Burst Tries: A Fast, Efficient Data Structure for String Keys

Steen Heinz Justin Zobel Hugh E. Williams

School of Computer Science and Information Technology,

RMIT University

Presented by Margot Schips

22 April 2010

Outline

- Introduction to Burst Tries
  - What's a Trie?
- Applications
- Review of Data Structures
- Burst Trie, how does it work?
  - Operation
  - Heuristics
  - Data Structures
- Experiments and Results
- Conclusions

Introduction to Burst Tries

- This paper proposes a new data structure, the Burst Trie, that has significant advantages over existing options for managing larges sets of distinct strings in memory.
- Advantages of Burst Tries
  - They use about the same memory as a binary search tree
  - They are as fast as a trie
  - A Burst trie maintains the strings in sorted or near-sorted order (though not as fast as a hash table).

So what's a 'Trie'?

- A trie is a compact data structure for representing a set of strings, such as all the words in a text.

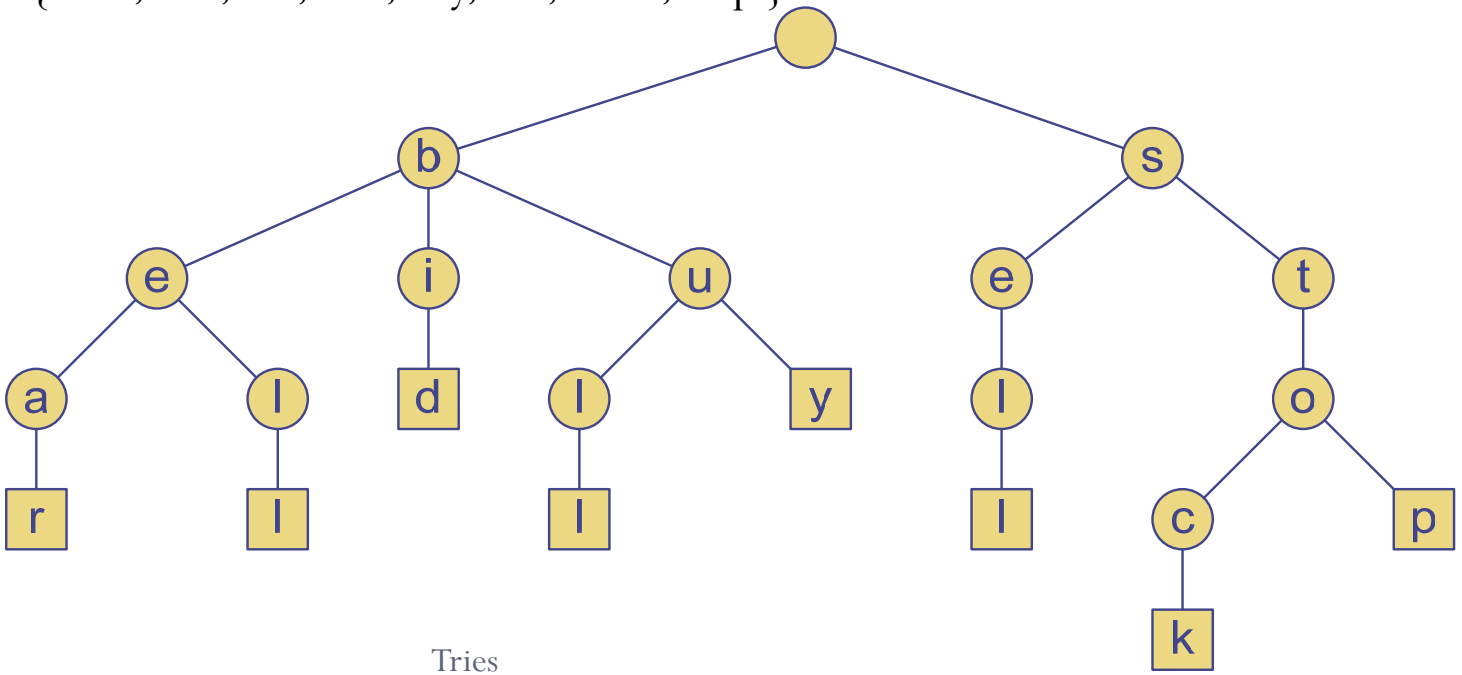
Why use a 'Trie'?

- Preprocessing the pattern speeds up pattern matching queries.
- A trie supports pattern matching queries in time proportional to the pattern size.

Standard Tries

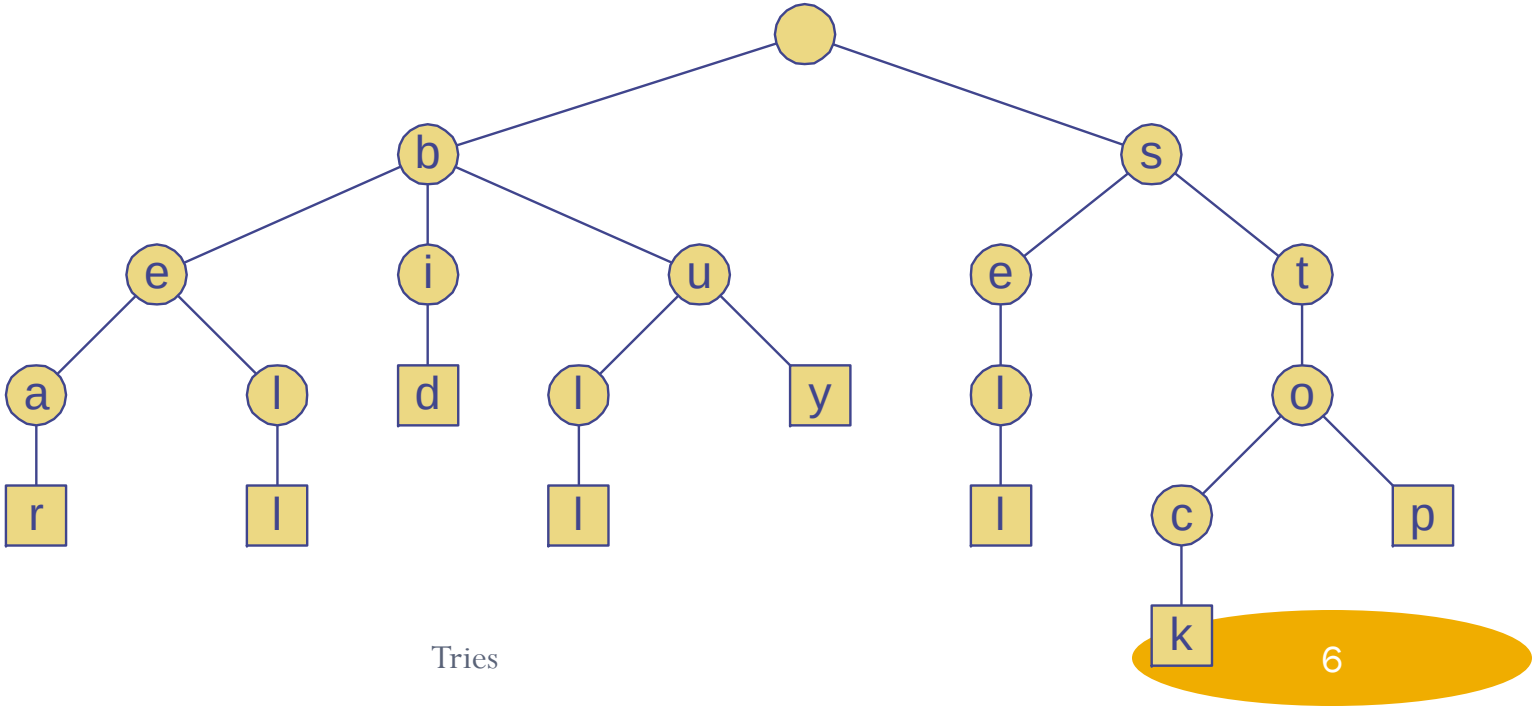
- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings

  S = { bear, bell, bid, bull, buy, sell, stock, stop }



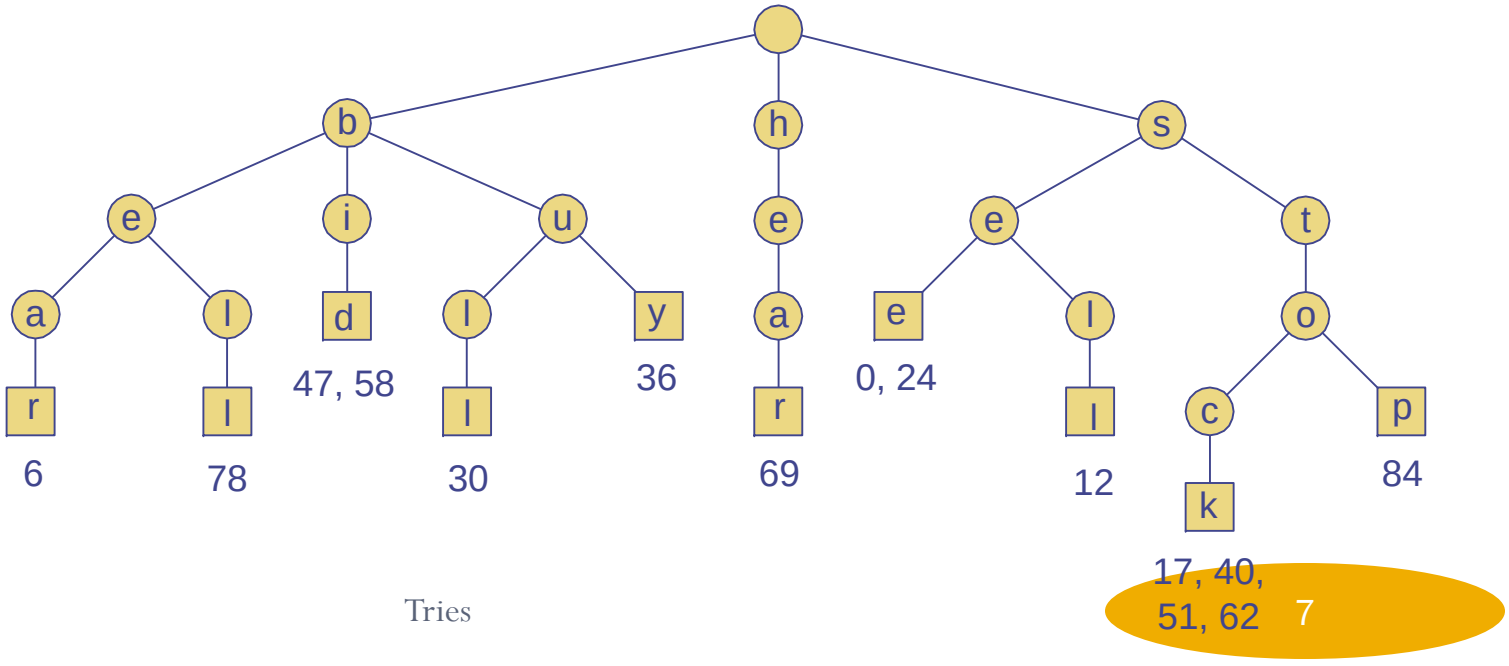
Analysis of Standard Tries

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  - n total size of the strings in S
  - m size of the string parameter of the operation
  - d size of the alphabet



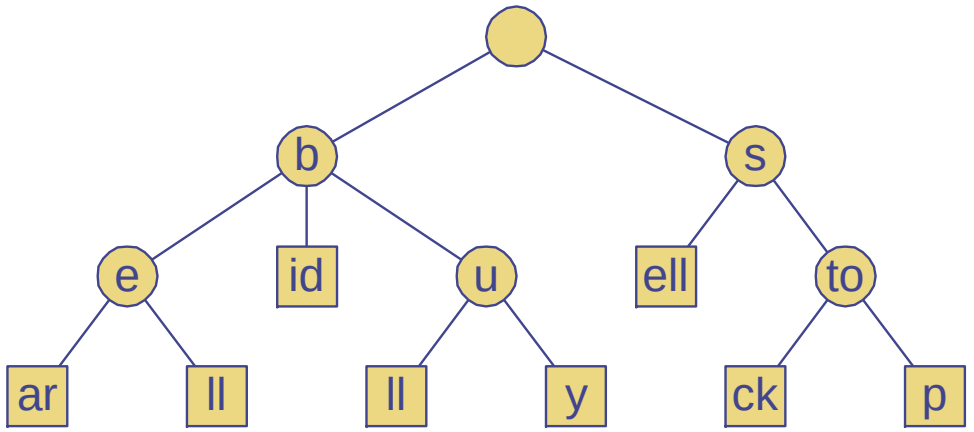
Word Matching with a Trie

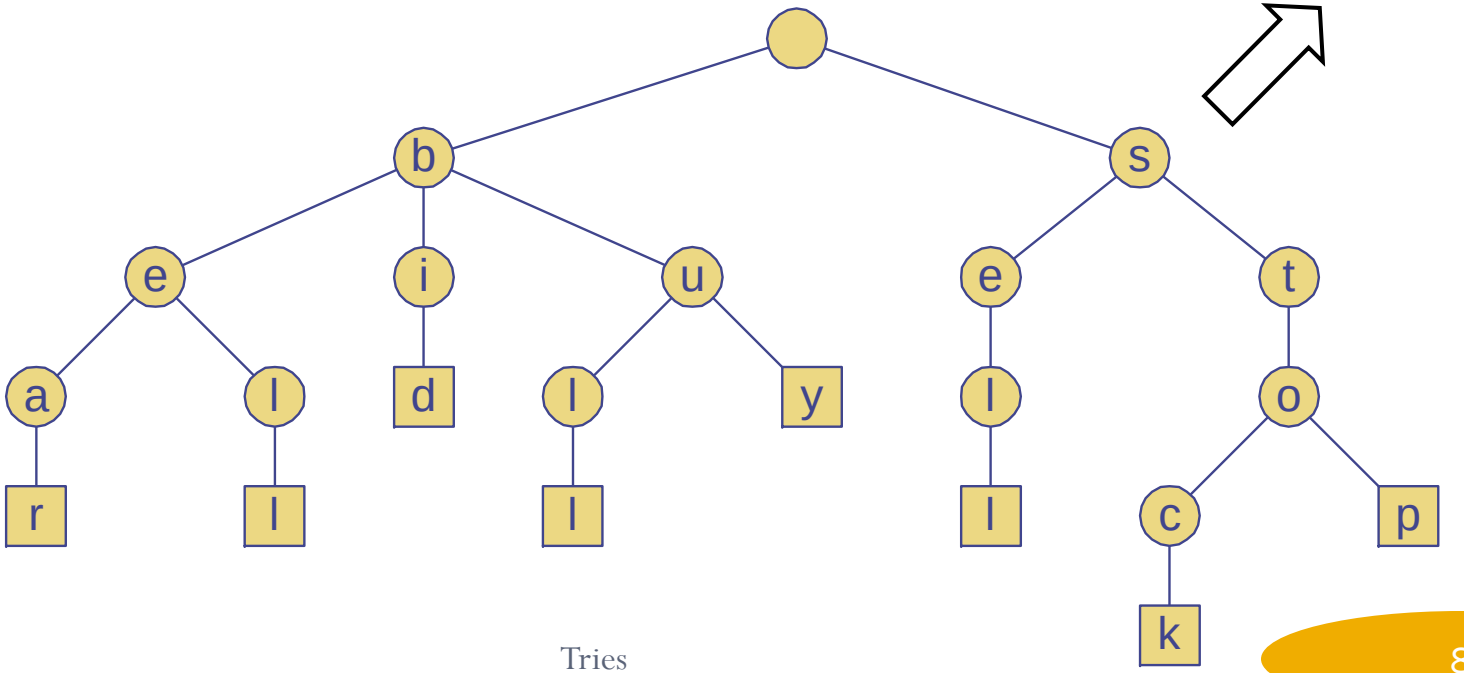
- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text



Compressed Tries

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of "redundant" nodes





Applications

- Many applications depend on efficient management of large sets of distinct strings in memory.
  - News archives
  - Law collections
  - Business reports
- These collections contain many millions of distinct words, the number growing more or less linearly with collection size
- During index construction for text databases a record is held for each distinct word in the text, containing the word itself and information such as counters.

Indexing: efficiency

- Indexing storage needs
  - Vocabulary
  - Term occurrence count
  - Individual term locations
- This *Vocabulary Accumulation* needs to be stored in a data structure.
- Goal: to maintain a set of records for which a string (term) is the key.
- Memory provides faster access than disk, so a compact data structure is preferred.

Review of Data Structures

- Hash Table
  - Efficient and relatively fast (improved by bitwise hash function, chaining and move-to-front in chains), but unsorted strings are slower to update.
- Tries and Ternary Search Trees
  - Fast, but space intensive
- Binary Search Trees including splay trees, randomised search trees and red-black tress
  - Less space-intensive, but slow

Burst Trie, how does it work?

- A Burst Trie uses containers to store keys/values before creating branches.
- When the containers are full, they "burst" and are turned into branches.
- The benefit is that a more efficient data structure for small sets of keys/values can be used in the container, making it faster than a conventional tree.

Burst Trie, how does it work?

- This structure is a collection of small data structures, which we call *containers,* that are accessed via a conventional trie.
- Searching involves using the first few characters of a query string to identify a particular container, then using the remainder of the query string to find a record in the container.
- A container can be any data structure that is reasonably efficient for small sets of strings, such as a list or a binary search tree.

Burst Trie, how does it work?

- Initially, a burst tree consists of a single container.
- When a container is deemed to be inefficient, it is *burst*, that is, replaced by a trie node and a set of child containers which between them partition the original container's strings.
- Two major design decisions to be explored for burst tries:
  - what data structure to use for a container
  - how to determine when a container should be burst

Existing Data Structures, (BST)

Binary Search Tree (BST)

- Allocation of strings to nodes determined by insertion order
- Advantages
  - Assuming a stable distribution in the text collection, with skew distribution it is expected that common words will occur close to the beginning of the text collection and are therefore close to the root of the BST.
  - Access to common terms is fast, since first levels of tree are kept in cache.
- Disadvantages
  - If BST is sorted or distribution changes (ex. Docs in multiple languages), behavior degrades. Long, non-branching paths, or 'sticks', create time-consuming searches.

Existing Data Structures (BST)

AVL and Red-black trees

- Advantages
  - Balanced trees are reorganized on insertion or deletion; eliminates long sticks found in unbalanced trees
  - O(log n) upper limit to length of search path
- Disadvantages
  - Rebalancing doesn't favor frequent access to common words at the top of the tree. Common words are just as likely to be placed in a leaf.
- Experiments show faster search with red-black tress in data with low skew, BST is better for typical vocabulary accumulation

Existing Data Structures (BST)

Splay Trees (BST variant)

- Potential Advantages
  - On each search, node accessed is moved to root by splaying, a series of node rotations
  - Intuitively, it seems that commonly-accessed node would remain near root and would adapt quickly to change. Search time is O(log n)
- Disadvantages
  - Splay Trees require more memory as efficient implementation requires each node to have a pointer to its parent.
  - In reality, common words actually don't stay near the top, they get shifted down as newly-accessed words rise.
  - Reorganization is expensive, better to reorganize at measured intervals. Experiments used interval of 4.

Existing Data Structures (BST)

Randomised Search Tree (RST)

- Treap: uses inorder traversal in sort order for strings
- Tree reorganized using rotations to restore treap property after each search
- RSTs give a similar performance as Splay Trees, so not tested in this study

Existing Data Structures

Hash Tables

- Chaining is most efficient hash table for vocabulary accumulation. A large array is used to index a set of linked lists of nodes. To search an array, the index is computed by hashing the query string, then the string is sought in the linked list for that index.

Existing Data Structures (Hash)

- For peak performance using hash tables
  - Distinct strings must be sorted once vocabulary has been accumulated because order of slots does not correspond to string sort order.
  - Hashing needs to be fast. Instead of standard modulus hashing, a bit-wise algorithm is employed
  - Hash Table must be sufficiently large such that the number of slots needs to be a significant fraction of the number of distinct strings.
  - OR can use smaller tables if, on each search, the accessed node is moved to the front of the list. This has same efficiency as large tables and, most importantly for this application, with move-to-front chains efficiency declines much more slowly with increasing vocabulary size.

Existing Data Structures (Hash)

- With move-to-front chains and vocabulary accumulation, over 99% of searches terminate at the first node in the slot. It is for this reason the small number of total string inspections required that, as can be seen in our experiments, hashing is the fastest of all the methods tested in this study.

Existing Data Structures (Tries)

- Trie
  - A *trie* is an alternative to a BST for storing strings in sort order.
  - A node in a standard trie is an array of pointers, one for each letter in the alphabet, with an additional pointer for the empty string.
  - A leaf is a record concerning a particular string.
  - Search in a trie is fast, requiring only a single pointer traversal for each letter in the query string. That is, the search cost is bounded by the length of the query string.

Existing Data Structures (TST)

Ternary Search Trees (TSTs)

- TST's are a variant forms of tries with reduced space requirements and compact tries.
  - Each node represents a single character *c*, and has three pointers. The left (respectively, right) pointer is for strings that start with a character that alphabetically precedes (respectively, follows) *c*.
  - Thus a set of TST nodes connected by left and right pointers are a representation of a trie node. These can be rebalanced on access.
  - The central pointer is for strings starting with *c*, thus corresponding to the `*c*' pointer of a trie node. TSTs are slower than tries, but more compact.

Hash vs. Tree

- In earlier work comparing the tree data structures discussed above, it was observed that compared to hashing, trees had three sources of inefficiency.
  - First, the average search lengths were surprisingly high, typically exceeding ten pointer traversals and string comparisons (even on moderate-sized data sets with highly skew distributions). In contrast, a search under hashing rarely requires more than a string traversal to compute a hash value and a single successful comparison.
  - Second, for structures based on BSTs, the string comparisons involved redundant character inspections, and were thus unnecessarily expensive. For example, given the query string 'middle' and given that, during search, 'michael' and 'midfield' have been encountered, it is clear that all subsequent strings inspected must begin with the prefix 'mi'.
  - Third, in tries the set of strings in a subtrie tends to have a highly skew distribution: typically the vast majority of accesses to a subtrie are to find one particular string. Thus use of a highly time-efficient, space-intensive structure for the remaining strings is not a good use of resources.

Why Burst Tries?

- So far trees are slow, redundant, and inefficient for high skew data such as text data

- Can we improve upon their design?

- Yes!

Burst Tries, goals

- The primary design goal for the burst trie is to reduce the average number of string comparisons required during a search to less than two.
- This requires an adaptive structure that stores more frequent terms such that they can be retrieved more rapidly than less frequent terms.
- As a secondary goal, any gains in performance should not be offset by impractical memory requirements, as observed in tries.

Burst Trie, cont

- A burst trie is an in-memory data structure, designed for sets of records that each have a unique string that identifies the record and acts as a key.
- A string s with length n consists of a series of symbols or characters c_i for $i = 0, \ldots, n$, chosen from an alphabet A of size $|A|$. We assume that $|A|$ is small, typically no greater than 256.

Burst Trie Components

- A burst trie consists of three distinct components, a set of Records, a set of Containers, and an Access Trie:
- *Records.* A *record* contains a string; information as required by the application using the burst trie (i.e., for information such as word locations) and pointers as required to maintain the container holding the *record*. Each string is unique.

Burst Trie Components

- *Containers.* A *container* is a small set of records, maintained as a simple data structure such as a list or a BST.
- For a *container* at depth *k* in a burst trie (depth is discussed below), all strings have length at least k, and the first *k* characters of all strings are identical. It is not necessary to store these first *k* characters. Thus a particular *container* at depth 3 containing “author" and “automated" could also contain “autopsy" but not “auger".
- Each *container* also has a header, for storing the statistics used by heuristics for bursting.

Burst Trie Components

- *Access trie*. An *access trie* is a trie whose leaves are containers. Each node consists of an array *p*, of length $|A|$, of pointers each of which may point to either a trie node or a container, and a single empty-string pointer to a record. The $|Aj|$ array locations are indexed by the characters $c \in A$. The remaining pointer is indexed by the empty string. The depth of the root is defined to be 1. Leaves are at varying depths.

Burst Trie Components

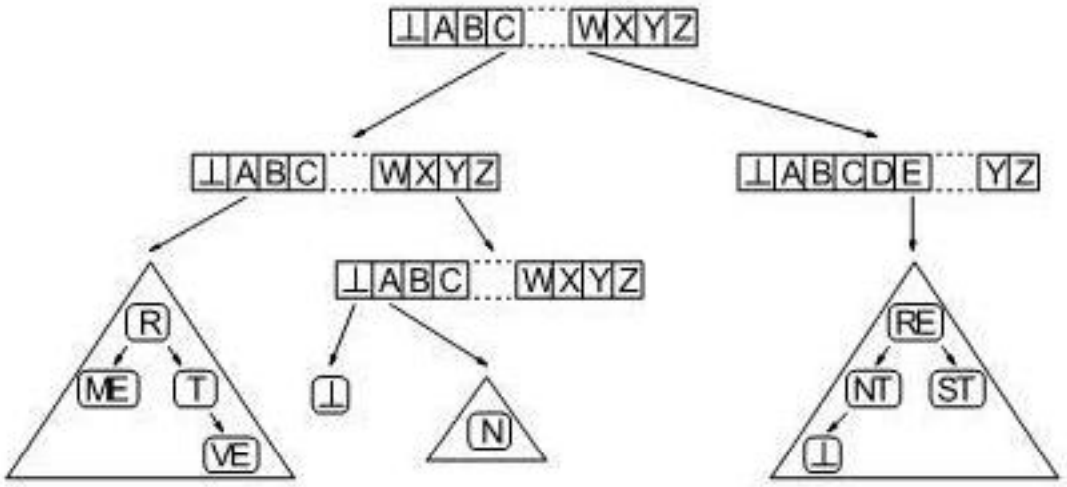


Figure 1: *Burst trie with BSTs used to represent containers.*

Figure 1 shows an example of a burst trie storing ten records whose keys are 'came', 'car', 'cat', 'cave', 'cy', 'cyan', 'we', 'went', 'were', and 'west' respectively.

Burst Trie Operations

- There are three main operations applied to the tree:
  - Searching
  - Insertion
  - Bursting

Burst Trie Operations, Searching

- The *access trie* is traversed according to the leading characters in the query string. Initially the current object is the root of the access trie and the current depth i is 1.
- While the current object is a trie node t of depth $i \leq n$,
  - (a) Update the current object to be the node or *container* pointed to by the c_i^{th} element of t's array p, and
  - (b) Increment i.
- If the current object is a trie node t of depth $i = n + 1$, the current object is the object pointed to by the empty-string pointer, which for simplicity can be regarded as a *container* of either zero or one *records*. Otherwise, if the current object is null the string is not in the burst trie, and search terminates.

Burst Trie Operations, Searching

- If $i \leq n$, use the remaining characters c_i , …,c_n to search the *container*, returning the *record* if found or otherwise returning null.
- Note that in many languages the most common words are typically short. Thus these terms are typically stored at an empty-string pointer, and are found after simply following a small number of access trie pointers with no search in a container at all.

Burst Trie Operations, Insertion

- Consider input of a new string $c_{1,\ldots,}c_n$ of length n
- Stage 1 of the search algorithm above is used to identify the *container* in which the *record* should be inserted. This *container*, which can be empty, is at depth k. For the special case of arriving at a trie node at depth $k = n + 1$, the *container* is under the empty-string pointer.
- If $k \leq n$, the standard insertion algorithm for the *container* data structure is used to add the record to the *container*, using only the suffix characters $c_{k+1,\ldots,}c_n$. Otherwise, the *record* is added under the empty-string pointer.

Burst Trie Operations, Insertion

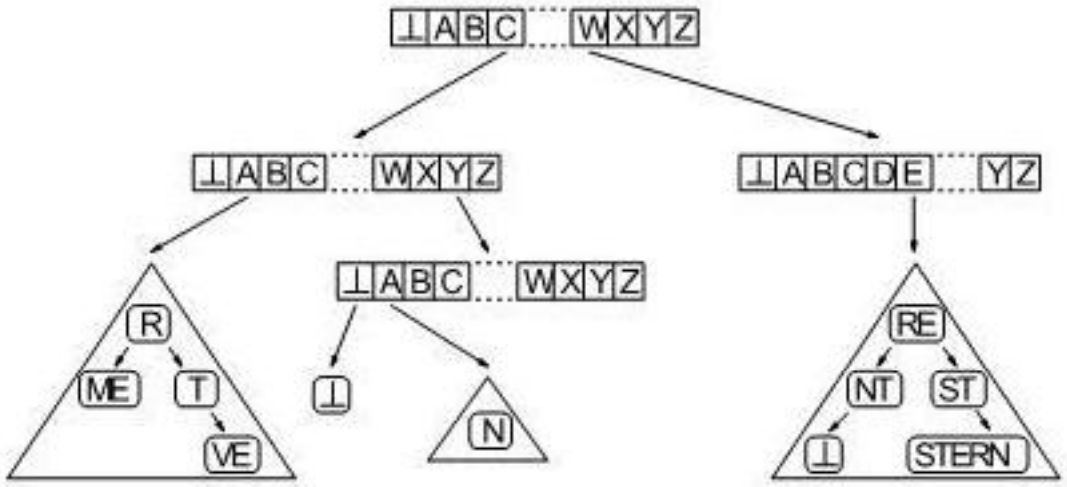
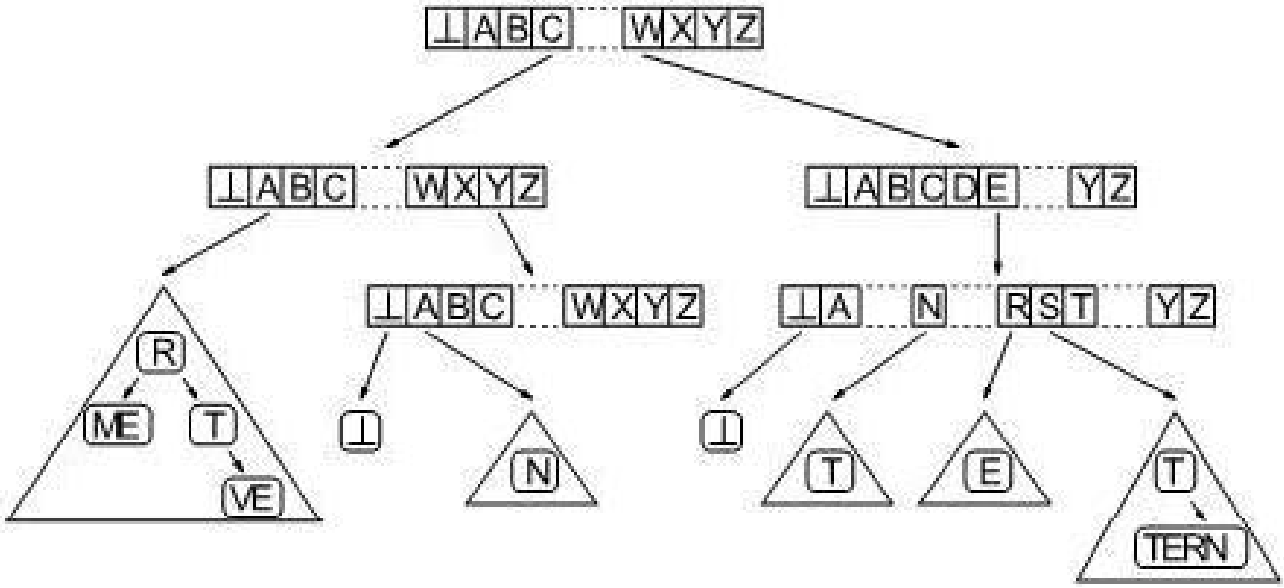


Figure 2: *Burst trie after insertion of a record with key "western".*

Figure 2 shows the example burst trie after the insertion of the record with key ‘western’.

Burst Trie Operations, Bursting

- Bursting is the process of replacing a *container* at depth *k* by a trie node and a set of new *containers* at depth *k*+1, which between them contain all the *records* in the original *container*.



Burst Trie Heuristics

- A successful bursting strategy should ensure that common strings are found quickly via the *access trie*, without excessive searching within *containers*.
- Appropriate management of *containers* (such as move-to-front in lists) should also be helpful in reducing search costs.
- *Containers* that are rarely accessed should not be burst, since *containers* should be more space-efficient than trie nodes.
- A goal of bursting is to transform *containers* that have high average search costs, without bursting unnecessarily.

Burst Trie Heuristics

- A design goal for the tested heuristics was that they could be cheaply evaluated.
  - They do not require a thorough inspection of a *container* to decide whether to burst.
  - Instead, the heuristics use counters for each *container*.
- The Three Heuristics tested
  - Ratio
  - Limit
  - Trend

Burst Trie Heuristics, Ratio

- The *Ratio* heuristic utilizes 2 counters to keep track of 2 values
  - the number of times a *container* has been searched
  - The number of searches that have ended successfully at the root node of that *container* (direct hits)
- We calculate the ratio *R* between the number of accesses and the number of *direct hits* for each *container*. This ratio is used to determine whether to burst.
- Potential Advantage
  - *R* is only tested once the total number of accesses exceeds some threshold, thereby *containers* that are rarely accessed are burst
- Potential Disadvantage
  - additional memory required to maintain two counters per *container*, and the number of tests required at each access.

Burst Trie Heuristics, Ratio

- *Ratio* Example
  - If less than, say, $R = 80\%$ of all accesses into a *container* are *direct hits* and there have been at least, say, 100 accesses to the *container* in total, it should be burst.

Burst Trie Heuristics, Limit

- The *Limit* heuristic bursts a *container* whenever it contains more than a fixed number *L* of *records*.
- Potential Advantages
  - By eliminating large *containers*, we limit total search costs.
  - Compared to *Ratio,* only a single counter is required, and this need only be tested on insertion.
  - Provides reasonable performance even if the distribution of strings is completely uniform.
- Potential disadvantages.
  - The burst trie with *Limit* is likely to be slow if there is a *container* with less than *K* terms but all are very common, so that there are many accesses to the *container* but not many direct hits. However, this can only arise if these terms and no others share a common prefix, not a probable occurrence.
  - *Containers* with only rare strings are burst.

Burst Trie Heuristics, Trend

- The *Trend* heuristic allocates each newly-created *container* a set amount of capital *C*.
- The current capital is modified on each access. On a direct hit, the capital is incremented by a bonus *B*. If a *record* is accessed that is already in the *container*, but is not a direct hit, the capital is decremented by a penalty *M*. When the capital is exhausted, the *container* is burst.
- Thus if the accesses are sufficiently skew, the bonuses will offset the penalties, and the *container* will not burst.
- The use of a start capital ensures that even *containers* with a uniform distribution will not burst until after a certain number of accesses.
- Testing revealed this good choice of *Trend* parameters: start capital *C* of 5000, bonus *B* of 4, and penalty *M* of 32.

Container Data Structures

- Data Structures should allow reasonable efficient insertion and search for *records* over small data sets. This study considers:
  - Linked Lists
  - Binary Search Trees
  - Splay Trees

Container Data Structures

- Data Structures should allow reasonable efficient insertion and search for records over small data sets. This study considers:
  - Linked Lists
  - Binary Search Trees
  - Splay Trees
- (Hash tables were not considered as they are not efficient for small numbers of items.)

Container Data Structures, Linked List

- Advantages
  - Low overhead = space efficient

- Disadvantages
  - Access costs are high in unordered list
    - Solution: Move-to-Front list (most recently accessed node moved to front position. Provides adaptation to local changes in vocabulary.

Container Data Structures, BST

- Advantages
  - Shorter average search paths than list
- Disadvantages
  - Position based on order of insertion, not frequency
    - Solution: when a BST is burst, sort the records by decreasing frequency, then distribute to the new *containers*.

Container Data Structures, Splay Trees

- Advantage
  - Since frequently-accessed *records* should be kept near the root, the structure provides adaptation to changing vocabulary and, in common with move-to-front lists, may lead to less bursting.

- Disadvantage
  - For efficient splaying, a record in a splay tree requires three pointers, two to its children and one to its parent. Thus they use the most space of any of the *container* structures we consider.

Experiments

Test Data, Text

•The data sets drawn from the large Web track in the TREC (Text Research Conference) project. The five groups contain collections of web pages extracted from the Internet Archive for use in TREC retrieval experiments.

•These collections show the skew distribution that is typical of text.

•A word, in these experiments, is an alphanumeric string containing no more than two integers and not starting with an integer.

•All words are converted to lower case. XML and HTML markup, special characters, and punctuation are skipped.

Table 1: *Statistics of text collections drawn from TREC.*

	Web S	Web M	Web L	Web XL	Web XXL	TREC1
Size (Mb)	100	2,048	10,240	15,360	45,229	1,206
Distinct words	114,109	1,212,885	3,604,125	4,617,076	8,634,056	503,242
Word occurrences ($\times 10^6$)	6	131	631	963	2,704	179
Documents	18,726	347,075	1,780,983	2,680,922	7,904,237	510,634
Parsing time (sec)	2.4	49.4	238.0	363.9	1,014.0	69.3

Test Data, Non-text

- Non-text collections used for comparison
  - Genomic Data: a collection of nucleotide strings, each typically thousands of nucleotides in length, with an alphabet of 4 characters . It is parsed into shorter strings by extracting n-grams of length 9. (Such 9-grams are commonly used by genomic search utilities to locate regions where a longer inexact match may be found.)
  - Music Data: consisting of pieces in MIDI format stored in a textual representation, using an alphabet of 27 characters. Extract n-grams of length 7 from the music data, an approach used in indexing MIDI files.
- These n-grams are extracted by sliding a window over the data and taking every consecutive block of *n* characters.

Test Data, Non-text

- Table 2 shows the statistics of the non-text collections. They do not show the skew distribution that is typical of text.

Table 2: *Statistics of non-text collections.*

	Genomic 9-grams	Music 7-grams
Size (Mb)	1,993	20
Distinct n-grams	262,146	3,147,961
N-gram occurrences ($\times 10^6$)	1,877	20
Documents	2,509,087	67,528
Parsing time (sec)	1,365.0	11.7

This low-skew data sets are expected to perform poorly using Burst Tries

Methodology

- Goals
- Explore the different burst heuristics and choices of container structure to identify which yield good performance,
- Compare burst tries to the other principal data structures for vocabulary accumulation in terms of running time and memory usage.
- The data structure stores for each term its total occurrence frequency and the number of documents in which it occurs.

Reference Data Structures

- Measurements of time and space requirements of five reference data structures: compact tries (which we simply refer to as tries), TSTs, BSTs, splay trees, and hash tables.
- For splay trees, two variants of splaying are tested: where the tree is splayed at every access, and where it is splayed intermittently (at every fourth access)
- For hash tables, a hash with 2^{20} slots and a bit-wise hash function was used.

Reference Data Structures

Table 3: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of each text collection, for each reference data structure.*

	Web S	Web M	Web L	Web XL	Web XXL	TREC1
Trie	2.2 (23)	54.4 (246)	—	—	—	68.7 (100)
TST	3.1 (9)	80.0 (84)	386.2 (246)			94.8 (35)
BST	5.9 (4)	148.0 (39)	740.2 (118)	1,129.0 (151)	3,230.3 (285)	171.6 (16)
Splay, all acc.	6.7 (4)	176.0 (44)	881.5 (131)	1,336.8 (169)	4,065.1 (318)	215.7 (18)
Splay, int.	5.7 (4)	151.0 (44)	753.9 (133)	1,145.4 (169)	3,331.1 (318)	174.9 (18)
Hashing	1.7 (6)	44.1 (38)	218.0 (108)	323.9 (138)	921.4 (256)	48.4 (18)

Table 4: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of the non-text collections, for each reference data structure.*

	Genomic 9-grams	Music 7-grams
Trie	2,553.4 (7.7)	—
TST	4,200.6 (12.3)	35.9 (179.9)
BST	9,940.7 (8.5)	95.6 (96.1)
Splay, all acc.	15,385.2 (9.5)	78.0 (108.1)
Splay, int.	13,316.4 (9.5)	88.5 (108.1)
Hashing	1,940.3 (11.5)	43.3 (88.1)

Results, TREC1 RATIO

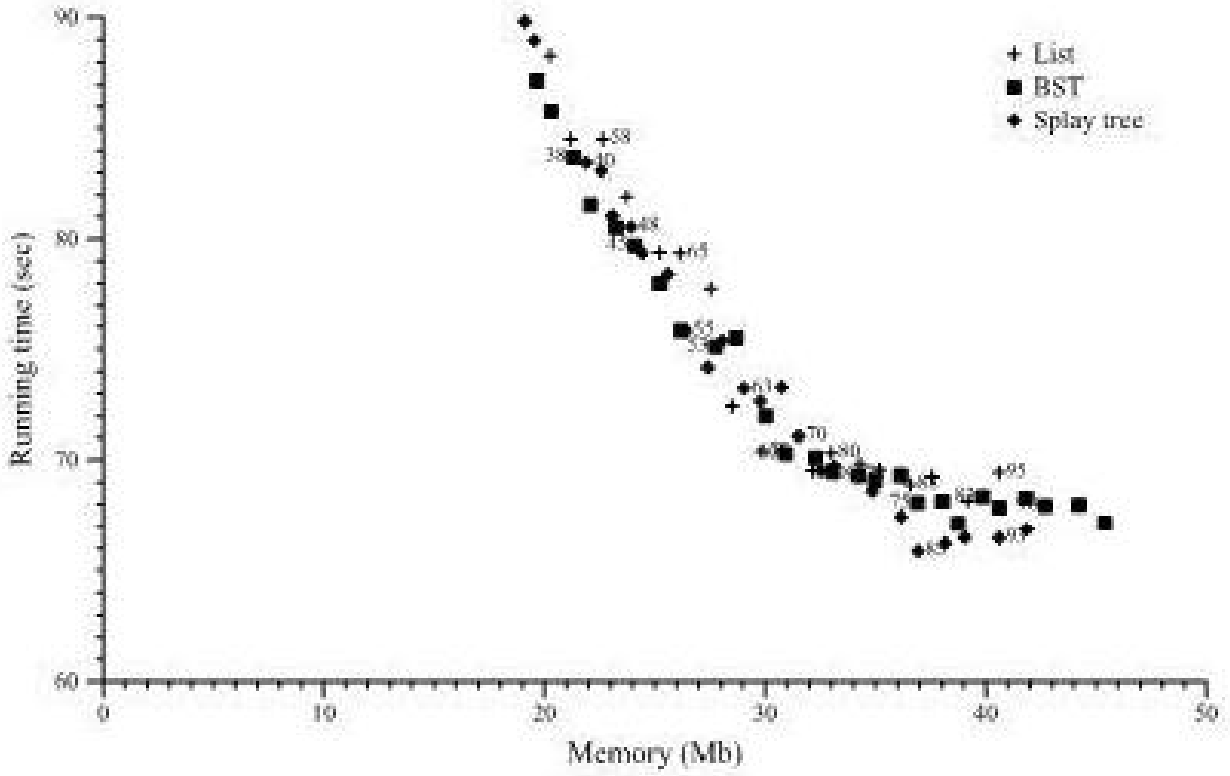


Figure 4: *Space and time for burst trie with* Ratio *burst heuristic and a variety of parameters, on* TREC1. *The parameter value is shown to the side of every third point.*

Results, WEB M RATIO

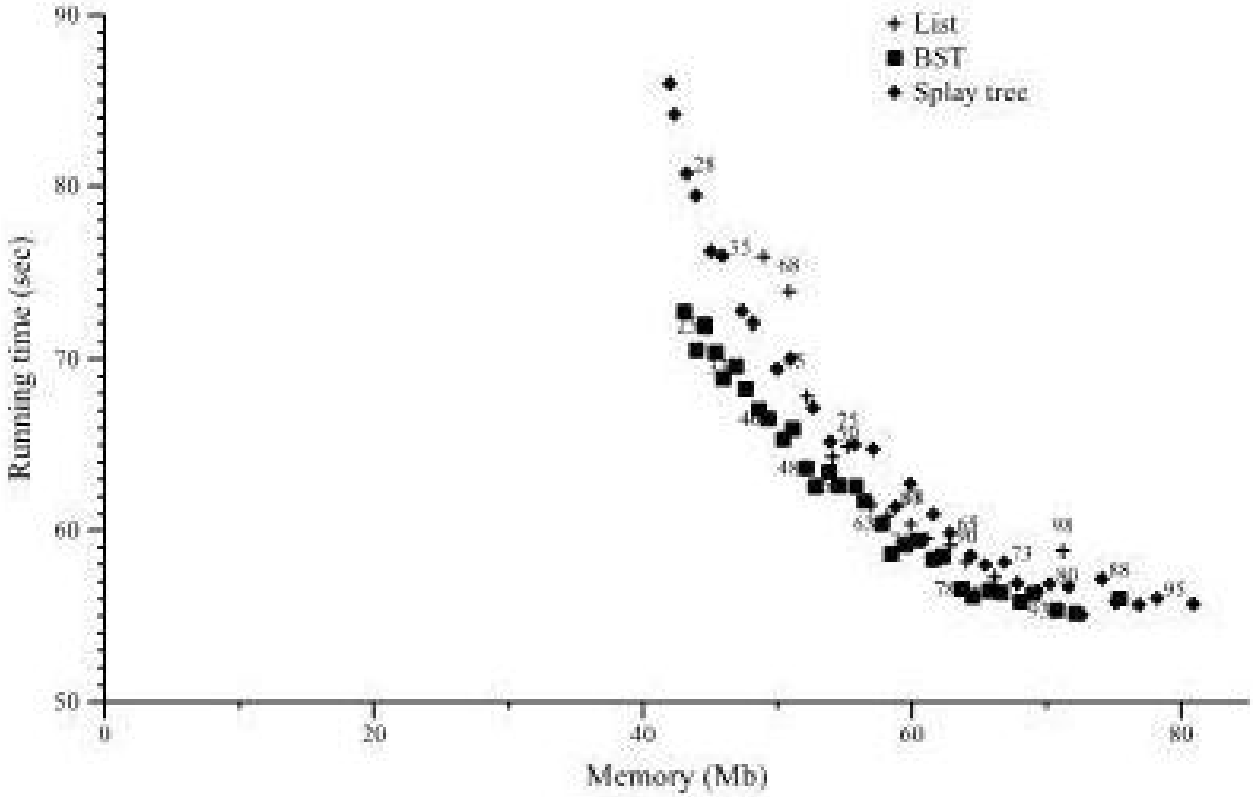
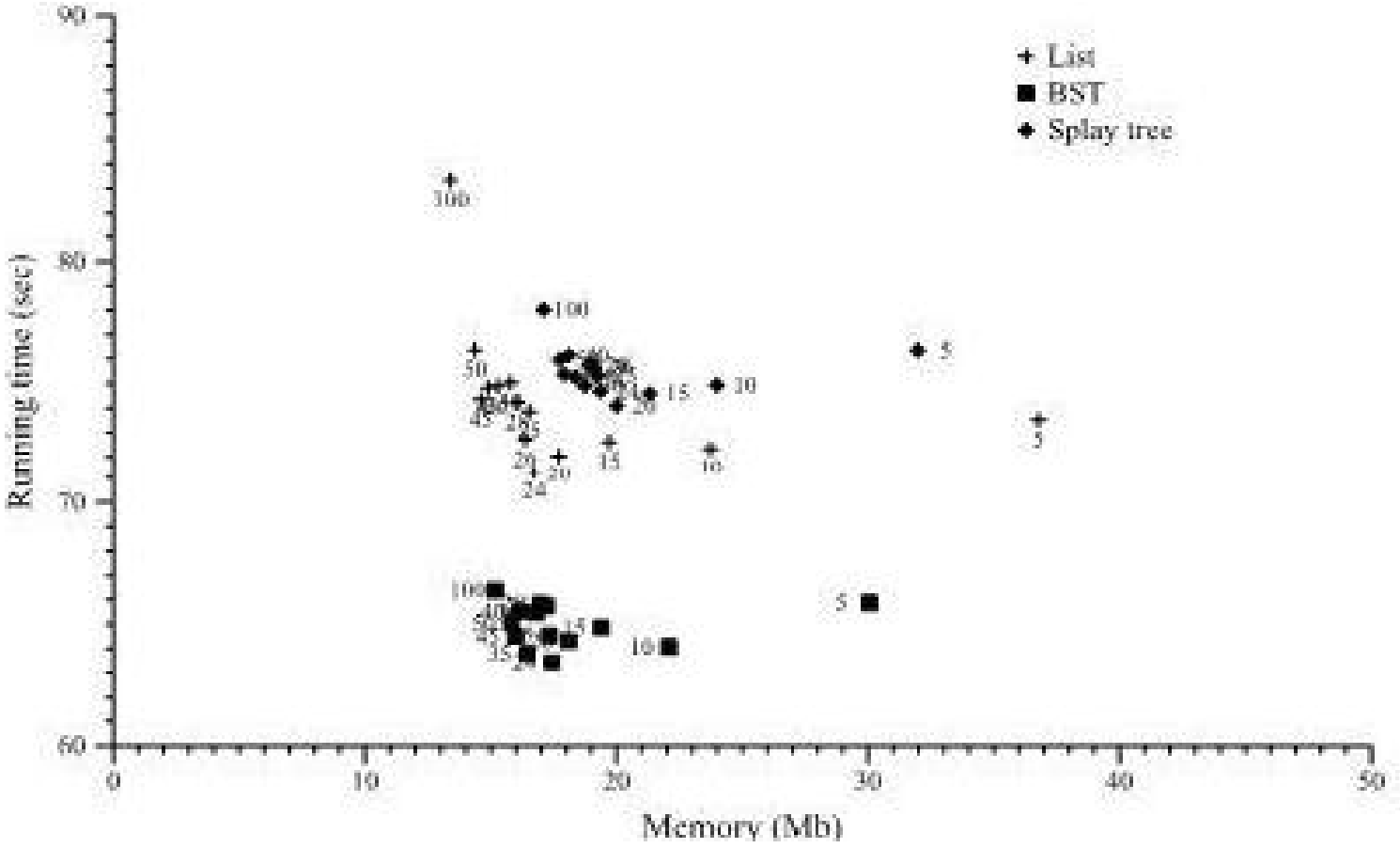


Figure 5: *Space and time for burst trie with* Ratio *burst heuristic and a variety of parameters, on* Web M. *The parameter value is shown to the immediate right of every third point.*

Results, RATIO

- As container structures, lists, BSTs and splay trees all yield a similar curve.
- However, the ratios that achieve particular points on the curve vary. For example, a BST with ratio 45 is about as efficient as a list with ratio 65. This is because a list of *n* nodes uses less space than a BST of *n* nodes, but is less efficient; lists require a higher ratio for the same efficiency.
- Similar results were obtained from the other 3 data sets

Results, TREC1 LIMIT



Results, WEB M LIMIT

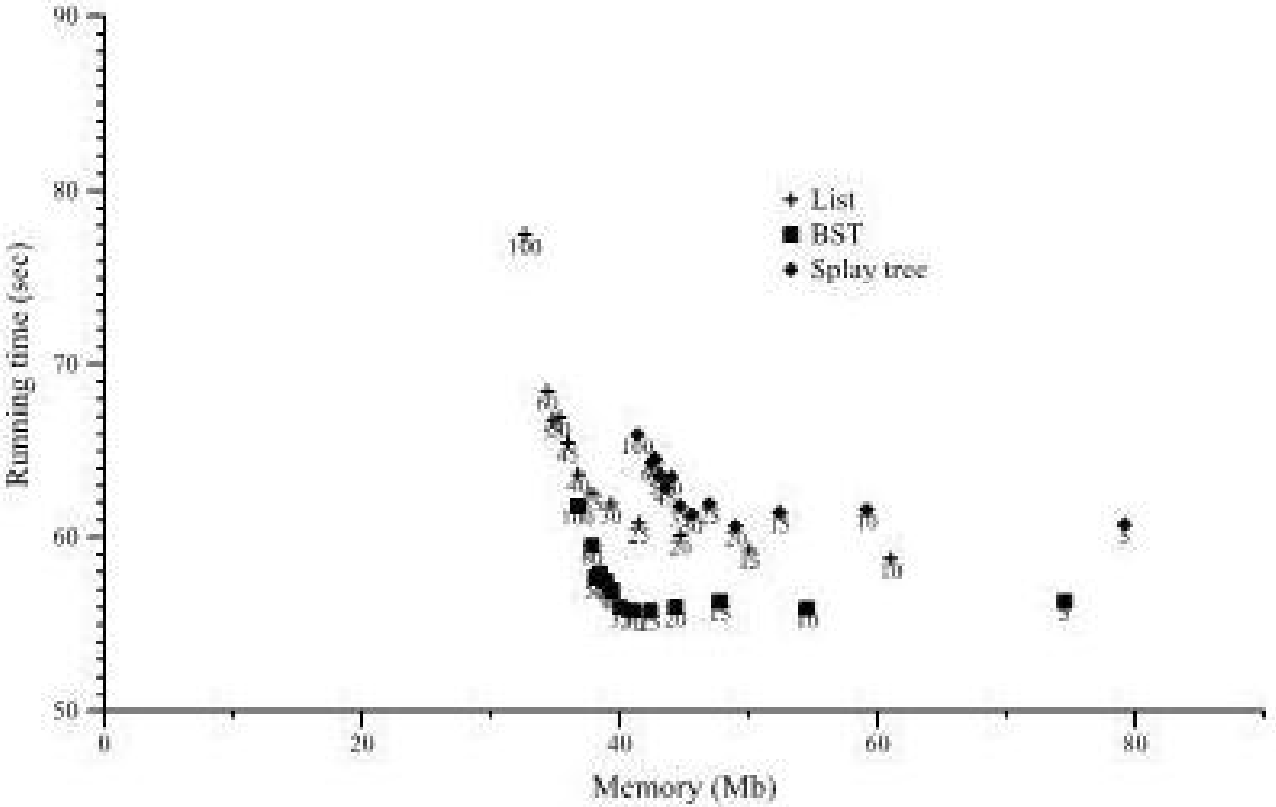


Figure 7: *Space and time for burst trie with* Limit *burst heuristic and a variety of parameters, on* Web M. *The parameter value is shown immediately below every point.*

Results, LIMIT

- For all but extremely small containers, *Limit greatly outperforms Ratio* for all container structures.
- Also, BST containers are faster for given memory than either list or splay tree containers.
- The best results with *Limit show that burst tries can be much more ecient than any* other tree or trie structure. For example, the slowest run with BSTs as containers with limit *L = 100 uses only 15 Mb and requires 66 seconds. This is less memory than any existing tree* structure tested, even standard BSTs, and is faster than tries and almost three times faster than BSTs or splay trees. Only hashing is faster, by about 27%.
- Results for BST containers with the *Limit* heuristic and all text data sets show that the gains achieved by burst tries, in comparison to the reference data structures, are consistent for all these data sets.

Results, LIMIT, cont

- Results for the non-text data sets show that with the genomic data the burst trie did not do well: it is slower than the TST and half the speed of hashing. Nonetheless, it is still much faster than the other tree structures, and is the most compact of all data structures tested. With the music data the burst trie showed the best observed performance observed, faster than any other data structure tested including hash tables with good space utilization.

Table 7: *Running time in seconds (in parentheses, peak memory usage in megabytes) to accumulate the vocabulary of the non-text collections, for burst tries with BST containers and the* Limit *heuristic.*

	Genomic 9-grams	Music 7-grams
$L = 35$	4333.2 (7.3)	29.2 (92.7)
$L = 100$	5475.9 (7.3)	36.2 (87.6)

Results TREND TREC 1

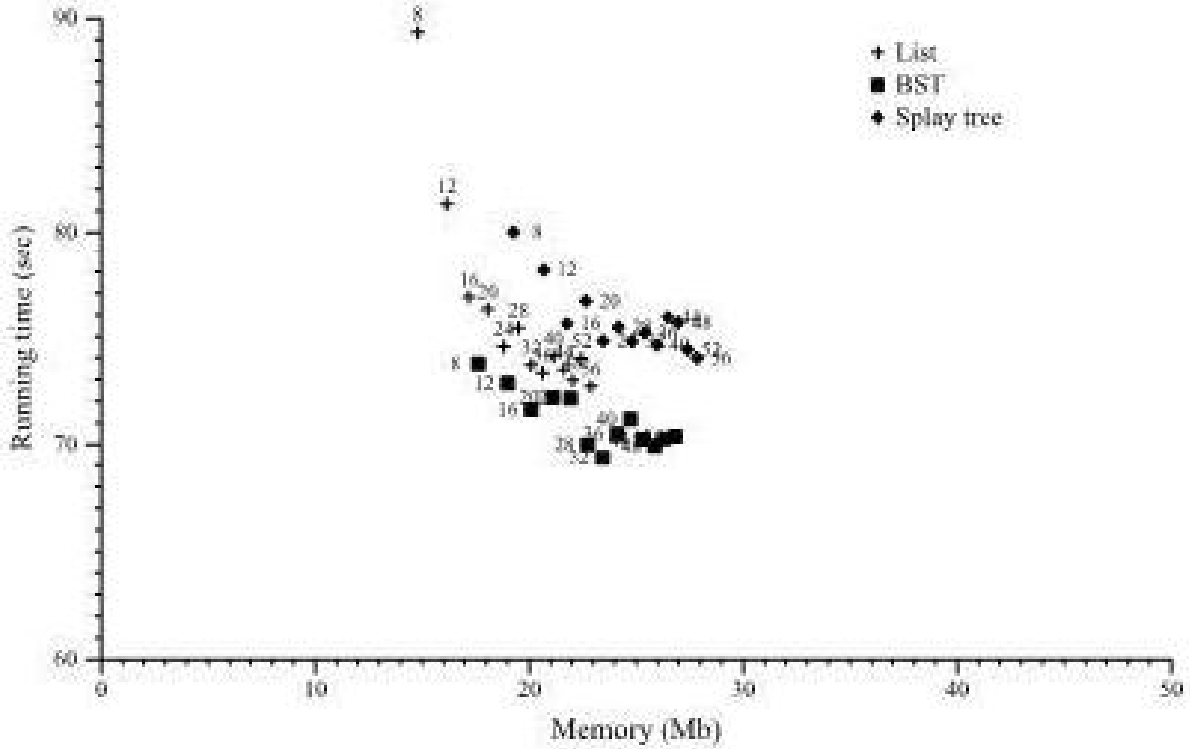


Figure 9: *Space and time for burst trie with* Trend *burst heuristic and a variety of container structures, on* TREC1. *The parameter value M is shown immediately adjacent every point; we fixed $C = 5000$ and $B = 4$.*

Results TREND

- Results on TREC 1 are typical of all text data sets where *M* is varied and the other parameters are held constant.
- It was expected that the *Trend* heuristic would be the best of all, as it was likely to lead to containers being burst more rapidly if they held common terms, and more slowly otherwise, but the results do not approach those of *Limit*.
- Despite wide investigation, further testing did not identify combinations of parameters with significantly better performance than that shown in these graphs.

Conclusions

- Burst Tries are highly efficient for managing large sets of strings in memory.
- Use of Containers allow dramatic space reductions with no impact on efficiency.
- Experiments show that Burst Tries are
  - Faster than compact tries, using 1/6 of the space
  - More compact than binary trees or splay trees and are over
  - two times faster
  - Close to hash tables in efficiency, yet keep the data in sort order.
- The Burst Trie is dramatically more efficient than any previously known structure for the task of managing sorted strings.

Resources

1. Burst Tries: A Fast, Efficient Data Structure for String Keys Heinz, Zobel, Williams; School of Computer Science and Information Technology, RMIT University (Australia) http://www.cdf.toronto.edu/~csc148h/fall/assignment3/bursttries.pdf
2. Data Structures and Algorithms in Java (4th edition), John Wiley & Sons, Inc., 2004. Michael T. Goodrich and Roberto Tamassia http://ww3.algorithmdesign.net/handouts/Tries.pdf
3. D. Harman. Overview of the second text retrieval conference (TREC-2). *InformationProcessing &Management, 31(3):271{289, 1995.*