Indexes and Algorithms for Scalable and Flexible Instant Overview Search

Pavlos Fafalios

Master's Thesis

July 2012

Supervisor: Yannis Tzitzikas

FORTH-ICS
Information Systems Laboratory

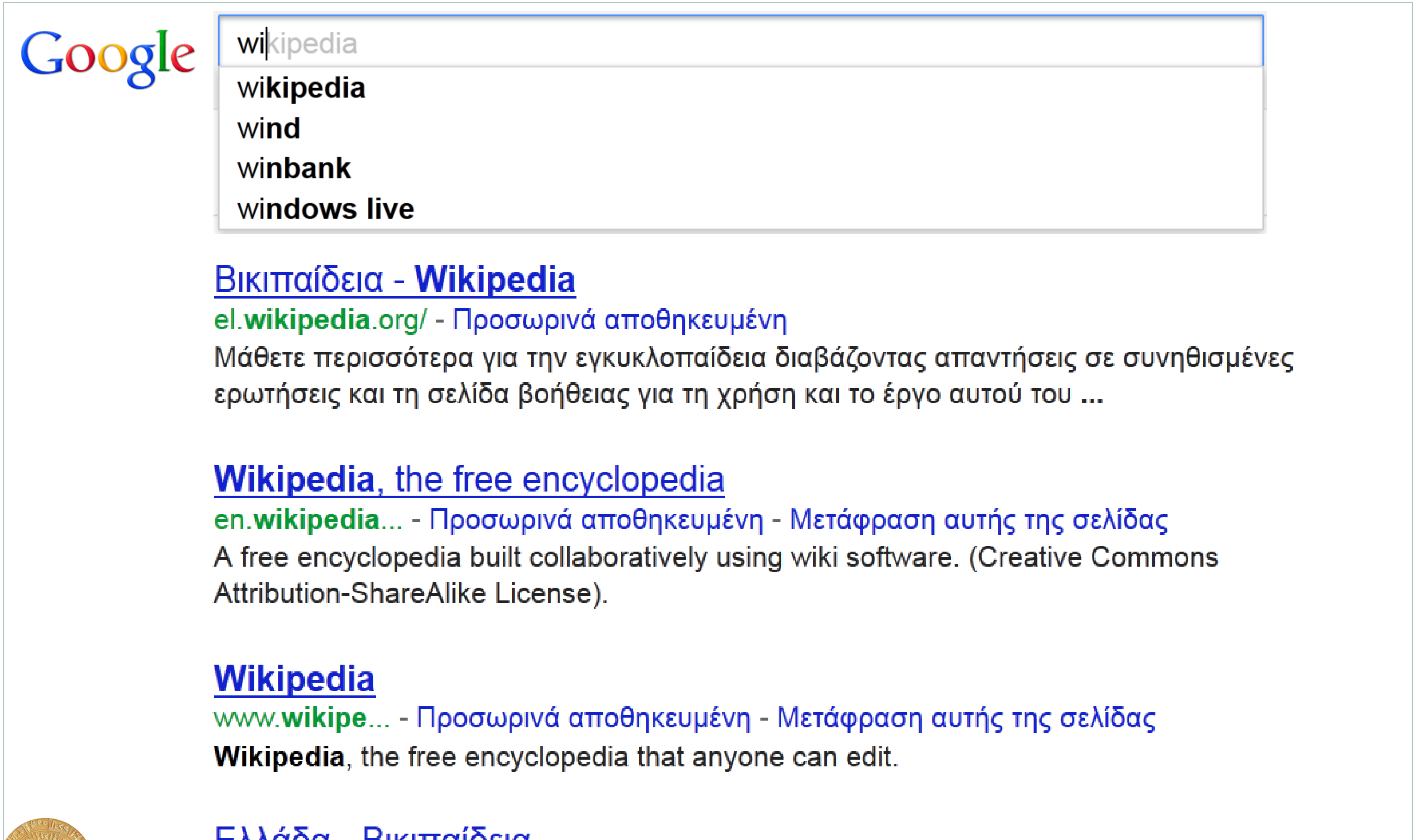
University of Crete
Computer Science Department

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)**

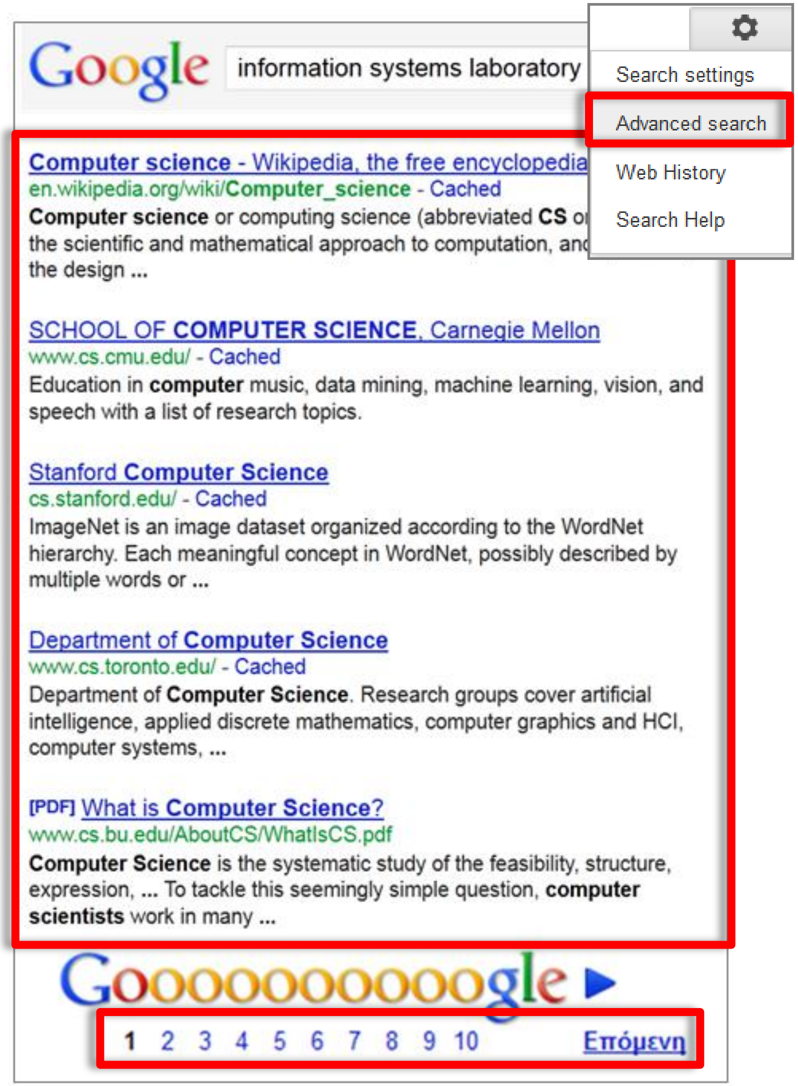
Introduction

What is Instant Search?

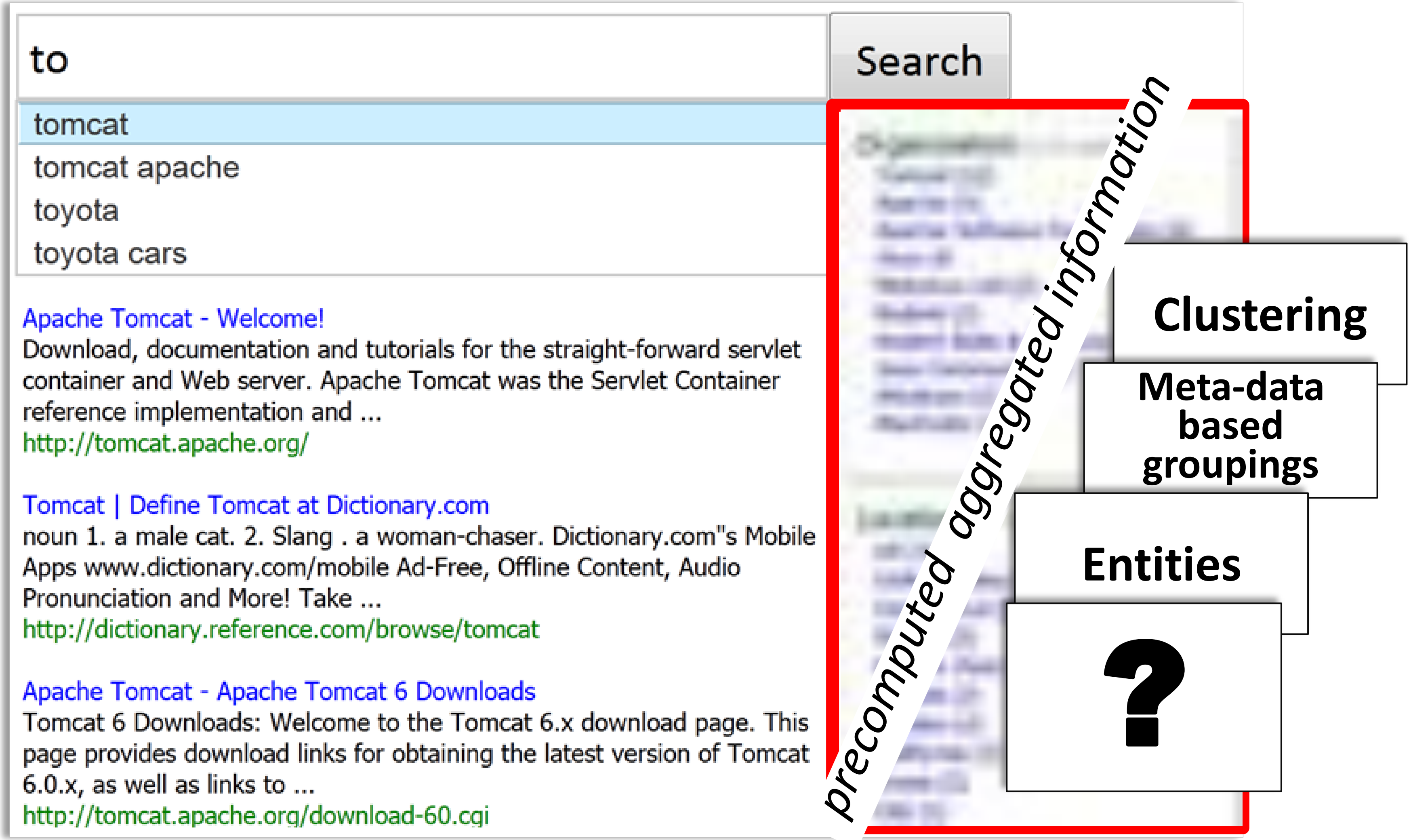


The problem

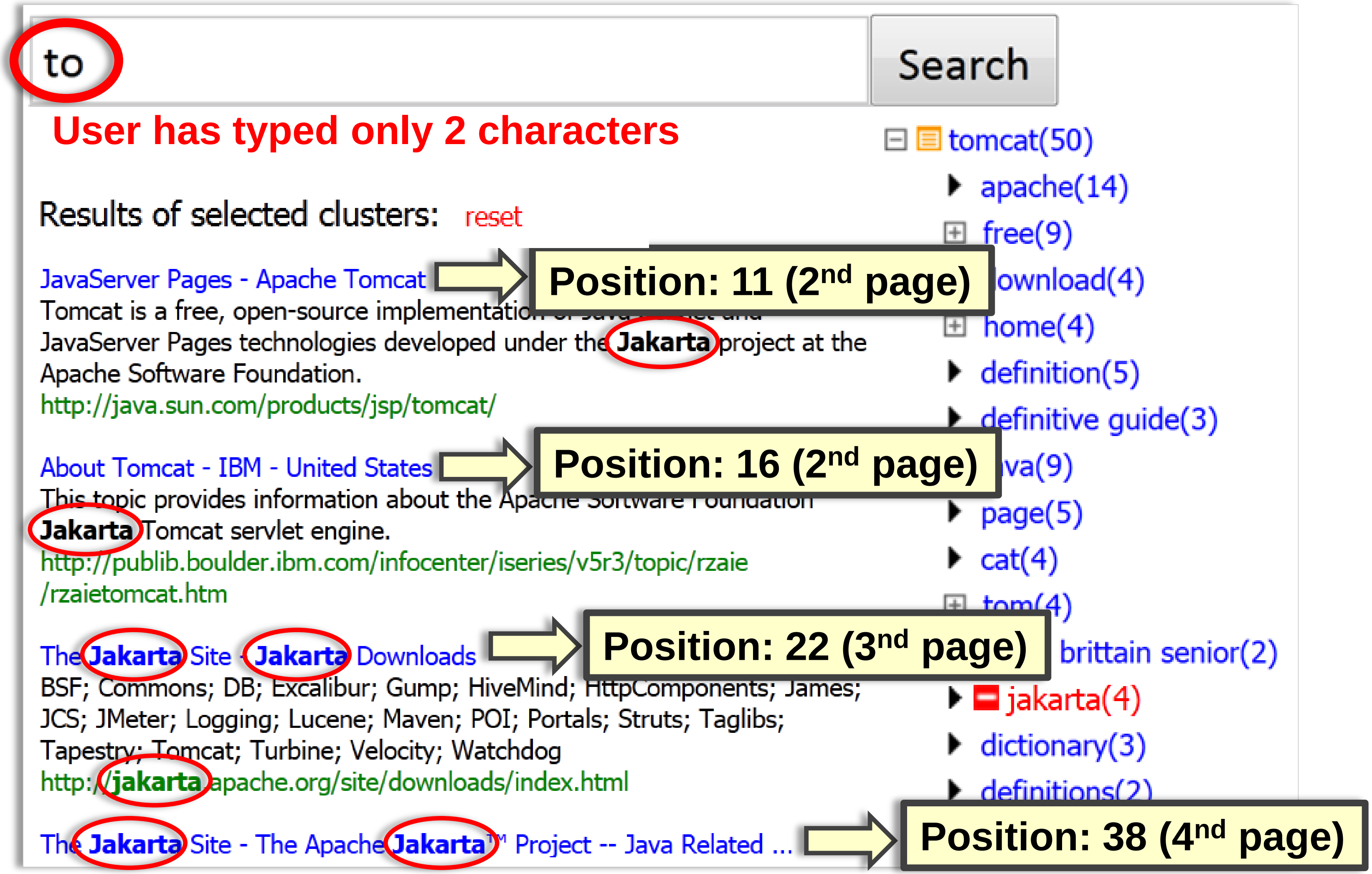
- Most web search engines return a ranked list of results
- Users have to explore the answer linearly
- In practice, users tend to look only at the first page of results, missing useful hits
- Users rarely exploit the available metadata during their searches *(advanced search)*
- In case user has limited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



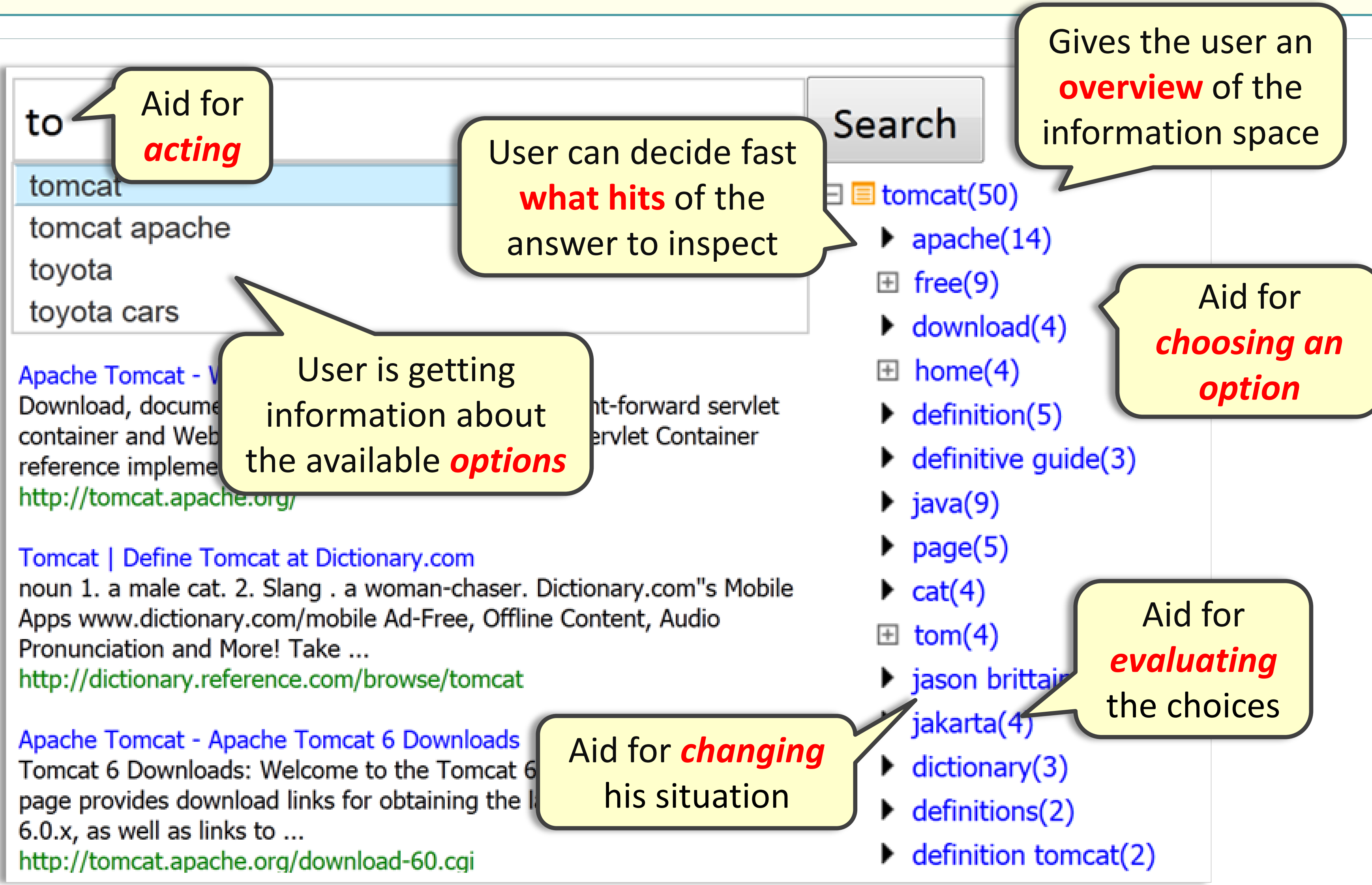
What is **I**nstant **O**verview **S**earch (**IOS**)



What is **I**nstant **O**verview **S**earch (**IOS**)



IOS from a **Decision Making** point of view



Key Challenges

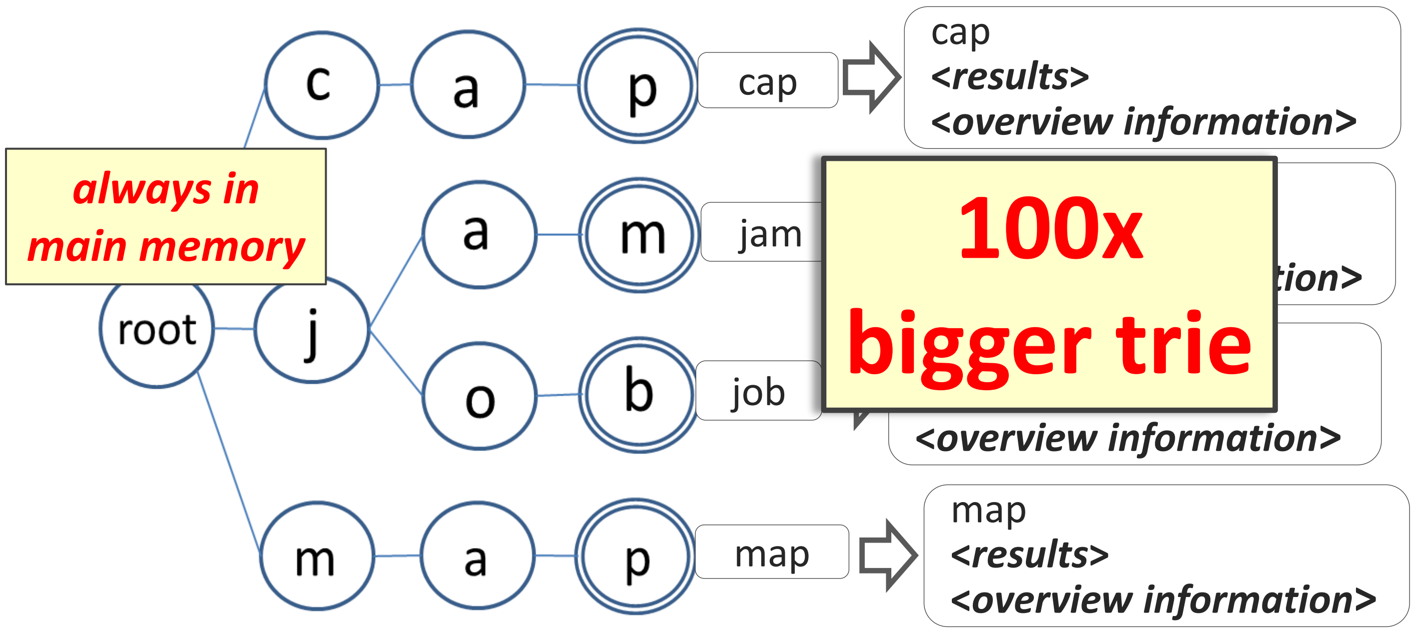
- Efficiency
  - *Offer real-time interaction*
- Scalability
  - *Exploit the available main memory and disk*
- Flexibility
  - *Reduce user's effort*

Using very modest hardware!

Trie-based Index Structures

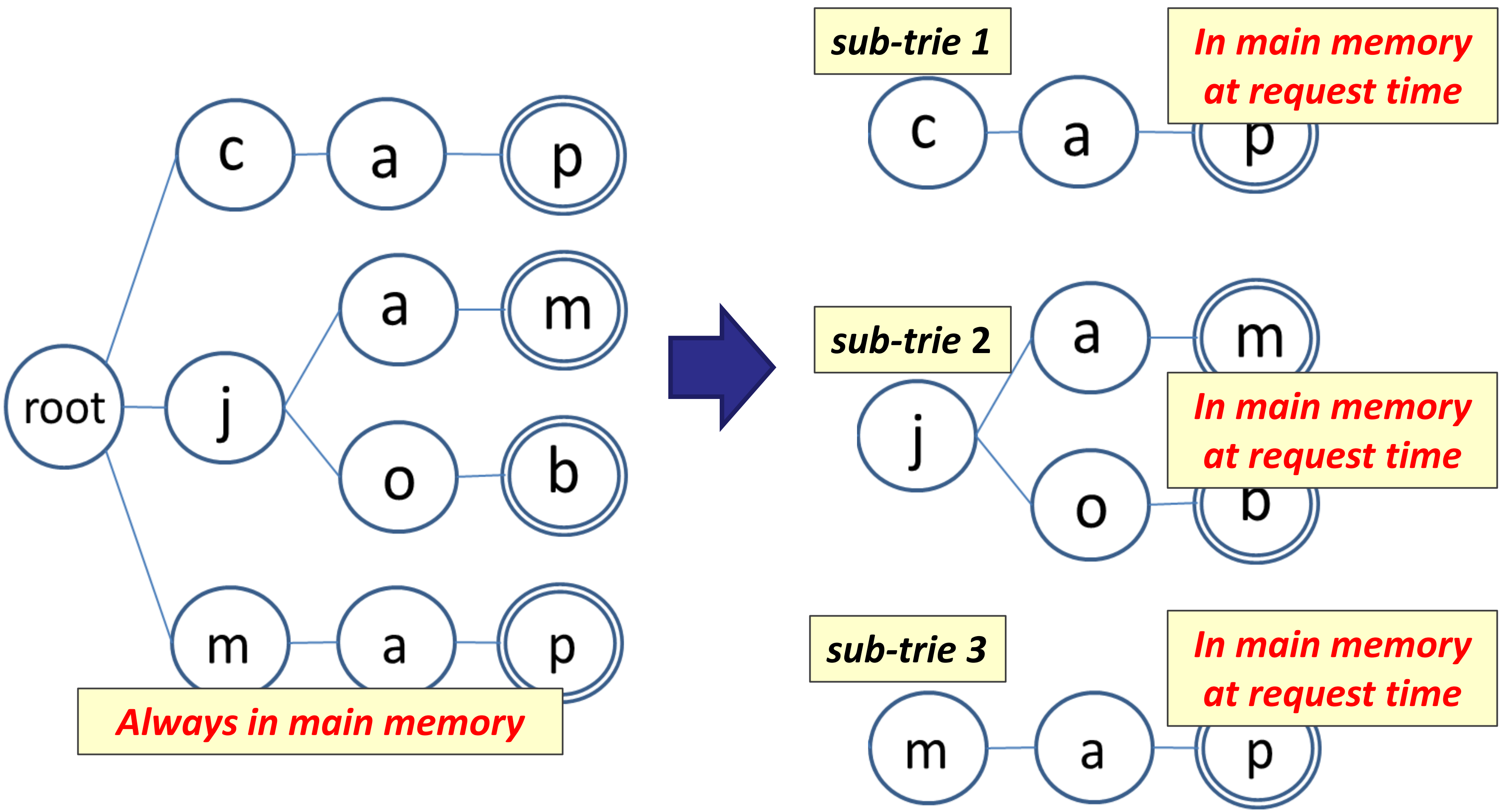
Trie-based Index Structures

- ***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

Trie Partitioning (1/3)



Trie Partitioning (2/3)

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**

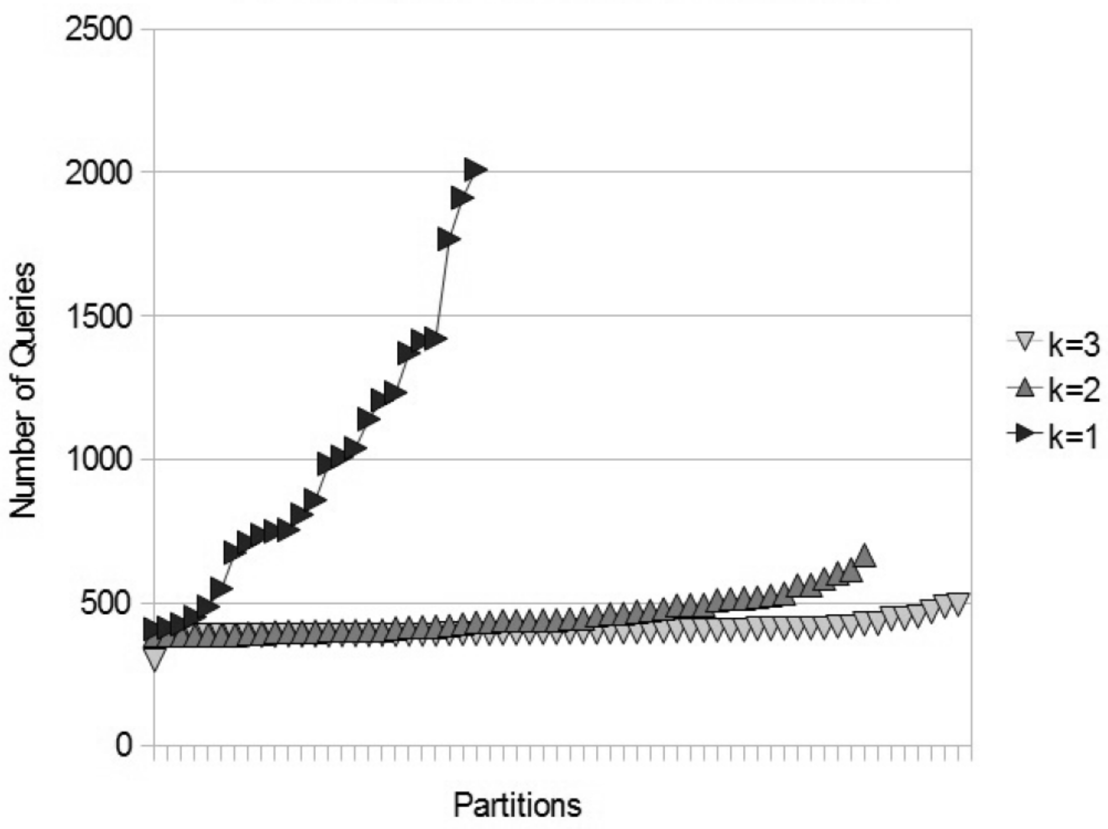
We decide to store at least 2 queries in each subtrie.

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)

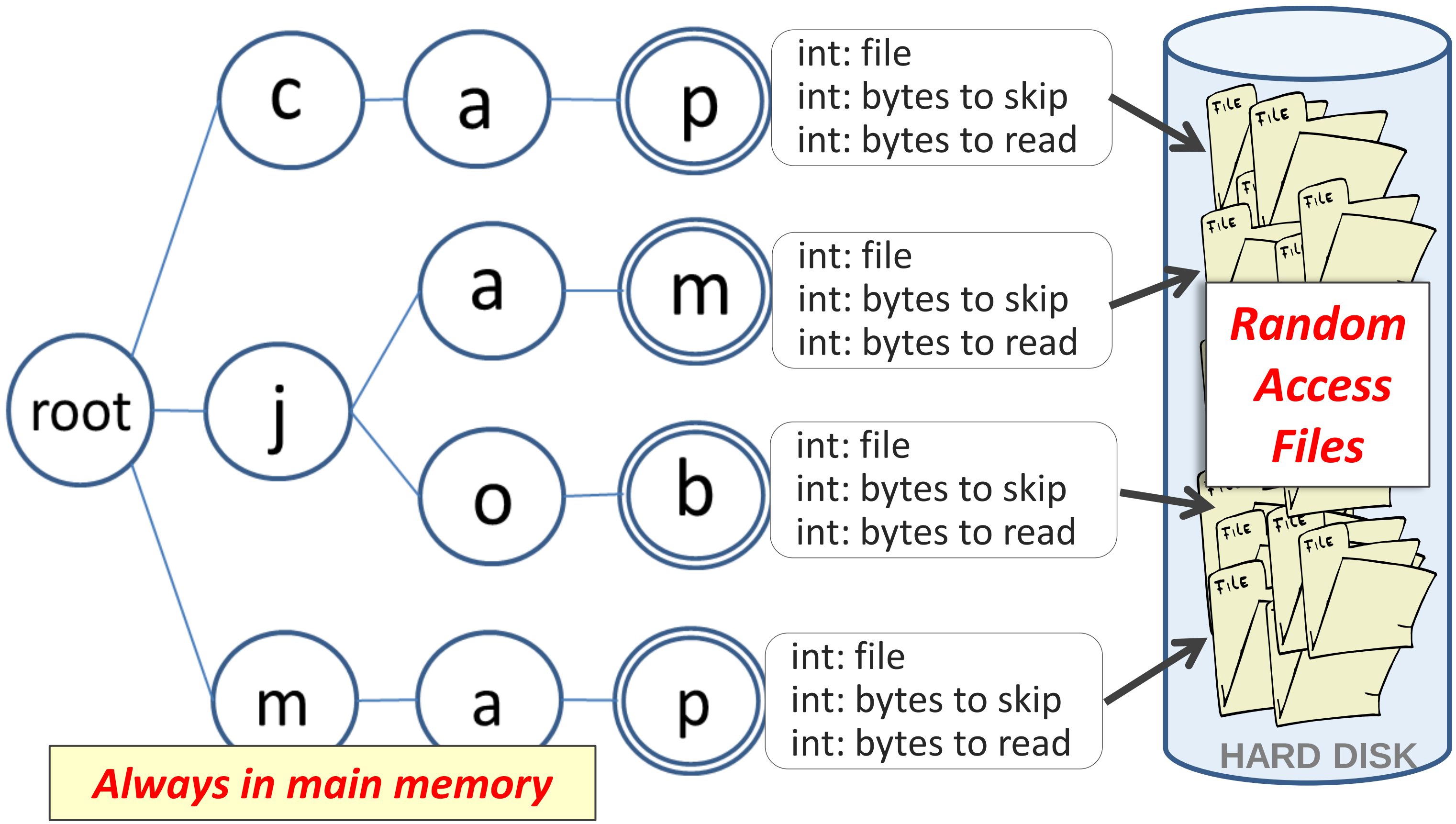
Trie Partitioning (3/3)

Distributions of Queries to Partitions based on the **first *k*** characters

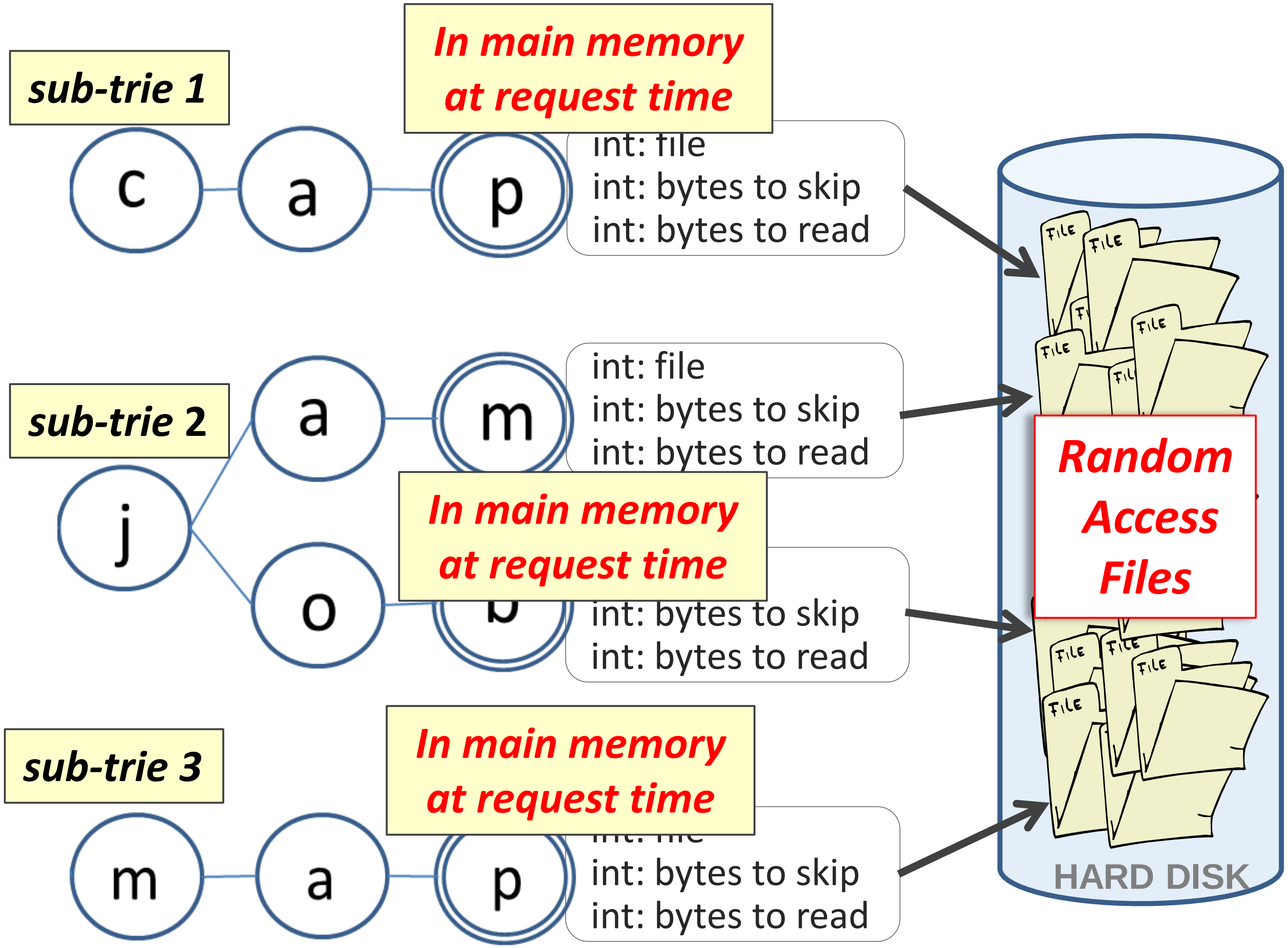


D. Kastrinakis and Y. Tzitzikas
"Advancing query autocompletion services with more and better suggestions"
ICWE 2010

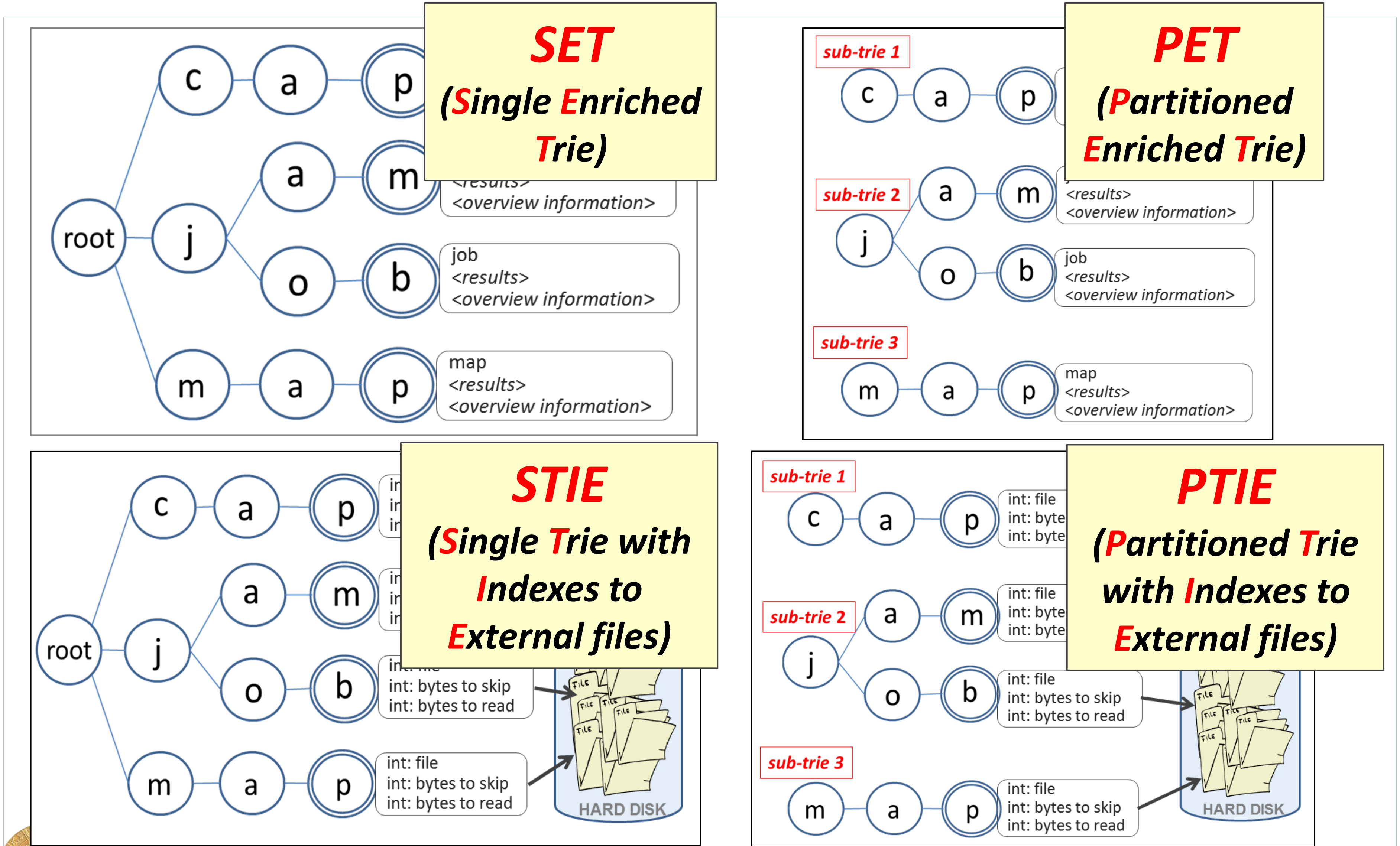
Indexes to External Files



Trie Partitioning and Indexes to External Files



Trie-based Index Structures – **Synopsis**



Throughput and Caching

Throughput and Caching (1/3)

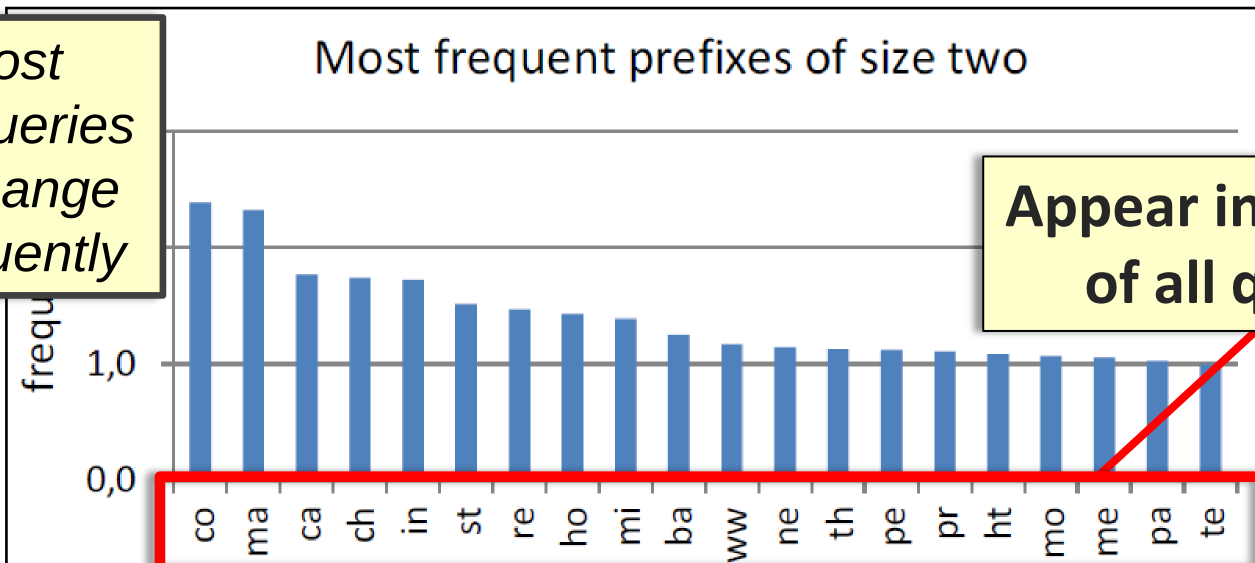
- The problem:
  - A large number of users start typing queries at the same time.
  - How does each index approach react?
- SET and STIE:
  - The trie is loaded only once (at system start-up)
  - The number of requests the system can serve depends on the server's request/session capacity
  - **No problem 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

Query log of 40,000 queries

Throughput and Caching (3/3)

- **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

Catch emerging temporal trends

- **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 start typing the query ***"avensis toyota"***
  - The trie (or subtrie) contains the query ***"toyota avensis"*** but not the query ***"avensis toyota"***
  - After having type **"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 <u>each word</u> 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*** <u>most frequent</u> (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)*

	Input String	**Suggestions**
Initial:	toy	**toy** story, **toy**ota, **toy**ota cars, **toy**otomy
Next:	**toyot**	**toyot**a, **toyot**a cars, **toyot**omy
Next:	**toyot**a c	**toyota c**ars, corolla **toyota**

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

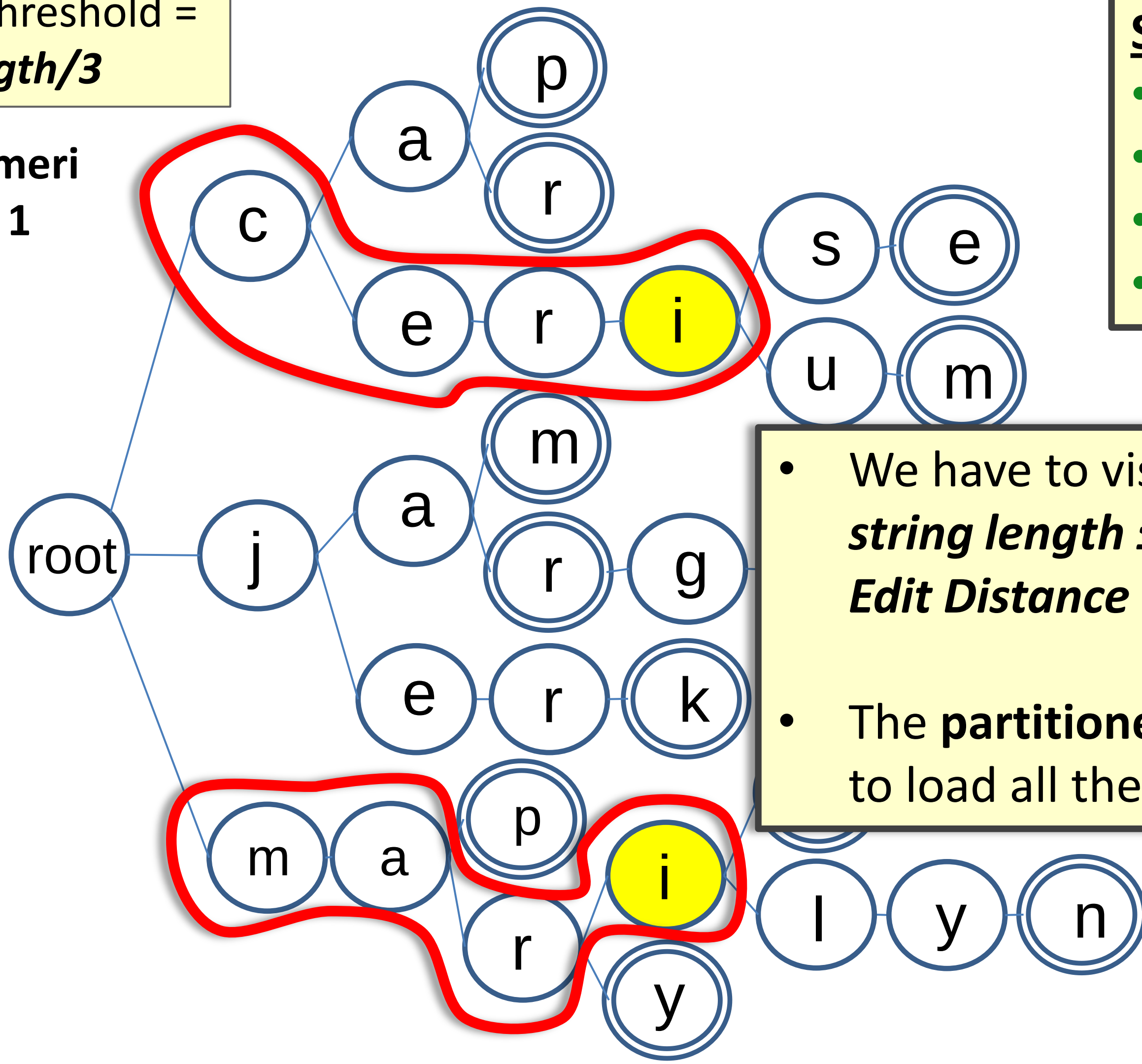
- Motivation:
  - A user start typing *"merilyn",* but actually he would like to type *"marilyn"*
  - The trie (or subtrie) contains the queries "*marilyn*", *"marilyn monroe"* and "m*a*rilyn manson", but not any query starting from *"merilyn"*
  - The user will never get these suggestions!
- Solution:
  - 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 =
input length/3

- User’s input: **meri**
- Edit Distance: **1**



Suggestions:

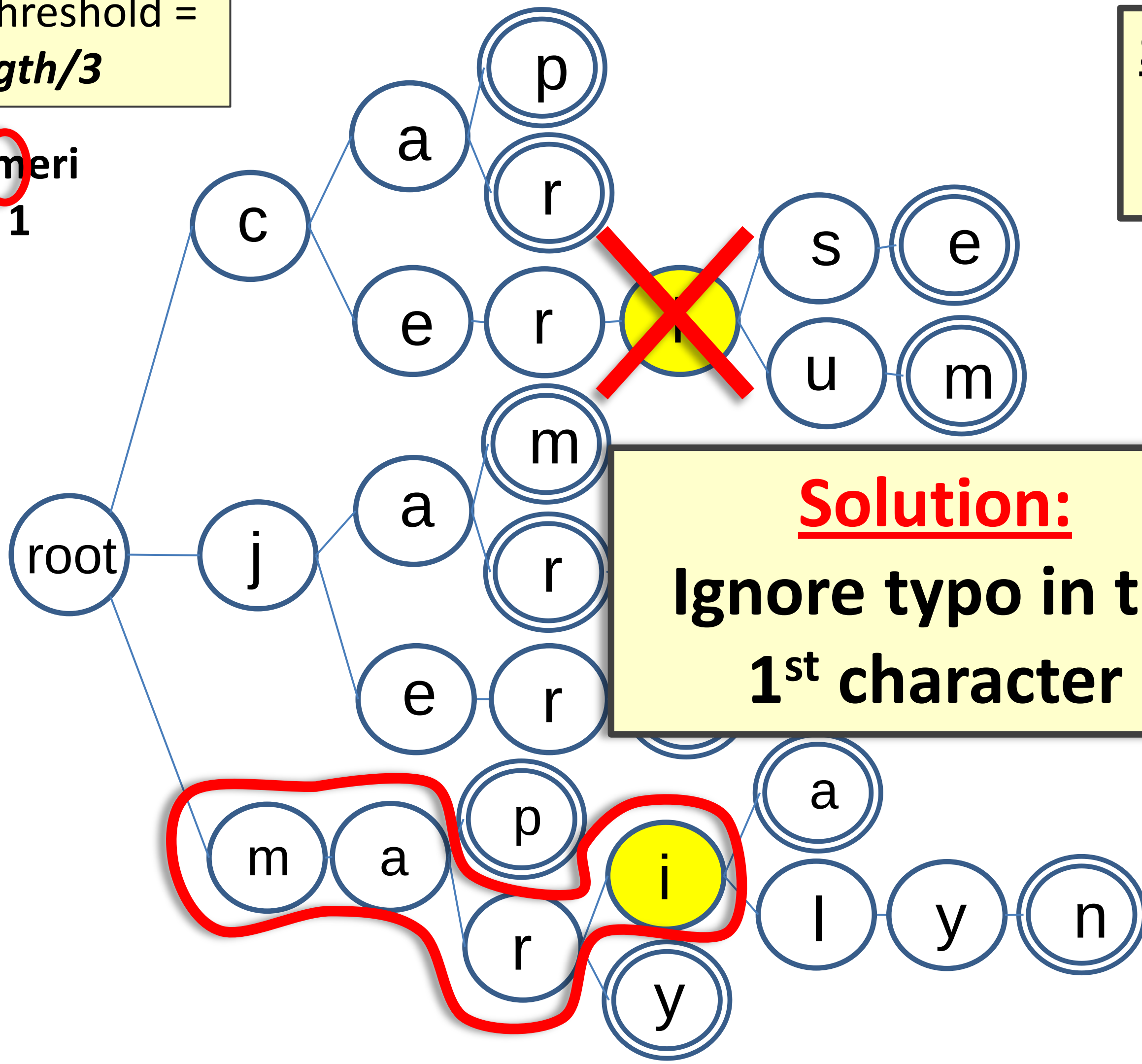
- *cerise*
- *cerium*
- *maria*
- *marilyn*

- We have to visit all nodes of ***string length ≤ input length + Edit Distance Threshold***
- The **partitioned** indexes have to load all the subtries!

Typo-Tolerant Query Suggestions – **Detect the Active Nodes**

Edit Distance Threshold = ***input length/3***

- User's input: **meri**
- Edit Distance: **1**



Suggestions:

- *maria*
- *marilyn*

Solution:

Ignore typo in the 1st character

Experimental Evaluation

1) of the index structures

2) of various caching schemes

3) of the “flexible” recommendations

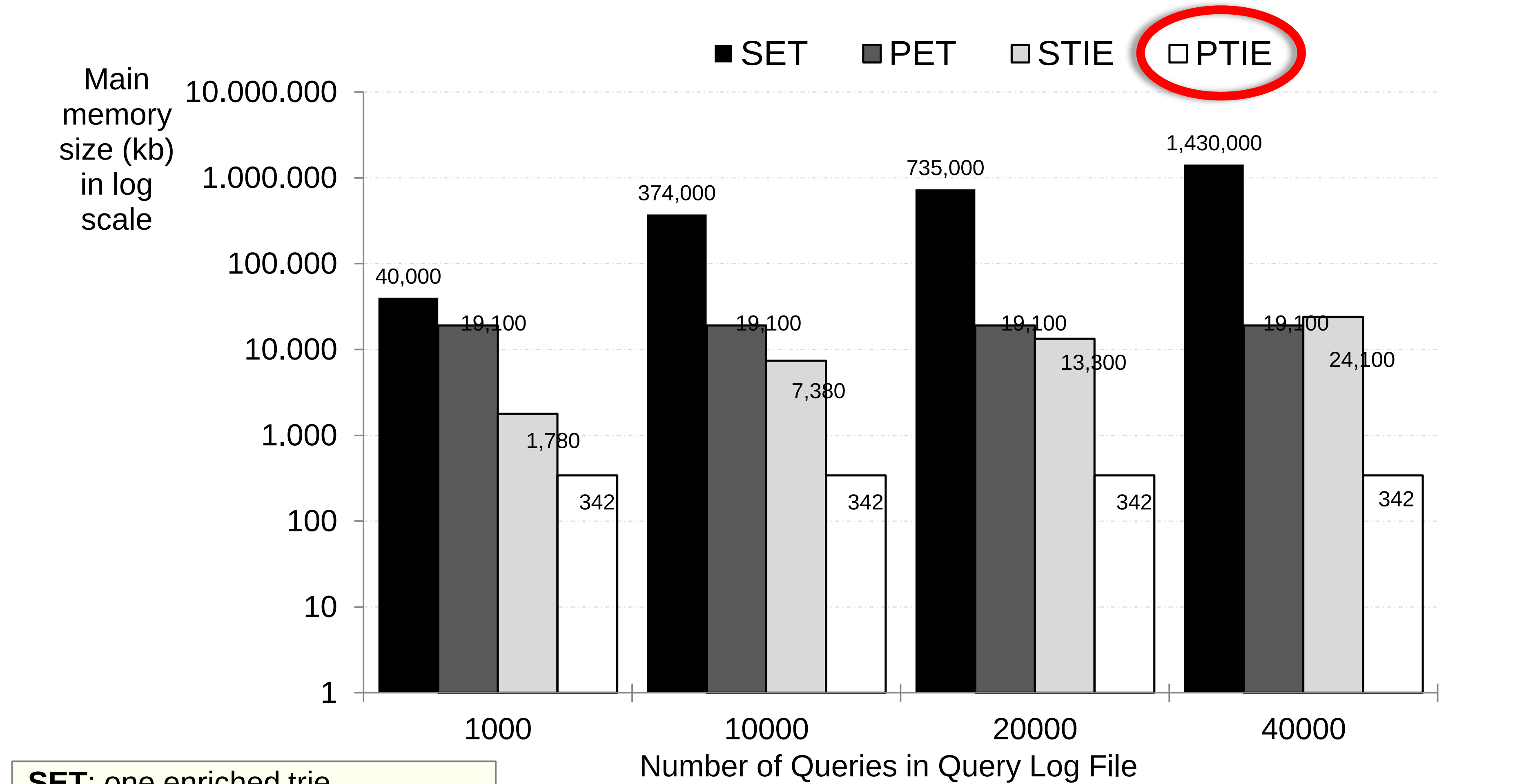
Evaluation of the Index Structures (**SET**, **PET**, **STIE**, **PTIE**)

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

- 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)

Num. of log's queries	Num. of unique queries	Avg num. of words per query	Num. of distinct words	Avg num. of chars per query
1,000	578	2.23	950	15.5
10,000	5,341	2.3	6,225	16
20,000	10,518	2.34	10,526	16.2
40,000	20,184	2.35	17,179	16.2

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

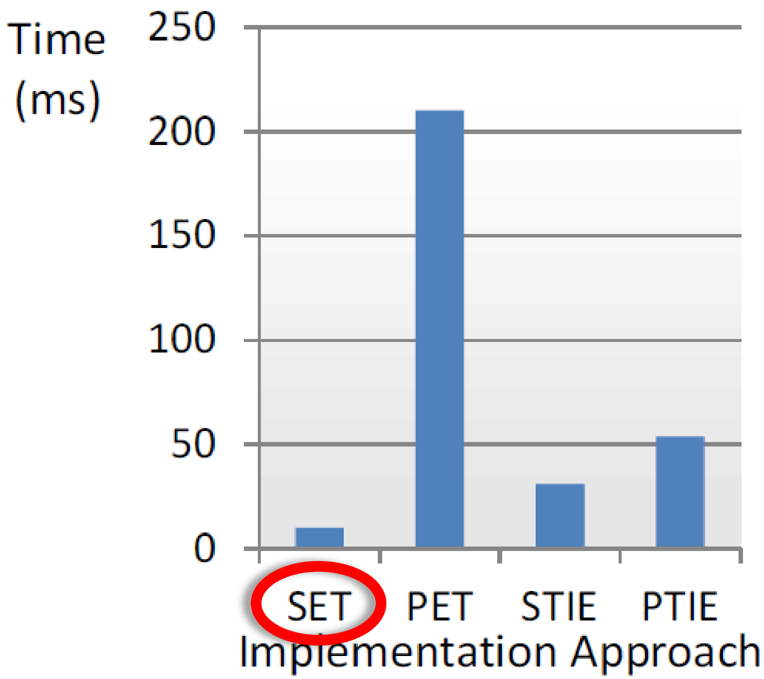
PET and PTIE: 50 entries / subtrie

Trie Size to be loaded in main memory (2/2)

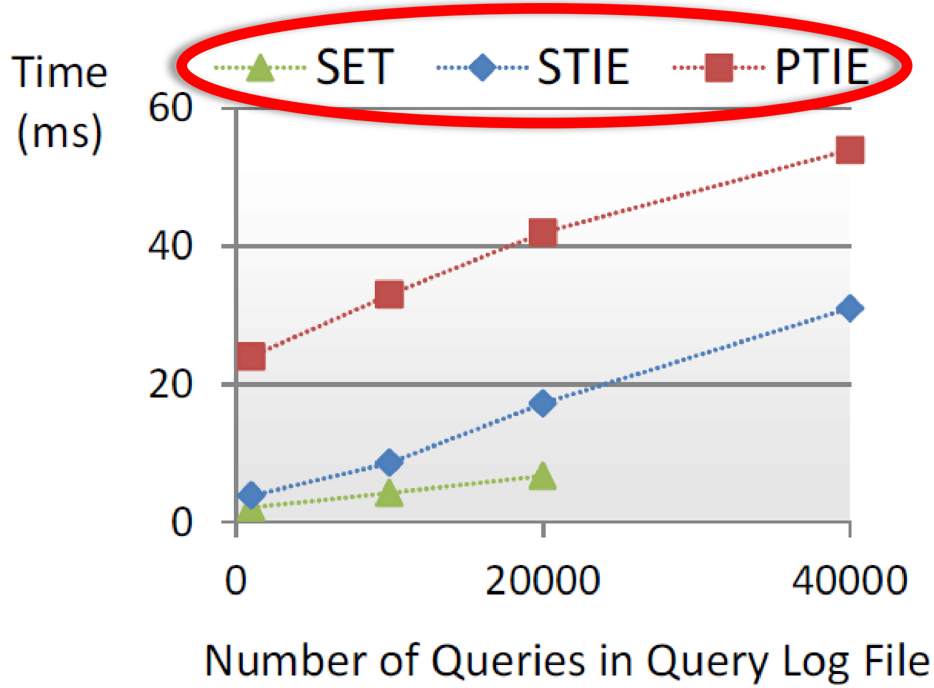
- The size of the proposed index structures is affected **only** by the size of the query log and in particular by the **number of distinct queries**.

- The size of the dataset/collection does **not** affect the size of the index.

Average Retrieval Time



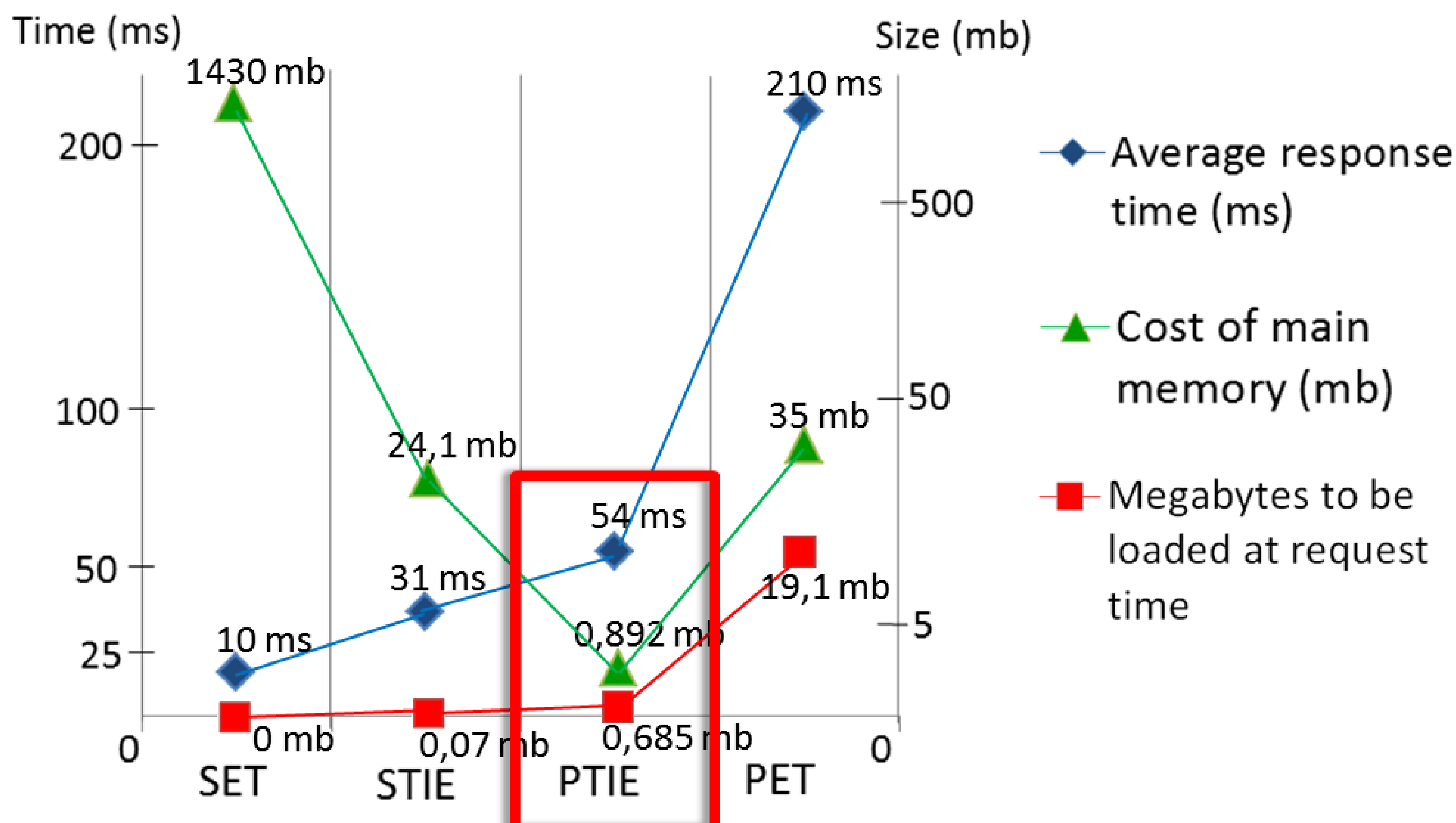
(Query log of 40,000 queries)



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

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

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

- 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

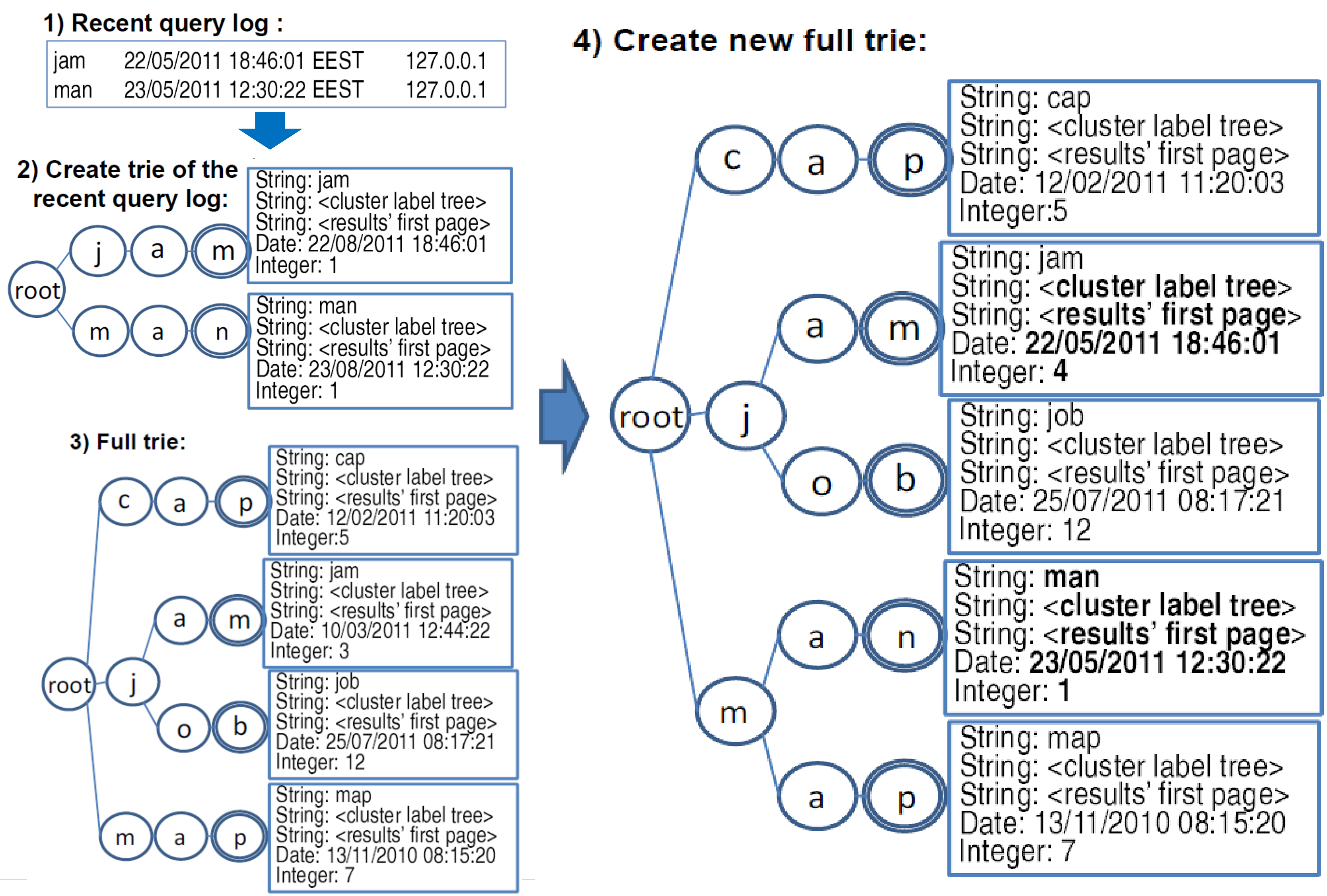
Trie construction has to be done once, but periodically

- 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)

Number of log's queries	Query log file analysis time (ms)	Results and clusters retrieval time (ms)	Trie creation time (ms)	Total time (sec)
1,000	4	592,515	1,259	594
10,000	9	5,415,150		
20,000	12	10,802,970		
40,000	16	21,105,780		

Update the trie (or subtries) incrementally

Incremental Trie Update



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

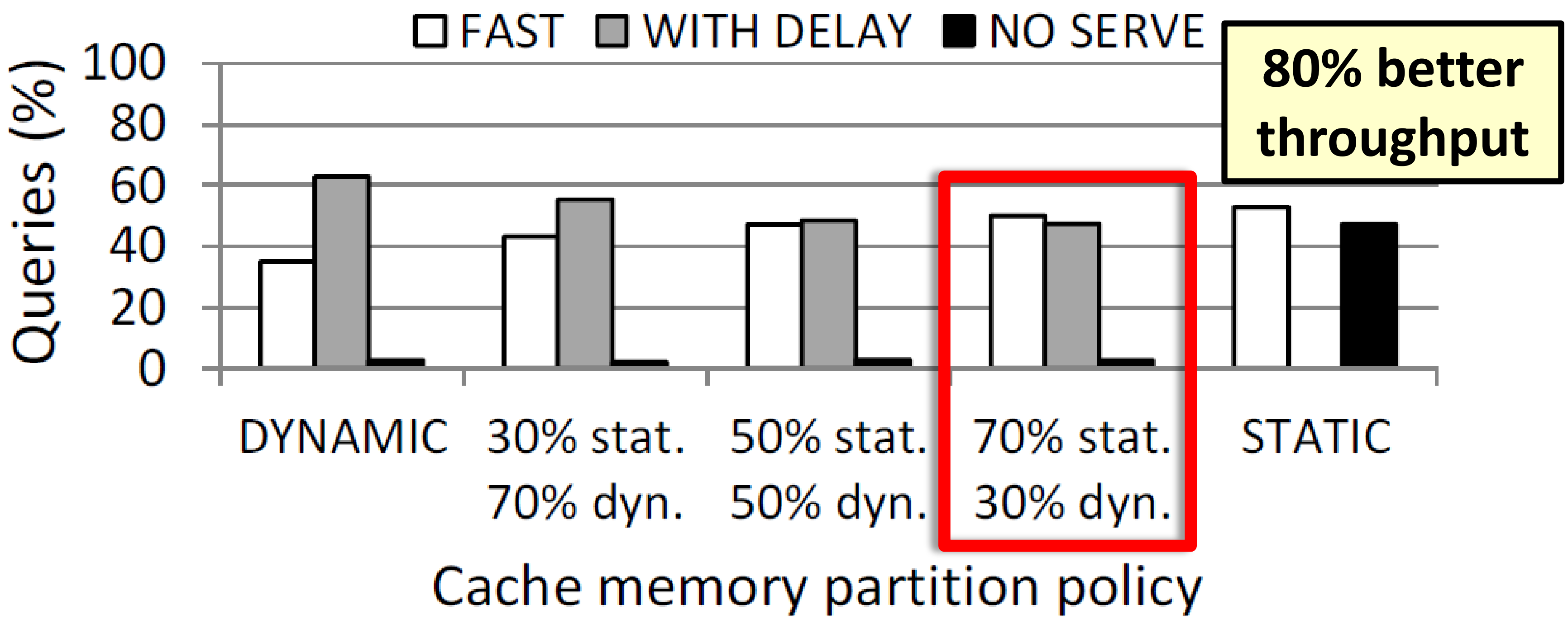
Evaluation of Caching Schemes – Data Set and Setup

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

PTIE: many subtries with indexes

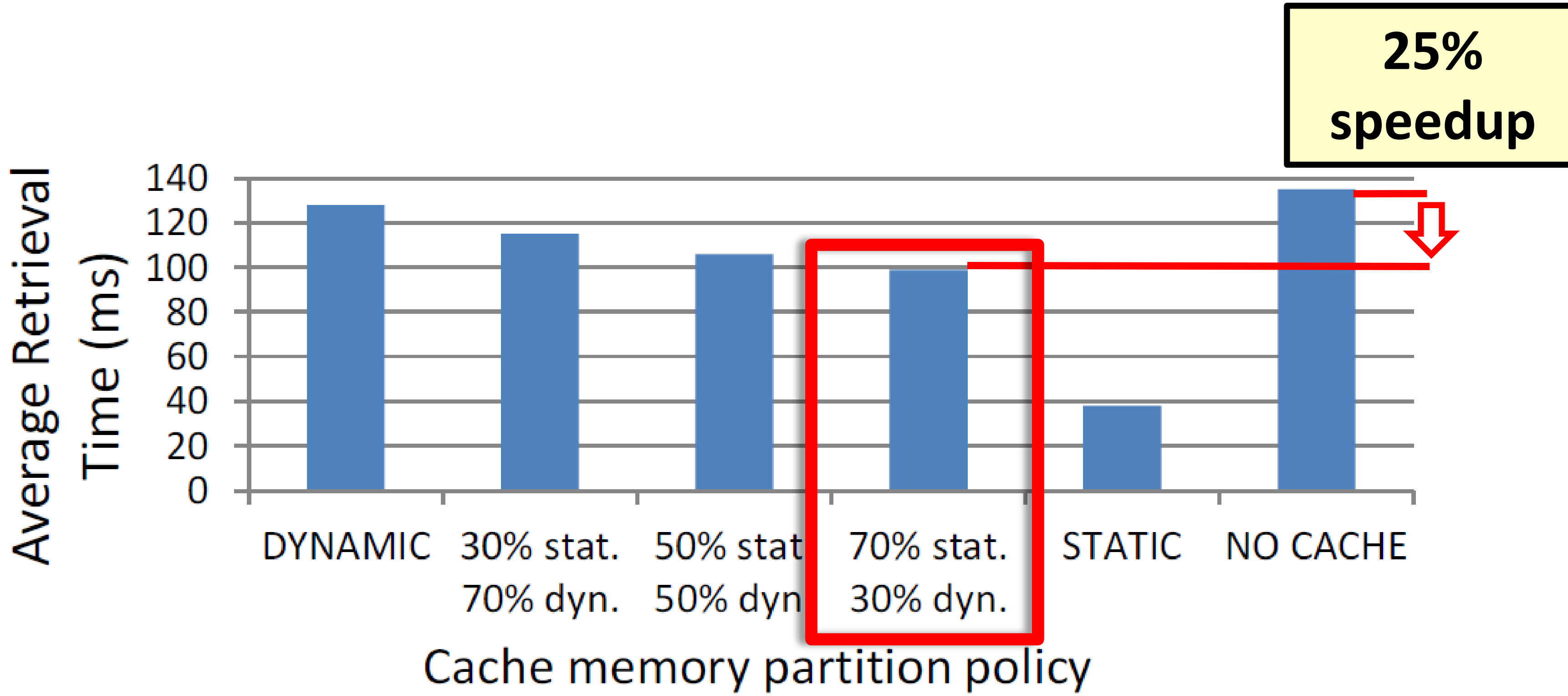
- *10,000* random queries (selected from the query log)
- Query rate = *8 queries/second*
- Memory Capacity = *60 subtries*
  - 17.4% of all subtries can fit in main memory at the same time
- Time threshold = *10 sec.*
  - The time that a subtrie is considered in use
  - 10*8=**80 queries** have to be 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**



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

STIE: one trie with indexes
PTIE: many subtries with indexes

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

Query length	STIE	PTIE
2-word queries	29 ms	182 ms
4-word queries	37 ms	492 ms
8-word queries	48 ms	829 ms
12-word queries	58 ms	1,054 ms

STIE: one trie with indexes
PTIE: many subtries with indexes

Retrieval Time – **Typo-Tolerant Suggestions**

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

Query length	**Detect the active nodes**	**Ignoring typo in the 1st char**
4-char queries	96 ms	28 ms
8-char queries	142 ms	39 ms
12-char queries	225 ms	36 ms
16-char queries	305 ms	32 ms

- **PTIE** must offer this functionality:
  - Only by ignoring typo in the 1st 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

Number of Additional Suggestions (1/2)

- Word-Order Independent Suggestions

Query length	Having typed the first 2 chars of the last word	Having typed the first 3 chars of the last word	Having typed the first 4 chars of the last word
2-word queries	1.6	0.6	0.4
3-word queries	10.1	4.7	4
4-word queries	20.9	13.2	12.3

Number of Additional Suggestions (2/2)

- Typo-tolerant Suggestions

Query length	Having typed the first 4 chars (edit distance = 1)	Having typed the first 8 chars (edit distance = 2)	Having typed the first 12 chars (edit distance = 4)	Having typed the first 16 chars (edit distance = 5)
Detect the active nodes	71.4	7.3	7.1	3.3
Ignore typo in the 1st character	48.6	6	5.7	2.8

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.