Optimizing Runtime with Memoization for In-Memory Query Perturbation

Ilica Mahajan
ipm2111@columbia.edu
Columbia University

Sughosh Kaushik
svk2117@columbia.edu
Columbia University

ABSTRACT

Query perturbation methods offer journalists new ways of finding leads and fact-checking the veracity of existing claims that are made using data [12]. Existing perturbation APIs require extensive knowledge of SQL and Python to utilize, which most data journalists do not possess. Instead, they tend to use Pandas for data analysis. We introduce Pert-Q-Pan, a Pandas compatible API to conduct in-memory query perturbation. A simple interface allows users to specify their query and the parameter space they wish to search, which will be turned into a parallelizable execution plan behind the scenes. Since such searches are computationally expensive, we offer one main optimization technique not implemented in previous literature: memoization during processing. We discuss the tradeoffs of caching and multiprocess and benchmark our implementation against brute-force search techniques, and with or without the use of our caching methods.

ACM Reference Format:
Ilica Mahajan and Sughosh Kaushik. 2020. Optimizing Runtime with Memoization for In-Memory Query Perturbation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the field of computational journalism has emerged and offered the advances of computer science to push forward what is possible in the area of journalism. Research on databases is one such area of computer science where gains involving data management and various querying abilities can help journalists find leads and fact-check dubious claims [9].

In the current world where data is often used to justify real-world policy decisions, or evaluate the effectiveness of a candidate idea, it can be difficult to discern the quality of a statistical argument. At the same time, having the ability to explore a dataset to grow an understanding of how particular metrics change when the input variables are altered is integral for making better decisions, whether that be for distributed systems management or for evaluating policies' influence on different groups. For example, Simpson's paradox shows that at different aggregation levels, it is possible to arrive at opposing understandings of a dataset. So, it is worth exploring different aggregation levels and parameters to create a more holistic understanding of a dataset.

A similar problem exists when querying a database. It is possible to arrive at an opposite conclusion simply by querying a dataset differently, or, as is usually the case, by slightly varying the parameters in the query. For example, when analyzing a time-series dataset, depending on the points one decides to sample, you could arrive at very different conclusions.

Varying the parameters of a query is called query perturbation and has been proposed as a novel technique to fact-check claims [2, 6]. This exciting development offers new tools to journalists who have access to the underlying datasets used in making a claim. In order to be able to conduct existing query perturbation methods mentioned in [2], the authors developed a novel system, Perada, on top of SQL and Python, and implemented several optimizations to be able to run the system at scale.

At its heart, query perturbation is akin to running gridsearch to look for the optimal parameters when fitting machine learning models to datasets. But, as is noted by the existing literature, running such nested searches is prohibitively computationally expensive [2]. While a dataset may not be large, the search space of perturbations may be, depending on the desired ranges of parameters to search.

The main contributions of the existing literature on query perturbation is on the optimization such that running such perturbations becomes possible. The optimizations include parallelization, intelligent caching schemes for post-processing on the intermediate result set, and being strategic in not calculating unnecessary parts of the search space (pruning) [2]. The challenge lies in identifying and implementing automatic optimizations that are general enough to be useful for a large array of workloads and access patterns. Perada does make an attempt to automate many of its offered optimizations, but relies on the user to specify the best set of optimizations for their workload. That is, in order to take full advantage of these optimization gains, Perada's API requires the user to understand the offered optimizations.

Additionally, because Perada is built on SQL, requires that the data be in a SQL database and the user should be an experienced programmer. Additionally, running a Spark cluster is also necessary. These prove to be particularly complicated for those who are not well versed in highly technical solutions. While today's journalists are quickly adopting new skills and technologies, the majority of data journalists, such as those that graduate from Columbia Journalism School's data journalism classes, usually interact with in-memory datasets on a single computer, either through R or Pandas in Python. Thus, this powerful tool, query perturbation, is out of their reach.

The aforementioned reasons give us the motivation to research and extend the Pandas framework to support query perturbation

Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

in order to check the robustness of a claim or to discover new leads. We adapted Perada to construct a Pandas compatible API that will operate entirely in-memory, and circulate through a variety of parameter values for a query locally [2]. One of our main goals was to build an interface that is intuitive and simple for any working data journalist to be able to take advantage of. With the information provided by the user of the query template, the parameters to vary, and the intended search space, we divide the work, and distribute it amongst several Python processes in order to parallelize the workload.

Since this will still be computationally expensive and therefore time intensive, we introduce one main in-memory optimization to run query perturbation not discussed in the original Perada implementation: caching during main stage processing. This optimization is promising as when traversing ranges of the parameter space, while holding one parameter constant, it is possible to reuse smaller ranges of computations when varying a secondary parameter. We will test on generated datasets with between 10,000 to 500,000 rows, and 5 columns which mimics the size of some datasets that journalists work with, and vary two parameters. This basic design, inspired by a real-world dataset and journalism article that could have benefited from query perturbation, gives a good base case with which to explore multiprocessing and caching optimizations.

We discuss the trade offs of the proposed optimization. We also benchmark and understand the run times of our system in a variety of experiments where we vary the parallelization, and compare the runtime footprint of execution when toggling our cache optimization and our optimized system against a simple, single process, non-optimized grid-search implementation.

In the following section, we discuss the related work on query perturbation and caching strategies for queries. In Section 3, we outline our approach, and in section 4, we lay out further technical details. Section 5 discusses our experiments and findings, and finally, Section 6 concludes and offers some directions for future work on in-memory query perturbation systems.

2 RELATED WORK

2.1 Query Perturbation

The database field has a long history of research that is applicable to journalism, but for a long time none of these connections were made. More recently, database researchers brainstormed ways to contribute to journalism and started to draw these connections [9] [10] [12], building on work about parametric query optimization [3]. They devised a model whereby they argued that any claim made in journalism is equivalent to levying a query with a certain set of parameters on a database. So, to check how well a claim holds up to a variety of query parameters, an analyst would run query perturbation, shedding light on how sensitive the result is to the parameters. In actuality, however, such an idea was computationally infeasible on its own, and thus a segment of the community set out to build a system with enough improvements and optimizations on the naive idea of doing a parameter gridsearch. Because gridsearch suffers from the curse of dimensionality [1], this is especially pertinent with many query parameters. This work resulted in systems that could efficiently and intelligently understand how sensitive a query is to its parameters without a brute force search of the entire space. The advances they offer come in the form of caching optimizations, parallelization, and intelligently pruning aspects of the space that are unnecessary to search. On top of that, they then established metrics and thresholds for determining the strength and reasonableness of a claim [12]. However, all this work, including the optimizations, only considers a large distributed system.

The Perada system is built on top of Spark, Redis, SQL and Python. Their system is highly distributed as running perturbations are, as they say, "embarassingly parallel." As such, we drew inspiration from the original system for our in-memory version, and brought multiprocessing into our system, to gain runtime speed in a one computer situation. Perada also describes a caching strategy, but applies this only to post-processing the results of the perturbation, and do not consider how to cache the intermediate results during the perturbation. We implement a caching strategy during the perturbation itself, to make gains from already computed subranges of parameters being searched. The following section will describe our approach in more detail.

2.2 Pandas

The pertinent question of whether to use Pandas or SQL is whether a DataFrame abstraction is more productive than a SQL abstraction from the standpoint of journalists and analysts. There are two main reasons that it often is. One is the domain knowledge required to operate SQL as opposed to using Pandas framework. The other, more importantly, is that going from raw data all the way to data visualization in one code base is convenient using Pandas. The present implementation of Perada [2] is built on top of SQL and thus journalists need to be able to write complex SQL queries in order to use it effectively. In this paper we present query perturbation as a functionality of the Pandas framework.

Robbert van der Gugten [11] talks about key optimization points that are relevant to our implementation. Index optimization is very helpful for fast merging or joining tables, which is a frequently used operation in the given scenario. Also, vectorized operations provide speedups as opposed to an iterative approach. Lastly, executing filter operations as early as possible, especially during multiple sequences of operations, can provide a significant performance improvement. An example is executing a filter after an inner join between two tables which would cause performance degradation. Finally, the presentation by McKinney discusses important data structures and more technical details used to achieve speedups in Pandas [8]. No existing Pandas functions allow for query perturbation on Pandas' dataframes.

2.3 Memoization and Caching in Queries

Caching, and more specifically, memoization have been common techniques across computing to attain faster processing times by avoiding re-computation of already calculated results. Memoization is the re-use of already computed function calls when the same inputs are provided. In the database literature, Hellerstein et. al. examined hybrid caching schemes against the traditional and widely used sorting technique in Object-Relational and Object-Oriented database management systems where users are allowed to invoke expensive user defined methods [5]. Their analysis of the costs of user defined functions and review of main memory caching systems was instrumental in helping us test our caching techniques, even if we rarely exceed the main memory for our system.

2.4 Map Reduce

The advantages of caching requires a later reduction ability to be able to reuse previous computations. The MapReduce framework offered lessons in parallelization as well as reduction [4]. MapReduce was a breakthrough paper pioneering re-imagining computation of large workloads on a parallel system architecture. It introduces a new programming model defining two functions map and reduce. The map function converts the input key-value pairs to an intermediate value and the reduce function merges all the intermediate values of a given intermediate key to output the final result. This is quite similar to the steps that query perturbation goes through. We take inspiration from the MapReduce programming model in adopting the mapping of input to intermediate values and reducing the intermediate values associated with its respective keys into the final output, with a carry variable to allow for weighting in the reduction.

3 APPROACH OVERVIEW

Query perturbation solves the instance of one query template and many different parameter settings that are needed to achieve a desired calculation or outcome. Perturbation analysis has two main steps: evaluating the perturbation and then analyzing the result set. As such, to borrow some language from Perada, there exists a query template q, a database on which it acts, D, and a combination of parameters drawn from a larger parameter space of ranges of those parameters to cycle through (p). Evaluating the query can be thought of as a function: q(p). If two parameters vary with ten values each, the parameter space has hundred possible combinations to search. The inputs to Perada or our system, Pert-Q-Pan, defined by the user, are the database, D, in question, the query template, q, the parameters to search, and the ranges to search over, P. Finally, once the results of the query are calculated, q(p), to obtain any interesting results, a post-processing function X operates on the result set to obtain a final value that one would care about.

In Perada, cunits, or pieces of work are distributed amongst a large system that uses Spark to run all the perturbations. The intermediate results are sent to a master than then does the final "reduction" or post processing function X.

Two types of caching are used. The first is a global key value cache, built on top of Redis, that is shared by all cunit workers that are operating on a single unit of work. This memoization cache allows for a lookup of the exact keys that are currently being computed in case there was a failure or a duplication of assigned inputs to the distributed workers. This cache is not used during the main perturbation to speed up any computations by using partially pre-computed results, which is the novel improvement our system offers. Instead, this cache is used for helping calculate the final results of the post-processing function X. For Perada, it also aids in pruning unnecessary computations from occurring if the system is certain that in the post-processing of the intermediate results, the computation would not, for example, make the top ten values if that is what the end user is interested in. A second local SyncSql cache is used for more robust pruning.

We present a Pandas compatible API that allows the below example discussed in 3.1 to be trivial to locate for a user. While Perada is implemented on Spark using Python and SQL, we implemented a Perada-like system that takes in a query and a set of parameters, and issues native Pandas methods. Our system, Pert-Q-pan, which stands for Perturbed Queries in Pandas, allows for query perturbation to check the robustness of a claim made on a database, or to locate interesting values. We implement multiprocessing and caching to improve the performance of the system and hypothesize that these two improvements can significantly improve run-time. Indeed, we find that they do, but that there are trade-offs when using both.

3.1 A Real World Example

In reality, many journalistic investigative articles can be framed as the above describes: queries with perturbed parameters. For example, an article recently published by The Markup, titled "Facebook Charged Biden a Higher Price Than Trump for Campaign Ads," isolates a two month period during which the average cost of an ad on Facebook for presidential nominee Biden was significantly higher than that for Trump. It reads, "The difference was especially stark in advertisements aimed primarily at Facebook users in swing states in July and August, where Biden's campaign paid an average of $34.34 per 1,000 views, more than double Trump's average of $16.55." [7]. This can essentially be re-framed as a query on a database, one where the parameters are searched for to produce interesting results, which in this case is a stark difference paid by two different campaigns during a comparable time period.

One plausible technique to isolate such an interesting claim would be to consider graphing the values in question and guessing at parameter values that would produce the max difference between the candidates to illustrate the argument of the article that Facebook is charging two candidates drastically different amounts. Since the cost of an ad varied over time across different states for different candidates with different numbers of target audience members reached, identifying a particular claim like this is non-trivial. As provided by the article, the entire analysis was done in Pandas, but it is unclear how the start and end dates were selected. To re-frame in perturbation terms, the parameters varied were a start date and a length of time to consider. The parameterized query template would calculate the mean cost value for both Biden and Trump ads between the start date and the length added to the start date, and take the difference between the two. The post-processing query, X, would look for the maximum value from the result set of all the differences calculated for all combinations of start date and length.

3.2 Multiprocessing

The workloads described are easily parallel. Each unit of work, which is one combination of parameter values out of the parameter space, can be run in a separate process. The same code is run, just with different inputs. Setting up the multiprocessing does take some overhead, and therefore it may cause diminishing returns when upping the parallelizations if the overhead time dominates the run time. For example, a variety of pre-filtering techniques may dominate the runtime and might be chosen to be performed before the dataset is handed to the child processes. The output from the parallelization is all of the outputs from all the jobs, which would then be handed over to the post-processing function X to calculate the final result.

Pert-Q-Pan does have options to filter out the columns and rows that are not applicable to the perturbation in question, and these have differing impacts on the runtime as discussed in the experiments section.

Perada describes one multiprocessing optimization which they term grouping. Grouping refers to the process of dividing data into groups for efficient evaluation of perturbation [2]. It may, for example, be worthwhile to hand every child process a grouped dataset where one parameter value is fixed, and the whole range of perturbations for the other parameter are considered to avoid having to filter the original dataset in a variety of ways, reducing the amount of total jobs required and operating on the same subset of data for the life of a subprocess. While we do not currently take advantage of dividing the work in this way, it would be possible to explore such an optimization in the future.

3.3 Caching to Reduce Computation Time

Perada does implement caching but does so only as a tool for post-processing on the result set, as mentioned above.

In contrast with that, we decided to implement a form of memoization in the main processing of the query perturbation. By defining a computation in terms of sub groups that can be combined, we can combine sub sections of the space to form larger sections, saving computation time at the expense of memory.

It is worth noting that while inserts into the cache should be idempotent, in our situation this is not really a concern as every unit of work submitted to the executor will never have the exact same parameter inputs.

One problem with the above is that since we are blind to the actual parameters and their values, and treat them as arbitrary objects, we require the user to specify an optional overlap function that helps us key the cache, and the query template itself must handle a reduction like behavior by handling previous results, and a carry variable that helps the user weight the previous result in the final combined result with the current computation. This does introduce some more complexity on the part of the user, but the trade-off is potentially very advantageous in run time.

4 TECHNICAL DETAILS

The user must specify the parameters, the ranges, the query template and then run the perturb function on the query object. The main point of entry for a user is the pQuery object, off which the perturb method can be activated. The perturb method handles kicking off the execution and returning the results object.

The pQuery object, which stands for perturbed query, and the pParam object, which stands for perturbed parameters, are interfaces that are defined by us that will aid users in building an executable query. First, a user specifies the parameters of interest.

The pParam class looks like the following:

```
class pParam:
    # Dictionary with the parameters to search
    # and the ranges to search in
    param_name_to_range = {}

    # Add a search space to a given parameter name
    def add_param(param:str, range:Object)
```

And can be used in the following way:

```
params = pParam()\
        .add_param("start_date", all_dates)\
        .add_param("length", range(1, 30))
```

Here, the parameter name is added along with the ranges to search. The pQuery object class looks like the following:

```
class pQuery:

    # To define a pQuery object, the user must
    # provide a dataframe, the query template,
    # q, and the post-processing function X.
    def __init__(df: Pandas.DataFrame,
                 query_template: Callable,
                 agg_func: Callable)

    # To take advantage of the ability to optimize
    # to avoid recomputing outputs for segments of
    # inputs when we have already computed segments
    # that makeup a larger segment, users define an
    # overlap function
    def add_overlap_fn(
        overlap_fn(*parameters):Callable)

    # To kick of the actual execution of
    # everything specified
    def perturb(params: pParamObject) -> pResultObject

    # This is the function that is submitted
    # to the multiprocessor for running each
    # job with the specified combination of
    # parameter inputs to try in the query
    # template
    def run(query_fn: Callable,
            df: Pandas.Dataframe,
            shared_cache: Manager.dict,
            combo: Tuple)
```

The run function is called from the perturb function. The perturb function traverses the ranges of the parameters and creates a tuple for every set of combinations of the parameter inputs. Each specific combination of parameter values is submitted to the multiprocessor along with the dataframe to act on, the query template, and the shared cache, if one is being used. The shared cache is ensured to be a shared piece of memory between the processes through the Manager class, which grants read and write access to all child processes.

It is important to note that processes and threads are different in Python. Multithreading in Python allows computation to progress in separate threads, but only one CPU process is technically operating at any given time. Thus, we opted to use multiprocessing to truly parallelize the workload, and have multiple CPU cores that are continuously making progress at the same time, with a maximum number of available processes calculated by the pQuery object. The managing of shared memory needs to be handled carefully, and thus we turn to Python's builtin Manager class to handle caching across processes. But, everything else does not rely on sharing any other shared writable data, only reading.

This can be used in the following way:

```
result = pQuery(df, query_template_2, max)\
        .add_overlap_function(overlap_fn)\
        .perturb(params)
```

The following is an example query template function, which should take a few required parameters if the caching optimization is to be taken advantage of:

```
def query_template_2(df: Pandas.DataFrame,
                     params_tuple: Tuple,
                     previous_params: Tuple=None,
                     previous_result: Tuple=None,
                     carry_variable: int=None):
```

The downside of implementing caching was that some additional considerations must be undertaken by the user. They must handle previous specific param input values, the result of the query template on those values, and any carry variable that is required for weighting the previous result. For example, if we are trying to find the mean value of a column, it would be possible to cache previous mean values for a subset of that column that is smaller than the current consideration of the range. If so, we can use the knowledge that the previous calculation for the mean consisted of 11 rows (the carry variable), and then calculate the mean on any remaining rows that weren't considered in the cached version. Say we have 3 rows that weren't previously considered. We could calculate a mean on those three rows, and weight that means by 3/14, weight the previous mean by 11/14. The previous result gives us the previous mean, the carry variable tells us how many rows were considered earlier to aid in weighting, and the previous params help us figure out which remaining rows have not been processed. Much of this is inspired by the MapReduce foundational paper, which describes in detail how to carry out reductions such as these [4].

It will also be beneficial to make optimizations with regards to the intermediate tables that are calculated and possibly stored. Depending on the necessary computations, it may be more efficient to create a separate materialized view or intermediate table that is stored to do the perturbation on depending on what the query looks like to keep computation numbers to a minimum. As we can relate this to conducting a grid search, while holding any set of parameters constant, it would likely be beneficial to conduct computations against a Pandas dataframe where only a subset of the data is held. Advanced users will be able to aid optimization by specifying any queries they believe would be an adequate intermediate table.

The biggest problem in not being able to achieve sufficient results or any of these executions not being possible is how long the computation time would take, which is based on the factor of how many computations need to run and the size of the dataset. The limited computational power of a single machine, as well as the limited operating memory, may make certain perturbations unfeasible, in which case users should turn to the Perada system.

5 EXPERIMENTS

Our experiments focus on toggling the multiprocessing and caching to understand how they affect the run-time. In the same way that the gains of the Perada paper came from the optimizations they had to offer, with the optimizations we are implementing to the in-memory system, we hypothesized that we would be able to outperform a simple brute-force grid-search of the parameter space on a given query. Thus, our main bench-marking occurs against brute-force searching of a parameter space. We utilize a workload of a varying specified number of rows and columns, which a variable parameter search space. The workloads are all randomly generated datasets with a specific format. We vary the caching, parallelism, and run all combinations of the parameter space. The following experiments entail the findings.

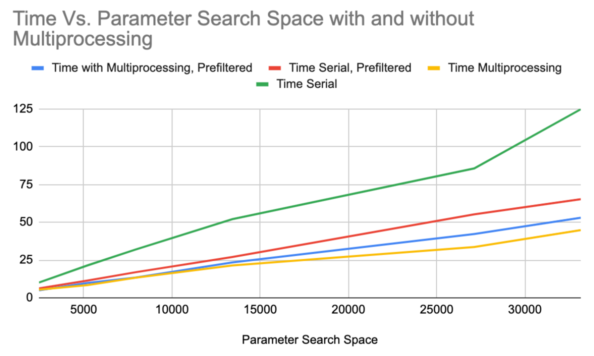

Figure 1: Run times (seconds) versus the parameter search space (number of combinations of specific parameter values to try) for different environment setups. We toggle the multiprocessing and whether or not the dataset filtering is occurring in the main or child processes. Indeed, multiprocessing is beneficial in terms of run time and this performance improvement is larger the larger the parameter search space is.

5.1 Experiment 1: Varying Parameter Search Space to Understand Multiprocessing Gains

The first experiment is devised to test how multiprocessing improves runtime as the parameter search space is increased. As predicted, multiprocessing does allow similar workloads to finish faster that if they are done as one long serial process. Multiprocessing in this experiment means utilizing all available 8 processes that the single machine we tested on was able to run at a time.

The fake dataset used for all the experiments was modelled after the analysis from The Markup piece on Trump and Biden Ad prices that was discussed earlier. The two parameters being ranged were the start date and a length of time, and the value being considered was either the maximum average price paid by a candidate or the maximum difference of the averages paid by the two candidates. The dataset contained 5 columns, and a variable number of rows. In Figure 1, 10,000 rows was chosen as a believable possible size for such a dataset.

The ranges of these two parameters were increased for each run to build the above results. The four different environments are multiprocessing, multiprocessing with pre-filtering, serial-processing and serial-processing with pre-filtering.

The term pre-filtering refers to reducing dimensions of the parameter space. This includes removing columns and rows that do not contribute to the computation of the result. Prefiltering indicates that filtering down to the portion of data selected for this computation was done in the main process, and then this reduced dataframe is passed to the child processes.

From the graph it is evident that multiprocessing does produce a win for run time. Pre-filtering the dataset causes the whole run time to take longer as the filtering causes a large amount of overhead in the main thread, but we suspect that its memory footprint would be lower, which could potentially be desired or advantageous. Future experiments could investigate the memory footprint of such choices.

5.2 Experiment 2: Understanding the Influence of the Cache

Similar to former experiment, we ran our system in different environment settings and noted the run-time. We toggle both the cache and the multiprocessing. Again, multiprocessing in this case is the maximum parallelization possible of 8 processes. The four settings are multiprocessing with cache, multiprocessing without the cache, serial execution with the cache, and serial execution without the cache. 10,000 rows were used, and a 2345 search space where 20 values was the range of the length and the start date varied from January 1 to December 1.

The built in Pandas mean function, which was being used by the user defined query template, is a hyper-optimized function that has a low computational complexity. In order to simulate more computationally complex user defined functions, we introduced a sleep time of a constant multiplied by the length of the input dataset raised to some exponent, e. The value of e was varied in the experiments. This helps us model potentially more complex user defined functions that would take an increasingly longer period of time to run, and thus we can see more performance improvement from the cache. More complex user defined functions benefit more from caching, which is understandable given what we can learn from Hellerstein et. al [5].

It can be deduced from the values in Table 1 and 2 that for this workload, caching gives us a performance improvement, albeit a different magnitude in the serial and multiprocessing environments. This difference in performance improvement can be reasoned that serial execution has a greater number of overlapping computations since computations are not potentially happening at the same time when they could have been reused. The specified overlap function, which is what is specified to allow Pert-Q-Pan to know what to retrieve from the cache, was looking for any entries that had the same start date, and a length that was less than the current computation. When the system was running in the multiprocess environment, the possible value that a current computation could have reused was being processed in parallel, and hence was not available yet for reuse. If we have unlimited CPU cores, we could process the entire perturbation space in parallel, but ultimately do a lot of excess computation. This highlights the trade off of caching and multiprocessing.

One possible modification would be to schedule the specific parameter values, or pieces of work that need to be computed, optimized for cache hits when queuing up the pieces of work assigned to the multiprocessor.

5.3 Experiment 3: Varying the parallelism of the Multiprocessor

Table 3 shows the degree to which the amount of parallelization improves runtime. The number of processes allowed to run was varied, with 100,000 rows, 90 values possible for the length parameter, and a total search space of 10,050 possible combinations. As we can see, the biggest wins in the case come from just moving from one to two processes, and there are diminishing returns after that. That is because Python has significant overhead for multiprocessing, and some shared resources could not be fully parallelized, but perhaps this could be improved in the future while still supporting the same workloads.

5.4 Experiment 4: Caching in Post Processing

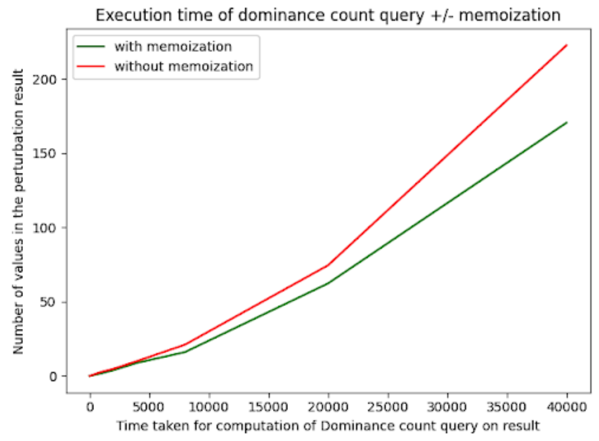

Figure 2: Runtime of caching versus not caching in post processing.

This experiment is devised to check the performance of post processing with and without caching. Caching here helps to avoid re-computation of queries that have already been computed before. In this paper we have implemented dominance count on the perturbation result. Several useful functions like ranking, clustering and many more can be implemented on the perturbation result. It can be deduced from Figure (2) that for very few values (<5000) caching doesn't provide any speed-ups. But for more than 25000 values, there is a significant difference in the computation time of post-processing query.

Exponent e	No Cache, Multiprocessing	Cache, Multiprocessing	Percent Improvement
0	6.201	5.273	14.97
1	5.425	5.151	5.05
1.25	6.517	6.441	1.17
1.5	10.716	9.928	7.35
1.75	27.71	26.013	6.12
1.9	56.252	50.512	10.20
2	92.711	79.027	14.76
2.1	150.415	110.464	26.56
2.15	199.793	142.274	28.79
2.2	271.692	175.849	35.28
2.3	458.008	269.64	41.13

Table 1: This table indicates the improvement our caching offers, while multiprocessing is turned on. Time is in seconds. The exponent is a variable parameter that indicates or models how time intensive any UDF is. The larger the exponent, the more time intensive the modelled UDF is. As we can see, caching partial computed values offers decent performance speed ups, though the speed ups at low values for e are inconsistent since the overhead of the runs likely dominates the run time.

Exponent e	No Cache, Serial	Cache, Serial	Percent Improvement
0	6.541	10.921	-66.96
1	13.268	12.712	4.19
1.25	24.653	15.924	35.41
1.5	61.887	22.806	63.15
1.75	204.681	35.747	82.54
1.9	416.466	49.598	88.09
2	675.801	62.967	90.68
2.1	1217.815	86.474	92.90
2.15	1567.611	95.833	93.89
2.2	-	110.178	-
2.3	-	154.512	-

Table 2: This table indicates the improvement our caching offers, while multiprocessing is turned off. Time is in seconds. The exponent is a variable parameter that indicates or models how time intensive any UDF is. The larger the exponent, the more time intensive the modelled UDF is. As we can see, caching partial computed values offers incredible performance speed ups in the serial case, though the speed ups at low values for e are inconsistent since the overhead of the runs likely dominates the run time. In the serial case, the previous computed values are placed into the cache and then immediately used by the next computation, so we see more improvement in the serial case over the multiprocessing case. This highlights the trade off of caching and multiprocessing.

Multiprocessing	Time, Cache off	Time, Cache on
1	98.917	113.26
2	61.541	71.131
3	59.866	62.797
4	61.431	66.59
5	62.439	66.078
6	61.063	70.579
7	60.865	67.048
8	66.189	64.049

Table 3: Runtimes, in seconds, while varying the amount of parallelization possible with the cache turned off and on.

5.5 Comparing the usage of Perturbation Analysis using Pandas and SQL

The framework of Perada has not been made public. Hence here we draw the comparison between SyncSQL which is also built on top of SQL as mentioned in [2] and Pert-Q-Pan.

Below is the implementation in both Pandas and SyncSQL for the query perturbation problem statement as follows:

There are two tables containing information about baseball games of players across seasons. The first table is PLAYER_INFO which contains information about the player like age, attributes, player_ID, and others. The second table is SCORE_STATS that has player_ID mapped to the statistics of games that a specific player P has played across different seasons(year). Now the task is to get the hits and home runs of a player averaged over a given number of seasons(between start_year and end_year). Once the result of the task is attained, the dominance count of how many more players have home-runs and hits greater than the given player P between start_year and end_year is calculated.

Pandas implementation

```
params = pParam()
.addParam('season', range(9,14))

presult = pQuery(df)
.add_filter_stage(["hits","home-runs"])
.add_groupby_stage(["player_name"])
.add_agg_stage(avg,["hits","home-runs"])
.perturb(param)

domcount = presult
.domCount({"hits":presult.x, "home-runs":presult.y})
```

SyncSQL implementation

```
DECLARE @x as INT, @y AS INT
SET @x,@y = SELECT AVG(hits), AVG(home-runs)
FROM (PLAYER_INFO JOIN SCORE_STATS ON PLAYER_ID)
WHERE SEASON >= start_year AND SEASON <= end_year
GROUP_BY PLAYER_NAME
SELECT MAX( domcount + (CASE WHEN hits > @x
OR home-runs > @y THEN 1 ELSE 0)) FROM cache
WHERE hits >= @x AND home-runs >= @y;
```

Comparing the above implementations of query perturbation in Pandas and SyncSQL, it is evident that the former offers a very simplistic framework for usage. The length of the SQL query is long, and the knowledge of SQL required to write the SQL query is cumbersome. As opposed to the SQL counterpart, Pandas implementation offers an easy to use API interface with method chaining. People with minimal programming experience and coding knowledge can delve quickly into getting insights into the data rather than on the implementation of the perturbation.

6 FUTURE WORK

As mentioned earlier, grouping the parameters before handing them off to the multiprocessor would likely improve runtimes, especially when an expensive overhead calculation is necessary on the grouped parameter. This would be a natural extension of the Pert-Q-Pan system, and it would be interesting to explore its effects. The other area where interesting developments could be pursued is in instituting a lineage capture during the processing, but more importantly, the post-processing part of the perturbation. This would help inform what computations to prune or cancel/ignore in the main phase of the perturbation. This would help cut down on the number of jobs that would need to be run, as some parts of the search space could easily be ignored if we know it will not end up in the final result from the post-processing query. For example, if we are looking for the top ten results, if we know that at a particular value of parameter A, there exists no value of parameter B that will allow the result to make it into the top ten, we can simply avoid calculating any of those jobs in that part of the parameter search space.

One other optimization we can take advantage of is to utilize our knowledge about some of the slower aspects of Pandas that a typical user may not know and rewrite the users' code to be more efficient when possible. For example, Pandas' apply function is notoriously slow, and in many cases can be replaced with vectorization [5]. In the Facebook Ads story, there are several points at which the author uses the apply function to map over or filter a column, which could be rewritten given our knowledge of Pandas.

Additionally, we currently have no support for failures since failure is unlikely in single machine operations. Still, if failure or errors occur, the whole process must be restarted from the beginning. A checkpoint system could be implemented to allow for recovery to prevent having to start from the beginning again. Finally, an interesting variation on the caching strategy might involve randomly sampling from the parameter space when assigning work to the multiple processes, to increase the chances of usable pre-computed chunks from the cache.

7 CONCLUSIONS

Query perturbation is a unique novel tool that has much to offer data journalists. By allowing journalists to check the robustness of claims, and discover interesting data points, new stories are made possible. However, the current technology is out of reach for most journalists that do their data analysis work largely in Pandas.

We introduce Pert-Q-Pan, an in-memory Pandas compatible API for conducting query perturbation. We optimize around multiprocessing capabilities of Python, memoization for caching, and discuss some possible tradeoffs to consider when turning both of these optimizations on. The simplistic API and examples allow any data journalist to run query perturbation on their single machine, in memory dataset.

REFERENCES

[1] R. E. Bellman. Dynamic programming. princeton university press. 1957.
[2] J. Y. Brett Walenz. Perturbation analysis of database queries. *VLDB*, 2016.
[3] S. Ganguly. Design and analysis of parametric query optimization algorithms. 1998.
[4] S. G. Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. 2008.
[5] J. F. N. Joseph M. Hellerstein. Query execution techniques for caching expensive methods. 1996.
[6] P. K. A. C. L. J. Y. C. Y. B. W. S. R. Jun Yang, You Wu. Query perturbation analysis: An adventure of database researchers in fact-checking. 2018.
[7] J. Merrill. Facebook charged biden a higher price than trump for campaign ads. *https://themarkup.org/election-2020/2020/10/29/facebook-political-ad-targeting-algorithm-prices-trump-biden*, 2020.
[8] D. R. a. P.G.Selinger, M.M.Astrahan. Data structures for statistical computing in python. in proc. of the 9th python in science conf. (scipy 2010), pages 56–61. python foundation, 2010. 2010.
[9] J. Y. C. Y. Sarah Cohen, Chengkai Li. Computational journalism: A call to arms to database researchers. 2011.
[10] P. K. A. Stavros Sintos and J. Yang. Selecting data to clean for fact checking: Minimizing uncertainty vs. maximizing surprise. 2019.
[11] R. van der Gugten. Advanced pandas: Optimize speed and memory. *Medium.com*, 2019.

[12] C. L. J. Y. C. Y. You Wu, Pankaj K. Agarwal. Toward computational fact-checking. 2014.