Differentiating Relational Queries

Paul Peseux
supervised by M.Berar, T.Paquet & V.Nicollet
Litis Normandie & Lokad
Paris, France
paul.peseux@lokad.com

ABSTRACT

This work is about performing automatic differentiation of a query in the context of relational databases and queries. This is done in order to perform optimization through gradient descent in these relational databases. This work describes a form of automatic differentiation for a subset of relational queries.

1 INTRODUCTION

Modern Differentiable Programming applied to Deep Learning concentrates on dense and regular problems such as images [13] [12] and sound [6], or studies ways to project unstructured problems into this framework [11] (e.g. auto-encoders for text data). Its success is partly due to automatic differentiation [21]. Parallel to this many business domains have a very well-defined structure, but this structure is relational. For example supply chain data is organized in relational databases and experts are used to working with these. A canonical example coming from this domain: items in the product table come from suppliers in the supplier table and are stored in warehouses in the warehouse table; the problem's structure is completely different from Computer Vision or Natural Language Processing, two hot topics in Machine Learning (ML). As people that understand the supply chain complexity work with relational databases, it seems to be the adequate place to let them build their own models and optimize them. One of the main ways to optimize a model is through gradient-based methods; if the model is written with queries then we need to differentiate them to optimize the model. Letting experts build white box models will help them to check the sanity of models, which is very difficult to do on black box models, such as deep neural networks. It is called Interpretable ML in [19] and directly applies to the supply chain, where thousands of orders are placed everyday for a single company. Furthermore [20] has shown the advantages when performing an optimization in the database system itself, limiting data transfer costs, over pulling the data out to an external ML-oriented system.

Many sub parts of languages have been differentiated: Python [17] [4], C [8], Julia [10], Swift [18], F# [3] ... A more complete reference can be found at [7]. However, there are only a few attempts at differentiating SQL·like programming languages, to our knowledge. Thus we study a way to *differentiate relational queries* in order to perform optimization through gradient descent.

Proceedings of the VLDB 2021 PhD Workshop, August 16th, 2021. Copenhagen, Denmark.

2 ADSL

[8] [10] [2] [21] [...] proposes to differentiate subsets of common programming languages. What these initiatives have in common is the purpose of differentiating a pre-existing language. It is an interesting task, while being complicated, as those languages are not crafted for differentiation. This is especially true for relational programming languages.

We introduce ADSL[1], which is A Differentiable Sub Language that is intended to lower relational language. It is a simple language where Automatic Differentiation is a first class citizen. This idea is similar to [1] [9] [14]. ADSL is closed by differentiation: the adjoint, i.e. the derived program, of an ADSL program is also a differentiable ADSL program. This closure gives immediate access to higher order derivatives, which are sometimes used [15] [5]. ADSL is a simple SSA language that supports loops and conditional. Its major specificity is its projectors and broadcasts support.

According to the definition below, an ADSL program is a list of Statements $< S >$, whose grammar is defined by:

$\langle\ S\ \rangle$	::=	.	
	|	$\langle\ v \leftarrow e \rangle$	Variable assignment
	|	$\langle\ Cond\ (\ v \quad \Psi \quad P_T \quad P_E \quad \Phi)\rangle$	Conditional
	|	$\langle\ For\ (\ \chi \quad P \quad \Xi)\rangle$	Loop
	|	$\langle\ Return\ v\ \rangle$	Output of a program
$\langle\ e\ \rangle$	::=	.	
	|	$\langle\ v\ \rangle$	Variable
	|	$\langle\ f\rangle$	Scalar
	|	$\langle\ b\ \rangle$	Boolean
	|	$\langle\ v + w\ \rangle$	Variable Addition
	|	$\langle\ Call1\ op\ v\ \rangle$	Function Call
	|	$\langle\ Call2\ op\ v\ w\ \rangle$	Function Call (2 parameters)
	|	$\langle\ Param\ i\ \rangle$	Parameter access
	|	$\langle\ Const\ i\ \rangle$	Constant access
	|	$\langle\ v \quad \triangleleft \quad \beta \quad \rangle$	Broadcast Projector
	|	$\langle\ v \quad \triangleright \quad \alpha \quad \rangle$	Aggregation Projector
	|	$\langle\ Pred\ \rangle$	Predicate
$\langle\ Pred\ \rangle$	::=	.	
	|	$\langle\ And\ v\ w\ \rangle$	
	|	$\langle\ Or\ v\ w\ \rangle$	
	|	$\langle\ Not\ v\ \rangle$	
	|	$\langle\ v < w\ \rangle$	
	|	$\langle\ v \leq w\rangle$	

ADSL is tight but it is enough to tackle many real business problems, such as those encountered in the supply chain. Its main characteristic is to be easily and fully differentiable. Projectors

[1] Adsl library can be found at https://github.com/Lokad/Adsl

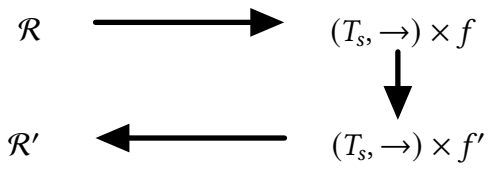

Figure 1: Path to differentiation.

(broadcasts and aggregations) are overused (with INNER JOIN and GROUP BY SQL operator) while lowering a relational query.

Remark 1. *We special case the addition. At first glance, it might seem unclear why it is not included in Call2. It is due to our automatic differentiation implementation and goes beyond the scope of the paper.*

3 DERIVED QUERY

In this section we describe our approach to derive relational query. This approach is based on *compilation*.

Let $\mathcal{R}$ be a relational query that creates the float column *Loss* in the table *OT*. We assume that $\mathcal{R}$ involves a float column X in the table *PT*. In optimization or Machine Learning, such an objective function is often called *loss*. *OT* stands for Observation Table and *PT* for Parameter Table. It is possible that $OT = PT$.

Our main goal is to minimize the scalar:

```
SELECT sum(Loss) FROM OT
```

Differentiating a relational query means that we want to create the column $Loss'$ in *PT* that is the derivative of $OT.Loss$ with respect to $PT.X$. Doing so will unlock optimization through gradient-based methods. As it appears hard to differentiate arbitrary query, we reduce our scope to a subset of queries that should be wide enough to cover many industrial cases. First we do not consider a query that involves a Common Table Expression: these have to be inlined in the query. It drastically helps query compilation to ADSL. Second $PT.X$ should appear once and only once.

Let T_s be the set of tables used in $\mathcal{R}$. Then $\{OT, PT\} \subset T_s$.

Let's introduce the relation"$T_A \longrightarrow T_B$" when the primary key of T_A is a foreign key in T_B. It is said that T_A broadcasts into T_B.

Here is a simple way to create such T_A and T_B in SQL:

```
CREATE table TA AS
    SELECT foreignKey AS primaryKey
    FROM TB
    GROUP BY foreignKey
```

$(T_s, \longrightarrow)$ naturally forms a graph. Then we can compile the query $\mathcal{R}$ to the pair $(T_s, \longrightarrow) \times f$ where f is an ADSL program. Evaluation of $(T_s, \longrightarrow) \times f$ gives $OT.Loss$.

As f is an ADSL program it is possible to differentiate it with respect to the input associated to $PT.X$ as f'.

We state that with

$$\mathcal{R}' = (T_s, \longrightarrow) \times f'$$

evaluation of $\mathcal{R}'$ gives the expected $OT.Loss'$, which is represented in Figure 1.

We believe that this schema is a good way to differentiate relational queries: if the output is a relational query then we can use all the optimizations and parallelizations developed for regular ones.

4 GRAPH POINT OF VIEW

In this section we introduce notations on graphs that will be applied to the SQL Table tree in order to simplify it and facilitate its compilation to ADSL.

Definition 4.1 (Polytree). A Polytree is a directed acyclic graph whose underlying undirected graph is a tree.

For example, any tree structure of a website is a Polytree.

Definition 4.2 (Cross Edge). A cross-edge is a pair of edges in a graph $(A \longrightarrow B, C \longrightarrow B)$ which indicates that B comes from a **cross** operation between A and C.

Here is a simple way to create such an edge in SQL:

```
CREATE table B AS
    SELECT * FROM A
    CROSS JOIN C
```

Let P be a Polytree with cross-edges: $(N, E, (e_i, e_j))$

Definition 4.3 (PolyStar). Let's define a PolyStar $P\star$ as

$$P\star = \{(P, n) \mid P \text{ a Polytree with cross-edges \& n a node of } P\} \quad (1)$$

A PolyStar is a PolyTree with a special focus on a specific node of the graph.

Let $(P, ot) \in P\star$. We call

- an *upstream* node a node n of P such that $n \longrightarrow ot$.
- an *upstream cross* node a cross node n of P such that one of its parents is an upstream node.
- an *observation-cross* a cross node of P such that one of its parents is ot.
- a *downstream node* a node d of P such that is not an observation-cross node and that $ot \longrightarrow d$.
- We call a *full* node the remaining nodes of P.

By construction, there is no path between a full node and ot.

4.1 SQL Table tree simplification

A relational query in SQL creates many tables, even though some could be grouped. For example,

```
SELECT Loss FROM OT
```

creates another table with a bijection from its index to *OT* index. Thus we introduce a novel join operator that helps us to simplify the table tree: TOTAL JOIN. T_1 TOTAL JOIN T_2 ON $\langle\theta\rangle$ is the same semantic as T_1 INNER JOIN T_2 ON $\langle\theta\rangle$ with the additional constraint that for each line of T_1, there is *exactly* one line of T_2 that corresponds. To make a successful T_1 TOTAL JOIN T_2 ON $\langle\theta\rangle$ it is sufficient that θ columns are a primary key in T_2 and a foreign key in T_2, but it is not necessary.

Thanks to this join operator that is reminiscent of [16], we can gather tables in the graph that come from this operation. Indeed, creating a new table is thus equivalent to adding a column to the origin table. Then the compilation in ADSL from any join operator that is **not** a TOTAL JOIN gives a projector (a broadcast or

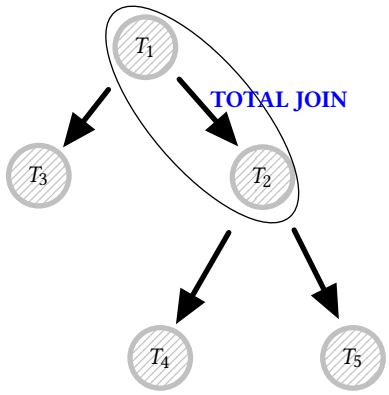

Figure 2: Simplification thanks to TOTAL JOIN

an aggregator). If a TOTAL JOIN is used then operations can be performed line by line thus we can compile it to a scalar operation such as *+*, *Call1*, *Call2* ...

Remark 2 (Sufficient condition). *If the query is written without any Common Table Expression and involves $PT.X$ only once, then $T_s, OT \in P\star$ where T_s are the used tables in $\mathcal{R}$.*

4.2 A supply chain example

In this section we take a real case from the supply chain industry to illustrate our previous formalization.

Let's consider that our database contains information on products that a company sells. It has the *Product* table recording the products. These products are organized by categories. The *Orders* table records products orders. Assuming that we also have a *Week* table whose primary is the week number, we would write:

```
CREATE table Category AS
    SELECT category FROM Product
    GROUP BY category

CREATE table CategoryWeek AS
    SELECT * FROM Category
    CROSS JOIN Week

CREATE table ProductWeek AS
    SELECT * FROM Product
    CROSS JOIN Week
```

Then we get Figure 3.

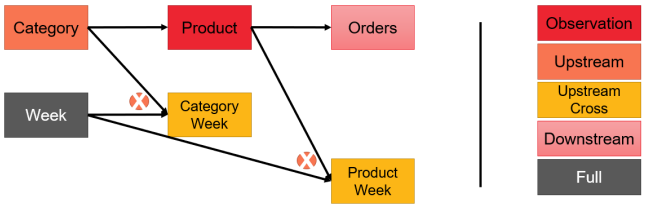

Figure 3: PolyStar

4.3 Why all these notations?

All these notations help us to compile the query as easily as possible. While computing a line of $PT.X'$ from $(T_s, \longrightarrow) \times f'$, an input coming from

- the observation table gives a scalar
- an upstream table gives a scalar
- a full table gives a vector of the size of the full table itself.
- an upstream-cross table gives a vector of the size of the left table used in the cross operation
- a downstream table gives a vector of certain size.

In summary, we introduce the TOTAL JOIN operator to turn the SQL Table tree into a PolyStar. Once it is a PolyStar, its lowering, i.e. compilation, to ADSL is simplified.

5 EXPERIMENTS

5.1 Dataset

We used the Chicago taxi rides dataset that can be found here [2]. We chose this dataset because it has also been used by [20] which partly motivated our work.

For each ride, we use the taxi identifier, distance (in miles) and the tips (in dollar). In this example, the Observation table is the *Trips* table and the *Taxis* table is an upstream table:

```
CREATE table Taxis AS
    SELECT taxiId, 1 AS a FROM Trips
    GROUP BY taxiId
```

5.2 Linear Regression

We use linear regression to predict the tip based on the trip distance.

$$tips = a \times distance + b \tag{2}$$

It is an interesting example to perform a benchmark but according to us, it does not illustrate the *relational* aspect of the dataset. Thus we also used an augmented version of this model where the slope depends on the taxi identifier, the intercept remains shared among taxis:

$$tips_{\text{taxiId}} = a_{\text{taxiId}} \times distance + b \tag{3}$$

This example illustrates how the relational information between the *Trips* table and the *Taxis* table has to be used.

```
DECLARE @intercept FLOAT;
SET @intercept = 0.0;

SELECT
  tripId, taxiId,
  (Estimated - Tips)^2 AS Loss,
FROM (
  SELECT
    *,
    Taxis.a * Trips.distance + @intercept
        AS Estimated
  FROM Trips
  INNER JOIN Taxis
  WHERE Trips.taxiId = Taxis.taxiId );
```

[2] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew

$$\frac{\partial}{\partial a}(ax+b-y)^2 = 2a(ax+b-y)$$

thus the derived query (with respect to the slope a) should be:

```
DECLARE @intercept FLOAT;
SET @intercept = 0.0;

SELECT
  tripId, taxiId,
  2 * a * (Estimated - Tips) AS Gradient,
FROM (
  SELECT
    *,
    Taxis.a * Trips.distance + @intercept
        AS Estimated
  FROM Trips
  INNER JOIN Taxis
  WHERE Trips.taxiId = Taxis.taxiId );
```

For the sake of notation, slopes are initialized at 1 and intercept at 0.

Such a model has a straight forward explanation; the model can be white boxed. Taxi's slope shows its ability to get tips. It is called Interpretable ML [19]. We've used linear regression for simplicity sake, but unlocking differentiable programming, i.e. access program derivative, to relational programming language unlocks an amazing variety of other models. All experiments were run on *Azure* with ¨Standard_L8s_v2¨, a 8 *vCPU* machine, running at 2.557 *GHz* with a disk of 1.9TB NVMe. Our prototype and experiments were run on a supply chain Domain Specific Language. It is a Python·like implementation of SQL narrowed for supply chain problems. Tests were carried out five times and the average runtimes recorded. In Table 1, we present our result for 10 epochs of gradient descent on the Chicago dataset.

Table 1: Runtime for Linear Regressions

Trips	Taxis	Shared Slope (sec)	Taxi's Slope (sec)
10^3	479	5.6×10^{-2}	6.1×10^{-2}
10^5	1037	3.14×10^{-1}	4.27×10^{-1}
1.95×10^7	9201	4.47×10^2	5.38×10^2

In Table 1, Shared Slope is the implementation relative to equation 2 and Taxi's Slope is relative to equation 3. We've not reproduced all experiments from [20] as our focus is to differentiate a relational query that involves table relationships. In our example this is the relationship between *Trips* and *Taxis* tables.

6 CONCLUSION

In this work we've presented a concrete approach to perform differentiation on relational query. Our claim is that derived query should also be a query. Thus we have introduced a dedicated programming language ADSL that is closed by differentiation. Thanks to the introduced operator TOTAL JOIN and PolyStar, we can clarify the roles of different tables in the relational query to differentiate. Our implementation allows us to efficiently tackle real world problems such as those encountered in a supply chain, for example. We hope that relational programming language will consider Automatic Differentiation as first class citizens in the future, this would strengthen "Query 2.0" [22] and unlock many interesting applications. This would help every engineer working on relational databases to develop efficient white-box models by easily plugging their expertise into it.

ACKNOWLEDGMENTS

This work was supported by ANRT French program and Lokad.

REFERENCES

[1] Martín Abadi and Gordon Plotkin. 2019. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages* 4 (12 2019), 1–28. https://doi.org/10.1145/3371106
[2] Atilim Baydin, Barak Pearlmutter, Alexey Radul, and Jeffrey Siskind. 2018. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research* 18 (04 2018), 1–43.
[3] Atilim Günes Baydin, Barak A. Pearlmutter, and J. Siskind. 2016. DiffSharp: An AD Library for .NET Languages. *ArXiv* abs/1611.03423 (2016).
[4] Olivier Breuleux and Bart van Merriënboer. 2017. Automatic Differentiation in Myia.
[5] M. Cerezo and Patrick Coles. 2020. Impact of Barren Plateaus on the Hessian and Higher Order Derivatives.
[6] Hoon Chung, Sung Joo Lee, Hyeong Bae Jeon, and J. Park. 2020. Semi-Supervised Speech Recognition Acoustic Model Training Using Policy Gradient. *Applied Sciences* 10 (2020), 3542.
[7] Autodiff.org community. 2020. Tools for Automatic Differentiation. http://www.autodiff.org/?module=Tools.
[8] L. Hascoët and V. Pascual. 2013. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.* 39 (2013), 20:1–20:43.
[9] Y. Hu, L. Anderson, Tzu-Mao Li, Q. Sun, N. Carr, Jonathan Ragan-Kelley, and F. Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ArXiv* abs/1910.00935 (2020).
[10] Michael Innes. 2018. Don't Unroll Adjoint: Differentiating SSA-Form Programs. (10 2018).
[11] Mike Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and Will Tebbutt. 2019. A Differentiable Programming System to Bridge Machine Learning and Scientific Computing. *ArXiv* abs/1907.07587 (2019).
[12] Tzu-Mao Li. 2019. Differentiable Visual Computing. *ArXiv* abs/1904.12228 (2019).
[13] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, F. Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)* 37 (2018), 1 – 13.
[14] Carol Mak and C. Ong. 2020. A Differential-form Pullback Programming Language for Higher-order Reverse-mode Automatic Differentiation. *ArXiv* abs/2002.08241 (2020).
[15] Andrea Mari, Thomas Bromley, and Nathan Killoran. 2020. Estimating the gradient and higher-order derivatives on quantum hardware.
[16] Frank McSherry. 2010. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. *Commun. ACM* 53 (09 2010), 89–97. https://doi.org/10.1145/1559845.1559850
[17] B. V. Merrienboer, D. Moldovan, and Alexander B. Wiltschko. 2018. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. *ArXiv* abs/1809.09569 (2018).
[18] Marc Rasi Bart Chrzaszcz Richard Wei, Dan Zheng. 2020. Differentiable Programming Manifesto. https://github.com/apple/swift/blob/main/docs/DifferentiableProgramming.md.
[19] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1 (05 2019), 206–215. https://doi.org/10.1038/s42256-019-0048-x
[20] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, A. Kemper, Stephan Günnemann, and T. Neumann. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW*.
[21] Bart van Merriënboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. (10 2018).
[22] Wu Weiyuan, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven Training Data Debugging for Query 2.0. 1317–1334. https://doi.org/10.1145/3318464.3389696