RuSSIR
Russian Summer School in Information Retrieval 2008

Data structures in Information Retrieval

Search!

RuSSIR
Russian Summer School in Information Retrieval 2008

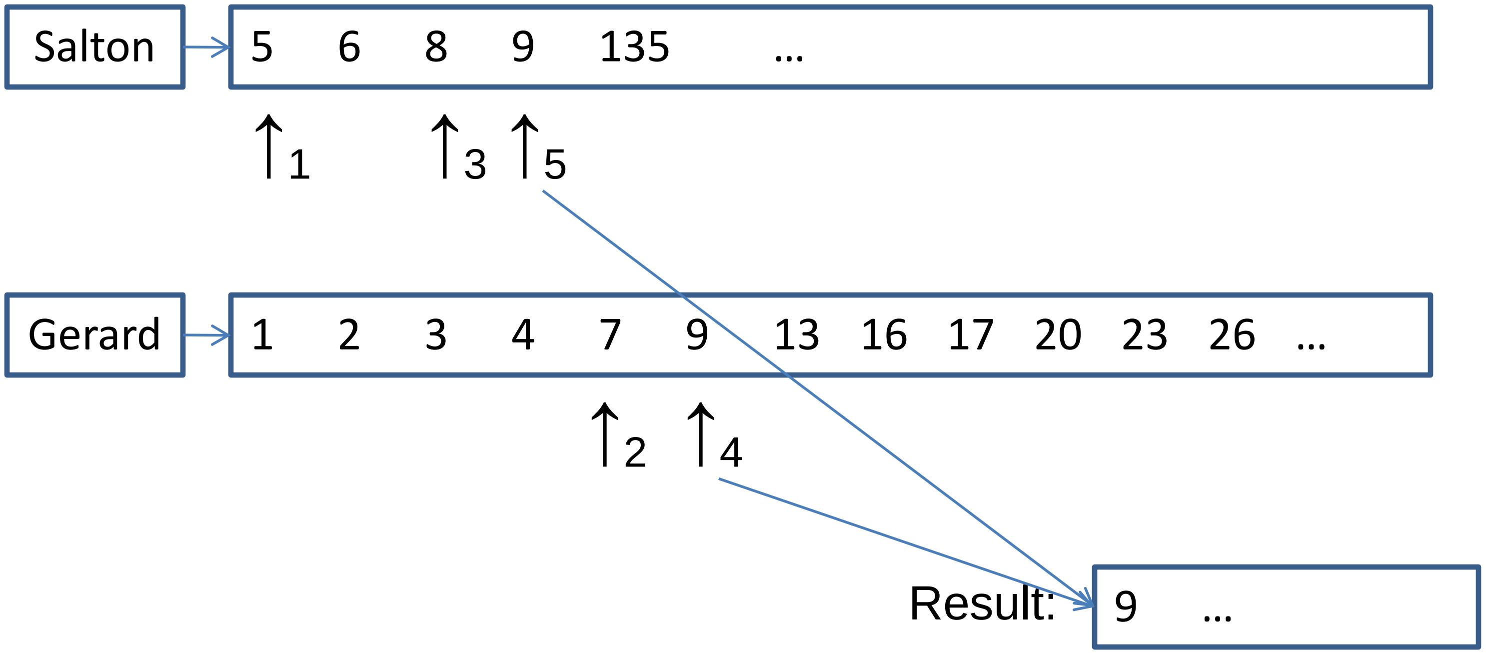
Search! [pseudocode]

```
Result = SearchResult()
Index = InvertedIndex()
for word in (query):
  for documentID in index[word]:

      Result.UpdateScore(word,documentID)


Result.SortByScore()
```

Problems

- Huge postlists
- Huge results
- Search in cluster

Multiway join

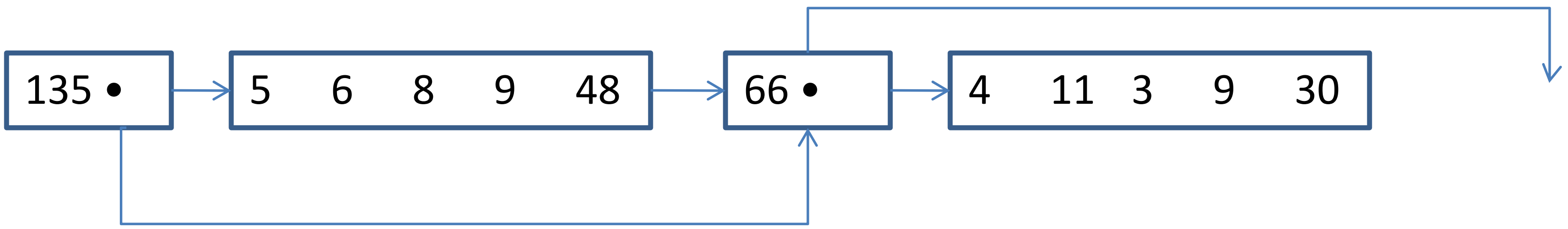


Assumption: result document contains all words

Idea: Move forward pointer with smallest value

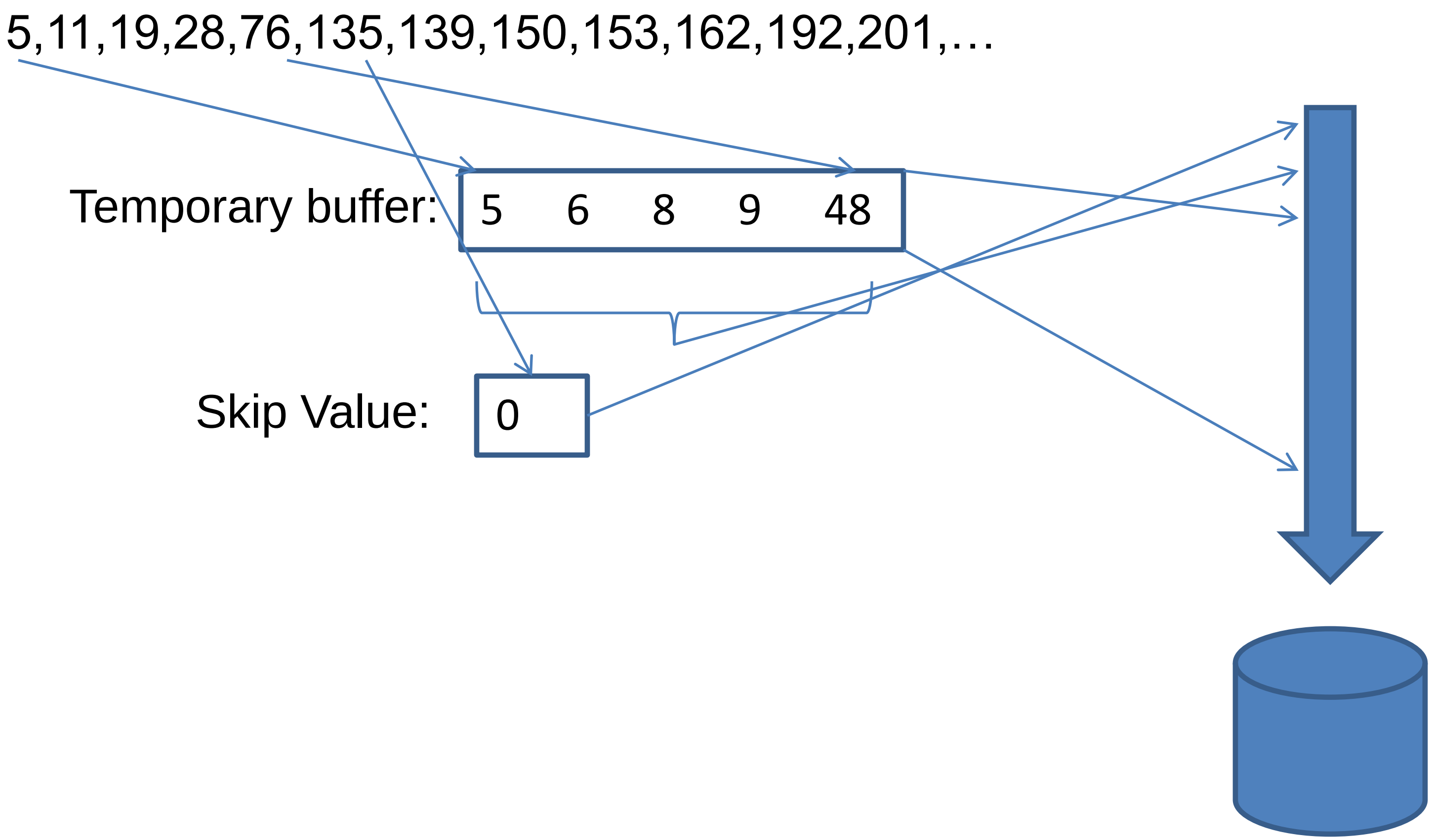
Skip lists

5,11,19,28,76,135,139,150,153,162,192,201,…



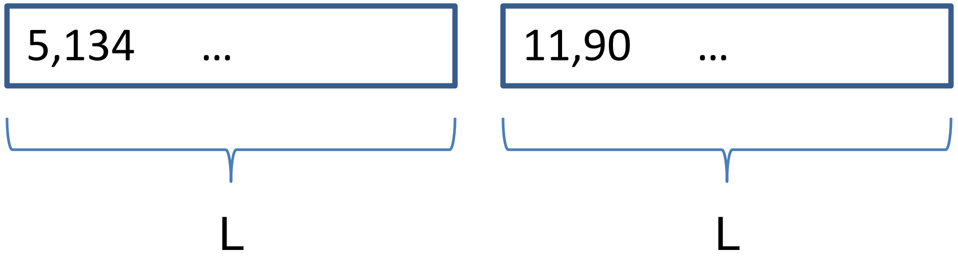
Reduce number of decompressions/comparisons

Skip list construction



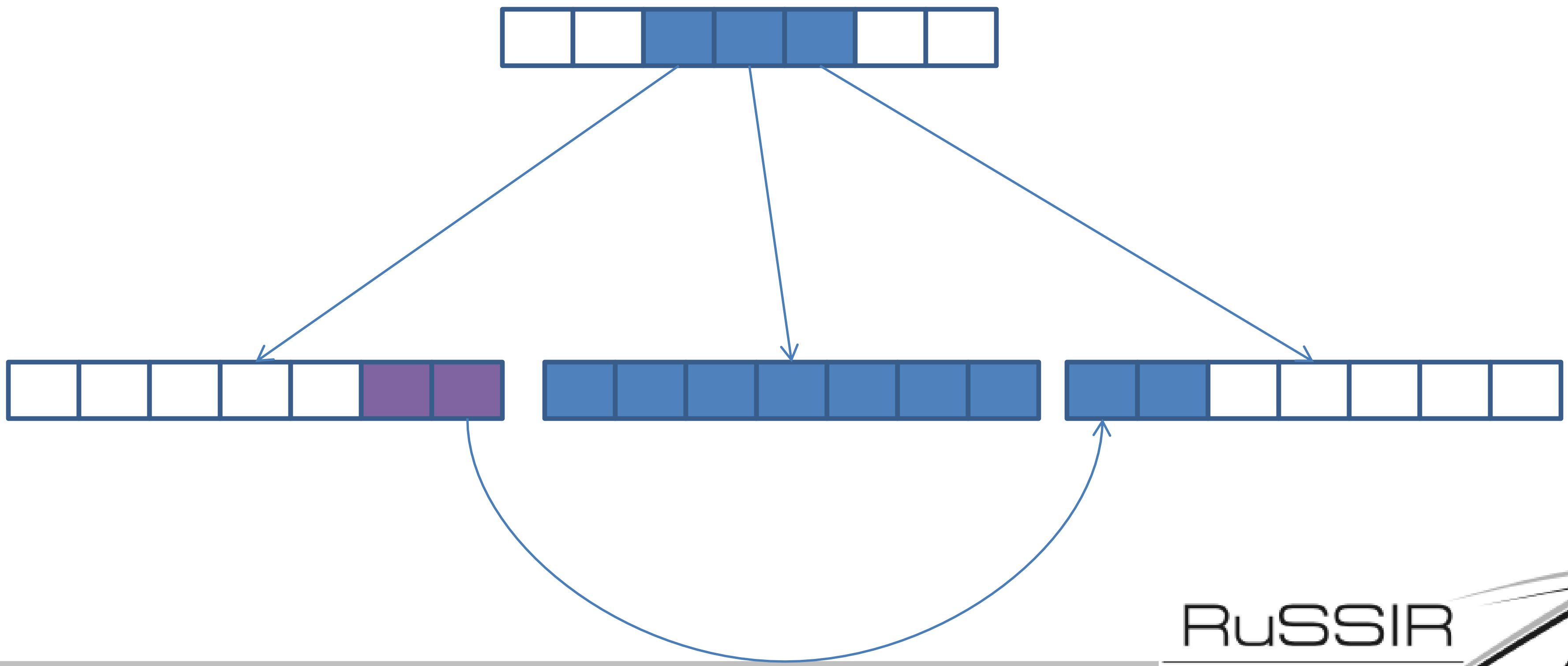
Block coding skip list

Block compression provides "automatic" skip lists



B-tree

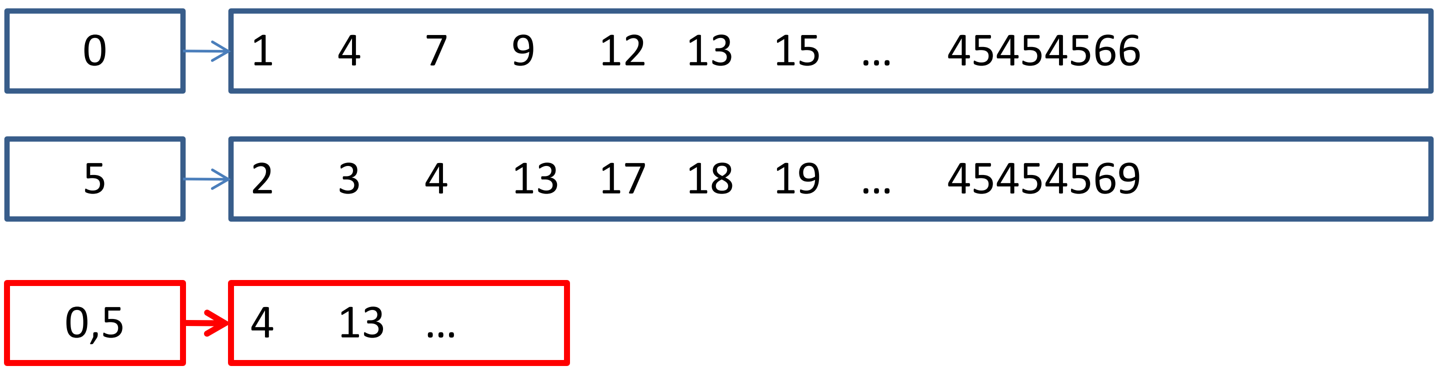
Another "automatic" skip list



Auxiliary index

Skip lists don’t help for frequent words

Example: 0 1 5



How to select phrases

- Frequent words
- Frequent phrases
  - From collection sample
  - Post-processing of index
- From query logs (static caching)

Early cancelation

User can't read list of 1,000,000 documents

Select top N (1000)

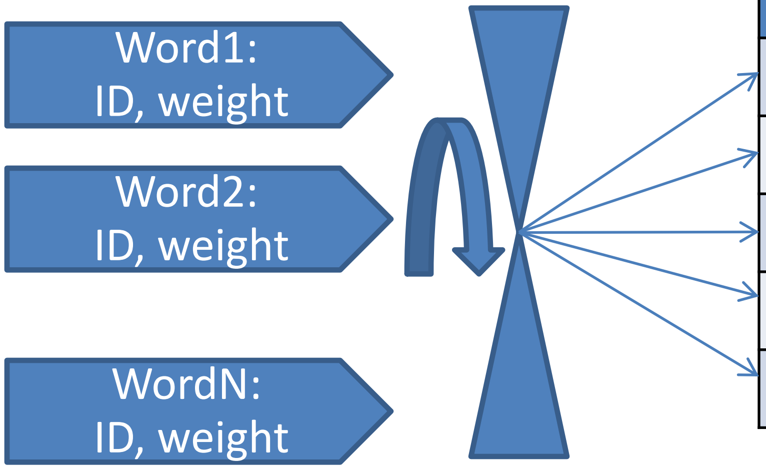
Rank = F (Word1, Word2,…, WordN)

WordX ↑ → Rank ↑

Rearranging data

Sort post-lists by weight

Word2: 150 <1000>, 28<620>, 19<610>, 5<605>,11<605>, 135<604>,…

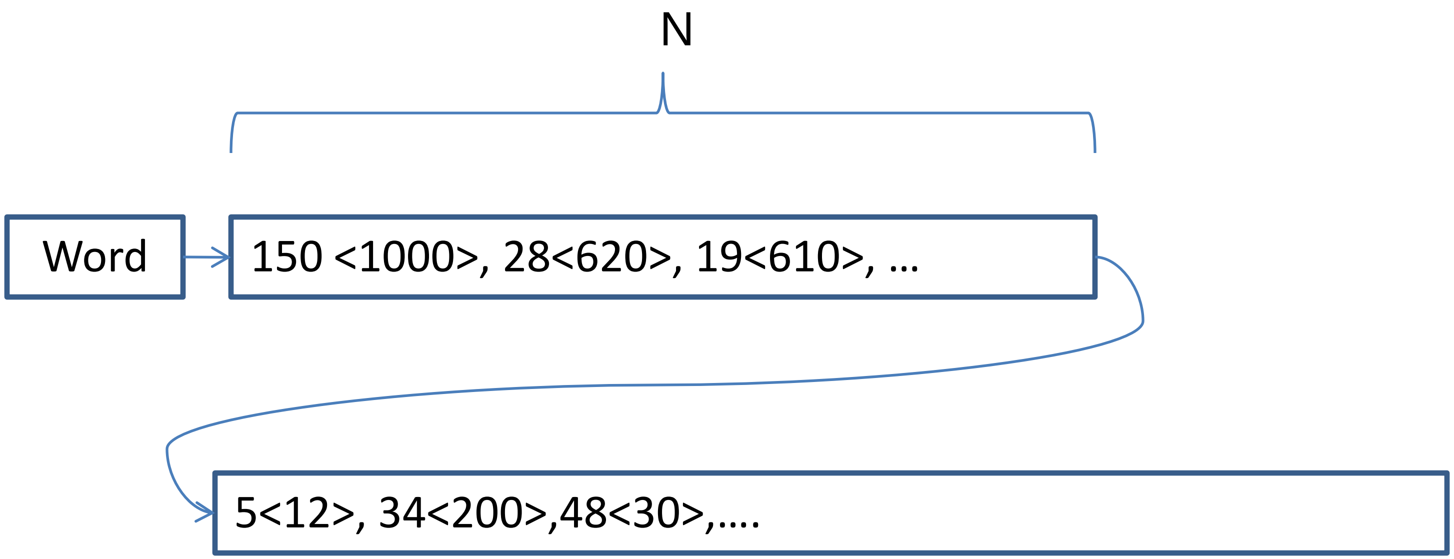


Word1	Word2		WordN
1000			
	450		300

Counter: N filled

Disadvantage: compression!!!

Rearranging part of data



Pruning

Facts:

Up to **75%** of index postings are never being in result

Idea: Remove "not important" postings

Pruning Approaches

"A"	{32[10],5[12],16[1],100[8]}
"B"	{16[23],124[24]}
"C"	{56[1]}

1. Term-oriented
2. Document-oriented
3. Language model pruning

Term-oriented Pruning

Term-oriented:

Remove “smallest” postings in list

“A”	{32<10>,5<12>,16<2>,100<8>,56<5>}
“B”	{16<23>,124<24>}
“C”	{56<1>}

Document-oriented pruning

Term-oriented:

Remove “smallest” postings in a document

“A”	{32<10>,5<12>,16<2>,100<8>,56<5>}
“B”	{16<23>,124<24>}
“C”	{56<1>}

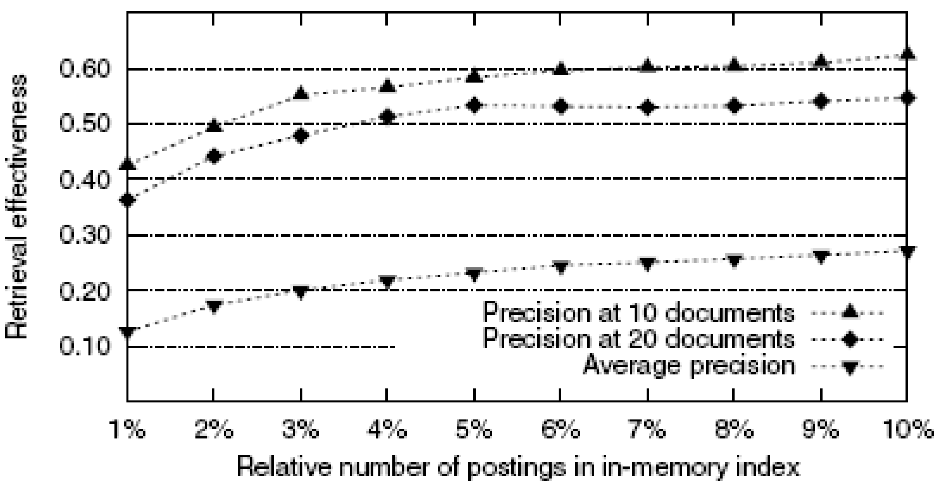
Pruning + LM

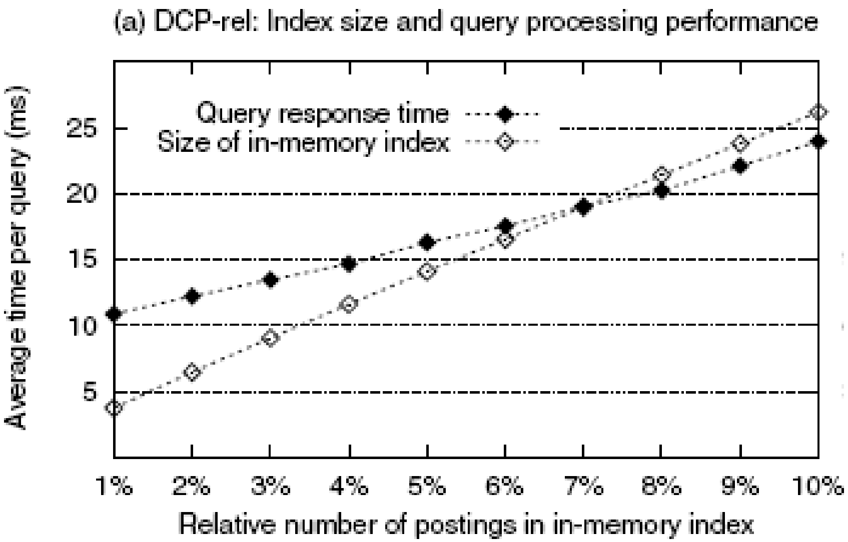
$(\text{Word}_1, \text{Word}_2, \ldots \text{Word}_n) \rightarrow P$

Language model pruning:

1. Create “Language model”
2. For every document select terms “not matching” the model
3. Put them into posting

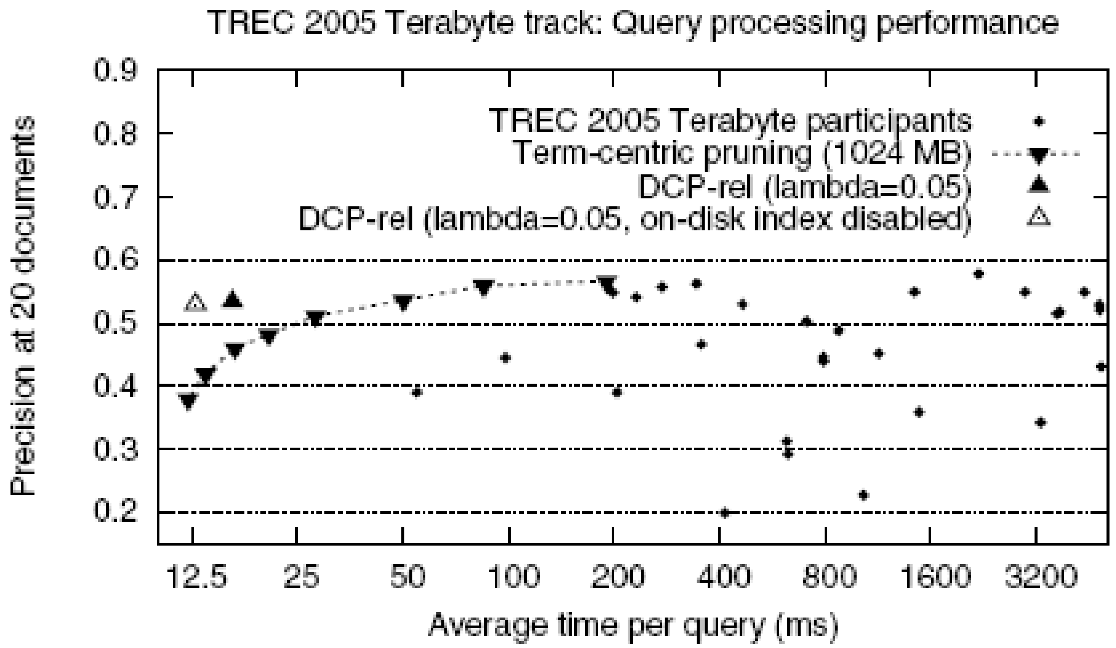
Pruning: reduce size





S.Buttcher, C.Clarke **A Document-Centric Approach to Static Index Pruning in Text Retrieval Systems,** *CIKM'06*

Pruning: improve speed



Allocating memory
[high performance]

1. Never alloc
2. Never, never realloc
3. Avoid free

Why: slow, fragmentation, concurrency, disk IO

Solution:

1. Use fixed size structures (FORTRAN)
2. Allocate everything in advance
3. Grow in big blocks

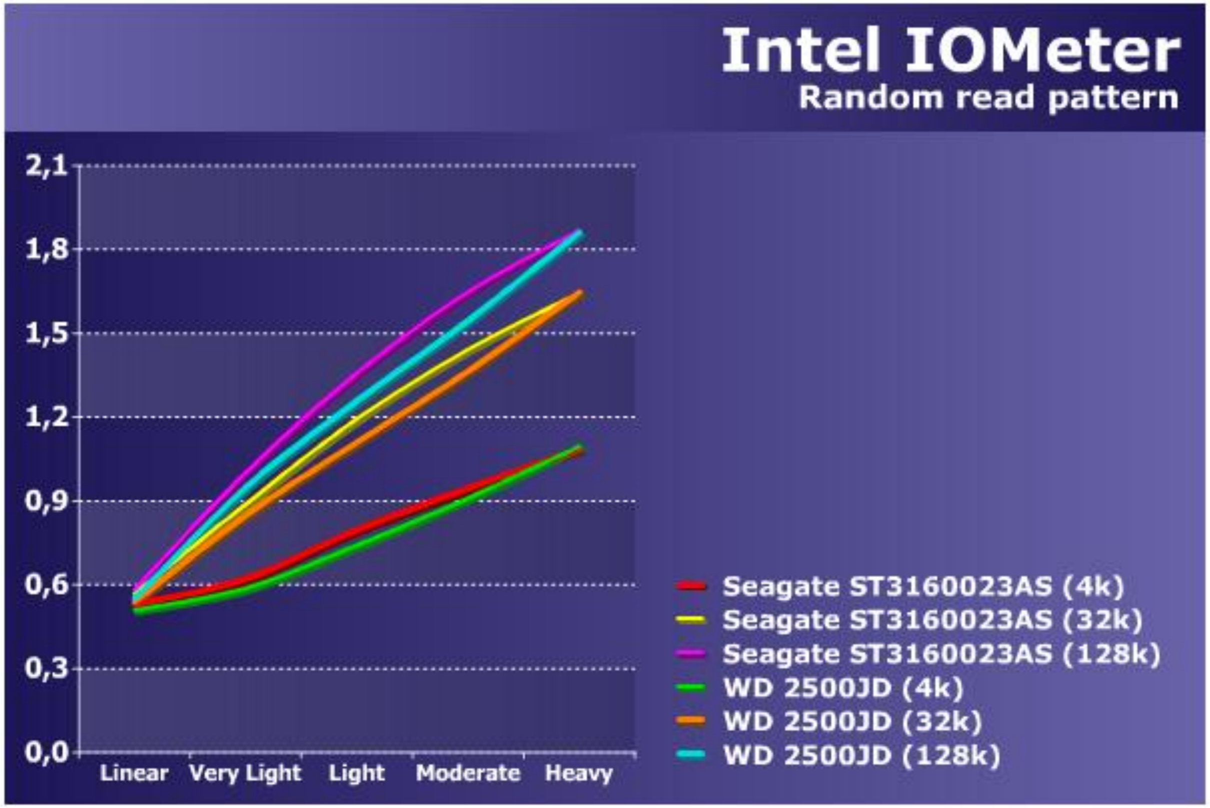
Implementation in different languages

C++: control everything, custom STL allocators

Java, .Net: use object pools, preallocate arrays

Python, Perl: standard structures, KISS

Disk access policy



Linear access is good, avoid "seeks"

Caching

Modern commodity computers 16 Gb

Dictionary size: 100-200 MB (mlns words)

Temporary search structures (1000 pq):

1000*1000*256 = 256 Mb

We have GBs of free memory!!!

Caching approaches

- Post-lists
- Search results
- Temporary results

Caching Postlists

- Compressed – duplicate OS file cache (do it if you can better)
- Decompressed – not effective memory usage (only if very complex compression and slow CPU)

Caching results

Pluses:

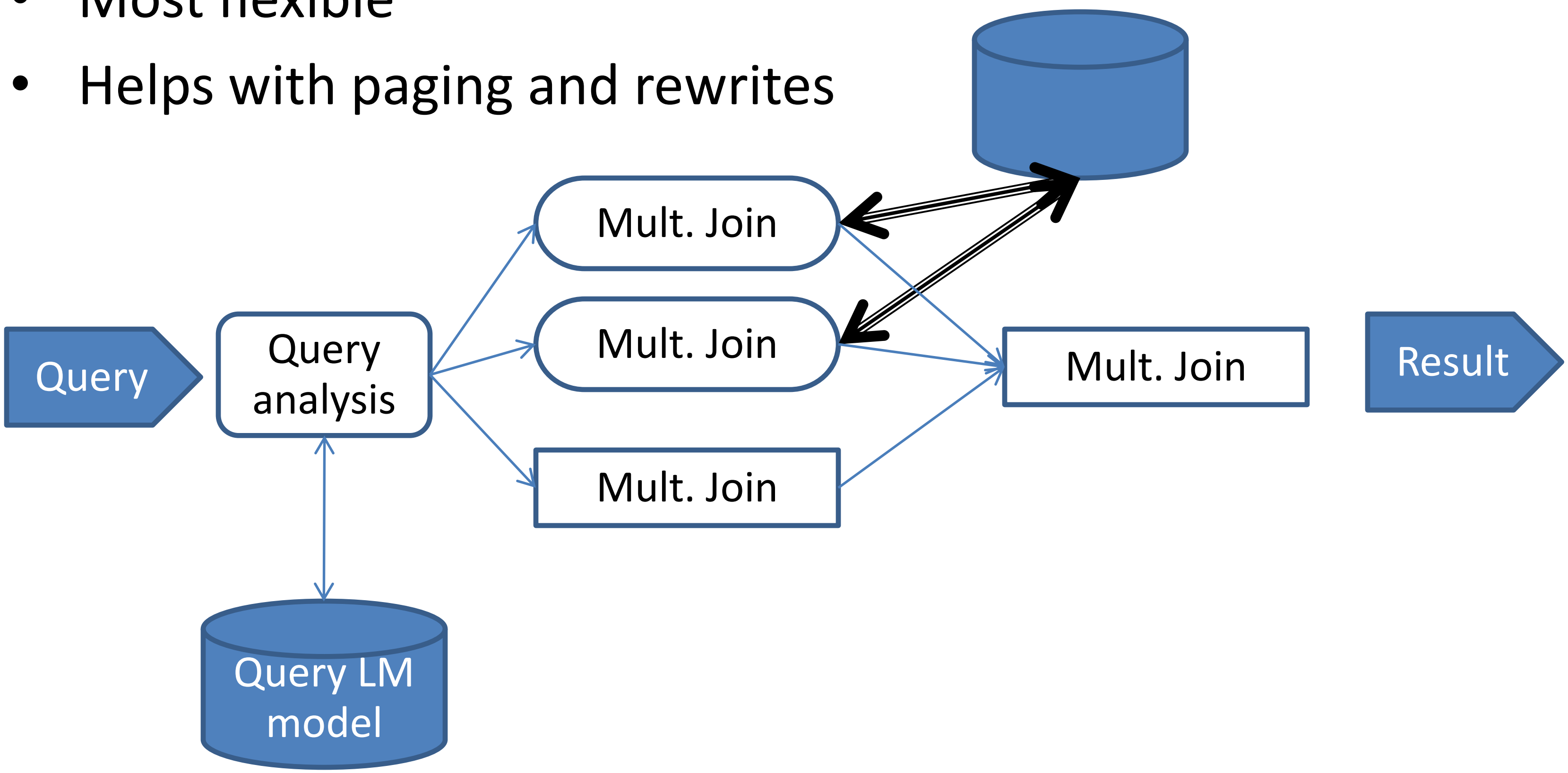
1. Simple
2. Helps with list paging

Minuses:

Most heavy queries don't repeat

Caching temporary results

- Most flexible
- Helps with paging and rewrites



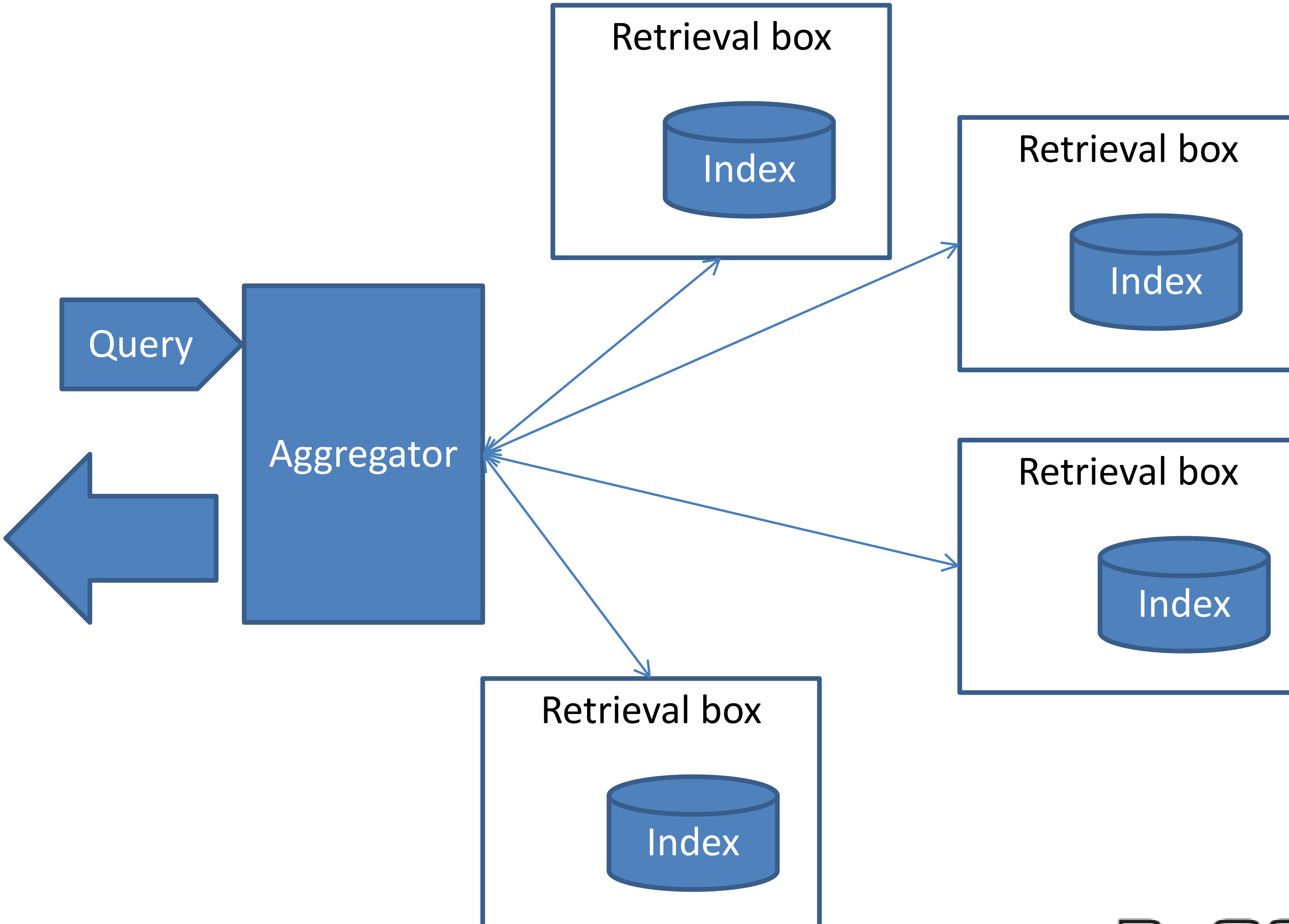
Caching temporary results [example]

Query: “Britney Spears album”

Subquery	Probability
Britney Spears album	1e-9
Britney Spears	1e-7 ←
Britney album	3e-9
Spears album	4e-9

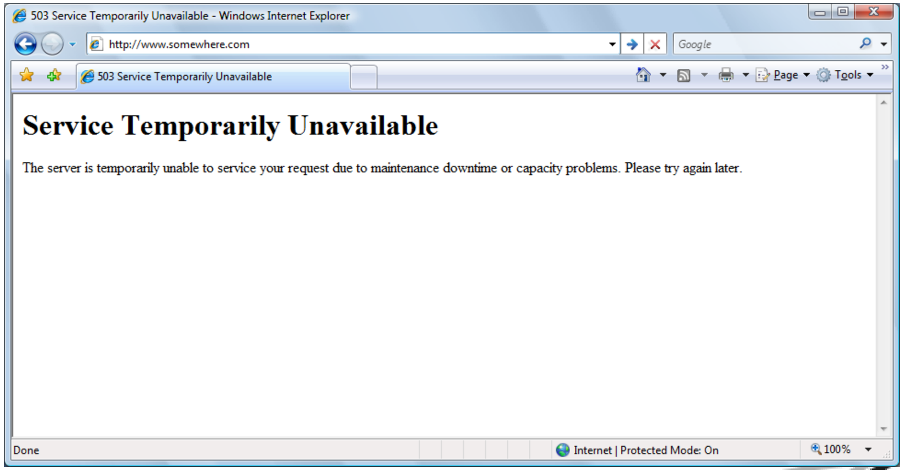
Problem: early cancelation? “Britney Spears naked”

Search in cluster



Interface in cluster

- RPC with timeouts
- Stateless protocol
- Failure prediction

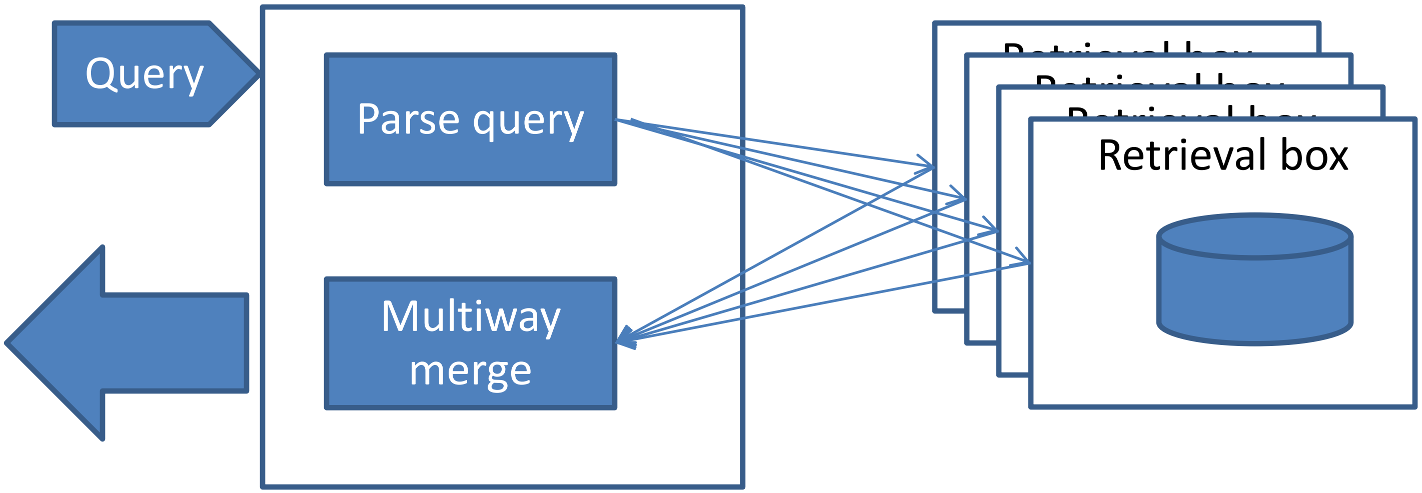


Index in cluster

Divide data

- By key (by words)
- By data (by documents)

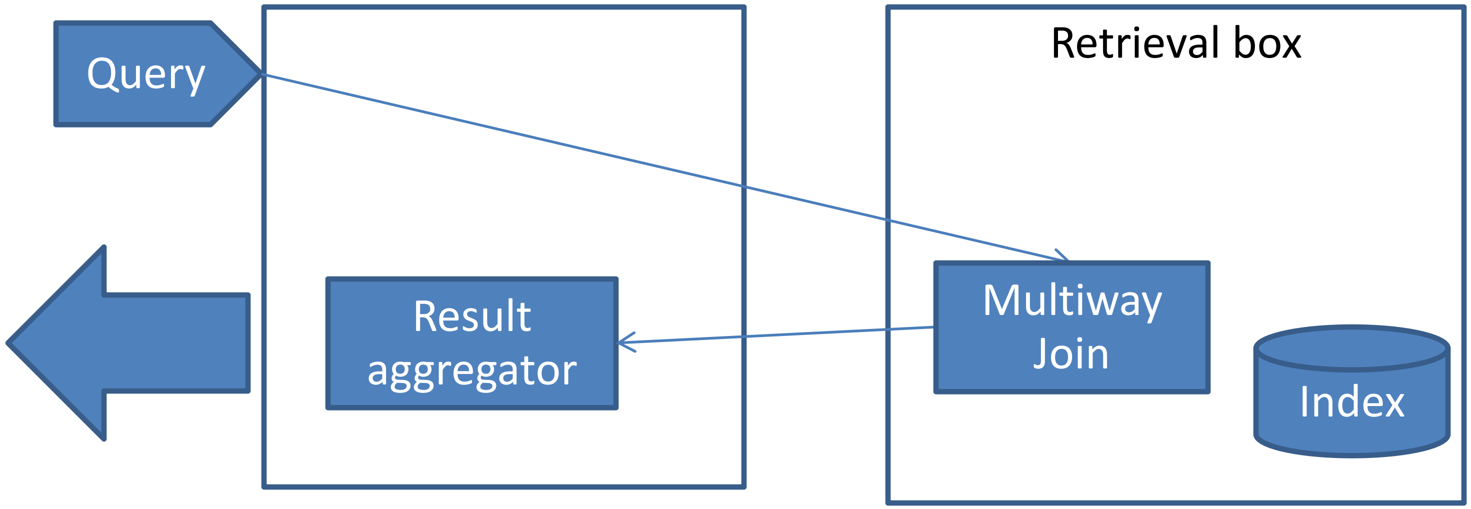
Distributed by words



Distribution function: letters, hash

Disadvantages: balance, early cancelation

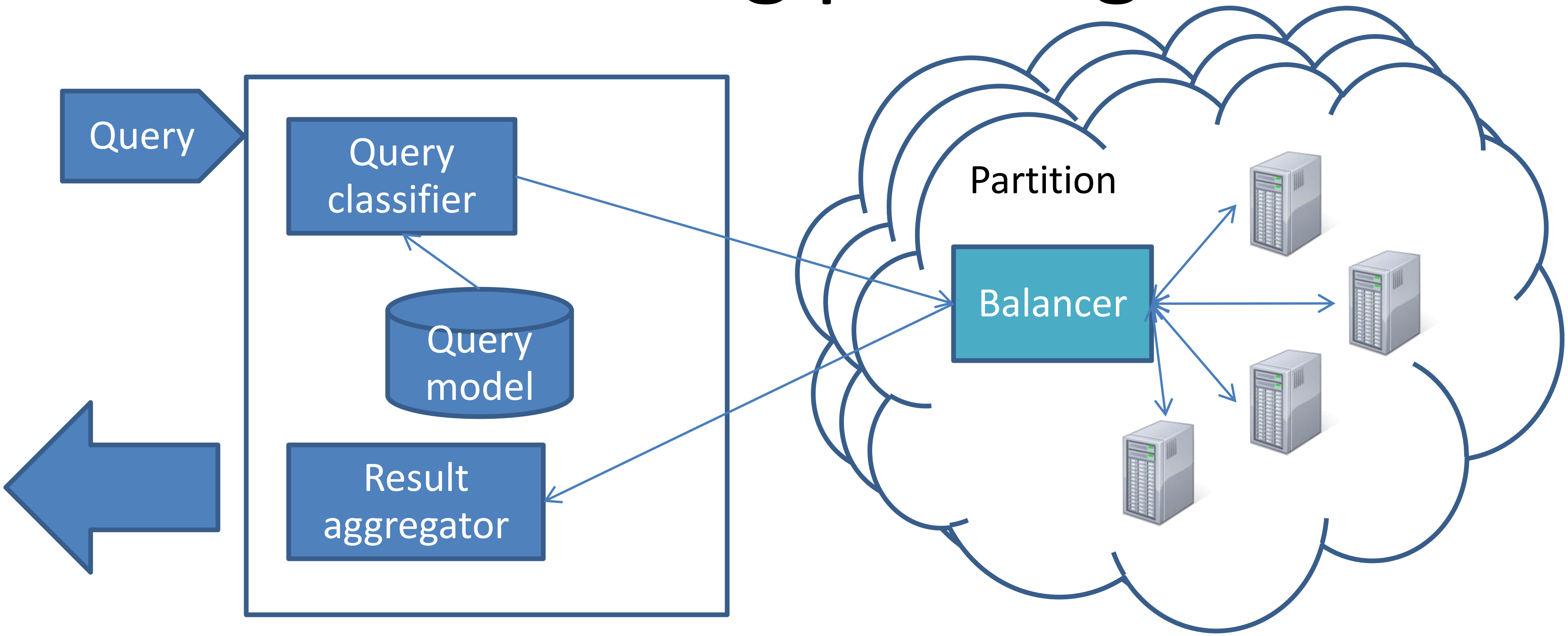
Distributed by documents



Distribution function: DocID, SiteID

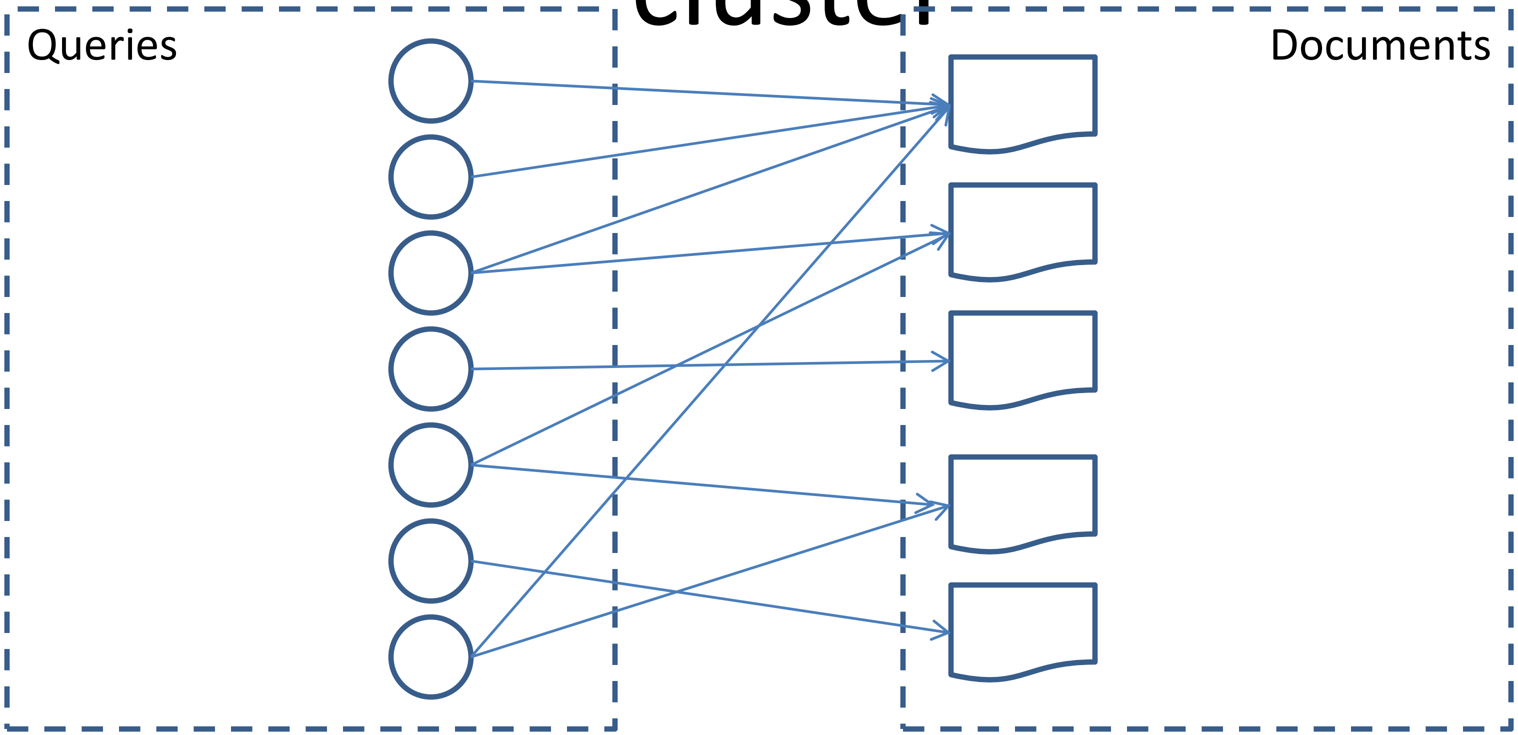
Problems: query to all servers (scalability)

Clustering pruning



Partition based on: document quality, geographical, topics, update frequency

How to build the query model for cluster



1. Query-click graph
2. Find subgraphs (minimal cut) – create models
3. Train classifiers

Summary

- Multiway join
- Skip lists/reordering
- Pruning
- Hardware issues
- Search in cluster

Q&A