Winning* the SIGMOD 2013 programming contest

Henrik Mühe  henrik.muehe@in.tum.de
Florian Funke  florian.funke@in.tum.de
<table>
<thead>
<tr>
<th>Rank</th>
<th>Team</th>
<th>Small (sec)</th>
<th>Big (sec)</th>
<th>New (sec)</th>
<th>Upload Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Campers (TUM)</td>
<td>0.081</td>
<td>1.938</td>
<td>7.515</td>
<td>Apr 15 - 09:50pm</td>
</tr>
<tr>
<td>2</td>
<td>RotaFortunae (Saint Petersburg University)</td>
<td>0.158</td>
<td>1.969</td>
<td>9.394</td>
<td>Apr 15 - 08:25pm</td>
</tr>
<tr>
<td>3</td>
<td>mofumofu (Tohoku University)</td>
<td>0.065</td>
<td>1.507</td>
<td>10.343</td>
<td>Apr 13 - 06:59pm</td>
</tr>
<tr>
<td>4</td>
<td>glhf</td>
<td>0.137</td>
<td>2.100</td>
<td>11.795</td>
<td>Apr 15 - 06:38pm</td>
</tr>
<tr>
<td>5</td>
<td>phoenix (Peking University)</td>
<td>0.585</td>
<td>2.320</td>
<td>12.794</td>
<td>Apr 15 - 05:24pm</td>
</tr>
<tr>
<td>6</td>
<td>StrongAccept (Tsinghua University)</td>
<td>0.396</td>
<td>3.019</td>
<td>12.848</td>
<td>Apr 15 - 08:28pm</td>
</tr>
<tr>
<td>93</td>
<td>ePetra</td>
<td>22.463</td>
<td>N/A</td>
<td>N/A</td>
<td>Apr 08 - 12:05pm</td>
</tr>
<tr>
<td>53</td>
<td>JoblessCoders</td>
<td>30.927</td>
<td>N/A</td>
<td>N/A</td>
<td>Apr 15 - 07:47pm</td>
</tr>
<tr>
<td>54</td>
<td>TangYuan</td>
<td>43.174</td>
<td>N/A</td>
<td>N/A</td>
<td>Mar 03 - 09:01am</td>
</tr>
<tr>
<td>55</td>
<td>TangYuan</td>
<td>43.798</td>
<td>N/A</td>
<td>N/A</td>
<td>Mar 07 - 10:33pm</td>
</tr>
</tbody>
</table>
The Challenge

System

Q1: visa work aupair
Q2: justin bieber
Q3: alien attack
Q4: ...
...
Q9999999: ....

Data Flow
The Metrics:
Exact Match

Query matches a document iff all query words are contained in the document.
The Metrics: Hamming Distance

Query matches a document iff all query words are within hamming distance $d$ of at least one word inside the document.

Hamming?

1 position differs $\Rightarrow$ HD=1
The Metrics: Levenshtein Distance

Query matches a document iff all query words are within Levenshtein distance $d$ of at least one word in the document.
Levenshtein Examples

\[
\text{levenshtein}
\begin{align*}
\text{henrik} & \quad \text{jenrik} \\
= 1 \quad (= \text{hd}())
\end{align*}
\]

\[
\text{levenshtein}
\begin{align*}
\text{alfons} & \quad \text{fonts}
= 3
\end{align*}
\]
Levenshtein Definition

`levenshtein(a,b) :=`  
Lowest number of  
- Replace  
- Insert  
- Remove  
  to change a into b  

$O(|a|*|b|) \Leftrightarrow \text{terrible}$
Baseline

- tar.gz download, fully functional
- Naive 'nested-loop' style
- Unbearably slow
- Horrible, horrible code

```c
int cur=0;
ia=0;
for(ib=0;ib<=nb;ib++)
    T[cur][ib]=ib;
cur=1-cur;
```
**Baseline Analysis**

$ ./testdriver
Start Test ...
Your program has passed all tests.
Time=30704[30s:704ms]

$ perf record ./testdriver && perf report

### Samples: 122K of event 'cycles', Event count (approx.): 115188817384

<table>
<thead>
<tr>
<th>Function</th>
<th>Time Percent</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>EditDistance(char*, int, char*)</td>
<td>72.69%</td>
<td>115188817384</td>
</tr>
<tr>
<td>MatchDocument</td>
<td>15.17%</td>
<td></td>
</tr>
<tr>
<td>__strcmp_sse42</td>
<td>10.78%</td>
<td></td>
</tr>
<tr>
<td>HammingDistance(char*, int, char*)</td>
<td>0.45%</td>
<td></td>
</tr>
<tr>
<td>strcmp@plt</td>
<td>0.33%</td>
<td></td>
</tr>
<tr>
<td>_Z15HammingDistancePciS_i@plt</td>
<td>0.22%</td>
<td></td>
</tr>
<tr>
<td>_Z12EditDistancePciS_i@plt</td>
<td>0.22%</td>
<td></td>
</tr>
<tr>
<td>_IO_vfscanf</td>
<td>0.05%</td>
<td></td>
</tr>
<tr>
<td>__memmove_sse3_back</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>native_write_msr_safe</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>__ticket_spin_lock</td>
<td>0.01%</td>
<td></td>
</tr>
</tbody>
</table>
API

- StartQuery
- EndQuery
- MatchDocument
- GetNextAvailRes
The Magic Sauce

1. Massive parallelism
2. Architecture-aware optimizations
3. Efficient computation of metrics
4. Filtering
5. Indexing
6. Caching
1. Parallelism & Concurrency
1. Parallelism & Concurrency

MatchDocument

- Spawn async task with subtasks for each match type
- Parallelize Hamming & Levensthein distance
- Avoid sync points

Intel® TBB
Inherent Optimization Potentials
Deduplication

- Remove all duplicates in document
- Match every query word only once (even if it is in multiple queries)
Caveats

Q1: henrik mühe
Q2: henrik database
Q3: henrik funfacts

QueryWords: henrik, mühe, database, funfacts
Caveats

Q1: henrik mühe
Q2: henrik database
Q3: henrik funfacts

QueryWords: henrik, mühe, database, funfacts

Document.probe(henrik) -> false
What about: mühe, database, funfacts
Cover Pruning

- For every word, determine which words can be skipped.
  - Full computation too expensive
  - When a query is added, remove word from invalidated dependency sets
  - Do not re-add
  - Recompute when queries have changed substantially

- Skip vector in hot loop

- Harmless race condition
2. Architecture-Aware Optimizations

- SIMD: Single Instruction Multiple Data
  - Hamming/Edit Distance
  - Filter computation

- Special Instructions
  - CRC32
3. Efficient computation of metrics
Improving Exact Match

Insert all query words into Hashmap
Signature: hash<QueryWord, vector<Query>>

1. Probe each document word &
   Mark QueryWord as matched
2. Count matching words per query
3. Generate result
Improving Hamming Distance

Materialize all and add to Exact Matcher?
Improving Hamming Distance

Materialize all and add to Exact Matcher?

For word with length 10 and distance 3 roughly

\[
\begin{align*}
  d=1 & \quad 10 \times 25 \\
  d=2 & \quad + (10 \times 25)^2 \\
  d=3 & \quad + (10 \times 25)^3 \\
\end{align*}
\]

\[\gg 15 000 000\]
Improving Hamming Distance

Hamming is essentially the sum of byte-wise XOR

\[
x = \text{aaaaabbbb} \\
y = \text{bbaaaabbb} \\
\text{sum} \ (11001000) = 3 = \text{hamming}(x, y)
\]
Improving Hamming Distance

SIMD easy solution:

POPCNT(PCMPESTRM)

SIMD fastest solution:

CENSORED
/// Compute levenshtein distance recursively
inline uint32_t levenshtein_rec(StringRef a, StringRef b) {
    // If one of the strings is empty, return the number of characters left
    if (a.length() == 0) return b.length();
    if (b.length() == 0) return a.length();

    // If the first two characters are equal, the edit distance is the edit
    // distance between the two suffixes
    if (a[0] == b[0]) return levenshtein_rec(a.substring(1), b.substring(1));

    // If they are not equal, try insert, remove and substitution
    // Pretend a is b with an extra letter in front
    uint32_t dInsert = levenshtein_rec(a.substring(1), b);
    // Pretend a is b with the first letter removed
    uint32_t dRemove = levenshtein_rec(a, b.substring(1));
    uint32_t dSubst = levenshtein_rec(a.substring(1), b.substring(1));

    // Return the best of the three possibilities above and add one for the
    // insert/remove/substitution we did
    return std::min(dInsert, std::min(dRemove, dSubst)) + 1;
}
Improving Edit Distance

- Superset of Hamming Operations
- Literature Research
  - Validation:
    - Levenshtein Automata
  - Improved Algorithms
    - Memoization (matrix)
    - Less memoization (column)
    - Bit-parallel Levenshtein
Figure 2: A finite automaton accepting strings less than three edits from “food”
4. Filtering
4. Filtering

- Determine if two words can be within edit/hamming distance
- Filter computation should be cheaper than metric invocation...
- Filters
  - Length
  - QGram
  - ...
  - Frequency

\[
\left| qg(a, q) \cap qg(b, q) \right| < (\max(|a|, |b|) - q + 1) - q \cdot d
\]
Frequency Filter

Looking at the histograms of two words:

\[ x = \text{aaabbb} \]
\[ y = \text{aacbba} \]

Define \textit{delta} operation

Max possible \textit{delta}: 2d-lengthdiff
5. Indexing
5. Indexing

- Physically reorganize words by some order relation
- Limit search space to a collocated subset
- Orders
  - Length
- Build column store
- Additive pointer arithmetics in hot loop
6. Caching

- Observation: People make the same mistakes again and again
- Remember last match
  - for each query word
  - for each distance
- Probing a hashtable is a lot cheaper than finding an edit distance match in an entire doc
6. Caching

- Observation: People make the same mistakes again and again
- Remember last match
  - for each query word
  - for each distance
- Probing a (good!) hashtable is a lot cheaper than finding an edit distance match in an entire doc
$ ./testdriver
Start Test ...
Your program has passed all tests.
Time=30704[30s:704ms]

VS.

<table>
<thead>
<tr>
<th>Team</th>
<th>Small (sec)</th>
<th>Big (sec)</th>
<th>New (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Campers (TUM)</td>
<td>0.081</td>
<td>1.938</td>
<td>7.515</td>
</tr>
</tbody>
</table>