Algorithms and Data Structures

Self-Organizing Lists

Ulf Leser

Assumptions for Searching

- Until now, we always assumed that every element of our list is searched with the same probability, i.e., with the same frequency
- Accordingly, we treated all elements of the list equal
- We may sort the list by properties of its values, but we did never consider properties of their usage
- This setting sometimes is inadequate

Searches on the Web [Germany, 2010, Google Zeitgeist]

Schnellst wachsende Suchbegriffe

1. wm 2010
2. chatroulette
3. ipad
4. dsds 2010
5. immobilienscout24
6. iphone 4
7. facebook
8. zalando
9. google street view
10. studi vz

Die häufigsten Suchbegriffe

1. facebook
2. youtube
3. berlin
4. ebay
5. google
6. wetter
7. tv
8. gmx
9. you
10. test

Meist gesuchte Personen

1. lena meyer-landrut
2. jörg kachelmann
3. daniela katzenberger
4. justin bieber
5. shakira
6. katy perry
7. david guetta
8. miley cyrus
9. rihanna
10. megan fox

Beliebte Produkte

1. ipod
2. handy
3. schuhe
4. fernseher
5. iphone
6. notebook
7. wii
8. ipad

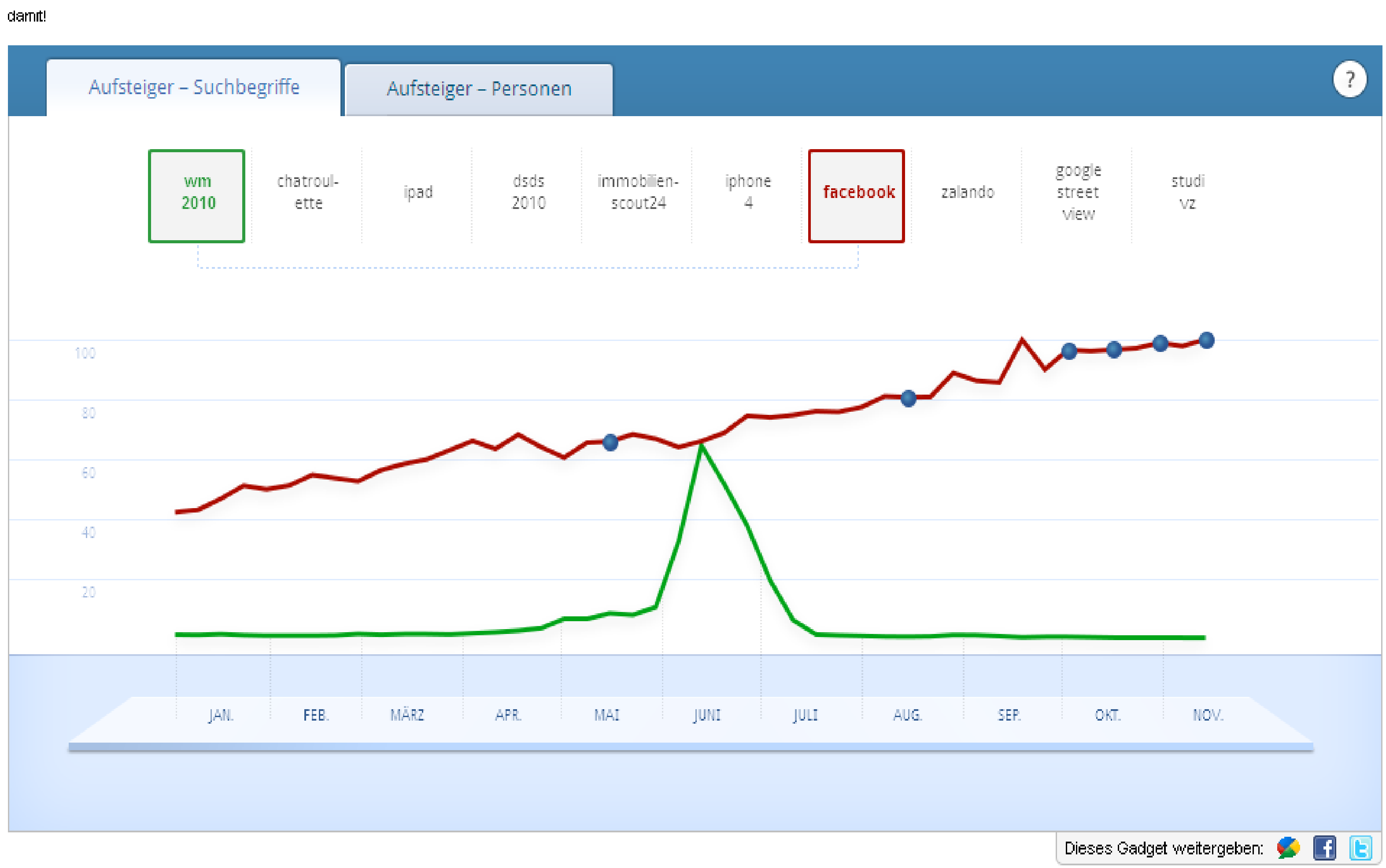
Meist gesuchte Nachrichten

1. bayern
2. menowin fröhlich
3. jörg kachelmann
4. stuttgart 21
5. iphone
6. fc bayern
7. aschewolke
8. daniela katzenberger

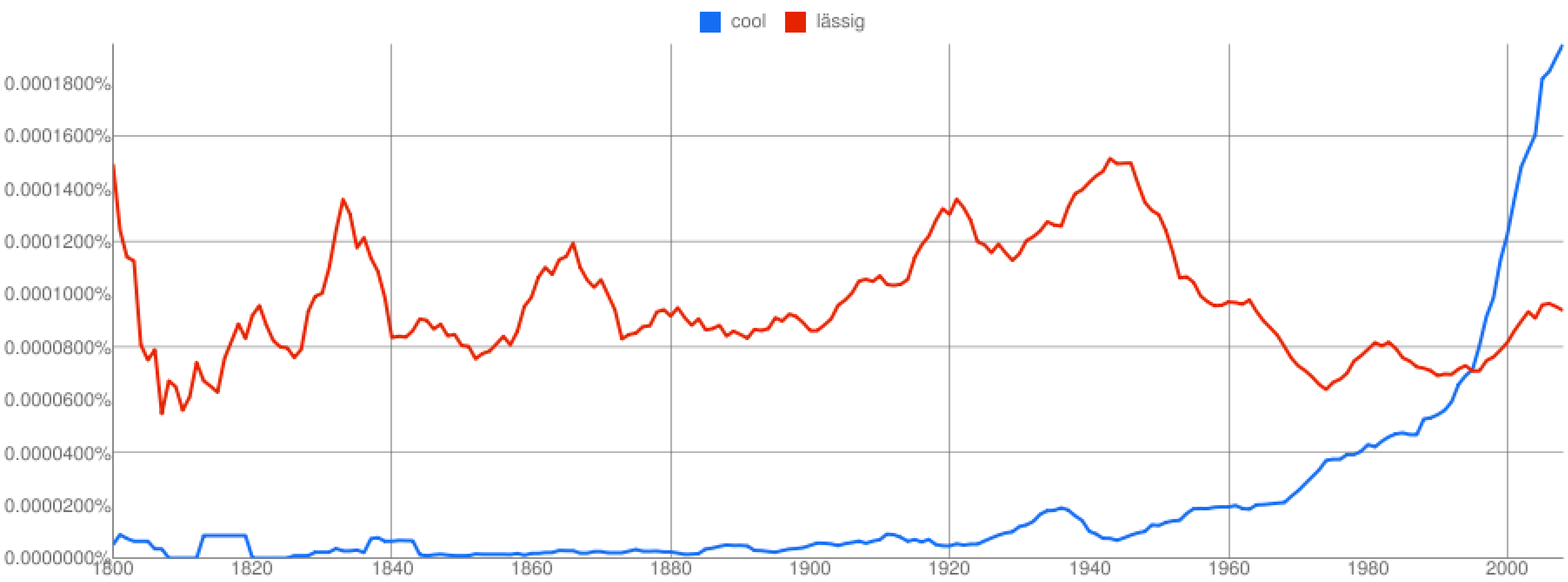
Beliebte Bildersuchen

1. ipad
2. lena meyer-landrut
3. larissa riquelme
4. mehrzad marashi
5. menowin fröhlich
6. vampire diaries
7. frisuren 2010
8. kesha

Changing Frequencies [Google Zeitgeist]

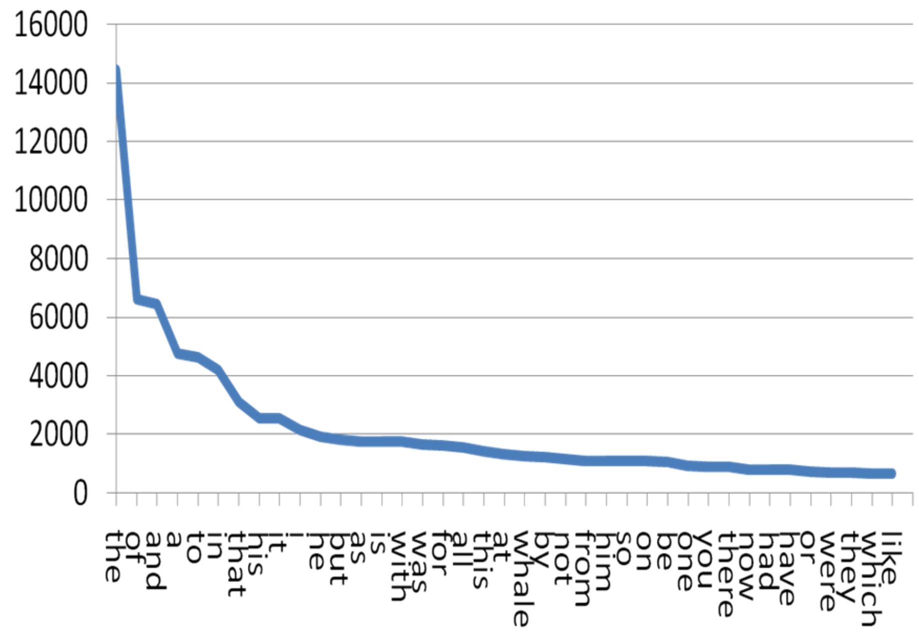


Changing Word Usage [Google n‘gram viewer]



Zipf-Distribution

- Many events are not equally but Zipf-distributed
  - Let f be the frequency of an event and r its rank in the list of all events sorted by frequency
  - Zipf's law: $f \sim k/r$ for some constant k
- Examples
  - Search terms on the web
  - Purchased goods
  - Words in a text
  - Sizes of cities
  - Opened files in a OS
  - ...



Source: http://searchengineland.com/the-long-tail-of-search-12198

Changing the Scenario

- Assume we have a list L of values
- L is searched very often
- But: Not all values in L are searched with the same frequencies
- How can we organize L such that a series of searches are as fast as possible?
- Let L organize itself depending on its usage

Content of this Lecture

- Self-Organizing Lists
  - Fixed frequencies
  - Dynamic frequencies
- Organization Strategies
- Analysis

Simple Case: Fixed Frequencies

- For simplicity, we assume L has $n=|L|$ different values
- Assume that we know the relative frequency p_i with which each of the n values in L will be searched ($1 \le i \le n$)
- Example: Assume p_i is distributed with $p_i = 1/(1+i)^2 * c$
  - Assume $n=25$
  - c: normalization factor to ensure $\sum p_i = 1$
  - Yields something like 41%, 18%, 10%, 6%, 4%, 3%, 2%, 1%, ...

Analysis

- What are the expected costs for a series of searches?
- Option 1: Assume L is sorted by value and we search L with log(n) comparisons upon each search
  - Expected cost for 100 searches: 100*log(n) ~ 500
- Option 2: Assume L is sorted by p_i and we search L linearly upon each search
  - In 41% of cases 1 access; in 18% 2; in 10% 3; ...
  - For 100 searches: 1*41+2*18+3*10+4*6+5*4+6*3+ ... = 386

Other Distributions

- Using $p_i=1/(1+i)^3*c$, we have 200 accesses for the frequency-sorted list, but still ~500 for the value-sorted list
  - Access frequencies: 62, 18, 7, 4, ...
- But: For $p_i=1/n$, we have 1336 versus ~500 accesses
  - Equal distribution, access frequencies: 4, 4, 4, 4, ...
- Summary
  - Sorting the list by „popularity“ may make sense
  - Gain (or loss) in efficiency can be computed before-hand if frequency of accesses are known (and do not change)

Content of this Lecture

- Self-Organizing Lists
  - Fixed frequencies
  - Dynamic frequencies
- Organization Strategies
- Analysis

Self-Organizing Lists

- More interesting scenario
  - Access frequencies are not known in advance
  - Access frequencies change over time
  - Implication: It is not optimal to log searches for some time, then compute popularity, then re-sort list
- Changing the list
  - After each access, we may change the order in the list
  - Searching the (currently) i'th element of the list costs i operations
    - I.e., L is implemented as linked list
    - Using arrays doesn't help – we don't know where the searched value is
- This scenario is called a self-organizing linear list (SOL)

Re-Organization Strategies

- Many proposals in the literature
- Many are very application specific
- Three popular general strategies
  - MF, move-to-front:
    After searching an element e, move e to the front of L
  - T, transpose:
    After searching an element e, swap e with its predecessor in L
  - FC, frequency count:
    Keep an access frequency counter for every element in L and keep L sorted by this counter. After searching e, increase counter of e and move “up” to keep sorted’ness

Application: Caching

- Often, the user wants to read more data from disk than there is main memory
  - Especially if there are more than one user
- Reading from disk is ~1000 times slower than from memory
- Caching: OS keep data (blocks) in memory for which it expects that they will be reused (in the near future)
- There is not enough space to keep all ever used blocks
- Thus, when loading new blocks, the OS has to evict blocks from the cache – which ones?
  - Those that probably will not be reused in the near feature

Caching and SOLs

- The OS could keep a SOL S with all block IDs sorted by their popularity
- The top-k of these blocks are cached
- When loading a new block b, the OS ...
  - Evicts the k'th block in S from memory
  - Loads b into the free space
  - Re-organize S to reflect the change in popularity of b
- Prominent strategies in caching
  - Most recently used: Popularity is the time stamp of the last usage
  - Most frequently used: Popularity is the number of access until now
- See course on Operating Systems (or/and Databases)

Content of this Lecture

- Self-Organizing Linear Lists
- Organization Strategies
- Analysis

Properties

- MF
  - If a rare element is accessed, it “jams” the list head for some time
  - Bursts of frequent same-element accesses are well supported
  - No problem with changes in popularity (trends)
- T
  - Problems with fast changing trends – slow adaptation
  - Frequently accessing same-elements well supported
    - After some swing-in time
- FC
  - Requires O(n) additional space
  - Re-sorting requires WC O(log(n)) time (binsearch in L[1...e])
    - Rather O(1) on average
  - Slow adaptation to changing trends – old counts dominate list head

Examples

- For each strategy, we can find sequences of accesses that are very well supported and others that are not
- Example: L={1,2,...7}, n=7
  - S_1: {1,2,...7, 1,2,...7, 1,2,... ... ...7} (ten times)
  - S_2: {1,1,1,1,1,1,1,1,1,1, 2,2,2,... ... 6, 7,7,7,7,7,7,7,7,7,7}
  - Each sequence performs 70 searches, each element is accessed with the same relative frequency 1/7
- Assume any static order
  - There are seven different costs 1, ... 7
  - Each cost is incurrent 10 times
  - Thus, the average cost will be

$$\frac{1}{10*n}*\left(\sum_{i=1}^{n}10*i\right)=4$$

MF: Average Cost

Almost worst case

- MF / S_1
  - In the first subsequence, we require i ops for the i'th access
  - L then looks like 7,6,5,4,3,2,1
  - We require 7 ops per element for every further subsequence
  - Together
    - Much worse than static order

$$\frac{1}{10*n}\left(\sum_{i=1}^{n} i + 7*9*7\right) = 6.7$$

Almost best case

- MF / S_2
  - First subsequence requires 10=1+9 ops
  - Second requires 2+9
  - Third requires 3+9
  - Together
    - Much better than static order

$$\frac{1}{10*n}\left(\sum_{i=1}^{n} i + 9*7*1\right) = 1.3$$

FC: Average Cost

- FC / S_1 (all counters are initialized with 0)
  - First subsequence costs ∑i and doesn't change order
    - Assuming stable sorting; now all counters are 1
  - Same for all other subsequences
  - Together
    - Ignoring re-sorting costs

$$\frac{1}{10*n}*10*\left(\sum_{i=1}^{n} i\right) = 4$$

- FC / S_2
  - First subsequence costs 10 and no change in order
  - Second subsequence costs 20 and no change in order
  - Same for all other subsequences
  - Together
    - Ignoring re-sorting costs

$$\frac{1}{10*n}*\left(\sum_{i=1}^{n} 10*i\right) = 4$$

T: Average Cost

- T/ S_1
  - First subsequence costs $\sum i = 28$
  - Order now is 2,3,4,5,6,7,1 – next subseq costs 7+1+2+...5+7 = 29
  - Order now is 3,4,5,6,2,7,1 – next subseq costs 7+... = 30
  - ...

Access	3	4	5	6	2	7	1	Costs
1	3	4	5	6	2	1	7	7
2	3	4	5	2	6	1	7	5
3	3	4	5	2	6	1	7	1
4	4	3	5	2	6	1	7	2
5	4	5	3	2	6	1	7	3
6	4	5	3	6	2	1	7	5
7	4	5	3	6	2	7	1	7

Optimal Strategies

- "Optimality" of a strategy depends on the sequence of accesses
- Conventional worst-case estimation uses worst-case for every single access, which is $O(n)$ for every strategy, and thus $O(n^2)$ for a sequence of n searches
- This is overly pessimistic: Accesses (by self-org) influence the cost of subsequent accesses
- Using a clever trick, we can derive estimates about the relative costs for different strategies over any sequence
- This trick is called amortized analysis
- This will take some time