LiquidXML: Self-adaptive Views in ViP2P

Asterios Katsifodimos
INRIA Saclay & Université Paris-Sud 11

Joint work with:

Ioana Manolescu
Jesus Camacho-Rodriguez

Outline

- Introduction
- The ViP2P Platform
- LiquidXML

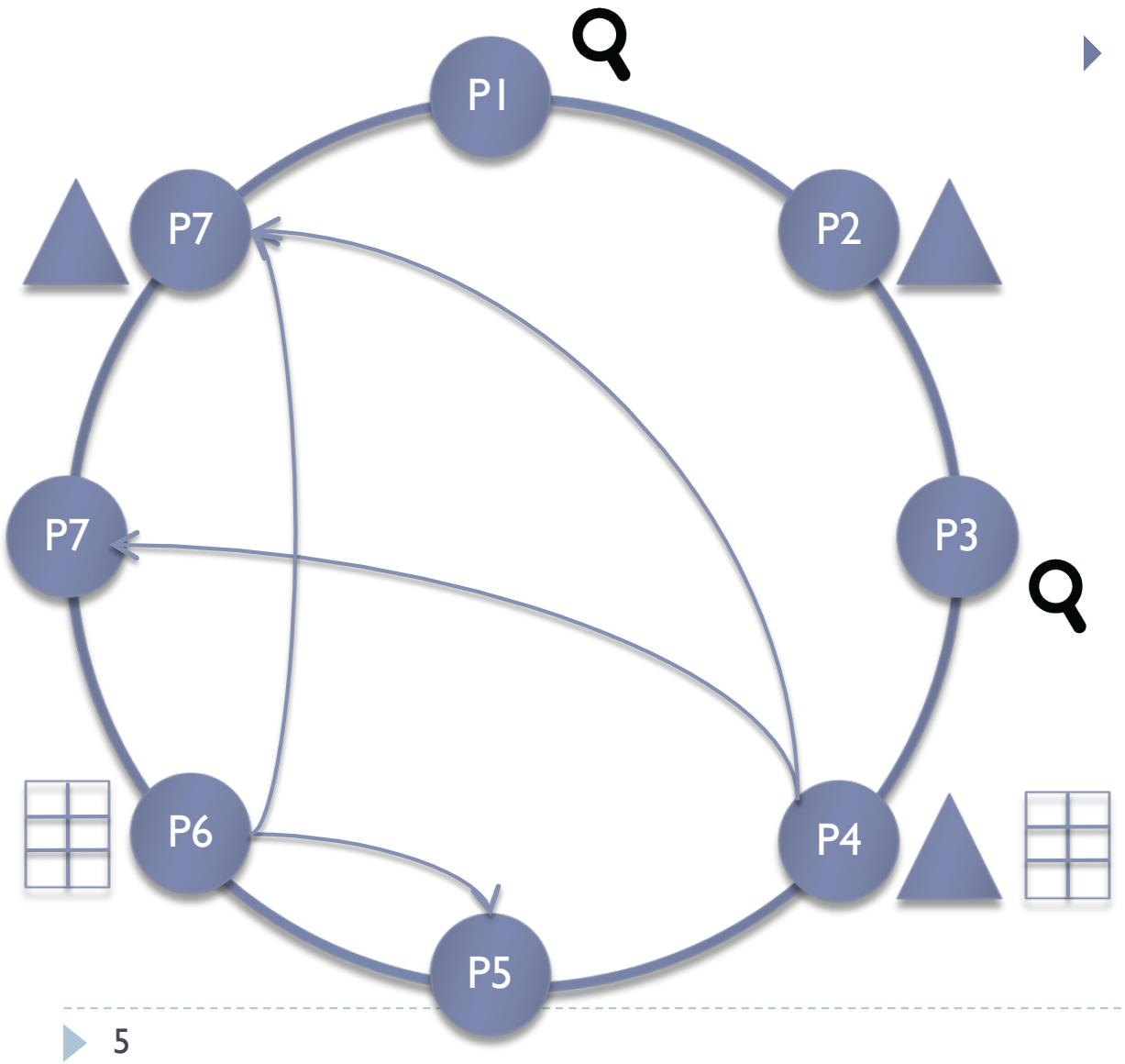
The ViP2P platform

XML Data Management on DHT based P2P networks using materialized views

Views in Peer to Peer (ViP2P)

- Platform for efficient, scalable management of XML data
  - Using materialized views
  - Powerful view based query rewriting
- Fully implemented (Java)
- Has been successfully deployed in up to 1000 peers
- More information: http://vip2p.saclay.inria.fr/

The ViP2P Platform

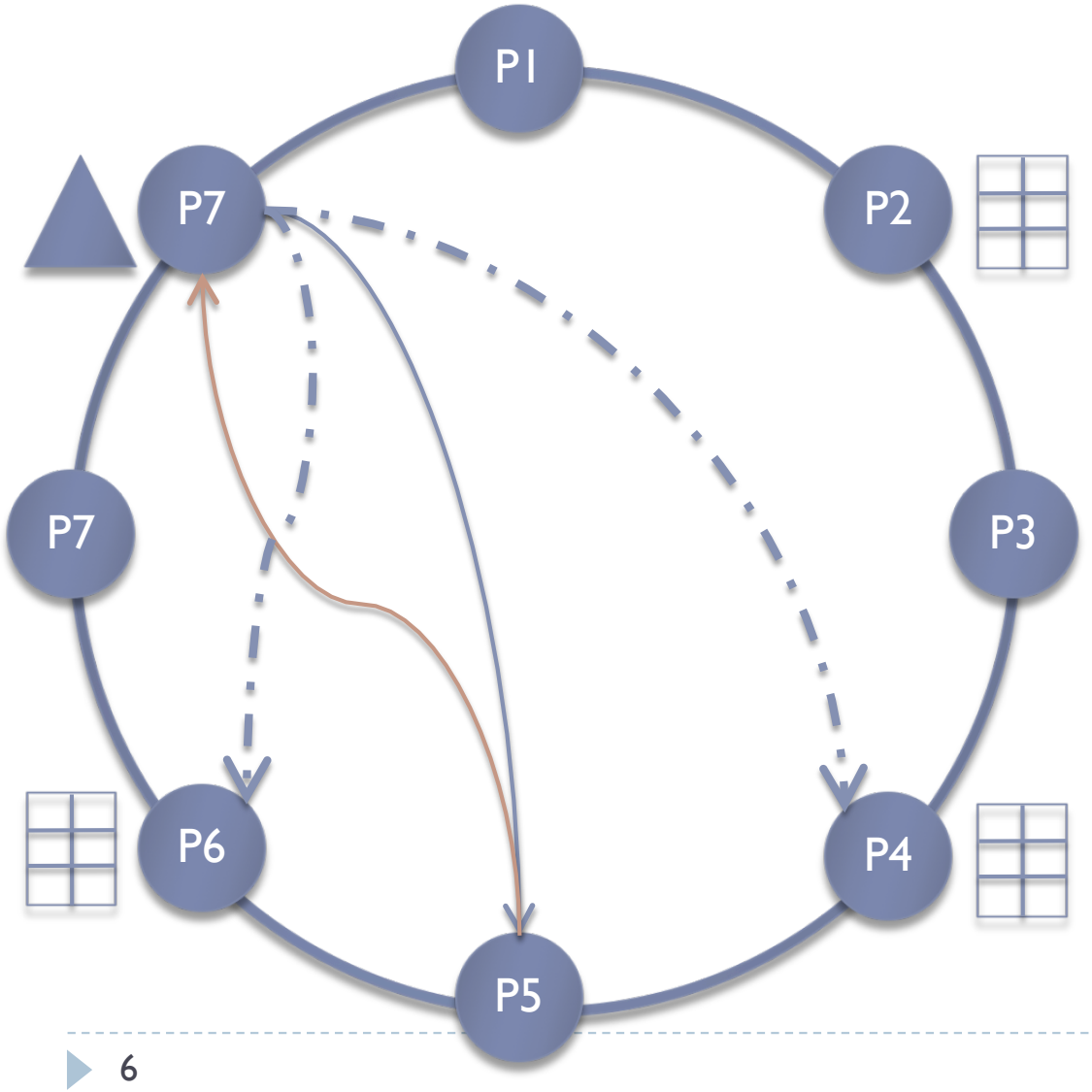


- Peers can:
  - Publish XML documents
  - Store materialized views
  - Advertise (index) them on the network
  - Pose queries
  - Answer them using views

Document

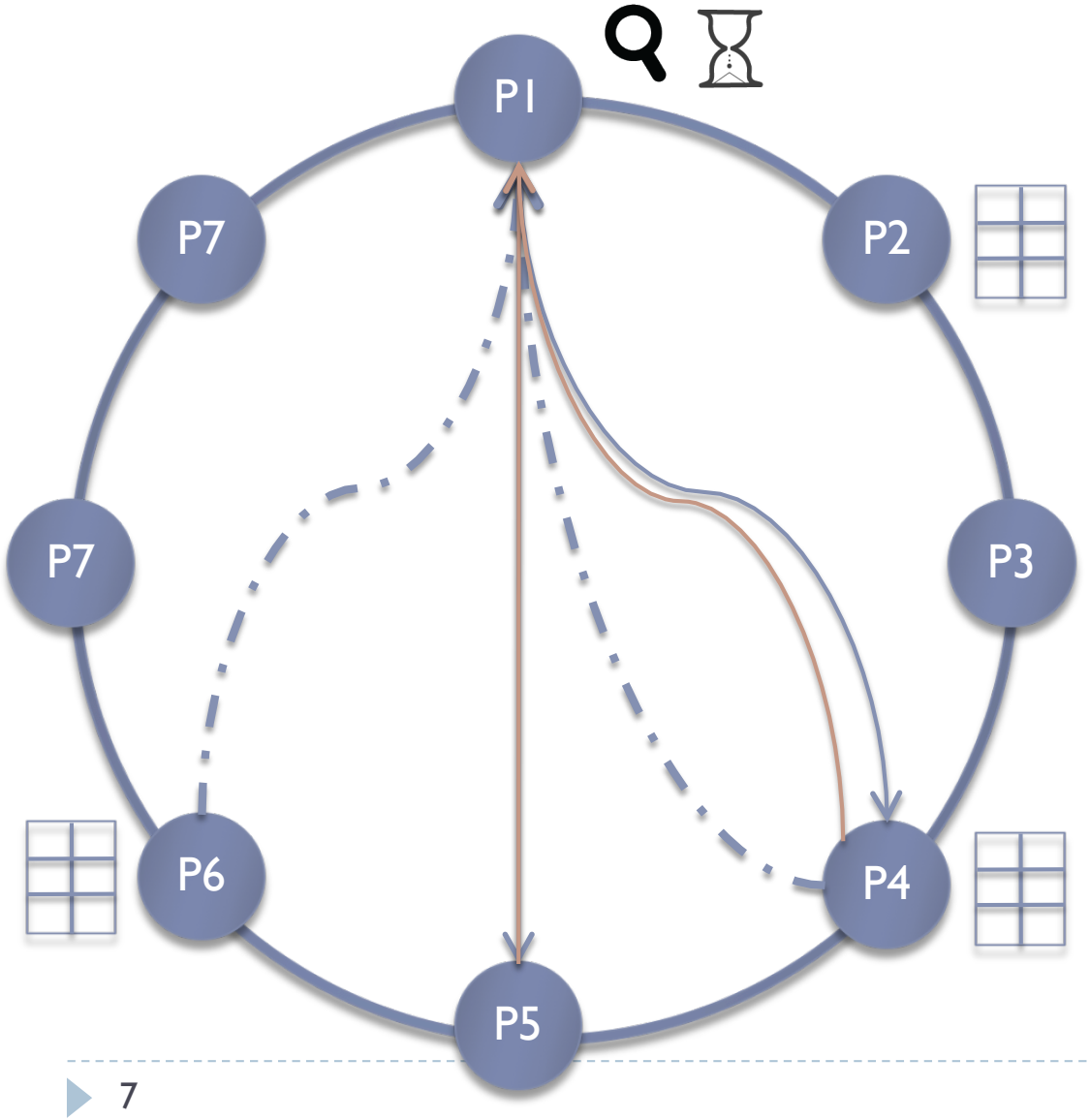
Materialized View

View materialization in ViP2P



- A document arrives at Peer 7
- Peer 7 looks up for views that it's document could contribute ($V(d) \neq \emptyset$)
- Retrieves the view definitions
- Extracts & pushes data to the view holders

Query rewriting in ViP2P



- A query arrives at Peer1
- Peer 1 looks up for views that can answer it's query
- Retrieves the view definitions
- Rewrites the query
- Executes the physical plan

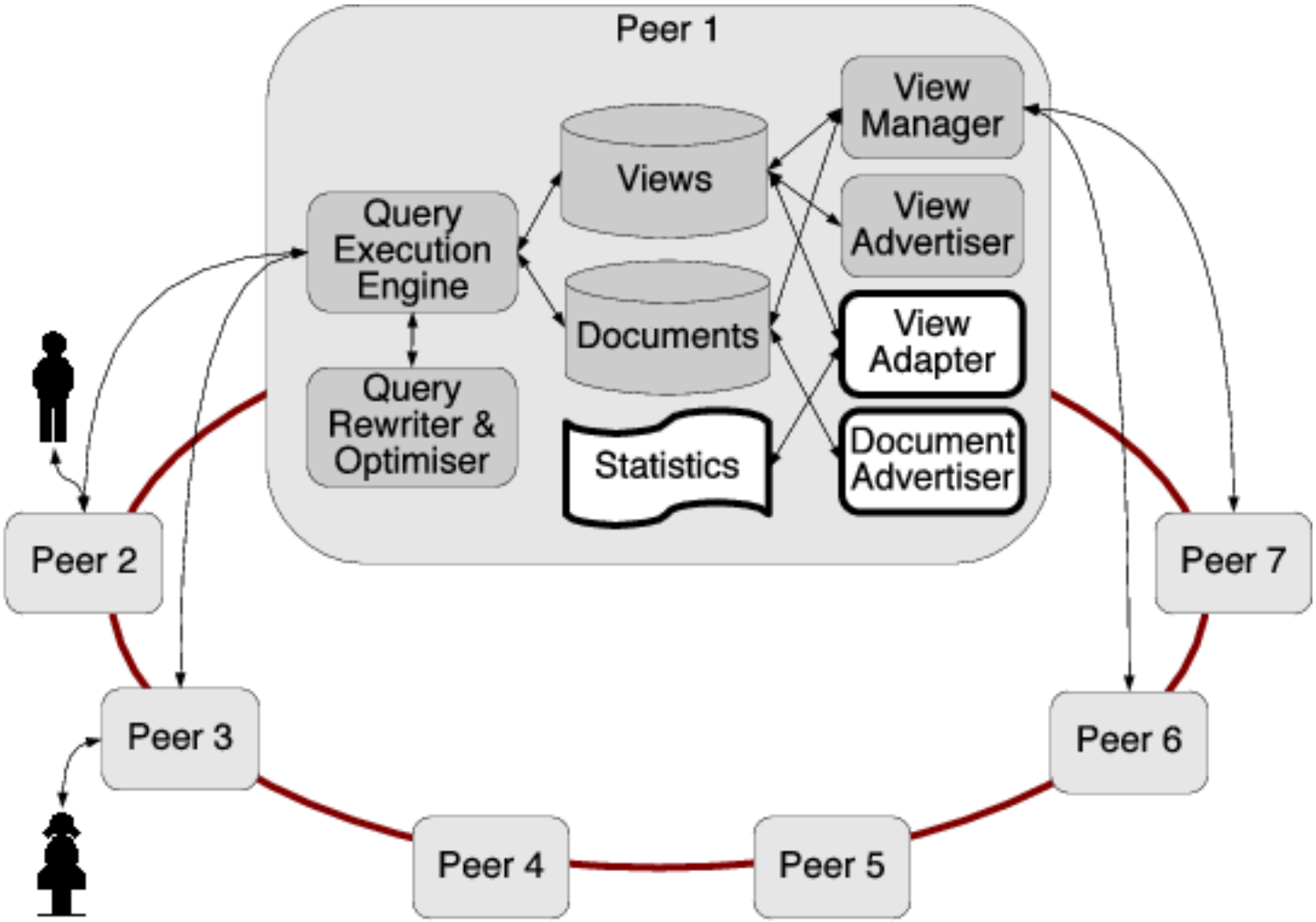
LiquidXML

Self-adaptive views for ViP2P

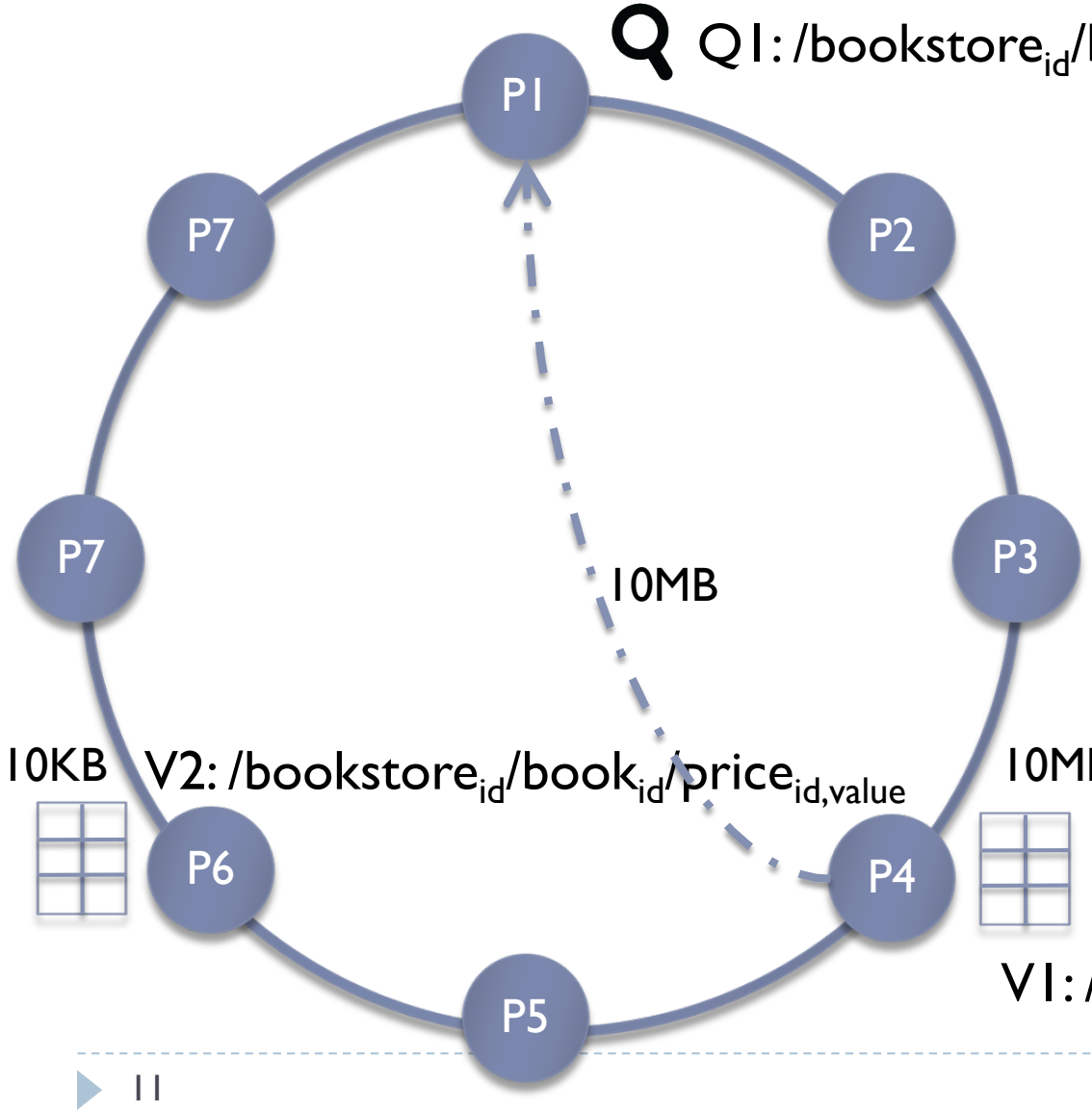
From static to liquid views

- ViP2P drawbacks:
  - Views do not adapt to the query & data workload
  - Users may not be willing to explicitly declare views
  - Users cannot predict all the possible queries
    - Although the data is out there, views might not have been declared
      - Missing views can lead to unanswered queries
- LiquidXML:
  - All queries can be answered (even without views)
  - Views are built according to the query & data workload

LiquidXML Architecture



From static to liquid views cont.



- Available views are V1 & V2
- Query Q1 arrives at Peer1
- Only V1 can answer the query and it will cost 10MBs of transfer
- What if P1 decides to materialize:

V3: //bookstore$_{id}$/book$_{id}$/author$_{id,value}$

V1: /bookstore$_{-d}$/book$_{content}$

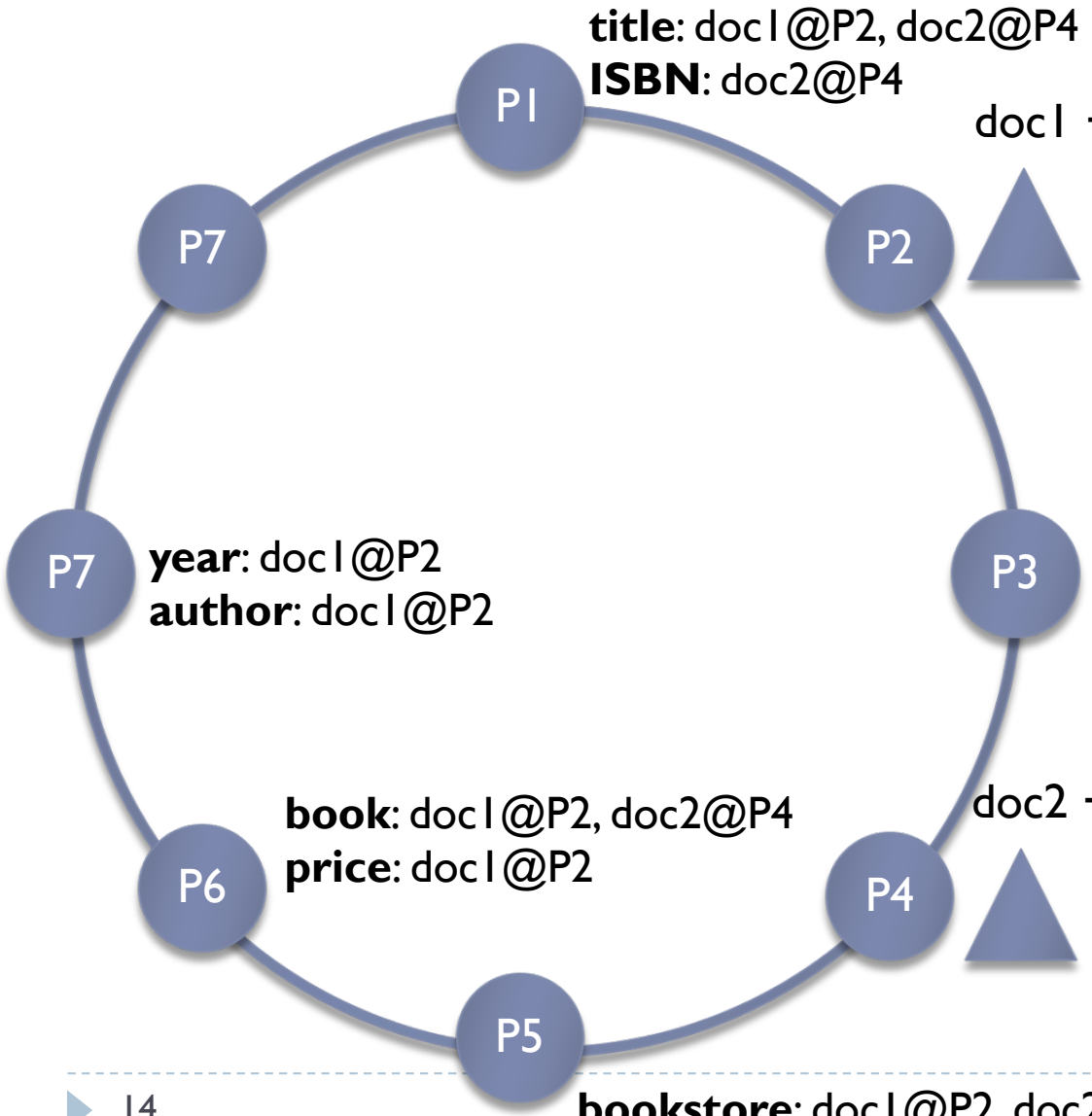
LiquidXML: **ingredients**

- Peers:
  - Have a limited *space budget* for storing views
  - Keep query statistics (frequency)
  - Index document synopses for published documents:
    - Allows to make selectivity estimations
- Baseline views
  - Document-level views (allow to answer all queries)
- Adaptive views
  - Peers materialize views trying to reduce query response time
  - Views evolve with time

LiquidXML: Document level views

- The network starts without any views
- Baseline (document-level) views
  - For each tag *t*, index the documents containing it in the DHT

Document-level views



title: doc1@P2, doc2@P4
ISBN: doc2@P4

doc1 →

```
<?xml version="1.0" encoding="UTF-8">
<bookstore>
<book>
<title>Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
</book>
</bookstore>
```

year: doc1@P2
author: doc1@P2

book: doc1@P2, doc2@P4
price: doc1@P2

doc2 →

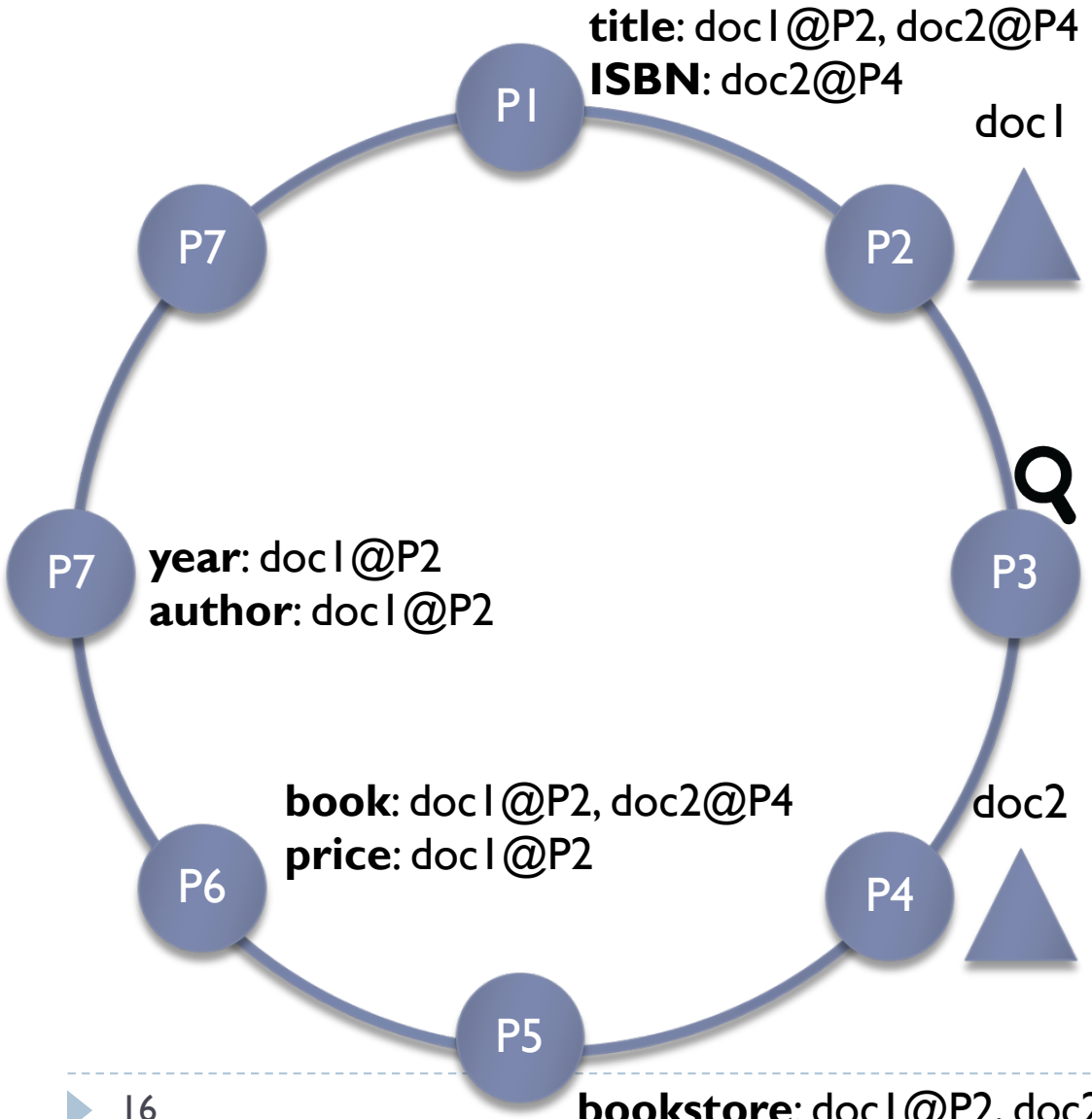
```
<?xml version="1.0" encoding="UTF-8">
<bookstore>
<book>
<title>Everyday Italian</title>
<ISBN>2FSA2412</ISBN>
</book>
</bookstore>
```

bookstore: doc1@P2, doc2@P4

Query answering with document-level views

- All queries can be answered using baseline document-level views
  - By locating relevant documents and
  - Shipping the query to the respective peers
  - Merging the results received

Query answering with document-level views



Q1: /bookstore$_{id}$/book$_{id/}$/ISBN$_{id,value}$

bookstore: doc1@P2, **doc2@P4**
book: doc1@P2, **doc2@P4**
ISBN: **doc2@P4**

Ship the query to **P4** and get results

LiquidXML

Adapting to the query workload

Adapting to the query workload : **the recipe**

1. Collect queries keeping statistics for an amount of time
2. *Find candidate views
    1. Subtrees that are "frequent" in the query workload
3. For each candidate view calculate:
    1. its **cost** (in terms of space)
    2. its **benefit** in case they are materialized
       (according to the query workload)
4. Materialize the set of candidate views that:
    1. fill the available space budget and
    2. achieve the maximum benefit

*work in progress

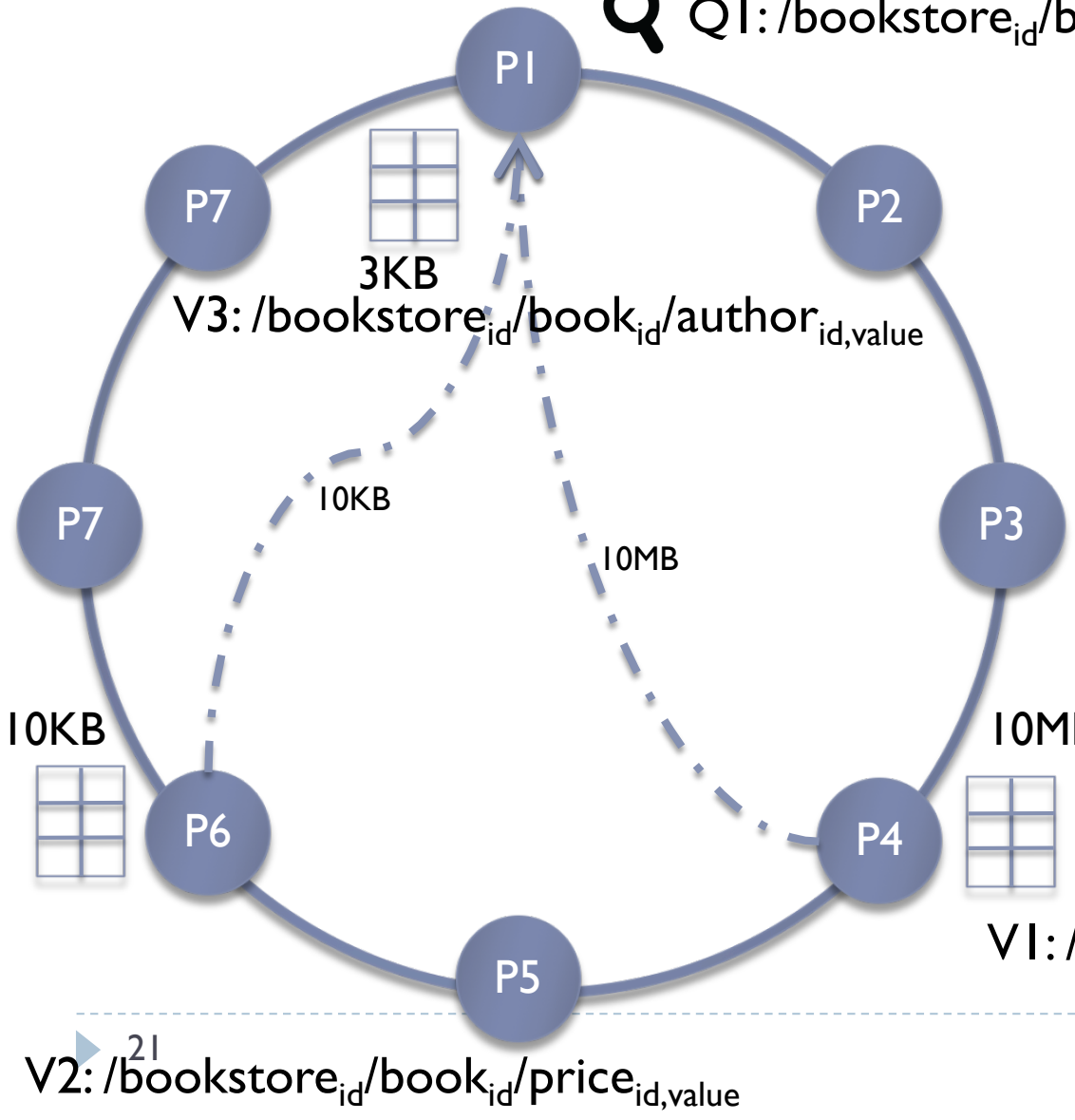
View cost: size estimation

- Synopses of all published documents are indexed in the DHT
- For estimating the size of a view **v**:
  - Locate the documents that contribute to **v**
    - Using document-level views
  - Retrieve their synopses
  - Calculate each document's **(d_i)** contribution to the view ***v*** ($|v(d_i)|$)
  - The total size of **v (|v|)** is the sum of contributions of individual documents

View benefit estimation

- If a candidate view is materialized it changes the rewriting of queries
- According to the query workload we can calculate the benefit of materializing a candidate view
- Example…

Cost & Benefit of materializing a view



Q1: /bookstore$_{id}$/book/$_{id}$[price$_{id,value}$, author$_{id,value}$]

- Available views are V1 & V2
- Query Q1 arrives at Peer1
- Only V1 can answer the query and it will cost 1MB of transfer
- Answering Q1 would cost 10MBs of transfer + computation
- If P1 decides to materialize V3
- Answering Q1 would require 10KBs of data transfer

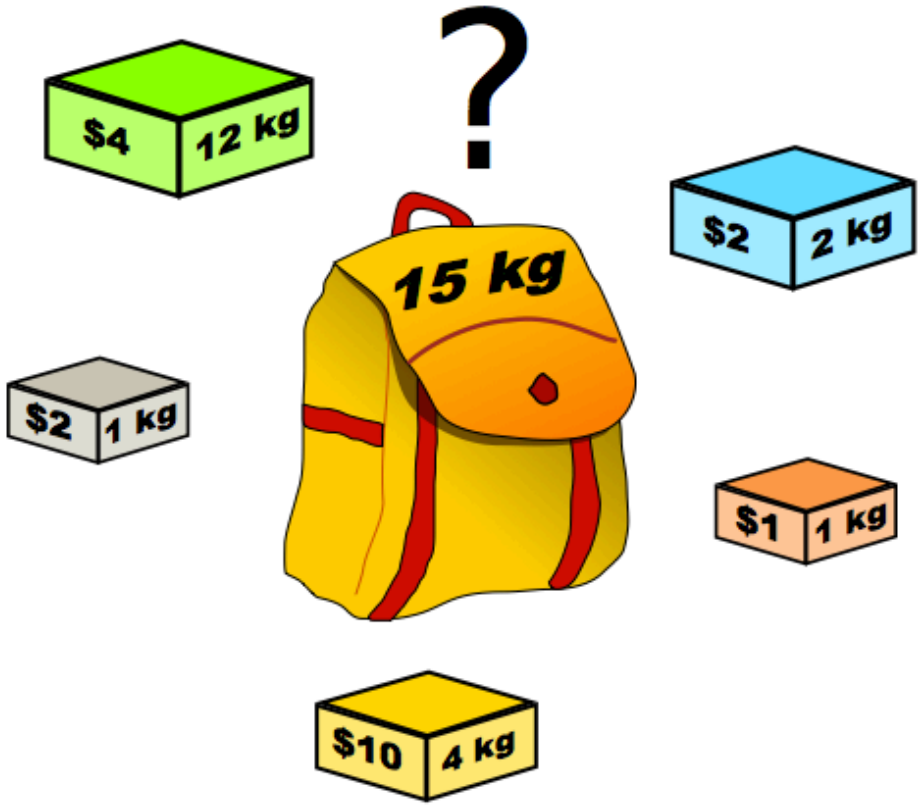
BENEFIT: ~ 9.99 MBs

COST: 3KB on P1

V1: /bookstore$_{-d}$/book$_{content}$

Filling the space budget with views

- Peers have to fill their space budget with views
- Cost and benefit of candidates is calculated
- Which is the best set of views to fill their budget?
  - Knapsack problem



$ → view benefit
kg → view cost

Questions comments

Backup

Details not mentioned

- Space budget is divided in three regions:
  - Compulsory
    - Used for storing document-level views
  - Selfish
    - Used for storing views that speed-up the execution of own queries
  - Collaborative
    - Used for storing views that speed-up the execution of others' queries