

POSTED ON JUN 27, 2016 TO [ANDROID](#)



# Memory optimization for feeds on Android

UDI COHEN

Millions of people use Facebook on Android devices every day, scrolling through News Feed, Profile, Events, Pages, and Groups to interact with the people and information they care about. All of these different feed types are powered by a platform created by the Android Feed Platform team, so any optimization we make to the Feed platform could potentially improve performance across our app. We focus on scroll performance, as we want people to have a smooth experience when scrolling through their feeds.

To help us achieve that, we have several [automatic tools](#) that run performance tests on the Feed platform, across different scenarios and on different devices, measuring how our code performs in runtime, memory use, frame rate, and more. One of those tools, [Traceview](#), showed a relatively high number of calls to the `Long.valueOf()` function, which led to objects accumulating in memory and causing the app to stall. This post describes the problem, the potential solutions we weighed, and the set of optimizations we made to improve the Feed platform.

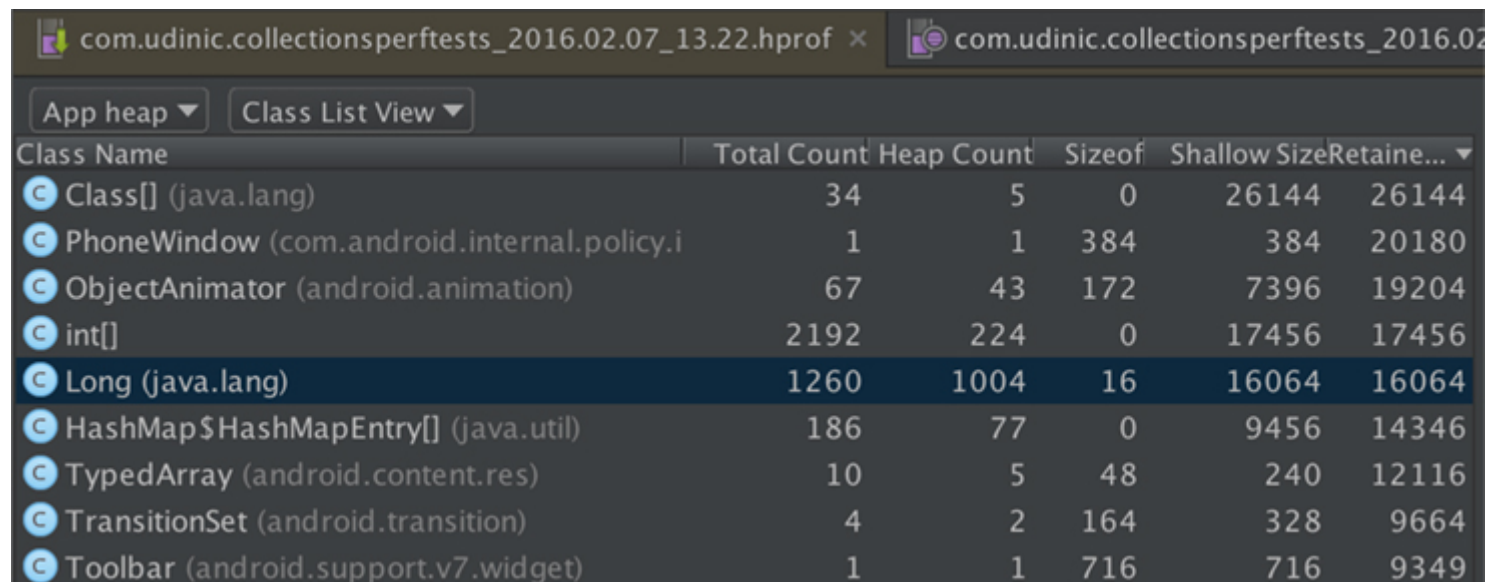
## The downside of convenience

After noticing the high number of calls to the `Long.valueOf()` function in one of our method profiling reports from Traceview, we ran further tests that confirmed that as we scrolled through News Feed, there were an unexpectedly high number of invocations for this method.

	Name	Invocation Count	Inclusive Time (µs)	Exclusive Time (µs)
▼ Thread: main			2,243,017 100.0%	
	java.lang.Object.<init>	30,678	86,681 3.9%	86,681 3.9%
	java.util.Collections.secondaryHash	22,508	382,004 17.0%	218,713 9.8%
	java.util.Collections.secondaryHash	22,508	77,273 3.4%	77,273 3.4%
	java.lang.Number.<init>	22,484	197,432 8.8%	133,490 6.0%
	java.lang.Long.valueOf	22,482	518,289 23.1%	172,426 7.7%
	java.lang.Long.<init>	22,482	345,863 15.4%	148,447 6.6%

When we looked at the stacktrace, we found that the function was not being called explicitly from Facebook's code but implicitly by code inserted by the compiler. This function was called when assigning a primitive **long** value where a **Long** object is expected. Java supports both Object and primitive representation of the simple types (e.g., integer, long character) and provides a way to seamlessly convert between them. This feature is called [autoboxing](#), because it “boxes” a primitive type into a corresponding Object type. While that's a convenient development feature, it creates new objects unbeknownst to the developer.

And these objects add up.



Class Name	Total Count	Heap Count	Sizeof	Shallow Size	Retained Size
Class[] (java.lang)	34	5	0	26144	26144
PhoneWindow (com.android.internal.policy.i	1	1	384	384	20180
ObjectAnimator (android.animation)	67	43	172	7396	19204
int[]	2192	224	0	17456	17456
<b>Long (java.lang)</b>	<b>1260</b>	<b>1004</b>	<b>16</b>	<b>16064</b>	<b>16064</b>
HashMap\$HashMapEntry[] (java.util)	186	77	0	9456	14346
TypedArray (android.content.res)	10	5	48	240	12116
TransitionSet (android.transition)	4	2	164	328	9664
Toolbar (android.support.v7.widget)	1	1	716	716	9349

In this heap dump taken from a sample app, **Long** objects have a noticeable presence; while each object is not big by itself, there are so many that they occupy a large portion of the app's memory in the heap. This is especially problematic for devices running the Dalvik runtime. Unlike ART, which is the newer Android runtime environment, Dalvik doesn't have a generational garbage collection, known to be more optimal for handling many small objects. As we scroll through News Feed and the number of objects grows, the garbage collection will cause the app to pause and sweep unused objects from the memory. The more objects that accumulate, the more frequently the garbage collector will have to pause the app, causing it to stutter and stall and making for a poor user experience.

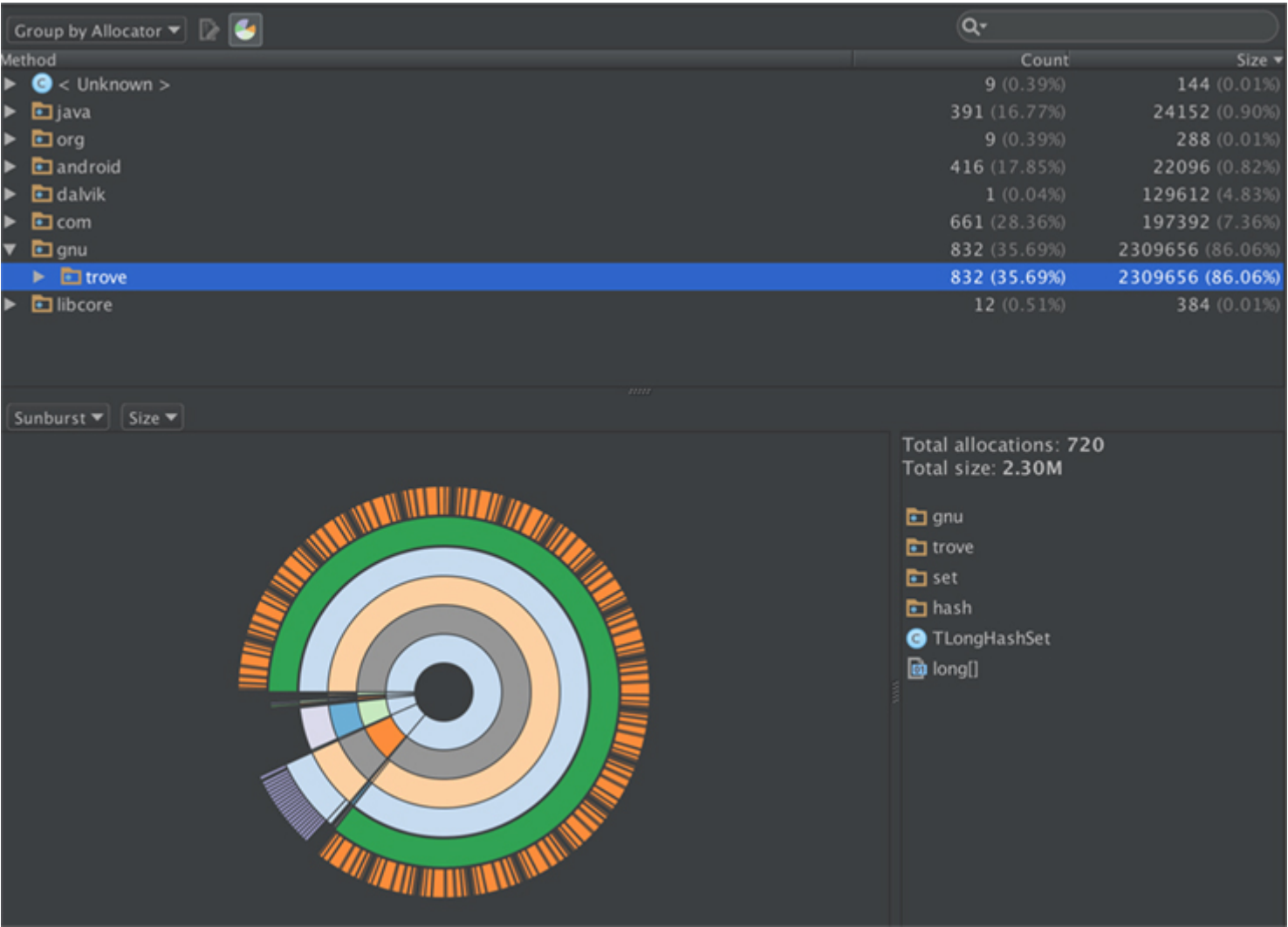
Luckily, tools such as Traceview and Allocation Tracker can help find where these calls are made. After reviewing the origins of these autoboxing occurrences, we found that the majority of them were done while inserting **long** values into a **HashSet<Long>** data structure. (We use this data structure to store hash values of News Feed stories, and later to check if a certain story's hash is already in the Set.) HashSet provides quick access to its items, an important feature allowing interaction with it as the user is scrolling through News Feed. Since the hash is calculated and stored into a primitive **long** variable, and our **HashSet** works only with objects, we get the inevitable autoboxing when calling `setStories.put(lStoryHash)`.

As a solution, a **Set** implementation for primitive types can be used, but that turned out not to be as straightforward as we expected.

## Existing solutions

There are a few existing Java libraries that provide a **Set** implementation for primitive types. Almost all of these libraries were created more than 10 years ago, when the only Java running on mobile devices was [J2ME](#). So, to determine viability, we needed to test them under Dalvik/ART and ensure they could perform well on more constrained mobile devices. We

created a small testing framework to help compare these libraries with the existing **HashSet** and with one another. The results showed that a few of these libraries had a faster runtime than **HashSet**, and with fewer **Long** objects, but they still internally allocated a lot of objects. As an example, **TLongHashSet**, part of the [Trove](#) library, allocated about 2 MB worth of objects when tested with 1,000 items:



Testing other libraries, such as [PCJ](#) and [Colt](#), showed similar results.

It was possible that the existing solutions didn't fit our needs. We considered whether we could instead create a new Set implementation and optimize it for Android. Looking inside Java's **HashSet**, there's a relatively simple implementation using a single **HashMap** to do the heavy lifting.

```
public class HashSet<E> extends AbstractSet<E>
implements Set<E>, ... {

    transient HashMap<E, HashSet<E>> backingMap;

    ...

    @Override public boolean add(E object) {
        return backingMap.put(object, this) == null;
    }

    @Override public boolean contains(Object object)
    {
        return backingMap.containsKey(object);
    }

    ...
}
```

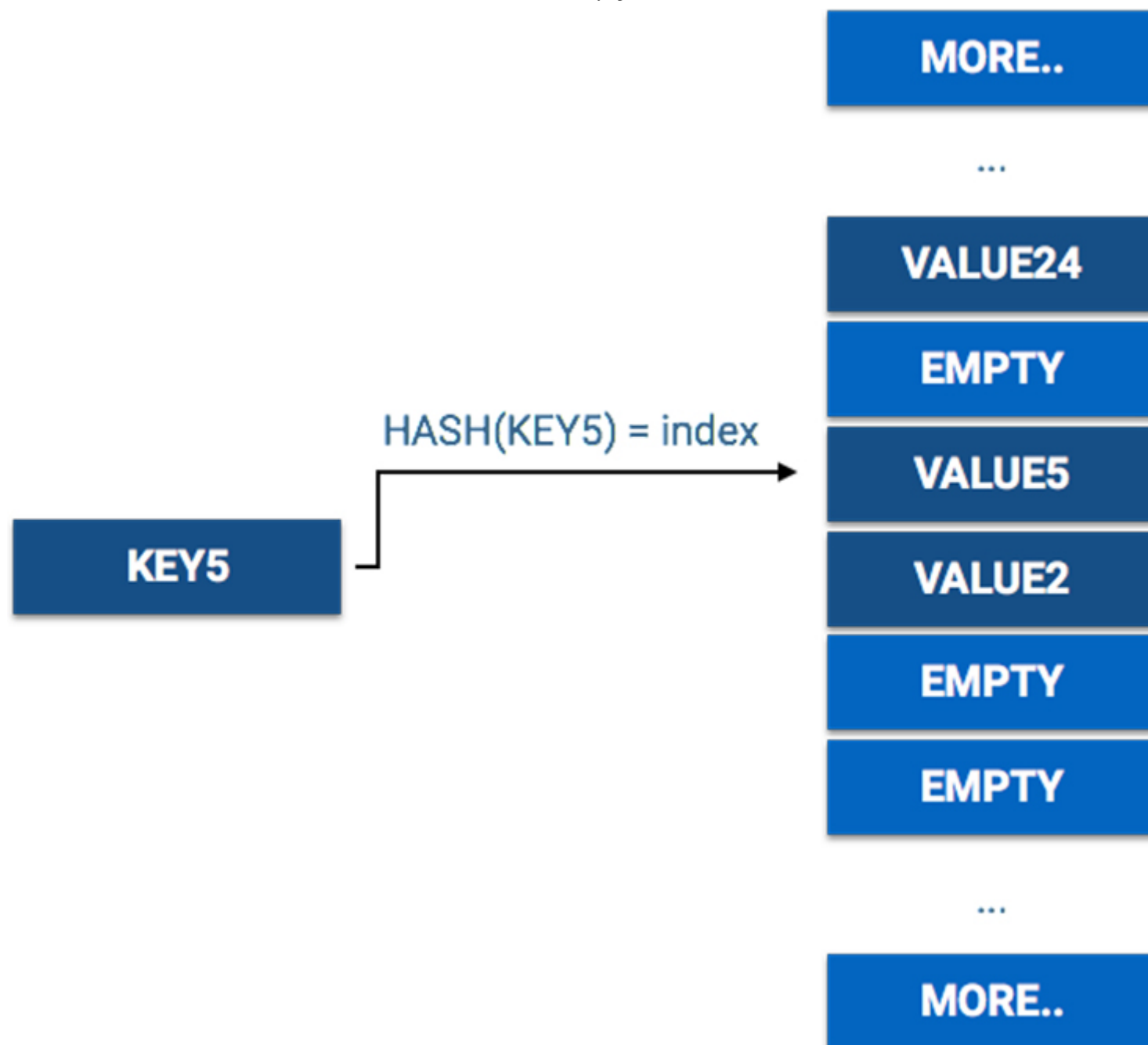
Adding a new item to the **HashSet** means adding it to the internal **HashMap** where the object is the key and the **HashSet**'s instance is the value. To check object membership, **HashSet** checks whether its internal **HashMap** contains the object as a key. An alternative to **HashSet** could be implemented using an Android-optimized map and the same principles.

## Introducing LongArraySet

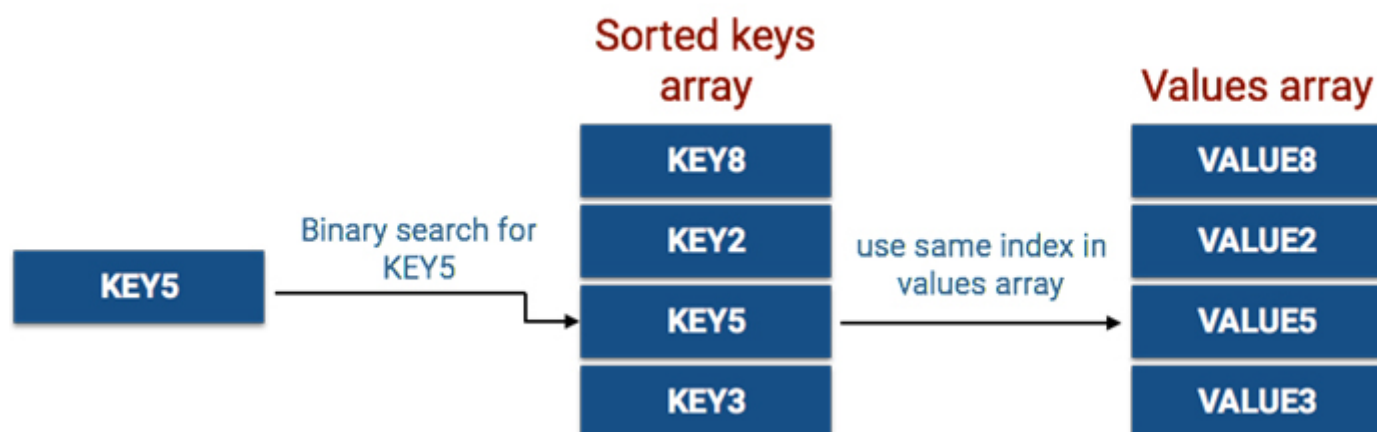
You may already be familiar with **LongSparseArray**, a class in the Android Support Library that acts as a map using a primitive **long** as a key. Example usage:

```
LongSparseArray<String> longSparseArray = new
LongSparseArray<>();
longSparseArray.put(3L, "Data");
String data = longSparseArray.get(3L); // the value
of data is "Data"
```

**LongSparseArray** works differently than **HashMap**, though. When calling **mapHashMap.get(KEY5)**, this is how the value is found in the **HashMap**:



When retrieving a value using a key on a **HashMap**, it's accessing the value in the array using a hash of the key as the index, a direct access in  $O(1)$ . Making the same call on a **LongSparseArray** looks like this:



**LongSparseArray** searches a sorted keys array for the key's value using a binary search, an operation with runtime of  $O(\log N)$ . The index of the key in the array is later used to find the value in the values array.

**HashMap** allocates one big array in an effort to avoid collisions, which are causing the search to be slower. **LongSparseArray** allocates two small arrays, making its memory footprint smaller. But to support its search algorithm, **LongSparseArray** needs to allocate its internal arrays in consecutive memory blocks. Adding more items will require allocating new arrays when there's no more space in the current ones. The way **LongSparseArray** works makes it less ideal when holding more than 1,000 items, where these differences have a more significant impact on performance. (You can learn more about **LongSparseArray** in the [official documentation](#) and by watching this [short video](#) by Google.)



Since the `LongSparseArray`'s keys are of a primitive `long` type, we can create a data structure with the same approach as `HashSet` but with a `LongSparseArray` as the internal map instead of `HashMap`.

And so `LongArraySet` was created.

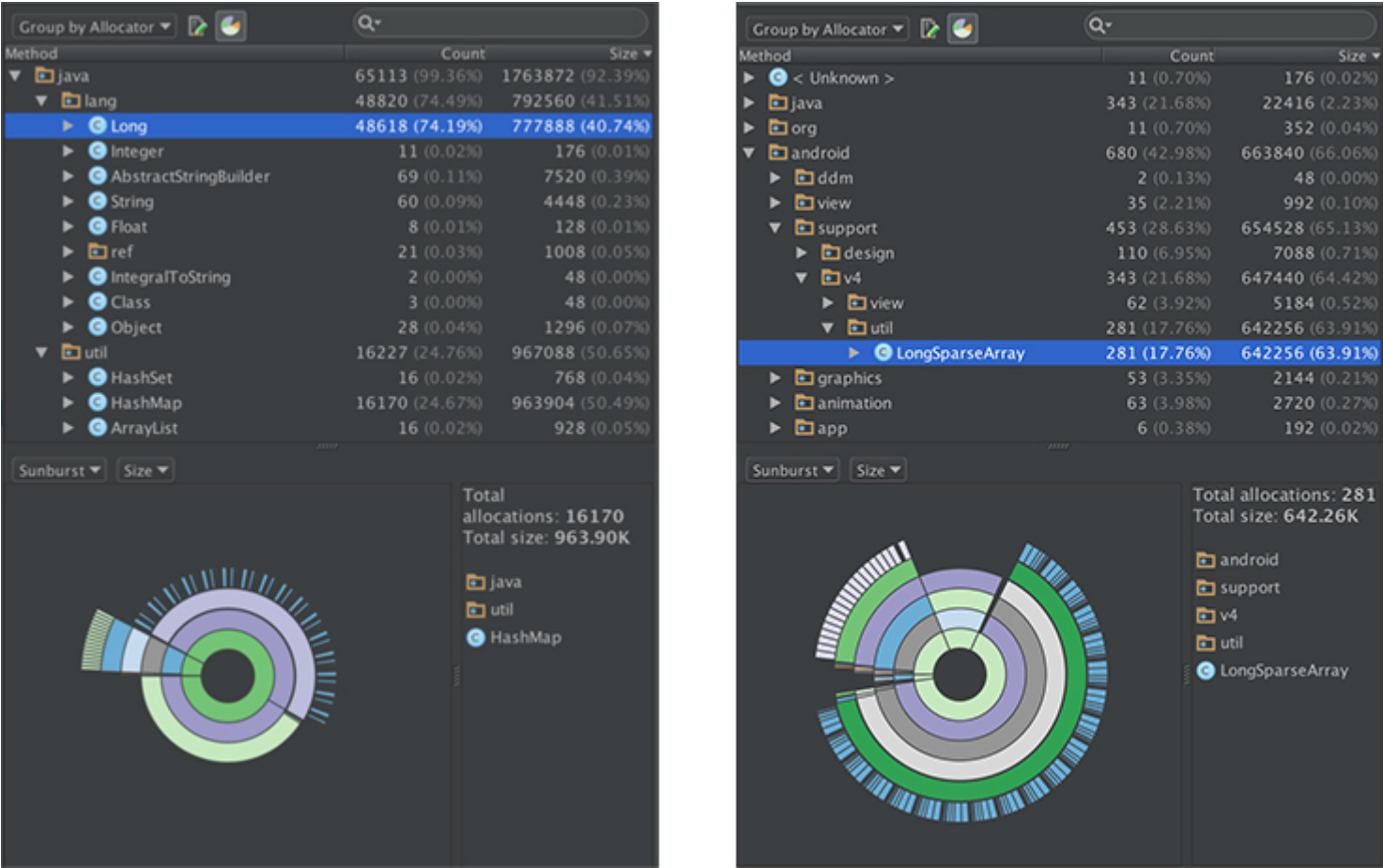
The new data structure looked promising, but the first rule of optimization is “always measure.” By using the same testing framework from earlier, we compared the new data structure with `HashSet`. Each data structure was tested by adding X number of items, checking the existence of each item, and later removing all of them. We ran tests using different numbers of items (X=10, X=100, X=1,000 ...) and averaged the time it took to complete each operation per item.

The runtime results (time shown is in nanoseconds):

	A	B	C	D
1	10 Items			
2		ADD	CONTAINS	REMOVE
3	HashSet	1519.79	690.64	707.03
4	LongArraySet	1056	320.32	342.98
5	change	-30.50%	-53.60%	-51.50%
6				
7	100 Items			
8		ADD	CONTAINS	REMOVE
9	HashSet	3102.66	570.63	606.01
10	LongArraySet	802.32	244.77	336.95
11	change	-74.10%	-57.10%	-44.40%
12				
13	1000 Items			
14		ADD	CONTAINS	REMOVE
15	HashSet	1267.6	788.48	698.38
16	LongArraySet	2116.23	332.84	404.65
17	change	66.90%	-57.80%	-42.10%
18				
19	10000 Items			
20		ADD	CONTAINS	REMOVE
21	HashSet	1042.11	650.66	748.46
22	LongArraySet	8966.69	376.68	428.94
23	change	760.40%	-42.10%	-42.70%

We saw a runtime improvement for the `contains` and `remove` methods using the new data structure. Also, as the number of items increased in the array set, it took more time to add new items. That's consistent with what we already knew about `LongSparseArray` — it doesn't perform as well as `HashMap` when the number of items is more than 1,000. In our use cases, we're dealing with only hundreds of items, so this is a trade-off we're willing to make.

We also saw a big improvement related to memory. In reviewing the heap dumps and Allocation Tracker reports, we noticed a decrease in object allocations. Here's a side-by-side allocation report for the `HashSet` and `LongArraySet` implementations, when adding 1,000 items for 20 iterations:



In addition to avoiding all the **Long** object allocations, **LongSparseArray** was more memory-efficient internally, with about 30 percent fewer allocations in this scenario.

## Conclusion

By understanding how other data structures work, we were able to create a more optimized data structure for our needs. The less the garbage collector has to work, the lower the likelihood of dropped frames. Using the new **LongArraySet** class, and a similar **IntArraySet** for the primitive **int** data type, we were able to cut down a significant number of allocations in our entire app.

This case study demonstrates the importance of measuring our options to ensure that we were introducing an improvement. We also accepted that while this solution is not perfect for all use cases, since this implementation is slower for very large data sets, it was an acceptable trade-off for us to optimize our code.

You can find the source code for the two data structures [here](#). We are excited to continue working on challenges and optimizing our Feed platform and to share our solutions with the community.

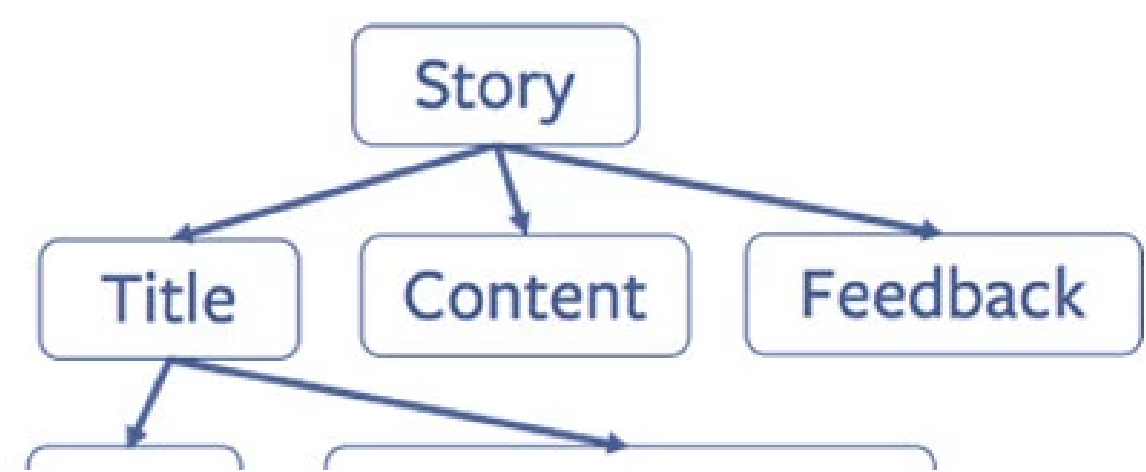
TAGS: [MOBILE](#) [NEWS FEED](#) [PERFORMANCE](#) [TESTING](#)

Like

Share

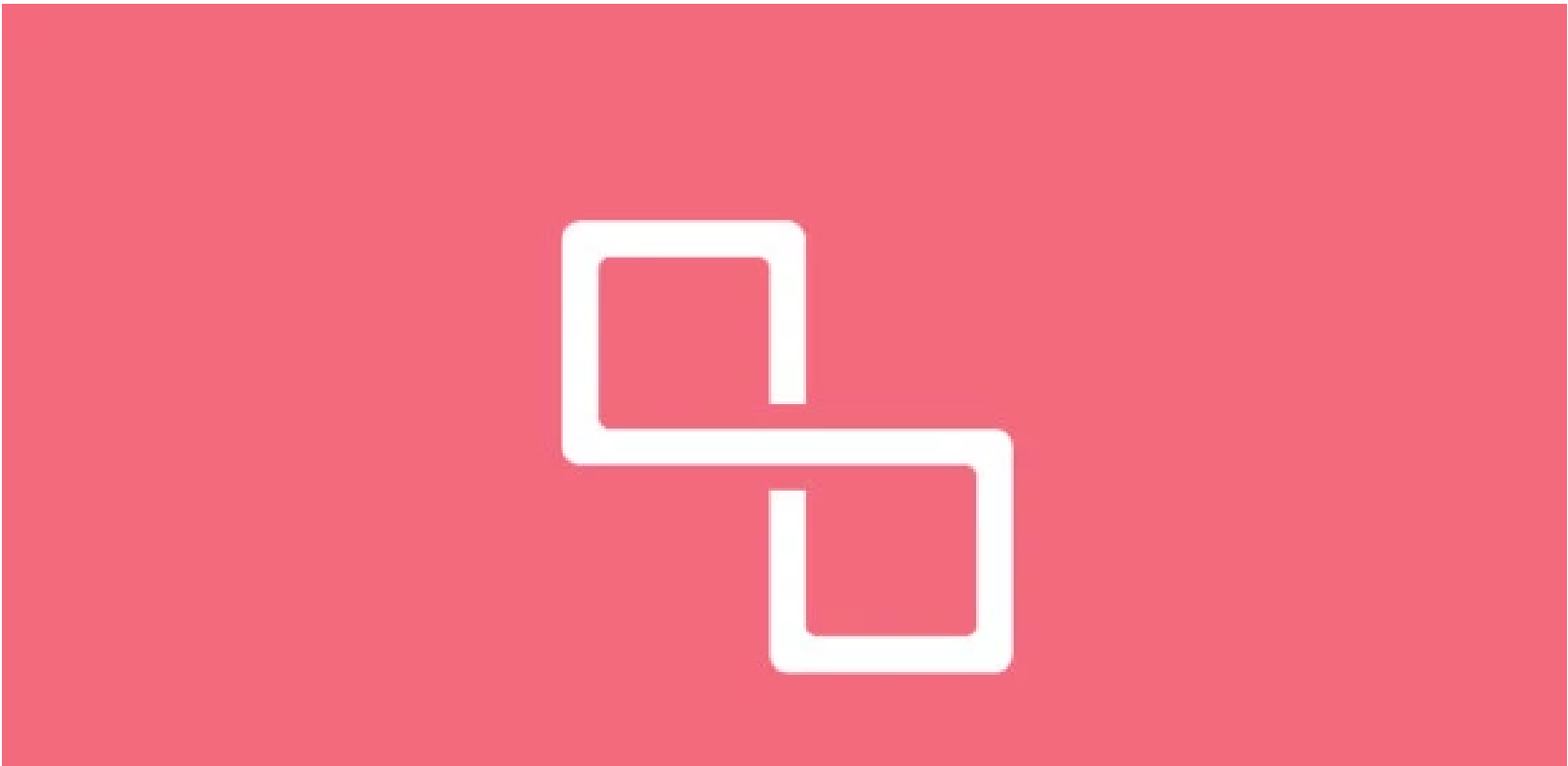
 722 people like this. Be the first of your friends.

Related Posts



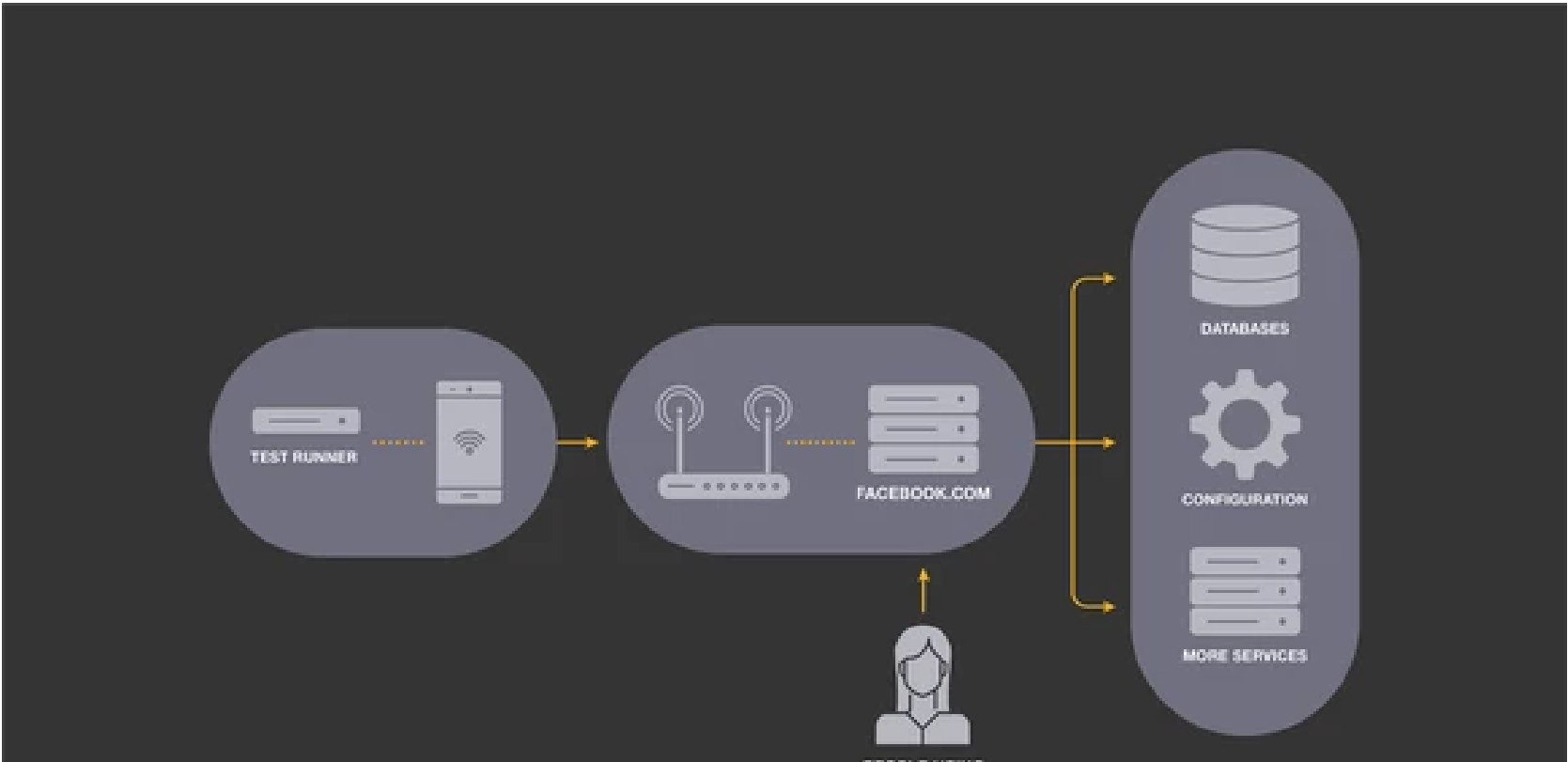
09.26.2017

[Multithreaded rendering on Android with Litho and Infer](#)



04.18.2017

[Open-sourcing Litho, a declarative UI framework for Android](#)



10.19.2018

[MobileLab: Highly accurate testing to prevent mobile performance regressions](#)



## Related Positions

[Partner Engineer, Android](#)

[MENLO PARK, US](#)

[Instagram - Software Engineer, Android \(Live Infrastructure\)](#)

[SAN FRANCISCO, US](#)

[Software Engineer, Mobile, Android NDK](#)

[MENLO PARK, US](#)

[Android UI Engineer](#)

[MENLO PARK, US](#)

[Android Software Engineer](#)

[SEATTLE, US](#)

See All Jobs

## Join Our Engineering Community

Available  
Positions

[Partner Engineer,  
Android](#)  
[MENLO PARK, US](#)  
[Instagram - Software  
Engineer, Android](#)

Stay  
Connected



Facebook  
Engineering

Like

Open  
Source

Facebook believes in building community through open source technology. Explore our latest projects in Artificial Intelligence,

([Live Infrastructure](#)),  
[SAN FRANCISCO, US](#)  
[Software Engineer, Mobile, Android NDK](#)  
[MENLO PARK, US](#)  
[Android UI Engineer](#)  
[MENLO PARK, US](#)  
[Android Software Engineer](#)  
[SEATTLE, US](#)

See All Jobs



@fbOpenSource

Follow



Facebook Research

Like



Facebook Developers

Like



RSS

Subscribe

Data Infrastructure,  
Development Tools,  
Front End,  
Languages,  
Platforms, Security,  
Virtual Reality, and  
more.



ANDROID



iOS



WEB



BACKEND



HARDWARE

Learn More