Special Topics:
Self-Driving Database Management Systems

Training Data Collection

@Dhruv_Arya // 15-799 // Spring 2022
QPPNet
→ Hierarchical Neural Networks for behaviour modeling
→ Query plan modeled at both the operator-level and the plan-level
→ Easy to transfer to new queries as long as the query uses operators that have already been learnt
Lots of data needed to train models

Current data sources -> Manually configured templates for synthetic data generation or Real-world databases

Even when we have a benchmark -> generating labels (e.g. execution time, latency) is expensive
→ Lots of data needed to train models
→ **Synthetically generate more data**

→ Current data sources -> Manually configured templates for synthetic data generation
→ **Automatically learn workload structures**

→ Even when we have a data source -> generating labels (e.g. execution time, latency) is expensive
→ **Forecast labels for data**
DataFarm – A High Level Overview
High Level Overview

I. Abstract Plan Generator:
Generates a generic plan.

II. Synthetic Job Instantiator:
Picks the tables and the parameters.

III. Behaviour Modeling:
Predict query execution time.
→ Model the plan as a Markov decision process
→ Build a transition probability graph from a given operator to child operators
Abstract Plan Generator

→ Model the plan as a Markov decision process
→ Another transition probability matrix for transitioning into a parent operator
→ Comes into play when the Join operator is encountered
→ Parent branches can’t have additional joins
Abstract Plan Generator

→ Does not specify the tables nor the selectors and the parameters.
→ The abstract plan generator does not know the structure of the data
Plan Instantiation

Use meta-data to instantiate the abstract plans.
Plan Instantiator --- Input Data

→ Table Schema
→ Fields used for joins
→ Cardinality
→ Parameter values distribution

```json
{
    "dataBaseID": "IMDB",
    "tables": [
        {
            "tableName": "title.rating",
            "rawCardinality": 99381,
            "typeSchema": "(String, float, int)",
            "fields": [
                "titleId",
                "averageRating",
                "numVotes"
            ],
            "filterFieldValue": {
                "releaseYear": {
                    "selectivity": "0.95"
                },
                "values": [
                    "1869.0"
                ]
            }
        }
    ]
}
```
Plan Instantiator --- Input Data

Two Interfaces for the users:
→ Database Manager: Add new relations
→ Table Manager: Specify the table-specific statistics

```json
{
  "dataBaseID": "IMDB",
  "tables": [
    {
      "tableName": "title.rating",
      "rawCardinality": 99381,
      "typeSchema": "(String, float, int)",
      "fields": [
        "titleId",
        "averageRating",
        "numVotes"
      ],
      "filterFieldValue": {
        "releaseYear": {
          "selectivity": "0.95"
        },
        "values": [
          "1869.0"
        ]
      }
    }
  ]
}
```
Iteratively instantiate nodes from the Data Source to the Data Sink

Instantiate other sub-branch on encountering Join

Use statistics from Table Manager to instantiate parameter values
Efficiently predicting some label for the given data.

Some label = execution time
Label Forecasting – Feature Extraction

→ Tables used -> represented as a one-hot encoding
→ Operator-level statistics
→ Complexity of operators computed
→ Map operator complexity = Number of rows scanned x Number of fields
→ Cardinality estimates based on user-supplied data
→ Principal Component Analysis for identifying features with most relevant variance
Train a Quantile RegressionForest Model
Gives the distribution of the output variable given the input variable
Prediction value: L
Prediction Interval: $[u_{low}(\hat{i}), u_{high}(\hat{i})]$

**Algorithm 1: Active labeling.**

1. $L \leftarrow \emptyset; L_{ex} \leftarrow \emptyset; \hat{J}_{noex} \leftarrow \emptyset; J_{ex} \leftarrow \emptyset; J_{noex} \leftarrow \emptyset; U \leftarrow \emptyset; i \leftarrow 0$
2. $F \leftarrow \text{FeatureExtractor}.\text{transform}(J)$
3. $S_{ex} \leftarrow \text{JobExecSampler}.\text{initialize}(F, \kappa)$
4. while not earlyStop($U, \lambda$) or $i < MAX_i$ do
   5. $L_{ex} \leftarrow L_{ex} \cup R.\text{submit}(S_{ex})$
   6. $J_{ex} \leftarrow J_{ex} \cup S_{ex}$
   7. $J_{noex} \leftarrow J \setminus J_{ex}$
   8. $M \leftarrow \text{ModelBuilder}(F[J_{ex}], L_{ex})$
   9. $\hat{J}_{noex} \leftarrow \text{Forecaster}(F[J_{noex}], M)$
   10. $u \leftarrow \text{UncertaintyEstimator}(\hat{J}_{noex}, M)$
   11. $U \leftarrow U \cup \{u\}$
   12. $S_{ex} \leftarrow \text{JobExecSampler}.\text{nextExecs}(J_{noex}, u, \eta)$
   13. $i \leftarrow i + 1$
5. end
6. $L \leftarrow L_{ex} \cup \hat{J}_{noex}$
7. return $L$
Active Learning for the training:
Start by executing initial few jobs to get ground truth labels
Iteratively select jobs such that the distribution of high-, medium-, and low-cardinality jobs is similar

Algorithm 1: Active labeling.

\[
\text{Input:} \quad \text{jobs instances } J; \text{ computational resources } R; \text{ number of init. jobs } K; \text{ threshold } \eta; \text{ early stopping threshold } \lambda; \text{ max iterations } MAX_i \\
\text{Output:} \quad \text{set of labels } L \\
1 \quad L \leftarrow \emptyset; L_{ex} \leftarrow \emptyset; L_{noex} \leftarrow \emptyset; J_{ex} \leftarrow \emptyset; J_{noex} \leftarrow \emptyset; U \leftarrow \emptyset; i \leftarrow 0; \\
2 \quad F \leftarrow \text{FeatureExtractor.transform}(J); \\
3 \quad S_{ex} \leftarrow \text{JobExecSampler.initialize}(F, K); \\
4 \quad \text{while not earlyStop}(U, \lambda) \text{ or } i < MAX_i \text{ do} \\
5 \quad \quad L_{ex} \leftarrow L_{ex} \cup R\text{.submit}(S_{ex}); \\
6 \quad \quad J_{ex} \leftarrow J_{ex} \cup S_{ex}; \\
7 \quad \quad J_{noex} \leftarrow J \setminus J_{ex}; \\
8 \quad \quad M \leftarrow \text{ModelBuilder}(F[J_{ex}], L_{ex}); \\
9 \quad \quad L_{noex} \leftarrow \text{Forecaster}(F[J_{noex}], M); \\
10 \quad \quad u \leftarrow \text{UncertaintyEstimator}(L_{noex}, M); \\
11 \quad \quad U \leftarrow U \cup \{u\}; \\
12 \quad \quad S_{ex} \leftarrow \text{JobExecSampler.nextExecs}(J_{noex}, u, \eta); \\
13 \quad \quad i \leftarrow i + 1; \\
14 \quad \text{end} \\
15 \quad L \leftarrow L_{ex} \cup L_{noex}; \\
16 \quad \text{return } L
\]
Experimental Setup

→ Apache Flink
→ 4 x Intel Xeon 2.40GHz CPU +16 GB RAM
→ IMDB Dataset – 5 Tables
→ TPC-H:
  a. 1GB, 5GB, 10GB, 50GB ---
  b. Q1, Q3, Q11, Q13, Q17, and Q21 only
  c. Implemented in Flink

```
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from lineitem
where l_shipdate <= date '1998-12-01' - interval ':1' day
  group by l_returnflag, l_linestatus
  order by l_returnflag, l_linestatus;
```

Generated Workloads

→ W1: 2000 jobs based on TPC-H <- 50 Abstract plans per dataset -> 10 instantiations each * four TPC-H datasets

→ W2: 1000 jobs based on IMDB <- 50 Abstract plans -> 20 instantiations each
Characteristics of the Generated Data

(a) W1

(b) W2
Operator Frequency Comparison
## Prediction Accuracy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>2,000</td>
<td>142</td>
<td>93</td>
<td>0.67</td>
<td>&lt;0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>532</td>
<td>73</td>
<td>0.75</td>
<td>0.65</td>
</tr>
<tr>
<td>W2</td>
<td>1,000</td>
<td>141</td>
<td>86</td>
<td>0.16</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>418</td>
<td>58</td>
<td>0.52</td>
<td>0.08</td>
</tr>
</tbody>
</table>

(a) W1 – 142 executed jobs.  
(b) W2 – 418 executed jobs
Effectiveness

→ Comparison of Random Forest Regressors
→ Trained on Predicted Labels vs Ground Truth
→ Ground Truth model only 1% more accurate — 74% vs 73% variance explained
DataFarm vs TDGen

→ Execution Time estimation model trained on jobs generated by DataFarm and TDGen
→ TDGen – doesn’t consider input data distribution
Active Labeling and Cardinality Estimates Effectiveness

→ Real Cardinality numbers extracted from Flink
→ Chosen method is significantly better only for W2

(b) W1 - $R^2$ validation scores.
(d) W2 - $R^2$ validation scores.
model uncertainty vs training data size

→ Early stop --- when there is a significant drop in uncertainty

\[ \tilde{u} = \frac{\sum_{\hat{l}} \hat{L}_{noex} u_{high}(\hat{l}) - u_{low}(\hat{l})}{|\hat{L}_{noex}|} \]

→ \( u_{high} \) – 75% percentile prediction
→ \( u_{low} \) – 25% percentile prediction
→ Only on the unexecuted jobs

(a) W1 - Model’s uncertainty.

(b) W2 - Model’s uncertainty.
→ Data sampled uniformly during the generation phase
→ $R^2 = -0.40$

Input Data Importance – W2
Outliers

→ Ground Truth of X% of data points in W1 changed during the training process
→ The uncertainty value of the QRF model also becomes high
PARTING THOUGHTS

→ This approach does not fully simulate a workload ---
timing/frequency aspects not covered

→ Need better evaluation metrics

→ Only OLAP data right now?
References
