Data structures in Information Retrieval

Building index

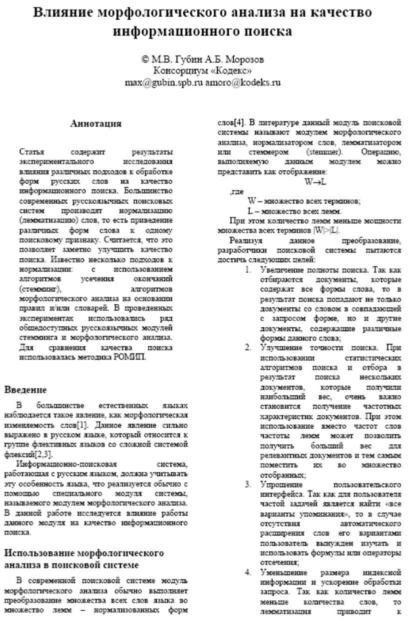
Build index (pseudocode)

```
Index = new Index()
for document in (collection):
  for word in (document):
     index[word].insert(document.id);
```

Questions

1. Implementation iterators/extractors
2. Indexing big collections/speed
3. Compression of indexes
4. Updating index
5. Parallelization/ cluster use

Linearization (word extraction)



Влияние морфологического анализа на качество информационного поиска

© М.В. Губин А.Б. Морозов
Консорциум «Кодекс»
max@gubin.spb.ru amoro@kodeks.ru

Аннотация

Статья содержит результаты экспериментального исследования влияния различных подходов к обработке форм русских слов на качество информационного поиска. Большинство современных русскоязычных поисковых систем производят нормализацию (лемматизацию) слов, то есть приведение различных форм слова к одному поисковому признаку. Считается, что это позволяет заметно улучшить качество поиска. Известно несколько подходов к нормализации: с использованием алгоритмов усечения окончаний (стемминг), алгоритмов морфологического анализа на основании правил и/или словарей. В проведенных экспериментах использовались ряд общедоступных русскоязычных модулей стемминга и морфологического анализа. Для сравнения качества поиска использовалась методика РОМИП.

Введение

В большинстве естественных языках наблюдается такое явление, как морфологическая изменяемость слов[1]. Данное явление сильно выражено в русском языке, который относится к группе флективных языков со сложной системой флексий[2,3].

Информационно-поисковая система, работающая с русским языком, должна учитывать эту особенность языка, что реализуется обычно с помощью специального модуля системы, называемого модулем морфологического анализа. В данной работе исследуется влияние работы данного модуля на качество информационного поиска.

Использование морфологического анализа в поисковой системе

В современной поисковой системе модуль морфологического анализа обычно выполняет преобразование множества всех слов языка во множество лемм – нормализованных форм слов[4]. В литературе данный модуль поисковой системы называют модулем морфологического анализа, нормализатором слов, лемматизатором или стеммером (stemmer). Операцию, выполняемую данным модулем можно представить как отображение:

$$W \to L$$

,где

W – множество всех терминов;

L – множество всех лемм.

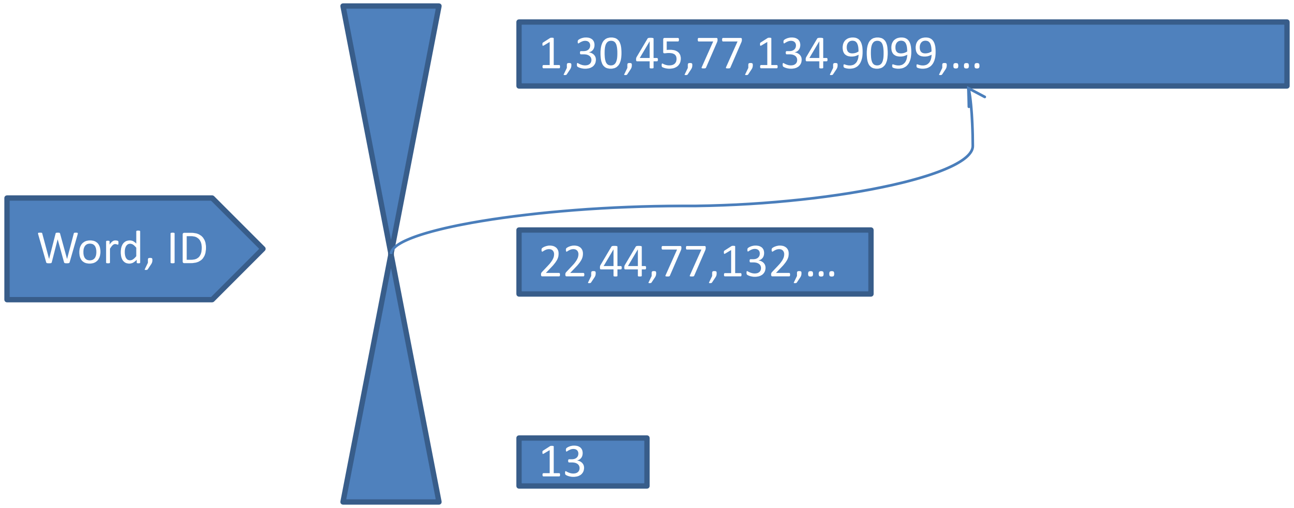
При этом количество лемм меньше мощности множества всех терминов $|W|>|L|$.

Реализуя данное преобразование, разработчики поисковой системы пытаются достичь следующих целей:

1. Увеличение полноты поиска. Так как отбираются документы, которые содержат все формы слова, то в результат поиска попадают не только документы со словом в совпадающей с запросом форме, но и другие документы, содержащие различные формы данного слова;
2. Улучшение точности поиска. При использовании статистических алгоритмов поиска и отбора в результат поиска нескольких документов, которые получили наибольший вес, очень важно становится получение частотных характеристик документов. При этом использование вместо частот слов частоты лемм может позволить получить больший вес для релевантных документов и тем самым поместить их во множество отобранных;
3. Упрощение пользовательского интерфейса. Так как для пользователя частой задачей является найти «все варианты упоминания», то в случае отсутствия автоматического расширения слов его вариантами пользователь вынужден изучать и использовать формулы или операторы отсечения;
4. Уменьшение размера индексной информации и ускорение обработки запроса. Так как количество лемм меньше количества слов, то лемматизация приводит к

(“To”, 12)
(“BE”,12)
(“or”,12)
(“not”,12)
(“to”, 12)
(“be”,12)

Build one piece of index



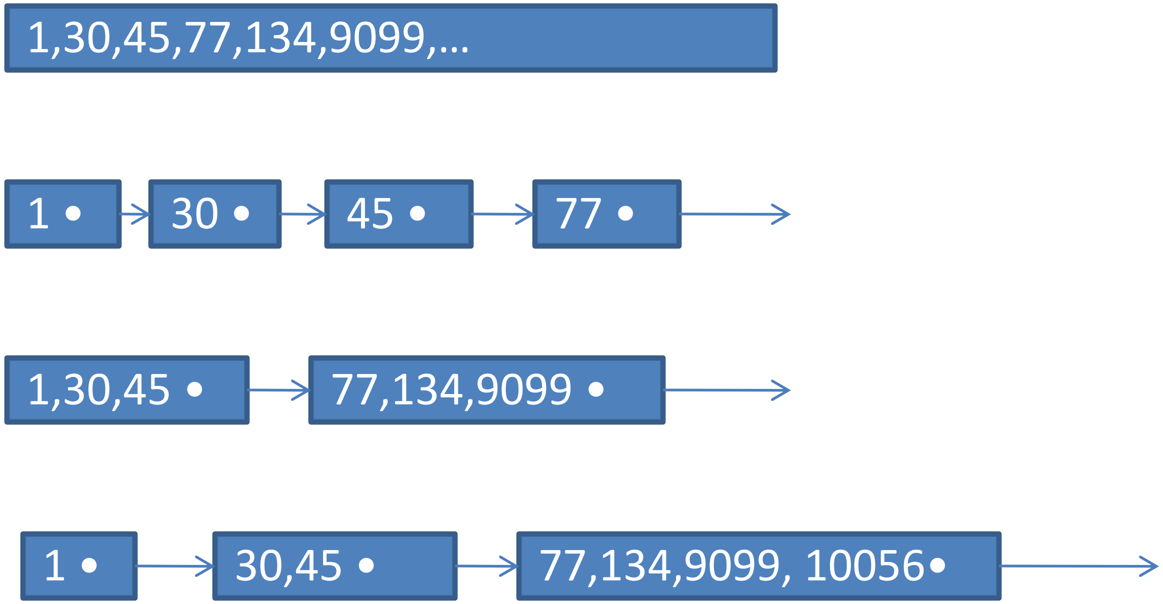
Word, ID

1,30,45,77,134,9099,…

22,44,77,132,…

13

Post lists in memory



Growing array: 1,30,45,77,134,9099,...

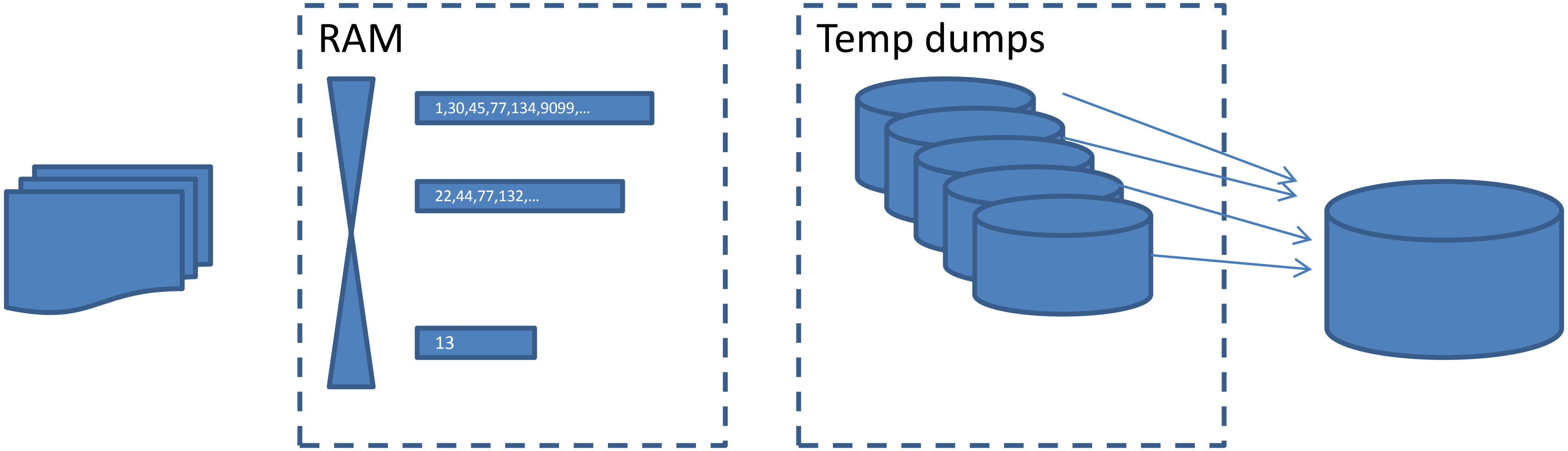
List: 1 • → 30 • → 45 • → 77 • →

List of arrays: 1,30,45 • → 77,134,9099 • →

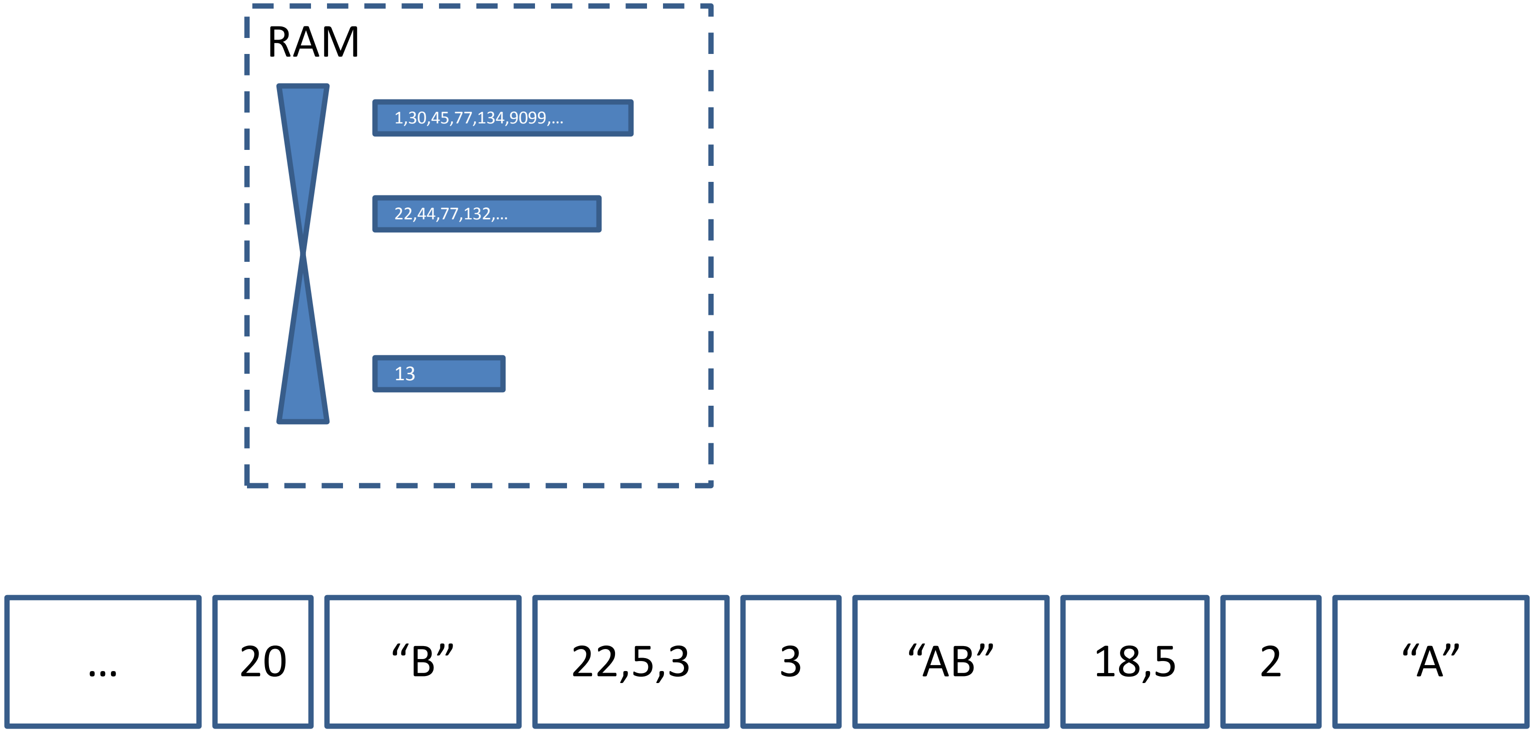
List of adapt. arrays: 1 • → 30,45 • → 77,134,9099, 10056• →

Decrease number of allocations, avoid fragmentation

Build big indexes



Temporary dump



RAM

1,30,45,77,134,9099,...

22,44,77,132,...

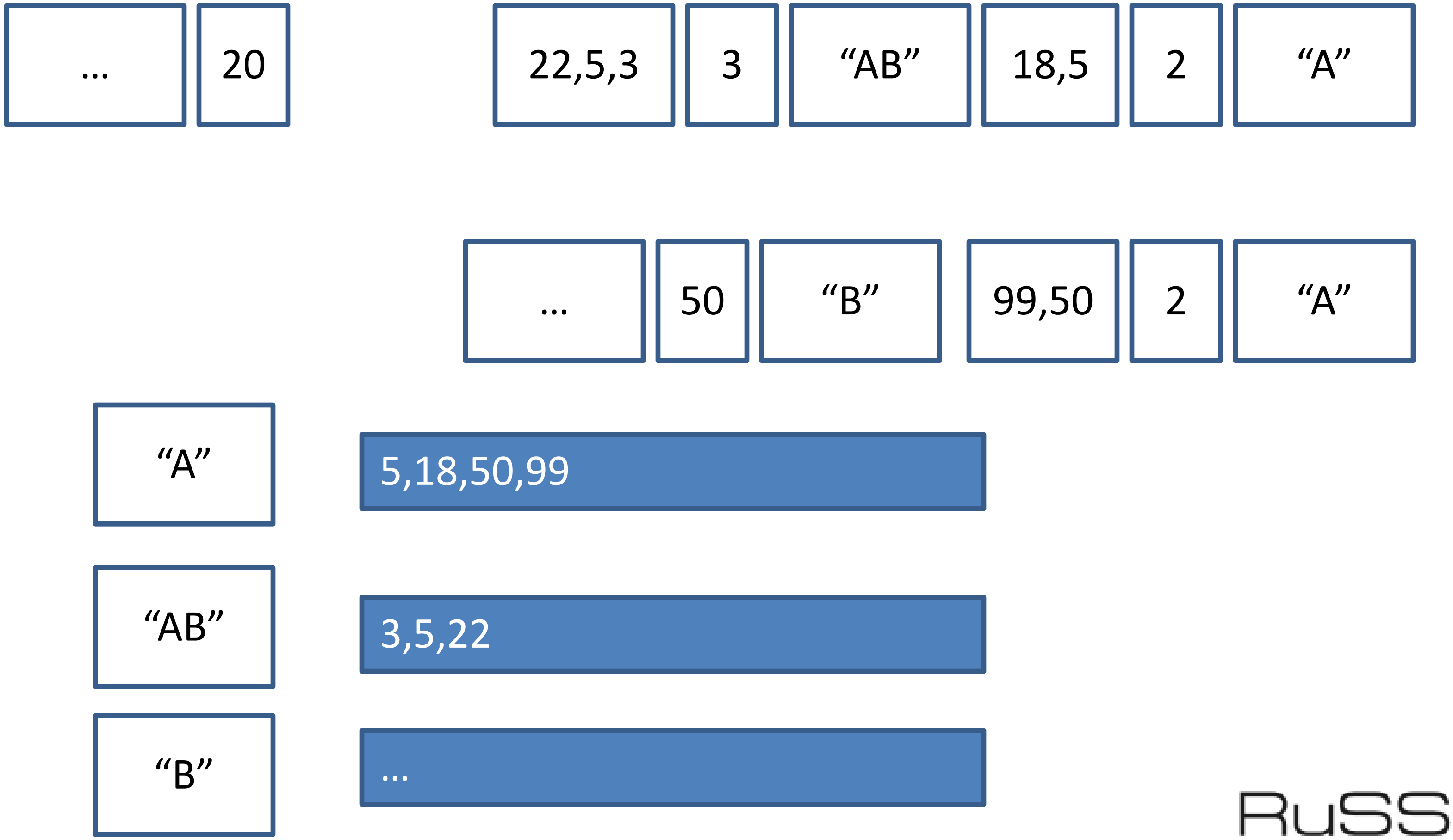
13

…	20	“B”	22,5,3	3	“AB”	18,5	2	“A”

Multiway merge [merging dumps]

1. Load first words into memory
2. Select smallest words
3. Merge post-lists for the words
4. Move to next words in pr. dumps
5. Remove empty dumps
6. Goto 2 if dumps are not empty

Multiway merge [example]



… | 20 | 22,5,3 | 3 | “AB” | 18,5 | 2 | “A”

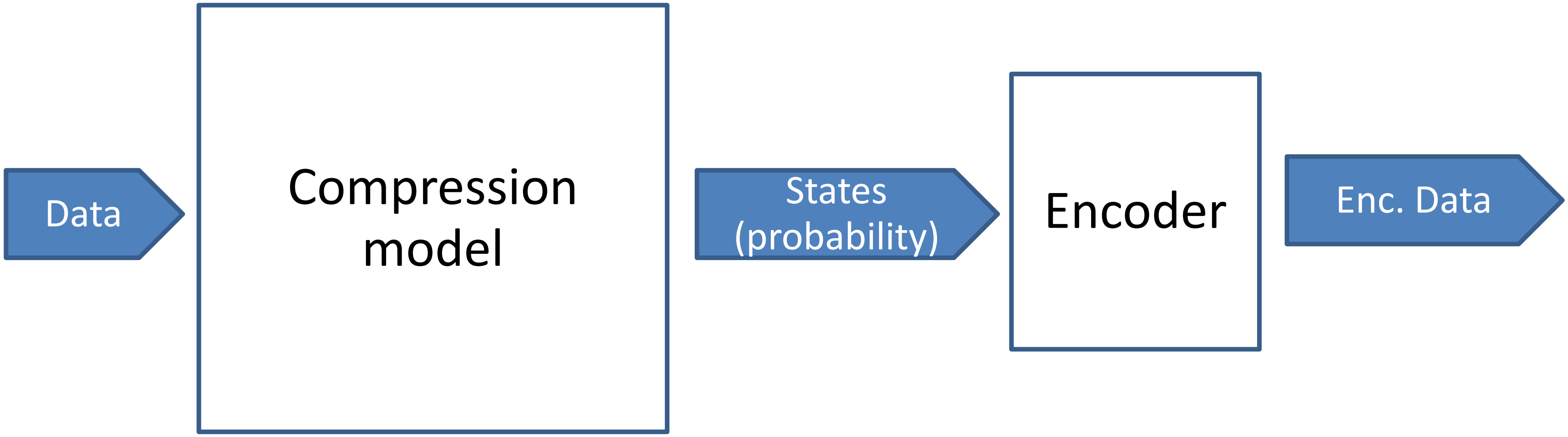
… | 50 | “B” | 99,50 | 2 | “A”

“A”	5,18,50,99
“AB”	3,5,22
“B”	…

Multiway merge [summary]

1. Linear number of operations
2. Fixed amount of RAM in memory
3. Linear access to data

Compression



Data

Compression model

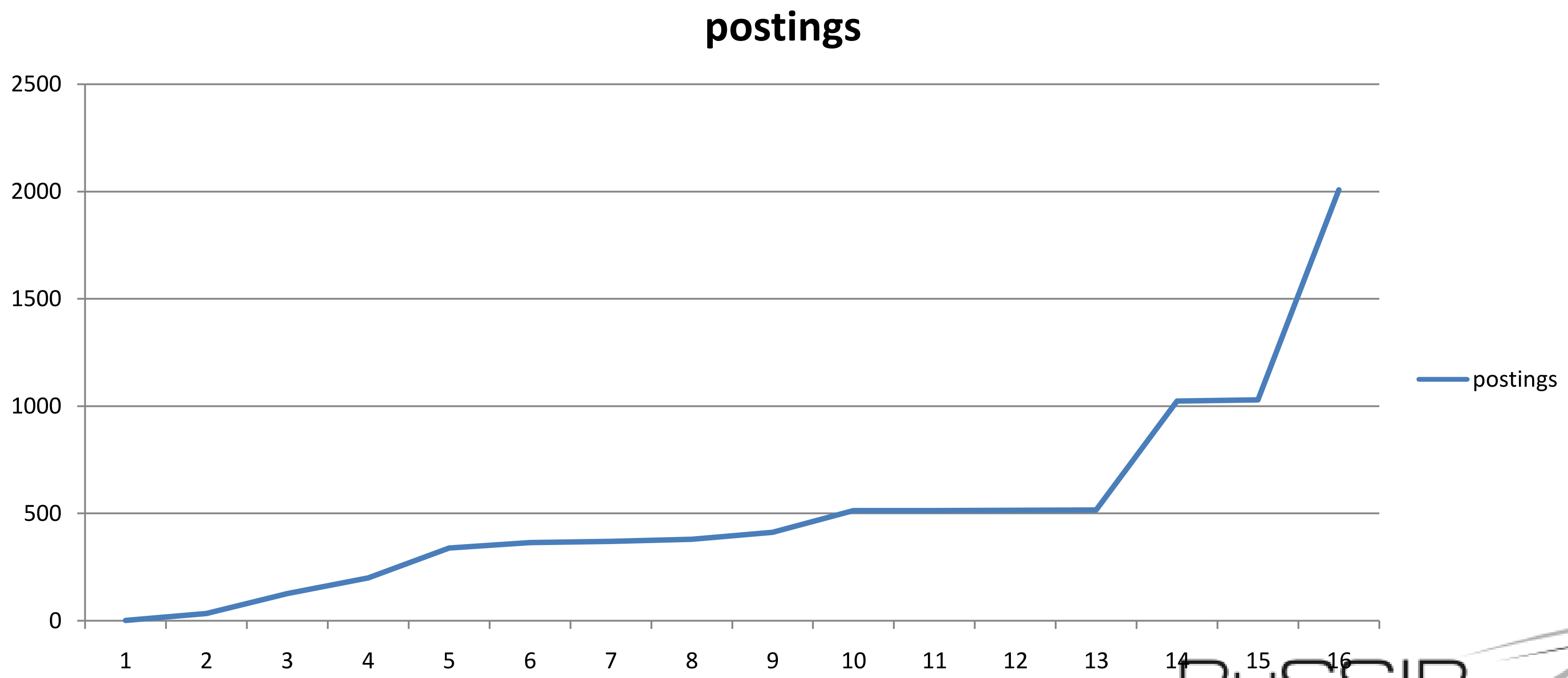
States (probability)

Encoder

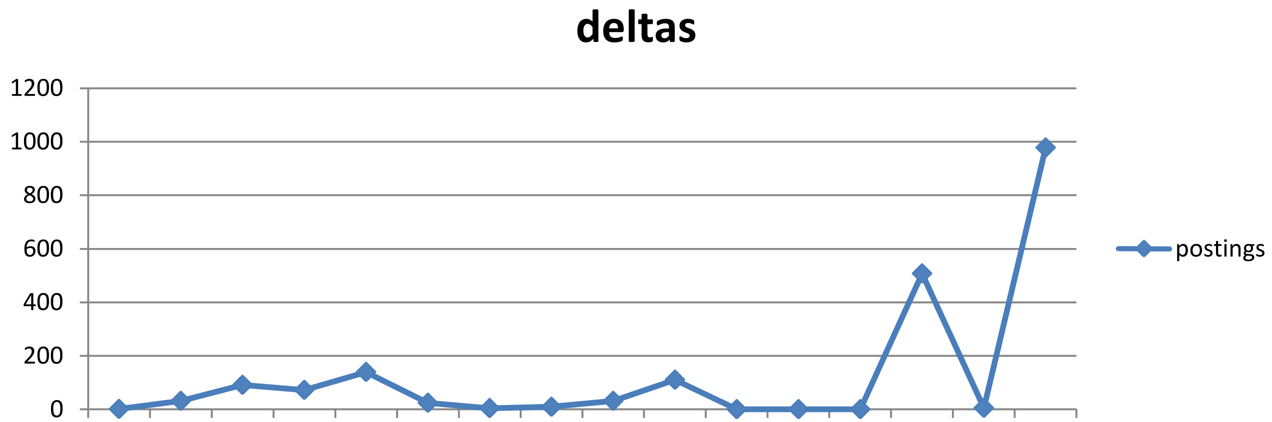
Enc. Data

Model for post-lists

<2,35,127,200,340,365,370,380,412,513,514,515516,1024,1030,2008>



Model for post-lists



$$N = ID_{cur} - ID_{prev} - 1$$

Universal code

- Binary code
- Prefix code (without out-of-band markers)
- If integers distribution $p(i+1) < p(i)$ → E(codelength) = E(optimal_code)+C

Unary code

n	Unary code
1	1
2	01
3	001
4	0001
5	00001
6	000001
7	0000001
8	00000001
9	000000001
10	0000000001

Elias code

Beta code:

1. Unary code for $[\log_2 i]$
2. i in binary without leading 1

1	1
2	010
3	011
4	00100

Elias ω code (universal case)

```
write(0)
while([$\log_2$ i]!= 0)
        prefix($C_b$(i))
        i = [$\log_2$ i]
```

1023	**11100111111111110**
1024	111010100000000000
1025	111010100000000010

Rice-Golomb code

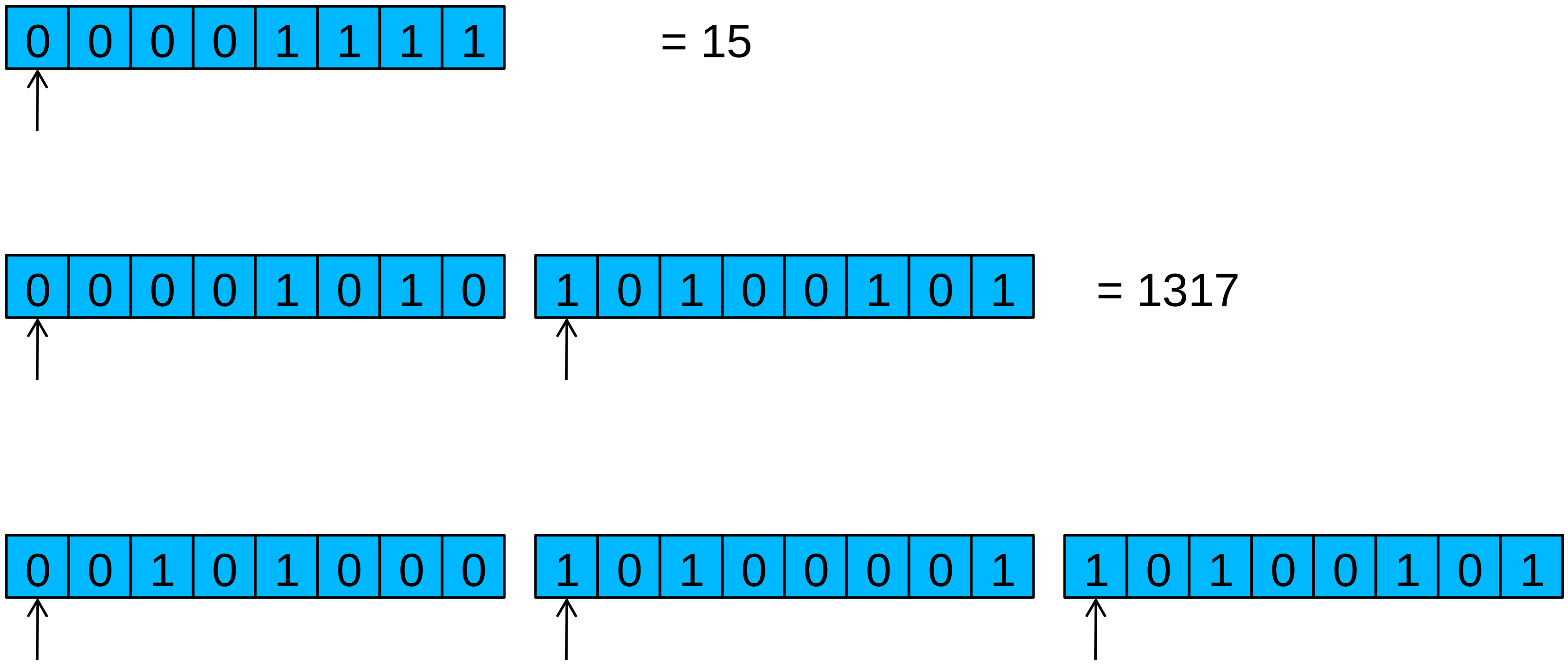
Param Q

X:Q ->unary code
X%Q ->binary code

Optimal for geometric distribution

Value	Quotient	Remainder	Code
0	0	0	1 00
1	0	1	1 01
2	0	2	1 10
3	0	3	1 11
4	1	0	0 1 00
5	1	1	0 1 01
6	1	2	0 1 10
7	1	3	0 1 11
8	2	0	00 1 00
9	2	1	00 1 01
10	2	2	00 1 10
11	2	3	00 1 11
12	3	0	000 1 00
13	3	1	000 1 01
14	3	2	000 1 10
15	3	3	000 1 11

Variable-byte code



0 0 0 0 1 1 1 1 = 15

0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 = 1317

0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 1 0 1

= 329893

Block coding

1. Divide post-list into blocks
2. Encode every block independently
3. Use all postings in block (more information)

How to define block:

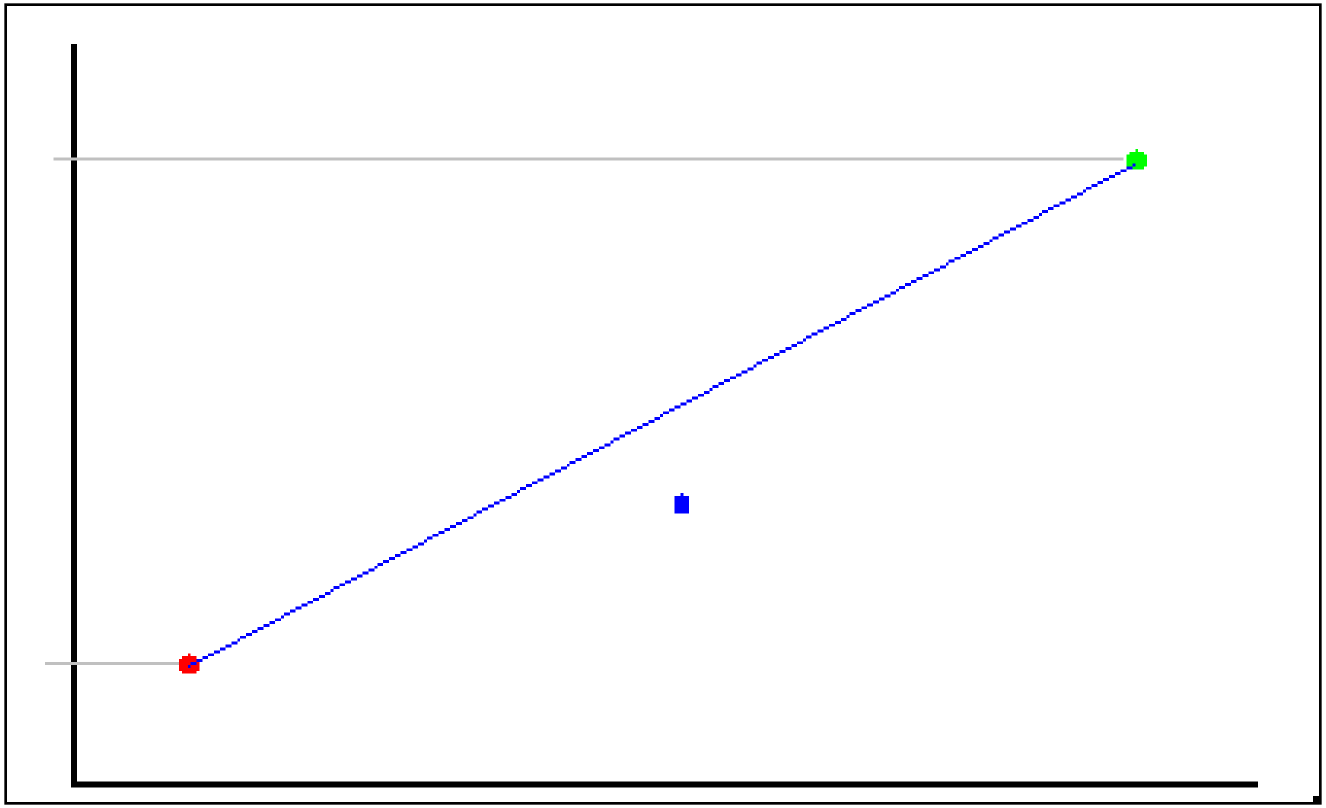
- Fixed number of postings
- Fixed encoded size

Interpolative encoding

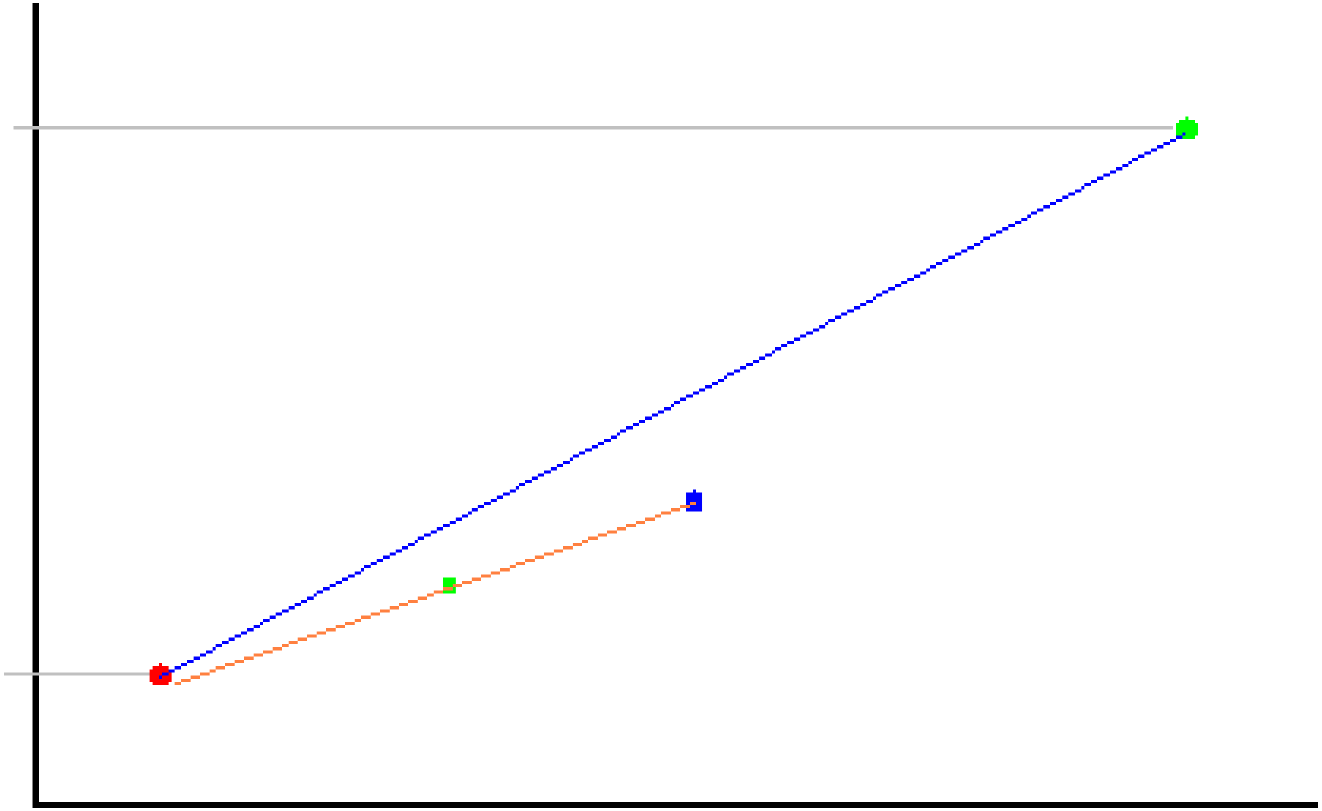
Example of **block coding**:

1. Encode first post (Elias, etc…)
2. Encode delta with last code in block
3. Encode middle with truncated binary
4. Repeat recursively prev. step

Interpolative encoding(2)



Interpolative encoding(3)



Index update polices

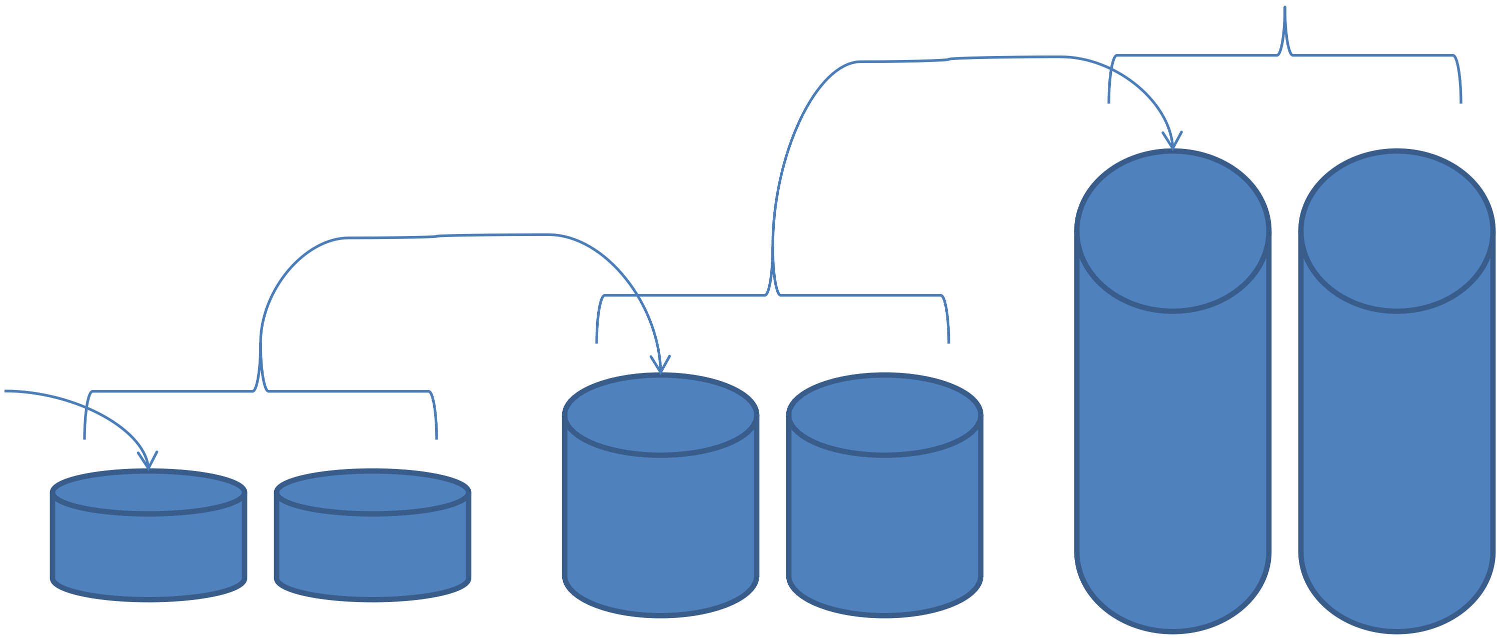
- Full rebuild (bulk data load)
- Rebuild/merge strategies
- In-place update

Full rebuild

Conditions:

1. Static collection
2. Big changes in collection
3. Indexing and Search are physically separated

Logarithm merge [rebuild/merge]



Inverted index over B-Tree

- Well-studied “classical” structure
- Many good implementations
- Good solution “little memory” and “in-place update”

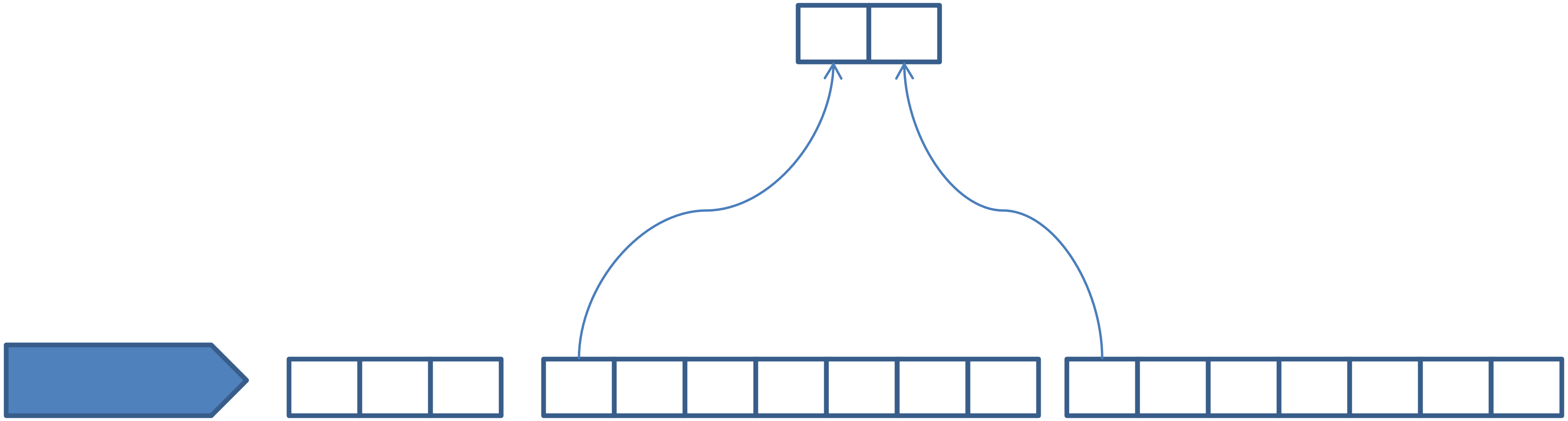
Tree slot:

Word	Block of post-list(compressed)

Cmp. function: (w1<w2)?-1:(w1==w2)?d1-d2:1

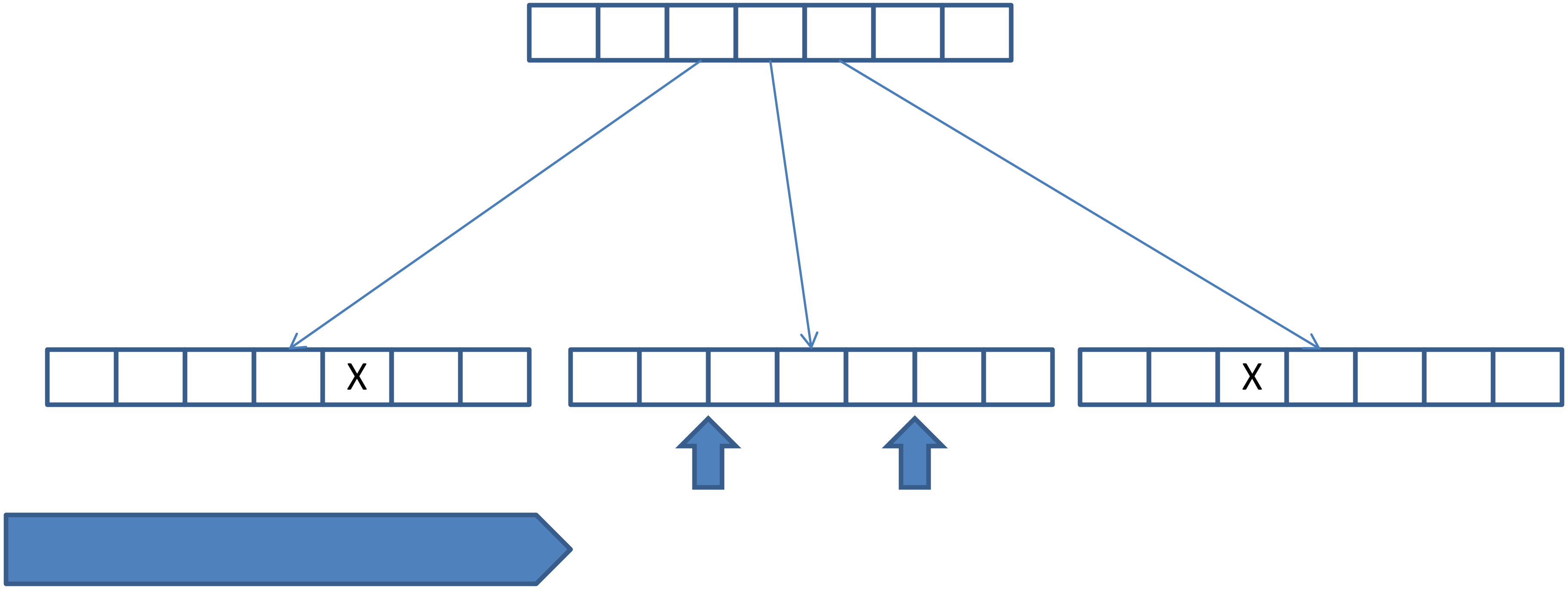
Building Inverted Index over B-Tree

Full rebuild B-Tree bulk load:



As effective as any other approach

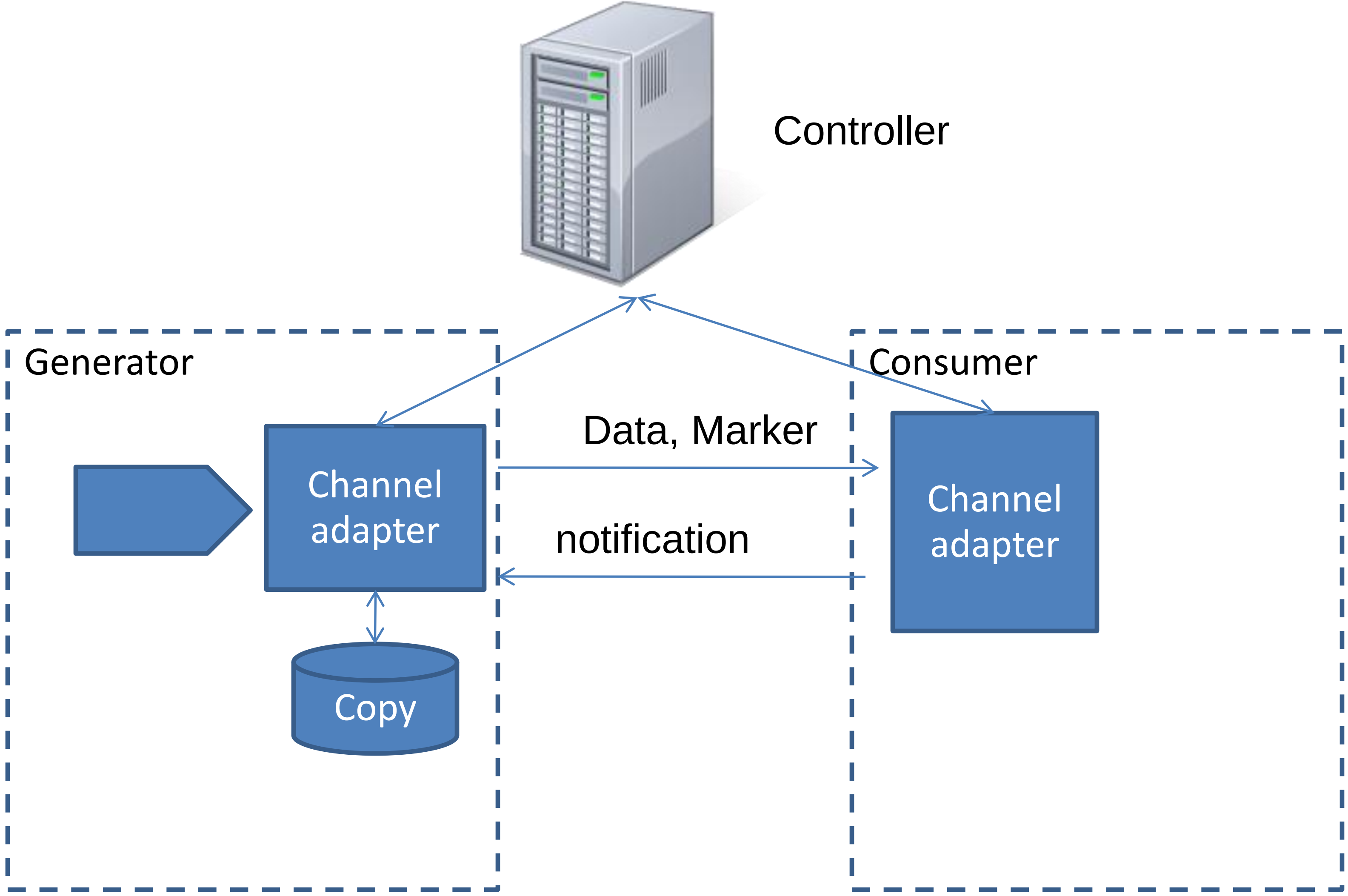
Update B-Tree in-place



Building in cluster

- Distribute process over cluster
- Effectively use resources
- Failure handling

Channels



Controller

Generator

Consumer

Data, Marker

notification

Channel adapter

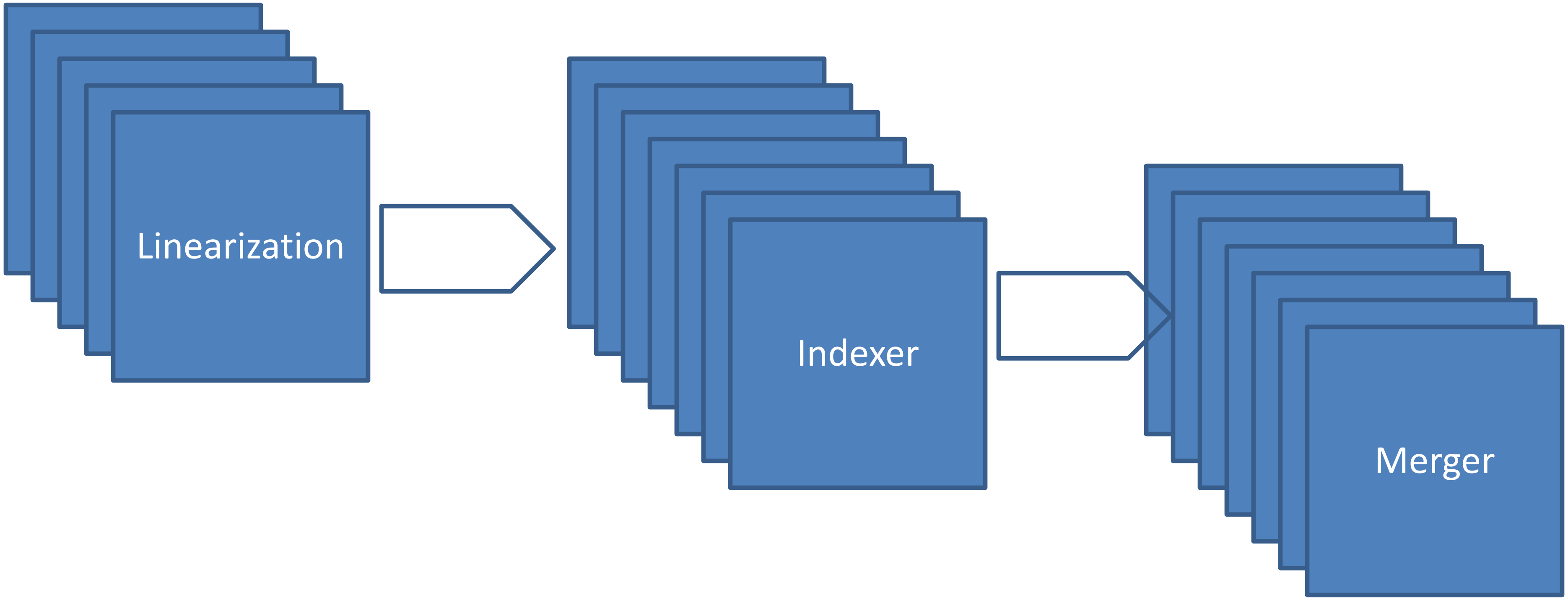
Channel adapter

Copy

Channel adapter logic

1. Find consumer
2. Sending to consumer and making copy
3. Wait for notification
4. If consumer failure Goto 1

Build index with channels



Map Reduce

- Use ideas of functional programming
- Simple paradigm, well understandable for programmers
- Match algorithms

Functional programming

- Avoids states and side effects
- “Pure functions” – get structures and return new structures

Map(fun,list) - apply fun for every element and return new list

Reduce(fun,list) – apply fun(prevVal,list_elem) to every list element and return fun. result

Build index in map-reduce [map]

```
def map(document):
    for each word w in document.text:
        Emit (w,(document.id,));
```

Build index in map-reduce [reduce]

```
Def Reduce(Word,ids_list)
   joined_list = ()
   for lst in ids_list:
        joined_list.append(joined_list)

   Emit(Word,joined_list)
```

Map-reduce enviroment

- Distribute tasks in cluster
- Move elements of list (group by key)
- Restart tasks in case of failure

Summary

- Build index in memory and merge
- Compress post-lists
- Build in cluster

Q&A