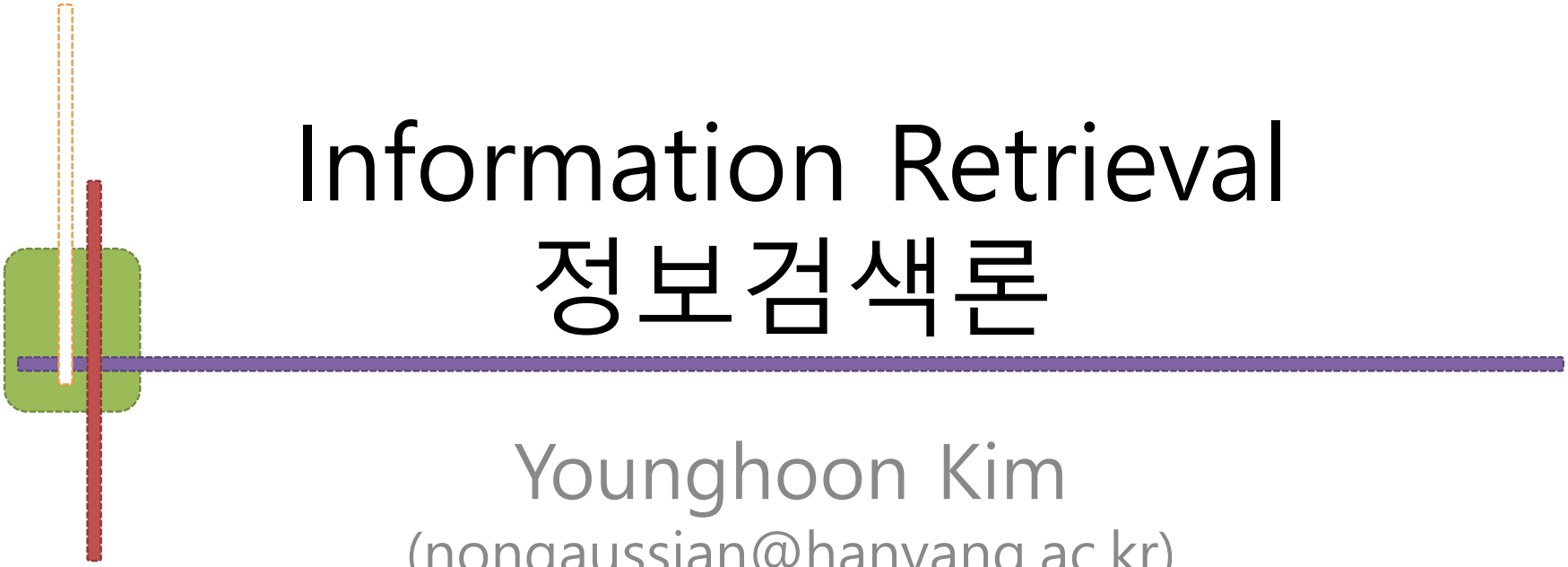


Information Retrieval
정보검색론

Younghoon Kim
(nongaussian@hanyang.ac.kr)

TERM-DOCUMENT INCIDENCE MATRICES

Unstructured data in 1620

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?
- One could `grep` all of Shakespeare's plays for ***Brutus*** and ***Caesar,*** then strip out lines containing ***Calpurnia***?
- Why is that not the answer?
  - Slow (for large corpora)
  - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

Term-document incidence matrices

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Brutus AND ***Caesar*** BUT NOT ***Calpurnia***

1 if play contains word, 0 otherwise

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for ***Brutus, Caesar*** and **not *Calpurnia*** (complemented) ➔ bitwise *AND*.
  - 110100 *AND*
  - 110111 *AND*
  - 101111 =
  - **100100**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Answers to query

- Antony and Cleopatra, Act III, Scene ii

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius ***Caesar*** dead,
he cried almost to roaring; and he wept
when at Philippi he found ***Brutus*** slain.

snippet

- Hamlet, Act III, Scene ii

Lord Polonius: I did enact Julius ***Caesar*** I was killed i' the Capitol; ***Brutus*** killed me.

Bigger collections

- Consider N = 1 million documents, each with about 1000 words.
- Avg. 6 bytes/word including spaces/punctuation
  - 6GB of data in the documents.
- Say there are M = 500K *distinct* terms among these.

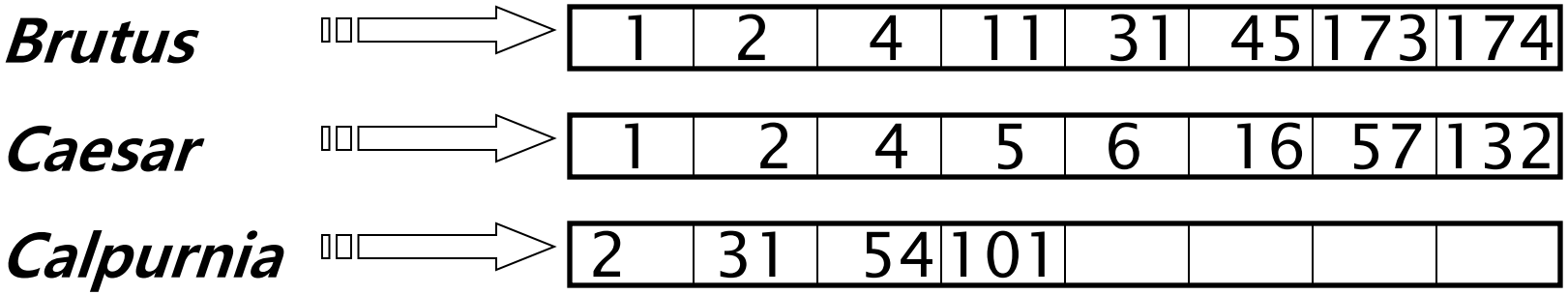
Can't build the matrix

- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
  - matrix is extremely sparse.
- What's a better representation?
  - We only record the 1 positions.

THE INVERTED INDEX
THE KEY DATA STRUCTURE
UNDERLYING MODERN IR

Inverted index

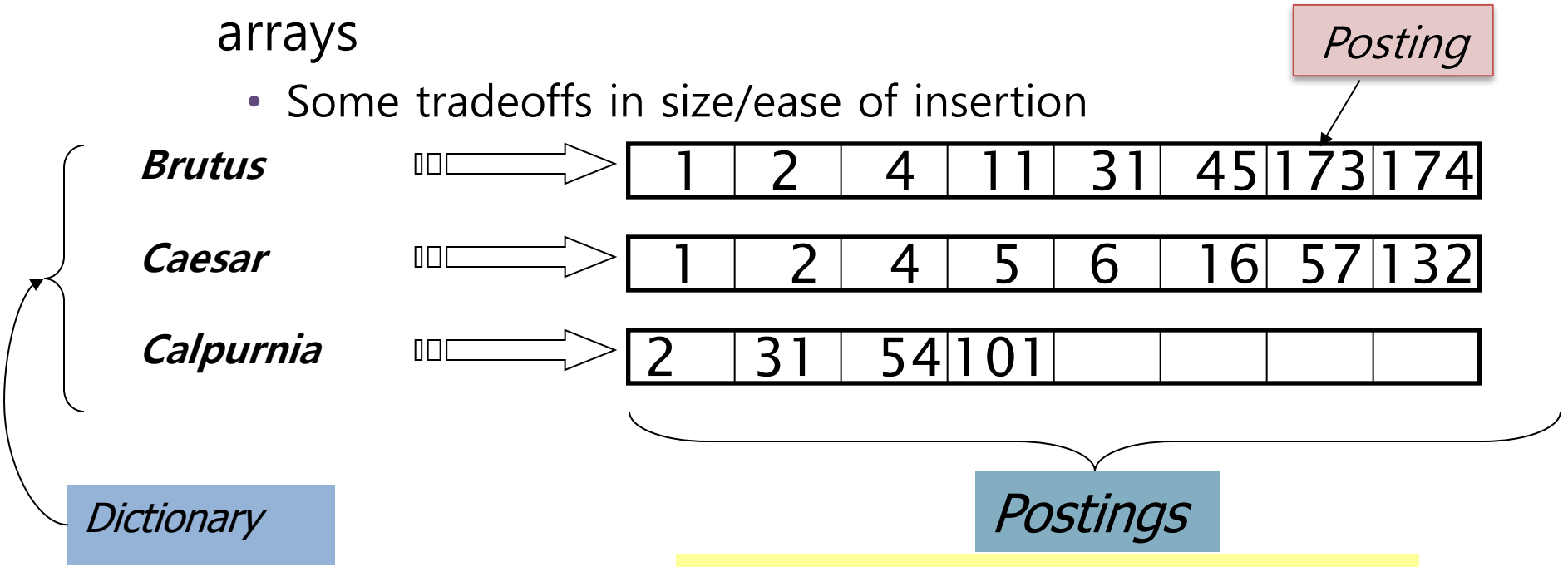
- For each term *t*, we must store a list of all documents that contain *t*.
  - Identify each doc by a **docID**, a document serial number
- Can we used fixed-size arrays for this?



What happens if the word ***Caesar*** is added to document 14?

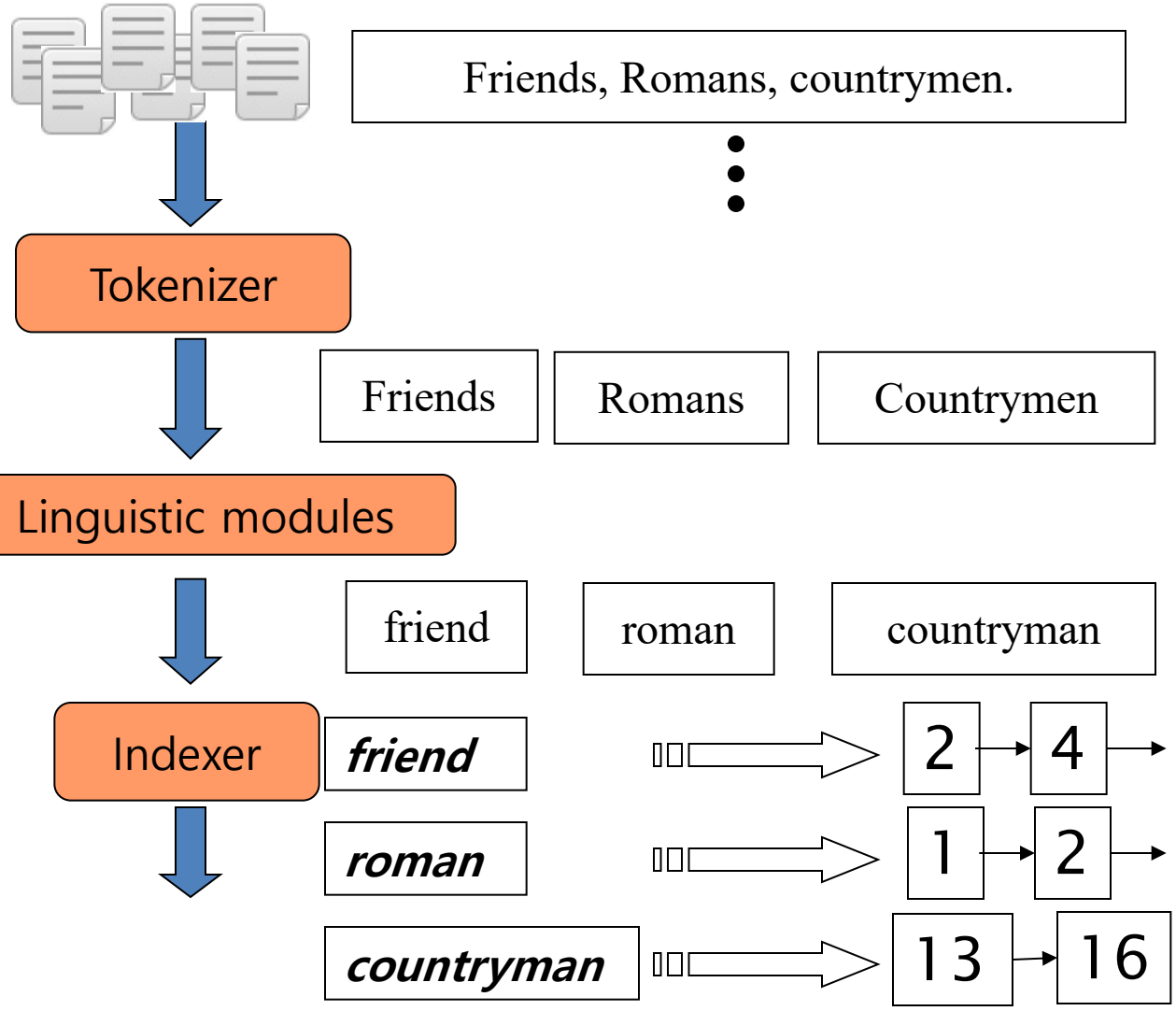
Inverted index

- We need variable-size posting lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion



Sorted by docID (more later on why).

Inverted index construction



Initial stages of text processing

- Tokenization
  - Cut character sequence into word tokens
    - Deal with ***"John's"***, ***a state-of-the-art solution***
- Normalization
  - Map text and query term to same form
    - You want ***U.S.A.*** and ***USA*** to match
- Stemming
  - We may wish different forms of a root to match
    - ***authorize***, ***authorization*** → ***authoriz***
- Stop words
  - We may omit very common words (or not)
    - ***the, a, to, of***

Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then docID

Most expensive indexing step

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Split into dictionary and postings
- Doc. frequency information is added.

Why frequency?
Will discuss later.

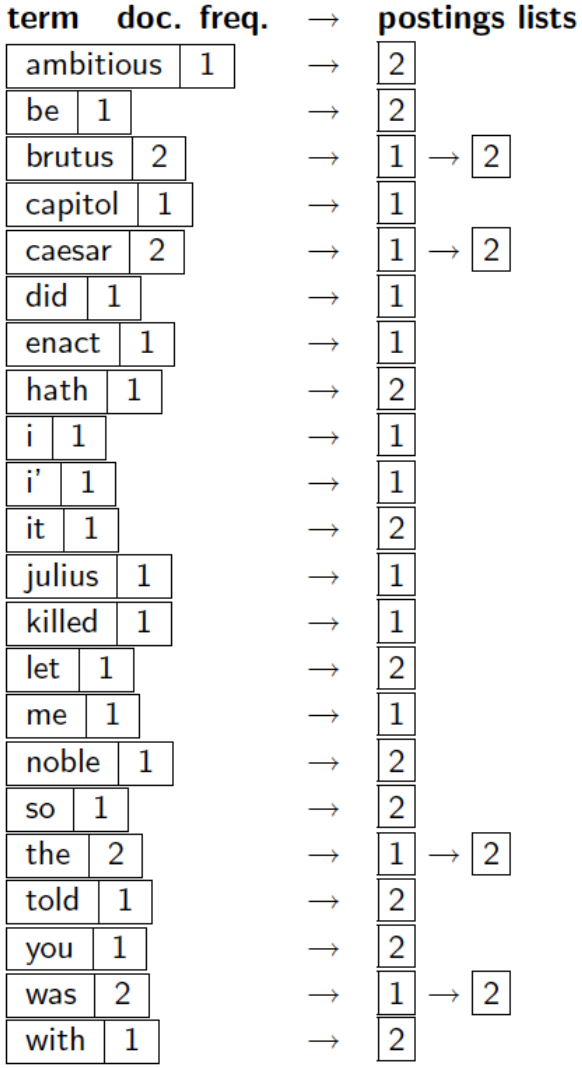
Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Where do we pay in storage?

Terms and counts

Lists of docIDs



IR system implementation

- How do we index efficiently?
- How much storage do we need?

Implementation

- Maintain an inverted index with RDBMs
  - Can we represent it with tables?
- Data structure for storing an inverted index
  - B-tree, B^+-tree

Exercise 1.2

- Draw the inverted index that would be built for the following document collection.
  - Doc 1: **b**reakthrough **d**rug **f**or **s**chizophrenia
  - Doc 2: **n**ew **s**chizophrenia **d**rug
  - Doc 3: **n**ew **a**pproach **f**or **t**reatment **o**f **s**chizophrenia
  - Doc 4: **n**ew **h**opes **f**or **s**chizophrenia **p**atients

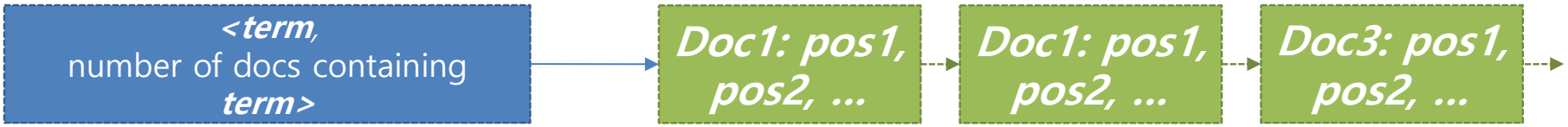
POSITIONAL INDEXES

Phrase queries

- We want to be able to answer queries such as ***"Hanyang university"*** – as a phrase
- Thus the sentence *"I went to university near Hanyang high school"* is not a match.
  - The concept of phrase queries has proven easily understood by users; one of the few "advanced search" ideas that works
  - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only <*term* : *docs*> entries

Positional indexes

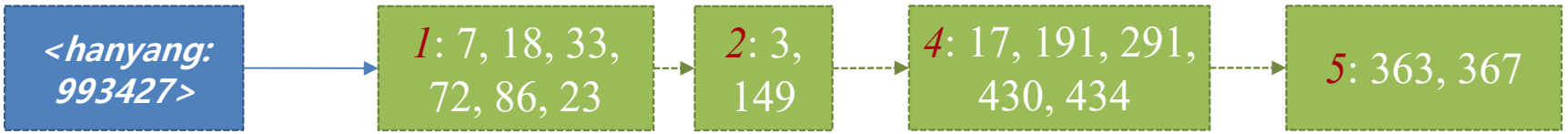
- In the postings, store, for each ***term*** the position(s) in which tokens of it appear:



Dictionary

Posting list

<example>



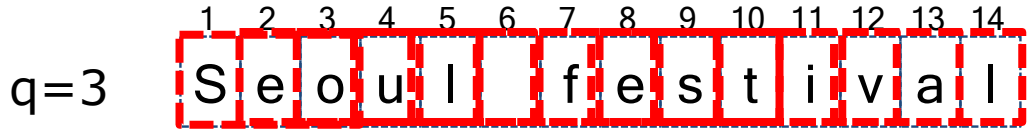
Efficient Processing of Substring Match Queries with Inverted q-gram Indexes

Younghoon Kim et al. ICDE 2010.

SUBSTRING MATCHING WITH AN INVERTED INDEX

Inverted q-gram Index

- q-gram term
  - For a given string, any substring of length q



- Posting list
  - For each q-gram, a list of record ids including the q-gram

Article

rid	title
r_1	Seoul festival
r_2	Samsung Electronics
r_3	Busan film festival
r_4	Activities in Seoul

Posting list of 'Seo'

rid
r_1
r_4

Posting list of 'fes'

rid
r_1
r_3

Posting list of 'tiv'

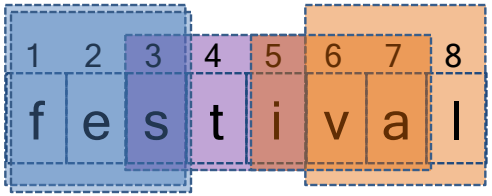
rid
r_1
r_3
r_4

Traditional Query Processing

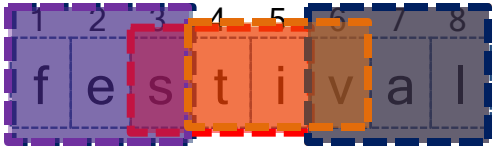
- Covering q-gram set: the character at every position is covered by at least a q-gram
- Non-covering q-gram set: the character at a position may not covered by any q-gram
- Traditional methods explore covering q-gram sets
  - MAX-COVER [E. S. Adams and A. C. Meltzer, 93]
    - Uses all the q-grams in query string
  - OPT-MINC [Y. Ogawa and T. Matsuda, 98]
    - Uses q-grams that takes smallest I/O cost among the covering q-grams with minimum size

Query=festival, q=3

Non-covering q-gram set: {fes,sti,val}



Covering set with minimum size:
{fes,sti,val}
or {fes,tiv,val}



Query Processing I/O Cost with Covering q-gram Sets

- Query processing I/O cost consists of
  - The number of read pages of posting lists
  - The number of retrieved pages of the result

Article

rid	title
r_1	Seoul festival
r_2	Alternative Life
r_3	Survival of the fittest
r_4	GNU menifesto

Query: festival

rid	title
r_1	Seoul festival

{fes,tiv,val}: a covering q-gram set

L_1 (g_1=fes)

rid
r_1
r_4

L_4(g_4=tiv)

rid
r_1
r_2

L_6(g_6=val)

rid
r_1
r_3

• Assume reading each entry in posting list and each record takes **1** page and **3** pages repectively

I/O cost for reading 3 posting lists = 6

\+

I/O cost for reading result records = 3

= 9 pages

Non-covering q-gram Sets may be Better

- Non-covering q-gram set
  - The character at some position may not covered by any q-gram

Article

rid	title
r_1	Seoul festival
r_2	Alternative Life
r_3	Survival of the fittest
r_4	GNU menifesto

Query: festival

rid	title
r_1	Seoul festival
r_4	GNU menifesto

{fes}: a non-covering q-gram set

L_1 (g_1=fes)

rid
r_1
r_4

• Assume reading each entry in posting list and each record takes **1** page and **3** pages repectively

Less than the cost of a covering set {fes,tiv,val} = 9

I/O cost for reading one posting list = 2

\+

I/O cost for reading result records = 6

= 8 pages

Non-covering q-gram Sets may be Better

- Non-covering q-gram set
  - The character at some position may not covered by any q-gram

Article

rid	title
r_1	Seoul festival
r_2	Alternative Life
r_3	Survival of the fittest
r_4	GNU menifesto

Query: festival

rid	title
r_1	Seoul festival

{fes, tiv}: a non-covering q-gram set

L_1 (g_1=fes)

rid
r_1
r_4

L_4(g_4=tiv)

rid
r_1
r_2

• Assume reading each entry in posting list and each record takes **1** page and **3** pages repectively

I/O cost for reading 2 posting lists = 4

+

I/O cost for reading result records = 3

=

7 pages

We have to choose q-gram set judiciously

Which Q-gram Set to Choose?

- Problem formulation
  - Given
    - q-gram length q
    - A query string
    - A table of strings and its inverted q-gram index
  - Select the subset of q-grams from the query string with the minimum I/O cost
  - Remember that query processing cost is

[I/O cost for reading posting lists + I/O cost for retrieving the records in the intersection result]

Challenges

- Estimation of intersection result size
  - Minhash technique [Zhiyuan Chen, Flip Korn, Nick Koudas and S. Muithukrishnan, 2000]
    - A Monte-Carlo technique to estimate intersection set sizes
    - Captures the correlations of sets well with small space overhead
- Exploring all q-gram sets
  - For a query string of length n, $O(2^n)$ q-gram sets possible
  - Too expensive to enumerate all subsets

OPT-NAÏVE: An Optimal Algorithm

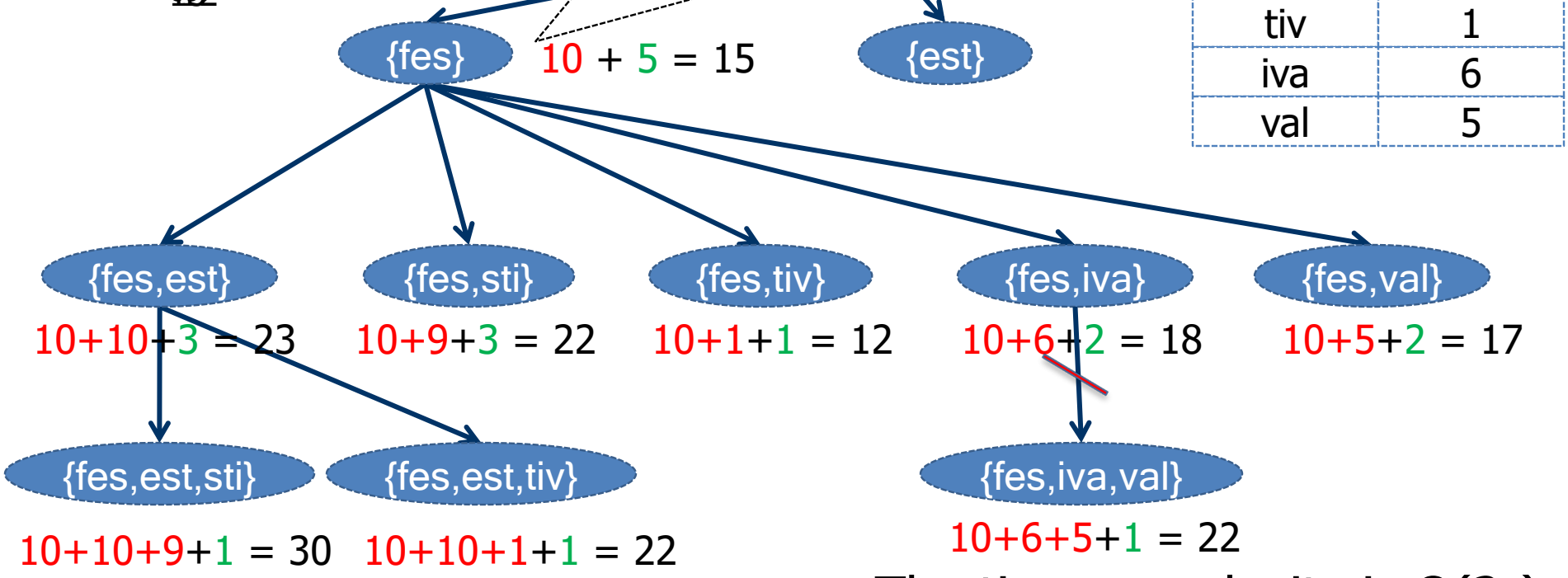
- Enumerate all possible q-gram subsets

Query = festival, q = 3
Buffer size = 1

Minimum cost so far = ∞

Cost of reading posting list = 10
Cost of retrieving the records in intersection results = 5

q-gram	List size
fes	10
est	10
sti	9
tiv	1
iva	6
val	5



The time complexity is $O(2^n)$

OPT-QSP: An Optimal Algorithm

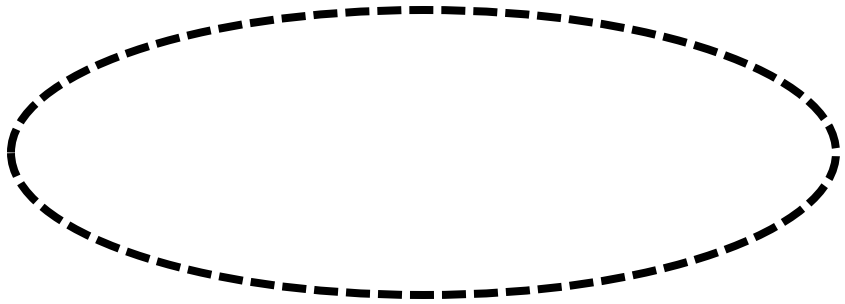
- Differences from OPT-NAÏVE
  - Branch and bound
  - Keep the minimum cost so far and use it for pruning
  - Do not explore q-gram sets which are guaranteed to be worse than the minimum cost so far
    - Costs of reading posting lists always increases by adding a q-gram
    - If costs of reading posting lists is larger than the minimum cost so far, we don't expand more

APR-GRQ: An Approximate Algorithm

- A greedy algorithm
- In each step of greedy selection,
  - Select the q-gram with the best improvement
  - If there is no improvement, exit this loop

Illustration of APR-GRQ

Query string = festival, q = 3
Buffer size = 1

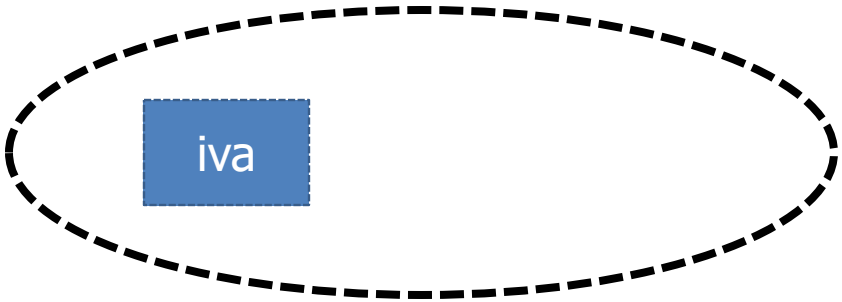


Cost of current set = [illegible]

fes	101
est	23
sti	48
tiv	124
iva	11
val	312

Illustration of APR-GRQ

Query string = festival, q = 3
Buffer size = 1

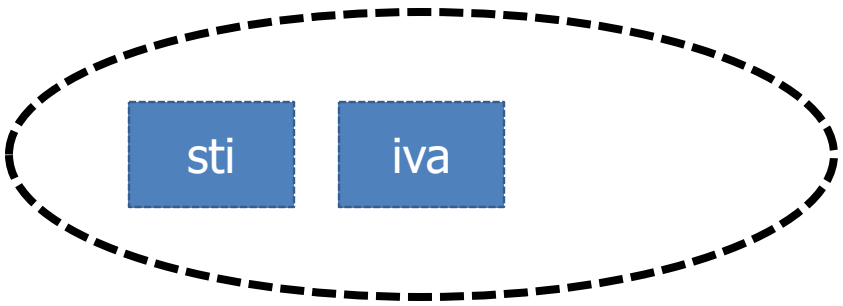


Cost of current set = [illegible]

fes	109
est	39
sti	10
tiv	120
val	242

Illustration of APR-GRQ

Query string = festival, q = 3
Buffer size = 1



Cost of current set = 10

fes 33

sti 142

tiv 45

val 98

Time complexity is $O(n^2)$
- Cost estimation O(n) times for each step
- Maximum number of step is n

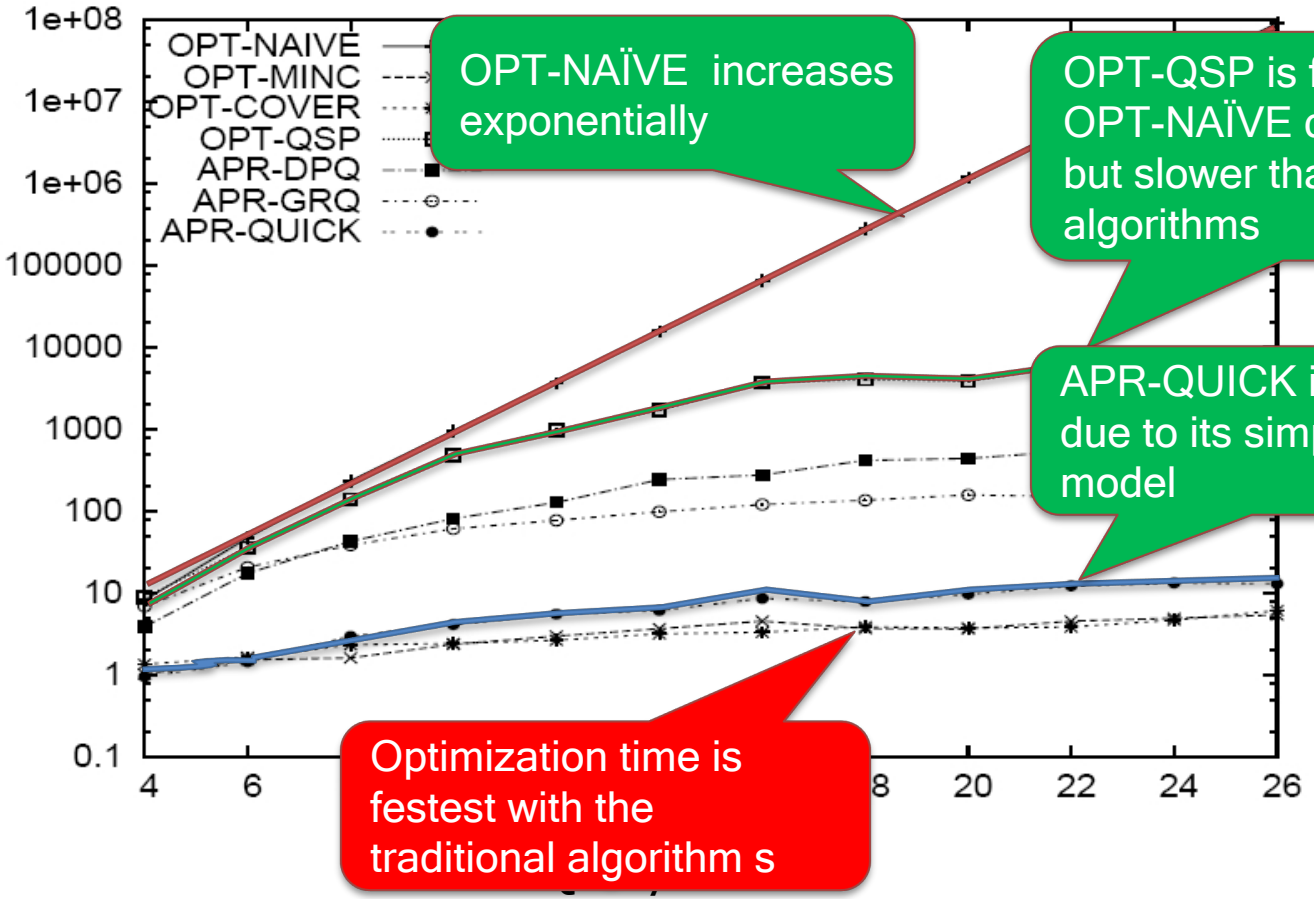
Experiments

- Test bed
  - OS: Linux/Ubuntu
  - 2.66GHz Intel CPU with 2G bytes of memory
- Datasets
  - DBLP titles - 1,000,000 records with avg. of 67 bytes
  - Times corpus - 100,000 articles with avg. of 2578 bytes
- Queries used
  - 100 random queries generated for each dataset
- Implemented inverted q-gram index
  - B+ Tree
  - Extensible Hash
  - Use our own buffer manager (default: 64 MBs)
  - Flushed buffer for each query execution

q-gram Set Selection Time

- Time overhead for q-gram set selection with varing query length from 4 to 26 with B+ tree

Set selection time overhead (usec) in log-scale

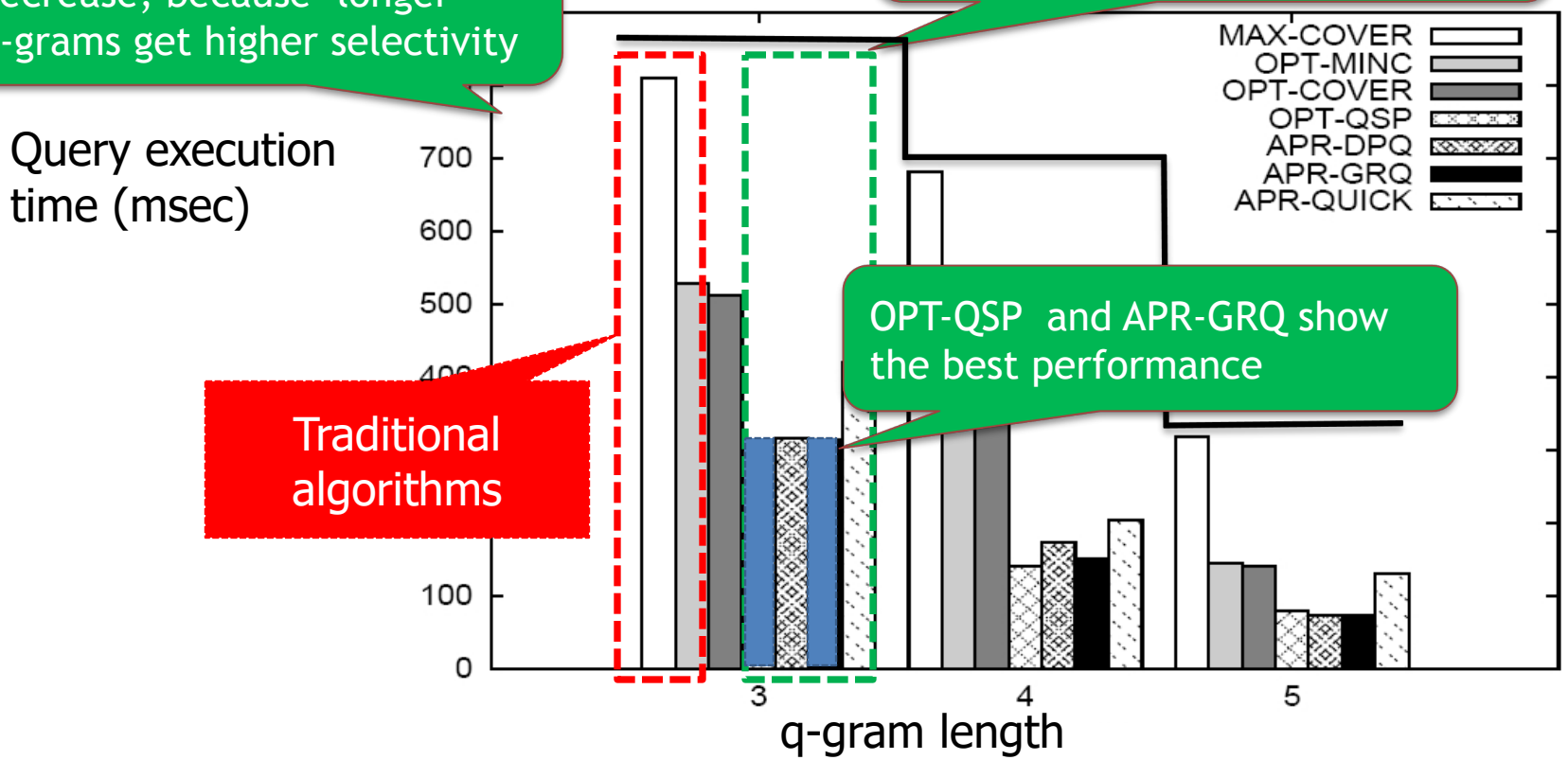


Query Execution Time with Varing q-gram Length

- Total query execution time with varying q-gram length 3 to 5 with B+ tree
- Query length is evenly distr

The execution time gradually decrease, because longer q-grams get higher selectivity

Our proposed algorithms are much faster than the traditional algorithms



Query Execution Time with Varying Query Length

- Total query execution time with varying query length from 4 to 20 with B+ tree