Indexes and Algorithms for Scalable and Flexible Instant Overview Search

Pavlos Fafalios
Master’s Thesis
July 2012

Supervisor: Yannis Tzitzikas
Outline

• Introduction
  – What is Instant Search
  – What is Instant Overview Search (IOS)
  – Key Challenges

• Our approach
  (1) Trie-based Index Structures
  (2) Throughput and Caching
  (3) “Flexible” Recommendations

• Experimental Evaluation

• Server’s Benefits

• Conclusion and Further Research

• (Demos)
Introduction
What is Instant Search?

Google

wikipe\id\ia

wikipedia
wind
winbank
windows live

Wikipedia, the free encyclopedia

Wikipedia, the free encyclopedia that anyone can edit.
The problem

- Most web search engines return a ranked list of results
- Users have to explore the answer linearly
- In practice, users tend to look only at the first page of results, missing useful hits
- Users rarely exploit the available metadata during their searches (advanced search)
- In case user has limited knowledge about the underlying data, he has to use a try-and-see approach
What is Instant Overview Search (IOS)

Clustering

Meta-data based groupings

precomputed aggregated information

Entities

?
What is Instant Overview Search (IOS)

User has typed only 2 characters

Position: 11 (2nd page)

Position: 16 (2nd page)

Position: 22 (3rd page)

Position: 38 (4th page)
iOS from a **Decision Making** point of view

- **Aid for acting**
  - User is getting information about the available options

- **Gives the user an overview of the information space**

- **User can decide fast what hits of the answer to inspect**

- **Aid for choosing an option**
  - User is getting information about the available options

- **Aid for evaluating the choices**
  - Gives the user an overview of the information space

- **Aid for changing his situation**
  - User can decide fast what hits of the answer to inspect
Key Challenges

• Efficiency
  – *Offer real-time interaction*

• Scalability
  – *Exploit the available main memory and disk*

• Flexibility
  – *Reduce user’s effort*

*Using very modest hardware!*
Trie-based Index Structures
Trie-based Index Structures

- **Trie** is an ordered tree data structure that is used to store an associative array where the keys are usually strings
  - Looking up a string of $m$ chars has complexity $O(m)$

- **Our approach:**
  - Enrich the trie that is used for autocompletion with the results of the pre-processing steps

```
always in main memory

100x bigger trie
```
Trie Partitioning (1/3)

Always in main memory
Example:

Query log:
apple
api
alpha>alert
basket
bingo
blank
blanket
blue
clown
cowboy
cow

We decide to partition the trie based on the first 2 characters ($k=2$)

- 2 queries start with ap
- 2 queries start with al
- 1 query start with ba
- 3 queries start with bl
- 1 query start with cl
- 2 queries start with co

We decide to store at least 2 queries in each subtrie.

5 subtries are created:
- 1 subtrie for the queries that start with ap (which contains 2 queries)
- 1 subtrie for the queries that start with ba and bi (which contains 4 queries)
- 1 subtrie for the queries that start with co (which contains 2 queries)
- 1 subtrie for the queries that start with al and cl (which contains 3 queries)
- 1 subtrie for the queries that start with bl (which contains 3 queries)
Distributions of Queries to Partitions based on the first $k$ characters

D. Kastrinakis and Y. Tzitzikas
“Advancing query autocompletion services with more and better suggestions”
ICWE 2010
Indexes to External Files

```
root
  c a p
    a m
      o b
        m a p

int: file
int: bytes to skip
int: bytes to read
```

*Random Access Files*

HARD DISK

Always in main memory
Trie Partitioning and Indexes to External Files

```plaintext
sub-trie 1
  c  a  p

sub-trie 2
  a  m  j  o  p

sub-trie 3
  m  a  p

In main memory at request time
int: file
int: bytes to skip
int: bytes to read

HARD DISK
Random Access Files
```
Trie-based Index Structures – **Synopsis**

**SET**  
*(Single Enriched Trie)*

**PET**  
*(Partitioned Enriched Trie)*

**STIE**  
*(Single Trie with Indexes to External files)*

**PTIE**  
*(Partitioned Trie with Indexes to External files)*
Throughput and Caching
The problem:
- A large number of users start typing queries at the same time.
- How does each index approach react?

SET and STIE:
- The trie is loaded only once (at system start-up)
- The number of requests the system can serve depends on the server’s request/session capacity
- **No problem of overloading!**

PET and PTIE:
- Require loading multiple subtries, i.e. the appropriate subtrie for each user’s keystroke
- **The system can get overloaded!**

**SET**: one enriched trie  
**PET**: many enriched subtries  
**STIE**: one trie with indexes  
**PTIE**: many subtries with Indexes
Throughput and Caching (2/3)

• Solution:
  – Keep in memory a number of subtries 
    *(which? the more frequent? the latest?)*

• **Static** Cache
  – Keep in cache the most frequent subtries based on a past log analysis

*The most popular queries do not change very frequently*

*Appear in the 28% of all queries*
• **Dynamic** Cache
  – Start from an empty cache and put in it each requested subtries
  – If the cache is full, replace an existing cached subtrie (e.g. the less frequent) with the new one
  – Periodically refresh the cache by removing the old subtries

• **Hybrid** Cache
  – Combine dynamic and static approach
  – Keep always in memory the most frequent subtries *(static part)*, and keep an amount of memory for loading subtries that are not in the static part *(dynamic part)*

  – *How to partition the available main memory?*
On “Flexible” Recommendations:

1) Tolerate Different Word Orders
2) Tolerate Typos
Relaxing the Word Order

• Motivation:
  – A user starts typing the query “avensis toyota”
  – The trie (or subtrie) contains the query “toyota avensis” but not the query “avensis toyota”
  – After having typed “avensis t”, the query “toyota avensis” is not suggested

• Solution:
  – Load also the suggestions starting from “t” that contain “avensis”
Relaxing the Word Order – Implementation Approaches

- Implementation Approaches:
  
  (A) Check all possible $m!$ permutations (where $m$ is the number of words of user’s input string).

  
  Trie traversals: $m!$  
  Max subtrie loadings: $m$

  (B) Check for queries that start from each word of user’s input and contain at least one of the remaining words.

  Trie traversals: $m$  
  Max subtrie loadings: $m$

  (C) Check for queries that start with the $k$ most frequent (in the query log) words of user’s input and contain at least one of the remaining words ($k < m$).

  Trie traversals: $k$  
  Max subtrie loadings: $k$
Relaxing the Word Order – **Incremental Suggestions**

- **Common case:**
  - *While user is typing a query, the old input is part** (substring) **of the new input (i.e. user has not changed the string that he has already typed)*

<table>
<thead>
<tr>
<th>Input String</th>
<th>Suggestions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial:</strong> toy</td>
<td>toy story, toyota, toyota cars, toyotomy</td>
</tr>
<tr>
<td>Next: toyot</td>
<td>toyota, toyota cars, toyotomy</td>
</tr>
<tr>
<td>Next: toyota c</td>
<td>toyota cars, corolla toyota</td>
</tr>
</tbody>
</table>

1. We just **filter** the last retrieved suggestions according to the new input
2. If user start typing a **new word**:  
   - we first filter the existing suggestions, and then  
   - we search for suggestions that start with only the new word and contain at least one of the first words
On “Flexible” Recommendations:

1) Tolerate Different Word Orders
2) Tolerate Typos
Typo-Tolerant Query Suggestions

• Motivation:
  – A user start typing “merilyn”, but actually he would like to type “marilyn”
  – The trie (or subtrie) contains the queries “marilyn”, “marilyn monroe” and “marilyn manson”, but not any query starting from “merilyn”
  – The user will never get these suggestions!

• Solution:
  – Load also the suggestions that their *beginning substring* is “similar” to the query that user is typing
  – For example, compute the *Edit (Levenshtein) Distance* between user’s input and the *beginning substring* of each full query in the log

*Edit Distance* is the minimum number of edits (insertions, deletions, substitutions) needed to transform one string into the other.
Typo-Tolerant Query Suggestions – **Detect the Active Nodes**

- **Edit Distance Threshold** = \( \text{input length}/3 \)
- User’s input: **meri**
- Edit Distance: **1**

**Suggestions:**
- **cerise**
- **cerium**
- **maria**
- **marilyn**

- We have to visit all nodes of string length \( \leq \text{input length} + \text{Edit Distance Threshold} \)
- The **partitioned** indexes have to load all the subtries!
Typo-Tolerant Query Suggestions – Detect the Active Nodes

Edit Distance Threshold = \textit{input length/3}

- User’s input: \textcolor{red}{meri}
- Edit Distance: \textcolor{red}{1}

**Solution:**
Ignore typo in the 1\textsuperscript{st} character

Suggestions:
- \textcolor{red}{maria}
- \textcolor{red}{marilyn}
Experimental Evaluation

1) of the index structures
2) of various caching schemes
3) of the “flexible” recommendations
Evaluation of the Index Structures (SET, PET, STIE, PTIE)

• Evaluation Aspects:
  – Trie Size to be loaded in main memory
  – Average Retrieval Time
  – Construction and Update Time

• Data Sets
  – 4 query logs of different sizes
    (each one is a subset of a random log sample from a real query log)

<table>
<thead>
<tr>
<th>Num. of log’s queries</th>
<th>Num. of unique queries</th>
<th>Avg num. of words per query</th>
<th>Num. of distinct words</th>
<th>Avg num. of chars per query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>578</td>
<td>2.23</td>
<td>950</td>
<td>15.5</td>
</tr>
<tr>
<td>10,000</td>
<td>5,341</td>
<td>2.3</td>
<td>6,225</td>
<td>16</td>
</tr>
<tr>
<td>20,000</td>
<td>10,518</td>
<td>2.34</td>
<td>10,526</td>
<td>16.2</td>
</tr>
<tr>
<td>40,000</td>
<td>20,184</td>
<td>2.35</td>
<td>17,179</td>
<td>16.2</td>
</tr>
</tbody>
</table>

SET: one enriched trie
PET: many enriched subtries
STIE: one trie with indexes
PTIE: many subtries with indexes
Trie Size to be loaded in main memory (1/2)

- **SET**: one enriched trie
- **PET**: many enriched subtries
- **STIE**: one trie with indexes
- **PTIE**: many subtries with indexes

**PET and PTIE**: 50 entries / subtrie
The size of the proposed index structures is affected only by the size of the query log and in particular by the number of distinct queries.

The size of the dataset/collection does not affect the size of the index.
Average Retrieval Time

**Graphs:**
- Left graph: Time (ms) vs. Implementation Approach (SET, PET, STIE, PTIE)
- Right graph: Time (ms) vs. Number of Queries in Query Log File

**Key Points:*
- **SET:** one enriched trie
- **PET:** many enriched subtries
- **STIE:** one trie with indexes
- **PTIE:** many subtries with indexes

**Notes:**
- PET and PTIE: 50 entries / subtrie
Trade-off

SET: one enriched trie
PET: many enriched subtries
STIE: one trie with indexes
PTIE: many subtries with indexes

Query log of 40,000 queries
PTIE over a very large query log

- Synthetic query log of 1 million queries
- Synthetic precomputed information of 1 terabyte
- We measure the average time for retrieving:
  - The suggestions
  - The results of the top suggestion
  - The supplementary information of the top suggestion
for a random input string without using any cache

Average Retrieval Time ≈ 135ms

PTIE: many subtries with indexes
Selecting the Right Index

• Rules:

1. If the entire **SET** fits in memory, then this is the faster choice
2. If **SET** does not fit in memory then the next choice to follow is **STIE**
3. If neither **SET** nor **STIE** fit in memory then **PTIE** approach has to be used

• **PTIE** is the more scalable approach, since:
  - It can be adopted even if the available main memory has very small size
  - It’s very efficient with low retrieval time
  - It can be used even with very large query log and very large amounts of precomputed information
Trie Construction

• Main Tasks:
  1. Analyze the query log
  2. Execute each distinct query and get the required information (*top results, cluster label tree, etc.*)
  3. Create the (sub)trie file(s)

• Task 2 is the most time consuming, requiring about 1 second per query (in our setting)

<table>
<thead>
<tr>
<th>Number of log’s queries</th>
<th>Query log file analysis time (ms)</th>
<th>Results and clusters retrieval time (ms)</th>
<th>Trie creation time (ms)</th>
<th>Total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>4</td>
<td>592,515</td>
<td>1,259</td>
<td>594</td>
</tr>
<tr>
<td>10,000</td>
<td>9</td>
<td>5,415,150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20,000</td>
<td>12</td>
<td>10,802,970</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40,000</td>
<td>16</td>
<td>21,105,780</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Trie construction has to be done once, but periodically

Update the trie (or subtries) incrementally
Incremental Trie Update

1) Recent query log:
- **jam**: 22/05/2011 18:46:01 EEST 127.0.0.1
- **man**: 23/05/2011 12:30:22 EEST 127.0.0.1

2) Create trie of the recent query log:
- Jam
- Man

3) Full trie:
- Cap
- Man
- Job
- Map

4) Create new full trie:
- String: cap
  - String: <cluster label tree>
  - String: <results' first page>
  - Date: 12/02/2011 11:20:03
  - Integer: 5
- String: jam
  - String: <cluster label tree>
  - String: <results' first page>
  - Date: 22/08/2011 18:46:01
  - Integer: 1
- String: man
  - String: <cluster label tree>
  - String: <results' first page>
  - Date: 23/08/2011 12:30:22
  - Integer: 1
- String: job
  - String: <cluster label tree>
  - String: <results' first page>
  - Date: 22/05/2011 18:46:01
  - Integer: 4
- String: map
  - String: <cluster label tree>
  - String: <results' first page>
  - Date: 13/11/2010 08:15:20
  - Integer: 7
Experimental Evaluation

1) of the index structures
2) of various caching schemes
3) of the “flexible” recommendations
Evaluation of Caching Schemes

• Comparative evaluation of the following schemes:
  1. Full Static cache
  2. Full Dynamic cache
  3. Hybrid (static: 30%, dynamic: 70%)
  4. Hybrid (static: 50%, dynamic: 50%)
  5. Hybrid (static: 70%, dynamic: 30%)
  6. No cache
Caching Schemes – **Evaluation Criteria**

1. **Number of Served Queries**
   - Number of queries that are served fast
     *(the requested subtrie is in cache)*
   - Number of queries that are served with delay
     *(the requested subtrie is not in cache and the system has to load it)*
   - Number of queries that cannot be served
     *(the requested subtrie is not in cache and the cache is full and in use)*

2. **Average Retrieval Time**
   - The average time to retrieve all the information
Evaluation of Caching Schemes – Data Set and Setup

• Synthetic query log of 1 million distinct queries
  – 344 subtries of 615 MB total size, using PTIE

• 10,000 random queries (selected from the query log)

• Query rate = 8 queries/second

• Memory Capacity = 60 subtries
  – 17.4% of all subtries can fit in main memory at the same time

• Time threshold = 10 sec.
  – The time that a subtrie is considered in use
  – 10*8=80 queries have to be served at the same time

• Static Cache: We load the more frequent subtries after a query log analysis
Caching Schemes – **Served Queries**

**No Cache:** The system can serve up to 60 requests at the same time, all with delay, i.e. 20 of the 80 requests (25%) will not be served.
Caching Schemes – Average Retrieval Time

The graph shows the average retrieval times for different cache memory partition policies. The policies include Dynamic (70% static, 30% dynamic), 30% static and 70% dynamic, 50% static and 50% dynamic, and 70% static and 30% dynamic. The graph also includes Static and No Cache as comparison points.

The graph highlights that the 70% static and 30% dynamic policy shows a 25% speedup compared to other policies.
Experimental Evaluation

1) of the index structures
2) of various caching schemes
3) of the “flexible” recommendations
Evaluation of the Flexible Recommendations

• Evaluation Criteria
  – Retrieval Time
    • STIE: Synthetic log of 200,000 queries
    • PTIE: Synthetic log of 1 million queries
    • 1,000 random queries from the log
    • No incremental suggestions
    • No caching scheme (for PTIE)
  – Number of Additional Suggestions
    • Real query log with 22,251 distinct queries
Retrieval Time – **Word-Order Independent Suggestions**

- Time for retrieving suggestions that *start from a word and contain at least one of the remaining words*
- For each random query, we keep only the first 2 characters from the last word

<table>
<thead>
<tr>
<th>Query length</th>
<th>STIE</th>
<th>PTIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-word queries</td>
<td>29 ms</td>
<td>182 ms</td>
</tr>
<tr>
<td>4-word queries</td>
<td>37 ms</td>
<td>492 ms</td>
</tr>
<tr>
<td>8-word queries</td>
<td>48 ms</td>
<td>829 ms</td>
</tr>
<tr>
<td>12-word queries</td>
<td>58 ms</td>
<td>1,054 ms</td>
</tr>
</tbody>
</table>

**STIE**: one trie with indexes  
**PTIE**: many subtries with indexes
Retrieval Time – *Typo-Tolerant Suggestions*

- **Time of STIE** for retrieving the suggestions

<table>
<thead>
<tr>
<th>Query length</th>
<th>Detect the active nodes</th>
<th>Ignoring typo in the 1&lt;sup&gt;st&lt;/sup&gt; char</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-char queries</td>
<td>96 ms</td>
<td>28 ms</td>
</tr>
<tr>
<td>8-char queries</td>
<td>142 ms</td>
<td>39 ms</td>
</tr>
<tr>
<td>12-char queries</td>
<td>225 ms</td>
<td>36 ms</td>
</tr>
<tr>
<td>16-char queries</td>
<td>305 ms</td>
<td>32 ms</td>
</tr>
</tbody>
</table>

- **PTIE** must offer this functionality:
  - Only by ignoring typo in the 1<sup>st</sup> character
  - Only for the subtries that lie in the cache (in case trie partitioning is not based on the first character)

---

**STIE**: one trie with indexes

**PTIE**: many subtries with indexes
• **Word-Order Independent Suggestions**

<table>
<thead>
<tr>
<th>Query length</th>
<th>Having typed the first 2 chars of the last word</th>
<th>Having typed the first 3 chars of the last word</th>
<th>Having typed the first 4 chars of the last word</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-word queries</td>
<td>1.6</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>3-word queries</td>
<td>10.1</td>
<td>4.7</td>
<td>4</td>
</tr>
<tr>
<td>4-word queries</td>
<td>20.9</td>
<td>13.2</td>
<td>12.3</td>
</tr>
</tbody>
</table>
### Number of Additional Suggestions (2/2)

- **Typo-tolerant Suggestions**

<table>
<thead>
<tr>
<th>Query length</th>
<th>Having typed the first 4 chars (edit distance = 1)</th>
<th>Having typed the first 8 chars (edit distance = 2)</th>
<th>Having typed the first 12 chars (edit distance = 4)</th>
<th>Having typed the first 16 chars (edit distance = 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect the active nodes</td>
<td>71.4</td>
<td>7.3</td>
<td>7.1</td>
<td>3.3</td>
</tr>
<tr>
<td>Ignore typo in the 1st character</td>
<td>48.6</td>
<td>6</td>
<td>5.7</td>
<td>2.8</td>
</tr>
</tbody>
</table>
Server’s Benefits
Benefits for the Server’s Side

- **Less incoming queries** which are not really useful for the end users
- **Reduced computational cost** per received query
- **Less monetary cost** (at a meta-search level)
- **Less network connections**

*In particular, the only real price to pay is actually the space required for storing the precomputed information*
Conclusion and Further Research
Conclusion

- **IOS**: a *search-as-you-type* functionality that predicts our search and shows results and *supplementary information* before finish typing

1) With a **partitioned trie-based index structure** we can efficiently support recommendations for *millions* of distinct queries and *terabytes* of precomputed information

2) An **hybrid (70% static/30% dynamic) caching scheme** seems to be the more appropriate, yielding about *80% better throughput* and *25% speedup*

3) Tolerating **typos** and **different word orders** reduces user’s effort and increases the exploitation of the precomputed information and the number of suggestions

- **IOS** is also **beneficial** for the server’s side
Further Research

• Analyze how exactly users exploit the precomputed information that appear instantly
  – *Very fast eye-tracking equipment*
  – *Methods for analyzing the gathered information*
  – *Where and how to display the recommended information?*

• Personalized recommendations
  – *E.g. according to the collaborative approach*
Thank you!

Questions?

Running prototypes:  http://www.ics.forth.gr/isl/ios

More information:

P. Fafalios and Y. Tzitzikas,
“Exploiting Available Memory and Disk for Scalable Instant Overview Search”,
12th International Conference on Web Information System Engineering, WISE’11, Sydney, Australia, October 2011

P. Fafalios, I. Kitsos and Y. Tzitzikas,
“Scalable, Flexible and Generic Instant Overview Search”,
Demo Paper, 21st International Conference on World Wide Web, WWW’12, Lyon, France, April 2012
(It was presented also at the 11th Hellenic Data Management Symposium (HDMS'12), Chania, Greece, June 2012)

P. Fafalios, I. Kitsos, Y. Marketakis, C. Baldassarre, M. Salampasis and Y. Tzitzikas,
“Web Searching with Entity Mining at Query Time”,
5th Information Retrieval Facility Conference, IRFC’12, Vienna, July 2012.