Clojure High Performance Programming
Second Edition

Clojure High Performance Programming, Second Edition, starts by helping you classify various use cases with respect to performance and analyze various performance aspects. You will also learn the performance vocabulary that experts use throughout the world and discover various Clojure data structures, abstractions, and their performance characteristics. Further, the book will guide you through enhancing performance by using Java interoperability and JVM-specific features from Clojure. It also highlights the importance of using the right concurrent data structure and Java concurrency abstractions. This book also sheds light on performance metrics for measuring, how to measure, and how to visualize and monitor the collected data.

With this book, you will learn to run a performance profiler, identify bottlenecks, tune performance, and refactor code to get better performance.

Who this book is written for
This book is intended for intermediate Clojure developers who are looking to get a good grip on achieving optimum performance. Having a basic knowledge of Java would be helpful.

What you will learn from this book
- Identify performance issues in Clojure programs by using different profiling tools
- Master techniques to achieve numerical performance in Clojure
- Use the Criterium library to measure the latency of Clojure expressions
- Exploit Java features in Clojure code to enhance performance
- Avoid reflection and boxing with type hints
- Understand Clojure's concurrency and state-management primitives in depth
- Measure and monitor performance, and understand optimization techniques

Clojure High Performance Programming
Second Edition

Become an expert at writing fast and high performant code in Clojure 1.7.0

Shantanu Kumar

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Clojure Abstractions'
- A synopsis of the book’s content
About the Author

Shantanu Kumar is a software developer living in Bengaluru, India. He works with Concur Technologies as a principal engineer, building a next-generation stack in Clojure. He started learning computer programming when he was at school, and has dabbled in several programming languages and software technologies. Having used Java for a long time, he discovered Clojure in early 2009 and has been a fan of it ever since.

When not busy with programming or reading up on technical stuff, he enjoys reading non-fiction and cycling around Bengaluru. Shantanu is an active participant in The Bangalore Clojure Users Group, and contributes to several open source Clojure projects on GitHub. He is also the author of the first edition of the book Clojure High Performance Programming, Packt Publishing.
Preface

Since the first edition of this book was published in November 2013, Clojure has seen a much wider adoption and has witnessed many success stories. The newer versions of Clojure fortify its performance story while staying close to its roots—simplicity and pragmatism. This edition significantly updates the book for Clojure 1.7, and adds a new chapter on the performance measurement.

The Java Virtual Machine plays a huge role in the performance of the Clojure programs. This edition of the book increases the focus on the JVM tools for performance management, and it explores how to use those. This book is updated to use Java 8, though it also highlights the places where features from Java 7 or 8 have been used.

This book is updated mainly to be more of practical use to the readers. I hope that this edition will better equip the readers with performance measurement and profiling tools and with the know-how of analyzing and tuning the performance characteristics of Clojure code.

What this book covers

Chapter 1, *Performance by Design*, classifies the various use cases with respect to performance, and analyzes how to interpret their performance aspects and needs.

Chapter 2, *Clojure Abstractions*, is a guided tour of various Clojure data structures, abstractions (persistent data structures, vars, macros, transducers, and so on), and their performance characteristics.

Chapter 3, *Leaning on Java*, discusses how to enhance performance by using Java interoperability and features from Clojure.

Chapter 4, *Host Performance*, discusses how the host stack impacts performance. Being a hosted language, Clojure has its performance directly related to the host.
Preface

Chapter 5, Concurrency, is an advanced chapter that discusses the concurrency and parallelism features in Clojure and JVM. Concurrency is an increasingly significant way to derive performance.

Chapter 6, Measuring Performance, covers various aspects of performance benchmarks and measuring other factors.

Chapter 7, Performance Optimization, discusses systematic steps that need to be taken in order to identify that the performance bottlenecks obtain good performance.

Chapter 8, Application Performance, discusses building applications for performance. This involves dealing with external subsystems and factors that impact the overall performance.
Clojure has four founding ideas. Firstly, it was set up to be a functional language. It is not pure (as in purely functional), but emphasizes immutability. Secondly, it is a dialect of Lisp; Clojure is malleable enough that users can extend the language without waiting for the language implementers to add new features and constructs. Thirdly, it was built to leverage concurrency for the new generation challenges. Lastly, it was designed to be a hosted language. As of today, Clojure implementations exist for the JVM, CLR, JavaScript, Python, Ruby, and Scheme. Clojure blends seamlessly with its host language.

Clojure is rich in abstractions. Though the syntax itself is very minimal, the abstractions are finely grained, mostly composable, and designed to tackle a wide variety of concerns in the least complicated way. In this chapter, we will discuss the following topics:

- Performance characteristics of non-numeric scalars
- Immutability and epochal time model paving the way for performance by isolation
- Persistent data structures and their performance characteristics
- Laziness and its impact on performance
- Transients as a high-performance, short-term escape hatch
- Other abstractions, such as tail recursion, protocols/types, multimethods, and many more
Non-numeric scalars and interning

Strings and characters in Clojure are the same as in Java. The string literals are implicitly interned. Interning is a way of storing only the unique values in the heap and sharing the reference everywhere it is required. Depending on the JVM vendor and the version of Java you use, the interned data may be stored in a string pool, Permitgen, ordinary heap, or some special area in the heap marked for interned data. Interned data is subject to garbage collection when not in use, just like ordinary objects. Take a look at the following code:

```
user=> (identical? "foo" "foo") ; literals are automatically interned
true
user=> (identical? (String. "foo") (String. "foo")) ; created string
is not interned
false
user=> (identical? (.intern (String. "foo")) (.intern (String.
"foo")))
true
user=> (identical? (str "f" "oo") (str "f" "oo")) ; str creates
string
false
user=> (identical? (str "foo") (str "foo")) ; str does not create
string for 1 arg
true
user=> (identical? (read-string "\"foo\"") (read-string "\"foo\"")) ;
not interned
false
user=> (require '[clojure.edn :as edn]) ; introduced in Clojure 1.5
nil
user=> (identical? (edn/read-string "\"foo\"") (edn/read-string
"\"foo\")
false
```

Note that identical? in Clojure is the same as == in Java. The benefit of interning a string is that there is no memory allocation overhead for duplicate strings. Commonly, applications on the JVM spend quite some time on string processing. So, it makes sense to have them interned whenever there is a chance of duplicate strings being simultaneously processed. Most of the JVM implementations today have an extremely fast intern operation; however, you should measure the overhead for your JVM if you have an older version.

Another benefit of string interning is that when you know that two string tokens are interned, you can compare them faster for equality using identical? than non-interned string tokens. The equivalence function first checks for identical references before conducting a content check.
Symbols in Clojure always contain interned string references within them, so generating a symbol from a given string is nearly as fast as interning a string. However, two symbols created from the same string will not be identical:

```
user=> (identical? (.intern "foo") (.intern "foo"))
true
user=> (identical? (symbol "foo") (symbol "foo"))
false
user=> (identical? (symbol (.intern "foo")) (symbol (.intern "foo")))
false
```

Keywords are, on the basis of their implementation, built on top of symbols and are designed to work with the `identical?` function for equivalence. So, comparing keywords for equality using `identical?` would be faster, just as with interned string tokens.

Clojure is increasingly being used for large-volume data processing, which includes text and composite data structures. In many cases, the data is either stored as JSON or EDN (http://edn-format.org). When processing such data, you can save memory by interning strings or using symbols/keywords. Remember that string tokens read from such data would not be automatically interned, whereas the symbols and keywords read from EDN data would invariably be interned. You may come across such situations when dealing with relational or NoSQL databases, web services, CSV or XML files, log parsing, and so on.

Interner is linked to the JVM *Garbage Collection* (GC), which, in turn, is closely linked to performance. When you do not intern the string data and let duplicates exist, they end up being allocated on the heap. More heap usage leads to GC overhead. Interning a string has a tiny but measurable and upfront performance overhead, whereas GC is often unpredictable and unclear. GC performance, in most JVM implementations, has not increased in a similar proportion to the performance advances in hardware. So, often, effective performance depends on preventing GC from becoming the bottleneck, which in most cases means minimizing it.

**Identity, value, and epochal time model**

One of the principal virtues of Clojure is its simple design that results in malleable, beautiful composability. Using symbols in place of pointers is a programming practice that has existed for several decades now. It has found widespread adoption in several imperative languages. Clojure dissects that notion in order to uncover the core concerns that need to be addressed. The following subsections illustrate this aspect of Clojure.
We program using logical entities to represent values. For example, a value of 30 means nothing unless it is associated with a logical entity, let’s say age. The logical entity age is the identity here. Now, even though age represents a value, the value may change with time; this brings us to the notion of state, which represents the value of the identity at a certain time. Hence, state is a function of time and is causally related to what we do in the program. Clojure’s power lies in binding an identity with its value that holds true at the time and the identity remains isolated from any new value it may represent later. We will discuss state management in Chapter 5, Concurrency.

Variables and mutation

If you have previously worked with an imperative language (C/C++, Java, and so on), you may be familiar with the concept of a variable. A variable is a reference to a block of memory. When we update its value, we essentially update the place in memory where the value is stored. The variable continues to point to the place where the older version of the value was stored. So, essentially, a variable is an alias for the place of storage of values.

A little analysis would reveal that variables are strongly linked to the processes that read or mutate their values. Every mutation is a state transition. The processes that read/update the variable should be aware of the possible states of the variable to make sense of the state. Can you see a problem here? It conflates identity and state! It is impossible to refer to a value or a state in time when dealing with a variable—the value could change at any time unless you have complete control over the process accessing it. The mutability model does not accommodate the concept of time that causes its state transition.

The issues with mutability do not stop here. When you have a composite data structure containing mutable variables, the entire data structure becomes mutable. How can we mutate it without potentially undermining the other processes that might be observing it? How can we share this data structure with concurrent processes? How can we use this data structure as a key in a hash-map? This data structure does not convey anything. Its meaning could change with mutation! How do we send such a thing to another process without also compensating for the time, which can mutate it in different ways?

Immutability is an important tenet of functional programming. It not only simplifies the programming model, but also paves the way for safety and concurrency. Clojure supports immutability throughout the language. Clojure also supports fast, mutation-oriented data structures as well as thread-safe state management via concurrency primitives. We will discuss these topics in the forthcoming sections and chapters.
Collection types

There are a few types of collections in Clojure, which are categorized based on their properties. The following Venn diagram depicts this categorization on the basis of whether the collections are counted (so that \texttt{counted?} returns \texttt{true}) or associative (so that \texttt{associative?} returns \texttt{true}) or sequential (so that \texttt{sequential?} returns \texttt{true}):

![Venn diagram of collection types]

The previous diagram illustrates the characteristics that different kinds of data structures share. The sequential structures let us iterate over the items in the collection, the item count of counted structures can be found constant with respect to time, and associative structures can be looked at with keys for corresponding values. The \texttt{CharSequence} box shows the character sequence Java types that can be converted to a Clojure sequence using \texttt{(seq charseq)}.

Persistent data structures

As we’ve noticed in the previous section, Clojure’s data structures are not only immutable, but can produce new values without impacting the old version. Operations produce these new values in such a way that old values remain accessible; the new version is produced in compliance with the complexity guarantees of that data structure, and both the old and new versions continue to meet the complexity guarantees. The operations can be recursively applied and can still meet the complexity guarantees. Such immutable data structures as the ones provided by Clojure are called \textbf{persistent data structures}. They are "persistent", as in, when a new version is created, both the old and new versions "persist" in terms of both the value and complexity guarantee. They have nothing to do with storage or durability of data. Making changes to the old version doesn’t impede working with the new version and vice versa. Both versions persist in a similar way.
Among the publications that have inspired the implementation of Clojure’s persistent data structures, two of them are well known. Chris Okasaki’s [*Purely Functional Data Structures*](#) has influenced the implementation of persistent data structures and lazy sequences/operations. Clojure’s persistent queue implementation is adapted from Okasaki’s *Batched Queues*. Phil Bagwell’s *Ideal Hash Tries*, though meant for mutable and imperative data structures, was adapted to implement Clojure’s persistent map/vector/set.

### Constructing lesser-used data structures

Clojure supports a well-known literal syntax for lists, vectors, sets, and maps. Shown in the following list are some less-used methods for creating other data structures:

- **Map** (`PersistentArrayMap` and `PersistentHashMap`):
  
  ```clojure
  {:a 10 :b 20} ; array-map up to 8 pairs
  {:a :b 2 :c 3 :d 4 :e 5 :f 6 :g 7 :h 8 :i 9} ; hash-map for 9 or more pairs
  ```

- **Sorted map** (`PersistentTreeMap`):
  
  ```clojure
  (sorted-map :a 10 :b 20 :c 30) ; (keys ..) should return sorted
  ```

- **Sorted set** (`PersistentTreeSet`):
  
  ```clojure
  (sorted-set :a :b :c)
  ```

- **Queue** (`PersistentQueue`):
  
  ```clojure
  (import 'clojure.lang.PersistentQueue)
  (reduce conj PersistentQueue/EMPTY [:a :b :c :d]) ; add to queue
  (peek queue) ; read from queue
  (pop queue) ; remove from queue
  ```

As you can see, abstractions such as TreeMap (sorted by key), TreeSet (sorted by element), and Queue should be instantiated by calling their respective APIs.

### Complexity guarantee

The following table gives a summary of the complexity guarantees (using the Big-O notation) of various kinds of persistent data structures in Clojure:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Persistent List</th>
<th>Persistent HashMap</th>
<th>Persistent ArrayMap</th>
<th>Persistent Vector</th>
<th>Persistent Queue</th>
<th>Persistent TreeMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
### Chapter 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Persistent List</th>
<th>Persistent HashMap</th>
<th>Persistent ArrayMap</th>
<th>Persistent Vector</th>
<th>Persistent Queue</th>
<th>Persistent TreeMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>conj</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>first</td>
<td>O(1)</td>
<td>O(&lt;7)</td>
<td>O(&lt;7)</td>
<td></td>
<td>O(&lt;7)</td>
<td></td>
</tr>
<tr>
<td>rest</td>
<td>O(1)</td>
<td>O(&lt;7)</td>
<td>O(&lt;7)</td>
<td></td>
<td>O(&lt;7)</td>
<td></td>
</tr>
<tr>
<td>doseq</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>nth</td>
<td>O(n)</td>
<td>O(&lt;7)</td>
<td>O(&lt;7)</td>
<td></td>
<td>O(&lt;7)</td>
<td></td>
</tr>
<tr>
<td>last</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>O(&lt;7)</td>
<td>O(1)</td>
<td>O(&lt;7)</td>
<td>O(&lt;7)</td>
<td>O(log n)</td>
<td></td>
</tr>
<tr>
<td>assoc</td>
<td>O(&lt;7)</td>
<td>O(1)</td>
<td>O(&lt;7)</td>
<td></td>
<td>O(log n)</td>
<td></td>
</tr>
<tr>
<td>dissoc</td>
<td>O(&lt;7)</td>
<td>O(1)</td>
<td>O(&lt;7)</td>
<td></td>
<td>O(log n)</td>
<td></td>
</tr>
<tr>
<td>peek</td>
<td>O(&lt;7)</td>
<td>O(1)</td>
<td>O(&lt;7)</td>
<td></td>
<td>O(log n)</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td></td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>

A **list** is a sequential data structure. It provides constant time access for count and for anything regarding the first element only. For example, `conj` adds the element to the head and guarantees $O(1)$ complexity. Similarly, `first` and `rest` provide $O(1)$ guarantees too. Everything else provides an $O(n)$ complexity guarantee.

Persistent hash-maps and vectors use the trie data structure with a branching factor of 32 under the hood. So, even though the complexity is $O(\log_{32} n)$, only $2^{32}$ hash codes can fit into the trie nodes. Hence, $\log_{32} 2^{32}$, which turns out to be $6.4$ and is less than 7, is the worst-case complexity and can be considered near-constant time. As the trie grows larger, the portion to copy gets proportionately tiny due to structure sharing. Persistent hash-set implementation is also based on hash-map; hence, the hash-sets share the characteristics of the hash-maps. In a persistent vector, the last incomplete node is placed at the tail, which is always directly accessible from the root. This makes using `conj` to the end a constant time operation.

Persistent tree-maps and tree-sets are basically sorted maps and sets respectively. Their implementation uses red-black trees and is generally more expensive than hash-maps and hash-sets. A persistent queue uses a persistent vector under the hood for adding new elements. Removing an element from a persistent queue takes the head off `seq`, which is created from the vector where new elements are added.
The complexity of an algorithm over a data structure is not an absolute measure of its performance. For example, working with hash-maps involves computing the hashCode, which is not included in the complexity guarantee. Our choice of data structures should be based on the actual use case. For example, when should we use a list instead of a vector? Probably when we need sequential or last-in-first-out (LIFO) access, or when constructing an abstract-syntax-tree (AST) for a function call.

**O(<7) implies near constant time**

You may know that the Big-O notation is used to express the upper bound (worst case) of the efficiency of any algorithm. The variable $n$ is used to express the number of elements in the algorithm. For example, a binary search on a sorted associative collection, such as a sorted vector, is a logarithmic time, that is an $O(\log_2 n)$ or simply an $O(\log n)$ algorithm. Since there can be a maximum of $2^{32}$ (technically $2^{31}$ due to a signed positive integer) elements in a Java collection and $\log_2 2^{32}$ is 32, the binary search can be $O(\leq 32)$ in the worst case. Similarly, though operations on persistent collections are $O(\log_{32} n)$, in the worst case they actually turn out to be $O(\log_{32} 2^{32})$ at maximum, which is $O(<7)$. Note that this is much lower than logarithmic time and approaches near constant time. This implies not so bad performance for persistent collections even in the worst possible scenario.

**The concatenation of persistent data structures**

While persistent data structures have excellent performance characteristics, the concatenation of two persistent data structures has been a linear time $O(N)$ operation, except for some recent developments. The concat function, as of Clojure 1.7, still provides linear time concatenation. Experimental work on Relaxed Radix Balanced (RRB) trees is going on in the core.rrb-vector contrib project (https://github.com/clojure/core.rrb-vector), which may provide logarithmic time $O(\log N)$ concatenation. Readers interested in the details should refer to the following links:

- Phil Bagwell's talk at http://www.youtube.com/watch?v=K2NYwP90bNs
Sequences and laziness

"A seq is like a logical cursor."

— Rich Hickey

Sequences (commonly known as seqs) are a way to sequentially consume a succession of data. As with iterators, they let a user begin consuming elements from the head and proceed realizing one element after another. However, unlike iterators, sequences are immutable. Also, since sequences are only a view of the underlying data, they do not modify the storage structure of the data.

What makes sequences stand apart is they are not data structures per se; rather, they are a data abstraction over a stream of data. The data may be produced by an algorithm or a data source connected to an I/O operation. For example, the `resultset-seq` function accepts a `java.sql.ResultSet` JDBC instance as an argument and produces lazily realized rows of data as seq.

Clojure data structures can be turned into sequences using the `seq` function. For example, `(seq [:a :b :c :d])` returns a sequence. Calling `seq` over an empty collection returns nil.

Sequences can be consumed by the following functions:

- `first`: This returns the head of the sequence
- `rest`: This returns the remaining sequence, even if it's empty, after removing the head
- `next`: This returns the remaining sequence or nil, if it's empty, after removing the head

Laziness

Clojure is a strict (as in, the opposite of "lazy") language, which can choose to explicitly make use of laziness when required. Anybody can create a lazily evaluated sequence using the `lazy-seq` macro. Some Clojure operations over collections, such as `map`, `filter`, and more are intentionally lazy.

Laziness simply means that the value is not computed until actually required. Once the value is computed, it is cached so that any future reference to the value need not re-compute it. The caching of the value is called memoization. Laziness and memoization often go hand in hand.
Laziness in data structure operations

Laziness and memoization together form an extremely useful combination to keep the single-threaded performance of functional algorithms comparable to its imperative counterparts. For an example, consider the following Java code:

```java
List<String> titles = getTitles();
int goodCount = 0;
for (String each: titles) {
    String checksum = computeChecksum(each); 
    if (verifyOK(checksum)) {
        goodCount++;
    }
}
```

As is clear from the preceding snippet, it has a linear time complexity, that is, $O(n)$, and the whole operation is performed in a single pass. The comparable Clojure code is as follows:

```clojure
(->> (get-titles)
    (map compute-checksum)
    (filter verify-ok?)
    count)
```

Now, since we know `map` and `filter` are lazy, we can deduce that the Clojure version also has linear time complexity, that is, $O(n)$, and finishes the task in one pass with no significant memory overhead. Imagine, for a moment, that `map` and `filter` are not lazy—what would be the complexity then? How many passes would it make? It's not just that map and filter would both have taken one pass, that is, $O(n)$, each; they would each have taken as much memory as the original collection in the worst case, due to storing the intermediate results.

It is important to know the value of laziness and memoization in an immutability-emphasizing functional language such as Clojure. They form a basis for amortization in persistent data structures, which is about focusing on the overall performance of a composite operation instead of microanalyzing the performance of each operation in it; the operations are tuned to perform faster in those operations that matter the most.

Another important bit of detail is that when a lazy sequence is realized, the data is memoized and stored. On the JVM, all the heap references that are reachable in some way are not garbage collected. So, as a consequence, the entire data structure is kept in the memory unless you lose the head of the sequence. When working with lazy sequences using local bindings, make sure you don't keep referring to the lazy sequence from any of the locals. When writing functions that may accept lazy sequence(s), take care that any reference to the lazy seq does not outlive the execution of the function in the form of a closure or some such.
Constructing lazy sequences

Now that we know what lazy sequences are, let's try to create a retry counter that should return true only as many times as the retry can be performed. This is shown in the following code:

```clojure
(defn retry? [n]
  (if (<= n 0)
    (cons false (lazy-seq (retry? 0)))
    (cons true (lazy-seq (retry? (dec n))))))
```

The `lazy-seq` macro makes sure that the stack is not used for recursion. We can see that this function would return endless values. Hence, in order to inspect what it returns, we should limit the number of elements as shown in the following code:

```clojure
user=> (take 7 (retry? 5))
(true true true true true false false)
```

Now, let's try using it in a mock fashion:

```clojure
(loop [r (retry? 5)]
  (if-not (first r)
    (println "No more retries")
    (do
      (println "Retrying")
      (recur (rest r)))))
```

As expected, the output should print `Retrying` five times before printing `No more retries` and exiting as follows:

```text
Retrying
Retrying
Retrying
Retrying
Retrying
No more retries
nil
```

Let's take another simpler example of constructing a lazy sequence, which gives us a countdown from a specified number to zero:

```clojure
(defn count-down [n]
  (if (<= n 0)
    '(0)
    (cons n (lazy-seq (count-down (dec n))))))
```
We can inspect the values it returns as follows:

```
user=> (count-down 8)
(8 7 6 5 4 3 2 1 0)
```

Lazy sequences can loop indefinitely without exhausting the stack and can come in handy when working with other lazy operations. To maintain a balance between space-saving and performance, consuming lazy sequences results in the chunking of elements by a factor of 32. That means lazy seqs are realized in a chunk-size of 32, even though they are consumed sequentially.

### Custom chunking

The default chunk size 32 may not be optimum for all lazy sequences—you can override the chunking behavior when you need to. Consider the following snippet (adapted from Kevin Downey's public gist at [https://gist.github.com/hiredman/324145](https://gist.github.com/hiredman/324145)):

```clojure
(defn chunked-line-seq
  "Returns the lines of text from rdr as a chunked[size] sequence of strings. rdr must implement java.io.BufferedReader."
[^java.io.BufferedReader rdr size]
(lazy-seq
  (when-let [line (.readLine rdr)]
    (chunk-cons
      (let [buffer (chunk-buffer size)]
        (chunk-append buffer line)
        (dotimes [i (dec size)]
          (when-let [line (.readLine rdr)]
            (chunk-append buffer line))))
      (chunk buffer))
  (chunked-line-seq rdr size))))
```

As per the previous snippet, the user is allowed to pass a chunk size that is used to produce the lazy sequence. A larger chunk size may be useful when processing large text files, such as when processing CSV or log files. You would notice the following four less-known functions used in the snippet:

- `clojure.core/chunk-cons`
- `clojure.core/chunk-buffer`
- `clojure.core/chunk-append`
- `clojure.core/chunk`
While `chunk-cons` is the equivalent of `clojure.core/cons` for chunked sequences, `chunk-buffer` creates a mutable chunk buffer (controls the chunk size), `chunk-append` appends an item to the end of a mutable chunk buffer, and `chunk` turns a mutable chunk buffer into an immutable chunk.

The `clojure.core` namespace has several functions related to chunked sequences listed as follows:

- `chunk`
- `chunk-rest`
- `chunk-cons`
- `chunk-next`
- `chunk-first`
- `chunk-append`
- `chunked-seq?`
- `chunk-buffer`

These functions are not documented, so although I would encourage you to study their source code to understand what they do, I would advise you not to make any assumptions about their support in future Clojure versions.

**Macros and closures**

Often, we define a macro so as to turn the parameter body of code into a closure and delegate it to a function. See the following example:

```
(defmacro do-something
  [& body]
  `(do-something* (fn [] ~@body)))
```

When using such code, if the body binds a local to a lazy sequence it may be retained longer than necessary, likely with bad consequences on memory consumption and performance. Fortunately, this can be easily fixed:

```
(defmacro do-something
  [& body]
  `(do-something* (^:once fn* [] ~@body)))
```
Notice the `:^:once` hint and the `fn*` macro, which make the Clojure compiler clear the closed-over references, thus avoiding the problem. Let's see this in action (Alan Malloy's example from https://groups.google.com/d/msg/clojure/Ys3kEz5c_eE/3St2AbIc3zMJ):

```clojure
user> (let [x (for [n (range)] (make-array Object 10000))
  f (^:once fn* [ ] (nth x 1e6))] ; using `:^:once`
  (f))
#<Object [Ljava.lang.Object;@402d3105>
user> (let [x (for [n (range)] (make-array Object 10000))
  f (fn* [ ] (nth x 1e6))] ; not using `:^:once`
  (f))
OutOfMemoryError GC overhead limit exceeded
```

The manifestation of the previous condition depends on the available heap space. This issue is tricky to detect as it only raises `OutOfMemoryError`, which is easy to misunderstand as a heap space issue instead of a memory leak. As a preventive measure, I would suggest using `:^:once` with `fn*` in all cases where you close over any potentially lazy sequence.

### Transducers

Clojure 1.7 introduced a new abstraction called transducers for "composable algorithmic transformations", commonly used to apply a series of transformations over collections. The idea of transducers follows from the `reducing function`, which accepts arguments of the form `(result, input)` and returns `result`. A reducing function is what we typically use with `reduce`. A transducer accepts a reducing function, wraps/composes over its functionality to provide something extra, and returns another reducing function.

The functions in `clojure.core` that deal with collections have acquired an `arity-1` variant, which returns a transducer, namely `map`, `cat`, `mapcat`, `filter`, `remove`, `take`, `take-while`, `take-nth`, `drop`, `drop-while`, `replace`, `partition-by`, `partition-all`, `keep`, `keep-indexed`, `dedupe` and `random-sample`.

Consider the following few examples, all of which do the same thing:

```clojure
user=> (reduce ((filter odd?) +) [1 2 3 4 5])
9
user=> (transduce (filter odd?) + [1 2 3 4 5])
9
user=> (defn filter-odd? [xf]
  (fn
   ([] (xf)))
```
Here, \((\text{filter odd?})\) returns a transducer— in the first example the transducer wraps over the reducer function \(+\) to return another combined reducing function. While we use the ordinary \(\text{reduce}\) function in the first example, in the second example we use the \(\text{transduce}\) function that accepts a transducer as an argument. In the third example, we write a transducer \(\text{filter-odd?}\), which emulates what \((\text{filter odd?})\) does.

Let’s see how the performance varies between traditional and transducer versions:

```clojure
;;; traditional way
user=> (time (dotimes [_ 10000] (reduce + (filter odd? (range 10000)))))
"Elapsed time: 2746.782033 msecs"
nil
;;; using transducer
(def fodd? (filter odd?))
user=> (time (dotimes [_ 10000] (transduce fodd? + (range 10000)))))
"Elapsed time: 1998.566463 msecs"
nil
```

### Performance characteristics

The key point behind transducers is how orthogonal each transformation is allowed to be, yet highly composable also. At the same time, transformations can happen in lockstep for the entire sequence instead of each operation producing lazy chunked sequences. This often causes significant performance benefits with transducers. Lazy sequences are still going to be useful when the final result is too large to realize at once—for other use cases transducers should fit the need aptly with improved performance. Since the core functions have been overhauled to work with transducers, it makes sense to model transformations more often than not in terms of transducers.
Transients

Earlier in this chapter, we discussed the virtues of immutability and the pitfalls of mutability. However, even though mutability is fundamentally unsafe, it also has very good single-threaded performance. Now, what if there was a way to restrict the mutable operation in a local context in order to provide safety guarantees? That would be equivalent to combining the performance advantage and local safety guarantees. That is exactly the abstraction called **transients**, which is provided by Clojure.

Firstly, let's verify that it is safe (up to Clojure 1.6 only):

```clojure
user=> (let [t (transient [:a])]
    @(future (conj! t :b)))
IllegalAccessError Transient used by non-owner thread  clojure.lang.PersistentVector$TransientVector.ensureEditable (PersistentVector.java:463)
```

As we can see previously, up to Clojure 1.6, a transient created in one thread cannot be accessed by another. However, this operation is allowed in Clojure 1.7 in order for transducers to play well with the core.async (https://github.com/clojure/core.async) library — the developer should maintain operational consistency on transients across threads:

```clojure
user=> (let [t (transient [:a])] (seq t))
IllegalArgumentException Don't know how to create ISeq from: clojure.lang.PersistentVector$TransientVector  clojure.lang.RT.seqFrom (RT.java:505)
```

So, transients cannot be converted to seqs. Hence, they cannot participate in the birthing of new persistent data structures and leak out of the scope of execution.

Consider the following code:

```clojure
(let [t (transient [])]
    (conj! t :a)
    (persistent! t)
    (conj! t :b))
IllegalAccessError Transient used after persistent! call clojure.lang.PersistentVector$TransientVector.ensureEditable (PersistentVector.java:464)
```

The `persistent!` function permanently converts transient into an equivalent persistent data structure. Effectively, transients are for one-time use only.
Conversion between persistent and transient data structures (the transient and persistent! functions) is constant time, that is, it is an \(O(1)\) operation. Transients can be created from unsorted maps, vectors, and sets only. The functions that mutate transients are: conj!, disj!, pop!, assoc!, and dissoc!. Read-only operations such as get, nth, count, and many more work as usual on transients, but functions such as contains? and those that imply seqs, such as first, rest, and next, do not.

**Fast repetition**

The function clojure.core/repeatedly lets us execute a function many times and produces a lazy sequence of results. Peter Taoussanis, in his open source serialization library Nippy (https://github.com/ptaoussanis/nippy), wrote a transient-aware variant that performs significantly better. It is reproduced, as shown, with his permission (note that the arity of the function is not the same as repeatedly):  

```clojure
(defn repeatedly*   
  "Like \`repeatedly\` but faster and returns given collection type."
  [coll n f]
  (if-not (instance? clojure.lang.IEditableCollection coll)
    (loop 
      [v coll idx 0]
      (if (>= idx n)
        v
        (recur (conj v (f)) (inc idx))))
    (loop 
      [v (transient coll) idx 0]
      (if (>= idx n)
        (persistent! v)
        (recur (conj! v (f)) (inc idx))))))
```

**Performance miscellanea**

Besides the major abstractions we saw earlier in the chapter, there are other smaller, but nevertheless very performance-critical, parts of Clojure that we will see in this section.

**Disabling assertions in production**

Assertions are very useful to catch logical errors in the code during development, but they impose a runtime overhead that you may like to avoid in the production environment. Since assert is a compile time variable, the assertions can be silenced either by binding assert to false or by using alter-var-root before the code is loaded. Unfortunately, both the techniques are cumbersome to use. Paul Stadig’s library called assertions (https://github.com/pjstadig/assertions) helps with this exact use-case by enabling or disabling assertions via the command-line argument -ea to the Java runtime.
Clojure Abstractions

To use it, you must include it in your Leiningen project.clj file as a dependency:

```clojure
:dependencies [;; other dependencies...
  [pjstadig/assertions "0.1.0"]]
```

You must use this library's `assert` macro instead of Clojure's own, so each `ns` block in the application should look similar to this:

```clojure
(ns example.core
 (:refer-clojure :exclude [assert])
 (:require [pjstadig.assertions :refer [assert]])
)
```

When running the application, you should include the `-ea` argument to the JRE to enable assertions, whereas its exclusion implies no assertion at runtime:

```
$ JVM_OPTS=-ea lein run -m example.core
$ java -ea -jar example.jar
```

Note that this usage will not automatically avoid assertions in the dependency libraries.

**Destructuring**

Destructuring is one of Clojure's built-in mini languages and, arguably, a top productivity booster during development. This feature leads to the parsing of values to match the left-hand side of the binding forms. The more complicated the binding form, the more work there is that needs to be done. Not surprisingly, this has a little bit of performance overhead.

It is easy to avoid this overhead by using explicit functions to unravel data in the tight loops and other performance-critical code. After all, it all boils down to making the program work less and do more.

**Recursion and tail-call optimization (TCO)**

Functional languages have this concept of tail-call optimization related to recursion. So, the idea is that when a recursive call is at the tail position, it does not take up space on the stack for recursion. Clojure supports a form of user-assisted recursive call to make sure the recursive calls do not blow the stack. This is kind of an imperative looping, but is extremely fast.
When carrying out computations, it may make a lot of sense to use `loop-recur` in the tight loops instead of iterating over synthetic numbers. For example, we want to add all odd integers from zero through to 1,000,000. Let's compare the code:

```
(defn oddsum-1 [n]  ; using iteration
  (->> (range (inc n))
    (filter odd?)
    (reduce +)))

(defn oddsum-2 [n]  ; using loop-recur
  (loop [i 1 s 0]
    (if (> i n)
      s
      (recur (+ i 2) (+ s i)))))
```

When we run the code, we get interesting results:

```
user=> (time (oddsum-1 1000000))
"Elapsed time: 109.314908 msecs"
250000000000
user=> (time (oddsum-2 1000000))
"Elapsed time: 42.18116 msecs"
250000000000
```

The `time` macro is far from perfect as the performance-benchmarking tool, but the relative numbers indicate a trend—in the subsequent chapters, we will look at the `Criterium` library for more scientific benchmarking. Here, we use `loop-recur` not only to iterate faster, but we are also able to change the algorithm itself by iterating only about half as many times as we did in the other example.

### Premature end of iteration

When accumulating over a collection, in some cases, we may want to end it prematurely. Prior to Clojure 1.5, `loop-recur` was the only way to do it. When using `reduce`, we can do just that using the `reduced` function introduced in Clojure 1.5 as shown:

```
;; let coll be a collection of numbers
(reduce (fn ([x] x) ([x y] (if (or (zero? x) (zero? y)) (reduced 0) (* x y))))
  coll)
```
Here, we multiply all the numbers in a collection and, upon finding any of the numbers as zero, immediately return the result zero instead of continuing up to the last element.

The function `reduced?` helps detect when a reduced value is returned. Clojure 1.7 introduces the `ensure-reduced` function to box up non-reduced values as reduced.

**Multimethods versus protocols**

Multimethods are a fantastic expressive abstraction for a polymorphic dispatch on the dispatch function's return value. The dispatch functions associated with a multimethod are maintained at runtime and are looked up whenever a multimethod call is invoked. While multimethods provide a lot of flexibility in determining the dispatch, the performance overhead is simply too high compared to that of protocol implementations.

Protocols (defprotocol) are implemented using reify, records (defrecord), and types (deftype, extend-type) in Clojure. This is a big discussion topic—since we are discussing the performance characteristics, it should suffice to say that protocol implementations dispatch on polymorphic types and are significantly faster than multimethods. Protocols and types are generally the implementation detail of an API, so they are usually fronted by functions.

Due to the multimethods' flexibility, they still have a place. However, in performance-critical code it is advisable to use protocols, records, and types instead.

**Inlining**

It is well known that macros are expanded inline at the call site and avoid a function call. As a consequence, there is a small performance benefit. There is also a `definline` macro that lets you write a function just like a normal macro. It creates an actual function that gets inlined at the call site:

```clojure
(def PI Math/PI)
(definline circumference [radius]
  `(* 2 PI ~radius))
```

Note that the JVM also analyzes the code it runs and does its own inlining of code at runtime. While you may choose to inline the hot functions, this technique is known to give only a modest performance boost.
When we define a var object, its value is looked up each time it is used. When we define a var object using a :const meta pointing to a long or double value, it is inlined from wherever it is called:

```
(def ^:const PI Math/PI)
```

This is known to give a decent performance boost when applicable. See the following example:

```
user=> (def a 10)
user=> (def ^:const b 10)
user=> (def ^:dynamic c 10)
user=> (time (dotimes [_ 100000000] (inc a)))
"Elapsed time: 1023.745014 msecs"
nil
user=> (time (dotimes [_ 100000000] (inc b)))
"Elapsed time: 226.732942 msecs"
nil
user=> (time (dotimes [_ 100000000] (inc c)))
"Elapsed time: 1094.527193 msecs"
nil
```

**Summary**

Performance is one of the cornerstones of Clojure's design. Abstractions in Clojure are designed for simplicity, power, and safety, with performance firmly in mind. We saw the performance characteristics of various abstractions and also how to make decisions about abstractions depending on performance use cases.

In the next chapter, we will see how Clojure interoperates with Java and how we can extract Java's power to derive optimum performance.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.