Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik
Database Systems Research Group

Bachelor Thesis:

# Exploring Ways to Evaluate Functional Style User-Defined Functions in Parallel

## An Exploration of Naive Parallelization in and around Databases.

Tim Fischer

*27.08.2020*

**Reviewers:**

Christian Duta
Research Assistant
Database Systems Research Group

Torsten Grust
Professor
Database Systems Research Group

# ABSTRACT

When analyzing recursive functions it is common to look at the function calls by building a call graph for an example input. By doing so a lot of the time we face the fact that we should be able to parallelize those parts of said call graph, which are mostly independent of each other. But some languages and systems make this process harder for us than it should be. One such system is the DBMS PostgreSQL, even though it allows for recursive UDFs they are poorly optimized and only viable for inputs yielding minimal amounts of recursive steps. To parallelize this directly is a chore in and of itself, but we can make use of a method introduced by Christian Duta and Torsten Grust in their 2020 paper “Functional-Style SQL UDFs With a Capital 'F'" [Duta20]. They propose a method to translate a given '*functional-style*' UDF into a query that first builds the call graph and then walks said call graph to calculate the result. Injecting parallelization into this two phase approach is what this work focuses on. We propose a simple method of extending the two phases such that the building of the call graph also partitions said call graph into disjointed, or at least mostly disjointed, sub-problems and that the evaluation uses said information to actually run these sub-problems in parallel. We found during our testing, that there can be major benefits for certain problems, but also that not all recursive problems lend themselves to being parallelized. We found that in general the more tree-like the call graph is, the more benefits can be gained from parallelization.

# CONTENTS

# 1

# INTRODUCTION

There are a lot of common and highly studied methods to speed up computation. Some rely on simply choosing a more intelligent way of storing and using intermediate computational results. Others go a step further and make clever use of some underlying properties the target problem may have. But one that comes to mind fairly naturally is simply doing multiple things at once, in other words simply process some parts in parallel. Though this may seem trivial at first, it is far from it. Proper parallelization requires a lot of work, doesn't generalize easily and more often then not is simply not worth the overhead required to do it properly. Putting that aside for a moment though, parallelization can bring massive performance benefits when it is optimized properly for the target problems. These benefits are most pronounced when the given input to a problem yields a massive callgraph, i.e. requires a large amount of computation to be solved. One place where algorithms often encounter massive problem sizes is in the world of databases, where an algorithm may at times be required to run over tens of thousands of gigabytes of input data.

This thesis will study the initial benefits of parallelization in and around databases through the use of call graph analysis and parallelized sub-problem evaluation. Doing so requires access to some notion of a call graph and a complementary method of evaluating one as well. For this requirement, in combination with our target implementation domain, there exists a novel method introduced by Christian Duta and Thorsten Grust in their 2020 paper on “Functional-Style SQL UDFs With a Capital 'F'" [Duta20]. They propose a method of transforming recursive SQL UDFs into a single recursive query, that benefits from all the query plan optimization already present in RDBMSs. Their resulting queries operate through a two phased approach, in which they first build the call graph and then evaluate it through traversal techniques that integrate common optimization patterns such as call-sharing. The main reason they deemed this work as necessary, is that modern RDBMSs which implement recursive UDFs do so with barely any optimization.

## 1.1 BASIC PREMISE

The basic premise this thesis builds upon is that we can accommodate parallelization by extending this two phase approach through a partitioning step, which divides the call graph into one rootgraph and multiple parallelizable subgraphs containing no external dependencies. The evaluation step is

extended to use this partitioning to evaluate the subgraphs in parallel.

The remainder of this thesis will focus on a the partitioning of DAGs, multiple options for parallel sub graph evaluation, analyzing the benefits and drawbacks of parallelizing the workload for different problems and discussing the overall impact and possible improvements.

# 2

# PARALLELIZATION

It is helpful to take a more detailed look at a general overview of what this method tries to achieve. The core idea is that we can take a callgraph and divide it into a so called root graph and multiple subgraphs:

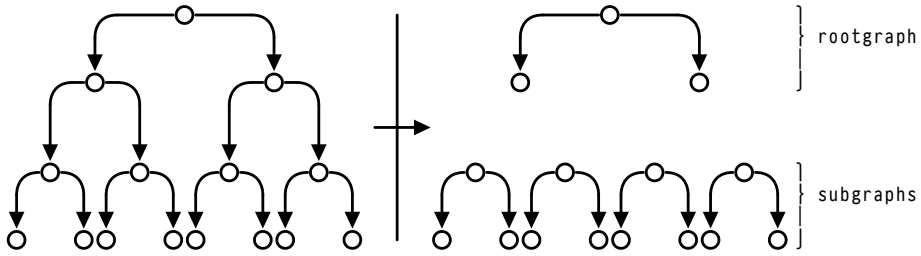


**Figure 1:** *Partitioning example*

Once we have this we can evaluate the problem by first solving the subgraphs in parallel and then stitch their results and rootgraph together for a final evaluation step:

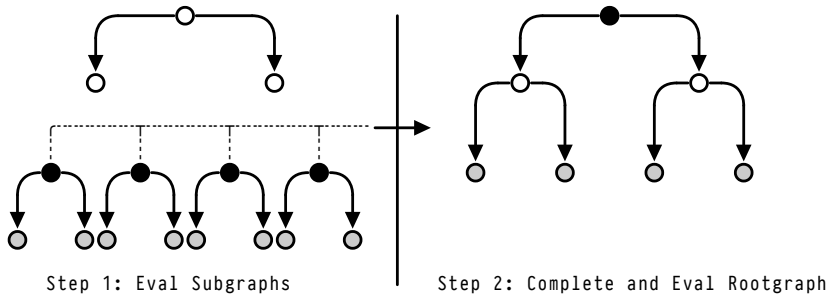


**Figure 2:** *Evaluation example*

This method doesn't allow for chains of dependent subproblems. Including this could be a potiential

area for future improvements, as large subgraphs could also be parallelized.

## 2.1 PARTITION BOUNDARY

Partitioning a callgraph is far from trivial, as callgraphs are DAGs and partitioning those is NP-hard, as Feldmann and Foschini found out [Feldmann12]. Furthermore it turns out that if we want to find a balanced partitioning, unless $P = NP$, we will have major difficulties finding an efficient alogrithm, as Andreev and Räcke pointed out that this special form is even NP-complete [Andreev04]. But the type of "partitioning" we are targeting here is very specific and allows us to perform trivial linear time graph traversals that are informed by some simple heuristics.

In layman's terms, what we are looking for can be defined as follows:

> Chopping up a call graph into a root graph and multiple, as loosely connected as possible, sub graphs.

In more concrete terms this can be described as finding a *partition boundary* in the call graph. This boundary can be further defined as a cut $C$ in a DAG $G = (V, T)$ such that the $C$ splits $G$ into $G_i = (V_i, E_i),\ i \in [1, \ldots, n]$ subgraphs with the following two properties:

1. $\forall i > 1 : V_1 \cap V_i = \emptyset$, and
2. $E / (\bigcup_{i \in [1,\ldots,n]} E_i) = C$.

These two requirements can be read as:

1. $G_1$ is the "rootgraph", i.e. the part of the call graph between the initial call, aka. the "root", and the partition boundary. And this sub graph must be completely disjointed from all other subgraphs.
2. The cut contains all the edges that aren't in any subgraph, i.e. the edges that we wish to construct in parallel later on.

### 2.1.1 FINDING PARTITION BOUNDARIES IN POLYTREES

The core idea behind defining this *partition boundary* in such a way is that it can be easily determined using trivial graph traversals such as depth-first and breadth-first traversal. These graph traversals can easily be guided through the use of simple heuristics. We can demonstrate how this guidance can be implemented through the use of some simple python code. Let's have a look at a very simple case were we assume the input is a Polytree in adjacency-lists format and we do not prematurely terminate the traversal:

```
def subtree(
    source: T,
    tree: Dict[T, Set[T]],
) -> Dict[T, Set[T]]:
    stack = [(None, source)]                          # < Stack contains edges and
    sub_tree: Dict[T, Set[T]] = defaultdict(set)      #   starts with a pseudo edge
                                                      #   to the root node.
    while len(stack) > 0:
        parent, current = stack.pop()
        stack.extend(
            (current, child)
            for child in tree[current]
            if child in tree
        )
        if parent is not None:                        # < Skip the root node.
            sub_tree[parent].add(current)

```

```
    return sub_tree
```

**Listing 1:** *Implementation of BF-traversal on Polytrees*

The simplest heuristic in graph traversals is only allowing for nodes with a maximum depth, i.e. distance to the root node. To visualize this let's take a look at an example:

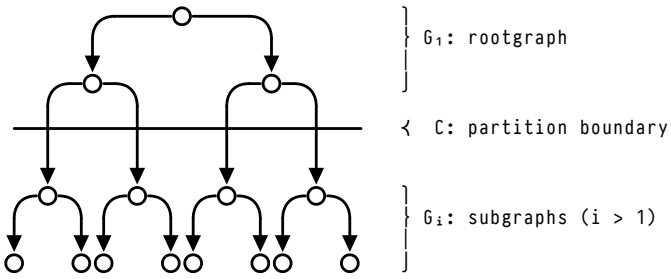


**Figure 3:** *Partitioning cut in a Polytree with `max-depth(2)`*

We can implement this into the BF-traversal from before by simply terminating the traversal once we've reached a given maximal depth:

```python
def subtree_max_depth(
    source: T,
    tree: Dict[T, Set[T]],
    max_depth: Union[int, float]
) -> Dict[T, Set[T]]:
    stack = [(None, source, 1)]
    sub_tree: Dict[T, Set[T]] = defaultdict(set)

    while len(stack) > 0:
        parent, current, depth = stack.pop()
        if depth <= max_depth:                          # < Cut-off nodes that exceed
            stack.extend(                               #   maximum depth.
                (current, child, depth + 1)
                for child in tree[current]
                if child in tree
            )
            if parent is not None:
                sub_tree[parent].add(current)

    return sub_tree
```

**Listing 2:** *BF-traversal with `max-depth` heuristic*

To calculate the *partition boundary*, or rather the subgraphs $G_i$, we need to first traverse the given polytree until we reach a given maximal depth using `subtree_max_depth` (Listing 2) to find our rootgraph $G_1$. Afterwards we can make use of `subtree` (Listing 1) to find all the remaining subgraphs $G_i; i > 1$:

```python
def partition(
    tree: Dict[T, Set[T]],
    max_depth: Union[int, float]
) -> List[Dict[T, Set[T]]]:
    root = list(gather_roots(tree))[0]                  # < This assumes given tree could
                                                        #   actually be a forest.

    root_tree = subtree_max_dist(root, tree, max_depth)    # < Calculate the root tree based
                                                           #   on the max-depth heuristic.
    return [root_tree] + [
        subtree(child, tree)                            # < Calculate the sub-tree
        for leaf in gather_leaves(root_tree)            #   attached to all nodes that
        for child in tree[leaf]                         #   are direct children of the
    ]                                                   #   leaves of the root-tree.
```

**Listing 3:** *Partitioning using constrained BF-traversal*

This algorithm is effectively a simple BF-traversal with a short stop in between. We can easily see that the time complexity must be linear, i.e. $\mathcal{O}(n)$.

### 2.1.2 EXTENSION TO DAGS

The prior section used a example heuristic on *Polytrees*. However, our callgraphs will be proper *DAGs*. Extending the example algorithms to handle DAGs as well can be done by simply using the BF-traversal for graphs, i.e. adding a "*visitation check*":

```
def subgraph(
    source: T,
    graph: Dict[T, Set[T]],
) -> Dict[T, Set[T]]:
    stack = [(None, source)]
    sub_graph: Dict[T, Set[T]] = defaultdict(set)
    seen: Set[T] = set()                                # < Visitation data.

    while len(stack) > 0:
        parent, current = stack.pop()

        if current not in seen:                         # < Only follow edges with
          stack.extend(                                 #   unvisited target nodes.
              (current, child)
              for child in graph[current]
              if child in graph
          )
          if parent is not None:
              sub_graph[parent].add(current)
          seen |= current                               # < Mark the current node as
                                                        #   visited.
    return sub_graph
```

**Listing 4:** *Implementation of BFS on DAGs*

### 2.1.3 EDGE CASES

The first edge case that this method produces is when two or more callgraphs overlap directly after the *partition boundary*. For example:

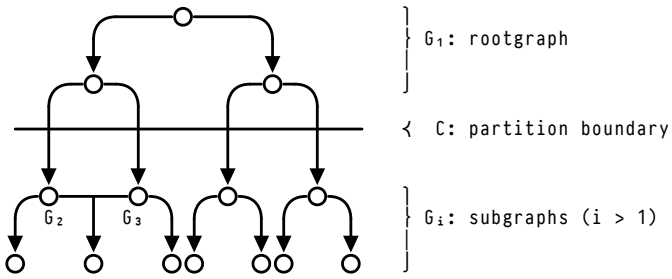


**Figure 4:** *Overlapping subgraphs $G_2$ and $G_3$*

In this case the two subgraphs $G_2$ and $G_3$ are either fully identical or one is completely covered by the other. These two possibilities and their impact on performance will be discussed in more detail later in this thesis.

Another case that we need to account for is an overlap immediately before the *partition boundary*. This causes the algorithm to produce two (or more) identical subgraphs. Though the demonstrated python implementation does not necessarily suffer from this, it does require extra attention to detail during the implementation to ensure that such unnecessary work is avoided.

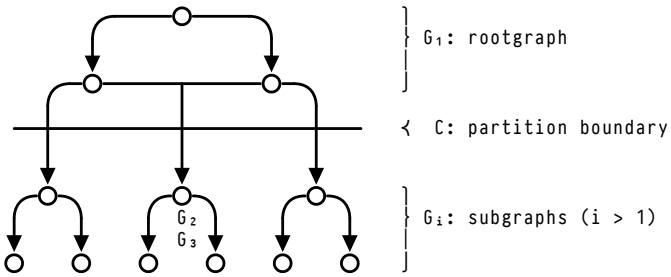


**Figure 5:** *Duplicate subgraph root nodes in $G_2$ and $G_3$*

## 2.2 EVALUATION

Once we have identified a proper partitioning boundary the actual evaluation becomes pretty easy. All we need to do is evaluate all the subgraphs in parallel and "prime" the rootgraph using those results. Afterwards we just need to evaluate the rootgraph to retrieve the total result. This process can be seen in Figure 2 at the beginning of this chapter.

# 3

# IMPLEMENTATION

## 3.1 GENERAL LAYOUT

During the adaptation of the original method some changes were required:

- The callgraph needs to be materialized to facilitate proper access between processes/threads/cursors.
- Instead of a single query that handles the two phases as common table expressions (CTE), said CTEs were extracted into their own functions. Leaving us with one for building the callgraph and one for evaluating it.

The general layout of the implementation consists of a few functions, these being:

- `build_callgraph`: building a callgraph.
- `_eval_graph`: the original callgraph evaluation method.
- `*_serial_eval`, `*_parallel_eval`: wrapping extensions to `_eval_graph`, where `*` describes the implementation method.

## 3.2 PARTITIONING

The original method builds the callgraph using a recursive Common Table Expression (CTE) that is implemented via a BF-traversal, so all we need to do is implement the heuristic as an additional terminal condition. Doing this only requires us to carry along a depth value during the recursive steps and check said value versus a given input parameter. To avoid this value influencing the `UNION` inside the recursive CTEs, we also needed to define a new datatype for this depth value that is basically just an integer that doesn't implement any ordering operators and yields a constant value for comparisons.

```
CREATE FUNCTION build_callgraph(args args, param param)
RETURNS TABLE (...) $$
  WITH RECURSIVE
    root_call_graph(...) AS ...,
    cut_leaves(...) AS ...,
    partitioned_call_graph(...) AS ...
```

```
7
8    SELECT ... FROM root_call_graph
9      UNION ALL
10   SELECT ... FROM cut_leaves
11     UNION ALL
12   SELECT ... FROM partitioned_call_graph;
13 $$ LANGUAGE SQL;
```

**Listing 5:** *General UDF layout for building callgraphs*

The CTEs in the wrapped query correlate to the following parts in listing 3:

1. `root_call_graph` — Line 8: `subtree_max_dist`
2. `cut_leaves` — Line 12+13: `gather_leaves`
3. `partitioned_call_graph` — Line 11: `subtree`

This implementation calculates all the subgraphs serially, i.e. not in parallel. One possible improvement would be to do said calculation in parallel as well, as we need to ensure "interoperability" between the calculation of individual subgraphs. We've forgone this improvement for this thesis but acknowledge that this could lead to a significant speed up in callgraph building times.

## 3.3 EVALUATION

The main reason for multiple implementations in the thesis is for comparing different approaches to solving this problem. For this we decided on the following:

1. First, we need some parallel implementation that runs within Postgres. The easiest way to do this is by using `dblink` to open multiple cursors and punt the work over to those.
2. And second, we need some parallel implementation that runs outside of Postgres to check if process boundaries have any major impact. As I am most fluent in python at the time of this thesis I decided on using that in conjunction with the `psycopg2` library to interface with a Postgres instance.

Both of these work by checking the callgraph table for all existing subgraph labels. They dispatch multiple workers to process each subgraph using `_eval_graph` and using those calculation results to complete the rootgraph. Afterwards they run `_eval_graph` once more on the rootgraph to complete the calculation.

### 3.3.1 PL/PGSQL + DBLINK

As stated before this implementation uses `dblink` to open multiple "remote" cursors and punt out the work over to those. These are cursors are not very "remote" as they are on the same Postgres instance, but from the perspective of this implementation they are treated as proper remote cursors. So a possible future improvement would be to setup up some kind of Postgres cluster and split up the workload between multiple machines. Though this "improvement" must contend with network delay and as such only is viable for very large callgraphs.

### 3.3.2 PYTHON 3 + PSYCOPG2

Python and parallelism are two things that are notorious for being hard to do get working properly together. However we don't need to do a lot of the work in python. The program only needs to coordinate and compose the calls of multiple UDFs inside our Postgres instance. For this we make use of the `psycopg2` library to open multiple serverside cursors on the instance and keep track of

computation results via `Future` objects spawned by a `concurrent.futures.ThreadPoolExecutor`. We can poll these futures for resolution via `concurrent.futures.as_completed` and process the resolved ones in the main thread. The use of threads does not negatively impact the performance and saves us some overhead versus subprocesses. Pythons GIL doesn't affect this implementation to heavily since all the threads spawned by the executor are doing, is effectively waiting for the computation in Postgres to complete. This results in the main thread holding on to the GIL during most of the execution time.

## 3.4 TEST PROBLEMS

### 3.4.1 DUMMY PROBLEM: "THRESHOLD"

The first experiment function is one we decided to dub "threshold" it doesn't calculating anything meaningful to our knowledge but it only produces callgraphs that are true Polytrees and as such is a wonderful candidate for analyzing the maximum effectiveness thanks to its fully disjointed subgraphs. We've defined it as follows:

$$threshold(i,j) = \begin{cases} 1 & \text{if } |i-j| < 2, \\ threshold\left(i, \frac{j-i}{2}\right) + threshold\left(\frac{j-i}{2}, j\right) & \text{else} \end{cases}$$

```
CREATE FUNCTION threshold(i int, j int) RETURNS int
AS $$
    SELECT CASE
      WHEN ABS(threshold.i - threshold.j) < 2 THEN 1
      ELSE (
        threshold(threshold.i, threshold.i + (threshold.j-threshold.i)/2) +
        threshold(threshold.i + (threshold.j-threshold.i)/2, threshold.j)
      )
    END;
$$ LANGUAGE SQL STABLE STRICT;
```

**Listing 6:** *Threshold as fsUDF*

### 3.4.2 BINOMIAL COEFFICIENT

For the second experiment we chose the standard recursive definition of the binomial coefficient, as this function yields callgraphs that are proper DAGs and isn't to complicated. This recursive definition is as follows:

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \vee k = n, \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{else} \end{cases}$$

```
CREATE FUNCTION bin_coef(n real, k real) RETURNS real
AS $$
  SELECT CASE
    WHEN k = 0 OR k = n THEN 1
    ELSE bin_coef(n - 1, k) + bin_coef(n - 1, k - 1)
  END;
$$ LANGUAGE SQL;
```

**Listing 7:** *Binomial Coefficient as fsUDF*

### 3.4.3 DYNAMIC TIME WARP

And last but not least we needed a function that is a bit more in-depth and inter-operates with persistent data inside the database, to make sure that this doesn't interfere with performance through postgres query planning. To cover this we opted to use the dynamic time warp function, which is the same one used as an example in the original paper by Duta and Grust [Duta20], as it reads data from two tables and is fairly simple mathematically:

$$dtw(i,j) = \begin{cases} 0 & \text{if } i = j = 0, \\ \infty & \text{if } i = 0 \vee j = 0, \\ |x_i - y_j| + \min \left\{ \begin{matrix} dtw(i-1, j-1) \\ dtw(i-1, j) \\ dtw(i, j-1) \end{matrix} \right\} & \text{else} \end{cases}$$

```
CREATE TABLE TX (
  pos serial PRIMARY KEY,
  v double precision
);

CREATE TABLE TY (
  pos serial PRIMARY KEY,
  v double precision
);

CREATE FUNCTION dtw(i int, j int)
RETURNS double precision AS
$$
  SELECT CASE
    WHEN i = 0 AND j = 0 THEN 0 :: double precision
    WHEN i = 0 OR  j = 0 THEN 'infinity' :: double precision
    ELSE (
      SELECT abs(X.v - Y.v) + LEAST(dtw(i - 1, j - 1), dtw(i - 1, j), dtw(i, j - 1))
      FROM TX AS X, TY AS Y
      WHERE (X.pos,Y.pos) = (i,j)
    )
  END;
$$ LANGUAGE SQL;
```

**Listing 8:** *DTW as fsUDF*

# 4

# RESULTS

## 4.1 METHODOLOGY

### 4.1.1 HARDWARE AND SOFTWARE

All results where captured on a `lenovo 4180AJ3 ThinkPad T420` with an `Intel i5-2520M (4) @ 3.2 GHz` processor and `7846 MiB` of RAM running `Manjaro Linux` using the version `5.4.47_rt28-1` Linux Kernel. To constrain the postgres process' access to system resource these experiments were run inside a docker container using the official `postgres:12.3-alpine` image configured with access to 3 CPU-cores and `5 GB` of memory. The python parts were also run inside the same container using an installation of `Python 3.8.3`. The evaluation of the experiments was done on the host machine, instead of the container, using an installation of `anaconda3-2020.02`.

### 4.1.2 SETUP

1. Between each call to the callgraph building function the callgraph table was cleared via `DELETE FROM call_graph` and all invalidated rows properly deleted from disk through `VACUUM FULL`.
2. The experiment coordination was handled via a python script running inside the container and the measurements were written to `stdout` and to disk using `tee`.
3. In every experiment the partitioning was run with a `max-depth` parameter of 1.
4. In every experiment we used the following input series to maximize the graph sizes and symmetries:

$$
\begin{aligned}
&\mathit{threshold}(1, j \cdot 10^{f}), && j \in \{1,2,5,7\}, \forall f \in [1,5] \\
&\mathit{binom}\left(i, \left\lfloor \frac{i}{2} \right\rfloor\right), && i \in [1,131] \\
&\mathit{dtw}(i,i), && i \in [1,50]
\end{aligned}
$$

5. `sql_baseline` refers to the original method from the paper by Duta and Grust [Duta20].

## 4.2 OVERALL TIME

The following plots plot the total evaluation time, including the callgraph building time, over the total amount of nodes in the callgraph. This was chosen instead of plotting the time over the inputs to ensure a more “readable” plot.

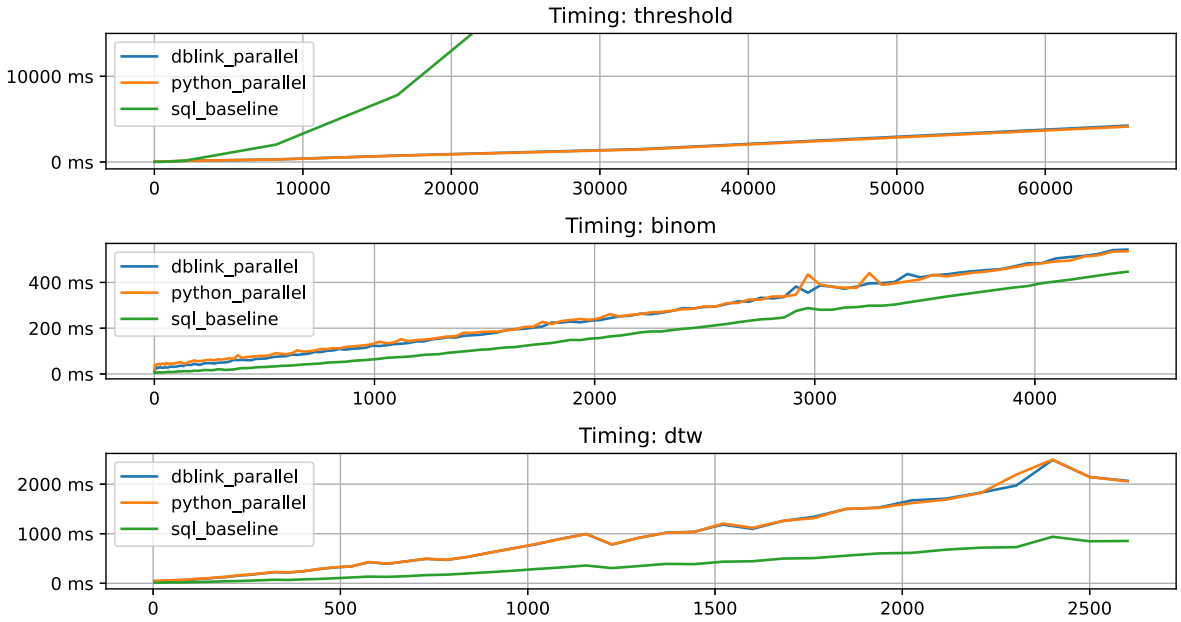


**Figure 6:** *Runtimes over Node amounts*

These plots show that our method can produce a performance increase, that being in `threshold`. And that there isn't much difference between our implementation when compared to the original method. But they also demonstrate that in the case of `binom` and `dtw` we loose more than we gain. After looking a bit deeper at were we lost all the benefits we found the following:

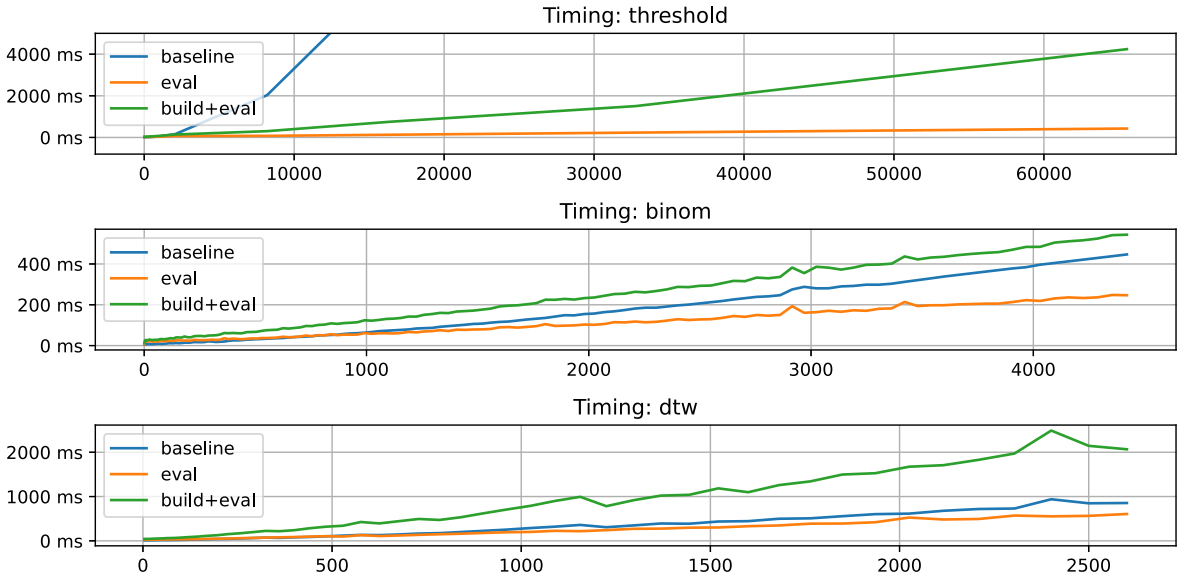


**Figure 7:** *Split overview over time influences in `dblink_parallel`*

This shows that most of the time is lost during the building/partitioning of the callgraph. To ensure the viability of this method the callgraph generation needs some major work. This could for example include the parallel generation of the subgraphs.

## 4.3 SYSTEM AND PARAMETER INFLUENCE

Another major factor on any parallelization method of course is the amount work one can actually do in parallel. Throughout all the experiments so far the docker container was able to make use of up to 3 CPU cores, but limiting this to just 2 CPU cores we see that even `threshold` doesn't show any performance benefits:

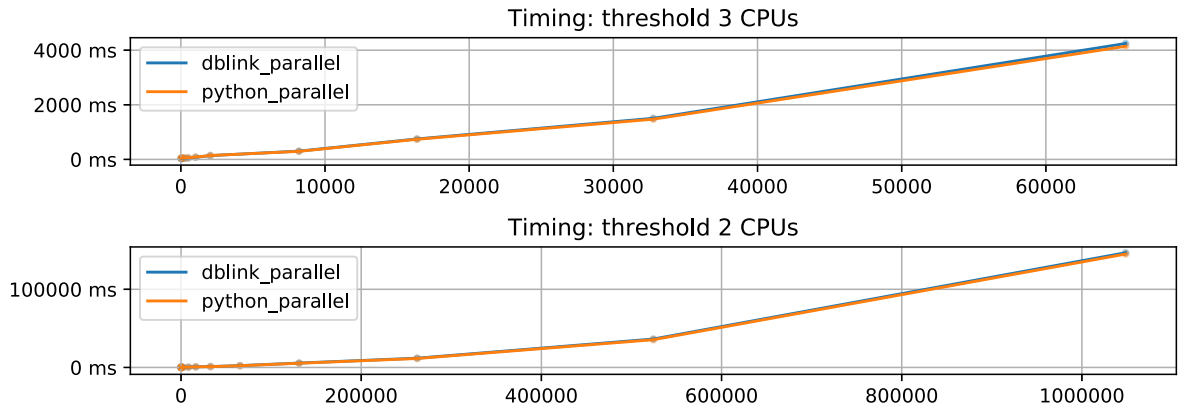


**Figure 8:** *Runtimes on 2 vs. 3 CPU cores*

## 4.4 CALLGRAPH COMPARISON

As stated before we see some stunted performance when it comes to the experiments that have overlapping subgraphs, but how large is this overlap.

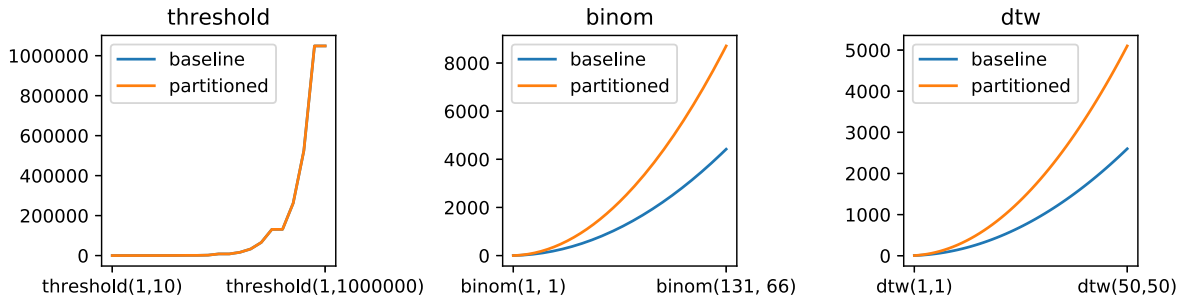


**Figure 9:** *Node amounts over inputs*

As is easily visible, threshold has zero difference between the baseline callgraph and the partitioned one, which is to be expected as we chose this dummy problem specifically for this reason. But the other two experiments show a significant difference, which can be attributed to us allowing the subgraphs to overlap. It is pretty obvious that processing a callgraph that is effectively double the size will take more time unless we throw massive amounts of resources at it.

Another way to inspect subgraph overlap is to look at it in terms of a heatmap where the overlap between each subgraph can be seen intuitively. When the heatmap “lights up all over like a christmas-tree” we know that there is to much overlap.

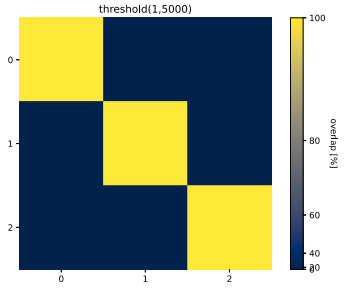


**Figure 10:** *Overlap heatmap of threshold on an average input*

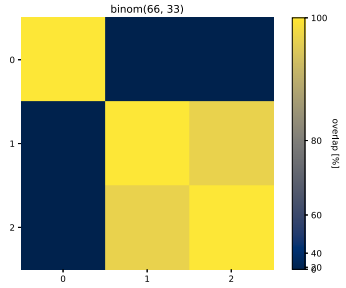


**Figure 11:** *Overlap heatmap of `binom` on an average input*

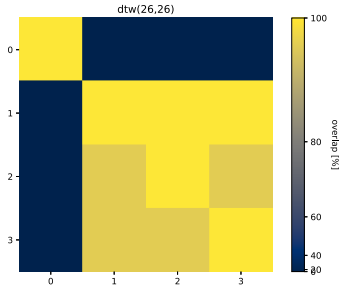


**Figure 12:** *Overlap heatmap of `dtw` on an average input*

These heatmaps show that `threshold` has zero overlap, which is to be expected thanks to the Polytree nature of its callgraphs. But the other two, `binom` and `dtw`, have partially overlapping subgraphs and fully occluded subgraphs respectively. In this method "overlap" indicates duplicate work. Though we knew this from the beginning, we naively assumed that small amounts of overlap wouldn't impact the performance to much. But the chosen experiments tend to produce very large amounts of overlap. For example the following is the callgraph for `dtw(2,2)`:

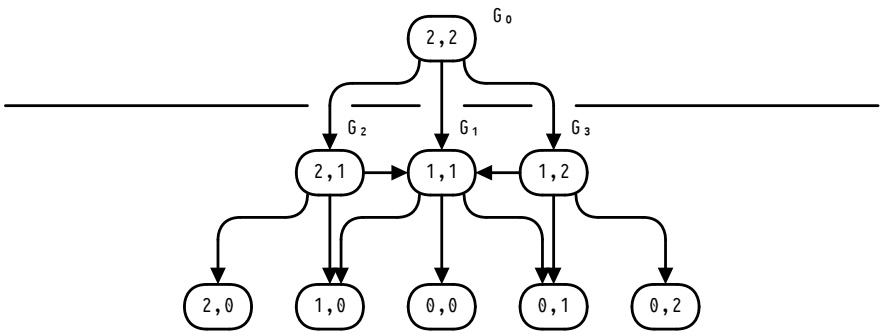


**Figure 13:** *`dtw(2,2)` partitioned to a depth of 1*

As we can see the the two outermost subgraphs $G_2$ and $G_3$ both contain $G_1$. We could eliminate this duplicate work by only evaluating subgraphs that are not completely contained in others. This would require multiple values from a subgraph evaluation, which is slightly outside of the scope of this thesis and is best left for future work.

# 5

# CONCLUSION

Proper parallelization can yield massive performance benefits. However, it needs to cover many edge cases and is heavily reliant on proper partitioning of the workload. As `threshold` demonstrated major benefits are possible, and easily reachable if the callgraph is a Polytree. In the cases of `dtw` and `binom` however we had major overlap in the subgraphs, and these benefits were nowhere to be found. There is a lot more optimization possibilities left in this method.

## 5.1 POSSIBLE IMPROVEMENTS

Some of the possible future improvements to this method could be:

1. Speeding up callgraph building time by ways of parallelization.
2. Balance subgraph sizes using a better heuristic.
3. Eliminating fully overlapped subgraphs by allowing multiple return values during the parallel evaluation.
4. Re-write as proper PSQL-Extension, for example in C.
5. Also parallelize subgraph evaluation.

# 6

# BIBLIOGRAPHY

[**Duta20**] Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA. ACM, New York, NY, USA, Pages 1273 – 1287. https://doi.org/10.1145/3318464.3389707

[**Andreev04**] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. June 2004, Pages 120–124. https://doi.org/10.1145/1007912.1007931

[**Feldmann12**] Andreas Emil Feldmann and Luca Foschini. 2012. Balanced partitions of trees and applications. In Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS'12). February 29 - March 13, 2012, Paris, France, Pages 100 – 111. https://doi.org/10.4230/LIPIcs.STACS.2012.100

# DECLARATION OF AUTHORSHIP

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This paper was not previously presented to another examination board and has not been published.

City, date and signature