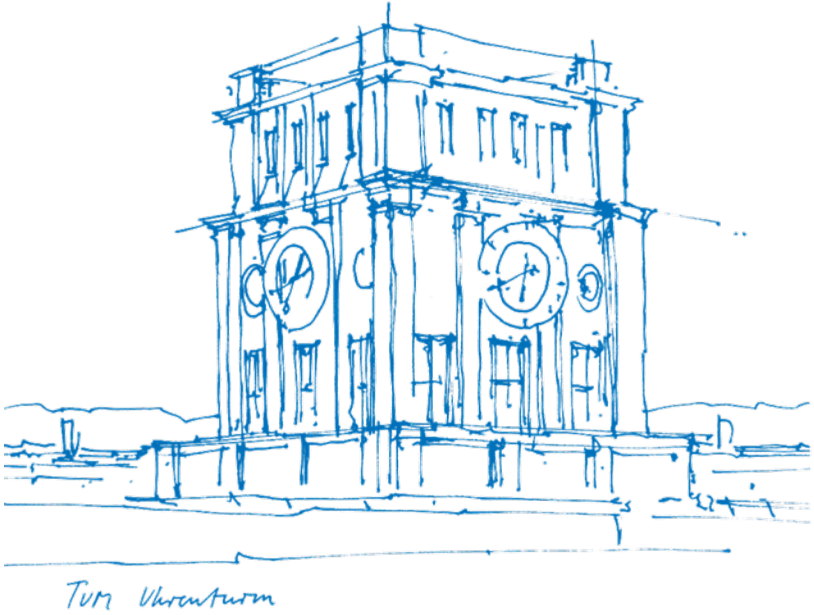
Chair III: Database Systems
Chair XXV: Data Science and Engineering
Department of Informatics
Technical University of Munich

In-Database Machine Learning with SQL on GPUs

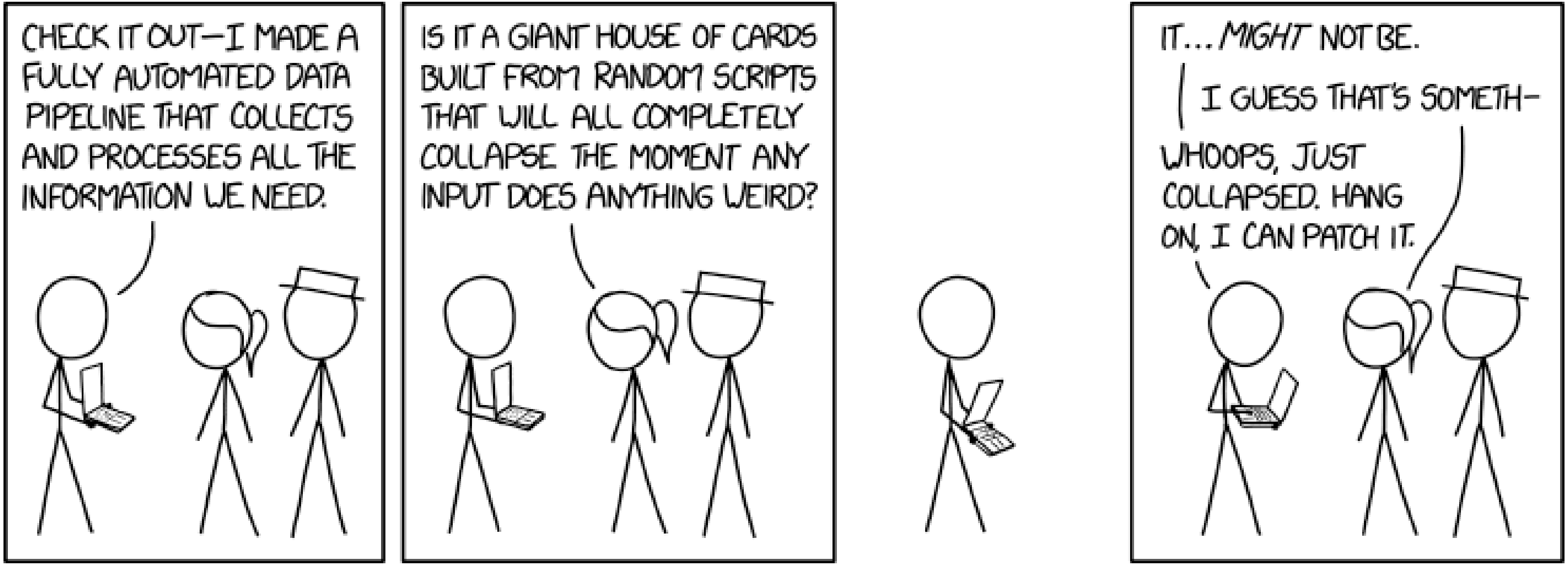
Maximilian E. Schüle, Harald Lang, Maximilian Springer,
Alfons Kemper, Thomas Neumann, Stephan Günnemann

Tampa, Florida, USA, July 6-7, 2021



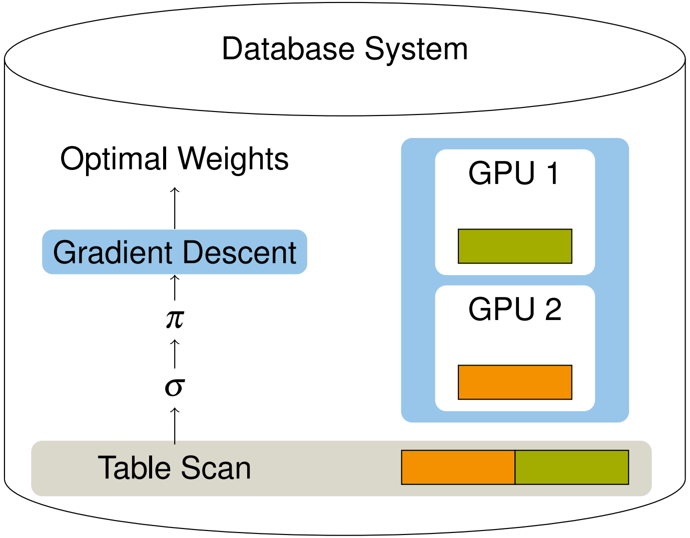
In-Database Machine Learning: Problem

xkcd.org #2054 CC BY-NC 2.5



In-Database Machine Learning: Solution

In-Database Machine Learning



- SQL sufficient for machine learning (ML)
  - Turing-complete with recursive tables
  - Streams for continuous learning
  - Sample operator for stochastic gradient descent
- Idea
  - Data preprocessing using SQL
  - No need for data extraction out of a database system
  - Continuously train models using operators for gradient descent with GPU support
  - Label data within the database system using SQL

Structure

ML in SQL-92
Gradient descent with recursive tables
Machine learning pipeline in SQL

UMBRA

ML Operators
Automatic Differentiation
Gradient Descent Operator

GPU support
GPU co-processing
Evaluation

ML in SQL-92

ML in SQL-92: Gradient Descent with Recursive SQL

A *loss function* $l_{X,y}(\vec{w})$ measures the deviation (*residual*) between all approximated values $m_{\vec{w}}(X)$ and the given labels $\vec{y}$, for example, mean squared error:

$$l_{x,y}(a,b) = (a \cdot x + b - y)^2 \tag{1}$$

$$\nabla l_{x,y}(a,b) = \begin{pmatrix} \partial l / \partial a \\ \partial l / \partial b \end{pmatrix} = \begin{pmatrix} 2(ax+b-y) \cdot x \\ 2(ax+b-y) \end{pmatrix}. \tag{2}$$

To minimise $l_{X,y}(\vec{w})$, gradient descent updates the weights per iteration by subtracting the loss function's gradient times the learning rate γ.

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \nabla l_{X,\vec{y}}(\vec{w}_t), \tag{3}$$

$$\vec{w}_\infty \approx \lim_{t \to \infty} \vec{w}_t. \tag{4}$$

```sql
create table data (x float, y float);
insert into data ...

with recursive gd (id, a, b) as (
 select 0,1::float,1::float
UNION ALL
 select id+1,
   a-0.05*avg(2*x*(a*x+b-y)),
   b-0.05*avg(2*(a*x+b-y))
 from gd, (select * from data )
 where id<5 group by id,a,b)
select * from gd order by id;
```

Listing 1: Gradient descent (batch).

Five iterations, loss function with two weights (8). First, the weights get initialised, then each iteration updates the weights (3) based on manually derived gradients (2) and $\gamma = 0.05$.

ML in SQL-92: Gradient Descent with Recursive SQL

A *loss function* $l_{X,y}(\vec{w})$ measures the deviation (*residual*) between all approximated values $m_{\vec{w}}(X)$ and the given labels $\vec{y}$, for example, mean squared error:

$$l_{x,y}(a,b) = (a \cdot x + b - y)^2 \tag{1}$$

$$\nabla l_{x,y}(a,b) = \begin{pmatrix} \partial l/\partial a \\ \partial l/\partial b \end{pmatrix} = \begin{pmatrix} 2(ax+b-y)\cdot x \\ 2(ax+b-y) \end{pmatrix}. \tag{2}$$

To minimise $l_{X,y}(\vec{w})$, gradient descent updates the weights per iteration by subtracting the loss function's gradient times the learning rate γ.

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \nabla l_{X,\vec{y}}(\vec{w}_t), \tag{3}$$

$$\vec{w}_\infty \approx \lim_{t\to\infty} \vec{w}_t. \tag{4}$$

```
create table data (x float, y float);
insert into data ...

with recursive gd (id, a, b) as (
 select 0,1::float,1::float
UNION ALL
 select id+1,
   a-0.05*avg(2*x*(a*x+b-y)),
   b-0.05*avg(2*(a*x+b-y))
 from gd, (select * from data tablesample reservoir(1))
 where id<5 group by id,a,b)
select * from gd order by id;
```

Listing 2: Gradient descent (stochastic).

Five iterations, loss function with two weights (8). First, the weights get initialised, then each iteration updates the weights (3) based on manually derived gradients (2) and $\gamma = 0.05$.

ML in SQL-92: Gradient Descent with Recursive SQL

A *loss function* $l_{X,y}(\vec{w})$ measures the deviation (*residual*) between all approximated values $m_{\vec{w}}(X)$ and the given labels $\vec{y}$, for example, mean squared error:

$$l_{x,y}(a,b) = (a \cdot x + b - y)^2 \tag{1}$$

$$\nabla l_{x,y}(a,b) = \begin{pmatrix} \partial l/\partial a \\ \partial l/\partial b \end{pmatrix} = \begin{pmatrix} 2(ax+b-y)\cdot x \\ 2(ax+b-y) \end{pmatrix}. \tag{2}$$

To minimise $l_{X,y}(\vec{w})$, gradient descent updates the weights per iteration by subtracting the loss function's gradient times the learning rate γ.

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \nabla l_{X,\vec{y}}(\vec{w}_t), \tag{3}$$

$$\vec{w}_\infty \approx \lim_{t\to\infty} \vec{w}_t. \tag{4}$$

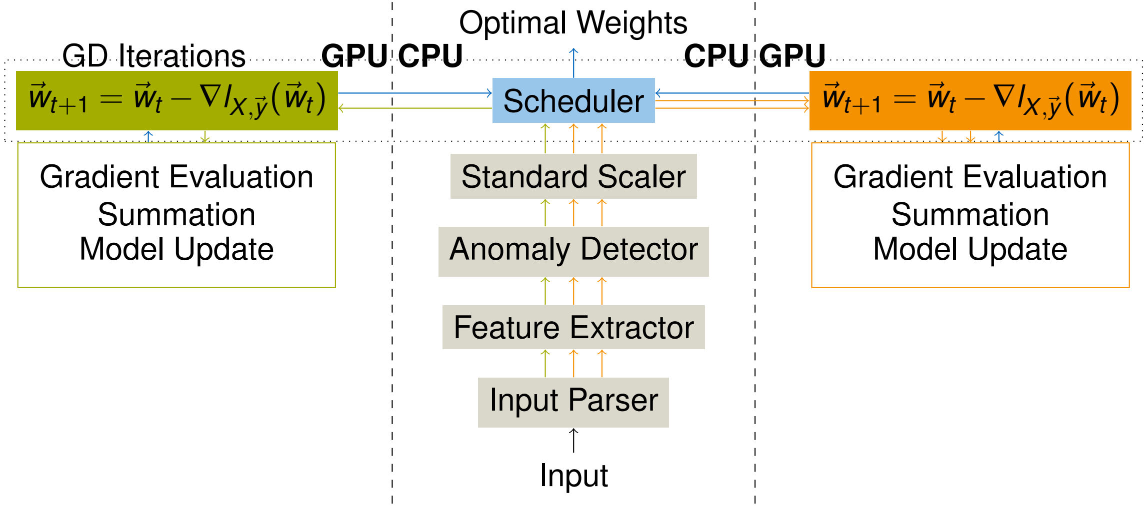
```sql
create table data (x float, y float);
insert into data ...

with recursive gd (id, a, b) as (
 select 0,1::float,1::float
UNION ALL
 select id+1,
   a-0.05*avg(2*x*(a*x+b-y)),
   b-0.05*avg(2*(a*x+b-y))
 from gd, (select * from data tablesample reservoir(8))
 where id<5 group by id,a,b)
select * from gd order by id;
```

Listing 3: Gradient descent (mini-batch).

Five iterations, loss function with two weights (8). First, the weights get initialised, then each iteration updates the weights (3) based on manually derived gradients (2) and $\gamma = 0.05$.

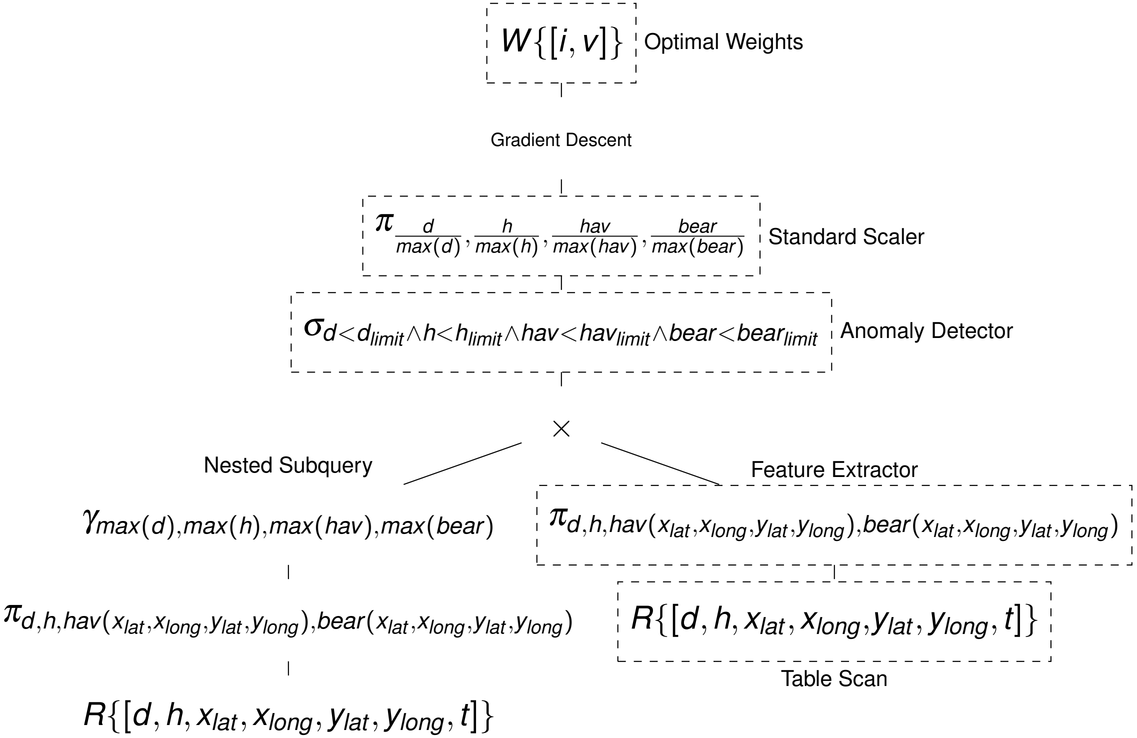
ML in SQL-92: Components of a Machine Learning Pipeline



Machine learning pipeline proposed by Derakhshan et. al. (EDBT'19)

- **Scheduler** manages gradient descent iterations until the weights converge.
- **Standard Scaler** scales all attributes in the range $[0, 1]$ to equal each attribute's impact on the model.
- **Anomaly Detector** deletes tuples on anomalies. An anomaly occurs when at least one attribute in a tuple passes over or under a predefined threshold.
- **Feature Extractor**: extracts features from data chunks.
- **Input Parser**: parses input CSV files and stores the data in chunks.

ML in SQL-92: Machine Learning Pipeline in Relational Algebra



- **Standard Scaler**: projection and nested subqueries to extract the attribute's extrema (view `normalised`).
- **Anomaly Detector**: user-defined limits in a selection (view `normalised`).
- **Feature Extractor**: a simple projection in SQL, day and hour from timestamps, distance metrics from given coordinates (view `processed`).
- **Input Parser**: a simple table scan or a foreign table as input for continuous views (table `taxidata`).

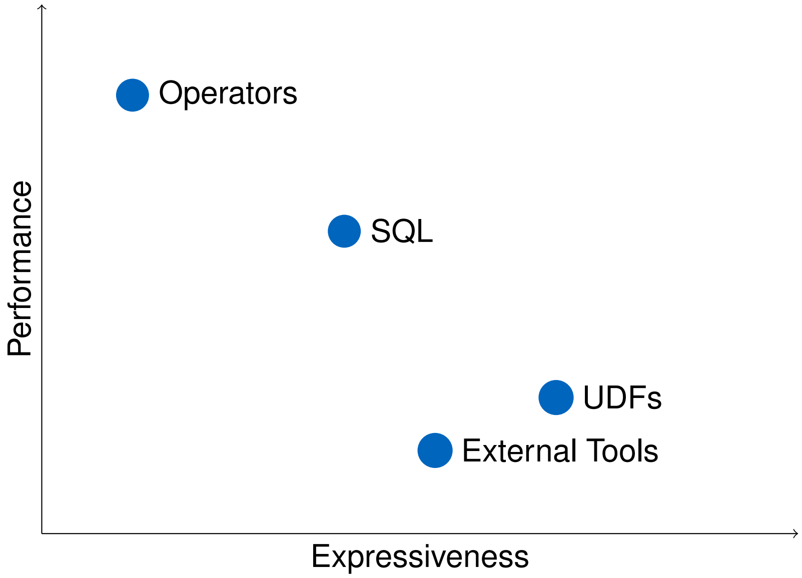
ML in SQL-92: Machine Learning Pipeline in SQL

```
create foreign table taxidata(id int, pickup_datetime date, dropoff_datetime date,
  passengers float, pickup_longitude float, pickup_latitude float, dropoff_longitude float,
  dropoff_latitude float, duration float) server stream;
copy taxidata from './taxidata.csv' delimiter ',';
create view processed as (select hour,day,duration,ACOS(SIN(plat)*SIN(dlat)+COS(plat)*COS(dlat)[...]
create view normalised(hour, day, distance, bearing, duration) as (
  select cast(hour as float)/(select max(hour)+1 from processed), [...]
  from processed where distance < 1000);

with recursive gd (id, a1, a2, a3, a4, b) as ( select 0, 1::float, 1::float, 1::float, 1::float, 1::float
  UNION ALL select id+1,
    a1-0.001*avg(2*hour*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
    a2-0.001*avg(2*day*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
    a3-0.001*avg(2*distance*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
    a4-0.001*avg(2*bearing*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
    b -0.001*avg(2*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration))
  from gd, (select * from normalised tablesample reservoir (10)) where id<50 group by id,a1,a2,a3,a4, b)
select id, avg(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)^2
  from gd,normalised where id=50;
```

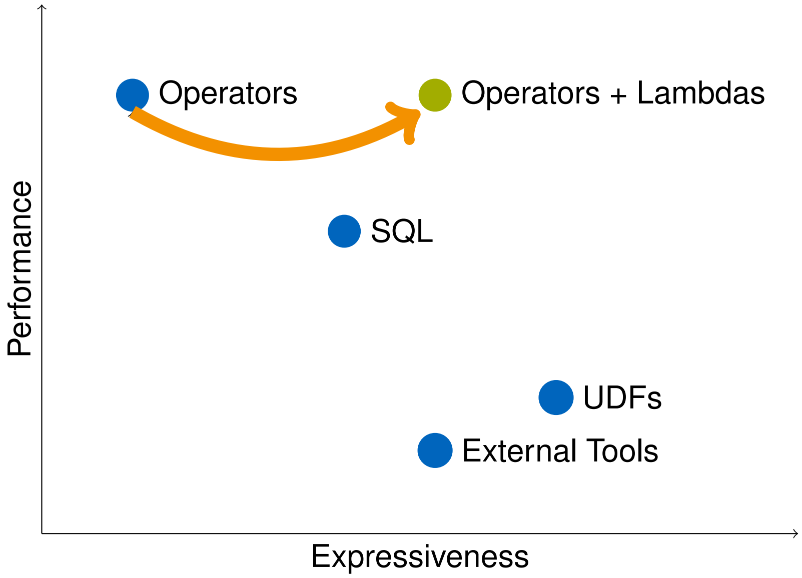
ML Operators

ML Operators: Why Lambda Functions in SQL?



- SQL
  - Turing-complete with recursive tables
  - queries get optimised before execution
  - statements must be expressed in relational algebra
- Operators (Table Functions)
  - purpose-specific but high-performant
  - require development by a database engineer
- User-Defined Functions (UDFs)
  - allow procedural language statements in SQL
  - not as performant as operators
- External Tools
  - database system as storage layer only
  - time consuming extraction necessary

ML Operators: Why Lambda Functions in SQL?



- SQL
  - Turing-complete with recursive tables
  - queries get optimised before execution
  - statements must be expressed in relational algebra
- Operators (Table Functions)
  - purpose-specific but high-performant
  - require development by a database engineer
- User-Defined Functions (UDFs)
  - allow procedural language statements in SQL
  - not as performant as operators
- External Tools
  - database system as storage layer only
  - time consuming extraction necessary
- Operators + Lambdas
  - customisation of operators

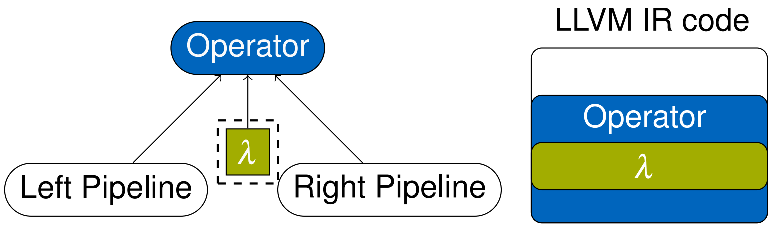
ML Operators: Lambda Functions in HyPer and Umbra

- HyPer and Umbra: code-generating database systems
- produce LLVM IR (Intermediate Representation)
- Lambda expressions: inject code into regular operators
- composed of *lambda arguments* to identify tuples and
- a *lambda body* to formulate an expression

$$\lambda(name_1, name_2, \ldots)(expr) \quad (5)$$

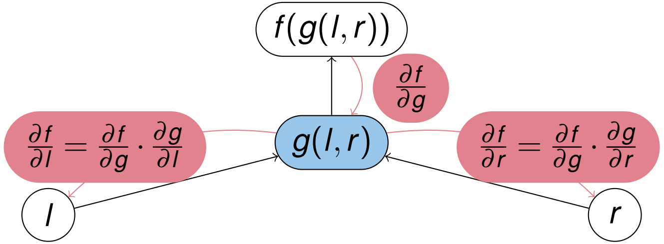
- Example: k-Means with injected distance metric

$$\lambda(S,T)((S.x - T.x)^2 + (S.y - T.y)^2) \quad (6)$$



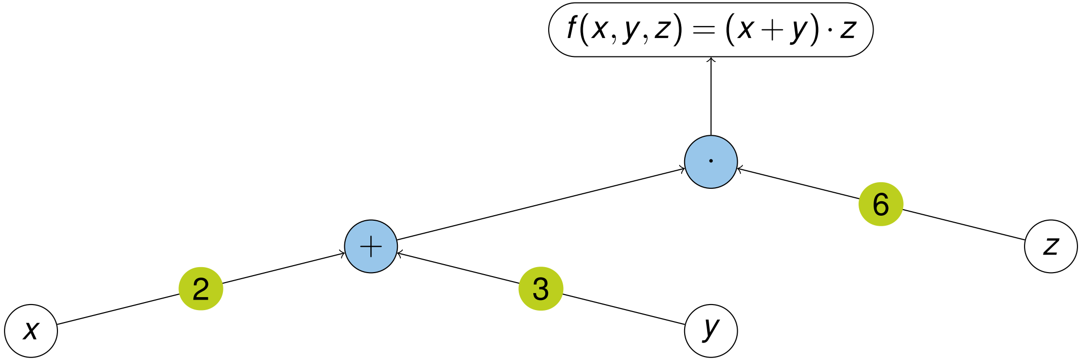

```
CREATE TABLE data(x float, y int);
CREATE TABLE centre(x float, y int);
INSERT INTO ...
SELECT * FROM kmeans(
  (SELECT x,y FROM data),
  (SELECT x,y FROM centre),
  -- distance function and max. number of iterations
  λ(a,b) (a.x-b.x)^2+(a.y-b.y)^2, 3);
```

ML Operators: Automatic Differentiation as Operator



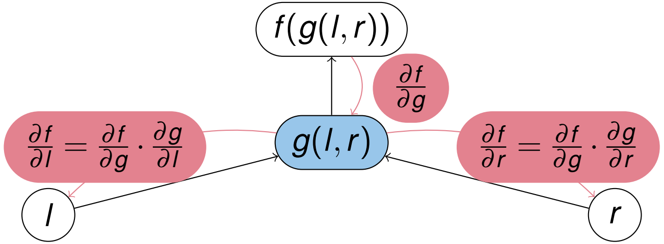
Automatic differentiation using backward mode

- applying the chain rule to backpropagate the loss
- no need for manually derived gradients
- subexpressions are cached in LLVM registers for reuse
- expose as SQL operator



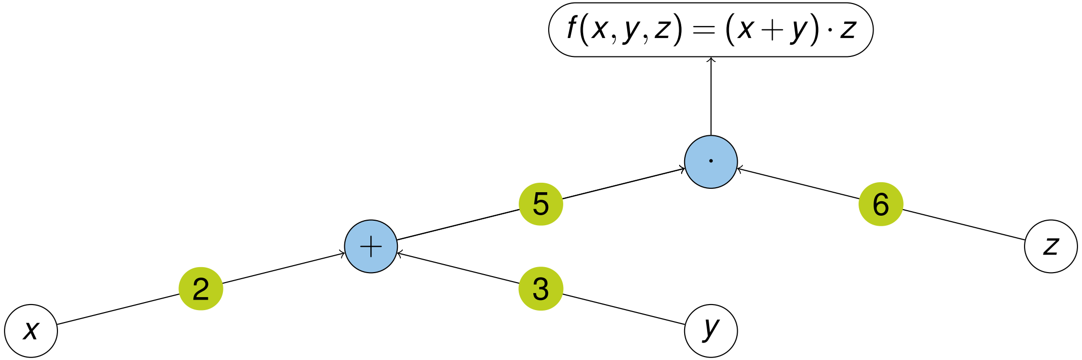

```
select * from umbra.derivation(
  TABLE(select 2 x,3 y,6 z),
  lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
```

ML Operators: Automatic Differentiation as Operator



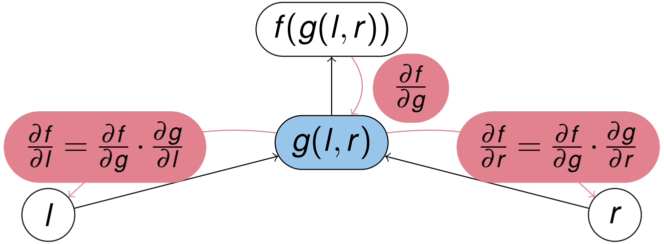
Automatic differentiation using backward mode

- applying the chain rule to backpropagate the loss
- no need for manually derived gradients
- subexpressions are cached in LLVM registers for reuse
- expose as SQL operator



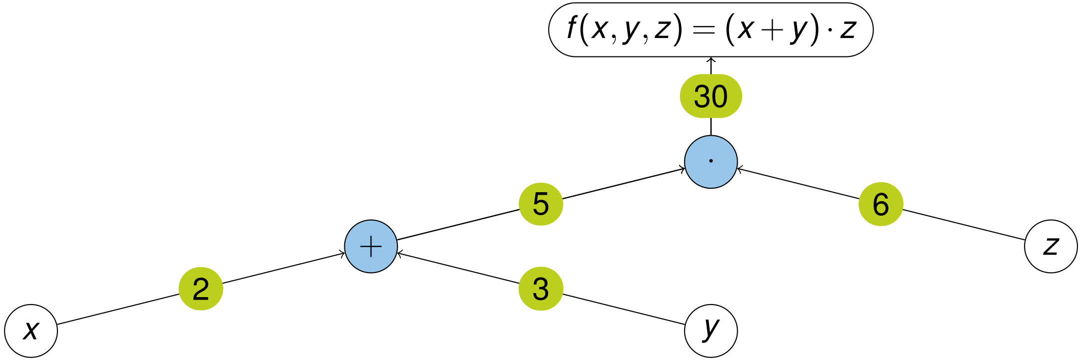

```
select * from umbra.derivation(
  TABLE(select 2 x,3 y,6 z),
  lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
```

ML Operators: Automatic Differentiation as Operator



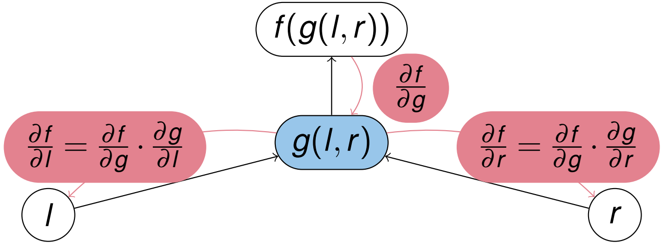
Automatic differentiation using backward mode

- applying the chain rule to backpropagate the loss
- no need for manually derived gradients
- subexpressions are cached in LLVM registers for reuse
- expose as SQL operator



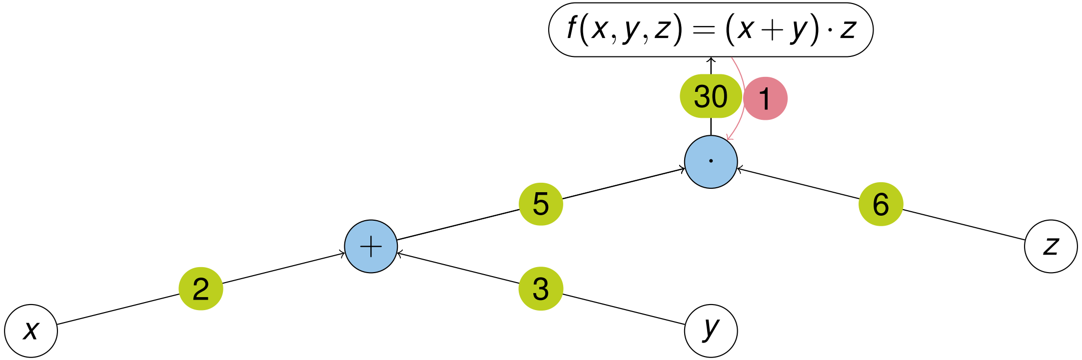

```
select * from umbra.derivation(
  TABLE(select 2 x,3 y,6 z),
  lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
```

ML Operators: Automatic Differentiation as Operator



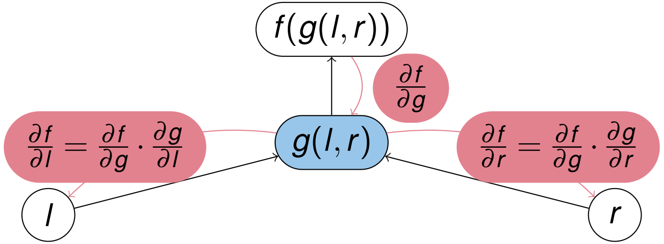
Automatic differentiation using backward mode

- applying the chain rule to backpropagate the loss
- no need for manually derived gradients
- subexpressions are cached in LLVM registers for reuse
- expose as SQL operator



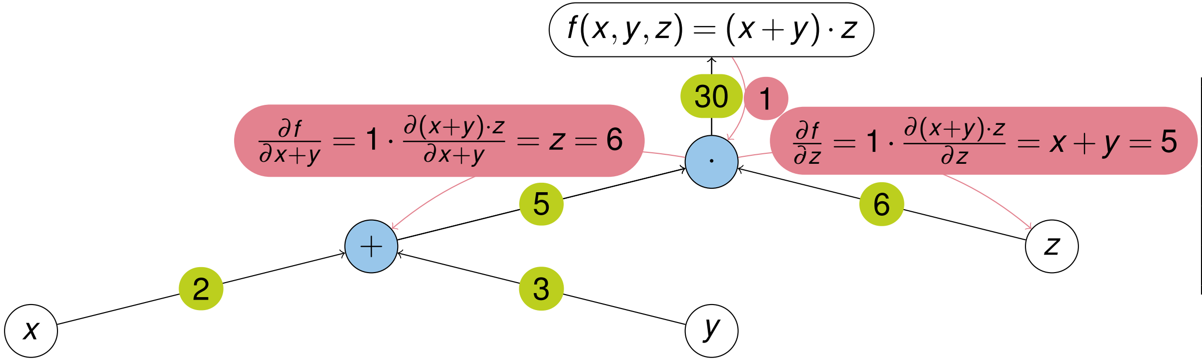

```
select * from umbra.derivation(
  TABLE(select 2 x,3 y,6 z),
  lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
```

ML Operators: Automatic Differentiation as Operator



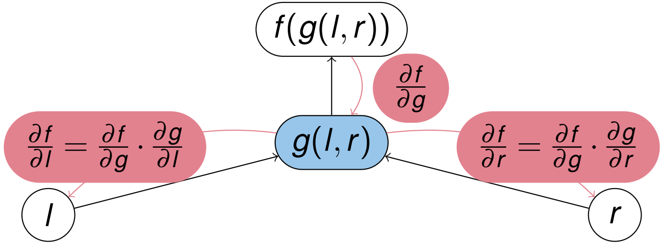
Automatic differentiation using backward mode

- applying the chain rule to backpropagate the loss
- no need for manually derived gradients
- subexpressions are cached in LLVM registers for reuse
- expose as SQL operator



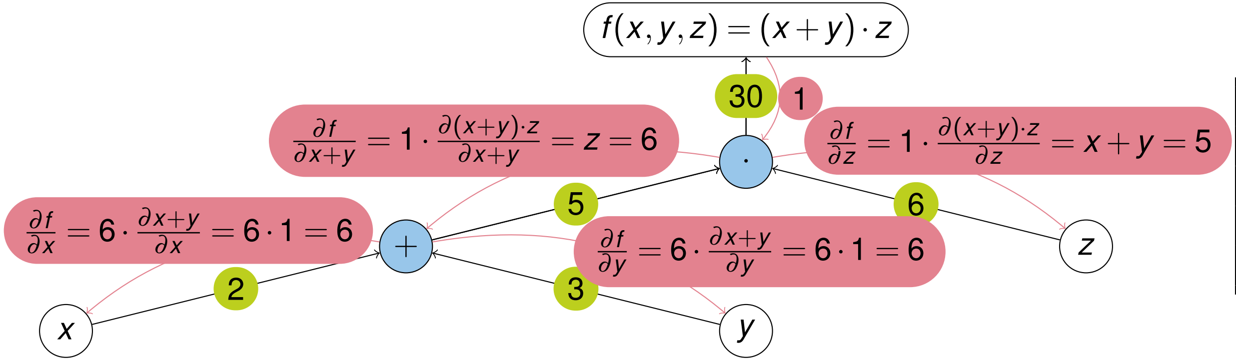

```
select * from umbra.derivation(
  TABLE(select 2 x,3 y,6 z),
  lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
```

ML Operators: Automatic Differentiation as Operator



Automatic differentiation using backward mode

- applying the chain rule to backpropagate the loss
- no need for manually derived gradients
- subexpressions are cached in LLVM registers for reuse
- expose as SQL operator




```
select * from umbra.derivation(
  TABLE(select 2 x,3 y,6 z),
  lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
```

ML Operators: Automatic Differentiation for Gradient Descent

Manually Derived

```
create table data (x float, y float);
insert into data ...

with recursive gd (id, a, b) as (
  select 1,1::float,1::float
UNION ALL
  select id+1,
    a-0.05*avg(2*x*(a*x+b-y)),
    b-0.05*avg(2*(a*x+b-y))
  from gd, data where id<5
  group by id,a,b)
select * from gd order by id;
```

Automatically Derived

```
create table data (x float, y float);
insert into data ...

with recursive gd (id, a, b) as (
  select 1,1::float,1::float
UNION ALL
  select id+1, a-0.05*avg(d_a), b-0.05*avg(d_b)
  from umbra.derivation(TABLE (
    select id,a,b,x,y from gd,data where id<5),
    lambda (x) ((x.a * x.x + x.b - x.y)^2))
  group by id,a,b)
select * from gd order by id;
```

ML Operators: Training a Feed-Forward Neural Network

Fully connected neural network with one hidden layer of size h, L output vector of probabilites, two weight matrices $w_{xh} \in \mathbb{R}^{|\vec{x}| \times h}$ and $w_{ho} \in \mathbb{R}^{h \times |L|}$, an activation function (applied elementwise), model function $m_{w_{xh}, w_{ho}}(\vec{x}) \in \mathbb{R}^{|L|}$, forward pass and loss:

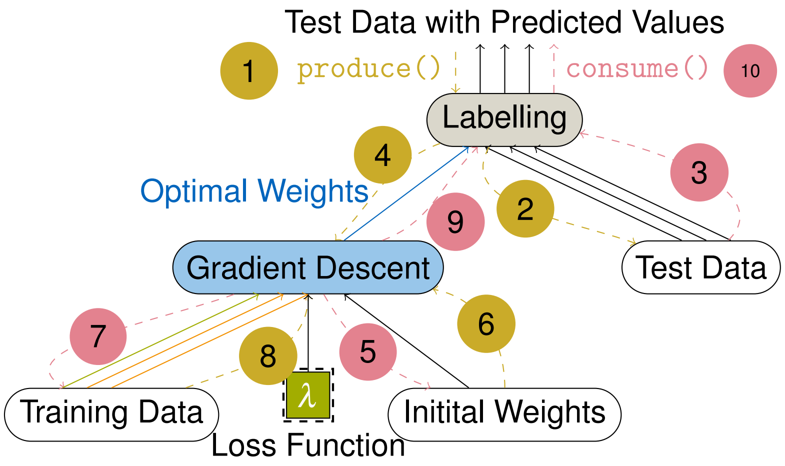
$$m_{w_{xh}, w_{ho}}(\vec{x}) = sig(sig(\vec{x}^T \cdot w_{xh}) \cdot w_{ho}), \tag{7}$$

$$l_{w_{xh}, w_{ho}}(\vec{x}, \vec{y}) = (m_{w_{xh}, w_{ho}}(\vec{x}) - \vec{y})^2. \tag{8}$$

```sql
with recursive gd (id,w_xh,w_ho) as (
   select 0, array_fill(0.1::float,array[4,10]), array_fill(0.1::float,array[10,3])
union all
   select id+1, w_xh - 0.1 * avg(d_w_xh), w_ho - 0.1 * avg(d_w_ho)
   from umbra.derivation(TABLE(
     select * from data,gd where id < 10),
     lambda(x)(( sig(sig(x.img*x.w_xh)*x.w_ho) - one_hot)^2 ))
   group by id, w_ho, w_xh)
select * from gd order by id;
```

Listing 4: Training a neural network when applying matrix algebra on arrays.

ML Operators: Gradient Descent as Operator



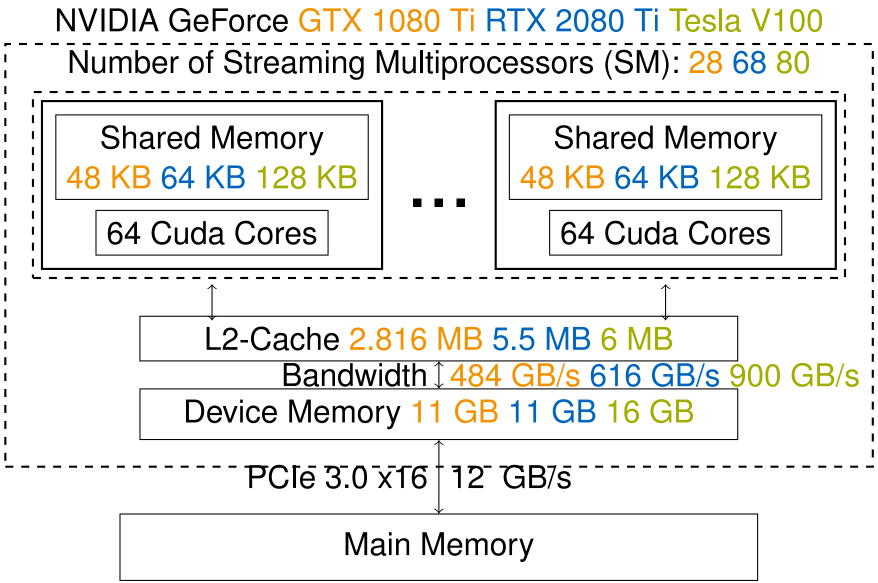
Dedicated operator for gradient descent

- **Input**: training data, initial weights and the loss function
- **Output**: optimal weights
- allows to call specialised libraries and off-loading to GPU

```
select * from umbra.gd(
   TABLE (select * from data), TABLE (select 10::float a, 10::float b),
   lambda (x,y) ((y.a * x.x + y.b - x.y)^2), 1, 0.05, 10);
```

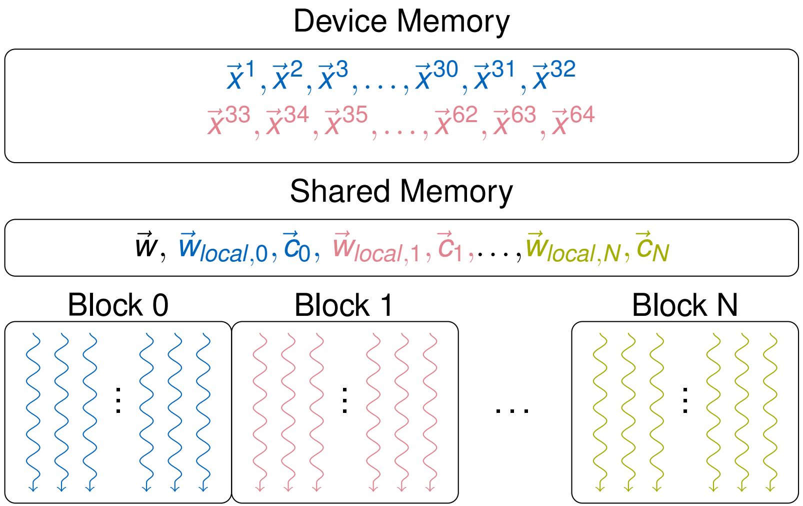
GPU Co-Processing

GPU Co-Processing: GPU Architecture



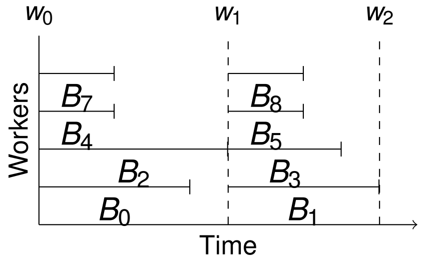
- Each GPU device owns one global memory (device memory) and an L2 cache.
- Core components: streaming multiprocessors with an attached shared memory
- Parallel threads perform the same instructions simultaneously
- 32 threads in a bundle: *warp*, multiple warps: *block*
- **Challenge**: map mini-batches of data to blocks
- **Parameter**: number of warps per block

GPU Co-Processing: Multiple Learner per GPU

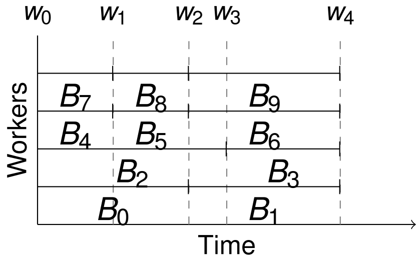


- **Goal**: utilise all GPU threads even with small batch sizes
- **Solution**: multiple independent learners per GPU
- Each block = one learner, responsible for a mini-batch
- Each learner maintains local weights $\vec{w}_{local}$ and the difference $\vec{c}_{local}$ to the global weights $\vec{w}$.
- Minimum batch size: one warp (minimum block size) with 32 threads
- Maximum number of learners = number of possible warps

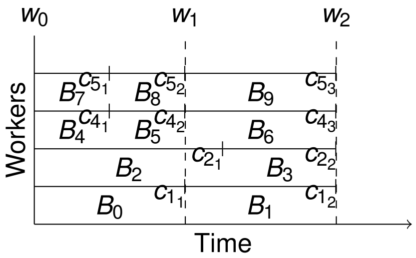
GPU Co-Processing: Synchronisation



Synchronised threads



Worker threads (global updates)



Worker threads (local models)

- **Synchronised threads**: same weights with an individual mini-batch, the main worker collects the calculated gradients and takes their average to update the weights, workers might drive idle and waiting for input
- **Worker threads (global updates)**: independent workers have to fetch their mini-batches on their own, global atomic counter as a batch identifier. Weights are updated globally. Assuming a low learning rate, weights are changing marginally and locks can be omitted similar to HogWild.
- **Worker threads (local models)**: local models known from Crossbow: Each learner maintains local weights. For every learner t a vector called corrections $\vec{c}_t$ stores the differences to the global weights. After each iteration, the corrections of all learners are summed up to form the global weights.

Evaluation

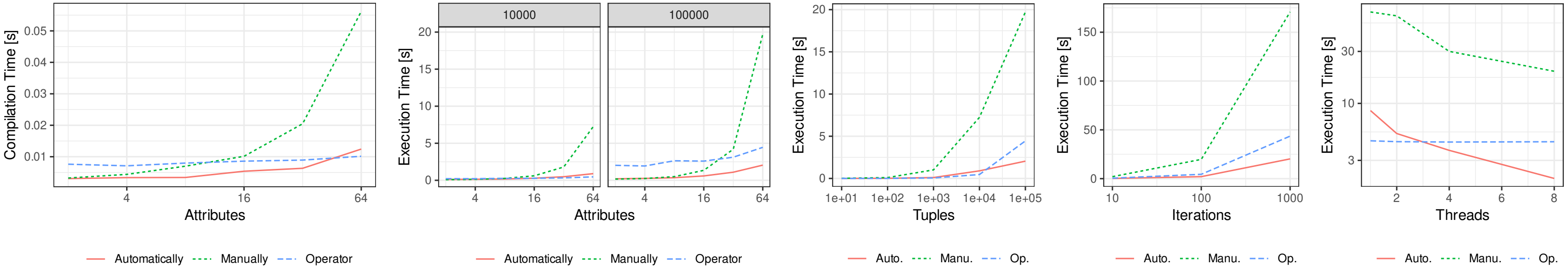
Evaluation: Set-Up

- **System**: Intel Xeon Gold 5120 processors, 4x14 CPUs (2.20 GHz), Ubuntu 20.04.01 LTS , 256 GiB RAM.
- **GPU**: either four GPUs (NVIDIA GeForce GTX 1080 Ti/RTX 2080 Ti) or one NVIDIA Tesla V100.
- **Models**: linear regression and feed-forward neural network with a single hidden layer for image recognition.
- **Data**: synthetic data, New York taxi data set (January 2015, 2.65 GiB), (Fashion-)MNIST data set

	#attr.	*#training*	*#validation*
New York Taxi	4 + 1	61,664,460	15,416,115
Synthetic	99 + 1	10	10
MNIST	784 + 1	60,000	10,000
Fashion-MNIST	784 + 1	60,000	10,000

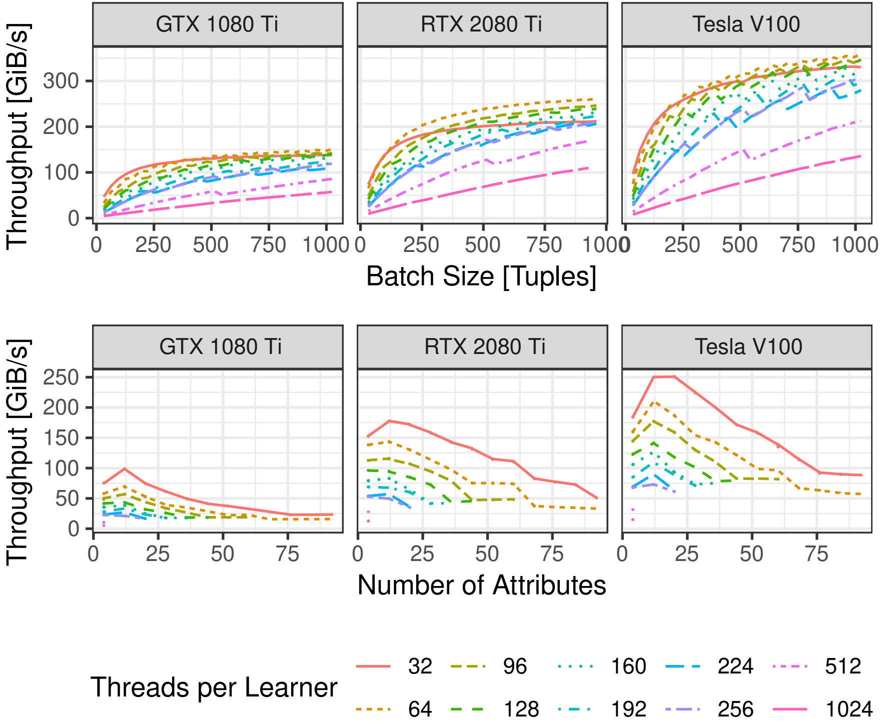
Table: Training and validation data sets used with linear regression and a neural network respectively.

Evaluation: Automatically vs. Manually Derived



- batch gradient descent (the batch size corresponds to the number of tuples), linear model, synthetic data
- recursive tables with either manually or automatically derived gradients, and a dedicated (single-threaded) operator
- automatic differentiation: speeds up compilation time and execution time (subexpressions are cached in registers for reuse)
- also visible when the batch size, the number of iterations or the number of threads is varied
- parallelisation when using recursive tables

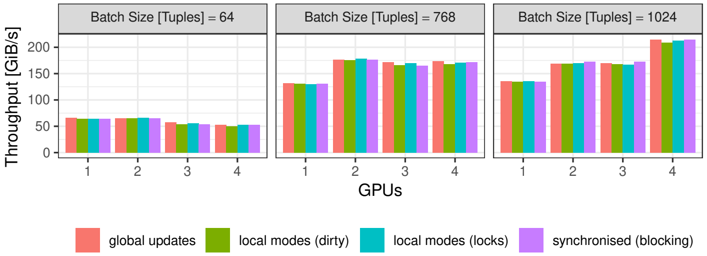
Evaluation: GPU co-processing (Learners, Linear Regression)



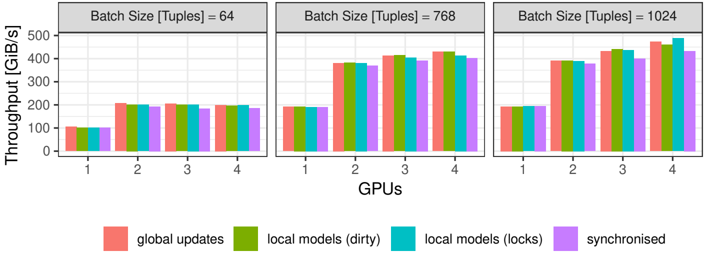
- vary the number of threads per block (32 to 1,024 threads, 4 attributes) or number of attributes (32 threads per block)
- a small number of threads per learner: a higher throughput for small batch sizes.
- highest throughput when batch size is a multiple of the block size
- local maximum (spikes): batch size = multiple of a block size

Evaluation: GPU co-processing (Linear Regression)

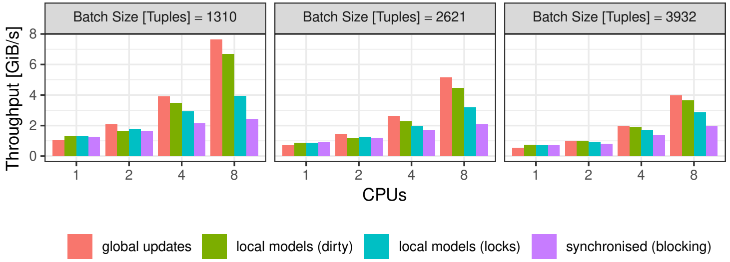
NVIDIA GeForce GTX 1080 Ti



NVIDIA GeForce RTX 2080 Ti



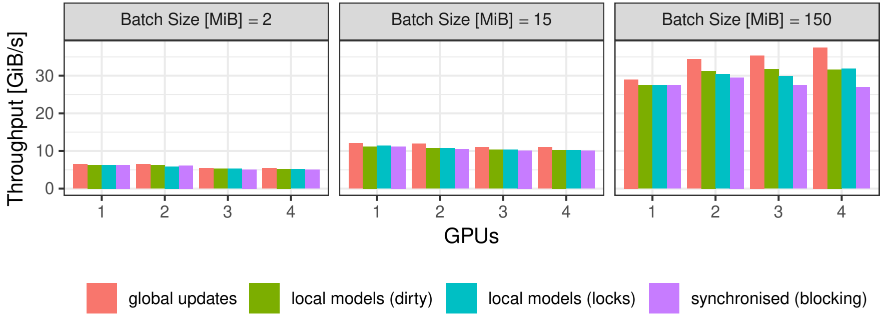
CPU (Intel Xeon Gold 5120)



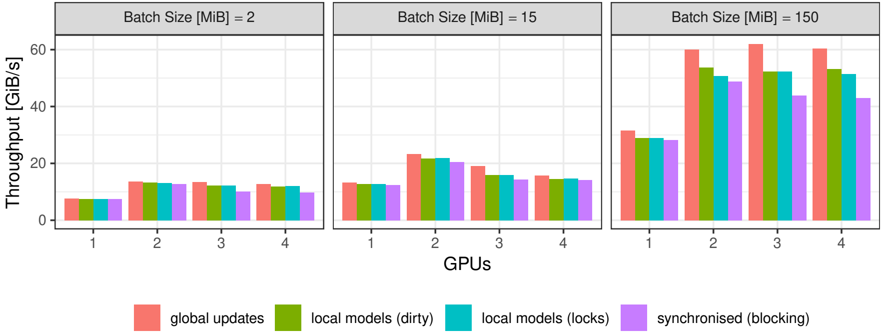
- no synchronisation, global updates (*global updates*), local models with locking of the critical section (*local models (locks)*) or without locking (*local models (dirty)*), (*synchronised (blocking)*).
- CPU: linear speed-up when no synchronisation takes place
- locks: lower throughput, blocking threads cause underutilisation
- GPU: the larger the batch size (less synchronisation), the higher the scale-up as (parallel workers work independently)
- local models: inter-GPU communication decreases the performance with the third additional device

Evaluation: GPU co-processing (Neural Network)

NVIDIA GeForce GTX 1080 Ti

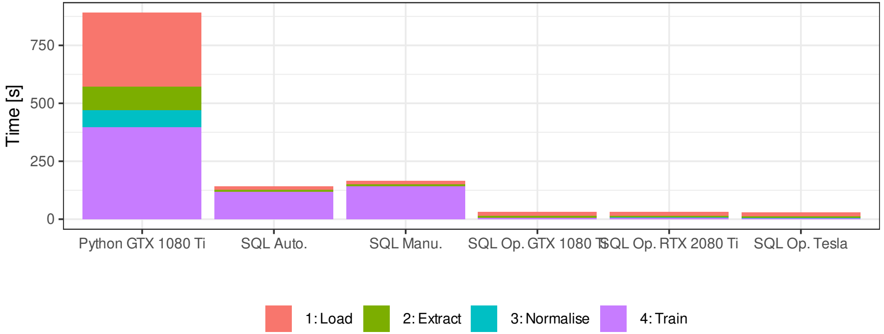


NVIDIA GeForce RTX 2080 Ti



- one additional worker increases the throughput
- for any further workers, the inter-GPU communication decreases the runtime
- small batch sizes: best result on two GPU devices
- larger batch sizes: every additional device allows a higher throughput

Evaluation: End-to-End



- training of one epoch (New York taxi data: $13 \cdot 10^6$ tuples)
- ML pipeline in Python using Keras vs. SQL within Umbra
- Steps: data loading from CSV, feature extraction and normalisation either with NumPy or SQL-92 queries, and training
- much time spent on data loading from CSV and preprocessing (no longer required within a database system or highly parallelised)
- gradient descent using recursive tables: comparable performance to library functions
- all outperformed by our operator that off-loads training to GPU

Conclusion

- in-database machine learning pipeline expressed in pure SQL based on sampling, continuous views and recursive tables
- operator for automatic differentiation and one for gradient descent
- off-load training to GPU units
- training algorithms as GPU kernels and fine-tuned learners at hardware level to increase the learning throughput
- automatic differentiation accelerated both the compile time and the execution time by the number of cached expressions
- fine-tuned learners at hardware level: highest possible throughput for small batch sizes
- end-to-end machine learning pipeline in SQL: comparable performance to traditional machine learning frameworks

Thank you for your attention!