

---

# **sortedfile Documentation**

***Release 0.1***

**David Wilson**

January 21, 2014







<http://github.com/dw/sortedfile>

When handling large text files it is often desirable to access some subset without first splitting, or using a database where import and index creation is required. When data is already sorted, such as with logs or time series, this can be exploited to efficiently locate interesting subsets. This module is analogous to the [bisection](#) module, allowing search of sorted file content in logarithmic time.

Given a 1 terabyte file, 40 seeks are required, resulting in an *expected* 600ms search on a mechanical disk under pessimistic assumptions. With SSDs having <0.16ms seeks the same scenario could yield 150 searches/second, and significantly more when allowing for caching.



---

## Common Parameters

---

In addition to those described later, each function accepts the following optional parameters:

- key:** Indicates a function (in the style of `sorted(..., key=)`) that maps lines to ordered values to be used for comparison. Provide `key` to extract a unique ID or timestamp. Lines are compared lexicographically by default.
- lo:** Lowest offset in bytes, useful for skipping headers or to constrain a search using a previous search. For line oriented search, one byte prior to this offset is included in order to ensure the first line is considered complete. Defaults to 0.
- hi:** Highest offset in bytes. If the file being searched is weird (e.g. a UNIX special device), specifies the highest bound to access. By default `getsize()` is used to probe the file size.





---

## Functions

---

Five functions are provided in two variants, one for variable length lines and one for fixed-length records. Fixed length versions are more efficient as they require  $\log(\text{length})$  fewer steps.

For line oriented functions, a *seekable file* is any object with functional `readline()` and `seek()`, whereas for record oriented functions it is any object with `read()` and `seek()`.

### 2.1 File Search

`sortedfile.bisect_seek_left(fp, x, lo=None, hi=None, key=None)`

Position the sorted seekable file *fp* such that all preceding lines are less than *x*. If *x* is present, the file is positioned on its first occurrence.

`sortedfile.bisect_seek_right(fp, x, lo=None, hi=None, key=None)`

Position the sorted seekable file *fp* such that all subsequent lines are greater than *x*. If *x* is present, the file is positioned past its last occurrence.

`sortedfile.bisect_seek_fixed_left(fp, n, x, lo=None, hi=None, key=None)`

Position the sorted seekable file *fp* such that all preceding *n* byte records are less than *x*. If *x* is present, the file is positioned on its first occurrence.

`sortedfile.bisect_seek_fixed_right(fp, n, x, lo=None, hi=None, key=None)`

Position the sorted seekable file *fp* such that all subsequent *n* byte records are greater than *x*. If *x* is present, the file is positioned past its last occurrence.

### 2.2 File Iteration

`sortedfile.iter_exclusive(fp, x, y, lo=None, hi=None, key=None)`

Iterate lines of the sorted seekable file *fp* satisfying  $x < \text{line} < y$ .

`sortedfile.iter_inclusive(fp, x, y, lo=None, hi=None, key=None)`

Iterate lines of the sorted seekable file *fp* satisfying  $x \leq \text{line} \leq y$ .

`sortedfile.iter_fixed_exclusive(fp, n, x, y, lo=None, hi=None, key=None)`

Iterate *n* byte records of the sorted seekable file *fp* satisfying  $x < \text{record} < y$ .

`sortedfile.iter_fixed_inclusive(fp, n, x, y, lo=None, hi=None, key=None)`

Iterate *n* byte records of the sorted seekable file *fp* satisfying  $x \leq \text{record} \leq y$ .

## 2.3 Generic Search

These purely implement the bisection algorithm, using a user-provided function to access keys to compare. `lo` and `hi` must always be specified.

`sortedfile.bisect_func_left(x, lo, hi, func)`

Bisect *func*(*i*), returning an index such that preceding values are less than *x*. If *x* is present, the returned index is its first occurrence. EOF is assumed if *func* returns None.

`sortedfile.bisect_func_right(x, lo, hi, func)`

Bisect *func*(*i*), returning an index such that consecutive values are greater than *x*. If *x* is present, the returned index is past its last occurrence. EOF is assumed if *func* returns None.

## 2.4 Utilities

`sortedfile.extents(fp, lo=None, hi=None)`

Return a tuple of the first and last lines from the seekable file *fp*.

`sortedfile.extents_fixed(fp, n, lo=None, hi=None)`

Return a tuple of the first and last *n* byte records from the seekable file *fp*.

`sortedfile.getsize(fp)`

Return the size of *fp* if it is a physical file, `StringIO`, or `mmap.mmap`, otherwise raise `ValueError`.

`sortedfile.warm(fp, lo=None, hi=None)`

Encourage the seekable file *fp* to become cached by reading from it sequentially.

---

## Example

---

Dump the past 15 minutes syslog:

```
YEAR = time.localtime().tm_year

def parse_ts(s):
    """Parse a UNIX syslog date out of 's'."""
    tt = time.strptime(s[:15], '%b %d %H:%M:%S')
    return time.struct_time((YEAR,) + tt[1:])

it = sortedfile.iter_inclusive(
    fp=open('/var/log/messages'),
    x=time.localtime(time.time() - (60 * 15)),
    y=time.localtime(),
    key=parse_ts)
sys.stdout.writelines(it)
```



---

## Performance

---

Tests use a 100GB file containing 1.073 billion 100 byte records with the record number left justified to 99 bytes followed by a newline, allowing both line and record oriented search. The key function uses `str.partition()` to extract the record number before passing it to `int()`, emulating extraction from a record populated with data other than whitespace.

### 4.1 Cold Cache

After clearing the buffer cache on a 2010 Macbook Pro with a Samsung HN-M500MBB:

```
$ ./bench.py
770 recs in 60.44s (avg 78ms dist 33080mb / 12.74/sec)
```

And the fixed record variant:

```
$ ./bench.py fixed
1160 recs in 60.28s (avg 51ms dist 35038mb / 19.24/sec)
```

19 random searches per second on a billion records, not bad for budget spinning rust. `bench.py` could be tweaked to more thoroughly dodge the various caches in play, but seems a fair test as-is.

Reading 100 consecutive records following each search provides some indication of throughput in a common case:

```
$ ./bench.py fixed span100
101303 recs in 60.40s (avg 0.596ms / 1677.13/sec)
```

### 4.2 Hot Cache

`bench.py warm` is more interesting: instead of load uniformly distributed over the set, readers only care about recent data. Requests are generated for the bottom 4% of the file (i.e. 4GB or 43 million records), with an initial warming that pre-caches this region. `mmap.mmap` is used in place of `file` for its significant performance edge when IO is fast (e.g. cached).

After warmup, `fork()` to avail of both cores:

```
$ ./bench.py warm mmap smp
611674 recs in 60.00s (avg 98us dist 0mb / 10194.00/sec)
```

And the fixed variant:

```
$ ./bench.py fixed warm mmap smp
751375 recs in 60.01s (avg 79us dist 0mb / 12521.16/sec)
```

Around 6250 random reads per second per core over 43 million records from a set of 1 billion, using only sorted text and a 23 line function.

And for consecutive sequential reads:

```
$ ./bench.py fixed mmap smp warm span100
15396036 recs in 60.01s (avg 0.004ms / 256578.04/sec)
```

## 5.1 Threads and `mmap.mmap`

Since `mmap.mmap` does not drop the GIL during reads, page faults will hang a process attempting to serve cold data to clients using threads. `file` does not have this problem, nor does forking a process per client, or maintaining a process pool.

## 5.2 Buffering

When using `file`, performance may vary according to the buffer size set for the file and target workload. For random reads of single records, a buffer that approximates double the average record length will work better, whereas for searches followed by sequential reads a larger buffer may be preferable.

## 5.3 Interesting Uses

Since the input's size is checked on each call when `hi` isn't specified, it is trivial to have concurrent readers and writers, so long as writers take care to open the file as `O_APPEND`, and emit records no larger than the maximum atomic write size for the operating system. On Linux, since `write()` holds a lock, it should be possible to write records of arbitrary size.

However since each region's midpoint will change as the file grows, this mode may not interact well with caching without further mitigation. Another caveat is that under IO/scheduling contention, it is possible for writes from multiple processes to occur out of order, although depending on the granularity of the key this may not be a problem.

When dealing with many small objects (e.g. lists of strings or integers) that can be easily serialized in-order to a `StringIO`, RAM use can be greatly reduced while still allowing fast access. For example with lists of integers, memory usage drops by up to 90% on a 64 bit machine.

## 5.4 Improvements

It should be possible to squeeze more performance from `file` by paying attention to the operating system's needs, for example read alignment and `posix_fadvise`. Single-threaded `file` performance is significantly worse than `mmap.mmap`, this is almost certainly not inherent, more likely it is due to a badly designed test.

Additionally unlike `mmap.mmap`, calling `file.seek()` invokes a real system call, which may be generating more work than is apparent. The implementation could be improved to remove at least some of these calls.