A syllabification tool for very long poems in Italian

Scala project for the Languages and Algorithms for Artificial Intelligence course

Luca Salvatore Lorello[1] (ID. 919584)

Exam session: Summer 2020

[1]The author wishes to thank Mrs. Simona Scacchi, dutiful Italian teacher and friend, for her invaluable help.

Contents

Abstract

This work aims to produce, given an Italian text, a syllabified version (with syllables separed by #) of it, a dictionary of encountered syllables mapping each of them to their count, and order statistics on them (most frequent, least frequent and centiles). Peculiar aspects of the Italian language, in general, and of poetic style, in particular, are exploited to solve, up to a very satisfactory accuracy, the diphthong/hiatus ambiguity (which plays an important role in the syllabification process), and the algorithm is developed in order to be robust with respect to punctuation marks and capitalization.

Albeit, clearly language dependent, the software architecture aims to be easily adaptable to one's needs, while remaining scalable. For this purpose, the syllable-splitting algorithm is implemented in a functional-programming style and different software architectures are explored. Furthermore some unit tests are implemented and the entire tool's scalability is benchmarked against long texts based on Dante's *Commedia*.

Chapter 1

Problem analysis

1.1 Problem specification

This project consists of a Scala tool capable of taking arbitrarily long Italian (with a particular focus on poetry) texts encoded in UTF-8 and returning a *syllabified* version of them, in which all the syllable boundaries are marked with a # sign, while preserving other punctuation marks and being robust with respect to elisions [5] (marked in Italian by the ' apostrophe) and metric dieresis [4] (marked in poetic Italian by an umlaut over vowels). Since the algorithm processes *written* Italian text, while syllabification is a *phonological* phenomenon, the output's accuracy won't reach 100%, however synalepha [11] phenomena will be detected as well.

Together with the syllabification of input texts, this project outputs some metrics measured on them:

- Absolute count for each syllable (in a dictionary),
- Most frequent syllable,
- Median syllable,
- Least frequent syllable.

1.2 Italian rules for syllabification

A syllable is a group of phonemes which are emitted as a single sound. Any human language splits words into syllables in a process known as *syllabification*, this process imposes constraints on the combinations of phonemes allowed in a given language.

As a universal principle, a syllable must contain at least a *vocoid* and, optionally, antevocalic and/or postvocalic *contoids* [3]. A single syllable is phonologically a variation of intensity in the sound emitted by the larynx, with *peaks* corresponding to vocoids and *troughs* corresponding to contoids, syllabification can therefore be analyzed phonetically by isolating each trough-to-trough segment of the voice intensity [7]. Albeit simple and universal, this approach cannot be applied directly to written text, since the mapping between phonemes (sound units) and graphemes (text symbols) is not bijective, however in Italian vocoids always correspond to vowels (and therefore the first, and most important syllabification rule, can already be derived: a syllable must always contain at least a vowel) and the mapping presents very few exceptions.

For syllabification of a specific language, it's useful to use less general rules. Luckily, for the Italian language, these rules are well defined and only few cases of ambiguity arise when applied without phonetic informations (namely the position of the tonic accent inside words), moreover, heuristics exist for reducing these ambiguities further.

Before describing the syllabification rules for Italian, it's important to clarify some terms which will be used:

Digram — Sequence of two letters corresponding to a single sound, namely:

- gl + i, gn + vowel,
- sc + e/i,
- ch + e/i, gh + e/i,
- ci + a/o/u, gi + a/o/u (except when the i is accented, as in farmacìa).

Trigram — Sequence of three letters corresponding to a single sound, namely:

- gli + a/o/u,
- sci + a/o/u.

Semiconsonant — Vowel which is pronounced as if it was a consonant, can be only i/u in one of the following cases:

- i/u + accented vowel (eg. ièri, uòva, etc.),
- accented vowel + i/u,
- diphthongs,
- triphthongs.

Diphthong — Sequence of two vowels pronounced with a single sound, namely:

- i + a/e/o/u,

- u + a/e/i/o,
- a/e/o/u + i,
- a/e + u.

Triphthong Sequence of three vowels pronounced with a single sound, namely:

- iài, ièi,
- uài, uòi, (eg. guài),
- iuò (eg. aiuòla).

Hiatus Sequence of two/three vowels pronounced separately, namely:

- a + e/o (eg. maestro),
- e + a/o,
- o + a/e,
- ì/ù + vowel(s) (eg. farmacìa, which is also an exception to the digram definition),
- ri/bi/tri + vowel (eg. biella),
- metric dieresis.

Dieresis In poetry, is a variation of pronunciation (for metric purposes) which splits letters which are usually pronounced together into multiple sounds, ie. a diphthong becomes a hiatus, as in "così vid' ïo la gloriosa rota", where the syllable splitting is "co-sì-vi-d'ï-o-la-glo-rio-sa-ro-ta", instead of the prosaic split "co-sì-vi-d'io-la-glo-rio-sa-ro-ta".

Synalepha In poetry, when a word ends with a vowel and the next one begins with another vowel, the two words are (often) merged into one and the boundary belongs to the same syllable (even if the joined vowels would normally form a hiatus, eg. "Lì cominciò con forza e con menzogna" is split as "Lì-co-min-ciò-con-for-z**ae**-con-men-zo-gna", even though a-e should be split, as in m**a-e**-stro).

From the above terminology, two sources of ambiguity *arising when the accent is not marked* can be already be identified:

- ci/gi + vowel can be either a digram (and so will be pronounced with the same sound as the vowel) or one of the possible hiatus cases (i + vowel, pronounced as two separate sounds),
- i/u + vowel can be either a diphthong or a hiatus.

Heuristics able to solve these ambiguities with an high precision exist, even when the accent is not explicitly marked [8], however since hiatuses are statistically rare in current Italian (pronunciation of languages tends towards simplification over time, and diphthongs are easier to pronounce than hiatuses), in case of ambiguity, the proposed algorithm will default to considering two consecutive vowels a diphthong.

Another ambiguity, which cannot be solved without semantic context (not even by humans), arises on deciding whether to apply a synalepha or not (this is at complete discretion of the poet and the same words, appearing in different verses can be tied by a synalepha in one and split in the other). Since they are statistically frequent, the proposed algorithm will mark all the vocalic boundaries as possible synalepha.

The following rules are derived from [10, 9]:

- At the beginning of a word, a vowel or a diphthong form a syllable on their own (eg. **a**-mi-co, **au**-gu-ri, **e**-lo-qu-io),
- Diphthongs/triphthongs always belong to the same syllable (eg. a-**iuo**-la, u-ra-n**io**, ci-l**ie**-gia),
- Hiatuses (including dieresis, but excluding those appearing inside synalepha) are always split (eg. m**a-e**-stro, in-no-c**u-o**, in-vi-d**ï-o**-si, qu-**ï-e**-tar-mi),
- Digrams/trigrams are always together with the following vowel (eg. **ci**-lie-**gia**, **gno**-mo),
- A single consonant cannot be on it's own syllable (eg. **pa-lo**, a-**mi-co**),
- Double consonants (including cq) are always split (eg. a**c-q**ua, so**q-q**ua-dro, ri-du**r-r**e),
- Impure s (s + consonant) are always together with the following vowel (eg. **sce**l-le-ra-to, **sco-sta**r-si, **stra**p-pa-re),
- Other consecutive consonants are always split (eg. ac-cre-sci-me**n-t**o, a**n-t**ro-po-lo-gi-co), except when a mute consonant (b, c, d, g, p, t) is followed by a liquid one (l, r) (eg. pi-ro-**cla-sto**, ac-**cli**-ma-ta-to, an-**tro**-po-lo-gi-co, ac-**cre**-sci-men-to, s**tr**ap-pa-re), but not viceversa (cfr. a**r-t**o and an-**tr**o).

1.3 Related work

Nowadays any word processor or text preparation system ships with a syllabification engine, mainly used for correctly splitting a word near the page margin.

While syllabification principles are universal among human languages, the mapping between phonemes and graphemes is unique to each language and therefore different approaches need to be applied for each language.

The most used approach is based on lookup tables: each language has its own dictionary (which stores for each word its syllabification) and a newline is inserted at the closest splitting point. Although universal (and applicable to languages, like English, which don't have strict rules for mapping phonemes into graphemes), this approach tends not to scale well. Rule-based approaches, like the one proposed in this project, are implemented in more sophisticated text preparation systems, like LaTeX's `babel` package, and some computational linguistics pubblications [2, 1]. In text preparation systems the diphthong-hiatus ambiguity does not arise, thanks to the tendence of avoiding an end-of-line split between two vowels (no matter whether they form a diphthong or hiatus) in Italian, so a rule-based approach can be extremely compact and effective.

Chapter 2

Software design

2.1 Word splitting procedure

Syllabification is a sequential process which can be applied either to an entire text or to each word separately. In order to exploit a distributed architecture, however, it's wise to apply the algorithm to single words (which can be distributed to different nodes, regardless of their actual position inside the text).

While prosaic text has a clear boundary between words (ie. a whitespace character), with the only exception of elisions (eg. "l'amico", which is considered a single word), poetic works tend to mimic the spoken tendency of joining consecutive words, by means of extensive use of *synalepha* (which will alter the "natural" syllabification as well).

This requirement imposes to split the input into words when a whitespace (optionally with some punctuation marks, against which the procedure should be robust) is encountered, except in the following cases:

- An apostrophe is encountered (either before, after or with no whitespace at all),
- The whitespace separates two vowels (except when a newline separates two verses).

In order to improve modularity, two splitting classes are implemented: the first one (actually a singleton object) splits the input into big *chunks* (or `Pages`) of text (useful when the input is so big that even the word splitting procedure needs to be distributed); while the latter splits a single `Page` into words.

To improve code reuse, the shared splitting behavior is implemented in a `Splitter` trait which exposes a single `split(area: String): Queue[Int]` method taking an area of text and returning all the indexes in which a word split can be performed. Internally it applies a tail-recursive split on whitespaces, checking whether the split found is acceptable or not according to the aforementioned criteria.

The `ChunkSplitter` object exposes two methods (`loadString(text: String, chunks: Int, lookahead: Int): Queue[Page]` and `loadFile(file: String, chunks: Int, lookahead: Int): Queue[Page]`) which load a text (from a string or a file respectively), applying a preliminary split into `Page` instances. An obvious race condition on the input forces this initial split to be sequential, however, abstracting from loading overhead and assuming $\mathcal{O}(1)$ substring extraction by index, splits are performed (in a tail-recursive fashion) in $\mathcal{O}(\texttt{chunks} \cdot \texttt{lookahead})$ time, because the `ChunkSplitter` invokes the `split()` method only in substrings starting at $\texttt{i} \cdot \texttt{text.length()/chunks}$ and of length `lookahead`, for `i` in $0..\texttt{chunks} - 1$.

The `Page` case class models a single chunk of text. It's a string enriched with housekeeping informations required for later reconstruction of the original text, namely it's `id` and `start/end` indexes inside the original text, however in the current implementation these informations are not used, due to the word ordering being implicitly preserved during computations.

The `WordSplitter(val page: Page)` class' `getWords(): Queue[Word]` method splits words tail-recursively, producing an immutable queue of `Word`s.

2.2 Syllable splitting rules

Once words have been correctly split, their syllables can be computed independently on each of them, abstracting from their position inside the original text (as long as synalepha are considered a single word). This property allows to encapsulate the syllable splitting inside the `Word` case class and to distribute its computation easily.

The `Word` case class exposes the methods `toSyllabifiedString(): String` and `toQueueOfSyllables(): Queue[Syllable]`, the former returns a string containing the word itself with syllables delimited by an # character, while the latter creates a `Syllable` instance for each syllable and stores them in a queue.

Each of them invokes private methods which split syllables in two phases:

1. `performInitialSplits(str: String): String` (partially) splits syllables at easily identifiable boundaries:

    - double consonants,
    - multiple consonants,
    - dieresis,
    - hiatuses;

2. `performFinalSplits(str: String): String` splits syllables at harder to detect boundaries:

    - contoid vocoid - contoid vocoid,
    - vocoid - contoid vocoid,
    - vocoid - vocoid.

As already discussed, ambiguities arise when two consecutive vowels meet, to solve this problem, heuristics need to be chosen:

- `splitHiatus(str: String): String` splits only sure hiatuses (ie. those which can never be mistaken for diphthongs and those with a marked accent),

- `performFinalSplits(str: String): String` (in the vocoid - vocoid split) and the auxiliary method `clumpDiphthongs(str: String): String` consider two consecutive vowels diphthongs as often as possible (ie. when a sure triphthong cannot be detected), since hiatuses are already split when the method is invoked and the combination vowel + diphthong is statistically more common than a triphthong.

Since syllabification rules impose, not only splitting requirements (eg. between double consonants), but also clumping requirements, the special characters §, { and } are internally used (and then removed) to mark positions in which a split is forbidden, therefore these characters (along with #) should not appear in the input text.

2.3 Frequency analysis

In order to unify every possible variant of a `Syllable` (eg. uppercase, lowercase, with apostrophes, with diacritics, inside synalepha, etc.), a `CharacterSequence` trait (which is mixed-in `Page` and `Word` classes as well) is injected in the `Syllable` case class.

`CharacterSequence`'s `toPrettyString(): String` method converts the `str` attribute of the injected class to a lowercase version, without spaces, punctuation marks and diacritics (accents and dieresis) on vowels. The `toString(): String` method overrides the default behavior returning the `str` attribute for convenience.

2.3.1 Syllable count

The most natural way of counting elements of a collection of elements of type `K` is to use any collection implementing the `Map[K, Int]` trait, indexed with the distinct elements of the collection. This approach works reasonably fast (since it can be performed in $\mathcal{O}(n)$) as long as indexing can be performed in $\mathcal{O}(1)$ (which is the case for `HashMap`, the default implementation of `Map`).

`Map`s typically impose constraints on the key type, in the case of `HashMap` it must be hashable, ie. each element should be uniquely identified (and its identifier must never change during the collection's lifetime). Since `String` is an immutable type, a prettified `CharacterSequence` satisfies both the language specifications and the "semantic" requirement of aggregating different versions of the same syllable (eg. "l'a" and "la" are prettified in the same way and therefore counted together).

2.3.2 Order statistics

While selection algorithms (eg. Quickselect, Heapselect, etc.) have an average complexity of $\mathcal{O}(n)$, the need of extracting more than one selection statistic neglects their advantage over the "naive" sort ($\mathcal{O}(nlog(n))$) followed by indexing ($\mathcal{O}(1)$), the latter strategy is therefore preferred.

2.4 Integration and distributed architecture

Although the syllabified output is intrinsically sequential and the map of syllable occurrences is a shared resource in which multiple agents need to synchronize, the implementation will benefit from parallel or distributed execution. To prove this claim, the same algorithm is implemented in multiple architectures and their performance will be profiled to determine which scales better with respect to input size.

In general the workflow is to split the input into `Page`s, each page into `Word`s and each word into `Syllable`s. In order to preserve the ordering, the most natural choice for collections is the `Queue`, using its *immutable* variant allows to distribute its content in a straight forward manner, since it can be handled in a purely functional fashion.

Listing 2.1: Syllable extraction pseudocode

```
val pages: Queue[Page] = ChunkSplitter.loadFile(in,
    chunks, lookahead)
val wordSplitters: Queue[WordSplitter] = pages.map(p =
    > new WordSplitter(p))
val words: Queue[Word] = wordSplitters.flatMap(ws => w
    .getWords())
val syllables: Queue[Syllable] = words.flatMap(w => w.
    toQueueOfSyllables())
```

Once the queue of syllables is obtained, two substantially different operations need to be performed:

- Output the queue as a string,

- Count the occurrences of each syllable.

While the first operation requires to preserve positional information on each syllable (implicitly encoded by a syllable's position inside the queue), the latter can be efficiently computed only after aggregation, it's therefore more reasonable to compute them in the aforementioned order, instead of recomputing positions which will be inevitably lost during aggregation.

The first output can be simply computed exploiting the method `Seq.mkstring(str: String)` on the queue of syllables, or using directly the method `Word.toSyllabifiedString()` mapping each word into a string and then joining them.

The second output involves unification of different variants of the same syllable and the creation of the occurrence map. The last step can be performed by creating an intermediate queue of `Tuple2[String, Int]`, which associates each syllable to a single occurrence, and then aggregating all the tuples with the same key by summing their values.

Listing 2.2: Syllable counting pseudocode

```
val prettySyllables: Queue[String] = syllables.map(s =
    > s.toPrettyString)
val tuples: Queue[(String, Int)] = prettySyllables.map
     (ps => (ps, 1))
val count: Map[String, Int] = tuples.reduceByKey((a, b
    ) => a + b)
```

While this implementation looks straightforward, the method `reduceByKey` is not a native Scala method (it's implemented by Spark's framework) and a less abstract implementation needs to be performed: a queue of queues is computed by grouping each tuple by their key (the first element) and then each subqueue is mapped into a new tuple `(key, sum of values)`.

Listing 2.3: reduceByKey pseudocode

```
val queueOfQueues = tuples.groupBy(t => t._1)
val count: Map[String, Int] = queueOfQueues.mapValues(
    el => el.map(t => t._2).sum)
```

In case of multiple pages, a partial occurrence count for each of them can be obtained. In order to merge them, a reduction is performed by summing, for each key in the concatenated map (which overwrites duplicate keys with the value from the second map), the corresponding value with either the value of the first map (if the key was duplicated) or zero.

Listing 2.4: Map merging pseudocode

```
val count: Map[String, Int] = queueOfCounts.reduce(
  (m1, m2) => m1 ++ m2.map {
    case (k, v) => k -> (v + m1.getOrElse(k, 0))
    }
)
```

The occurrence count map is finally sorted and centiles are extracted as output.

2.4.1 Sequential architecture

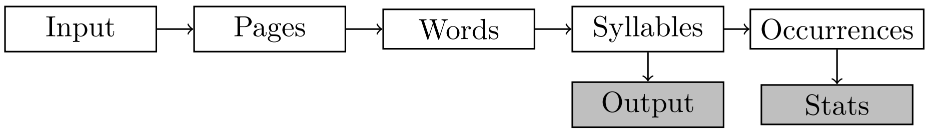

Figure 2.1: Sequential dataflow

In this straightforward architecture, a single computational unit is used to proceed from input to outputs. Even though the input can be split into multiple pages (and therefore partial occurrence maps can be computed for each of them), processing is performed in order.

2.4.2 Parallel architecture

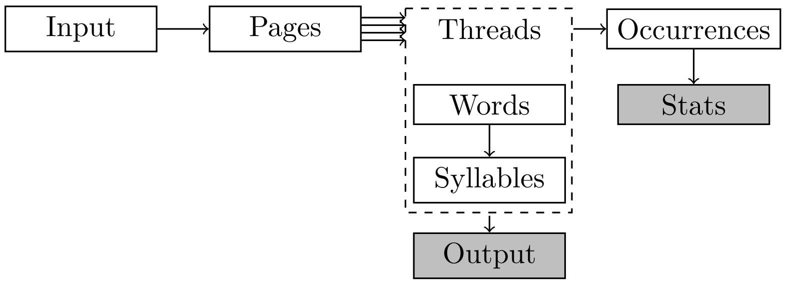

Figure 2.2: Parallel dataflow

Since syllabification is an instance of an *embarassingly parallel* [6] task, a parallel architecture can be devised, with no need for synchronization mechanisms, other than a *barrier* at the end of each computation.

In this architecture, a worker *thread* is started for each page. When syllabification is completed for the associated page, each thread stores internally the final results (the syllabified string and a partial occurrence count). When every thread has finished, the partial results are merged together sequentially (a simple append in the case of the syllabified output, a reduction for the partial occurrence maps).

2.4.3 First distributed architecture

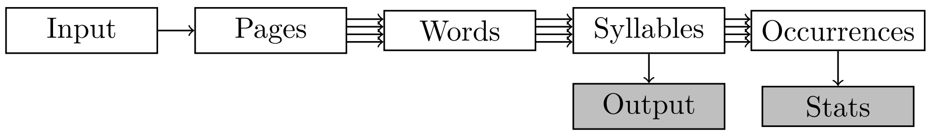

Figure 2.3: Distributed dataflow with maximum parallelism

Since every collection involved in the program is immutable, they can be parallelized into Spark's resilient distributed datasets and the output can be computed in a distributed fashion.

Data distribution can be implemented in two possible ways, by maximizing parallelism or by minimizing inter-node communication. The former approach is discussed in this section, while the latter in the following.

Since the disk is a shared resource, pages still need to be split sequentially, however as soon as a `WordSplitter` can be created for each of them, the collection can be distributed.

RDD transformations create two new RDDs (of words first and then of syllables), then the syllabified output is collected and the same RDD of syllables is further transformed into an RDD of occurrences (which is sorted by the distributed framework prior to collection).

While this approach tends to maximize the amount of data which is processed simultaneously, the first two transformations **increase** the number of elements (contrary to the usual approach of trying to reduce an RDD size at each step, to minimize communication, which is the bottleneck of this architecture), making this architecture potentially slow in some circumstances.

2.4.4 Second distributed architecture

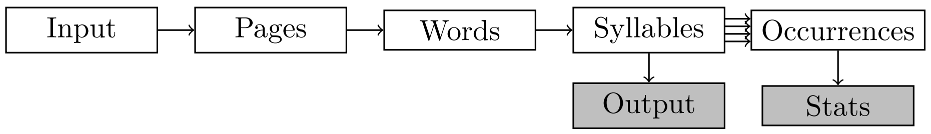

Figure 2.4: Distributed dataflow with minimum parallelism

By postponing parallelization as much as possible, it's possible to minimize communication, at the expenses of a lower degree of parallelism.

This architecture produces a queue of syllables sequentially and then parallelizes it into an RDD in order to perform only the last transformation into an RDD of occurrences.

This seemingly naive implementation is in fact better than the previous one, in case the trade off between network communication and sequential syllabification is heavily shifted in favor of the latter.

Chapter 3

Tests

3.1 Unit tests

The `Scalatest` framework was used to perform some unit tests which drove the development of word and syllable splitting algorithms.

Namely, four `FunSuite`s were implemented in order to test:

- the `CharacterSequence` trait behavior,
- the `Splitter` trait behavior,
- the syllabification of single words,
- the syllabification of entire verses.

Although complete coverage was not reached, test cases covered both common and exotic inputs, and all tests except two (caused by a failure of the chosen heuristics for those inputs) passed.

3.2 Profiling

In order to time the execution of the four proposed architectures, a `Profiler` trait was implemented and, in order to achieve higher granularity, their implementation was split into functions profiled separately.

The `Profiler` trait exposes two higher order methods: `profile[T](f: => T): (T, Long)` takes a function and times its execution (returning its return value and the time taken in nanoseconds), exploiting call-by-name parameter passing for a correct profiling, `smoothProfile[T](f: => T, times: Int): (T, Long)` profiles the function more than once and returns the average time (together with the last return value) in order to filter out noise caused by other processes running on the machine.

While optimization could not be turned off (many methods are tail recursive, therefore a compiler optimization is needed for large inputs), the tests have been performed on a fresh virtual machine (with no processes running other than the system daemons) mapped to 4 CPU cores (with 100% execution cap on the physical hardware) and 8*Gb* of RAM, and averaged over multiple runs whenever applicable (neither the distributed architectures nor the tests on large inputs are averaged, the formers to avoid underestimating execution time due to caching, the latters due to lack of time).

An initial implementation used `Lists` instead of `Queues`, in order to make the code "more functional", however their poor performance (every append operation worked, especially using only the `::` operator, in $\mathcal{O}(n)$, while queues can `enqueue` in $\mathcal{O}(1)$) ultimately lead to ruling them out. The following benchmarks are performed on both implementations, showing the dominance of performance of queues over lists also in practice, in terms of execution time.

The benchmarks were performed on the following files:

- `La Divina Commedia.txt` composed of 561097 UTF-8 characters,
- `La Divina Commedia large.txt` composed of 2805485 UTF-8 characters (original dataset copy-pasted 5 times),
- `La Divina Commedia huge.txt` composed of 5610970 UTF-8 characters (original dataset copy-pasted 10 times),
- in memory strings generated by appending programmatically `La Divina Commedia.txt` to itself between 1 and 20 times.

3.2.1 General assessment

A preliminary assessment was performed in order to compare the performance of list-based and queue-based implementations.

The following is the output for lists (related only to the original dataset):

```
Profiling the sequential execution on file "La Divina Commedia.txt",
    split into 4 chunks (profiling averaged over 5 runs)...
Total (averaged) running time: 47921.384041 ms, of which:
  43.373189 ms for reading the file and splitting into chunks,
  47829.655373 ms for splitting each word into syllables and building
    a dictionary of occurrences for each chunk,
  35.288367 ms for merging the dictionaries into one,
  13.067112 ms for sorting and extracting the centiles.

Profiling the parallel execution on file "La Divina Commedia.txt",
    split into 4 chunks (profiling averaged over 5 runs).
    Each chunk is processed by a different thread...
Total (averaged) running time: 22331.325159 ms, of which:
  4.747231 ms for reading the file and splitting into chunks (sequential),
  22294.380319 ms for splitting each word into syllables and building a
    dictionary of occurrences for each chunk (parallel),
  23.938964 ms for merging the dictionaries into one (sequential),
  8.258645 ms for sorting and extracting the centiles (sequential).

Profiling the distributed execution on file "La Divina Commedia.txt",
    split into 4 chunks (profiling CANNOT be averaged).
    Each chunk is processed by a different node...
Total (averaged) running time: 28251.505847 ms, of which:
  2071.806305 ms for initializing Spark,
  455.901307 ms for reading the file and splitting into chunks (sequential),
  12477.873347 ms for splitting each word into syllables and building a
    dictionary of occurrences for each chunk (distributed),
  13245.924888 ms for sorting and extracting the centiles and shutting down
    Spark (sequential).

Profiling the distributed execution on file "La Divina Commedia.txt"
    with no prior splitting (profiling CANNOT be averaged).
    The number of nodes is chosen by Spark...
Total (averaged) running time: 87137.502624 ms, of which:
  94.868064 ms for initializing Spark,
  85781.084591 ms for reading the file, splitting into syllables and
    distributing the dataset (sequential),
  68.578017 ms for building a dictionary of occurrences (distributed),
  1192.971952 ms for sorting, extracting the centiles and shutting down
    Spark (sequential).
```

As it can be noted, for every implementation, the slowest operation is the syllable splitting procedure, however a clear advantage is achieved when exploiting parallelism. The number of pages the input is split into also relates to the performance of the merge procedure (slower when merging fewer occurrence maps of large size, instead of many smaller maps), this hints a non-linear, probably polynomial, complexity (since $m \cdot \mathcal{T}(n/m) \ll \mathcal{T}(n)$).

The following is the output for queues (on the same input file):

```
Profiling the sequential execution on file "La Divina Commedia.txt",
    split into 4 chunks (profiling averaged over 5 runs)...
Total (averaged) running time: 31229.42195 ms, of which:
  59.813815 ms for reading the file and splitting into chunks,
  31101.113841 ms for splitting each word into syllables and building a
    dictionary of occurrences for each chunk,
  54.814361 ms for merging the dictionaries into one,
  13.679933 ms for sorting and extracting the centiles.

Profiling the parallel execution on file "La Divina Commedia.txt",
    split into 4 chunks (profiling averaged over 5 runs).
    Each chunk is processed by a different thread...
Total (averaged) running time: 13635.397876 ms, of which:
  4.850249 ms for reading the file and splitting into chunks (sequential),
  13585.609283 ms for splitting each word into syllables and building a
    dictionary of occurrences for each chunk (parallel),
  35.490544 ms for merging the dictionaries into one (sequential),
  9.4478 ms for sorting and extracting the centiles (sequential).

Profiling the distributed execution on file "La Divina Commedia.txt",
    split into 4 chunks (profiling CANNOT be averaged).
    Each chunk is processed by a different node...
Total (averaged) running time: 19099.583035 ms, of which:
  2420.161538 ms for initializing Spark,
  240.389256 ms for reading the file and splitting into chunks (sequential),
  8209.232922 ms for splitting each word into syllables and building a
    dictionary of occurrences for each chunk (distributed),
  8229.799319 ms for sorting and extracting the centiles and shutting down
    Spark (sequential).

Profiling the distributed execution on file "La Divina Commedia.txt"
    with no prior splitting (profiling CANNOT be averaged).
    The number of nodes is chosen by Spark...
Total (averaged) running time: 76563.562394 ms, of which:
  171.860143 ms for initializing Spark,
  75096.973943 ms for reading the file, splitting into syllables
    and distributing the dataset (sequential),
  53.080434 ms for building a dictionary of occurrences (distributed),
  1241.647874 ms for sorting, extracting the centiles and shutting down
    Spark (sequential).
```

The execution times are sensibly lower, demonstrating that the $\mathcal{O}(n)$ list append effectively acted as a bottleneck on most operations (including RDD transformations).

Further tests on the two implementations were run using the three datasets loaded from disk, in order to show the effect of varying input size.

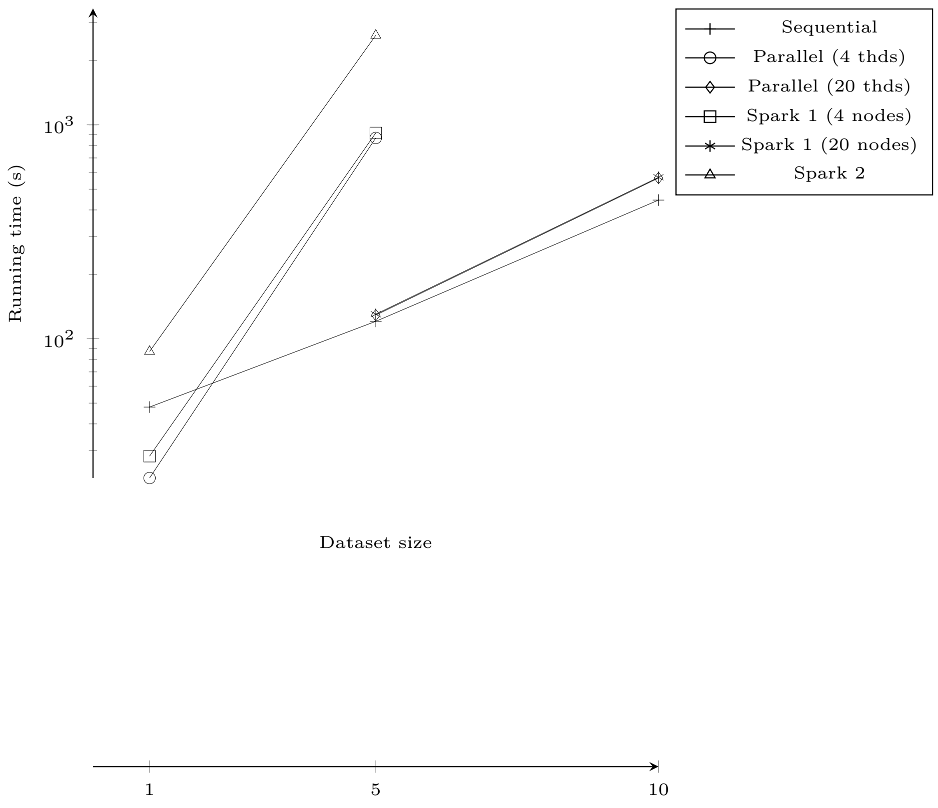

Figure 3.1: Scalability of execution times on list-based implementations

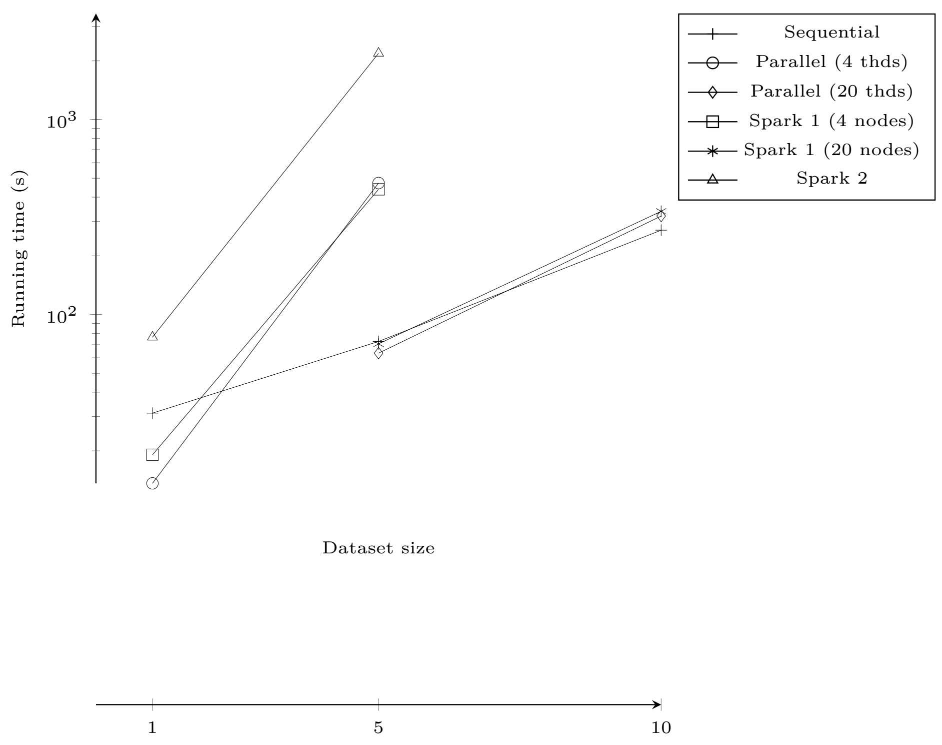

Figure 3.2: Scalability of execution times on queue-based implementations

For both implementations, it can be noted how the second Spark implementation (minimum inter-node communication) scales the worst, followed by low-parallelism implementations (parallel architecture and the first Spark implementation with few nodes). The best scalability is achieved by the sequential implementation and high-parallelism implementations, hinting the fact that for increasing datasets, the occurrence map merging operation becomes more and more relevant for the overall performance (the sequential implementation doesn't need to merge anything, while high-parallelism implementations need to merge only small-sized maps).

3.2.2 Execution time with respect to degree of parallelism

In order to further investigate the dependency of execution times with respect to the degree of parallelism, the parallel architecture was tested using a fixed input (the original dataset), varying the number of threads.

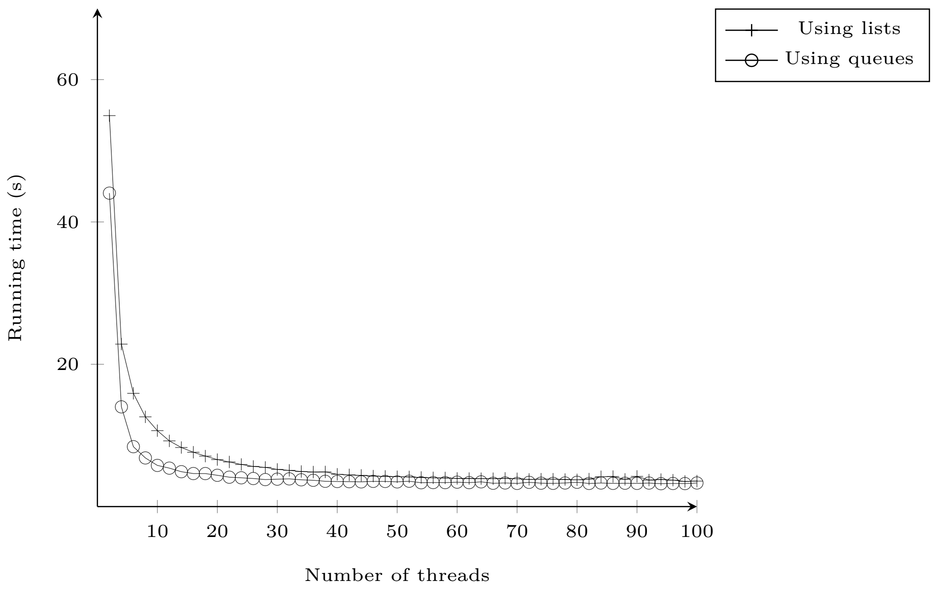

Figure 3.3: Efficiency with respect to the number of threads

It is important to notice that JVM's mapping is many-to-many, so the number of threads effectively running simultaneously will depend on the system's load, however it will never exceed the number of CPUs available on the machine (4 in this case). As such, the expected performance (on the parallel component only) should converge pretty soon (and after a given threshold context-switching overhead should cause a performance degradation).

Although this convergence can be clearly seen on the plot, it doesn't happen as soon as expected, due to the combination with the performance of the sequential part of the implementation (namely the occurrence maps merging, which, as stated before, performs better when merging many smaller maps instead of few larger ones).

Finally, as already demonstrated, the list-based implementation is in all cases outperformed by the queue-based one.

3.2.3 Comparing the most promising architectures

A final test was performed comparing the most promising architectures (sequential, parallel implementation with 20 threads and Spark implementation with maximal parallelism using 20 nodes) on the queue-based implementation to better assess the scalability.

The input was given a a string obtained by repeatedly concatenating in RAM the original dataset a given number of times, due to this lack of I/O operations and the fact that the complete architectures were profiled in a single function for each of them, compiler optimization achieved different results than the other tests (element-wise figures in this test are smaller than the ones obtained in other tests), however the general trend is preserved.

The sequential implementation clearly diverges from the other two, the larger the input becomes, while the parallel implementations tend to perform similarly (this is justified by the fact that on a single machine Spark will still be subjected to the same hardware limitations as the multithreaded implementation, and due to the fact that its core is not completely shut down between each experiment, leading to warm-restarts).

For clarity, both a linear and a semilog plot of the same results are shown, from both a polynomial trend clearly emerges.

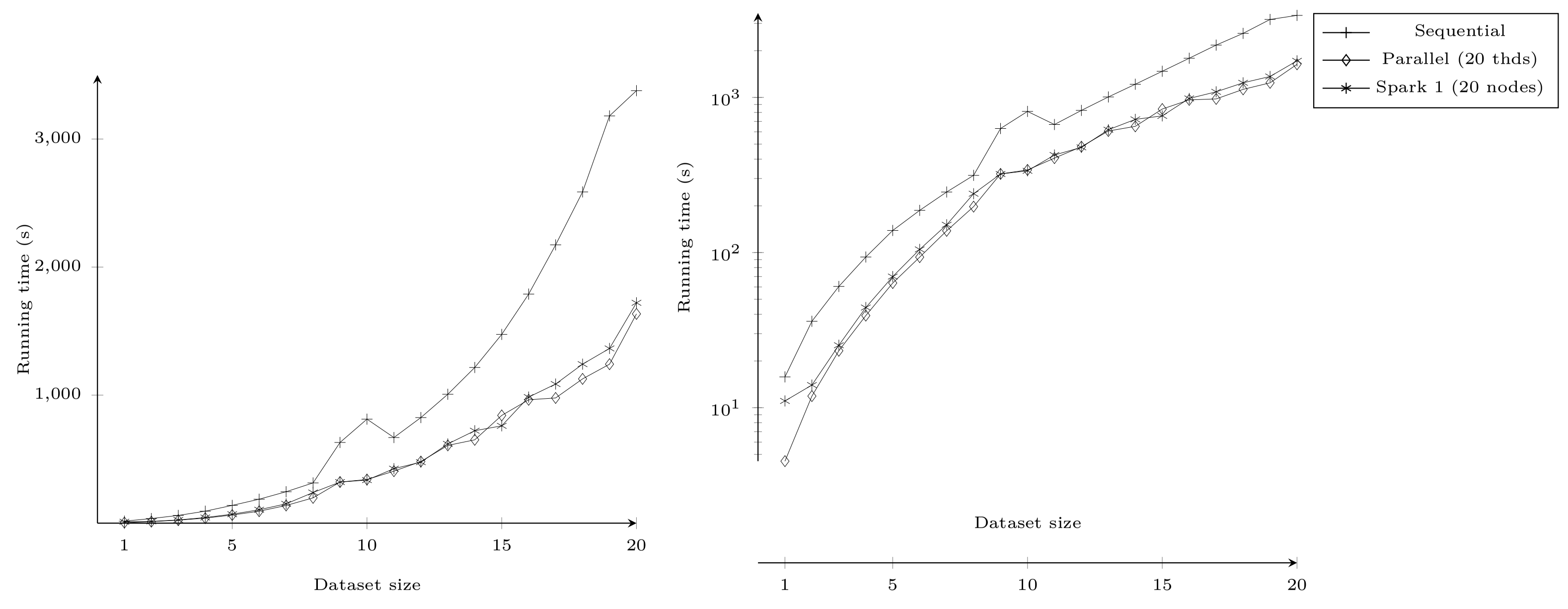

Figure 3.4: Running time of the queue-based implementation

Listings

Bibliography

[1] C. R. Adsett, Y. Marchand, V. Kes, et al. Syllabification rules versus data-driven methods in a language with low syllabic complexity: the case of italian. *Computer Speech & Language*, 23(4):444–463, 2009.

[2] L. Cioni. Rb-tree: un algoritmo per la sillabazione dell'italiano, 1996.

[3] T. De Mauro. *Linguistica elementare*. Editori Laterza, 9th edition, 1998, pages 35–37. ISBN: 8842069779.

[4] Diaeresis (prosody) - Wikipedia. URL: `https://en.wikipedia.org/wiki/Diaeresis_(prosody)`.

[5] Elision - Wikipedia. URL: `https://en.wikipedia.org/wiki/Elision`.

[6] Embarrassingly parallel - Wikipedia. URL: `https://en.wikipedia.org/wiki/Embarrassingly_parallel`.

[7] P. M. Federico Albano Leoni. *Manuale di fonetica*. Università / 357. Carocci editore, 3rd edition, 2002, pages 74–76. ISBN: 9788843021277.

[8] iato in "Enciclopedia dell'italiano". URL: `http://www.treccani.it/enciclopedia/iato_%28Enciclopedia-dell%27Italiano%29`.

[9] M. Sensini. *Lo spazio linguistico*, volume A (La riflessione sulla lingua). Arnoldo Mondadori Scuola, 10th edition, 2005, pages 15–17. ISBN: 9788824724838.

[10] L. Serianni. *Grammatica, sintassi, dubbi*. Garzanti Editore, 2nd edition, 1997, pages 37–38. ISBN: 8811504880.

[11] Synalepha - Wikipedia. URL: `https://en.wikipedia.org/wiki/Synalepha`.