Partitioning-based Distributed RDF Graph Management

Peng Peng
Hunan University, China
hnu16pp@hnu.edu.cn

Outline

1. Background
2. Partitioning-based Systems
3. Conclusions

Part 1

Background

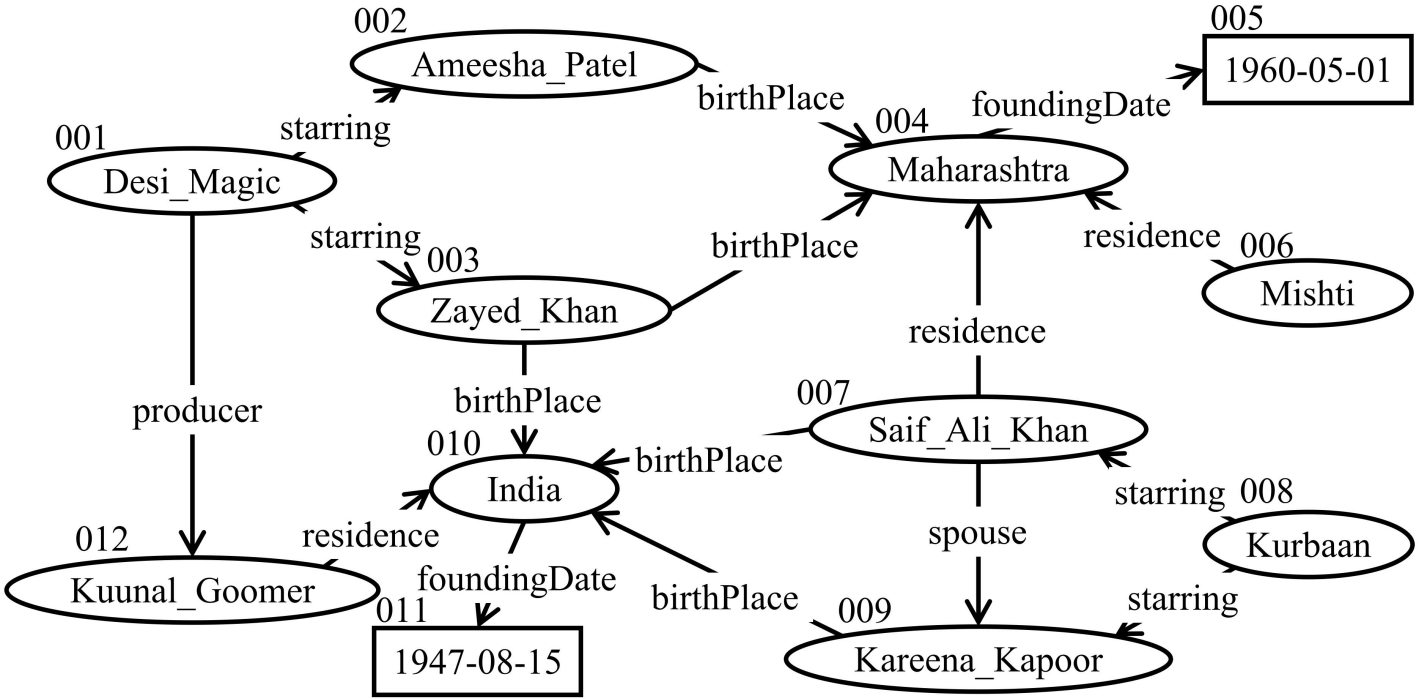
RDF Introduction

Resource Description Framework

- It is a data model proposed widely used for knowledge bases
- Everything is an uniquely named resource
- Properties of resources can be defined
- Relationships with other resources can be defined

RDF Graph

An RDF dataset can be represented as a graph

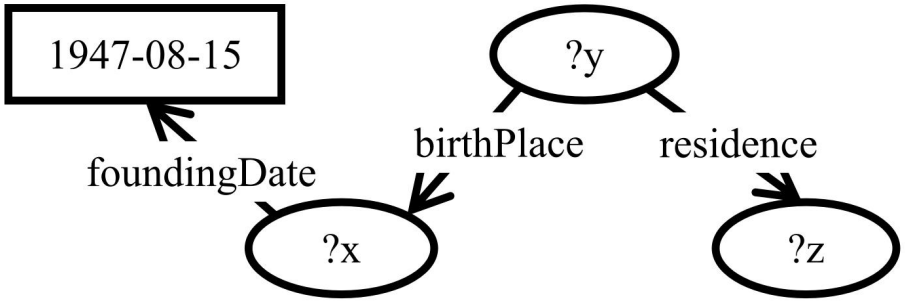


RDF Query Model

Query Model - ***S***PARQL ***P***rotocol ***a***nd ***R***DF ***Q***uery ***L***anguage

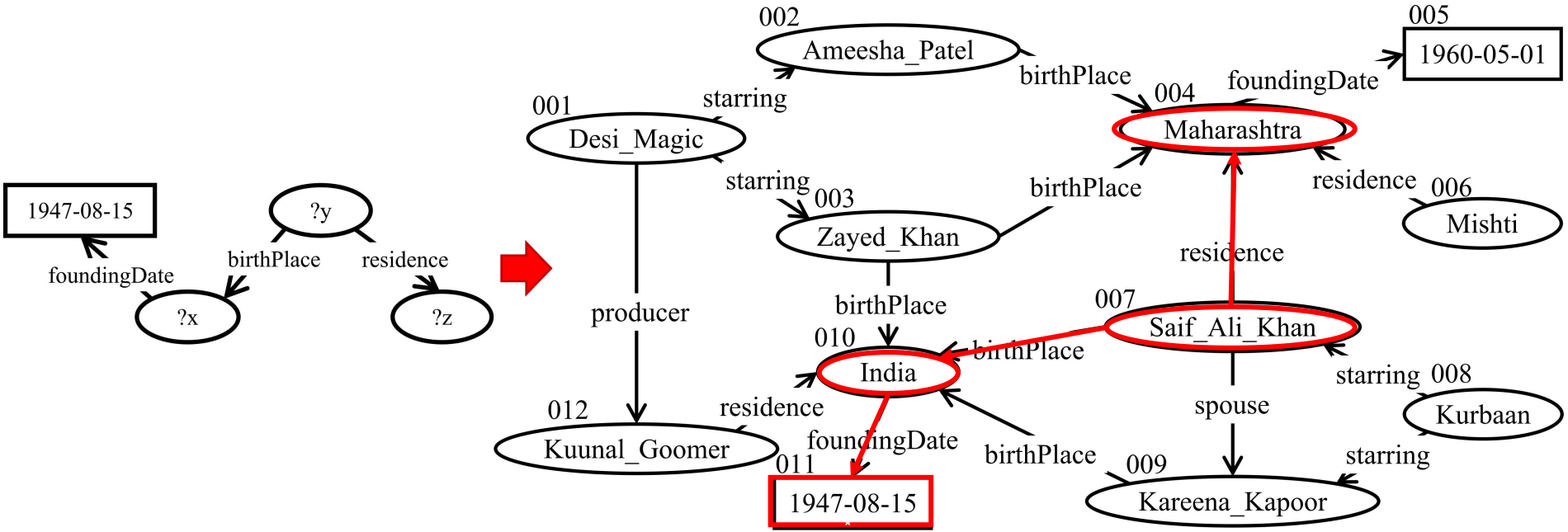
A SPARQL query is a set of triple patterns with variables

```
Select ?x where {
  ?y residence        ?z .
  ?y birthPlace       ?x .
  ?x foundingDate     1947-08-15 . }
```



RDF Query Model

Answering SPARQL query = subgraph matching using homomorphism



Large RDF Graphs

Now, RDF datasets become larger and larger

Max-Planck-Institute

284 million triples

Metaweb Company
acquired by Google in 2010

2.4 billion triples

Leipzig University
University of Mannheim
OpenLink Software

9.5 billion triples

It is important to design a distributed RDF system to manage the large RDF dataset.

Part 2

Partitioning-based Systems

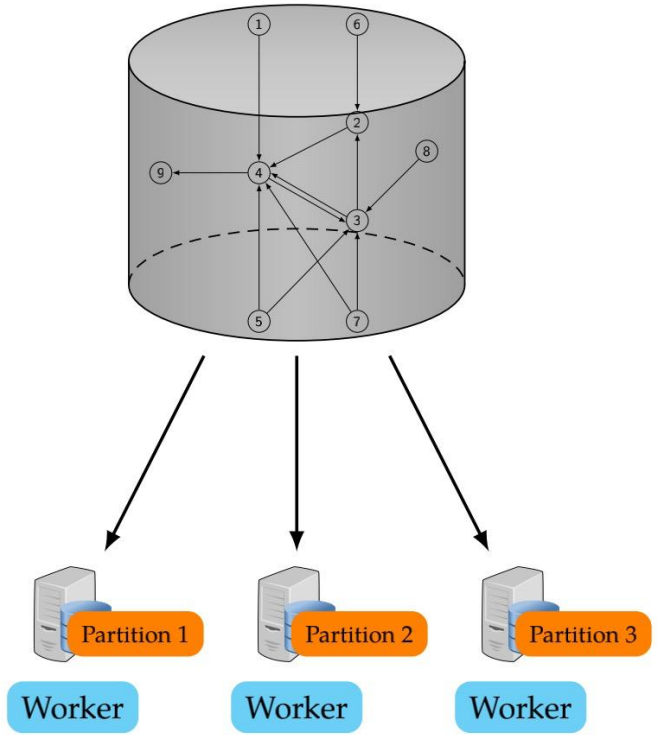
Partitioning-based Systems

- **Workload-agnostic approaches**
- Workload-aware approaches

Distributed RDF System Architecture

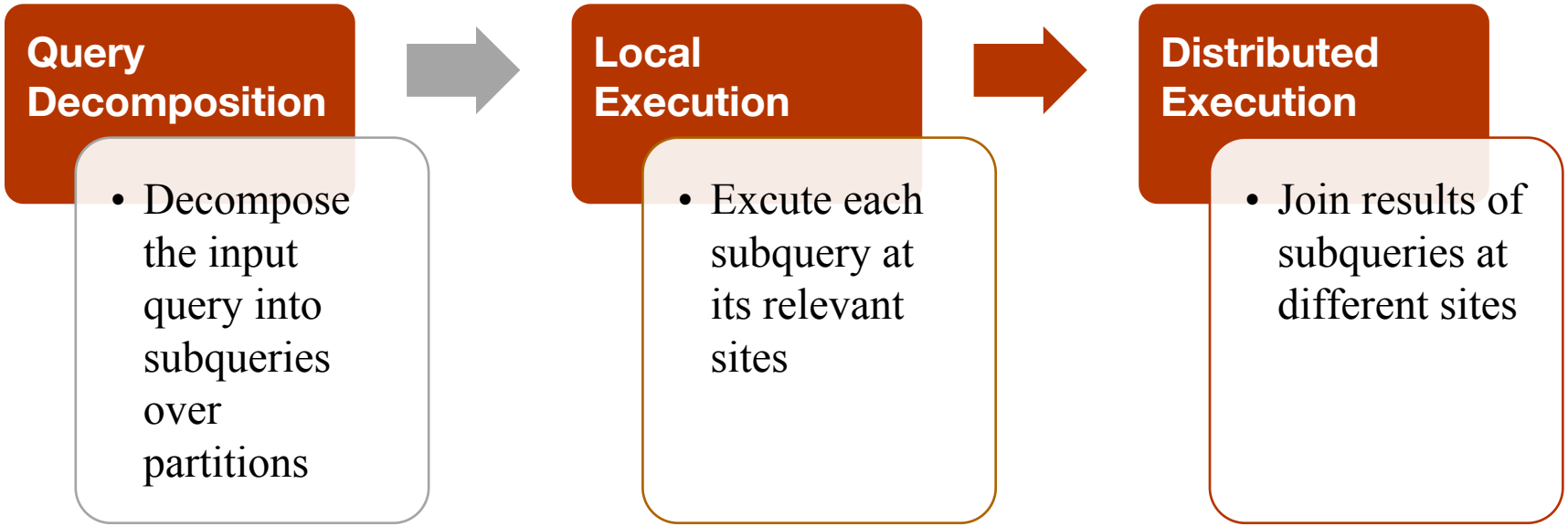
Partitioning–based Architecture

- An RDF graph is partitioned through different sites
- Objective: Parallelize query processing with as little inter-site communication as possible.



Distributed Query Execution Model

Partition the data and the query in such a way to minimize the inter-partition joins



Related Work

RDF graph partitioning approaches can be classified into three types:

- **Vertex-disjoint:** Put each vertex in one partition
  - Existing methods' objective is to minimize edge-cuts, rather than minimize the inter-partition joins
- **Edge-disjoint:** Put each edge in one partition
  - Existing methods are widely used in many cloud-based systems to prune irrelevant partitions and avoid too many scans in the cloud, but do not focus on avoiding inter-partition joins
- **Other:** Consider extra information, such as workloads

H-RDF-3X [Huang et al., VLDB 2011]

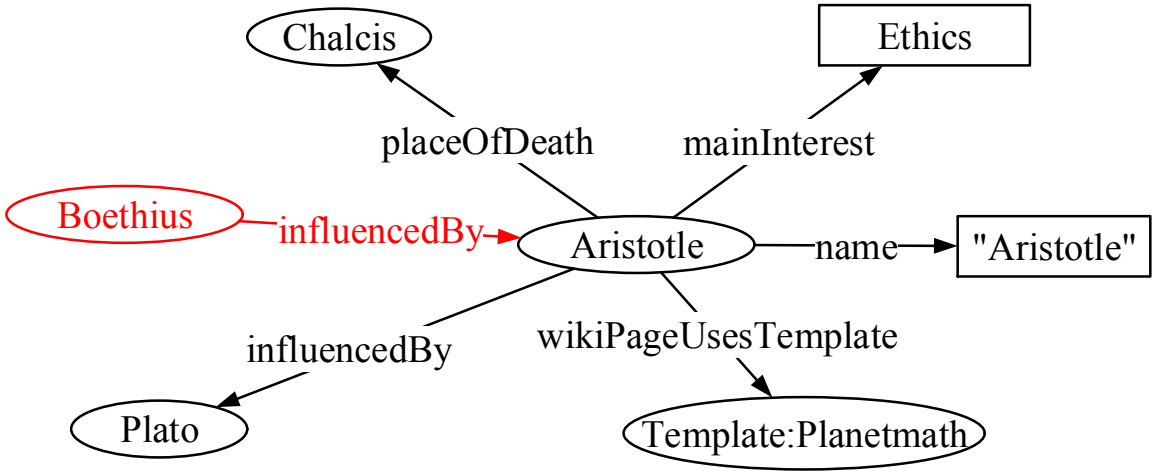
- Data is partitioned using METIS
- Replicate vertices using n-hop guarantee
- If the radius of Q is not larger than n, Q can be locally evaluated n each site
- If the radius of Q is larger than n, Q is decomposed into several subqueries Qi that can be independently evaluated; then their results are joined

SHAPE [Lee et al., VLDB 2013]

- Generate triple groups based on common subject or object
- Semantic hashing triple groups: URI references usually have a path hierarchy and URI references having a common ancestor are often connected together, so SHAPE places such URI references (vertices) in the same partition
- Allow some data replication between different partitions

Example Triple Groups

- Black subgraph is a subject-based triple group
- Red subgraph is an object-based triple group
- The whole graph is a subject-object-based triple group



Characteristics of RDF Graphs

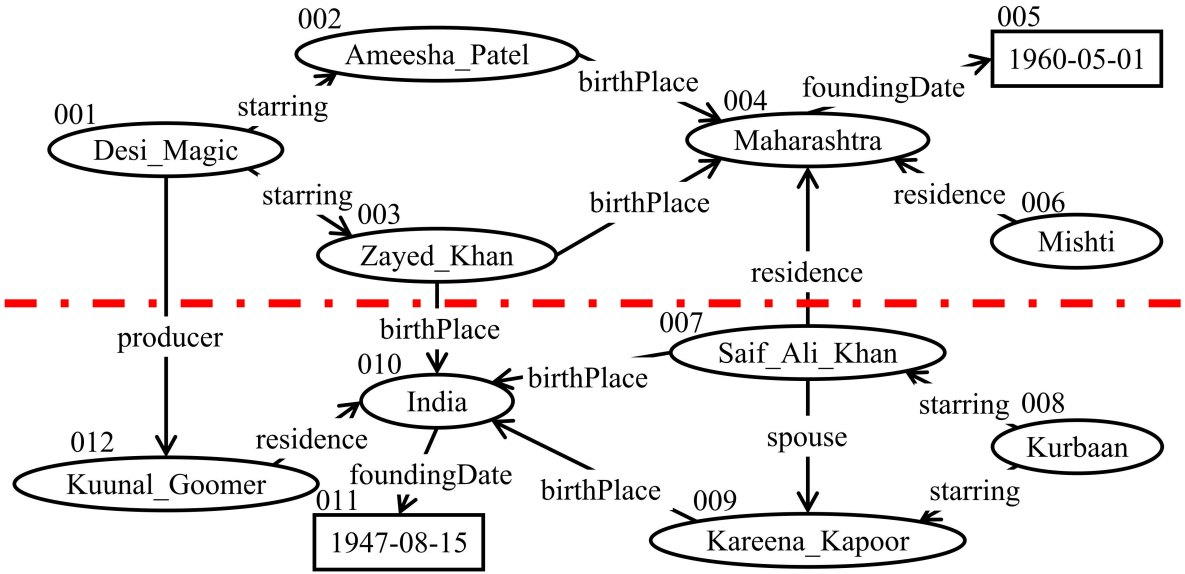
- LUBM is a benchmark that adopts an ontology for the university domain, and can generate synthetic RDF datasets of arbitrary sizes
- The URI references in LUBM are URLs with http as their schema, such as http://www.Department1.University2.edu/FullProfessor2/Publication14
- Then, all entities with the similar URI hierarchy are put into the same partition

Minimum Property-Cut [Peng et al., ICDE 2022]

- We propose a novel vertex-disjointed RDF Graph partitioning scheme, minimum property-cut (MPC), for distributed SPARQL query evaluation

Internal & Crossing Properties

- A property is called a **crossing property** if and only if there is at least one crossing edge with that property; otherwise, it is called an **internal property**

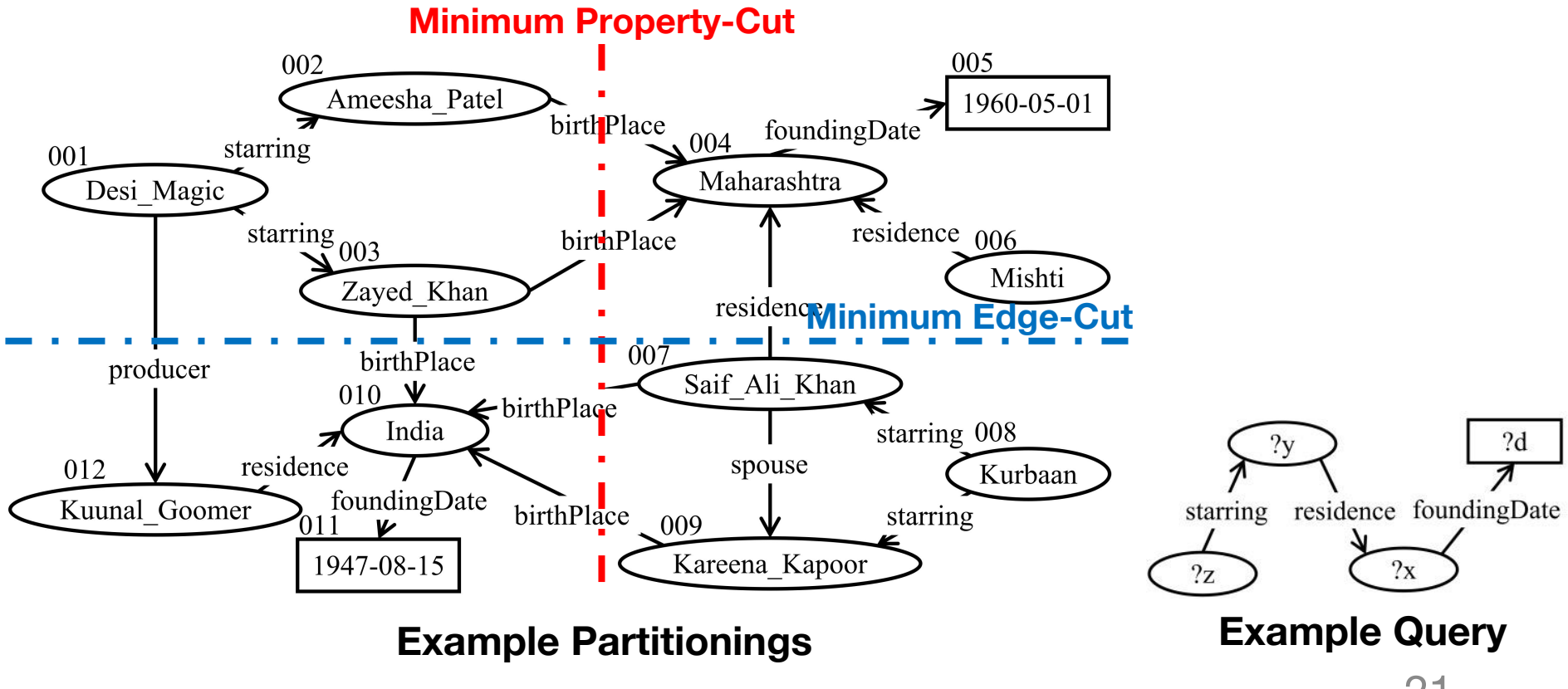


Motivations

- The main performance bottleneck in distributed SPARQL execution is inter-partition joins
  - If all properties in a query are internal → execute independently over each partition without inter-partition join

- **Objective**: Partition to maximize the number of internal properties
  - May increase edge-cuts, but minimizes number of unique property-cuts

Example for Comparison of Different Partitionings



Problem Definition

- Given an RDF graph G and a positive integer k, the **minimum property-cut (MPC)** partitioning of G is a partitioning $F = \{F_1, F_2, ..., F_k\}$ such that
  1. the number of crossing properties $|L_{cross}|$ is minimized (i.e. the number of internal properties $|L_{in}|$ is maximized);
  2. the size of F_i (i.e. $|V_i|$) is not larger than $(1 + \varepsilon) \times |V|/k$ for each F_i, where ε is a user-defined maximum imbalance ratio of a partitioning (i.e., how much difference can be in the relative sizes of partitions).

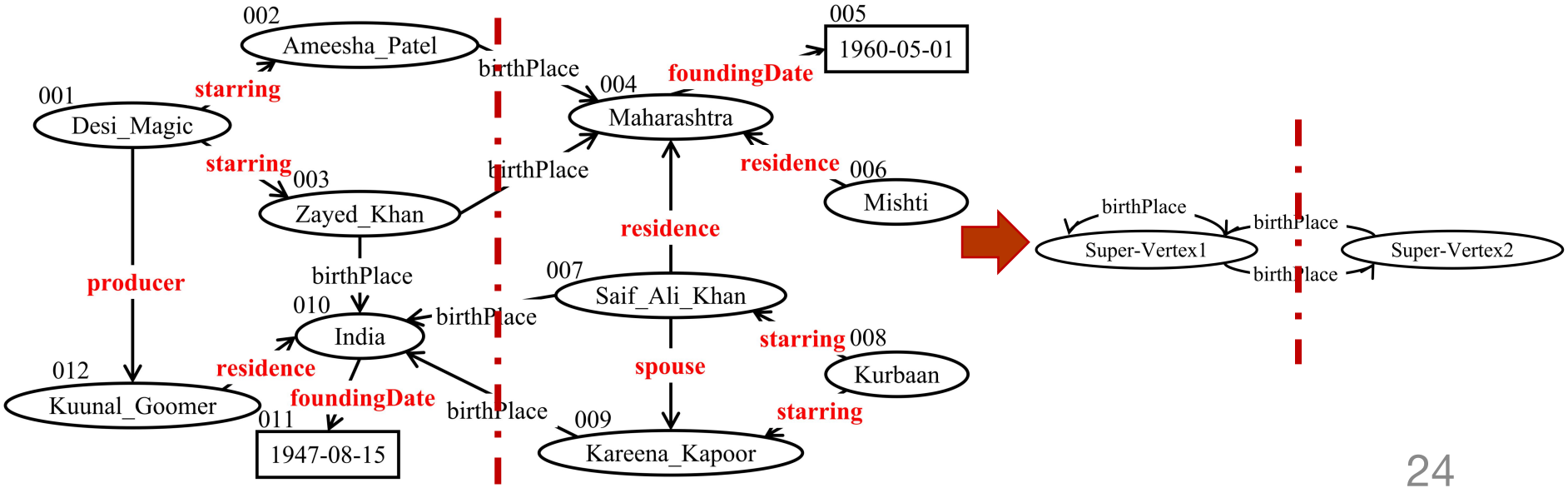
Hardness of MPC

Theorem: The *MPC* partitioning problem is NP-complete.

Proof: We reduce the NP-complete *minimum edge-cut* problem to the *MPC* problem by assigning each edge in an instance of *minimum edge-cut* graph partitioning with a distinct property

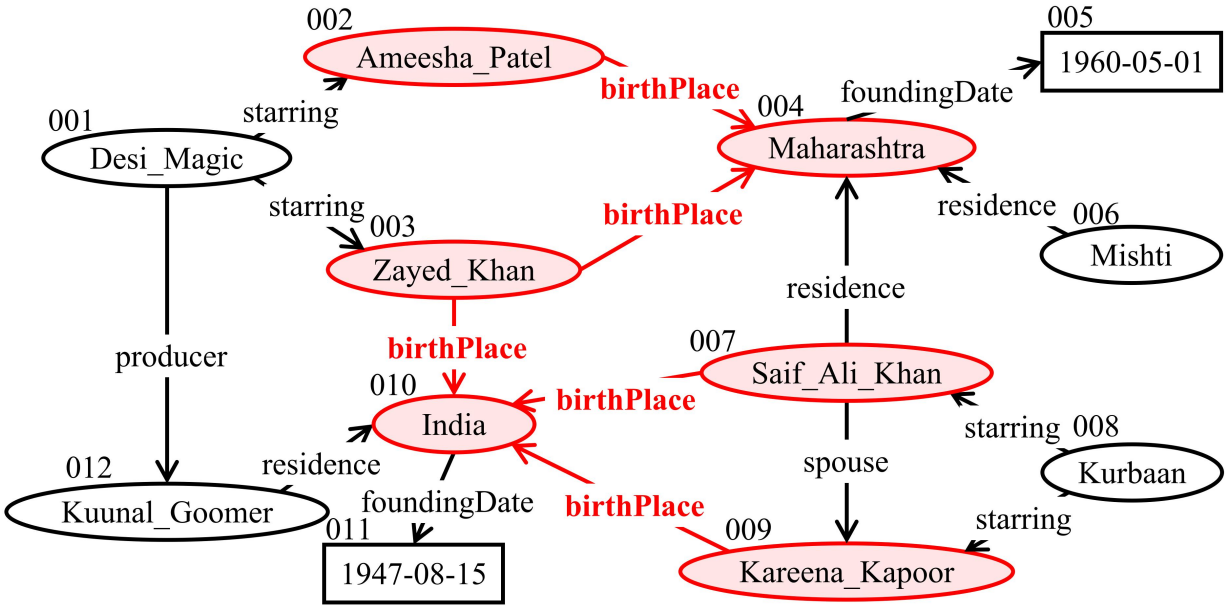
Framework of Minimize Property-Cuts

Selecting Internal Properties → Coarsening → Partitioning Coarsened Graph → Uncoarsening



Constraint of Internal Properties

If a property p is an internal property, any two vertices in a weakly connected component of p's induced graph should be in one partition.



If we want *birthPlace* to be an internal property, all the six vertices should be in one partition

Selecting Internal Property Cost

Given a set of properties *L'*, the cost of selecting them as internal properties is defined as the size of the largest WCC in the property induced subgraph *G[L']*

$$Cost(L') = \max_{c \in WCC(G[L'])} |c|$$

where *c* is a weakly connected component in *WCC(G[L']])* and |c| denotes the number of vertices in *c*.

Internal Property Selection Algorithm

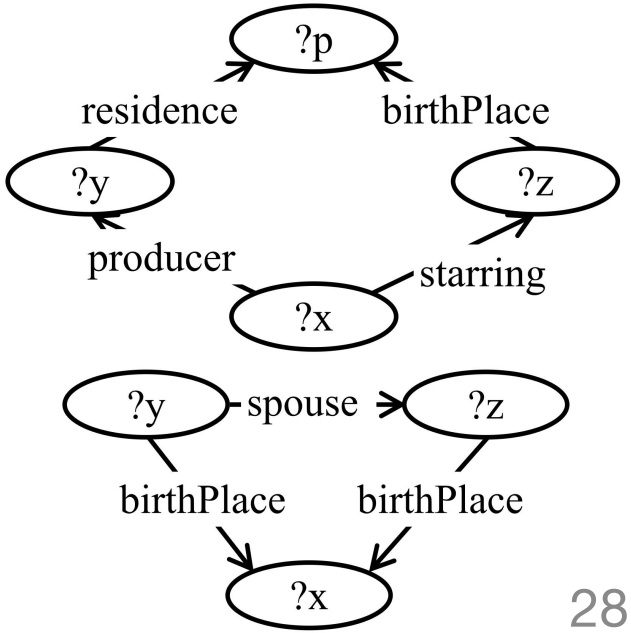
1. First, we initialize an empty set, L_{in}, for internal properties, and compute the WCCs of the property-induced subgraphs for each property
2. We iteratively select a property p that minimizes $Cost(L_{in} \cup \{p\})$ and inserted it into L_{in}. These steps are repeated until no more property can be selected

To compute and merge the WCCs, we propose to use the disjoint-set forest data structure

Independently Executable Queries

Independently executable queries (IEQs) are those that are "local" to a partition and can be executed without a join with another partition

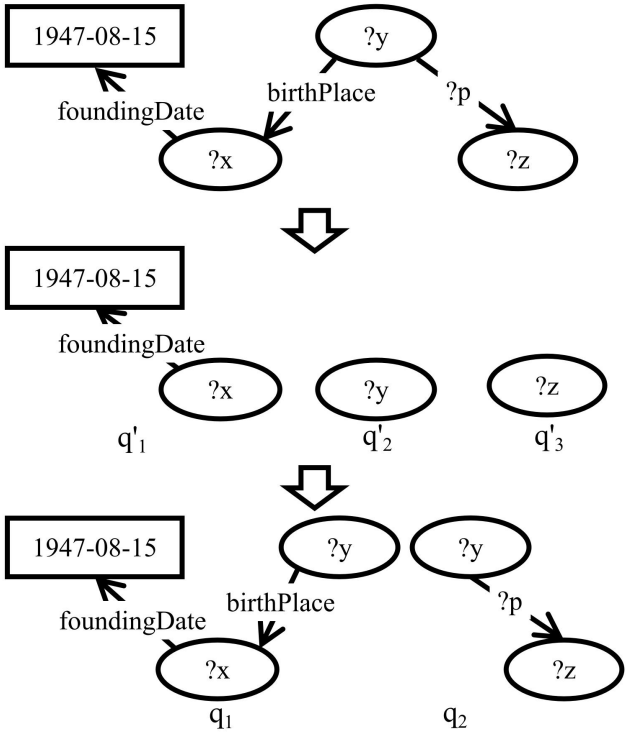
- Internal IEQs: Queries not containing any crossing properties
- Extended IEQs: Queries connected when crossing property edges are removed



Query Decomposition

Non-IEQ: We decompose into a set of IEQs and join the results

- Remove crossing property edges & edges with variables
- Form subqueries
- Add crossing property edges & edges with variables to subqueries



Query Execution

- When a non-IEQ Q is received, it is decomposed into a set of subqueries $\{q_1, q_2, ..., q_y\}$, and they are sent to each partition for independent execution

- Then, the subqueries' matches are joined to obtain the final result

Setting

- **Datasets:**

Dataset	#Entities	#Triples	#Properties
LUBM 100M	17,473,142	106,909,064	18
LUBM 1B	173,891,493	1,069,331,221	18
LUBM 10B	1,737,718,408	10,682,013,023	18
WatDiv 100M	5,212,745	108,997,714	86
WatDiv 1B	52,120,745	1,099,208,068	86
WatDiv 10B	521,200,745	10,987,996,562	86
YAGO2	21,073,153	284,417,966	98
Bio2RDF	804,671,979	4,426,591,829	1,581
DBpedia	139,493,254	1,111,481,066	124,034
LGD	311,153,753	1,292,933,812	33,348

- **Competitors:** Subject_Hash, METIS and VP
- **Environment:** 8 machines running Linux at Alibaba Cloud

Partitioning Quality

- **$|L_{cross}|$ & $|E^c|$:**

Datasets	MPC		Subject_Hash		METIS	
	$\|L_{cross}\|$	$\|E^c\|$	$\|L_{cross}\|$	$\|E^c\|$	$\|L_{cross}\|$	$\|E^c\|$
LUBM	5	29,971,560	14	62,377,786	13	21,853,766
WatDiv	17	95,497,642	31	95,786,527	31	58,100,544
YAGO2	5	128,758,514	45	142,274,454	43	58,912,138
Bio2RDF	36	2,044,500,633	398	2,184,075,117	-*	-
DBpedia	64	504,897,118	33,966	681,734,289	17,807	171,944,344
LGD	6	518,035,967	2,012	676,620,154	2,010	296,443,817

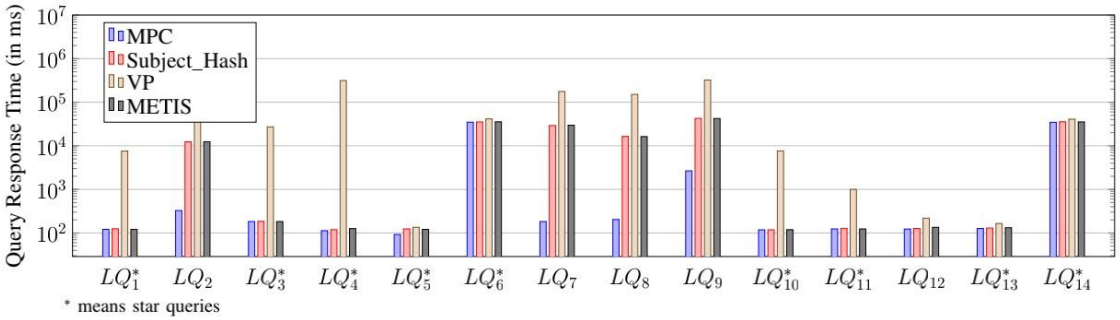
* - means that data size is beyond the capacity of METIS.

- **Percentage of IEQs:**

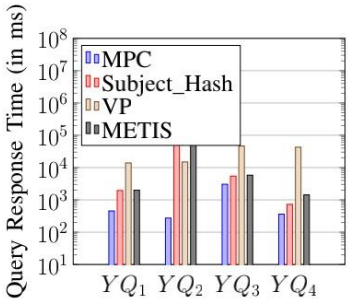
	MPC	VP	Subject_Hash / METIS	Subject_Hash+	METIS+
LUBM	100%	28.57%	71.43%	71.43%	71.43%
WatDiv	60%	0%	50%	50%	50%
YAGO2	100%	0%	0%	0%	0%
Bio2RDF	100%	40%	80%	80%	80%
DBpedia	75.19%	24.25%	46.87%	51.87%	51.90%
LGD	99.95%	83.51%	96.95%	96.98%	96.98%

Online Performance Comparison

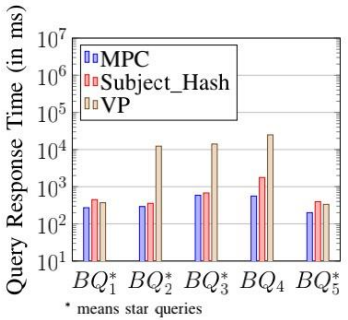
- **On Benchmark Queries:**



(a) LUBM

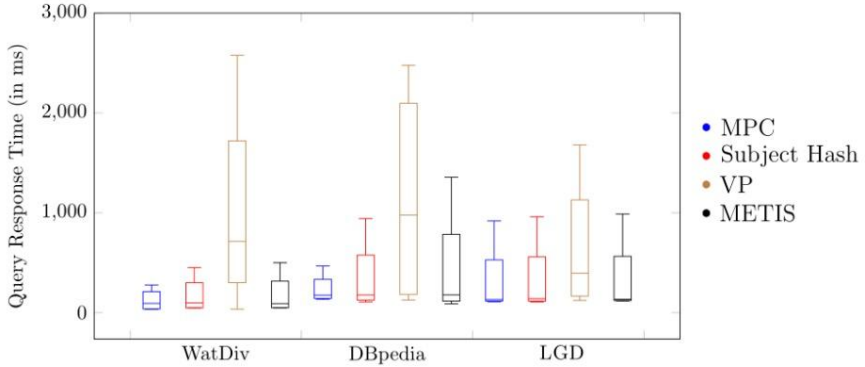


(b) YAGO2



(c) Bio2RDF

- **On Real Query Logs:**

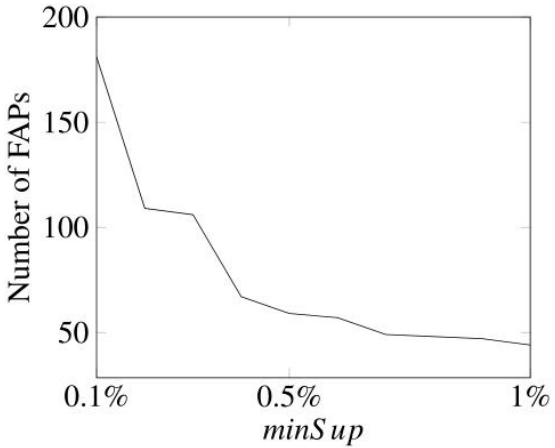


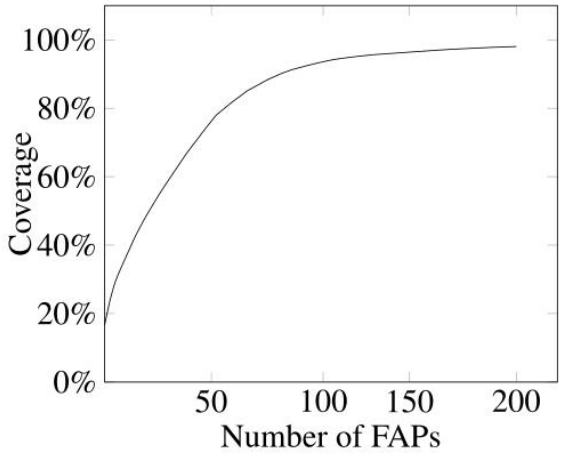
Partitioning-based Systems

- Workload-agnostic approaches
- **Workload-aware approaches**

Why Query Workload Matters ?

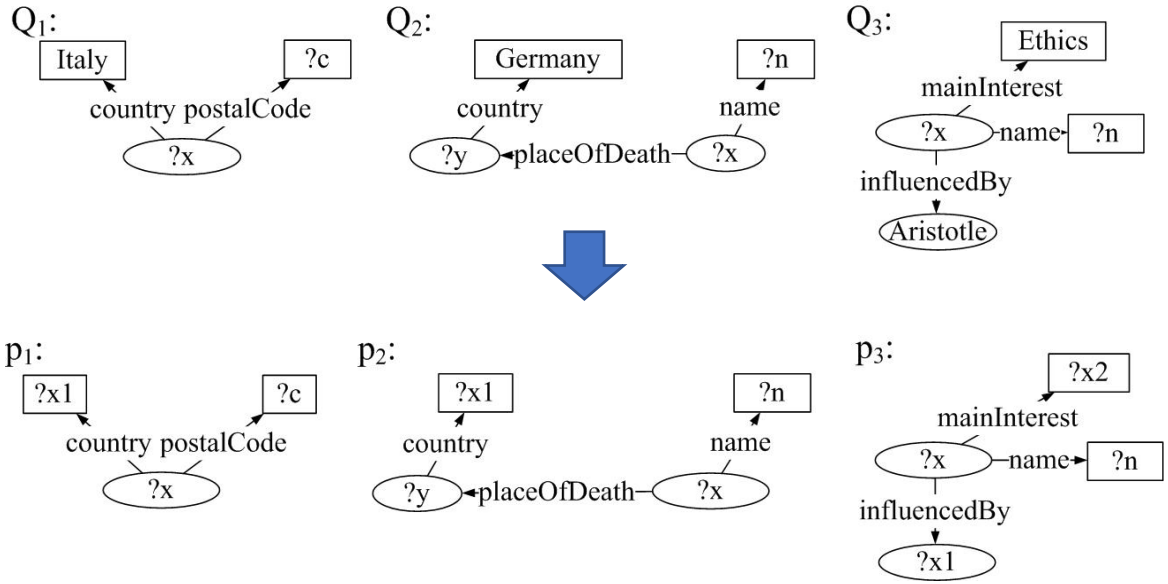
- A few pattern can cover most queries in the real query log





Normalization of the Workload

- Before frequent access pattern mining, we normalize all SPARQL queries in the workload to avoid overfitting



Access Frequency

- The **access frequency** of a pattern *p* is the number of queries in a workload $\mathcal{Q}$,, where a pattern *p* is a subgraph

- A pattern *p* is **frequent access pattern** if its access frequency is no less than a threshold, *minSup*.

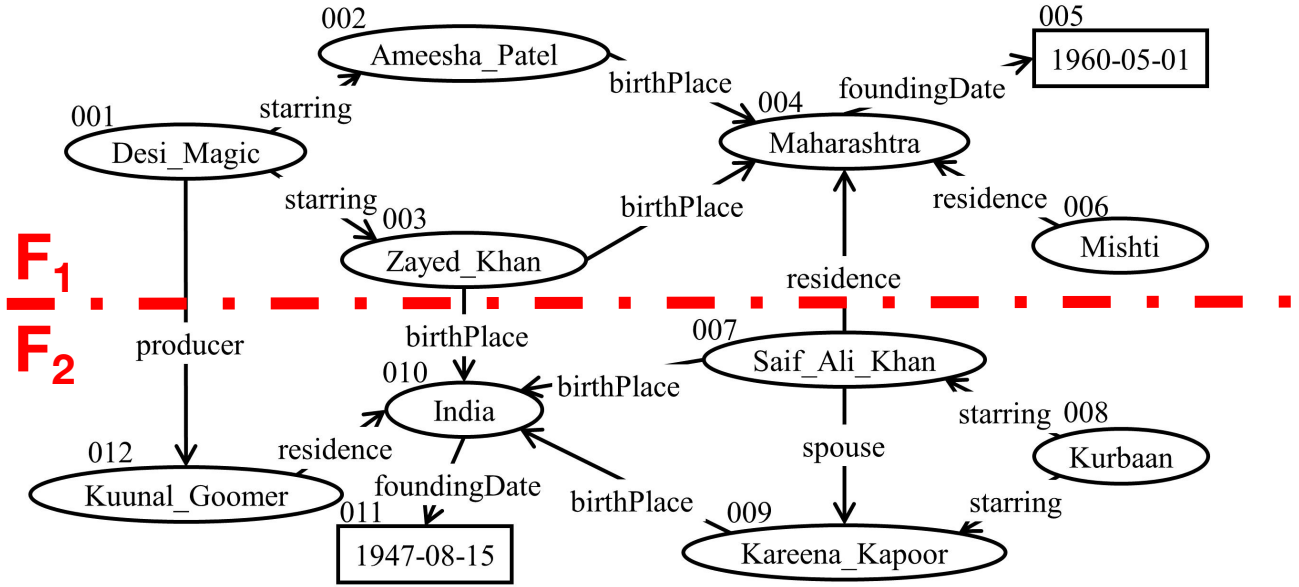
- Most workload-aware appoarches are based on frequent access patterns

WARP [Hose et al., ICDE Workshops 2013]

- WARP proposes a distributed SPARQL engine that combines a graph partitioning technique with workload-aware replication of triples across partitions

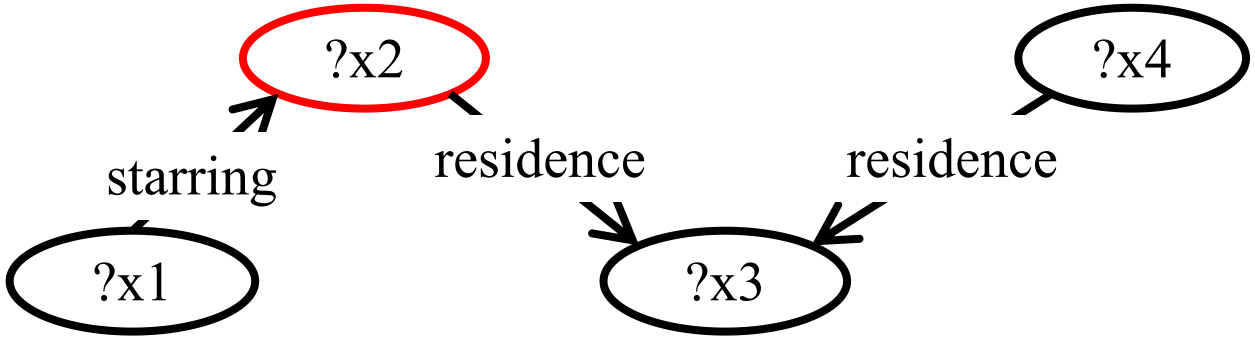
WARP

- First, WARP partitions the RDF graph using METIS



WARP

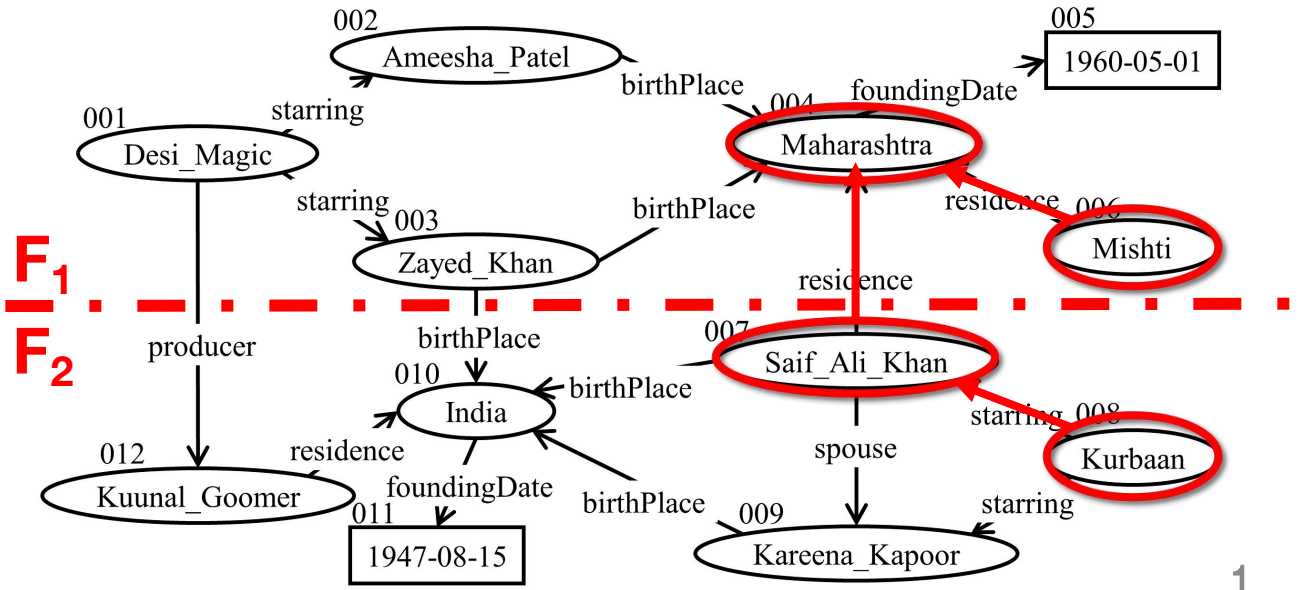
- Second, WARP find some frequent query patterns that are not independently executable and select one query vertex as *seed*



WARP

- Last, WARP replicate the matches of pattern at the partition that the mapping of the seed is at

Seed vertex ?x2 is mapped to 007, so the match is replicated at F_2



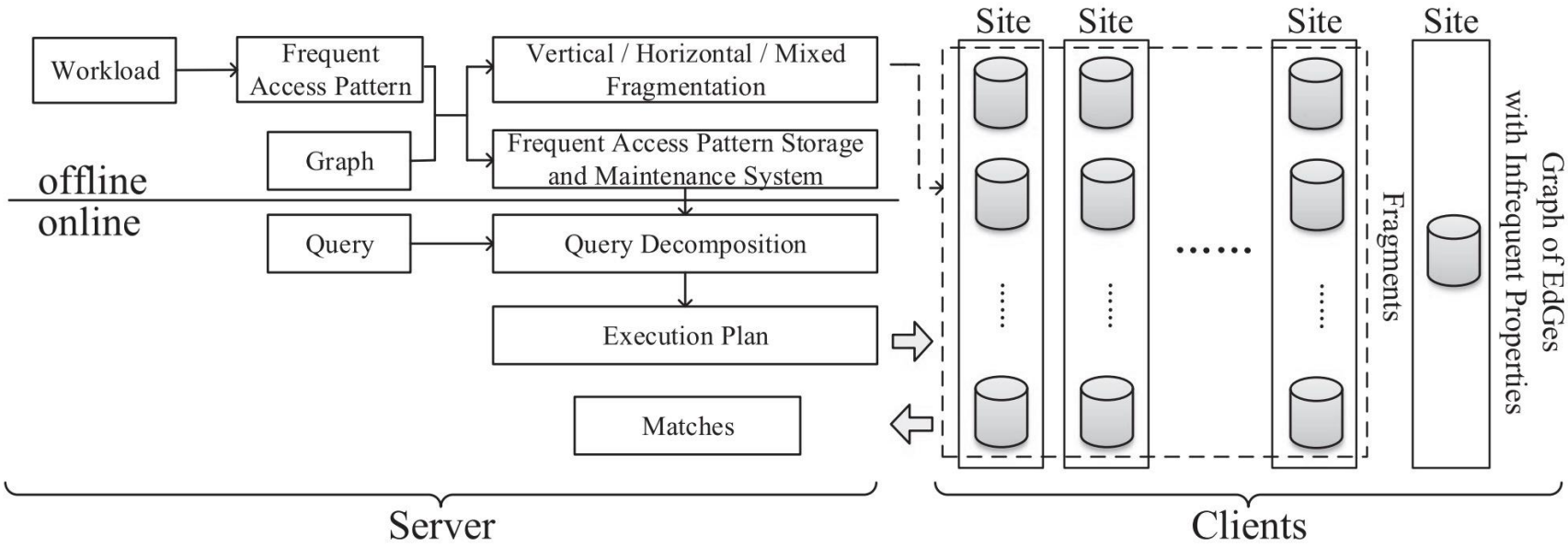
WARP

- **Advantages:** if a query is isomorphic to the frequent query pattern, there is no inter-partition join and the query performance is very high

VF, HF & MF [Peng et al., EDBT 2016, TKDE 2019]

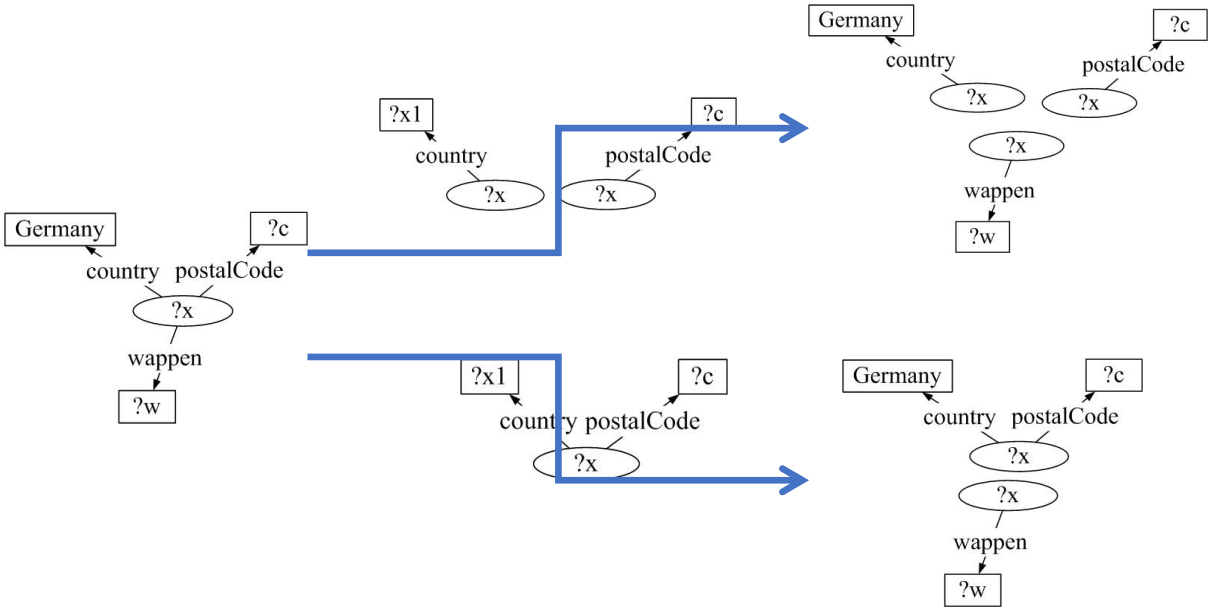
- We first mine and select frequent subgraph patterns, named **frequent access patterns**, in the query workload
- We propose three fragmentation strategies, **vertical, horizontal and mixed fragmentation,** based on these patterns and an allocation algorithm to distribute the fragments to the sites in the distributed system
- We propose a cost-aware query optimization method to evaluate a SPARQL query

Framework


Workload
Frequent Access Pattern
Graph
Vertical / Horizontal / Mixed Fragmentation
Frequent Access Pattern Storage and Maintenance System
offline
online
Query
Query Decomposition
Execution Plan
Matches
Server
Site
Site
Site
Site
Fragments
Graph of EdGes with Infrequent Properties
Clients

Frequent Access Pattern Selection

- Not all patterns need to be selected
- Larger patterns have large benefits



Size-increasing Benefit

- The benefit of selecting frequent access pattern p for hitting the query Q is

$$Benefit(p,Q) = |E(p)| \times use(Q,p)$$

where use(Q,p)=1 if pattern p is a subgraph of Q

- Given a set of frequent access patterns P over the workload $\mathcal{Q}$, its benefit is

$$Benefit(P,\mathcal{Q}) = \sum_{Q_i \in \mathcal{Q}, i=1,\ldots,r} \max_{p \in P}\{Benefit(p,Q_i)\}$$

Storage Constraint

- We normalize a storage constraint to a value SC

$$\sum_{p \in P'} |E([\![p]\!]_G)| \leq SC$$

where $E([\![p]\!]_G)$ means the size of all subgraph matches in G homomorphic to p

Problem Hardness

- Finding a set of frequent access patterns with the largest benefit while subject to the storage constraint is NP-hard

- Thus, we propose a heuristic algorithm which can guarantee the data integrity

Frequent Access Pattern Selection Algorithm

Select all patterns with one edge to guarantee data integrity

Greedy selection of frequent access pattern

The approximation ratio is

$$\min\{\frac{1}{\max_{p\in P}|E(p)|}, \frac{1}{2}(1-\frac{1}{e})\}$$

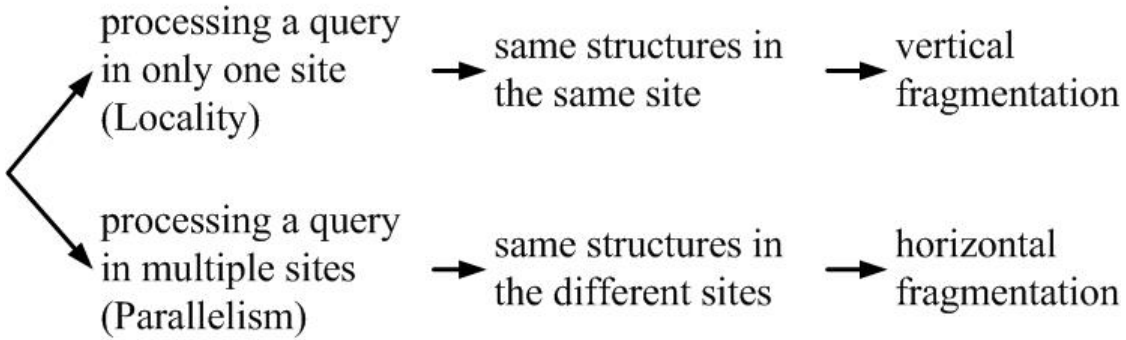
Algorithm 1: Frequent Access Pattern Selection Algorithm

Input: A set of frequent access patterns $P = \{p_1, p_2, ..., p_x\}$
Output: A set $P' \subseteq P$ to generate fragments

1 $P' \leftarrow \emptyset$;
2 $TotalSize \leftarrow 0$;
3 **for** *each* $p \in P$ *and* p *has only one edge* **do**
4 $\quad P' \leftarrow P' \cup \{p\}$;
5 $\quad P \leftarrow P - \{p\}$;
6 $\quad TotalSize \leftarrow TotalSize + |E([\![p]\!]_G)|$;
7 $P_1 \leftarrow argmax\{\frac{Benefit(\{p_i\},Q)}{|E([\![p_i]\!]_G)|} : p_i \in P, |E([\![p_i]\!]_G)| + TotalSize \leq SC \wedge |E(p)| > 1\}$;
8 $P_2 \leftarrow \emptyset$;
9 $TotalSize' \leftarrow 0$;
10 **while** $TotalSize' \leq SC - TotalSize$ **do**
11 $\quad$ Find the frequent access pattern $p' \in P - P'$ with the largest additional value of $\frac{Benefit(\{p'\}\cup P',Q)-Benefit(P',Q)}{|E([\![p']\!]_G)|}$;
12 $\quad P_2 \leftarrow P_2 \cup \{p'\}$;
13 $\quad P \leftarrow P - \{p'\}$;
14 $\quad TotalSize' \leftarrow TotalSize' + |E([\![p']\!]_G)|$;
15 **if** $Benefit(P' \cup P_1, Q) \geq Benefit(P' \cup P_2, Q)$ **then**
16 $\quad$ Return $P' \cup P_1$;
17 Return $P' \cup P_2$;

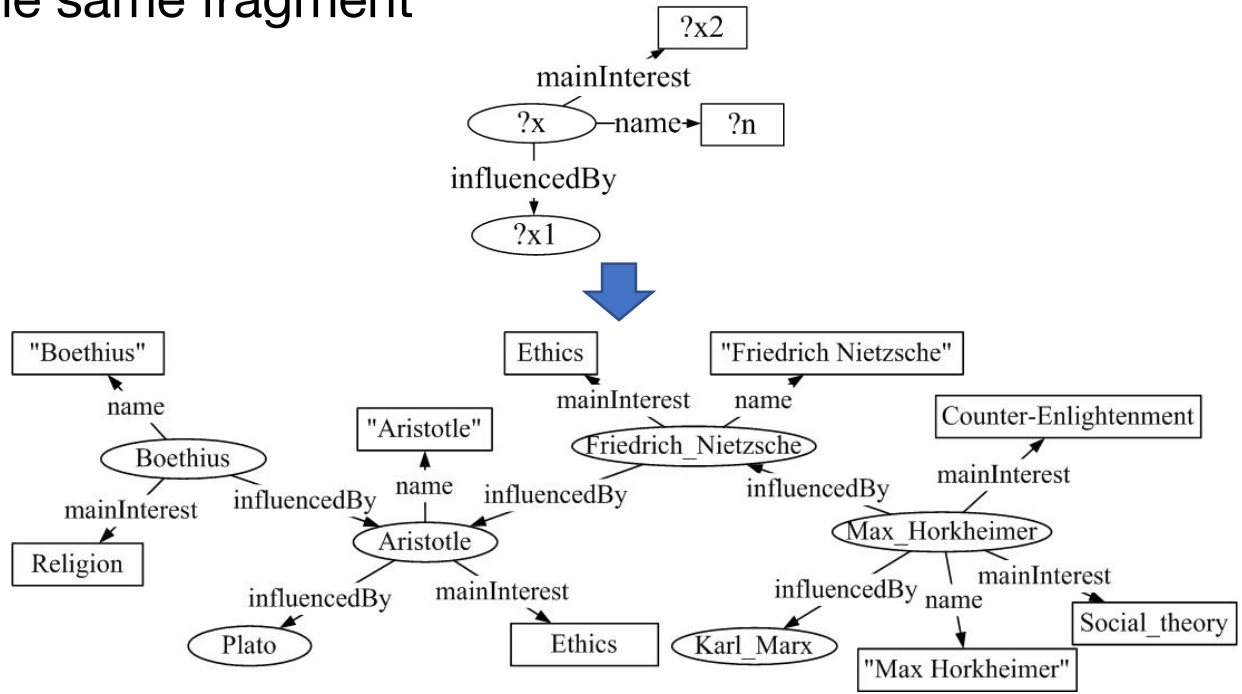
Fragmentation Framework

- In this paper, we present two fragmentation strategies: vertical and horizontal



Vertical Fragmentation

- All matches homomorphic to the same frequent access pattern into the same fragment



Horizontal Fragmentation

- We put matches of one frequent access pattern into the different fragments

- We extend the concepts of **simple predicate** and **minterm predicate** divide the RDF graph horizontally.

Structural Simple Predicate

- Given a frequent access pattern *p* with variables set $\{var_1, var_2, \ldots, var_n\}$, a **structural simple predicate** *sp* defined on Q, has the following form.

$$sp : p(var_i)\ \theta\ Value$$

where $\theta \in \{=, \neq\}$ and Value is a constant constraint for var_i chosen from a query containing *p* in Q,.

Structural Minterm Predicate

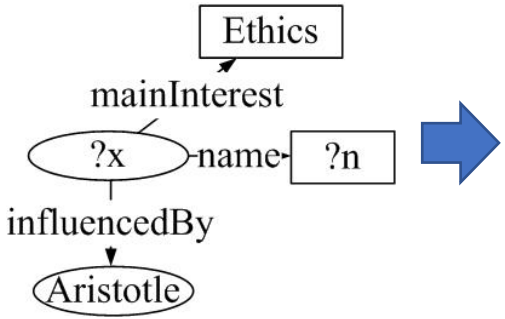
- Given a set of structural simple predicates $SP = \{sp_1, sp_2, ..., sp_y\}$

  for frequent access pattern p, a structural minterm predicate mp_i is

$$mp_i = \bigwedge_{sp_k \in SP} sp_k, 1 \le k \le y$$

  where $sp_k{}^* = sp_k$ or $sp_k{}^* = \neg sp_k$

Example Structural Simple and Minterm Predicates



$$sp_1 : p(?x1) = Aristotle;$$

$$sp_2 : p(?x1) \neq Aristotle;$$

$$sp_3 : p(?x2) = Ethics;$$

$$sp_4 : p(?x2) \neq Ethics.$$

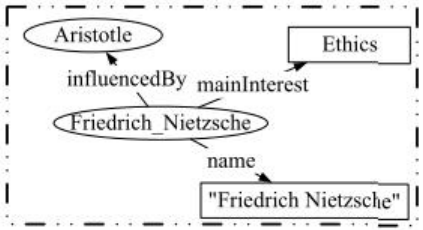
$$mp_1 : p(?x0) = Aristotle \wedge p(?x1) = Ethics;$$

$$mp_2 : p(?x0) = Aristotle \wedge p(?x1) \neq Ethics;$$

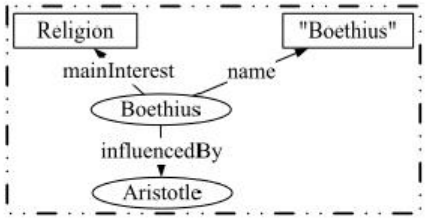
$$mp_3 : p(?x0) \neq Aristotle \wedge p(?x1) = Ethics;$$

$$mp_4 : p(?x0) \neq Aristotle \wedge p(?x1) \neq Ethics.$$

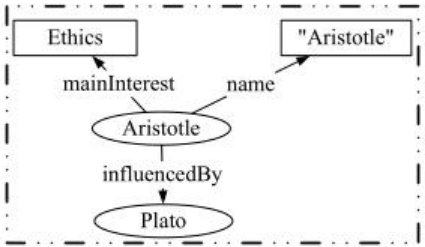
Example Horizontal Fragments



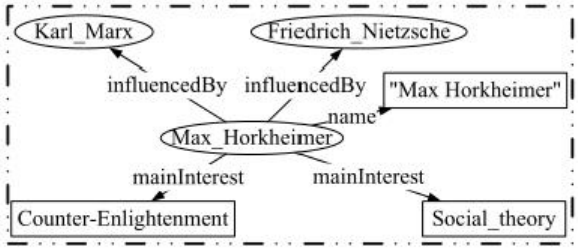
(a) Example Horizontal Fragment Generated from mp_1



(b) Example Horizontal Fragment Generated from mp_2



(c) Example Horizontal Fragment Generated from mp_3



(d) Example Horizontal Fragment Generated from mp_4

Mixed Fragmentation

- In our mixed fragmentation, we keep some frequent access patterns (P^v) for the vertical fragmentation but others (P^h) are designed for horizontal fragmentation

Classification of Frequent Access Patterns

- Given a frequent pattern *p*, let *M(p)* denote all structural minterm predicates that are derived from pattern *p*.
- The classification strategy is based on the access frequencies of the FAPs and corresponding structural minterm predicates to divide *M(p)* into two subsets:

$$M'(p) = \{mp | mp \in M(p) \wedge acc(mp) \geq minSup\}$$
$$M''(p) = \{mp | mp \in M(p) \wedge acc(mp) < minSup\}$$

Mixed Fragmentation Strategy

- Given a selected frequent access pattern p, we have the following fragmentation strategy:
  1. If $\sum_{mp \in M''(p)} acc(m\ p) \geq minSup$, we vertically partition the RDF graph based on pattern p, i.e., $p \in P^v$;
  2. If $\sum_{mp \in M''(p)} acc(m\ p) < minSup$, we horizontally partition the RDF graph based on frequent structural minterm predicates $M'(p)$ that are derived from pattern p, i.e., $p \in P^h$.

Allocation

- Two fragments should be placed in one site if they are often accessed together

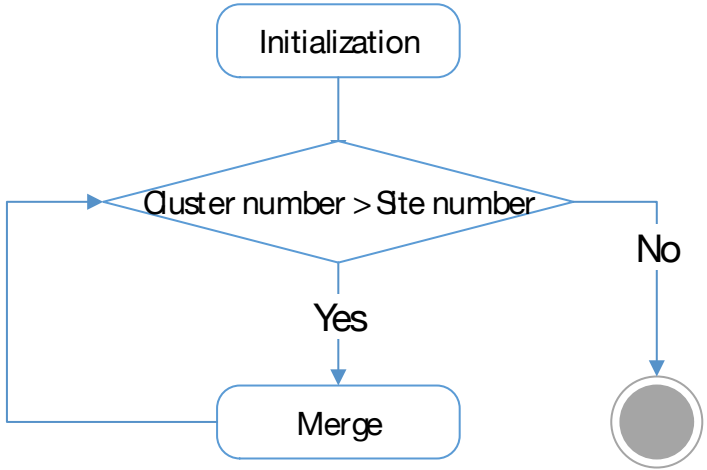
Fragment Affinity Metric

- The fragment affinity metric between two fragments F and F' with respect to the workload $Q = \{Q_1, Q_2, ..., Q_r\}$ is defined as follows

1. Given two vertical fragments F and F' generated from frequent access patterns p and p', $aff(F, F') = \sum_{k=1}^{r} use(Q_k, p) \times use(Q_k, p')$;
2. Given two horizontal fragments F and F' generated from structural minterm predicates mp and mp', $aff(F, F') = \sum_{k=1}^{r} use(Q_k, mp) \times use(Q_k, mp')$;
3. Given two fragments F and F' in mixed fragmentation generated from FAP p and structural minterm predicate mp, if mp is not generated from p, $aff(F, F') = \sum_{k=1}^{r} use(Q_k, p) \times use(Q_k, mp)$; otherwise, $aff(F, F') = 0$

Allocation Algorithm

- We extend a clustering algorithm, PNN, to cluster all fragments

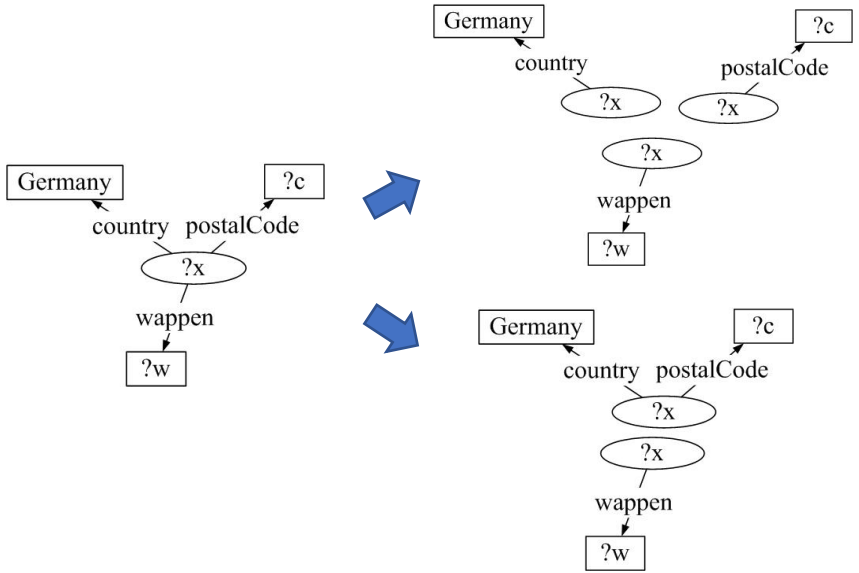


Data Dictionary

- The global statistics of fragmentation and allocation need to be stored and maintained in data dictionary
  1. fragment definitions
  2. fragment sizes
  3. site mappings
  4. access frequencies
  5. ……

Query Decomposition

- During query evaluation, we first decompose the query based on the frequent access patterns



Cost of Query Decomposition

- The cost of a decomposition is

$$\cos t(D) = \prod_{q \in D} card(q_i)$$

where D is a decomposition result and $card(q_i)$ is the number of matches for q_i

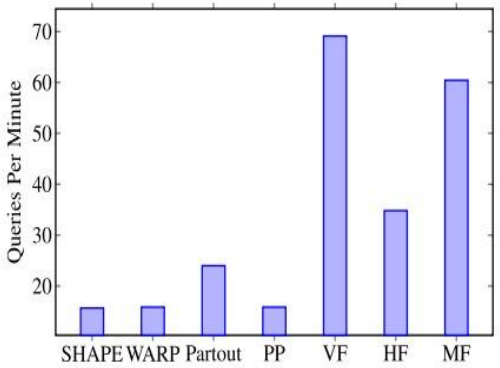
Query Optimization and Execution

- We extend the algorithm of System-R to find the optimal execution plan for joining the results of all subqueries
- Each subquery is executed in the corresponding sites in parallel
- We join them together according to the optimal execution plan

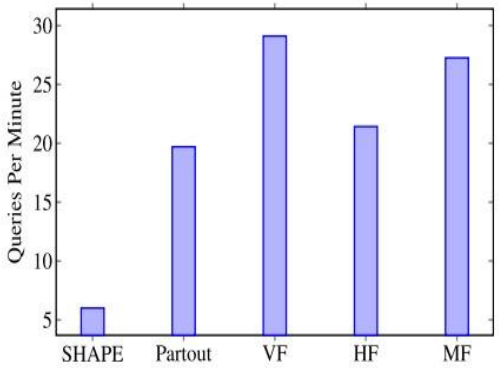
Experiments

- **Datasets:** DBPedia (about 330 millions triples) and WatDiv (50 millions to 250 million triples)
- **Workload:** 8 millions queries of DBPedia and 2 thousands queries of WatDiv
- **Competitor:** WARP and SHAPE
- **Environment:** 10 machines running Linux in a LAN

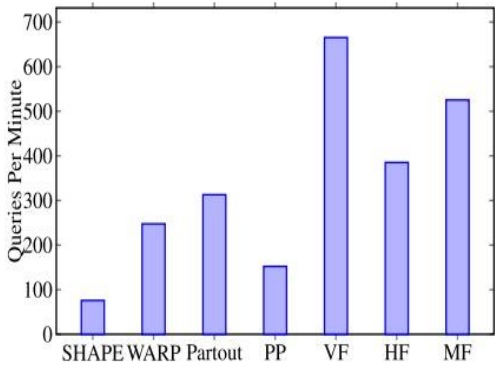
Throughput



(a) **DBpedia**

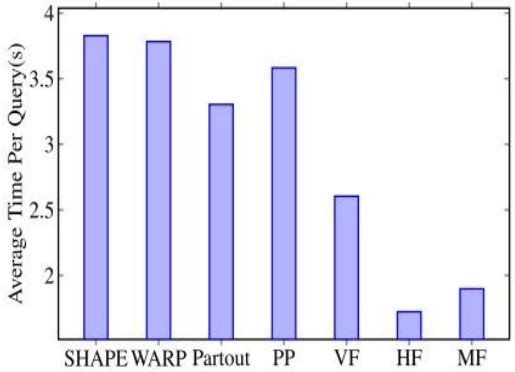


(b) **LinkedGeoData**

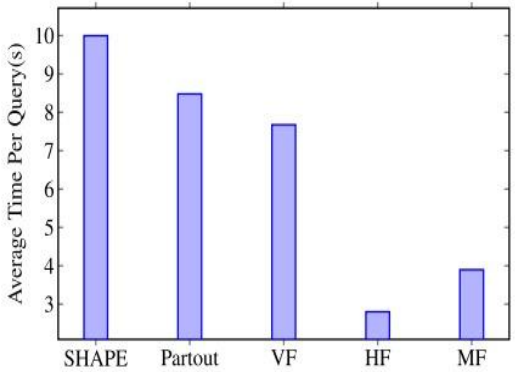


(c) **WatDiv**

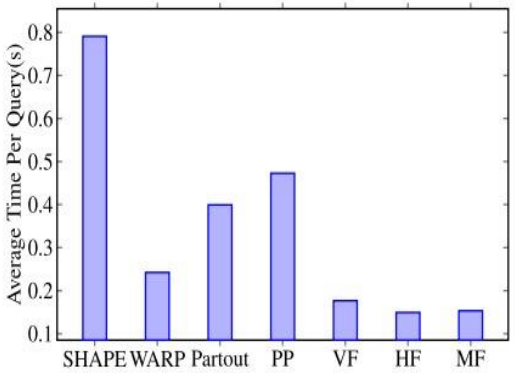
Response Time



(a) **DBpedia**



(b) **LinkedGeoData**



(c) **WatDiv**

Part 3

Conclusions

Conclusions

- We introduce partitioning-based distributed RDF systems and classify them into two categories: **workload-agnostic** and **workload-aware** approaches

THANK YOU

References

- Jiewen Huang, Daniel J. Abadi, Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. Proc. VLDB Endow. 4(11): 1123-1134 (2011)
- Kisung Lee, Ling Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. Proc. VLDB Endow. 6(14): 1894-1905 (2013)
- Yuanbo Guo, Zhengxiang Pan, Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. J. Web Semant. 3(2-3): 158-182 (2005)
- Peng Peng, M. Tamer Özsu, Lei Zou, Cen Yan, Chengjun Liu. Accelerating Partial Evaluation in Distributed SPARQL Query Evaluation. Accepted in ICDE 2022
- Katja Hose, Ralf Schenkel. WARP: Workload-aware replication and partitioning for RDF. ICDE Workshops 2013: 1-6
- Peng Peng, Lei Zou, Lei Chen, Dongyan Zhao. Adaptive Distributed RDF Graph Fragmentation and Allocation based on Query Workload. IEEE Trans. Knowl. Data Eng. 31(4): 670-685 (2019)
- Peng Peng, Lei Zou, Lei Chen, Dongyan Zhao. Query Workload-based RDF Graph Fragmentation and Allocation. EDBT 2016: 377-388