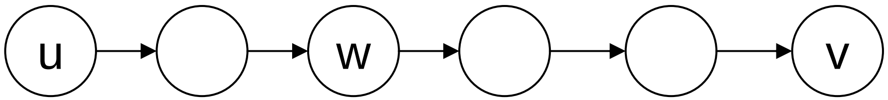
Advanced Divide-and-Conquer Algorithms for Computing Two-Hop Covers for Large Collections of XML Documents

Oberseminar AG 5, WS ´04/´05

Andreas Broschart
Supervisor: Dr.-Ing. Ralf Schenkel

HOPI

- index for XML document collection,
  use Two-Hop Cover concept (Cohen et al.)
  => compressed storage of transitive closure (on element level)



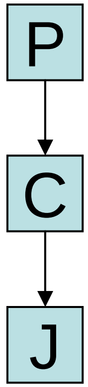
$L_{out}(u) = \{w, ...\}$
$L_{in}(v) = \{w, ...\}$

w center node

$L_{out}(u) \cap L_{in}(v) \neq \emptyset \Leftrightarrow$ there is a connection from u to v

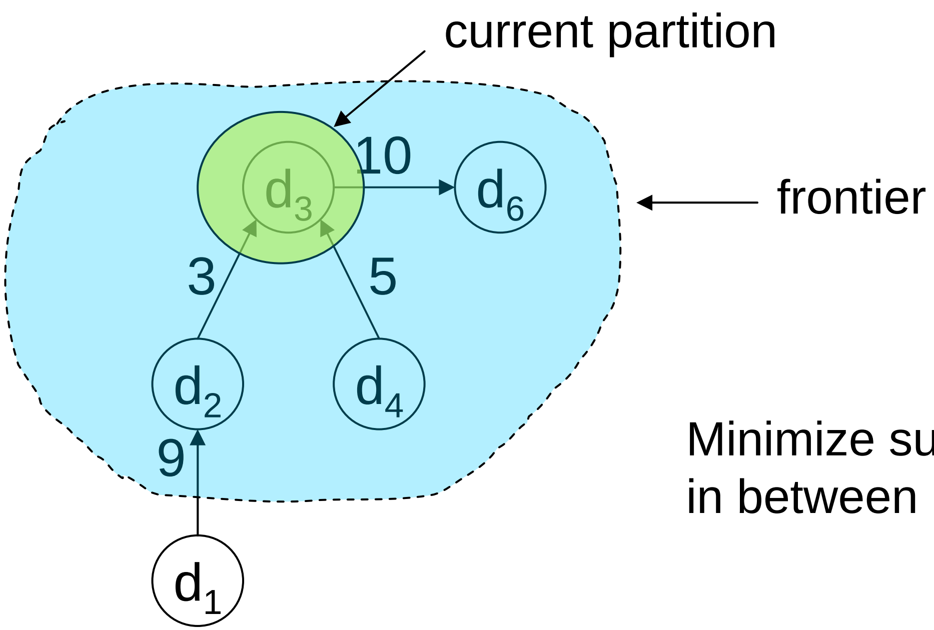
Computation of HOPI and goals

- compute HOPI using divide-and-conquer algorithm:
  - Compute the partitioning for the document collection
  - Compute the single partition covers
  - Join the partition covers



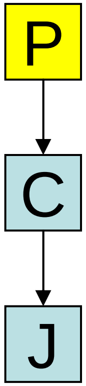
- Our goals:
  - reduce the size of the computed 2-hop cover
  - reduce the time needs

Partitioning process – example for frontier



Minimize sum of edge weights in between partitions!

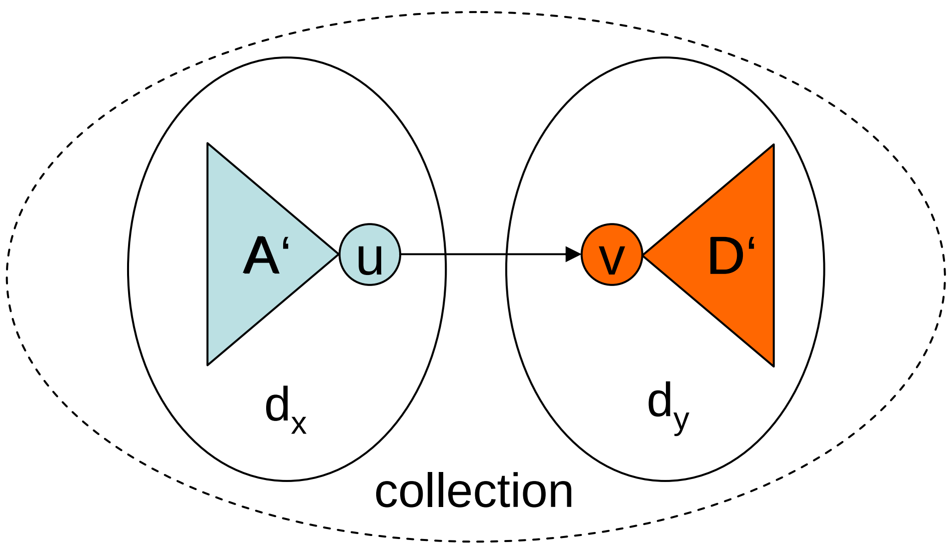
So far:
Edge weight: count the number of links in between two documents



Variation of edge weights

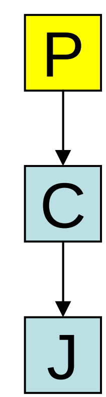
New:

- #connections induced by two documents: A‘*D‘
- #elements connected by two documents: A‘+D‘



Computation of A‘, D‘: easy (pre/postorder)

Computation of A‘, D‘: use skeleton graph [ICDE2005]

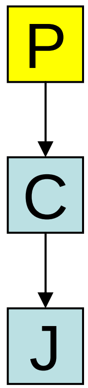


New connection based partitioner

- old approach counts number of elements in each partition
  => no uniform distribution of connections over partitions

- new approach creates transitive closure of partition‘s element graph
  => limit: size of transitive closure

- Two variants:
  - optimistic approach:
    assume that candidate document fits into the current partition
    (with possibility to do rollback)
  - pessimistic approach:
    estimate the number of new connections

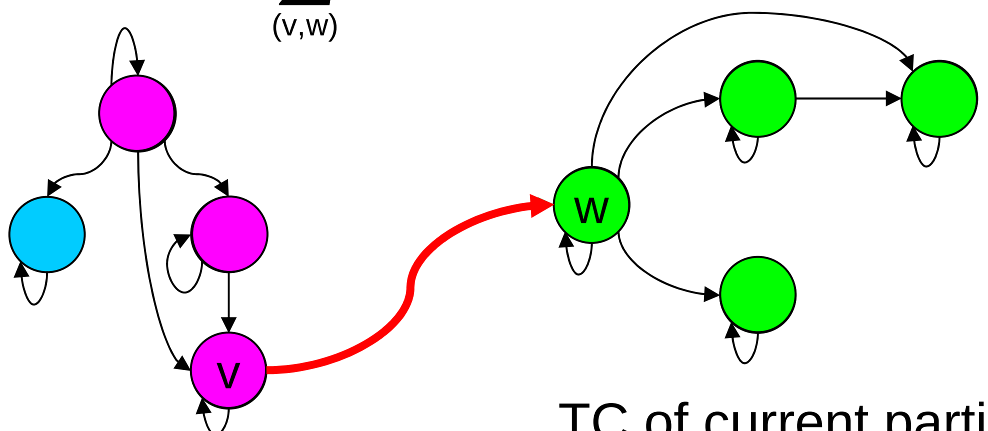


Estimation

before candidate document is assigned to current partition:

- compute transitive closure for element graph of candidate document
- consider all links (v,w) from candidate document to current partition and vice versa

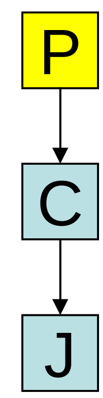
$$\#\text{new connections} = \sum_{(v,w)} \#\text{ancestors}_{TC}(v) * \#\text{descendants}_{TC}(w)$$



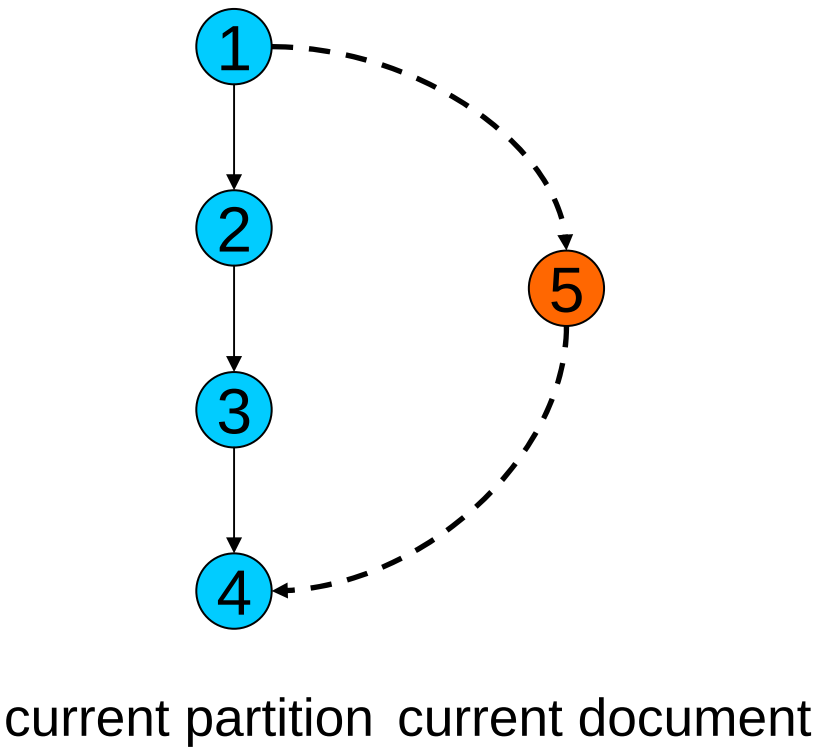
TC of current partition

TC of candidate document

connect every ancestor of v with every descendant of w:
estimation=3*4=12 is correct.
But: we can also over- and underestimate!



Optimistic partitioning with rollback

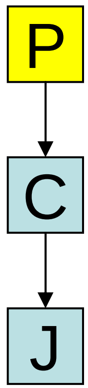


TC
(1,1,0)
(1,2,1)
(1,3,2)
(1,4,3)
(2,2,0)
(2,3,1)
(2,4,2)
(3,3,0)
(3,4,1)
(4,4,0)
(5,5,0)
(5,4,1)
(5,5,0)
(5,5,0)

Log
new (5,5,0)
new (1,5,1)
new (5,4,1)
update (1,4,3)

Rollback!

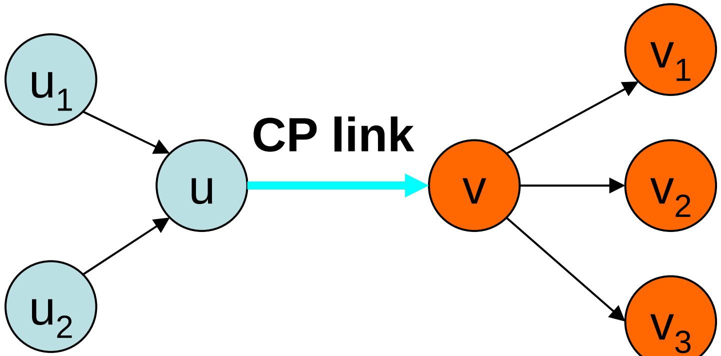
Rollback finished!

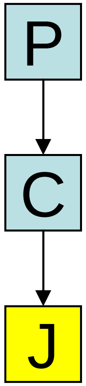


How do we connect the partition covers?

- for each cross partition link (u,v):
  - get known ancestors of u within 2-hop labeling
  - get known descendants of v within 2-hop labeling

- choose v as center node for connecting the partition covers



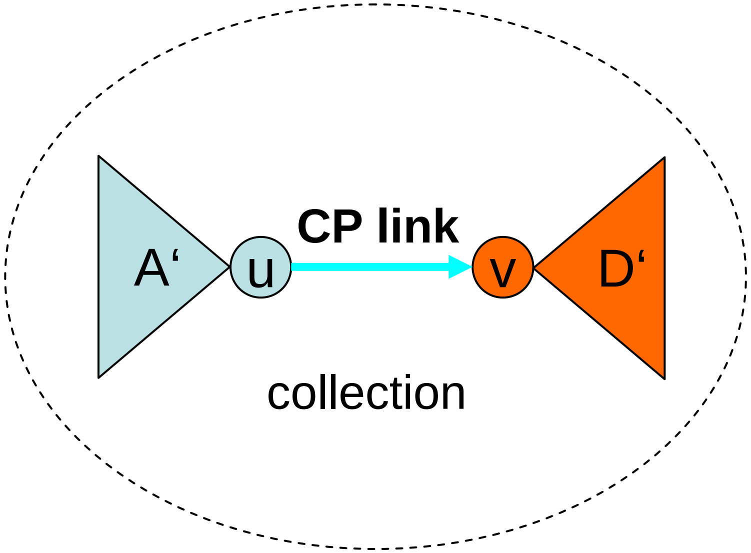


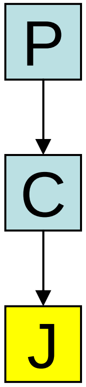
Connecting the partition covers

Join partition covers along cross-partition links in different orders:

Up to now:
- Order by (linktarget ID, linksource ID) ascending

New:
- Order by A'*D' descending
- Order by A'*D' ascending
- Order by A'+ D' descending
- Order by A'+ D' ascending
- Order by max {A', D'} descending
- Order by min {A', D'} ascending



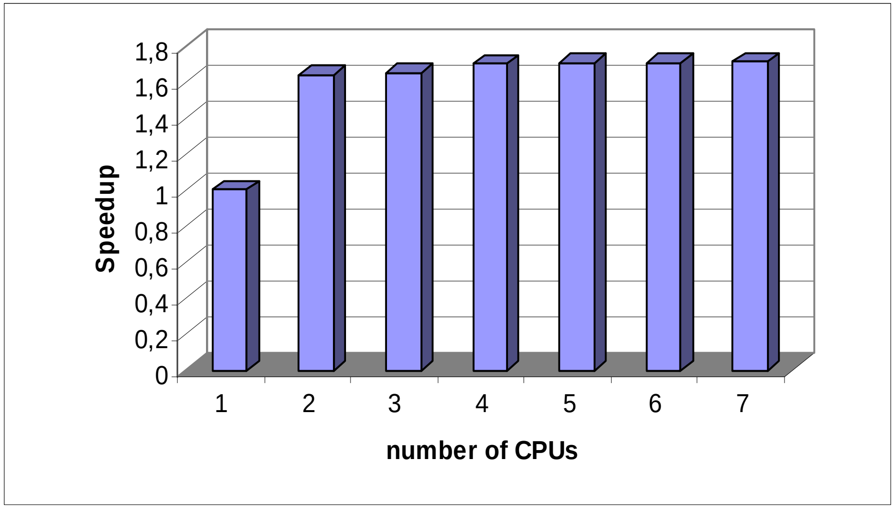


Experimental setup

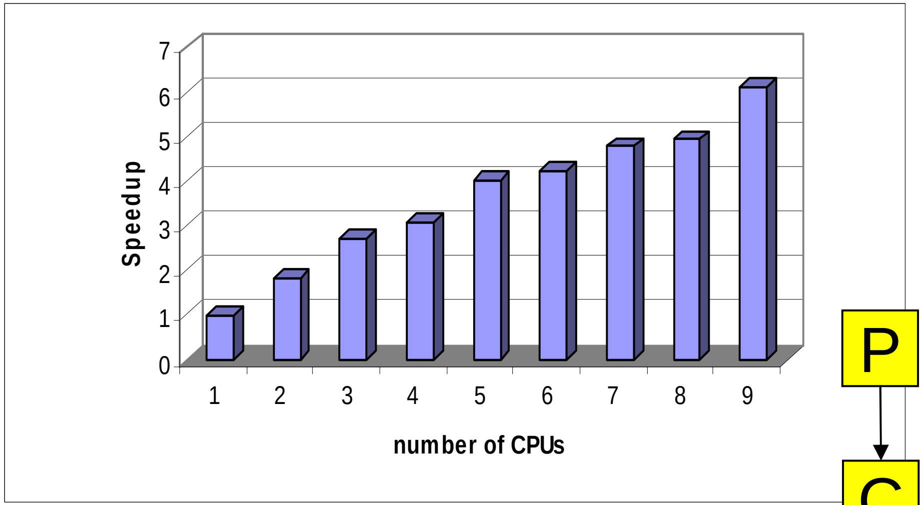
- DBLP fragment with 6,210 documents
- 168,991 elements, 162,781 edges, 25,368 links
- Transitive closure: 344,992,370 connections
- CPU: Intel Pentium 4, 3 GHz
- RAM: 1 GB
- HDD: 120 GB
- OS: Windows XP Professional
- VM: SUN Java 1.4.2
- DBS: Oracle 9.2

Comparing the old and new partitioning approach

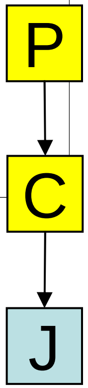
- old partitioning approach computes much faster (3 min vs. 8 min - 30 min)
- new partitioning approach fills the partitions in a balanced way => better scalability when computing partition covers simultaneously



element based partitioning



connection based partitioning



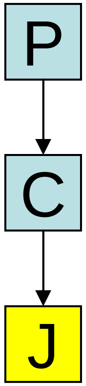
Variation of cover join order

Base line: element based partitioning approach, edge weight: #links

cover join order	cover size	time [sec]
(oid2, oid1) ascending	16,750,820	193,390

Connection based partitioning approach, edge weight: #links

cover join order	cover size	time [sec]
(oid2, oid1) ascending	16,649,966	250,589
A‘*D‘ descending	13,843,540	120,959
A‘*D‘ ascending	21,802,078	229,417
max{A‘,D‘} descending	12,186,321	158,224
min{A‘,D‘} ascending	16,771,056	212,919
A‘+D‘ descending	12,186,889	107,121
A‘+D‘ ascending	22,446,682	207,797



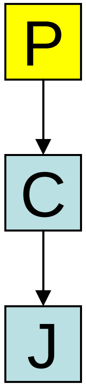
Variation of edge weights

Base line: element based partitioning approach, cover join order: (oid2, oid1) asc.

edge weight	cover size	time [sec]
#Links	16,750,820	193,390

Connection based partitioning approach, cover join order: max{A‘,D‘} desc.

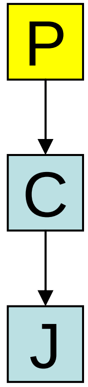
edge weight	cover size	time [sec]
#Links	12,186,321	158,224
A‘+D‘	10,186,488	91,528
A‘*D‘	10,410,923	104,534



Variation of transitive closure size

- cover size shrinks with increasing transitive closure size
- required time shrinks with increasing transitive closure size (up to a certain amount of connections)

#conns/part.	cover size	time[sec]
1 Mio.	10,186,488	91,528
5 Mio.	9,606,602	76,649
10 Mio. (*)	9,444,487	77,478



(*): computation on server due to large memory needs during partitioning

Summary experiments

best approach in our experiments:

- connection based partitioning, TC_{max}=10 Mio. connections, edge weight: A‘+D‘, cover join order: max{A‘,D‘} descending

with respect to baseline:

- size of 2-hop cover decreased from 16,750,820 to 9,444,487 entries representing 344,992,370 connections
  => savings ~44%
  => compression ratio of 36.5
- simultaneously time need decreased from 193,390 sec to 77,478 sec
  => savings ~60%

Future work

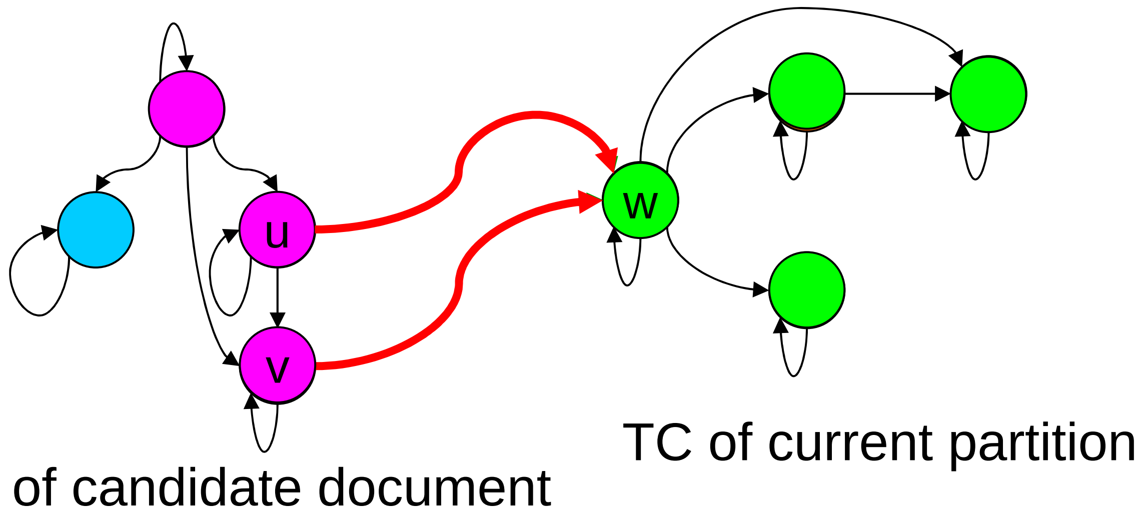
- multithreaded connection based partitioner
- multithreaded computation of partition covers
- local improvement methods for existing valid partitionings

(Kernighan-Lin, Fiduccia-Mattheyses, Simulated Annealing, ...)

$\Rightarrow$ less cross partitioning links

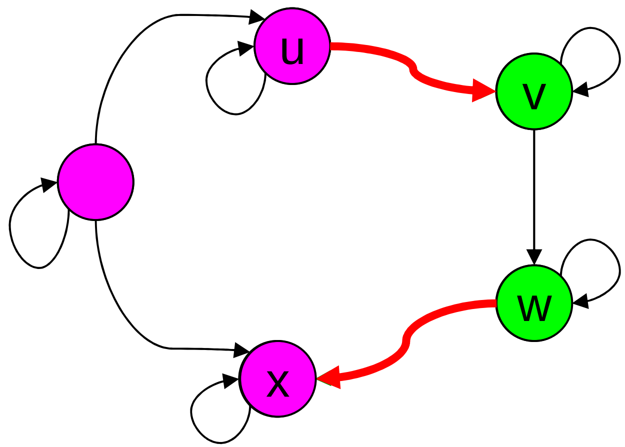
- usage of 2-hop cover algorithm in general graph applications, beyond usage of indexing xml document collections

Overestimation



evaluate (u, w): u has 2 ancestors, w has 4 descendants.
I.e. estimation=2*4=8 connections.
evaluate (v, w): v has 3 ancestors, w has 4 descendants.
I.e. estimation=3*4=12 connections.
Together with previous estimation: 20 connections.
Estimation too high: we only need 12 connections
Document fits into partition but is rejected => too small partitions

Underestimation



TC of candidate document TC of current partition

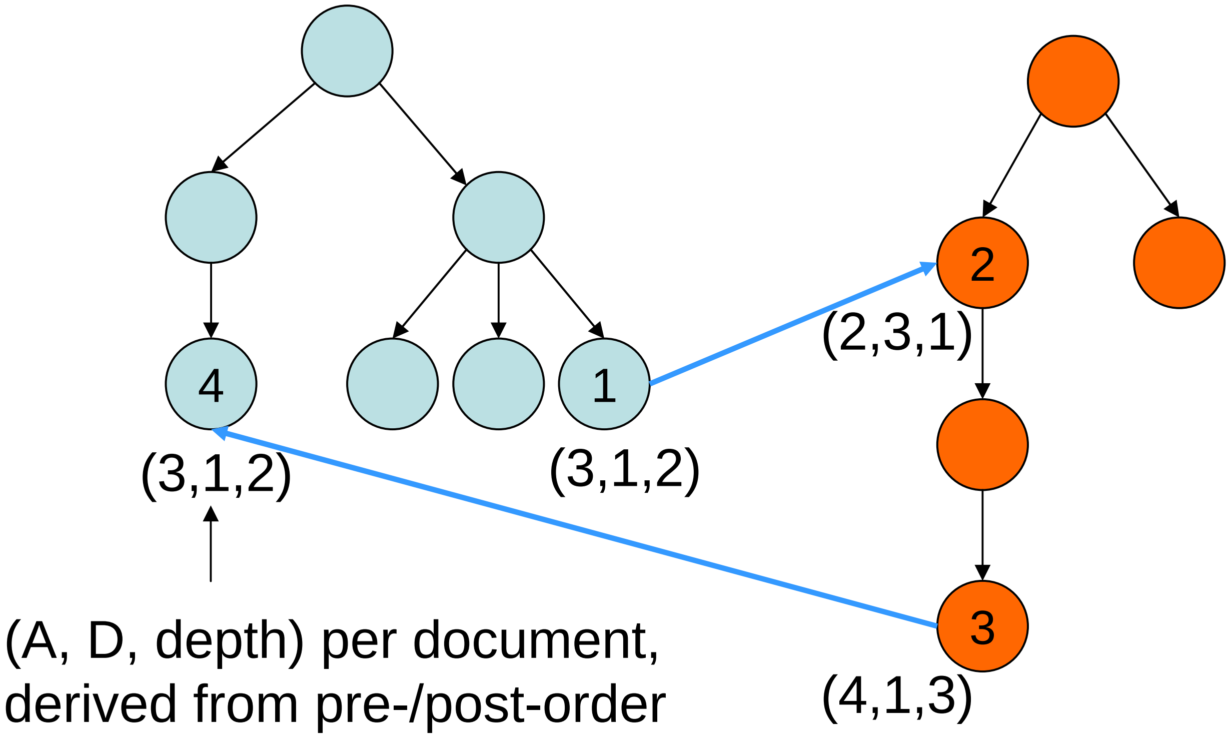
evaluate (u, v): u has 2 ancestors, v has 2 descendants.
I.e. estimation=2*2=4 connections.

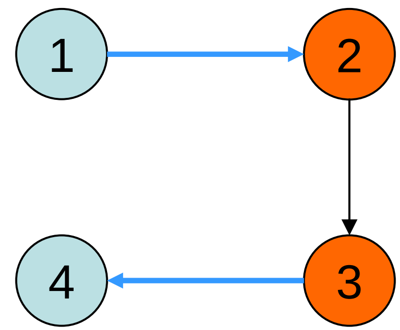
evaluate (w, x): w has 2 ancestors, x has 1 descendants.
I.e. estimation=2*1=2 connections.

Together with previous estimation: 6 connections.
Estimation too low: we need 7 connections - (u,x) not considered
=> partition gets too big

Computation of A‘ and D‘

Skeleton graph

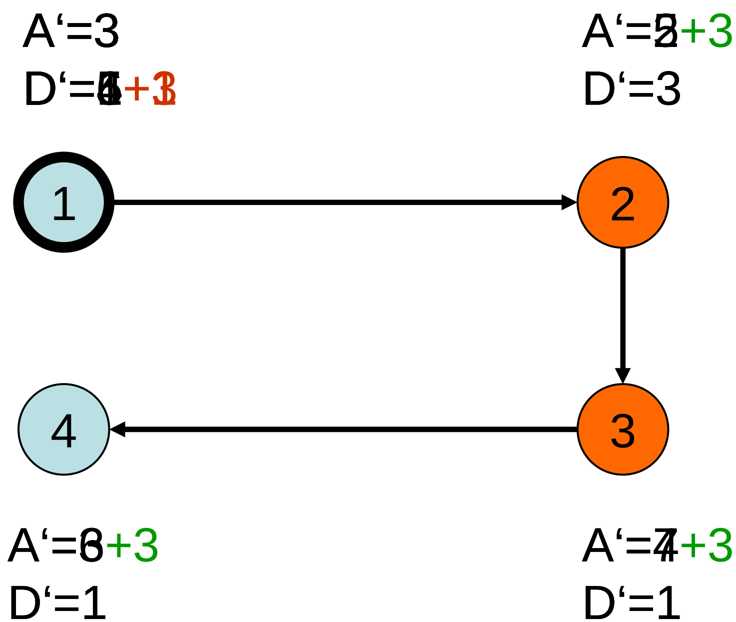




We want to compute the number of ancestors A‘
and descendants D‘ in the whole collection
Cost for computation of transitive closure too high!
=> Approximation by skeleton graph

Approximation of A‘ and D‘ (collectionwide)

- BFS starting with each node on skeleton graph
- Starting node gets descendants D of each visited node
- Visited node gets ancestors A of starting node



node	1	2	3	4
A	3	2	4	3
D	1	3	1	1

D‘(1)=D(1)+D(2)+D(3)+D(4)=6 approximates too big, but always upper bound.
Correct value: D‘(1)=D(1)+D(2)+D(4)=5.