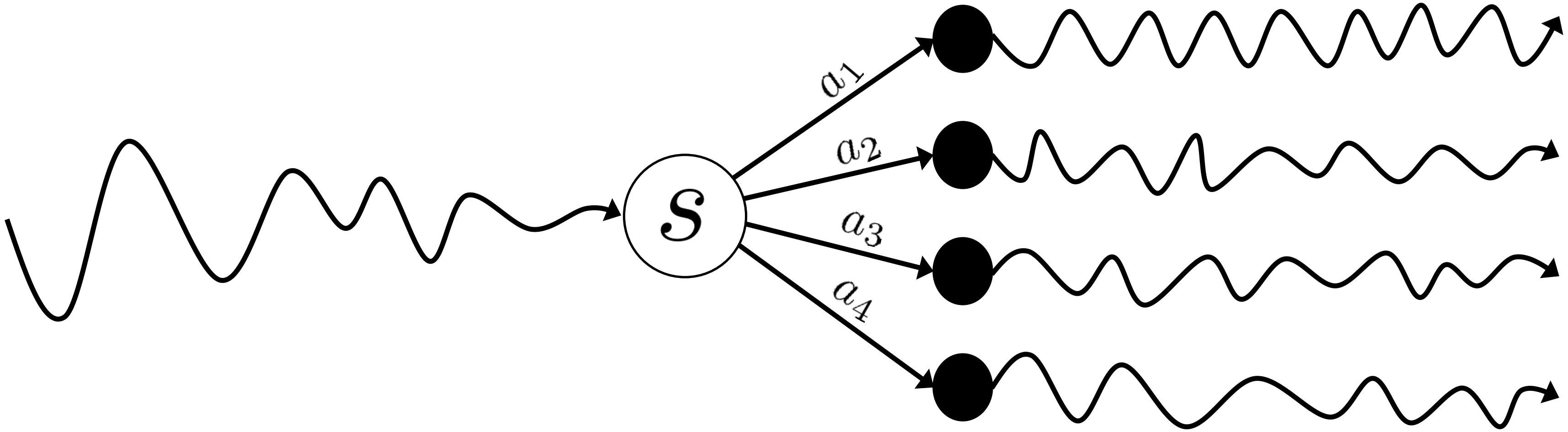
Life of π



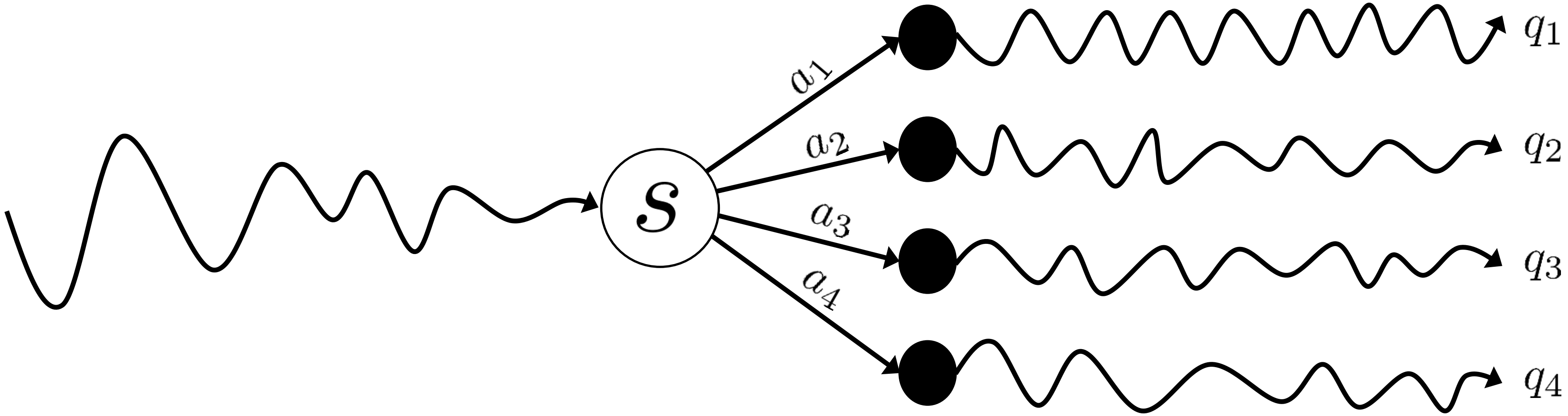
Life of π



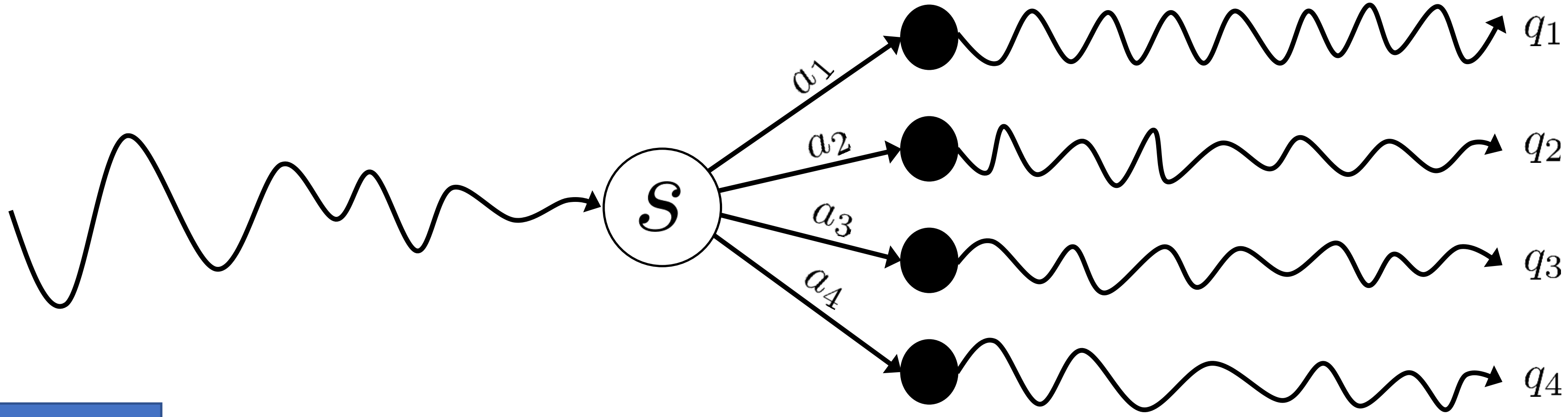
Life of π



Life of π



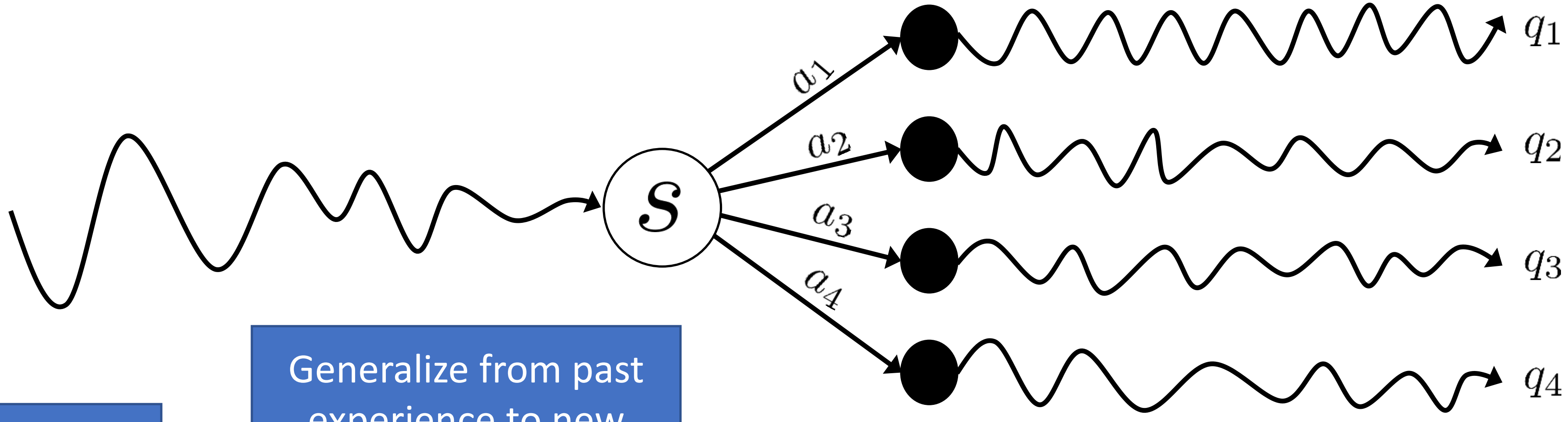
Life of π



Use ML to predict future costs!

$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i)$$

Life of π

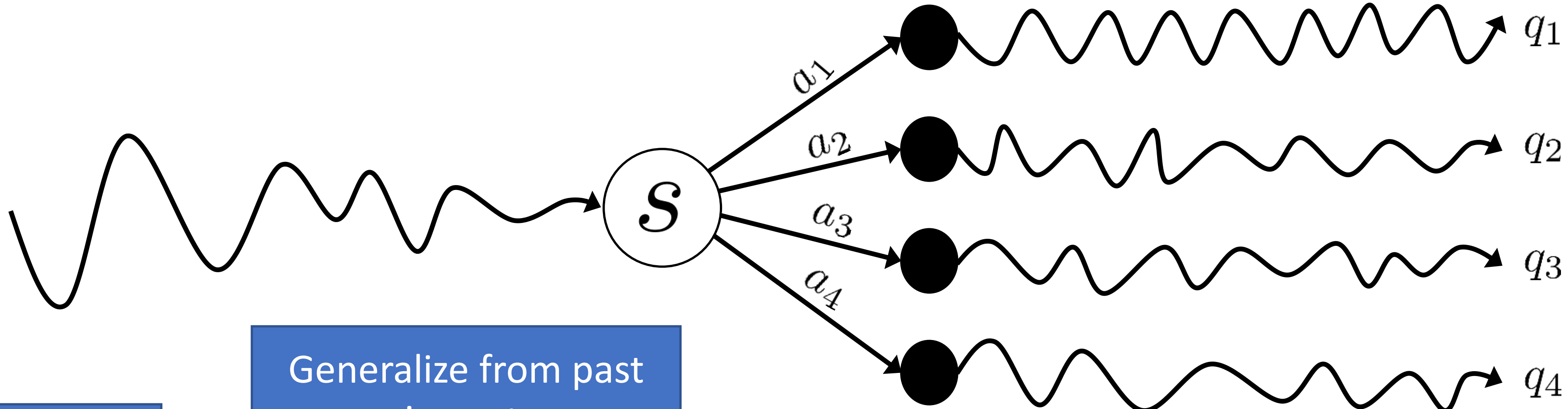


Use ML to predict future costs!

Generalize from past experience to new situations

$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i)$$

Life of π

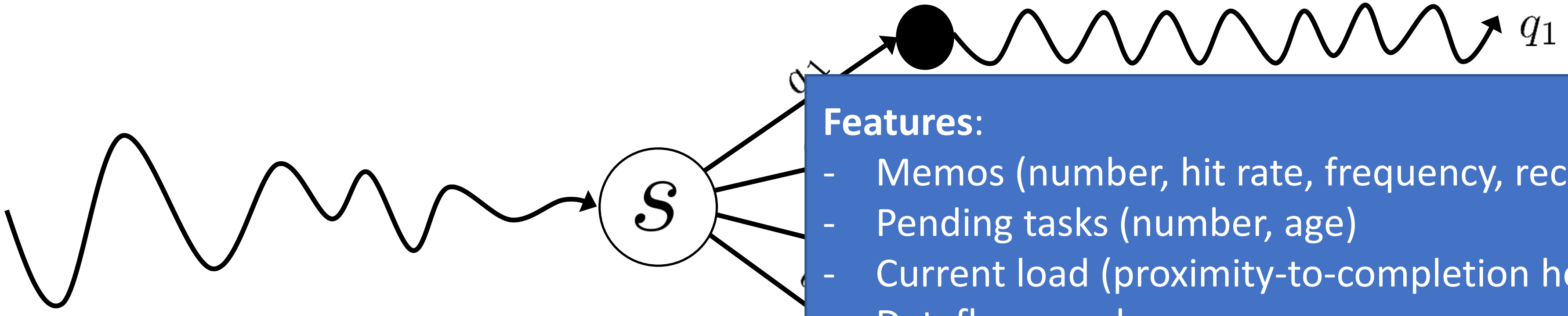


Use ML to predict future costs!

Generalize from past experience to new situations

$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i) = w^\top \Phi(s, a_i)$$

Life of π



Features:
- Memos (number, hit rate, frequency, recency)
- Pending tasks (number, age)
- Current load (proximity-to-completion heuristics)
- Dataflow graph
  - Will this be used again?
  - Are other jobs blocking on it?

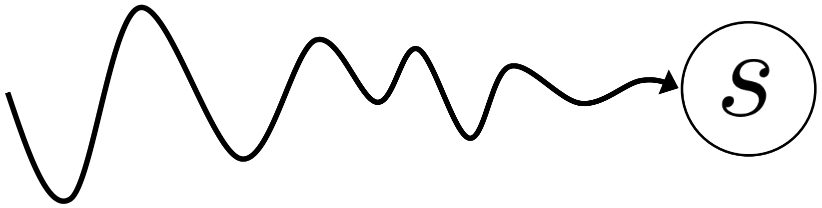
Use ML to predict future costs!

Generalize from past experience to new situations

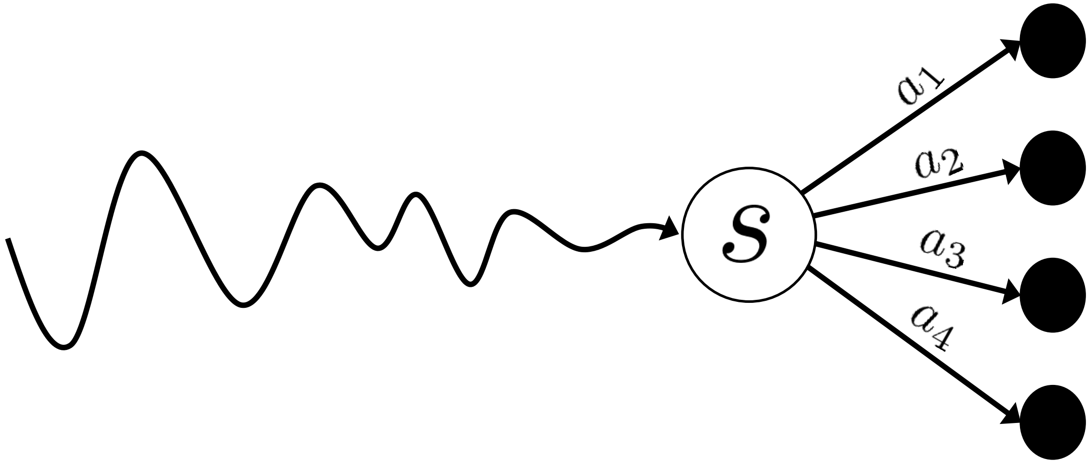
$$\mathbb{E}\left[q_i\right] \approx \widehat{q}(s, a_i) = w^\top \Phi(s, a_i)$$

Back to π

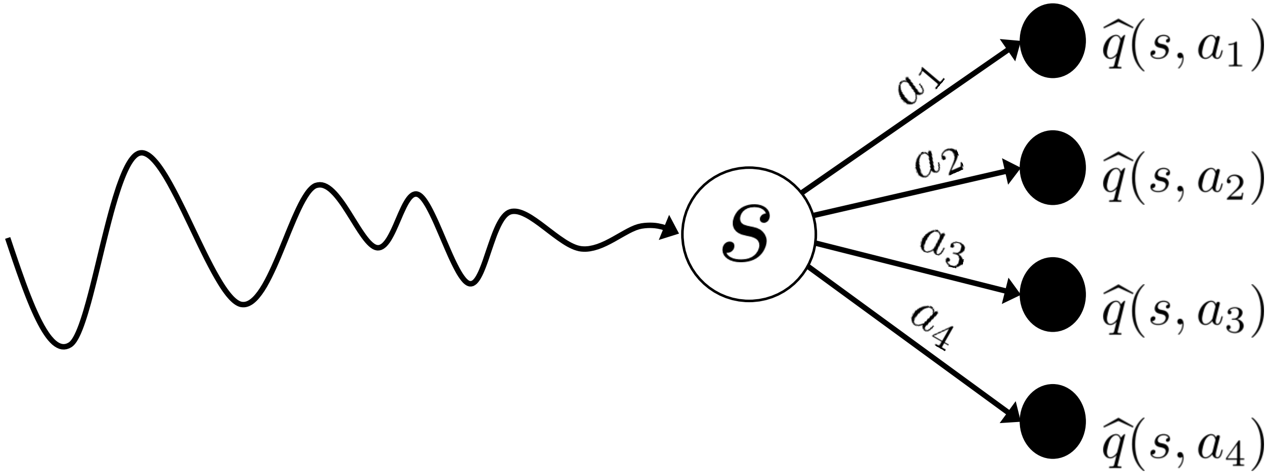
Back to π



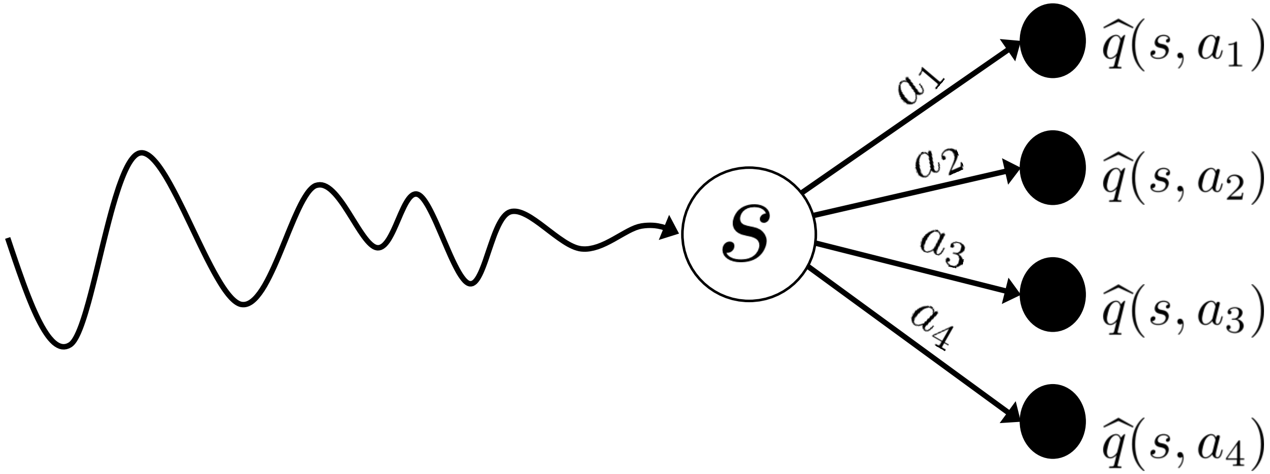
Back to π



Back to π

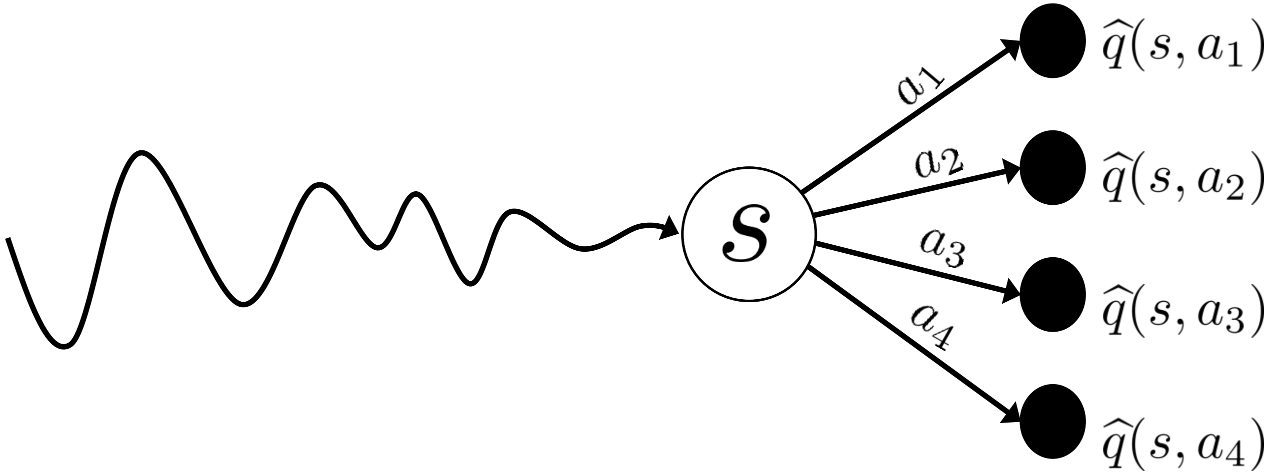

s
a_1
a_2
a_3
a_4
$\widehat{q}(s, a_1)$
$\widehat{q}(s, a_2)$
$\widehat{q}(s, a_3)$
$\widehat{q}(s, a_4)$

Back to π



What should π do in this state?

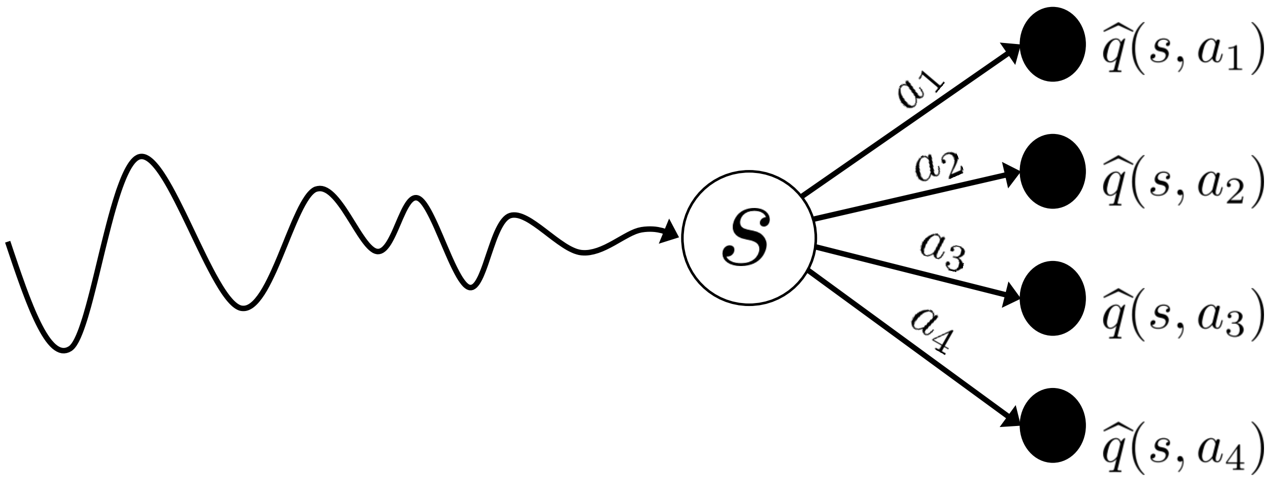
Back to π



What should π do in this state?

$$\pi(s) = \underset{a}{\operatorname{argmin}} \ \widehat{q}(s, a)$$

Back to π

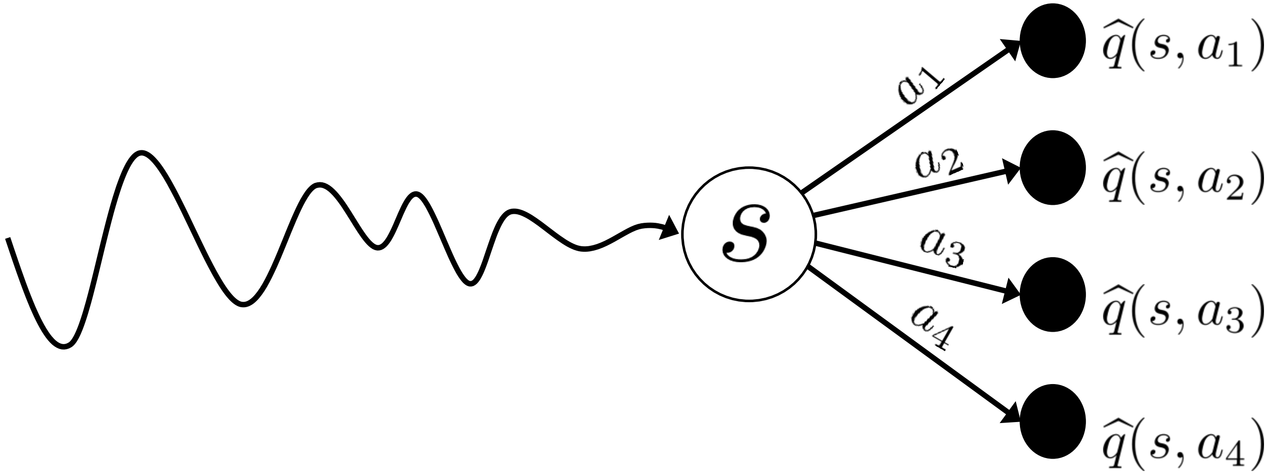


What should π do in this state?

$$\pi(s) = \underset{a}{\operatorname{argmin}}\ \widehat{q}(s, a)$$

Predicting the future requires richer features than simply learning to take good actions.

Back to π

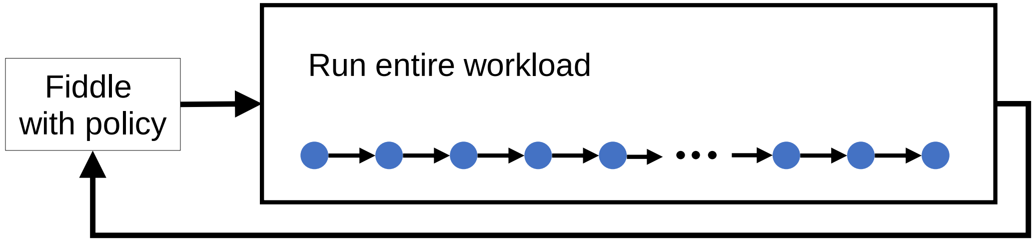


What should π do in this state?

$$\pi(s) = \underset{a}{\operatorname{argmin}}\ \widehat{q}(s, a)$$

Predicting the future requires richer features than simply learning to take good actions.

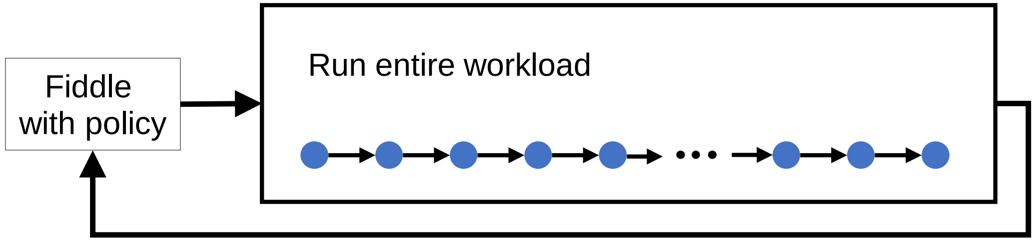
We need π to be really fast to execute!



Fiddle with policy

Run entire workload

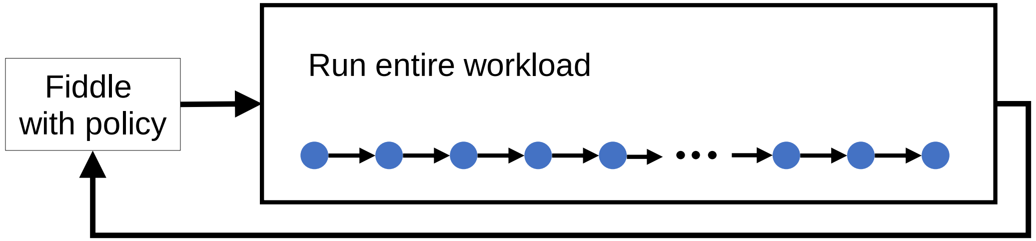
Feedback $\rho(\pi)$



Fiddle with policy

Run entire workload

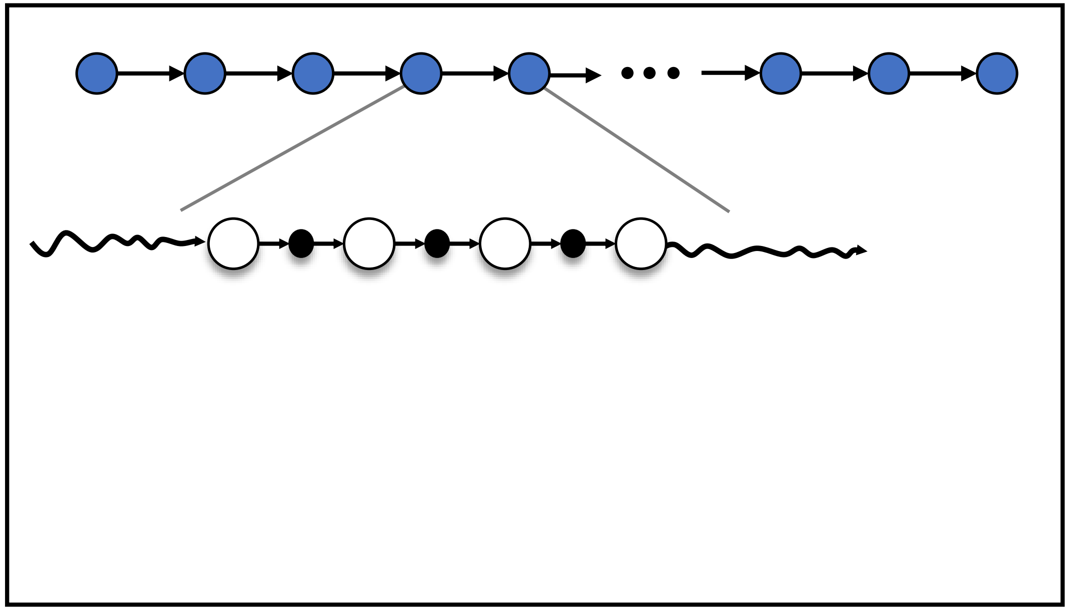
Feedback $\rho(\pi)$

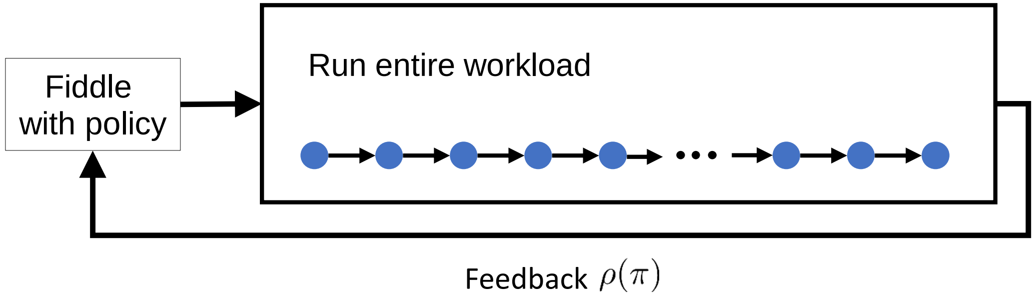


Fiddle with policy

Run entire workload

Feedback $\rho(\pi)$

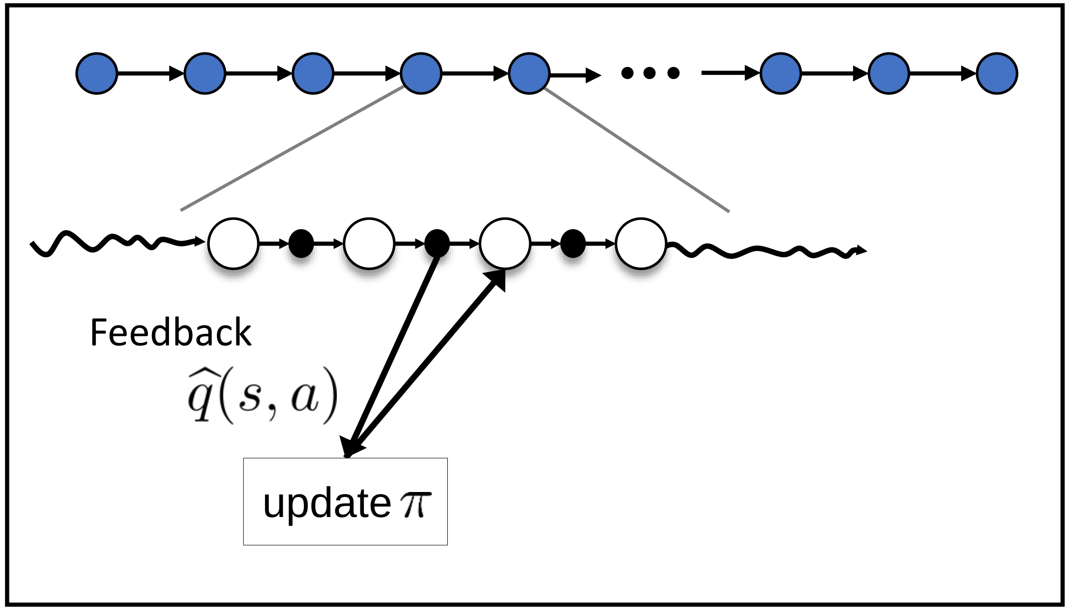




Fiddle with policy

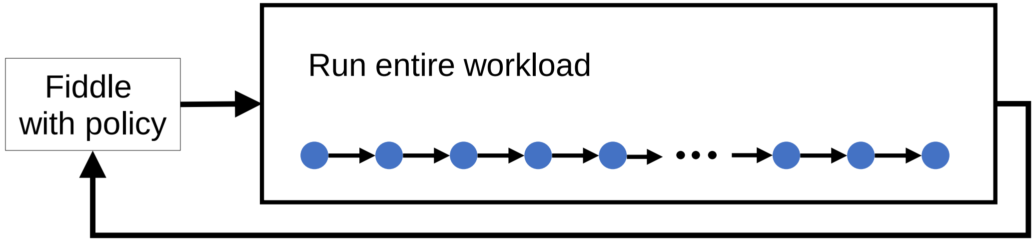
Run entire workload

Feedback $\rho(\pi)$



Feedback $\widehat{q}(s,a)$

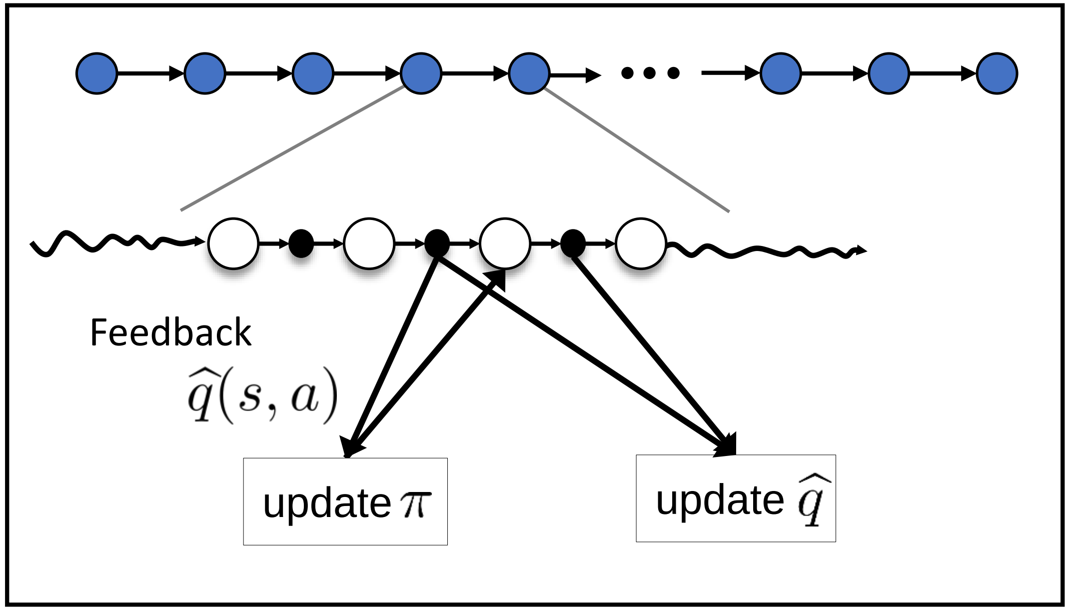
update π



Fiddle with policy

Run entire workload

Feedback $\rho(\pi)$



Feedback $\widehat{q}(s, a)$

update π

update $\widehat{q}$