Learning Models over Relational Databases

fdbresearch.github.io  relational.ai

Dan Olteanu
Oxford & relationalAI

ICDT & EDBT Keynote
Lisbon, March 2019
Acknowledgments

**FDB team, in particular:**

Jakub  
Max  
Milos  
Ahmet  
Fabian

**relationalAI team, in particular:**

Mahmoud  
Hung  
Long
ML is emerging as general purpose technology

- Just as computing became general purpose 70 years ago

A core ability of intelligence is the ability to predict

- Turn information you have into information you need

The quality of the prediction is increasing as the cost per prediction is decreasing
Most Enterprises Rely on Relational Data for Their ML Models

Retail: 86% relational
Insurance: 83% relational
Marketing: 82% relational
Financial: 77% relational

(based on 2017 Kaggle survey of 16,000 ML practitioners)
Last decades have witnessed massive adoption of the Relational Model.

Many human hours invested in building relational models.

Relational databases are rich with knowledge of the underlying domains.

Availability of curated data made it possible to learn from the past and to predict the future for both humans (BI) and machines (ML).
Current State of Affairs in Building Predictive Models

Stores
- store_id
- zipcode
- area_sq_ft
- avghhi
- distance_to_comp1
- distance_to_comp2

Demographics
- zipcode
- population
- ethnicities
- households
- median_age

Inventory
- store_id
- date
- item_id
- inventory_units

Weather
- store_id
- date
- rain
- snow
- thunder
- min_temperature
- max_temperature
- mean_wind

Items
- item_id
- category
- subcategory
- category_cluster
- price

Data matrix
Features
Samples

Current ML technology throws away the relational structure that can help build faster better models.
Current ML technology throws away the relational structure that can help build faster better models.
Learning over Relational Databases: Revisit from First Principles
Structure-aware versus Structure-agnostic Learning

Structure-agnostic learning requires:

1. Materialisation of the query result
   (Recomputation in case of data updates)
2. Data export from DBMS and import into ML tool
3. One/multi-hot encoding of categorical variables
Structure-aware versus Structure-agnostic Learning

Structure-agnostic learning requires:

1. Materialisation of the query result
   (Recomputation in case of data updates)

2. Data export from DBMS and import into ML tool

3. One/multi-hot encoding of categorical variables

All these steps are very expensive and unnecessary!
Structure-aware versus Structure-agnostic Learning

Structure-aware learning avoids the three expensive steps.
Structure-aware Learning

FASTER even than

Feature Extraction Query!
Example: A Retailer Use Case

<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>23 GB</td>
</tr>
</tbody>
</table>
Structure-aware versus Structure-agnostic Learning

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>Database</td>
<td>2.1 GB</td>
</tr>
<tr>
<td>Join</td>
<td>152.06 secs</td>
</tr>
<tr>
<td>Export</td>
<td>351.76 secs</td>
</tr>
<tr>
<td>Shuffling</td>
<td>5,488.73 secs</td>
</tr>
<tr>
<td>Query batch</td>
<td>–</td>
</tr>
<tr>
<td>Grad Descent</td>
<td>7,249.58 secs</td>
</tr>
<tr>
<td>Total time</td>
<td>13,242.13 secs</td>
</tr>
</tbody>
</table>
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 600× 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.
TensorFlow’s Behaviour is the Rule, not the Exception!

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, factorization machines, $k$-means, ..

This is to be contrasted with the scalability of DBMSs!
How to Achieve
This Performance Improvement?
Idea 1: Turn the Learning Problem into a Database Problem
Through DB Glasses, Everything is a Batch of Queries

- **Gradient Computation** → **Query Batch**

- **Decision Tree Split Cost** → **Query Batch**

- **$k$-means Clustering Assignment** → **Query Batch**
Models under Consideration

<table>
<thead>
<tr>
<th>So far:</th>
<th>On-going:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Polynomial regression</td>
<td>• Boosting regression trees</td>
</tr>
<tr>
<td>• Factorization machines</td>
<td>• AdaBoost</td>
</tr>
<tr>
<td>• Classification/regression trees</td>
<td>• Sum-product networks</td>
</tr>
<tr>
<td>• Mutual information</td>
<td>• Random forests</td>
</tr>
<tr>
<td>• Chow Liu trees</td>
<td>• Logistic regression</td>
</tr>
<tr>
<td>• $k$-means clustering</td>
<td>• Linear algebra:</td>
</tr>
<tr>
<td>• $k$-nearest neighbours</td>
<td>• QR decomposition</td>
</tr>
<tr>
<td>• PCA</td>
<td>• SVD</td>
</tr>
<tr>
<td>• SVM</td>
<td>• low-rank matrix factorisation</td>
</tr>
</tbody>
</table>
Models under Consideration

So far:

- Polynomial regression
- Factorization machines
- Classification/regression trees
- Mutual information
- Chow Liu trees
- $k$-means clustering
- $k$-nearest neighbours
- 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
Example: Ridge Linear Regression

Linear function: \( f_\theta(x) = \langle \theta, x \rangle = \theta_0 x_0 + \theta_1 x_1 + \ldots \)

- Training dataset \( D \) defined by feature extraction query and consists of tuples \((x, y)\) of feature vector \( x \) and response \( y \)
- Parameters \( \theta \) obtained by minimising the objective function:

\[
J(\theta) = \frac{1}{2|D|} \sum_{(x, y) \in D} (\langle \theta, x \rangle - y)^2 + \frac{\lambda}{2} \|\theta\|^2
\]
We can solve $\theta^* := \arg\min_\theta J(\theta)$ by repeatedly updating $\theta$ in the direction of the gradient until convergence:

$$\theta := \theta - \alpha \cdot \nabla J(\theta).$$

**Model reformulation idea**: Decouple

- data-dependent ($x, y$) computation from
- data-independent ($\theta$) computation

in the formulations of the objective $J(\theta)$ and its gradient $\nabla J(\theta)$. 
\[ J(\theta) = \frac{1}{2|D|} \sum_{(x,y) \in D} (\langle \theta, x \rangle - y)^2 + \frac{\lambda}{2} \| \theta \|_2^2 \]

\[ = \frac{1}{2} \theta^\top \Sigma \theta - \langle \theta, c \rangle + \frac{sY}{2} \]

\[ \nabla J(\theta) = \Sigma \theta - c + \lambda \theta, \text{ where} \]
\[ J(\theta) = \frac{1}{2|D|} \sum_{(x,y)\in D} (\langle \theta, x \rangle - y)^2 + \frac{\lambda}{2} \| \theta \|_2^2 \]

\[ = \frac{1}{2} \theta^\top \Sigma \theta - \langle \theta, c \rangle + \frac{s_Y}{2} \]

\[ \nabla J(\theta) = \Sigma \theta - c + \lambda \theta, \text{ where} \]

matrix \( \Sigma = (\sigma_{ij})_{i,j\in[n]} \), vector \( c = (c_i)_{i\in[n]} \), and scalar \( s_Y \) are:

\[ \sigma_{ij} = \frac{1}{|D|} \sum_{(x,y)\in D} x_i x_j \quad c_i = \frac{1}{|D|} \sum_{(x,y)\in D} y \cdot x_i \quad s_Y = \frac{1}{|D|} \sum_{(x,y)\in D} y^2 \]
$\Sigma, c, s_Y$ can be Expressed as Batch of Queries

Queries for $\sigma_{ij} = \frac{1}{|D|} \sum_{(x,y) \in D} x_i x_j^\top$ (w/o factor $\frac{1}{|D|}$):
The natural text is as follows:

\[ \Sigma, c, s_Y \text{ can be Expressed as Batch of Queries} \]

Queries for \( \sigma_{ij} = \frac{1}{|D|} \sum_{(x,y) \in D} x_i x_j \) (w/o factor \( \frac{1}{|D|} \)):

- \( x_i, x_j \) continuous

\[
\text{SELECT SUM (} x_i \ast x_j \text{) FROM } D ;
\]

where \( D \) is the feature extraction query over the input DB.
$\Sigma, c, s_Y$ can be Expressed as Batch of Queries

Queries for $\sigma_{ij} = \frac{1}{|D|} \sum_{(x,y) \in D} x_i x_j^\top$ (w/o factor $\frac{1}{|D|}$):

- $x_i, x_j$ continuous
  
  ```sql
  SELECT SUM (x_i * x_j) FROM D;
  ```

- $x_i$ categorical, $x_j$ continuous
  
  ```sql
  SELECT x_i, SUM(x_j) FROM D GROUP BY x_i;
  ```

where $D$ is the feature extraction query over the input DB.
Σ, c, s_Y can be Expressed as Batch of Queries

Queries for $\sigma_{ij} = \frac{1}{|D|} \sum_{(x,y) \in D} x_i x_j^\top$ (w/o factor $\frac{1}{|D|}$):

- $x_i, x_j$ continuous
  
  $$\text{SELECT } \text{SUM} (x_i \times x_j) \text{ FROM } D;$$

- $x_i$ categorical, $x_j$ continuous
  
  $$\text{SELECT } x_i, \text{SUM}(x_j) \text{ FROM } D \text{ GROUP BY } x_i;$$

- $x_i, x_j$ categorical
  
  $$\text{SELECT } x_i, x_j, \text{SUM}(1) \text{ FROM } D \text{ GROUP BY } x_i, x_j;$$

where $D$ is the feature extraction query over the input DB.
Query batch sizes in practice:

<table>
<thead>
<tr>
<th>Application/Dataset</th>
<th>Retailer</th>
<th>Favorita</th>
<th>Yelp</th>
<th>TPC-DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covariance Matrix</td>
<td>937</td>
<td>157</td>
<td>730</td>
<td>3,299</td>
</tr>
<tr>
<td>Decision Tree (one node)</td>
<td>3,150</td>
<td>273</td>
<td>1,392</td>
<td>4,299</td>
</tr>
<tr>
<td>k-means</td>
<td>44</td>
<td>19</td>
<td>38</td>
<td>92</td>
</tr>
</tbody>
</table>
Aggregation is the Aspirin to All Problems

Query batch sizes in practice:

<table>
<thead>
<tr>
<th>Application/Dataset</th>
<th>Retailer</th>
<th>Favorita</th>
<th>Yelp</th>
<th>TPC-DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covariance Matrix</td>
<td>937</td>
<td>157</td>
<td>730</td>
<td>3,299</td>
</tr>
<tr>
<td>Decision Tree (one node)</td>
<td>3,150</td>
<td>273</td>
<td>1,392</td>
<td>4,299</td>
</tr>
<tr>
<td>k-means</td>
<td>44</td>
<td>19</td>
<td>38</td>
<td>92</td>
</tr>
</tbody>
</table>

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.
Natural Attempt:

Use Existing DB Technology to Compute the 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

Throughput vs. Fraction of Retailer Stream Processed

---

Azure DS14, Intel Xeon, 2.40GHz, 112GB, 1 thread; one hour timeout
Idea 2: Exploit the Problem Structure to Lower the Complexity

<table>
<thead>
<tr>
<th>author</th>
<th>book</th>
<th>place</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lewis</td>
<td>Narnia</td>
<td>Eagle&amp;Child</td>
<td>pipe</td>
</tr>
<tr>
<td>Lewis</td>
<td>Narnia</td>
<td>Eagle&amp;Child</td>
<td>chips</td>
</tr>
<tr>
<td>Tolkien</td>
<td>Hobbit</td>
<td>Eagle&amp;Child</td>
<td>pint</td>
</tr>
<tr>
<td>Tolkien</td>
<td>Hobbit</td>
<td>Eagle&amp;Child</td>
<td>chips</td>
</tr>
<tr>
<td>Tolkien</td>
<td>Hobbit</td>
<td>Eagle&amp;Child</td>
<td>pint</td>
</tr>
</tbody>
</table>
Algebraic structure: (semi)rings $(\mathcal{R}, +, \cdot, 0, 1)$

- Distributivity law $\rightarrow$ Factorisation

  Factorised Databases $[\text{VLDB’12+’13}]$

  Factorised Machine Learning

  $[\text{SIGMOD’16, VLDB’16, SIGREC’16, DEEM’18, PODS’18+’19}]$
Structure-aware Tools of a Database Researcher

Algebraic structure: (semi)rings \((\mathcal{R}, +, *, 0, 1)\)

- Distributivity law → Factorisation
  
  Factorised Databases
  
  \[\text{Factorised Databases} \quad [\text{VLDB’12+’13}]
  
  Factorised Machine Learning
  
  \[\text{Factorised Machine Learning} \quad [\text{SIGMOD’16, VLDB’16, SIGREC’16, DEEM’18, PODS’18+’19}]
  
- Additive inverse → Uniform treatment of updates
  
  Factorised Incremental Maintenance
  
  \[\text{Factorised Incremental Maintenance} \quad [\text{SIGMOD’18}]\]
Structure-aware Tools of a Database Researcher

Algebraic structure: (semi)rings \((\mathcal{R}, +, \ast, 0, 1)\)

- Distributivity law \(\rightarrow\) Factorisation
  
  Factorised Databases
  Factorised Machine Learning
  
  \([\text{VLDB}'12+'13]\)
  \([\text{SIGMOD}'16, \text{VLDB}'16, \text{SIGREC}'16, \text{DEEM}'18, \text{PODS}'18+'19]\)

- Additive inverse \(\rightarrow\) Uniform treatment of updates
  
  Factorised Incremental Maintenance
  
  \([\text{SIGMOD}'18]\)

- Sum-Product abstraction \(\rightarrow\) Same processing for distinct tasks
  
  DB queries, Covariance matrix, PGM inference, Matrix chain multiplication
  
  \([\text{SIGMOD}'18]\)
Combinatorial structure: query width and data degree measures

- **Width measure** $w$ for FEQ $\rightarrow$ Low complexity $\tilde{O}(N^w)$

  factorisation width $\geq$ fractional hypertree width $\geq$ sharp-submodular width

  worst-case optimal size and time for factorised joins

  [ICDT'12+'18,TODS'15,PODS'19]
Combinatorial structure: query width and data degree measures

- **Width measure** $w$ for FEQ $\rightarrow$ Low complexity $\tilde{O}(N^w)$
  - factorisation width $\geq$ fractional hypertree width $\geq$ sharp-submodular width
  - worst-case optimal size and time for factorised joins
  
  \[ \text{[ICDT'12+’18, TODS’15, PODS’19]} \]

- **Degree** $\rightarrow$ Adaptive processing depending on high/low degrees
  - worst-case optimal incremental maintenance for triangle count \[ \text{[ICDT’19a]} \]
  - evaluation of queries with negated relations of bounded degree \[ \text{[ICDT’19b]} \]
Combinatorial structure: query width and data degree measures

- **Width measure** $w$ for FEQ $\rightarrow$ Low complexity $\tilde{O}(N^w)$
  
  factorisation width $\geq$ fractional hypertree width $\geq$ sharp-submodular width
  
  worst-case optimal size and time for factorised joins
  
  [ICDT'12+, TODS'15, PODS'19]

- **Degree** $\rightarrow$ Adaptive processing depending on high/low degrees
  
  worst-case optimal incremental maintenance for triangle count
  
  evaluation of queries with negated relations of bounded degree
  
  [ICDT'19a]

- **Functional dependencies** $\rightarrow$ Learn simpler, equivalent models
  
  reparameterisation of polynomial regression models and factorisation machines
  
  [PODS'18, TODS'19]
Statistical structure: sampling for joins and models, coresets

- Sampling through joins: ripple/wander joins  \[\text{SIGMOD'99+’16}\]

- Sampling specific to classes of models  \[\text{SIGMOD'19}\]

- Succinct approximate data representations
  coresets for $k$-means with constant-factor approximation  \[\text{submission’19}\]
Case in Point (1):
Factorised Query Computation
⇒
Exponential Time/Size Improvement
Example: Factorised Query Computation

<table>
<thead>
<tr>
<th>Orders (O for short)</th>
<th>Dish (D for short)</th>
<th>Items (I for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>day</td>
<td>dish</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
</tr>
<tr>
<td>Steve</td>
<td>Friday</td>
<td>hotdog</td>
</tr>
<tr>
<td>Joe</td>
<td>Friday</td>
<td>hotdog</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Factorised Query Computation

<table>
<thead>
<tr>
<th>Orders (O for short)</th>
<th>Dish (D for short)</th>
<th>Items (I for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>day</td>
<td>dish</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
</tr>
<tr>
<td>Steve</td>
<td>Friday</td>
<td>hotdog</td>
</tr>
<tr>
<td>Joe</td>
<td>Friday</td>
<td>hotdog</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider the natural join of the above relations:

\[
O(\text{customer, day, dish}), D(\text{dish, item}), I(\text{item, price})
\]

<table>
<thead>
<tr>
<th>customer</th>
<th>day</th>
<th>dish</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Factorised Query Computation

<table>
<thead>
<tr>
<th>customer</th>
<th>day</th>
<th>dish</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
</tbody>
</table>

An algebraic encoding uses product ($\times$), union ($\cup$), and values:
There are several algebraically equivalent factorised joins defined by distributivity of product over union and their commutativity.
Observation:

- price is under item, which is under dish, but only depends on item,
- .. so the same price appears under an item regardless of the dish.

Idea: Cache price for a specific item and avoid repetition!
COUNT(*) computed in one pass over the factorisation:

- values $\mapsto 1$,
- $\cup \mapsto +$, $\times \mapsto \ast$. 
COUNT(*) computed in one pass over the factorisation:

- values $\mapsto 1$,
- $\cup \mapsto +$, $\times \mapsto *$. 
SUM(dish * price) computed in one pass over the factorisation:

- Assume there is a function $f$ that turns dish into numbers.
- All values except for dish & price $\mapsto 1$,
- $\cup \mapsto +$, $\times \mapsto \ast$. 
SUM(dish * price) computed in one pass over the factorisation:

- Assume there is a function $f$ that turns dish into numbers.
- All values except for dish & price $\mapsto 1$,
- $\cup \mapsto +$, $\times \mapsto \ast$. 
Case in Point (2):
Sum-Product Ring Abstraction
⇒
Sharing Aggregate Computation
Shared Computation of Several Aggregates (1/2)

Ring for computing \( \text{SUM}(1) \), \( \text{SUM}(\text{price}) \), \( \text{SUM}(\text{price} \times \text{dish}) \):

- Elements = triples of numbers, one per aggregate
- Sum (+) and product (*) now defined over triples
  They enable shared computation across the aggregates
Ring for computing SUM(1), SUM(price), SUM(price * dish):

- Elements = triples of numbers, one per aggregate
- Sum (+) and product (*) now defined over triples
  They enable shared computation across the aggregates
Ring Generalisation for the Entire Covariance Matrix

Ring \((\mathcal{R}, +, \ast, 0, 1)\) over triples of aggregates \((c, s, Q) \in \mathcal{R}\):

\[
\begin{pmatrix}
\sum(1) & \sum(x_i) & \sum(x_i \ast x_j)
\end{pmatrix}
\]

\[
(c_1, s_1, Q_1) + (c_2, s_2, Q_2) = (c_1 + c_2, s_1 + s_2, Q_1 + Q_2)
\]

\[
(c_1, s_1, Q_1) \ast (c_2, s_2, Q_2) = (c_1 \cdot c_2, c_2 \cdot s_1 + c_1 \cdot s_2, c_2 \cdot Q_1 + c_1 \cdot Q_2 + s_1 s_2^T + s_2 s_1^T)
\]

\[
0 = (0, 0_{n \times 1}, 0_{n \times n})
\]

\[
1 = (1, 0_{n \times 1}, 0_{n \times n})
\]

- \(\sum(1)\) reused for all \(\sum(x_i)\) and \(\sum(x_i \ast x_j)\)
- \(\sum(x_i)\) reused for all \(\sum(x_i \ast x_j)\)
Idea 3: Lower the Constant Factors
1. **Specialisation** for workload and data
   - Generate code specific to the query batch and dataset
   - Improve cache locality for hot data path

2. **Sharing low-level data access**
   - Aggregates decomposed into views over join tree
   - Share data access across views with different output schemas

3. **Parallelisation**: multi-core (SIMD & distribution to come)
   - Task and domain parallelism

[DEEM’18, SIGMOD’19]
Added optimisations for covariance matrix computation:

specialisation $\rightarrow$ + sharing $\rightarrow$ + parallelization

AWS d2.xlarge (4 vCPUs, 32GB)
Case in Point (3):
Code Optimisations
⇒
Non-trivial Speedup
Sharing Computation for a Query Batch in Favorita

Sales: date, store, item, units, promo
Holidays: date, htype, locale, transferred
Stores: store, city, state, stype, cluster
Items: item, family, class, perishable
Transactions: date, store, txns
Oil: date, price

Aggregates to compute over the join of relations:

\[ Q_1: \text{SUM}(f(\text{units})) \]

\[ Q_2: \text{SUM}(g(\text{item}) \cdot h(\text{date}, \text{family})) \text{ GROUP BY \text{store}} \]

\[ Q_3: \text{SUM}(f(\text{units}) \cdot g(\text{item})) \text{ GROUP BY \text{family}} \]
Sharing Computation for a Query Batch in Favorita

Sales: date, store, item, units, promo
Holidays: date, htype, locale, transferred
Stores: store, city, state, stype, cluster
Items: item, family, class, perishable
Transactions: date, store, txns
Oil: date, price

Aggregates to compute over the join of relations:

\[ Q_1: \text{SUM} \left( f(\text{units}) \right) \]
\[ Q_2: \text{SUM} \left( g(\text{item}) \cdot h(\text{date, family}) \right) \quad \text{GROUP BY store} \]
\[ Q_3: \text{SUM} \left( f(\text{units}) \cdot g(\text{item}) \right) \quad \text{GROUP BY family} \]
For each query, decide its output (root) node in the join tree

Break down each query into directional views over the join tree

Merge/share/group views for different queries

Computational unit: group of views
Parallelisation: Dependency Graph of View Groups

- **Task parallelism:** Evaluate independent groups in parallel
- **Domain parallelism:** Partition the large relation used for each group
Traverse Sales as a trie following an order of its join variables
Lookup into incoming views, e.g., $V_i^{(1)}$, as early as possible
Code Generation for Executing View Group 6 over Sales

\[ V_1^{(1)} \xrightarrow{\text{item}} \]
\[ V_1^{(2)} \xrightarrow{\text{item}} \]
\[ V_H \rightarrow \text{date} \]
\[ V_T \rightarrow \text{store} \]

\[ \alpha_4 = 0; \]
\[ \text{foreach } i \in \pi_{\text{item}}(S \bowtie_{\text{item}} V_1^{(1)} \bowtie_{\text{item}} V_1^{(2)}) \]
\[ \alpha_i^{(1)} = V_1^{(1)}(i) \]
\[ \alpha_5 = g(i); \]
\[ \alpha_3 = 0; \]
\[ \text{foreach } d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_H \bowtie_{\text{date}} V_T) \]
\[ \alpha_H = V_H(d); \]
\[ \text{foreach } y \in \pi_{\text{family}}(\sigma_{\text{item}=i} V_1^{(2)}); \]
\[ \alpha_6 + = h(d, y) \cdot V_1^{(2)}(i, y). \]
\[ \alpha_2 = 0; \]
\[ \text{foreach } s \in \pi_{\text{store}}(\sigma_{\text{item}=i, \text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_T) \]
\[ \alpha_T = V_T(d, s); \]
\[ \alpha_1 = 0; \]
\[ \text{foreach } u \in \pi_{\text{units}}(\sigma_{\text{item}=i, \text{date}=d, \text{store}=s} S); \]
\[ \alpha_1 + = f(u); \]
\[ \alpha_2 + = \alpha_1 \cdot \alpha_T; \]
\[ \text{if } Q_5(s) \text{ then } Q_5(s) + = \alpha_7 \cdot \alpha_8 \cdot \alpha_T \text{ else } Q_5(s) = \alpha_7 \cdot \alpha_8 \cdot \alpha_T; \]
\[ \alpha_3 + = \alpha_2 \cdot \alpha_H; \]
\[ \alpha_4 + = \alpha_3 \cdot \alpha_i^{(1)} \]
\[ Q_1 = \alpha_4; \]

Insert code for partial aggregates as early as possible
\[ \forall i \in \pi_{\text{item}}(S \bowtie \text{item } V_i^{(1)} \bowtie \text{item } V_i^{(2)}) \]
\[ \alpha_i^{(1)} = V_i^{(1)}(i); \]
\[ \alpha_3 = 0; \]
\[ \forall d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie \text{date } V_H \bowtie \text{date } V_T) \]
\[ \alpha_H = V_H(d); \]
\[ \alpha_2 = 0; \]
\[ \forall s \in \pi_{\text{store}}(\sigma_{\text{item}=i, \text{date}=d} S \bowtie \text{store } \sigma_{\text{date}=d} V_T) \]
\[ \alpha_T = V_T(d, s); \]
\[ \alpha_1 = 0; \]
\[ \forall u \in \pi_{\text{units}} \sigma_{\text{item}=i, \text{date}=d, \text{store}=s} S : \alpha_1 \ += f(u); \]
\[ \alpha_2 \ += \alpha_1 \cdot \alpha_T; \]
\[ \alpha_3 \ += \alpha_2 \cdot \alpha_H; \]
\[ \alpha_4 \ += \alpha_3 \cdot \alpha_i^{(1)}; \]
\[ Q_1 = \alpha_4; \]

**Q_1**: SUM (f(units))
Code Generation for Executing View Group 6 over Sales

\[ V_{(1)}^{(1)} \rightarrow \text{item} \]
\[ V_{(2)}^{(1)} \rightarrow \text{item} \]
\[ V_H \rightarrow \text{date} \]
\[ V_T \rightarrow \text{store} \]

\[ \alpha_4 = 0; \]

\[ \text{foreach } i \in \pi_{\text{item}}(S \Join \text{item} V_{(1)}^{(1)} \Join \text{item} V_{(2)}^{(1)}) \]
\[ \alpha_i^{(1)} = V_{(1)}^{(1)}(i); \]
\[ \alpha_5 = g(i); \]
\[ \alpha_3 = 0; \]

\[ \text{foreach } d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \Join \text{date} V_H \Join \text{date} V_T) \]
\[ \alpha_H = V_H(d); \]

\[ \alpha_2 = 0; \]

\[ \text{foreach } s \in \pi_{\text{store}}(\sigma_{\text{item}=i,\text{date}=d} S \Join \text{store} \sigma_{\text{date}=d} V_T) \]
\[ \alpha_T = V_T(d, s); \]
\[ \alpha_1 = 0; \]

\[ \text{foreach } u \in \pi_{\text{units}}(\sigma_{\text{item}=i,\text{date}=d,\text{store}=s} S : \alpha_1 += f(u); \]
\[ \alpha_2 += \alpha_1 \cdot \alpha_T; \]

\[ \alpha_3 += \alpha_2 \cdot \alpha_H; \]

\[ \alpha_4 += \alpha_3 \cdot \alpha_i^{(1)}; \]

\[ Q_1 = \alpha_4; \]

\[ V_{S \rightarrow i}: \text{SUM } (f(\text{units}) \cdot g(\text{item})) \text{ GROUP BY item} \]
\( V_{(1)} \rightarrow \text{item} \)

\( V_{(2)} \rightarrow \text{item} \)

\( V_H \rightarrow \text{date} \)

\( V_T \rightarrow \text{store} \)

\( \alpha_4 = 0; \)

\( \text{foreach } i \in \pi_{\text{item}}(S \bowtie_{\text{item}} V_{(1)} \bowtie_{\text{item}} V_{(2)}) \)

\( \alpha_i^{(1)} = V_i^{(1)}(i); \)

\( \alpha_5 = g(i); \)

\( \alpha_3 = 0; \)

\( \text{foreach } d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_H \bowtie_{\text{date}} V_T) \)

\( \alpha_H = V_H(d); \quad \alpha_6 = 0; \)

\( \text{foreach } y \in \pi_{\text{family}}(\sigma_{\text{item}=i} V_{(2)} : \alpha_6 += h(d, y) \cdot V_i^{(2)}(i, y); \)

\( \alpha_2 = 0; \quad \alpha_7 = \alpha_6 \cdot \alpha_5 \cdot \alpha_H; \)

\( \text{foreach } s \in \pi_{\text{store}}(\sigma_{\text{item}=i, \text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_T) \)

\( \alpha_T = V_T(d, s); \quad \alpha_1 = 0; \quad \alpha_8 = |\sigma_{\text{item}=i, \text{date}=d, \text{store}=s} S|; \)

\( \text{foreach } u \in \pi_{\text{units}}(\sigma_{\text{item}=i, \text{date}=d, \text{store}=s} S : \alpha_1 += f(u); \)

\( \alpha_2 += \alpha_1 \cdot \alpha_T; \)

\( \text{if } Q_2(s) \text{ then } Q_2(s) += \alpha_7 \cdot \alpha_8 \cdot \alpha_T \text{ else } Q_2(s) = \alpha_7 \cdot \alpha_8 \cdot \alpha_T; \)

\( \alpha_3 += \alpha_2 \cdot \alpha_H; \)

\( \alpha_4 += \alpha_3 \cdot \alpha_i^{(1)}; \quad V_{S \rightarrow I}(i) = \alpha_3 \cdot \alpha_5; \)

\( Q_1 = \alpha_4; \)

\( Q_2: \text{SUM } (g(\text{item}) \cdot h(\text{date}, \text{family})) \quad \text{GROUP BY store} \)
Three-step recipe for efficient machine learning over databases:

1. Turn the learning problem into a database problem
2. Exploit the problem structure to lower the complexity
3. Specialise and optimise the code to lower the constant factors

Q.E.D.
Call to arms

We need more sustained work on theory and systems for Structure-aware Approaches to Data Analytics


[VLDB’13] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. *Aggregation and Ordering in Factorised Databases*. In VLDB 2013


[DEEM’18] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. AC/DC: In-Database Learning Thunderstruck. In DEEM@SIGMOD 2018

[ICDT’18] Ahmet Kara and Dan Olteanu. Covers of Query Results. In ICDT 2018

[PODS’18] Mahmoud Abo Khamis, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-Database Learning with Sparse Tensors. In PODS 2018


