The Relational Data Borg is Learning

fdbusresearch.github.io relational.ai

Dan Olteanu
University of Zurich

VLDB 2020 Keynote
Virtual Tokyo, Sept 1, 2020
Acknowledgments

**FDB team, in particular:**

Ahmet  Amir  Haozhe  Max  Milos

**RelationalAI team, in particular:**

Hung  Long  Mahmoud  Molham
Reasons for DB research community to feel empowered:

1. **Pervasiveness of relational data** in data science
   - Hard fact

2. **Widespread need for efficient data processing**
   - Core to our community’s *raison d’être*

3. **New processing challenges** posed by data science workloads
   - DB’s approach reminiscent of Star Trek’s Borg Collective

These reasons also serve as motivation for our work.
Co-opt technology and knowledge of alien species to become ever more powerful and versatile
Relational Data Borg

Assimilate ideas and applications of related fields to adapt to new requirements and become ever more powerful and versatile.

Unlike in Star Trek, the Relational Data Borg

• moves fast
• has great skin complexion and
• is reasonably happy
Borg Cube vs Data Cube
Resistance IS futile in either case
Relational Data is Ubiquitous

Kaggle Survey: Most Data Scientists use Relational Data at Work!

(based on 2017 Kaggle survey of 16,000 ML practitioners)
State of Affairs in Learning over Relational Data

Inventory ⋊ ⋉ Stores ⋊ ⋉ Items ⋊ ⋉ Weather ⋊ ⋉ Demographics

Feature Extraction Query
Inventory ⋊ Stores ⋊ Items ⋊ Weather ⋊ Demographics

10,000s of Features

Relational Data

Training Dataset
ML Tool
Model

Structure-Agnostic Learning:
1. Unnecessary data matrix materialization
   - Relational structure carefully crafted by domain experts thrown away
2. Expensive data move
   - Training dataset can be order-of-magnitude larger than the input DB
3. Bloated one-hot encoding
4. High maintenance cost
   - Recomputation from scratch after updates
5. Limitations inherited from both DB and ML tools
State of Affairs in Learning over Relational Data

Feature Extraction Query

10,000s of Features

Structure-Agnostic Learning:

1. Unnecessary data matrix materialization
   Relational structure carefully crafted by domain experts thrown away

2. Expensive data move
   Training dataset can be order-of-magnitude larger than the input DB

3. Bloated one-hot encoding

4. High maintenance cost
   Recomputation from scratch after updates

5. Limitations inherited from both DB and ML tools
State of Affairs in Learning over Relational Data

Structure-Agnostic Learning:

1. **Unnecessary** data matrix materialization
   
   Relational structure carefully crafted by domain experts thrown away

Feature Extraction Query

\[
\text{Inventory} \times \text{Stores} \times \text{Items} \\
\times \text{Weather} \times \text{Demographics}
\]
State of Affairs in Learning over Relational Data

Structure-Agnostic Learning:

1. **Unnecessary** data matrix materialization
   
   Relational structure carefully crafted by domain experts thrown away

2. **Expensive** data move

   Training dataset can be order-of-magnitude larger than the input DB
Structure-Agnostic Learning:

1. **Unnecessary** data matrix materialization
   - Relational structure carefully crafted by domain experts thrown away
2. **Expensive** data move
   - Training dataset can be order-of-magnitude larger than the input DB
3. **Bloated** one-hot encoding
State of Affairs in Learning over Relational Data

Feature Extraction Query

10,000s of Features

Relational Data

Structure-Agnostic Learning:

1. **Unnecessary** data matrix materialization
   - Relational structure carefully crafted by domain experts thrown away
2. **Expensive** data move
   - Training dataset can be order-of-magnitude larger than the input DB
3. **Bloated** one-hot encoding
4. **High** maintenance cost
   - Recomputation from scratch after updates
State of Affairs in Learning over Relational Data

Structure-Agnostic Learning:

1. **Unnecessary** data matrix materialization
   
   Relational structure carefully crafted by domain experts thrown away

2. **Expensive** data move

   Training dataset can be order-of-magnitude larger than the input DB

3. **Bloated** one-hot encoding

4. **High** maintenance cost

   Recomputation from scratch after updates

5. **Limitations** inherited from both DB and ML tools
Structure-Aware Learning over Relational Data

Relational Data

Feature Extraction Query
+ Feature Aggregates

Batch of Aggregate Queries

Optimisation

10,000s of Features

Training Dataset

ML Tool

Model
Conjecture

The learning time and accuracy of the model can be drastically improved by exploiting the structure and semantics of the underlying multi-relational database.
Structure-aware Learning FASTER than Feature Extraction Query alone

<table>
<thead>
<tr>
<th>Relation</th>
<th>Cardinality</th>
<th>Arity (Keys+Values)</th>
<th>File Size (CSV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inventory</td>
<td>84,055,817</td>
<td>3 + 1</td>
<td>2 GB</td>
</tr>
<tr>
<td>Items</td>
<td>5,618</td>
<td>1 + 4</td>
<td>129 KB</td>
</tr>
<tr>
<td>Stores</td>
<td>1,317</td>
<td>1 + 14</td>
<td>139 KB</td>
</tr>
<tr>
<td>Demographics</td>
<td>1,302</td>
<td>1 + 15</td>
<td>161 KB</td>
</tr>
<tr>
<td>Weather</td>
<td>1,159,457</td>
<td>2 + 6</td>
<td>33 MB</td>
</tr>
<tr>
<td>Join</td>
<td>84,055,817</td>
<td>3 + 41</td>
<td>23GB</td>
</tr>
</tbody>
</table>
Train a linear regression model to predict *inventory* given all features

<table>
<thead>
<tr>
<th>Time</th>
<th>Size (CSV)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PostgreSQL+TensorFlow</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>–</td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td>152.06 secs</td>
</tr>
<tr>
<td><strong>Export</strong></td>
<td>351.76 secs</td>
</tr>
<tr>
<td><strong>Shuffling</strong></td>
<td>5,488.73 secs</td>
</tr>
<tr>
<td><strong>Query batch</strong></td>
<td>–</td>
</tr>
<tr>
<td><strong>Grad Descent</strong></td>
<td>7,249.58 secs</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td>13,242.13 secs</td>
</tr>
</tbody>
</table>

Our approach (SIGMOD’19) is 2,160 times faster while being more accurate (RMSE on 2% test data) compared to TensorFlow. TensorFlow trains one model. Our approach takes < 0.1 sec for any extra model over a subset of the given feature set.
Train a linear regression model to predict *inventory* given all features

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL+TensorFlow</th>
<th>Our approach (SIGMOD’19)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Size (CSV)</td>
</tr>
<tr>
<td>Database</td>
<td>–</td>
<td>2.1 GB</td>
</tr>
<tr>
<td>Join</td>
<td>152.06 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Export</td>
<td>351.76 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Shuffling</td>
<td>5,488.73 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Query batch</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Grad Descent</td>
<td>7,249.58 secs</td>
<td>–</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td><strong>13,242.13 secs</strong></td>
<td><strong>6.13 secs</strong></td>
</tr>
</tbody>
</table>

2, 160× faster while being more accurate (RMSE on 2% test data)
Structure-aware versus Structure-agnostic Learning

Train a linear regression model to predict *inventory* given all features

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL+TensorFlow</th>
<th>Our approach (SIGMOD’19)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Size (CSV)</td>
</tr>
<tr>
<td>Database</td>
<td>–</td>
<td>2.1 GB</td>
</tr>
<tr>
<td>Join</td>
<td>152.06 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Export</td>
<td>351.76 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Shuffling</td>
<td>5,488.73 secs</td>
<td>23 GB</td>
</tr>
<tr>
<td>Query batch</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Grad Descent</td>
<td>7,249.58 secs</td>
<td>–</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td><strong>13,242.13 secs</strong></td>
<td><strong>6.13 secs</strong></td>
</tr>
</tbody>
</table>

2, 160× faster while being more accurate (RMSE on 2% test data)

*TensorFlow* trains one model. *Our approach* takes < 0.1 sec for any extra model over a subset of the given feature set.
Similar behaviour (or outright failure) for more:

- **datasets**: Favorita, TPC-DS, Yelp, Housing
- **systems**:  
  - used in industry: R, scikit-learn, Python StatsModels, mlpack, XGBoost, MADlib  
  - academic prototypes: Morpheus, libFM
- **models**: decision trees, factorisation machines, $k$-means, ..

This is to be contrasted with the scalability of DBMSs!
How to achieve this performance improvement?
Idea 1: Turn the ML Problem into a DB Problem
### Workload | Query Batch
--- | ---
Linear Regression | $\text{SUM}(X_i \ast X_j)$
Covariance Matrix | $\text{SUM}(X_i) \text{ GROUP BY } X_j$
 | $\text{SUM}(1) \text{ GROUP BY } X_i, X_j$
Decision Tree Node | $\text{VARIANCE}(Y) \text{ WHERE } X_j = c_j$
Mutual Information | $\text{SUM}(1) \text{ GROUP BY } X_i$
Rk-means | $\text{SUM}(1) \text{ GROUP BY } X_j$
 | $\text{SUM}(1) \text{ GROUP BY } \text{Center}_1, \ldots, \text{Center}_k$
Through DB Glasses, Everything is a Batch of Queries

<table>
<thead>
<tr>
<th>Workload</th>
<th>Query Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Regression</td>
<td>$\text{SUM}(X_i \cdot X_j)$ [ WHERE $\sum_k X_k \cdot w_k &lt; c$ ]</td>
</tr>
<tr>
<td>Covariance Matrix</td>
<td>$\text{SUM}(X_i) \text{ GROUP BY } X_j$ [ WHERE \ldots ]</td>
</tr>
<tr>
<td>(Non)poly. loss</td>
<td>$\text{SUM}(1) \text{ GROUP BY } X_i, X_j$ [ WHERE \ldots ]</td>
</tr>
<tr>
<td>Decision Tree Node</td>
<td>$\text{VARIANCE}(Y) \text{ WHERE } X_j = c_j$</td>
</tr>
<tr>
<td>Mutual Information</td>
<td>$\text{SUM}(1) \text{ GROUP BY } X_i$</td>
</tr>
<tr>
<td>R$k$-means</td>
<td>$\text{SUM}(1) \text{ GROUP BY } X_j$</td>
</tr>
<tr>
<td></td>
<td>$\text{SUM}(1) \text{ GROUP BY } \text{Center}_1, \ldots, \text{Center}_k$</td>
</tr>
</tbody>
</table>
# Through DB Glasses, Everything is a Batch of Queries

## Workload Query Batch

<table>
<thead>
<tr>
<th>Workload</th>
<th>Query Batch</th>
<th># Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Regression</td>
<td>( \text{SUM}(X_i \times X_j) ) [ WHERE ( \sum_k X_k \times w_k &lt; c ) ]</td>
<td>814</td>
</tr>
<tr>
<td>Covariance Matrix</td>
<td>( \text{SUM}(X_i) \text{ GROUP BY } X_j ) [ WHERE ... ]</td>
<td></td>
</tr>
<tr>
<td>(Non)poly. loss</td>
<td>( \text{SUM}(1) \text{ GROUP BY } X_i, X_j ) [ WHERE ... ]</td>
<td></td>
</tr>
<tr>
<td>Decision Tree Node</td>
<td>\text{VARIANCE}(Y) \text{ WHERE } X_j = c_j</td>
<td>3,141</td>
</tr>
<tr>
<td>Mutual Information</td>
<td>( \text{SUM}(1) \text{ GROUP BY } X_i )</td>
<td>56</td>
</tr>
<tr>
<td>Rk-means</td>
<td>( \text{SUM}(1) \text{ GROUP BY } X_j ) \text{ SUM}(1) \text{ GROUP BY } Center_1, \ldots, Center_k</td>
<td>41</td>
</tr>
</tbody>
</table>

(# Queries shown for Retailer dataset with 39 attributes)

Queries in a batch:

- Same aggregates but over different attributes
- Expressed over the same join of the database relations

AMPLE opportunities for sharing computation in a batch.
Models under Consideration

So far:
- Polynomial regression
- Factorisation machines
- Classification/regression trees
- Mutual information
- Chow Liu trees
- $k$-means clustering
- $k$-nearest neighbours
- (robust, ordinal) PCA
- SVM

On-going:
- Boosting regression trees
- AdaBoost
- Sum-product networks
- Random forests
- Logistic regression
- Linear algebra:
  - QR decomposition
  - SVD
  - low-rank matrix factorisation

All these cases can benefit from **structure-aware computation**
Natural Attempt:

Use Existing DB System to Compute Query Batch
Existing DBMSs are **NOT** Designed for Query Batches

Relative Speedup for **Our Approach** over DBX and MonetDB

C = Covariance Matrix; R = Regression Tree Node; AWS d2.xlarge (4 vCPUs, 32GB)
Existing DSMSs are **NOT** Designed for Query Batches

Task: Maintain the covariance matrix over Retailer

- Round-robin insertions in all relations
- All maintenance strategies implemented in DBToaster

![Graph showing throughput comparison between F-IVM, higher-order IVM, and first-order IVM. The graph indicates that F-IVM has the highest throughput at approximately 1E+07 tuples/sec, followed by higher-order IVM and then first-order IVM.](image)

Azure DS14, Intel Xeon, 2.40GHz, 112GB, 1 thread; one hour timeout