

Flink Frequent Item Detection

Aaryan Mohindru, Eric Fournier, Neel Joshi, Yan Mazheika

Motivation

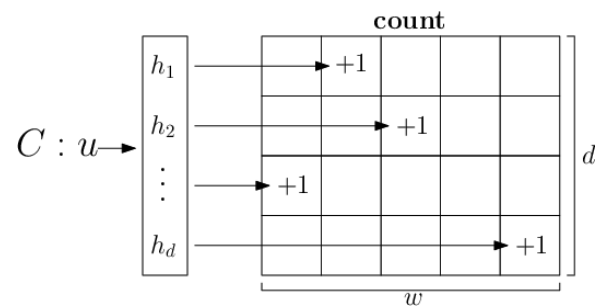
Online storefronts such as Amazon conduct millions of transactions every day for a wide range of products. Analyzing trends in this data can require intensive computation. We implemented a space-efficient distributed system to determine which items and categories are trending. The system empowers businesses and customers alike to make informed decisions about products.

Design Description

The [Amazon Sales Dataset](#) contains ~1,400 rows of item purchases from Amazon, including product names and categories. To construct an unbounded stream from this data, we created a rate-limited data source which generates random purchases from a distribution over item categories. This distribution can be tuned by specifying weights for each category in [resources/weights.yaml](#). If the total weight of these specified categories is less than 100%, the remaining weight is uniformly distributed over unspecified categories. The dataset itself is stored as a CSV file, and the logic to parse it leverages [OpenCSV](#) and resides in [stream/PurchaseGenerator](#).

To estimate frequent items in the stream, we implemented the [count-min sketch](#) (CMS) data structure in [cms2D/Sketch](#). CMS uses sublinear space, which makes it perfect when handling large data streams. Each [Sketch](#) encapsulates a 2D array of estimates. Each row in the array is associated with a [MurmurHash3.hash32x86](#) hash function. When an item arrives, its category is passed through each hash function, and the hashes are used to increment the counters of

the corresponding rows. Notice counters never underestimate with this scheme, so the minimum counter is the best estimate for a given category's frequency.



To achieve a distributed CMS, incoming items are randomly distributed across `NUM_CORES` workers using a `RandomKeySelector`. Workers run the `WindowCMS` process, which creates and updates a local `Sketch` object. After a set time interval, the workers submit their local sketches to a global coordinator running the `Merger` process. Sketches are merged by simply adding corresponding entries.

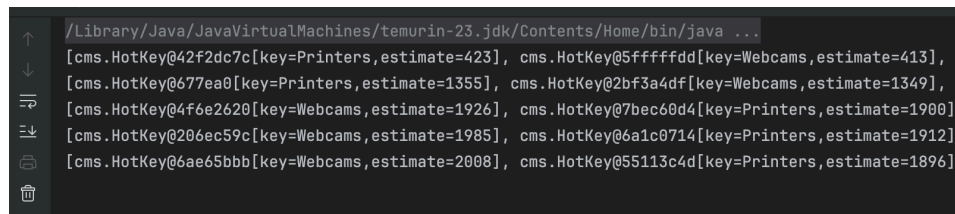
Assuming the number of categories can be arbitrarily large, it would be costly to extract estimates for every category from a merged CMS. Instead, our solution has each worker maintain a size-limited `TreeSet`, which stores the top `MAX_HOT_KEYS` categories with the largest local estimates in `HotKey` objects. `TreeSet` was chosen since it supports item insertion and deletion in logarithmic time. Workers emit these categories along with their sketches. The coordinator then uses this reduced set on the merged CMS to determine frequent categories, significantly improving performance.

| <div><div></div><div>C</div></div> HotKey |
|---|
| <div><div><div>key: String</div><div>estimate: int</div></div></div> |
| <div><div><div>HotKey(key: String, estimate: int)</div><div>getKey(): String</div><div>setKey(key: String): void</div><div>getEstimate(): int</div><div>setEstimate(estimate: int): void</div><div>compareTo(other: HotKey): int</div><div>toString(): String</div></div></div> |

This works because purchases are distributed randomly across the workers, which preserves relative frequencies in expectation. For instance, if there are 5 workers and a stream of 75 printers and 25 webcams, each worker is expected to receive 15 printers and 5 webcams. Thus, if printers are globally popular, they're expected to be locally popular and likely represented in a **HotKey** emitted by a worker

Testing

We tested our program by creating custom category distributions and observing the output. Below, we set “Webcams” and “Printers” to 25% each, meaning that of the ~200 categories, “Webcams” and “Printers” should appear as trending:

A terminal window with a dark background and light text. The prompt is `/Library/Java/JavaVirtualMachines/temurin-23.jdk/Contents/Home/bin/java ...`. The output consists of six lines of JSON-like arrays, each containing two objects. Each object has a `key` (either "Printers" or "Webcams") and an `estimate` value. The estimates for "Printers" are 423, 1355, 1926, 1985, 1912, and 1896. The estimates for "Webcams" are 413, 1349, 1900, 1908, 2008, and 1896. The objects are separated by commas, and the arrays are enclosed in square brackets.

```
/Library/Java/JavaVirtualMachines/temurin-23.jdk/Contents/Home/bin/java ...  
[cms.HotKey@42f2dc7c[key=Printers,estimate=423], cms.HotKey@5ffffdd[key=Webcams,estimate=413],  
[cms.HotKey@677ea0[key=Printers,estimate=1355], cms.HotKey@2bf3a4df[key=Webcams,estimate=1349],  
[cms.HotKey@4f6e2620[key=Webcams,estimate=1926], cms.HotKey@7bec60d4[key=Printers,estimate=1900]  
[cms.HotKey@206ec59c[key=Webcams,estimate=1985], cms.HotKey@6a1c0714[key=Printers,estimate=1912]  
[cms.HotKey@6ae65bbb[key=Webcams,estimate=2008], cms.HotKey@55113c4d[key=Printers,estimate=1896]
```

There are other three test distributions included as yaml files in **resources**. To run a test, simply change the **WEIGHTS_FILE** variable in **distribution/CustomDistribution**.

1. **test1.yaml**: This file doesn't specify any categories, meaning all categories are weighted equally.
2. **test2.yaml**: This file sets “Printers” to 99% and “Webcams” to 1%.
3. **test3.yaml**: This file sets “Printers” to 60%, “Webcams” to 20%, and “Basic Cases” to 20%.

For the above tests, if the program works properly, the relative frequencies should be reflected in the estimates for the categories reported by the program. The logs included show this is exactly the case:

```

C:\Program Files\Java\jdk-23\bin\java.exe ...
19:37:24,274 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - No tokens obtained so skipping notifications
19:37:24,547 WARN org.apache.flink.runtime.webmonitor.WebMonitorUtils [] - Log file environment variable 'log.file' is not set.
19:37:24,547 WARN org.apache.flink.runtime.webmonitor.WebMonitorUtils [] - JobManager log files are unavailable in the web dashboard. Log file location not found in environment variable 'log.file' or configuration
19:37:24,684 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - No tokens obtained so skipping notifications
19:37:24,684 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - Tokens update task not started because either no tokens obtained or none of the tokens specified its renewal date
[[key=Smartphones,estimate=241], [key=Digital Kitchen Scales,estimate=213], [key=Pens,estimate=202], [key=Sandwich Makers,estimate=191], [key=Digital Scales,estimate=188], [key=HEPA Air Purifiers,estimate=186], [key=Earbuds,
[[key=Split-System Air Conditioners,estimate=486], [key=Digital Kitchen Scales,estimate=486], [key=Pens,estimate=452], [key=Hard Disk Bags,estimate=424], [key=Basic Cases,estimate=415], [key=Laundry Baskets,estimate=402], [
[[key=Smartphones,estimate=532], [key=Digital Kitchen Scales,estimate=512], [key=Sandwich Makers,estimate=469], [key=Rotl Makers,estimate=469], [key=Hard Disk Bags,estimate=469], [key=Macro & Ringlight Flashes,estimate=466],

```

```

C:\Program Files\Java\jdk-23\bin\java.exe ...
19:38:05,498 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - No tokens obtained so skipping notifications
19:38:05,971 WARN org.apache.flink.runtime.webmonitor.WebMonitorUtils [] - Log file environment variable 'log.file' is not set.
19:38:05,971 WARN org.apache.flink.runtime.webmonitor.WebMonitorUtils [] - JobManager log files are unavailable in the web dashboard. Log file location not found in environment variable 'log.file' or configuration
19:38:06,109 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - No tokens obtained so skipping notifications
19:38:06,110 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - Tokens update task not started because either no tokens obtained or none of the tokens specified its renewal date
[[key=Printers,estimate=3639], [key=Webcams,estimate=377]]
[[key=Printers,estimate=4631], [key=Webcams,estimate=391]]

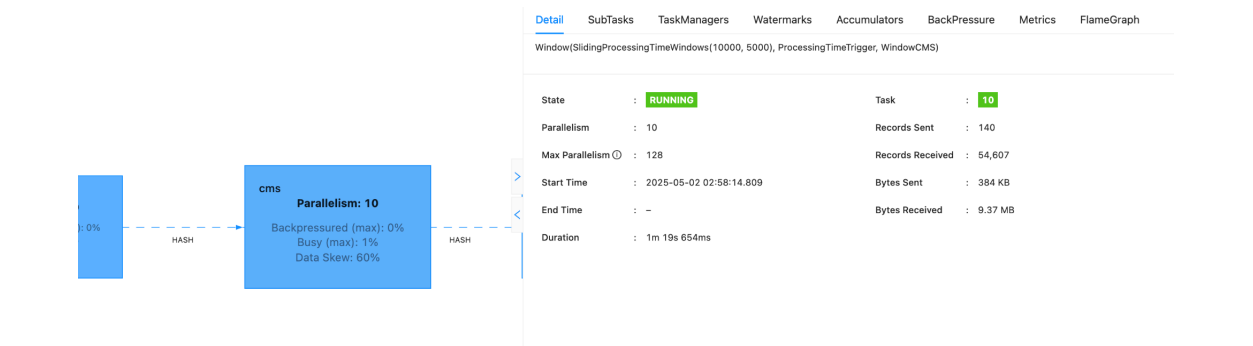
```

```

C:\Program Files\Java\jdk-23\bin\java.exe ...
19:38:50,221 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - No tokens obtained so skipping notifications
19:38:50,495 WARN org.apache.flink.runtime.webmonitor.WebMonitorUtils [] - Log file environment variable 'log.file' is not set.
19:38:50,495 WARN org.apache.flink.runtime.webmonitor.WebMonitorUtils [] - JobManager log files are unavailable in the web dashboard. Log file location not found in environment variable 'log.file' or configuration
19:38:50,631 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - No tokens obtained so skipping notifications
19:38:50,631 WARN org.apache.flink.runtime.security.token.DefaultDelegationTokenManager [] - Tokens update task not started because either no tokens obtained or none of the tokens specified its renewal date
[[key=Printers,estimate=989], [key=Basic Cases,estimate=295], [key=Webcams,estimate=283]]
[[key=Printers,estimate=2350], [key=Basic Cases,estimate=750], [key=Webcams,estimate=748]]
[[key=Printers,estimate=2768], [key=Webcams,estimate=928], [key=Basic Cases,estimate=914]]

```

We also used our Flink dashboard to verify the program’s finer-grained qualities, such as load balancing between subtasks. To do this, we compared input/output rates of the subtasks for all states in the pipeline.

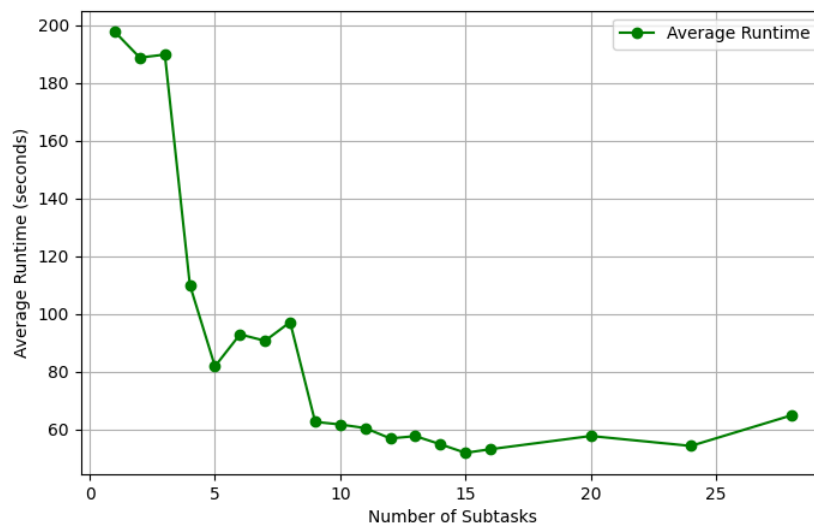


By running the sanity tests and looking at the dashboard, we were able to confirm that our CMS pipeline was accurately identifying popular categories and doing so in an evenly distributed manner across all subtasks.

Experimental Results

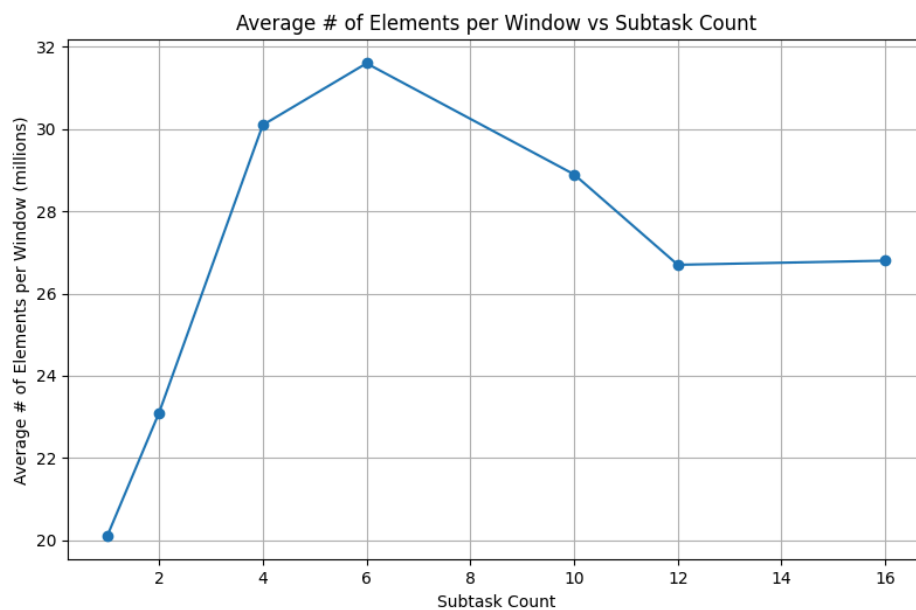
We conducted several performance experiments on our Count Min Sketch processing pipeline. We ran each experiment three times and took the average metric. The system had 16 available cores and 8GB of allocated JVM heap memory throughout all of our testing.

The first was unrestricting the throughput rate of input elements and examining how the system performs under load. We allowed 100 million purchases to flow unrestricted and measured the runtime while varying subtask count. We found that more subtasks helped processing time with diminishing returns until multi-processing started diminishing capability at 16 cores, the number of physical cores available.



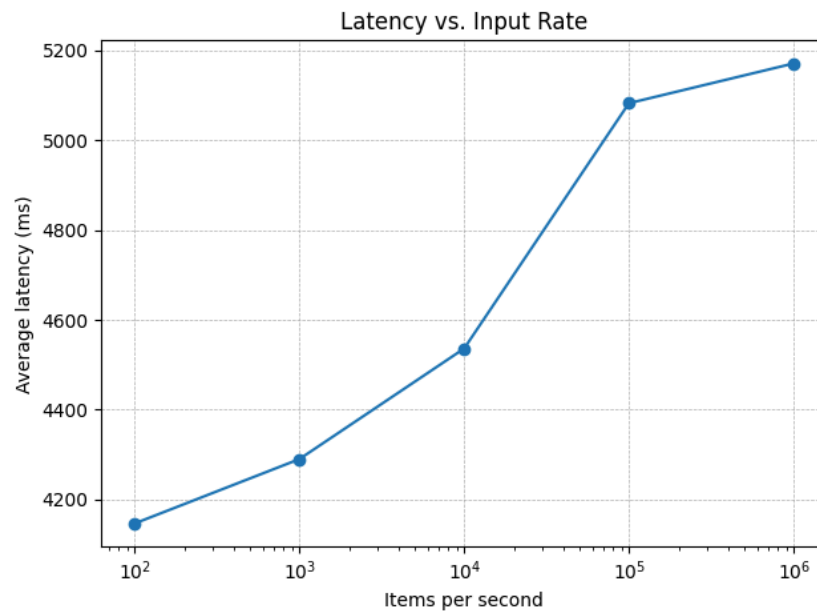
Our results suggest sharply diminished returns after 9 subtasks, most likely caused exhaustion of allocated memory due to increased stream buffering. While examining a system monitor, we observed heap usage close to 8GB while CPU-usage in percent remained stable at around 30% for subtask counts at or above 6. Additionally, performance degradation may stem from uneven load balancing, amplified by random assignment of elements to subtasks and by time-scheduling when subtasks counts are above available cores.

We also wanted to see the maximum throughput of elements a window could reliably ingest and analyze. We examined the throughput of one isolated 10-second window without observing the effects of buffering on subsequent windows. This measure isolates the effects from earlier stream build-up and highlights the increase in performance of having additional subtasks. The graph below displays the total number of elements streamed across all available subtasks per one time window.



This graph highlights an interesting relationship between intake rate and performance. We found that despite having a lower stream input rate, performance was better at higher thread counts (>10). This is due to each subtask processing fewer elements and potentially being unbalanced. The lower input rate for higher thread counts can also be attributed to a larger share of memory being reserved for CMS-arrays/other Java objects and the fact that memory is a bottleneck after 4-6 cores.

Lastly, we tested latency against various stream input rates on a single core. Latency was measured as the time on `Purchase` object creation to be fully processed in the `WindowCMS` routine. The graph above shows the results.



Overall, the main factor in reducing scalability is memory. We found that with more subtasks, we can increase performance at the cost of being able to buffer less elements per window. Furthermore, a larger input of elements causes latency to decrease. We expect that with more allocated memory, the runtime to resolve a bottleneck would continue to decrease.

Possible Improvements

<<ADD MORE DETAILED EXPLANATIONS BELOW>>

There are four main areas that could be improved:

1. Add more interesting distributions: Exponential, geometric, etc.
2. Optimize memory further (lot of objects created and used)
3. Customize checkpoints

4. Automatic-scaling of nodes