INSTRUCT: Space-Efficient Structure for Indexing and Complete Query Management of String Databases

Sourav Dutta, Arnab Bhattacharya
sodutta3@in.ibm.com, arnabb@iitk.ac.in

IBM Research Lab, New Delhi
Dept. of Computer Science and Engineering, Indian Institute of Technology, Kanpur

COMAD
8th December, 2010

Motivation

- Explosion of sequence data
- String databases are reaching terabytes of storage
- Modern applications no longer limited to exact string matching
- They demand intelligent prefix, suffix and substring search facilities
- Current storage and indexing techniques fail to cater to all these issues simultaneously
- They also waste storage space by not reusing the common characters
- We introduce INSTRUCT
  - INdexing STrings by Re-Using Common Triplets

Existing structures

- Hash tables
  - Fastest exact search operation
  - However, do not support other search operations

Existing structures

- Hash tables
  - Fastest exact search operation
  - However, do not support other search operations
- Binary and ternary search trees (BST and TST)
  - Efficiently supports exact string search only

Existing structures

- Hash tables
  - Fastest exact search operation
  - However, do not support other search operations
- Binary and ternary search trees (BST and TST)
  - Efficiently supports exact string search only
- Suffix and prefix trees
  - Efficiently supports exact as well as prefix/suffix search
  - Fails for substring search

Existing structures

- Hash tables
  - Fastest exact search operation
  - However, do not support other search operations
- Binary and ternary search trees (BST and TST)
  - Efficiently supports exact string search only
- Suffix and prefix trees
  - Efficiently supports exact as well as prefix/suffix search
  - Fails for substring search
- Trie family
  - Re-uses space, but only for common prefix
  - Does not support substring search
  - May use dictionary compression to reduce storage
  - May use merging of buckets to re-use space

Existing structures

- Hash tables
  - Fastest exact search operation
  - However, do not support other search operations
- Binary and ternary search trees (BST and TST)
  - Efficiently supports exact string search only
- Suffix and prefix trees
  - Efficiently supports exact as well as prefix/suffix search
  - Fails for substring search
- Trie family
  - Re-uses space, but only for common prefix
  - Does not support substring search
  - May use dictionary compression to reduce storage
  - May use merging of buckets to re-use space
- *n*-gram indexing
  - Expensive merge operations for generating results

Structure of INSTRUCT

- Keys are composed from an alphabet set Σ of size k
- Maximum length of any key is l
- Basic idea is to index *triplets* or *3-grams*
- Collection of k nodes, each corresponding to a character in Σ
- Each node consists of a $k \times k$ matrix
- Cell in node c_1 at row c_2 and at column c_3 represents the triplet $c_1 c_2 c_3$
- When a particular triplet is present in a key, the corresponding cell is marked

Structure of INSTRUCT

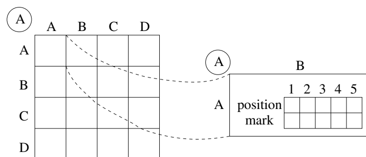
- Keys are composed from an alphabet set Σ of size k
- Maximum length of any key is l
- Basic idea is to index *triplets* or *3-grams*
- Collection of k nodes, each corresponding to a character in Σ
- Each node consists of a $k \times k$ matrix
- Cell in node c_1 at row c_2 and at column c_3 represents the triplet $c_1 c_2 c_3$
- When a particular triplet is present in a key, the corresponding cell is marked
- Position information of a triplet in a key is also incorporated
- Each cell is broken into an array of size l to denote which position the triplet occurs
- This is called the position array

Implementation of INSTRUCT

- Implemented as bit vectors – regular 4-dimensional array of size $k^3 l$
- Bit operations are faster and easier

Implementation of INSTRUCT

- Implemented as bit vectors – regular 4-dimensional array of size $k^3 l$
- Bit operations are faster and easier
- When a particular bit at node c_1, row c_2, column c_3 and position w is set, it indicates that there exists a key in the database with the triplet $c_1 c_2 c_3$ at position w



Mark array

- INSTRUCT structure by itself does not disambiguate among all keys

Mark array

- INSTRUCT structure by itself does not disambiguate among all keys
- Consider strings 'ABCA' and 'DBCD' to be present
  - P[A][B][C][1] = P[B][C][A][2] = P[D][B][C][1] = P[B][C][D][2] = 1
- Searching for strings 'ABC' and 'ABCD' would return a false positive as P[A][B][C][1] and P[B][C][D][2] bits are appropriately set
- This problem occurs as the history regarding the key(s) of which a triplet is a part of, is lost

Mark array

- INSTRUCT structure by itself does not disambiguate among all keys
- Consider strings 'ABCA' and 'DBCD' to be present
  - P[A][B][C][1] = P[B][C][A][2] = P[D][B][C][1] = P[B][C][D][2] = 1
- Searching for strings 'ABC' and 'ABCD' would return a false positive as P[A][B][C][1] and P[B][C][D][2] bits are appropriately set
- This problem occurs as the history regarding the key(s) of which a triplet is a part of, is lost
- To *alleviate* the problem, another l element bit array called mark is used
- A set bit in mark implies that there exists at least one key that *ends* at that position with that triplet
  - Thus, M[B][C][A][2] = M[B][C][D][2] = 1
  - Search for 'ABC' is correctly reported as false now
  - Search for 'ABCD' is still returned as a false positive

Container

- When a mark bit is set, a container is allocated to hold the keys ending at that position with that particular triplet
  - Container with M[B][C][D][2] = 1 stores the string 'DBCD'
  - Container with M[B][C][A][2] = 1 stores the string 'ABCA'
  - So, search for 'ABCD' will now return false
- This guarantees completely accurate results – no false positives or false negatives

Container

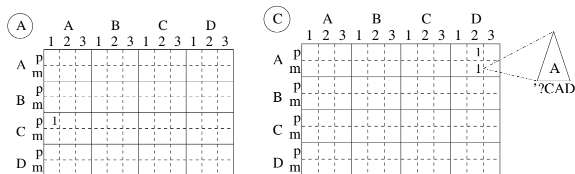
- When a mark bit is set, a container is allocated to hold the keys ending at that position with that particular triplet
  - Container with M[B][C][D][2] = 1 stores the string 'DBCD'
  - Container with M[B][C][A][2] = 1 stores the string 'ABCA'
  - So, search for 'ABCD' will now return false
- This guarantees completely accurate results – no false positives or false negatives
- Container can be
  - Binary search tree (BST): faster searching, slower insertion
  - List: slower searching, faster insertion

Analysis of INSTRUCT structure

- Main indexing structure requires only $2k^3l$ bits
- All possible strings of length l are indexed without increasing this size
- Bit implementation enables the use of standard bit manipulation operations like RIGHT SHIFT, AND, etc., thereby making them efficient
- For very large databases, if the complete index structure does not fit into the main memory, the various nodes of INSTRUCT can be stored on disks and can be independently fetched and processed for various triplets
- For keys of length 1 and 2 having no well defined triplets, special containers are maintained

Insertion

- Insertion is done by simply setting the bits corresponding to each triplet and each position
- The final triplet also sets the corresponding mark bit and the string is inserted into the corresponding container



Insertion of key 'ACAD'

First triplet ('ACAD') Last triplet ('ACAD')

Exact string search

- Exact search is done similarly by checking the bits corresponding to each triplet and each position
- If any bit is unset, the key is reported to be absent
- Even if all the bits are set, the final container needs to be searched
- The above procedure is called index strategy

Exact string search

- Exact search is done similarly by checking the bits corresponding to each triplet and each position
- If any bit is unset, the key is reported to be absent
- Even if all the bits are set, the final container needs to be searched
- The above procedure is called index strategy
- Alternatively, *only* the mark bit corresponding to the final triplet can be checked
- If it is unset, the key is absent; otherwise, the container is searched
- This procedure is called direct strategy

Exact string search

- Exact search is done similarly by checking the bits corresponding to each triplet and each position
- If any bit is unset, the key is reported to be absent
- Even if all the bits are set, the final container needs to be searched
- The above procedure is called index strategy
- Alternatively, *only* the mark bit corresponding to the final triplet can be checked
- If it is unset, the key is absent; otherwise, the container is searched
- This procedure is called direct strategy
- Direct strategy is better when
  - Size of database is large as most bits are then likely to be set
- Index strategy is better when
  - Length of key is large as then chances of hitting a negative is more
  - Size of alphabet is large as then chances of hitting the same character, and therefore, the same triplet, is less

Analysis of exact string search

- Key of length n
- For index strategy, container will be searched if and only if for all corresponding triplets and positions, the bits are set
- In other words, the database contains all such keys

Analysis of exact string search

- Key of length n
- For index strategy, container will be searched if and only if for all corresponding triplets and positions, the bits are set
- In other words, the database contains all such keys
- Number of keys in database having length at least w is $f(w)$
- Assume all characters to be equi-probable (having probability $1/k$)
- Probability that at least one key contains character c_1 at position w is

$$\begin{aligned} P_w &= 1 - P(\text{no key contains } c_1) \\ &= 1 - (P(\text{key contains character other than } c_1))^{f(w)} \\ &= 1 - (1 - 1/k)^{f(w)} \end{aligned} \quad (1)$$

Analysis of exact string search (contd.)

- Probability that triplet $c_1 c_2 c_3$ occurs at position w is

$$\begin{aligned} P_{w,3} &= P_w . P_{w+1} . P_{w+2} \\ &= \left(1 - (1 - 1/k)^{f(w)}\right) . \left(1 - (1 - 1/k)^{f(w+1)}\right) . \\ &\qquad \left(1 - (1 - 1/k)^{f(w+2)}\right) \\ &\simeq 1 - \sum_{i=w}^{w+2} (1 - 1/k)^{f(i)} \text{ [ignoring higher order terms]} \qquad (2) \end{aligned}$$

Analysis of exact string search (contd.)

- The last bit must be set in the mark array as well
- Number of keys in database having length exactly w is $g(w)$
- Probability that at least one key ends at character c_1 at position w is

$$P_{w_e} = 1 - (1 - 1/k)^{g(w)} \tag{3}$$

- Probability that triplet $c_1 c_2 c_3$ ends at position w is

$$P_{w_e,3} \simeq 1 - \sum_{i=w}^{w+1} (1 - 1/k)^{f(i)} - (1 - 1/k)^{g(w+2)} \tag{4}$$

Analysis of exact string search (contd.)

- Probability that all triplets of the search key are present in the database at corresponding positions is

$$P_n = \left(\prod_{j=1}^{n-3} P_{j,3} \right) . P_{n-2_e,3}$$

$$= \prod_{j=1}^{n-3} \left(1 - \sum_{i=j}^{j+2} (1 - 1/k)^{f(i)} \right) .$$

$$\left(1 - \sum_{i=n-2}^{n-1} (1 - 1/k)^{f(i)} - (1 - 1/k)^{g(n)} \right)$$

$$\simeq 1 - \sum_{j=1}^{n-2} \sum_{i=j}^{j+2} (1 - 1/k)^{f(i)} + (1 - 1/k)^{f(n)} - (1 - 1/k)^{g(n)} \quad (5)$$

Analysis of exact string search (contd.)

- Since each of the $f(i)$ and $g(i)$ terms are bounded by m, P_n can be upper bounded as

$$
\begin{aligned}
P_n &\leq 1 - \sum_{j=1}^{n-2} \sum_{i=j}^{j+2} (1 - 1/k)^m \\
&= 1 - 3(n-2)\,(1 - 1/k)^m \qquad (6)
\end{aligned}
$$

Expected running time

- Assume that search through index structure requires T_s time
- Search through container requires T_c time
- With probability P_n, a container is searched
- Therefore, expected running time is

$$T_i = (1 - P_n) T_s + P_n (T_s + T_c) \tag{7}$$

Expected running time

- Assume that search through index structure requires T_s time
- Search through container requires T_c time
- With probability P_n, a container is searched
- Therefore, expected running time is

$$T_i = (1 - P_n) T_s + P_n (T_s + T_c) \tag{7}$$

- The alternate direct strategy checks for the last mark bit only
- If it is set, a container is searched
- Therefore, expected running time is

$$T_d = P_{n-2_e,3} T_c \tag{8}$$

Comparison

- It is beneficial to search through the index structure when

$$
\begin{aligned}
T_i &\leq T_d \\
\text{or, } T_s &\leq (P_{n-2_e,3} - P_n) T_c \\
\text{or, } \frac{T_s}{T_c} &\leq 3(n-3)\left(1 - \frac{1}{k}\right)^m
\end{aligned} \tag{9}
$$

- Thus, index strategy is better when
  - n increases
  - k increases
  - m decreases
- Conforms with intuition and experiments

Suffix search

- A suffix search is almost same as exact string search
- However, since length of key containing suffix is not known, all possible lengths must be searched
- Suppose query suffix is $c_1 c_2 \dots c_f$
- If mark bit for last triplet is set at position p, then position bit for last but one triplet must be set at exactly $p - 1$
- Number of such positions p may be more than one

Suffix search

- A suffix search is almost same as exact string search
- However, since length of key containing suffix is not known, all possible lengths must be searched
- Suppose query suffix is $c_1 c_2 \dots c_f$
- If mark bit for last triplet is set at position p, then position bit for last but one triplet must be set at exactly $p - 1$
- Number of such positions p may be more than one
- Efficiently implemented using bit vector operations
    - Suppose mark array for last triplet is L
    - Position array for last but one triplet is RIGHT SHIFT-ed and then AND-ed with L
    - Containers corresponding to bits still set are searched

Prefix search

- Key idea: Prefix is *reverse* of suffix
- All database keys are reversed and stored in a separate INSTRUCT structure
- This is called the *reverse* INSTRUCT structure
- This structure is invoked only for prefix search, and therefore, can be maintained on disk

Substring Search

- Key idea: Any substring, when sufficiently shifted, becomes a prefix

Substring Search

- Key idea: Any substring, when sufficiently shifted, becomes a prefix
- Extra $l - 1$ reverse INSTRUCT structures
- i^{th} structure stores the original key shifted by i places
- However, containers store the entire key to facilitate returning the key
- Substring search now maps to prefix searches in these extra structures
- For the last triplet, only the position bit and not the mark bit is checked

Substring Search

- Key idea: Any substring, when sufficiently shifted, becomes a prefix
- Extra $l - 1$ reverse INSTRUCT structures
- i^{th} structure stores the original key shifted by i places
- However, containers store the entire key to facilitate returning the key
- Substring search now maps to prefix searches in these extra structures
- For the last triplet, only the position bit and not the mark bit is checked
- Space requirement increases by a factor of l
- Reverse structures are brought to memory only on demand

Substring Search

- Key idea: Any substring, when sufficiently shifted, becomes a prefix
- Extra $l - 1$ reverse INSTRUCT structures
- i^{th} structure stores the original key shifted by i places
- However, containers store the entire key to facilitate returning the key
- Substring search now maps to prefix searches in these extra structures
- For the last triplet, only the position bit and not the mark bit is checked
- Space requirement increases by a factor of l
- Reverse structures are brought to memory only on demand
- Expected number of prefix searches for a substring query of length s is

$$E_{prefix} \leq l \times \left(1 - 3(s-2)\left(1 - 1/k\right)^m\right) \tag{10}$$

Experimental setup

- Two real datasets
  1. English dictionary
  2. Protein sequences
- Synthetic datasets generated by controlling following parameters
  1. Total number of keys, m
  2. Size of alphabet, k
  3. Largest key length, l
  4. Length of query string, n
  5. Probability distribution of characters – uniform or Zipfian
- Out of total keys generated, 2/3rd is inserted
- Rest 1/3rd is used to trigger unsuccessful searches
- Half of keys inserted, i.e., 1/3rd of total is used to trigger successful searches

Real datasets

Dataset	Keys m	Symbols k	Length l	Total number of characters	Container size		False positive
					Max.	Avg.	
English dictionary	179,935	26	45	1,198,635	601	7.5	0.019
Protein sequences	38,627	21	2512	5,846,331	205	1.3	0.161

- False positive measures the rate when a container is searched for an unsuccessful key
- For dictionary dataset, the index structure prunes almost all unsuccessful searches

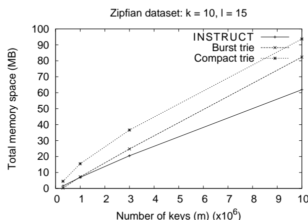
Real datasets

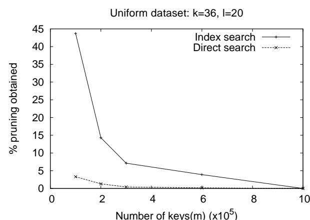
Index structure	Total memory	Time to insert	Searching time		
			Succ	Unsucc	Total
INS. BST	1.50 MB	1.42 s	0.51 s	0.54 s	1.05 s
INS. List	1.50 MB	1.29 s	0.59 s	0.58 s	1.17 s
Burst tr.	1.53 MB	1.61 s	0.64 s	0.66 s	1.30 s
Compact tr.	2.38 MB	1.82 s	0.65 s	0.65 s	1.31 s

Index structure	Total memory	Time to insert	Searching time		
			Succ	Unsucc	Total
INS. BST	15.73 MB	4.89 s	2.28 s	2.21 s	4.49 s
INS. List	15.73 MB	4.66 s	2.44 s	2.16 s	4.60 s
Burst tr.	15.89 MB	5.64 s	2.64 s	2.67 s	5.31 s
Compact tr.	25.71 MB	9.29 s	2.70 s	2.37 s	5.07 s

- INSTRUCT has lower storage requirements
- It also requires lesser running time
- Choice of containers does not matter when average size of container is low

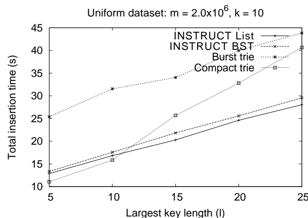
Effect of number of keys

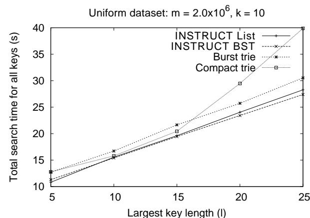




- INSTRUCT requires the least amount of memory
- Size grows linearly with number of keys due to size of containers
- Pruning for index search is high for small number of keys
- Pruning for direct search is always low

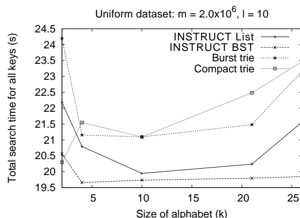
Effect of largest key length

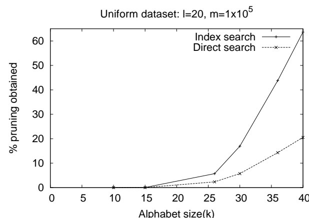




- INSTRUCT requires setting of bits only and is, therefore, faster
- With increase in key length, INSTRUCT can prune unsuccessful searches more and is, therefore, faster

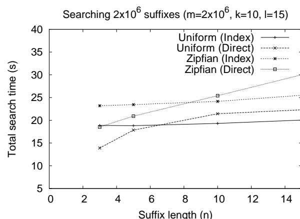
Effect of size of alphabet

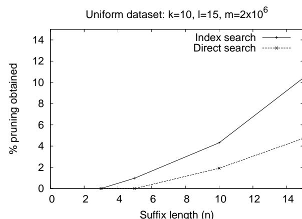




- For small alphabets, there is practically no pruning and container sizes are extremely large
- This leads to higher searching time
- Pruning increases exponentially with size of alphabet

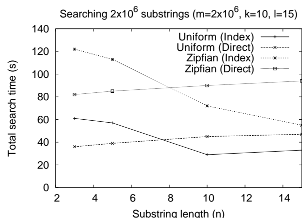
Effect of query length on suffix search

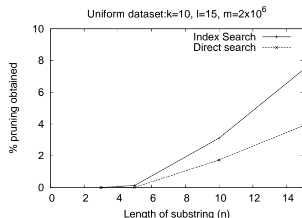




- For short suffixes, direct strategy performs better as it bypasses searching through the INSTRUCT structure
- Index strategy performs better as length of suffix increases

Effect of query length on substring search





- Similar behavior as in suffix search
- Absolute running times are higher
- Absolute pruning ratios are smaller

Summary of experiments

- For an expanding database, container should be implemented as a list; otherwise, BST is better
- For large databases of more than 10^6 keys, direct strategy is better than index strategy
- When query string length is more than 9 or alphabet size is more than 15, index strategy performs better
- INSTRUCT has better or comparable running times with other competing structures
- In general, INSTRUCT is the best choice for memory purposes

Conclusions

- We designed a simple indexing structure, INSTRUCT, that requires the least amount of space
- It supports the full range of string queries including exact, suffix, prefix and substring search
- INSTRUCT procedures can be easily implemented as parallel algorithms
- Actual effects of parallelization need to be measured
- Choice of other structures such as hash table for containers needs to be explored

Conclusions

- We designed a simple indexing structure, INSTRUCT, that requires the least amount of space
- It supports the full range of string queries including exact, suffix, prefix and substring search
- INSTRUCT procedures can be easily implemented as parallel algorithms
- Actual effects of parallelization need to be measured
- Choice of other structures such as hash table for containers needs to be explored

THANK YOU!

Conclusions

- We designed a simple indexing structure, INSTRUCT, that requires the least amount of space
- It supports the full range of string queries including exact, suffix, prefix and substring search
- INSTRUCT procedures can be easily implemented as parallel algorithms
- Actual effects of parallelization need to be measured
- Choice of other structures such as hash table for containers needs to be explored

THANK YOU!

Questions?