Fast Search Engine Vocabulary Lookup

Fei (Xiang-Fei Jia)

Andrew Trotman

Jason Holdsworth

University of Otago

02 Dec 2011

About the Research

- An inverted file typically has two parts, a vocabulary of unique terms and a list of postings.
- The vocabulary is normally stored in alphabetical order so that it can be binary-searched.
- If the vocabulary is large, it can be represented as a 2-level B-tree and only the root of the tree is held in memory. The leaves are retrieved from disk only when required at runtime.

About the Research: the issues

- First, disks are an order of magnitude slower than main memory and it can take a long time to read the leaves.
- Second, the size of the leaves has a big impact on the performance. If the size if large, it can take a long time to read. However, large leaves might benefit from disk caching. On the other hand, small leaves are fast to read, but cannot benefit much from the caching.
- Last, how the leaves should be compressed.

Outline

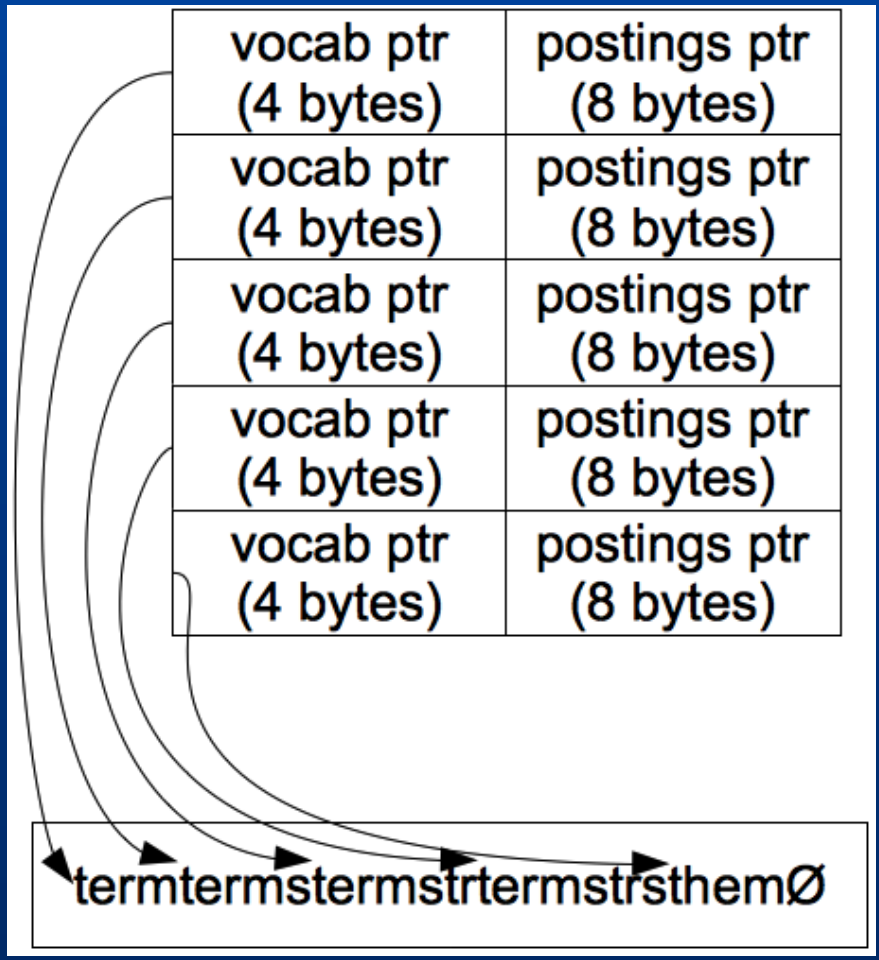
- In-memory Structures
- Structures based on 2-level B-tree
- Experiments and Results

In-memory: *fixed*

termØ (24 bytes)	postings ptr (8 bytes)
termsØ (24 bytes)	postings ptr (8 bytes)
termstrØ (24 bytes)	postings ptr (8 bytes)
termstrsØ (24 bytes)	postings ptr (8 bytes)
themØ (24 bytes)	postings ptr (8 bytes)

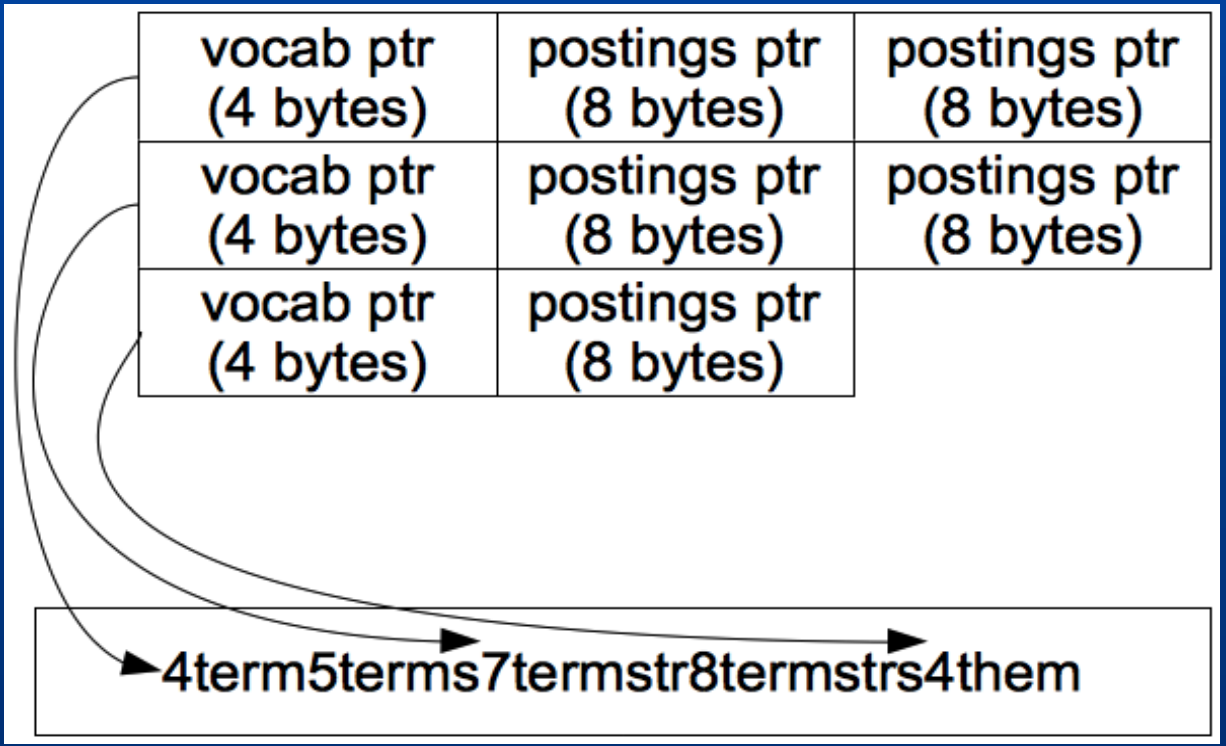
Manning et al, *Introduction to Information Retrieval*, 2008

In-memory: *String*



Manning et al, *Introduction to Information Retrieval*, 2008

In-memory: *blocked-k*



Takes $O(\log_2(n/k)+k)$ to search, where *k* is the blocking factor and *n* is the total number of the terms in the leaf

Manning et al, *Introduction to Information Retrieval*, 2008

2-level B-tree

- For large data collections and systems with limited resources, it is not feasible to store the whole vocabulary in memory. Instead, we discuss how to use 2-level B-tree to store vocabulary terms. The root of the B-tree can be loaded in memory while the leaves can be stored on disk and will be retrieved only when required.

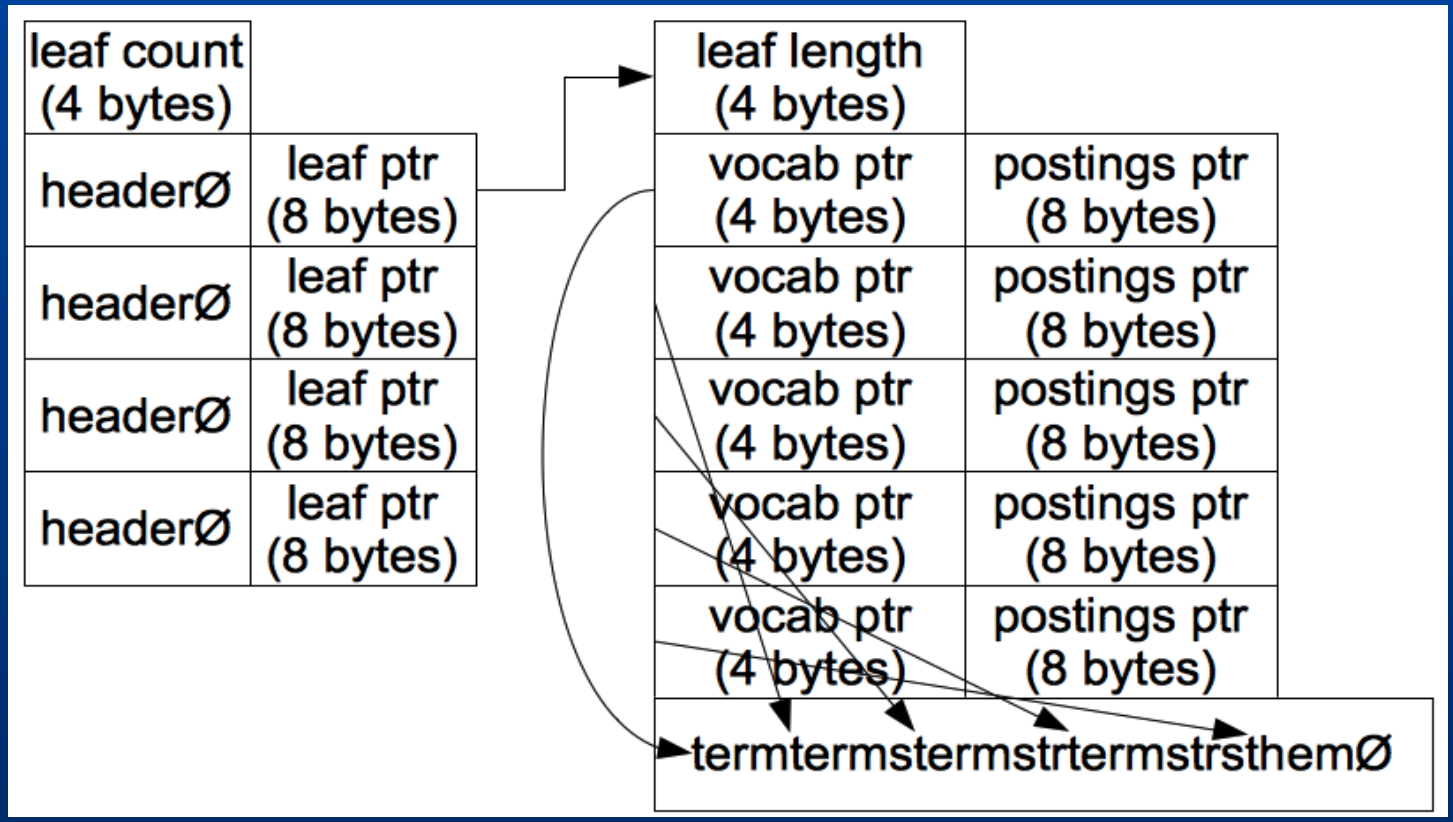
2-level B-tree: fixed, string, blocked

- Converted the original in-memory *fixed*, *string*, *blocked-k* into 2-level B-Tree.
- The header (root) is constructed to allow fast lookup using binary search to locate the leaf.
- The sizes of the leaves are aligned in disk sectors so that leaves can be stored efficiently on disk and read quickly from disk.
- As many terms as possible are inserted into the leaf and the remaining bytes padded with zeros.

2-level B-tree: fixed

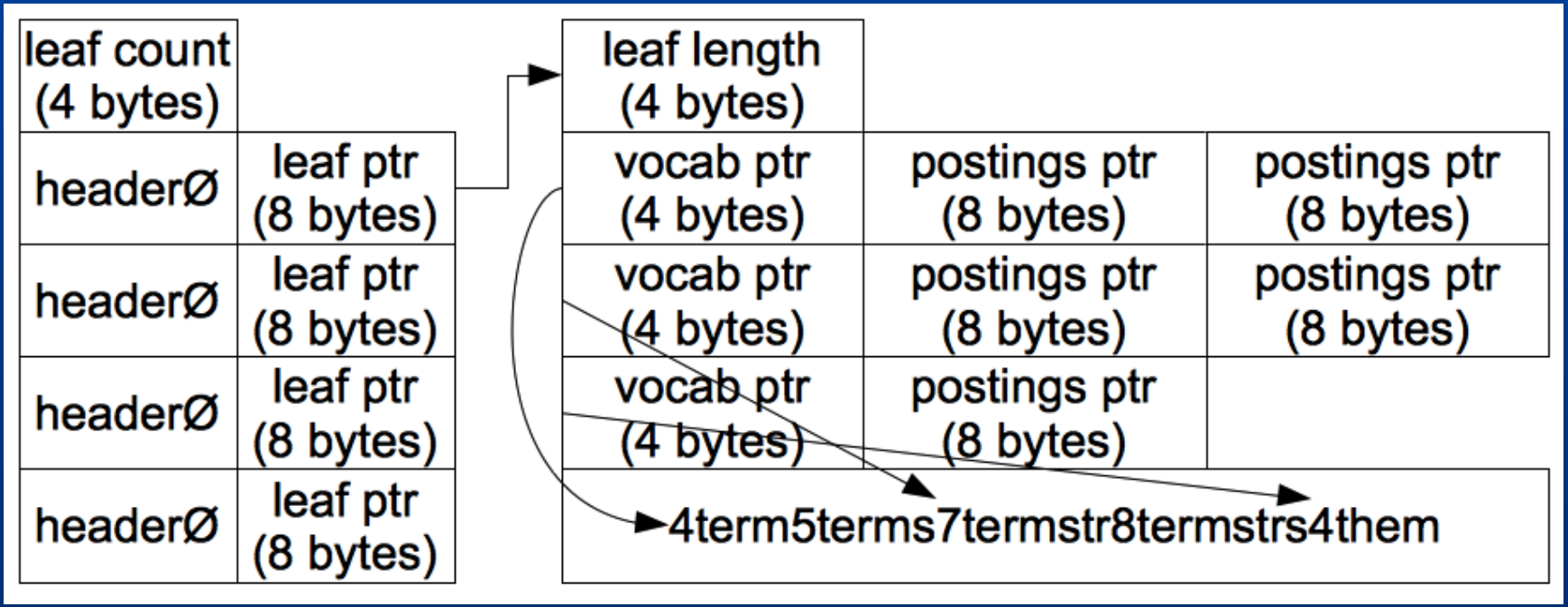
- No need to store the header. Instead the header is constructed at startup time.

2-level B-tree: string



Binary search in the leaves is allowed.

2-level B-tree: blocked-k

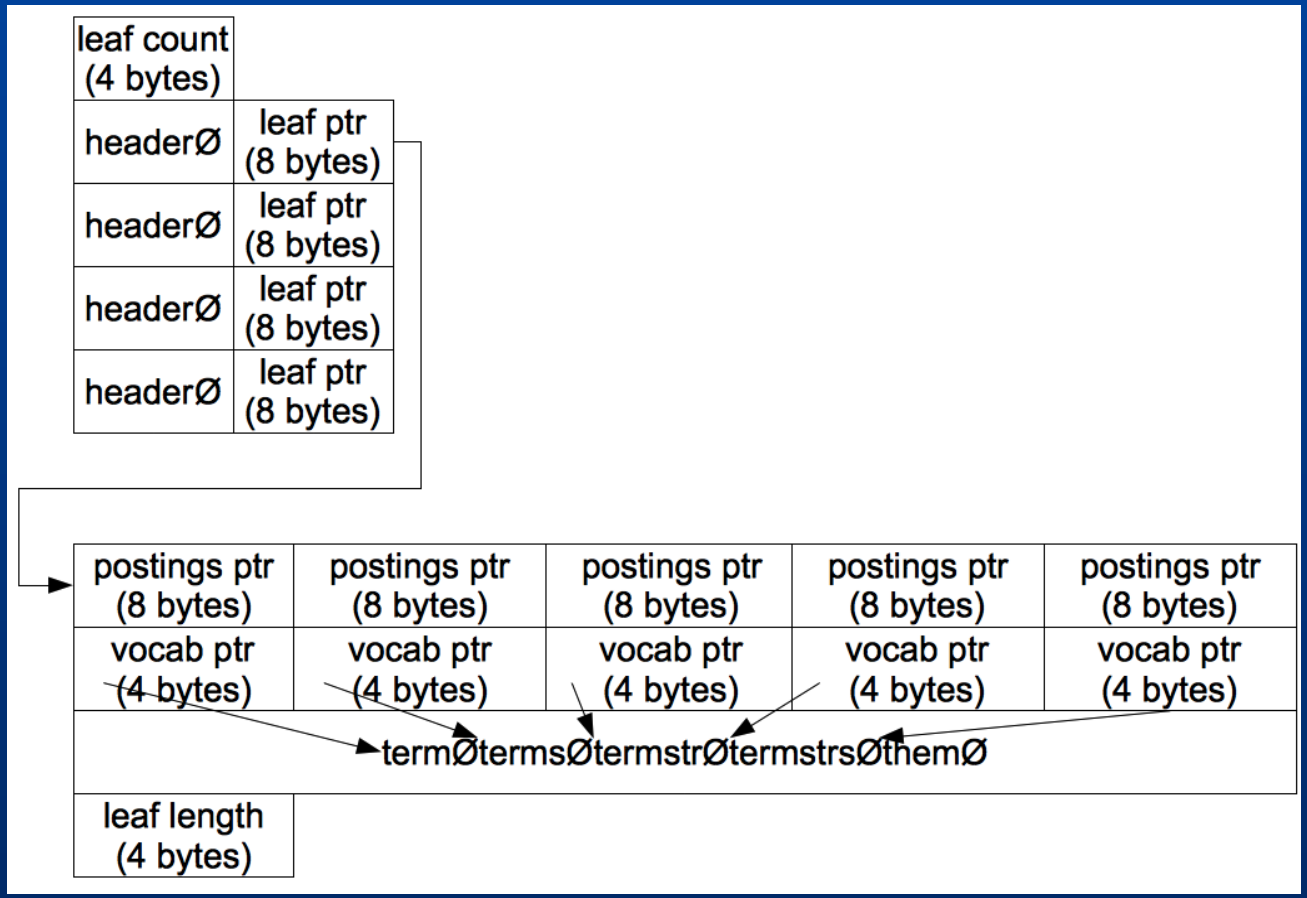


Binary search in the leaves is allowed, but takes $O(\log_2(n/k)+k)$.

2-level B-tree: *embed*, *embedfront*

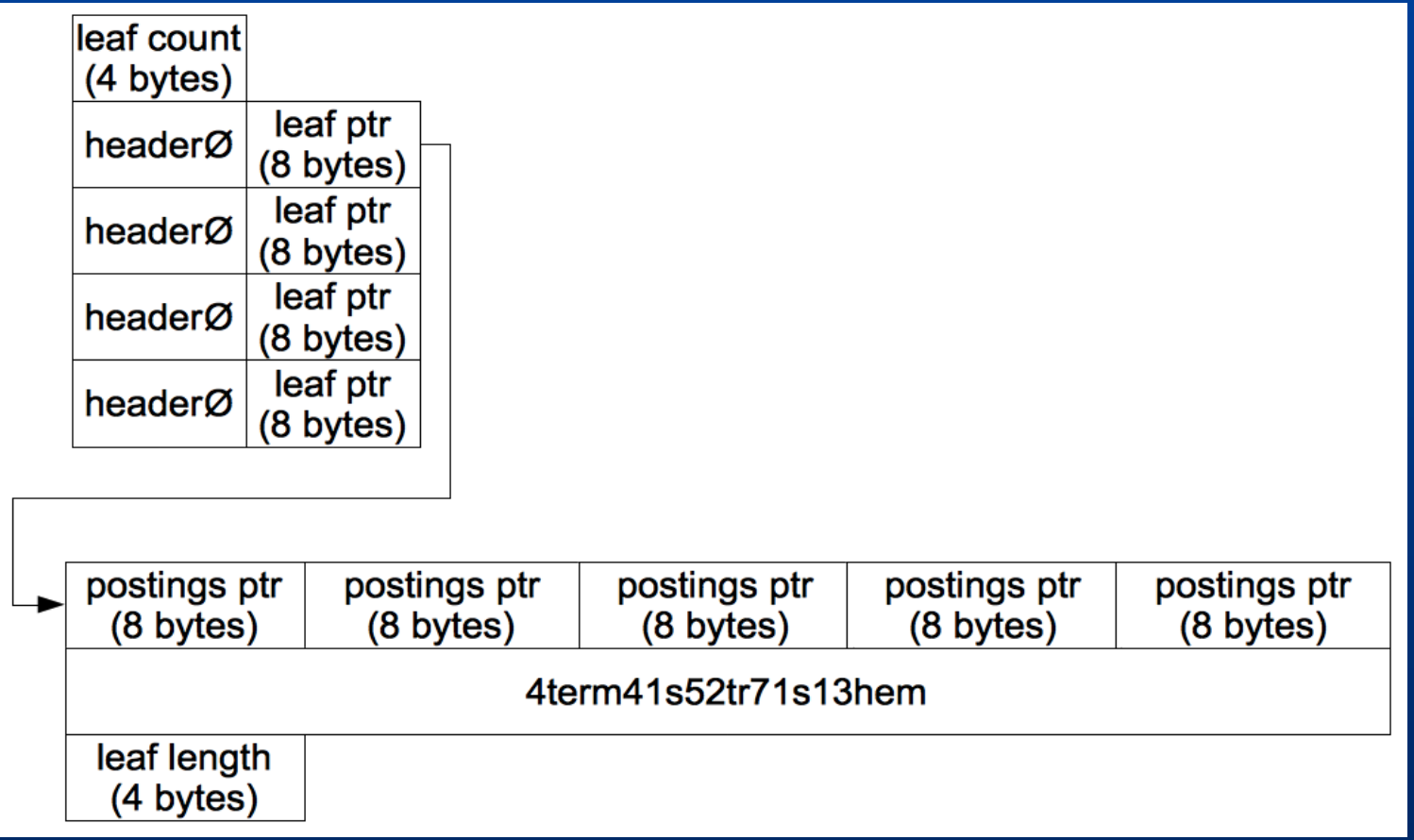
- They are based on the assumption that similar data types should be grouped together as CPUs are good at caching and fast at reading data of a fixed length.
- The data types used in the leaf structure are word-aligned as CPUs read/write data in units of words.

2-level B-tree: *embed*



Binary search in the leaves is allowed.

2-level B-Tree: *embedfront*



Binary search in the leaves is not allowed.

2-level B-tree: *embedfixed*

- The *embedfixed* algorithm further extends *embed* by providing compression.
- Terms are split into two parts, the common prefix and the rest of the terms (the suffixes).
- A Leaf only contains terms with the same common prefix, and only the suffixes of the terms are stored in the leaf.
- The common prefixes are stored in the header.
- Essentially, *embedfixed* is a form of Trie.

2-level B-tree: *embedfixed*

- Construction of the leaf
  - Instead of allocating as many terms as possible into each leaf, a leaf only contains terms with the same common prefix. The leaf size thus depends on the number of common prefix characters and how many terms share the that common prefix.

2-level B-tree: *embedfixed*

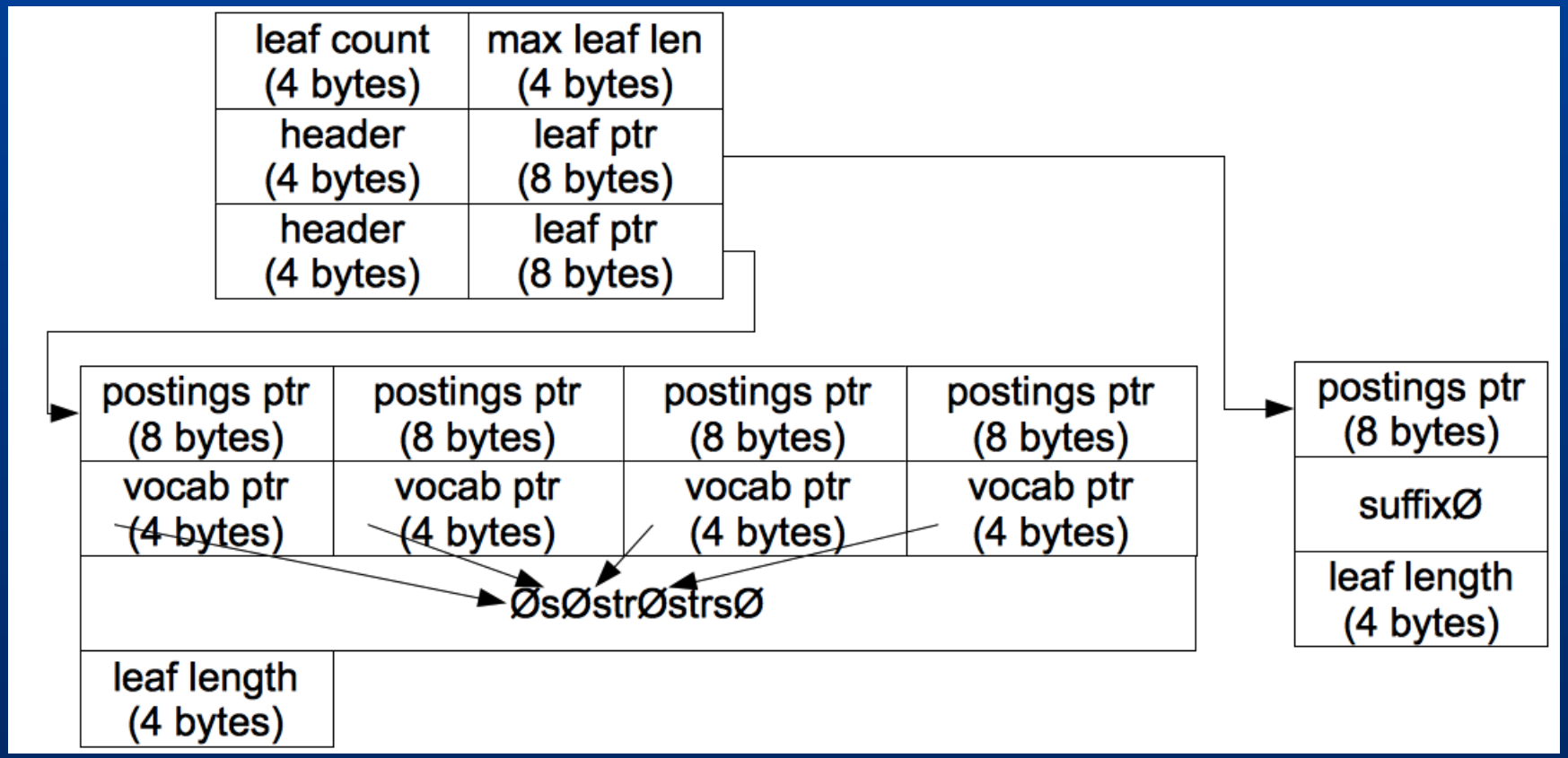
- Construction of the header
  - The construction of the header is also different from the previous algorithms. Instead of storing the whole terms in the header, only the characters of the common prefix are stored, and all common prefixes are of the same length.

2-level B-tree: *embedfixed*

- Advantages:
  - The header structure has a smaller footprint since only partial terms are stored.
  - During lookup, a shorter string comparison can locate the leaf containing the term.
  - If the partial terms are treated as integers, integers comparison can be used for fast lookup instead of standard string comparison.

2-level B-tree: *embedfixed*

Example terms “term,terms,termstr,termstrs,them”



Binary search in the leaves is allowed.

Experiments

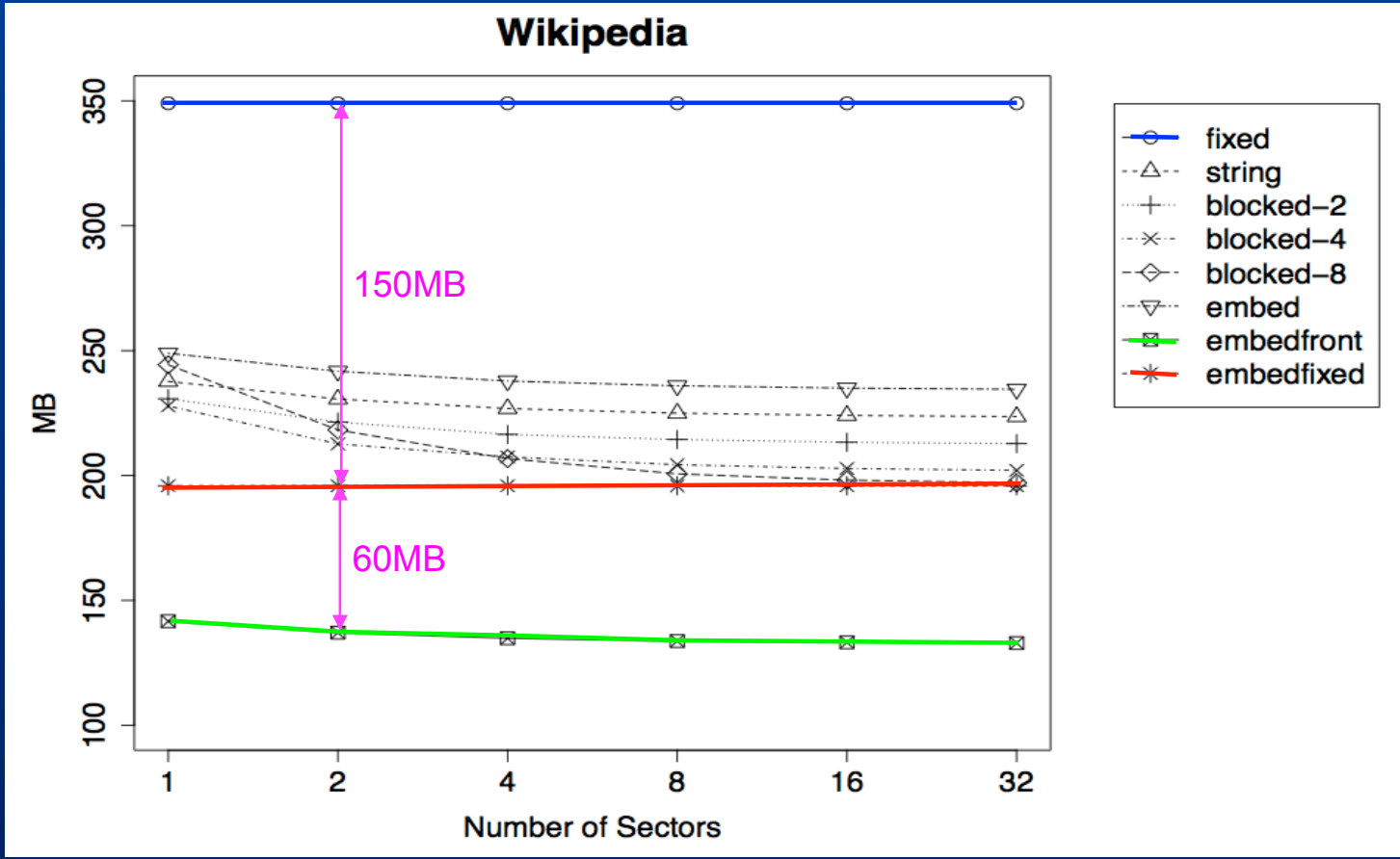
- Two collections were used, the INEX 2009 Wikipedia collection and the uni-gram corpus from the Web 1T 5-gram version 1.
- The queries were from the Million Queries Track in TREC 2007.
- A simulation program was written for the experiments. (source code available)
- Four sets of experiments were conducted.

Results: Experiment One

- The first set of experiments examined the storage space and the wastage due to disk sector alignment.
- The sector has a size of 512 bytes.

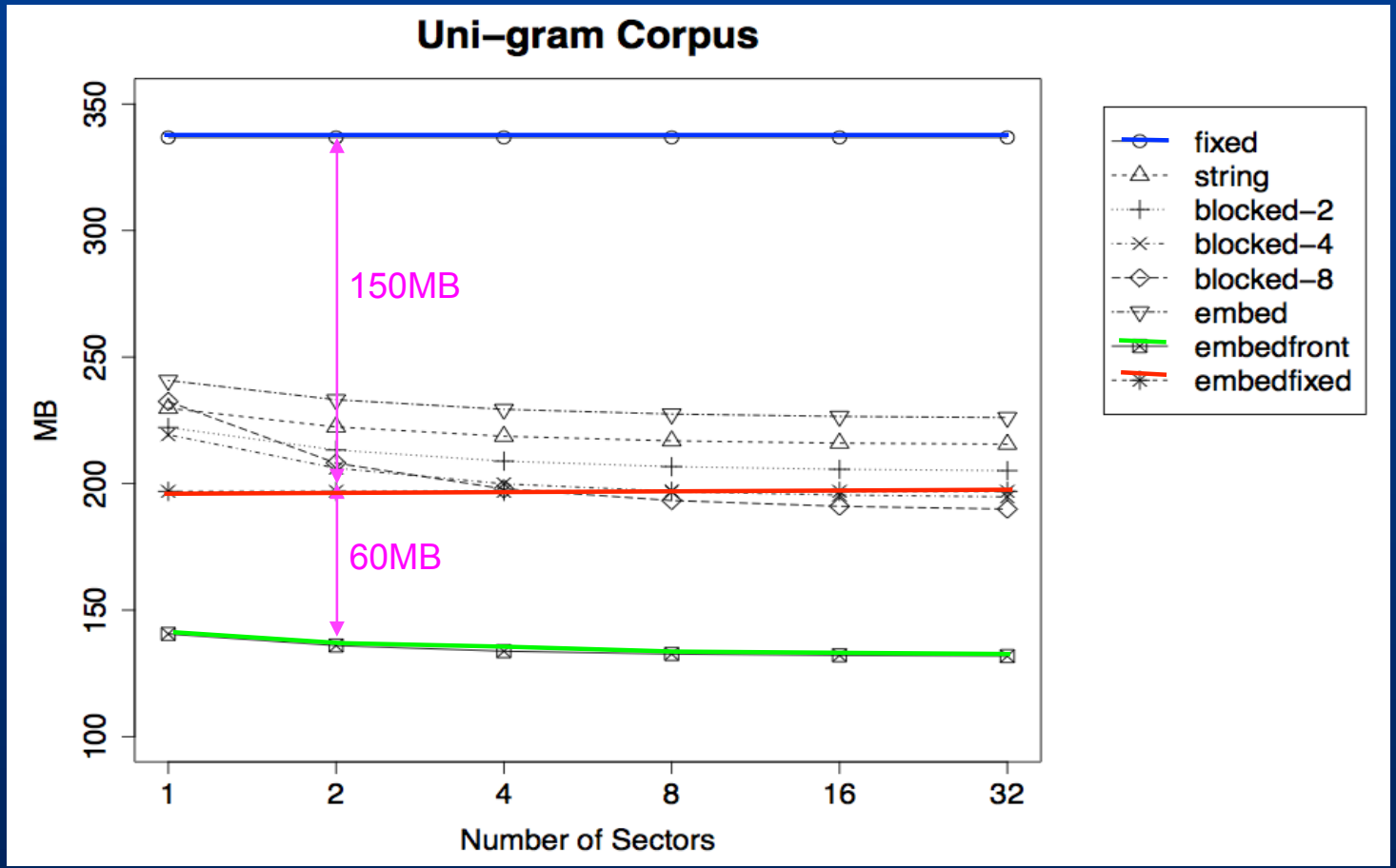
Results: Experiment One

- The total storage space



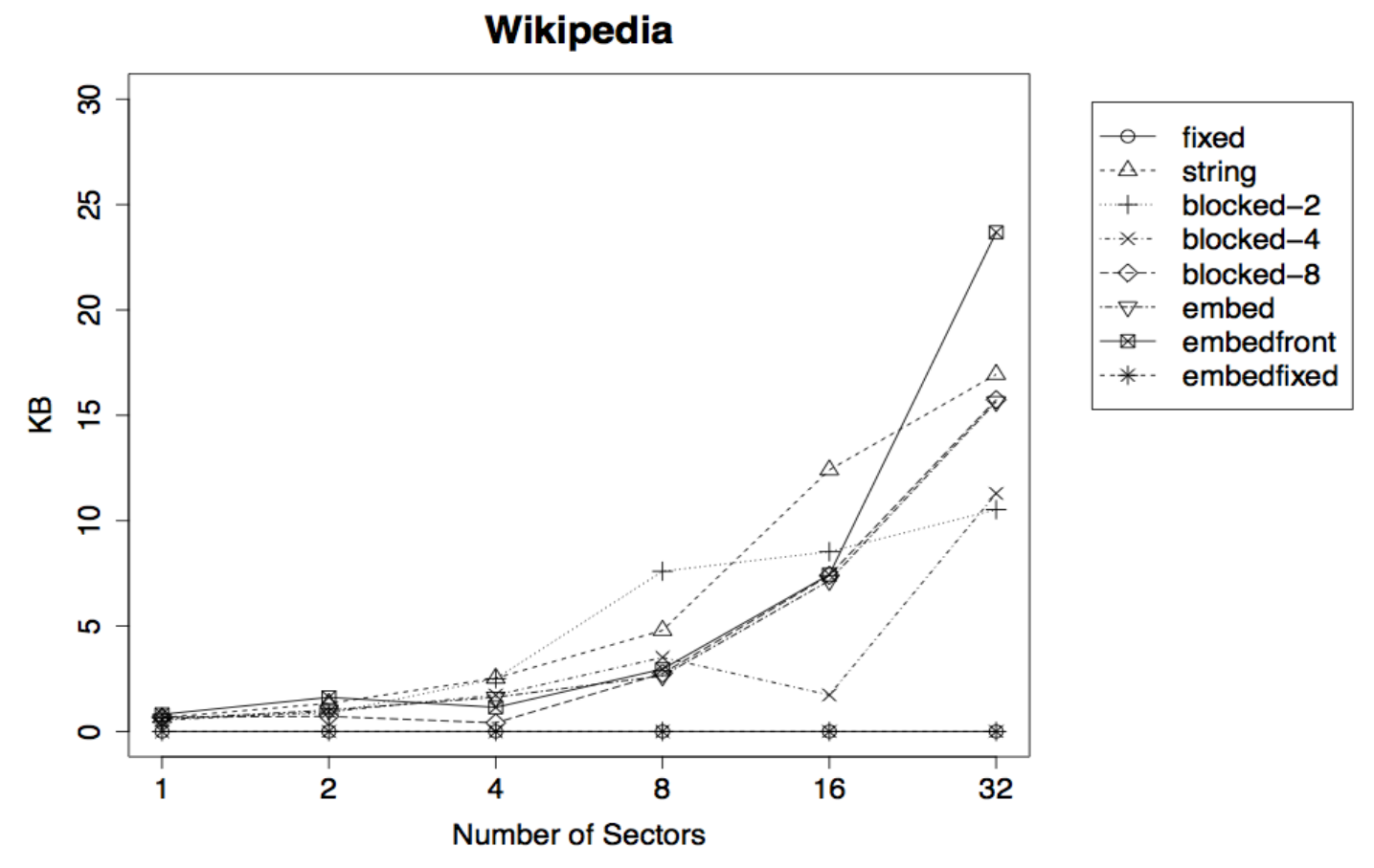
Results: Experiment One

- The total storage space



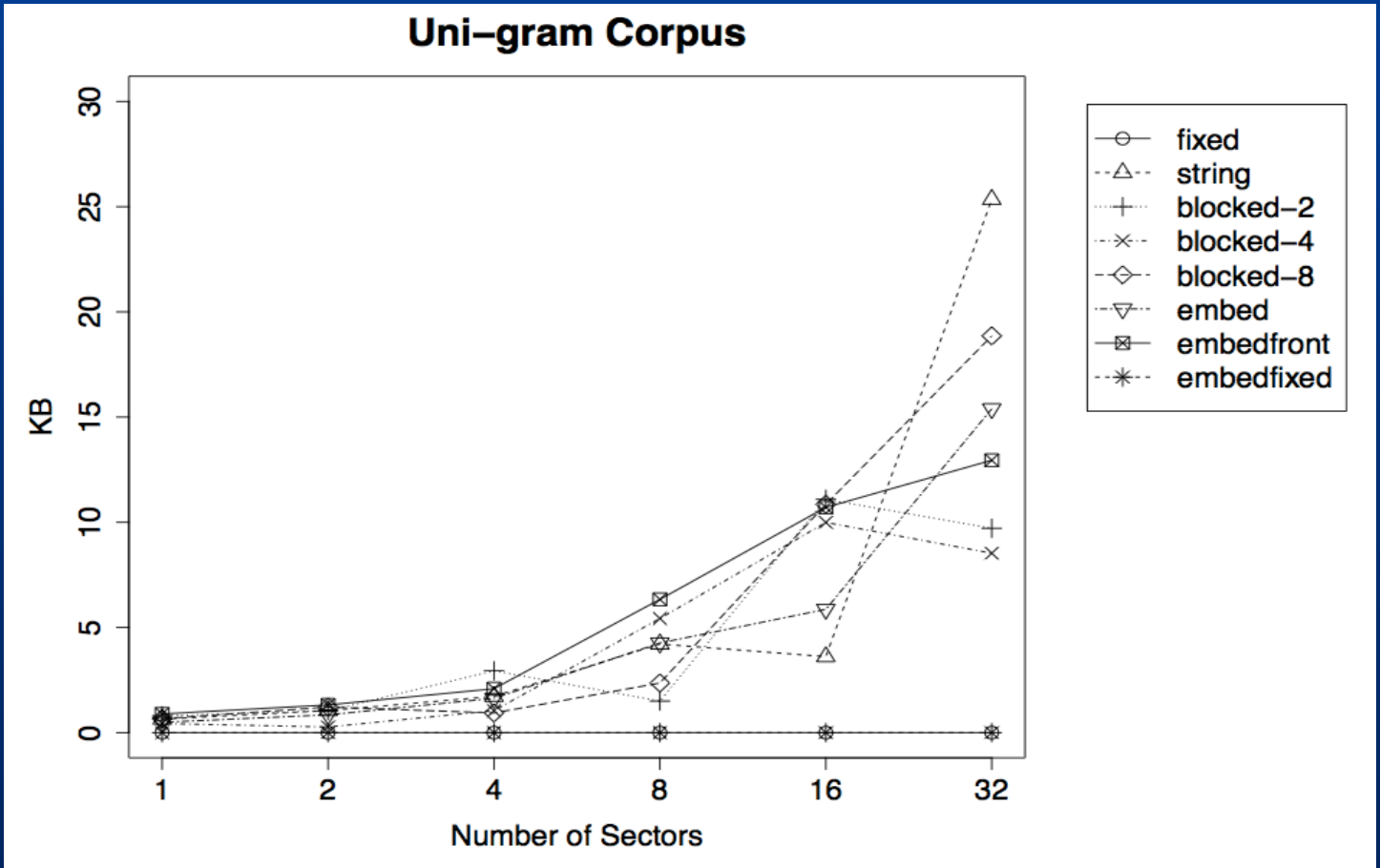
Results: Experiment One

- The wastage of the storage space



Results: Experiment One

- The wastage of the storage space

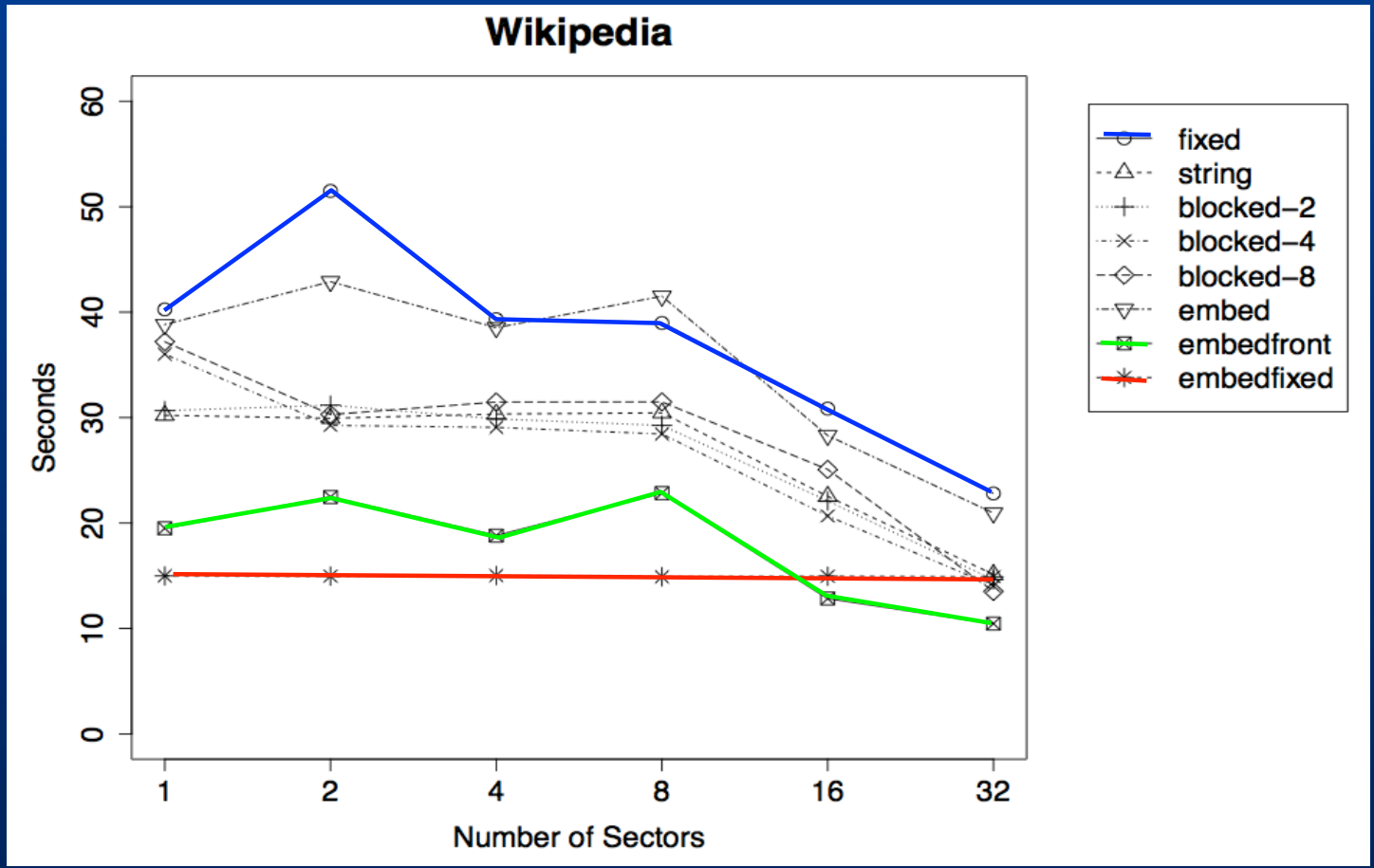


Results: Experiment Two

- The second set of the experiments examined the search performance when the header of the vocabulary structures was loaded into memory and only the required leaves were retrieved from disk.

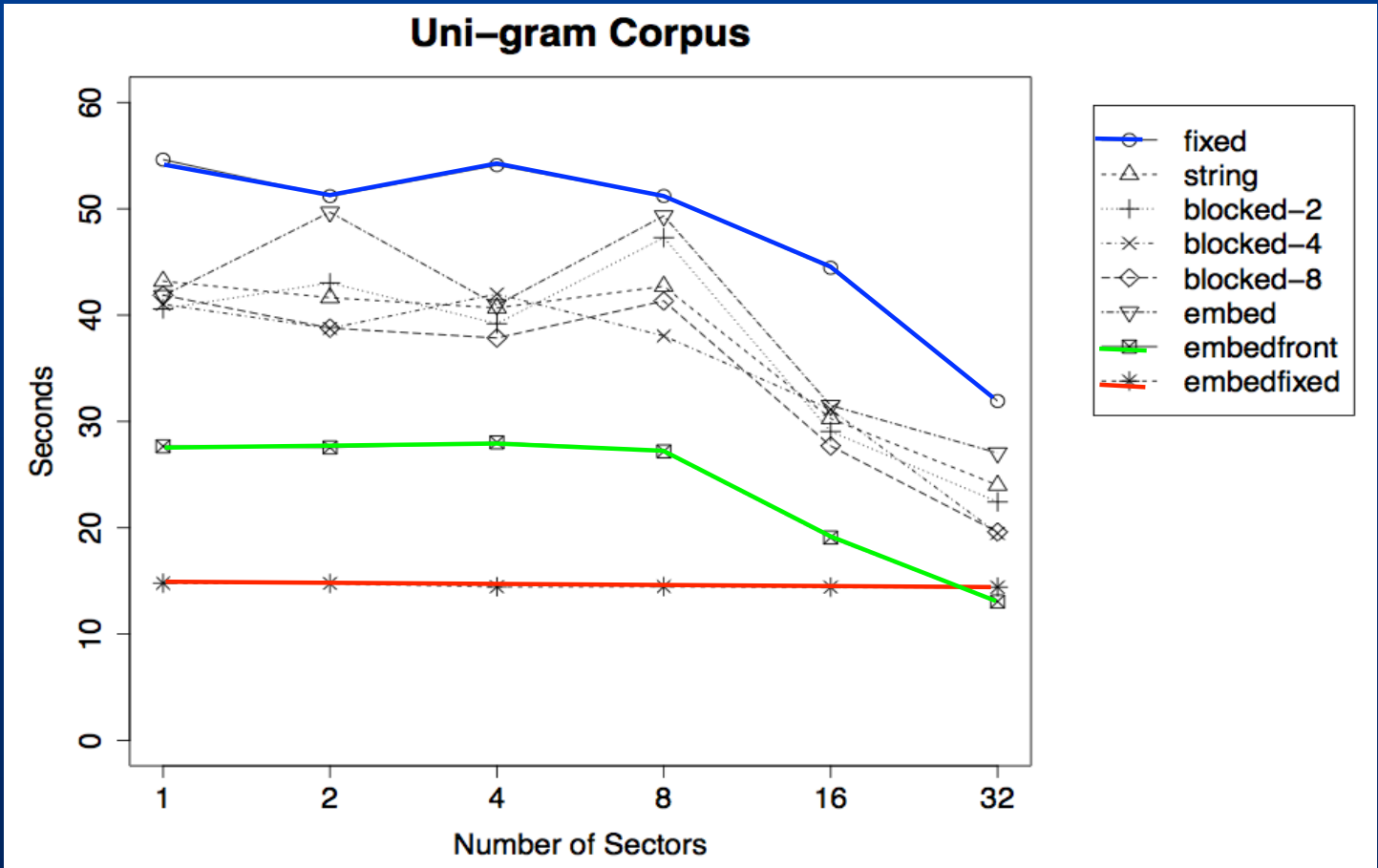
Results: Experiment Two

- The I/O time



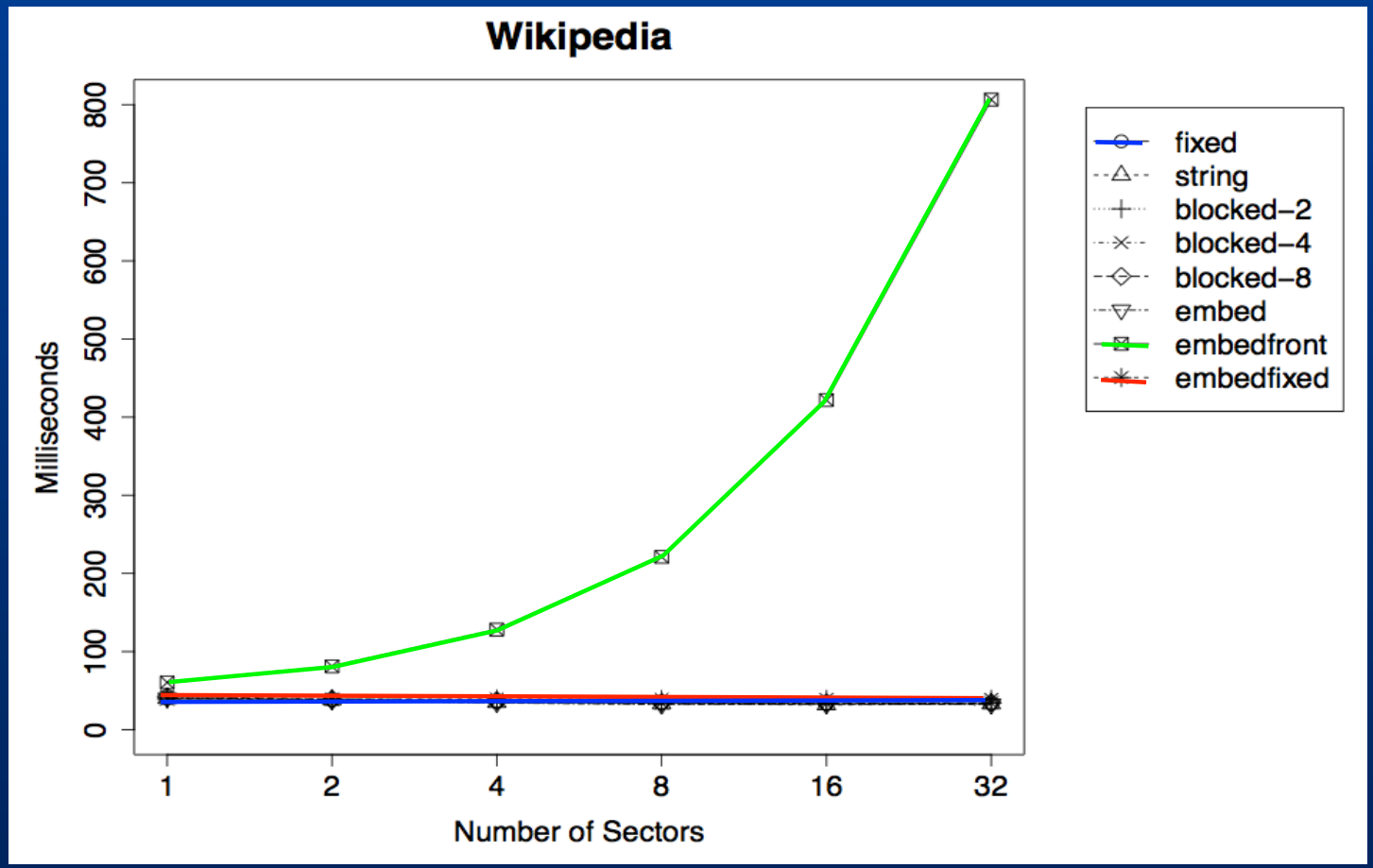
Results: Experiment Two

- The I/O time



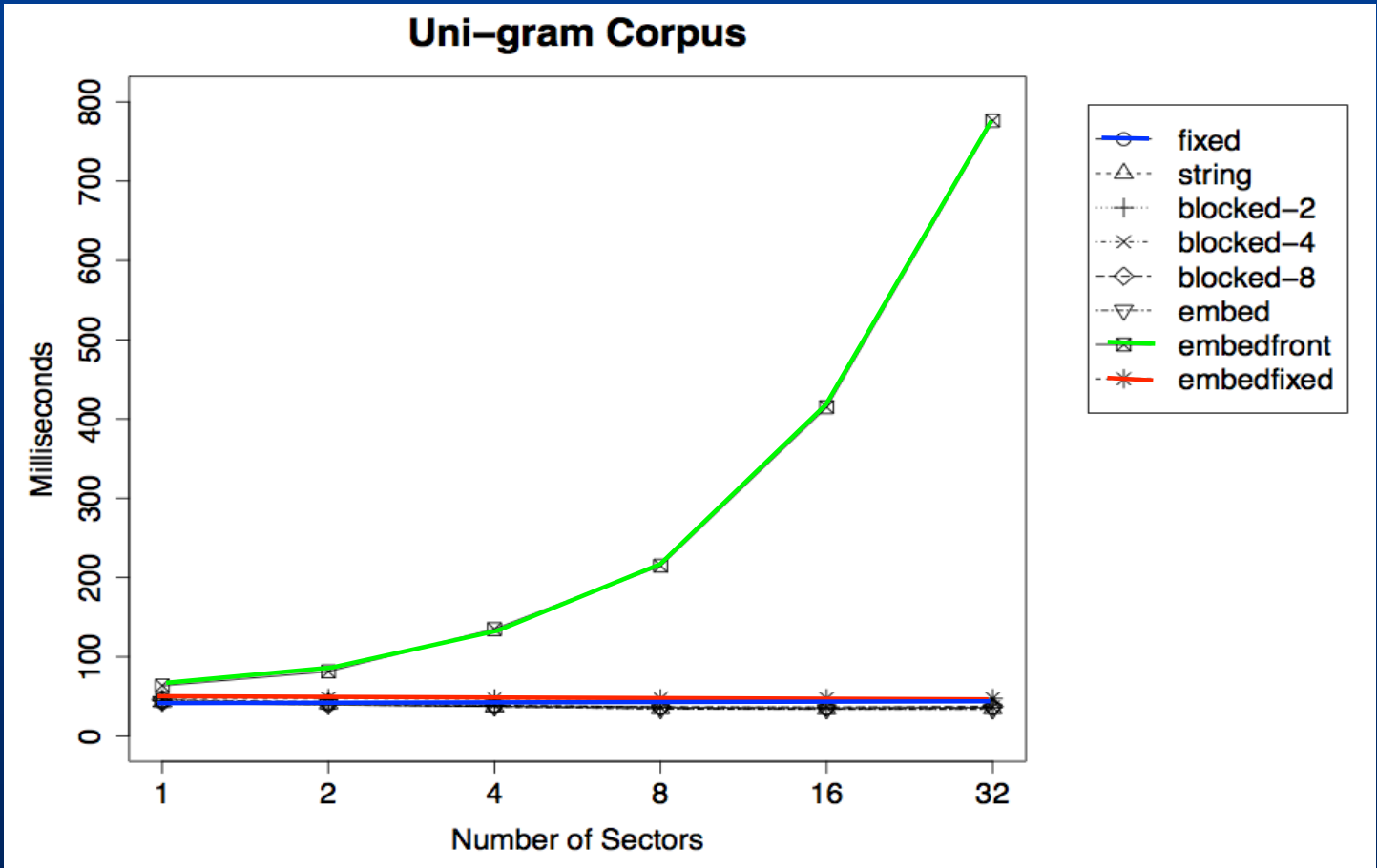
Results: Experiment Two

- The search time



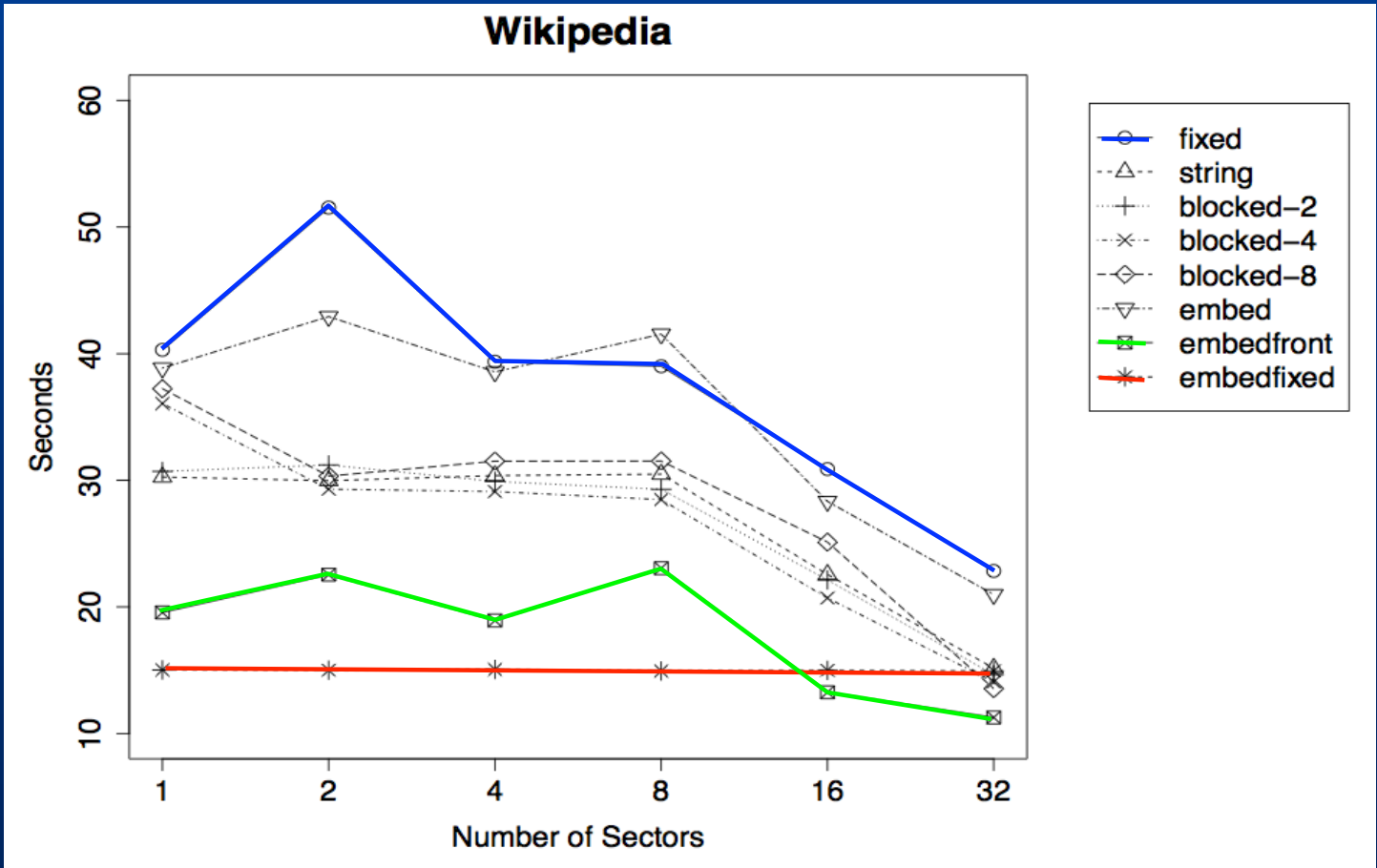
Results: Experiment Two

- The search time



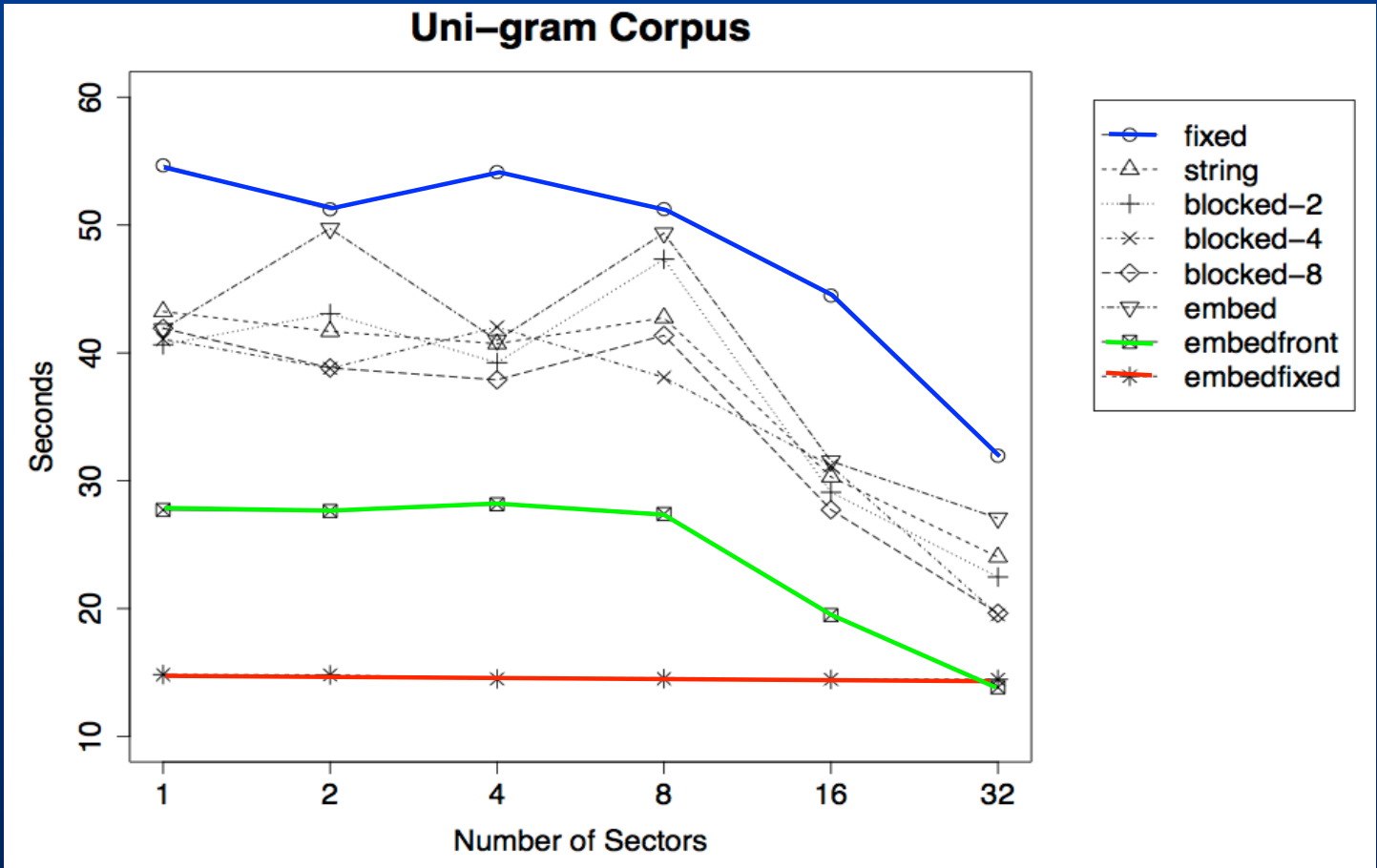
Results: Experiment Two

- The total time



Results: Experiment Two

- The total time

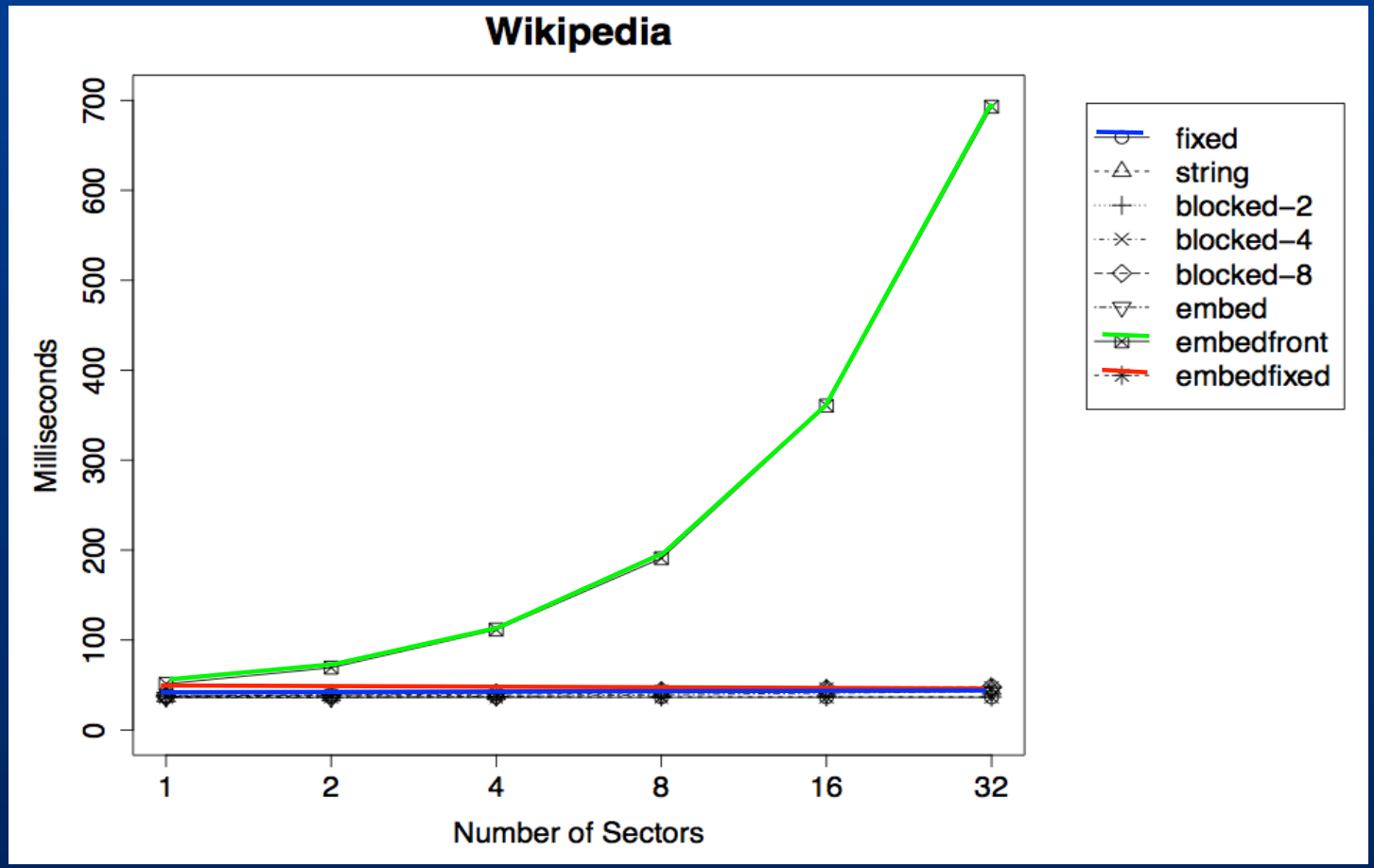


Results: Experiment Three

- The third set of experiments examined the search performance when the whole vocabulary was loaded into memory.

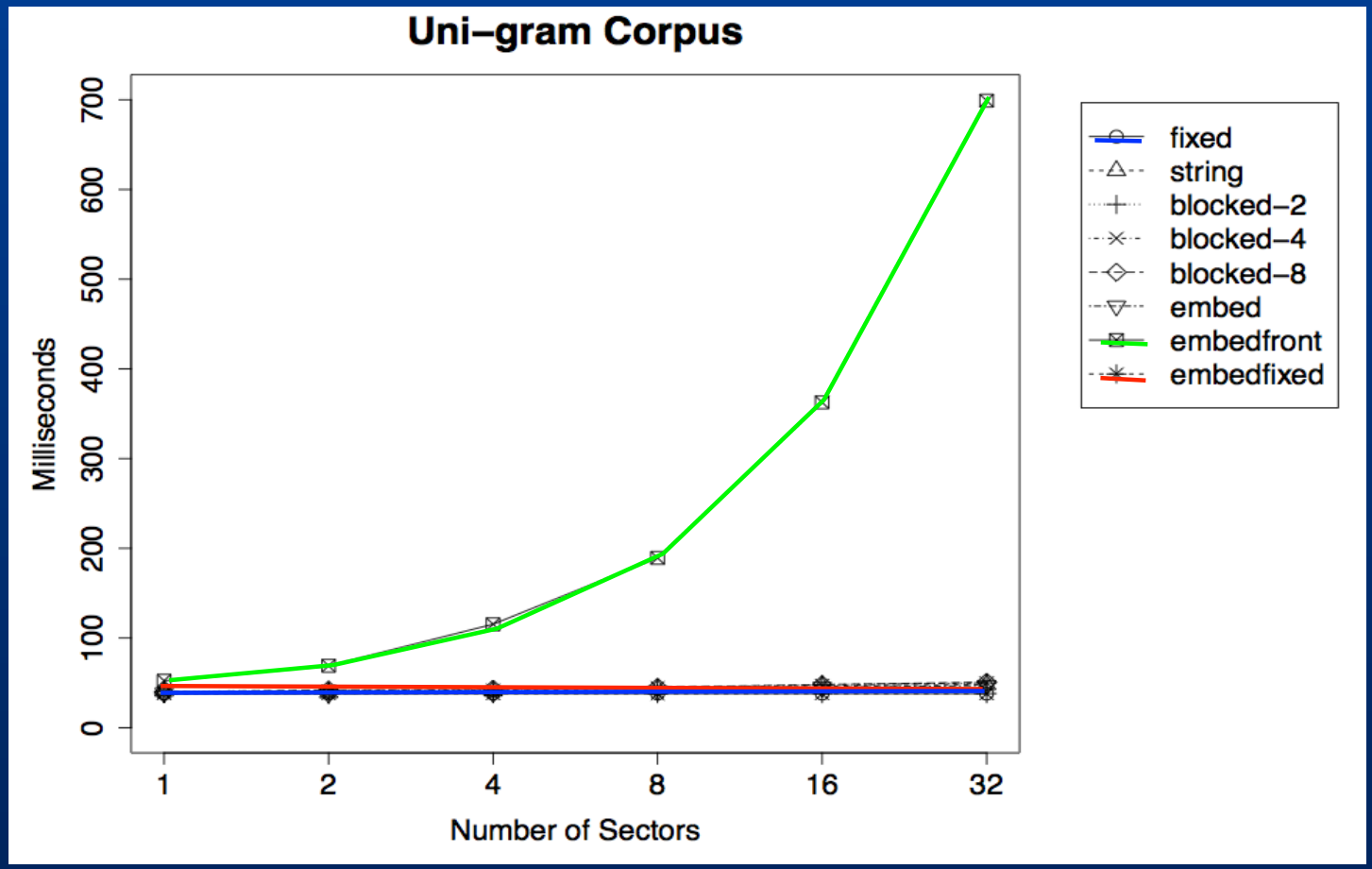
Results: Experiment Three

- The total time



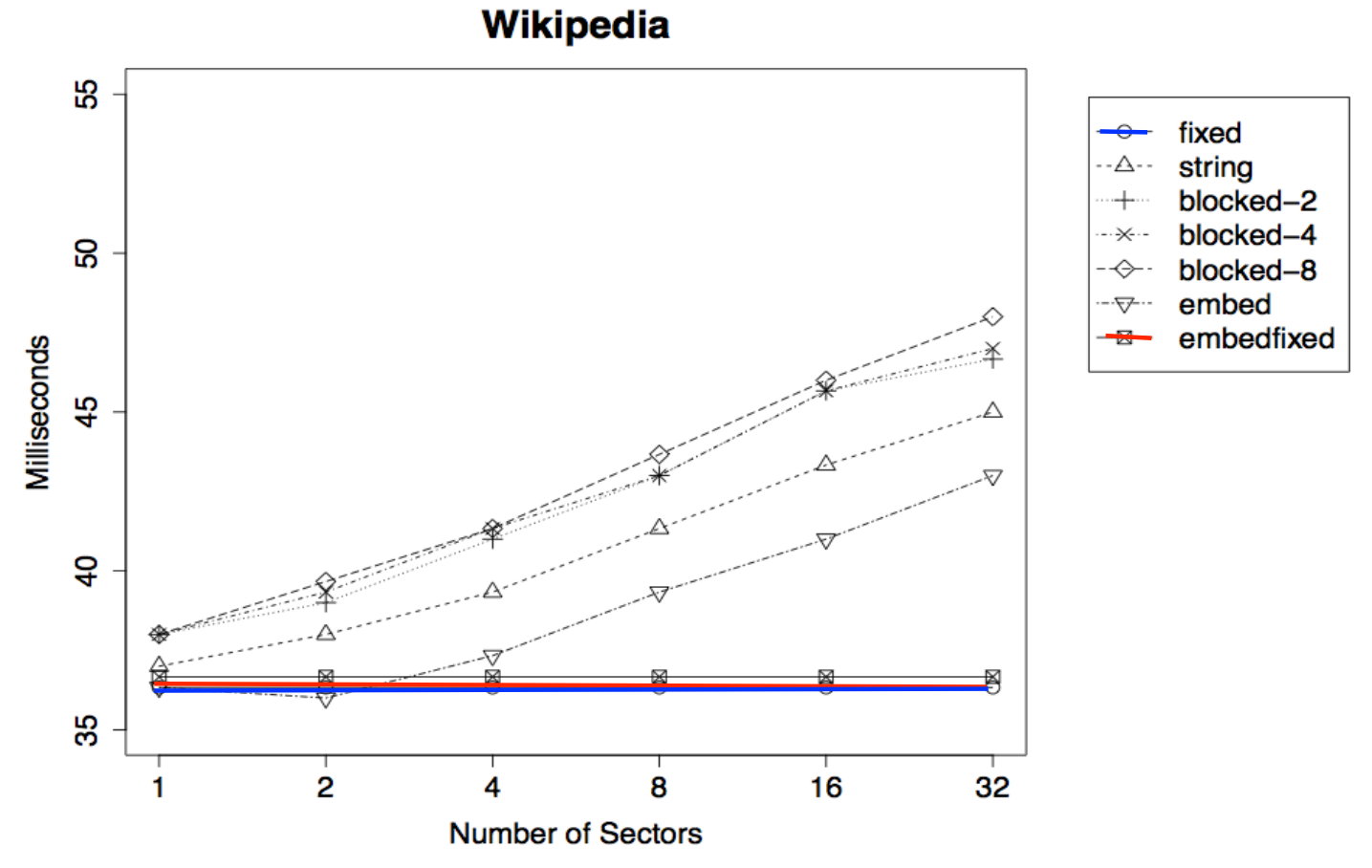
Results: Experiment Three

- The total time



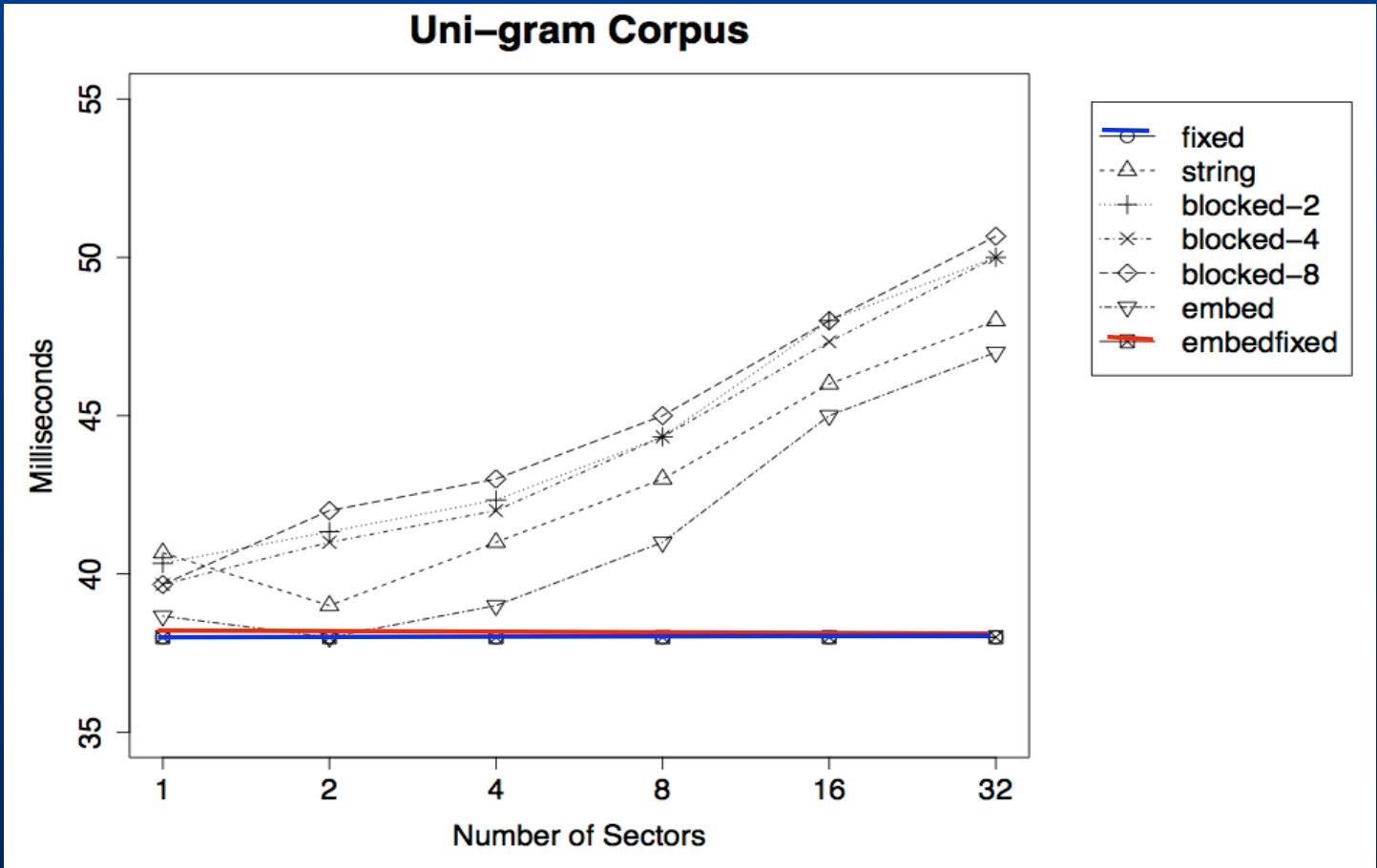
Results: Experiment Three

- The total time without embedfront



Results: Experiment Three

- The total time without embedfront

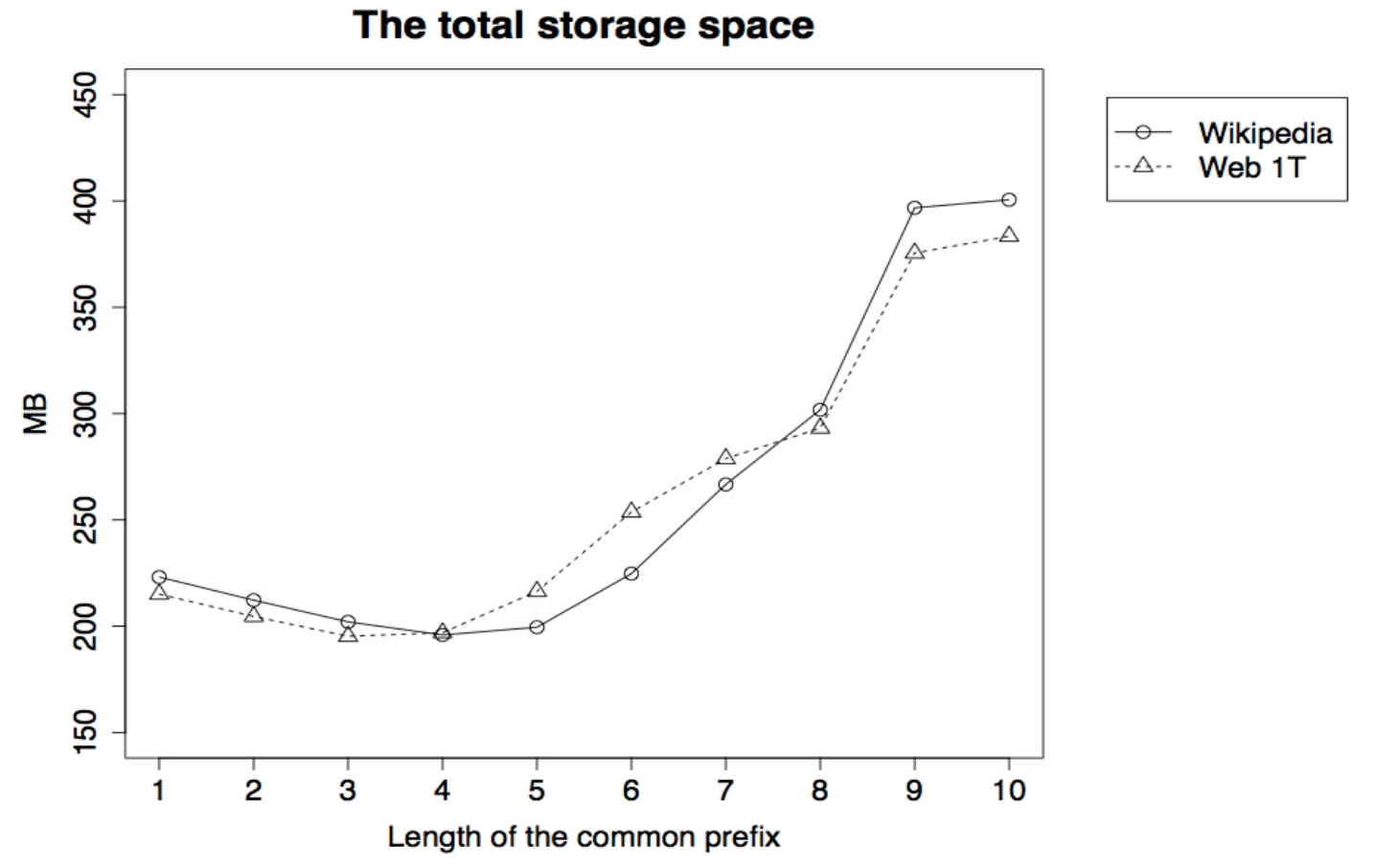


Results: Experiment Four

- The last set of experiments explored various lengths of the common prefix in the header for *embedfixed*.

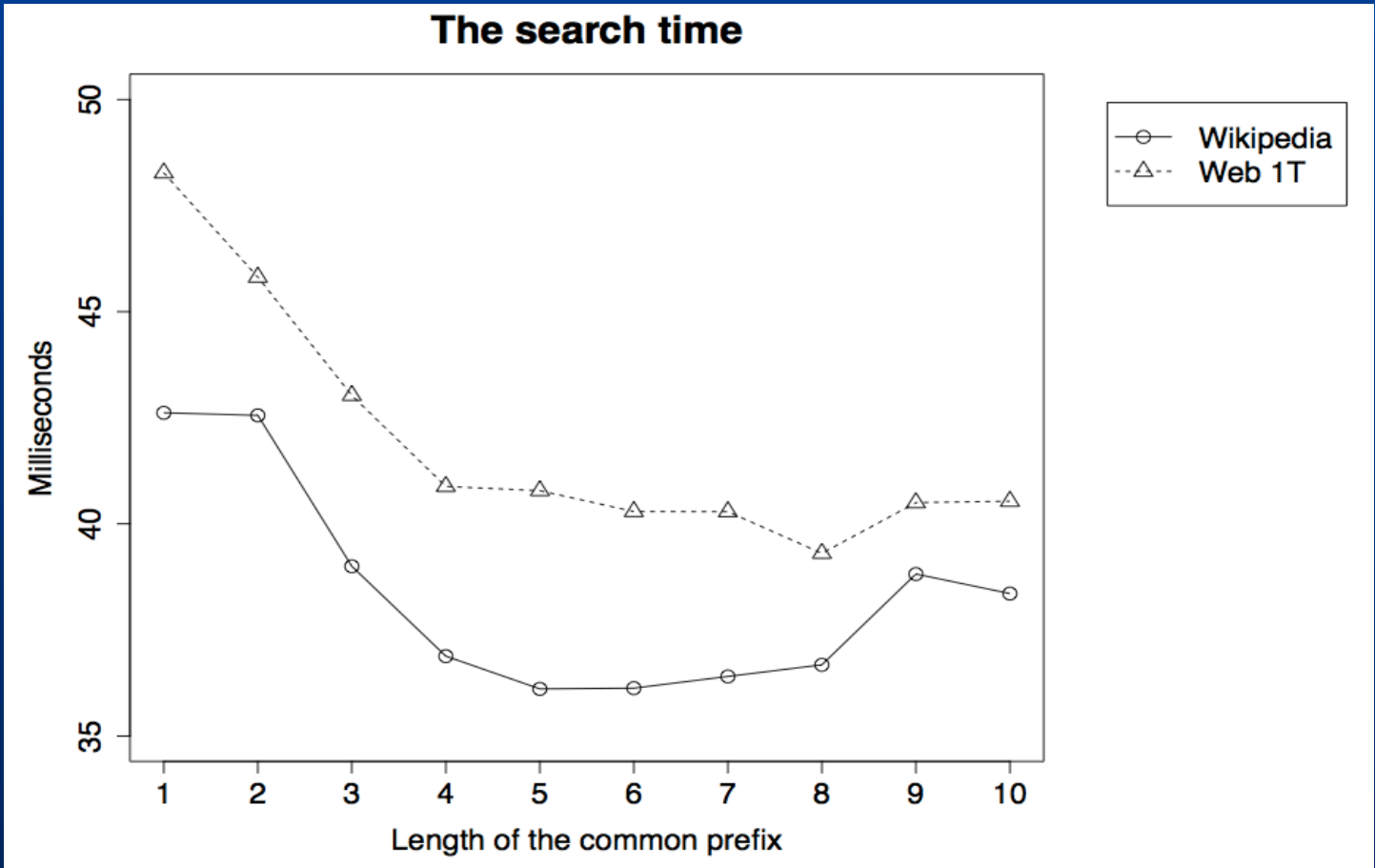
Results: Experiment Four

- The total storage space



Results: Experiment Four

- The search time



Conclusion

- We have conducted experiments on a number of algorithms, including 2-level *fixed*, *string*, *blocked-k*, *embed*, *embedfront*, *embedfixed*.
- The embedfixed algorithm provides a simple but effective encoding method, with an average search time of $O(\log_2(lc)) + O(\log_2(ll))$ where *lc* is the number of leaves in the header and *ll* is the number of entries in the leaf.
- The embedfixed provides the best trade-off between storage space and fast lookup.

Future Work

- Conducts experiments on different hardware, for example smart phones and SSD hard drives.
- Compare the proposed algorithms with other algorithms, for example B-Tries and hashing.

Acknowledgement

- Thanks Google for the travel support

Question