KYUNG HEE UNIVERSITY

Department of Computer Science & Engineering
Ubiquitous Computing Lab

Ph.D. Dissertation Presentation

November 06th, 2020

A Cache Based Method to Improve Query Performance of Linked Open Data Cloud

Mr. Usman Akhtar

Department of Computer Science and Engineering
Kyung Hee University

Advisors

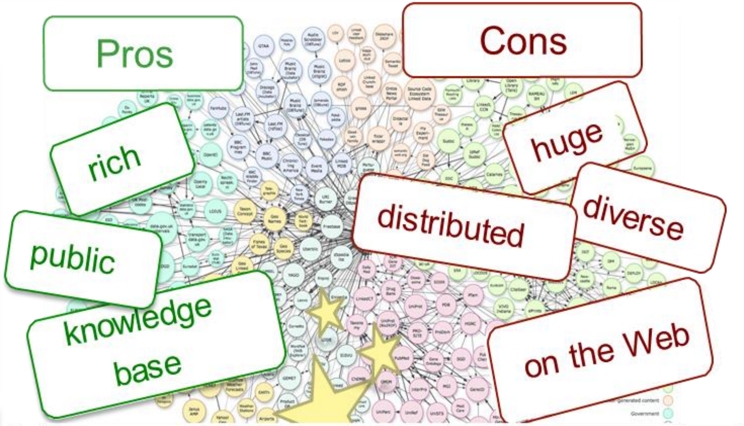
Prof. Sungyoung Lee
Prof. Eui-Nam Huh

Contents

Background

Linked Data Dynamics

- Linked Open Data Cloud (LOD) is a distributed knowledge base on the web that handles a large number of requests from applications consuming these data [1,2].
- Understanding the evolution of the Linked Data Cloud (LOD) is important for applications [5,6].
  - e.g., Query Caching, Web Crawling, and knowledge graph search engines.



LOD: a rich, huge, diverse, public and distributed knowledge base on the Web.

Querying

- Traditional ways of querying LOD are as follows:
  - Data Dumps [6].
  - Querying endpoints [7].

Data Dumps

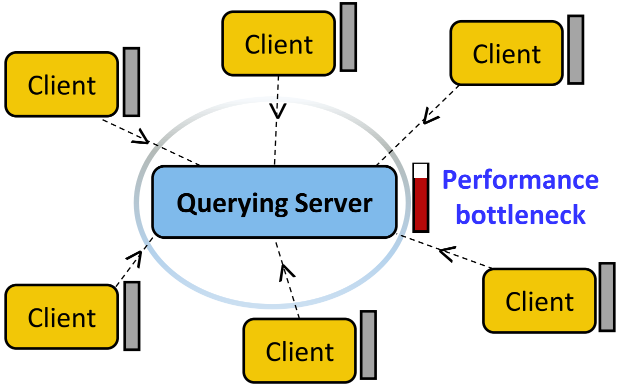
Cons: Dump the data locally and allows to setup own private querying endpoints.

- Out-of-date data
- No longer query the web
- Infrastructure cost

Querying endpoints

Cons: Public endpoints are often unreliable.

- Low availability (Downtime)
- High querying cost
- Hosting endpoints are expensive



Server perform all the processing

Motivation

- To unlock the full potential of Linked data sources, we need flexible ways to query them [8].
  - ✦ Following benefits drive our research:
    - ◆ High-availability to the server.
    - ◆ High query performance

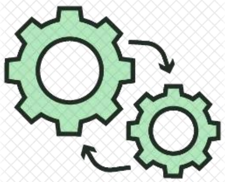
Limitations	Benefits	Limitations
– Low availability	+ High availability	– Low availability
– Out-dated data	+ Decentralized caching	– Hosting endpoints are expensive
– No longer query the web.	+ Low server cost	– High client cost

Data dumps (Existing) — **Proposed** (Caching) — **SPARQL endpoint** (Existing)

Linked Data Caching

- (Existing) Server-side caching
  - + Server manages request from cache
  - + Dependent on the database
  - – Not flexible approach
  - – High server cost
  - – Low availability
- (Proposed) Client-side caching
  - + Moving intelligence from server to the client
  - + More flexible
  - + High availability
  - + Low server cost
  - – Not fully explored, still in the early days of the research

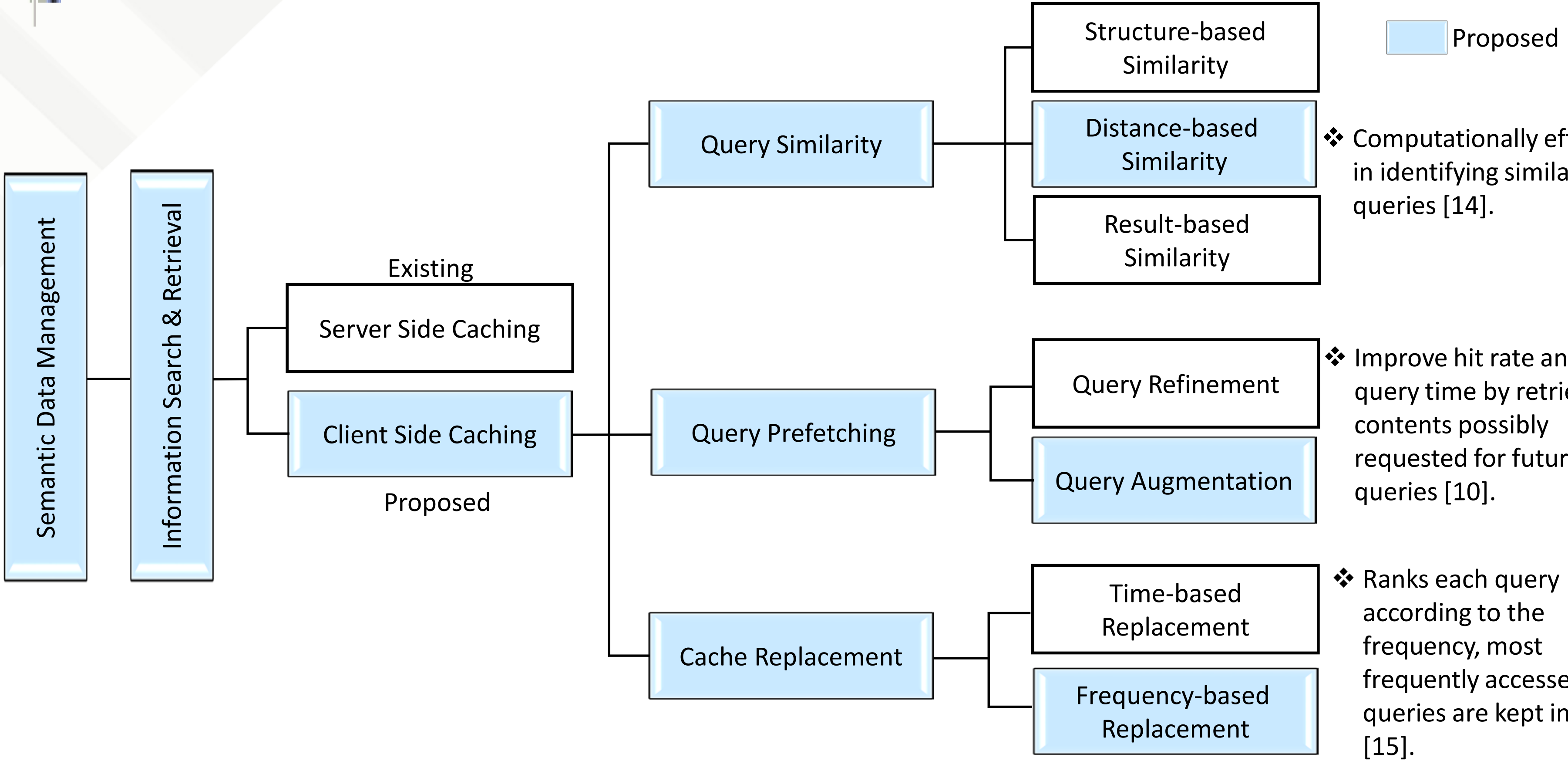
Methodology



- I **Query Prefetching** — A **Similar query requests are served from cache**
- I **Decentralized caching** — A **Higher hit rates**

I Idea A Advantage

Research Taxonomy



- ❖ Computationally efficient in identifying similar queries [14].
- ❖ Improve hit rate and query time by retrieving contents possibly requested for future queries [10].
- ❖ Ranks each query according to the frequency, most frequently accessed queries are kept in cache [15].

Research Problem

Problem Statement

Reliable query access of public linked datasets largely remains challenge due to low availability, especially under high work loads, therefore, preventing it's use in real-world applications [1-4],[8].

Goal

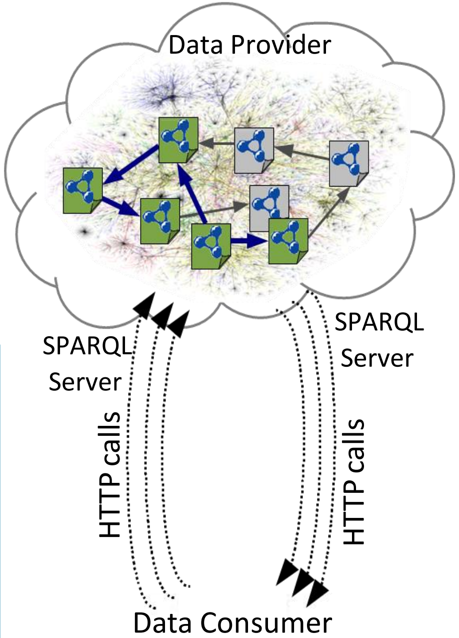
- To devise a scalable and sustainable approach that identifies the possible bottlenecks of querying web of data with high availability by moving intelligence from the server to the client. Thus, achieving a better query performance for interactive exploration of LOD.

Objectives

- To design a client-side caching by balancing the cost of query processing between data providers and data consumers.
- Leverage the tradeoffs between the data availability and improve query performance by answering queries from client's cache.

Challenges

- Challenge 1: Local caches become outdated, due to the continuous evolution of LOD. In order to utilize best resources, improvised change metric is required to quantify the changes occurs in the LOD cloud [11][12].
- Challenge 2: Queries issued by the end user are similar in pattern, challenge is to find identical queries, as two queries may be structurally similar but different in content [9][10].
- Challenge 3: Cache has a limited space, it is critical to fill it with valuable content, therefore, how to design an efficient cache replacement policy? [13]



Problem Illustration

Server perform all the processing

Related Works

		Existing work problems			
Categories	**Methodologies**	**Advantages**	**Method**	**Limitation**	**Overheads**
Query Similarity & Prefetching (Challenge 1&2)	**[SQC]** Improving the performance of semantic web applications with SPARQL query caching [16]	• Cache complete triples query results. • Introduce a proxy layer to cache repeated query results	Structure based similarity	**Server side caching** Only consider repeated queries.	**High**
	[PFU] Proactive Policy for Efficiently Updating Join Views on Continuous Queries Over Data Streams and Linked Data [11]	• Proposed maintenance policy that update the cache prior to query execution	Content based similarity	**Server side caching** Only update the local cache at system idle time.	**High**
	[CIR] Caching intermediate result of SPARQL queries [17]	• Adaptive cache to store intermediate results of SPARQL queries	Result based similarity	**Client side caching** No cache replacement policy is introduced.	**High**
	[SDC] Semantic data caching and replacement [18]	• Proposed a semantic region based caching and a distance measure to update cache	Distance based similarity	**Server side caching** Only considered the structure similarity while creating a semantic region.	**High**
	[CAS] Towards content aware SPARQL caching for semantic web application [19]	• Introduced a query containment which evaluated whether a query can be answered from cache or not.	Content based similarity	**Client side caching** Containment checking is computationally expensive task.	**High**
Cache Replacement (Challenge 3)	**[GAW]** Graph-aware, workload-adaptive SPARQL query caching [20]	• Work-load adaptive caching to reduce the SPARQL query response time	Result based similarity	**Server side caching** Time based cache replacement	**High**
	[Autosparql] Let user query your knowledge base [21]	• Proposed machine learning approach to leverage the query processing.	Structure based similarity	**Server-Side Caching** The feature modeling approach in their work is time consuming	**High**
Query Similarity, prefetching & Cache Replacement	Proposed method	**O** (Alleviate burden on querying endpoints by identifying queries learnt from client historical patterns)	**O** (Distance based similarity & Frequency based cache replacement)	**O** (Local data Cache need to be updated during system idle time)	**Low**

Our Approach: Limitation, Objective & Solutions

Limitations & Challenges

Challenge: 1

- Unable to quantify the evolution of LOD and changes occurs in the cloud.
- Unable to prioritize recent changes occurs in the cloud

Proposed Solution

Solution 1(a)

- Dynamic function to quantify the evolution of LOD.
  - AACP Algorithm
  - PASU Algorithm

Objectives

- Identifies the changes occurs in the LOD.
- Predicting change for better resource utilization

Limitations & Challenges

Challenge: 2

- Unable to identifies the similar structure queries that aggravate the burden on query.

Proposed Solution

Solution 1(b)

- Query Augmentation to prefetch contents possibly requested in future.
  - Query Suggestion
  - Query Relaxation

Objectives

- Remove the burden on the querying endpoints.
- Result of similar queries are placed in cache for future access.

Limitations & Challenges

Challenge: 3

- Unable to replace less valuable items from cache.

Proposed Solution

Solution 2

- Adaptive cache replacement when cache become full.
  - Frequency based cache replacement

Objectives

- Identifies the less valuable items from cache.
- Improve cache hit rates

Thesis Map

User

LOD Cloud

Proposed Solutions

Solution 1a

Change Metric

Problem: Linked Data Evolution

Result: Quantify the changes occurs in the cloud such as addition and deletion of items

Algorithm 1 & 2
AACP Algorithm
PASU Algorithm

Problem: Change Prioritization

Result: Identify the recent changes occurs in the cloud using weight function

Solution 1b

Algorithm 3
Query Augmentation

Problem: Similar structure queries

Result: Prefetches the result of similar queries and placed in cache for future queries

Solution 2

Algorithm 4
Cache Replacement

Problem: Replace cache

Result: Frequency based cache replacement for replacing less valuable cache items

Outcomes

- + Low overheads
- + High Query performance

Challenges

C1: Linked data evolution

C 2: Query Similarity

C 3: Cache Replacement

Challenge 1, 2

Challenge 3

Existing Approach

- Unable to quantify the evolution of LOD.
- Less flexible
- Dump all the data locally.

Existing Approach

- Unable to prioritize recent changes occurs in the cloud
- Issued irrelevant queries
- Performance bottleneck

Existing Approach

- Only consider the structure similarity.
- Aggravate the burden on query endpoint by issuing similar structure queries.

Existing Approach

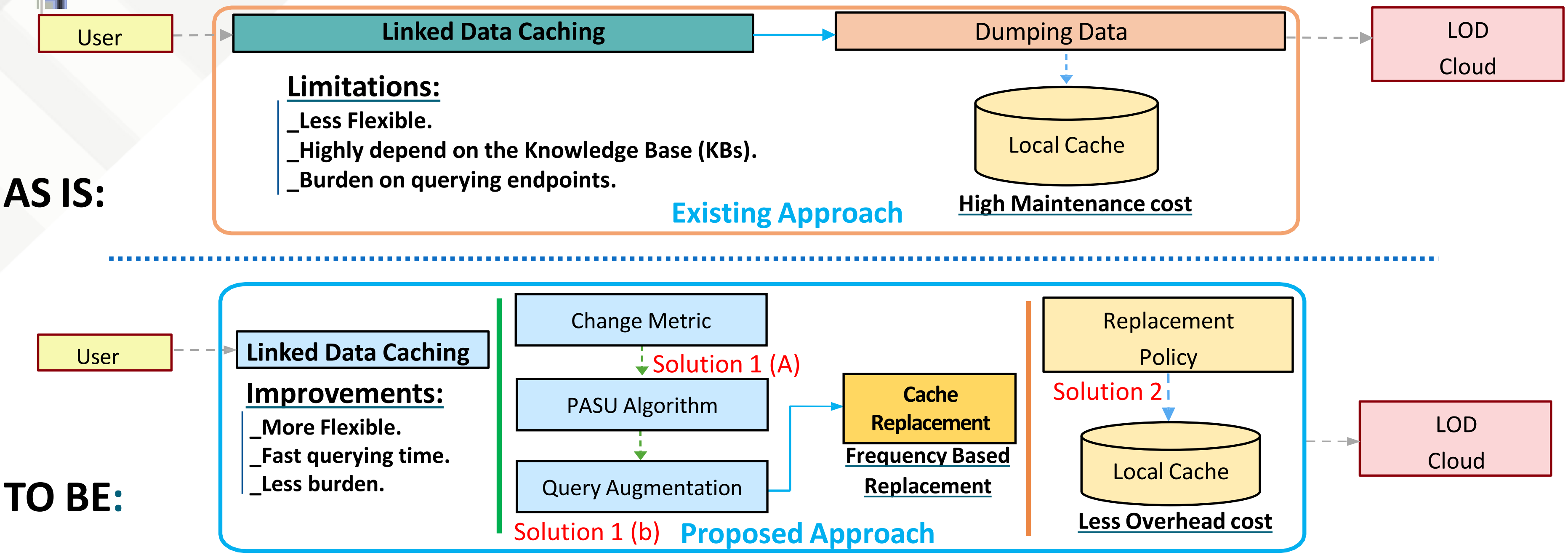
- The cost of cache maintenance is high
- Introduced the time based cache replacement
- High overhead

Outcomes

- High Overheads
- Low query performance

Proposed Solutions

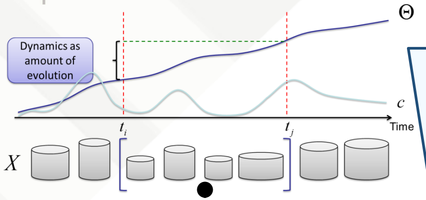
(Abstract)



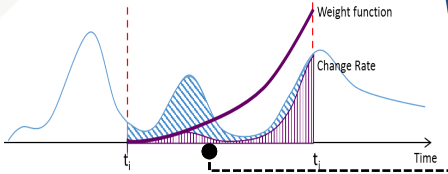
- **Thesis Contributions**
  - Comprehensively utilize client side Linked Data caching for better query performance.
    - **Solution 1(a):** Proposed change metric to quantify the evolution of Linked Open Data (Published: IEEE Access).
    - **Solution 1(b):** Proposed query augmentation to alleviate the burden on server (Published: Computing, Springer).
    - **Solution 2:** Proposed frequency based cache replacement to replace less valuable cache items (Published: Applied Sciences, MDPI).

Proposed Solutions

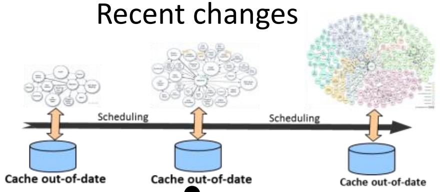
(Detailed)



Linked Data evolution



Recent changes



Cache Maintenance

Solution-1a
(Quantify evolution of LOD)

Dynamic Function

Change Prioritization

Maintenance Policy

Change Metric

Quantify Evolution of LOD

Highlights of the proposed solutions

- The construction of the Dynamic function to quantify the evolution of LOD.
- Prioritize the recent changes in the cloud and update the local cache.
- Prefetching the result of previously issued queries and store in cache for future queries.

Solution-1b
(Based on Prefetching)

Query Suggestion

Query Relaxation

Query Prefetching

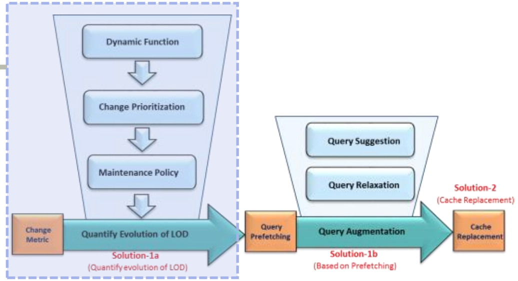
Query Augmentation

Solution-2
(Cache Replacement)

Cache Replacement

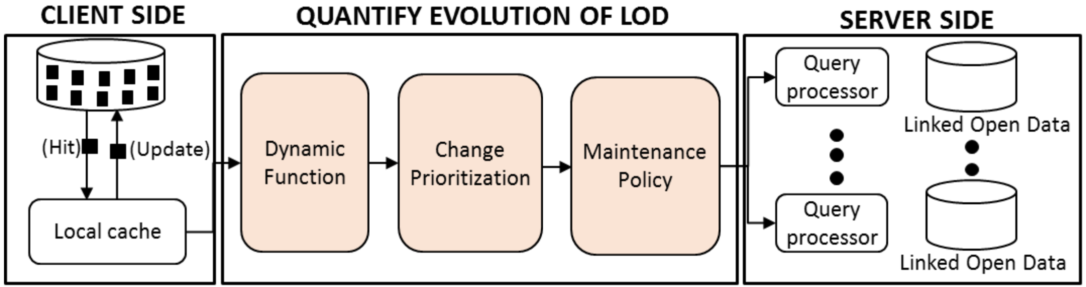
Solution 1(a): Quantify Evolution of LOD (1/4)

- **Change Metric:**
  - **Problem:** Due to the evolution of the LOD sources, local data caches become outdated.
  - **Ideal Case:** Quantifies the changes occured in the recent past.
  - **Solution:** Proposed change metric to quanity the evolution of the LOD cloud.



Proposed Methodology

Workflow to obtain the model for Linked Data evolution



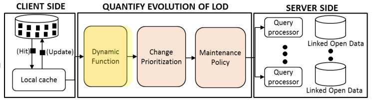
(Challenge 1a & 1b)

- **Proposed Uniqueness:**
  - **I. Dynamic function**
    - Distance-based approach to identifies changes in LOD cloud.
  - **II. Change Prioritization**
    - Weight-based function to assign importance to the recently change items in LOD.
  - **III. Maintenance Policy**
    - Our maintenance policy update the local cache based on the preference score.

Differences from Existing Approaches

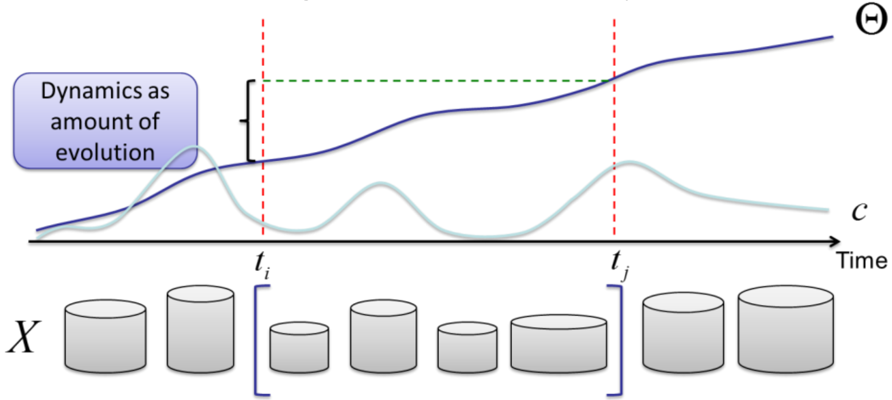
- Compared to data dumps, our approach is more flexible to identify the changes in the LOD cloud.
- Change-aware maintenance for effectively updating local data cache

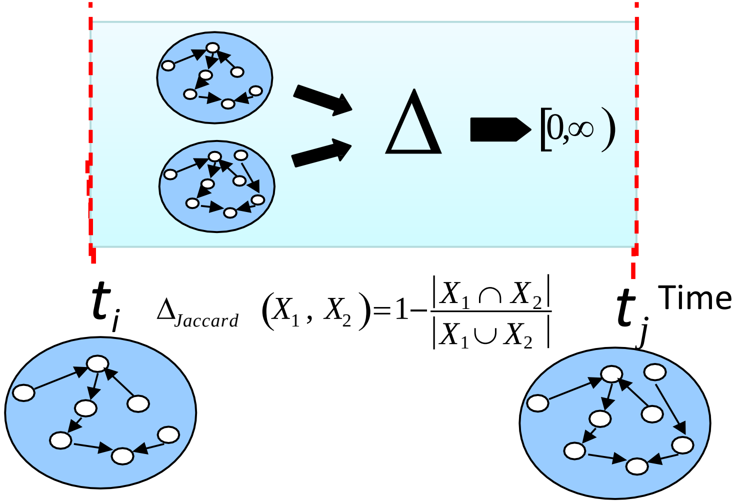
Solution 1(a): Quantify Evolution of LOD (2/4)



I. Dynamic Function (Θ)

- **Problem:** Linked Data Cloud (LOD) are highly dynamic in nature e.g., contents gets added, updated and removed.
- **Ideal Case:** Quantifies changes accours in the cloud.
- **Solution:** Change metric based on the dynamic function.





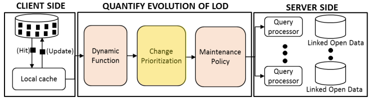
Differences from Existing Approaches

- Existing approach **dump** all the data into local data caches.
- The existing change metric are **less dynamic** unable to identify which data sources are added, updated and removed.

Salient Features and Benefits

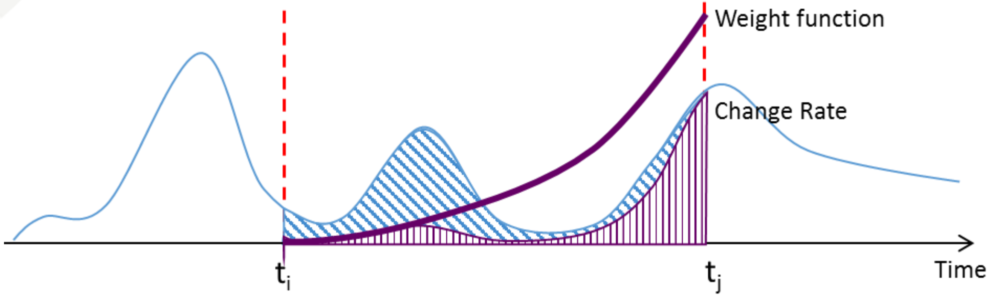
- Proposed function able to **quantify** the addition and deletion of triples with a real number.
- As compared with existing approaches, our approach is **more flexible** to identifies the changes in the LOD cloud.

Solution 1(a): Quantify Evolution of LOD (3/4)



II. Change Prioritization

- **Problem:** Impact of the data evolution is **independent** of the time e.g., contents are added, updated and removed.
- **Ideal Case:** It is desireable to **priotize** changes in certain period of time such as **recent** past.
- **Solution:** Application-Aware Change Prioritization (AACP), assign more weights to changes occurs in the recent past.



Proposed Algorithm (AACP)

$Input$: Linked Data Set

$Output$: Recently changed Item

$For\ t_i \leftarrow X_i$

$ComputeAge$

$If\ \ t$ is timestamp of the record then

$t = t_p$ where t_p is the present time

$return\ w(X_i, t_i)$

Existing Algorithm

$Input$: Linked Data Set

$Output$: Sources Based on the Page Rank

$For\ t_i \leftarrow X_i$

$ComputePageRank$

$If\ \ PR$ is rank of the page record then

$PR > X_i$ where X_i is the data sets

$return\ PR(X_i)$

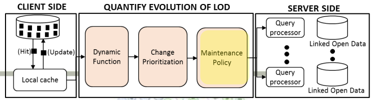
Differences from Existing Approaches

- Assign importance of the **recently changed** items in LOD cloud.
- Our approach consider the dynamic resource based on the **last modified date** as compared to existing which is mainly focus on the page rank.

Salient Features and Benefits

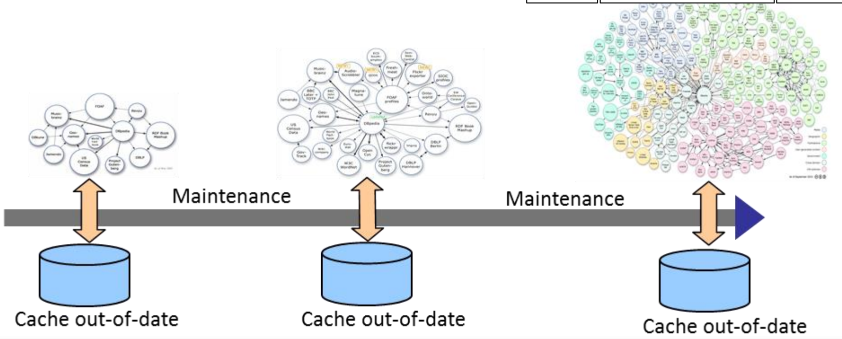
- **More flexible** for data analytics applications that consume the evolving LOD cloud.
- **Less space** is required to update the recently changed items.
- **Offline process** to evaluate the resources.

Solution 1(a): Quantify Evolution of LOD (4/4)



III. Maintenance (Update) Policy

- **Problem:** Due to the evolution of the LOD sources, local data caches become outdated.
- **Ideal Case:** Maintenance policy only visits the sources that have been changed in the recent past.
- **Solution:** Preference-Aware Source Update (PASU), update the local cached based on the preference score



Proposed Maintenance Policy

Preference Aware Source Updates (PASU):

Existing approach prefetches all the data and store in the local cache. The proposed maintenance policy will update the local data caches according to the preference score:

I. *Init ()*

II. *Update Function ()*

III. *Estimate Score ()*

Update local cache based on the preference score

$Init()$

$N = 0; \Rightarrow$ Total number of access

$X = 0; \Rightarrow$ number of detected changes

$T = 0; \Rightarrow$ sum of the times from changes

$Update(T_i, I_i)$ Update Variables

$\mathrm{N} = \mathrm{N}+1;$

If $(T_i \angle I_i)$ *then* $\Rightarrow$ Has the element changed?

$UpdateCache \leftarrow JobScheduler$

else

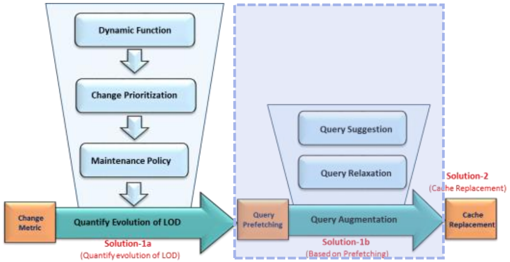
$T = T + I_i; \Rightarrow$ The element has not changed.

$Estimate()$

$return\ \frac{X}{T}; \Rightarrow$ return the estimated value.

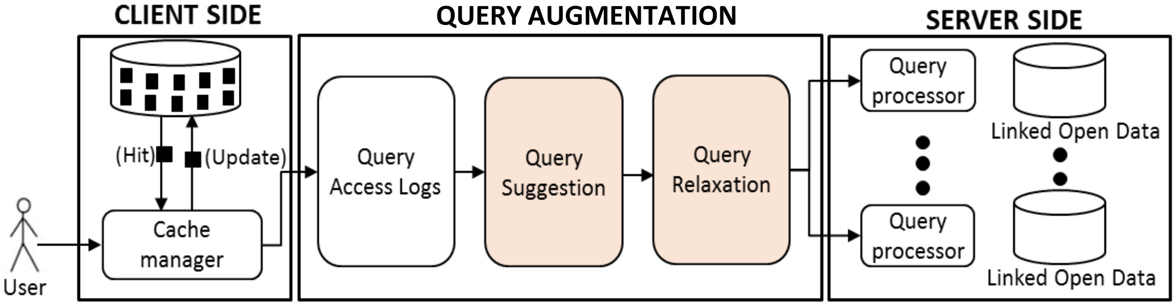
Solution 1(b): Query Augmentation

- **Query Augmentation:**
  - **Problem:** More than half of the querying endpoints, which is <95% of availability.
  - **Ideal Case:** Retrieve contents that are possible be requested for future access.
  - **Solution:** Proposed query augmentation to alleviate the burden on server and prefetches the result of similar queries for future access.



Proposed Methodology

Workflow to obtain the model for Query Augmentation



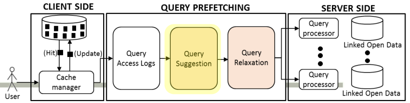
(Challenge 1a & 1b)

- **Proposed Uniqueness:**
  I. **Query Suggestion**
  - Our approach considers both structure and content-wise similarity based on the distance score.

  II. **Query Relaxation**
  - Our approach attempts to modify the queries to retrieve additional information relevant for future access.

Differences from Existing Approaches

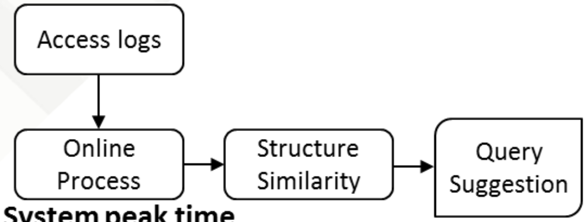
- Existing approaches only consider structure similarity. Two queries potential are same but yields different results.
- Existing approach aims at improving the recall in retrieval effectiveness.



Solution 1(b): Query Augmentation

i. Query Suggestion → Comparison

Existing Approach



System peak time

Algorithm: Query Suggestion

Input : Query logs

Output : *MatchingPatterns*

$Score \leftarrow 0$

$foreach\ (Q_1, Q_2) \in mappings = 1\ do$

$Score\ \leftarrow score + \Delta(Q_1, Q_2)$

else

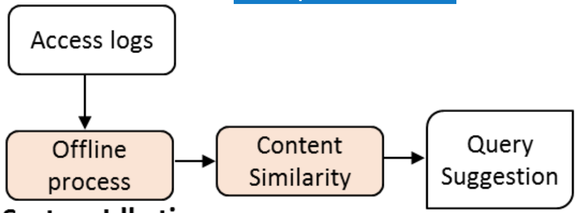
$if\ \ Q_1 \neq Q_2\ \ then$

return ∞

Existing Approach Limitations

- Only consider the structure-wise similarity.
- Online process, compute during system peak time

Proposed Idea



System Idle time

Algorithm: Query Suggestion

Input : Query logs

Output : *MatchingQueries*

$if\ \ Q_1 \neq Q_2$ then

$return \leftarrow 0$

$foreach\ \ (Q_1 \in Q_2) = mappings.values\ then$

$foundmapping \leftarrow ture$

break

else

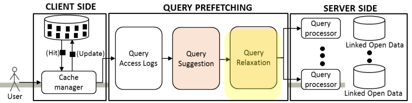
$if\ \ Subject \in Subject.Count\ \ then$

$Subject.Count.IncreaseCount()$

return *result*

Proposed approach benefits

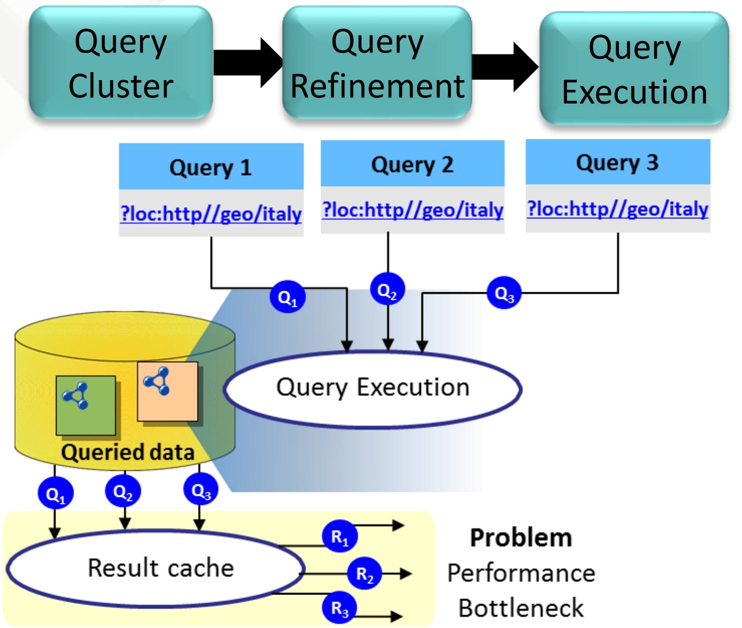
+ Consider the structure and content-wise similarity.
+ As an offline process, compute only during the system idle time.



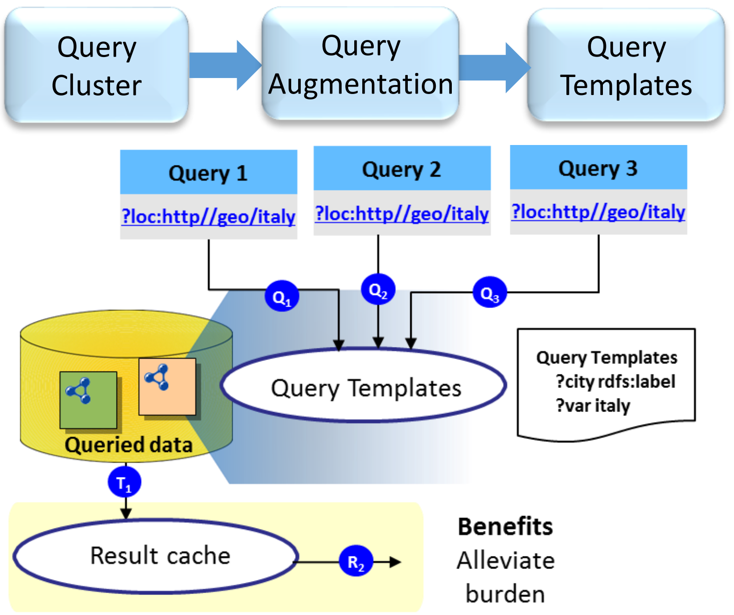
Solution 1(b): Query Augmentation

ii. Query Relaxation → Comparison

Existing Approach



Proposed Idea



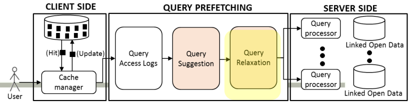
Existing Approach Limitations

- Aggravate the burden on query endpoint by issuing similar structure queries
- Performance bottleneck

Proposed approach benefits

+ Similar structure queries are execute only once.
+ Store the additional facts useful for subsequent queries.

Solution 1(b): Query Augmentation



Existing Approach

Query Relaxation → Comparison

Proposed Idea

Algorithm: Query Refinement

Input : $C = \{Q_1, Q_2, ..., Q_n\}$
Output : $Q_E = QueryExecution$
$foreach\ Q \in C\ do$
$m \leftarrow \mathrm{DistanceScore}(D)$
$foreach\ C \in m\ do$
$Q \leftarrow Distance(D)$
$FoundMapping \leftarrow True$
$Break$
return Q_E

Algorithm: Query Template

Input : $C = \{Q_1, Q_2, ..., Q_n\} : SimilarQueries$
Output : $Q = QueryTemplate$
$foreach\ Q_p \in C\ do$
$m \leftarrow TriplePatternMatching(P_{Q_i})$
$foreach\ T_p \in m\ do$
$Q \leftarrow replace(Q_i, Q_j)$
$Until$
$T \subseteq T.getVariables()$
$\mathrm{Q.addtoProjection(T.getvariable())}$
return Q

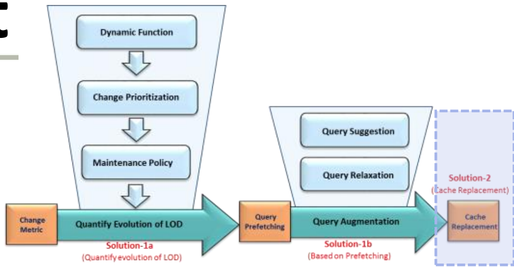
Algorithm: Central Concept Fetching (CCF)

Input : $T_Q = \{Q_1, Q_2, ..., Q_n\}$
Output : $Occurance\ of\ Frequent\ Subjects$
$Subject.Count \leftarrow 0$
$foreach\ Q_p \in T = condition\ do$
$S \leftarrow \Theta(P_{Q_i})$
$Where\ S \neq 0\ do$
$foreach\ Q_p \in S\ do$
$S \leftarrow S \cup \Theta(P_{Q_i})$
$else$
$(S, P, O) \leftarrow \Theta(P_{Q_i})$
$Subject.Count.IncreaseCount()$
return $getHighestCount$

Differences from Existing Approaches

- We utilize the template based prefetching to alleviate the burden of execution of similar queries.
- By using the augmentation of central concept fetching, we retrieve additional information that are useful for future queries.

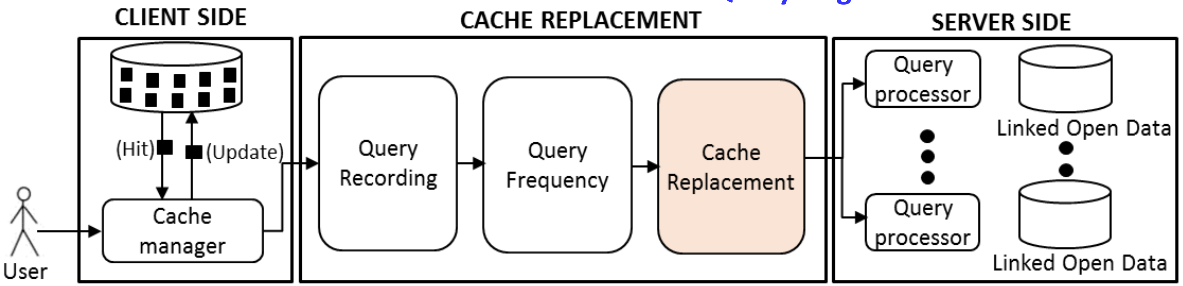
Solution 2: Linked Data Cache Replacement



Proposed Methodology

- **Cache Replacement:**
  - **Problem:** Cache replacement is a problem to replace the cache with valuable content.
  - **Ideal Case:** The request of similar queries are sent from the cache to improve cache hit rates.
  - **Solution:** Our idea is to replace cache based on the accessed frequencies highly accessed querie are kept in cache for future queries.

Workflow to obtain the model for Query Augmentation



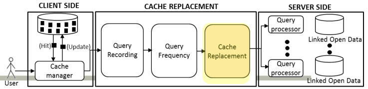
(Challenge 2)

- **Proposed Uniqueness:**
  - **Cache Replacement**
    - We proposed an frequency-based cache replacement, when the cache become full, the replacement is based on the access frequencies.

Differences from Existing Approaches

- The cache replacement is performed when the local cache become full.
- The proposed cache replacement is based on the frequency of the previous accessed queries.
- Higher queries are placed in cache for future access

Solution 2: Linked Data Cache Replacement



Cache Replacement → Comparison

Existing Approach

- **Existing approach proposed a time-based cache replacement:**
  - Do not consider the frequency of the access data.
  - Poor accuracy of the cache replacement reduces the hit rates.
  - When the caches become full, existing approaches triggers the full cache replacement.

Limitation of Existing approach

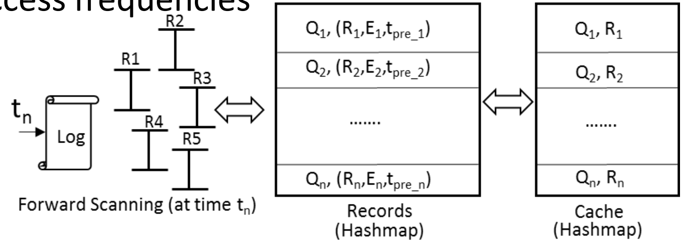
- Time based cache replacement.
- Low cache hit rates due to inefficient approach
- Space and performance overheads

Proposed approach benefits

+ Predict when the cache need to be updated.
+ Frequency based cache replacement.
+ Less space and performance overheads.

Proposed Idea

- **We proposed an offline process for cache replacement, following are the steps involved in our approach:**
  + Log record accesses
  + Forward scanning to identify access frequency
  + Exponential smoothing approach to estimate access frequencies



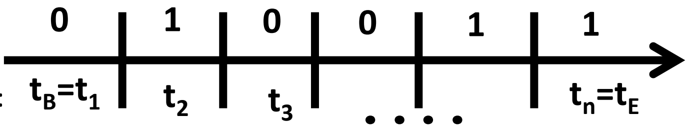
- **Calculation of access frequencies using exponential smoothing:**

$$w(t_k) = \alpha * x_{t_k} + (1-\alpha) * w(t_{k-1})$$

$W(t_k)$: Score at time t_k X_{t_k}: observed value at time t_k

α : decay constant In our scenario, x_{t_k} = 1 if observed, 0 otherwise

Example timeline for single record:



Experimental Environments

(Solution - 1a,1b & 2)

- **Setup Configurations:**
  - We have evaluated our proposed on real Linked datasets e.g., DYLDO & BTC datasets [12].
  - All the experiments were performed on 4x AMD A8-7650 Radeon R7, 64bit Ubuntu LTS and OpenLink Virtuso Server
- **Dataset selection:**
  - **DYLDO Dataset:** On average the size of the each snapshots is 1.35 GB (149 Weekly Crawls)
  - **BTC Datasets:** Collected from multi-crawler frameworks, the size of each snapshots was 3.7 GB.
- **Evaluation metric:**
  - **Accuracy:** We have evaluated the effectiveness of our approach by using the precision and recall.

$$\text{Precision} = \frac{\sum |X_{c,t} \cap X'_{c,t}|}{\sum X'_t \, |X'_{c,t}|} \qquad \text{Recall} = \frac{\sum |X_{c,t} \cap X'_{c,t}|}{\sum X'_t \, |X_{c,t}|}$$

  - **Effectivity:** Shows the runtime overhead cause due to irrelevant execution

$$Effectivity(\%) = \frac{R_{relavantquery}}{T_{totalexecution}}$$

  - **Performance:** Examine the hit rates and space overhead.

DYLDO Datasets Characteristics

PLD	Avg. triples per snapshots	Avg. triples added	Avg. triples remove	Description
Identica.ca	1,341,045	2930	2563	Open source social engine
Loc.gov	369,884	2220	1890	Library congress
Linkedct.org	1,782,884	4263	3529	Live data browser
Dbtropes.org	4,080,910	5414	45518	Online wrapper
Neuinfo.org	2,065,028	3580	4080	Neuroscience information

BTC Datasets Characteristics

PLD	Avg. triples per snapshots	Avg. triples added	Avg. triples remove	Description
Berkeleybop	55,124,003	3920	2563	Social engine
Bio2rdf.org	20,168,230	4263	3529	Disease data
Data.gov.uk	13,302,277	5414	45518	DBpedia
Dataincubator	1,729,455	3810	2108	Data science
Freebase	25,488,720	2630	9080	Wikipedia

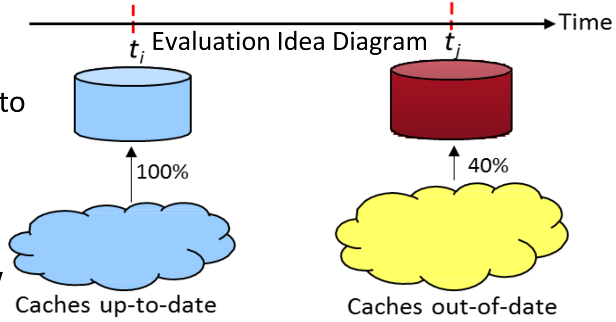
- **Candidate Approaches:**
  - ✓ **PageRank**
  - ✓ **Size**
  - ✓ **Age**
  - ✓ **ChangeRatio**
  - ✓ **ChangeRate**
  - ✓ **TLR**
  - ✓ **LRU**
  - ✓ **LFU**
  - ✓ **SQC**
  - ✓ **Proposed**

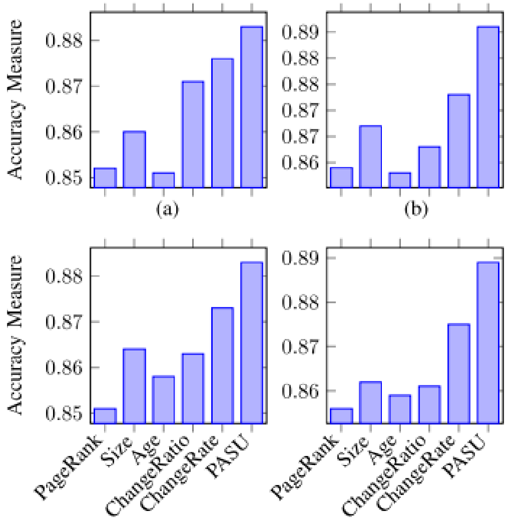
Experimental Results

(Solution-1a & 1b)

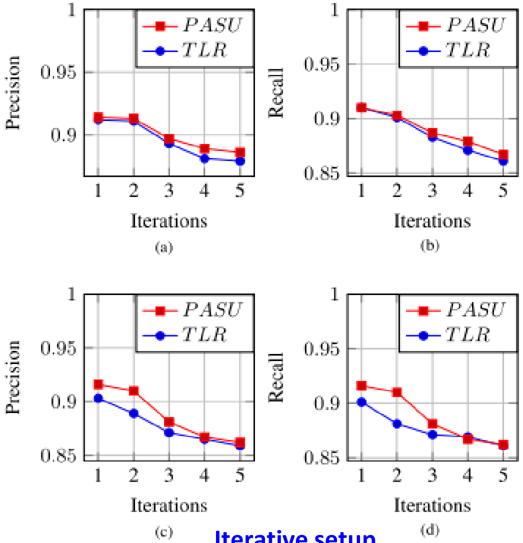
Maintenance Quality$_{(1/2)}$

- **Goal:** The goal is to evaluate the effectiveness of the maintenance policy in order to keep the caches up-to-date.
- **Single Setup:** In single setup, we utilized the quality of the updates performed by the maintenance for a single iterations.
- **Iterative Setup:** In an iterative approach, the goal is to estimate the accuracy, how good is update policy for maintaining up-to-date caches for longer period of time.





Single setup



Iterative setup

Findings

- The proposed approach reported to outperform other approaches by achieving an **F-measure score** of **90%**.

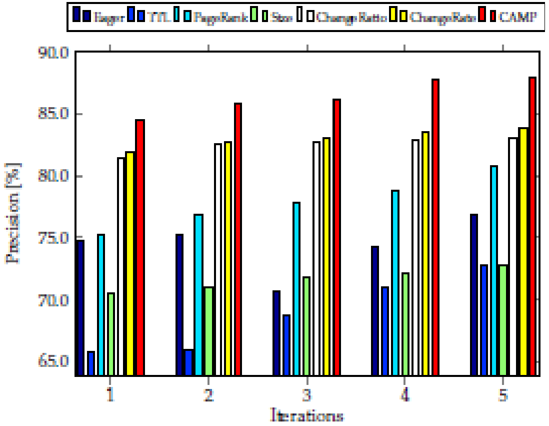
Accuracy measurement

- **Most accurate:** proposed approach
  - Maintain the caches up-to-date
- All other strategies shows the uniform loss of the quality
  - Execute irrelevant updates.
  - Massive amount of overhead, resulting in low a low effectivity.
- **Worst accuracy:** PageRank and Age based strategies

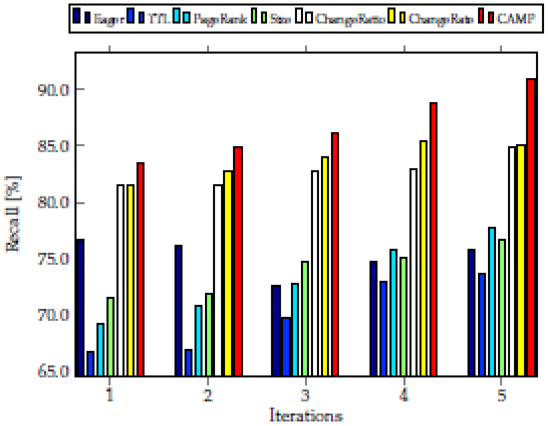
Experimental Results

(Solution-1a & 1b)

- **Maintenance Quality$_{(2/2)}$**
  - **Goal:** We utilized the quality of the updates performed by maintenance policies under consideration.
- **Accuracy measurement**
  - **Most accurate:** Proposed approach
    - Our proposed approach outperform the existing approaches, achieving 91% (precision) and 89% (reca accuracy in LOD datasets.
- **Findings**
  - Our proposed approach only updates the relevant data sources with less overhead and delays.
    - PageRank, Size, TTL performed worst because these strategies were executing the irrelevant queries.
    - ChangeRatio and ChangeRate only captured changed items and their efficiency degraded with each iteration overtime.
- **Summary**
    - Our approach only executed the relevant data updates with less drop and delay.
    - Existing strategies execute massive amount of overhead, resulting in low effectivity.



(a) Precision on DYLDO data



(b) Recall on DYLDO data

Figure. Quality of Updates performed by the proposed approach as compared to the existing approaches.

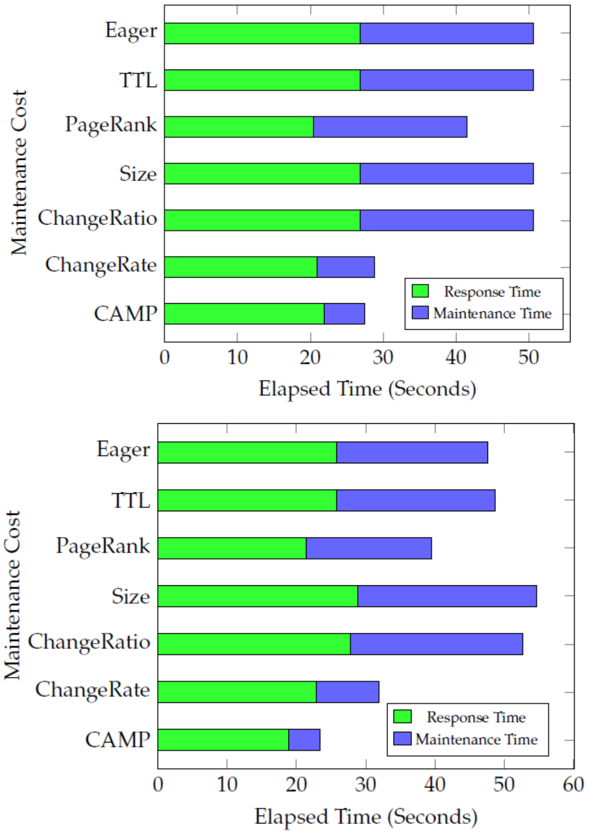
Table. Evaluating the effectivity of the update strategies.

Update strategies	Total Query Execution	Irrelevant	Relevant	Effectivity	Runtime (sec)
PageRank	32,690	30,650	2,040	6.20	800
Size	28,521	16,448	12,072	42.3	560
Age	29,128	10,560	18,960	63.9	500
ChangeRatio	29,550	9,800	19,750	66.3	320
ChangeRate	27,690	4,500	25,062	75.6	220
Proposed	**19,250**	**1,900**	**27,890**	**93.5**	**33**

Experimental Results

(Solution-1a & 1b)

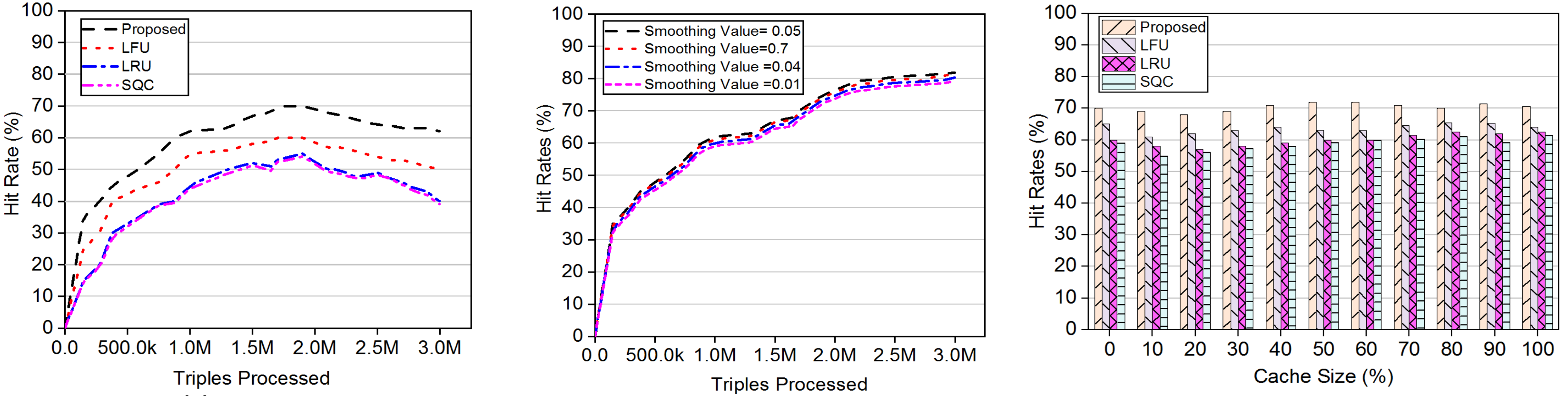
- **Maintenance Cost**
  - **Goal:** The goal of the maintenance cost is the time taken to perform maintenance operations.
- **Performance measurement**
  - **Response Time:** Proposed approach
    - As compared with the existing approach, we perform offline maintenance task with less response time.
  - **Maintenance Time:** Proposed approach
    - The maintenance time is less as compared to the existing approaches.
    - Existing policies often run in the background, they produced high latency.
- **Findings**
  - Existing approaches perform maintenance during system peak time.
    - Strategies such as PageRank, Size, eager performed worst because they trigger the maintenance during system peak time.
    - The proposed approach produce a lower elapsed time of 5s as compared to state-of-the-art approaches.
- **Summary**
  - As compared with the existing approach, existing approach produce less response time while update the local data caches.



Showing the comparison with other state-of-the-art approaches on **(a) DYLDO** and **(b) BTC** datasets.

Experimental Results

(Solution - 2)



(a) Hit rates **(b)** Exponential smoothing constant **(c)** Varying size of the cache

Hit rate achieved by proposed approach as compared to LRU, LFU and SQC algorithms

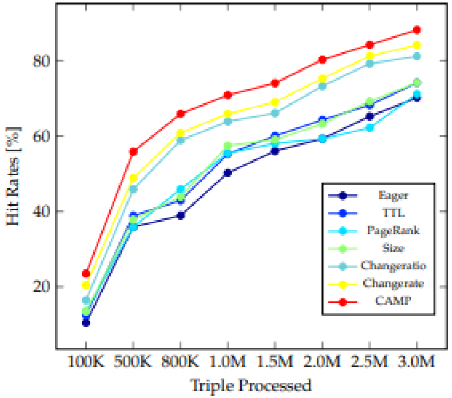
- **Cache Hit Rates (1/2)**
  - **Goal:** The goal is to measure the **performance** of query time in terms of better hit rates.
- **Performance Evaluation**
  - **Hit rates comparison**
    - Existing approaches such as **LRU (Least Recently Used)**, **LFU (Least Frequently Used)** and **SQC (SPARQL Query Caching)** and measure the efficiency in terms of average hit rate.
    - The proposed approach performed better in terms of **higher hit rates** with varying size of the cache.

- **Findings**
  - The proposed approach **increase** the hit rates by **5.46%** and **reduce** the **query time** by **6.34%**.
    - LFU technique remains accurate for cache with small size.
    - We noticed that the choice of the smoothing constant effect the accuracy, therefore we have set its value to **0.05**.

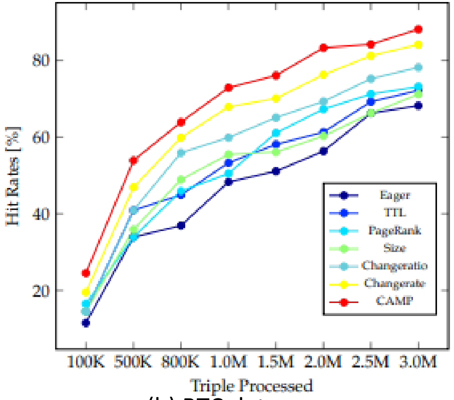
Experimental Results

(Solution - 2)

- **Cache Hit Rates** (2/2)
  - **Goal:** We measure the **performance** of query times in terms of better hit rates.
- **Performance Evaluation**
  - **Hit rates: Proposed approach**
    - As compared with the existing approach, our approach perform better in terms of **higher** hit rates.
- **Findings**
  - On average, the proposed approach out perform the existing approaches in terms of higher hit rates.
    - Proposed approach achieved higher hit rates (90%) as compared to the **Eager** (70%), **ChangeRate** (69%).
    - **PageRank**, **TTL** and **size** performed worst in term of cache hit rates.
- **Summary**
  - Effectiveness of the proposed approach to state-of-the-art approaches, namely **eager**, **TTL**, **PageRank**, **Size**, **ChangeRatio** and **ChangeRate**.
  - The proposed approach outperform the existing approaches in terms of lower maintenance cost, higher maintenance quality and better hit rates.



(a) DYLDO data



(b) BTC data

Showing the comparison of hit rates (a) DYLDO and (b) BTC datasets.

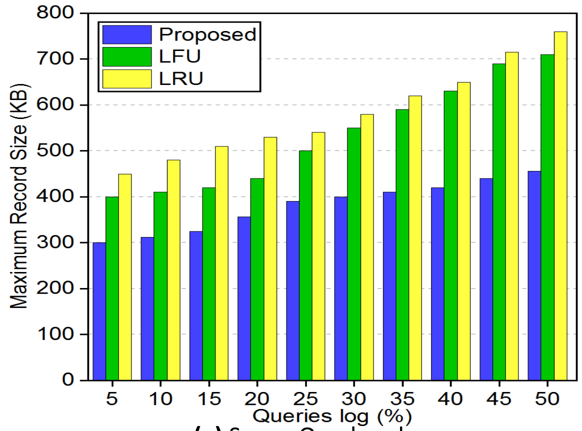
Experimental Results

(Solution - 2)

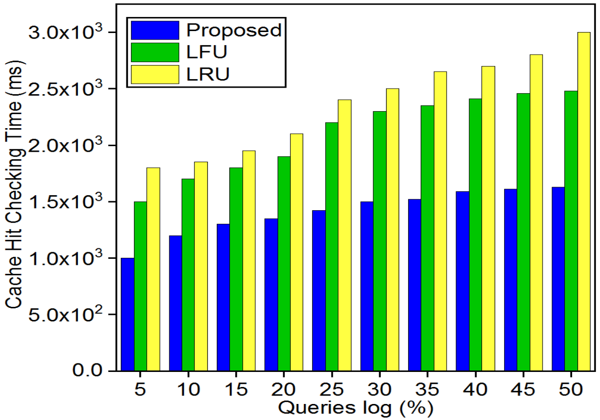
- **Space and Time Overhead comparison**
  - **Goal:** The goal of this evaluation is to measure the **space** and **time** overhead of the proposed approach.

- **Performance Evaluation**
  - **Space & time overhead comparison**
    - We measure the maximum **space consumption** of each approach based on the maximum number of the query records each algorithm stores in its cache.
    - Proposed approach consume **less space**, and also the query response time is better then existing.

- **Findings**
  - On average the hit checking time of the proposed approach is **280ms**, which is **10 times better** than other approaches.
    - Strategies such as LFU and LRU performed worst in case of space overhead.
    - In case of the time overhead, LRU and LFU take higher checking time as compared to the proposed approach.



(a) Space Overheads

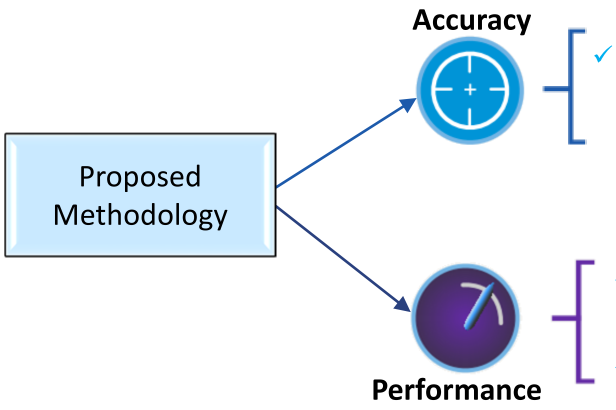


(b) Time Overheads

Showing the overhead comparison of proposed approach

Conclusion

- **Improve the accuracy and querying performance of Linked Open Data cloud.**



- ✓ Higher accuracy of the proposed maintenance policy by achieving an F-measure score of 90%.

- ✓ Increase the hit rates by 5.46% and reduce the query time by 6.34%.
- ✓ Less overhead (10 times better)

- **Future works**
  - Improved methods for effective ways of prefetching by utilizing the parallelizing algorithms to run on separate machine.
  - Improved the storing and querying over evolving data and replace the cache replacement during system idle time.

Publications

Published Status

- **Patent (01)**
  - **Domestic:** (01)
- **SCI / SCIE Journals (09)**
  - **SCI:** First Author (01)
    - **Computing (SCI)**, Springer (**IF: 2.25**)
  - **SCIE**: First Author (02)
    - **IEEE** Access (**SCIE**) (**IF: 3.7**)
    - **MDPI**, Applied Science (**SCIE**) (**IF: 2.47**)
  - **Co-author**: (06)
- **Conferences (12)**
  - **International:**
    - First Author (06)
    - Co-author (02)
  - **Domestic**
    - First author (04)

Total Publications

(27)

First Author Publications

(18)

Submitted Status

- **Patent (04)**
  - **Domestic:** 04
- **Conference (01)**
  - **International**
    - First author: 01

Selected References

1. Folz, Pauline, Hala Skaf-Molli, and Pascal Molli. "**CyCLaDEs: a decentralized cache for triple pattern fragments.**" In *European semantic web conference*, pp. 455-469. Springer, Cham, 2016.
2. Gyrard, Amelie, Martin Serrano, and Ghislain A. Atemezing. "**Semantic web methodologies, best practices and ontology engineering applied to Internet of Things.**" In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 412-417. IEEE, 2015.
3. Dividino, Renata Queiroz, Thomas Gottron, Ansgar Scherp, and Gerd Gröner. "**From Changes to Dynamics: Dynamics Analysis of Linked Open Data Sources.**" In *PROFILES@ ESWC*. 2014.
4. Dividino, Renata, Thomas Gottron, and Ansgar Scherp. "**Strategies for efficiently keeping local linked open data caches up-to-date**." In *International Semantic Web Conference*, pp. 356-373. Springer, Cham, 2015.
5. Verborgh, Ruben, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. "**Web-Scale Querying through Linked Data Fragments.**" In *LDOW*. 2014.
6. Folz, Pauline, Hala Skaf-Molli, and Pascal Molli. "**CyCLaDEs: a decentralized cache for Linked Data Fragments.**" In *ESWC: Extended Semantic Web Conference*. 2016.
7. Akhtar, Usman, Anita Sant'Anna, and Sungyoung Lee. "**A Dynamic, Cost-Aware, Optimized Maintenance Policy for Interactive Exploration of Linked Data.**" *Applied Sciences* 9, no. 22 (2019): 4818.
8. Akhtar, Usman, Anita Sant'Anna, Chang-Ho Jihn, Muhammad Asif Razzaq, Jaehun Bang, and Sungyoung Lee. "**A cache-based method to improve query performance of linked Open Data cloud**." *Computing* (2020): 1-21.
9. Dividino, Renata Queiroz, and Gerd Gröner. "**Which of the following SPARQL Queries are Similar? Why?**." In *LD4IE@ ISWC*. 2013.
10. Lorey, Johannes, and Felix Naumann. "**Detecting SPARQL query templates for data prefetching.**" In *Extended Semantic Web Conference*, pp. 124-139. Springer, Berlin, Heidelberg, 2013.
11. Chun, Sejin, Jooik Jung, and Kyong-Ho Lee. "**Proactive policy for efficiently updating join views on continuous queries over data streams and linked data.**" *IEEE Access* 7 (2019): 86226-86241.
12. Akhtar, Usman, Muhammad Asif Razzaq, Ubaid Ur Rehman, Muhammad Bilal Amin, Wajahat Ali Khan, Eui-Nam Huh, and Sungyoung Lee. "**Change-aware scheduling for effectively updating linked open data caches.**" *IEEE Access* 6 (2018): 65862-65873.
13. Verborgh, Ruben, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. "**Web-Scale Querying through Linked Data Fragments.**" In *LDOW*. 2014.
14. Hasan, Rakebul. "Predicting SPARQL query performance and explaining linked data." In *European Semantic Web Conference*, pp. 795-805. Springer, Cham, 2014.
15. Zhang, Wei Emma, Quan Z. Sheng, Yongrui Qin, Kerry Taylor, and Lina Yao. "Learning-based SPARQL query performance modeling and prediction." *World Wide Web* 21, no. 4 (2018): 1015-1035.
16. Martin, Michael, Jörg Unbehauen, and Sören Auer. "**Improving the performance of semantic web applications with SPARQL query caching**." In *Extended Semantic Web Conference*, pp. 304-318. Springer, Berlin, Heidelberg, 2010.
17. Yang, Mengdong, and Gang Wu. "**Caching intermediate result of SPARQL queries**." In *Proceedings of the 20th international conference companion on World wide web*, pp. 159-160. 2011.
18. Dar, Shaul, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, and Michael Tan. "**Semantic data caching and replacement**." In *VLDB*, vol. 96, pp. 330-341. 1996.
19. Shu, Yanfeng, Michael Compton, Heiko Müller, and Kerry Taylor. "**Towards content-aware SPARQL query caching for semantic web applications.**" In *International Conference on Web Information Systems Engineering*, pp. 320-329. Springer, Berlin, Heidelberg, 2013.
20. Papailiou, Nikolaos, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. "**Graph-aware, workload-adaptive SPARQL query caching**." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1777-1792. 2015.
21. Lehmann, Jens, and Lorenz Bühmann. "**Autosparql: Let users query your knowledge base.**" In *Extended semantic web conference*, pp. 63-79. Springer, Berlin, Heidelberg, 2011.

Thank you
for your attention

Q & A ?

Appendix

Idea Diagram

(Challenge 1 & 2)

1 Existing Method

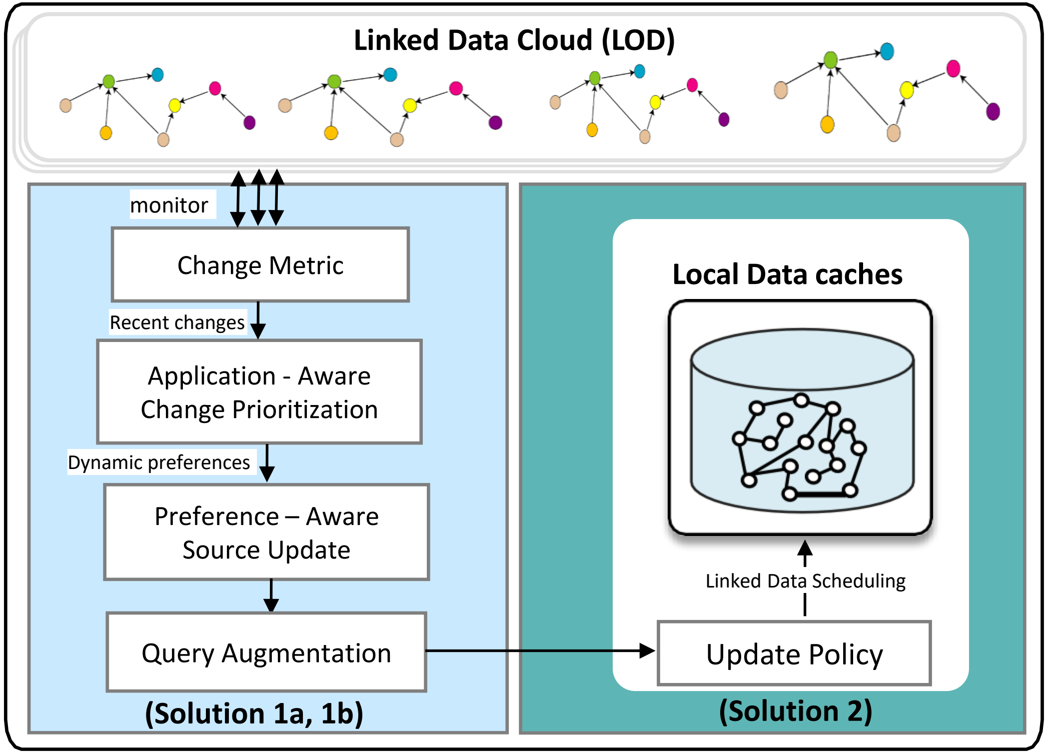
- Unable to quantify the evolution of LOD
- Unable to prioritize recent changes occurs in the cloud
- Aggravate the burden on query endpoint by issuing similar structure queries.

(Challenge 3)

2 Existing Method

- Unable to update the cache prior to the query execution
- Update of the local data caches during system peak time.
- Time based cache replacement is inefficient to replace content from cache.

Proposed approach - Overview



(Solution 1a & 1b)

1a Proposed Solution

- + A dynamic function to deal with the evolution of the Linked Open Data
- + A change-aware algorithm to update the content of local data caches.

1b Proposed Solution

- + Query augmentation to alleviate the burden on the server
- + Prefetching the result of similar queries for future access

(Solution 2)

2 Proposed Solution

- + Frequency-based cache replacement to eliminate less valuable content from cache.