

# On the Feasibility of Peer-to-peer Web Indexing and Search

Jingyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David R Karger, Robert Morris

# Introduction

- “*Is P2P Web search likely to work?*”
  - Good stress test
  - More resistant to maliciousness
  - No central point of failure

# Fundamental Constraints

- Total estimated index size is  $6 \times 10^{13}$
- Storage Constraints
  - 60k PCs at 1GB/PC
- Communication Constraints
  - 1MB/s assumed

# Partition by Document

- Documents divided up by host
- Each peer maintains its own index
- Each query is broadcast to all peers
- Estimated to exceed communication budget by 6x

# Partition by Keyword

- Keywords are divided up by host
- Each peer maintains the posting list for its associated keywords
- Problem: Index Intersection
  - On average, 530x communication budget
  - As bad as **4000x**, or worse

# So why bother with Partition by Keyword?

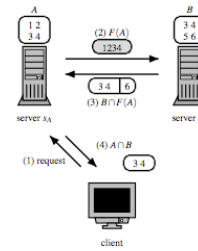
- A lot of existing research on fast inverted index intersection
- Using these techniques, Partition by Keyword can be made reasonable

# Caching and Precomputation

- Caching gains are modest: 38%
  - A lot of queries only show up once
- Precomputation: 50%
  - Intersection of only the most popular query term pairs

# Bloom Filters

- Idea
  - At node A, hash the matching documents for term X in to a table and send it
  - At node B, find out which documents for term Y hash in to the same places in the table
  - Send those document IDs back to A and filter out false positives



(b) Bloom filters help reduce the bandwidth requirement of "AND" queries. The gray box represents the Bloom filter  $F(A)$  of the set  $A$ . Note the false positive in the set  $B \cap F(A)$  that server  $s_B$  sends back to server  $s_A$ .

From [17]

# Bloom Filters

- One round gives compression ratio of 13
- For small searches, four rounds gives compression ratio of 40
- By compressing the filters themselves, get up to compression ratio of 50

# Gap Compression

- Basic idea: reduce the amount of space needed to express each Doc ID
- Why use 160 bits when we can use 32 or less?
- Compression ratio of 30

# Adaptive Set Intersection

- Uses structure in posting lists to avoid sending entire lists
- Example: no need to send entire list if we can tell that the intersection is null ahead of time
- Compression ratio of 40 with Gap Compression

# Clustering

- Cluster similar documents in to close docIDs
- Idea: improve adaptive set intersection
- With AS, compression ratio is 75

# Compromising Result Quality

- Try to return partial results by attempting to only send the best matches
  - Not as good as centralized search
  - Can be mitigated with feedback
- 50x reduction in in communication cost
  - Particularly good for previously expensive queries

# Compromising P2P Structure

- Break homogeneous structure of the P2P network to take advantage of network structure
- For example, replicate the inverted index per ISP instead of having one copy

# Conclusion

- Lots of ways to improve Partition by Keyword
- Compromising the Result Quality seems to be the biggest win
- In terms of making the largest queries reasonable

| Technique              | Improvement |
|------------------------|-------------|
| Caching                | 1.5×        |
| Precomputation         | 2×          |
| Bloom Filters          | 50×         |
| Gap Compression (GC)   | 30×         |
| Adaptive Set (AS) + GC | 40×         |
| Clustering + GC + AS   | 75×         |

Table 1: Optimization Techniques and Improvements

# Conclusion

- Addresses unstructured keyword search and not much else
- No concrete development

# Efficient Peer-To-Peer Searches Using Result Caching

Bobby Bhattacharjee, Sudarshan Chawathe,  
Vijay Gopalakrishnan, Pete Keleher, Bujor Silaghi

# Introduction

- Lookup is easy in P2P
- Search is less obvious
- Idea: cache conjunctions of search terms
- Need: efficient data structure for the cache
  - View Tree

# Data & Query Model

- Each data item has a unique name
- Each item has searchable meta-data
- Meta-data are ordered sets of name/value pairs
  - Boolean or real valued

# View Trees

- A view is some conjunction of search terms that we can cache
- Basic Idea: organize the views in to a trie spread across the DHT
- Each node of the trie has its view hashed in to the DHT, and it knows its children

# Searching

- First, find a prefix
- Then do a depth first search to add more useful terms

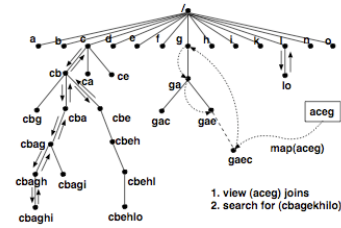


Figure 2: Example of a node joining the View Tree.

# Balancing the Tree

- By default, the trie will be unbalanced toward search terms that show up early in the normalized form
- Idea: have a standard permutation after normalizing to randomize position in the trie

# Maintaining the Tree

- Mostly, we can discard things when indexes get old and nodes disappear
- Tree integrity maintained with soft-state
- New nodes can be added in the middle of the tree by reassigning child pointers

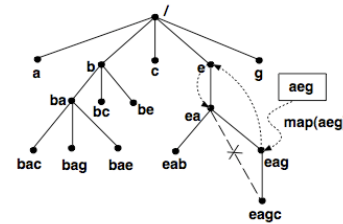


Figure 3: Maintaining the tree when a new node joins.

# Results

- Experiment 1: By varying the levels of caching, we find that retaining even only two levels reduces communication by 92%
- Experiment 2: By varying use of the “working set”, we find that performance is dependent on locality
  - But entirely random queries still benefit

# Conclusion

- Concrete tool for improving search in P2P systems
- Do the experiments reflect real query traffic?
  - Addresses keyword search
  - What about range search?
    - Presumably, “att  $\geq$  constant” can be cached as a term