Université de Montréal

Measuring RocksDB Performance and Adaptive Sampling for Model Estimation

par

Jean Laprés-Chartrand

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Recherche Opérationnelle

January 24, 2022

© Jean Laprés-Chartrand, 2021

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé

Measuring RocksDB Performance and Adaptive Sampling for Model Estimation

présenté par

Jean Laprés-Chartrand

a été évalué par un jury composé des personnes suivantes :

Jean-Yves Potvin
(président-rapporteur)

Fabian Bastin
(directeur de recherche)

Abdelhakim Hafid
(membre du jury)

Abstract

This thesis focuses on two topics, namely statistical learning and the prediction of key performance indicators in the performance evaluation of a storage engine.

The part on statistical learning presents a novel algorithm adjusting the sampling size for the Monte Carlo approximation of the function to be minimized, allowing a reduction of the true function at a given probability and this, at a lower numerical cost. The sampling strategy is embedded in a trust-region algorithm, using the Fisher Information matrix, also called BHHH approximation, to approximate the Hessian matrix. The sampling strategy is tested on a logit model generated from synthetic data. Numerical results exhibit a significant reduction in the time required to optimize the model when an adequate smoothing is applied to the function.

The key performance indicator prediction part describes a novel strategy to select better settings for RocksDB that optimize its throughput, using the log files to analyze and identify suboptimal parameters, opening the possibility to greatly accelerate modern storage engine tuning.

Mots-Clés

optimization, statistical learning, RocksDB, LevelDB, Fisher information

Résumé

Ce mémoire s'intéresse à deux sujets, un relié à l'apprentisage statistique et le second à la prédiction d'indicateurs de performance dans un système de stockage de type clé-valeur.

La partie sur l'apprentissage statistique développe un algorithme ajustant la taille d'échantillonnage pour l'approximation Monte Carlo de la fonction à minimiser, permettant une réduction de la véritable fonction avec une probabilité donnée, et ce à un coût numérique moindre. La stratégie d'échantillonnage est développée dans un contexte de région de confiance en utilisant la matrice d'information de Fisher, aussi appelée approximation BHHH de la matrice hessienne. La stratégie d'échantillonnage est testée sur un modèle logit généré à partir de données synthétiques suivant le même modèle. Les résultats numériques montrent une réduction siginificative du temps requis pour optimiser le modèle lorsqu'un lissage adéquat est appliqué.

La partie de prédiction d'indicateurs de performance décrit une nouvelle approche pour optimiser la vitesse maximale d'insertion de paire clé-valeur dans le système de stockage RocksDB. Les fichiers journaux sont utilisés pour identifier les paramètres sous-optimaux du système et accélérer la recherche de paramètres optimaux.

Mots-Clés

optimisation, apprentissage statistique, RocksDB, LevelDB, Information de Fisher

Table des matières

Abstract 5

Résumé 7

Mots-Clés 7

Liste des tableaux 11

Liste des figures 13

Liste des sigles et des abréviations 15

Remerciements 17

Introduction 19

RocksDB 19

Statistical Learning 19

Thesis structure 20

Chapitre 1. RocksDB Performance Tuning Based on Log Files 21

1.1. Introduction 21

1.2. RocksDB description 22

1.2.1. RocksDB Architecture 22

1.2.2. Configuration Space 24

1.2.3. Existing performance models 26

1.2.4. Objective 27

1.3. Observations 28

1.4. Model 28

1.4.1. Flush Thread Utilization 30

1.4.2. Compaction Thread Utilization 30

1.4.3. Level 1 Utilization 31

1.4.4. Parameter Selection 33

1.5. Exploration 34

1.6. Conclusion 34

Chapitre 2. Statistical Learning 37

2.1. Introduction 37

2.2. Mathematical Formulation 41
2.2.1. Notations and Basic Assumptions 41
2.2.2. Assumptions 42
2.2.3. Basic Trust-Region model 42
2.2.4. Trust-region subproblem 43
2.2.5. Approximation of the Hessian matrix 44

2.3. Algorithms 46
2.3.1. Sample size strategy 46
2.3.2. Relation to other methods 51
2.3.3. Novelty of our method 54

2.4. Numerical Experiments 55
2.4.1. Multinomial Logit Model 55
2.4.2. Results 56
2.4.3. Hessian Approximation 59

2.5. Conclusion 60

Conclusion 61

Appendix A. Multinomial logit derivatives 63

Liste des tableaux

2.1 Sampling strategies nomenclature . . . 51
2.2 Smoothing nomenclature . . . 51
2.3 BHHH approximation benchmark . . . 57
2.4 Optimization time, BHHH approximation . . . 59
2.5 True Hessian matrix benchmark . . . 60
2.6 Times and results for BTR algorithm with the true Hessian matrix . . . 60

Liste des figures

1.1 RocksDB schematic representation 23
1.2 A small extract of the log file from which required metrics were measured. 28
1.3 Algorithmic complexity of merging sorted lists is $\mathcal{O}(n)$ with respect to the total size of the list to be merged. 29
1.4 The distribution of the time between the creation and the deletion of a memtable is asymmetric. 29
1.5 Graphical representation of the behavior of the files in a level during a cycle 32
1.6 Graphical representation of a cycle 32
2.1 Unconstrained sampling 58
2.2 Distance to solution and sample size for BHHH approximation 58

Liste des sigles et des abréviations

BTR	Basic Trust Region
KOPS	Kilo Operations Per Seconds
KPI	Key Performance Indicator
KV	Key-Value
LSM-tree	Log-Structured Merge-Tree
MLE	Maximum Likelihood Estimation
SSD	Solid State Drive
SAA	Sample Average Approximation
TCG	Truncated Conjugate Gradient

Remerciements

I would like to thank Doctor Fabian Bastin and Doctor Étienne Elie for their help and guidance in the domains of statistical learning and settings optimization, respectively, as well as for sharing their expertise for the realization of this thesis.

I would like to thank as well for their help Morgan Githinji, Jeremy Rieussec, Elizabeth Michel, and notably François Corneau-Tremblay for his help with git and Julien Codsi for leading by example with his suboptimal Julia code.

This work would not have been possible without the generous support of Intel Corporation, where the data and ideas for the first chapter of this thesis have been collected during a nine months internship. I am very thankful for the generous support and opportunity Intel has provided.

Introduction

In this fast and competitive world, optimization is taking an ever-growing place in most fields related to computer science and operations research. Such fields include but are not restricted to key-value pair (KV-pair) storage and statistical learning, two topics covered in this thesis.

RocksDB

RocksDB is key-value storage engine based on a log-structured merge-tree (LSM-tree) structure that reaches a high insertion throughput by using an in-memory write buffer called memtable to avoid small random write requests to storage and only write long sequential write requests to storage, taking full advantage of modern storage systems. RocksDB is however difficult to tune as it is characterized by more than 50 different configurable settings, each one being capable to impact various key performance indicators (KPIs). We propose a novel approach based on RocksDB traces files to measure the storage system internal metrics. This allows a better understanding of its behavior and pinpoints non-optimal settings, thus guiding the search for more efficient configurations.

Statistical Learning

Modern statistical learning problems are now, more then ever, complex. The calibration, sometime called training, of such models, for instance neural networks composed of many layers, leads to a high dimensional optimization problem, relying on datasets of ever increasing size. Stochastic first-order optimization algorithms are usually used to solve such high-dimensional problems. In such methods, the gradient of the function can be easily approximated by means of Monte Carlo sampling, decreasing the time required per iteration. Use of second-order optimization algorithms is less common as the storage of the Hessian matrix, or some approximation of it, is usually impractical or even impossible.

Multiple techniques have been developed to use second-order optimization algorithms without the costly storage of the Hessian matrix. In this thesis, we explore such techniques and develop a sampling strategy based on a second-order approximation of the function,

reducing the sample size while guaranteeing a reduction in the function to minimize with a significant probability.

Thesis structure

This thesis is partitioned into two parts. The first chapter covers the write throughput optimization for RocksDB. It is divided into three sections: a description of RocksDB, a review of analytical models describing RocksDB performance, and finally a presentation of the proposed new methodology using RocksDB log files to optimize its settings. The second chapter covers the statistical learning part of this thesis. It is divided into three sections: a mathematical formulationof the problem under consideration, a description of the proposed algorithm, and finally some numerical experiments. A general conclusion follows.

Chapitre 1

RocksDB Performance Tuning Based on Log Files

1.1. Introduction

A log-structured merge-tree (LSM-tree) is a data structure that stores data in the form of key-value (KV) pairs [? ?]. It has two main components. The first component, named "the write buffer", resides in memory and serves to buffer and alleviate costly random write operations to the storage device. The second component hosts most of the KV and resides in storage. When the write buffer exceeds a specified size, it is merged with the second component.

In many LSM-tree based architectures, the storage component is divided into multiple layers, called levels, with higher levels containing older KV pairs and lower levels storing the most recently written KV pairs. Levels are composed of multiple files containing KV pairs in a sorted fashion. Due to the architecture of the system, the same key can be associated to multiple values placed in different levels. A KV pair has at most, a single value at any level greater than or equal to one but can be associated to multiple values at level 0. If a key is associated to multiple values, the most recent version of the KV-pair is either the version in the write buffer or the version that was in the lowest level.

RocksDB [?] is a LSM-tree based storage engine created by Facebook, available at the address `http://rocksdb.org/`. It is used in a variety of applications : database storage engines, stream processing, logging services [?], index services, caching on solid state drives (SSDs) [?], ... It is an evolution of LevelDB [?], another storage engine based on LSM-tree, but takes better advantage of parallelism to get better throughput at the cost of marginal worst read latency [?]. RocksDB is highly customizable to accommodate users requirements and systems, and meets various types of use. The set of possible settings is therefore very large, and testing all parameters combinations is often too time consuming to be considered. Furthermore, testing a wide spectrum of settings requires to write large amount of data

to the storage device, which will inevitably reduce its life expectancy through write/erase cycles. Thus, the optimal combination of setting parameters is difficult to obtain.

Interesting work has been done to automatically optimize RocksDB settings [**?**] and develop an analytical model of its behavior. Such models include RocksDBs slowdown throughput [**?**] and space amplification [**?**]. In this chapter, we further investigate the possibilities to improve RocksDB operations, more specifically to optimize RocksDB write throughput. To the best of our knowledge, our approach is novel as it relies on a newly established set of equations linking the write throughput of RocksDB to several of its settings. Furthermore, we used and analyzed a single run of RocksDB to generate a log file in order to measure RocksDB internal thread usage and get a better perspective and accurate prediction of its behavior.

The rest of the chapter is organized as follows. We present the architecture of RocksDB, along with the main adjustable parameters, in the second section, followed by usual Key Performance Indicators (KPIs) of interest and their existing mathematical models. In the third section, we describe how a single run can be used to measure different metrics required for the optimization of write throughput. In the fourth section, we build a simple set of equations that describe the insertion rate by means of the measures previously obtained. We then suggest how to use these expressions to make suggestion on better settings for RocksDB, and we conclude.

1.2. RocksDB description

In this section, we describre the architecture of RockDB and its configuration space. We then review the main performance models in the litterature, and discuss the objective to consider when optimizing RocksDB.

1.2.1. RocksDB Architecture

As illustrated in Figure 1.1, RocksDB's main components are the memtables (mutable and immutable) denoted as MM, the levels (subdivided into SSTables), and the write-ahead log (WAL). The File (SST i j) is the j-th file of level i. In the example, there are three levels but the number of levels is usually larger.

RocksDB stores KV in ordered **levels**, numbered from 0. levels contain older KV, while lower levels store the most recently written KV pairs. The number of levels can dynamically changes with the data size. The KV pairs are pushed into deeper levels through an operation called leveled compaction. Each level is composed of **Sorted Static Tables** (SST) files. As their names indicate, SST files are composed of a sorted set of KV pairs. In levels greater than 0, the key range is partitioned and each SST file contains a non-overlapping range of keys. Due to the nature of the flush operations, SST files at level 0 have a key range possibly

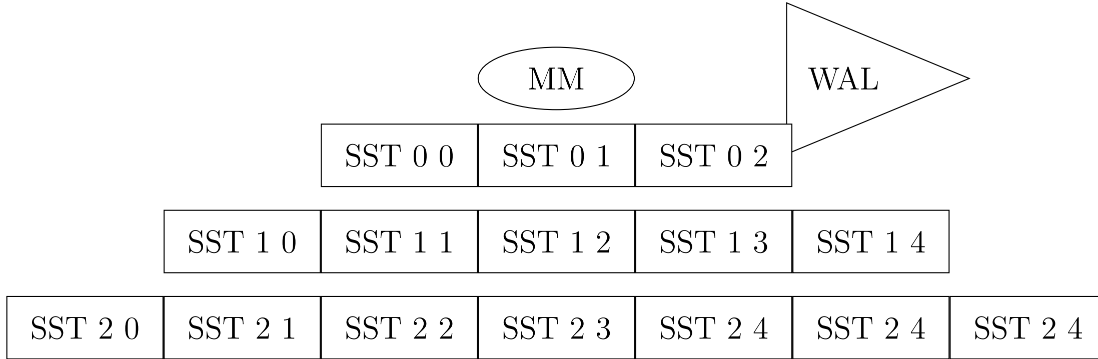


Fig. 1.1. RocksDB schematic representation

covering the whole key space, sometimes called key range, which is the set of keys that can be accessed by the user. Each level has a target size, set to x times the previous level size. x is often chosen to be equal to 10 but other techniques can be used [**?**]. The SSTs are composed of blocks, blocks are chunks of memories in which KV pairs can be accessed. To access the value associated to a key, the block must first be located in storage, loaded in memory from the storage and then searched. The blocks can contain many KV pairs.

The insertion of new KV pairs is done by means of the **memtables**, that are in-memory structures used as write buffers. They are implemented as a skip list, a structure that allows both insertion and extraction at a low computational cost [**?**]. Once a memtable is filled, it is set to immutable and a new empty mutable memtabe is created. Immutable memtables are eventually flushed to Level 0.

The **manifest** contains the information about all the files in RocksDB, the key range of all SST files, and the locations of the blocks. The **WAL** is used to recover unsynchronized data in the event of a failure. Every time a new KV pair is written, it is written into the memtable (in memory) and the WAL (also in memory). The WAL content is transfered to storage more frequently to ensure that recently written KV pairs are not lost in case of a system malfunction.

For each level k, we compute a score s_k, defined as the ratio between the level size, defined as the number of bits of the KV pairs stored in it, and the level target size:

$$s_k = \frac{\text{size of level } k}{\text{target size of level } k}.$$

The level k with the highest score is chosen for compaction. Then, the oldest available SST files are chosen to be merged into the next level, $k+1$. A file is available if it is not already in a compaction process. All SST files at level $k+1$ that contain KV pairs that overlap with the key range of the selected file at level k are also chosen. Once all the files that will be involved in the compaction are selected, they are locked, meaning that they cannot be involved in other compaction. The compacting process is simply the merging of the selected

files and the creation of new files. The files that were selected are used to create new files that contain their KV pairs and are then deleted. This process allows duplicated keys that possibly coexisted at levels k and $k+1$ to be merged, leaving the most recent KV pairs in the new file. Once the files have been merged, the manifest is updated, the newly generated SST files are written into storage, and all the files that were used are discarded as the KV pairs they were containing now exist in the new SST files. Compactions can occur simultaneously as long as they involve different files and multiple compactions can occur at the same level.

The Flush operation simply moves the KV pairs from immutable memtables into a new SST file at level 0. The key range of the newly created SST file can possibly cover the whole key range of the data set as the keys contained in this new file are only related by their ages while the keys of deeper files are related by their age and key names.

The location of a KV pair is not known, and we have to search for it when a reading request is made. The search stops when the KV pair has been found or when the last level has been explored. Structures are searched first in the mutable memtable, next in immutable memtables, and finally in the levels starting from 0 to the last level. Each file at level 0 must be searched as the key range of those files can include the key requested. For all levels greater than 0, a single file has to be searched as it is possible to find the only file that can possibly contain the key searched since the key range of those files are disjoint.

Hash tables can guarantee that a KV pair is not present in a given level, that can be skipped. Mutable and immutable memtables, and SST files at level 0 are usually cached in memory while higher levels are stored on the storage device. If the key was stored in a memtable, RocksDB can directly return the KV pair to the user. In the case where the key is on the storage device, it is costly to search for. The location of the block possibly containing the KV pair is found using the manifest; if the block is already present in memory, it is directly searched. If it wasn't already in memory, it is searched in the storage device. In the worst case, the read request targets an old KV pair that is stored in the deepest level. In such a case, if none of the blocks possibly containing the KV pair is in memory, and the hash tables are not used or do not indicate that the KV pair is not present at any level, then RocksDB must generate a costly random read request to storage per level.

1.2.2. Configuration Space

RocksDB has more than 50 tunable integer parameters. Thus, the number of possible configurations is very large, and changing one parameter can have a positive effect on a particular KPI, while reducing the performance with respect to other KPIs. Adjusting RocksDB settings to better suit encountered workload is thus difficult. We describe below the most commonly used parameters, their effect on some common KPIs, and their usual values.

S_B: the **size of the write Buffer**, a multiple of 32MB varying between 1 and 16. A bigger write buffer can lead to better throughput but might also increase tail latency [?] and memory usage.

T_F: the **maximum number of background flush operations** and T_C **maximum number of compaction threads** are parallelism-related configurations. For a write heavy workload, they can drastically increase throughput but will also increase CPU usage and so increase latency and tail latency [?]. Flushes and compactions are background operations of low priority, but are also time expensive compared to read requests. This means that if many of those background operations are occurring simultaneously, the application level request latency can increase. Increasing the maximum number of background compactions has been observed to shorten compaction cycles and improve performance [?]. Increasing the number of flushing threads is often reported to have no or negligible effect on KPIs [?]

S_C: **the read cache size**, affecting the read throughput. When the read cache size is increased, the probability of a block to be already cached after a get request is increased and the latency and tail latency are reduced at the cost of more memory usage.

B_b: the **bloom filter bit size** (bloom filter is disabled by default). B_b impacts the get latency and get tail latency by reducing the number of disk IO access. For random read heavy workload, [?] found that 12 bit bloom filter outperformed 1 bit bloom filter by up to 213 % and that the bloom filter function consumed only 0.9% of the CPU resources. The impacts of the bloom filter are larger on the memory utilization observing 344.1 MB memory footprint with 12 bit keys.

B_s: the **block size** (1 to 16KB and 4KB by default), hardware dependent. As the block size gets bigger, less memory is used in the manifest to keep the location of each block. Since an entire block must be read from storage if the KV pair requested is not in memory, more bandwidth might be used with bigger block size but this could be balanced by key request time locality as loaded blocks will reside in memory for a period of time after their request. [?] observed that increasing the block size decreased the database duplication but had a negative impact on read performance so they ended up not including this configuration to their final configuration.

w: boolean parameter allowing to **disable the WAL**. It is a common and easy optimization when data durability is not required, for instance when using Redis On Flash (ROF), a in-memory KV-Storage engine that uses SSD as memory extension [?].

t_s: **slowdown trigger**, allowing the activation of a mechanism in RocksDB that slows down the rate of KV pair insertion when there are too many files (12 files for slowdown trigger by default) in level 0. This feature is required to limit the size of level 0 and

give the system a chance to gain back its optimal level of size amplification, where the latter is the size of the of the RocksDB instance divided by the size of all the KVs it contains. As described in [**?**], if the limit is too low, the system will often slowdown causing long tail latencies as a write request occurring during the slowdown takes a longer time to perform. In contrast, a slowdown trigger which is too high might cause a compaction depth that can stop all traffic for even longer periods of time.

t_t**: the stop trigger**, similar to the slowdown trigger but completely stopping the system when it is reached (20 files for the stop trigger by default), causing even harsher penalty to the measured KPIs. [**?**] observed increased performance of 10 % by selecting higher values for the slowdown and stop triggers.

1.2.3. Existing performance models

To the best of our knowledge, a limited number of metrics have been proposed in the literature to quantity the performance of RocksDB. We found two analytical models describing the behavior of LSM-tree based algorithms. The first one is computing the write throughput when a slowdown trigger has been reached for RocksDB [**?**], and the second one is computing RocksDB space amplification based on key distributions [**?**].

?] describe the throughput of RocksDB once the slowdown trigger has been reached. Let λ_a be the application level throughput and λ_s be the system level throughput. These quantities respond to the equality

$$\lambda_a = \frac{t}{\tau + t} \cdot \lambda_s,$$

where τ is the delay period time. The delay period time is a period of time during which the put rate will be slowed down. t is the median latency, i.e. the median time to complete a write operation. They consider as a use case RocksDB for a 3D XPoint SSD, the background processing capacity when compaction happens is 190 thousand operations per second (kops), $t = 15ms$, and $\tau = 1024ms$. This results to an application level throughput of $15/(1024 + 15) \cdot 190kops = 2.74kops$. This value is coherent with the throughput they observed with 90% write workload.

?] describe the space amplification of RocksDB based on basic key distribution assumptions. When running benchmarks to evaluate the performance of a KV storage engine like RocksDB or LevelDB, two key distributions are often used: the uniform distribution and the Zipf distribution. Both assume that the key set can be mapped to the set of integers $K = [1, N]$, where N is the total number of keys. Both distributions assume that the key requests are independent and identically distributed (i.i.d). The probability-mass functions are $f_X(k) = 1/N$, $k \in K$ and $f_X(k) = (1/k^s)/(\sum_{n \in K} 1/n^s)$ for the uniform and Zipf distributions, respectively. Only cases with $0 < f_X(k) < 1$ are of interest since if $f_X(k) = 0$, then

the key k as a null probability of being requested and if $f_X(k) = 1$, then the workload is composed of a single key.

Let us now define $\text{Unique}(p)$, the expected number of unique keys appearing when p requests are made, and $\text{Unique}^{-1}(u)$, the expected number of requests with u different keys. It is easy to see that Unique is a bijection, $Unique^{-1}$ exists for all $0 \leq u < N$ and is the inverse of $Unique$. Mathematically speaking,

$$\text{Unique}(p) = N - \sum_{k \in K} (1 - f_X(k))^p,$$

while $\text{Unique}^{-1}(u)$ can be numerically computed by solving $\text{Unique}(p) = u$ for the unknown p. Finally, $\text{Merge}(u, v)$ is the expected size of a merged SST file created by merging two tables of sizes u and v respectively.

$$\text{Merge}(u, v) = \text{Unique}(\text{Unique}^{-1}(u) + \text{Unique}^{-1}(v)).$$

Using the previously defined distributions, they ran simulations of LevelDB behavior, and only tracked the simulated files. This resulted in a much faster simulation while giving an accurate approximation of space amplification. They used their model to optimize their use of LevelDB with respect to write amplification, varying the target size of each level. Using their approaches, they were able to reduce write amplification up to 9.4%. The use of a simulation instead of direct calls to LevelDB allow to speed up the optimization process. For instance, for 100 millions unique keys, the optimisation took 2.63 seconds in their experiments, evaluating 17,391 different parameter sets. On the opposite, running a single set of parameters on a real instance require 45 minutes to measure write amplification.

1.2.4. Objective

Finding the optimal configuration for RocksDB is a long and arduous task that depends on hardware, workload, Key Performance Indicator (KPI) and basic performance requirements. The usual KPIs of interest are amplification factors (read, write and space amplification). Read amplification is the average number of disks read per query, write amplification is the ratio of data written to storage to data sent to RocksDB, and space amplification is the sum of bytes written to storage to number of bytes sent to RocksDB. Other KPI's can be of interest, such as throughput (read, write or a combination of both) latency and tail latency, latency being the response time of the system and tail latency is usually the 90th quantile or 99th quantile longest response times. Furthermore, system utilization (CPU, bandwidth and memory utilization) can be used as KPI as users might be running other resource heavy workload on the same system and minimizing the resources used by an instance of RocksDB can free resources for other applications.

1.3. Observations

```
2019/01/29-10:13:09.998658) EVENT_LOG_v1 {"time_micros": 1548774789998651, "job":
1075, "event": "compaction_finished", "compaction_time_micros": 1624173,
"output_level": 4, "num_output_files": 4, "total_output_size": 284470987,
"num_input_records": 617033, "num_output_records": 614428, "num_subcompactions": 1,
"num_single_delete_mismatches": 0, "num_single_delete_fallthrough": 0, "lsm_state":
[3, 0, 8, 32, 238, 1589]}
2019/01/29-10:13:10.001703 7f81313ff700 [db/compaction_job.cc:1153] [default] [JOB
1076] Generated table #135591: 39995 keys, 18516303 bytes
2019/01/29-10:13:10.001726 7f81313ff700 EVENT_LOG_v1 {"time_micros":
1548774790001713, "cf_name": "default", "job": 1076, "event":
"table_file_creation", "file_number": 135591, "file_size": 18516303,
"table_properties": {"data_size": 18342184, "index_size": 229901, "filter_size":
50117, "raw_key_size": 1119860, "raw_average_key_size": 28, "raw_value_size":
31996000, "raw_average_value_size": 800, "num_data_blocks": 7999, "num_entries":
39995, "filter_policy_name": "rocksdb.BuiltinBloomFilter", "kDeletedKeys": "0",
"kMergeOperands": "0"}}
```

Fig. 1.2. A small extract of the log file from which required metrics were measured.

For our experiments, our goal was to optimize the write throughput of RocksDB with no constraints on other KPIs, amplification factors or system utilization. We extracted information from the log file generated by RocksDB to analyze the internal behavior of RocksDB. The log file contains information about compactions (which files were used, which files were created, the time the compaction started, the time it finished, sizes of each file, etc), similarly for the flush operation (which memtables were used and their sizes, which SSTables were created, time it started and finished, etc). Also, the log file contains information about which threads were involved in each operation, slowdown and shutdown trigger information, see Figure 1.2 for an extract.

The compaction times were highly correlated to the input size with a correlation coefficient of 0.999, as shown in Figure 1.3. The Compaction time was 5.54 MB per milliseconds.

Due to the fixed size of the memtables and fixed number of immutable memtable required to be flushed, all flushes involved a similar number of keys and the relation between the flush time and size, both in terms of bytes and number of keys, was not found to be linear. The average time for a flush was 0.27 seconds and the distribution of the time was asymmetric, see Figure 1.4.

1.4. Model

Three parts of RocksDB algorithm were thought to possibly be limiting the put rate during the benchmark, the flushing threads utilization, the compaction threads utilization and the level 1 utilization.

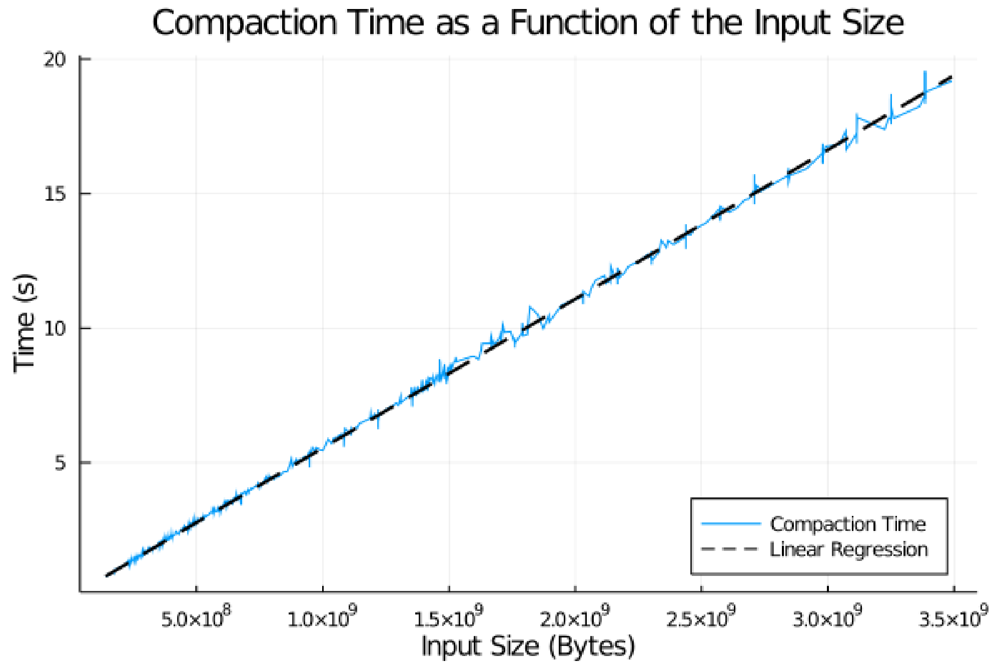


Fig. 1.3. Algorithmic complexity of merging sorted lists is $\mathcal{O}(n)$ with respect to the total size of the list to be merged.

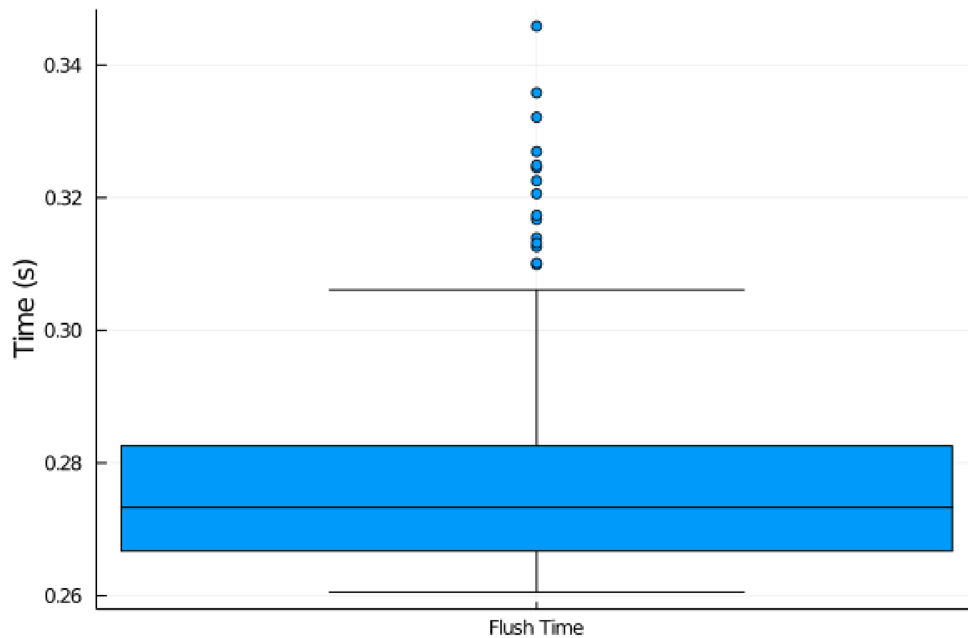


Fig. 1.4. The distribution of the time between the creation and the deletion of a memtable is asymmetric.

1.4.1. Flush Thread Utilization

The first possible bottleneck studied was the flush threads utilization. This possible bottleneck would appear in the cases where the put rates would exceed the rate at which the flushing thread can flush KV pairs to the level 1 through the flushes of immutable memtables.

The maximum rate at which one flushing thread can flush keys from the memtable into level 0 files is given by

$$\lambda_F^1 = \frac{N_M \cdot K_M}{t_F},$$

where N_M, is the average number of keys in the memtables, K_M, the average number of memtables per flushes and t_F, the average time to execute a flush operation. This maximum rate must be multiplied by the number of flushing threads T_F in use, leading to a maximum flush rate of

$$\lambda_F = T_F \lambda_F^1 = \frac{T_F \cdot N_M \cdot K_M}{t_F}.$$

In our run of RocksDB with seven flushing threads, we observed the transfer to storage of one memtable in average per flush, an average of 127098 keys per memtable, an average flushing time of 0.27 seconds, and a put rate of 71,682 keys per second while the upper bound delimited by the flushing threads was

$$\lambda_F = \frac{7 \cdot 1 \cdot 127098}{0.27} = 3{,}295{,}133.$$

This means that the observed put rate of the system was well under the maximum flushing rate of the flushing threads and an optimal configuration of RocksDB would have a smaller number of flushing threads.

1.4.2. Compaction Thread Utilization

The second possible bottleneck in the RocksDB architecture is the compaction thread that is either not fast enough or not numerous enough to move keys from level 0 to deeper levels, causing the system to hit a slowdown trigger.

We assume that $R_{i \to j}$, the ratio of compaction that occurs between levels i and j is known and fixed for all levels i, j. The maximum rate at which one thread can execute compaction λ_C^1 has a similar form as the maximum flushing rate and is given by

$$\lambda_C^1 = \frac{1}{t_C},$$

where t_C is the average time for a compaction. Note that λ_C^1 is a measure in compaction per second while λ_F^1 and λ_F were measures in key per second. The maximum rate of compaction of T_C compaction threads is then

$$\lambda_C = T_C \cdot \lambda_C^1.$$

From the maximum rate of compaction, we can extract the maximum rate of compaction between level 0 and level 1 as

$$\lambda_{0\to1} = R_{0\to1}\lambda_C,$$

since a proportion $R_{0\to1}$ of compaction must occur between level 0 and level 1. Due to overlapping key ranges of files in level 0, all files of level 0 are involved during level 0-1 compactions. The maximum rate at which keys are passed from level 0 to level 1 will be called dispersion rate λ_0 and is given by

$$\lambda_0 = \lambda_{0\to1} \cdot N_0 \cdot K_0,$$

where N_0 is the average number of files at level 0 when compaction 0-1 occurs and K_0 is the average number of keys per file in level 0.

Putting all the equations together, we have

$$R_{0\to1} \cdot \frac{T_C}{t_C} \cdot N_0 \cdot K_0,$$

where λ_0 is the maximum rate at which the level 0 moves keys in level 1. In our experiment, we have observed

$$\lambda_0 = 0.08728 \cdot \frac{16}{7.25} \cdot 4.54320 \cdot 178584 = 156{,}279.$$

While this upper bound is much lower than the flushing rate upper bound, it is still more than twice the observed put rate. Furthermore, we argue that reducing the number of flushing threads might have a detrimental effect on the performance of RocksDB by putting too much pressure on the utilization of the compaction threads.

1.4.3. Level 1 Utilization

The last possible bottleneck that we see in the architecture of RocksDB is in the level 1 utilization. This bottleneck comes from the fact that for a compaction to occur, RocksDB must find free files (files that are not already involved in a compaction) in oversized levels. While for most levels this is perfectly fine as multiple compactions can occur simultaneously on non overlapping key regions of a level, compactions from level 0 to level 1 involve all, or almost all the files. If at a given time, RocksDB is ready to do a compaction between level 0 and level 1 and some files in level 1 are already used in a compaction to level 2, RocksDB will have to delay the level 0-1 compaction causing the system to accumulate more files at level 0 during this delay and hitting a slowdown trigger. More generally, it appears that RocksDB utilization will consist of cycles. During the first part of the cycle, a compaction between level 0 and level 1 will occur, raising the size of level 1 and will follow compactions between levels 1 and 2, see Figure 1.5 and 1.6. For this reason, it seems reasonable that the level 1 utilization could be the bottleneck in RocksDB architecture.

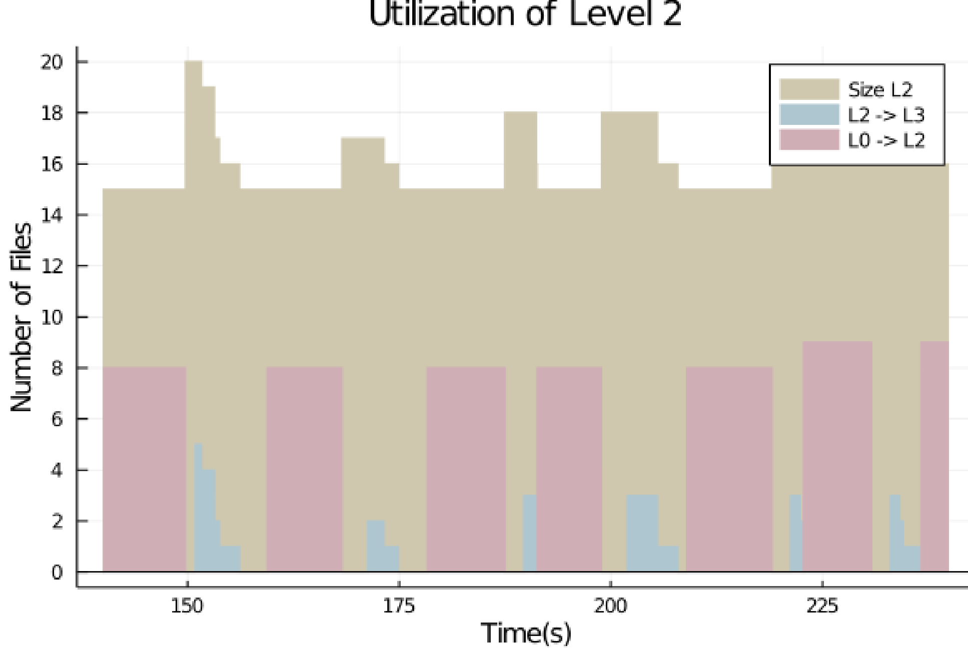


Fig. 1.5. Graphical representation of the behavior of the files in a level during a cycle

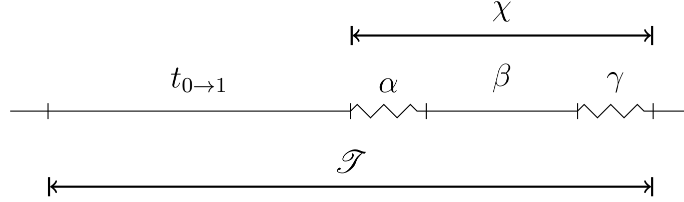


Fig. 1.6. Graphical representation of a cycle

This bottleneck assumes that the cycles are too long and limit the rate at which the level 1 can absorb KV pairs from level 0. The time required to cycle is composed of four parts: the time to execute a compaction from level 0 to level 1 $t_{0\to1}$, the time α required to acquire the required resources for a compaction from level 1 to level 2, the time β to execute all the compactions between level 1 and level 2 and finally, the time γ to acquire the resources for a compaction between levels 0 and 1. The cycle ends when the next compaction between level 0 and level 1 starts, see Figure 1.6. To simplify the notation, we will call χ the total time between the end of a compaction from level 0 to level 1 and $\mathscr{T}$ the total time for a cycle. χ and $\mathscr{T}$ satisfy the relations

$$\chi = \alpha + \beta + \gamma,$$
$$\mathscr{T} = t_{0\to1} + \alpha + \beta + \gamma.$$

The number of keys absorbed by level 1 during a cycle is simply the number of keys inserted to RocksDB during the previous period. Assuming that each cycle takes the same time and

that the number of keys in level 0 during compaction n_0 is constant, the maximum rate at which keys can be moved to level 1 is

$$\lambda_1 = \frac{n_0}{\mathscr{T}}.$$

During our test of RocksDB, we have observed $\alpha = 1.563$, $\beta = 3.574$, $\gamma = 1.269$, and $\chi = 6.407$, giving us a period of $\mathscr{T} = 12.81$ seconds during which the average number of keys in a file in level 0 is 237490 and the average number of files from level 0 in a compaction 0-1 is 4.543, meaning that the average number of keys being moved to level 1 per cycle is $4.543 \cdot 237490 = 1.078 \cdot 10^6$. Thus, level 1 can absorb 84224 keys per second which is only about 17% higher than the observed throughput.

1.4.4. Parameter Selection

For the current system, running the same workload, we argue that many of the observed measurements will not change significantly when changing most of RocksDB's settings. For instance, the speed of compaction (in MB per seconds) is hardware dependent and should not change when changing most settings. However, the average flush time will change when modifying the write buffer size. While there is a linear relation between the compaction time and its input size, the same is not known to be true yet for the flushing time and further investigation will be required to find such a relation. The flushing time should however be independent of most other settings. The distribution of α and γ are synchronization dependent, they depend on the time required for the system to acquire needed resources to execute a compaction and should slightly vary with changes in settings. For simplicity purpose, we will assume that they stay constant when varying other settings. Considering all those parameters as constants can help us suggest better settings to optimize the put throughput for this workload on this architecture.

Since we observed such a low utilization of the flushing threads, we argue that **reducing the number of flushing thread** will not decrease the throughput while decreasing the CPU utilization. Based on our equations, a single flushing thread might be enough to keep up with the current write throughput. Reduction of the number of flushing threads is coherent with the literature [**?**]. Furthermore, the maximum rate of at which a single flushing thread can disperse the keys in level 0 is 470 thousand KV pairs per second which is much higher than the current insertion rate. We then argue that a single flushing thread would be sufficient with this insertion rate.

While slightly under utilized, the compaction threads were utilized too much for their number to be reduced and that reducing their number might impact the write throughput of the system. We then propose to **leave the number of compaction threads at their current number**.

It is more difficult to alleviate the level 1 utilization bottleneck from the system. For this reason, if we allowed the level 1 to be bigger, the distribution of α and γ would stay the same but compactions involving the level 1, both $t_{0\rightarrow1}$ and β would be longer resulting in a better utilization of the level 1 and a better write throughput.

1.5. Exploration

Tuning involves many KPIs, either as a form of multi-objective optimization, or implementing requirement constraints such as keeping a KPI under a given limit while maximizing one or more other KPIs. Multi-objective optimization can be complex because the change in a parameter can have beneficial effects on a given KPI while deteriorating others. For example, enabling hashing can significantly reduce latency, tail latency and bandwidth usage but will increase CPU and memory utilization. This means that for most use cases, no trivial solution to maximize all KPIs exists. The case where KPIs are used as constraints is not trivial either. In this case, KPI constraints can generate problems were no solution exists or the available set of settings is small, garnering poor solutions.

Furthermore, this works opens the door for many other projects. First of all, we believe that the problem can be extended into two-steps. During the first step, the user optimizes his architecture on which the LSM-tree based algorithm will run. In the second step, the user optimizes his LSM-tree based storage engine settings. We argue that such a methodology can not only help the user make better choices when determining which hardware to buy, but it also predicts the performance it will be able to achieve. To achieve such results, further works must be done, exploring the optimization of other KPIs, and the exact impact of other settings on the system performance.

This work can also be extended to in-storage optimization. Modern SSDs use complex algorithms to take advantage of their parallel architecture. SSDs and their Flash Translation Layer (FTL) specifically, behave in a lot of ways similar to KV storage engines. That is, logical block addresses behave as a key and physical block addresses behave as values. Non-optimal FTL management has created the need for open channel SSDs. We argue that the FTL management problem is not so far of the optimization of a regular LSM-tree based tuning algorithm and can give boosts in performance if further studied.

1.6. Conclusion

We argue that our method to optimize the settings of RocksDB using the log files is new and has the potential to help many accelerate their search for better settings. It also achieves gains in performance without having to run many different parameter configurations, thus reducing the number of runs required to find a suitable solution. Our analysis of the log file allowed us to find and measure the linear correlation between the time of a single

compaction process and its input size. From our measurements, we were able to build three sets of equations linking the maximum write throughput to many RocksDB settings which allowed us to propose better settings for RocksDB that fit our workload on our architecture without the need to run all the possible combinations of settings, a technique often used to optimize RocksDB performance.

Chapitre 2

Statistical Learning

2.1. Introduction

We consider unconstrained optimization problems of the form

$$\min_{x \in \mathbb{R}^p} f(x) = \mathbb{E}_\xi \left[F(x, \boldsymbol{\xi}) \right] \tag{2.1.1}$$

where the expectation is taken with respect to the distribution of $\boldsymbol{\xi}$. To distinguish between random data and their numerical values we will use the bold script like $\boldsymbol{\xi}$ for the random vector, and ξ for a particular realization $\boldsymbol{\xi}$. The functions $F(\cdot, \xi) : \mathbb{R}^p \to \mathbb{R}$, for *almost every* $\xi \in \mathbb{R}^m$, are deterministic, differentiable, relatively cheap to compute and can be nonconvex.

We will focus on statistical learning problems which involve smooth sample average problems where gradients are computed relatively easily. Such examples arise in estimation of *mixed-logit models* (MLM) [**?**], *maximum likelihood estimation* (MLE), data-fitting problems with least-squares regression, or in *Machine Learning* (ML) with the training of deep *Neural Networks* (NN) for multiclass classification problems in the presence of very large data sets (see [**?**] for a survey on ML problems).

However, the analytical expression for the mathematical expectation (2.1.1) is rarely available, or even unknown, for various reasons. In the stochastic optimization literature, the distribution of $\boldsymbol{\xi}$ is typically known, but (2.1.1) requires the computation of multidimensional integrals in the continuous case, which is numerically impracticable, and, in the discrete case, the number of realizations (also callled scenarios) of $\boldsymbol{\xi}$ can be very large or even infinite. In machine learning problems, the distribution is usually unknown and (2.1.1) is replaced by the empirical risk on the available observations. One possibility for dealing with this issue is to approximate the original objective function using a sample $\{\xi_1, \ldots, \xi_N\}$ of N *independently and identically distributed* (i.i.d.) realizations of $\boldsymbol{\xi}$. We then form the sample

average approximation (SAA) problem

$$\min_{x \in \mathbb{R}^p} \tilde{f}_N(x,\xi_1,\dots,\xi_N) = \frac{1}{N}\sum_{i=1}^{N} F(x,\xi_i). \tag{2.1.2}$$

As underlined in (2.1.2), $\tilde{f}_N(x,\xi_1,\dots,\xi_N)$ is a realization of the random variable $\tilde{f}_N(x,\boldsymbol{\xi_1},\dots,\boldsymbol{\xi_N})$. For simplicity, we will omit from now the dependence on the random variables, which should be understood from the context, leading to the problem

$$\min_{x \in \mathbb{R}^p} \tilde{f}_N(x) = \frac{1}{N}\sum_{i=1}^{N} F(x,\xi_i). \tag{2.1.3}$$

Since each $\boldsymbol{\xi_i}$ follows the same probability distribution than $\boldsymbol{\xi}$, we have that for any $x \in \mathbb{R}^p$, $\mathbb{E}_{\boldsymbol{\xi}}[F(x,\boldsymbol{\xi_i})] = f(x)$ and hence $\mathbb{E}[\tilde{f}_N(x)] = f(x)$, ensuring $\tilde{f}_N(x)$ to be an unbiased estimator of $f(x)$. Moreover, the application of the pointwise Law of Large Numbers (LLN) implies that $\tilde{f}_N(x)$ converges *with probability one* (w.p.1) to $f(x)$ as $N \to \infty$. In that case, $\tilde{f}_N(x)$ is said to be a *strongly consistent* estimator of $f(x)$.

Problem (2.1.3) can be solved with standard optimization methods since it is deterministic. The approach is referred to as SAA or *sample-path optimization*. Classical assumptions to ensure that SAA optimization exhibits good asymptotic behavior are the uniform law of large numbers on the approximate function and its gradient, implying that $\mathbb{E}_{\boldsymbol{\xi}}[\nabla F(x,\boldsymbol{\xi})] = \nabla f(x)$, and under additional conditions, $\mathbb{E}_{\boldsymbol{\xi}}[\nabla^2 F(x,\boldsymbol{\xi})] = \nabla^2 f(x)$. By optimizing the SAA problem, one can hope to obtain an approximate solution that is close to a solution of the true stochastic problem when N is large enough.

For instance, in maximum likelihood estimation with discrete variables, we aim to maximize, or equivalently minimize the opposite of, the probability that a set of i.i.d. random variables $X_i, i = 1,\dots,N$ have taken the values $x_i, i = 1,\dots,N$. Assuming that the i-th observation X_i has taken the value x_i with the probability $\mathcal{F}(x,x_i)$ for an unknown and searched parameter vector θ, the joint probability to observe the realizations $x_1, x_2, \dots, x_n$ is given by

$$\prod_{i=1}^{N} \mathcal{F}(\theta, X_i).$$

The logarithmic operator allows to transform this product into a sum, facilitating the computation of the gradients while improving the numerical stability and preserving the argument of the maximum. Dividing by N, we form the average log-likelihood function

$$\ell(\theta) = \frac{1}{N}\sum_{i=1}^{N} \log \mathcal{F}(\theta, X_i),$$

which has the form required by (2.1.3) by taking $F(\theta,\xi_i) := -\log \mathcal{F}(\theta,\xi_i)$.

Another family of algorithms to consider are *stochastic approximation* (SA) methods. They are iterative procedures which sample small independent data sets at every iteration

to efficiently approximate properties of f such as zeros or extrema. These methods have been widely studied in the last decades, and have become very popular within the ML community. In particular, we can mention first-order methods like stochastic gradient descent, whose convergence properties were analyzed in depth by ?], its variants that use averaging of gradients [?], averaging of iterates [?], momentum and acceleration [? ? ?] or adaptive learning rates [? ?]. We refer the reader to ? ?] and references therein for examples and theoretical details. These methods only require moderate computational cost per iteration allowing for scalable training on large datasets.

The choice between these two extremes, SAA and SA, outlines the well-known tradeoff between inexpensive noisy steps and expensive but more reliable steps. The availability of parallel computing environments, however, has developed a fairly different tradeoff. It has become more reasonable to consider large samples as the Monte Carlo estimates of the function and gradients are sums for which each term can be computed in parallel.

The need to make SAA implementable, computationally efficient and competitive with SA methods have led to various refinements such as *Retrospective Approximation* (RA) or *variable sample methods*. The idea is to use relatively rough approximations at early stages of the optimization procedure and gradually increase the accuracy at the late stages to reach high accuracy. This way one hopes to save computational effort and yet solve the original problem eventually by benefiting from the convergence theory from SAA.

Homem-de-Mello [?] provides general results on the scheduling problem, under which variable-sample methods yield consistent estimators as well as bounds on the estimation error. Other authors [? ? ?] increase the sample that approximates the objective function by a certain percentage in each iteration. In particular, ?] provide guidance on how much sampling should be undertaken under various contexts in order to ensure that the resulting iterates are provably efficient, where "efficient" implies convergence at the fastest possible theoretical rate. Another approach [? ? ? ?] consists to propose dynamic sample sizes in a linesearch framework to ensure that the sampled gradient is close enough to the true gradient to have a decrease in the true objective function sufficiently often. In [? ? ? ? ? ?], the authors adjust the sample size to make sure that the progress is statistically significant and not only due to noisy estimates. Roughly speaking, the function value of the next iterate should be outside the confidence interval of the current function value. Finally, some authors provide more elaborate schemes. For example, ?] design a variable-number sample-path scheme which integrates Bayesian techniques to determine a satisfactory non-decreasing sequence $\{N_k\}$ to ensure the accuracy of the approximation, while ?] develops a discrete-time optimal-control problem to adaptively select sample sizes. We position our work in between SA and SAA with a variable sample-path method, where the Monte Carlo sample size is dynamically updated at every iteration to ensure a decrease in the true objective function with high probability. In the following, the relation of our method to works mentioned above

is clearly stated. Note that the sample size update can be non-monotone. For instance, when leaving a critical point neighborhood, like a saddle point, the sample can be decreased.

After discussing solutions for approximating the objective function, a reasonable question is: which algorithm should be considered? Facing the computational challenges brought by the large scale nature of modern *big-data* problems [**?**], many of the recent research efforts have been centered around designing variants of classical algorithms which improve upon the cost per iteration, while maintaining the original iteration complexity. First-order methods have been very popular for the interesting trade-off they offer between their relatively simple implementation and their good convergence properties. For example, Keskar and Socher [**?**] highlight that state-of-the-art optimizers, like Adam, Adagrad or RMSprop, have been found to generalize poorly compared to SGD at later stages of training in ML problems. Moreover, they suffer from several issues. They are sensitive to poor initialization [**?**], and their performance highly depends on hyperparameters values, which can be cumbersome to tune [**?**]. Also, they can be blind to pathological curvature, showing slow progress in regions where the Hessian matrix is ill-conditioned like flat regions. Therefore, second-order stochastic methods have also been proposed, because using knowledge of the curvature allows to both escape more easily from such regions, and provide adaptive learning rates by re-scaling the gradient. They also need far fewer iterations as they benefit from better rates of convergence. Nevertheless, their main drawback is that, due to the huge number of parameters in the model, it is practically impossible to compute, store or worse, solve a linear system involving the Hessian matrix. Within this context, the central question is how to select a useful training sample for computing Hessian estimates or Hessian-vector products that is significantly smaller than the sample used for function and gradient computations, but still keeps the good convergence properties from second-order methods. A good analysis is proposed by Bellavia and al. [**?**] in a context of optimizing large sums of convex functions with linesearch procedures. A more general framework for Hessian subsampling considerations can be found in [**? ?**]. Also, trust-region methods have been shown in the last decades to be very efficient, and exhibit good convergence properties [**?**]. These algorithms progress by minimizing a model of the objective function in a region around the current iterate where the model is deemed of good quality, and the region can be adaptively expanded or shrinked accordingly. Finding the solution to the constrained model minimization is referred to as solving the trust-region *subproblem*. A popular approximation used is the truncated second-order Taylor expansion of the objective. The goal of trust region is to extend convergence properties of Newton methods. At each iteration, when using a quadratic model, the step can be efficiently obtained using the *truncated conjugate gradient* method, which only requires Hessian-vector products, and does not require the computation of the Hessian matrix.

Recent works study subsampled Newton-type methods in trust region and cubic regularization, showing great theoretical and empirical potential [**? ? ?**]. **?**] stress the

importance of adaptive methods in stochastic optimization to overcome the constraint of hyperparameters which are hugely time demanding to fine tune and provide general analysis frameworks for stochastic trust region and linesearch. **?**] extend the work of **?**] and **?**], and provide convergence rates of stochastic trust-region algorithms, where random models of the objective function are used at each iteration to compute the next potential iterate. Borrowing ideas from *derivative-free optimization*, the approximating models are only assumed to be sufficiently accurate with high probability, and no considerations are made on their construction process. The study shows that convergence properties of standard optimization methods, like trust-region algorithms, can be transferred to their stochastic counterparts under mild assumptions. We have therefore decided to integrate our variable-sample strategy in a subsampled Hessian-free trust-region method. *Outer-product* (OP) approximations will be used to efficiently compute Hessian-vector products, and an estimate of the solution of the trust-region subproblem will be obtained by means of the truncated conjugate gradient technique [**? ?**].

2.2. Mathematical Formulation

2.2.1. Notations and Basic Assumptions

We denote by k the iteration index, and x_k the current iterate. $\|\cdot\|$ will denote the L_2-norm, specifically $\|x\| = \sqrt{\sum_{j=1}^{p} x_{(j)}^2}$, where $x_{(j)}$ is the jth Cartesian coordinate, and $\|\cdot\|_1$ the L_1-norm defined by $\|x\|_1 = \sum_{j=1}^{p} |x_{(j)}|$. f_k, g_k and H_k refer to $f(x_k)$, the gradient $\nabla f(x_k)$ and the Hessian matrix $\nabla^2 f(x_k)$, respectively. The stochastic nature of the problem allows us to consider SA or SAA estimates, so we will use the notation $\tilde{\cdot}$ to denote Monte Carlo approximations and the distinction will be made by the index. Formally, when considering SA estimates, we will have $\mathcal{S}_k = \{1, 2, \ldots, N_k\}$ such that $\tilde{f}_k = \frac{1}{N_k} \sum_{i \in \mathcal{S}_k} F(x_k\,,\, \boldsymbol{\xi}_i)$ where $\{\boldsymbol{\xi}_i\}_{i=1,\ldots,N_k}$ are i.i.d replicates of the random variable $\boldsymbol{\xi}$, and $\tilde{g}_k = \nabla \tilde{f}_k$. In a SAA situation, we will draw a sample $\mathcal{S}_k \subset \{1, \ldots, N\}$ of size $|\mathcal{S}_k| = N_k$ to compute the function estimate $\tilde{f}_k = \frac{1}{N_k} \sum_{i \in \mathcal{S}_k} F_i(x_k)$ where $F_i(x_k) = F(x_k, \xi_i)$ and the gradient approximation $\tilde{g}_k \stackrel{def}{=} \nabla \tilde{f}_k = \frac{1}{N_k} \sum_{i \in \mathcal{S}_k} \nabla F_i(x_k)$. To simplify notations we will usually write $\tilde{f}_k$ or $\tilde{g}_k$ for both situations, and will specify when necessary. The sample used to compute $\tilde{f}_k$ and $\tilde{g}_k$ will be the same, except stated otherwise. Similarly, a sample $\mathcal{H}_k$, which can be a subset of $\mathcal{S}_k$ or not, will be drawn to calculate a subsampled version of the Hessian or its approximation.

When considering a sequence $(\boldsymbol{Y}_n)$ of random variables, we will write $\boldsymbol{Y}_n \xrightarrow{d} \boldsymbol{Y}$ whenever $(\boldsymbol{Y}_n)$ converges *in distribution* to a random variable $\boldsymbol{Y}$, $\boldsymbol{Y}_n \xrightarrow{p} \boldsymbol{Y}$ when the convergence is *in probability*, and $\boldsymbol{Y}_n \xrightarrow{\text{a.s.}} \boldsymbol{Y}$ when the convergence holds *almost surely* (or with probability one).

2.2.2. Assumptions

Some assumptions must be made to ensure the good behavior of the optimization algorithm.

(1) f is bounded from below on $\mathbb{R}^p$.
(2) The iterates x_k for $k \in \mathbb{N}$ stay in a compact set $\Theta \subset \mathbb{R}^p$.
(3) For $\mathbb{P}$-almost every ξ, the function $F(\cdot\,,\,\xi)$ is continuously differentiable on Θ, and for every $x \in \mathbb{R}^p$, $F(x\,,\,\cdot)$ is $\mathbb{P}$-mesurable.
(4) The family $F(x,\xi)$, $x \in \mathbb{R}^p$, is dominated by a $\mathbb{P}$-integrable function $K(\xi)$, i.e. $\mathbb{E}_{\mathbb{R}}[K]$ is finite and $|G(x,\xi)| \leq K(\xi)$ for all $x \in \mathbb{R}^p$ and $\mathbb{P}$-almost every ξ.
(5) The Hessian of $f(\cdot)$ is uniformly bounded on $\mathbb{R}^p$, i.e. there exists a constant $\kappa_{bhm} > 0$ such that

$$\forall x \in \mathbb{R}^p,\ \|\nabla^2 f(x)\| \leq \kappa_{bhm}$$

Consequently, ∇f is Lipschitz-continuous with constant L.
(6) There exists a constant $\kappa_{bhhh} > 0$ such that

$$\forall x \in \mathbb{R}^p,\ \text{ for } \mathbb{P}\text{-almost all } \xi \in \mathbb{R}^m,\ \|\nabla F(x,\xi)\nabla F(x,\xi)^T\| \leq \kappa_{bhhh}.$$

It is important to note that assumptions 2, 3 and 4 together imply that there exists a *uniform law of large numbers* (ULLN) on Θ for the approximation $\tilde{f}_N(x)$ of $f(x)$ (see [**?**]), meaning that

$$\sup_{x\in\Theta}\left|\tilde{f}_N(x) - f(x)\right| \xrightarrow{a.s.} 0.$$

Assumption 3 obviously implies that $F(\cdot\,,\,\xi)$ is continuous $\mathbb{P}$-almost surely. This and assumption 4 are typical assumptions of stochastic programming theory (see for instance [**?**]). The stronger form of 3 is justified by our interest in first-order optimality conditions, which requires the objective function gradient.

2.2.3. Basic Trust-Region model

Our optimization scheme is based on the basic trust-region algorithm (see **?** , Chapter 6] for more details) and its main components are presented in Algorithm 1. At each iteration k, a model $m_k(s)$ is built around the current iterate x_k, which serves as a local approximation of f in a ball $\mathcal{B}_k \stackrel{def}{=} \{x_k + s \,|\, \|s\| \leq \Delta_k\}$. A popular choice is the quadratic approximation:

$$m_k(s) = f_k + g_k^T s + \frac{1}{2} s^T B_k s \tag{2.2.1}$$

where B_k is a symmetric approximation of the Hessian matrix $\nabla^2 f(x_k)$.

The model $m_k(s)$ is (approximately) minimized in $\mathcal{B}_k$ to produce a step s_k computed by the truncated conjugate gradient. The ratio ρ_k defined in (2.2.2) aims to measure the

adequacy between the objective function decrease and the model decrease. If the model fits the function well ($\rho_k \geq \eta_2$), the trust region can be expanded. On the other hand, if the model differs greatly from the function, that means that the trust region was too large and hence should be reduced. If ρ_k is very small ($\rho_k < \eta_1$), the step is rejected and the radius is decreased. Trust-region algorithms aim to extend Newton method properties with global convergence guarantees. Notably, far from the solution, if the model is not very accurate, the trust-region step will have performance similar to gradient descent. But, close to a local minimum, if the model Hessian approximates the true Hessian with increasing precision as the iterates get closer to the critical point, the trust-region boundaries will become inactive and the steps will become similar to Newton step, allowing for local superlinear convergence rates.

Algorithm 1 Basic Trust-Region Algorithm (BTR)

Step 0: Initialization: An initial point x_0 and an initial trust-region radius Δ_0 are given. The constants η_1, η_2, γ_1, and γ_2 are also given and satisfy

$$0 < \eta_1 \leq \eta_2 < 1 \quad \text{and} \quad 0 < \gamma_1 \leq \gamma_2 < 1 \quad \text{and} \quad \gamma_3 \geq 1.$$

Compute $f(x_0)$ and set $k = 0$.

Step 1: Model definition: Define a model m_k in $\mathcal{B}_k$.

Step 2: Step calculation: Compute a step s_k that "sufficiently reduces the model" m_k and such that $x_k + s_k \in \mathcal{B}_k$.

Step 3: Acceptance of the trial point: Compute $f(x_k + s_k)$ and define

$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)}. \tag{2.2.2}$$

If $\rho_k \geq \eta_1$, then define $x_{k+1} = x_k + s_k$; otherwise define $x_{k+1} = x_k$.

Step 4: Trust-region radius update: Set

$$\Delta_{k+1} \in \begin{cases} [\gamma_3 \Delta_k, \infty] & \text{if } \rho_k \geq \eta_2; \\ [\gamma_2 \Delta_k, \Delta_k] & \text{if } \rho_k \in [\eta_1, \eta_2); \\ [\gamma_1 \Delta_k, \gamma_2 \Delta_k] & \text{if } \rho_k < \eta_1. \end{cases}$$

Increment k by 1 and repeat **Steps 1 to 4** until a stopping criterion is met.

2.2.4. Trust-region subproblem

The main computational bottleneck in BTR is the calculation of s_k in step 2, because, when considering unconstrained optimization, solving a quadratic model is equivalent to finding the solution of a linear system if the model is strictly convex. One popular method is the classical CG method as it only involves matrix-vector products and if the system is well conditioned, a point close to the solution can be found in a few iterations, at a much lower computational cost. However, if we do not know if our quadratic model is strictly convex, we must take precautions to deal with nonconvexity if it arises. To tackle all these

issues, the Steihaug-Toint Truncated Conjugate Gradient (TCG) [**?** **?**]. In this technique a sequence of iterates is produced. If the model minimizer lies within the trust region and the matrix is well conditioned, an approximation is obtained after a few iterations. Otherwise, when the minimizer lies outside the trust region or if a negative curvature is encountered, the solution is a well chosen point on the boundary and the procedure stops. The last iterate computed will be the trial step s_k considered in step 3 (see **?** , Chapter 5] for more details). The first iterate generated is the *Cauchy point*, which is the model minimizer along the steepest descent direction, and subsequent iterates give lower values of the model. Thus, for $k > 0$, each solution s_k satisfies the required condition of "sufficient decrease" in the model guaranteeing the global convergence to a first-order critical point in a deterministic setting.

2.2.5. Approximation of the Hessian matrix

In some statistical learning situations, like maximum likelihood estimation or least-squares regression, it is possible to consider an approximation of the Hessian, called *outer-product* (OP) due to its particular structure

$$\tilde{B}_k \stackrel{def}{=} \frac{1}{|\mathcal{H}_k|} \sum_{i \in \mathcal{H}_k} \nabla F_i(x_k) \nabla F_i(x_k)^T,$$

where $\tilde{B}_k$ is the approximation of the Hessian matrix. The terms $\nabla F_i(x_k)$ will generally be already computed as they are required for the sampled gradient, and we will assume throughout the paper that $\mathcal{H}_k \subset \mathcal{S}_k$. The outer-product special structure enables to cheaply compute its product with any vector v, without storing the matrix, by using the following trick,

$$\tilde{B}_k v = \frac{1}{|\mathcal{H}_k|} \sum_{i \in \mathcal{H}_k} \Big(\nabla F_i(x_k)^T v \Big) \nabla F_i(x_k).$$

When considering *maximum likelihood estimation* (MLE), the loss-function can be interpreted as log-probabilities, i.e $F_i(x) = -\log \Big(p(\xi_i|x) \Big)$ where $(\xi_i)_{i \in \{1,\dots,n\}}$ represents the observed data and $p(\cdot|x)$ the *probability distribution function* (PDF) of the data parameterized by x. The problem can be expressed as a stochastic problem of the form $\mathbb{E}_{\boldsymbol{\xi} \sim \pi}[F(x, \boldsymbol{\xi})]$ where π is the true population distribution. An alternative interpretation of the problem is to identify the right probability amongst a family of distributions $\mathcal{F} = \Big\{ p_x = p(\cdot \,|\, x) \ s.t. \ x \in \Xi \subset \mathbb{R}^p \Big\}$.

In a MLE context, an OP approximation of the Hessian, called *BHHH*, can be considered near the solution. This technique was reported in [**?**], and later used by [**?**] for the estimation of mixed-logit model parameters. Formally, if $\widehat{x}_N$ is the *maximum likelihood estimator* of the true parameters x^* then $\sqrt{N}\Big(\widehat{x}_N - x^*\Big) \xrightarrow{d} \mathcal{N}\Big(0, J(x^*)^{-1}\Big)$, where $J(x^*)$ is

the *Fisher information matrix* defined as

$$J(x^*) = \mathbb{E}_{\boldsymbol{\xi}}\Big[\nabla_x \log p(\boldsymbol{\xi}|x^*) \nabla_x \log p(\boldsymbol{\xi}|x^*)^T\Big] = \mathbb{E}_{\boldsymbol{\xi}}\Big[\nabla_x F(x^*, \boldsymbol{\xi}) \nabla_x F(x^*, \boldsymbol{\xi})^T\Big],$$

where $\nabla_x \log p(\boldsymbol{\xi}|x)$ is called the score. If the problem is well specified, meaning that there exists x^* such that $\pi = p(\cdot|x^*)$ and the assumptions to interchange expectation and derivatives are satisfied, the *information identity* property states that $\mathbb{E}_{\boldsymbol{\xi}}[\nabla_x \log p(\xi|x^*)] = 0$, and that the Fisher information matrix represents the variance of the score, and is equal to the expected Hessian matrix at the true parameters

$$J(x^*) = -\mathbb{E}_{\boldsymbol{\xi}}\Big[\nabla_x^2 \log p(\boldsymbol{\xi}|x^*)\Big] = \mathbb{E}_{\boldsymbol{\xi}}\Big[\nabla_x^2 F(x^*, \boldsymbol{\xi})\Big]. \tag{2.2.3}$$

See [**?**] for details. Therefore, we can see from equation (2.2.3) that $J(x^*) = H(x^*)$. It then feels reasonable to consider an approximation of the Hessian of the form,

$$B(x) \overset{def}{=} \mathbb{E}_{\boldsymbol{\xi}}\Big[\nabla_x F(x, \boldsymbol{\xi}) \nabla_x F(x, \boldsymbol{\xi})^T\Big].$$

The BHHH approximation has several advantages over the true Hessian matrix of the function.

* $\tilde{B}_k v$ is cheap to compute because all $\nabla F_i(x_k)$ are already calculated for the gradient estimation $\tilde{g}_k$.
* If the number of observation used to build the BHHH approximation is small compared to the dimension of the problem, there is a significant gain in storage compared to the Hessian.
* $B(x)$ is positive semidefinite. Therefore, if $B(x)$ is invertible, it is positive definite and $B(x)^{-1} g_k$ will be a direction of descent. Even if it is not invertible, it is can be used in a conjugate gradient strategy, and combined with the use of a trust-region approach, it can present the following advantages:
  - if OP is a **poor approximation**, the trust-region radius will shrink, ensuring the step to be close to a steepest descent. However, the computed step will be better, as the Hessian approximation is cheap to obtain, but still better than the identity matrix for which the model minimum is the Cauchy step. Therefore, the update will generally be better than gradient descent.
  - if OP is a **good approximation**, the method will be close to Newton method, potentially achieving a near-quadratic convergence.
* Stalls in the algorithm's progress could indicate that conditions to ensure some statistical properties of OP approximation are not verified. So, a more complex model or of another type should be considered.
* As investigated by [**?**] for the nonlinear least-square situation, secant methods could be used to cheaply estimate the error between the OP approximation and the Hessian matrix.

* Since we use the information matrix as an approximation of the Hessian matrix, it is reasonable to use the Hessian matrix or some approximation of it as an estimate of the information matrix.

2.3. Algorithms

At the beginning of the trust-region algorithm, the iterates may be far from an optimum. Instead of directly using the full dataset, the use of a small initial sample allows rapid progress in the early stages with low computational costs. While a larger sample yields high accuracy near the solution, such accuracy is not required at the beginning of the optimization. Changing the sample size can save computational effort and accelerate the optimization. Two different schemes are proposed to implement this strategy.

2.3.1. Sample size strategy

At iteration k, we compute the next point by $x_{k+1} = x_k + s_k$, where s_k is the step obtained with Steihaug-Toint truncated CG. The sampling strategy aims at deriving a sample size that ensures a decrease in the true objective function with high probability, from the knowledge of the Monte Carlo estimation $\tilde{f}_k(x_{k+1}) - \tilde{f}_k(x_k)$. Formally, it can be stated as follows:

$$\mathbb{P}(\mu_k \leq 0) \geq 1 - \alpha_k \qquad \text{where } \mu_k = f(x_{k+1}) - f(x_k) \text{ and } \alpha_k \in (0\,;\,1).$$

If we assume N_k is large enough, the *Central Limit Theorem* (CLT) implies that:

$$\tilde{\mu}_k - \mu_k \sim \mathcal{N}\left(0\,,\,\frac{\sigma_k^2}{N_k}\right) \tag{2.3.1}$$

where, $\tilde{\mu}_k(\boldsymbol{\xi_1}, \ldots, \boldsymbol{\xi_{N_k}}) = \tilde{f}_k(x_{k+1}, \boldsymbol{\xi_1}, \ldots, \boldsymbol{\xi_{N_k}}) - \tilde{f}_k(x_k, \boldsymbol{\xi_1}, \ldots, \boldsymbol{\xi_{N_k}})$ is the difference between the Monte Carlo estimation of the function between iterates k and $k+1$ and so, an unbiased estimator of the difference of the function at the two points. $\sigma_k^2 = \mathbb{V}\text{ar}_{\boldsymbol{\xi}}\Big(F(x_{k+1}, \boldsymbol{\xi}) - F(x_k, \boldsymbol{\xi})\Big)$ is the variance of the difference of the function estimated at the two points. (2.3.1) implies that:

$$\frac{\sqrt{N_k}}{\sigma_k}\Big(\mu_k - \tilde{\mu}_k\Big) \xrightarrow{d} \mathcal{N}(0\,;\,1) \qquad \text{as } N_k \to +\infty. \tag{2.3.2}$$

Therefore, we can use the one-sided $100(1-\alpha_k)\%$ asymptotic confidence interval to compute the required sample size that generates a decrease in f with at least probability $1 - \alpha_k$ asymptotically, where $\alpha_k \in \Big[0; \frac{1}{2}\Big)$, namely,

$$\mathbb{P}\left[\mu_k \leq 0\right] \geq 1 - \alpha_k \quad \Leftrightarrow \quad \mathbb{P}\left[\frac{\sqrt{N_k}}{\sigma_k}\Big(\mu_k - \tilde{\mu}_k\Big) \leq -\frac{\sqrt{N_k}}{\sigma_k}\tilde{\mu}_k\right] \geq 1 - \alpha_k \tag{2.3.3}$$

Let us denote Φ and Φ^{-1} the cumulative function for the standard Normal distribution and its inverse, respectively, and $z_{1-\alpha_k}$ will refer to the $(1-\alpha_k)$-quantile $\Phi^{-1}(1-\alpha_k)$. Therefore,

the approximation (2.3.2) gives us,

$$\mathbb{P}\left[\frac{\sqrt{N_k}}{\sigma_k}\Big(\mu_k - \tilde{\mu}_k\Big) \leq -\frac{\sqrt{N_k}}{\sigma_k}\tilde{\mu}_k\right] \approx \mathbb{P}\left[Y \leq -\frac{\sqrt{N_k}}{\sigma_k}\tilde{\mu}_k\right] \qquad \text{where } Y \sim \mathcal{N}(0\,;\,1),$$

and,

$$\mathbb{P}\left(Y < -\frac{\sqrt{N_k}}{\sigma_{N_k}}\tilde{\mu}_k\right) \geq 1 - \alpha_k \quad \Leftrightarrow \quad -\frac{\sqrt{N_k}}{\sigma_{N_k}}\tilde{\mu}_k \geq \underbrace{z_{1-\alpha_k}}_{>0} \tag{2.3.4}$$

$$\Leftrightarrow \quad N_k \geq \frac{\sigma_k^2\,(z_{1-\alpha_k})^2}{\tilde{\mu}_k^2} \tag{2.3.5}$$

The equivalence from (2.3.4) to (2.3.5) is justified because, $-\tilde{\mu}_k > 0$ and $\alpha_k \in \Big[0; \frac{1}{2}\Big)$, so $(z_{1-\alpha_k}) > 0$. The next sample size is then

$$N_{k+1} = \left\lceil \frac{\sigma_k^2\,(z_{1-\alpha_k})^2}{\tilde{\mu}_k^2} \right\rceil$$

We now discuss the intuition behind this method and interpret each term in the sample size computation. The variance σ_k^2 of the individual decrease $F(x_{k+1}, \boldsymbol{\xi}) - F(x_k, \boldsymbol{\xi})$ is in the numerator. When the individual decreases agree, the variance will be small and so will the required sample size. Otherwise, a larger sample is needed to avoid collecting "bad" information when there is high variance. $\Phi^{-1}(1-\alpha_k)^2 = (z_{1-\alpha_k})^2$ in the numerator is also explainable. When α_k is close to 1/2, $z_{1-\alpha_k}$ is close to zero, and the samples gets smaller, allowing more deviation for the sampled decrease around its mean. But, when α_k is close to 0, $(z_{1-\alpha_k})^2$ gets bigger and $\tilde{\mu}_k$ is constrained to be close to a *true* decrease μ_k, therefore demanding a bigger sample. Finally, $\tilde{\mu}_k^2$ is in the denominator so that when the iterates get closer to a solution, $\tilde{\mu}_k^2$ will converge to 0 and the sample size will increase to get better accuracy near the optimum.

In a more global consideration, the next sample size N_{k+1} is derived to balance the squared standard deviation of the individual decrease and the squared sampled decrease. This means that, when a step produces a high decrease in the SAA function approximation, it will likely dominate the noise induced by the Monte Carlo estimates, so we can allow ourselves higher variability in the sample estimate with a smaller sample. However, when the estimated decrease is small, there is more sensitivity to noisy estimates, so a bigger sample is required. Furthermore, by examining the formula's structure, one can expect the sample size scheme to exhibit some desirable properties. When iterates get closer to a first-order critical point, the decrease in the SAA estimates gets smaller because the region is locally flat around the critical point. Consequently, the sample size will increase to ∞. This is essential to have better precision near an optimum to ensure convergence to a solution of the *true* problem.

Unfortunately, the true variance σ_k^2 is unknown. It is possible to use the empirical estimate $\hat{\sigma}_{N_k}^2$ computed with the following formula,

$$\hat{\sigma}_{N_k}^2 = \frac{1}{N_k - 1} \sum_{i \in \mathcal{S}_k} \left(\Big(F_i(x_{k+1}) - F_i(x_k) \Big) - \tilde{\mu}_k \right)^2 \tag{2.3.6}$$

When the sample size is large, it can imply high computational costs, but also expensive memory costs as it requires to store all $\{F_i(x_k)\}_{i \in \mathcal{S}_k}$.

We propose an approximation that is algorithmically efficient since it only involves quantities already computed and stored. The value of σ_k is obtained by using a first-order Taylor approximation of $F(x_k, \xi)$ around x_k, for $\xi \in \mathbb{R}^p$,

$$F(x_k + s_k, \xi) \approx F(x_k, \xi) + \nabla F(x_k, \xi)^T s_k$$

and by computing $\tilde{f}_k(x_{k+1})$ accordingly. We obtain a closed expression containing the Fisher information matrix outer-product approximate,

$$\mathbb{V}\mathrm{ar}\left[F(x_k + s_k, \boldsymbol{\xi}) - F(x_k, \boldsymbol{\xi})\right] \approx \mathbb{V}\mathrm{ar}\left[s_k^T \nabla F(x_k, \boldsymbol{\xi})\right] \approx s_k^T\, \mathbb{V}\mathrm{ar}\left[\nabla F(x_k, \boldsymbol{\xi})\right] s_k \tag{2.3.7}$$

And,

$$\begin{aligned}
\mathbb{V}\mathrm{ar}\left[\nabla F(x_k, \boldsymbol{\xi})\right] &= \mathbb{E}\left[\Big(\nabla F(x_k, \boldsymbol{\xi}) - \nabla f(x_k)\Big)\Big(\nabla F(x_k, \boldsymbol{\xi}) - \nabla f(x_k)\Big)^T\right] \\
&= \mathbb{E}\left[\nabla F(x_k, \boldsymbol{\xi}) \nabla F(x_k, \boldsymbol{\xi})^T\right] - \nabla f(x_k) \nabla f(x_k)^T \\
&\approx \tilde{B}_k(x_k) - \nabla \tilde{f}_k(x_k) \nabla \tilde{f}_k(x_k)^T
\end{aligned}$$

Therefore, we will approximate $s_k^T\, \mathbb{V}\mathrm{ar}\left[\nabla F(x_k, \boldsymbol{\xi})\right] s_k$ by,

$$s_k^T \left(\tilde{B}_k(x_k) - \nabla \tilde{f}_k(x_k) \nabla \tilde{f}_k(x_k)^T \right) s_k = s_k^T \tilde{B}_k(x_k) s_k - \left(\nabla \tilde{f}_k(x_k)^T s_k \right)^2 \tag{2.3.8}$$

So,

$$\hat{\sigma}_{N_k}^2 \approx s_k^T \tilde{B}_k(x_k) s_k - \left(\nabla \tilde{f}_k(x_k)^T s_k \right)^2 \tag{2.3.9}$$

From this, we can find a first update rule for the sample size

$$\begin{aligned}
N_{k+1} &= \frac{\left(s_k^T \tilde{B}_k(x_k) s_k - \left(\nabla \tilde{f}_k(x_k)^T s_k \right)^2 \right) (z_{1-\alpha_k})^2}{\tilde{\mu}_k^2} \\
&\approx \frac{\left(s_k^T \tilde{B}_k(x_k) s_k - \tilde{\mu}_k^2 \right) (z_{1-\alpha_k})^2}{\tilde{\mu}_k^2} \\
&\approx \frac{s_k^T \tilde{B}_k(x_k) s_k z_{1-\alpha_k}^2}{\tilde{\mu}_k^2}
\end{aligned}$$

Thus,

$$N_{k+1} = \left\lceil \frac{s_k^T \tilde{B}_k(x_k) s_k (z_{1-\alpha_k})^2}{\tilde{\mu}_k^2} \right\rceil \tag{2.3.10}$$

The constant $z_{1-\alpha_k}^2$ does not impact much the quantity N_{k+1} and is only simpler to code. When examining this expression closely, we can observe that all these quantities are computed at the kth iteration of the algorithm. Indeed, $s_k^T \tilde{B}_k(x_k) s_k$ needs to be computed for the evaluation of ρ_k, and $\nabla \tilde{f}_k(x_k)$ is already stored in memory to construct the trust-region model.

This approximation of the variance has little inconvenient over the true variance (2.3.6). While the approximation is less precise when the length of the step s_k is big, long steps should occur only when we are far from the solution where there is a lot of potential to reduce the true function and it is possible to have significant reduction with small, or minimal sample size. On the other hand, when the step is small and close to the solution, the approximation of the variance gets better and allow an accurate sample size.

If we assume that the BTR sub-problem is solved using the quasi-Newton step, each step s_k is taken in the direction

$$s_k \approx -\tilde{B}_k(x_k)^{-1} \nabla \tilde{f}_k(x_k)$$

Let us consider again the first-order Taylor expansion, but this time for $\tilde{f}_k$ around x_k,

$$\tilde{\mu}_k = \tilde{f}_k(x_k + s_k) - \tilde{f}_k(x_k) \approx \nabla \tilde{f}_k(x_k)^T s_k = \tilde{g}_k^T s_k$$

To simplifiy the notation, we will omit the k index in the following calculation.

$$\begin{aligned}
\frac{\left(s^T \tilde{B} s - \left(\tilde{g}^T s\right)^2\right)(z_{1-\alpha})^2}{\tilde{\mu}^2} &= \frac{\left((-\tilde{B}^{-1}\tilde{g})^T \tilde{B} (-\tilde{B}^{-1}\tilde{g}) - \left(-\tilde{g}^T \tilde{B}^{-1}\tilde{g}\right)^2\right)(z_{1-\alpha})^2}{\left(\tilde{g}^T s\right)^2} \\
&= \frac{\left(\tilde{g}^T (\tilde{B}^{-1})^T \tilde{B} \tilde{B}^{-1} \tilde{g} - \left(\tilde{g}^T \tilde{B}^{-1}\tilde{g}\right)^2\right)(z_{1-\alpha})^2}{\left(-\tilde{g}^T \tilde{B}^{-1}\tilde{g}\right)^2} \\
&= \frac{\left(\tilde{g}^T \tilde{B}^{-1}\tilde{g}\right)\left(1 - \left(\tilde{g}^T \tilde{B}^{-1}\tilde{g}\right)\right)(z_{1-\alpha})^2}{\left(\tilde{g}^T \tilde{B}^{-1}\tilde{g}\right)^2} \\
&= \frac{\left(1 - \left(\tilde{g}^T s\right)\right)(z_{1-\alpha})^2}{(-\tilde{g}^T s)} \\
&= -\frac{(z_{1-\alpha})^2}{\tilde{g}^T s} + (z_{1-\alpha})^2
\end{aligned}$$

Once again, the second term of the last equation is disregarded since the value of $(z_{1-\alpha})^2$ is insignificant in comparison to $\dfrac{(z_{1-\alpha})^2}{\tilde{g}^T s}$. For instance choosing $\alpha = 0.001$ (an extreme value for α as it forces the probability of reducing the true function to be far too high) we have

$(z_{1-\alpha})^2 = 9.55 < 10$ while the first term will be in the order of thousands or hundreds of thousands near the solution. This leads to the next update rule

$$N_{k+1} = \left\lceil \frac{(z_{1-\alpha_k})^2}{\tilde{g}_k^T s_k} \right\rceil \tag{2.3.11}$$

From (2.3.11) some remarks must be made. In the BTR algorithm, if there is long series of bad iterations where the quadratic model is a bad approximation of the function to optimize, the trust region will shrink until the quadratic approximation is reasonably good. For demonstration purpose, we assume that at a given iteration following a series of bad steps, the region size is $\hat{\Delta}$ such that $\hat{\Delta} < 1$ and that the step computed is much smaller than the step found by solving the unconstrained minimization of the quadratic approximation of the function. For this example, we will use the solution $\hat{s}$ given by

$$\hat{s} = \frac{\dot{\Delta}}{\|\tilde{B}^{-1}\tilde{g}_k\|}\tilde{B}^{-1}\tilde{g}_k \tag{2.3.12}$$

where $\dot{\Delta}$ is found by a line search in the Newton direction constrained by the trust-region size.

$$\dot{\Delta} = \min_{\Delta \in [0,\hat{\Delta}]} f_k\left(x_k + \Delta\frac{\tilde{B}^{-1}\tilde{g}_k}{\|\tilde{B}^{-1}\tilde{g}_k\|}\right).$$

Using (2.3.12) as the solution of the trust-region sub-problem, and considering the sample size as a function of the region size $\hat{\Delta}$, the sample size can be arbitrarily large in function of the basic trust region. Using $\hat{\mu}_k = \frac{\tilde{B}^{-1}\tilde{g}_k}{\|\tilde{B}^{-1}\tilde{g}_k\|}$, we have

$$N_{k+1}(\hat{\Delta}) = \frac{(z_{1-\alpha_k})^2}{\hat{\Delta}\cdot\hat{\mu}_k} \leq \frac{(z_{1-\alpha_k})^2}{\dot{\Delta}\cdot\hat{\mu}_k}$$

As $\hat{\Delta} \to 0$, we have

$$\lim_{\hat{\Delta}\to 0} N_{k+1}(\hat{\Delta}) = \infty$$

From this example, we see that the sample size can increase drastically in a case where the region size is too small to generate a step long enough to create a significant reduction in the function. While a special case of the last version of the sampling strategy, the behavior of exploding sample size has been observed in numerical experiments with all sampling strategies developed here and with other solutions of the basic trust sub-problem such has truncated conjugate gradient. This issue forces us to add more restrictions on the sample size algorithm. Namely, we add bounds on the sample size $b_1 N_k \leq N_{k+1} \leq b_2 N_k$ with $0 \leq b_1 \leq 1$ and $1 \leq b_2$. The case where $b_1 = 1$ is simply a non-decreasing sampling and the case with $b_2 = \infty$ is an unbounded dynamic sampling. We suggest $b_1 = 0.75$ and $b_2 = 2$ as bounds for the dynamic sampling algorithm. Other restrictions have been explored such as letting the sample size be changed only if the step s_k is strictly inside the trust region, $\|s_k\| < \Delta_k$, but it did not lead to interesting results.

Due to the large number of sampling strategies and types of restrictions on the sample size, the following nomenclatures will be used in the remaining of this chapter. We first describe the sampling strategies in Table 2.1, then restrictions on the growth and decrease of the sample size in Table 2.2.

Name	Equation	Acronym
Dynamic sampling	(2.3.10)	DS
Newton Based Dynamic Sampling	(2.3.11)	NDS

Table 2.1. Sampling strategies nomenclature

Name	b_1	b_2
Naive	0.75	2
No Smoothing	0	∞
Monotonous	1	2

Table 2.2. Smoothing nomenclature

2.3.2. Relation to other methods

Several works propose dynamic sampling size strategies with similar structures. First, we will present the methods derived in linesearch frameworks. They aim at ensuring a decrease in the true objective function with high probability by finding a sufficiently large sample to guarantee the sampled gradient to be "close" enough to the true gradient. Then, the step taken will be a descent direction sufficiently often to obtain good convergence properties. The methods diverge in the interpretation of "close". The main three interpretations found in the literature are the norm test, the inner-product test and the acute-angle test. Our method relates to these methods because our goal is also to ensure a decrease in the true objective function, but, in a trust-region framework, the step taken is not necessarily in the steepest descent direction. However, the truncated-CG algorithm ensures a decrease in the Monte Carlo function $\tilde{f}_k$, meaning that, $\tilde{f}_k(x_k + s_k) - \tilde{f}_k(x_k) < 0$. The question is then, what sample size will guarantee a decrease in the true objective function? Secondly, we will discuss the relation to methods based on construction of confidence intervals.

The **norm test** proposed by **?**] is designed to control the deviation in norm of the sampled gradient to the true gradient. Formally, they want $\delta_{S_k} \stackrel{def}{=} \|\nabla \tilde{f}_k(x_k) - \nabla f(x_k)\| \leqslant \theta \|\nabla \tilde{f}_k(x_k)\|$, where $\theta \in [0; 1)$ is a user-specified parameter. When θ is close to 0, the sampled gradient is highly constrained to be in a small ball around the true gradient, but when θ is near to 1, noisier estimates are allowed. To deal with the stochastic nature of the problem, they intend to control the quantity $\mathbb{E}[\delta_{S_k}^2] = \| \mathbb{V}\text{ar}\left(\nabla \tilde{f}_k(x_k)\right)\|_1$ where the variance

is computed component-wise, therefore leading to the following update formula,

$$|S_{k+1}| = \frac{\| \mathbb{V}\mathrm{ar}_{i\in S_k}\left(\nabla F_i(x_k)\right)\|_1}{\theta^2 \|\nabla \tilde{f}_k(x_k)\|^2}$$

where $\| \mathbb{V}\mathrm{ar}_{i\in S_k}\left(\nabla F_i(x_k)\right)\|_1 = \dfrac{1}{m-1}\Big(\nabla F_i(x_k) - \nabla f(x_k)\Big)^T\Big(\nabla F_i(x_k) - \nabla f(x_k)\Big)$. **?**] designed an **inner product test** by building up on the work from **?**]. Instead of imposing a condition on the deviation in norm, they design a sample size strategy to ensure that $\nabla \tilde{f}_k(x_k)$ is a descent direction for f by requiring that $\nabla \tilde{f}_k(x_k)^T \nabla f(x_k) > 0$. To do so, they work on the stochastic condition,

$$\mathbb{E}\left[\left(\nabla \tilde{f}_k(x_k)^T \nabla f(x_k) - \|\nabla f(x_k)\|^2\right)^2\right] \le \theta^2 \|\nabla f(x_k)\|^4 \qquad \text{, for some } \theta > 0$$

allowing them to derive the following formula,

$$N_{k+1} = \left\lceil \frac{\mathbb{V}\mathrm{ar}_{i\in S_k}\left(\nabla F_i(x_k)^T \nabla \tilde{f}_k(x_k)\right)}{\theta^2 \|\nabla \tilde{f}_k(x_k)\|^4} \right\rceil. \tag{2.3.13}$$

The idea can be summed up by building the one-sided $100(1-\alpha_k)\%$ confidence interval for $\nabla \tilde{f}_{\mathcal{S}_k}(x_k)^T \nabla f(x_k)$, and finding the adequate sample size S_k for the lower bound to be positive. They also add more conditions to ensure that the sampled gradient is not too orthogonal to the true gradient. Some issues can also occur when starting with very small samples, and the sample size can stay constant across iterations. To tackle this issue they put a condition to ensure an increase in the sample size when it stays constant too long, e.g. for r iterations.

The same formula is also derived by **?**] in a linesearch framework by invoking, under suitable assumptions, the CLT for the sampled gradient, showing a more direct connection to the *Fisher information matrix*,

$$\nabla \tilde{f}_k(x_k) - \nabla f(x_k) \sim \mathcal{N}\left(0\,, \frac{\mathbb{V}\mathrm{ar}_{\boldsymbol{\xi}}\left(\nabla F(x, \boldsymbol{\xi})\right)}{N}\right).$$

?] proposed an **acute-angle test** where they choose to control the quantity $\mathbb{E}\left[\left\| \dfrac{\tilde{g}_k}{\|\tilde{g}_k\|} - \dfrac{\tilde{g}_k^T \nabla f}{\|\tilde{g}_k\| \|\nabla f\|} \dfrac{\nabla f}{\|\nabla f\|} \right\|^2\right]$ to keep the sampled gradient at a small angle from the true one.

Our method can be seen as an extension to the trust-region framework of the technique designed by **?**] and **?**]. Indeed, in the following calculations, we show how our formula includes the linesearch case, for which the step taken at every iteration is of the form $s_k = -\nu_k \nabla \tilde{f}_{S_k}(x_k)$, where $\nu_k > 0$ is the steplength. The numerator in (2.3.13) can be rewritten

as follows,

$$
\begin{aligned}
&\mathbb{V}\mathrm{ar}_{i\in S_k}\left(\nabla F_i(x_k)^T\nabla \tilde{f}_k(x_k)\right)\\
&= \frac{1}{N_k-1}\sum_{i\in\mathcal{S}_k}\left(\nabla F_i(x_k)^T\nabla \tilde{f}_k(x_k) - \nabla \tilde{f}_k(x_k)^T\nabla \tilde{f}_k(x_k)\right)^2\\
&= \frac{1}{N_k-1}\sum_{i\in\mathcal{S}_k}\nabla \tilde{f}_k(x_k)^T\left(\nabla F_i(x_k) - \nabla \tilde{f}_k(x_k)\right)\left(\nabla F_i(x_k) - \nabla \tilde{f}_k(x_k)\right)^T\nabla \tilde{f}_k(x_k)\\
&= \nabla \tilde{f}_k(x_k)^T\,\mathbb{V}\mathrm{ar}_{i\in\mathcal{S}_k}\left(\nabla F_i(x_k)\right)\nabla \tilde{f}_k(x_k)\\
&= \frac{1}{\nu_k^2}\,s_k^T\,\mathbb{V}\mathrm{ar}_{i\in\mathcal{S}_k}\left(\nabla F_i(x_k)\right)s_k,
\end{aligned}
$$

and

$$
\begin{aligned}
\tilde{\mu}_k &= \tilde{f}_k(x_k+s_k) - \tilde{f}_k(x_k) \approx \nabla \tilde{f}_k(x_k)^T s_k = -\nu_k\nabla \tilde{f}_k(x_k)^T\nabla \tilde{f}_k(x_k) = -\nu_k\|\nabla \tilde{f}_k(x_k)\|^2\\
\left(\tilde{\mu}_k\right)^2 &= \nu_k^2\|\nabla \tilde{f}_k(x_k)\|^4.
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
N_{k+1} &= \left\lceil \frac{\mathbb{V}\mathrm{ar}_{i\in S_k}\left(\nabla F_i(x_k)^T\nabla \tilde{f}_k(x_k)\right)}{\theta^2\|\nabla \tilde{f}_k(x_k)\|^4}\right\rceil = \left\lceil \frac{s_k^T\,\mathbb{V}\mathrm{ar}_{i\in\mathcal{S}_k}\left(\nabla F_i(x_k)\right)s_k}{\theta^2(\nu_k)^2\|\nabla \tilde{f}_k(x_k)\|^4}\right\rceil\\
&= \left\lceil \frac{s_k^T\,\mathbb{V}\mathrm{ar}_{i\in\mathcal{S}_k}\left(\nabla F_i(x_k)\right)s_k}{\theta^2\left(\tilde{\mu}_k\right)^2}\right\rceil.
\end{aligned}
$$

We recover the same formula as **?**] but for a trust-region framework, considering that $1/\theta$ plays an equivalent role to $z_{(1-\alpha)}$.

We now discuss the relations to works based on construction of confidence intervals and measures of the sample variability around the current function value. **?**] developed a dynamic sample size scheme in a trust-region framework. **?**] then transposed this method to linesearch algorithms and later extended it to nonmonotone linesearch methods by adding some safeguards [**?**]. A confidence interval is built around the current function value,

$$
\mathcal{I}_k = [\tilde{f}_k - \epsilon_{\alpha,N_k}\,;\,\tilde{f}_k + \epsilon_{\alpha,N_k}] \quad \text{with } \epsilon_{\alpha,N_k} = (z_{1-\alpha})\frac{\hat{\sigma}^F(x_k,N_k)}{\sqrt{N_k}},
$$

where $\hat{\sigma}^F(x_k,N_k)^2 = (N_k-1)^{-1}\sum_{i=1}^{N_k}\left(\nabla F(x_k\,,\,\xi_i) - \nabla \tilde{f}_k(x_k)\right)^T\left(\nabla F(x_k\,,\,\xi_i) - \nabla \tilde{f}_k(x_k)\right)$ is the centered sample variance estimator of the the standard deviation of the random variable $F(x_k,\boldsymbol{\xi})$. That is, the goal is then to find an optimal sample size to balance the model decrease and the width of the confidence interval. Roughly speaking, if the decrease in the function value is large compared to the width of the confidence interval then the sample size

is decreased at the next iteration. In the opposite case, when the decrease is relatively small in comparison with the precision, the sample size is increased.

?] propose a method for *stochastic root-finding problems* (SRFP) where the goal is to find a zero of a vector-valued function $h : \mathcal{D} \subset \mathbb{R}^q \to \mathbb{R}^q$, i.e to find $x^* \in \mathcal{D}$ such that $h(x^*) = \gamma$, assuming one exists, where $\gamma \in \mathbb{R}^q$ is the target value. Transposed to our context, where the vector-valued function under scrutiny will be the gradient function ∇f and γ is the null vector, the formula can be stated as follows,

$$N_k = \inf \left\{ m \in \mathbb{N}^+ \mid \left\| m^{-1} \sum_{i=1}^{m} \nabla F(x_k\,,\, \boldsymbol{\xi}_i) - \gamma \right\| > c\, \frac{\hat{\sigma}^F(x_k, m)}{\sqrt{m}} \right\}, \tag{2.3.14}$$

with $0 < c < 1$. The underlying idea behind strategy (2.3.14) is to continue the sampling at x_k until the algorithm is reasonably certain that the deviation of $\nabla \tilde{f}_{N_k}(x_k)$ from the target is mainly due to the bias $\|\nabla \tilde{f}_{N_k}(x_k) - \gamma\|$ rather than a consequence of the sampling variability. **?**] propose two simple adaptive sampling rules similar to (2.3.14). We will only present the first as it is more related to our method. They replace the $\|\nabla \tilde{f}_{N_k}(x_k)\|$ on the left-hand side with a geometrically decreasing deterministic sequence γ^{-k}, for some $\gamma \in (1, \overline{\gamma})$, where $\overline{\gamma}$ is defined based on some prior curvature information. They realize the convergence analysis in a convex a strongly convex setting.

?] extend the idea in a derivative-free optimization context where at every iteration a quadratic trust-region model is constructed by interpolation. Omitting details for clarity (see the full paper for deeper insights), the sample size is updated according to a formula of the form,

$$N_k = \max \left\{ \lambda_k\,,\, \inf \left\{ m \in \mathbb{N}^+ \;\middle|\; \frac{\hat{\sigma}^F(x_k, m)}{\sqrt{m}} \le \frac{\kappa_{as} \Delta_k^2}{\sqrt{\lambda_k}} \right\} \right\}.$$

where $\{\lambda_k\}$ is a sample size lower bound sequence such that $k^{1+\epsilon} = \mathcal{O}(\lambda_k)$ and κ_{as} is a constant. The main idea is to sample until the estimated standard error drops below a slightly deflated square of the trust-region radius.

2.3.3. Novelty of our method

The originality of our work from the literature is manifold. The information geometry is used in an algorithmically efficient manner by using a Monte Carlo estimate of the *Fisher information matrix* to approximate the Hessian, but also to compute the sample size, and therefore, no extra computation is needed because the quantity $s_k^T \, \mathbb{V}\text{ar}_{i \in \mathcal{S}_k} \Big(\nabla F_i(x_k) \Big) s_k$ is already computed to evaluate the model accuracy ρ_k. **?**], **?**], and Metel [**?**] work in linesearch framework, so the step taken is proportional to the sampled gradient. They only need to control this quantity for it to be a descent direction with high probability to ensure a decrease in the objective function sufficiently often. Their work only differ in the way they do so. However, in a trust-region algorithm, the step taken can be very different to the

steepest descent. Therefore, it feels natural to use that step in the sample size calculation. Lastly, the works mentioned above build a confidence interval around the current function value $\tilde{f}_{\mathcal{S}_k}(x_k)$ in order to gauge if the progress is statistically significant and not just an effect of the noisy estimates. We instead build a confidence interval for the true decrease in the objective function, which requires evaluating a difference of function values and the variance considered in the calculations is different, which enables a connection to the *Fisher information matrix*. Also, studying differences allows us to consider variable strategies for variance reduction.

2.4. Numerical Experiments

2.4.1. Multinomial Logit Model

To ease the development of the sampling method, a simple log-likelihood maximization was used to test and give a first insight into the performance and possible flaws in our sampling algorithm. The numerical experiments were conducted using a synthetic dataset. Such datasets have many advantages over real world datasets as they can be designed to satisfy all the assumptions, with a known true optimal solution. Moreover, there is no limit on the size of the population (the population is not even required to fit into storage) and the dimension of the problem can be set arbitrary. The synthetic model has been chosen to be a logit model with linear utility function. The logit model with linear utility is known to be convex, simplifying the optimization process.

In discrete choice theory [**?**], each individual $i \in X$, X being the total population, is assumed to be facing a set of alternatives $a(i)$. Random utility maximization models [**?**] assume that each individual chooses the alternative y_i that maximizes its utility function.

$$y_i = \max_{j \in a(i)} v_{i,j}(\beta^*),$$

where β^* is the true optimal parameters and $v_{i,j}$ is assumed to be partially random. The utility can be decomposed into two parts, an observed utility $u_{i,j}(\beta)$ and a random part $\epsilon_{i,j}$

$$v_{i,j}(\beta) = u_{i,j}(\beta) + \epsilon_{i,j}.$$

If we assume that the $\epsilon_{i,j}$ are i.i.d. and follow a Gumbel distribution of scale parameter equal to one, for a given β, the probability that individual i chooses alternative j is then given by

$$\mathbb{P}_{ij}(\beta) = \mathbb{P}\left[v_{i,j}(\beta) \geq v_{i,k}(\beta)\ \forall k \in \{a(i) \backslash \{j\}\}\right] = \frac{\exp\left(u_{i,j}(\beta)\right)}{\sum_{k \in a(i)} \exp\left(u_{i,k}(\beta)\right)}. \tag{2.4.1}$$

(2.4.1) is also known as the softmax function in machine learning. For simplicity reason, we will drop the index j and use the $\mathbb{P}_i$ to define $\mathbb{P}_{i,y_i}$ the probability that individual i has chosen the alternative y_i. We can obtain an estimator of β^* by maximizing the log-likelihood that

the population made the choices they made. Introducing the notation $\ell_i(\beta) := \log \mathbb{P}_i(\beta)$, the function to maximize is the average log-likelihood

$$f(\beta) = \frac{1}{|X|} \sum_{i \in X} \log \mathbb{P}_i(\beta) = \frac{1}{|X|} \sum_{i \in X} \ell_i(\beta)$$

The gradient and Hessian matrix of both the softmax function and its logarithm are developed in Appendix A. In statistics, the maximization of the log-likelihood function $f(\beta)$ is called multinomial logistic regression, or in short, multinomial logit.

Linear utilities are the most popular utility formulations for multinomial logit models. Therefore, in our experiments, we consider a linear utility, defined by

$$v^1_{i,j}(\beta) = \langle \beta, x_{i,j} \rangle,$$

where $x_{i,j}$ are the explanatory variables of the individual i faced to the alternative j. The number of alternatives was set to 5 and the dimension of the problem was set to 10. The optimal solution β^* was set to one in every dimension. The explanatory variables were simply independent $U(0,1)$ random variables and a random variable following a Gumbel distribution with scale factor equal to one was added to each alternative in order to create a "perfect" dataset. One hundred thousand observations were generated.

2.4.2. Results

All tests were run using an Intel i5-5400U CPU running at 2.900GHz with 8 Gigs of RAM. The first serie of tests was done using the BHHH Hessian approximation. Three versions of the BHHH approximation implementation were tested. The first implementation simply stores the BHHH matrix, the second implementation stores each individual gradients and uses them to compute the product of the BHHH matrix with a given vector and the last implementation does not store any gradient and compute the gradients of each observation every time there is a matrix-vector product involving the BHHH matrix. The three versions have their own benefits and limitations. To determine which implementation was the best for the current problem, a population of 10 thousands observations was used and the time required to generate the matrix and the time to execute the matrix-vector product was computed. We can see from Table 2.3 the benefits of each implementation. The implementation storing the BHHH matrix and then computing the matrix-vector product is by far the fastest. It can almost be considered free but it is the implementation that requires the most time to generate the structure. The implementation with recomputed gradients and no storage has by far the cheapest cost to generate the structure but also has the most expensive matrix-vector product. Finally, the implementation where the gradients are stored is a good compromise between both metrics. During the TCG algorithm, the matrix-vector product can be required between one and ten times and, close to the solution, all iterations

might be required meaning that the last implementation is by far the worst for the current problem. Finally, even if the matrix-vector product is extremely cheap when the matrix is stored, the cost of generating the matrix is too high and the overall best implementation in both dimensions is the implementation storing all the gradients.

Implementation	Structure generation (ms)	Matrix-vector product (ms)
Gradients stored	32.023	0.089
Matrix stored	41.25	0.00015
No storage	0.0094	33.649

Table 2.3. BHHH approximation benchmark

In the second serie of tests, we compared the time and number of iterations required for both sampling strategies DS and NDS (see Table 2.1) crossed with all three smoothing strategies in Table 2.2 (naive, no smoothing and monotonous), as well as full batch (by using all the available observations). For all the tests, the stopping criterion was a robust first-order criterion, that is the optimization stopped only if for all the dimensions i, $g^i \leq \epsilon \max(x^i, 1)$, with ϵ set to 10^{-4}. It is a very harsh stopping criterion as it forces the solution to be extremely precise but allowed us to study the behavior of the sampling strategy close to the solution. Finally, during the optimization, three metrics were recorded at every iteration of the optimization algorithm: the distance to the solution, the Monte Carlo estimation of the function, and the sample size. The distance to the solution can be computed since we have fixed it ourself.

The first test using the BHHH approximation and the sampling strategy DS with no smoothing shows the unstable behavior of the sampling strategy when no smoothing strategy is used. We report in Figure 2.1 a portion of the optimization iterations. The graph clearly shows the sample size increasing massively and then reducing down to the minimal sampling authorized. For instance, at iteration 171, the sample size is 4686 and is then reduced down to 100 at iteration 172.

From Table 2.4, we can observe the behavior of the sampling size strategy when properly constrained. The monotonous smoothing was the smoothing that worked the best for both sampling strategies. This is not surprising since the function to optimize is convex and the algorithm has barely any advantage to reduce the sample size. It is interesting to observe what happened just before the iteration 175 with the sampling strategy DS and the naive smoothing, the optimization routine maintaining a small sample size while getting farther from the solution, as illustrated in Figure 2.2(a). Such a behavior is ruled out when using the monotonous sampling as the sample size would not have been allowed to decrease past iteration 144.

This test shows that the sampling strategy requires a smoothing strategy. The optimization routine with no smoothing for the sampling strategy took more time than using no

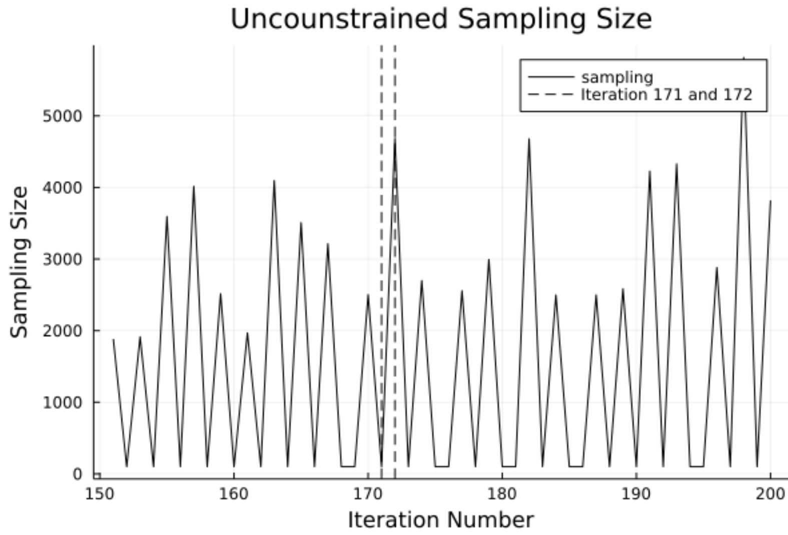
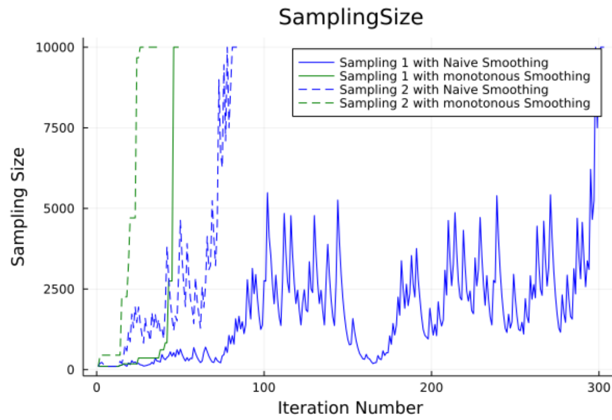
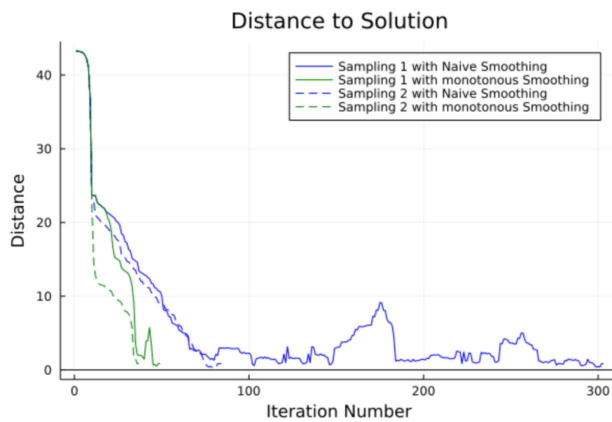


Fig. 2.1. Unconstrained sampling



(a) Sample size



(b) Distance to Solution

Fig. 2.2. Distance to solution and sample size for BHHH approximation

Sampling strategy	Smoothing strategy	Time (s)
Full batch	No smoothing	9.86
DS	No smoothing	769.22
DS	Naive	14.74
DS	Monotonous	1.46
NDS	No smoothing	16.87
NDS	Naive	5.55
NDS	Monotonous	3.28

Table 2.4. Optimization time, BHHH approximation

strategy and using the full population with times of 769.22 seconds, 16.87 seconds, and 9.86 seconds, respectively. The monotonous smoothing performed extremely well with 1.46 second and 3.28 seconds, being 3 times faster than the optimization routine without smoothing. While this exhibits the potential of the sampling strategy, it does not show which sampling strategy is the best. Depending on the smoothing technique, the best sampling strategy changes.

2.4.3. Hessian Approximation

As discussed in Section 2.2.5, there are multiple reasons to use the Hessian matrix and its approximation as an estimation of the covariance matrix of the score. In this section, we will test the sampling method with the BTR algorithm using the the Hessian matrix and the BFGS approximation of the the Hessian matrix. Similarly to the BHHH implementation, the Hessian matrix is only used in a matrix-vector product context, and two implementations can be used. The first one is to compute the Hessian and simply do a matrix vector product and the second one is to compute the gradient of the function $\mathcal{F}(x) = \nabla F(x) \cdot v$ every time a matrix-vector product is required.

Similarly to the BHHH approximation, both implementations have benefits, as reported in Table 2.5. Once the matrix has been computed, the matrix-vector product is almost free when the Hessian matrix is stored. On the other hand, if the TCG stops after a single iteration, computing the gradient of $\mathcal{F}(x)$ is much cheaper then computing the full Hessian matrix. The implementation chosen was to store the full Hessian matrix since the TCG algorithm take multiple iterations (multiple iterations require multiple matrix-vector products) to solve the quadratic problem near the solution. Furthermore, since the sample size is at its biggest near the solution, the last iterations are the most expensive and the implementation when computing the full matrix is the best for this particular model.

Both sampling strategies were tested and compared when the full batch is used. For the sampling strategies, both naive and monotonous smoothing were tested. From Table 2.6, we can observe that monotonous sampling outperformed the optimization time for the naive sampling by more that 50% with both forms of the sampling equations (DS and NDS).

Implementation	Structure generation (ms)	Matrix-vector product (ms)
Matrix stored	114.8	0.00016
Gradient of $\mathcal{F}(x)$	0.00019	53.76

Table 2.5. True Hessian matrix benchmark

Most importantly, DS works even when using the true Hessian as an approximation of the information matrix.

Sampling strategy	Smoothing strategy	Time (s)
Full batch	No smoothing	6.16
DS	Naive	1.99
DS	Monotonous	1.05
NDS	Naive	1.71
NDS	Monotonous	1.11

Table 2.6. Times and results for BTR algorithm with the true Hessian matrix

2.5. Conclusion

Both sampling strategies, while new in the literature, are simple generalizations of existing sampling strategies. It is unclear which sampling strategy is the best since some problems are solved faster with DS when compared to NDS, while better results are achieved with NDS over DS, depending on the Hessian matrix approximation used and the smoothing strategies. For instance, DS achieves better results than NDS when using monotonous sampling and true Hessian matrix is slightly slower when using the naive smoothing. Since both algorithms converge in similar times, the key to accelerate the optimization time seems to be hidden in the smoothing strategy used. The best performing smoothing strategy was the monotonous smoothing but its good performance could be explained by the fact that the current problem is strictly convex.

Conclusion

In this thesis, we have explored two interesting but distinct and unrelated optimization problems.

The first one concerns the optimization settings for RocksDB, a log-structured merge-tree based key-value storage engine. A literature review and a list of the most commonly used settings are presented in the first chapter followed by the description of our approach. To the best of our knowledge, our method is new in the literature, and presents a log-file based technique to identify non-optimal settings. Using this methodology, we were able to pin-point settings requiring changes in order to improve the write throughput of the system. This approach only requires a single run and thus has the potential to greatly accelerate the search for optimal settings for such a problem. A test was made using RocksDB but other popular LSM-tree based storage engines have similar log-files, meaning that our approach can be extended to other systems. Future work would require more experiments to validate the proposed methodology on RocksDB and other LSM-tree storage engines.

The second optimization problem is related to statistical learning. The goal is to find a sampling strategy that finds an adequate sample size at a given iteration on which to build the Monte Carlo estimation of the true function. The Monte Carlo estimation of the function is required to be sufficiently close to the true function to guarantee a reduction of the true function with sufficient probability. To do so, a first-order Taylor approximation of the function for each observation in the current sampling set is constructed. From this approximation and using the central limit theorem, the distribution of the decrease of the function is computed and approximated by a Normal distribution. Finally, the estimated parameters of the Normal distributions are used to approximate a sampling size that would have allowed a decrease of the function to minimize with sufficient probability. More work must be done to rigorously justify the assumptions and approximations required by the proposed sampling strategy, which was tested on a statistical learning problem with synthetic data. The sampling strategy has been found to be efficient, the optimization being significantly faster than a full batch method. The optimization time is nevertheless highly dependent on the smoothing technique used. This issue should be investigated in more details.

Appendix A

Multinomial logit derivatives

The multinomial logistic regression is typically performed by maximazing the log-likehood of the observed choices. It is thereful useful to be able to compute the gradient and the Hessian matrix of the log-likelihood function. The gradient of the softmax function (2.4.1) with respect to β is given by

$$\nabla \mathbb{P}_{i,j}(\beta) = \mathbb{P}_{i,j}(\beta) \cdot \sum_{k \in a(i)} \mathbb{P}_{i,k}(\beta) \cdot (\nabla u_{i,j}(\beta) - \nabla u_{i,k}(\beta)),$$

and the gradient of its logarithm $\ell_{i,j}(\beta) = \log \mathbb{P}_{i,j}(\beta)$ is

$$\nabla \ell_{i,j}(\beta) = \nabla u_{i,j}(\beta) - \sum_{k \in a(i)} \mathbb{P}_{i,k}(\beta) \nabla u_{i,k}(\beta).$$

The Hessian matrix of $\ell_{i,j}(\beta)$ is given by

$$\nabla^2 \ell_{i,j}(\beta) = \nabla^2 u_{i,j}(\beta) - \sum_{k \in a(i)} \left(\nabla \mathbb{P}_{i,k}(\beta) \nabla u_{i,k}^T(\beta) + \mathbb{P}_{i,k}(\beta) \nabla^2 u_{i,k}(\beta) \right).$$

Obviously, in the case of linear utilities, the Hessian matrix log-likelihood function simply becomes

$$\nabla^2 \ell_{i,j}(\beta) = - \sum_{k \in a(i)} \nabla \mathbb{P}_{i,k}(\beta) \nabla u_{i,k}^T(\beta).$$

It is also easy to derive the expression of the product between $\nabla^2 \ell_{i,j}(\beta)$ and a direction v as

$$\nabla^2 \ell_{i,j}(\beta) \cdot v = \nabla^2 u_{i,j}(\beta) \cdot v - \sum_{k \in a(i)} \left(\nabla \mathbb{P}_{i,k}(\beta) \nabla u_{i,k}^T(\beta) + \mathbb{P}_{i,k}(\beta) \nabla^2 u_{i,k}(\beta) \right) \cdot v,$$

and, for linear utilities,

$$\nabla^2 \ell_{i,j}(\beta) \cdot v = - \sum_{k \in a(i)} \nabla \mathbb{P}_{i,k}(\beta) \nabla u_{i,k}^T(\beta) \cdot v.$$