Generating Efficient Execution Plans for Vertically Partitioned XML Databases

Research paper review by

QING Pei, Edward 11500811g
LO Wing Yi, Wing 11523479g
SHAO Shuai, Philip 11552402g

April 10, 2012

What ?

Why ?

How ?

For this paper, as well as today's presentation, we raised three questions.

What ?

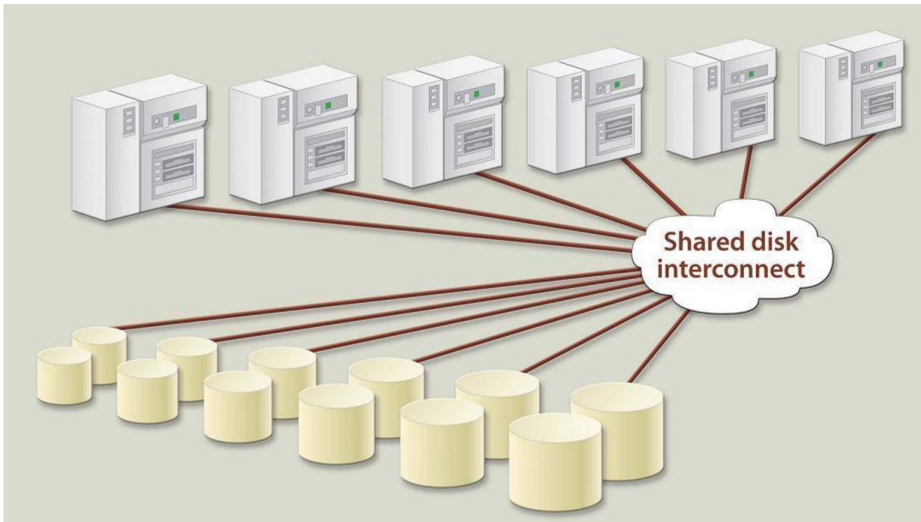
What is this paper talking about?

Query Processing

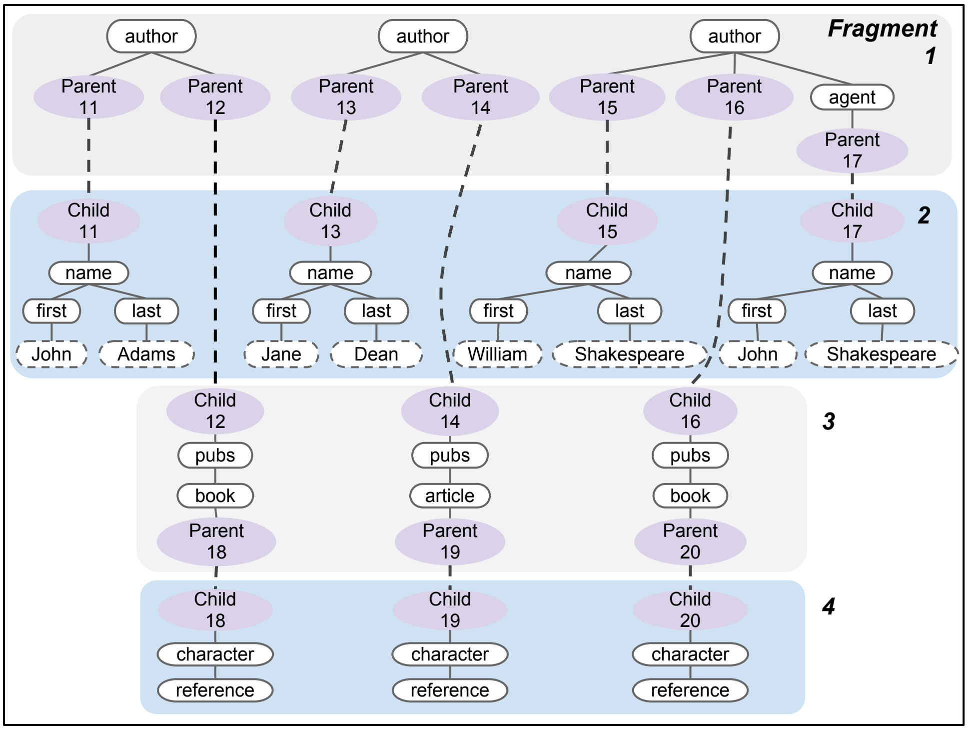
	Centralized	**Distributed**
RDBMS	✔	✔
XML	✔	***This paper***

It is focused on the query processing of distributed XML databases, other than a centralized one or relational databases.

XML in the Cloud



Several XML servers instead of a centralized one are distributed in the cloud



As shown in this picture, all data are orginally in one tree.
For better scalability, it has been vertically partitioned into four fragment.
So there are four (maybe five) distributed server here, restoring local data respectively.
That is what this paper is going to talk about.

Why ?

Now we come to the second question, why. Why this paper exist?

Distributed architecture
leads to
Different execution plans

For a single query, the **order** in which *joins* are performed results in various time consumed.

For a single query, there may exist different plans for local execution/joining operations.

Response time
=
local execution time
+
joining time

The equation
In naive distributed xml database query execution, if we want to execute a query, we have to access every tuple and every node on each distributed server. This is quite slow.
The join operation is also slow. Then it leads to even longer response time.
That's why we have this paper - to reduce the response time.

local execution time

snip(i): the number of document subtrees accessed by the local plan at *fragment i*

smaller snip(i) preferred

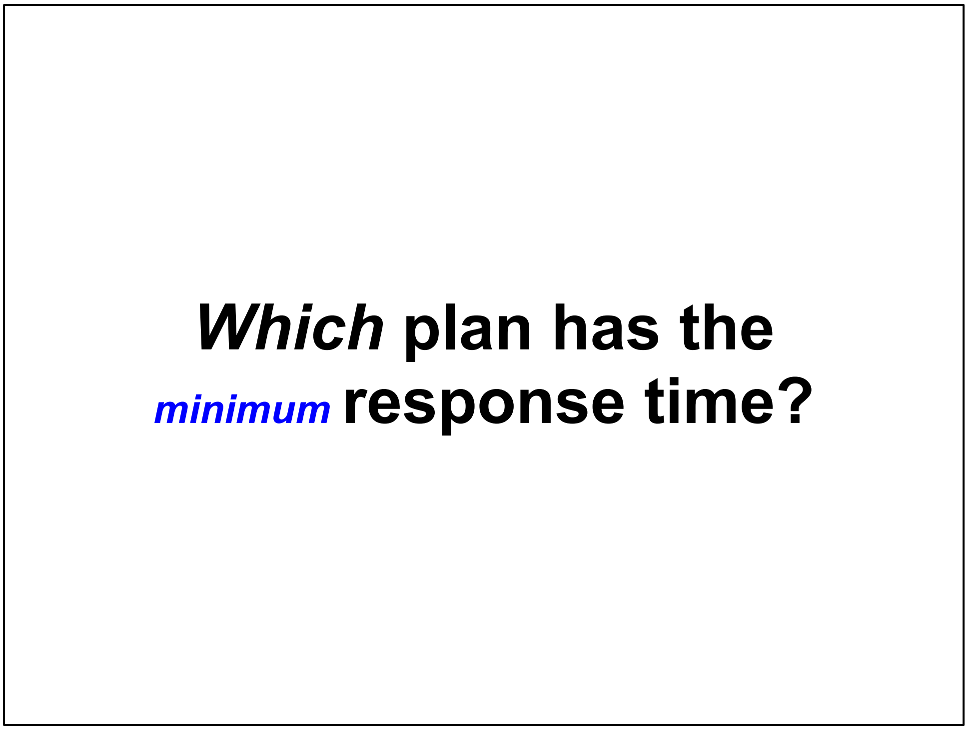
local execution time depends on snip

joining time

card(i): the number of tuples that are returned by the local plan when evaluated at *fragment i*

smaller card(i) preferred

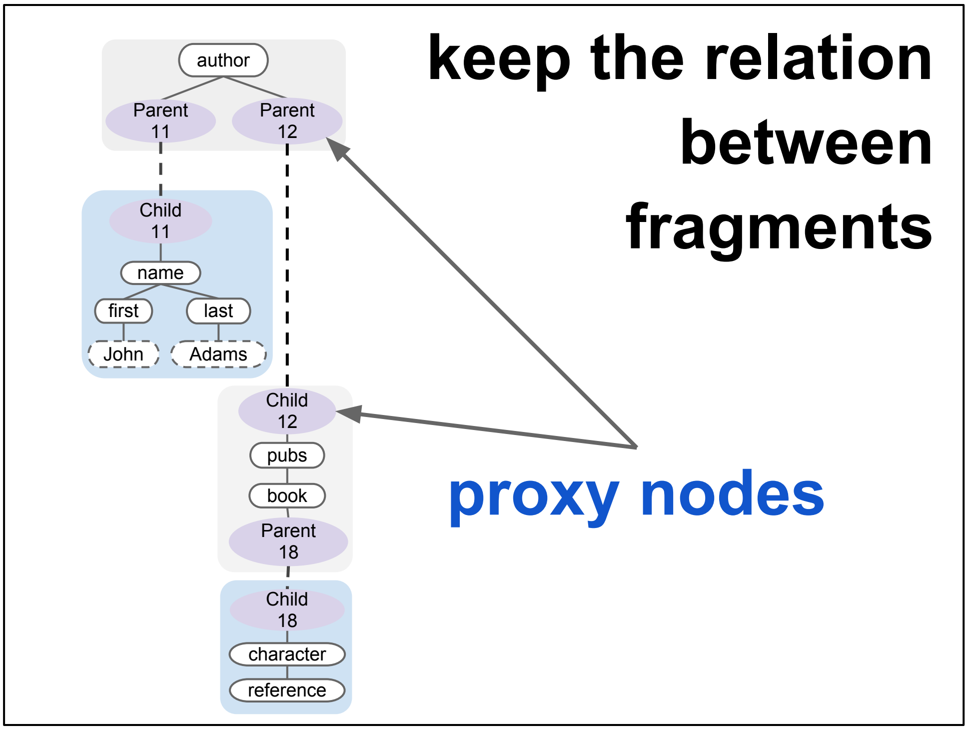
joining time depends on card



For the distributed plan, there are several optimizations for it.
This paper is trying to find fastest one.

How ?

Now, the third question, how.
How is it done?



One subtree of the original tree. Author and name elements are connected.
Some edges are broken during partitioning.
Proxy nodes are then introduced to recover the relations.

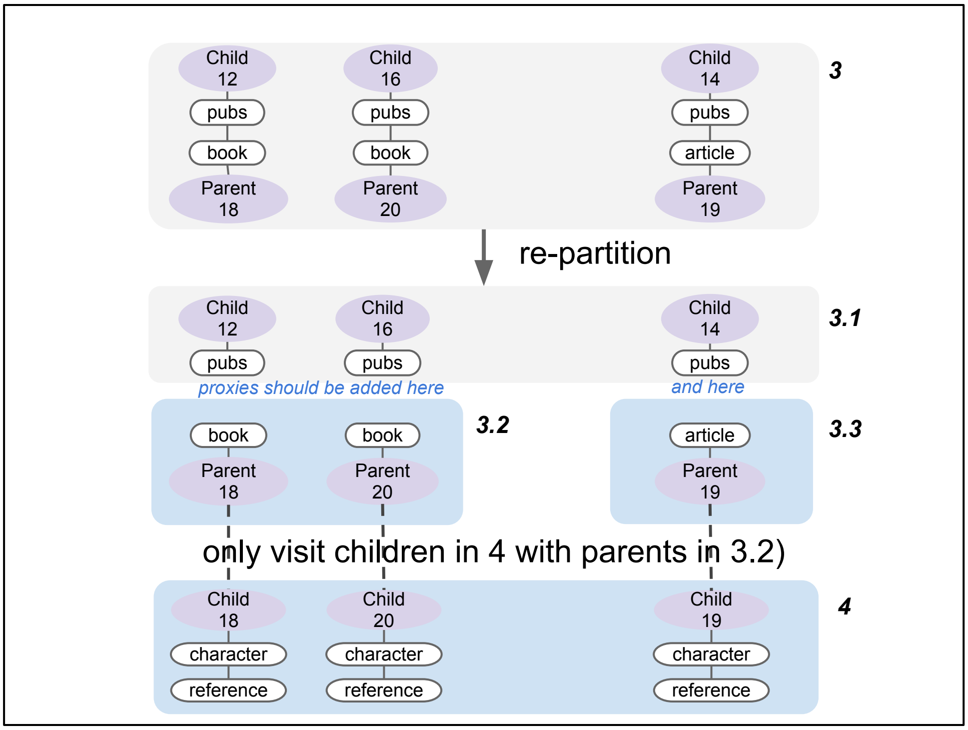
Optimizing distributed plans

There are several optimization plans, and this paper talks about two of them.

Optimizing distributed plans

Pushing Cross-Fragment Joins

fully works on left-deep plans

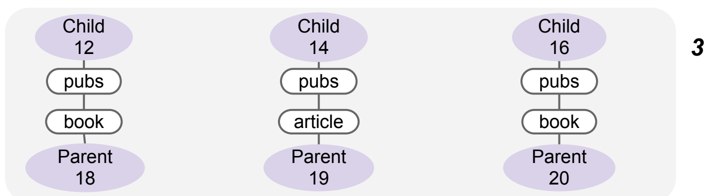


To illustrate how this works, we modify the example a little bit.
In the original partition, fragment three has been re-partitioned into two fragments so that the proxy nodes could work.
The proxy nodes reduce visits to unnecessary subtrees.

Optimizing distributed plans

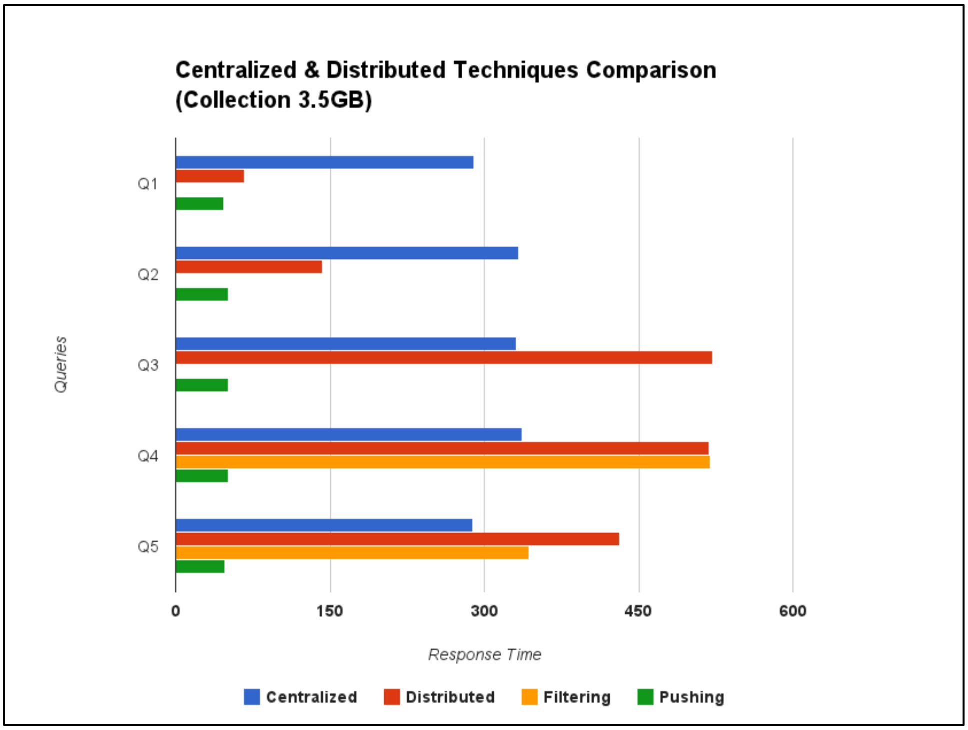
Label Path Filtering

//book//reference

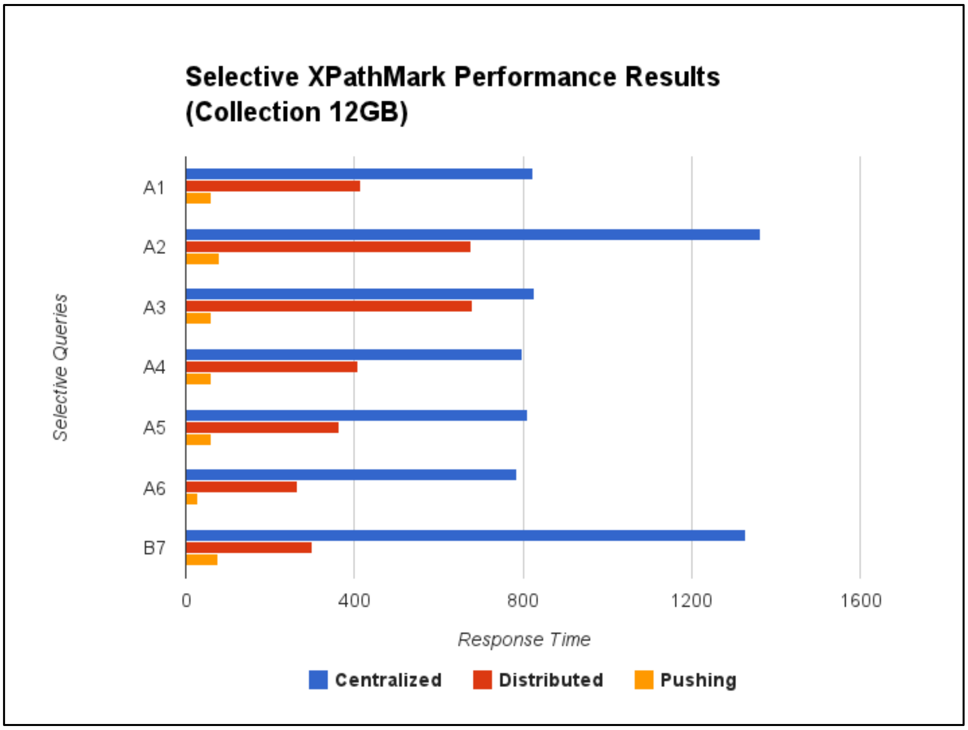


works in the situation that when a parent has several children that share its grandchild
It reduced the response time, snip, by avoiding access to some unnecessary subtree

Evaluation



We could see that Label Path filtering has its restrictions -
it works in the situation that when a parent has several children that share its grandchild



For selective XPath queries, the proposed optimizations greatly reduces the response time. It can get up to 10 times faster execution.

Conclusion

Greatly improves response time of querying large XML collections.

Small overhead. Choosing the fastest plan took **< 0.01** seconds.

Scalability.

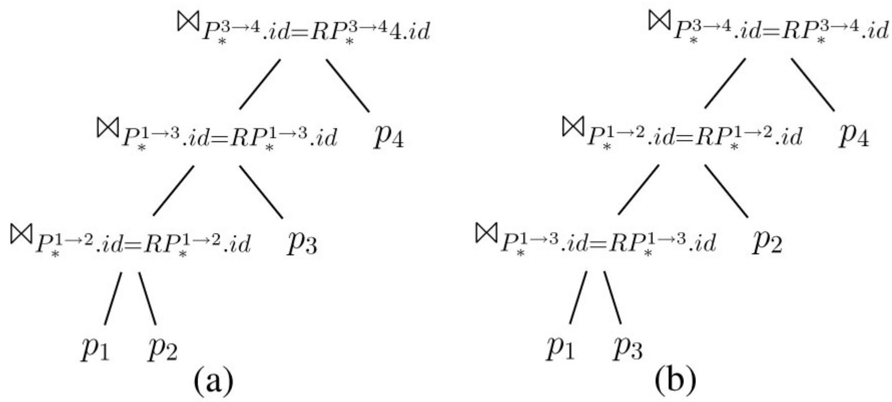
Q & A

Merci beaucoup

The end.

Appendix

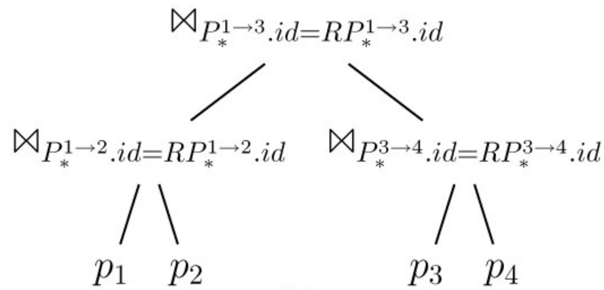
Distributed Execution Plans



left-deep execution plans

Appendix

Distributed Execution Plans



(c)

not a left-deep execution plan

Appendix

Queries used for evaluation

Q1 /open auction[initial > 200]//item//mail/from

Q2 /open auction[initial > 200][.//author/person/name[starts-with(., ’Ry’)]]//item//mail/from

Q3 /open auction[initial > 200][.//author/person/name[starts-with(., ’Ry’)]]//item//category/id

Q4 /open auction[initial > 200][.//author/person[profile/age > 30]/name[starts-with(., ’Ry’)]]//item//category/id

Q5 /open auction[initial > 200]//author/person[starts-with (name, ’Ry’)]/profile/interest/category/description

Appendix

Queries used for XPathMark

A1 /site/closed auctions/closed auction/annotation/description/text/keyword

A2 //closed auction//keyword

A3 /site/closed auctions/closed auction//keyword

A4 /site/closed auctions/closed auction [annotation/description/text/keyword]/date

A5 /site/closed auctions/closed auction[descendant:: keyword]/date

A6 /site/people/person[profile/gender and profile/age]/name

B7 //person[profile/@income]/name

Appendix

Queries used for Selective XPathMark

A1S /site/closed auctions/closed auction[price > 600]
/annotation/description/text/keyword

A2S //closed auction[price > 600]//keyword

A3S /site/closed auctions/closed auction[price > 600]
//keyword

A4S /site/closed auctions/closed auction[price > 600]
[annotation/description/text/keyword]/date

A5S /site/closed auctions/closed auction[price > 600]
[descendant::keyword]/date

A6S /site/people/person[starts-with(name, ’Ry’)]
[profile/gender and profile/age]/name

B7S //person[starts-with(name, ’Ry’)][profile/@income]/name

Appendix

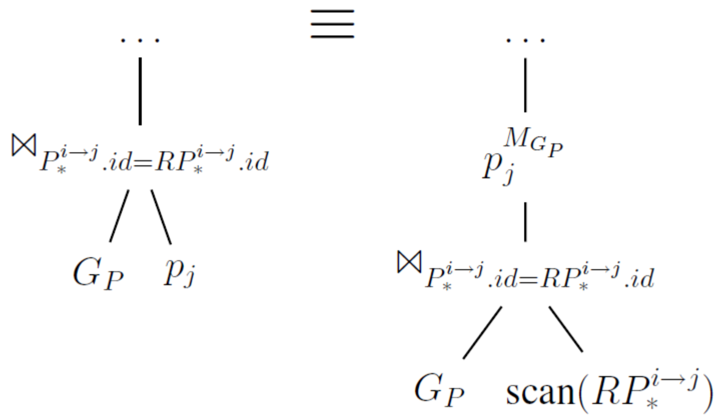


Figure 11: Cross-fragment join pushing rewrite

Appendix

$$p_j \equiv \begin{array}{c} p'_j \\ | \\ \sigma_{RP_*^{i\to j}.label \in L_j} \\ | \\ \text{scan}(RP_*^{i\to j}) \end{array}$$

Figure 12: Label path rewrite

Appendix

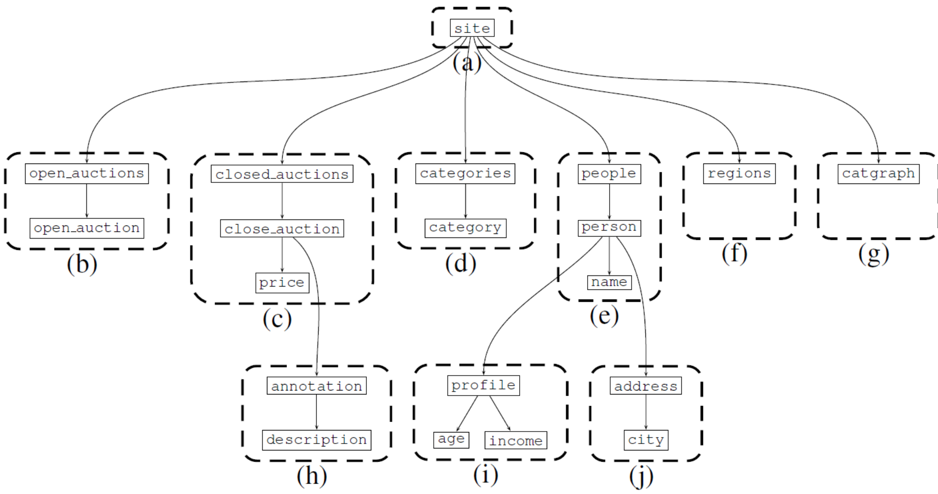


Figure 13: Fragmentation schema used in second experiment

Appendix

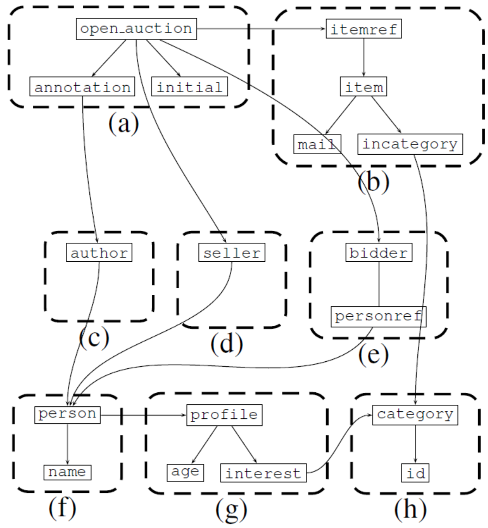


Figure 14: Fragmentation schema used in first experiment