

STA Proposal: Scala Standard Library Enhancements

This is a proposal for enhancements that could be made to the Scala standard library under the STA project.

1. Overview

This proposal has three parts:

- **Performance Analysis:** a comparison of `Vector` and `List`
- **Missing Pieces:** additions that fill a gap, where a closely related API already exists in the standard library but one natural member of the family is absent
- **Useful Methods:** the remaining proposed additions, which reduce boilerplate

2. Performance Analysis: `Vector` vs `List`

The deliverable is a report, not an API.

The report should answer:

- which operations matter most for real usage
- how performance changes with size
- where the crossover points are
- how to interpret results without overstating microbenchmark numbers

The report should describe:

- benchmark methodology
- warmup and GC handling
- input sizes and operations
- graphs and summary tables
- limitations and caveats

3. Missing Pieces

These are additions that fill an obvious gap: a closely related API already exists in the standard library, but one natural member of the family is missing.

3.1 `SeqSet` and `VectorSet`

Source: <https://github.com/scala/scala-library-next/issues/22>, <https://github.com/scala/scala-library-next/issues/168>

Intent: provide order-preserving set implementations.

Behavior:

- immutable
- preserves insertion order
- should outperform `ListSet` for the intended workloads

Rationale: `SeqMap` and `VectorMap` were added because `ListMap` is only appropriate, performance-wise, for small collections. The same is true of `ListSet`, so the ordered-set counterparts `SeqSet` and `VectorSet` are missing. They are also useful as dependencies for ordered map inversion.

3.2 `Iterable[+A].zipStrict` / `Iterator[+A].zipStrict`

Source: <https://contributors.scala-lang.org/t/standard-library-now-open-for-improvements-and-suggestions/7337/24>

Intent: zip two inputs only if they have exactly the same length.

Proposed signatures:

- on `Iterable` (via `IterableOps`): `def zipStrict[B](that: IterableOnce[B]): Option[CC[(A, B)]]`
- on `Iterator`: `def zipStrict[B](that: IterableOnce[B]): Option[Seq[(A, B)]]`

Behavior:

- defined on both `Iterable` (via `IterableOps`) and `Iterator`, like `zip` / `zipAll`; not on `IterableOnce`
- on `Iterable` the result is the receiver's own collection type, `Option[CC[(A, B)]]`, matching `zip`
- on `Iterator` the result is a strict `Option[Seq[(A, B)]]`: unlike `zip`, `zipStrict` cannot be lazy, because confirming equal length requires consuming both inputs fully, so a lazy `Iterator` result would be misleading
- returns `Some` when both sides have the same length
- returns `None` on a mismatch instead of throwing
- preserves left-to-right pairing order
- consumes both inputs as needed to determine whether they end together
- two empty inputs yield `Some(empty)`

Rationale: we already have `zip`, `zipAll`, and `zipWithIndex`, but no `zipStrict`, despite it being a commonly requested operation for when exact alignment matters.

3.3 `Iterable[+A].lazyZipAll`

Source: <https://github.com/scala/scala-library-next/issues/62>

Intent: zip two iterables lazily while filling missing elements with defaults.

Proposed signature:

- `def lazyZipAll[A1 >: A, B](that: Iterable[B])(thisDefault: A1, thatDefault: B): LazyZip2[A1, B, this.type]`

Behavior:

- shorter side is padded with its default value
- preserves the same pairing order as `lazyZip`
- should remain lazy rather than materializing immediately

Rationale: we have the strict `zip` and `zipAll`, and `lazyZip` was added, but `lazyZipAll` never was. It is the missing lazy counterpart of `zipAll`.

3.4 `Iterable[+A].groupFlatMap`

Source: <https://github.com/scala/scala-library-next/issues/135>

Intent: group elements and flat-map each one into zero or more outputs.

Proposed signature:

- `def groupFlatMap[K, B](key: A => K)(f: A => IterableOnce[B]): Map[K, CC[B]]`

Behavior:

- defined on `Iterable` (via `IterableOps`), not `IterableOnce`: the group value is the receiver's own collection type, which is only meaningful for a reusable collection
- the group value type `CC[B]` matches `groupMap`
- groups by derived key
- expands each input element into zero or more outputs
- preserves encounter order inside each group
- the result is a plain, unordered `Map`, matching `groupMap`

Rationale: we have `map` and `flatMap`, and `groupMap` was added, but not `groupFlatMap`. It is the grouping counterpart of `flatMap`.

4. Useful Methods

These are the remaining proposed additions: useful operations that reduce boilerplate, even though they do not fill a gap left by an existing family.

4.1 `Iterable[+A].groupByOrdered`

Source: <https://github.com/scala/scala-collection-contrib/issues/252>

Prior art: <https://github.com/sake92/squery/blob/main/squery/src/ba/sake/squery/utils/seqUtils.scala>

Intent: group elements by key while preserving first-seen key order, with overloads tuned to the shape of SQL join results.

Proposed signatures:

- `def groupByOrdered[K](extractKey: A => K): SeqMap[K, C]`
- `def groupByOrdered[K, V](extractKey: A => K, extractValue: A => V): SeqMap[K, CC[V]]`
- `def groupByOrderedOpt[K, V](extractKey: A => K, extractValue: A => Option[V]): SeqMap[K, CC[V]]`

Behavior:

- defined on `Iterable` (via `IterableOps`), not `IterableOnce`: the group value is the receiver's own collection type, which is only meaningful for a reusable collection
- the group value has the same collection type as the receiver, mirroring `groupBy` (`C`, the whole-element collection) and `groupMap` (`CC[V]`, the receiver's collection of extracted values)
- keys appear in encounter order; values inside each group preserve input order
- repeated keys append to the existing group
- empty input returns an empty `SeqMap`
- the operation is strict and single-pass over the source
- the single-argument overload keeps the whole element as the grouped value
- the two-argument overload extracts one value from each element
- the `opt` overload extracts an optional value: a `None` adds nothing to the group, but the key is still created with an empty collection

Rationale: this is the order-preserving analogue of `groupBy`. Its main use is re-nesting the flat, denormalized rows a SQL query returns into an ordered, hierarchical structure that mirrors the queried relational data. These methods do not perform or simulate a join: the database runs the join, and the method simply walks the rows it returned and collapses them into a `parent -> Seq(children)` map. The grouping is join-agnostic and works on any flat sequence; join results are just the motivating case. Once you have that nested structure, a natural next step is to serialize it — to JSON, for example. The result is a `SeqMap` rather than a `Map` because queries use `ORDER BY`, and you typically want the grouped output to preserve that order (a JSON array, for instance, is ordered).

Pick the overload that matches the join your rows came from:

- **groupByOrdered** — **rows from an inner join**: every row carries a real value, so values are grouped directly. A parent with no children never appears, because the inner join already omitted it.
- **groupByOrderedOpt** — **rows from a left / full outer join**: the value extractor returns `Option`. When a parent row has a `NULL` child, the extractor yields `None`; the parent key is still created (with an empty `seq`) but no value is appended, so every parent appears even when it has no children.
- **groupByOrdered (single-argument)**: keeps the whole element as the value, for when you want to group entire rows rather than one extracted column.

For reference, these are the join types that produce those rows: an **inner join** keeps only rows that match on both sides; a **left join** keeps every left-side row, filling the right with `NULL` when unmatched; a **full join** keeps every row from both sides, filling `NULL` on whichever side is missing.

4.2 `IterableOnce[+A].frequencies`

Source: <https://contributors.scala-lang.org/t/standard-library-now-open-for-improvements-and-suggestions/7337/7>

Intent: count how often each value appears.

Proposed signature:

- `def frequencies[A1 >: A]: Map[A1, Int]`

Behavior:

- stays on `IterableOnce`: it is a single-pass aggregation returning a fixed `Map[A1, Int]` (not the receiver's collection type), like `toMap` / `count` / `corresponds`, so it does not need `IterableOps` the way `groupByOrdered` does
- repeated elements increment the corresponding count
- empty input returns an empty map
- the result is strict
- the order of appearance of keys is not preserved; the result is an unordered `Map`
- the widened type parameter is needed for covariance reasons

Rationale: this is the direct counting equivalent of `groupMapReduce(identity)(_ => 1)(_ + _)`. Although `groupMapReduce` lives on `IterableOps`, that is grouping-family cohesion rather than a requirement — `frequencies` returns a fixed type, so it belongs with the single-pass aggregators on `IterableOnce`.

4.3 `IterableOnce[+A].mapAccumulate`

Source: <https://contributors.scala-lang.org/t/standard-library-now-open-for-improvements-and-suggestions/7337/46>

Intent: map while threading state through the traversal.

Proposed signature:

- `def mapAccumulate[B, S](z: S)(f: (A, S) => (B, S)): (CC[B], S)`

Behavior:

- returns mapped elements first and final state second
- the mapped collection has the receiver's element-collection type `CC[B]`, like `map`
- preserves encounter order
- runs strictly in a single pass
- empty input returns an empty collection and the initial state unchanged

Rationale: this is useful when stateful traversal is needed but a full custom loop would be boilerplate.

4.4 `IterableOnce[+A].takeTo / dropTo`

Source: <https://github.com/scala/scala-library-next/issues/29>

Intent: take or drop until and including the first matching terminator.

Proposed signatures:

- `def takeTo(p: A => Boolean): C`
- `def dropTo(p: A => Boolean): C`

Behavior:

- returns the receiver's own collection type `C`, like `takeWhile` / `dropWhile`
- `takeTo` includes the first matching element
- `dropTo` excludes the first matching element
- if no element matches, `takeTo` returns the whole input and `dropTo` returns empty
- preserves encounter order
- should work on single-pass sources

Rationale: this is a useful companion to `takeWhile` and `dropWhile`.

4.5 `IterableOnce[+A].mapWithIndex`

Source: <https://github.com/scala/scala-library-next/issues/78>

Intent: map elements with their index.

Proposed signature:

- `def mapWithIndex[B](f: (A, Int) => B): CC[B]`

Behavior:

- returns `CC[B]`, the receiver's collection type, like `map`
- indexing starts at zero
- order is preserved
- traversal is single-pass

Rationale: this removes a common source of manual zip-with-index boilerplate.

4.6 `Iterable[+A].runsWith` / `runBy` / `matchingRuns`

Source: <https://github.com/scala/scala-library-next/issues/30>

Intent: group adjacent elements into maximal runs.

Proposed signatures:

- `def runsWith(p: A => Boolean): Iterator[C]`
- `def runBy[B](f: A => B): Iterator[(B, C)]`
- `def matchingRuns(p: A => Boolean): Iterator[C]`

Behavior:

- defined on `Iterable` (via `IterableOps`), like `grouped` / `sliding`, which this generalizes (boundaries set by a predicate or key instead of a fixed size)
- each run is the receiver's own collection type `C`; runs are produced as a lazy outer `Iterator` (`Iterator[(B, C)]` for `runBy`), exactly like `grouped` returns `Iterator[C]`
- a new run starts when the predicate result or derived key changes
- each run preserves element order
- `matchingRuns` returns only runs whose elements satisfy the predicate
- not offered on `Iterator`: an `Iterator`-native version would need a dedicated lazy iterator (like `GroupedIterator`) and would have to yield a concrete `Seq` instead of `C`, so it is intentionally omitted

Rationale: this is a useful abstraction for grouping adjacent values without manual state tracking.

4.7 `Map[K, +V].getAndRemove`

Source: <https://github.com/scala/scala-library-next/issues/86>

Intent: retrieve a value and return the map with that entry removed.

Proposed signature:

- `def getAndRemove(key: K): (Option[V], C)`

Behavior:

- returns the value if present
- the returned map has the receiver's own type `C`, like `removed`
- returns the map without the key
- returns `None` and the original map when the key is absent

Rationale: this is useful when immutable map update-and-query steps need to stay concise.

4.8 `mutable.Map[K, V].merge`

Source: <https://github.com/scala/scala-library-next/issues/164>

Motivation: <https://github.com/zio/zio-kafka/pull/1079>

Intent: insert a value, or combine it in place with the existing value at a key.

Proposed signature:

- `def merge(key: K, value: V, remappingFunction: (V, V) => V): Option[V]`

Behavior:

- defined on `mutable.Map`, which mutates in place; `V` is invariant there, so no `[V1 >: V]` widening is needed
- inserts `value` when the key is absent
- combines the existing and new values via the function when the key is present
- never removes: the function returns a `V`, so the entry always ends up present (unlike Java's `merge`, which removes on a `null` result, since "we don't do nulls in Scala")
- returns the resulting value as `Option[V]`

Rationale: this is the idiomatic in-place "accumulate into this key" operation (combine-or-insert), the workhorse for counting and grouping. `updateWith` is cumbersome for the common never-remove case, and Java's `Map.merge` is the precedent. The concrete motivation: zio-kafka had to fall back to a `java.util.HashMap` purely to use `merge` while accumulating maximum offsets per partition (`acc.merge(tp, offset, _ max _)`), because Scala's `mutable.Map` has no equivalent, forcing Java-collection interop in the consumer runloop.

4.9 `Map[K, +V].invert`

Source: <https://contributors.scala-lang.org/t/standard-library-now-open-for-improvements-and-suggestions/7337/6>

Intent: swap keys and values, collecting duplicate values.

Proposed signatures:

- ON `Map`: `def invert[V1 >: V]: Map[V1, Set[K]]`
- ON `SeqMap`: `def invert[V1 >: V]: SeqMap[V1, SeqSet[K]]`
- ON `SortedMap`: `def invert[V1 >: V]: SeqMap[V1, SortedSet[K]]`

Behavior:

- each value becomes a key in the result; original keys that shared a value are grouped into a set
- the grouped set's flavor mirrors the receiver map's flavor: `Map` -> `Set`, `SeqMap` -> `SeqSet` (insertion order preserved), `SortedMap` -> `SortedSet` (using the `Ordering[K]` the sorted map already carries, so the keys stay sorted)
- unlike the collection-typed methods elsewhere in this proposal, `invert` cannot use a single `cc`-style return type: inverting turns the original values into the new keys, and the receiver's map factory cannot supply an ordering for those new keys (a `SortedMap[K, V]` has no `Ordering[V]`). Hence the explicit per-flavor variants
- for `SortedMap` the result's outer map is a `SeqMap`, preserving the order in which values were first seen during the sorted-by-key traversal, since the new keys `v1` have no available ordering
- the `SeqMap` variant returns `SeqSet[K]`, so it depends on `SeqSet` (§3.1) and must be sequenced after it

Rationale: this is the map analogue of swapping tuple positions, but for many-to-one relationships.

4.10 `Map[K, +V].unionWith`

Source: <https://contributors.scala-lang.org/t/standard-library-now-open-for-improvements-and-suggestions/7337/24>

Intent: merge two maps while combining values for overlapping keys.

Proposed signature:

- `def unionWith[V2 >: V](that: Map[K, V2], f: (K, V2, V2) => V2): CC[K, V2]`

Behavior:

- keys present in only one map are preserved
- overlapping keys are combined with the supplied function
- returns the receiver's own map type `cc[K, V2]`, like `concat`; the value type widens as needed

Rationale: this is a common merge pattern that is currently verbose to express.

4.11 Set[A].fullIntersection

Source: <https://github.com/scala/scala-library-next/issues/184>

Intent: return the overlap and the elements unique to each side.

Proposed signature:

- `def fullIntersection[B >: A](that: Set[B]): (C, C, CC[B])`

Behavior:

- returns three sets in the order `(elementsOnlyInThis, elementsInBoth, elementsOnlyInThat)`
- the first two use the receiver's own set type `C` (element type `A`); the third is `CC[B]` (the that-only elements, type `B`), like `intersect` / `diff`

Rationale: this gives one pass over the relationship between two sets rather than forcing multiple operations.

4.12 Seq[+A].deleted / updatedWith / splitAround

Source: <https://github.com/scala/scala-library-next/issues/186>, <https://github.com/scala/scala-library-next/issues/187>

Intent: provide small but common sequence-editing operations.

Proposed signatures:

- `def deleted(i: Int): C`
- `def updatedWith[B >: A](i: Int, f: A => Option[B]): CC[B]`
- `def splitAround[A1 >: A](separator: A1): (C, C)`

Behavior:

- `deleted` removes the element at the given index
- `updatedWith` optionally replaces the element at the given index
- `deleted` and `updatedWith` throw `IndexOutOfBoundsException` for an out-of-range index, matching `updated`
- `splitAround` splits around the first element equal to the separator and excludes that separator from both halves; only the first occurrence is the split point, so any later occurrences remain in the second part (e.g. `"bucket/path/prefix".splitAround('/') == ("bucket", "path/prefix")`)
- results use the receiver's own sequence type: `deleted` and each `splitAround` half return `C`, `updatedWith` returns `CC[B]`, like `patch` / `updated` / `splitAt`
- the surrounding order is preserved

Rationale: these methods remove common sequence boilerplate.

4.13 `String.splitToSeq`

Source: <https://github.com/scala/scala-library-next/pull/113>

Intent: return regex splits as a sequence instead of an array.

Proposed signatures:

- `def splitToSeq(regex: String): Seq[String]`
- `def splitToSeq(regex: String, limit: Int): Seq[String]`

Behavior:

- mirrors standard regex split behavior
- preserves token order
- returns a `Seq` result; the concrete type would most likely be `Vector`

Rationale: this avoids array handling in common string-processing code.

4.14 `scala.math.clamp`

Source: <https://github.com/scala/scala-library-next/issues/55>

Intent: constrain values to a closed interval.

Proposed signatures:

- `def clamp(value: Long, lower: Int, upper: Int): Int`
- `def clamp(value: Long, lower: Long, upper: Long): Long`
- `def clamp(value: Float, lower: Float, upper: Float): Float`
- `def clamp(value: Double, lower: Double, upper: Double): Double`
- `def clamp[T](value: T, lower: T, upper: T)(using Ordering[T]): T`

Behavior:

- values below the lower bound return the lower bound
- values above the upper bound return the upper bound
- values inside the range are returned unchanged
- the `Int`-returning overload takes a `Long` value with `Int` bounds, clamping a long into an int range, matching Java's `int clamp(long, int, int)`
- floating-point `NaN` should pass through unchanged
- throws `IllegalArgumentException` when `lower > upper`, matching Java's `Math.clamp`

Rationale: this is a small but common utility, and primitive overloads can avoid boxing overhead.