Bao: Learning to Steer Query Optimizers

Paper authors: R Marcus[1,2], P Negi[1], H Mao[1], N Tatbul[1,2], M Alizadeh[1], T Kraska[1] ([1]MIT CSAIL, [2]Intel Labs)
Presenter: Mihai-Ionut Enache

8 November 2021

Query Optimization - Background and Problem

- **Domain**: relational database management systems (RDMS) + others (NoSQL, Graph DBs etc.)
- **Definition**: **query plan** = list of steps followed by the database engine to execute a given query
- **Problem**: what is the optimal query plan?
    - What is “optimal”? Time / cost etc.?
    - What is the context? Static / dynamic schema?

Example

ID	YoE
1	7
2	3
3	12
4	9

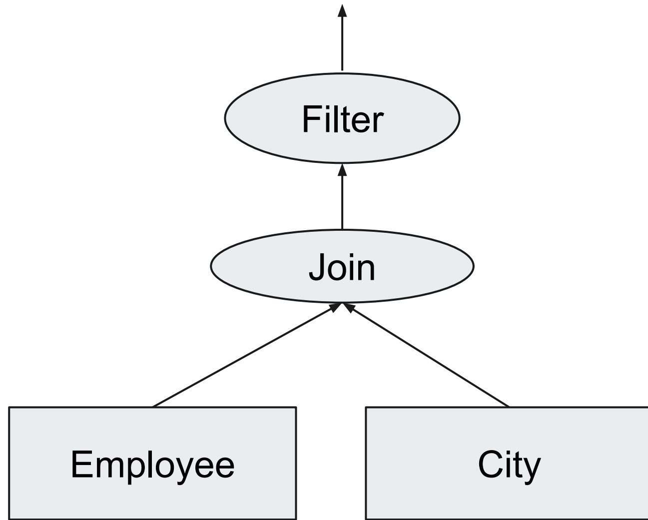
Table: Employee

Name	Employee_id
London	1
Bristol	2
London	3
Edinburgh	4

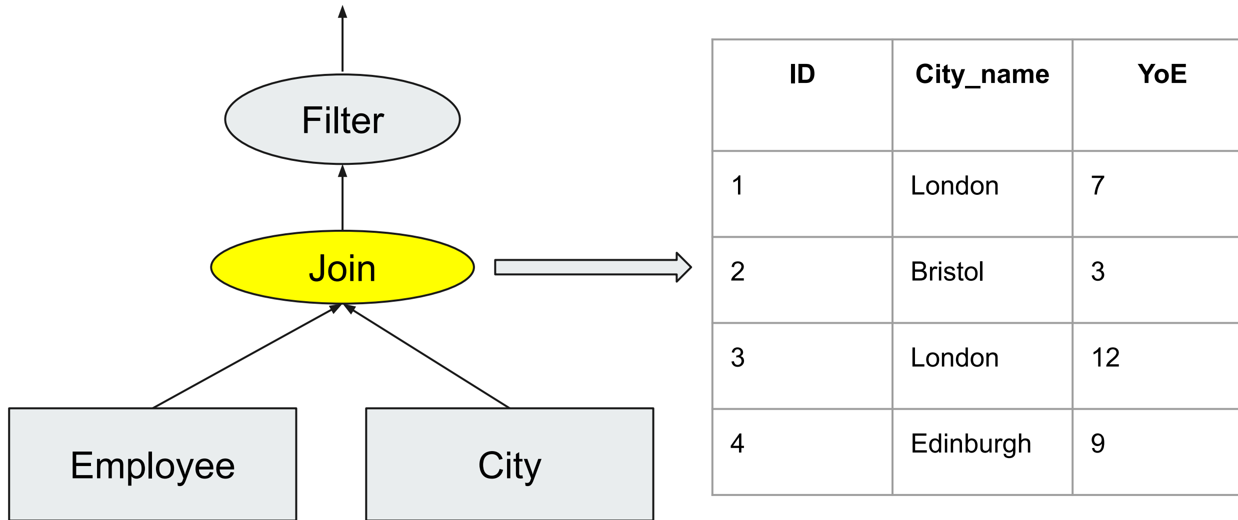
Table: City

```
SELECT name
FROM city
INNER JOIN employee on employee.id = city.employee_id
WHERE employee.yoe >= 10
```

Example Continued: Join Before Filter

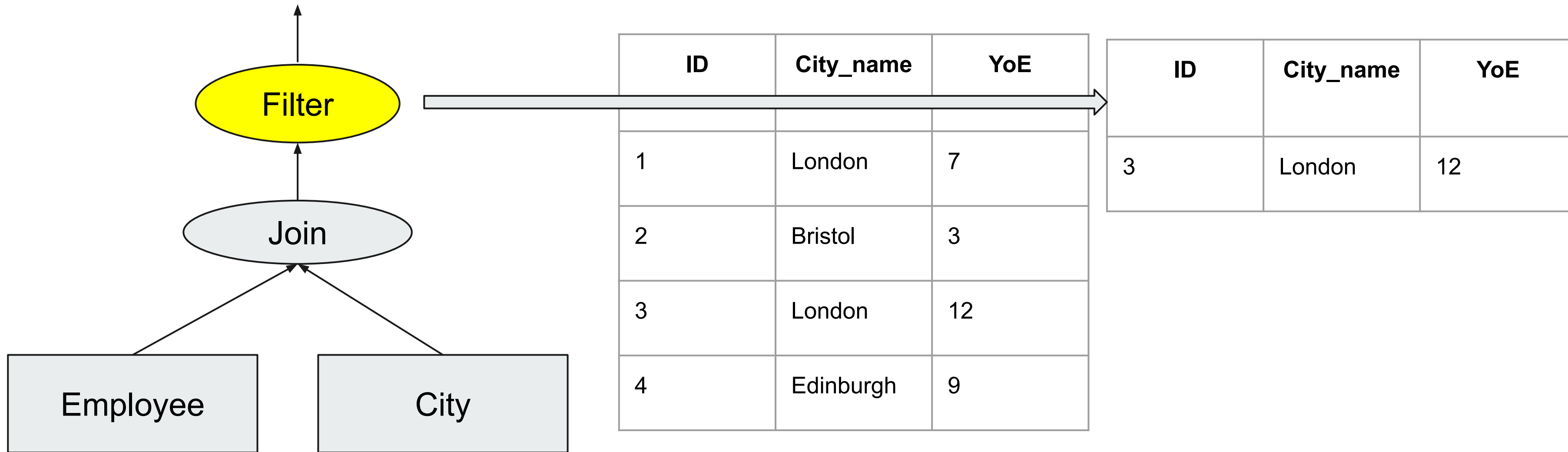


Example Continued: Join Before Filter



ID	City_name	YoE
1	London	7
2	Bristol	3
3	London	12
4	Edinburgh	9

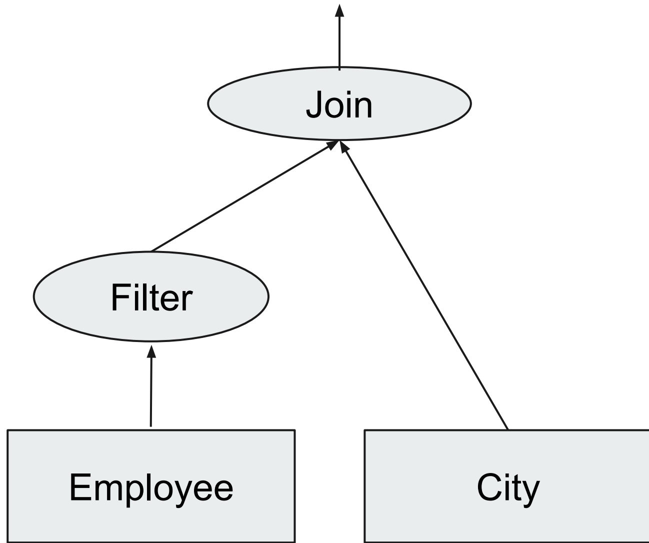
Example Continued: Join Before Filter



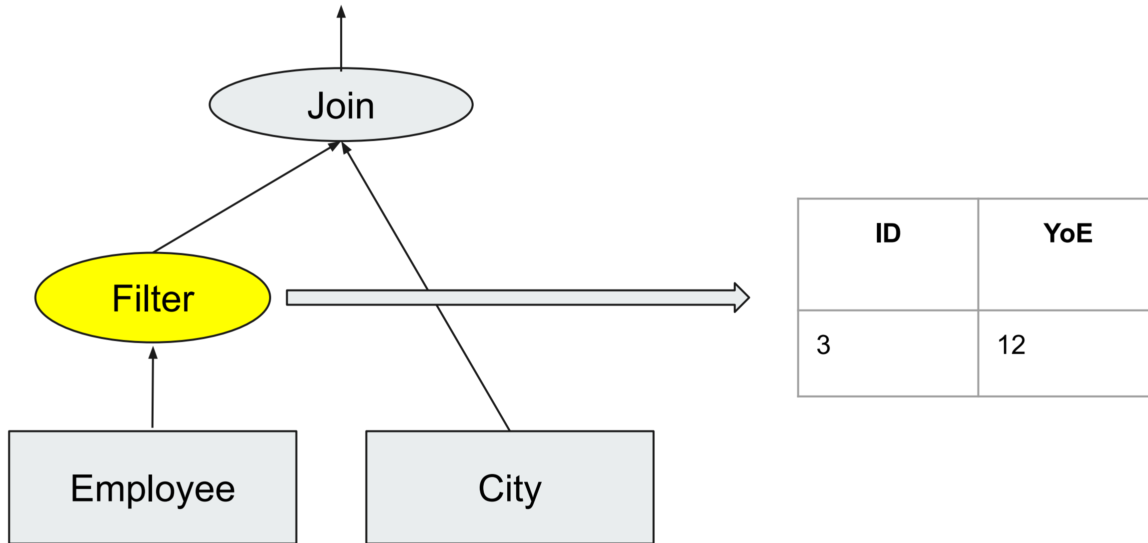
ID	City_name	YoE
1	London	7
2	Bristol	3
3	London	12
4	Edinburgh	9

ID	City_name	YoE
3	London	12

Example Continued: Filter Before Join



Example Continued: Filter Before Join



ID	YoE
3	12

Example Continued: Filter Before Join

Join

Filter

Employee

City

ID	YoE
3	12

ID	City_name	YoE
3	London	12

Motivation

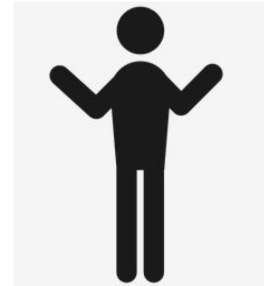
- Query execution has large impact on the application (e.g. user experience, memory footprint etc.)
- Traditional method: hand-tuned optimizers
- Recent trends: machine learning techniques, but there are limitations:
  - **Sample efficiency**: amount of data required is impractical
  - **Brittleness**: query / data / schema updates means we need to retrain
    - Many proposed schemes assume constant data / schema or need full re-training if they change
  - **Tail catastrophe**: learning techniques do better than hand-tuned on average, but how about worst case?
    - Tail latency = high-percentile latency = latencies that clients experience rarely
    - But worst case scenario is often critical to an application

New Idea!

- Information from traditional query optimizers is useful - don’t throw it away!
  - Has plenty of encoded ideas and algorithms, carefully planned and designed over the decades
- Instead, **steer** the query optimizer in the right direction using **hints**

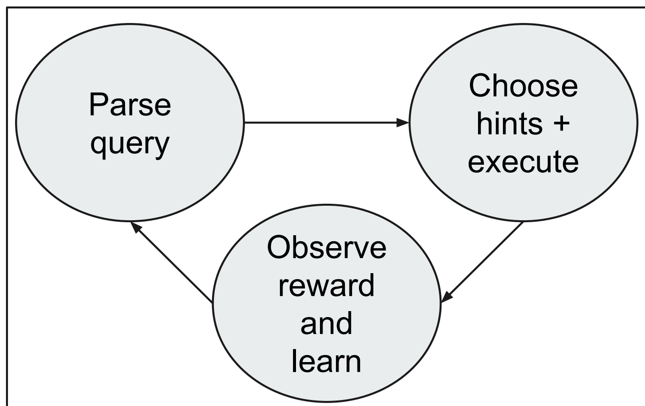
Multi-Armed Bandit Problem

- Fixed, finite set of resources
- Different choices
- Allocate resources for choices to maximize expected gain
- **Contextual multi-armed bandit problem**
  - Player has context information at the current iteration
  - Combines with rewards in the past
  - Tries to understand relation between context and rewards at each round to improve predictions
  - Reward is assumed to be independent given the context information



Introducing Bao

- Built on top of an optimizer
- Attempts to learn mappings from queries to subsets of hints
    - Given a query: **predict useful hints** to limit the search space of the optimizer
    - **Hint** = flag passed to the optimizer (e.g. “eliminate plans containing loop joins”)
- Treat each subset of hints as an arm in CMAB
- In short, what Bao does:



Paper’s Main Contribution

- **Bao**: system for query optimization that learns how to use hints to improve performance
- Simple predictive model and featurization scheme independent of workload / data / schema
- Shows the query system performs better than open source and commercial DBs
  - Cost
  - Latency
  - Adaptability

Bao Detailed

- **Goal**: produce set of hints that gives best performance
  - Use the underlying query optimizer to produce set of query plans
  - Transform each of them into a vector tree (nodes = feature vectors)
  - Feed trees into a Tree Convolutional Neural Network (TCNN) to predict the outcome of executing each plan
    - E.g.: predicts the time it would take to execute a plan
- **Challenge: exploration vs exploitation**
  - **Exploration** = try new things; **Exploitation** = do what you already know
  - Model as contextual multi-armed bandit problem: hint set = arm; query plans = contextual information
  - Solution: **Thompson sampling**

Thompson Sampling

- Bao uses predictive model M_θ to select a hint set (θ = weights)
  - Select query plan ⇒ execute ⇒ observe ⇒ add to Bao's experience E (update M_θ)
- M_θ is trained differently than standard ML
  - Most ML algorithms: use set of parameters that explain data, i.e. maximize P(θ | E)
    - Most likely parameters = expectation of the distribution: Exp[P(θ | E)]
  - But to balance exploitation and exploration we need to sample from θ
    - To maximize exploration: choose θ randomly
    - To maximize exploitation: choose θ = Exp[P(θ | E)]
    - Sampling offers a balance between the two

Bao's Architecture

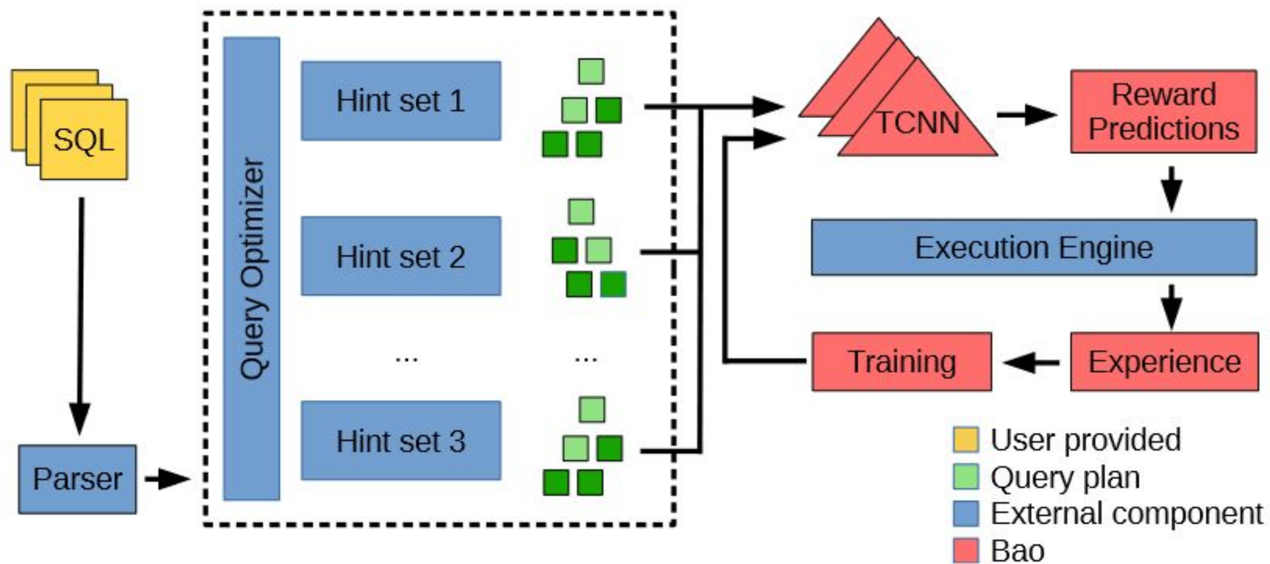


Figure 2: Bao system model

Bao’s Predictive Model

- Uses a Tree Convolutional Neural Network (TCNN)
- Need to map query plans to trees (input to TCNN)
- **Binarization:** transform query plan tree into a binary
- **Vectorization**: encode each query plan operator as vector
  - Vector = 3 parts:
    - Operator type
    - **Cardinality** and cost model information
      - Cardinality = how many unique values are in a column
    - Current state of disk (cache info)

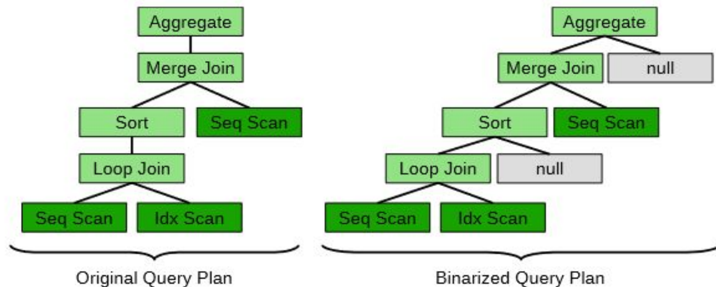


Figure 3: Binarizing a query plan tree

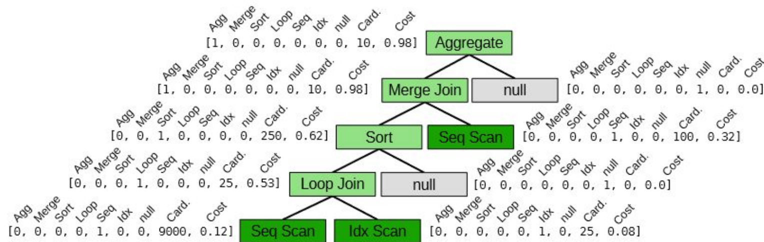


Figure 4: Vectorized query plan tree (vector tree)

Tree Convolutional Neural Networks

- Slide tree-shaped filters over the query plan tree (like image convolution)
  - Produce transformed query plans trees of the same size
  - Can stack TCNN operator ⇒ several layers of convolution
- Thompson sampling for training: use |E| random samples drawn with replacement from E

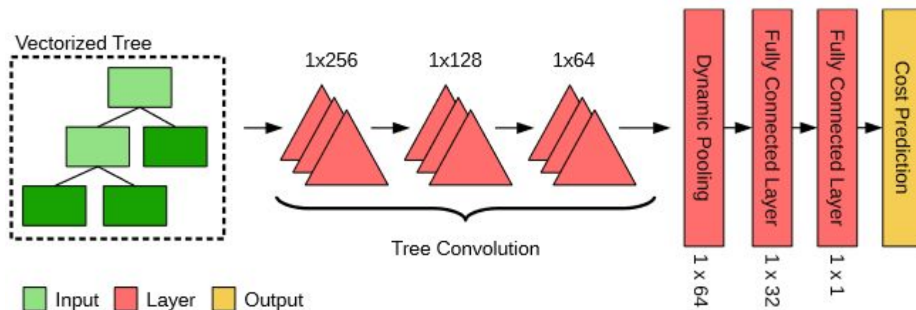


Figure 5: Bao prediction model architecture

Training Loop

- 2 problems:
  1. Sampling θ requires training the NN ⇒ expensive
  2. Experience E can grow unbounded
- Solutions:
  - Don’t retrain after every query; instead, do after every n-th query ⇒ training overhead reduced by factor of n
  - Only store the most recent k queries
  - User can tune n and k to trade-off quality and training overhead
- New optimization introduced:
  - Training NNS requires GPU; query processing requires CPU, RAM and disk ⇒ can overlap them
  - Enable GPU only when need to sample, otherwise detach
  - Useful on modern cloud platforms that charge per second

Experiments

- Focus on performance, but also on dollar cost
- Performed on the Google Cloud platform
- 2 approaches:
  - Compare with other systems: PostgreSQL and a commercial DB
  - Compare against the optimal choice
- 3 datasets:

	Size	Queries	WL	Data	Schema
IMDb	7.2 GB	5000	Dynamic	Static	Static
Stack	100 GB	5000	Dynamic	Dynamic	Static
Corp	1 TB	2000	Dynamic	Static[a]	Dynamic

Experiments: Tail Latency

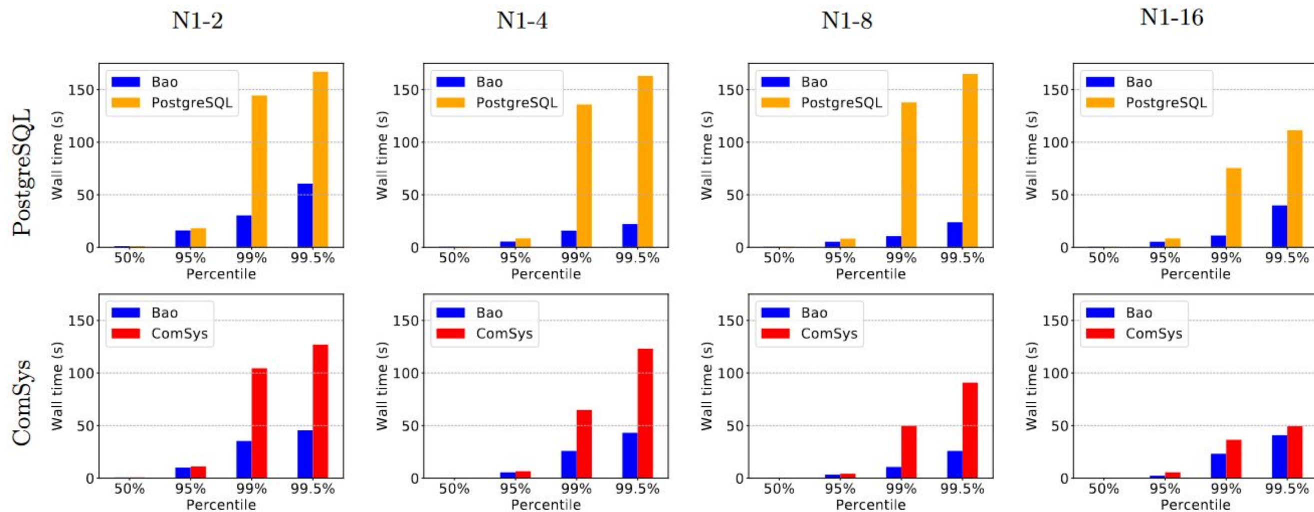
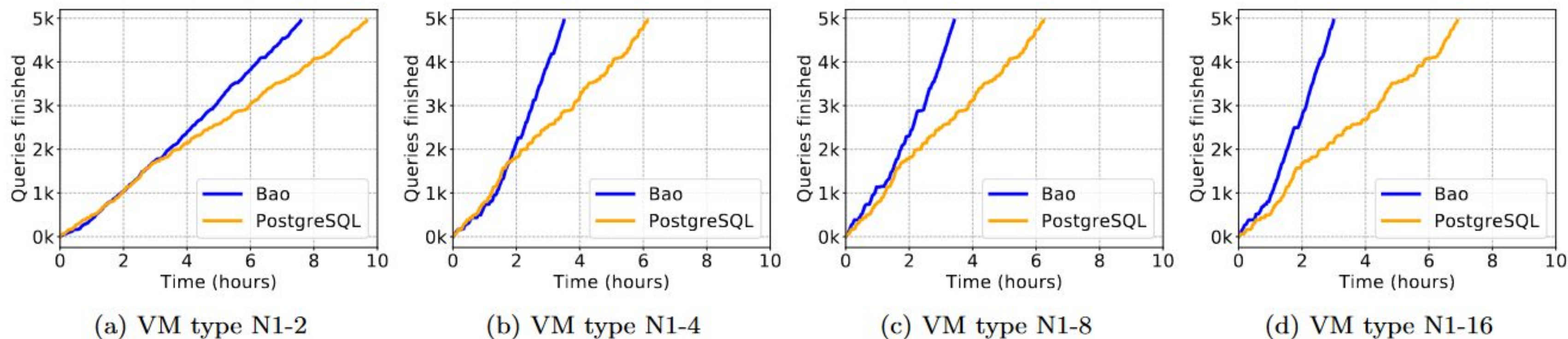


Figure 8: Percentile latency for queries, IMDb workload. Each column represents a VM type, from smallest to largest. The top row compares Bao against the PostgreSQL optimizer on the PostgreSQL engine. The bottom row compares Bao against a commercial database system on the commercial system's engine. Measured across the entire (dynamic) IMDb workload.

Experiments: Training Time and Convergence

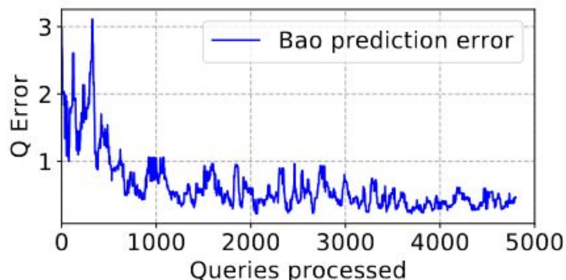


(a) VM type N1-2 (b) VM type N1-4 (c) VM type N1-8 (d) VM type N1-16

Figure 9: Number of IMDb queries processed over time for Bao and the PostgreSQL optimizer on the PostgreSQL engine. The IMDb workload contains 5000 unique queries which vary over time.

Experiments: Predictive Model Accuracy

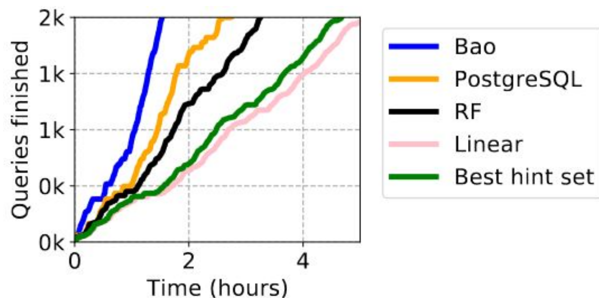
- As we make more decisions ⇒ experience grows ⇒ model is more accurate
- Defined in terms of Q-Error: $QError(x, y) = \max\left(\frac{x}{y}, \frac{y}{x}\right) - 1$
  - x = prediction, y = correct value



(a) Median Q-Error (0 is a perfect prediction) of Bao's predictive model vs. the number of queries processed. IMDb workload on N1-16 VM using PostgreSQL engine.

Experiments: Other ML Techniques

- NNs are costly
  - Sometimes, simpler approaches work better
  - Compare against Random Forests and Linear Regression
- Introduce “best hint set” = the set that performed best on average for all queries



(c) Random forest (RF) and linear models (Linear) used as Bao’s model. “Best hint set” is the single best hint set. IMDb, N1-16 VM, on PostgreSQL.

Other Experiments

- Other experiments in the paper include:
  - Required GPU time
  - Resiliency to hint sets that perform poorly
  - Optimization time
  - Regret over time and tails
  - Regression analysis

Related Work

- Cardinality estimation:
  - **Supervised learning**: recent work uses deep learning to learn cardinality estimators and query costs
  - **Unsupervised learning**: QuickSel (linear mixture models), Naru (Monte Carlo)
  - These provided better cardinality estimation, but **no evidence for better query performance**
- RL for query optimizers:
  - Neo: apply **deep reinforcement learning** to query latency
- Thompson sampling:
  - Been used for a while in statistics and decision making
  - Use custom set-up inspired from previous work: no need to define how to update **posterior belief**

Personal Opinion

- Pros:
  - Great idea to take advantage of decades of hand-tuned algorithms in traditional optimizers
  - Simple & efficient featurization scheme, agnostic of schema / data changes
  - TCNN: encapsulates information about the structure of the query plans in the features
  - Thompson sampling: great way to balance exploration vs exploitation
  - Convincing experiments: plenty of experiments, different approaches / contexts / baselines / datasets
  - (Mostly) Self-contained paper, good structure and flow
- Cons:
  - Focuses only on boolean hints (flags)
  - Transforms query plans in binary trees for convenience (they simplify TCNN) - can we do better?
  - Can only use hints to build query plans, i.e. can't build a new plan using custom strategies

Summary

- **Bao** - a bandit optimizer that steers query optimizers in the right direction using RL
- Problem as **CMAB** ⇒ well studied, efficient sampling algorithms
- Underlying optimizer ⇒ cost and cardinality estimation available ⇒ easier to adapt to data / schema changes
- Can immediately build on top of and improve the optimizer
  - Other systems have to learn those techniques
- Easy to integrate cache information
  - Reading from in-memory much better than disk
- Extensible architecture: easy to add / remove query hints
  - Little re-training time needed for new hints ⇒ can easily test new optimizers
  - Can do exception rules: can specifically exclude query hints if performing bad in practice