Learning Decision Trees over Factorized Joins

Lukas Kobis

University College

University of Oxford

Supervisor: Prof. Dan Olteanu

Final Honour School of Mathematics and Computer Science - Part C

Trinity 2017

Abstract

This thesis investigates the problem of learning decision trees over training datasets defined as the answers of arbitrary join queries over relational databases.

Existing systems do not integrate the learning and the querying components. We follow a new trend in research that does combine both. This comes with clear benefits:

The flat join result, that is commonly used by state-of-the-art approaches, can have a high degree of redundancy. We instead employ a factorization of the join answer, which avoids redundancy in the representation and, most importantly, in its computation.

Our approach can learn a decision tree over training data described by a query Q over a database $\mathbf{D}$ in $\mathcal{O}(|\mathbf{D}|^{\mathit{fhtw}(Q)})$ whereas any relational approach that relies on listing the full representation of the join answer takes time $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$. Note that $\mathit{fhtw}(Q) \leq \rho^*(Q)$ and the gap between the two quantities can be as large as the number of relations in the query Q. Thus, the speed-up can be exponential.

We show our implementation can be up to 2 orders of magnitude faster than a state-of-the-art implementation (MADlib) in learning regression trees for the considered dataset.

Acknowledgements

I would like to thank my supervisor Prof. Dan Olteanu. The weekly consultations brought the project forward and the discussions were very productive. I also want to thank Prof. Olteanu's DPhil student Maximilian-Joel Schleich who attended many of the meetings and helped me with many implementation specific questions.

I also wish to express my gratitude to Lewis Ruks, for proof-reading my thesis.

Last but not least, I want to thank my family and my girlfriend for their continuous support and encouragement.

Contents

1 Introduction **1**
- 1.1 Related Work . . . 2
- 1.2 Contributions . . . 2
- 1.3 Outline . . . 3

2 Preliminaries **4**
- 2.1 Decision trees . . . 4
  - 2.1.1 Example: learning a classification tree . . . 7
  - 2.1.2 Advantages and disadvantages . . . 9
- 2.2 Factorized databases . . . 11
  - 2.2.1 Factorized data representations . . . 11
  - 2.2.2 Aggregations over factorized representations . . . 14

3 Learning decision trees over factorized joins **18**
- 3.1 Conditioned aggregates . . . 18
- 3.2 Caching aggregation results . . . 18
- 3.3 Learning regression trees over factorized representations . . . 19
- 3.4 Learning classification trees over factorized representations . . . 20
- 3.5 Time complexity . . . 21

4 Implementation **22**
- 4.1 Reference implementation (MADlib) . . . 22
- 4.2 Implementation based on factorized joins . . . 24
  - 4.2.1 Memory allocation and Caching issues . . . 25
  - 4.2.2 Reusing aggregates . . . 26
  - 4.2.3 Order of iterations . . . 26
  - 4.2.4 Materializing the factorization vs. recomputing the join . . . 27
  - 4.2.5 An aside: sampling over a stream . . . 27

5 Experiments **28**
- 5.1 Benchmark Setup . . . 29
- 5.2 Implementations . . . 30
- 5.3 Dataset . . . 30
- 5.4 Accuracy . . . 31
- 5.5 Results . . . 32
  - 5.5.1 Size of the join result vs. size of the factorization . . . 32
  - 5.5.2 Total wall-clock time: MADlib vs. our implementations . . . 32
  - 5.5.3 Total wall-clock time: comparison of our implementations . . . 36

5.5.4 Wall-clock time per regression tree level 37
5.5.5 Wall-clock time per node 38
5.5.6 Exporting the data from PostgreSQL and loading it into R 39

6 Conclusions and future work **41**
6.1 Conclusions 41
6.2 Future work 42
6.2.1 Improve memory efficiency 42
6.2.2 Avoid more recomputations 42
6.2.3 Parallelization and distribution 42
6.2.4 Adapting f-trees 43
6.2.5 Verify results using more datasets 43

Bibliography **44**

Chapter 1

Introduction

Machine learning has led to advances in many applications over the years: smartphones can be controlled using speech-recognition systems [9], computers have reached nearly human-level accuracy in translation tasks in many languages [30], and self-driving cars utilising this technology can be spotted on various roads [7]. This is mainly due to advances in machine learning, brought about by the increased attention it has received recently, in both academia and industry [26]. However, large data sets are particularly common, with their size growing day by day. Working with such data sets is computationally expensive and more efficient methods of extracting knowledge from data will likely be essential in the near future. [14]

Many machine learning algorithms have been developed; one such method will be the focus of this thesis, that of *decision tree learning*. In a comparison between seven widely-used machine learning methods (using different performance metrics) a calibrated boosted decision tree algorithm led to the most accurate predictions for the models overall [3], and hence we chose this as our focus of study here.

The current machine learning algorithms have deficiencies in their handling of the data presented to them; in the commercial setting data is mainly stored in relational databases that consist of multiple tables. However, machine learning algorithms usually require the data to be represented as one large matrix. Thus, the relational tables are joined before the result gets passed to a conventional algorithm [16], and this operation is very expensive. Moreover, the join result may contain a large amount of redundant information that all commonly-used algorithms do not handle specifically. The benefits of fixing this flaw would be considerable, and thus there is an increasing interest in both academia and industry to integrate machine learning into database systems [8, 21, 11].

In the following we contribute to the intersection of machine learning and databases. The classical machine learning problem - decision tree learning - is treated from a database perspective.

1.1 Related Work

In 2015, a SIGMOD [1] panel was held on the crossover of the disciplines of machine learning an databases [21], and this is just one example of the increased attention that this 'intersection' has gained recently. In particular a number of papers have been written on the topic [8, 10, 15].

An approach, similar to what we propose in this thesis, detects patterns in the data matrix that come from the underlying relational structure of the data, with the aim of speeding up a linear regression learning task [22]. However, this approach does not fully capture join dependencies, and we aim to fully exploit the dependencies that result from join queries. We will follow the approach taken by Olteanu et al. that avoids the computation of the flat join completely and identifies patterns directly from the join query and the relational tables. Olteanu et al. use factorized databases to speed up learning regression models. By calculating various aggregates for each table and combining them appropriately, they are able to avoid materializing the join [19]. We will apply similar ideas to the task of decision tree learning.

To our knowledge there is no decision tree learning implementation available that leverages in-database knowledge to improve computation times. Thus, this thesis is the first in analyzing this problem setting. MADLib [8] is the end-to-end decision tree learning solution that is the closest to our approach, since it is an in-database framework that implements this algorithm. All other implementations require the data to be first exported from a relational database management systems (RDBMS). This adds additional time and increases the complexity of an end-to-end system.

1.2 Contributions

To sum up, the contributions of this thesis are as follows:

- We survey existing work on learning of decision trees and identify shortcomings;

[1]"The annual ACM SIGMOD/PODS conference is a leading international forum for database researchers, practitioners, developers, and users to explore cutting-edge ideas and results, and to exchange techniques, tools, and experiences." - `http://sigmod2017.org/`

- we propose a novel approach that fully exploits dependencies in the join result of relational tables when training decision trees;
- we show experimentally how this approach improves the running time of decision tree training significantly.
- Finally, we suggest ideas for possible future work and summarize possible extensions.

1.3 Outline

The thesis is organized as follows:

- In **Chapter 2**, we introduce the theoretical foundations of decision trees and factorized databases.
- In **Chapter 3**, we show how one can use factorized data representations to speed up the decision tree learning task.
- In **Chapter 4**, we give a detailed overview of the implementation and various problems we faced while implementing the previously described algorithms.
- In **Chapter 5**, we compare our implementation with an existing decision tree learning algorithm - namely, MADlib [2] [8] - a big data machine learning framework that works directly on an SQL database and is well-known for its efficiency. Furthermore, we compare it to a regression tree implementation in R [3] [28] - a programming language that is very popular amongst scientists and is used for many statistical computing tasks.

[2] `http://madlib.incubator.apache.org/`

[3] `https://www.r-project.org/`

Chapter 2

Preliminaries

2.1 Decision trees

This section gives an introduction to decision trees and how they can be used to solve machine learning tasks. Decision trees have been applied to many different fields: filtering noise from Hubble Space Telescope images [24], semiconductor manufacturing [13], three-dimensional object recognition [27], and drug analysis [5], to name a few.

This section is based on [17, chapter 16.2]. A more detailed survey was published by Safavian and Landgrebe [23].

Decision trees are a common model for various machine learning tasks, and before we introduce this, we familiarise the reader with the concept of Machine learning. Machine learning can be described in the following way: Suppose we have a relation $R = \{(x_1^{(i)}, ..., x_n^{(i)}, y^{(i)}) : 1 \leq i \leq m\}$. Our goal is to define a model that predicts y given a new unknown sample $(x_1, ..., x_n)$. Say the schema of R is $(X_1, ..., X_n, Y)$. We call the $X_1, ..., X_n$ *features* and Y is called the *target attribute*. We write $(x^{(i)}, y^{(i)})$ for $(x_1^{(i)}, ..., x_n^{(i)}, y^{(i)})$ to denote all features as a vector.

Decision trees partition the input space recursively, and handle every partition by a different local model, usually a constant function. Every internal node of the decision tree corresponds to one condition that partitions the input space and every leaf corresponds to one final partition, to which a local model is applied.

If the domain of the target attribute $\mathrm{dom}(Y)$ is finite we call the elements of $\mathrm{dom}(Y)$ labels and the corresponding decision tree is often referred to as a **classification tree**. If $\mathrm{dom}(Y)$ is infinite (usually $\mathbb{Z}$ or $\mathbb{R}$) we talk about **regression trees**. The literature often refers to these as CART models: **c**lassification **a**nd **r**egression **t**rees.

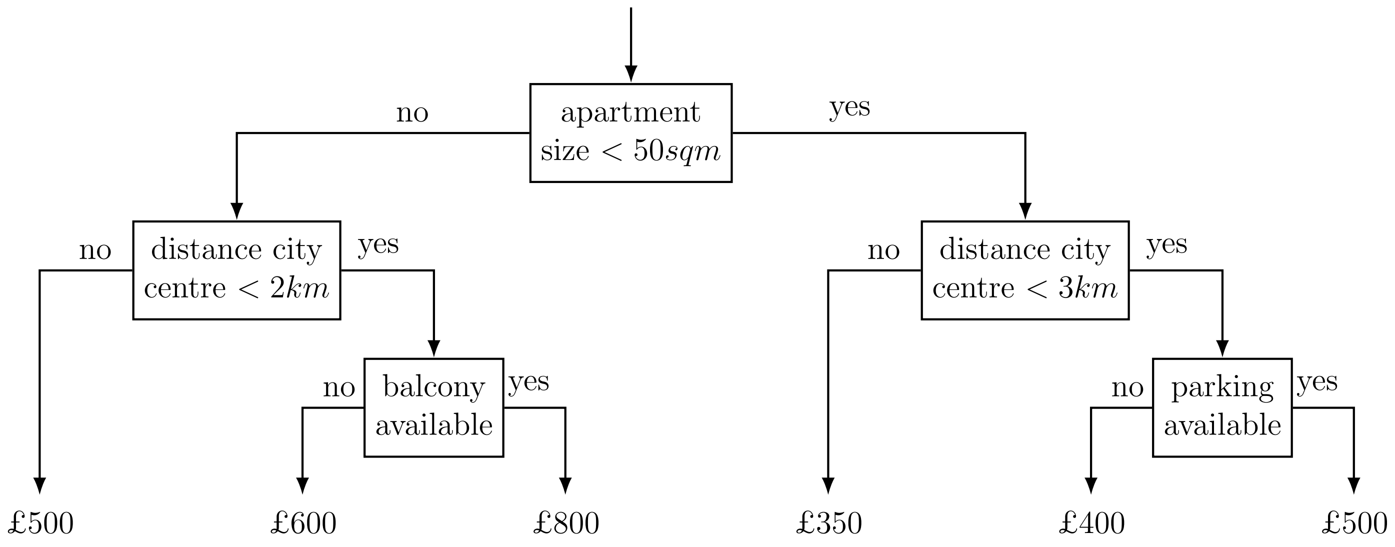


Figure 2.1: An example regression tree that models apartment rent prices.

A sample regression tree is illustrated in Figure 2.1. It models rent prices in £ of apartments based on various attributes. For example, the apartment size is a real-valued feature whereas the availability of a balcony is a discrete / binary feature.

For various definitions of optimally it is NP hard to find the optimal partitioning of the input space [12], but computing a decision tree greedily leads to very good results in practice.

Decision tree learning algorithms usually differentiate between continuous and categorical features. In our case continuous features have the domain $\mathbb{R}$ and the domain of a categorical feature is some finite set.

The greedy decision tree learning algorithm chooses, at each step, a feature j^* that splits the current data in an optimal way, i.e. for the categorical features we want to find:

$$(j^*, t^*) = \operatorname*{arg\,min}_{j, t \in \mathrm{dom}(j)} cost(\{(x, y) \in R : x_j = t\}) + cost(\{(x, y) : x_j \neq t\}) \tag{2.1}$$

and for real-valued features we want to find:

$$(j^*, t^*) = \operatorname*{arg\,min}_{j, t \in \mathrm{dom}(j)} cost(\{(x, y) \in R : x_j \leq t\}) + cost(\{(x, y) : x_j > t\}). \tag{2.2}$$

This split partitions the relation R into a disjoint union of two sub-relations. This split is represented by a decision tree node. Each node can then be split again recursively using the same method to build a decision tree.

For categorical features we could also start a new branch for each categorical value. These non-binary trees can be prone to overfitting or data fragmentation (the problem being that too little data falls in each subtree).

For continuous features it is impossible to compute the $\arg\max$ over all values in the domain since $\mathrm{dom}(j) = \mathbb{R}$. It is sufficient to let t range over all values that appear in R, but even so, this can still be too many values to feasibly compute the $\arg\max$. Thus, the $\arg\max$ is approximated in the continuous case by choosing an evenly distributed set of possible split points that t ranges over.

There are various termination conditions for this recursive procedure. One can limit the depth of the tree, the number of leaves, stop when the distribution of a node is homogeneous enough (for some appropriate measure), or stop when the number of samples for a node is too small. In practice a selection of these are usually employed.

Finally, we need to choose a *cost* function. Let us first consider the case of decision trees, i.e. $\mathrm{dom}(Y)$ is finite.

For every node with data R and every possible output $y \in \mathrm{dom}(Y)$ define

$$\pi_y = \frac{1}{|\mathcal{R}|} \sum_{(\_,y') \in R} \mathbb{1}(y = y').$$

Now we can define the *cost* function for a node by its misclassification rate, i.e. for every data point in the leaf we predict $y^* = argmax_y \pi_y$. Then the error is given by

$$M(\pi) = \frac{1}{|R|} \sum_{(\_,y) \in R} \mathbb{1}(y \neq y^*) = 1 - \pi_{y^*}.$$

One can also define the *cost* function given by the entropy of the distribution π.

$$H(\pi) = -\sum_y \pi_y \log(\pi_y).$$

Another commonly used *cost* function is the expected error rate, also known as the Gini index.

$$GI(\pi) = \sum_y \pi_y (1 - \pi_y).$$

π_y is the probability that a random sample belongs to category y and $1 - \pi_y$ is the probability that it will be misclassified.

There are various other *cost* functions, but the ones described above are the most prominent.

If we do not want to predict classes but predict real values we can use regression trees, i.e. $\mathrm{dom}(Y) = \mathbb{R}$. We predict the mean of the label variable in the data for a given node, i.e. $y^* = \frac{1}{|R|}\sum_{(\_,y)\in R} y$. Then the *cost* function is given by the mean squared error

$$cost(R) = \frac{1}{|R|}\sum_{(\_,y)\in R} (y - y^*)^2.$$

A pseudocode description of the decision tree learning algorithm is given in Algorithm 1. Learning regression trees requires some trivial modifications. Note that the *split*() function finds an optimal split with respect to one of the cost measures given above.

Algorithm 1: Learning a decision tree over training data given in relation R.

```
1 function $learn(R)$
2     $node = $ new $Node$
3     $node.prediction = \begin{cases} \arg\max_{(\_,y)\in R} \pi_y & \text{for classification trees} \\ \frac{1}{|R|}\sum_{(\_,y)\in R} y & \text{for regression trees} \end{cases}$
4     $(j^*, t^*, R_L, R_R) = split(node, R)$
5     if not $should\_split(R_L, R_R)$ then return $node$
6     $node.condition = \begin{cases} \lambda x : x_{j^*} = t^* & \text{for classification trees} \\ \lambda x : x_{j^*} < t^* & \text{for regression trees} \end{cases}$
7     $node.left = learn(R_L)$
8     $node.right = learn(R_R)$
9     return $node$
```

This algorithm runs in $\mathcal{O}(K \cdot T_{split})$ if we compute K nodes and the function *split*() takes time T_{split}. Let F_{disc} and F_{cont} be the sets of discrete and continuous features and suppose we use S split points for continuous features. Then,

$$T_{split} = \mathcal{O}\left(N \cdot \left(\sum_{f\in F_{disc}} |\operatorname{dom}(f)| + S \cdot |F_{cont}|\right)\right).$$

Recall that N is the number of samples in the training dataset.

2.1.1 Example: learning a classification tree

As an example consider the following input relation R:

X_1	X_2	X_3	Y
1	1	0	1
1	1	0	1
0	1	1	0
0	1	1	0
1	0	0	0
1	1	1	0
0	0	1	1
1	0	1	0

Table 2.1: Sample input relation R with schema (X_1, X_2, X_3, Y).

Calling $learn(R)$ would create a root node with prediction value 0, since most tuples have Y-value 0. Next, the algorithm finds the best split condition. In this example we will use the misclassification rate as the cost function: $cost(\cdot) = M(\cdot)$.

The costs for the first three possible splits are given in Table 2.2. Note that R_L contains all tuples that satisfy the split condition and R_R contains all tuples that do not.

split condition	$cost(R_L)$	$cost(R_R)$	$cost(R_L) + cost(R_R)$
$X_1 = 1$	2 / 5	1 / 3	11 / 15
$X_2 = 1$	2 / 5	1 / 3	11 / 15
$X_3 = 1$	1 / 5	1 / 3	8 / 15

Table 2.2: Costs associated to the three possible split conditions.

Thus, we choose $X_3 = 1$ as the split condition of the root node of the decision tree. We proceed recursively on R_L and R_R (see Table 2.3).

X_1	X_2	X_3	Y
0	1	1	0
0	1	1	0
1	1	1	0
0	0	1	1
1	0	1	0

X_1	X_2	X_3	Y
1	1	0	1
1	1	0	1
1	0	0	0

Table 2.3: R_L and R_R after the first split on $X_3 = 1$.

The resulting tree is shown in Figure 2.1.1, and has an error of 0 on the sample data given in R.

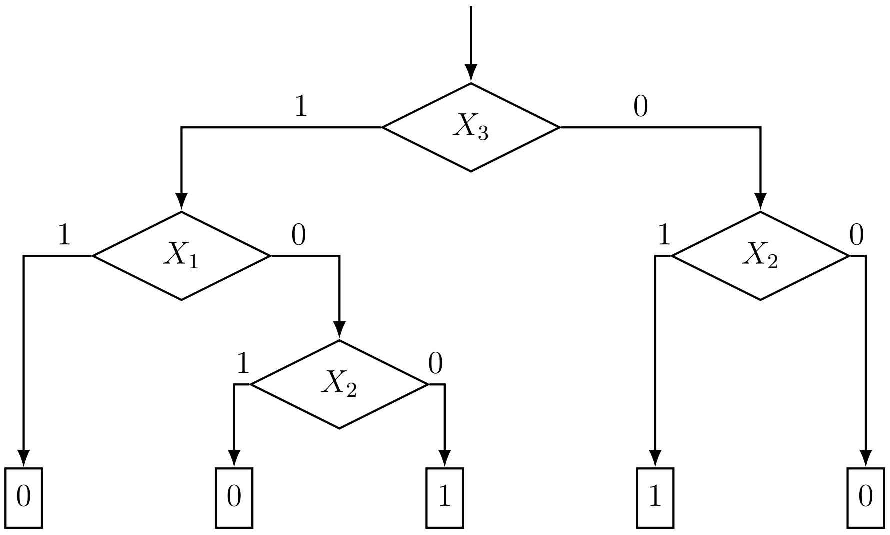


Figure 2.2: An example classification tree over the features X_1, X_2, and X_3, with domains $\{0, 1\}$. It fits the data given in relation R.

2.1.2 Advantages and disadvantages

Decision trees are very popular models because they can be easily interpreted by looking at the node conditions. Moreover, they can handle both discrete and continuous features naturally which makes them very attractive for real-world applications.

Deep trees usually over-fit the training data. To avoid this one can prune the tree accordingly. A complete tree can be pruned after learning, by removing the nodes that decrease the training error the least. One may use cross-validation to determine how many nodes one should prune. Other regularization methods have already been discussed above (limiting the tree height, not splitting nodes with too few examples, etc).

Moreover, decision trees are very unstable. Small changes in the input data potentially lead to completely different tree structures, as errors at the top of the tree effect all deeper levels. Thus, decision trees are called high-variance estimators.

The most obvious solution to this problem is to train multiple trees on randomly chosen subsets of the training data and combine their results. This technique is called **b**ootstrap **agg**regating, or bagging. However, this approach can lead to highly-correlated trees, so that averaging them does not reduce the variance enough. This may be remedied by selecting both a random subset of the training data and a random subset of feature variables for each tree. This methos is called random forest learning. Random forest models work very well on a wide range of tasks [3] and are used in many applications, for example in tracking human bodies in Microsoft's Kinect for XBox 360 [4].

Another approach to reduce the model's variance is called boosting. Boosting uses a learning algorithm - in our case a CART algorithm - and applies it multiple times to weighted training data. In the first training round all weights are equal. In subsequent rounds wrongly classified training samples are assigned higher weights.

Training multiple trees has many advantages; for example the training process can be parallelized easily. On the other hand, it makes the resulting model less interpretable. To get an idea of what the model does, one can calculate the relative importance of each feature by counting the number of times the feature occurs in any of the tree nodes.

2.2 Factorized databases

This section introduces factorized databases and gives an overview of previous work on the subject. Factorized databases are relational databases that use factorized data representations to reduce redundancy.

2.2.1 Factorized data representations

A factorized representation of a relation is an algebraic expression consisting of singletons, unions, and Cartesian products. Formally:

Definition 1. *A factorized representation (f-representation) E over a set S of attributes (schema) and domain D is a relational algebra expression of the form*

- $\varnothing$*, the empty relation over schema* S
- $\langle\rangle$*, the relation consisting of the empty tuple, if* $S = \varnothing$
- $\langle A : a\rangle$*, the unary relation with a single tuple with value* a*, if* $S = \{A\}$ *and* $a \in D$
- (E)*, where* E *is a factorized representation over* S
- $E_1 \cup ... \cup E_n$*, where each* E_i *is a factorized representation over* S
- $E_1 \times ... \times E_n$*, where each* E_i *is a factorized representation over* S_i *and* S *is the disjoint union of all* S_i*.*

Define the size of E *to be the number of singletons (*$\langle A : a\rangle$ *or* $\langle\rangle$*) in* E*.*

Factorized representations are used to avoid redundancy in the data representation.

Example 1. *For example, consider the relation* $R = \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$ *It can be represented by a flat representation as*

$$\begin{aligned} R =&(\langle A:0\rangle \times \langle B:0\rangle \times \langle C:0\rangle) \cup \\ &(\langle A:0\rangle \times \langle B:0\rangle \times \langle C:1\rangle) \cup \\ &(\langle A:0\rangle \times \langle B:1\rangle \times \langle C:0\rangle) \cup \\ &(\langle A:0\rangle \times \langle B:1\rangle \times \langle C:1\rangle) \cup \\ &(\langle A:1\rangle \times \langle B:0\rangle \times \langle C:0\rangle) \cup \end{aligned}$$

$$(\langle A:1\rangle \times \langle B:0\rangle \times \langle C:1\rangle) \cup$$
$$(\langle A:1\rangle \times \langle B:1\rangle \times \langle C:0\rangle) \cup$$
$$(\langle A:1\rangle \times \langle B:1\rangle \times \langle C:1\rangle).$$

Using the distributivity law of the Cartesian product over unions, the relation can be represented more succinctly:

$$R = (\langle A:0\rangle \cup \langle A:1\rangle) \times (\langle B:0\rangle \cup \langle B:1\rangle) \times (\langle C:0\rangle \cup \langle C:1\rangle).$$

The introduced relational representation scheme is complete. Every tuple can be written as a product of singletons and the relation is the union of all tuples. Thus, there always exists a flat factorization.

The example shows that factorizations are not unique. Ideally, we want to find the smallest representation of a relation.

If we generalize the example to tuples of length n, we can see that the flat representation has size $n \cdot 2^n$, whereas the factorized representation has size $2n$. Thus, factorized representations can be exponentially smaller than the equivalent flat representation.

Furthermore, tuples can be enumerated with constant delay from both a factorized and a flat representation [20], which makes them very useful in many database applications.

Example 2. *Taken from [25].*

In Figure 2.3(a) one can see a database with three relations (Shops, House, TaxBand) and their natural equijoin. The relation Shops lists zip codes and opening hours; the relation House lists zip codes, living areas, and prices; and the relation TaxBand lists living areas corresponding to their city tax bands.

The join result contains a lot of redundancies. For example, the value z_1 occurs in 24 tuples paired 8 times with each h_1, h_2, and h_3. Each H value is paired with the same combination of S and P values.

The first three tuples from the join result in Figure 2.3(a) can be expressed by the following flat relational algebra expression:

$$\begin{aligned} &\langle Z:z_1\rangle \times \langle H:h_1\rangle \times \langle S:s_1\rangle \times \langle P:p_1\rangle \times \langle T:t_1\rangle \\ \cup\, &\langle Z:z_1\rangle \times \langle H:h_1\rangle \times \langle S:s_1\rangle \times \langle P:p_1\rangle \times \langle T:t_2\rangle \\ \cup\, &\langle Z:z_1\rangle \times \langle H:h_1\rangle \times \langle S:s_1\rangle \times \langle P:p_1\rangle \times \langle T:t_3\rangle. \end{aligned}$$

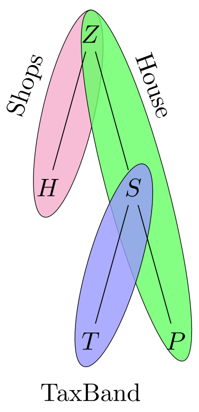
Shops	
Z	H
z_1	h_1
z_1	h_2
z_1	h_3
z_2	h_4

House		
Z	S	P
z_1	s_1	p_1
z_1	s_1	p_2
z_1	s_2	p_3
z_2	s_2	p_4

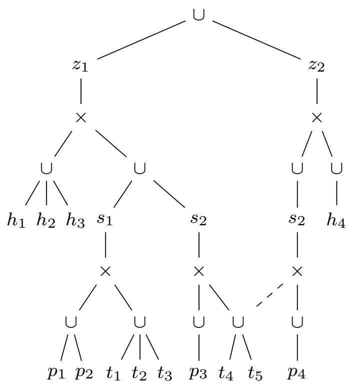
TaxBand	
S	T
s_1	t_1
s_1	t_2
s_1	t_3
s_2	t_4
s_2	t_5

Shops ⋈ House ⋈ TaxBand				
Z	H	S	P	T
z_1	h_1	s_1	p_1	t_1
z_1	h_1	s_1	p_1	t_2
z_1	h_1	s_1	p_1	t_3
z_1	h_1	s_1	p_2	t_1
z_1	h_1	s_1	p_2	t_2
z_1	h_1	s_1	p_2	t_3
z_1	h_1	s_2	p_3	t_4
z_1	h_1	s_2	p_3	t_5
....				
the above for h_2 and h_3				
....				
z_2	h_4	s_2	p_4	t_4
z_2	h_4	s_2	p_4	t_5

(a) Relations of database $\mathbf{D}$ and natural join $Q(\mathbf{D})$.



(b) F-tree F.



(c) Factorisation $F(\mathbf{D})$.

Figure 2.3: Taken from [25]. (a) Database $\mathbf{D}$ with relations House(Zipcode, Sqm, Price), TaxBand(Sqm, Tax), Shops(Zipcode, Hours), where the attribute names are abbreviated and the values are not necessarily distinct; (b) Nesting structure (f-tree) for the natural join of the relations; (c) Factorisation $F(\mathbf{D})$ of the natural join over F.

By applying the distributivity law of the Cartesian product over union, we get:

$$\langle Z : z_1\rangle \times \langle H : h_1\rangle \times \langle S : s_1\rangle \times \langle P : p_1\rangle \times \big(\langle T : t_1\rangle \cup \langle T : t_2\rangle \cup \langle T : t_3\rangle\big).$$

By applying this observation systematically on the whole join result we get the factorized representation depicted in Figure 2.3(c).

Figure 2.3(b) shows the nesting structure of the factorization tree given in Figure 2.3(c). This tree makes the conditional independence relationships clear. For example, given a Z value, the H values are independent of the S values and, given an S value, the T values are independent of the P values. Such trees are called f-trees, short for factorization trees [20]. Its nodes are attributes of the schema that satisfy the path constraint. That is, all attributes of a relation lie on the same root-leaf path. Also, every attribute is represented by exactly one node. Note that f-trees are not unique. The linked list of all attributes is always a valid f-tree and leads to the flat join representation. Thus, the f-tree determines the size of the factorization significantly.

We can reduce the size of the factorization further by reusing nodes: In Example 2 we can see that every s_2 value gets paired with both t_4 and t_5. Thus, we can cache the $t_4 \cup t_5$ node and reuse it. This is shown in Figure 2.3(c) by a dashed line. When using caching the corresponding factorization structures are called d-trees [20]. They can capture conditional independence relations between attributes. Since we do not focus on caching in this thesis we will mainly consider f-trees.

Good f-trees can be found by looking at cardinality estimates of relations and their join selectivities [25]. The following theorem bounds the size of the different representations:

Theorem 1. *Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits*

- *a flat representation of size* $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$ [1]
- *a factorization over f-trees* $\mathcal{O}(|\mathbf{D}|^{s(Q)})$ [20]
- *a factorization over d-trees* $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$ [20]

where $\rho^*(Q)$ is the fractional edge cover number, $s(Q)$ is the factorization number, and $fhtw(Q)$ is the fractional hypertree width of the query Q.

There are classes of databases $\mathbf{D}$ for which the obove size bounds are tight and worst-case optimal join algorithms to compute the join result in these representations [18, 20].

We know that

$$1 \leq fhtw(Q) \leq s(Q) \leq \rho^*(Q) \leq |Q|$$

holds where $|Q|$ is the number of relations in the query Q [25]. For example, for star queries we have $s(Q) = 1$ while $\rho^*(Q) = |Q|$. We have seen this exponential gap in the representation size already in Example 1. The gap between $fhtw(Q)$ and $s(Q)$ can be as large as $\log(|Q|)$ [20].

Next we will look at how to perform various aggregation tasks directly on the factorized representation without materializing the flat join result.

2.2.2 Aggregations over factorized representations

In the later sections we are interested in computing various aggregates, given an f-representation E over a schema S and some variable $T \in S$ (the target). For example, we want to find the number of elements in the relation represented by E and the sum of all T-values.

If we consider $\langle\rangle$ as an empty union, $\varnothing$ as an empty product, and $\langle A : a\rangle$ as $\langle A : a\rangle \times \langle\rangle$, then we only have the following two cases to consider when describing our recursive algorithms on the factorization tree structure:

$$\begin{tikzcd}
& \langle A : a \rangle \arrow[d, dash] & \\
& \times \arrow[dl, dash] \arrow[dr, dash] & \\
E_1 & \cdots & E_k
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& \cup \arrow[dl, dash] \arrow[dr, dash] & \\
E_1 & \cdots & E_k
\end{tikzcd}$$

In the follwing we will call these two cases α and β. The *count*(E) procedure is given in Algorithm 2.

Algorithm 2: Count aggregation procedure for an f-representation E over an f-tree.

```
count(E) {
  switch E {
    case α {
      if k = 0 then return 1
      else return $\prod_i count(E_i)$
    }
    case β {
      return $\sum_i count(E_i)$
    }
  }
}
```

The procedure *sum*(E, T) returns the sum of all T values in E. The pseudocode is given in Algorithm 3.

Algorithm 3: Sum aggregation procedure for an f-representation E over an f-tree with target T.

```
sum(E,T) {
  switch E {
    case α {
      if k = 0 and A = T then return a
      else if $E_j$ contains T then return $sum(E_j, T) \prod_{i \neq j} count(E_i)$
      else return 0
    }
    case β {
      return $\sum_i sum(E_i, T)$
    }
  }
}
```

In case α at most one subtree can contain the target variable T, since in the f-tree every attribute is represented by exactly one node.

For the regression tree learning algorithm, we also need to calculate the sum of the squares for some target attribute T. This can be done similarly, see Algorithm 4. The only difference to Algorithm 3 is the a^2 value in line 4.

Algorithm 4: Sum of squares aggregation procedure for an f-representation E over an f-tree with target T.

```
 1 $sum\_squares(E, T)$ {
 2   **switch** $E$ {
 3     **case** $\alpha$ {
 4       **if** $k = 0$ **and** $A = T$ **then return** $a^2$
 5       **else if** $E_j$ contains T **then return** $sum\_squares(E_j, T) \prod_{i \neq j} count(E_i)$
 6       **else return** 0
 7     }
 8     **case** $\beta$ {
 9       **return** $\sum_i sum\_squares(E_i, T)$
10     }
11   }
12 }
```

For the classification tree learning procedure we need to calculate histograms. That means given some target attribute T we want to calculate the distribution over dom(T). The pseudocode is given in Algorithm 5. Addition of two histograms and multiplication of a histogram and a scalar are defined point-wise. $\{a \to c\}$ denotes a histogram that contains c times the value a, and $\varnothing$ denotes an empty histogram.

Algorithm 5: Histogram aggregation procedure for an f-representation E over an f-tree with target T.

```
 1 $histogram(E, T)$ {
 2   **switch** $E$ {
 3     **case** $\alpha$ {
 4       **if** $k = 0$ **and** $A = T$ **then return** $\{a \to count(E)\}$
 5       **else if** $E_j$ contains T **then**
 6         **return** $\prod_{i \neq j} count(E_i) \cdot histogram(E_j, T)$
 7       **else return** $\varnothing$
 8     }
 9     **case** $\beta$ {
10       **return** $\sum_i histogram(E_i, T)$
11     }
12   }
13 }
```

It is easy to add caching to these algorithms. We can keep a hash map from nodes to aggregates and return the cached value in case we have computed it already. Due to memory constraints it usually only makes sense to cache values of attributes that appear very often in the factorized representation.

Furthermore, all the aggregation algorithms run in times scaling linear in the size of the factorized representation.

Chapter 3

Learning decision trees over factorized joins

We can obviously apply the standard learning algorithms after enumerating all tuples from the factorized data representation. In this section we show how to avoid this and learn decision trees over the factorized representation instead, which can be a lot more efficient.

3.1 Conditioned aggregates

The decision tree learning algorithm requires us to calculate counts restricted to the tuples of the join result that satisfy a certain condition.

Thus, we have to extend the aggregation functions by an argument for a condition C. A value $\langle A : a \rangle$ satisfies condition C if A is not restricted by C or if a satisfies all restrictions of A in C. Algorithm 6 shows the updated $count(E, C)$ pseudo code. The $sum(\cdot)$, $sum\_squares(\cdot)$, and $histogram(\cdot)$ aggregations work analogously.

In this modification we return 0 immediately if the value $\langle A : a \rangle$ does not satisfy the conditions C. Otherwise, $count(\cdot)$ performs exactly the same operations. Note that if conditions restrict variables of low depth in the f-tree, then this can prune large parts of the factorized representation and thus can increase efficiency.

3.2 Caching aggregation results

We are interested in many aggregates over similar conditions. Then, a large part of their computations will be similar or equal. For example, a count aggregation with the two conditions $C_1 = (H = h_1)$ and $C_2 = (H = h_2)$ will need to calculate the same counts over the S value subtree (this refers to Example 2).

Algorithm 6: Count procedure for an f-representation E over an f-tree restricted by condition C.

```
count(E, C) {
  switch E {
    case α {
      if ⟨A : a⟩ does not satisfy C then return 0
      else if k = 0 then return 1
      else return ∏_i count(E_i)
    }
    case β {
      return ∑_i count(E_i)
    }
  }
}
```

To do this, we can keep a hash map that maps the node in the factorization tree, and a condition restricted to those attributes that appear in the subtree to an aggregation value. These cached values can then be reused by later aggregations, if their condition restricted to the attributes in the subtree match one of the conditions in the hash map.

3.3 Learning regression trees over factorized representations

For learning regression trees we use the following cost function (see section 2.1):

$$cost(R) = \frac{1}{|R|} \sum_{(\_,y) \in R} (y - y^*)^2$$

where $y^* = \frac{1}{|R|} \sum_{(\_,y)\in R} y$ is the predicted value for the samples in relation R.

The cost function can be written as

$$\begin{aligned} cost(R) &= \frac{1}{|R|} \sum_{(\_,y) \in R} y^2 - 2yy^* + y^{*2} \\ &= \frac{1}{|R|} \sum_{(\_,y) \in R} y^2 - \frac{2y^*}{|R|} \sum_{(\_,y) \in R} y + y^{*2} \\ &= \frac{1}{|R|} \sum_{(\_,y) \in R} y^2 - \frac{2}{|R|^2} \left(\sum_{(\_,y) \in R} y \right)^2 + \left(\frac{1}{|R|} \sum_{(\_,y) \in R} y \right)^2 \end{aligned}$$

$$= \left(\frac{1}{|R|} \sum_{(\_,y)\in R} y^2 \right) - \left(\frac{1}{|R|} \sum_{(\_,y)\in R} y \right)^2 .$$

We can see that the cost function of a relation R only depends on $|R|$, $\sum_{(\_,y)\in R} y$, and $\sum_{(\_,y)\in R} y^2$. These are exactly the aggregates that we looked at in section 2.2.2. We can compute them efficiently over the factorized representation using the three algorithms described previously by setting the target attribute T to be the label attribute Y.

We use Algorithm 1 as the learning procedure. At every step we create a list of candidate conditions: For every discrete feature j, all conditions are of the form $x_j = t$ for $t \in \mathrm{dom}(j)$; for every continuous feature j, all conditions are of the form $x_j < t$ for t in the set of possible split points. The split points are computed before running the learning procedure by sampling data from the continuous features. Next, we take the conjunction of the condition of the current node, and one possible candidate condition, and run the aggregation algorithm on the factorized representation. We do this for all candidate conditions and find the one that minimizes the cost measure (see equations (2.1) and (2.2)).

3.4 Learning classification trees over factorized representations

There are various cost functions that can be used to learn classification trees (see section 2.1). All of them depend on the quantities

$$\pi_y = \frac{1}{|R|} \sum_{(\_,y')\in R} \mathbb{1}(y = y')$$

for $y \in \mathrm{dom}(Y)$. The values $\sum_{(\_,y')\in R} \mathbb{1}(y = y')$ are the ones that the histogram aggregation algorithm calculates and $|R| = \sum_y \sum_{(\_,y')\in R} \mathbb{1}(y = y')$. Thus, we can use the exact same methods described in the previous section.

Most applications have very few classes that we need to differentiate. Thus, keeping track of the full histogram is not an issue. For problems with many classes we might want to approximate histograms to decrease the memory usage of the algorithm. This can be done by using ideas from stream processing algorithms [2].

3.5 Time complexity

Recall the time complexity of learning decision trees from section 2.1:

$$\mathcal{O}\left(K \cdot N \cdot \left(\sum_{f \in F_{disc}} |\operatorname{dom}(f)| + S \cdot |F_{cont}|\right)\right).$$

K is the number of nodes we want to compute, N is the number of samples in the training dataset, F_{disc} and F_{cont} are the sets of discrete and continuous features, and S is the number of split points for continuous features.

However, in our problem setting the data is given as the result of a query Q over a relational database $\mathbf{D}$. The factorization with caching has size $\Theta(|\mathbf{D}|^{fhtw(Q)})$ (see Theorem 1, section 2.2.1). Since we can calculate all necessary aggregates in one pass over the factorized representation our algorithm has the following time complexity:

$$\mathcal{O}\left(K \cdot |\mathbf{D}|^{fhtw(Q)} \cdot \left(\sum_{f \in F_{disc}} |\operatorname{dom}(f)| + S \cdot |F_{cont}|\right)\right).$$

The flat join has size $\Theta(|\mathbf{D}|^{\rho^*(Q)})$ (see Theorem 1, section 2.2.1). Recall that $\rho^*(Q)$ is the fractional edge cover number and $fhtw(Q)$ is the fractional hypertree width of the query Q. Furthermore, $fhtw(Q) \leq \rho^*(Q)$ and the gap between the two quantaties can be as large as the number of relations in the query Q. Thus, our algorithm can be exponentially faster than a conventional decision tree learning algorithm that operates on the flat join.

Chapter 4

Implementation

This section covers the details of the implementation and challenges faced in transforming the theoretical ideas described earlier into functional C++ code.

4.1 Reference implementation (MADlib)

One big issue facing us is that the decision tree learning algorithms are not very specific; there are many different implementations that can be found online. To make our results comparable to a state-of-the-art decision tree learning implementation, we will choose one framework and implement the exact same algorithm.

We will use MADlib [1] [8] for this task - a big data machine learning framework that works directly on an SQL database and is well-known for its efficiency. Its website describes it as "an open-source library for scalable in-database analytics. It provides data-parallel implementations of mathematical, statistical, graph and machine learning methods for structured and unstructured data." Its low-level abstraction layer is written in C++, and the higher level functions are implemented in Python. It can run on multiple relational database management systems (RDBMSs), but for this thesis all experiments have been performed on PostgreSQL.

The framework provides a function for CART learning:

```
tree_train(
  training_table_name,
  output_table_name,
  id_col_name,
  dependent_variable,
  list_of_features,
```

[1] `http://madlib.incubator.apache.org/`

```
    list_of_features_to_exclude,
    split_criterion,
    grouping_cols,
    weights,
    max_depth,
    min_split,
    min_bucket,
    num_splits,
    pruning_params,
    surrogate_params,
    verbosity
  ).
```

The training data needs to be in the table `training_table_name` and we need to pass the name of the dependent variable (the target) and a list of features to the function. Depending on the column type of the target variable the function chooses whether to train a regression tree or a classification tree. There are many more options we can pass to the function. The most important ones are:

- `max_depth` (5): The maximum depth of the tree.
- `min_split` (2): The minimum number of elements that relation R needs to have such that we try to split the node further.
- `min_bucket` (1): The minimum size of relation R for terminal nodes.
- `num_splits` (100): The number of split points the algorithm uses for continuous features.

The values in parenthesis are the values that have been used for the experiments. Apart from the explanation of the parameters, there is no further documentation of how the algorithm works in detail. The detailed implementation has been taken from MADlib's GitHub page `github.com/apache/incubator-madlib` and refers to version 1.9.1.

To make the results of our implementation comparable to MADlib, we have stayed as close to the MADlib implementation as possible. Rather simple tasks can have a complex implementation: As an example Algorithm 7 shows how MADlib calculates the split points for continuous features.

Algorithm 7: Calculating split points for continuous features (MADlib implementation).

```
1 $n\_bins = 100$
2 $sample\_size = n\_bins \cdot n\_bins$
3 if $sample\_size < 10000$ then $sample\_size = 10000$
4 $actual\_sample\_size = sample\_size + 14 + \sqrt{196 + 28 \cdot sample\_size}$
5 $percentage = actual\_sample\_size / n\_rows$
6 run query
  SELECT * from training_table_name where random() < percentage
7 sort values by a continuous attribute
8 choose $num\_splits$ evenly spaced values for this attribute
```

4.2 Implementation based on factorized joins

The algorithms described earlier have been implemented in C++ to be as performant as possible. Due to time limitations we have only implemented regression tree learning for continuous features. In any case, the implementation should be easily adaptable for classification trees and discrete features also.

The implementation was based on the DFDB system that implements linear regression learning over factorized joins [19].

From a high-level perspective the program does the following: The tables are read from CSV files from disc, sorted, and then the factorized representation is built directly by the join algorithm. We use the worst-case optimal (up to log factors) join algorithm Leapfrog triejoin [29]. The factorized representation then gets passed to the regression tree learner.

The main issue for learning decision trees is that we need multiples passes over the data. Linear regression can be solved by computing the aggregates once on the whole dataset and then running a numerical optimization task on the result [19]. Decision tree learning requires many aggregations over the same data that are restricted by different conditions. We can still run multiple aggregations with one pass over the data, and it turns out that the order in which we run the aggregations (and the way we batch them together) significantly influences the running time. Furthermore, the factorized join result can be a very large tree structure, which makes iterating over it cache unfriendly. One has to find a compromise between keeping the factorization tree in memory and recomputing the join on the fly.

In the following we describe a few challenges faced when designing such a system.

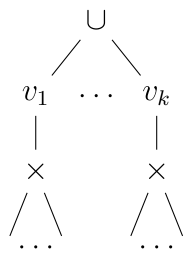
4.2.1 Memory allocation and Caching issues

In our first implementation we allocated many dynamic objects in the aggregation functions (for example, the recursive algorithms returned triples of numbers). After profiling the code, it can be seen that the program spends a majority of the running time allocating memory. Fixing these issues by using global variables for return values and other necessary data structures improves the performance of the code by up to 10x. Further improvements are possible by using plain C arrays instead of bloated STL containers.

Since we need to pass multiple times over the factorized representation of the join, we want to keep this expression in memory. For larger relational databases this can be a very large tree structure. Because of this it is essential to represent it as compactly as possible. The nodes of the tree are represented by the following struct:

```
struct Union {
  double* values;
  Union** children;
  unsigned int numberOfValues;
  unsigned short cacheIndex;
}.
```

Note that one node stores a union node with all its values and for each value the corresponding product nodes:



That means that `children[i][j]` points to the union node that is the j-th child of the product node corresponding to value i. Since the number of product nodes is determined by the f-tree, we do not need to store this information inside the node, and a count of the values `numberOfValues` is sufficient. The additional field `cacheIndex` keeps track of the position in the cache where we store aggregates (if caching for this node is enabled).

Not storing product nodes explicitly improves the memory footprint of the program significantly.

4.2.2 Reusing aggregates

Another problem facing us is deciding which aggregates we should store for later use. Reusing aggregates can improve the running time significantly, but it also increases the memory usage of the program. Finding a good compromise is essential. After various experiments on large datasets we have decided only to precompute the aggregates of union nodes without any restrictions by split conditions. When storing multiple aggregates for different split conditions, the memory limit is reached very quickly for every node. The operating system then swaps memory, i.e. it uses hard disc space as memory which slows down the computations more than the time improvements we may gain.

4.2.3 Order of iterations

When iterating the factorized representation, we can calculate multiple aggregates at once. Minimizing the number of iterations decreases the running time but calculating too many aggregates with one pass will make it impossible to keep all aggregates in the CPU cache. Thus, we have to find a compromise between the two.

At every step we have to calculate, for every leaf of the current regression tree $l_1, ..., l_n$, and for every attribute $a_1, ..., a_m$, every possible split condition. Say attribute a_i has splits $s^i_1, ..., s^i_l$. So the list of conditions that we need to calculate aggregates for is

$$\begin{gathered} c(l_1) \wedge a_1 < s^1_1, ..., c(l_1) \wedge a_1 < s^1_1 \\ ... \\ c(l_1) \wedge a_m < s^m_1, ..., c(l_1) \wedge a_m < s^m_1 \end{gathered} \tag{4.1}$$

for node 1, where $c(l_1)$ is the conjunction of conditions from the root node to leaf l_1. The other nodes $l_2, ..., l_n$ are treated similarly.

We implemented the folowing options:

- One aggregation (one condition) per iteration.
- One aggregation for each node and for each attribute per iteration, i.e. the aggregates for the conditions in every line in (4.1) are calculated in a single pass.
- One aggregation for all conditions for each node per iteration, i.e. the aggregates for all conditions in (4.1) are calculated in a single pass.

- All aggregates for a whole new regression tree level are calculates in a single pass, i.e. the conditions in (4.1) for all leaves $l_1, ..., l_n$.

The first and the last option lead to very bad results due to the reasons explained above. The second option performed depending on the data either better or worse than option three (see Chapter 5).

4.2.4 Materializing the factorization vs. recomputing the join

Since iterating over the in-memory representation of the factorized join takes a rather long time, due to the non-ideal locality of reference, it might make sense not to store the factorization at all and to compute the aggregates directly while joining the tables. This would also avoid keeping the factorized representation in memory at all which is impossible for very large join results.

Both variants are analyzed further in Chapter 5.

4.2.5 An aside: sampling over a stream

Ideally, we want to calculate the splitting points for continuous features while calculating the join. This boils down to calculating a uniform sample from a stream of values, i.e. the length is unknown. One could keep all values in memory and then sample uniformly, but we want to avoid this since the stream can be very long. The idea is the following: Whenever we get a value v we pair it with a random number $x \sim \mathcal{U}[0, 1]$ and store (x, v) in a max-heap of size k where k is the size of the sample we want to generate. When the heap is full and we want to add a new value (x', v'), we discard it if x' is greater than the current maximum in the heap; otherwise we remove the value with the maximum x-value from the heap and insert (x', v'). This way, the heap always contains the k values with lowest x-value. Since the x values are uniformly distributed, the heap contains a uniform sample of the stream at any point in time. Processing a heap of length n takes $\mathcal{O}(n \log n)$ time.

Chapter 5

Experiments

We compare our implementation against MADlib, a state-of-the-art in-databse machine learning framework. MADlib extends PostgreSQL with user defined aggregate functions (UDAFs) to perform analytics inside the database. Thereby, it avoids the cost of exporting data outside the database and importing it into a statistical package like R[1] [28]. This means that, like our approach, MADlib is an in-database solution to data analytics.

In addition, we look at the time it takes to export the data from a PostgreSQL database and import it into another machine learning framework. This is the usual machine learning workflow. In our case we load the data into the statistical framework R.

Our findings can be summarized by the following points:

- Our implementations yield almost identical qualitative results with respect to the training error.
- The running time of our algorithm is linear in the size of the factorized join representation.
- We verify this using a dataset that can be scaled according to some scale factor s such that the factorized representation grows linearly in s, whereas the flat join result has size $\mathcal{O}(s^3 \log s)$.
- For a moderate scale factor ($s = 7$) our implementation is 5 times as fast as MADlib. MADlib times out for larger scale factors, i.e. it takes more than 25 minutes.

[1] `https://www.r-project.org/`

- Deeper levels of regression trees have more nodes, thus our implementation takes more time. This is not true for MADlib. Thus, MADlib performs better for very deep trees.
- The time it takes to compute a new node has a very high variance, i.e. some nodes can be computed very quickly due to pre-aggregated values, whereas others can take very long. This could be an issue when parallelizing the computations.
- Our implementation offers even more advantages compared to systems that do not interact with a database directly. We compare our approach with the R framework. The data needs to be exported from PostgreSQL and then imported into the R system. For moderate scale factors ($s = 7$) our implementation has already learned a full regression tree before the data is ready to be processed by the R system.

In the following we describe the experiments in more detail and evaluate them appropriately.

5.1 Benchmark Setup

All experiments have been performed on an Intel(R) Core(TM) i7 2.2GHz/64bit with 16GB 1600MHz DDR3 RAM (MacBook Pro 15-inch, Mid 2015) running macOS 10.12.4. The C++ code has been compiled using Apple LLVM version 8.1.0 with the following optimization flags:

```
-Ofast
-mtune=native
-fassociative-math
-freciprocal-math
-fno-signed-zeros
-frename-registers
-fopenmp
```

We report wall-clock times by running each system five times and then reporting the minimum. [2]

[2]Minimum vs. average: Suppose the best case running time of the program is T. Any interference with other programs, process descheduling, etc. adds a positive random error to the measured time: $T + \epsilon_i$. The average time of n runs is then $T + \frac{1}{n}\sum_i \epsilon_i$, whereas the minimum is $T + min_i \epsilon_i$. Since

We run the benchmarks for MADlib with PostgreSQL 9.5.5 and MADLib 1.9.1. We tuned PostgreSQL for in-memory processing by setting its working memory to 14GB and shared buffers to 128MB, and by turning off parameters that affect performance (fsync, synchronous commit, full page writes, bgwriter LRU maxpages). We verify that it runs in memory by monitoring IO.

We set the following default parameters for MADlib: $max\_depth = 5$, $min\_split = 2$, $min\_bucket = 1$, and $num\_splits = 100$.

We use R version 3.3.3.

All of our systems are single threaded.

5.2 Implementations

We compare four implementations:

- **Mat-1**: Materializing the factorized representation and then calculating aggregates by running one aggregation for each node, and for each attribute, per iteration.
- **Mat-2**: Materializing the factorized representation and then calculating aggregates by running one aggregation, for all conditions, and for each node, per iteration.
- **Non-Mat**: Not materializing the factorized representation and recomputing the join for each level of the regression tree.
- **MADlib**.

We also analyze the time to export the data from PostgreSQL and import it into another machine learning framework. We use the programming language R for this purpose. Note that the regression tree learning implementation is different to ours. However, we can look at the export and import times.

5.3 Dataset

The experiments are run on a artificially generated housing dataset. In the following the tables and attributes are listed:

$min_i \epsilon_i \leq \frac{1}{n}\sum_i \epsilon_i$, the minimum yields a better estimate for T. A more elaborate discussion on this topic can be found at `https://mail.python.org/pipermail/python-dev/2016-June/145011.html`.

- House: postcode, livingarea, price, nbbedrooms, nbbathrooms, kitchensize, house, flat, unknown, garden, parking.
- Shop: postcode, openinghoursshop, pricerangeshop, sainsburys, tesco, ms.
- Institution: postcode, typeeducation, sizeinstitution.
- Restaurant: postcode, openinghoursrest, pricerangerest.
- Demographics: postcode, averagesalary, crimesperyear, unemployment, nbhospitals.
- Transport: postcode, nbbuslines, nbtrainstations, distancecitycentre.

All values are positive numbers and the natural join is a star join on the attribute `postcode`. There are 25000 postcodes and a scale factor s determines the number of records per postcode in each relation: There are s houses and shops per postcode, $\log s$ institutions per postcode, $s/2$ restaurants per postcode, and one record in Demographics and Transport per postcode. Thus, the size of the join result is $\mathcal{O}(s^3 \log s)$. The datasets have been generated for $s \in \{1, ..., 20\}$.

This set of datasets has been designed to analyze how data redundancy in the join result is exploited by various algorithms. The larger the scale factor s, the more redundant data is contained in the join result.

For the implementations that work on the factorizations, we use the most obvious f-tree that has the attribute `postcode` as a root and the other attributes of the 6 relations on disjoint root-leaf paths. Caching of factorized representations is disabled.

The task is to compute a depth 5 regression tree using the house price as a target and all other attributes as features.

5.4 Accuracy

We implemented the very same CART algorithm as MADlib (see section 4.1), but over factorized joins instead of flat ones.

Thus, all four implementations yield qualitatively very similar results, i.e. the resulting regression tree has a similar training error. The training error for Mat-1, Mat-2, and Non-Mat is exactly the same and difference of the training errors between our implementations and MADlib is not larger than 5%. Since the accuracies of the implementations are very similar, we only compare the wall-clock times that the implementations need to compute the regression trees.

5.5 Results

5.5.1 Size of the join result vs. size of the factorization

The graph depicted in Figure 5.1 compares the size of the flat join result (number of tuples times columns) and the size of the factorized representation (in number of value nodes in the factorization tree). Note that the scale is logarithmic. For $s = 18$, the factorized representation is ca. three orders of magnitude smaller than the flat join.

Even larger is the difference of the time to compute the flat join and the time taken in computing the factorized representation. For $s = 18$, it takes almost 17 minutes to write the flat join result to disk (it does not fit in memory) whereas the factorized representation can be computed ca. 1.3 seconds.

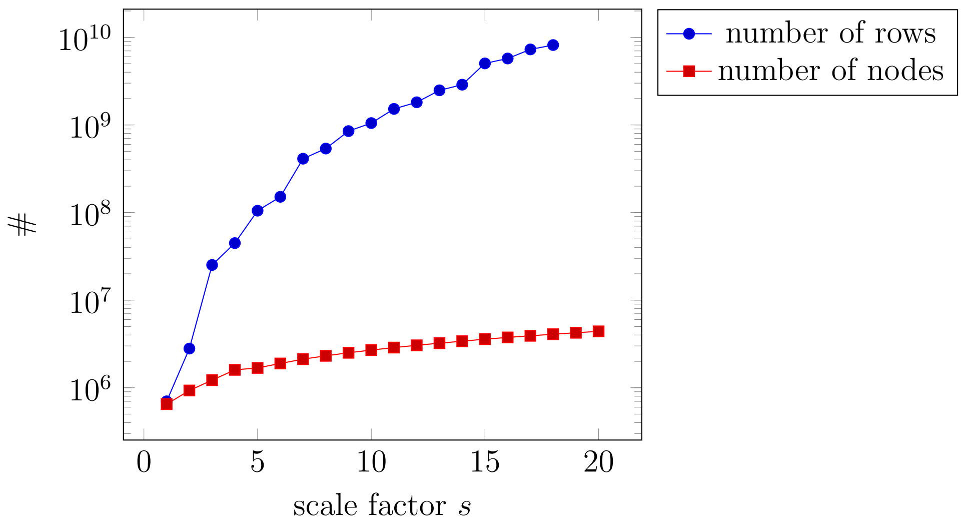


Figure 5.1: Size of the flat join result (number of tuples times columns) and the size of the factorized representation (in number of value nodes in the factorization tree) for $s \in \{1, ..., 20\}$. The scale is logarithmic. Calculating the flat join timed out for $s = 19$ and $s = 20$.

5.5.2 Total wall-clock time: MADlib vs. our implementations

Figure 5.2 shows the wall-clock time taken by the four implementations for $s \in \{1, ..., 20\}$. We can see that for $s = 7$, MADlib takes almost 20 minutes to complete the training task. We did not run MADlib on datasets generated by larger s values for obvious reasons. The $s = 20$ join result contains 400 million tuples, whereas for

$s = 7$ it only contains about 15 million. Still, the factorized implementations are faster at processing the $s = 20$ case than MATlib is at processing the $s = 7$ case.

The fact that MADlib is faster than our implementations for $s \leq 3$ is due to memory locality issues. MADlib needs to iterate over a list of values once, which is a very cache-friendly operation, whereas our implementations are building complicated tree structures that are iterated using pointers. For smaller s values, the data reduncancy that is exploited by our implementations is not enough to make up for the memory locality issues.

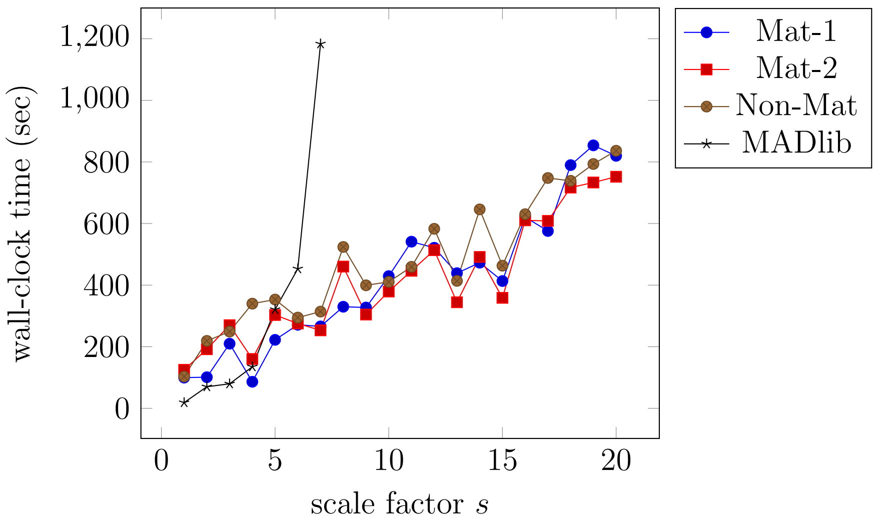


Figure 5.2: Wall-clock time plot of the four implementations for $s \in \{1, ..., 20\}$.

This comes from the fact that the time complexity of the algorithm is linear in the size of the factorization (up to a log-factor). For the specific f-tree chosen, the size of the factorization grows linearly in s; in particular there are

$$25000 \cdot (s \cdot C_H + s \cdot C_S + \log s \cdot C_I + s/2 \cdot C_R + C_D + C_T)$$

value nodes, where C_i are constants. C_H is the number of attributes in relation *House*, similarly for the other C_i. The factorization is depicted schematically in Figure 5.3.

An example regression tree computed by our software is depicted in Figure 5.4. The data has been generated randomly, so the result is not sensible in any practical sense.

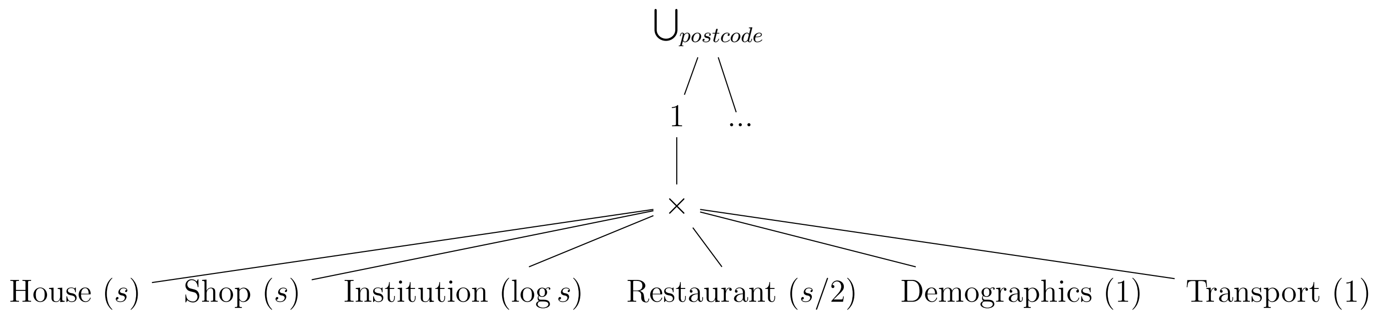


Figure 5.3: Schematic diagram of the factorization of the Housing dataset. The number in parentheses represents the number of values in the next union node.

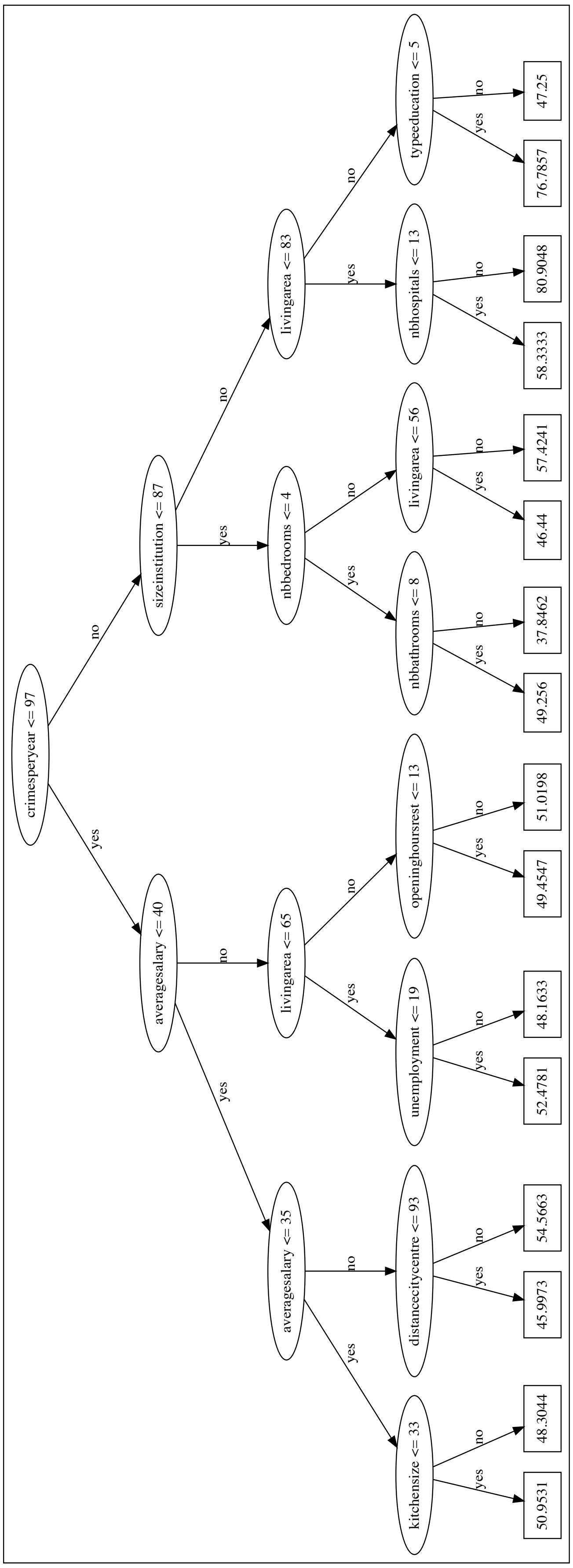


Figure 5.4: An example regression tree computed by Mat-1. This tree was trained with a maximum depth of 4 to make it fit on one page. All experiments were trained with maximum depth of 5.

5.5.3 Total wall-clock time: comparison of our implementations

Figure 5.5 shows the same graph without the MADlib times to present the differences between our implementations in more detail. We can see how the order and the bundling of the aggregations matters.

For every new node in the regression tree the algorithm needs to calculate three aggregates for each attribute (26), and for each possible split point (100). Thus, $3 \cdot 26 \cdot 100 = 7800$ aggregates are necessary per node.

Mat-1 can benefit a lot from the pre-aggregations; all conditions for one iteration over the factorized representation only restrict a few attributes. Thus, if a subtree of the factorization does not contain any of these attributes we can use the pre-aggregated values. However, Mat-1 needs the most passes over the factorization (26 per regression tree node).

Mat-2 needs to calculate aggregates for conditions dependent on every attribute in each pass, and so pre-aggregation does not help in this case. Since it aggregates for the 26 attributes at once, we only need one pass over the factorization per regression tree node.

Non-Mat computes a whole level of regression tree nodes per pass over the data. Note that level k has 2^k regression tree nodes, counting from $k = 0$. Thus, it calculates $7800 \cdot 2^k$ aggregates per pass. Recall that this algorithm does not materialize the factorized representation. This creates additional overhead for computing the join whilst calculating the aggregates, but it decreases the memory footprint of the application significantly. For very large datasets this is the only algorithm that can be feasibly executed without running out of memory.

Depending on the data and the resulting regression tree, Mat-1 can be faster than Mat-2 or Mat-2 can be faster than Mat-1 (see Figure 5.5). Non-Mat is never the fastest solution. However, its running time is very similar to Mat-1 and Mat-2 and it has the advantage that it uses a lot less memory. Thus, it is the only feasible choice for large datasets.

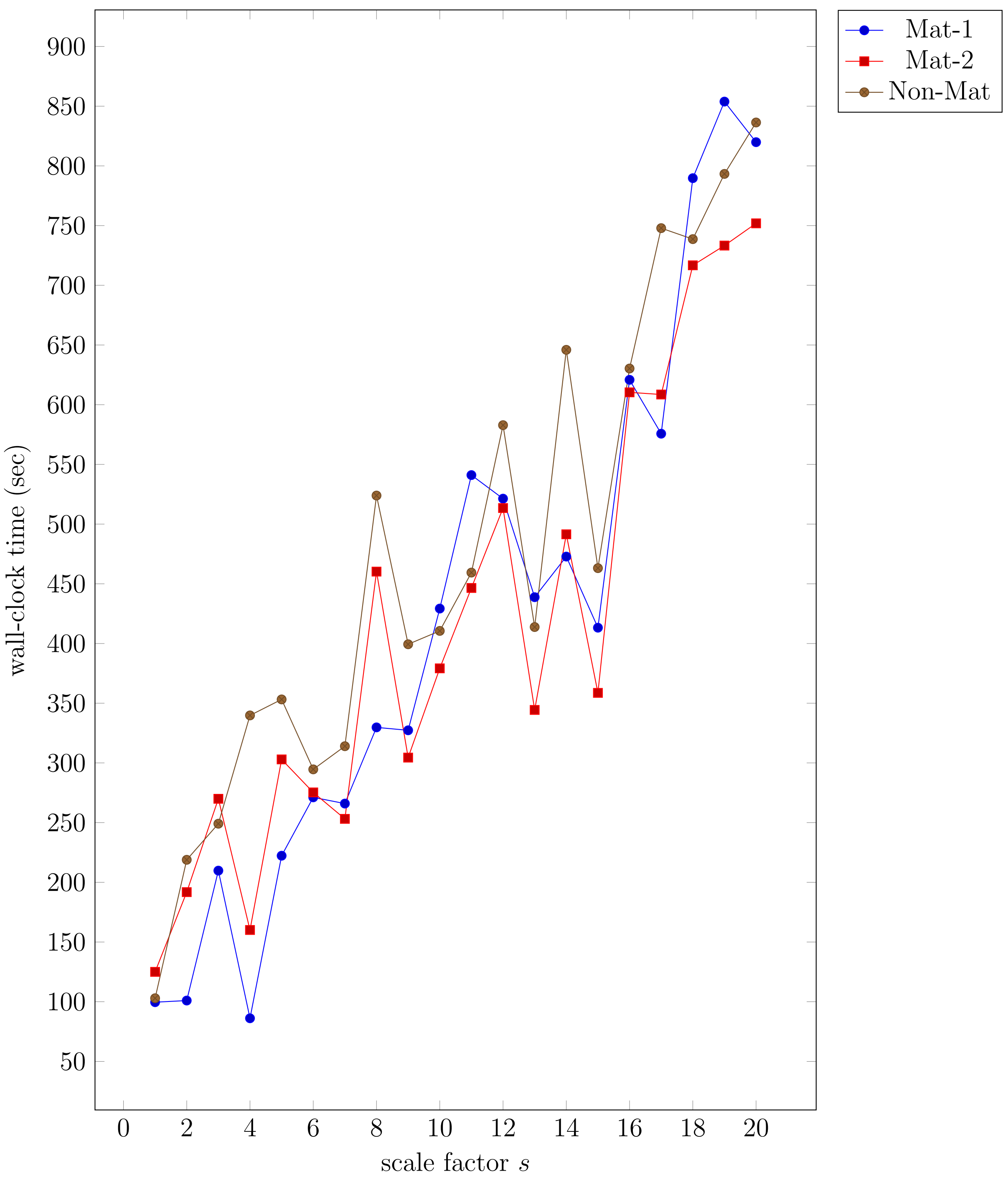


Figure 5.5: Running time plot of the three implementations based on factorization for $s \in \{1, ..., 20\}$.

5.5.4 Wall-clock time per regression tree level

Figure 5.6 shows the wall-clock time taken to compute each level of the regression tree for $s = 7$ (other s-values give the same qualitative results). Again, we compare the four implementations: Mat-1, Mat-2, Non-Mat, and MADlib. One can see that, even though level 4 contains 16 nodes and level 0 just one node, the time MADlib takes for each level is approximately constant.

This happend because the flat join result does not fit in memory and has to be read from disk multiple times. Thus, the iteration over the flat join takes up the majority of the time and outweighs the aggregation operations.

On the other hand, the time our implementations takes increases for each level. This means our implementations can be slower than MADlib for very deep trees. In practice, however, one usually trains shallower trees and builds a model based on an ensemble of independently trained trees (see section 2.1.2). For boosting based algorithms one sometimes even uses decision stumps, i.e. decision trees of depth one [6].

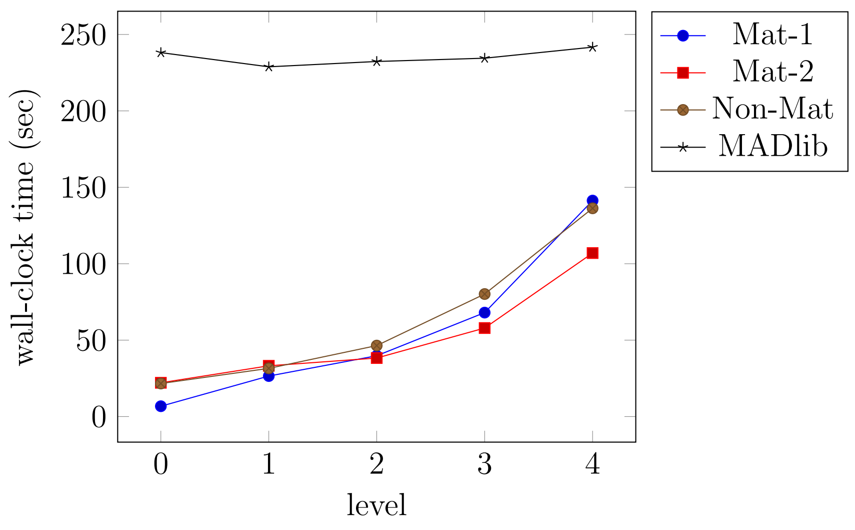


Figure 5.6: Running time of the four implementations for the five levels of the regression tree for $s = 7$.

5.5.5 Wall-clock time per node

Another interesting graph is shown in Figure 5.7. It depicts the wall-clock time per node for different levels of the regression tree, again for $s = 7$ (other s-values give qualitatively the same results). This specific graph is the result from the Mat-1 implementation, but the Mat-2 and Non-Mat graphs look very similar. Since every level of the regression tree contains multiple nodes, we plotted the mean of the wall-clock times, and the error bars show one standard deviation.

We can see that the times for nodes on the same level can differ significantly. For example, the computations for a level 4 node can take 1 second or almost 16 seconds.

This time depends on the f-tree positions of the attributes that are restricted by the conditions leading to the node. If all these attributes have a low depth in the f-tree, the pre-aggreation values can be used to prune a lot of the computation. We only need to explore a subtree of the factorization if it contains an attribute that is restricted by any of the conditions. Thus, if we know which attributes might be important for the regression tree then we can place them at low-depth positions in the f-tree. On the other hand, if the important attributes are very deep in the f-tree, the performance of the implementations based on factorization will perform poorly.

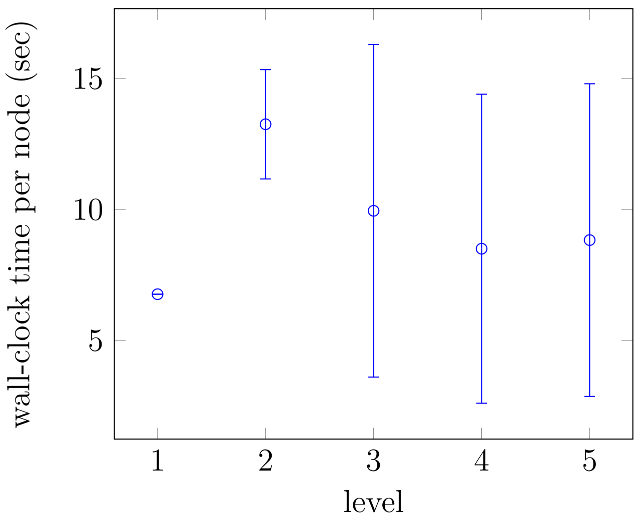


Figure 5.7: Running time per node for the five levels of the regression tree. The times plotted are from the Mat-1 implementation with $s = 7$. The graph plots the mean of the times of all nodes for each level, and the error bars show one standard deviation.

5.5.6 Exporting the data from PostgreSQL and loading it into R

R [28] is a programming language that is very popular amongst scientists and is used for many statistical computing tasks. It does not work directly on a database. Thus, we first have to compute the flat join in our relational database, then export the data to a file, then import the data into R, and finally we can learn the regression tree. For the learning task we use the package `rpart` [3]. We cannot compare MADlib and our implementation directly since the learning procedure implemented in R is different form the other approaches. This section aims to give an overview of the time needed to export the data from the database and import it into another system.

[3]`https://cran.r-project.org/web/packages/rpart/index.html`

Figure 5.8 shows the wall-clock time for exporting the data from PostgreSQL, importing it into R, and learning a regression tree using the rpart package. In comparison we plot the total time for learning a decision tree with Mat-1. We can see that for $s = 7$ the time to export and import the data already exceeds the time that Mat-1 needs to learn the full regression tree.

Even though we cannot compare directly the regression tree learning with rpart and our implementation it is clear that our approach is superior. Before the data is ready to be processed by the R system, our implementation has already learned a full regression tree.

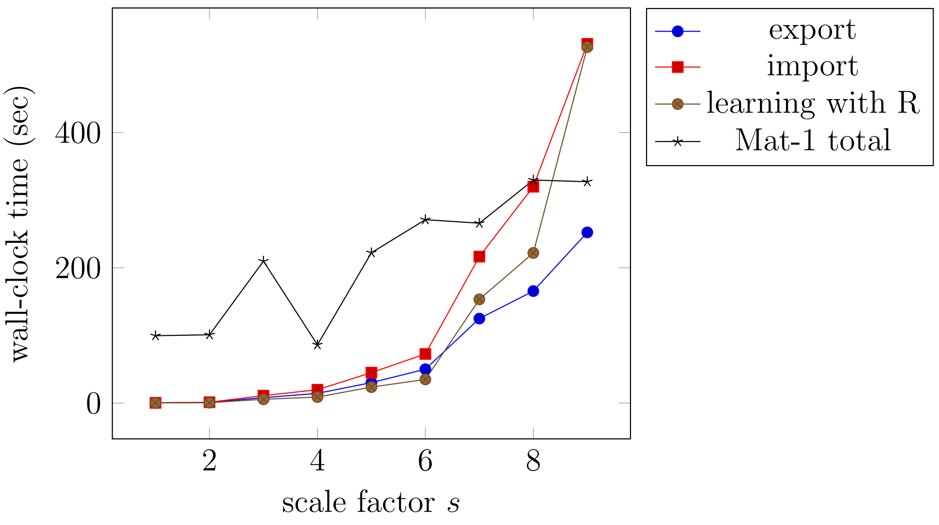


Figure 5.8: Wall-clock time for exporting the data from PostgreSQL, importing it into R, and learning a regression tree using the rpart package. In comparison we plot the total time for learning a decision tree with Mat-1. The times are shown for $s \in \{1, ..., 9\}$; the rpart package timed out for larger s values.

Chapter 6

Conclusions and future work

6.1 Conclusions

In this thesis we proposed a new method of learning decision trees over arbitrary joins of input relations. It outperformed a current state-of-the-art decision tree learning library (MADlib) that works directly on relational databases, on join results with a large degree of redundancy. Furthermore, we showed that even exporting the data from a database and loading it to another system can take more time than learning a full regression tree using our implementation. The superiority of our approach was shown theoretically by comparing the complexities of the algorithms and empirically on an artifically generated dataset for regression trees.

By rewriting the objective function of regression trees, we saw that it is only necessary to calculate three aggregates for various split conditions. These aggregates can be computed efficiently over joins by exploiting how these aggregates propagate over a factorized representation of the join. Similarly, for classification trees.

We then demonstrated three different implementations for learning regression trees; two of them materialized the factorized representation and kept it in memory for the multiple aggregations over this structure. They differ in the way aggregates are batched. These implementations are infeasible for very large datasets due to memory constraints. The third implementation does not materialize the join result. Instead it recomputes the join once for every level of the regression tree. This implementation is slighly slower than the others but it can be applied to arbitrarily large datasets.

The effectiveness of our implementations depends heavily on redundancies in the join result, and it should be investigated how well this approach works on real-world datasets. Furthermore, the work only focuses on regression trees. In theory, the results shown empirically should be observed in classification trees as well. This has to be analyzed by implementing the algorithms and performing benchmarks.

After learning least-squares regression models [19], this is the second machine learning model that was analyzed to exploit factorized joins. Applying this idea to other machine learning algorithms appears to be very promising. Moreover, there are various ways in which one may extend this work, and we shall discuss such methods briefly in the final section.

6.2 Future work

6.2.1 Improve memory efficiency

There are various ways to decrease the memory footprint of the implementation. For example, the *sum* and *sum_squares* aggregates are non-zero only on the paths from the target attribute to the root in the f-tree. Thus, we only need to store and propagate *count* aggregates for all the other nodes. By making sure that the target attribute has low depth in the f-tree, we can decrease the memory used for aggregations by factor of almost 3.

Furthermore, we can eliminate certain conditions completely upon calculating their count to be 0. For example, the condition $A < a$ can restrict the sample such that all B-values are in the range $[0, 10]$. Hence, we can eliminate all conditions of the form $B < b$ for $b \geq 10$ in this regression tree branch.

6.2.2 Avoid more recomputations

The current implementation only pre-computes non-restricted aggregates for all union nodes of the factorization tree. If enough memory is available, one can cache many more aggregates, especially the ones corresponding to conditions of the first nodes of the regression tree. Ideally, the implementation would get a maximum memory size as an input and reuse as many aggregates as possible within the memory constraint.

6.2.3 Parallelization and distribution

One could also extend the implementation to work on multiple CPU cores, and ultimately multiple machines, as the aggregation problem is inherently concurrent; every subtree of the factorizaed representation can be treated independently. Moreover, in practice one usually trains an ensemble of trees (see section 2.1.2). Thus, one could train each in parallel.

Random forests use a different set of attributes for each tree. Hence, using different f-trees for each regression tree might be sensible.

6.2.4 Adapting f-trees

As mentioned previously, the order of the attributes in the f-tree can influence the running time significantly. In the non-materializing algorithm (Non-Mat) one could reorder the f-tree before each join computation, depending on the current split conditions of the regression tree. This improvement will have a significant impact on the execution time, especially if the f-tree has long paths that can easily be reordered (like the Housing dataset analyzed previously).

6.2.5 Verify results using more datasets

So far we have only looked at the performance of Mat-1, Mat-2, and Non-Mat for different scale factors of the artificially generated Housing dataset. This experiment was carried out to highlight a dependence between the running time of our implementations and the redundancy in the join result.

For more practical results, one should use the algorithm on various real-world datasets to see what redundancies are appear in their joins.

Bibliography

[1] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.

[2] Yael Ben-Haim and Elad Tom-Tov. A streaming parallel decision tree algorithm. *Journal of Machine Learning Research*, 11(Feb):849–872, 2010.

[3] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

[4] Antonio Criminisi, Jamie Shotton, Ender Konukoglu, et al. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends® in Computer Graphics and Vision*, 7(2–3):81–227, 2012.

[5] Koudou Toussaint Dago, Rémy Luthringer, Régis Lengellé, Gérard Rinaudo, and Jean-Paul Macher. Statistical decision tree: A tool for studying pharmaco-eeg effects of cns-active drugs. *Neuropsychobiology*, 29(2):91–96, 1994.

[6] Yoav Freund, Robert Schapire, and N Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[7] Erico Guizzo. How googles self-driving car works. *IEEE Spectrum Online, October*, 18, 2011.

[8] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[9] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N

Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[10] Xiaohua Hu and Nick Cercone. Learning in relational databases: a rough set approach. *Computational intelligence*, 11(2):323–338, 1995.

[11] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 137–152. ACM, 2015.

[12] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[13] Keki B Irani, Jie Cheng, Usama M Fayyad, and Zhaogang Qian. Applying machine learning to semiconductor manufacturing. *iEEE Expert*, 8(1):41–47, 1993.

[14] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[15] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[16] R Menich and N Vasiloglou. The future of logicblox machine learning. *LogicBlox User Days*, 2013.

[17] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.

[19] Dan Olteanu and Maximilian Schleich. F: Regression models over factorized views. *Proceedings of the VLDB Endowment*, 9(13):1573–1576, 2016.

[20] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):2, 2015.

[21] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael Cafarella, Michael Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 283–284. ACM, 2015.

[22] Steffen Rendle. Scaling factorization machines to relational data. In *Proceedings of the VLDB Endowment*, volume 6, pages 337–348. VLDB Endowment, 2013.

[23] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[24] Steven Salzberg, Rupali Chandar, Holland Ford, Sreerama K Murthy, and Richard White. Decision trees for automated identification of cosmic-ray hits in hubble space telescope images. *Publications of the Astronomical Society of the Pacific*, 107(709):279, 1995.

[25] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.

[26] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[27] Lilly Spirkovska. Three-dimensional object recognition using similar triangles and decision trees. *Pattern Recognition*, 26(5):727–732, 1993.

[28] R Core Team. R: A language and environment for statistical computing. r foundation for statistical computing, vienna, austria. 2013, 2014.

[29] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[30] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean.

Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.