Quasar: A High-Performance Scoring and Ranking Library

May 20, 2019

Fei Chen
Senior Engineering Manager

Andris Birkmanis
Staff Software Engineer
Background
Scoring and Ranking Paradigm

Request → Candidate List of Items → Feature Computation → Scoring → Ranking → Final List of Items → List of Items

Relevance Service
Feature Computation and Scoring

item1 -> featureX1 -> score1

item2 -> featureX2 -> score2

itemN -> featureXN -> scoreN
## Spreadsheet Metaphor

<table>
<thead>
<tr>
<th>Item</th>
<th>FeatureX</th>
<th>FeatureY</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>item1</td>
<td>0.3</td>
<td>-0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>item2</td>
<td>0.9</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Ranking

<table>
<thead>
<tr>
<th>Item</th>
<th>X</th>
<th>Y</th>
<th>S1</th>
</tr>
</thead>
<tbody>
<tr>
<td>item1</td>
<td>0.1</td>
<td>0.3</td>
<td>0.5</td>
</tr>
<tr>
<td>item2</td>
<td>-0.3</td>
<td>0.2</td>
<td>0.8</td>
</tr>
<tr>
<td>item3</td>
<td>-0.4</td>
<td>0.1</td>
<td>0.6</td>
</tr>
<tr>
<td>item4</td>
<td>0.3</td>
<td>0.5</td>
<td>0.9</td>
</tr>
<tr>
<td>item5</td>
<td>0.3</td>
<td>0.1</td>
<td>0.7</td>
</tr>
<tr>
<td>item6</td>
<td>0.1</td>
<td>-0.2</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**TOP 3 BY S1**

<table>
<thead>
<tr>
<th>Item</th>
<th>X</th>
<th>Y</th>
<th>S1</th>
</tr>
</thead>
<tbody>
<tr>
<td>item2</td>
<td>-0.3</td>
<td>0.2</td>
<td>0.8</td>
</tr>
<tr>
<td>item4</td>
<td>0.3</td>
<td>0.5</td>
<td>0.9</td>
</tr>
<tr>
<td>item5</td>
<td>0.3</td>
<td>0.1</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Why Quasar?

Prior to Quasar
• Product-specific solutions
• Highly optimized for each product
• Little sharing/reuse across products

Business Challenges
Scale ML efforts when
• # of products increases
• # of engineers increases
• Model complexity increases

Key Business Metrics
• Productivity
• System efficiency

A Ubiquitous Library for
• Easily sharing modeling efforts
• Model portability
• Expressiveness and composability
• Performance tuned for various system constraints
Portability
Portability and Consistency

Quasar DAG for Training
- Input Data
- Feature Producer
- Features
- Learning Algorithm
- Labels

Quasar DAG for Scoring
- Input Data
- Feature Producer
- Features
- Trained Model Parameters
- Scorer
- Label
Quasar DSL by An Example

1. Problem
   - Generate an engaging feed for a member.

2. Input & Output
   **Input**
   - Member profile
   - List of all posts by member’s connections
   **Output**
   - Top posts ordered by relevance

3. Relevance Model
   - Do member’s interests match post topics?
   - Is the post fresh?
Quasar Scoring DAG of the Example

- **Expressive**
  - Arithmetic and logical expressions
  - Commonly used scoring models
- **Strongly Typed**
  - LONG, FLOAT, DOUBLE, STRING,
  - Dense/Sparse Vectors and Tensors
- **Declarative**
  - Specification order != execution order
  - Immutable types

- **Member**
  - **Interests**
  - **Topic Categories**
  - **Published Time**
- **Feed Item**
  - **Interest Match**
  - **Normalized Time**
- **Linear Model Score**
DSL for Model Composition: Ensembles

- **Member**
  - Interests
    - Interest Match
      - Linear Model Score
  - Feed Item
    - Topic Categories
    - Published Time
      - Normalized Time
        - Normalized View Times
        - Time Interaction
          - View Times
    - Weighted Score
DSL for Model Composition: Multi-Passes

- Member
  - Interests
  - Topic Categories
  - Published Time
- Feed Item
  - Normalized Time
  - View Times
  - Normalized View Times
  - Time Interaction
  - Weighted Score
  - Linear Model Score
  - Interest Match

FILTER BY
ORDER BY
Engine Optimization
Skipping Unneeded Feature Computation

*Required*

- Member
- Feed Item
- Interests
- Topic Categories
- Skills
- Content
- Interest Match
- Skill Match
- Content Match
Multi-pass Ranking: Avoid Repeating Computation

FILTER BY
Interest-match >0.5

FILTER BY
skill-match >0.7

Member

Feed
Item

Interests

Topic
Categories

Skills

Content

Interest
Match

Skill
Match

Content
Match
Batch-Column Order Evaluation
Compute Features with Code Locality

**Row-order** Feature Computation Flow

**Column-order** Feature Computation Flow
Engine Optimization: Multi-threaded Evaluation

<table>
<thead>
<tr>
<th>Item1</th>
<th>member</th>
<th>news feed</th>
<th>interest</th>
<th>category</th>
<th>Interest-match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item2</td>
<td>member</td>
<td>news feed</td>
<td>interest</td>
<td>category</td>
<td>Interest-match</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item3</td>
<td>member</td>
<td>news feed</td>
<td>interest</td>
<td>category</td>
<td>Interest-match</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item4</td>
<td>member</td>
<td>news feed</td>
<td>interest</td>
<td>category</td>
<td>Interest-match</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item5</td>
<td>member</td>
<td>news feed</td>
<td>interest</td>
<td>category</td>
<td>Interest-match</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item6</td>
<td>member</td>
<td>news feed</td>
<td>interest</td>
<td>category</td>
<td>Interest-match</td>
</tr>
</tbody>
</table>

Thread A

Thread B

Thread C
Post-adoption Challenges
Fragmentation Again

Planned a single solution
UDFs still fragmented
New strategy?
Feature Type Standardization

• Reducing degrees of freedom for types for features
  • Relations/Tensors
• Small composable operations
  • Relational algebra/Linear algebra/Einstein notation
• Automatic specialization and fusion of operations in codegen
• Stronger metadata management
  • Categorical and other dimensions

• Still leave ability to use Java UDFs as a fallback or for experimentation
Example of the new DSL

// Map operation
TENSOR[ActionType][VerbType][ObjectType]:FLOAT activity_log<i><j><k> = log(activity<i><j><k> + 1.0);

// Interaction
TENSOR[ActorType][VerbType][ObjectType]:FLOAT activity<i><j><k> = actor<i> * verb<j> * object<k>

// Join to get a subset
TENSOR[ActorType][VerbType][ObjectType]:FLOAT interestingActivities<i><j><k> = activity<i><j><k> * importantVerbs<j>;

// Lookup operation
FLOAT recent_viral_activity = activity_recent[VIRAL];
TENSOR[ActorType][ObjectType]:FLOAT likeActivities<i><k> = activityType<i>[LIKE]<k>;}
// Dot Product with explicit types
TENSOR[ActivityType]:FLOAT activities1 = ...;
TENSOR[ActivityType]:FLOAT activities2 = ...;
FLOAT similarity += activities1<i> * activities2<i>;

// Dot Product with inferred types
val activities1 = ...;
val activities2 = ...;
val similarity += activities1<i> * activities2<i>;
Future Directions

- Improving reusability via modules
- Enriching the standard feature type (nested relational?)
- Improving optimizations in codegen
- Backends other than JVM?
- Opensourcing
Thank you