Anatomy of a Large-Scale Hypertextual Web Search Engine by Sergey Brin and Lawrence Page (1997)

Presented By Wesley C. Maness

Outline

- Desirable Properties
- Problem; Google's reasons
- Architecture
- PageRank
- Open Problems & Future Direction

Desirable Properties wrt Google

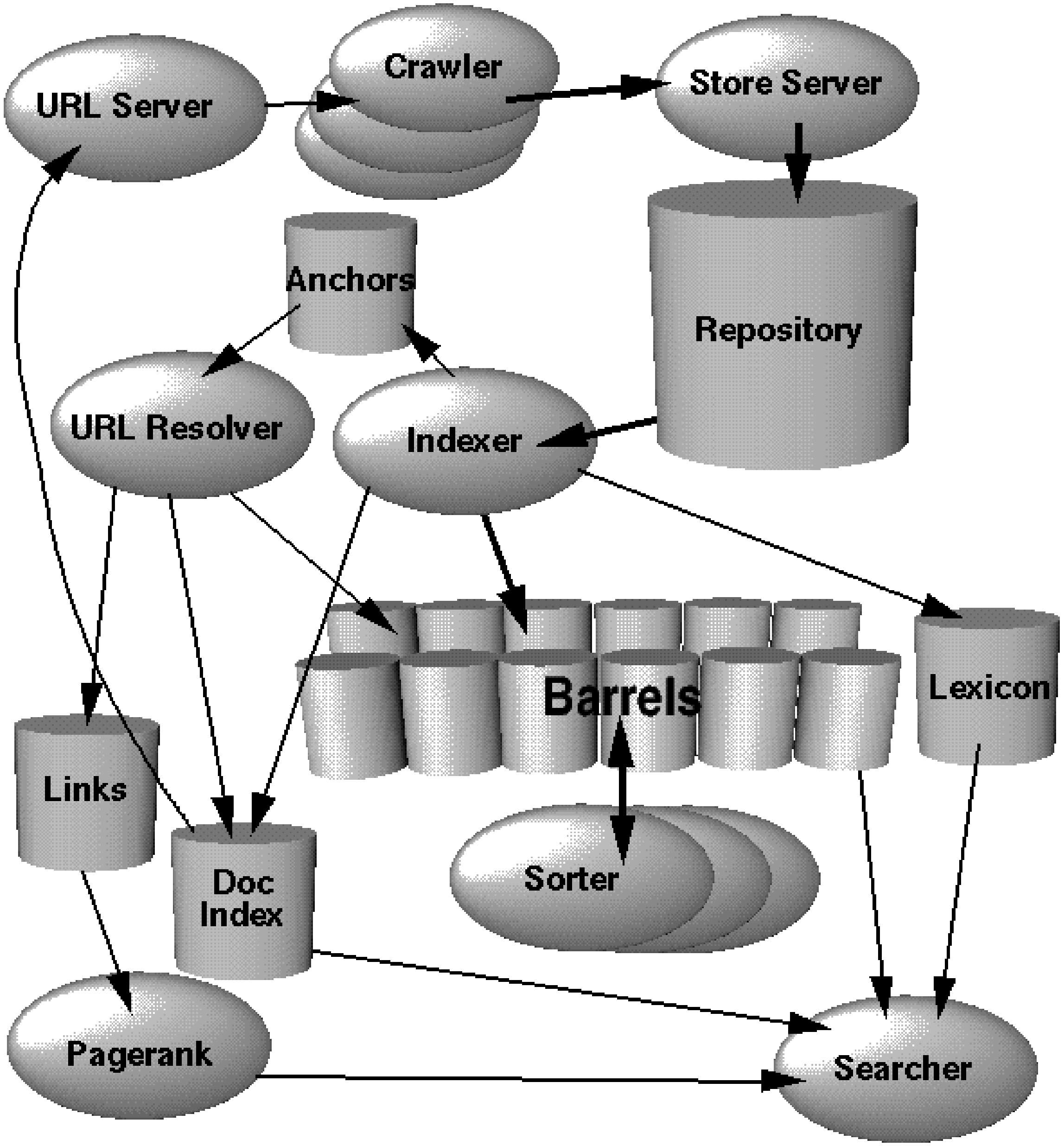
- Input
  - Keyword(s)
- Output
  - Will return to the user what the user wants/needs and NOT what the search engine ***thinks*** you want/need.

The Problems then and current

- It isn’t easy to search when you consider your search space and the properties of your search space.
- Web is vast and growing exponentially
- Web is heterogeneous
  - ASCII
  - HTML
  - Images
  - Video files
  - Java applets
  - Machine generated files (log files, etc.)
  - Etc.
- Web is volatile
- Distributed
- Freshness
- Human Maintained Lists cannot keep up
- External meta information that can be inferred from a document may or may not be accurate about the document
- Google had the solution then…

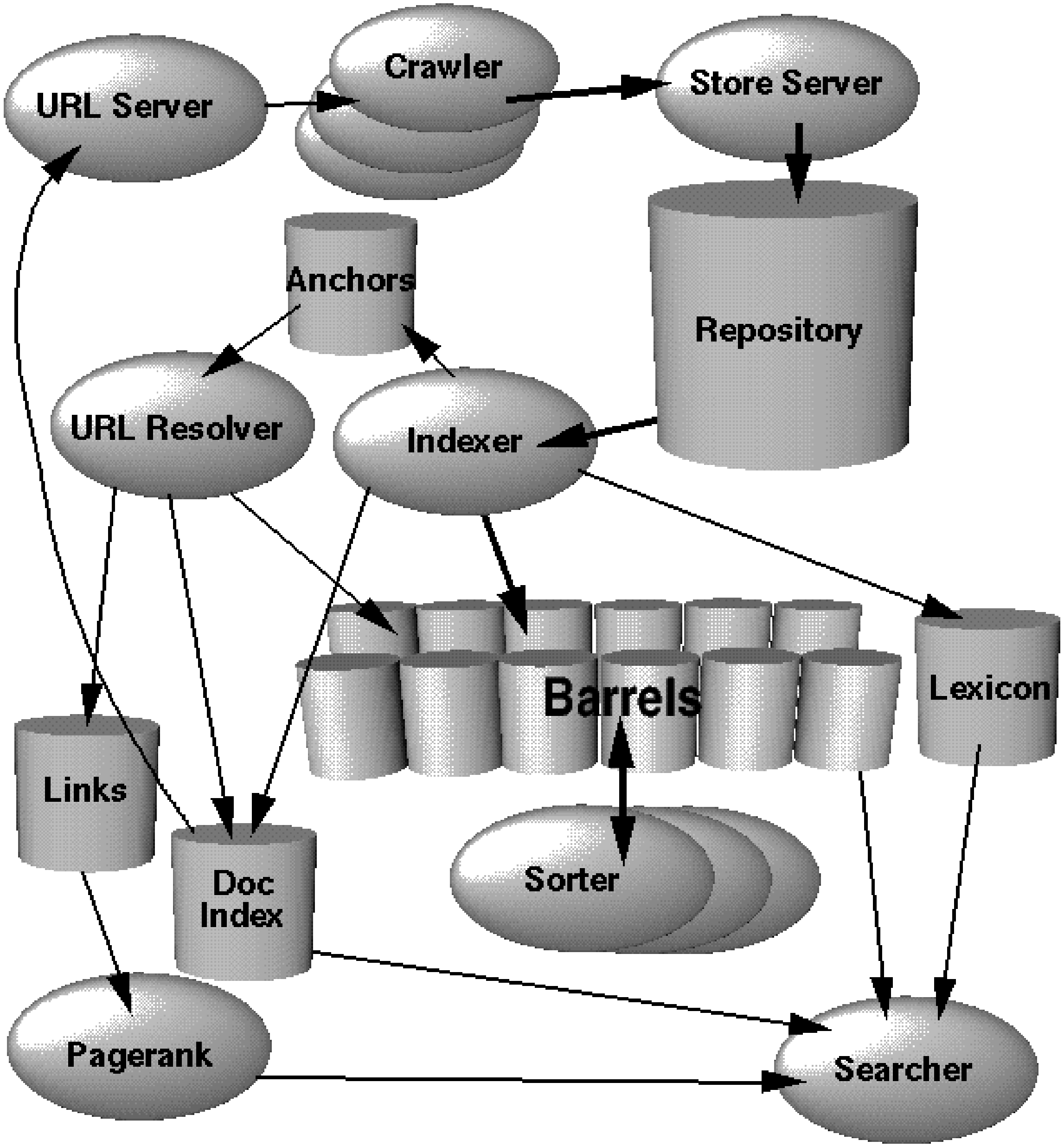
Google Architecture(1)

- **Crawler** – crawls the web
- **Urlserver**: sends links to the Webcrawler to navigate
- **Storeserver**: stores pages crawled by the webserver
- **Indexer**: retrieves the stored webpages
  - parses each document
  - converts the words into hit lists
  - distributes the words to barrels for storage
  - parses out all links and stores them in an anchor file
- **UrlResolver**: converts links to absolute Urls
  - Converts these Urls to DocID's
  - Stores them in the forward index
- **Sorter**: converts the barrels of DocID's to WordID's
  - Resorts the barrels by WordID's
  - Uses the WordID's to create an inverted Index
- **Searcher**: responds to querries using PageRank, inverted Index, and DumpLexicon



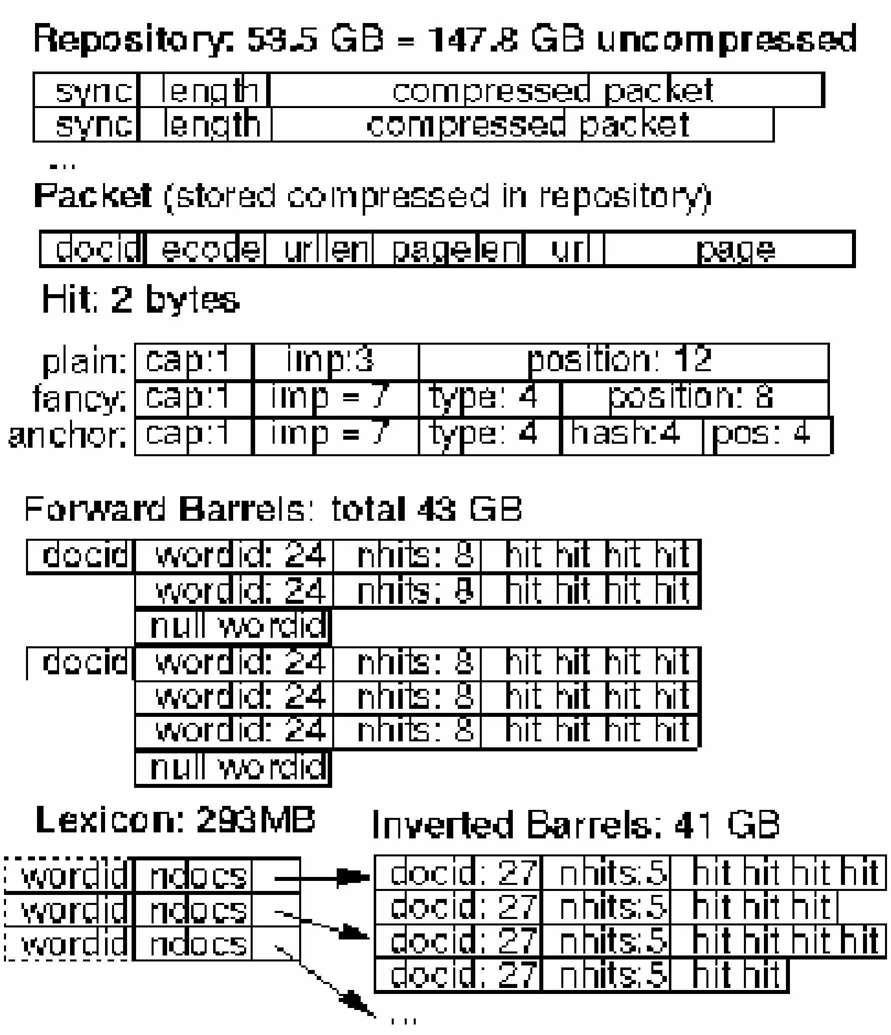
Google Architecture(2)

- **Repository** - Stores the html for every page crawled Compressed using zlib
- **Doc Index -** Keeps information about each document Sequentially stored, ordered by DocID Contains:
  - Current document status
  - Pointer into the repository
  - Document checksum
  - File for converting URLs to DocID's
  - If the page has been crawled, it contains: A pointer to DocInfo -> URL and title If the page has not been crawled, it contains: A pointer to the URLList -> Just the URL
- **Lexicon** is stored in memory and contains:
  - A null separated word list
  - A hash table of pointers to these words in the barrels (for the Inverted Index)
  - An important feature of the Lexicon is that it fits entirely into memory (~14 Million)



Google Architecture(3)

- **Forward Index -** Stored in (64)barrels containing:
  - A range of WordID's, The DocID of a pages containing these words, A list of WordID's followed by corresponding hit lists,Actual WordID's are not stored in the barrels; instead, the difference between the word and the minimum of the barrel is stored, This requires only 24 bits for each WordID,Allowing 8 bits to hold the hit list length
- **Inverted Index -** Contains the same barrels as the Forward Index, except they have been sorted by docID’s, All words are pointed to by the Lexicon, Contains pointers to a doclist containing all docID’s with their corresponding hit lists.
  - The barrels are duplicated
  - For speed in single word searches



Repository: 53.5 GB = 147.8 GB uncompressed

sync | length | compressed packet
sync | length | compressed packet
...

Packet (stored compressed in repository)

docid | ecode | urllen | pagelen | url | page

Hit: 2 bytes

plain: cap:1 | imp:3 | position: 12
fancy: cap:1 | imp = 7 | type: 4 | position: 8
anchor: cap:1 | imp = 7 | type: 4 | hash:4 | pos: 4

Forward Barrels: total 43 GB

docid | wordid: 24 | nhits: 8 | hit hit hit hit
wordid: 24 | nhits: 8 | hit hit hit hit
null wordid
docid | wordid: 24 | nhits: 8 | hit hit hit hit
wordid: 24 | nhits: 8 | hit hit hit hit
wordid: 24 | nhits: 8 | hit hit hit hit
null wordid

Lexicon: 293 MB

wordid | ndocs
wordid | ndocs
wordid | ndocs

Inverted Barrels: 41 GB

docid: 27 | nhits:5 | hit hit hit hit
docid: 27 | nhits:5 | hit hit hit
docid: 27 | nhits:5 | hit hit hit hit
docid: 27 | nhits:5 | hit hit
...

Hit Lists

- A list of occurrences of each word in a particular document
  - Position
  - Font
  - Capitalization
- The hit list accounts for most of the space used in both indices
- Uses a special compact encoding algorithm
  - Requiring only 2 bytes for each hit
- The hit lists are very important in calculating the Rank of a page
- There are two different types of hits:
- Plain Hits: (not fancy)
  - Capitalization bit
  - Font Size (relative to the rest of the page) -> 3 bits
  - Word Position in document -> 12 bits
- Fancy Hits (found in URL, title, anchor text, or meta tag)
  - Capitalization bit
  - Font Size - set to 7 to indicate Fancy Hit -> 3 bits
  - Type of fancy hit -> 4 bits
  - Position -> 8 bits
- If the type of fancy hit is an anchor, the Position is split:
  - 4 bits for position in anchor
  - 4 bits for hash of the DocID the anchor occurs in
- The length of the hit list is stored before the hits themselves

What is PageRank? And why?

- What is PageRank?:
  - Assumptions
    - A page with many links to it is more likely to be useful than one with few links to it
    - The links from a page that itself is the target of many links are likely to be particularly important
  - PageRank is a citation importance ranking
    - Approximated measure of importance or quality
    - Number of citations or backlinks

- Why?:
  - Attempts to model user behavior
  - Captures the notion that the more a page is pointed to by “important” pages, the more it is worth looking at, votes
  - Takes into account “assumed” global structure of web
  - Assumption: Important pages are pointed to by other important pages.
  - Link “A ⇒ B” often means “A thinks B is important”

$$PageRank(P) = (1-d) + d\left(\sum_{i=1}^{C(P)} \frac{PageRank(T_i)}{C(T_i)}\right)$$

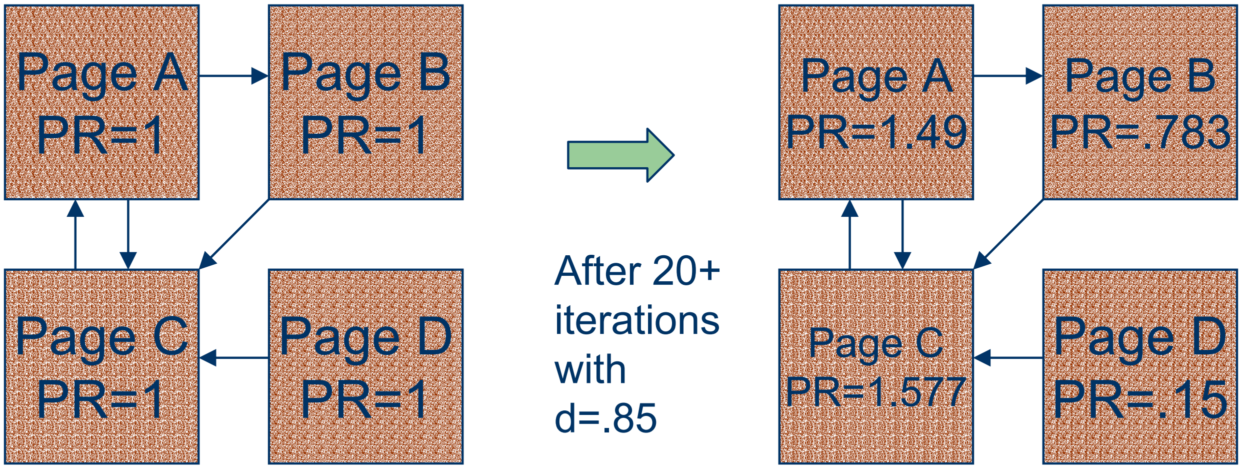
PageRank Calculation

- Variables:
  - d: damping factor, normally this is set to 0.85
  - Ti – page pointing to page P
  - PageRank(Ti): PageRank of page Ti pointing to page P
  - C(Ti): the number of links going out of page Ti

- How is it calculated?
  - 1. Spider the web to generate NxN link matrix A
    - A[i,j] = 1 iff page P_i contains link to page P_j
  - 2. Simple iterative algorithm:
    - Initialize PageRank[P_i]=1 for each page P_i
    - Repeat many times

- (Jan 1998) PR converges to a reasonable tolerance on a link database of 322Mill in 52 iterations. Half the data took 45 iterations.

PageRank Example



Page A PR=1

Page B PR=1

Page C PR=1

Page D PR=1

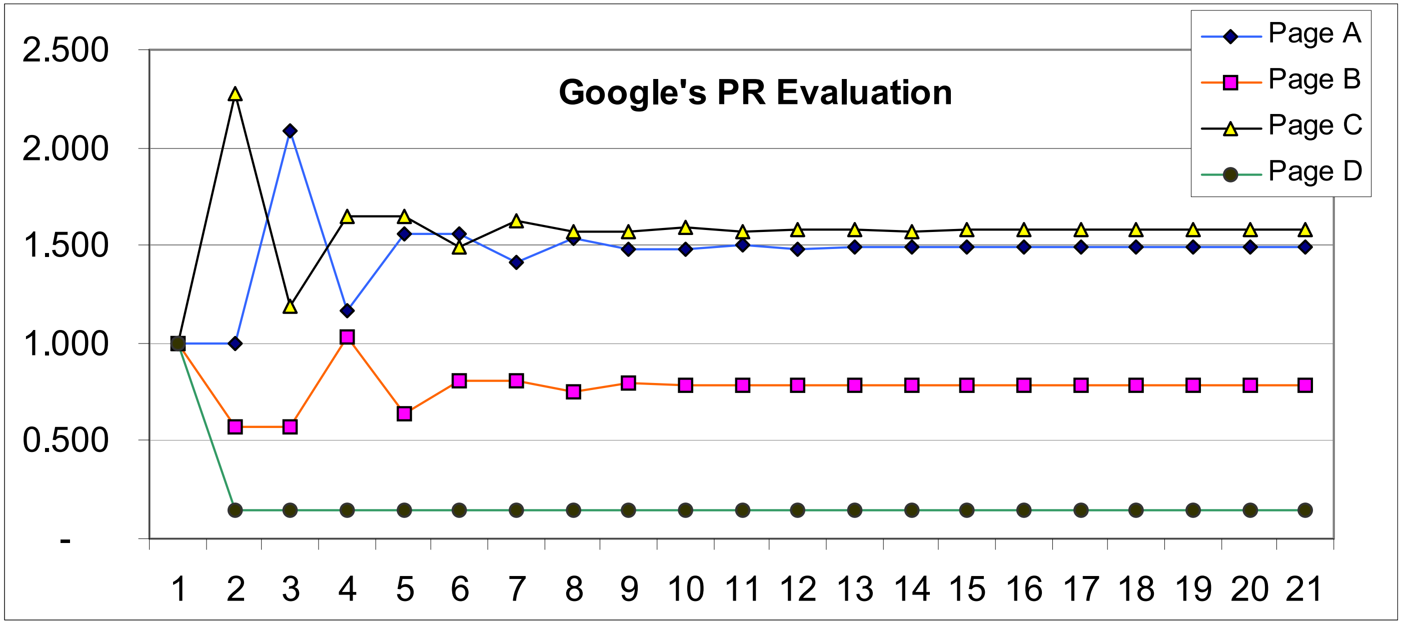
After 20+ iterations with d=.85

Page A PR=1.49

Page B PR=.783

Page C PR=1.577

Page D PR=.15



Sample Google Query Evaluation

1. Parse the query.
2. Convert words into wordIDs.
3. Seek to the start of the doclist in the short barrel for every word.
4. Scan through the doclists until there is a document that matches all the search terms.
5. Compute the rank (Would be a weighted computation of PR and the hitlist) of that document for the query.
6. If we are in the short barrels and at the end of any doclist, seek to the start of the doclist in the full barrel for every word and go to step 4.
7. If we are not at the end of any doclist go to step 4.
8. Sort the documents that have matched by rank and return the top k.

Summary of Key Optimization Techniques

- Each crawler maintains its own DNS lookup cache
- Use flex to generate lexical analyzer with own stack for parsing documents
- Parallelization of indexing phase
- In-memory lexicon
- Compression of repository
- Compact encoding of hitlists accounting for major space savings
- Document index is updated in bulk
- Critical data structures placed on local disk

Ongoing/Future Work

- The PageRank is dead argument from **The act of Google trying to "understand" the web caused the web itself to change.** Jeremy Zawodny – i.e. PageRank's assumption of the citation model had major impacts on web site layout, along with the ever-changing web. 'Google Bombing' – i.e. the web search for "miserable failure" due to bloggers. Also, AdWords->AdSense and the public assumption of a conspiracy – note the Florida Algorithm. More efficient means of rank calculation.

- Personalization for results – give you what you want, usage of cookies, etc. – based on previous searches. This works very well with contextual paid listings (purchase of Applied Semantics) Yahoo has the advantage of user-lock-in and being a portal. (Mooter accomplishes this by learning or remembering previous search results per user and re-ranks search results.

- Context Sensitive results

- Natural Language queries askMSR

- Cluster-based/Geographic-based search results (Mooter)

- Authority-based – Teoma's technology search results are weighted by authorities that are determined via a citation weighted model (similiary to PR) and cross-verified by human/subject-specific experts. - Highly accurate not scalable

Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks

Hong Ge

Peer-to-Peer Information Retrieval

- Distributed Hash Table (DHT)
  - CAN, Chord, Pastry, Tapestry, etc.
  - Scalable, fault tolerant, self-organizing
  - Only support exact key match
    - K_d=hash (“books on computer networks”)
    - K_q=hash (“computer network”)
- Extend DHTs with content-based search
  - Full-text search, music/image retrieval
- Build large-scale search engines using P2P technology

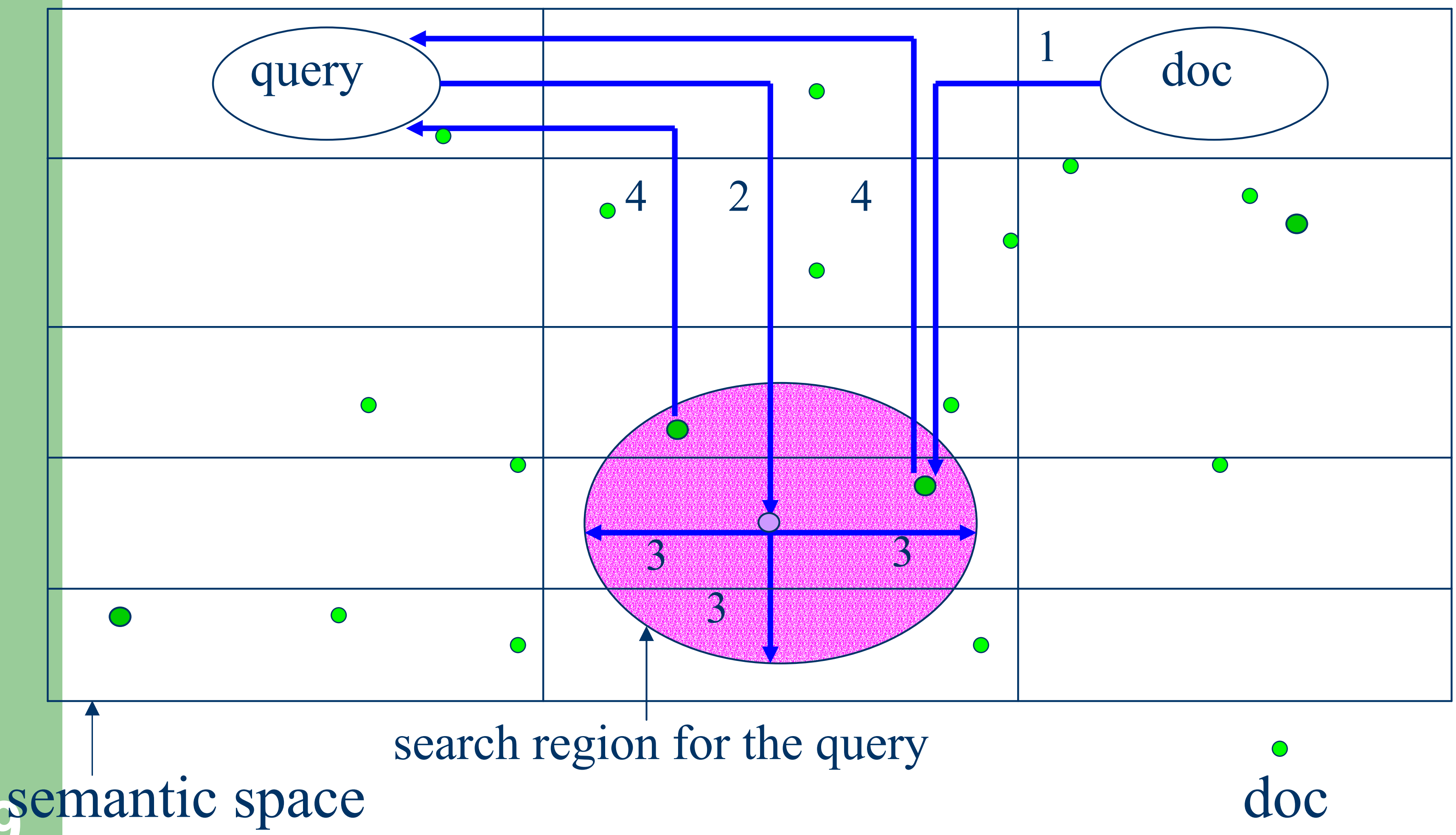
Focus and Approach in pSearch

- Efficiency
  - Search a small number of nodes
  - Transmit a small amount of data
- Efficacy
  - Search results comparable to centralized information retrieval (IR) systems
- Extend classical IR algorithms to work in DHTs, both efficiently and effectively

Outline

- Key idea in pSearch
- Background
  - Information Retrieval (IR)
  - Content-Addressable Network (CAN)
- P2P IR algorithm
- Experimental results
- Open issues

pSearch Illustration



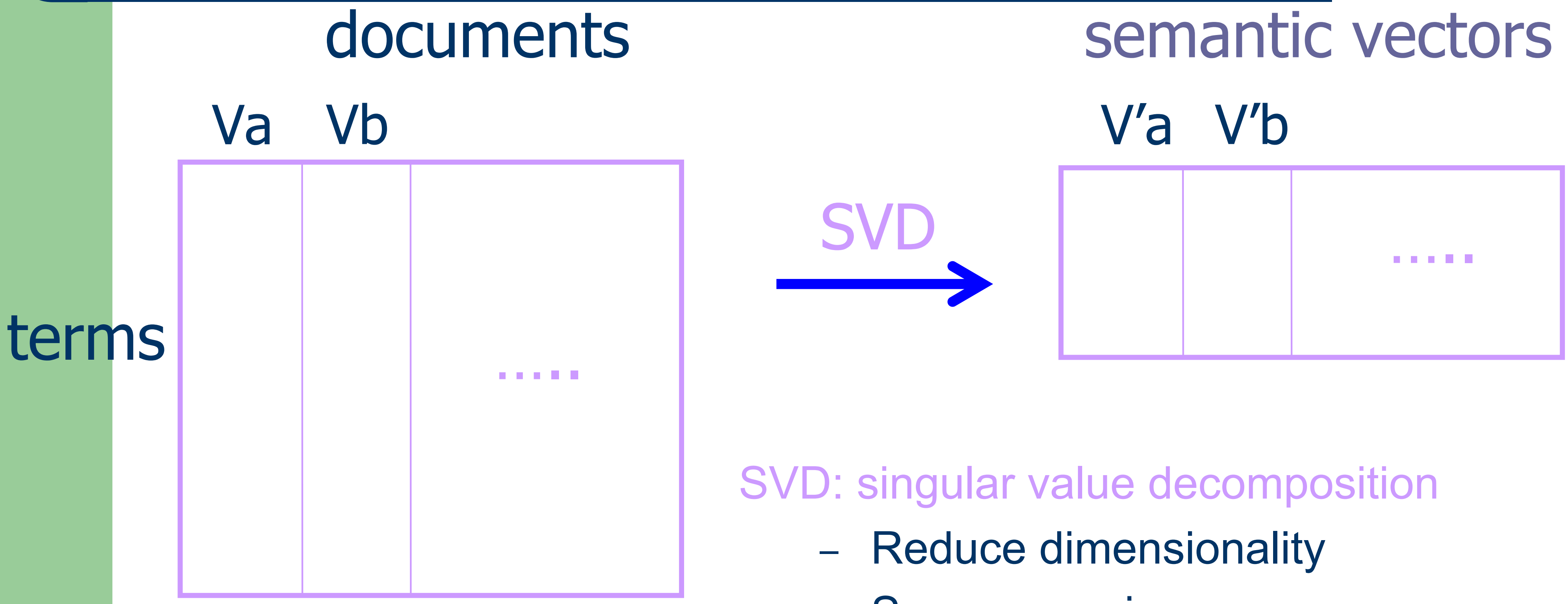
Background

- Statistical IR algorithms
  - Vector Space Model (VSM)
  - Latent Semantic Indexing (LSI)
- Distributed Hash Table (DHT)
  - Content-Addressable Network (CAN)

Background: Vector Space Model

  - *d* documents in our corpus
  - *t* terms (vocabulary)
  - Represented by a $t \times d$ *term-document* matrix A
- Elements a_{ij}
  - $a_{ij} = l_{ij}\, g_i$
    - g_i is a *global* weight corresponding to the importance of term *i* as an index term for the collection
      - Common words have low global weights
    - l_{ij} is a *local* weight corresponding to the importance of term *i* in document *j*

Background:
Latent Semantic Indexing



documents

Va Vb

terms

.....

SVD

semantic vectors

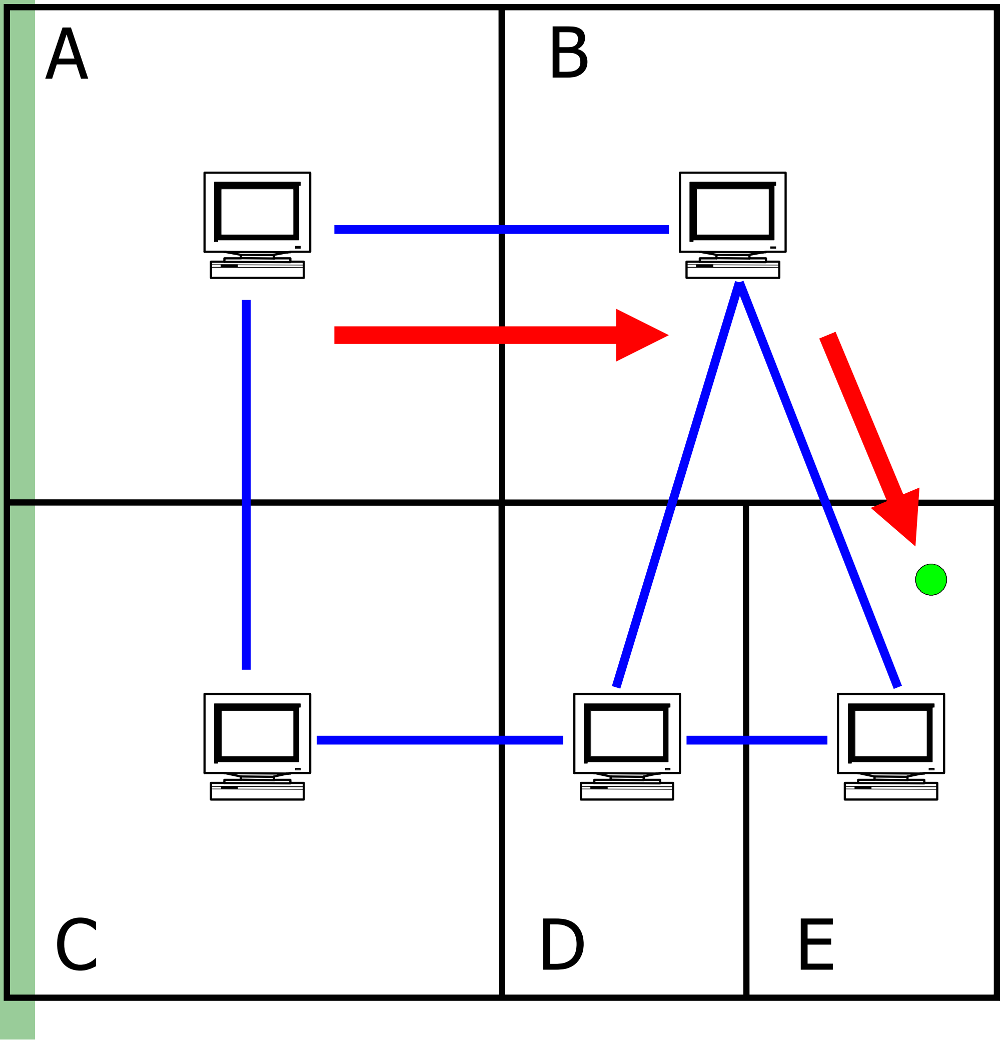
V'a V'b

.....

SVD: singular value decomposition

- Reduce dimensionality
- Suppress noise
- Discover word semantics
  - Car <-> Automobile

Background: Content-Addressable Network



- Partition Cartesian space into zones
- Each zone is assigned to a computer
- Neighboring zones are routing neighbors
- An object key is a point in the space
- Object lookup is done through routing

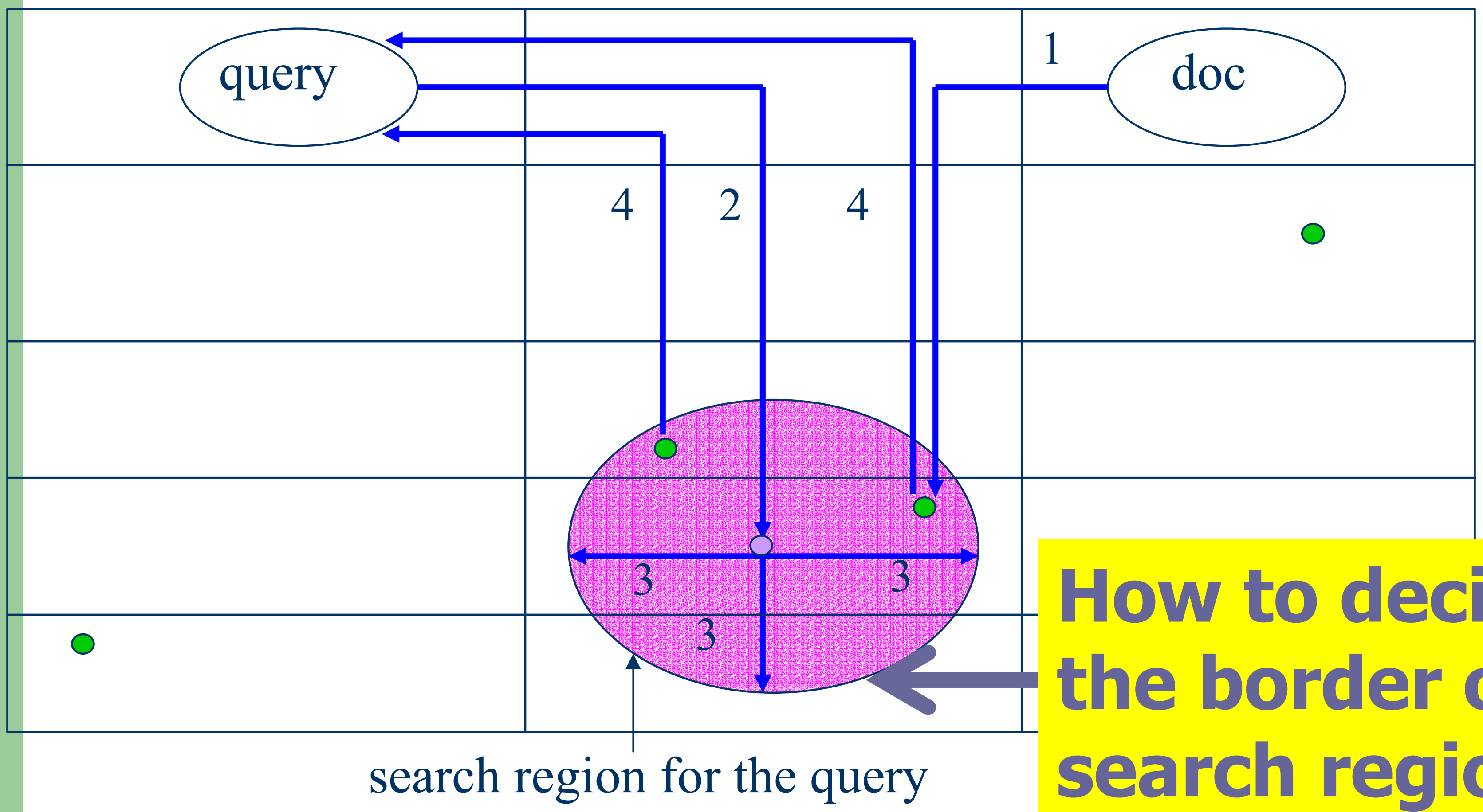
Outline

- Key idea in pSearch
- Background
  - Information Retrieval (IR)
  - Content-Addressable Network (CAN)
- P2P IR algorithm
- Experimental results
- Open issues and ongoing work
- Conclusions

pLSI Basic Idea

- Use a CAN to organize nodes into an overlay
- Use semantic vectors generated by LSI as object key to store doc indices in the CAN
  - Index locality: indices stored close in the overlay are also close in semantics
- Two types of operations
  - Publish document indices
  - Process queries

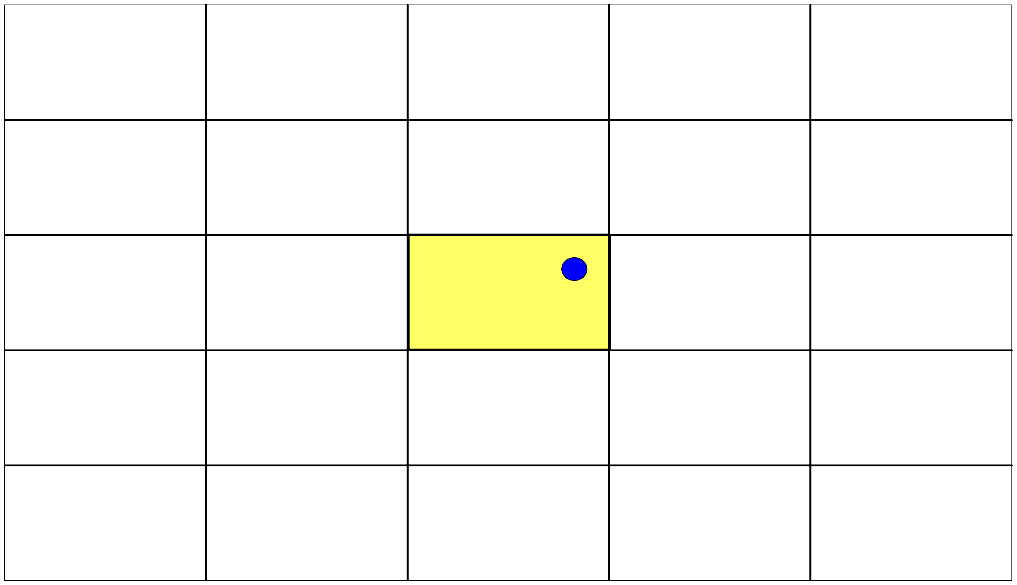
pLSI Illustration



How to decide the border of search region?

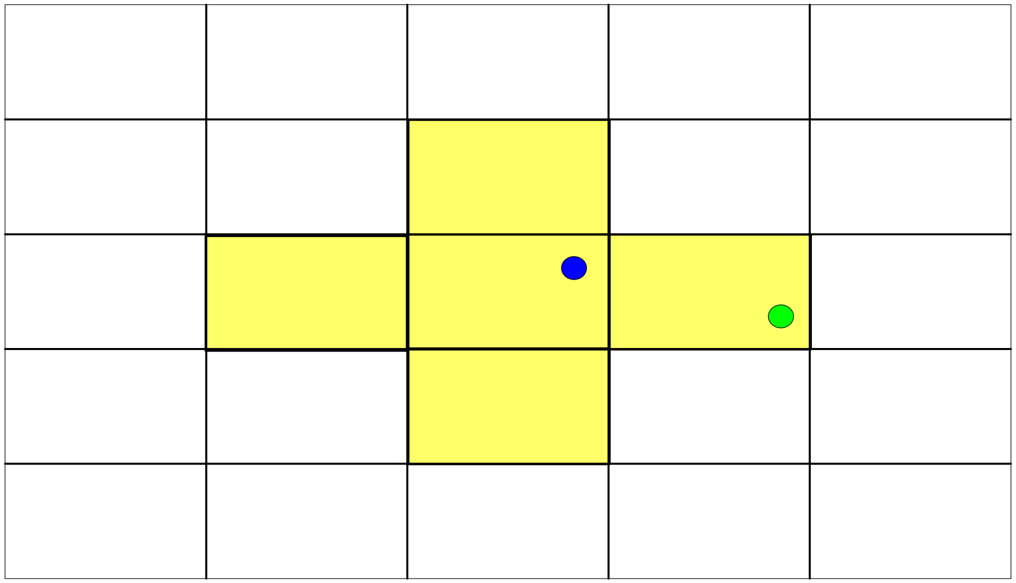
Content-directed Search

- Search the node whose zone contains the query semantic vector. (query center node)



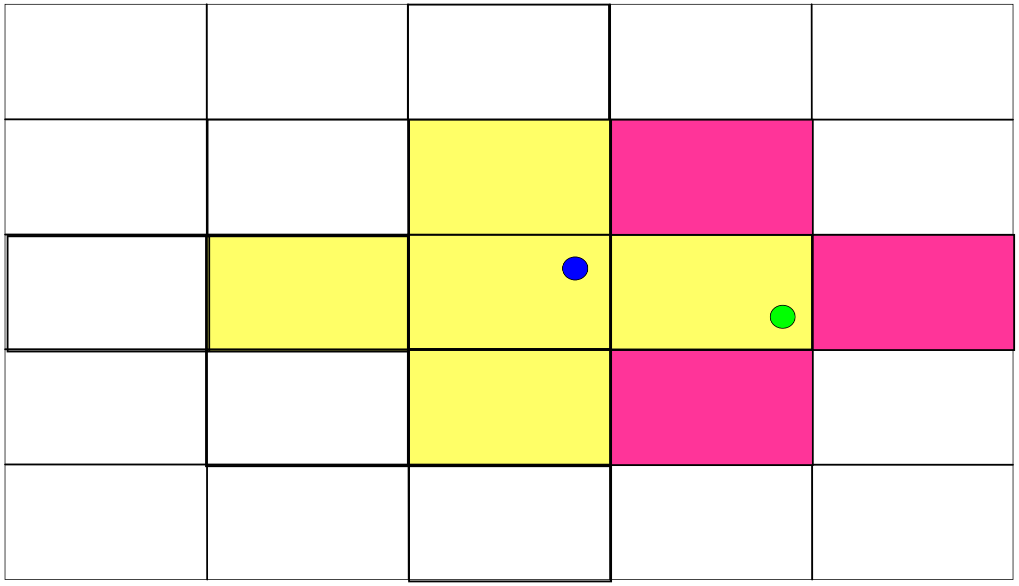
Content-directed Search

- Add direct (1-hop) neighbors of query center to pool of candidate nodes
- Search the most “promising” one in the pool suggested by samples



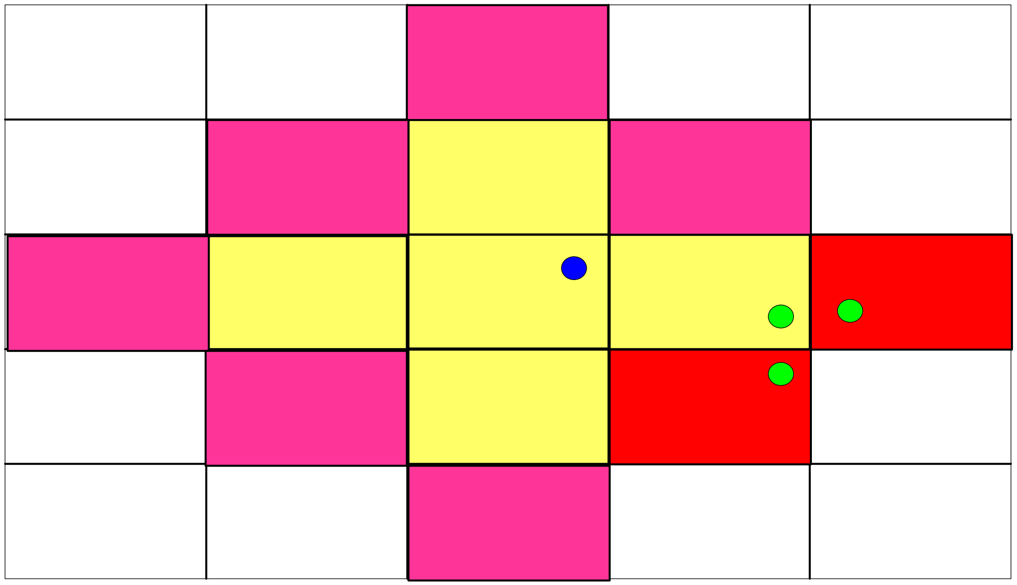
Content-directed Search

- Add its 1-hop neighbours to pool of candidate nodes



Content-directed Search

- Go on until it is unlikely to find better matching documents



pLSI Enhancements

- Further reduce nodes visited during a search
  - Content-directed search
  - Multi-plane (Rolling-index)
- Balance index distribution
  - Content-aware node bootstrapping

Multi-plane (rolling index)

- 4-d semantic vectors

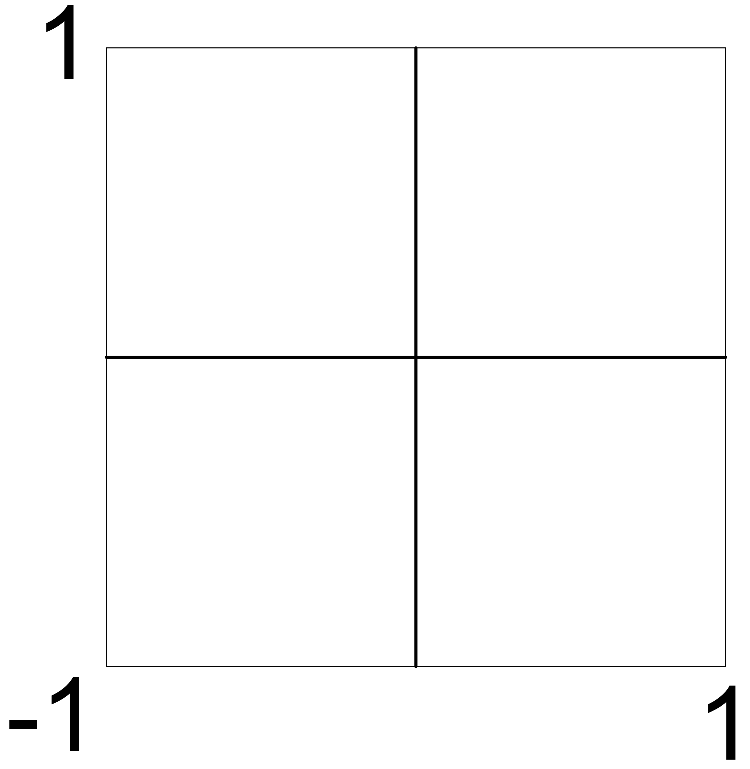
doc	query
-0.1	0.5
0.5	-0.1
0.5	0.6
-0.6	-0.5

Multi-plane (rolling index)

- 4-d semantic vectors
- 2-d CAN

doc

-0.1
0.5
0.5
-0.6

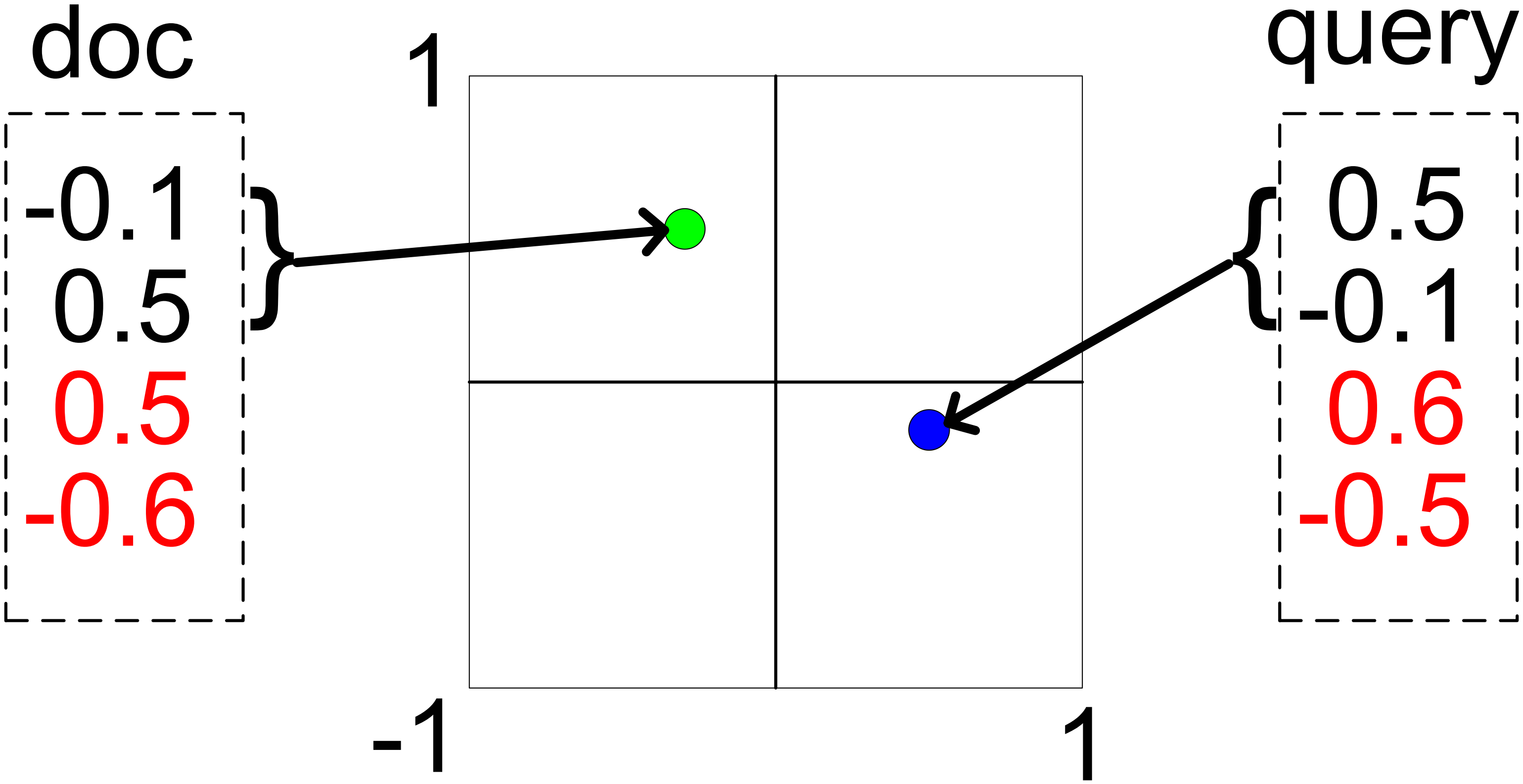


query

0.5
-0.1
0.6
-0.5

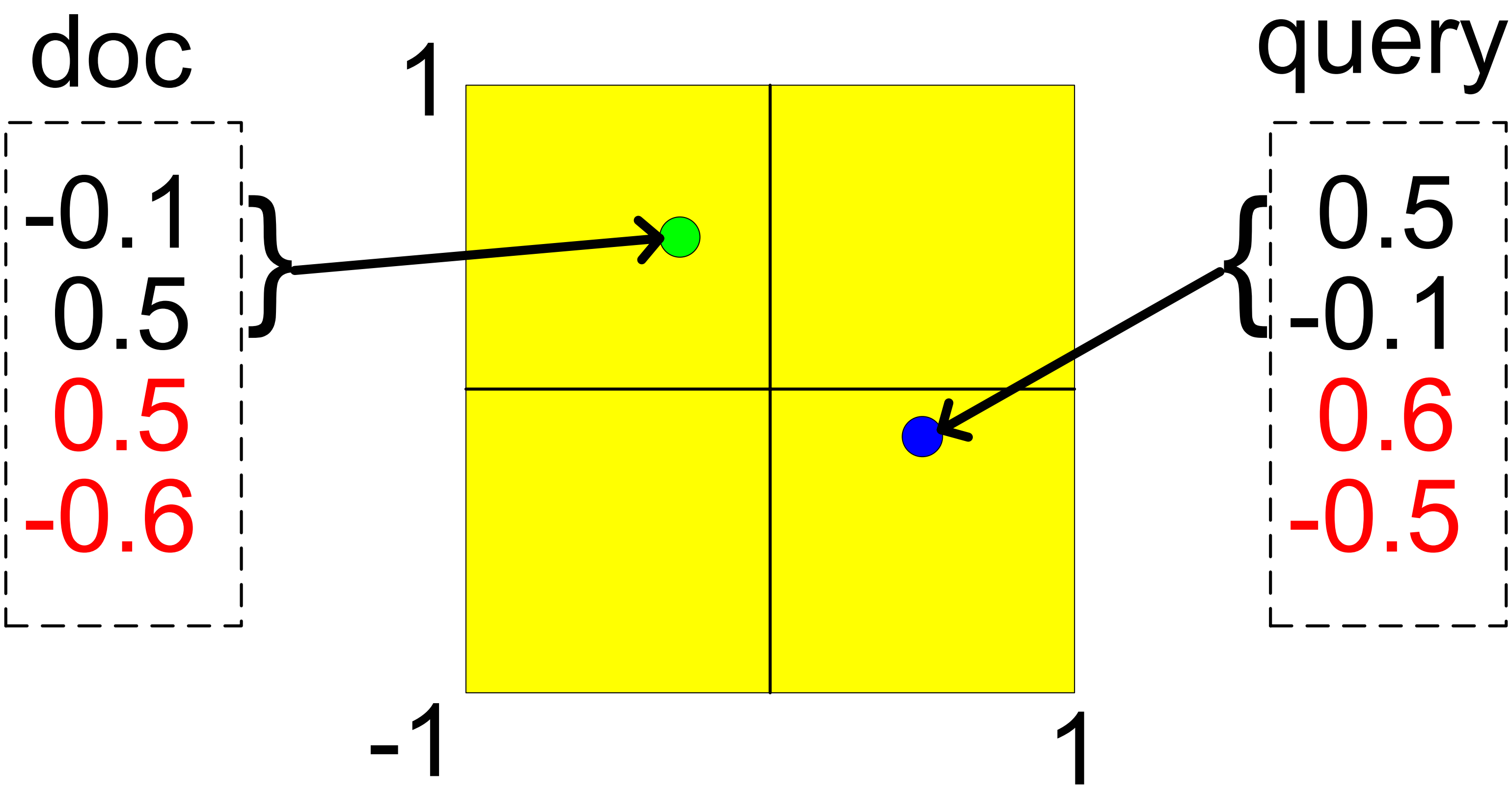
Multi-plane (rolling index)

- 4-d semantic vectors
- 2-d CAN



Multi-plane (rolling index)

- 4-d semantic vectors
- 2-d CAN



doc

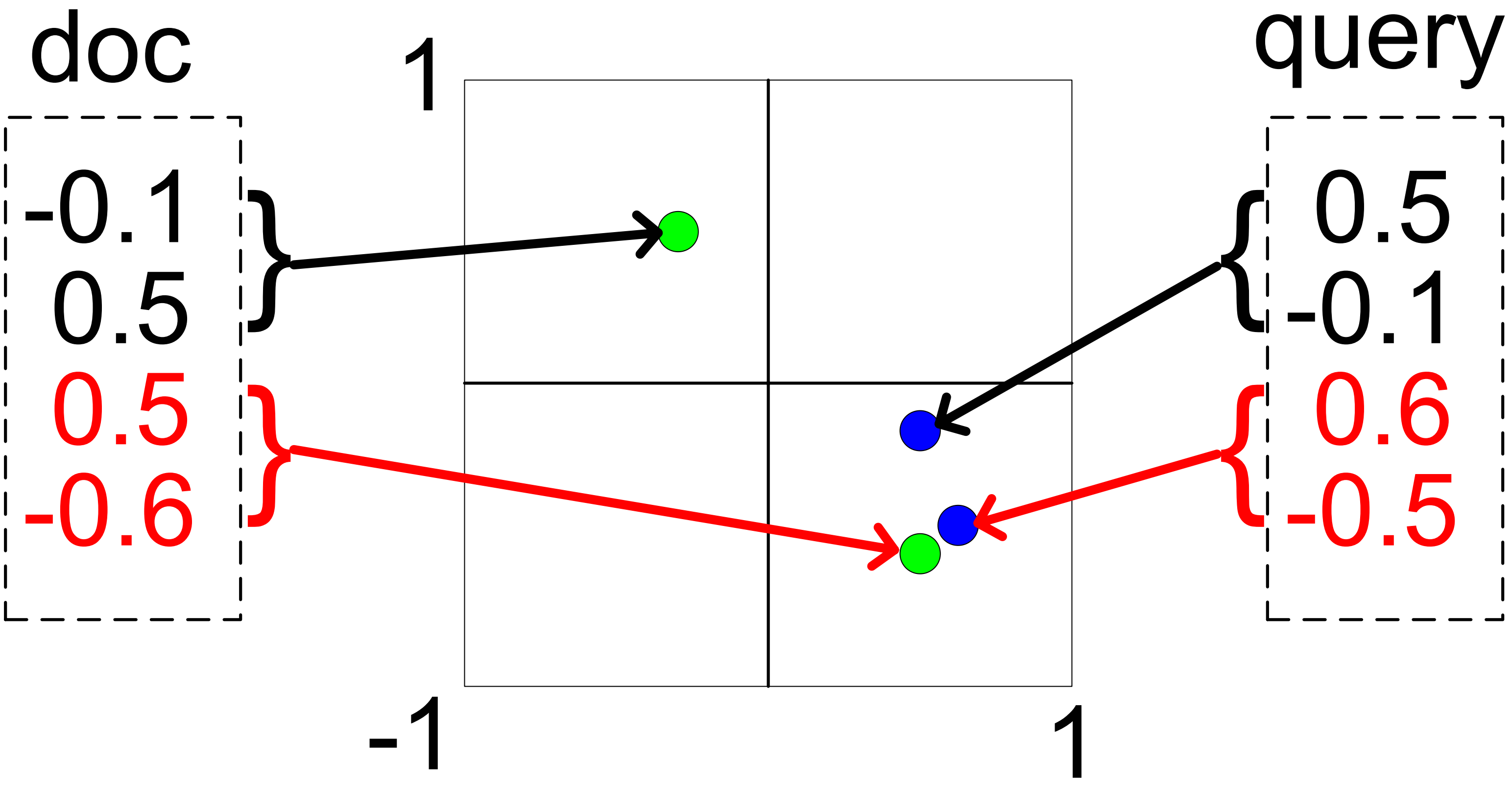
-0.1
0.5
0.5
-0.6

query

0.5
-0.1
0.6
-0.5

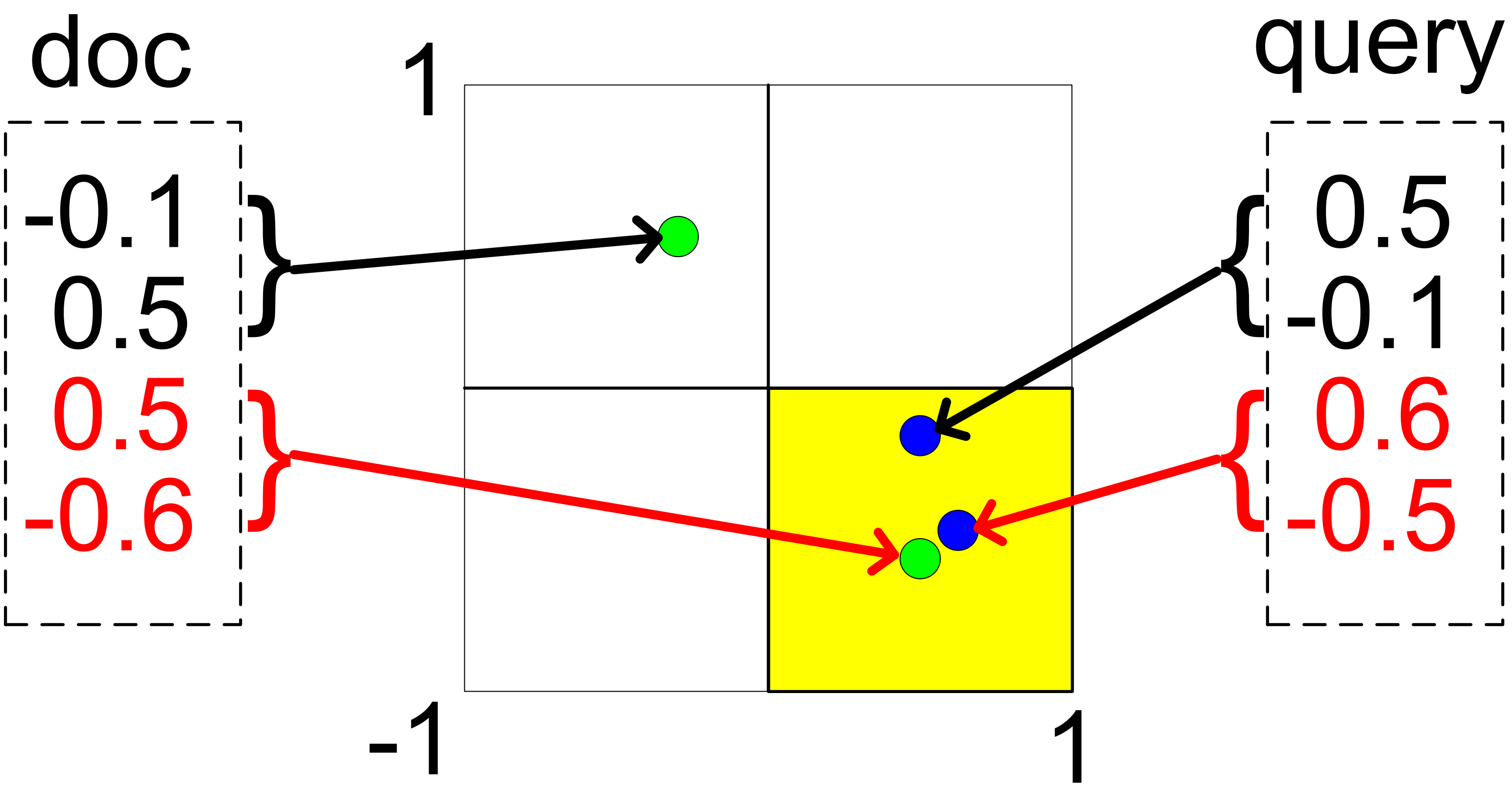
Multi-plane (rolling index)

- 4-d semantic vectors
- 2-d CAN



Multi-plane (rolling index)

- 4-d semantic vectors
- 2-d CAN

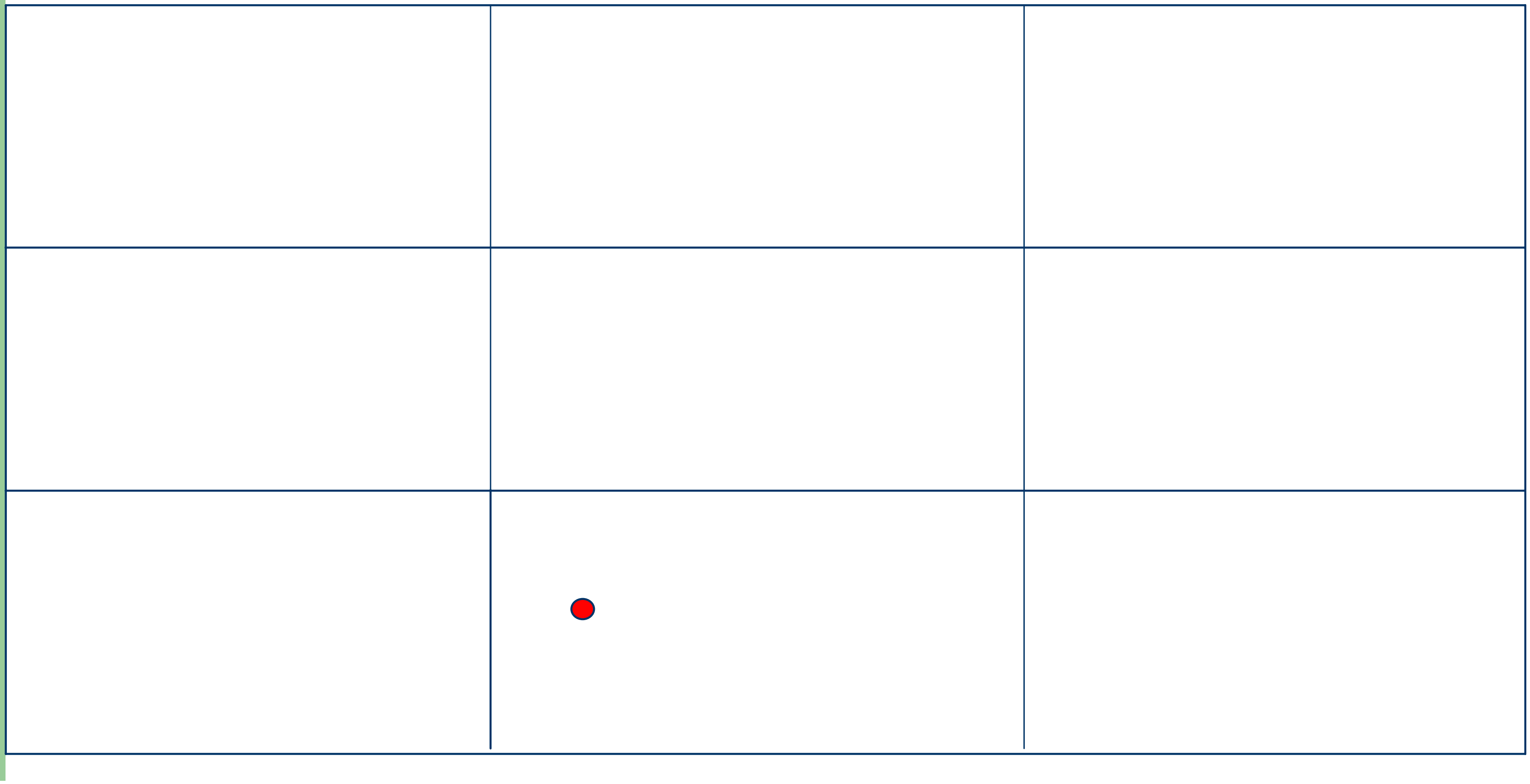


pLSI Enhancements

- Further reduce nodes visited during a search
  - Content-directed search
  - Multi-plane (Rolling-index)
- Balance index distribution
  - Content-aware node bootstrapping

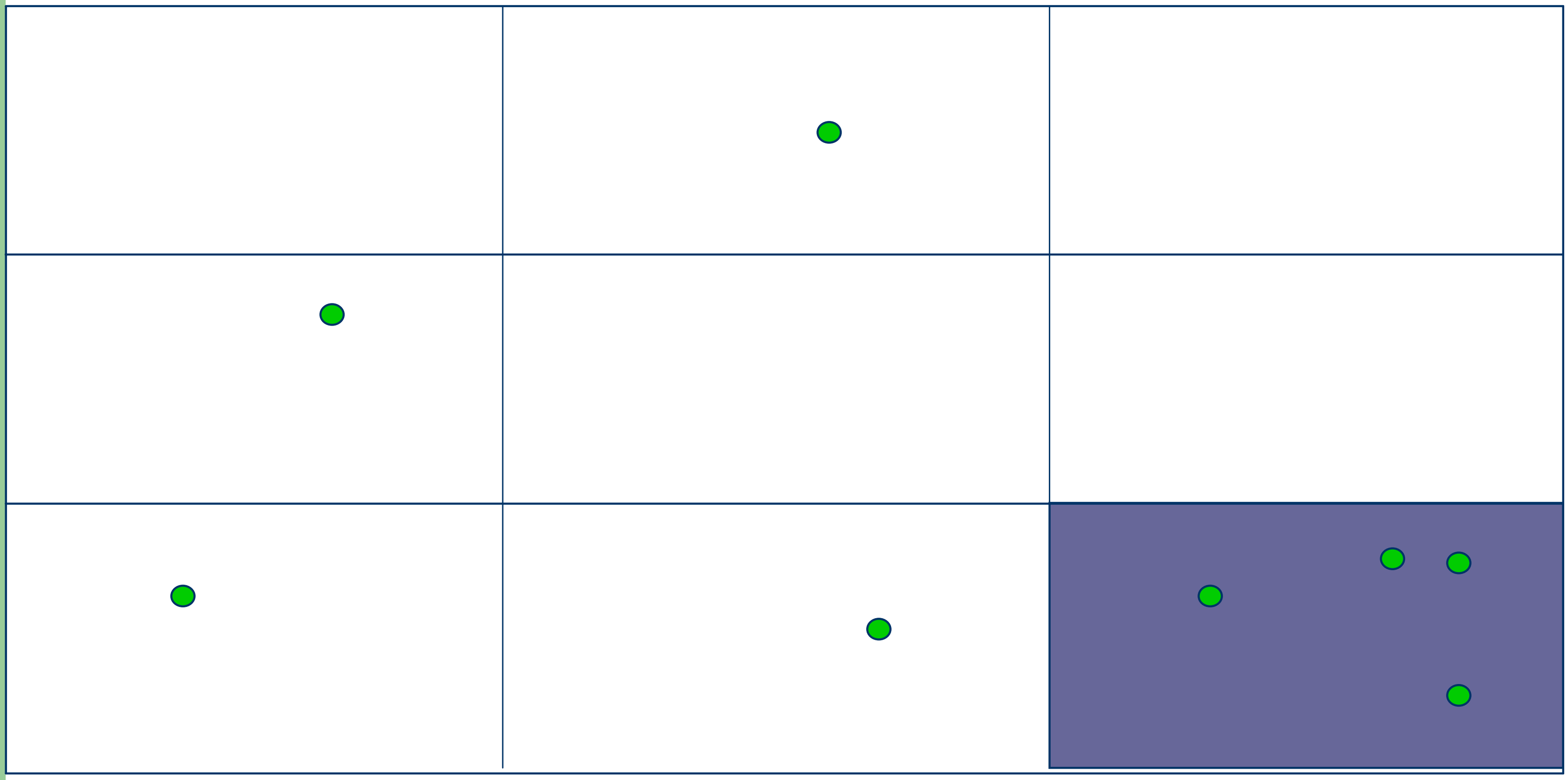
CAN Node Bootstrapping

- On node join, CAN picks a random point and splits the zone that contains the point



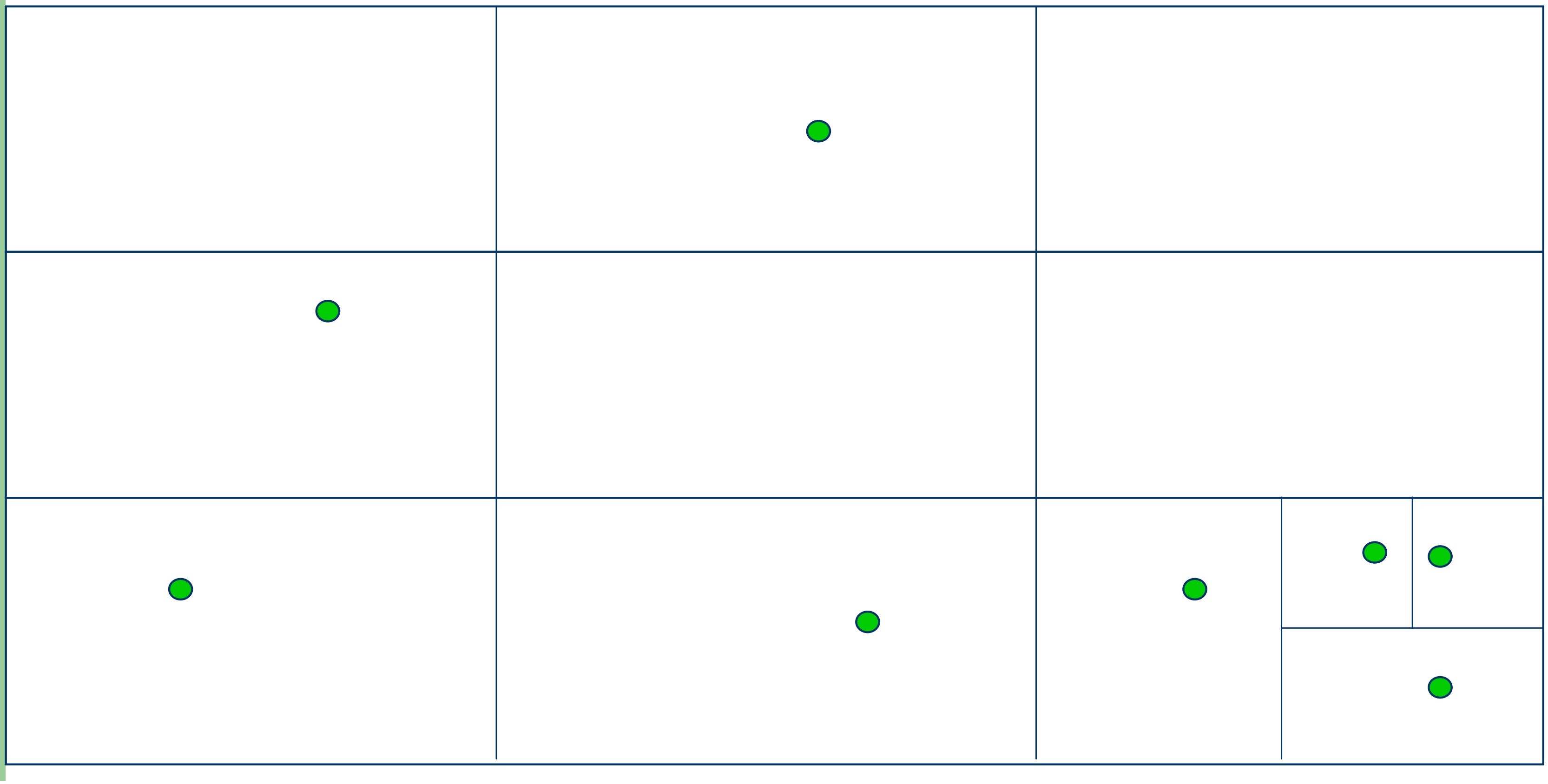
Unbalanced Index Distribution

- semantic vectors of documents



Content-Aware Node Bootstrapping

- pSearch randomly picks the semantic vector of an existing document for node bootstrapping



Experiment Setup

- pSearch Prototype
  - Cornell’s SMART system implements VSM
  - extend it with implementations of LSI, CAN, and pLSI algorithms
- Corpus: Text Retrieval Conference (TREC)
  - 528,543 documents from various sources
  - total size about 2GB
  - 100 queries, topic 351-450

Evaluation Metrics

- Efficiency: nodes visited and data transmitted during a search
- Efficacy: compare search results
  - pLSI vs. LSI

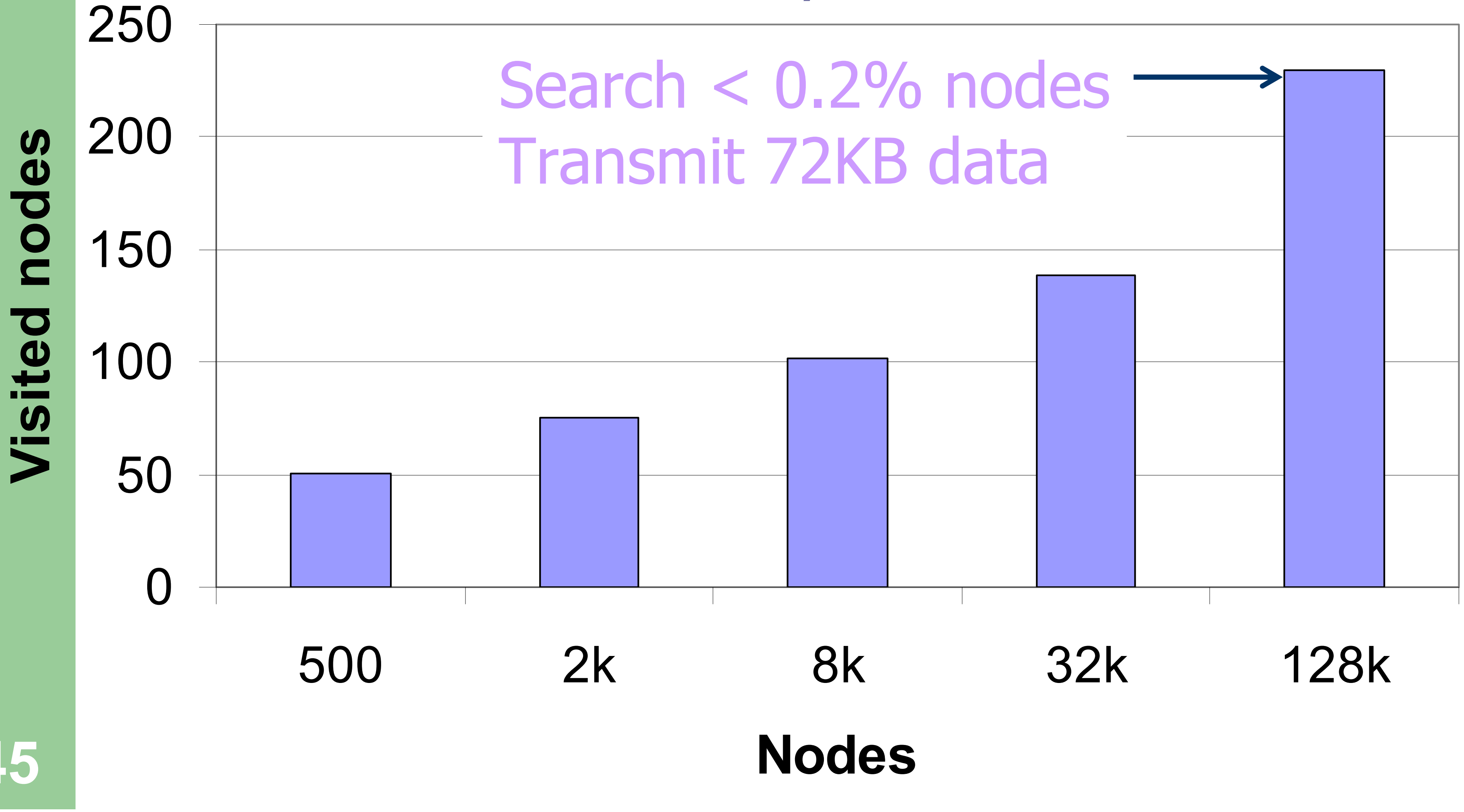
pLSI vs. LSI

$$\text{Accuracy} = \frac{|A \cap B|}{|A|} \times 100\%$$

- Retrieve top 15 documents
- A: documents retrieved by LSI
- B: documents retrieved by pLSI

Performance w.r.t. System Size

Accuracy = 90%



Search < 0.2% nodes
Transmit 72KB data

Open Issues

- Larger corpora
- Efficient variants of LSI/SVD
- Evolution of global statistics
- Incorporate other IR techniques
  - Relevance feedback, Google’s PageRank

Querying the Internet with PIER

(PIER = Peer-to-peer Information Exchange and Retrieval)

Presented by Zheng Ma

Yale University

Outline

- Inroduction
- What is PIER?
  - Design Principles
- Implementation
  - DHT
  - Query Processor
- Performance
- Summary

Introduction

- Databases:
  - powerful query facilities
  - declarative interface
  - potential to scale up to few hundred computers
- What about Internet? If we want well distributed system that has
  - query facilities (SQL)
  - fault tolerance
  - flexibility
- PIER is a query engine that scales up to thousands of participating nodes and can work on various data

What is PIER?

- Peer-to-Peer Information Exchange and Retrieval
- Query engine that runs on top of P2P network
  - step to the distributed query processing at a larger scale
  - way for massive distribution: querying heterogeneous data
- Architecture meets traditional database query processing with recent peer-to-peer technologies

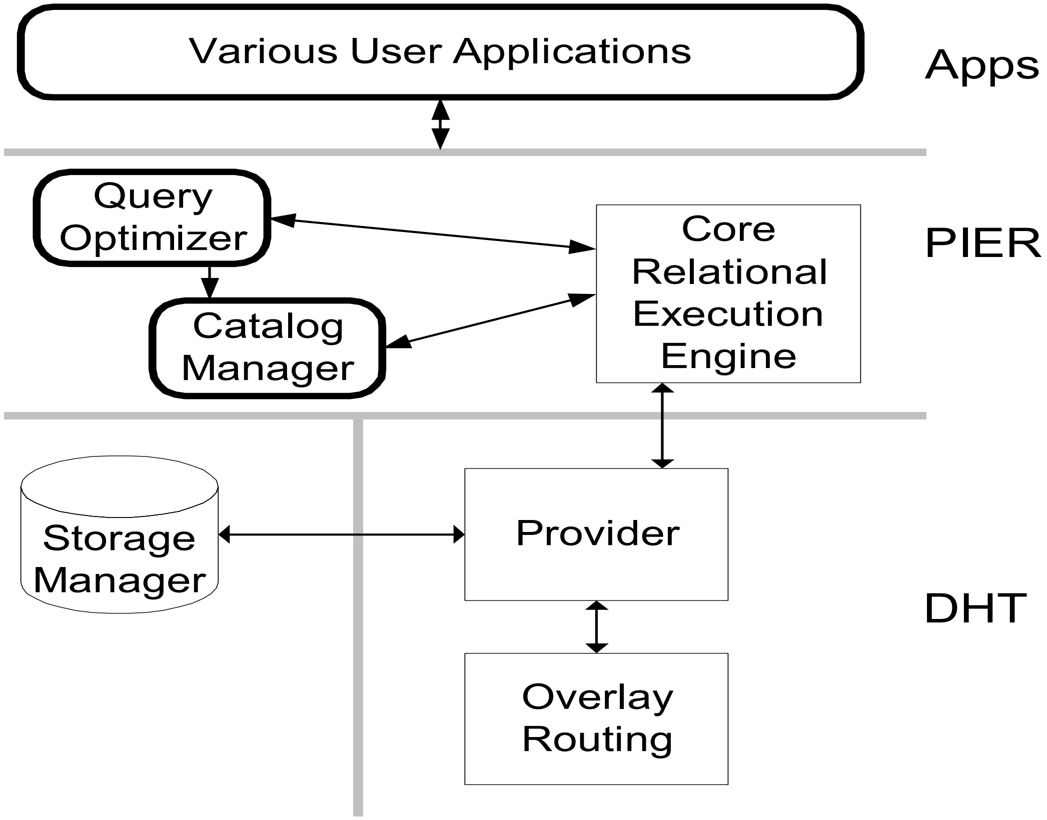
Design Principles

- Relaxed Consistency
  - adjusts availability of the system
- Organic Scaling
  - No need in *a priori* allocation of a data center
- Natural Habitats for Data
  - No DB schemas, file system or perhaps a live feed
- Standard Schemas via Grassroots Software
  - widespread programs provide *de facto* standards.

Outline

- Introduction
- What is PIER?
  - Design Principles
- Implementation
  - DHT
  - Query Engine
- Scalability
- Summary

Implementation – DHT



<< based on CAN

DHT structure:

- routing layer
- storage manager
- provider

DHT – Routing & Storage

Routing layer

maps a **key** into the **IP address** of the node currently responsible for that key. Provides exact lookups, callbacks higher levels when the set of keys has changed

Routing layer API

```
lookup(key) → ipaddr
join(landmarkNode)
leave()
locationMapChange
```

Storage Manager

stores and **retrieves** records, which consist of key/value pairs. Keys are used to locate items and can be any data type or structure supported

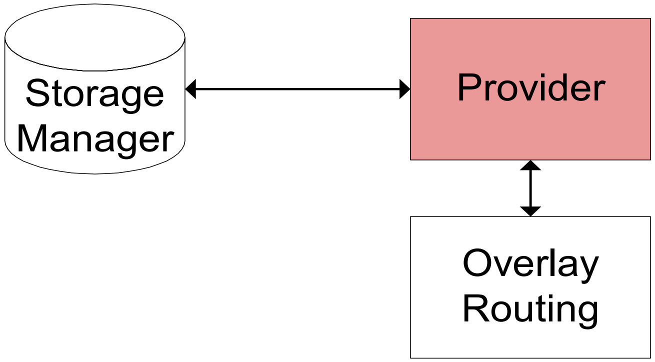
Storage Manager API

```
store(key, item)
retrieve(key)→ item
remove(key)
```

DHT – Provider (1)

Provider ties routing and storage manager layers and **provides** an interface

- Each object in the DHT has a *namespace*, *resourceID* and *instanceID*
- **DHT key = hash(*namespace*,*resourceID*)**



- *namespace* - application or group of object, table
- *resourceID* – *what* is object, primary key or any attribute
- *instanceID* – integer, to separate items with the same *namespace* and *resourceID*
- CAN's mapping of *resourceID*/Object is equivalent to an index

DHT – Provider (2)

Provider API

```
get(namespace, resourceID) → item
put(namespace, resourceID, item, lifetime)
renew(namespace, resourceID, instanceID, lifetime)
   → bool
multicast(namespace, resourceID, item)
lscan(namespace) → items
newData(namespace, item)
```

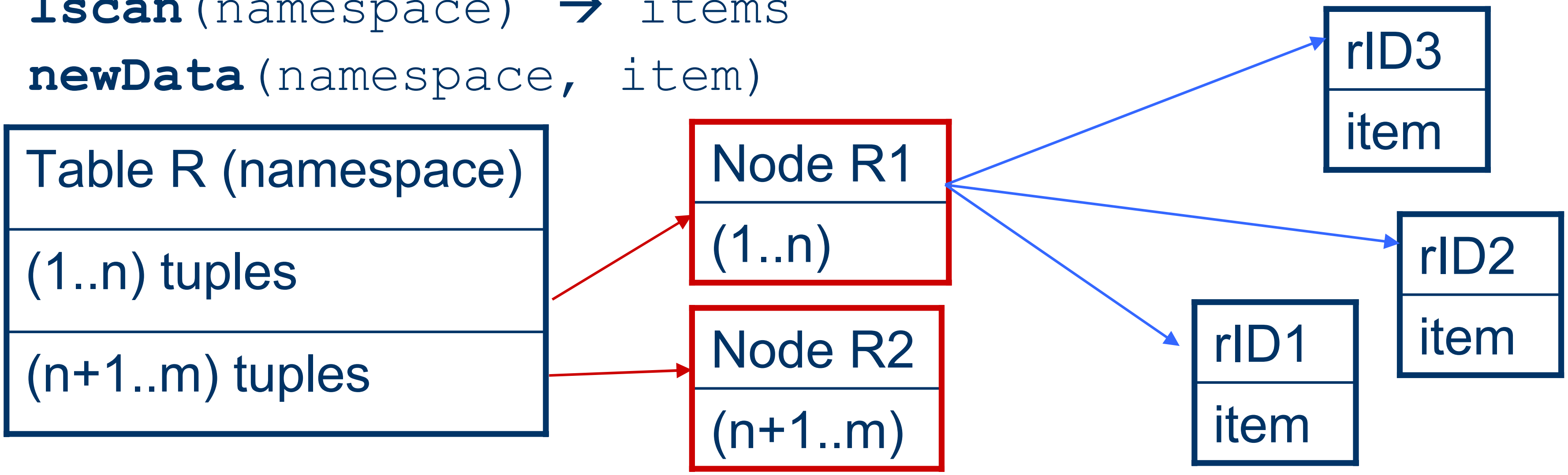


Table R (namespace)

(1..n) tuples

(n+1..m) tuples

Node R1

(1..n)

Node R2

(n+1..m)

rID3

item

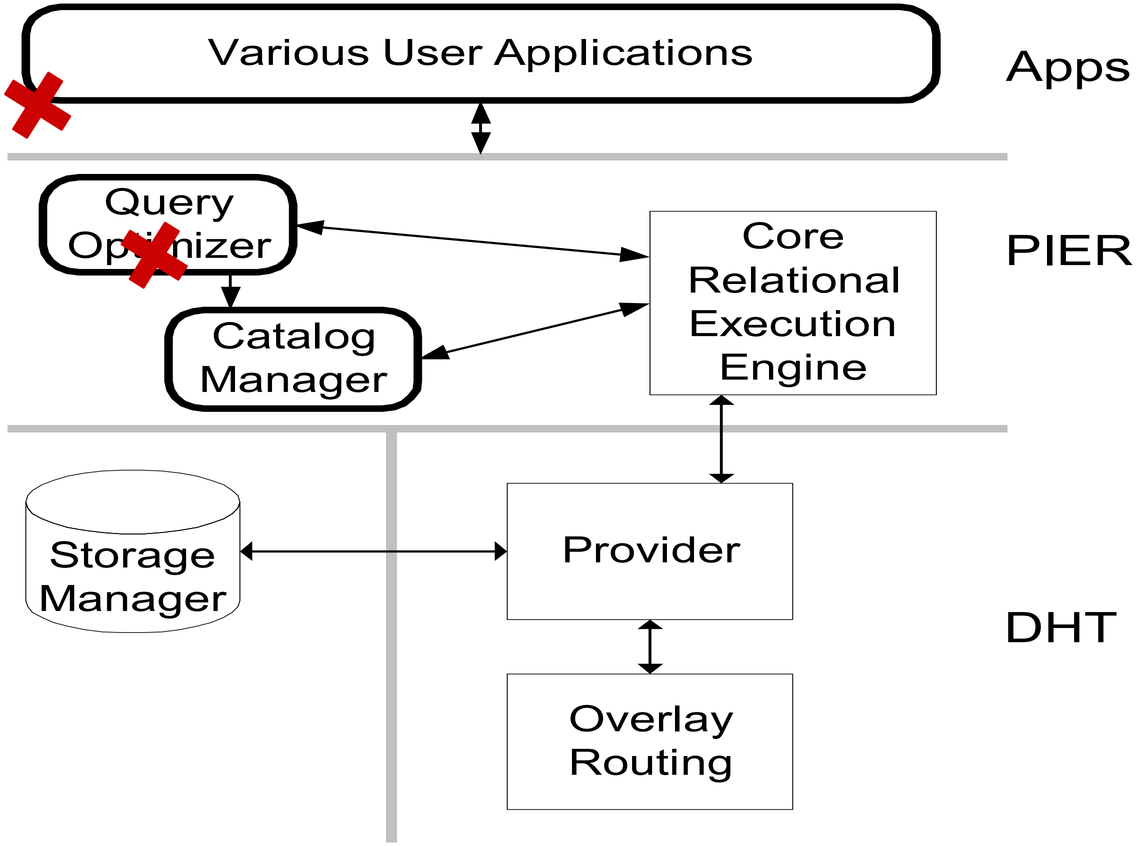
rID2

item

rID1

item

Implementation – Query Engine



<< query processor

QP Structure:

- core engine
- query optimizer
- catalog manager

Query Processor

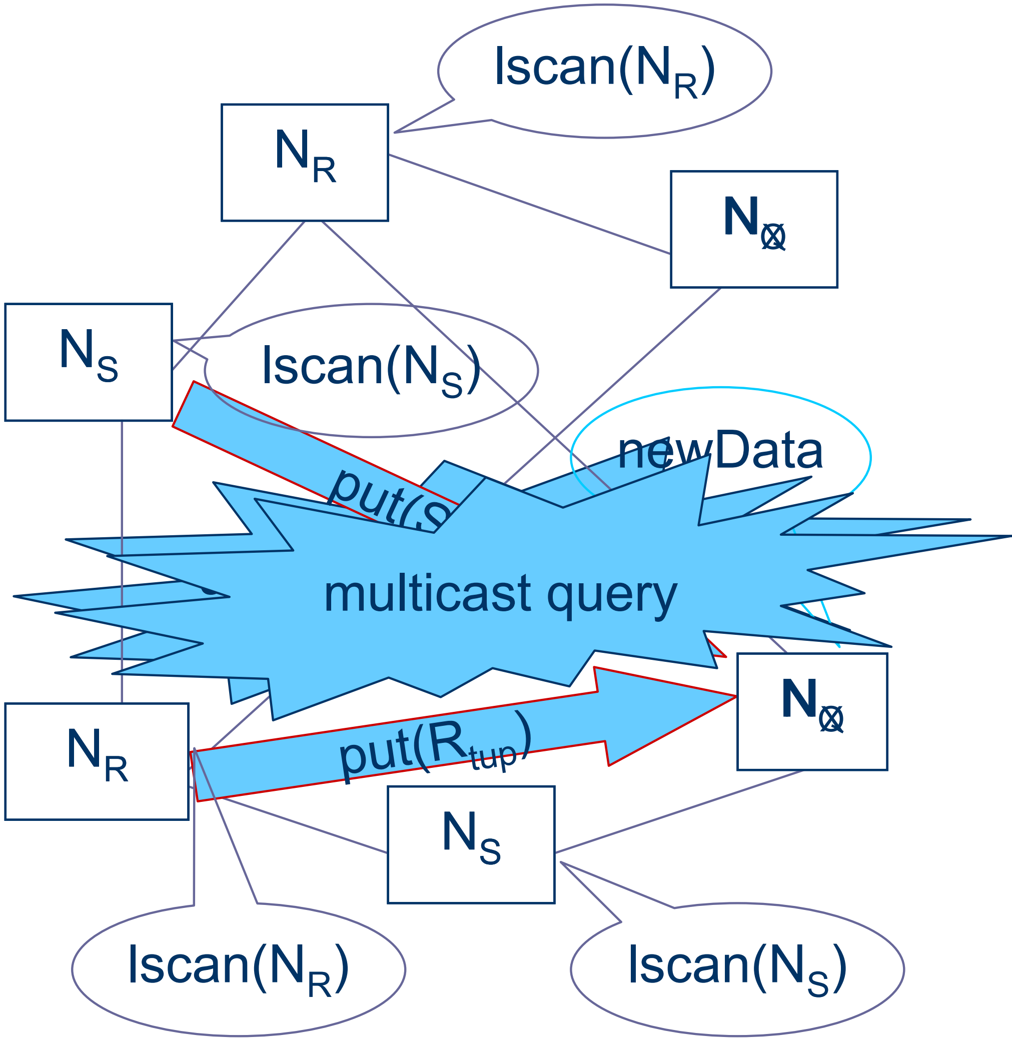
- How it works?
  - performs selection, projection, joins, grouping, aggregation
  - simultaneous execution of multiple operators pipelined together
  - results are produced and queued as quick as possible
- How it modifies data?
  - insert, update and delete different items via DHT interface
- How it selects data to process?
  - *dilated-reachable snapshot* – data, published by reachable nodes at the query arrival time

Query Processor – Joins (1)

Symmetric hash join

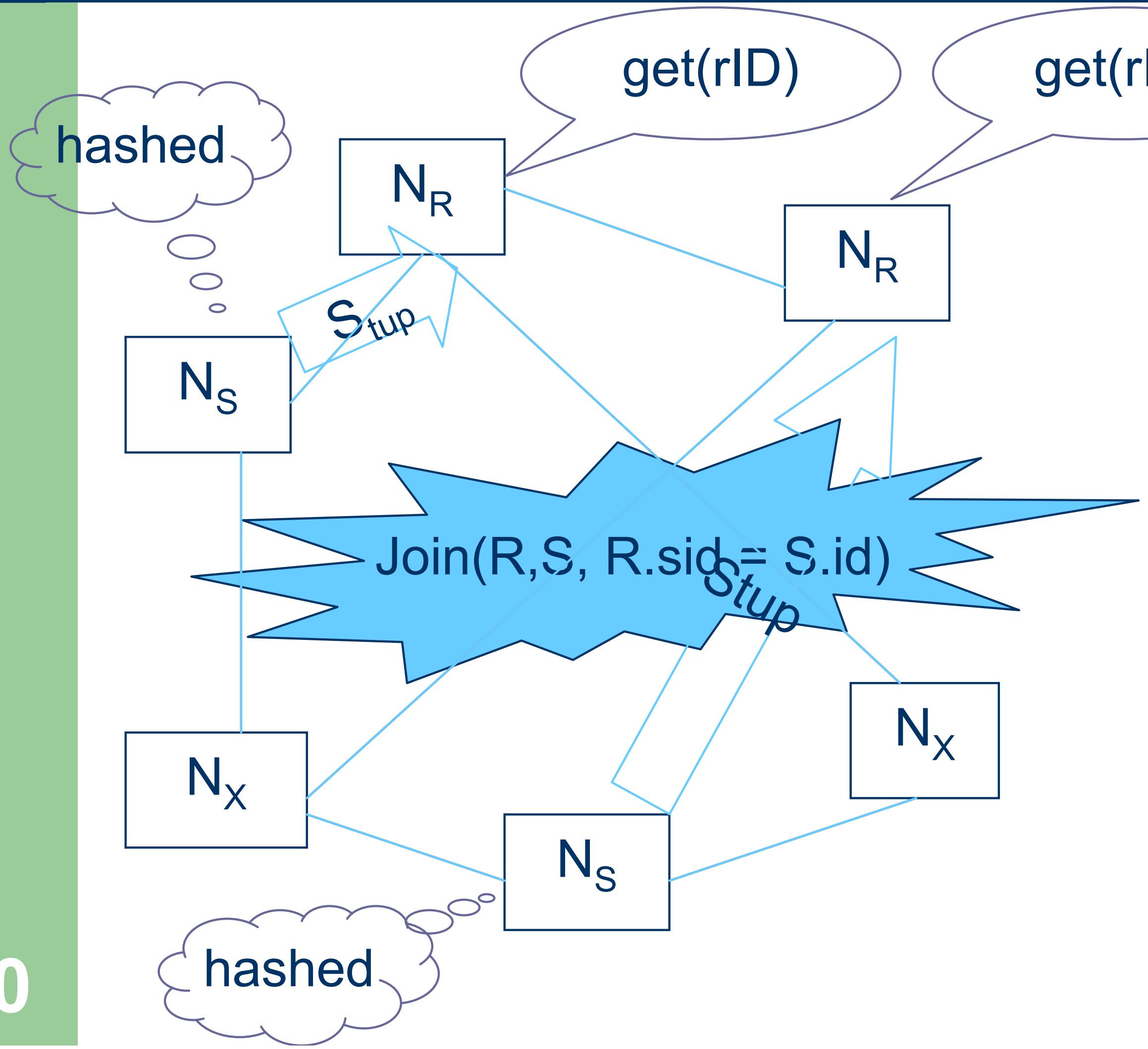
At each site

- (Scan) `lscan` N_R and N_S
- (Rehash) `put` into N_Q a copy of each eligible tuple
- (Listen) use `newData` to see the rehashed tuples in N_Q
- (Compute) join the tuples as they arrive to N_Q



*Basic, uses a lot of network resources

Query Processor – Joins (2)



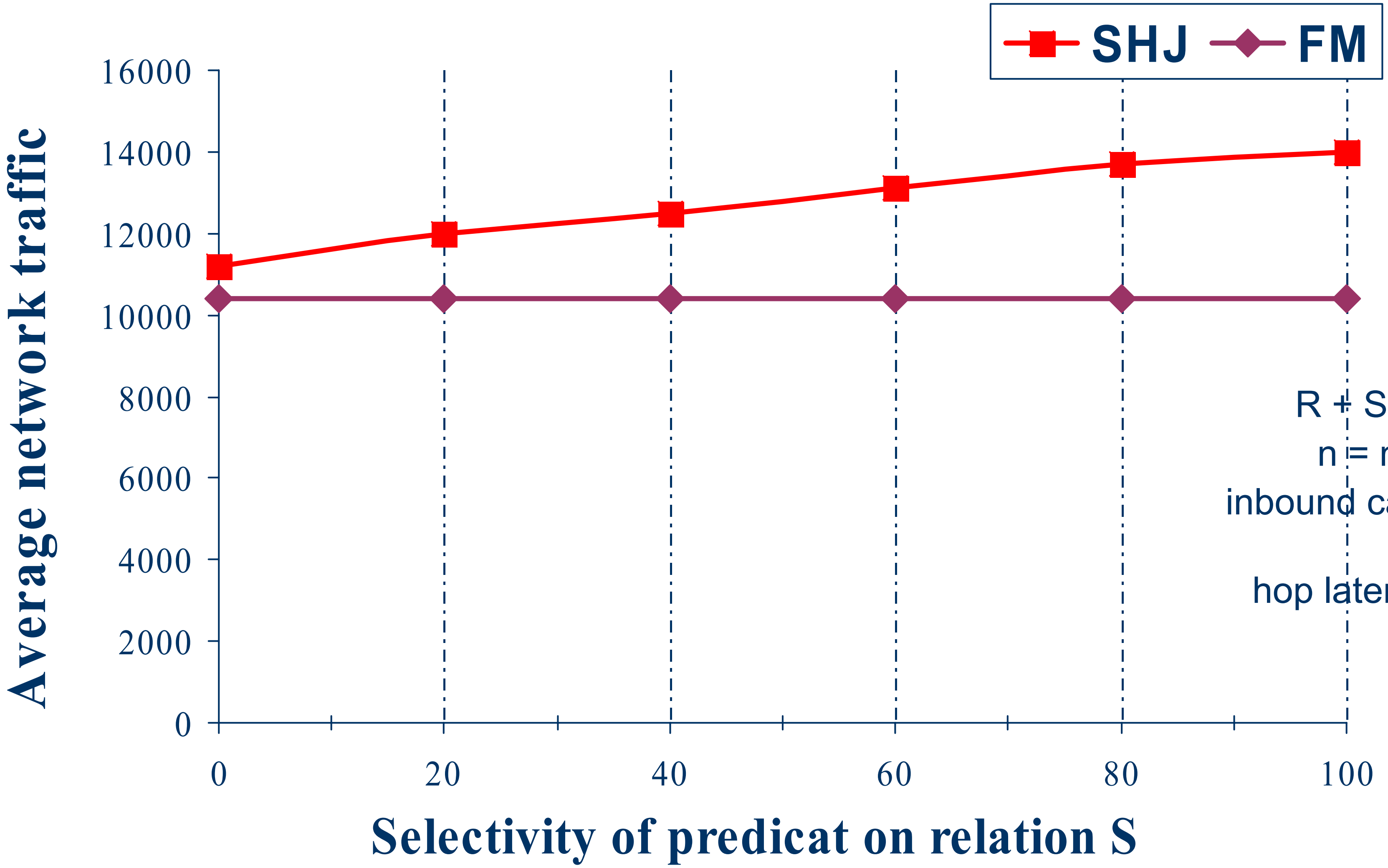
Fetch matches

At each site

- (Scan) `lscan(`N_R`)`
- (Get) for each suitable *R* tuple `get` for the matching *S* tuple
- When S tuples arrive at R, join them
- Pass results

*Retrieve only tuples that matched

Performance: Join Algorithms



R + S = 25 GB

n = m = 1024

inbound capacity = 10 Mbps

hop latency =100 ms

Query Processor – Join rewriting

Symmetric semi-join

- (Project) both *R* and *S* to their *resourceIDs* and join keys
- (Small rehash) Perform a **SHJ** on the two projections
- (Compute) Send results into **FM** join for each of the tables

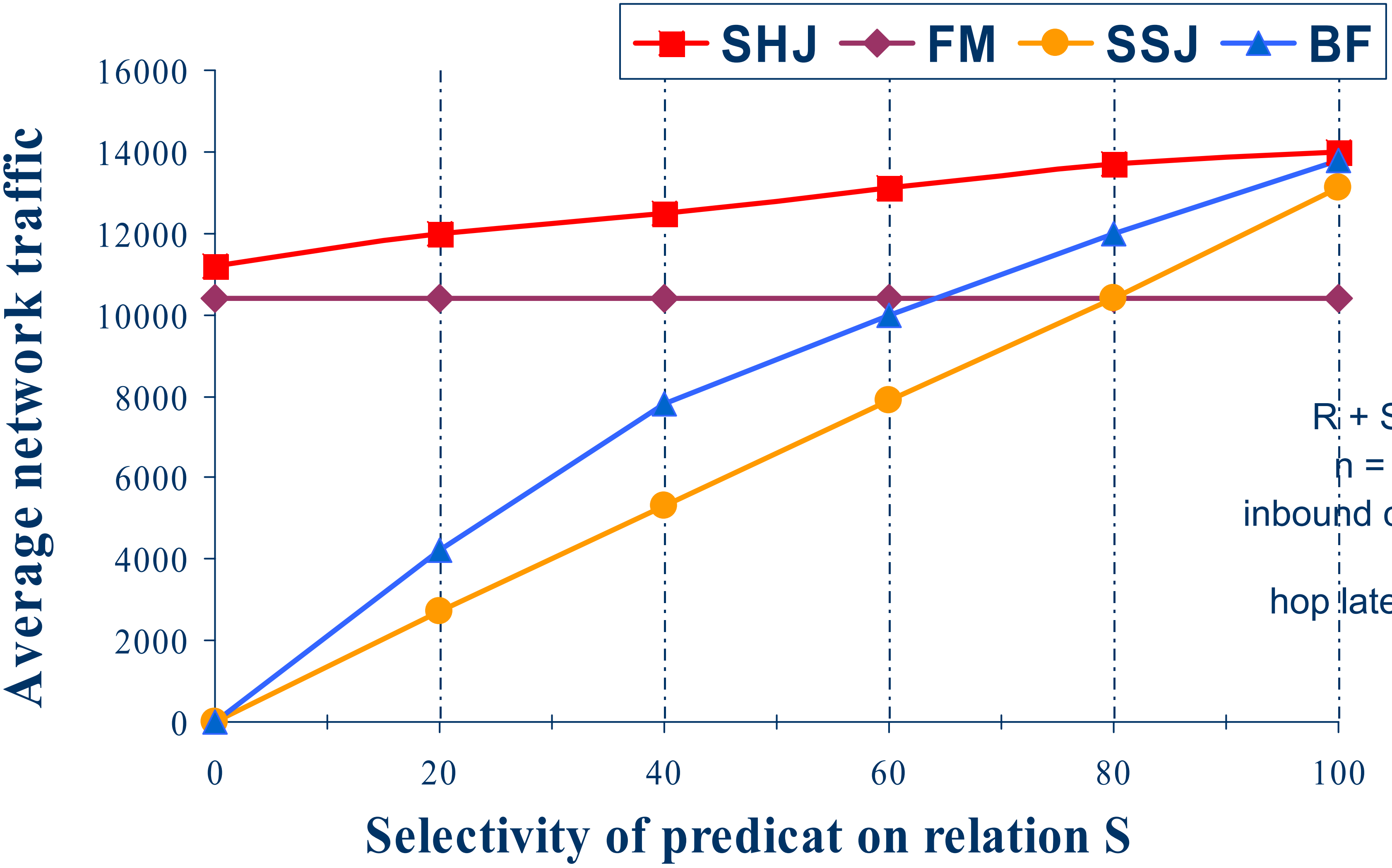
*Minimizes initial communication

Bloom joins

- (Scan) create Bloom Filter for a fragment of relation
- (Put) Publish filter for *R, S*
- (Multicast) Distribute filters
- (Rehash) only tuples matched the filter
- (Compute) Run **SHJ**

*Reduces rehashing

Performance: Join Algorithms



R + S = 25 GB

n = m = 1024

inbound capacity = 10 Mbps

hop latency =100 ms

Outline

- Introduction
- What is PIER?
  - Design Principles
- Implementation
  - DHT
  - Query Processor
- Scalability
- Summary

Scalability Simulation

Conditions

- |R| =10 |S|
- Constants produce selectivity of 50%

Query:

SELECT

R.key, S.key, R.pad

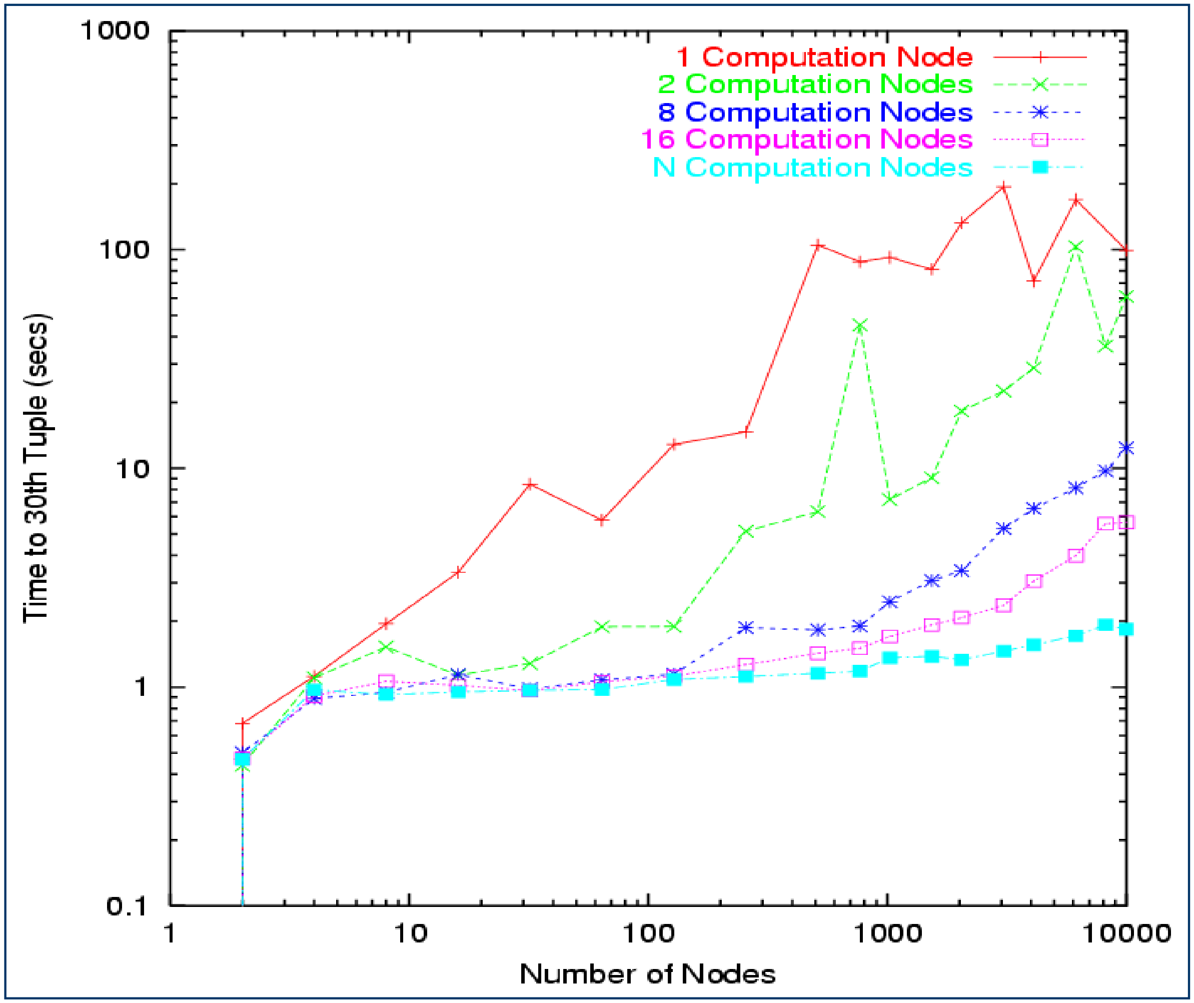
FROM R,S

WHERE R.n1 = S.key

AND R.n2 > const1

AND S.n2 > const2

AND f(R.n3,S.n3) > const3



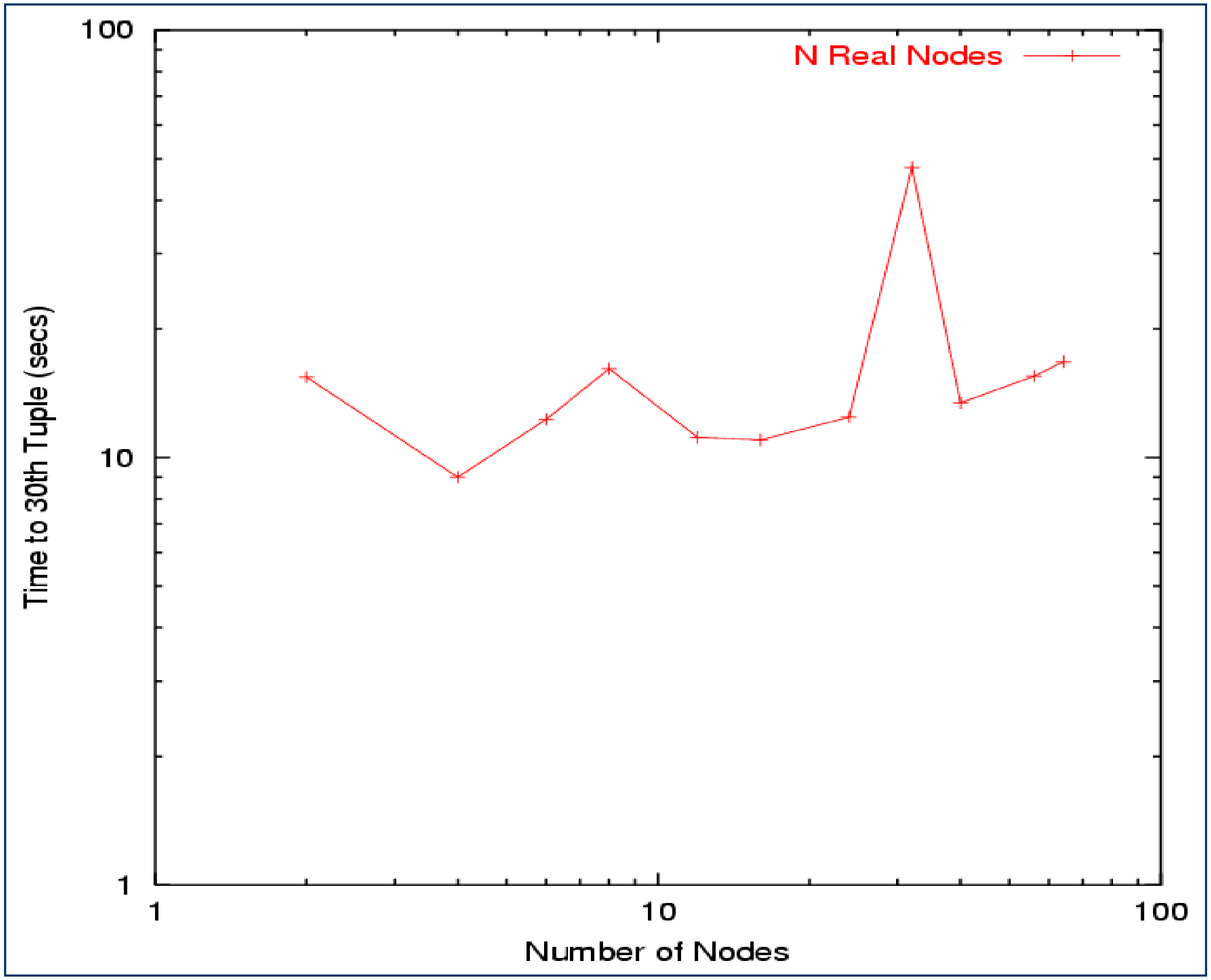
Experimental Results

Equipment:

- cluster of 64 PCs
- 1 Gbps network

Result:

- Time to receive 30-th result tuple practically remains unchanged as both the size and load are scaled up.



Summary

- PIER is a structured query system intended to run at the big scale
- PIER queries data that preexists in the wide area
- DHT is a core scalability mechanism for indexing, routing and query state management
- Big front of future work:
  - Caching
  - Query optimization
  - Security
  - …

Backup Slides

HitList

“Hitlist” is defined as list of occurrences of a particular word in a particular document including additional meta info:

- position of word in doc

- font size

- capitalization

- descriptor type, e.g. title, anchor, etc.

Inverted Index

- Contains the same barrels as the Forward Index, except they have been sorted by docID’s
- All words are pointed to by the Lexicon
- Contains pointers to a doclist containing all docID’s with their corresponding hit lists.
  - The barrels are duplicated
  - For speed in single word searches

Specific Design Goals

- Deliver results that have very high precision even at the expense of recall
- Make search engine technology transparent, i.e. advertising shouldn’t bias results
- Bring search engine technology into academic realm in order to support novel research activities on large web data sets
- Make system easy to use for most people, e.g. users shouldn’t have to specify more than a couple words

Crawling the Web

- Distributed Crawling system:
  - UrlServer
  - Multiple crawlers
- Issues:
  - DNS bottleneck requires cached DNS
  - Extremely complex and heterogeneous Web
- Systems must be very robust

Why do we need d?

- In the real world virtually all web graphs are not connected, i.e. they have dead-ends, islands, etc.
- If we don’t have d we get “ranks leaks”
  for graphs that are not connected, i.e. leads to numerical instability

Indexing the Web

- Parsing so many different documents is very difficult
- Indexing documents requires simultaneous access to the Lexicon
  - Creates a problem for words that aren’t already in the Lexicon
- Sorting is done on multiple machines, each working on a different barrel

Document Index

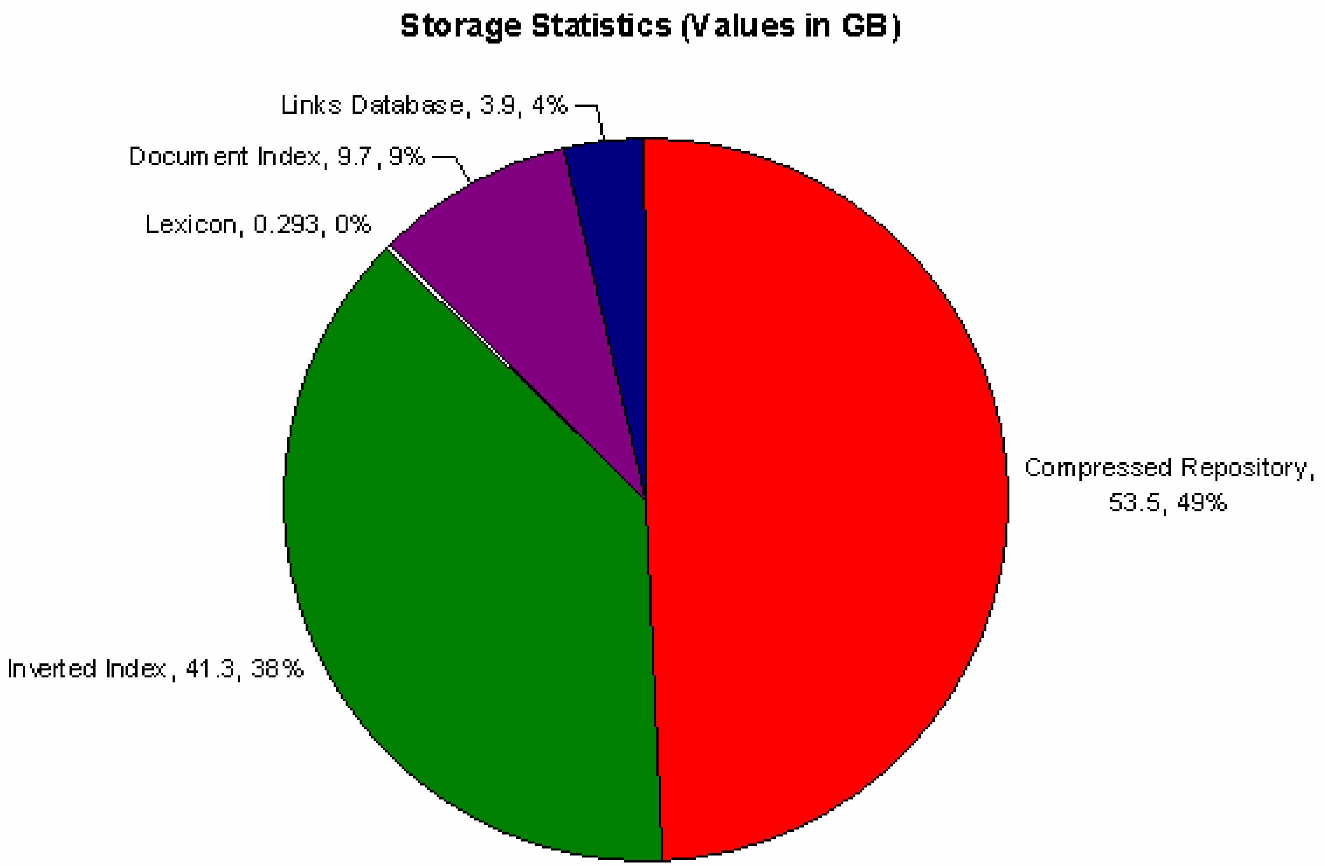
- Keeps information about each document
- Sequentially stored, ordered by DocID
- Contains:
  - Current document status
  - Pointer into the repository
  - Document checksum
  - File for converting URLs to DocID's
- If the page has been crawled, it contains:
  - A pointer to DocInfo -> URL and title
- If the page has not been crawled, it contains:
  - A pointer to the URLList -> Just the URL
- This data structure requires only 1 disk seek for each search

Lexicon

- The lexicon is stored in memory and contains:
  - A null separated word list
  - A hash table of pointers to these words in the barrels (for the Inverted Index)
- An important feature of the Lexicon is that it fits entirely into memory

Storage Requirements

At the time of publication, Google had the following statistical breakdown for storage requirements:



Single Word Query Ranking

- Hitlist is retrieved for single word
- Each hit can be one of several types: title, anchor, URL, large font, small font, etc.
- Each hit type is assigned its own weight
- Type-weights make up vector of weights
- # of hits of each type is counted to form count vector
- Dot product of two vectors is used to compute IR score
- IR score is combined with PageRank to compute final rank

Multi-word Query Ranking

- Similar to single-word ranking except now must analyze proximity
- Hits occurring closer together are weighted higher
- Each proximity relation is classified into 1 of 10 values ranging from a phrase match to “not even close”
- Counts are computed for every type of hit and proximity

Forward Index

- Stored in barrels containing:
  - A range of WordID's
  - The DocID of a pages containing these words
  - A list of WordID's followed by corresponding hit lists
- Actual WordID's are not stored in the barrels; instead, the difference between the word and the minimum of the barrel is stored
  - This requires only 24 bits for each WordID
  - Allowing 8 bits to hold the hit list length

References

1. Sergey Brin and Lawrence Page. “The Anatomy of a Large-Scale Hypertextual Web Search Engine”. WWW7 / Computer Networks 30(1-7): 107-117 (1998)
2. http://searchenginewatch.com/
3. http://www.searchengineshowdown.com/
4. http://www.robotstxt.org/wc/exclusion.html