Ranking Highscores

Evaluation of a dynamic Bucket with Global Query algorithm

Carl-Evert Kangas
Abstract

The task of ranking highscores in a computer game may sound like a trivial task. It turns out it is not, because the naive solution have a time complexity not suitable for online applications in terms of response time and running cost.

An overview of a few approaches to ranking is presented: how an N-ary tree could be used to do ranking and how to do linear approximation. Two ways of obtaining a model for doing linear approximation are demonstrated, a method called *Buckets with Global Query* is described and a method based on *Frugal Streaming* is elaborated on.

Finally, a variant of the *Buckets with Global Query* algorithm where the buckets are adjusted continuously according to the changes in the distribution of highscores is evaluated. The dynamic variant of the algorithm performs well in terms of accuracy for at least 100 000 highscore updates but have no significant gains in reduced CPU-time.

Att ranka rekord

En utvärdering av en dynamisk implementation av algoritmen *Buckets with Global Query*

Sammanfattning

Den naiva lösningen till att ranka ett rekord i ett spel är att räkna antalet rekord som är bättre än det rekord man har för handen. Lösningen skalar inte då den har en tidskomplexitet på $O(n)$ och kan därför inte användas i ett onlinespel då responstiden skulle bli för lång och kostnaden för hög.


Slutligen utvärderas en variant av *Buckets with Global Query* där modellen som approximationerna görs ifrån kontinuerligt uppdateras för att motsvara den verkliga distributionen av alla rekord. Den dynamiska varianten av algoritmen har god precision men är inte nämnvärt effektivare än den ursprungliga.
Acknowledgements

I would like to thank my supervisor Suna Bensch for all support and feedback during the writing of this thesis.

Special thanks to my partner Peter Sundqvist for moral as well as technical support during this project.
Contents

1 Introduction 1
  1.1 Ranking, a definition 1
  1.2 Problem description, outline and methods 2

2 Ranking algorithms 3
  2.1 Rank by counting 3
  2.2 Tree based approach 3
  2.3 Rank by approximation with linear interpolation 4
    2.3.1 Buckets with Global Query 6
    2.3.2 Online approaches, streaming algorithms 9

3 Improving the Bucket with Global Query-algorithm 11
  3.1 Hypothesis 11
  3.2 Data 12
  3.3 The experiment 13
  3.4 What and how to measure 14
  3.5 Limitations 14

4 Results 15
  4.1 Relative error 15
  4.2 Execution time 16

5 Discussion and future directions 19

References 21
1 Introduction

Ranking can informally be defined as “a relationship between a set of items such that, for any two items, the first is either ‘ranked higher than’, ‘ranked lower than’ or ‘ranked equal to’ the second”\(^1\). Usually a rank number is assigned to the items such that the highest number receive rank number one.

Ranking in computer games is mainly done to reward skilled or lucky players. Introducing a dimension of competition makes the game more exciting and hopefully – from the game makers point of view – more profitable. This can be done by creating a leaderboard that shows the highest scores or by giving some measure of improvement to the player, eg. number of ranks advanced by the last game. Other reasons for ranking skills in computer games include gambling and matching equally skilled players for challenges.

As a player of a computer game one wants to know if one are making progress in your gaming. However, games that can be considered having even a moderate success on today’s marketplaces have at least hundreds of thousands of players not knowing each other. Therefore, cheating by approximating ranks works perfectly fine most of the time. Also, approximated ranks come close to the real rank as we shall see. In some cases though an exact rank needs to be calculated. Obviously it would be embarrassing if two players were estimated to have rank number one. In general, higher ranks calls for greater precision.

1.1 Ranking, a definition

When ranking is mentioned in scientific writings it is usually in the sense of assigning a weight to an item in a set from a function of properties of that item. A search on a price comparison site for “red car” would assign a high weight to items that are cars and also happens to be red. Eventually a list of red cars will be presented.

When the word *ranking* appears in the rest of this thesis it means the procedure of getting the position for an item with a weight in a list sorted on the weight of the item – not how that specific weight is decided upon. Or to be precise; the procedure of getting the position for a user with a highscore from a large number of highscores in a database, indexed on the highscore.

It is unfortunate that there is no commonly used, distinct word for *ranking*. However, it should be clear from the context which definition is referred.

\(^1\) https://en.wikipedia.org/wiki/Ranking
1.2 Problem description, outline and methods

The overall goal is to investigate how to get the rank, exact or an approximation, for a new highscore in the context of an online computer game. Doing ranking of highscores by counting the number of highscores better than a specific score is not a feasible option. A tree-based approach to counting seems promising but have a number of drawbacks that will be discussed in Section 2.2. Two methods based on linear interpolation will be presented: *Buckets with Global Query* and a method relying on *Frugal streaming* (Sections 2.3.1 and 2.3.2) will be discussed, the latter one for demonstrative purposes. Finally, a variant of the *Buckets with Global Query*-algorithm with dynamic buckets will be presented and empirically evaluated in Chapters 3, 4 and 5.

The questions to be researched in this thesis are

1. How to get a rank for a score efficiently and in near real time when the set of scores is large?
2. What is the performance characteristics of Bucket with Global Query algorithm when adapting the buckets to changes in the underlying highscore distribution?

To attend to the first question, a review of the ranking approaches mentioned above will be provided in Chapter 2, starting by a naive approach derived from the definition of ranking, ending with a draft for a streaming and approximating approach. The algorithms presented in Chapter 2 do certainly not represent a complete listing of all possible ways of solving the problem stated but will nevertheless provide some insight in the area.

To answer the second question on the performance characteristics of the Buckets with Global Query with dynamic buckets, a system for evaluating the performance in terms of efficiency and precision is built. The precise method for this evaluation is left for Chapter 3.
2 Ranking algorithms

The main purpose of the algorithms discussed below in this section is to answer the question 
*Given a score, what is the rank of a user or item with this score?*. The opposite question 
– *Given a rank, what is the score for this rank?* is very similar but not addressed here even 
though most approaches to the ranking problem indirectly answers that question too.

Ranking algorithms can coarsely be divided in two categories: *exact* and *approximating* 
algorithms. The exact algorithms deal with the ranking problem by sorting and counting the 
number of users having a better score than the new score. If it is not necessary to obtain 
an exact rank or other factors such as speed or cost have to be considered an approximating 
algorithm can be a good compromise.

Approximating algorithms estimate the rank by interpolation within a segment of ranks. 
Suitable segments can be found with an *offline-method* such as scanning all scores period-
ically, keeping track of scores at the segments boundaries (See Section 2.3.1) or with an 
*online-method* with a streaming algorithm that maintains a model for estimating ranks by 
continuously updating a number of statistical measures.

2.1 Rank by counting

The naive approach to ranking a highscore is to count the number of highscores better 
than the one at hand. This way of getting the rank can be done with a simple query in 
SQL: `SELECT count(id) FROM Highscores WHERE Score > TheScore` or by increas-
ing a counter while iterating through an ordered set of highscores in case a non-relational 
database is used. In any case the items should be indexed on the highscore-field.

Obtaining rank for a score by this method implies scanning through all items having a score 
higher than the one you want to obtain the rank for. The time complexity of this approach 
is obviously $O(n)$ making it unusable for online applications in terms of cost and response 
time.

2.2 Tree based approach

One way to accomplish counting more efficiently is by storing the count of each score in 
an N-ary tree. $N$ is the maximum number of childrens a node can have. Each node defines 
$N$ score ranges or $N$ single scores along with their associated count. The leaves represents 
individual items from the domain of possible scores. The rank for a score is obtained by 
summing the number of better scores and adding one. An example is provided in Figure 1.

---

1 [http://googleappengine.blogspot.se/2009/01/google-code-jams-ranking-library.html]
Updating a score by this approach also implies updating the count in each node visited.

The time complexity of finding a rank in the tree is $O(\log n)$ where $n$ is the number of possible scores. The height of the tree is $\log N/n$. Reading or writing a node may be expensive in case it has to be read from or written to a database system, hence the number of ranges per node needs to be chosen so that the height of the tree remains reasonably low. What would be considered as reasonably low depends on details of the implementation such as the cost for database operations, the cost for CPU-time etc.

There are a number of things to consider with this approach. First, to get exact rankings the leaf nodes will be as many as there are distinct scores. This may be a problem if the range of scores is open in one or both directions. One way to get around that problem could be to not store the count of individual scores but instead count of ranges. This could be done in at least two ways; by defining the ranges in beforehand or by compressing the tree at some interval so that if a child’s score range has only a small number of scores, then do not build the tree further down from that child. That is essentially the solution proposed by Shrivastava et al[1] with their Quantile Digest algorithm. A ranking algorithm based on that solution would then not be able to give exact ranks for scores, but have to do approximations. Secondly, if the scores distribution is not known in beforehand the tree may need to be balanced during it’s lifetime in order for this class of algorithms to be efficient.

Figure 1: Example of how ranking is done using a tree-based approach. To get the rank for score 30, start by adding the counts for scores greater than 30 (numbers inside circles). There is 22 higher scores and rank for score 30 is thus 23.

This figure is a reproduction of a similar image from the article Fast and Reliable Ranking in Datastore (https://cloud.google.com/datastore/docs/articles/fast-and-reliable-ranking-in-datastore/), licensed under the Creative Commons Attribution 3.0 licence.

**2.3 Rank by approximation with linear interpolation**

Getting an exact rank for a score may not be crucial. One class of solutions is the ones using linear interpolation to get an approximate rank from a score range and known ranks at...
the lowest and highest scores. The ranks and score ranges needed for the interpolation can be acquired in different ways of which two will be described shortly: **Buckets with Global Query** and **Frugal Streaming**.

Approximating algorithms will almost by definition deviate from the **true value** and we need a way to express that deviation. One way could be expressing the error as the **absolute error** as in 2.1 [2].

\[ \varepsilon_{\text{abs}} = x_0 - x = \Delta x \]  

In 2.1 \( x \) is the true value and \( x_0 \) is in this case our approximated value. Another option would be expressing the error as the **relative error** which is the quota between the absolute error and the true value [2].

\[ \varepsilon_{\text{rel}} = \frac{x_0 - x}{x} = \frac{\Delta x}{x} \]  

A consequence of measuring the relative error when it comes to ranking is that a small rank estimate error will generate a large error when the estimating a high rank, and a relatively small error when estimating lower ranks with the same ranking error in absolute terms, for example see 2.3 and 2.4 where \( |\Delta x| = 1 \) in both cases.

\[ x_0 = 9, x = 10 \implies \varepsilon_{\text{rel}} = \frac{9 - 10}{10} = -0.1 \]  

\[ x_0 = 999, x = 1000 \implies \varepsilon_{\text{rel}} = \frac{999 - 1000}{1000} = -0.001 \]

The property of the relative error shown above coincides with a common requirement when ranking highscores in computer games, namely that higher ranks need to be approximated more precisely (that is, they should have a small \( \varepsilon_{\text{abs}} \)) than lower ranks. This makes the relative error a suitable quality measure for the approximations.

A way to actually avoid large errors for higher ranks may be accomplished by interpolating over fewer ranks when approximating ranks for higher scores while increasing the number of ranks to interpolate over when the score is low (Figure 2). This will keep the number of ranges low while maintaining high precision for the highest ranks.

However, the method described above may not be enough to handle the highest ranks. One reason for that may simply be that the requirements on the algorithm and ultimately the final solution do not allow approximations at all for the top ranks. To solve that problem the approximating algorithms can be paired with the **Rank by counting**-algorithm (see section 2.1) for the highest ranks. This should not be a problem since the counting involves only the critical and highest ranks which can easily be fetched if the data is indexed. When to not estimate can be decided by looking at variables such as the score or rank estimate to name a few examples.
2.3.1 Buckets with Global Query

An approach to ranking by doing linear interpolation is called *Buckets with Global Query*. A **bucket** corresponds to a data structure having a **start score**, **rank for the start score** and the **number of scores within the bucket**. A **bucket-table** consists of a number of buckets. An interpolant for a score that falls into a bucket can be created with data from that bucket and the start score from the following one.

The bucket-table is created by iterating through the whole, sorted set of high scores. Table 1 shows an example of what the table created could look like.

<table>
<thead>
<tr>
<th>Bucket no</th>
<th>Start score</th>
<th>Start rank</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 515</td>
<td>83</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>1 961</td>
<td>105</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>2 204</td>
<td>128</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>2 574</td>
<td>151</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>2 852</td>
<td>175</td>
<td>25</td>
</tr>
</tbody>
</table>

---

2Fast and Reliable Ranking in Datastore: https://cloud.google.com/datastore/docs/articles/fast-and-reliable-ranking-in-datastore/

3Storing both lowest and highest score for a bucket would result in having to update two buckets when a new high score would fall between the two.
So, for example, to approximate the rank for score 2050 which falls in bucket 6, start by calculating what the score range in bucket 6 is, in this case 2204 – 1961 = 243. 2050 is 89 scorepoints “in” the bucket and hence the rank is calculated as $83 + 23 \times \frac{89}{243} \approx 91$. The approximation is illustrated in Figure 3.

**Figure 3:** Linear interpolation

---

**Quality of approximations and the lifetime of a bucket-table**

The quality of the approximations made by this method depends on several factors. The frequency of the periodic scans needs to be high enough to keep the errors at an acceptable level. Also, the size of the buckets have an impact on the result. Yet another factor that cannot be handled in a simple way is when the score distribution within a bucket is skewed or not uniform within the score range as in Figure 4.

**Figure 4:** An example of an uneven distribution of highscores within a bucket. In this case, a score in the middle of the actual distribution in the beginning of bucket a would get a rank estimate based on the score range defined by start scores of bucket a and b, which in this example would result in a higher rank than the actual.

When a highscore update occurs, from a lower score to a higher, but the new score falls in the same bucket as the previous highscore, the subsequent approximation will be the same as before for any score since the bucket-table is unchanged. Because the underlying highscore distribution have changed the error of the next approximation may be different but not necessarily worse.
On the other hand, if a new highscore estimate goes from a lower rank bucket to a higher one, a subsequent scan of the highscores would result in a table where the new bucket will grow by one and, start-rank of buckets after the new bucket to and including the former one will increase by one and the bucket for the previous estimate would shrink by one.

Future highscore updates will add to the bucket-tables deviation from a theoretical true bucket-table and as a consequence the estimates will show a growing error with respect to the true rank until the bucket-table is recreated (Figure 5). It is worth noting that the error should not be expected to be zero even with a fresh bucket-table since approximations is precisely that – approximations.

**Figure 5:** Error in approximations will grow over time, until the bucket-table is recreated and the error level drops down to the initial error. Bucket-table recreated at \( t_0, t_1, t_2, \ldots \).
2.3.2 Online approaches, streaming algorithms

Frugal Streaming[3] is an algorithm for continuously estimating quantiles from a stream. It is an online and streaming algorithm, meaning that it “can process its input piece-by-piece in a serial fashion”⁴ and that it is “only allowed to access the input in streaming fashion”⁵ and “does not have random access to the tokens”.

Assume we know the number of highscores, \( n \). By using the Frugal Streaming algorithm on the incoming highscores we also know at least two quantiles, for example \( q_{25\%} \) and \( q_{50\%} \) that are marked in Figure 6. At \( q_{25\%} \) we have the highest score 25 % of the lowest ranked players (or lowest score of the other 75 %). The same reasonings apply to quartile \( q_{50\%} \), \( \frac{n}{2} \) players have a lower score and \( \frac{n}{2} \) have a higher score (For the moment, the possibility of an odd number of highscores is ignored).

**Figure 6**: Distribution of highscores with quantiles \( q_{0\%} \), \( q_{25\%} \) and \( q_{50\%} \) marked.

Since we have the ranks and estimate of the score at the two quantiles we can do linear interpolation between our quantiles. For example, at \( q_{25\%} \) the rank is \((1 - 0.25) \times n\), 75 % of all players have a higher score) and the score is the output of running the Frugal Streaming algorithm for those quantiles.

A drawback with this procedure is that every single new highscore would result in running the Frugal Streaming algorithm as many times as the number of quantiles needed to make precise enough approximations. Also, it needs some time to stabilize so it cannot be used from the absolute beginning when there are only a few highscores.

---

⁴ https://en.wikipedia.org/wiki/Online_algorithm
⁵ CS49: Data Stream Algorithms - Lecture Notes Fall 2011, Amit Chakrabarti
http://www.cs.dartmouth.edu/~ac/Teach/CS49-Fall11/Notes/lecnotes.pdf
3 Improving the Bucket with Global Query-algorithm

The experiment described in this chapter is to a great extent adapted to match the conditions of a specific implementation of the Buckets with Global Query-algorithm used by a company making games for the iOS and Android platforms. The back end for the game where the ranking is done is run on Google App Engine (GAE) which is a cloud computing service offering a scalable, pay-as-you-go platform for running web services among other things.

Since applications running on GAE do not have access to a file system, data is stored in the App Engine Datastore which is NoSQL key-value-store. While the application part of the GAE automatically scales as far as the configuration allows, the throughput when writing to the same object in the Datastore is very limited and measures need to be taken to avoid congestion.

In this case, highscores are stored the Datastore as entities with properties *username* and a *highscore*. When a player gets a new highscore it replaces the old highscore. Note that this implies that highscore entries never will get updated with a lower score than previously recorded for that user. Since score in this case is measured in milliseconds and lower timings are better a better score is a lower number.

3.1 Hypothesis

The current implementation starts a background job that builds the bucket-table every ten minutes. When the scan is complete a reference is set to the new table and the old one is discarded. The error is a priori assumed to be at its maximum level right before the switch (See Figure 5) and will in the forthcoming reasonings serve as an upper limit for what would be an acceptable performance.

The cost for ranking a single highscore is

\[ n \times C_{\text{estimate}} + \frac{C_{\text{table}}}{n} \]  

(3.1)

where \( n \) is the number of approximations made with one bucket-table (ie. the lifetime of the bucket-table), \( C_{\text{estimate}} \) is the execution time for making one estimate and \( C_{\text{table}} \) is the cost for recreating the bucket-table.

In the original Bucket with Global Query-algorithm, getting an approximate rank is a lookup-operation in a table (or similar operation on another data structure) and some simple arithmetics, hence the execution time cannot really be improved for that part. The cost for creating a bucket-table can not be radically improved either because it has to iterate over...
the set of highscores. But if the lifetime of the bucket-table could be extended, its addition to the cost per rank estimate would get lower. As shown in Section 2.3.1, the bucket-table in use starts to deviate from a true bucket-table when new highscores comes in. The tables score ranges, start ranks and bucket sizes simply don not correspond to the actual set of highscores.

So what if the table’s buckets were adjusted when a new highscore gets registered? The overhead caused by adjusting start ranks and bucket sizes will be small because it can be accomplished in a few lines of code, mostly concerned with increasing and decreasing integer primitive values. But there is also an overhead coming from storing the adjusted table which would apply to every single new highscore.

To sum up, the cost for approximating the rank for a highscore will be more expensive when also updating the bucket-table, but will the increased cost be compensated for by the expected increase in lifetime of the bucket-table? The cost for recreating the bucket table will remain the same. Two possible scenarios are sketched in Figure 7.

![Figure 7: Possible scenarios for cost growth.]

The hypothesis reads as follows: **Running the ranking service with the new implementation will be more efficient than the current implementation at the same level of error.**

### 3.2 Data

In this experiment only synthetic data will be used. The choice is both practical and a methodologically motivated. First, the production system as cannot be altered in such a way that real world data could be tapped within this experiments timeframe. Also, real world data in this system differs from one time period to another both in terms of the highscore distribution and throughput. And of course, data from one application differs from data from other applications. Using synthetic data make the results more general.

Random data is used in two ways. First, the experiment is started with a set of 100 000 highscores having a Gaussian distribution with mean 1 000 000 and variance 1 000 000, also scores below 1000 are not included\(^1\) (Figure 8). Second, the improvement for the new highscores used in the experiment are picked from a uniform distribution between 1 and 1000.

---

\(^1\)The real world highscores are similarly distributed to each other and somewhat similar to a Gaussian distribution. However, no effort has been made to do a statistical analysis of them.
The random number generator in Java is used and is always initialized with the same random seed to achieve repeatability.

### 3.3 The experiment

The experiment is implemented in a client-server-model where the client simulates playing a game and sends highscores to the server. The server responds with an approximated rank and the execution time for processing the request. The client is a regular Java program without bells and whistles and the server consists of a number of servlets designed to run on Google App Engine.

The client plays 100,000 rounds for different users, picked at random. They will always get a new highscore, i.e., they always win. The new highscore is better than the old one by a number drawn from a uniform pseudorandom distribution $1 - 1000$.

To be able to measure the relative error of the estimates the client starts by asking for a list of all highscores. When the client sends a new highscore to the server, it also updates its local highscore list. When the server responds with the rank estimate for the new score, that estimate is compared with the real rank and ultimately used for calculating the relative error.

### Technical details

Both client and server parts are run on the same physical computer.

Two libraries are used in the project. **Objectify** provides means for persisting Java objects to the App Engines Datastore. By a few annotations $^2$ to the Java-class (@Entity on the class and @Id on the String or Long field that will be used for creating the key) objects can be saved with a statement like `ofy().save().entity(theObject)`. On top of providing an easy way for persisting objects, Objectify also handles caching data in the memory cache. **Jackson** is a library for parsing JSON-data as well as mapping JSON to Java objects.

---

$^2$ https://docs.oracle.com/javase/tutorial/java/annotations/index.html
3.4 What and how to measure

To be able to test the hypothesis relative error and execution time will be logged per ranking request. As the true value for calculating the relative error, an exact rank is calculated for every approximation.

The execution time is measured by calls to `System.currentTimeMillis()` in the servlet receiving the ranking request.

3.5 Limitations

The experiment is only run using a local development server. While there is no obvious reason to believe that the conclusions could be applied to a production environment in the cloud in principle, there may be factors that need to be considered.

The experiment could have been set up to be run several times with different random seed. This would make the data gained smoother and conclusions more well founded.
4 Results

4.1 Relative error

Figure 9 shows the absolute value of the relative error (|ε_{rel}|) for 100 000 highscore updates. After 100 000 highscore updates the dynamic bucket-tables’ relative error remains practically constant at 0.05%. For the static bucket-table, the error grows from 7% to 0.25%.

Figure 9: Development of the relative error for 100 000 highscore updates.

Figure 10 is a plot of the same relative error but for approximations 1 – 10 000. It shows where the difference between the two algorithms starts to be significant. The relative error is practically the same after 4 000 updates, after that the original algorithm starts to deviate noticeably\(^1\).

\(^1\)The results would not be exactly the same if the experiment had been run a number of times with different random seeds to create an average relative error. This is also the reason for causing ambiguity by not specifying \(t_1\) and \(t_2\) exactly.
Table 2  Relative error with static buckets, %

<table>
<thead>
<tr>
<th>Approx $n$ ($\times 1000$)</th>
<th>[1-20)</th>
<th>[20-40)</th>
<th>[40-60)</th>
<th>[60-80)</th>
<th>[80-100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.07</td>
<td>0.11</td>
<td>0.16</td>
<td>0.21</td>
<td>0.25</td>
</tr>
<tr>
<td>Median</td>
<td>0.03</td>
<td>0.04</td>
<td>0.05</td>
<td>0.06</td>
<td>0.08</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>0.19</td>
<td>0.43</td>
<td>0.64</td>
<td>0.86</td>
<td>0.96</td>
</tr>
<tr>
<td>Maximum</td>
<td>6.34</td>
<td>20.5</td>
<td>21.9</td>
<td>28.4</td>
<td>33.7</td>
</tr>
</tbody>
</table>

Figure 10: Relative error for 10 000 highscore updates.

4.2 Execution time

The execution time is measured on a local development server and not on the cloud platform targeted. The local development server is an Intel Core i7-6700K with 16 GB of RAM which at the time of writing has the fastest single-thread performance available. The virtual machines (instances) provided by Google App Engine are far less powerful. For example, the least powerful instance have CPU-limit corresponding to 600 MHz on an unspecified architecture and 128 MB of RAM\(^2\). No measurements have been done to determine what kind of scaling factor would apply when translating the results to a production environment.

Cost of making one approximation

The cost for approximating one rank is composed of a share of the cost for building a bucket-table and the cost for making the approximation. With the dynamic bucket-table approach, adjusting and storing the bucket-table adds an additional cost to each approximation. The cost for making an approximation for the two methods compared are presented in Table 4. Both runs of the experiment had about one hundred outliers (> 100 ms) which affect the mean significantly by 0.6 ms in both cases.

\(^2\) https://cloud.google.com/appengine/docs/about-the-standard-environment#instance_classes
Table 3 Relative error with dynamic buckets

<table>
<thead>
<tr>
<th>Approx n (×1000)</th>
<th>[1-20)</th>
<th>[20-40)</th>
<th>[40-60)</th>
<th>[60-80)</th>
<th>[80-100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.06</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>Median</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>0.14</td>
<td>0.11</td>
<td>0.12</td>
<td>0.12</td>
<td>0.13</td>
</tr>
<tr>
<td>Maximum</td>
<td>6.76</td>
<td>2.50</td>
<td>4.40</td>
<td>4.21</td>
<td>5.83</td>
</tr>
</tbody>
</table>

Table 4 CPU-time in milliseconds (ms)

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Static buckets</th>
<th>Dynamic buckets</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>5.914</td>
<td>7.604</td>
<td>1.690</td>
</tr>
<tr>
<td>Median</td>
<td>5</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>17.418</td>
<td>19.029</td>
<td>1.611</td>
</tr>
<tr>
<td>Maximum</td>
<td>702</td>
<td>757</td>
<td>55</td>
</tr>
</tbody>
</table>

Cost of creating a bucket-table

Building a bucket-table for 100 000 highscores takes 10 205 ms in average. The time does not vary much between test runs (Table 5).

Table 5 Bucket-table creation time, 100 000 highscores. Sample size: 5.

<table>
<thead>
<tr>
<th>Bucket-table creation time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
</tr>
<tr>
<td>Median</td>
</tr>
<tr>
<td>Standard Deviation</td>
</tr>
<tr>
<td>Maximum</td>
</tr>
</tbody>
</table>
For the length of the experiment the relative error for the solution with the dynamic bucket-table remains practically constant\(^1\). Because the buckets for the higher ranks will grow in size (highscores always improve when updated), the approximations will get less accurate over time. The experiments were not run long enough to tell when the bucket-table would need to be recreated in the dynamic approach.

While the data gained do not tell when the bucket-table would need to be recreated we can still assume that it would last for 100,000 updates.

What an acceptable error level would be is a question that ultimately needs to be answered by stakeholders in charge of the application the rank approximations are used in. However, having a mean realative error of 0.05% when approximating a rank at position 30,000 means having an error in absolute ranks of 15 and could hardly be considered not precise enough in a computer game.

The hypothesis stated in Chapter 3 can be said to be true, however by an insignificant amount of CPU-time. Table 6 shows a summary of CPU-time used for the dynamic approach and the static approach.

<table>
<thead>
<tr>
<th></th>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recreate table every</td>
<td>100 000</td>
<td>5 000</td>
</tr>
<tr>
<td>Num bucket-table creations</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10 000</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>15 000</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>20 000</td>
<td>5</td>
</tr>
<tr>
<td>CPU-time bucket-table creation</td>
<td>10.25</td>
<td>204.1</td>
</tr>
<tr>
<td>CPU-time approximations</td>
<td>760</td>
<td>591</td>
</tr>
<tr>
<td>Total time (s)</td>
<td>771</td>
<td>796</td>
</tr>
</tbody>
</table>

\(^1\)Linear regression gives that it is actually *decreasing* with the data set used, albeit by a modest \(-10^{-11}\) per highscore update.

Next up

There are a number of interesting paths to follow from here and the number one and two priorities should be to perform the experiment on a real platform and doing the experiments with different random seeds.
Also, there are a lot of parameters that could be tweaked, such as properties of the distribution of the initial highscores and the new highscores, as well as the distribution of the players actually playing. The impact of the size of the buckets is not investigated at all in this thesis.

Optimizing the new implementation by utilizing the memory cache available in Google App Engine directly in the ranking function instead of relying on the cache functionality built into Objectify would probably result in significant performance gains.
References

