Integrating Web and Database Searches

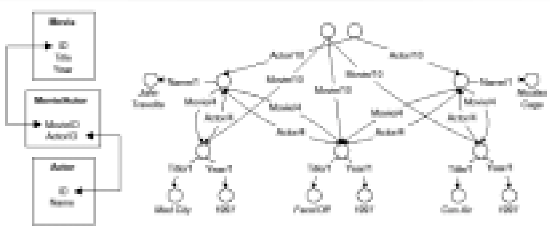
CPS 296.1

Topics in Database Systems

Roadmap

- Rank aggregation: merging ranked results from different searches
  - Fagin et al. “Optimal Aggregation Algorithm for Middleware.” *PODS*, 2001
- ➢ Proximity search: finding all shortest paths in the link structure of a database
  - Goldman et al. “Proximity Search in Databases.” *VLDB*, 1998
- ➢ WSQ: enhancing database queries with Web searches
  - Goldman and Widom. “WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web.” *SIGMOD*, 2000

Link structure of a database



- Table → records in the table; record → fields in the record; foreign key references; etc.
- Links have weights (≥1)
  - Smaller weight means closer relationship

Proximity search

Returns a “find set” *F*

Returns a “near set” *N*

Returns objects in *F*, ranked by proximity to objects in *N*

Proximity versus text distance

- Text distance works on one-dimensional text
- Proximity works on data with tree or graph structure, e.g., link structure of a database, XML, and structured documents
- Example

```
<student>
        <name>Lisa Simpson</name>
        <age>…</age><advisor>…</advisor>…
        … …
        <school>Springfield Elementary<school>
</student>
<student>
        <name>Wesley Crusher</name>
        … …
        <school>Starfleet Academy</school>
</student>
```


Issues

- How to come up with a meaningful link structure for a database
  - What are the nodes? Links? Weights?
  - ➢Not addressed by this paper
- How to define proximity
  - Between one object (in *F*) and another (in *N*)
  - Between an object (in *F*) and a set of objects (*N*)
  - ➢Not a focus of this paper
- How to process proximity queries efficiently

Proximity functions

- Distance between objects f and n:
  $d(f, n)$ = weight of the shortest path between f and n
- Bond between two objects f and n:
  $b(f, n)$ = (rank of f in F) × (rank of n in N) / $d(f, n)^t$
  - $b(f, n) \in [0, 1]$; bigger number means a stronger bond
  - t controls the impact of distance (the paper used $t = 2$)
- Proximity of f to N:
  - Additive (used in the paper): $\Sigma_{n \in N}\, b(f, n)$
  - Maximum: $\max_{n \in N}\, b(f, n)$
  - Belief: $1 - \Pi_{n \in N}\,(1 - b(f, n))$

7

Computing proximity

- Requires efficient computation of $d(f, n)$
  - Shortest-path problem
- Naïve approach: compute $d(f, n)$ at run-time
  - Response time is unacceptable
- ➢ Pre-compute $d(f, n)$ for all pairs of f and n
  - Only need those $d(f, n) \le K$
    - The paper used $K = 12$
    - Above this threshold, $b(f, n)$ becomes insignificant
  - You can write it in SQL!

8

Self-join algorithm

To compute shortest paths up to weight K

- Start with E_1 whose rows have form (v_i, v_j, w_{ij})
  - w_{ij} is the weight of the link from v_i to v_j
- In general, E_l contains information about all shortest paths consisting of up to 2^{l-1} links with weight not exceeding K
- Self-join E_l with itself to obtain E_{l+1}
- Stopping condition: $l = \lceil \log_2 K \rceil + 1$
  - At this point, we have seen all paths consisting of up to K links

9

SQL for the self-join

- Here we assume directed edges (the paper assumes undirected edges)
- Construct shortest paths with up to 2^l links from shortest paths with up to 2^{l-1} links
  - Intuition: any shortest path with 2 to 2^l links can be broken into two shortest paths with 1 to 2^{l-1} links each


```
select t1.v1 as new_v1, t2.v2 as new_v2,
       (t1.dist + t2.dist) as new_dist
from E_l t1, E_l t2
where t1.v2 = t2.v1 and t1.v1 <> t2.v2
and t1.dist + t2.dist <= K;
```

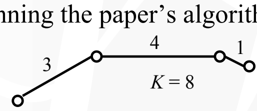
10

Bug in the paper?

- The paper has a stronger WHERE condition:

```
select t1.v1 as new_v1, t2.v2 as new_v2,
       (t1.dist + t2.dist) as new_dist
from E_l t1, E_l t2
where t1.v2 = t2.v1 and t1.v1 <> t2.v2
and t1.dist + t2.dist <= 2^l
and t1.dist + t2.dist <= K;
```

- Try running the paper's algorithm on



11

SQL for computing E_{l+1} from E_l

```
E_{l+1} :=
  select new_v1, new_v2, min(new_dist)
  from
   (select v1 as new_v1, v2 as new_v2,
           dist as new_dist
    from E_l)
   union
   (select t1.v1 as new_v1, t2.v2 as new_v2,
           (t1.dist + t2.dist) as new_dist
    from E_l t1, E_l t2
    where t1.v2 = t2.v1 and t1.v1 <> t2.v2
    and t1.dist + t2.dist <= K)
  group by new_v1, new_v2;
```

Can we replace it with E_1?

Why necessary?

Paths found by the second subquery make at least one hop

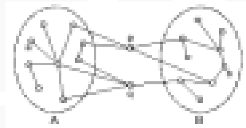
12

Why self-join?

- Self-join algorithm
  - "Squaring" E_i in each step
  - E_i contains all shortest paths with up to 2^{i-1} links
  - $O(\log K)$ joins required
- An alternative algorithm
  - "Multiplying" E_i with E_1 in each step
  - E_i contains all shortest paths with exactly i links
  - $O(K)$ joins required

Hub indexing

- Reduce the amount of shortest-path information that needs to be pre-computed and stored



- Find hubs, nodes whose removal disconnects the graph
- Any path that connects subgraphs (e.g., A and B) must go through hubs

➢ No need to remember the shortest paths between nodes in different subgraphs

An analogy

- Hierarchical path planning in AI (or MapQuest?)
- Example: from Durham, NC to Fremont, CA
  - Use major waypoints: RDU (Raleigh-Durham Airport), SJC (San Jose Airport)
  - Look up how to go from RDU to SJC
  - Look up how to go from Durham to RDU, and how to go from SJC to Fremont

Issues

- Constructing the hub index
- Using the hub index to look up shortest paths
- Picking hubs

Constructing the hub index (slide 1)

- Suppose we have already picked H, a set of hubs
- When running the self-join algorithm, do not generate any path that goes through a hub in H
  - select t1.v1 as new_v1, t2.v2 as new_v2,
    (t1.dist + t2.dist) as new_dist
    from E_i t1, E_i t2
    where t1.v2 = t2.v1 and t1.v1 <> t2.v2 and t1.v2 $\notin H$
    and t1.dist + t2.dist <= K;
  - But do generate paths that begin and/or end with hubs

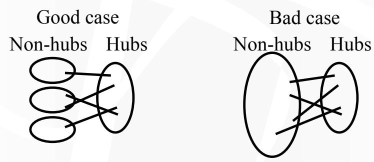
Constructing the hub index (slide 2)

- Output from the self-join algorithm contains shortest paths (without crossing any hubs) between:
  - Two non-hub nodes
  - A non-hub node and a hub
  - Two hubs
- Starting with the above output for two hubs, compute all shortest paths among hubs (now allowing crossing other hubs and non-hubs)
  - Compute in memory (assuming $|H|$ is small)

Hub index

- *Hub_Dist*: shortest distances between any two hubs
  - In memory
- *Dist*: shortest distances that do not cross any hubs
  - On disk

➢ Note that we do not assume hubs disconnect the graph (although that would help with the performance)



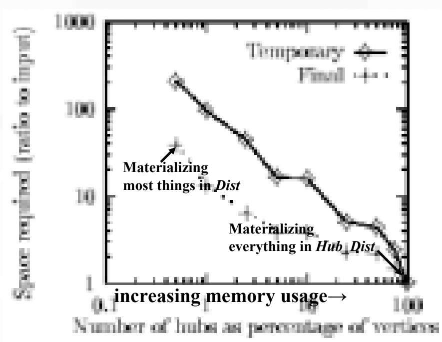
Using hub indexes

- Between two hubs *h* and *h*'
  - Return *Hub_Dist*(*h*, *h*') in memory
- Between a non-hub *n* and a hub *h*
  - Calculate *Dist*(*n*, *h*') + *Hub_Dist*(*h*', *h*) for all *h*' in *H*
  - Pick the smallest
- Between two non-hubs *n* and *n*'
  - Check *Dist*(*n*, *n*')
  - Calculate *Dist*(*n*, *h*) + *Hub_Dist*(*h*, *h*') + *Dist*(*h*', *n*') for all pairs of *h*, *h*' in *H*
  - Pick the smallest

Picking hubs

- Heuristic: select up to sqrt(*M*) nodes with highest number of links
  - *M* is the size of memory available for *Hub_Dist*
- Intuition
  - These nodes are more likely to be on shortest paths
    - For example, if we assume that each link has a fixed probability of being on a shortest path
  - If a node with *n* links is not chosen as hub, n^2 tuples will be generated by the self join

Effectiveness of hub indexes



Future work

- Better hub selection
- Optimizing a set of lookups (*F* and *N*)
  - Mentioned in paper
    - Get all data related to *F* (or *N*) in memory
    - Cluster nodes that often appear together in *F* and *N*
- Incremental maintenance
- Other index compression schemes (including lossy ones)
- More expressive queries
  - Find actors near (find movies near Cage)
  - Find movies *not* near Cage

WSQ (Web-Supported Query)

- Integrate
  - Keyword-based searches on the Web
  - SQL queries over a database
- Example: database with info about ACM SIG's
  - Rank SIG's by how often they appear on the Web near the keyword "Knuth"
- Issues
  - How to write the query (keyword searches + SQL)
  - How to process the query (requests to Web search engines + database query processing)

Integrating Web searches in SQL

- Virtual table
  - WebPages(T1, …, Tn, URL, Rank, …)
    - SearchExp (omitted here) depends on the search engine
    - T1, …, Tn are the search terms
    - URL and Rank are returned by the search engine
  - A convenient view over WebPages:
    WebCount(T1, …, Tn, Count) :=

```
select T1, …, Tn, count(*) as Count
from WebPages group by T1, …, Tn;
```

    - Not exactly (why?)
- Example: SIGs(Name, …)

```
select Name, Count
from SIGs, WebCount
where T1 = Name and T2 = 'Knuth'
order by Count desc;
```


Limited access patterns

- Virtual tables can be accessed only in certain ways
  - Example of a valid access:

```
select * from WebPages
where T1 = ??? and Rank <= 10;
```

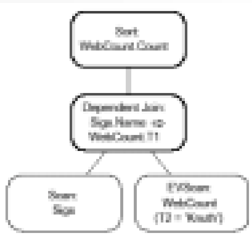
  - Examples of invalid accesses:

```
select * from WebPages;
select T1 from WebCounts where Count = 3;
```

- Issues
  - How to specify valid access patterns
  - How to process queries when access patterns are limited
  - ➢ Problem of "answering query using views"; to be addressed later in this course

Naïve query execution plan

- Synchronous iteration
- For each SIGs.Name
  - Issue a Web request to count number of pages containing both SIGs.Name and Knuth
  - Wait for the request to complete and return the joined tuple (that is, synchronization in every iteration)

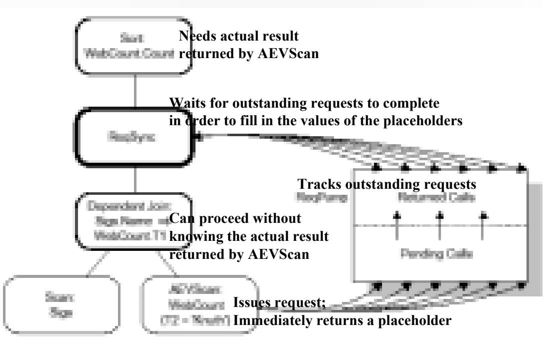


DJ obtains bindings from the left operand and pass them to the right operand

Problems and opportunities

- Problems
  - Latency of a single Web search request is very high
  - The database query processor is idle all the time
- Opportunities
  - Web can handle many concurrent requests
  - Parts of the query still can be processed without knowing the information returned from the Web searches
  - ➢ Asynchronous iteration

Asynchronous iteration plan



Tuple generation and cancellation

- Suppose a tuple *T* is waiting in ReqSync for a call *C* to complete
- What if *C* returns multiple (*n*) tuples?
  - ReqSync creates *n* – 1 additional copies of *T* and fills in the missing attribute values from the *n* returned tuples
- What if *C* returns one tuple?
  - ReqSync simply fills in the missing attribute values in *T* from the returned tuple
- What if *C* returns no tuple at all?
  - ReqSync removes *T*

How about parallel query processing?

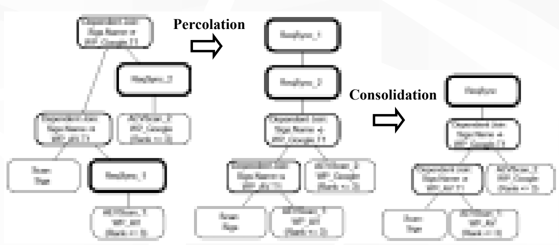
- Traditional parallel query processing techniques
  - Intra-operator parallelism
    - Multiple threads work on the same operator by dividing up its work (e.g., multiple threads send WebPages requests for an EVScan operator)
  - Inter-operator parallelism
    - Different threads work on different operators in the pipeline (e.g., a selection operator can work on the current input tuple while its child work on producing the next input tuple)
  - Key idea: different tuples can be processed independently
- Asynchronous iteration also recognizes the fact that different parts of a tuple can be processed independently

Generating asynchronous plans

- Query: find three top pages for each SIG from AltaVista and from Google
- Convert synchronous EVScan to asynchronous AEVScan
- Insert ReqSync directly above AEVScan

Transforming asynchronous plans

- ReqSync percolation: "pull up" ReqSync as much as possible
- ReqSync consolidation: replace multiple adjacent ReqSync's by one



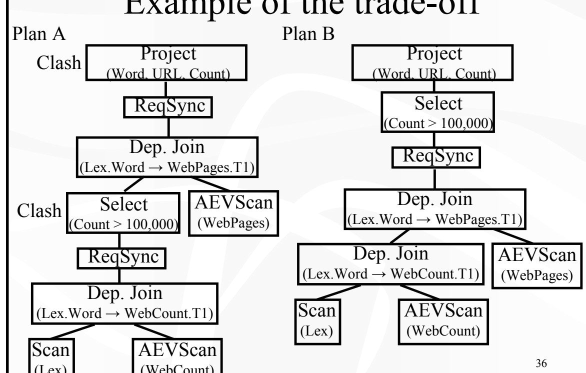
More on percolation

- Intuition: synchronize as late as possible to minimize wait by the database query processor
- Cannot pull up ReqSync through an operator *O* that clashes with it
  - *O* reads the missing information (e.g., a selection operator with condition Count > 100)
  - *O* projects away the placeholder (without the request identifier, ReqSync cannot perform tuple cancellation or generation properly)
  - *O* is an aggregation or existential operator (which requires knowing the exact count)

Avoiding clashes

- Transform the query plan to delay operations that clash with ReqSync
- Examples
  - Pull up projections that clash with ReqSync
  - Pull up selections that clash with ReqSync
    - Against the conventional wisdom of pushing down selections
    - More work versus more wait
  - Convert joins to cross products followed by selections

Example of the trade-off



Plan A versus Plan B

- Plan A
  - Too conservative
  - More waiting (for WebCount requests to complete)
- Plan B
  - Too aggressive
  - More work (many unnecessary WebPages requests)
- How would you execute this query?
  - A more adaptive query plan
  - Check out Avnur and Hellerstein. “Eddies: Continuously Adaptive Query Processing.” *SIGMOD*, 2000