The Tensor Data Platform
Towards an AI-centric Database System

Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, Matteo Interlandi
AI is growing... and having an impact on applications... and DBMS

Enter your favorite chart showing how AI is taking over the world

Unlocking the value of unstructured data at scale using BigQuery ML and object tables
Anatomy of next gen data-driven applications

1. Support for multimodal data (image, video, relational, audio, etc.)
   - Not many relational systems with proper image/video/etc. support
   - Many specialized systems are moving towards supporting "scalar" queries

2. Tight integration and interoperability with ML
   - Most systems either (partially) re-implement ML features in SQL...
   - ... Or call external ML runtimes

3. Native support for hardware acceleration
   - Most systems are built on single vendor tech (CUDA)
   - Supporting other stacks (AMD, Apple, etc.) requires nontrivial engineering effort

Claim: Building a data engine with all three is hard!
1. Support for multimodal data
   • Thanks to the Tensor abstraction

2. Native support for hardware acceleration
   • Large open-source communities with HW vendors involvement

3. Tight integration and interoperability with ML
   • ML capabilities embedded into the system and language (e.g., autodiff)

Question: Can we build a database on top of tensor runtimes?
AI-centric Database: Outline

1. Support for multimodal data

2. Native support for hardware acceleration

3. Tight integration and interoperability with ML
Def Tensor:

A multidimensional matrix that is a cornerstone data structure in AI
Tensor data representation

*Def Tensor:* A multidimensional matrix that is a cornerstone data structure in AI.
**Tensor data representation**

*Def Tensor:* A multidimensional matrix that is a cornerstone data structure in AI

We leverage `torch`, `torchaudio`, `torchvideo`, etc, for loading data into tensor format.

We have our own custom tensor class: `EncodedTensor = tensor + metadata`  
**PlainEncoding,**  
**DictionaryEncoding** (data tensor + 2-d dictionary metadata tensor)  
**ProbabilisticEncoding** (data tensor + a domain dictionary)  

---

**Table: Sales**

<table>
<thead>
<tr>
<th>saleid</th>
<th>prodid</th>
<th>date</th>
<th>region</th>
<th>receipt</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SQL on Images Demo
Quering images

The goal of this notebook is to show how image data can be loaded on TQP and how we can use TQP capabilities to query images.

1. Setup

2. Filter images based on Natural Language Query
AI-centric Database: Outline

1. Support for multimodal data

2. Native support for hardware acceleration

3. Tight integration and interoperability with ML
The Tensor Data Platform (TDP)

In process and **100% Python**!

Classical ML Inference: Hummingbird

SQL: TQP

Performance highlights

<table>
<thead>
<tr>
<th>Modality</th>
<th>ML (DNN)</th>
<th>ML (Classical)</th>
<th>SQL</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>video</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>audio</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tabular</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hardware

- CPU
- GPU
- TPU
- M1

[TDP] in process and 100% Python!

ML (DNN)

ML (Classical)

SQL

Graph

Common Tensor Library

Tensor Runtimes

Spark

SQL Server

DuckDB

RateUpDB

Performance highlights:

- TPCH SF 100: 103

Speedup (base/TDP)

- Spark: 103
- SQL Server: 5
- DuckDB: 10
- RateUpDB: 3

Classical ML Inference: Hummingbird

SQL: TQP
AI-centric Database: Outline

1. Support for multimodal data
2. Native support for hardware acceleration
3. Tight integration and interoperability with ML
SQL as a declarative language for Differentiable Programming

Gradients are the staple mechanism by which we learn in machine learning.

Tensor runtimes have a remarkable tool to compute gradients **Automatic Differentiation**

TDP extends SQL by taking advantage of automatic differentiation in PyTorch

Particularly, we add the following to SQL:
1. **Trainable User Defined Functions (UDFs) and Table Valued Functions (TVFs)**
2. **Differentiable Relational Operators** (e.g., Differentiable Group By, Aggregation, Filters, etc.)
Trainable SQL Queries

We can execute SQL queries that combines trainable operations with relational operators.

MNISTGrid Dataset

<table>
<thead>
<tr>
<th>Digit</th>
<th>Size</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Small</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Small</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Large</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Large</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Large</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Small</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Large</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Large</td>
<td>0</td>
</tr>
</tbody>
</table>

MNISTGrid Task

Compute the grouped (Digit, Size) counts from the image.

Trainable Query

```sql
SELECT Digit, Size, COUNT(*)
FROM parseMNISTGrid(MNISTGrid)
GROUP BY Digit, Size
```
Anatomy of a Trainable Query

```
SELECT Digit, Size, COUNT(*)
FROM parseMNISTGrid(MNISTGrid)
GROUP BY Digit, Size
```
SELECT Digit, Size, COUNT(*)
FROM parseMNISTGrid(MNISTGrid)
GROUP BY Digit, Size
Anatomy of a Trainable Query

```python
SELECT Digit, Size, COUNT(*)
FROM parseMNISTGrid(MNISTGrid)
GROUP BY Digit, Size

digit_parser = CNN(out_classes=10).to(device)
size_parser = CNN(out_classes=2).to(device)

@tdp_udf("Digit float, Size float")
def parseMNISTGrid(x: torch.Tensor) -> torch.Tensor:
    # Break up grid into a batch of 9 images
    grid = rearrange(x[0], "(h1 h2) (w1 w2) -> (h1 w1) 1 h2 w2", h1=3, w1=3)

    # Parse digits from images
    parsed_digits = digit_parser(grid)
digit_domain = np.arange(10)
encoded_digits = ProbabilisticEncoding.encode(parsed_digits, digit_domain)

    # Parse size from images
    parsed_sizes = size_parser(grid)
size_domain = np.arange(2)
encoded_sizes = ProbabilisticEncoding.from_encoded_data(parsed_sizes, size_domain)

    return encoded_digits, encoded_sizes
```

Trainable UDF
Anatomy of a Trainable Query

```sql
SELECT Digit, Size, COUNT(*)
FROM parseMNISTGrid(MNISTGrid)
GROUP BY Digit, Size
```

<table>
<thead>
<tr>
<th>Digit</th>
<th>Size</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Small</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Large</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Small</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Large</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Small</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Large</td>
<td>0</td>
</tr>
</tbody>
</table>

**Digit Parser**

**Size Parser**

**Trainable UDF**

**Differentiable Relational Operators**
Anatomy of a Trainable Query

**SELECT** Digit, Size, **COUNT(\*)**
**FROM** parseMNISTGrid(MNISTGrid)
**GROUP** **BY** Digit, Size

The query combines neural and relational operators and is end-to-end differentiable.
The alternative: pure Deep Learning

The standard way to tackle this problem would be to pose it as a multiple regression problem with a single monolithic neural network.

Disadvantages:
1. Entanglement of tasks (cannot separate digit classification from size classification or aggregation)
2. Cannot generalize to other tasks
3. Needs to learn from scratch what it means to group and count
Trainable Query vs pure Deep Learning

- **Datasets:**
  - MNISTGrid Train/Test: 5000/1000 Grids

- **Training Hyperparameters (Fixed):**
  - Learning Rate = 0.0001
  - Training Iterations = 40,000 iterations

- **Architecture (Varied):**
  - TDP Trainable Query (860K Parameters)
  - Pure Deep Learning CNN-Small (850K Parameters)
  - Resnet-18 (11.1M Parameters)

- 5 runs per architecture

Our approach trains significantly faster than a purely deep learning model

Our SQL can declaratively express Neurosymbolic [1] systems that are end-to-end trainable

[1] Neurosymbolic AI CACM oct 2022
Summary

🔥 The space of AI-powered databases is heating up

🚀 AI-centric Database could be a leap forward. Free-ride on:

1. $B of HW/SW investments for AI
2. Multimodal support
3. Seamless integration with latest and biggest ML models
4. Novel querying paradigms such as trainable queries

_exciting future directions

1. TensorFrame API
2. Expressing some ML tasks in a more natural way
   • Learning from Label Proportions
Thank you!

[Link to Gray Systems Lab](https://aka.ms/gsl)
ML-first user experience

ML within SQL: UDF-based programming model
- We use UDF to access the tensor API
- Still end-to-end on HW accelerators

```python
import torch
import torchtext

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

@tdp_udf("float")
def image_text_similarity(query: str, images: torch.Tensor) -> torch.Tensor:
    inputs = processor(text=[query], images=images, return_tensors="pt", padding=True)
    outputs = model(**inputs)
    scores = outputs.logits_per_image.flatten() / 30
    return scores
```
ML-first user experience

ML within SQL: UDF-based programming model
• We use UDF to access the tensor API
• Still end-to-end on HW accelerators

SQL within ML: Embedding queries into PyTorch programs
• Use the right tool for the right task
• Thanks to trainable SQL queries

---

```python
SELECT images
FROM Attachments
WHERE image_text_similarity("dog", images) > 0.80
```

```python
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

@tdp.udf("float")
def image_text_similarity(query: str, images: torch.Tensor) -> torch.Tensor:
    inputs = processor(text=[query], images=images, return_tensors="pt", padding=True)
    inputs.to(device)
    outputs = model(**inputs)
    scores = outputs.logits_per_image.flatten() / 30
    return scores

def train(compiled_query, num_iterations, optimizer, mnist_grids, target_counts):
    for i in range(num_iterations):
        optimizer.zero_grad()

        # Register MNISTGrid and perform inference with the query
tq.p.sql.register_tensor(mnist_grids[i], "MNIST_Grid")
predicted_counts = compiled_query.run()

        # Compute loss. Here we use MSE between the counts.
        loss = ((predicted_counts - target_counts[i])**2).mean()

        # Backpropagate and perform optimization step
        loss.backward()
        optimizer.step()
```
TQP

100% Python

TQP supports the full TPCH benchmark

Performance highlights

**SELECT**
  **MAX**(p.supplycost)  
  **AS** price,  
  s_name **AS** supp  
**FROM** supplier  
**JOIN** partsupp  
**ON**  
  ps_suppkey=s_suppkey  
**GROUP BY**  
  supplier.s_name  
**ORDER BY**  
  price **DESC**;
**TQP Scalability Comparison**

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Spark</th>
<th>SQL Server</th>
<th>DuckDB</th>
<th>RateUpDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>62</td>
<td>15</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>89</td>
<td>11</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>103</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

TQP: A100 with 80GB. Spark/SQLServer/DuckDB: 32 cores machine with 256GB. RateUp: Nvidia Quadro RTX 8000

- TQP has better perf/cost
- TQP has worst perf/cost
- Same perf/cost

1.5x
Differentiable Grouped Aggregation

Let’s see how we might make the “Group By + Aggregation” operation differentiable.

**Inventory**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>carrot</td>
<td>4.0</td>
</tr>
<tr>
<td>banana</td>
<td>carrot</td>
<td>2.0</td>
</tr>
<tr>
<td>apple</td>
<td>carrot</td>
<td>4.0</td>
</tr>
<tr>
<td>banana</td>
<td>potato</td>
<td>3.5</td>
</tr>
</tbody>
</table>

**Query**

```
SELECT Fruit, Vegetable, COUNT(*)
FROM Inventory
GROUP BY Fruit, Vegetable
```

**Query Answer**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>carrot</td>
<td>2</td>
</tr>
<tr>
<td>apple</td>
<td>potato</td>
<td>0</td>
</tr>
<tr>
<td>banana</td>
<td>carrot</td>
<td>1</td>
</tr>
<tr>
<td>banana</td>
<td>potato</td>
<td>1</td>
</tr>
</tbody>
</table>
Differentiable Grouped Aggregation

Let’s see how we might make the “Group By + Aggregation” operation differentiable.

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>carrot</td>
<td>4.0</td>
</tr>
<tr>
<td>banana</td>
<td>carrot</td>
<td>2.0</td>
</tr>
<tr>
<td>apple</td>
<td>carrot</td>
<td>4.0</td>
</tr>
<tr>
<td>banana</td>
<td>potato</td>
<td>3.5</td>
</tr>
</tbody>
</table>

We can do this in three steps:
1. Relax discrete data to continuous representation.
2. Create masks corresponding to each group.
3. Perform aggregation using the mask and data.
Differentiable Grouped Aggregation

Let’s see how we might make the “Group By + Aggregation” operation differentiable.

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>carrot</td>
<td>4.0</td>
</tr>
<tr>
<td>banana</td>
<td>carrot</td>
<td>2.0</td>
</tr>
<tr>
<td>apple</td>
<td>carrot</td>
<td>4.0</td>
</tr>
<tr>
<td>banana</td>
<td>potato</td>
<td>3.5</td>
</tr>
</tbody>
</table>

We can do this in three steps:
1. Relax discrete data to continuous representation. *(Assume data is pre-encoded)*
2. Create masks corresponding to each group. *(Needs to be differentiable)*
3. Perform aggregation using the mask and data. *(Needs to be differentiable)*
Differentiable Grouped Aggregation

Step 1: Relax discrete data to continuous representation.

We can use One Hot Encoding (OHE) for categorical columns.

We assume data is pre-encoded to this format before being fed into our differentiable operator.
### Differentiable Grouped Aggregation

Step 2: Create masks corresponding to each group.

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Inventory

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>banana</td>
<td>potato</td>
</tr>
</tbody>
</table>

Mask for Group: (apple, carrot)

With the OHE strategy of categorical data representation, creating a group mask requires only element-wise product (which is differentiable).
Differentiable Grouped Aggregation

Step 3: Perform aggregation using the mask and data.

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0.</td>
<td>1.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.0</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>1.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Inventory

Mask for Group: (apple, carrot)

Aggregation

\[ \sum \]

\[ 2.0 \]
Differentiable Grouped Aggregation (GROUP BY + COUNT)

Step 3: Perform aggregation using the mask and data.

<table>
<thead>
<tr>
<th>Inventory</th>
<th>Mask for Group: (apple, carrot)</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit</td>
<td>Vegetable</td>
<td>Price</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>1.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.0</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>4.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>3.5</td>
</tr>
</tbody>
</table>

We have only used product and sum, both of which are differentiable.
Differentiable Grouped Aggregation (GROUP BY + SUM)

Step 3: Perform aggregation using the mask and data.

<table>
<thead>
<tr>
<th>Inventory</th>
<th>Summands for Group: (apple, carrot)</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit</td>
<td>Vegetable</td>
<td>Price</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>1.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.0</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>1.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Query Answer

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>SUM(Price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>carrot</td>
<td>8.0</td>
</tr>
<tr>
<td>apple</td>
<td>potato</td>
<td>0.0</td>
</tr>
<tr>
<td>banana</td>
<td>carrot</td>
<td>2.0</td>
</tr>
<tr>
<td>banana</td>
<td>potato</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Query

SELECT Fruit, Vegetable, SUM(Price)
FROM Inventory
GROUP BY Fruit, Vegetable
Differentiable Grouped Aggregation (GROUP BY + MAX)

Step 3: Perform aggregation using the mask and data.

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0.</td>
<td>4.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>2.0</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>4.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Inventory

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>banana</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>carrot</td>
<td>3.93</td>
</tr>
<tr>
<td>carrot</td>
<td>potato</td>
<td>2.0</td>
</tr>
<tr>
<td>potato</td>
<td></td>
<td>3.5</td>
</tr>
</tbody>
</table>

Aggregation

Query Answer

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>SUM(Price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>carrot</td>
<td>3.93</td>
</tr>
<tr>
<td>banana</td>
<td>carrot</td>
<td>2.0</td>
</tr>
<tr>
<td>banana</td>
<td>potato</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Query

```
SELECT Fruit, Vegetable, MAX(Price)
FROM Inventory
GROUP BY Fruit, Vegetable
```
Differentiable Filtered Aggregation (WHERE + SUM)

**Inventory**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Vegetable</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0.</td>
<td>4.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>2.0</td>
</tr>
<tr>
<td>1.</td>
<td>0.</td>
<td>4.0</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
<td>3.5</td>
</tr>
</tbody>
</table>

apple  banana  carrot  potato

**Query**

```
SELECT SUM(Price)
FROM Inventory
WHERE Price > 2.5
```

**Aggregation**

$\sum_{i} \text{Sigmoid}(h^*(x - 2.5)) = 11.51$
Case Study: Multimodal Email Search

MAIDAP has been working with MSAI to explore multimodal search capabilities for outlook.

An example of relevant data analysis:

What is the count of the different types of image attachments in outlook emails?

Regular Images
Receipts
Company Logos

Surakav’s multimodal support makes it easy to answer such queries.
Tensors are the de facto data structure for multimodal computation

The tensor data structure has been used to represent numerous rich entities.

Surakav can exploit tensors for multimodal query support.