Towards a Hands-Free Query Optimizer through Deep Reinforcement Learning

Ryan Marcus*, Olga Papaemmanouil
Brandeis University

@RyanMarcus
ryan@cs.brandeis.edu

These slides: http://rm.cab/cidr19

Towards a Hands-Free Query Optimizer through Deep Reinforcement Learning

(putting Eugene Wu out of work)

Ryan Marcus*, Olga Papaemmanouil
Brandeis University

@RyanMarcus
ryan@cs.brandeis.edu

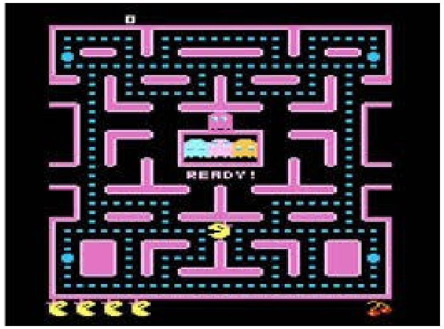
These slides: http://rm.cab/cidr19

Query Optimizers

- Extremely complex to develop
  - PostgreSQL: 40k LOC (12/27/2018)
  - SQL Server & Vertica: much higher
- Requires DBA tuning
  - *Thousands* of knobs (probably ~50 require changes)
- **Optimizer = expert system. Can we learn it instead?**

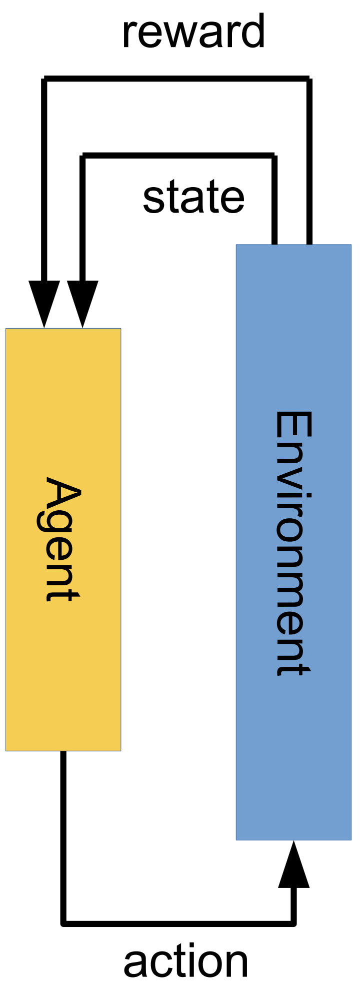
Learning Expert Systems

- Past 5 years: huge explosion in deep reinforcement learning
- AlphaGo, PPO, DQN, etc.
- Outperforming expert systems



Reinforcement Learning

- Agent observes a *state*
  - Info about the world
  - Set of possible actions
- Agent selects an action, gets:
  - A reward
  - New state
- Goal: maximize reward over time



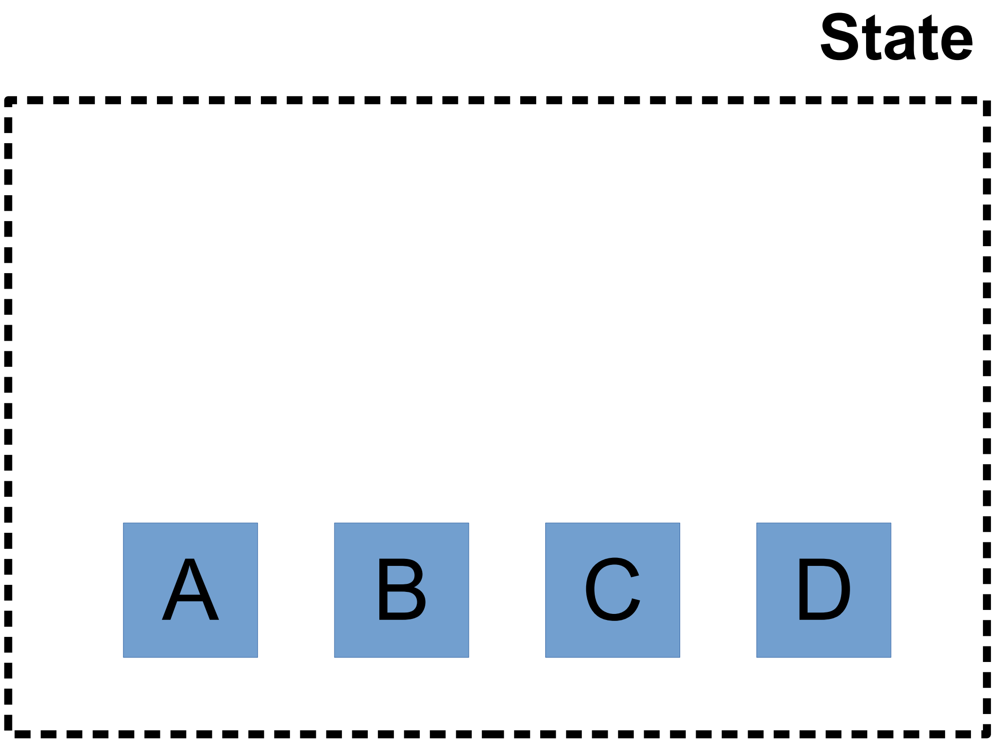
Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

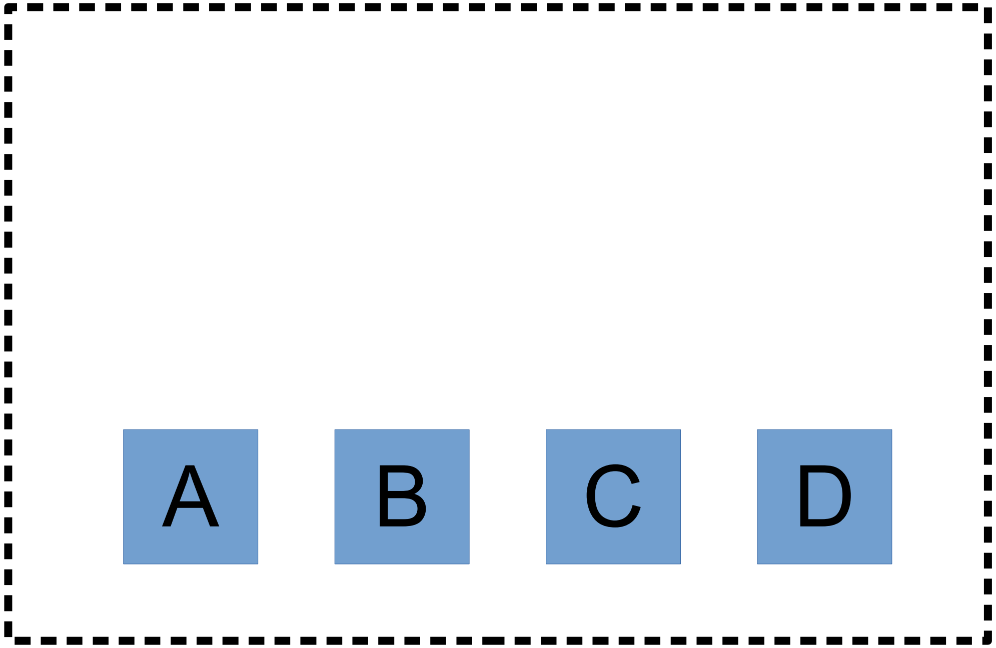


SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

State



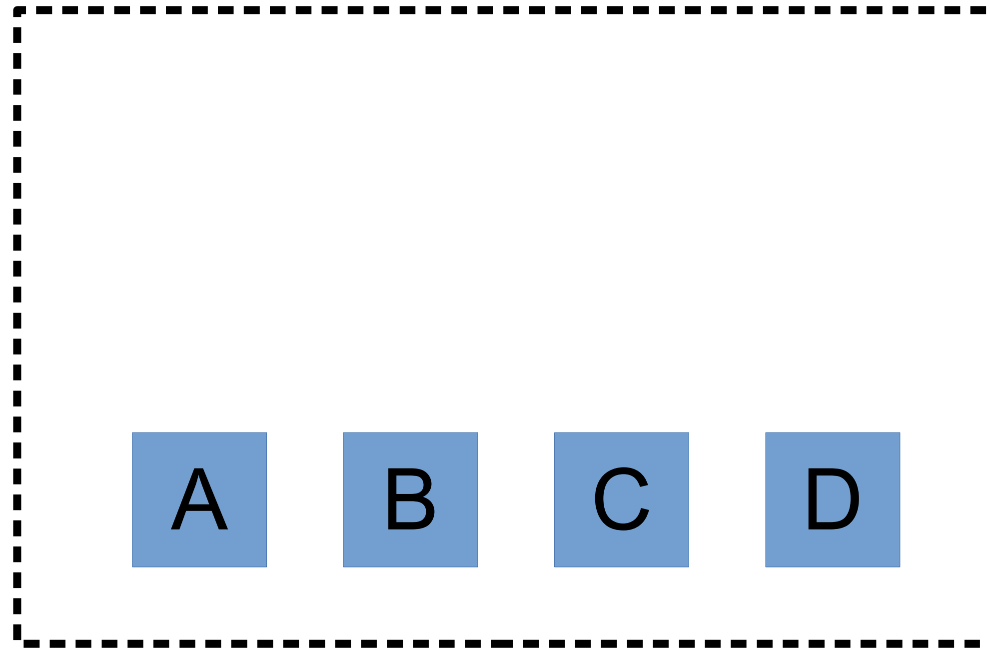
Possible actions:
(A, B), (B, A), (A, C), (C, A), (A, D), (D, A), (B, C), (C, B), (B, D), (D, B), (C, D), (D, C)

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

State



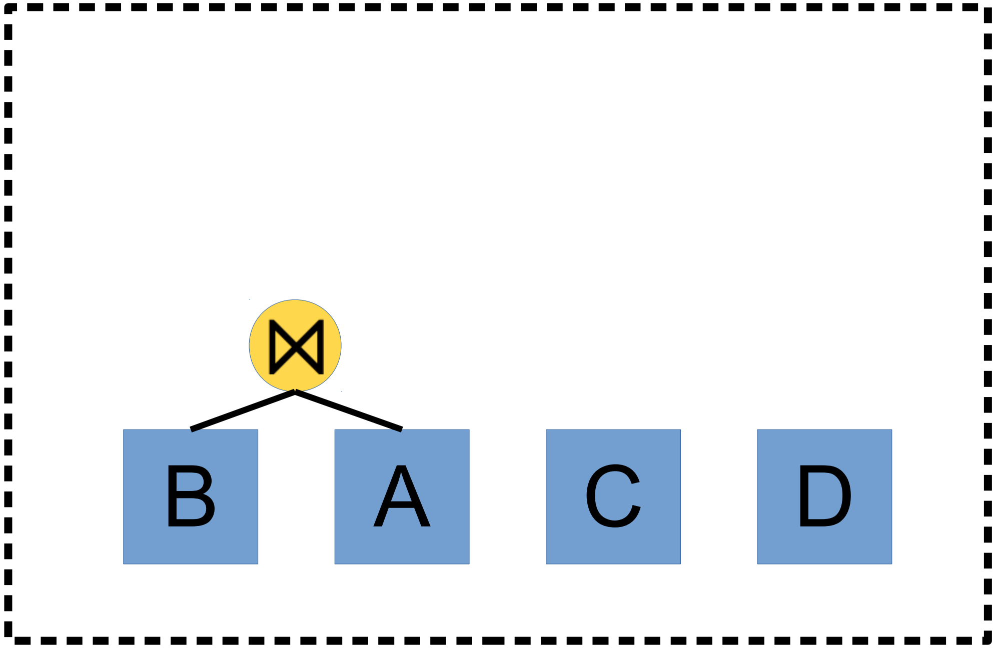
Possible actions:
(A, B), **(B, A)**, (A, C), (C, A), (A, D), (D, A), (B, C), (C, B), (B, D), (D, B), (C, D), (D, C)

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

State

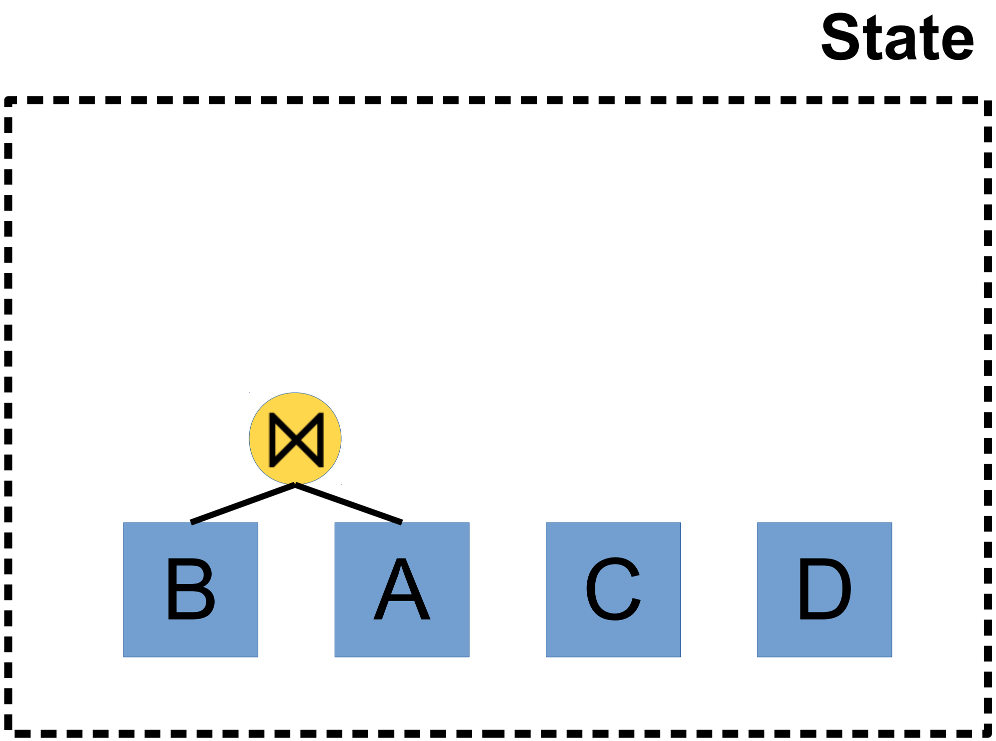


Possible actions:
([BA], C), (C, [BA]), ([BA], D), (D, [BA]), (C, D), (D, C)

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency



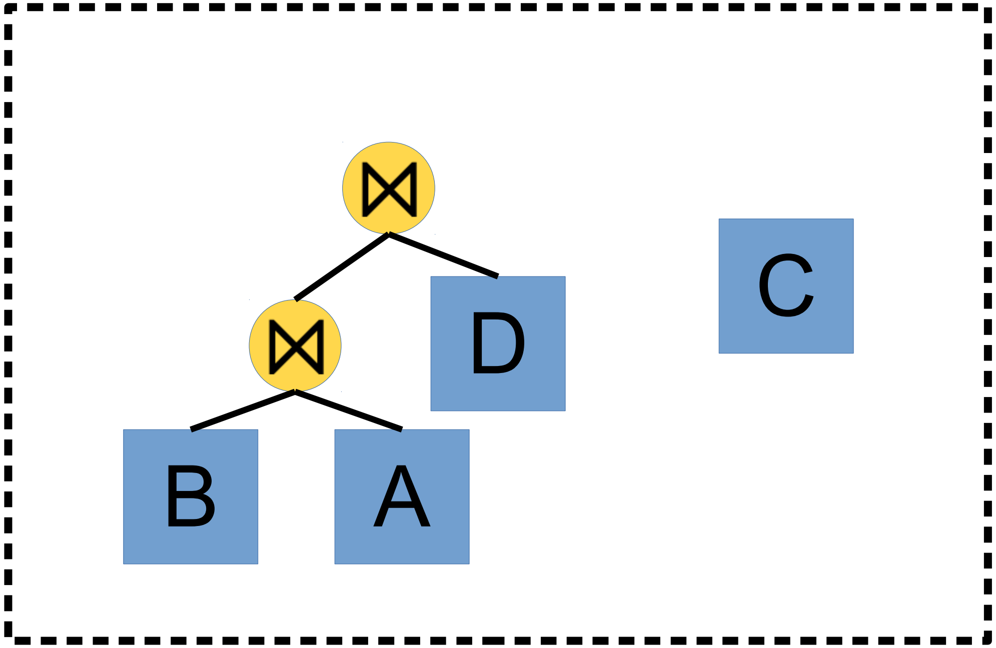
Possible actions:
([BA], C), (C, [BA]), **([BA], D)**, (D, [BA]), (C, D), (D, C)

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

State



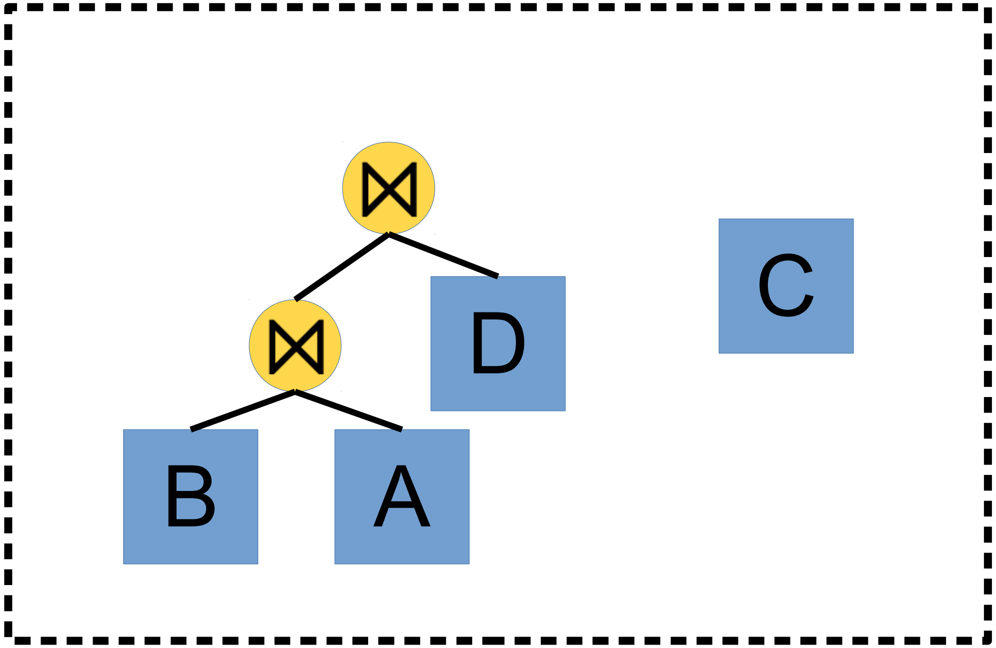
Possible actions:
([[BA]D], C), (C, [[BA]D])

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

State



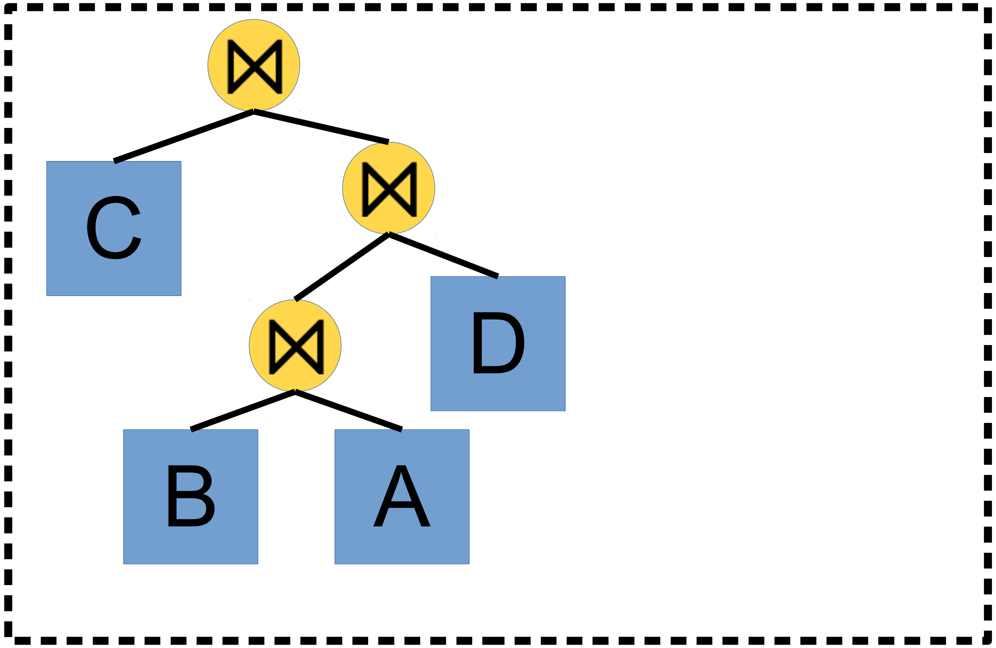
Possible actions:
([[BA]D], C), **(C, [[BA]D])**

SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

Reinforcement Learning

- Each state is a partial join order
- Each action fuses two partial orderings
- Reward is the query latency

State



Possible actions:

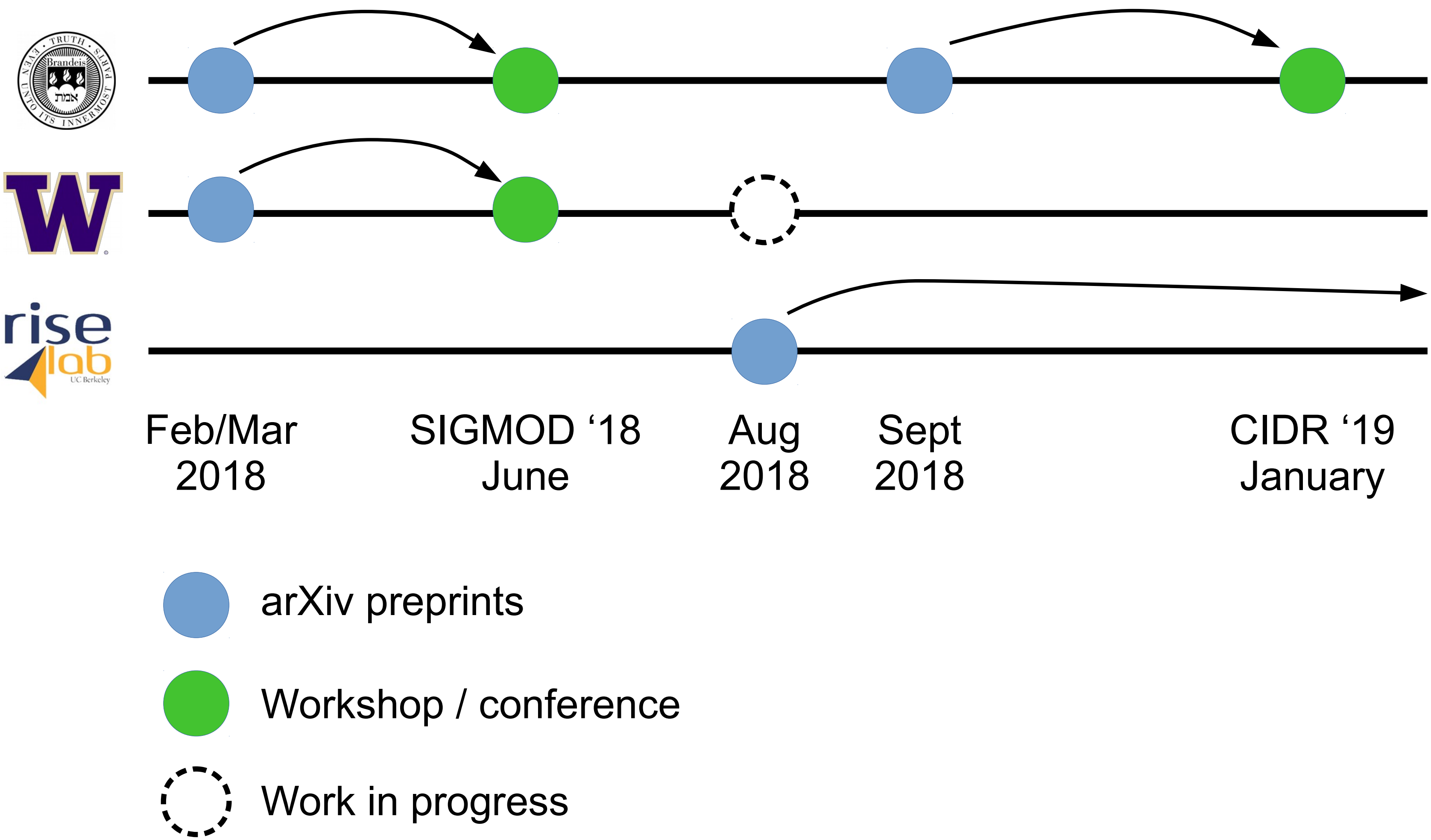
SELECT * FROM A, B, C, D WHERE A.attr1 = B.attr2 AND …;

The Dream

- We’ve described QO (partially) as an RL problem. So what?
- *Replace* optimizers with *off-the-shelf* deep reinforcement learning algorithm
- Totally “hands-free” – no configuration required.
  - Automatically tune to each DBMS
    - Column store, row store, XYZ-store…
  - Automatically adapt to shifts in workload

The Reality

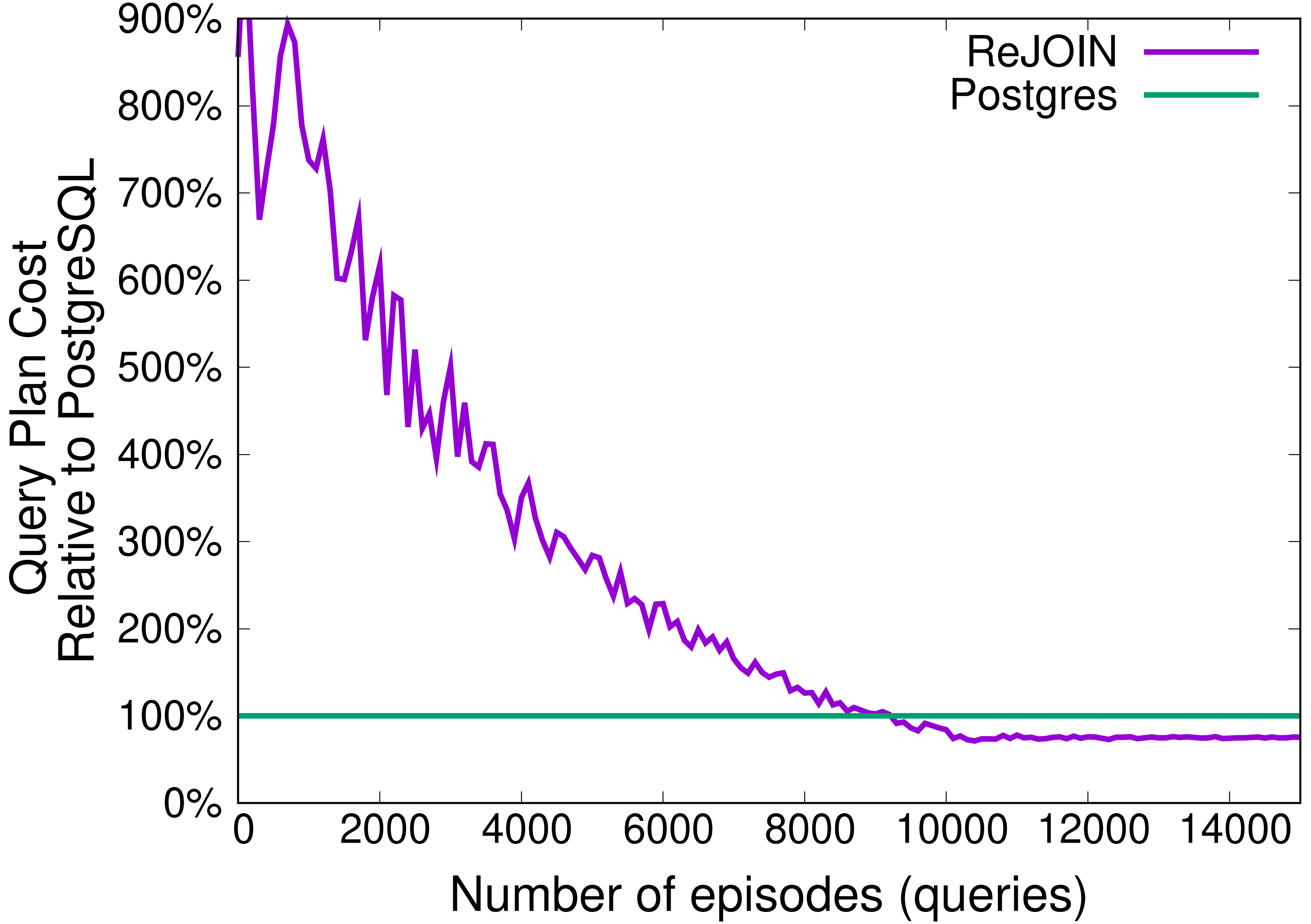
- Rapid, multi-faceted progress!



The Reality

- ReJOIN: deep reinforcement learning for join order enumeration
  - `http://rm.cab/rejoin`
- Promising results
  - Better join orderings than Postgres
- Problems
  - Only does join orderings
  - Uses optimizer cost model as a reward

ReJOIN



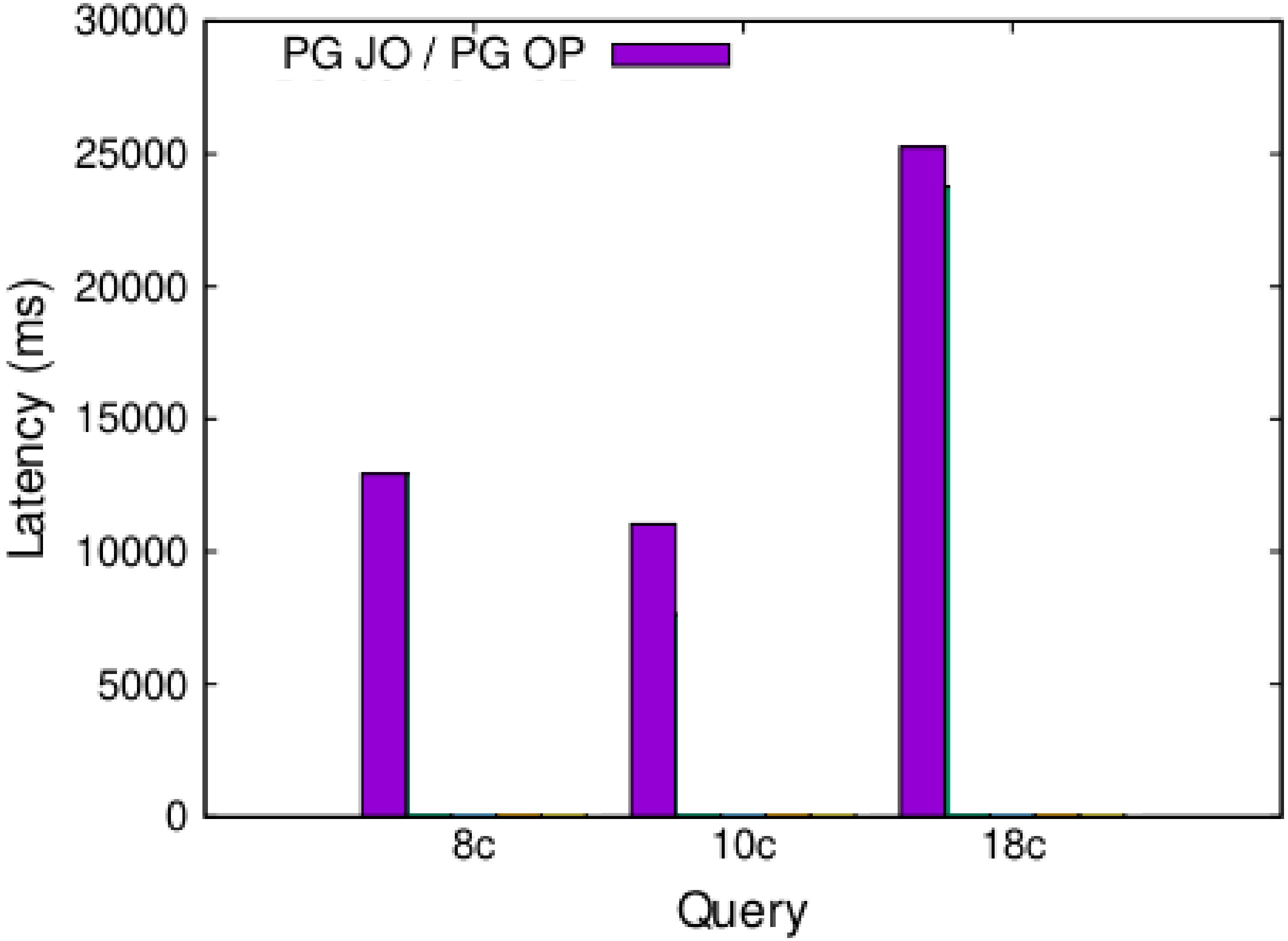
ReJOIN
Postgres
Query Plan Cost
Relative to PostgreSQL
900%
800%
700%
600%
500%
400%
300%
200%
100%
0%
0
2000
4000
6000
8000
10000
12000
14000
Number of episodes (queries)

Beyond Join Orders

- Problem 1: ReJOIN only does join order enumeration.
- Other optimizer decisions
  - Join operator selection?
  - Index selection?
  - Aggregate operator selection?
  - Early vs. late materialization?

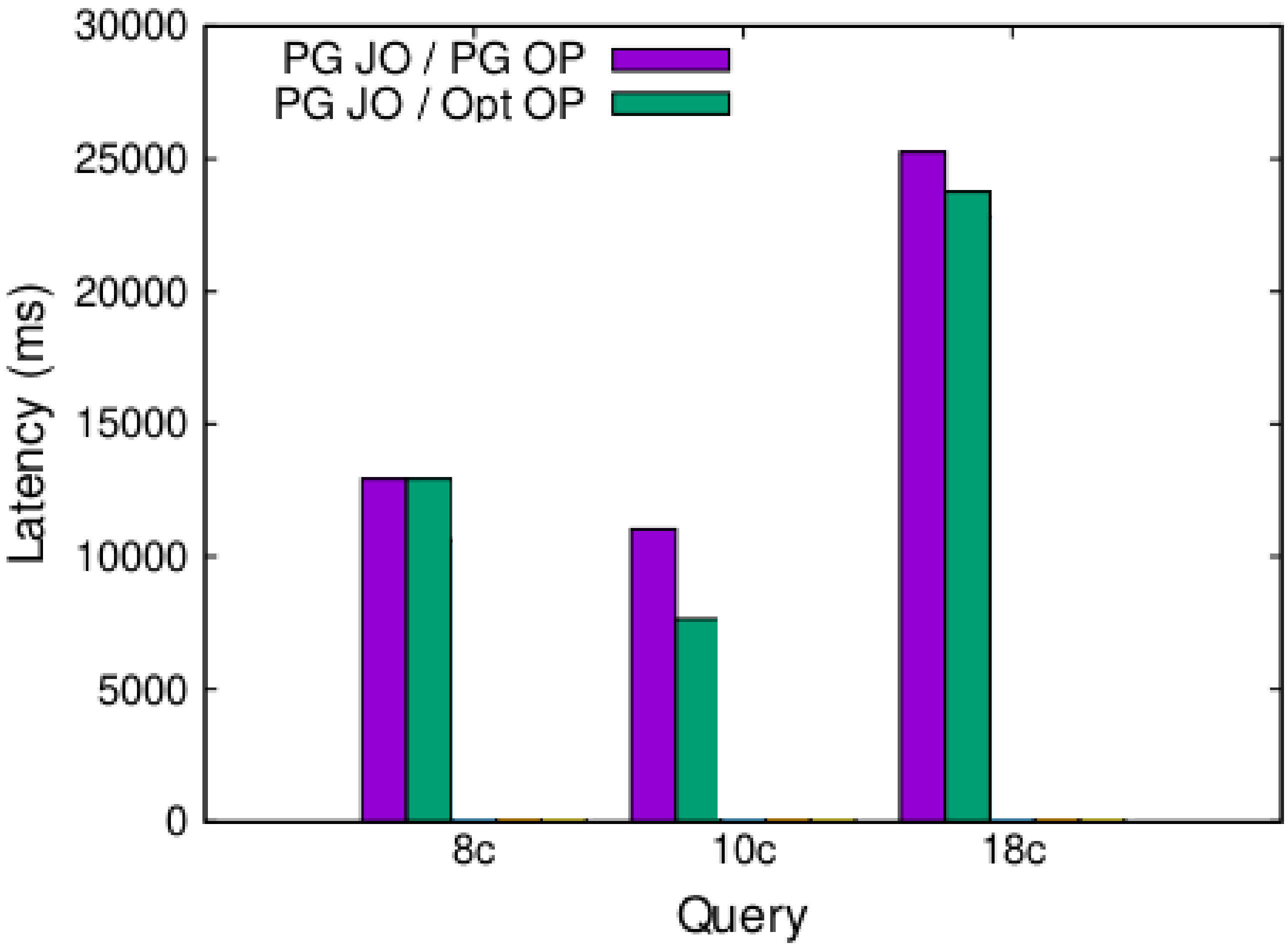
Beyond Join Orders

- Who cares? Join order is the hard part.
  - Yes and no...



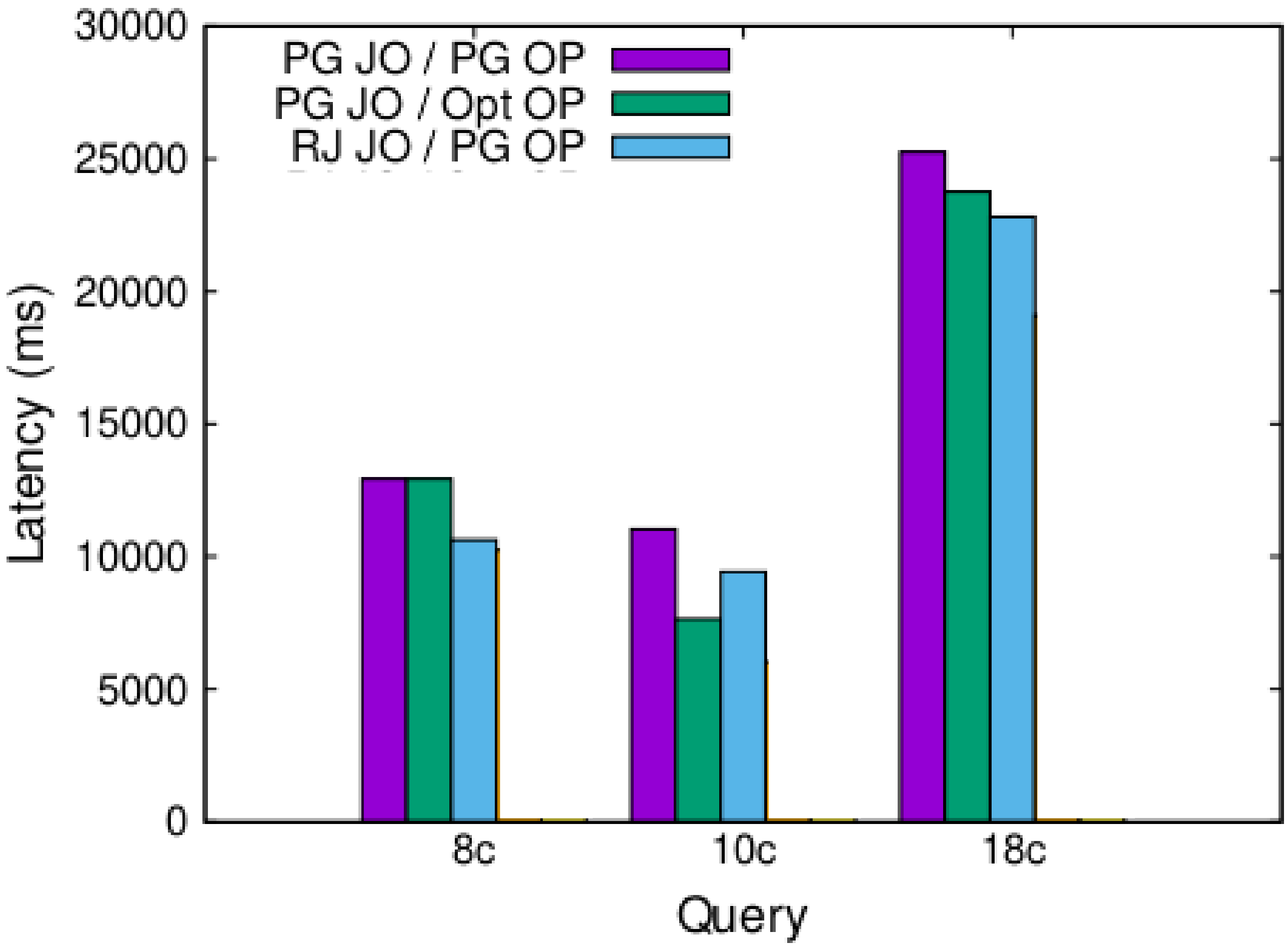
Beyond Join Orders

- Who cares? Join order is the hard part.
  - Yes and no...



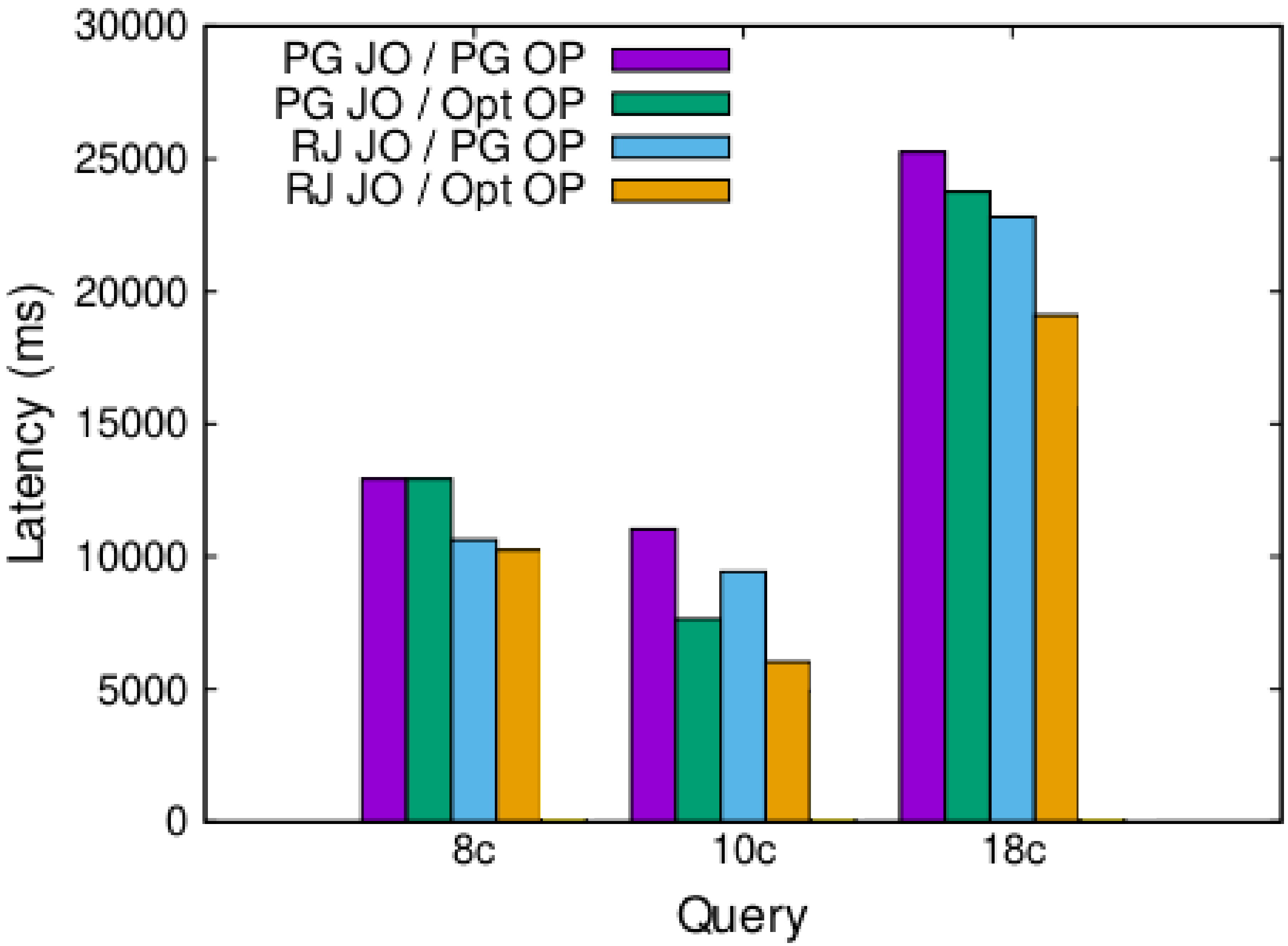
Beyond Join Orders

- Who cares? Join order is the hard part.
  - Yes and no...



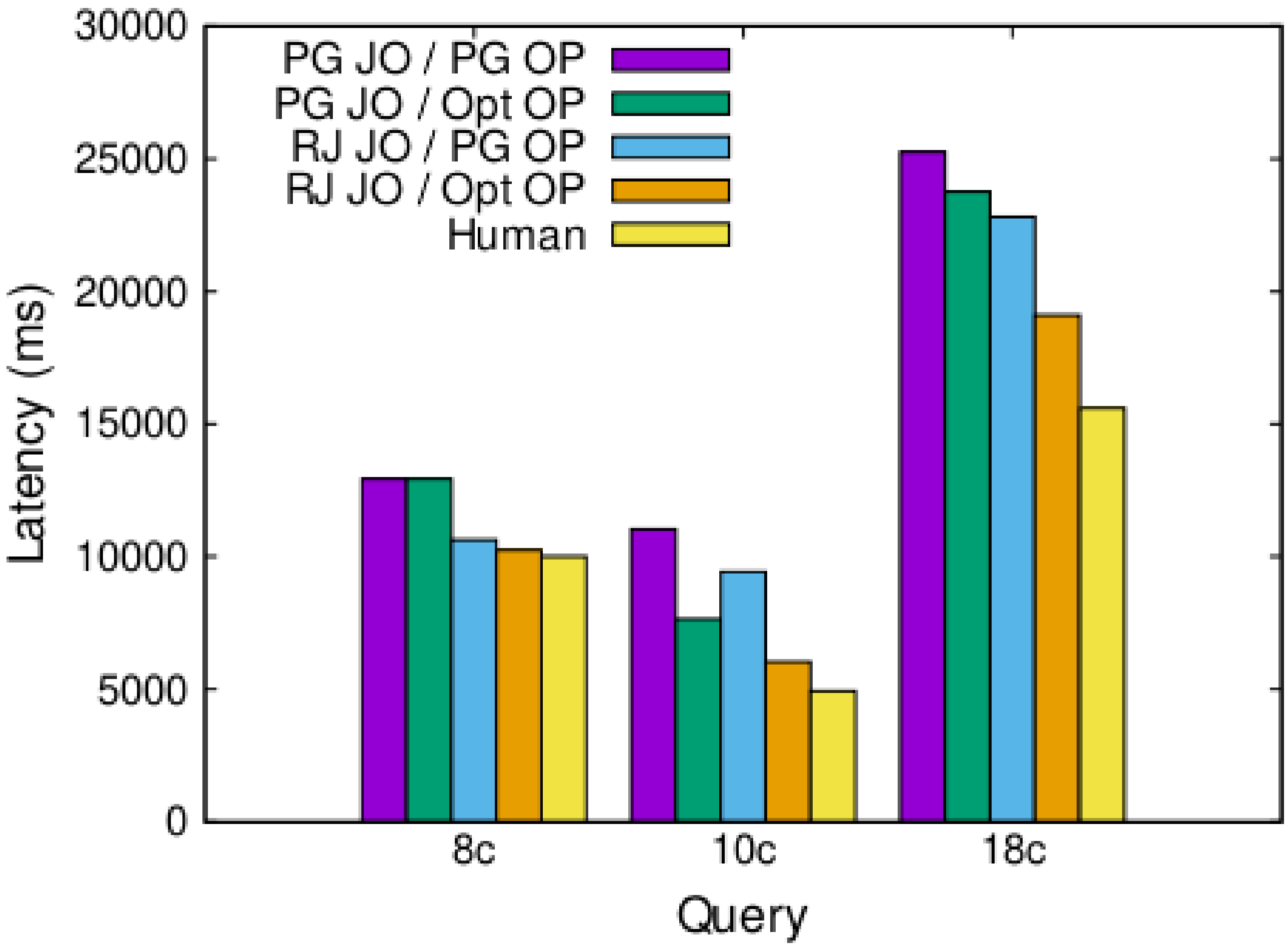
Beyond Join Orders

- Who cares? Join order is the hard part.
  - Yes and no...



Beyond Join Orders

- Who cares? Join order is the hard part.
  - Yes and no...



Cost Models

- Problem 2: ReJOIN depends on a cost model.
  - Cost models are complex, require development effort, tuning, etc.

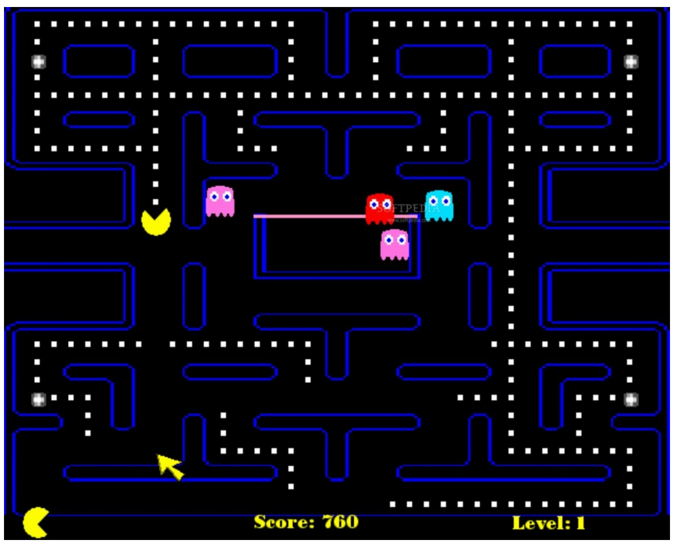
Why won't ReJOIN work?

- Why can't we just use the same approach as before?
  - Expand the action set
  - Plug in query latency as the reward signal
- In short, because the query latency doesn't behave well as a reward signal.

- Bad plans are *really* bad
- Rewards are *sparse*

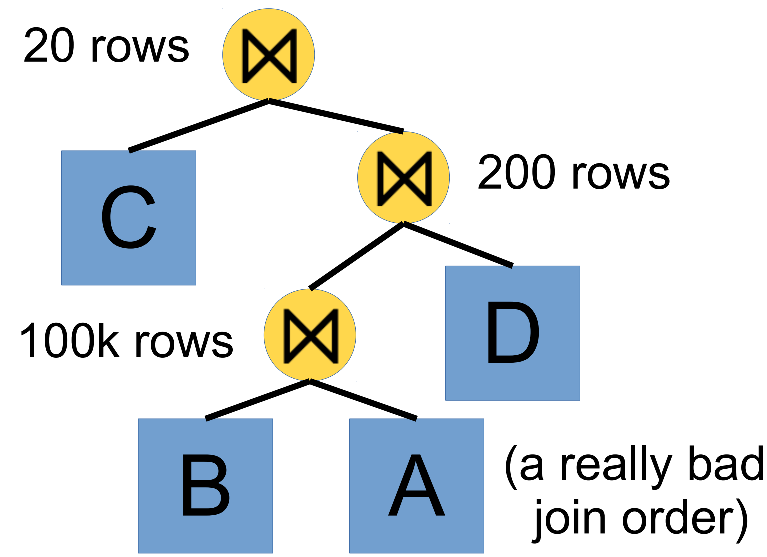
Bad plans are bad

What we want



Reading the score takes constant time

What we've got



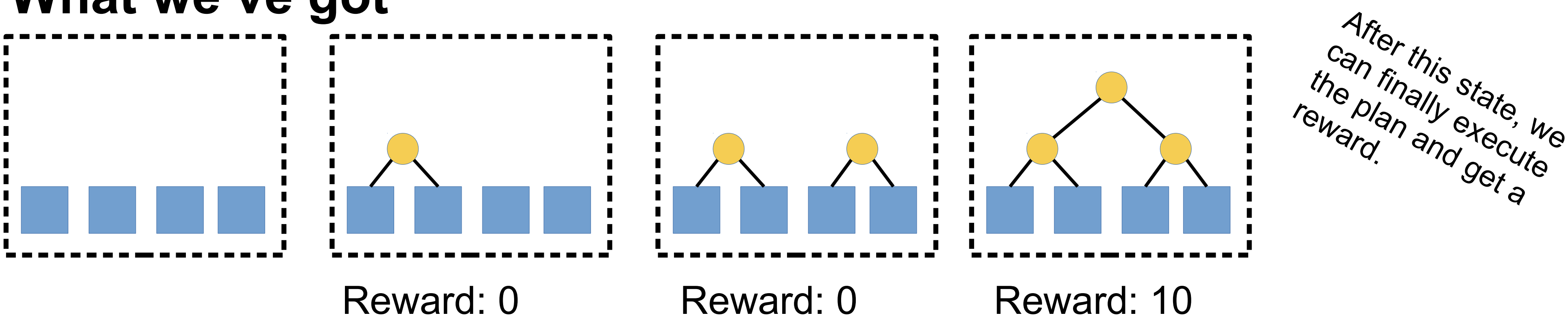
"Reading" the score takes a very long time!

- Good vs. bad join orders: seconds vs. days
- Sometimes even the best join order still takes minutes or hours
- … and we need 10k to converge!

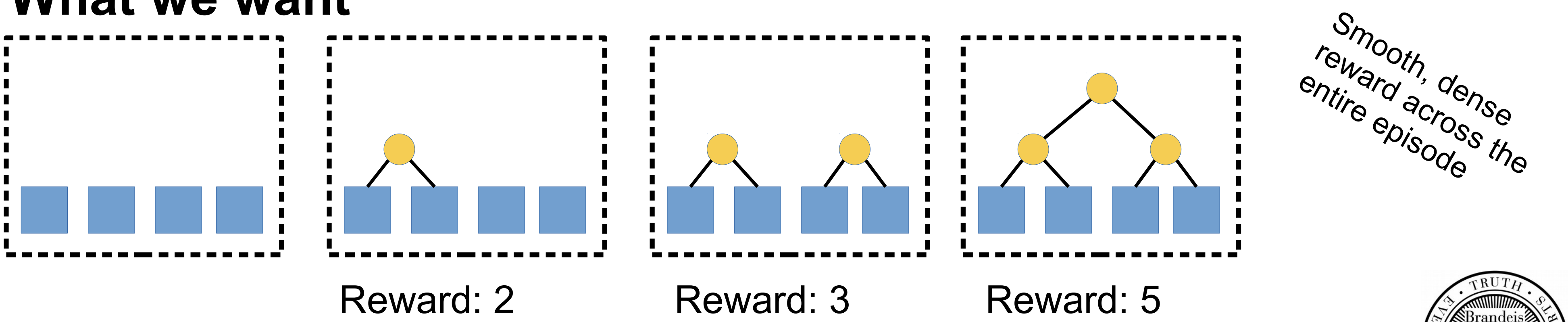
Sparse Rewards

- There are no intermediate rewards.

What we've got



What we want



Potential Solutions

- We describe three possible architectures:
  - **Learning from demonstration**
  - Cost-model bootstrapping
  - Incremental learning

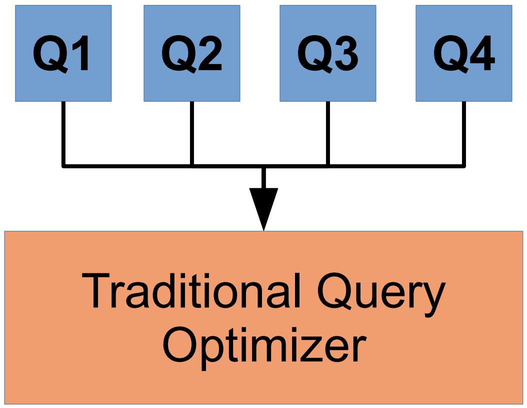
Learn from Demonstration

- "Cold start" learning occurs rarely in nature
  - Initial learning happens via *imitation*
- Can we learn from demonstration?
  - Traditional query optimizer = adult
  - DRL agent = child

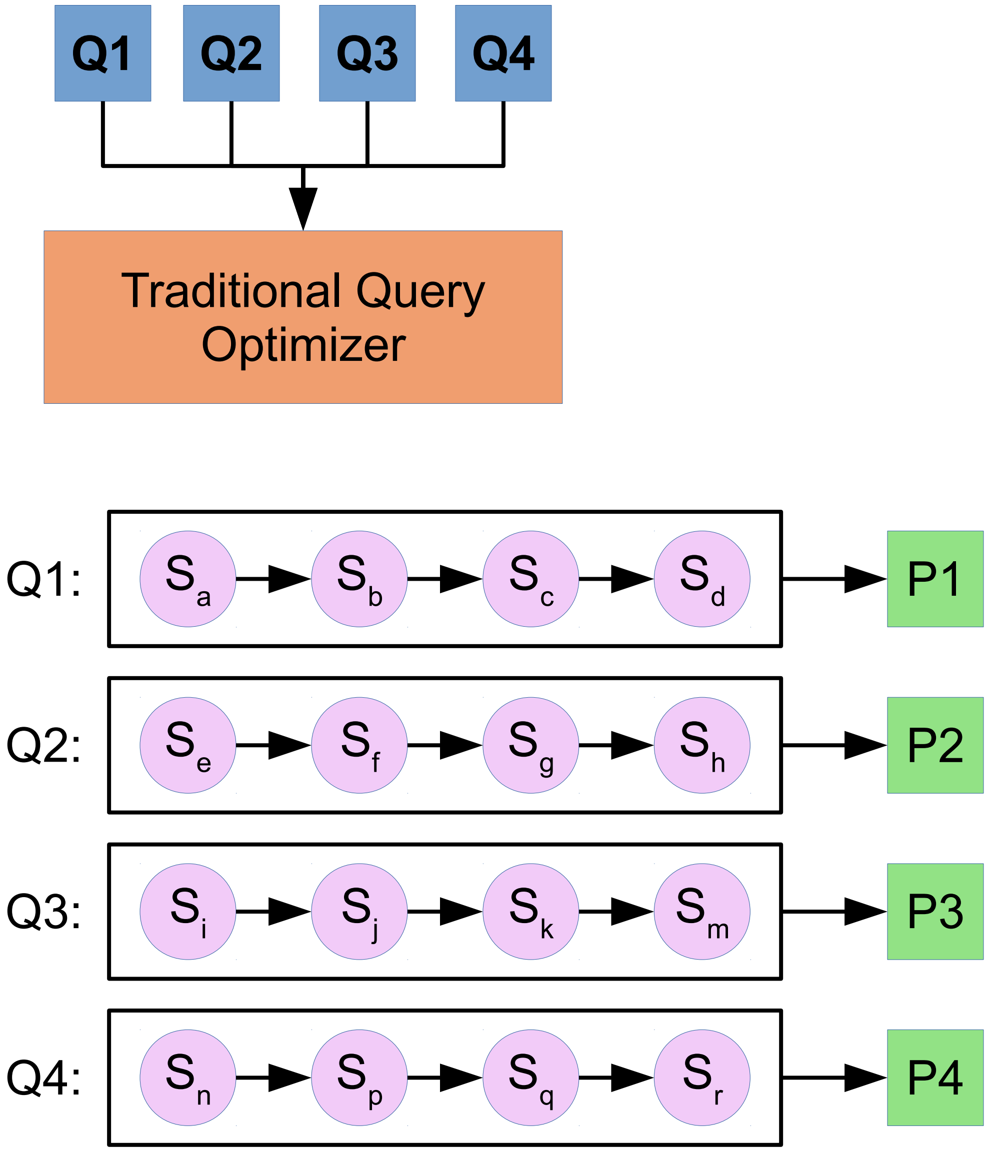
Learn from Demonstration

- Let $Q^*(s)$ be the best possible latency we could achieve from state (partial plan) *s*
  - A lot like an optimizer cost model
- Idea: use a neural network, Q(s), to estimate $Q^*(s)$
  - Initially, train this neural network through observation of the expert system
  - Then, refine it.

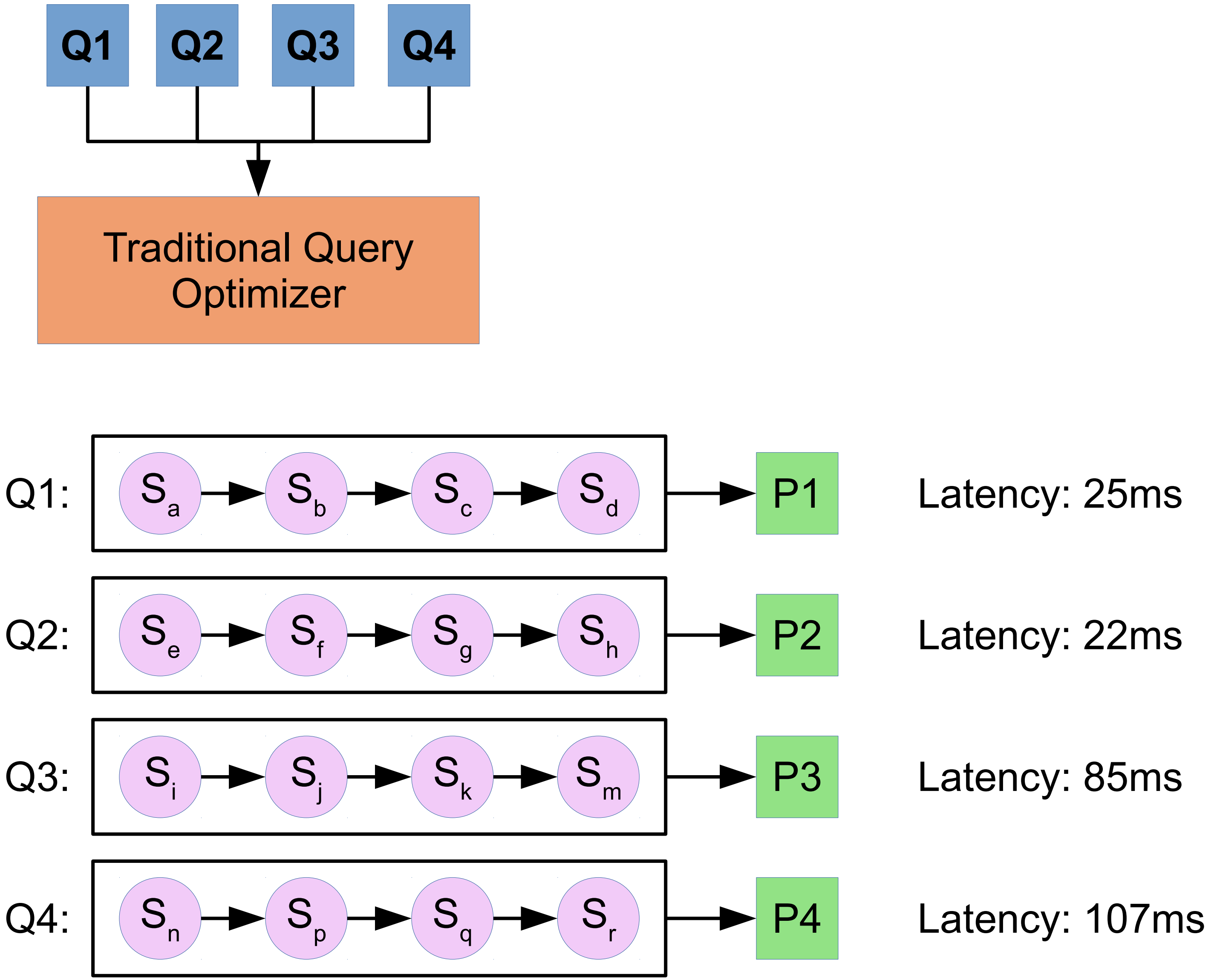
Learn from Demonstration



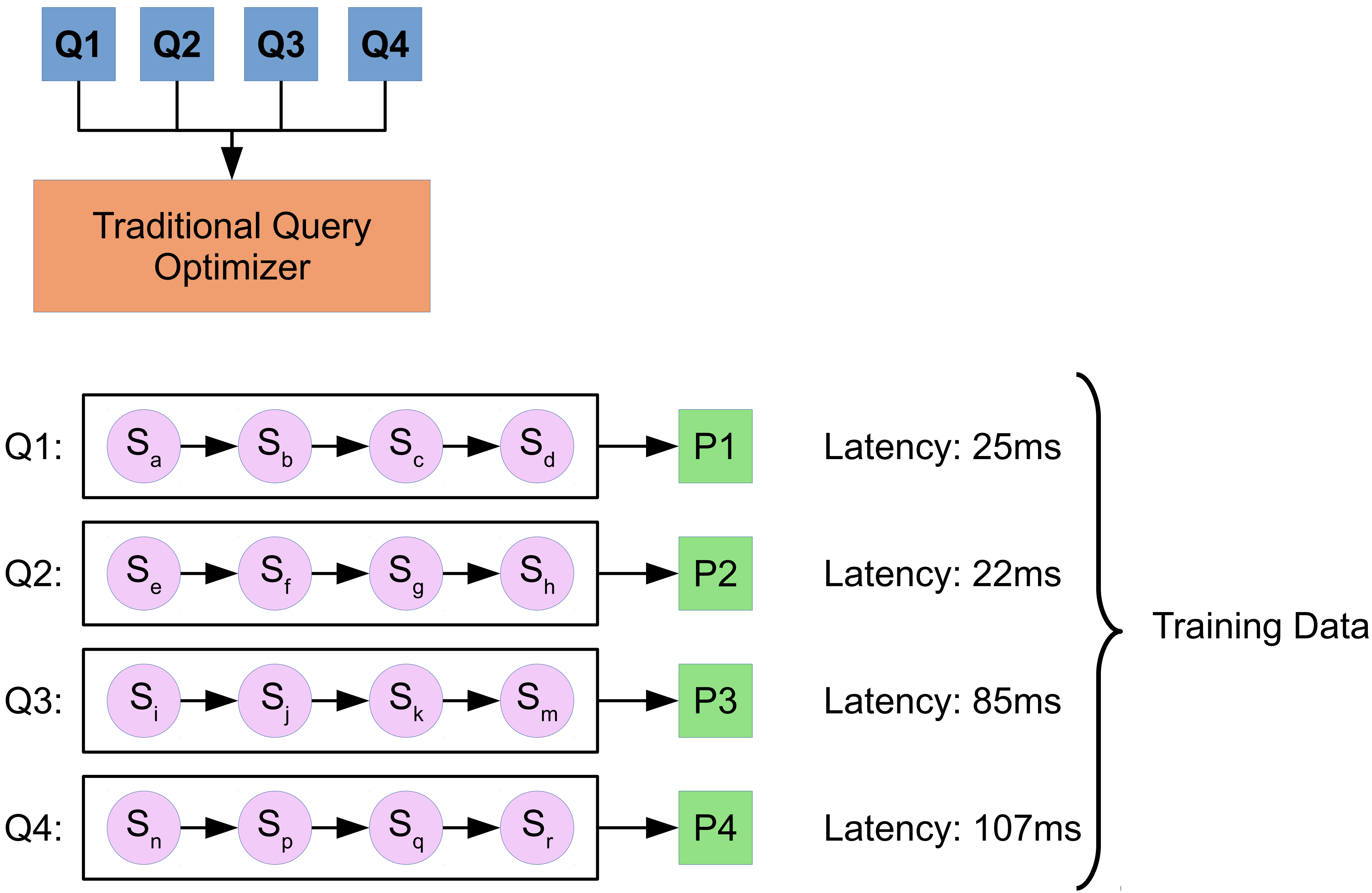
Learn from Demonstration



Learn from Demonstration

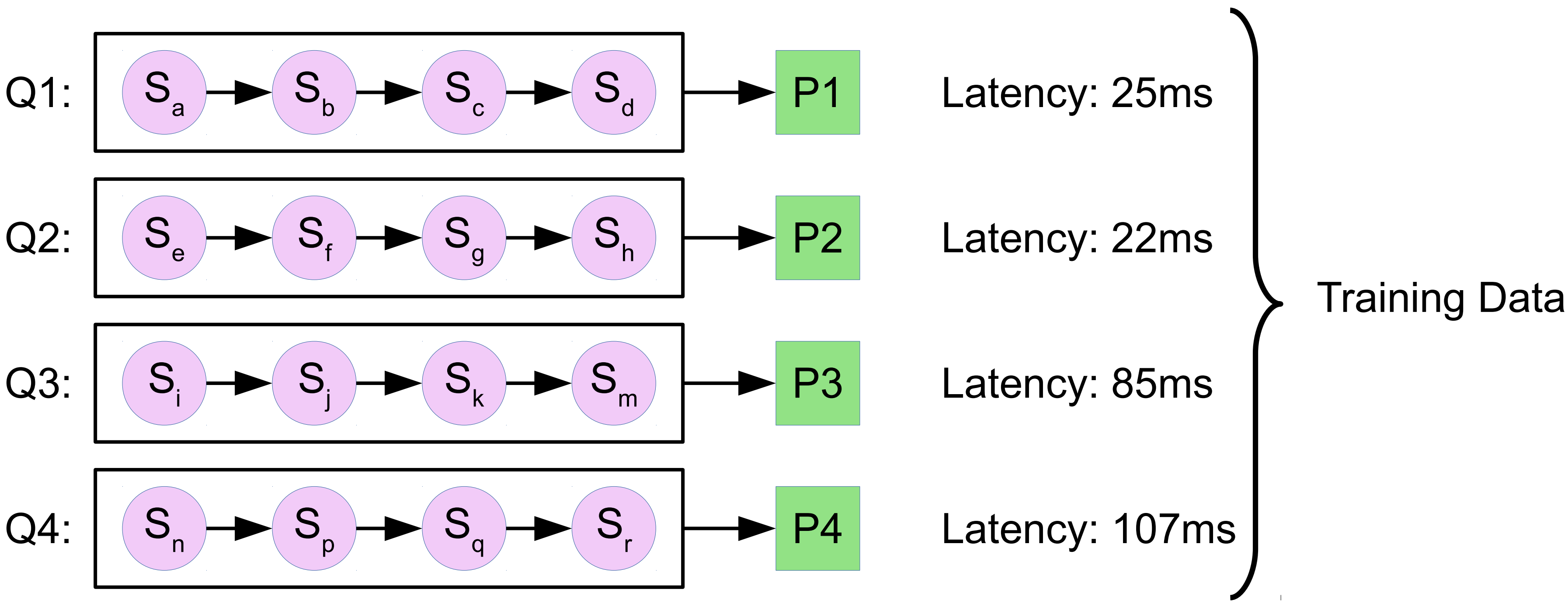


Learn from Demonstration

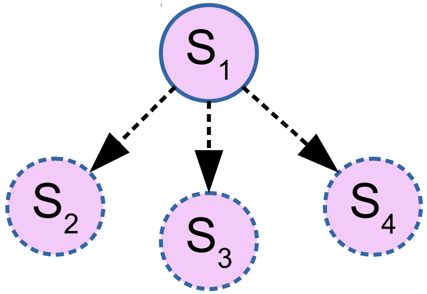


Learn from Demonstration

Q(S_a) = 25	Q(S_e) = 22	Q(S_i) = 85	Q(S_n) = 107
Q(S_b) = 25	Q(S_f) = 22	Q(S_j) = 85	Q(S_p) = 107
Q(S_c) = 25	Q(S_g) = 22	Q(S_k) = 85	Q(S_q) = 107
Q(S_d) = 25	Q(S_h) = 22	Q(S_m) = 85	Q(S_r) = 107



Learn from Demonstration

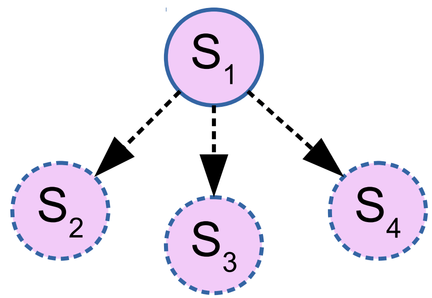


Learn from Demonstration

$Q(S_2) = 205$

$Q(S_3) = 87$

$Q(S_4) = 43$

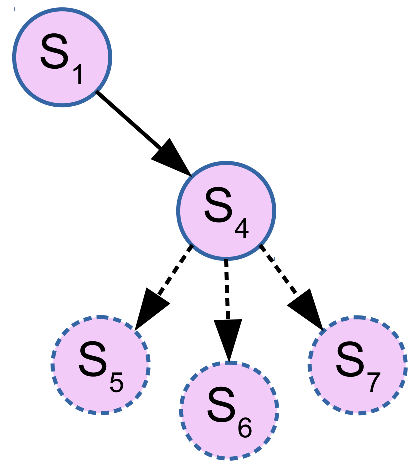


Learn from Demonstration

Q(S_5) = 36

Q(S_6) = 42

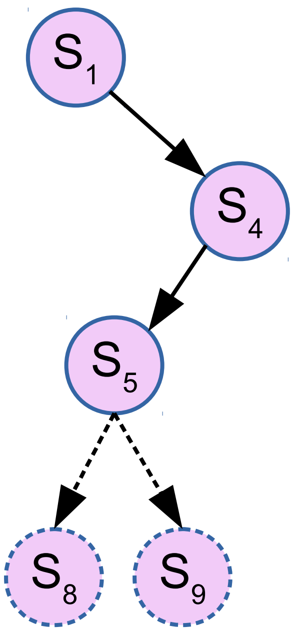
Q(S_7) = 88



Learn from Demonstration

Q(S_8) = 39

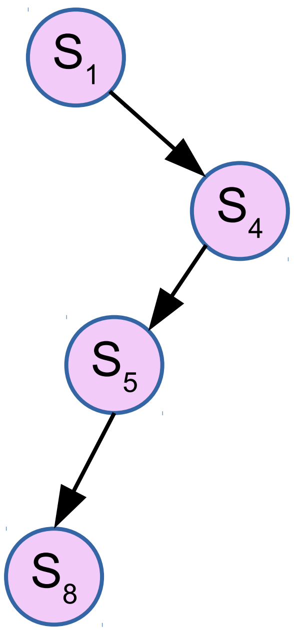
Q(S_9) = 60



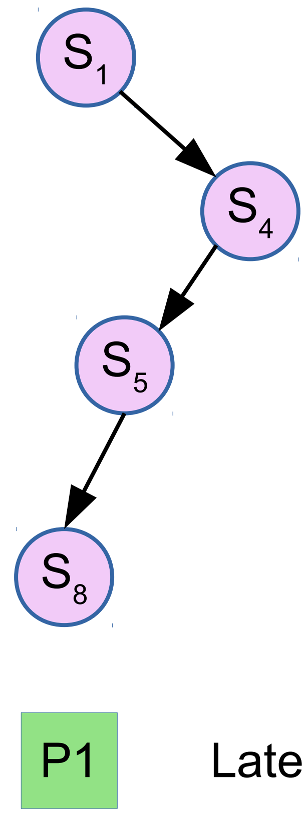
Learn from Demonstration

Q(S_8) = 39

Q(S_9) = 60



Learn from Demonstration



Latency: 40ms

Learn from Demonstration

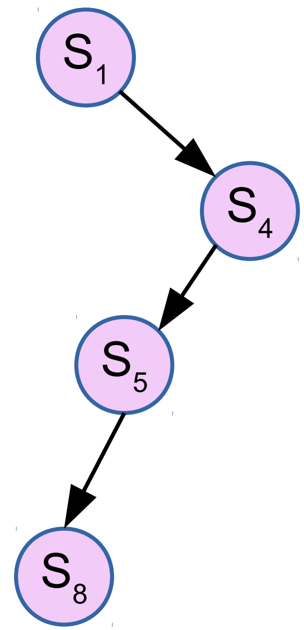
Predictions:

Q(S_1) = 25

Q(S_4) = 43

Q(S_5) = 36

Q(S_8) = 39



P1 Latency: 40ms

Learn from Demonstration

Predictions:

Q(S_1) = 25

Q(S_4) = 43

Q(S_5) = 36

Q(S_8) = 39

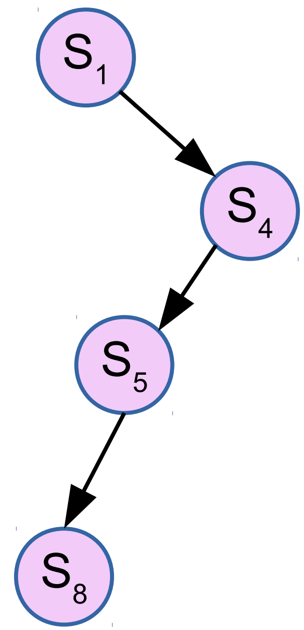
Update the network with:

Q(S_1) = 40

Q(S_4) = 40

Q(S_5) = 40

Q(S_8) = 40



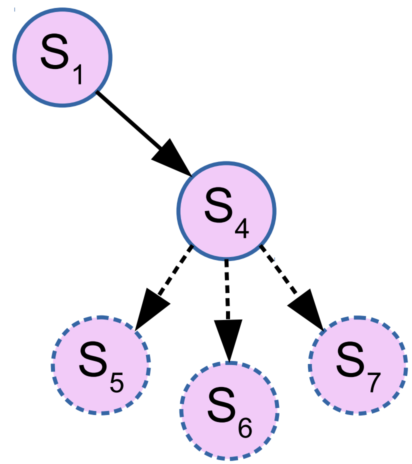
P1 Latency: 40ms

Learn from Demonstration

Q(S_5) = 36

Q(S_6) = 42

Q(S_7) = 88



Use the state with the lowest predicted latency

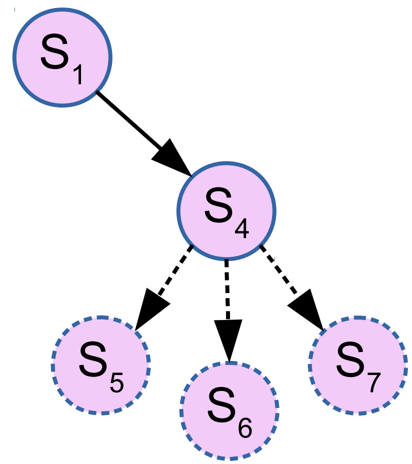
Result? Imitate & improve on the expert

Learn from Demonstration

$Q(S_5) = 36 \rightarrow 0.441$

$Q(S_6) = 42 \rightarrow 0.378$

$Q(S_7) = 88 \rightarrow 0.181$



Normalize the output, sample from the distribution

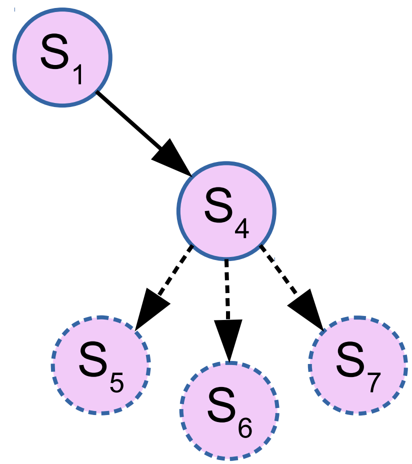
Result? Explore & exploit

Learn from Demonstration

Q(S_5) = 36 ± 20

Q(S_6) = 42 ± 28

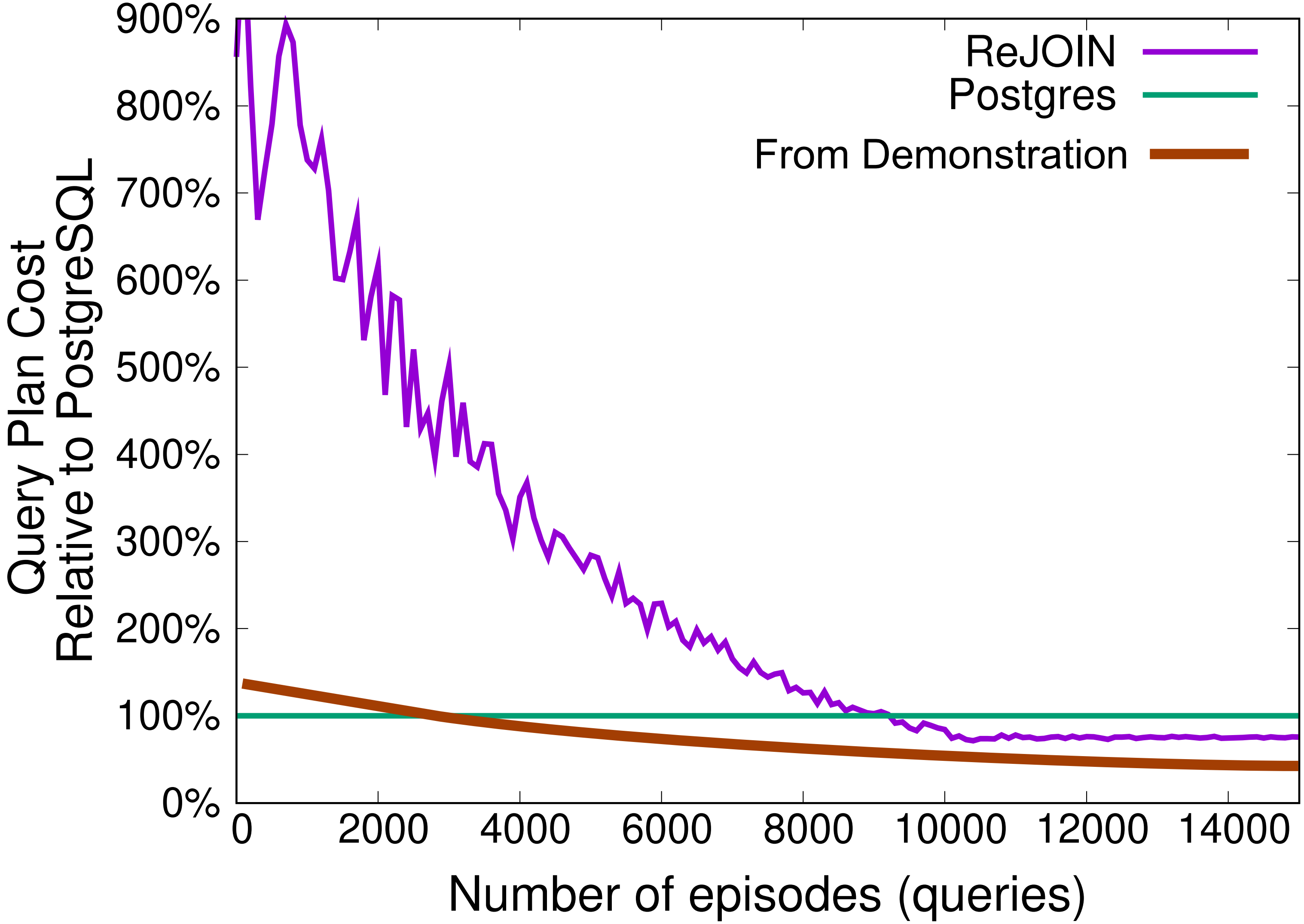
Q(S_7) = 88 ± 5



Use the variance of the predicated latency to decide when to “ask the expert” again

Result? “Active learning”

Learn from Demonstration



Desired behavior of a “learn from demonstration” system

Learn from Demonstration

- Take advantage of pre-existing optimizers
  - Bootstrap & surpass, hopefully!
- Drastically reduce convergence time, while:
  - Going beyond join ordering
  - Using query latency, not cost model

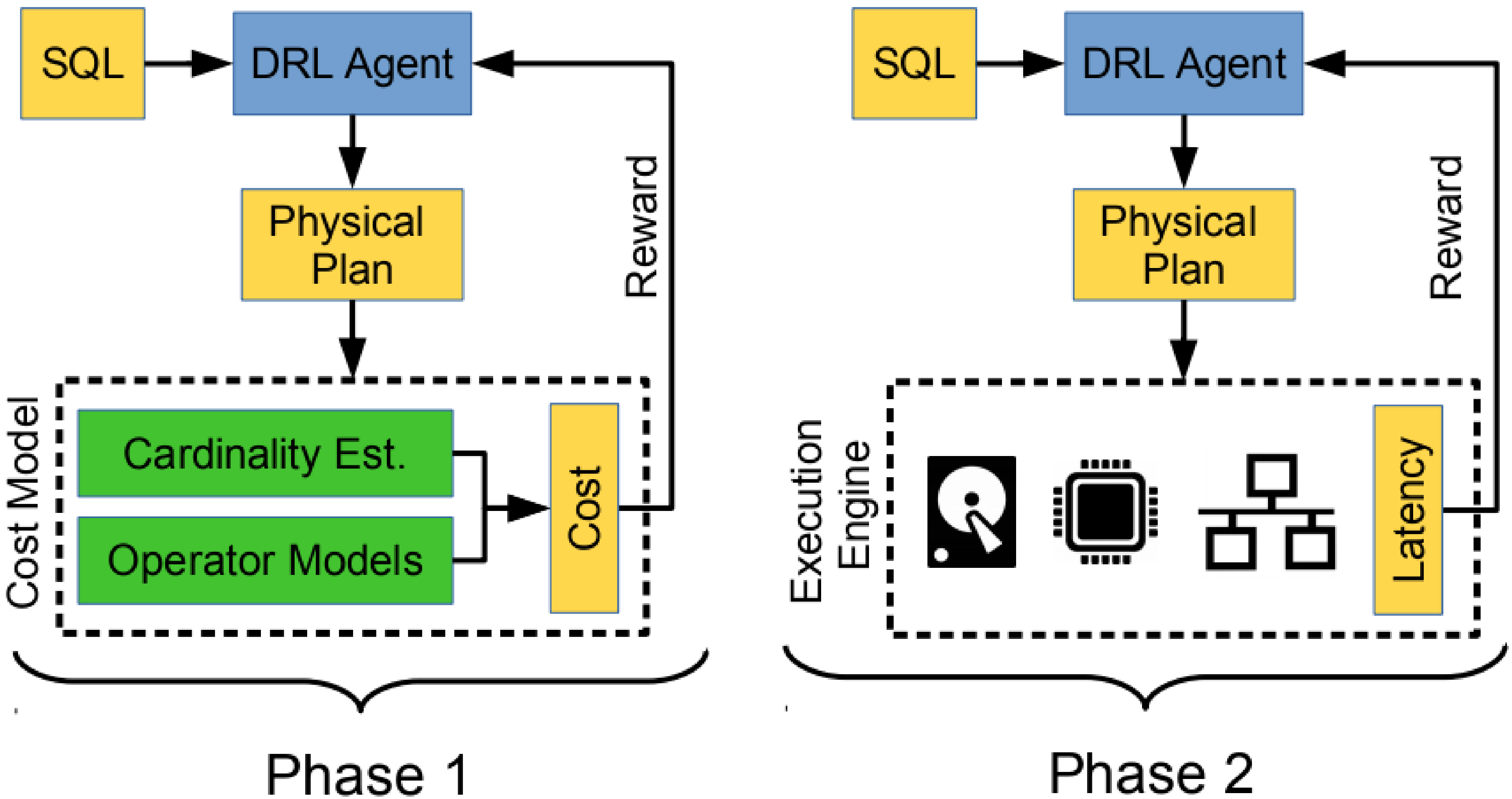
Learn from Demonstration

- Challenges & Opportunities
  - Trading off exploitation and exploration
  - Balancing expert / exploratory data
    - When do we “go back to the expert?”
  - Managing uncertainty
    - What to do when variance is high?
  - How good does the expert need to be?
    - Could we use something simple?

Potential Solutions

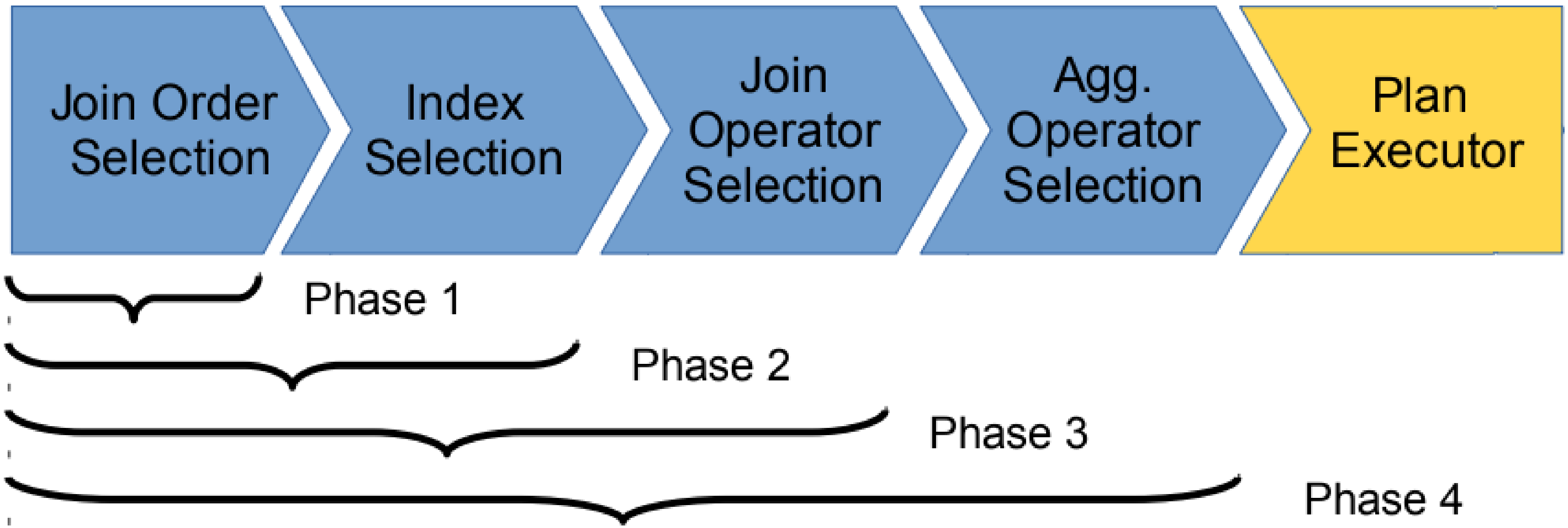
- We describe three possible architectures:
  - Learning from demonstration
  - **Cost-model bootstrapping**
  - **Incremental learning**

Cost-model Bootstrapping



Like practicing free throws before playing basketball

Incremental Learning



Instead of learning calculus from nothing, start with arithmetic, then geometry, then algebra, etc.

Conclusions

- Vast research space for DRL applications to query optimization
- Huge potential
  - For increasing query performance
  - For decreasing complexity

- These slides: `http://rm.cab/cidr19`
- Twitter: @RyanMarcus