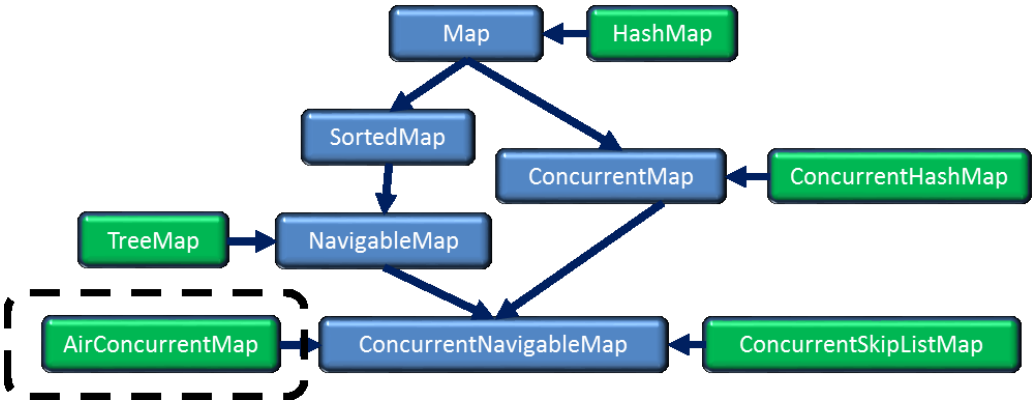
AirConcurrentMap Performance Testing

Extreme ConcurrentNavigableMap from BoilerBay

Java Version 3.0 23-6-2016

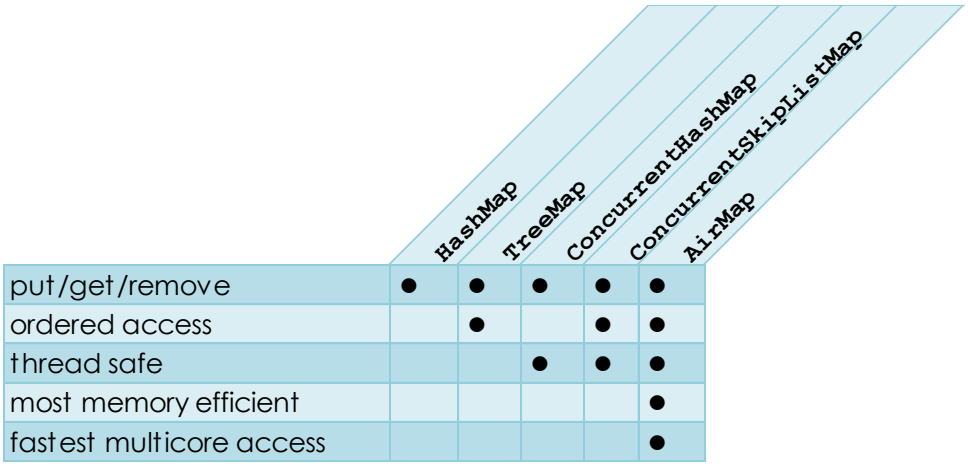
The AirConcurrentMap for Java product is a multi-core capable concurrent ordered Java Map implementation that out-competes the standard Java library Map implementations and others in multiple ways – as shown by the graphical test results in this document. It provides drop-in increased performance up to 90% faster for put(), get(), remove() and all other search operations, and up to four times faster for iteration, while increasing memory efficiency – giving at least 40% to 60% more capacity - for a wide range of usages we test here. There is also a *Visitor* API that improves scan speed with or without internal threading which is faster than the Java 8 streams system.

Here is where AirConcurrentMap fits into the standard Map hierarchy. In this diagram, the blue boxes are the standard Map interfaces, and the green boxes are AirConcurrentMap and the library Map implementations we test against. Note that AirConcurrentMap includes all of the capabilities of any other Map, hence is a compatible replacement for any.



Feature comparison

Below we show the most important characteristics of the java built-in Maps versus AirConcurrentMap. AirConcurrentMap includes all of the features.



	HashMap	TreeMap	ConcurrentHashMap	ConcurrentSkipListMap	AirMap
put/get/remove	●	●	●	●	●
ordered access		●		●	●
thread safe			●	●	●
most memory efficient					●
fastest multicore access					●

Performance Advantages

AirConcurrentMap is the Most Memory Efficient Map

The test results show that AirConcurrentMap is more memory efficient than any other standard Map above a small size threshold. In these tests, we show memory efficiency by growing the JVM by adding as many entries as possible. The limit is reached when either the performance drops to an unusable level or OutOfMemoryError is thrown. An unusable performance level usually occurs with all cores busy and overall system response very low. Note that each of these maps has a corresponding Set implementation, like AirSet. AirSet is even more memory efficient relative to the standard Set implementations than AirConcurrentMap is relative to the standard Map implementations.

Memory efficiency is vital, because the stability of an entire application is lost when OutOfMemoryError is thrown into any thread, no matter what Map implementation or other data structure is used. OOME can cause any data structures maintained by the application or libraries to become corrupted, and there is virtually no way to recover. Nulling out references in order to allow GC to reclaim space is not guaranteed to prevent OOME, because temporary Objects are frequently constructed both by the application and internally by the libraries. Also, the response of nulling-out references may be too late to prevent OOME reaching other threads. The only reliable solution is to exit the JVM immediately, before further program output also becomes corrupted. Sometimes, however, memory pressure will cause the JVM simply to stall with all cores busy in the

garbage collector, and the entire system, not just the Java application, will slow down dramatically or almost freeze. Running out of memory is to be avoided, and AirConcurrentMap will help.

AirConcurrentMap has the Fastest Iterators

The keyset, entrySet, and values iterators are faster than any other standard Map by up to four times, except that ConcurrentSkipListMap is slightly faster below a small entry count threshold. Look at the graphs below to see the enormous speed of the AirConcurrentMap iterators.

AirConcurrentMap is faster than ConcurrentSkipListMap

ConcurrentSkipListMap is AirConcurrentMap's only competitor for feature completeness, but AirConcurrentMap beats it in these tests for single-threaded or multi-threaded applications, for modifying or accessing the Map, over a wide range of thread and core counts, ranging from a small threshold up to a full JVM. We show graphically get and put speed versus entry count, entry count versus time for put plus mixtures of concurrent put, get, and remove. We show that ConcurrentSkipListMap has a 0.2 second 'flat spot' during some mixed loads while fast gets prevent put, so we sometimes omit those results.

AirConcurrentMap is the Fastest Way to Load a NavigableMap

AirConcurrentMap even beats the TreeMap class for continuous put, because TreeMap can only use one core. The only other way to build a standard NavigableMap is to use ConcurrentSkipListMap, which is also slower.

AirConcurrentMap is the fastest ConcurrentMap for multi-core

The standard ConcurrentMap implementations were tested with eight cores, and in the results AirConcurrentMap even beats the ConcurrentHashMap class, which is not even a NavigableMap. ConcurrentHashMap actually slows down in going from one to eight cores in these tests.

Visitor API provides fastest scans

There is a custom Visitor API which allows faster scanning than Iterators – about 2 times faster. The API uses a single 'visit(Object key, Object value)' method in a client class that extends Visitor that is a call-back from the Map. The client's Visitor extension classes are easily re-usable as

declarative components. The iteration keeps up with Java 8 parallel streams.

Parallel Visitors use an internal Thread pool

The fastest scanning possible over any tested Map, concurrent or not, ordered or not, is provided by the client class extending the ThreadedVisitor class. Two new methods are required. Performance is several times that of Java 8 parallel streams.

Usage Advantages

Trivial Installation

AirConcurrentMap can be plugged in immediately into any application or environment as a transparent replacement with no code modifications except for changing the statements where the Maps are constructed to construct instead a com.infinitydb.air.AirConcurrentMap. (Of course also the com.infinitydb.air.jar must be added to the classpath). Such program changes usually take only minutes, and the performance results can be tested immediately.

Universal Compatibility

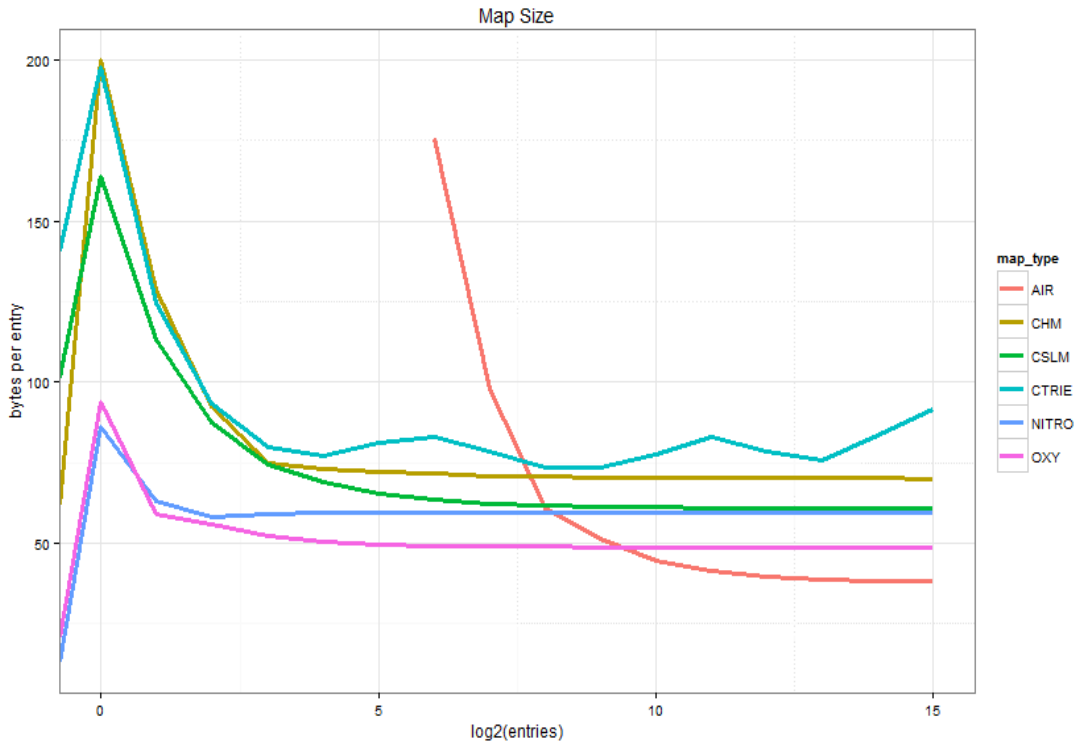
By being a ConcurrentNavigableMap, AirConcurrentMap includes all of the capabilities of any of the kinds of standard Maps defined in Java 1.6, so it is a drop-in replacement for any of them. For example even java.util.HashMap, or java.util.TreeMap, can be replaced easily to increase memory efficiency. (As for any SortedMap, keys must implement Comparable or else a custom Comparator must be supplied. Most classes likely to be used as keys are already Comparable, such as Strings or Numbers.) AirConcurrentMap will run in Java 1.6 or later, where the ConcurrentNavigableMap interface was defined. Java 1.5 compatibility may be released in the future.

Thread-safe

It is dangerous in a multi-threaded application to use the non-concurrent Map implementations, either out of habit, or for a perceived need for speed, because application bugs that allow shared access will cause undefined behavior such as map internal data structure corruption and incorrect results that can be very hard to discover, reproduce, and fix.

Memory Efficiency Comparison

This logarithmic diagram shows that AirConcurentMap in red becomes the most efficient Map above about 1024 Entries. It has about twice the capacity of ConcurrentSkipListMap or ConcurrentHashMap. The results are similar for TreeMap or HashMap. The Ctrie is the worst. The other very efficient Maps -'Nitro' and 'Oxy' - are in development by Boiler Bay. It is interesting to note that all Maps are particularly inefficient for a single Entry.



Performance Comparisons

(Please note that the results of performance measurements shown here cannot be taken as guarantees or warrantees of performance. Best efforts were made to keep all tests fair and unbiased.)

GC Limitations

Some of the tests shown here pre-allocate the JVM heap space to full size (using –Xms2g) in order to avoid garbage collection delay effects. Involving GC in the tests is important but makes the tests difficult to interpret due to occasional, significant slowdowns. GC becomes significant for most long-running applications such as web servers and also for single-run applications that are performance-limited by the time to load data into memory.

What We Test

We test put(), get(), and remove() for performance, plus iterators over the keySet, entrySet and values views and internally parallel visiting. There is much other functionality, but these are the core operations and many of the other operations are implemented as varieties of these, such as higher(), lower(), ceiling(), and floor(), which are the essential NavigableMap methods, and which therefore have the same performance characteristics.

The Test System

The test system uses the default 2GB memory limit and runs this 64-bit Java version:

```
$ java -version
java version "1.8.0"
Java(TM) SE Runtime Environment (build 1.8.0-b132)
Java HotSpot(TM) Client VM (build 25.0-b70, mixed mode)
```

with this hardware:

Intel Core i7-3615QM 2.3GHz 8GBRAM 64-bit Windows 7 64-bits.

The system is quad-core, with the usual Intel hyperthreading, which Windows treats as eight logical cores. Windows can execute one thread per logical core at any moment in time, and if there are more runnable threads than eight, Windows (like Linux or any OS) will switch from thread to thread (a 'context switch') at a rate in the millisecond range, giving each thread a relatively fair share of execution time overall, considered over a period of many milliseconds.

Hyperthreading

The eight logical cores in an Intel hyperthreaded processor like the i7 do not provide twice the performance of four physical cores. Instead, each of the four cores can execute two threads at once, and depending on many factors, the combined pair of threads will run from 1 times to 1.5 times as fast as one thread would on that core. Adding a second thread to a given busy core - thereby 'activating' the hyperthreading feature - never apparently slows down the execution of the first thread on that core. When the number of threads is eight, one should not expect eight times the performance of one thread. As the thread count increases, the best possible scaling will be approximately proportional to thread count up to four threads, and then there will be a linear scaling at half the slope up to eight, followed by a flattening out. In general, concurrent systems actually slow down rather than staying flat after the peak Thread count is reached due to inter-thread interference of various kinds, however AirConcurrentMap is relatively immune to excess Threading.

Another complication in multi-threading performance testing is the 'turbo-boost' power-saving feature of the i7, which increases the clock rate substantially when only one core is active.

Test Variables

We show most of the performance curves based on the number of entries currently in the Map, but some tests are cumulative entries over time. The keys are random Long's, with a single value which is a Long (with the value 42). We use custom random number generators – one per thread - to get the necessary speed. The shapes of the curves for String keys of 20 chars were similar but the curves are not shown. String or other keys should produce similar results because the performance is basically limited to the theoretical minimum of log2(n) comparisons.

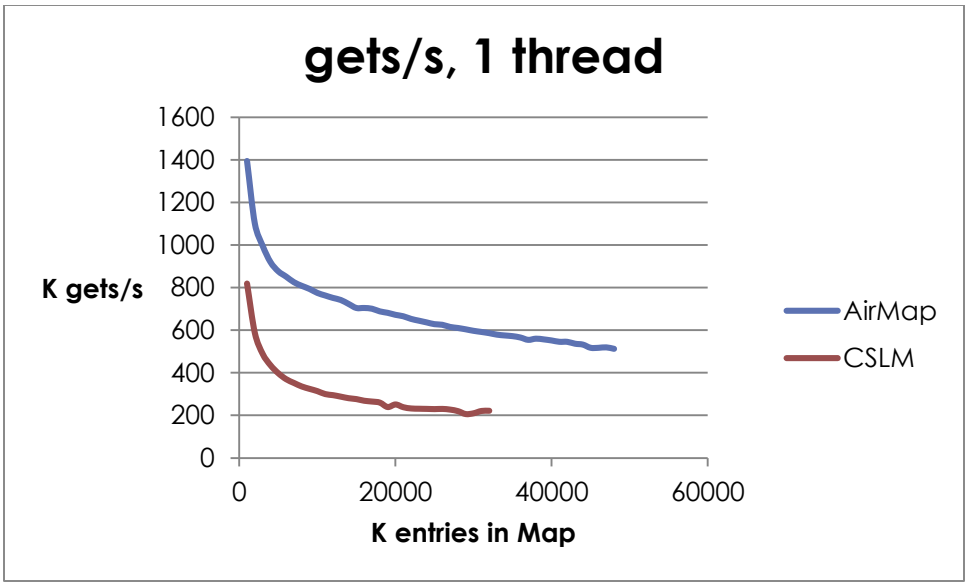
Note that the curves for the CSLM end with much fewer entries in the Map than AirConcurrentMap, because the tests are run until the JVM slows down to an unusable level, or else the JVM throws OutOfMemoryError into one or more threads. This is one way we demonstrate AirConcurrentMap's higher memory efficiency. We also do overall, non-graphical tests for all of the standard maps and AirConcurrentMap.

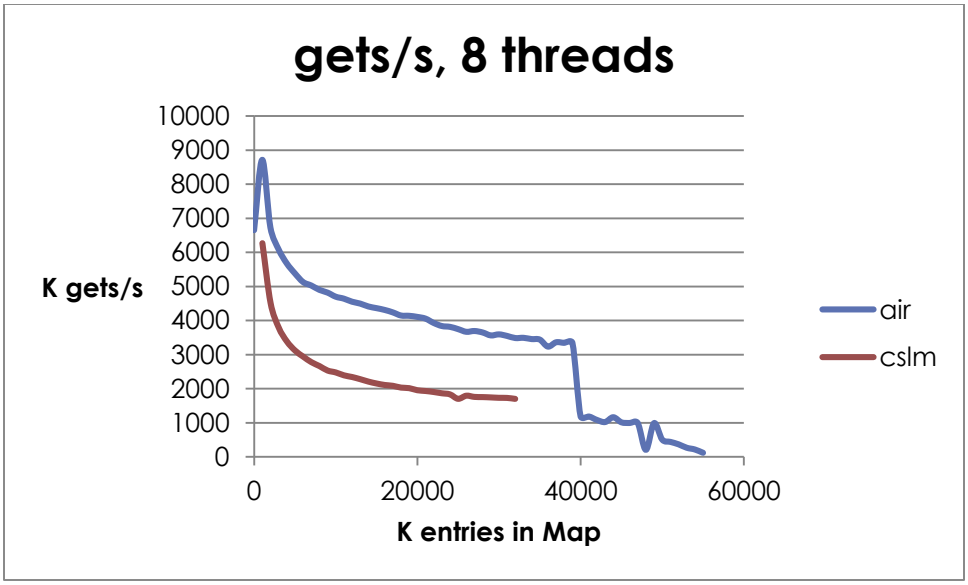
First we will test gets and puts separately, then mixtures, and also mixtures with put, get, and remove(), then iterators.

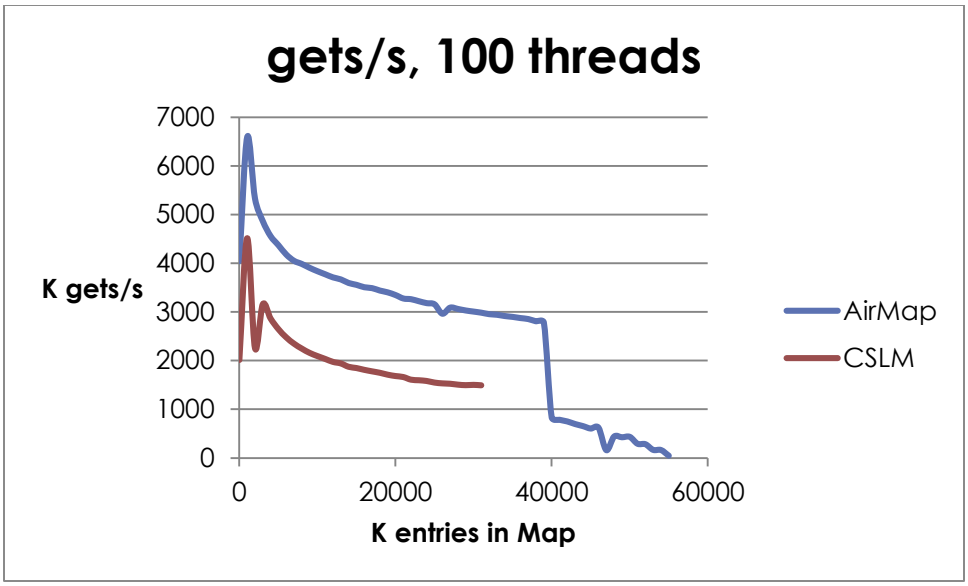
Get and Put Versus ConcurrentSkipList

The get tests

Below you can see that AirConcurrentMap's get() method is about twice as fast as java.util.concurrent.ConcurrentSkipListMap i.e. 'CSLM' for single-threaded access. This is the rough pattern shown by other tests as well. While AirConcurrentMap takes full advantage of multicore opportunities, it is also superior in single-thread situations as can be seen.

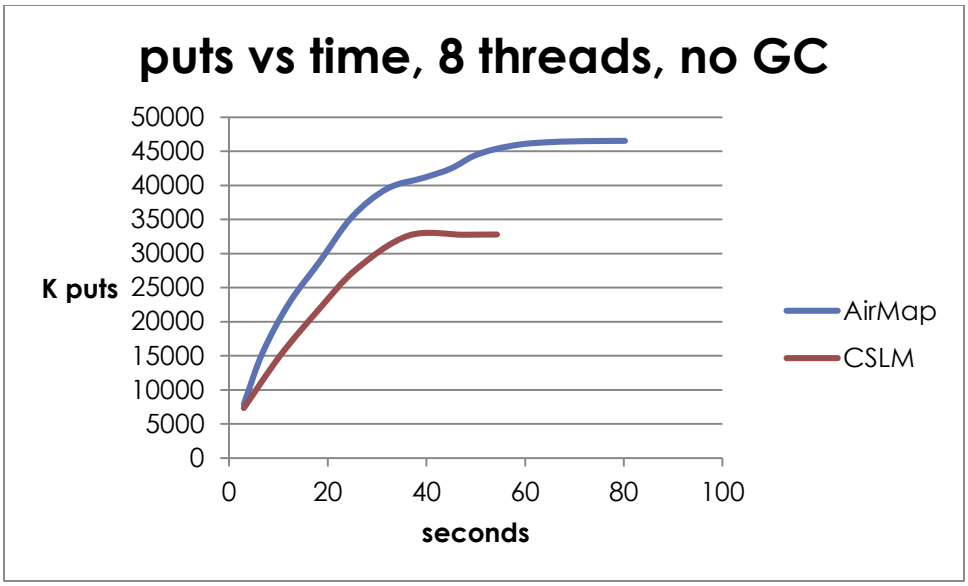

gets/s, 1 thread
K gets/s
1600
1400
1200
1000
800
600
400
200
0
0
20000
40000
60000
K entries in Map
AirMap
CSLM

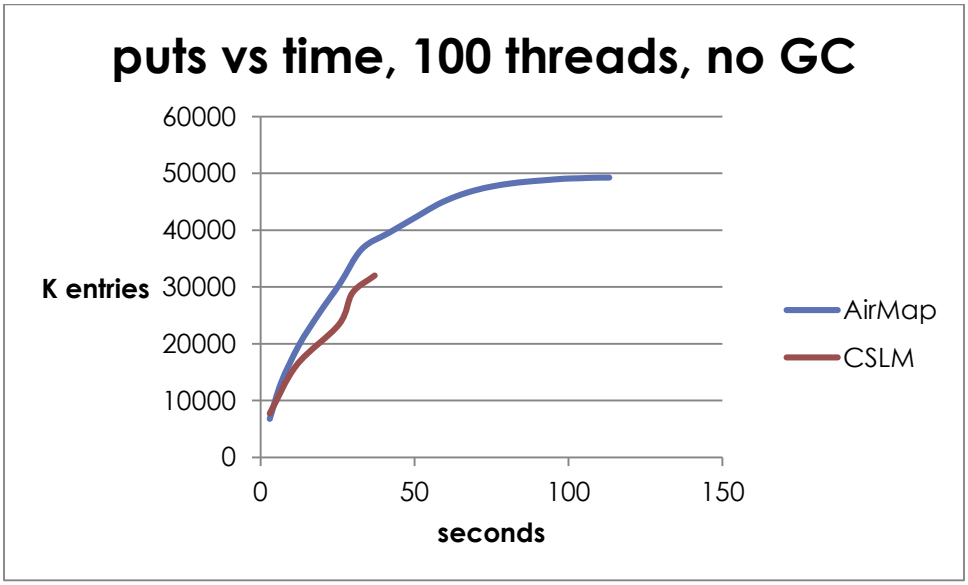

gets/s, 8 threads
K gets/s
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
20000
40000
60000
K entries in Map
air
cslm



The Put Tests

The put() method can be measured several ways. One way is to eliminate the complexity and unpredictable timing variations produced by the garbage collector by using the –Xms command-line switch to set the minimum JVM size equal to the maximum JVM size, so that the JVM will immediately allocate as much memory as it is allowed, and GC is *almost* eliminated. This provides repeatable and clear test results, as shown below for 1 thread, 8 threads, and 100 threads. Note again that the AirConcurrentMap curves end later as a result of the better memory efficiency.

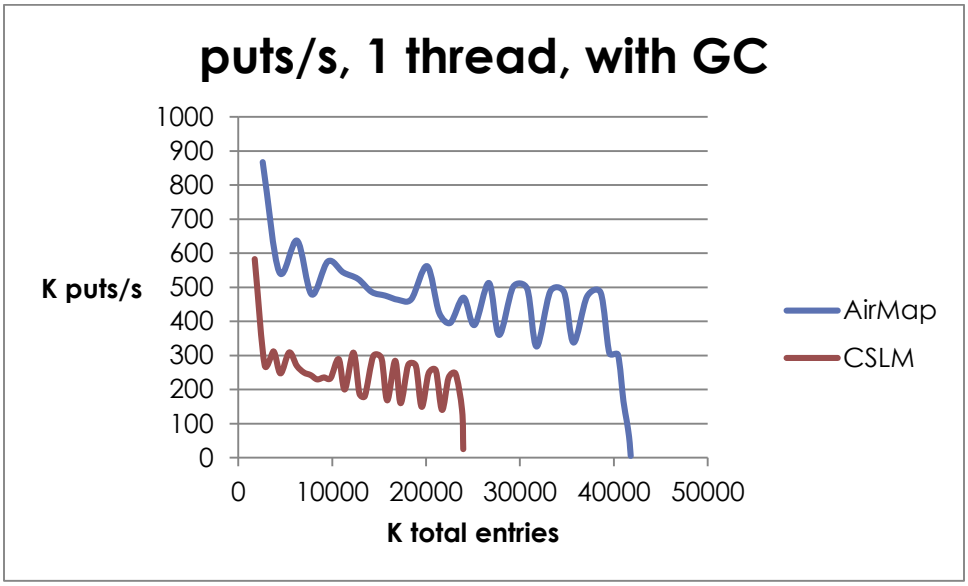


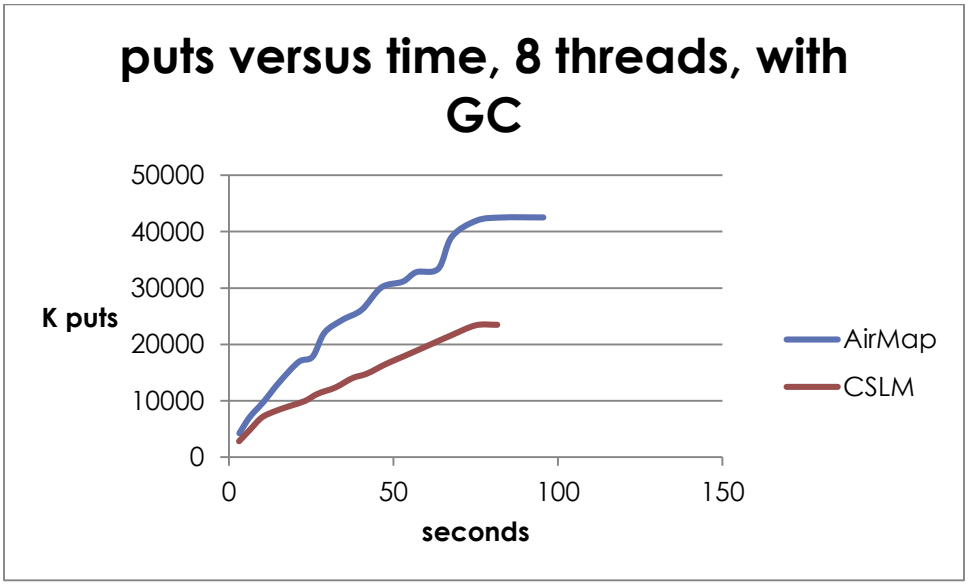


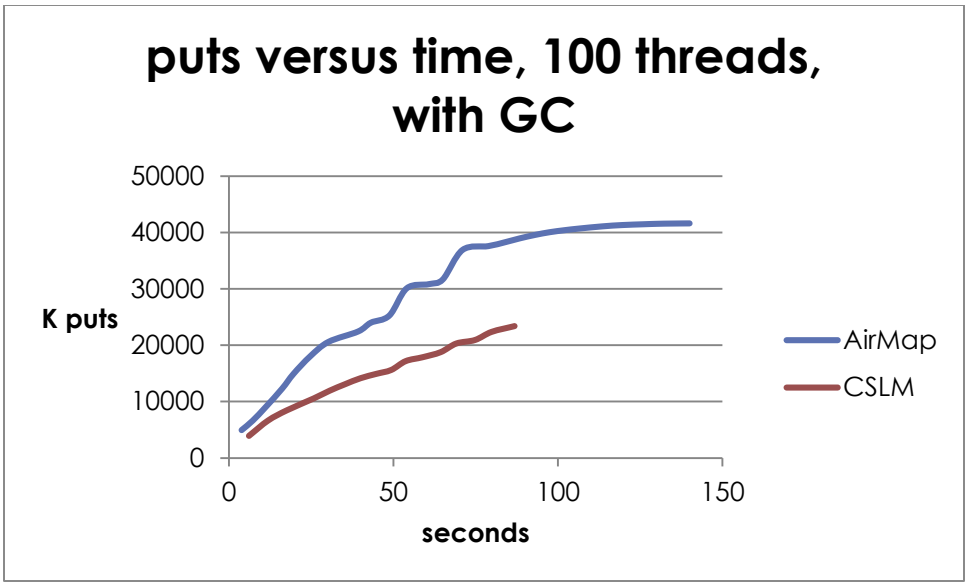
Put Tests with Garbage Collection

Now we do these put tests with GC, by allowing the JVM to choose its own smallest size and then grow as entries arrive. This clearly shows the effects of the GC occasionally taking some cycles, and causing ripples in the curves. Again, for the 8 and 100 thread cases, we show the cumulative puts versus time instead of puts per second to help average out these effects, which are worse in those cases. Note that in these cases CSLM can store somewhat over 20M entries, while AirConcurrentMap stores somewhat over 40M before the GC overhead becomes too great and the processes either slow down to an unusable level or throw

OutOfMemoryErrors.. The garbage collector selected is the default for ORACLE Java 8.

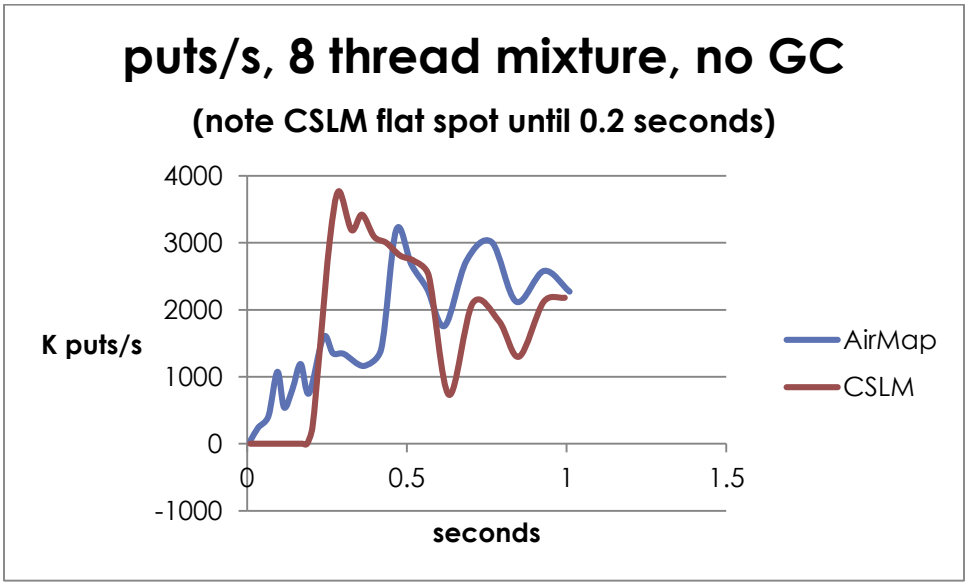




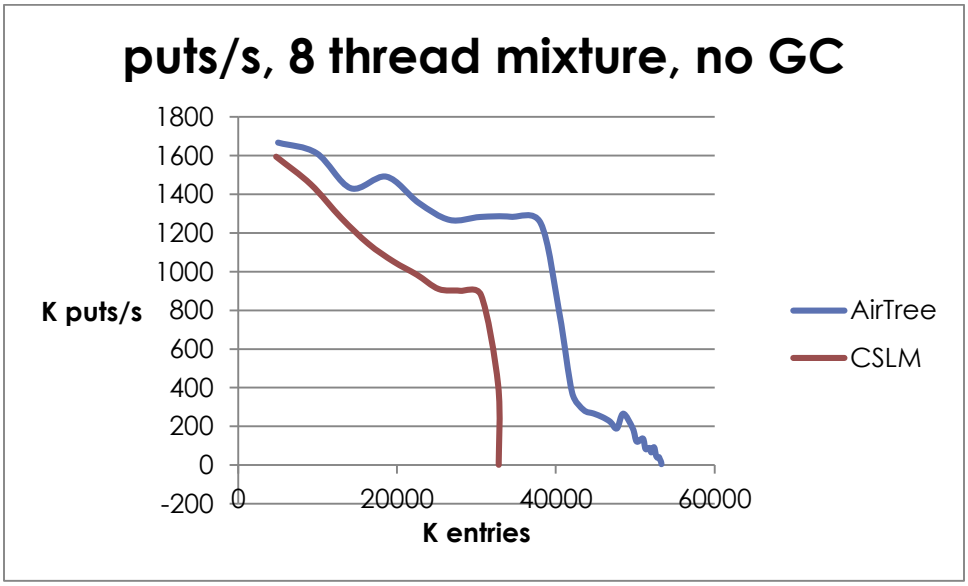


Mixture Tests and the CSLM 'Flat Spot'

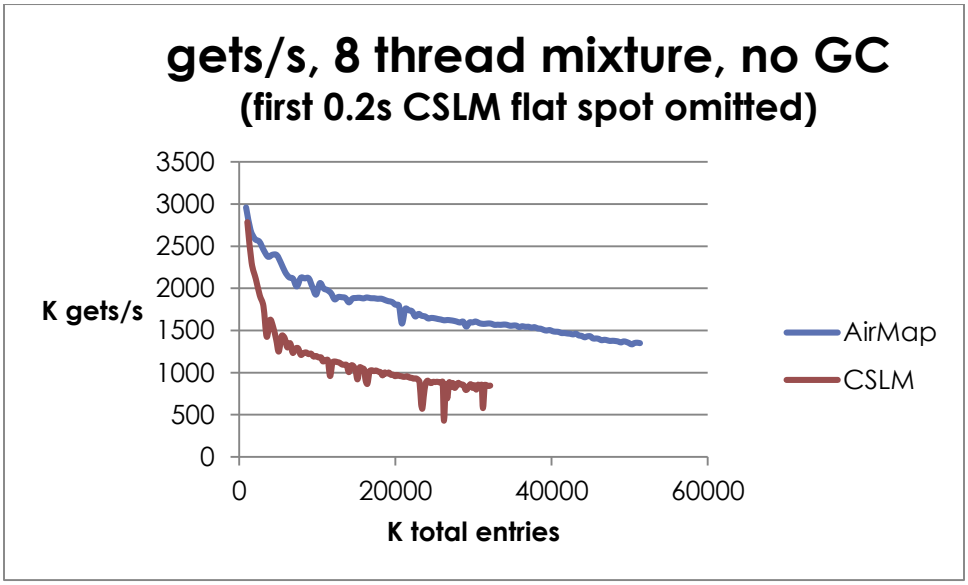
Testing mixtures of four put threads and four get threads is more complex. We are using no GC (by setting –Xms2g). It appears that CSLM has a 'flat spot' at the start of the run, while the Map is growing, that lasts about 0.2 seconds, apparently due to interference with the get threads. During that time, CSLM fails to add entries and put is stalled, but once a few entries actually reach the map, the gets no longer interfere so much, and the Map goes into a normal mixed put/get state.



Once CSLM gets past the flat spot, it increases its put speed for about 0.5 seconds, but with larger entry counts, AirConcurrentMap takes the lead:



During the flat spot, CSLM is reading an empty Map, so its get speed is extremely high. In order to compare AirConcurrentMap and CSLM during the mixed test for get, we omit the first 0.2 second, so that there are truly mixed puts and gets going on. The result is:



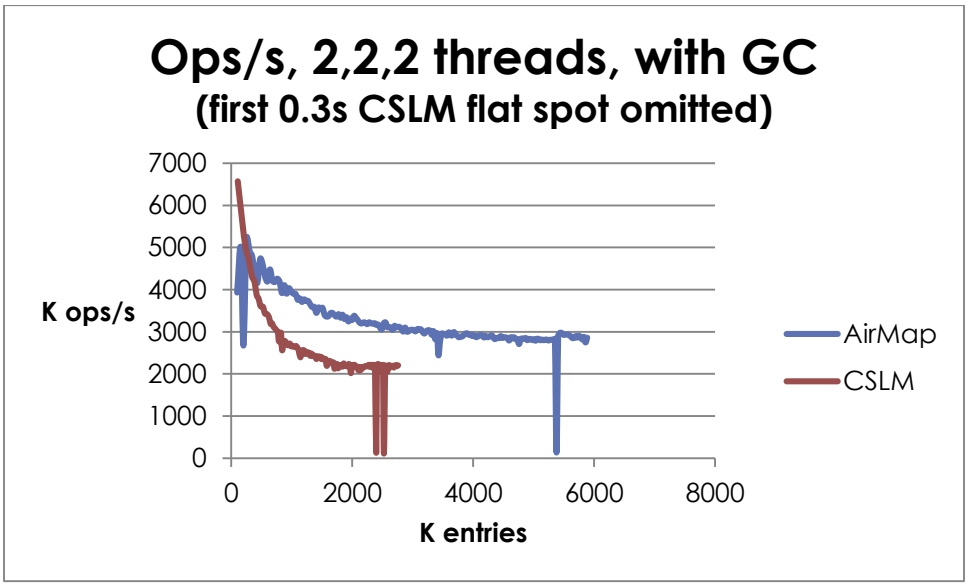
As can be seen, the two maps are approximately tied at the start, although again, during the initial 0.2 second, we omit the samples when the CSLM Map is empty due to its 'flat spot'. If the initial 0.2 seconds were to be shown in the graph, there would be a very tall relatively meaningless initial portion of the CSLM curve that would push the rest of the data down in the chart to near the bottom, where it would be difficult to

interpret. During the flat spot, CSLM is simply retrieving nulls, but it does so very quickly.

Mixed Tests Including Remove

When we test a mixture of put, get, and remove together, the results are hard to visualize because there are too many curves, so we just test the sum of the operations per second. Once again, we have removed the initial CSLM flat spot of approximately 0.2 seconds during which it did no puts. During that time, CSLM was returning null for the gets and removing no entries, so at that time those operations are artificially fast.

Here are the results for two threads each of put, get, and remove, with GC happening because we did not use the –Xms2g command-line switch. The 'drips' are the GC events. Note that the JVM in this test is never actually reaching its size limit, and the length of these curves is not meaningful, whereas it is meaningful in the other charts, where the AirConcurrentMap memory advantage shows up. The three kinds of operations are relatively balanced.



Put Performance Versus All Standard Maps

Below is a typical output of a test that compares the put performance of all of the standard Map implementations plus AirConcurrentMap, for single threads and for eight threads. We run each test by inserting random Long keys with a constant value until used memory reaches ½ of maximum memory, so there are no OutOfMemoryErrors, but there is some

GC. The first tested Map is tested twice to make sure memory is initialized fairly. The test system is the same as for the tests above.

Here are some conclusions we draw from a typical output of this test (the AirConcurrentMap is called 'LayeredAirTree' in the output):

- AirConcurrentMap is the most memory efficient – 40% to 60% more capacity
- AirConcurrentMap is the fastest for 8 threads (only 8 were tested)
- AirConcurrentMap is always faster than ConcurrentSkipListMap
- AirConcurrentMap is the fastest at loading a NavigableMap.

The NavigableMap implementations are TreeMap, CSLM, and AirConcurrentMap. When initializing or 'loading' a NavigableMap, of course we use multiple cores if the Map supports them: this means that TreeMap must use only one core. The ConcurrentHashMap actually slows down with 8 threads,

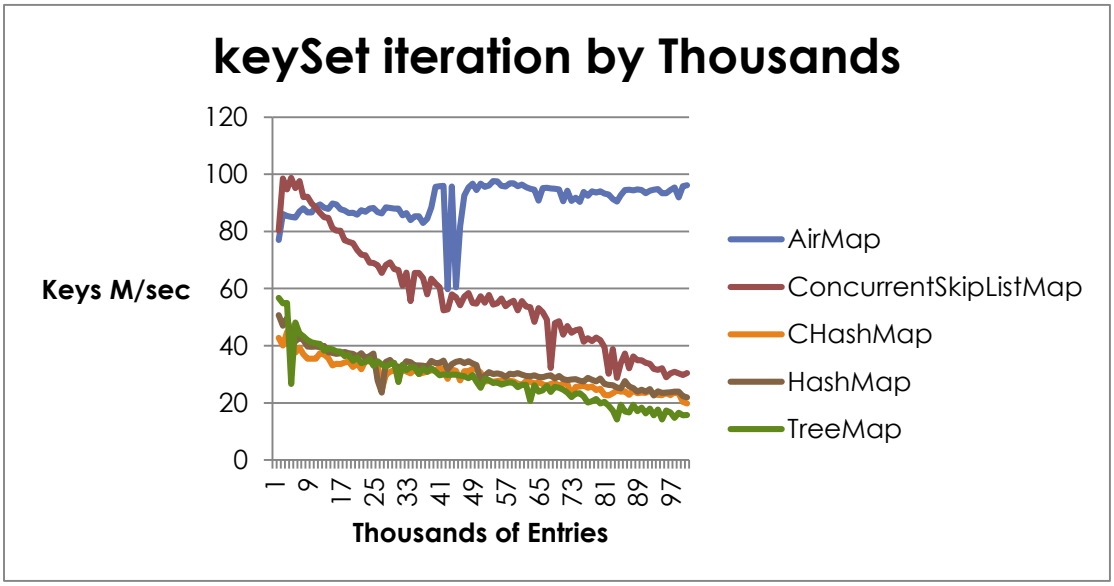
```
java.version=1.8.0
java.vm.version=25.0-b70
KeyType=LONG
Map memory efficiency test maxMemory=1858.60096
Multi-Threads Tests: threadCount=8
TestClass=java.util.concurrent.ConcurrentHashMap
        start run: memory total= 133.693 free= 132.468 used=   1.226
        end run:   memory total=1066.402 free= 131.362 used= 935.040
        entries=12.68M t=62.51 entries/s=202886
TestClass=java.util.concurrent.ConcurrentHashMap
        start run: memory total=1035.469 free=1034.476 used=   0.993
        end run:   memory total=1166.017 free= 215.756 used= 950.261
        entries=12.60M t=19.42 entries/s=648777
TestClass=java.util.concurrent.ConcurrentSkipListMap
        start run: memory total=1172.308 free=1171.212 used=   1.096
        end run:   memory total=1069.548 free= 140.237 used= 929.310
        entries=15.33M t=21.07 entries/s=727584
TestClass=com.infinitydb.air.LayeredAirTree
        start run: memory total=1034.420 free=1033.447 used=   0.973
        end run:   memory total=1307.050 free= 377.619 used= 929.431
        entries=24.04M t=18.45 entries/s=1303436
Single-Thread Tests
TestClass=java.util.HashMap
        start run: memory total=1316.487 free=1315.432 used=   1.055
        end run:   memory total=1350.042 free= 420.339 used= 929.702
        entries=12.95M t= 7.90 entries/s=1638293
TestClass=java.util.HashMap
        start run: memory total=1350.042 free=1349.021 used=   1.021
        end run:   memory total=1374.683 free= 432.198 used= 942.485
        entries=12.60M t= 6.43 entries/s=1960054
TestClass=java.util.concurrent.ConcurrentHashMap
        start run: memory total=1374.683 free=1373.691 used=   0.992
        end run:   memory total=1592.787 free= 643.311 used= 949.476
        entries=12.60M t=15.06 entries/s=837019
TestClass=java.util.TreeMap
        start run: memory total=1592.787 free=1591.800 used=   0.987
        end run:   memory total=1616.904 free= 685.503 used= 931.401
        entries=14.46M t=19.16 entries/s=755077
TestClass=java.util.concurrent.ConcurrentSkipListMap
        start run: memory total=1617.428 free=1616.456 used=   0.973
        end run:   memory total=1625.293 free= 694.580 used= 930.713
        entries=15.41M t=40.06 entries/s=384676
TestClass=com.infinitydb.air.LayeredAirTree
```

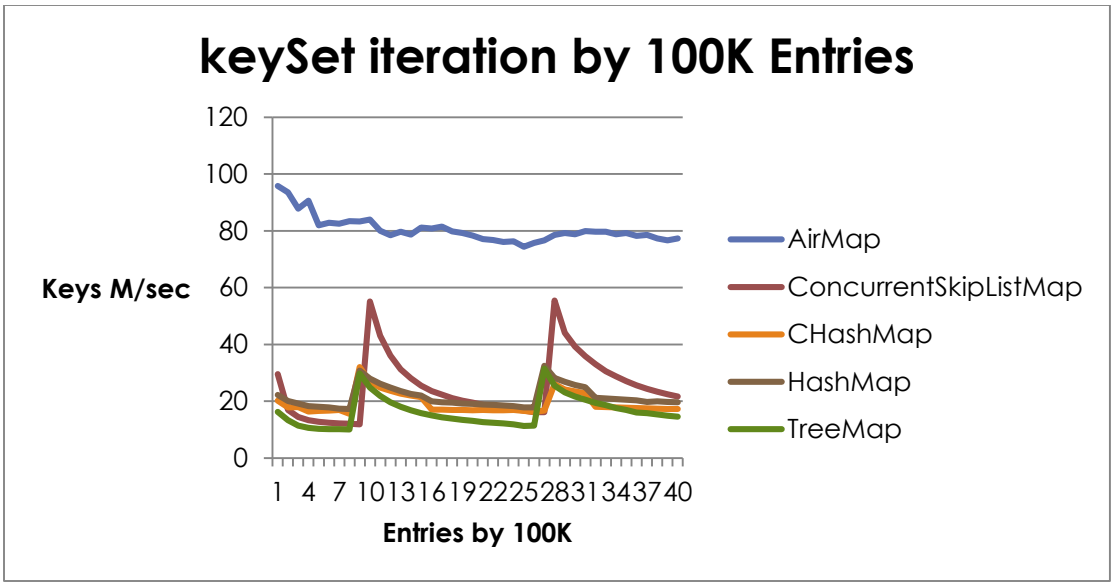
```
start run: memory total=1625.817 free=1624.838 used=   0.979
end run:   memory total=1626.341 free= 695.161 used= 931.180
entries=21.99M t=36.34 entries/s=605189
```

Iterators Versus All Standard Maps

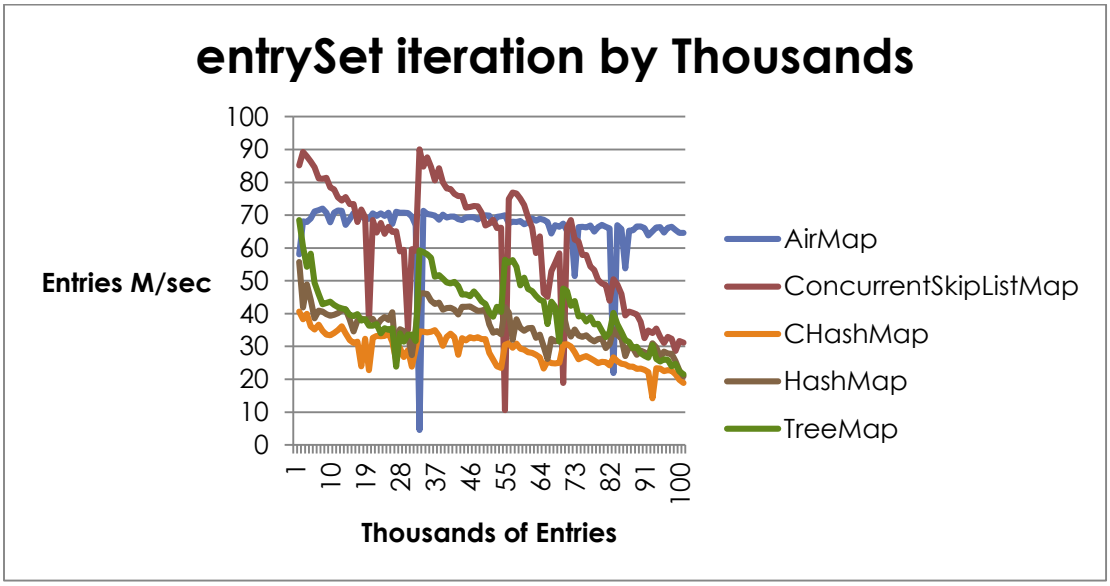
The iteration of AirConcurrentMap's keySet(), entrySet(), and values() is very fast, with the advantage increasing with set size until it is a multiple of the standard implementations. We have two graphs here for each kind of collection view: one by thousands of Entries, and one by 100's of Thousands of Entries. At the lowest sizes, the AirConcurrentMap collection views are sometimes somewhat slower than ConcurrentSkipListMap, but they are normally faster than all the rest, and all of the standard Maps quickly lose ground as the Maps fill up. The performance of the standard Maps is relatively unpredictable and variable, while the AirConcurrentMap sets are relatively steady over most of the range. The measurements are in terms of millions of Entries scanned per second.

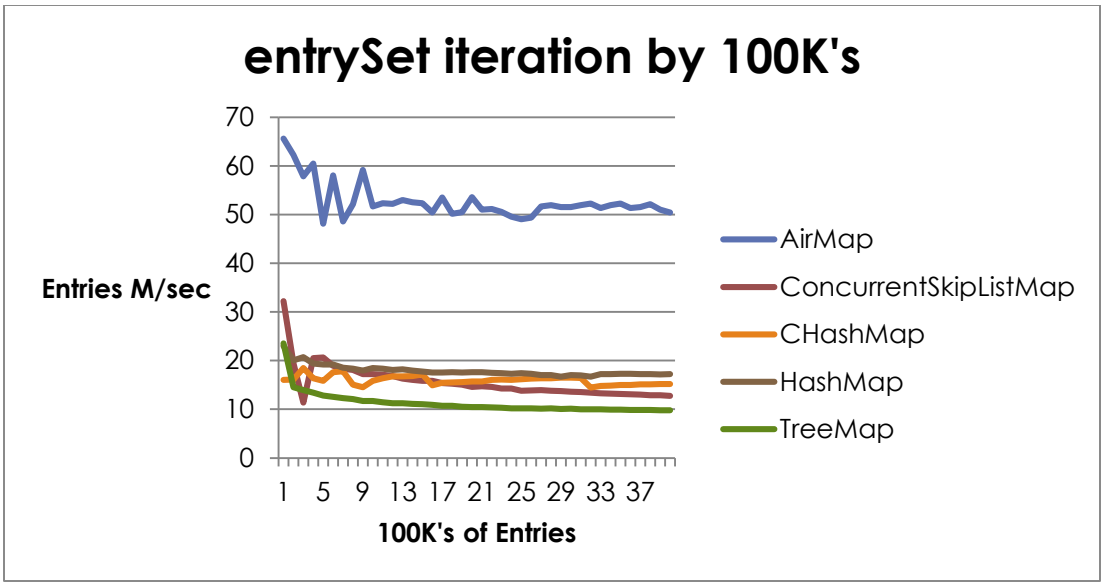
These tests are for one thread.



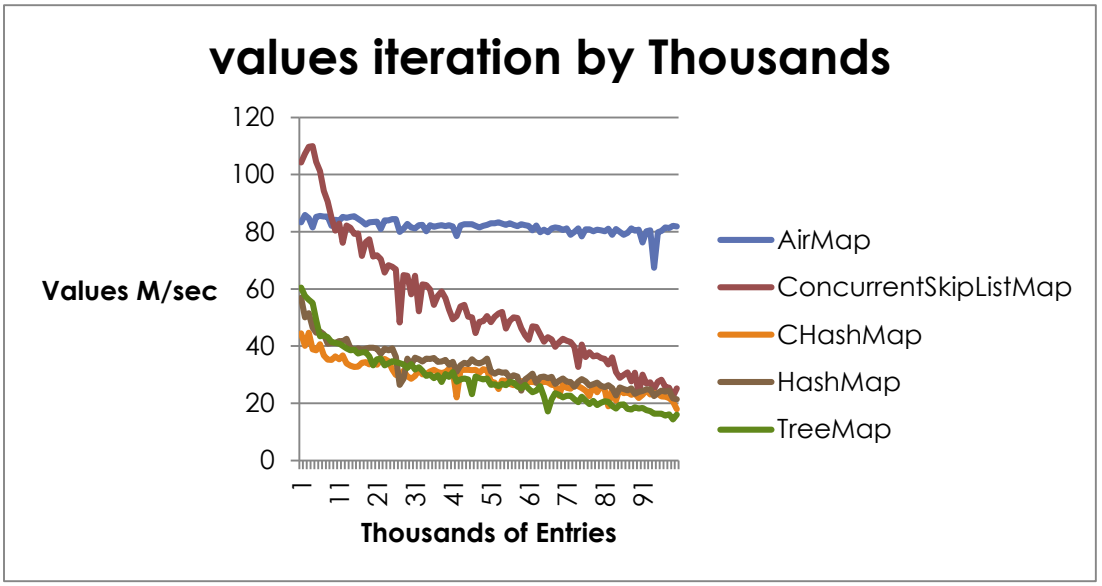


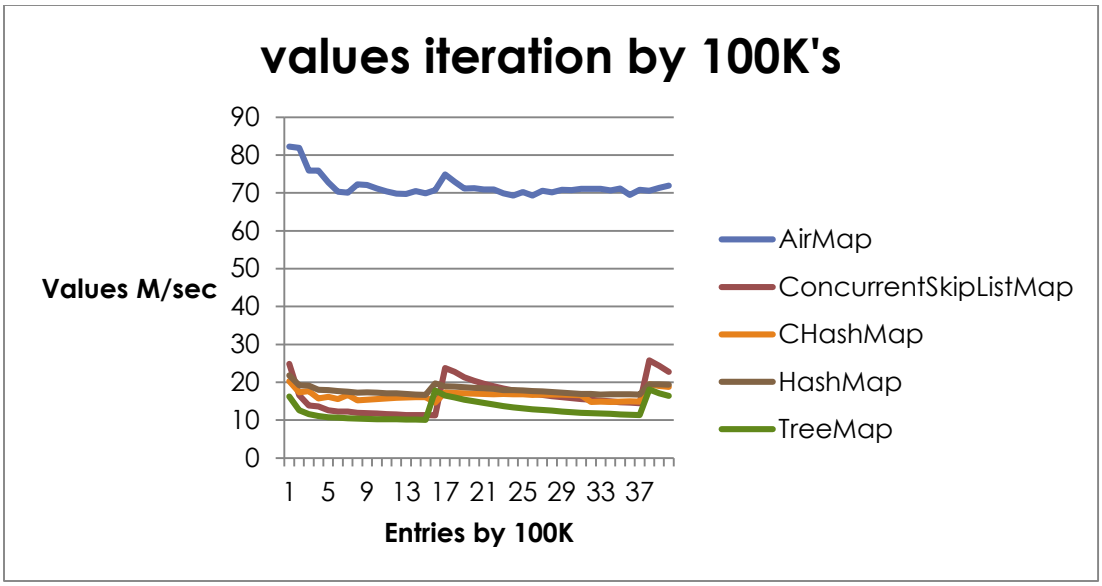
The confusing results for small sizes of entrySet() are probably due to the fact that Map.Entry Objects are being created very rapidly, and the garbage collector gets involved. This may account for the occasional 'drips' where performance drops for just a few samples. Again, AirConcurrentMap is relatively fast and steady anyway.



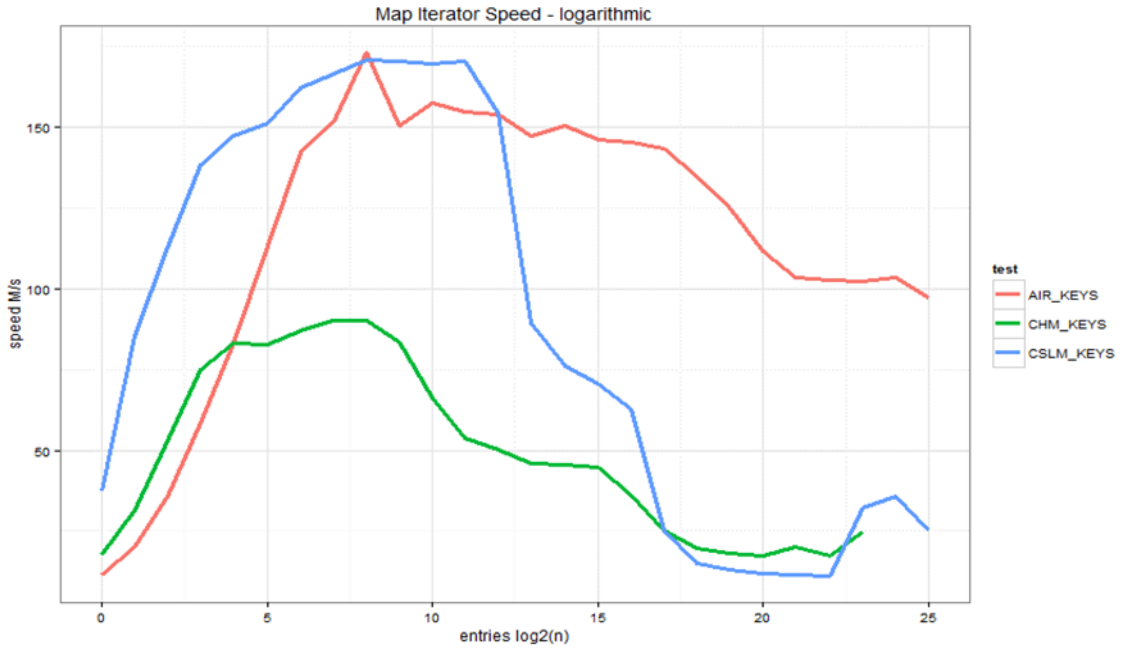

entrySet iteration by 100K's
Entries M/sec
70
60
50
40
30
20
10
0
1 5 9 13 17 21 25 29 33 37
100K's of Entries
AirMap
ConcurrentSkipListMap
CHashMap
HashMap
TreeMap

++


values iteration by Thousands
Values M/sec
120
100
80
60
40
20
0
1 11 21 31 41 51 61 71 81 91
Thousands of Entries
AirMap
ConcurrentSkipListMap
CHashMap
HashMap
TreeMap



Here is another summary of the performance, now with a logarithmic scale. Towards the right, above about $2^{10,}$ which is 1024 entries, AirConcurrentMap clearly takes over.



MapVisitors

There is another scanning performance improvement possible with a custom technique in AirConcurrentMap v2.0. Now, a MapVisitor class can

be extended by implementing the visitor(Object key, Object value) method to provide a re-usable component for scanning. For example, the following code creates a reusable printer visitor:

```
static class PrintVisitor extends MapVisitor<Object, Object> {
    public void visit(Object key, Object value) {
        System.out.println("key=" + key + " value=" + value);
    }
}
```

Which is used simply:

```
map.getVisitable().visit(new PrintVisitor());
```

A fast reusable summer looks like this:

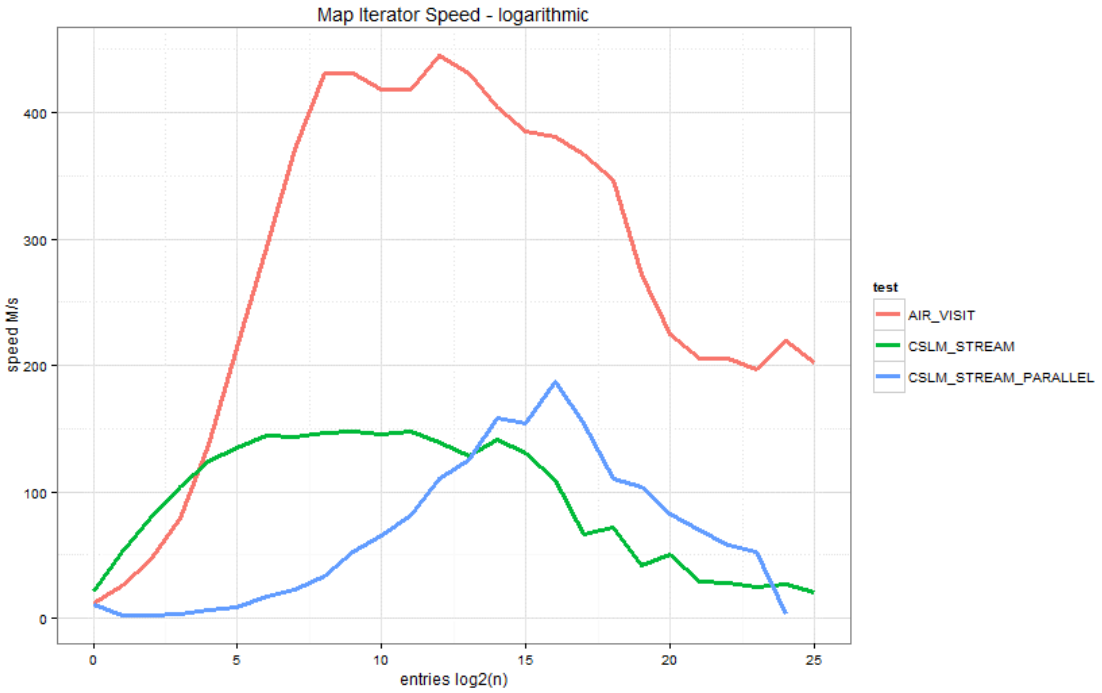
```
static class Summer extends MapVisitor<Object, Number> {
    // We could use double or float as well.
    long sum;
    // This is idiomatic.
    public long getSum(VisitableMap<?, Number> map) {
        sum = 0;
        map.getVisitable().visit(this);
        return sum;
    }

    @Override
    public void visit(Object key, Number value) {
        sum += value.longValue();
    }
}
```

Which is used this way:

```
System.out.println("sum=" + new Summer().getSum(map));
```

MapVisitor Versus Parallel Java Streams

The results show that the single-threaded Visitor is much faster than streams sequentially or in parallel. This is for simple counting, which shows the best case. Other slower operations such as summing or computing a maximum show a smaller difference, and in general as the operation takes longer, the performance advantage decreases. This reduction of relative performance is not experienced by the threaded technique described below.



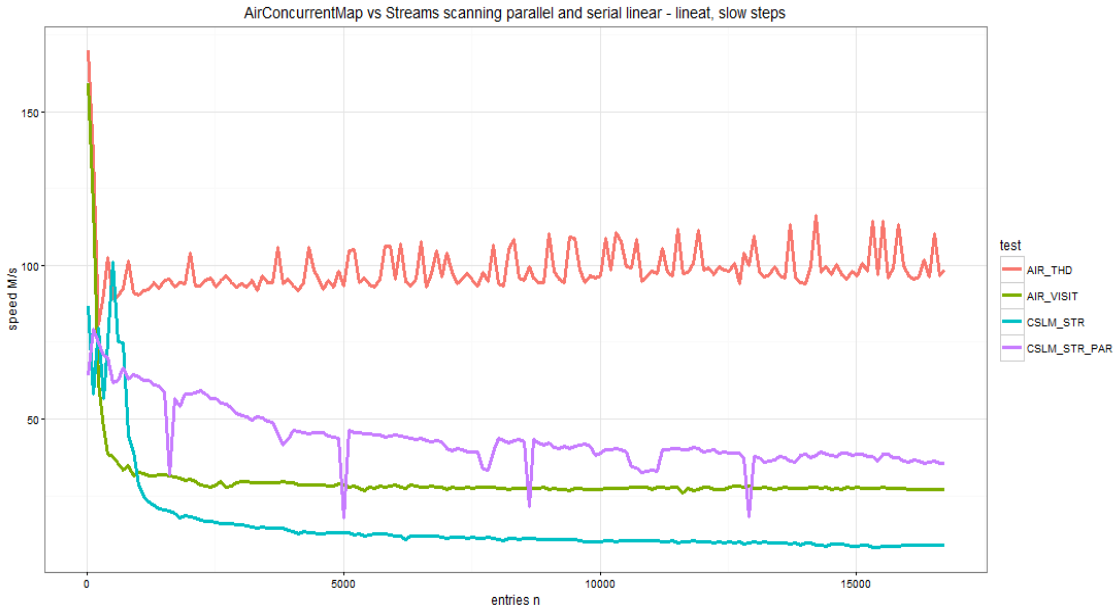
ThreadedMapVisitors

A further increase in scanning speed is available with the 'ThreadedMapVisitor subclass of MapVisitor. This is used interchangeably with a regular MapVisitor, except that it operates in parallel. There are two new threading-specific methods to implement:

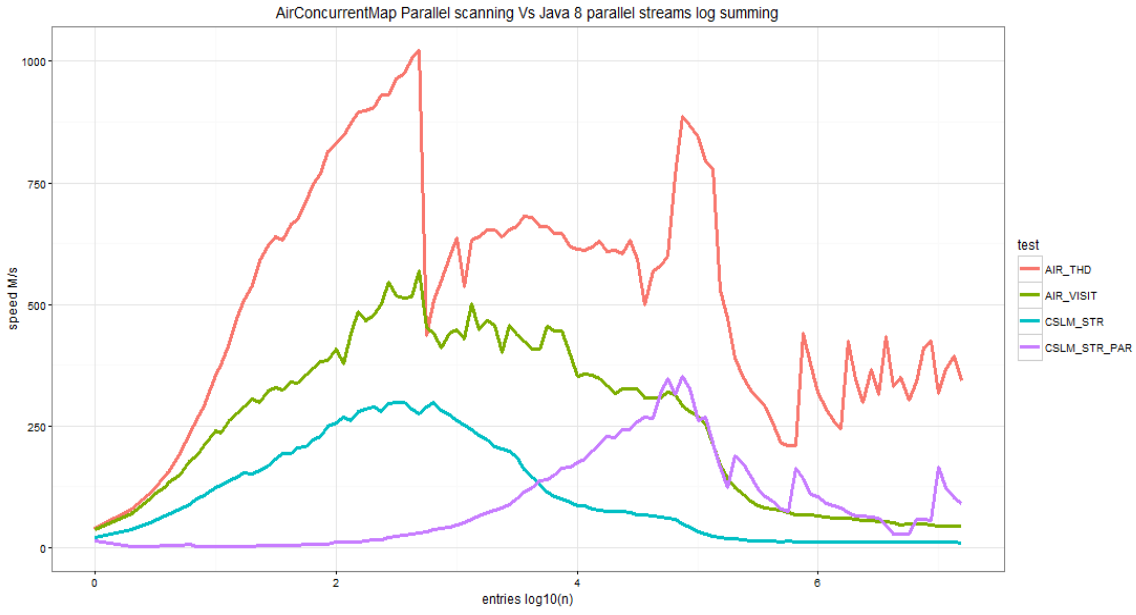
```
@Override
public ThreadedMapVisitor<Object, Number> split() {
    return new ThreadedSummer();
}

// The other threading-specific method
@Override
public void merge(ThreadedMapVisitor<Object, Number> visitor) {
    this.sum += ((ThreadedSummer)visitor).sum;
}
```

See the javadoc for more. As can be seen, the AirConcurrentMap threaded visitor technique is several times faster than the Java 8 streams system. The ThreadedMapVisitor is red, and the parallel streams is in purple. The operation is summing the values.

ThreadedMapVisitor Versus Java Parallel Streams



Here is the same information on a log10 basis. Again, ThreadedMapVisitor is at the top in red, then single-threaded MapVisitor in green. The Java 8 streams are at the bottom, and are 'peaky', so the programmer must guess at the production Map size and the system must show the guessed pattern. The parallel streams stall with small Maps, and the sequential streams stall with large Maps.



Background

Here we describe the basics of the various standard Map interfaces and their implementations.

Each standard Java Map interface comes with a standard implementation, and each has particular strengths and weaknesses, so the developer must choose an implementation carefully based on the features, performance, and efficiency that are needed. Often, the characteristics of the type of Map used by an application will be user-visible, due to the performance or memory efficiency of the Map, or due to the ability of the particular kind of Map to retrieve entries in ascending or descending key order and so on.

What is a 'Map'?

Maps are extremely common in Java applications. A Map is a collection of entries, and each entry in a Map is a 'key/value' pair. The key part of the entry identifies the entry uniquely in the Map, and it can be used to retrieve an entry directly using the get(Object key) method (we are ignoring generics). To add a key/value pair to the Map, the put(Object key, Object value) method is used, and it will first remove any previously existing key/value pair in the Map having the given key, and then put in the new key/value pair. For example, the key "MyKey" could be used to store the value "MyValue" using put("MyKey","MyValue"). Retrieving a single value is done with get("MyKey"). Both the key and the value are any Java Objects. Removing an entry can be done with remove(Object key). It is also possible to iterate quickly over the entries in a Map, or over the keys, or over the values. We will not try to cover all of the features of the Map interface.

What is a ConcurrentMap?

The ConcurrentMap is a special kind of Map that is used in Java applications where there are multiple threads putting, getting, and removing key/value entries at the same time. A ConcurrentMap is particularly important for performance when there are multiple cores available, because it allows multiple threads safely to access the data at the same time. The number of cores in all kinds of computers is increasing rapidly, at Moore's-law speed. As of this writing, many servers have dozens of cores, and even laptops can be quad-cores, with hyperthreading bringing the effective core count up to eight.

Taking advantage of this new kind of computing model is a challenge to developers, especially when data is to be shared between threads. The old route to allowing multiple threads to access and modify shared data is to wrap all Map methods using some form of locks - such as the Java synchronization mechanism – to prevent internal Map corruption.

Accidental corruptions from multi-thread access to single-threaded Maps have very complex unpredictable results. Locking, unfortunately, serializes access and prevents multiple core usage. Worse, locks can cause a dramatic drop in performance called 'convoying' as there are increasing numbers of threads contending for access to the shared data: the performance can drop to 10% or 1% of the single-threaded level or below. Such a scaling problem does not occur with a ConcurrentMap, as can be seen in the charts.

An example in which multi-core capability is important and natural is a web server, which may handle thousands of hits per second, with each hit being serviced by a thread temporarily allocated from a thread pool. These threads may need to share data so that users see a dynamic, up-to-date view of some kind of state. Each thread may need to deal with a significant amount of shared data on each page hit, so the Map may easily be a bottleneck for the entire server. There are countless other naturally occurring or 'embarrassingly' concurrent applications that need to share some data between threads. Also traditional single-threaded applications will need to become multi-threaded if they are to take advantage of the rapidly increasing core count.

The ConcurrentMap interface brings in some new special concurrency-related methods that allow atomic updates. One example is remove(Object key, Object value) , which removes an entry only if an entry with the given key exists and its value matches the given value. These improved methods increase performance by combining several operations into one, but more importantly, they do their operations atomically, avoiding the need to fall back to locking. For example, to get the same effect without the atomic replace() mentioned above, we would have to do: a lock acquisition, a get(), a test of the value, a possible put(), and then a lock release.

What is a NavigableMap?

A NavigableMap exposes the entries in key order, keeping the entries effectively sorted by key at all times. (The older SortedMap does also, but it has been superseded by NavigableMap in the standard Map implementations.) Thus iterators can scan the entries in ascending or descending key order, and optionally within a range of keys. A submap can be created that is a dynamic view of the original NavigableMap in ascending or descending key order and optionally over a restricted range of keys. The successor or predecessor entry with a given key can immediately be accessed, usually at the same speed as the simple get(). The availability of this feature can affect the entire structure of an application.

BoilerBay

BoilerBay provides the Java AirConcurrentMap implementation as a product to customers, as well as the technology behind AirConcurrentMap, which is patent-applied-for.

InfinityDB

For persisting data, use BoilerBay's InfinityDB, an extreme Map-like Java multi-core ordered key-value store. InfinityDB's performance is almost the same as AirConcurrentMap. The InfinityDB API is much more than key-value, however, providing ordered access and elegant data structuring capabilities. InfinityDB will at some time in the future acquire a ConcurrentNavigableMap interface, but its current API is capable of much more, and the two APIs will co-exist. InfinityDB provides dynamic paging of data from disk to allow databases to be transparently much larger than memory without the performance and space overhead of Object serialization, and there is no need to load and store the entire object graph in batches. InfinityDB is transactional, reliable, space efficient, fast, flexible, and simple. InfinityDB has thousands of active deployments in the field. InfinityDB uses the same patent-pending technology as AirConcurrentMap to achieve high performance. See:

boilerbay.com and infinitydb.com