SPARQL Query via Subgraph Matching

Siliang Xia

xiasiliang.str@gmail.com

- Introduction
- Problem
- Background and Related Work
- gStore: subgraph matching
- Evaluation
- Summary

Introduction

- RDF, SPARQL
  - The *RDF* (Resource Description Framework) data model was proposed for modeling Web objects as part of developing the semantic web.
  - In order to query RDF repositories, *SPARQL* query language has been proposed by W3C

  - RDF Storage and Indexing
  - Storage Service is part of Infrastructure service

Problem

- Answer SPARQL query over RDF data set
  - Given a certain query, return qualified data pieces from given data set

Related work

- Different Perspectives on RDF storage
  - The relational perspective
  - The entity perspective
  - The graph-based perspective

Related work

- The relational perspective.
  - simply to store all RDF triples in a single table over the relation schema since RDF data is of the form (subject, predicate, object)

Subject	Predict	Object
y:Abraham_Lincoln	hasName	"Abraham Lincoln"
y:Abraham_Lincoln	BornOnDate	"1809-02-12"
y:Abraham_Lincoln	DiedOnDate	1865-04-15
y:Abraham_Lincoln	DiedIn	y:Washington_D.C
y:Washington_D.C	hasName	"Washington D.C."
y:Washington_D.C	FoundYear	1790
y:Washington_D.C	rdf:type	y:city
y:United_States	hasName	"United States"
y:United_States	hasCapital	y:Washington_D.C
y:United_States	rdf:type	Country
y:Reese_Witherspoon	rdf:type	y:Actor
y:Reese_Witherspoon	BornOnDate	"1976-03-22"
y:Reese_Witherspoon	BornIn	y:New_Orleans,_Louisiana
y:Reese_Witherspoon	hasName	"ReeseWitherspoon"
y:New_Orleans,_Louisiana	FoundYear	1718
y:New_Orleans,_Louisiana	rdf:type	y:city
y:New_Orleans,_Louisiana	locatedIn	y:United_States

Related work

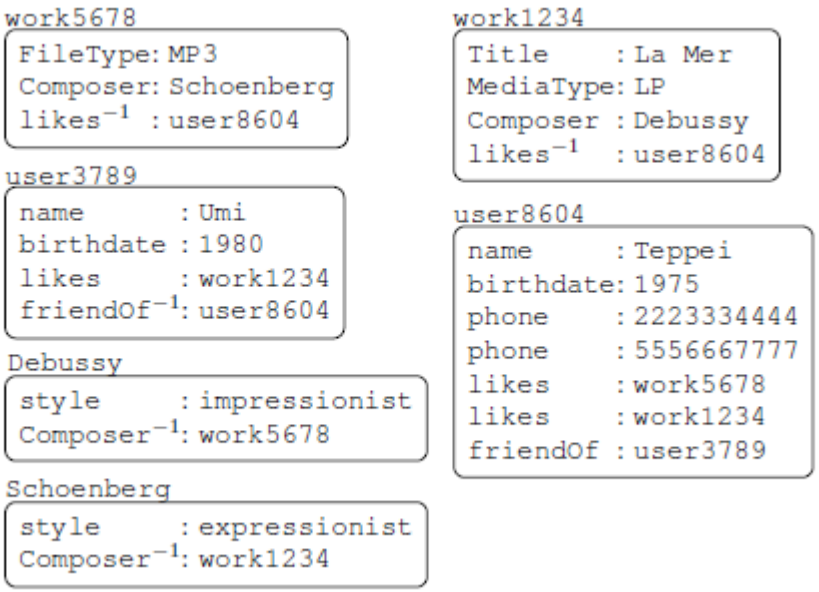
- The relational perspective.
- *1. One big triple table*
  - Store all RDF triples in a single three-column table
  - Manipulate all RDF triples in a uniform manner
  - Limitation: Require performing lots of self-joins over this table to answer a SPARQL query.
  - Some efforts to address this issue, such as, RDF-3x and Hexastore, which build several clustered B+-trees as the index for all 6 permutations of three columns.

Related work

- The relational perspective.
- *2. Vertically partitioned tables*
  - For each property, this approach builds a single two-column (subject, object) table ordered by
  - The advantage of the ordering is to perform fast merge join during query processing
  - this approach does not scale well as the number of properties increases.

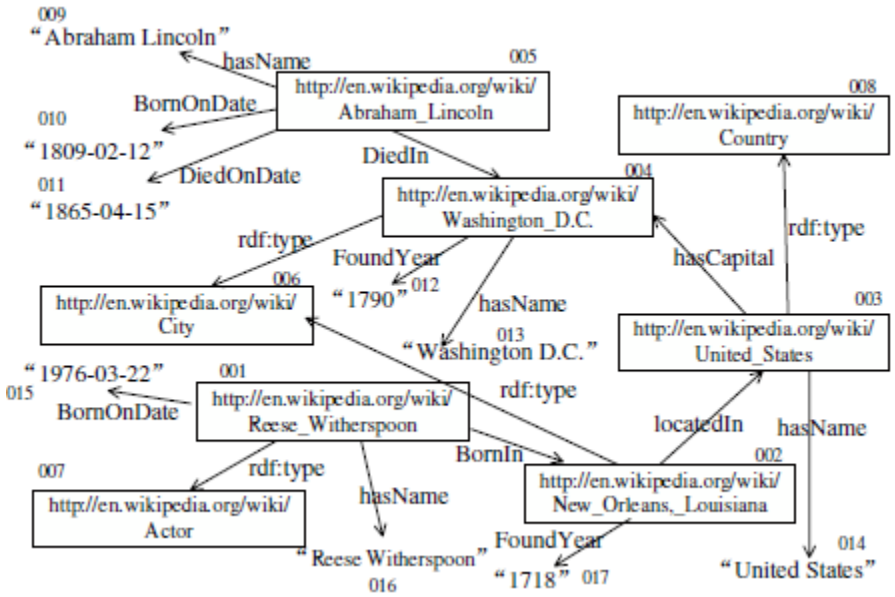
Related work

- The entity perspective.
  - resources in the RDF graph are interpreted as objects, or entities.
  - each entity is determined by a set of attribute-value pairs in the entity perspective



Related work

- The graph-based perspective
  - subjects and objects form the nodes
  - triples specify directed, labeled edges
  - focus is on supporting navigation in the RDF graph



Related work: limitations

- 1. they cannot answer SPARQL queries with wildcards in a scalable manner
  - Jena, Yars2, and Sesame 2.0: cannot work well in large RDF datasets, not scalable
  - SW-store RDF-3x: store RDF triples by replacing all literals with ids. In this way, they can only support exact queries. support only exact queries, can not deal with wildcards eg. reg(“*abc*”)

Related work: limitations

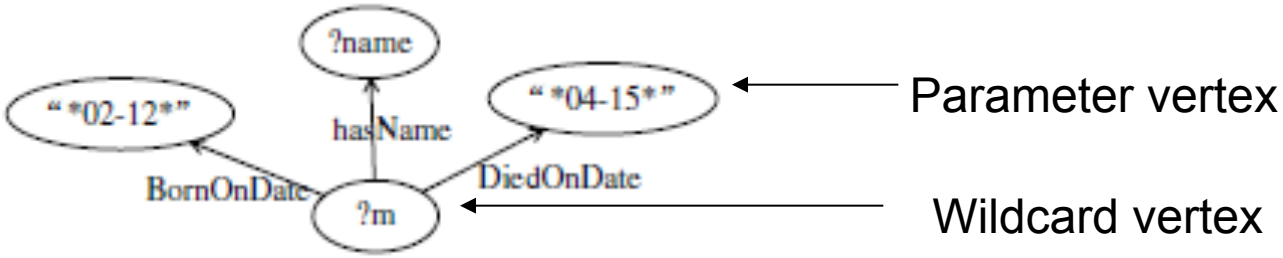
- 2. they cannot handle frequent updates in RDF repositories
  - it is very difficult for some existing systems to handle frequent updates in RDF repositories, forcing them to reprocess the dataset from scratch when there is an update.

gStore: overview

- Treat RDF datasets from a graph perspective
  - *RDF Graph*: a labeled, directed multiedge graph

gStore: overview

- Given a SPARQL query, we can also represent it by a query graph, *Q*.
- A SPARQL query is transformed into a subgraph matching query over a large RDF graph



gStore: overview

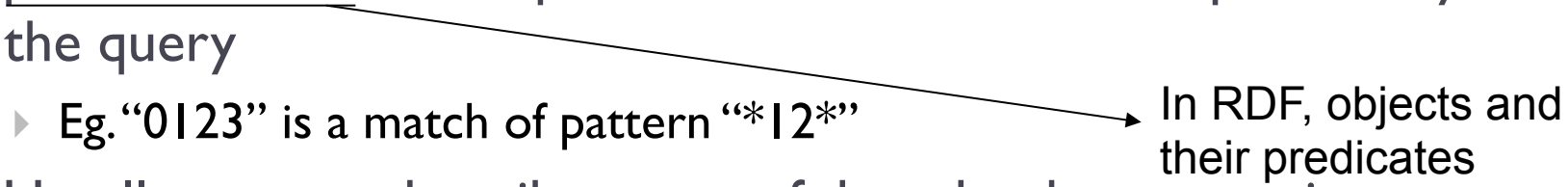
- Formal defination of RDF graph and Query graph

DEFINITION 2.1. *A* RDF graph *is denoted as* $G = \langle V, L_V, E, L_E \rangle$, *where (1)* $V = V_c \cup V_e \cup V_l$ *is a collection of vertices that correspond to all subjects and objects in RDF data, where* V_c, V_e, *and* V_l *are collections of class vertices, entity vertices, and literal vertices, respectively. (2)* L_V *is a collection of vertex labels. Given a vertex* $v \in V_l$, *its vertex label is its literal value. Given a vertex* $v \in V_c \cup V_e$, *its vertex label is its corresponding URI. (3)* $E = (v_1, v_2)$ *is a collection of directed edges that connect the corresponding subjects and objects. (4)* L_E *is a collection of edge labels. Given an edge* $e \in E$, *its edge label is its corresponding property.*

DEFINITION 2.2. *A* query graph *is denoted as* $Q = \langle V, L_V, E, L_E \rangle$, *where (1)* $V = V_c \cup V_e \cup V_l \cup V_p \cup V_w$ *is collection of vertices that correspond to all subjects and objects in a SPARQL query, where* V_p *and* V_w *are collections of parameter vertices and wildcard vertices, respectively, and* V_c *and* V_e *and* V_l *are defined in Definition 2.1. (2)* L_V *is a collection of vertex labels. For a vertex* $v \in V_p$, *its vertex label is* ϕ. *The vertex label of a vertex* $v \in V_w$ *is the substring without the wildcard. A vertex* $v \in V_c \cup V_e \cup V_l$ *is defined in Definition 2.1. (3) E and* L_E *are defined in Definition 2.1.*

gStore: overview

- What is a match ?
  - pieced of data that is qualified for certain criteria specified by the query
    - Eg. "0123" is a match of pattern "*12*"
  - Usually a query describes a set of data that has a certain common pattern.
    - Eg: regex expression describes the pattern of a string
  - Graph view of SPARQL processing: A SPARQL query is a graph pattern
  - Ansering SPARQL query is to find all instances (subgraph) from a given data graph G matching the pattern specified by a query graph Q



gStore: overview

- Formalized Mathematical definition of a *Match of a query*

DEFINITION 2.3. *Consider an RDF graph G and a query graph Q that has n vertices* $\{v_1, ..., v_n\}$. *A set of n distinct vertices* $\{u_1, ..., u_n\}$ *in G is said to be a* match *of Q, if and only if the following conditions hold:*

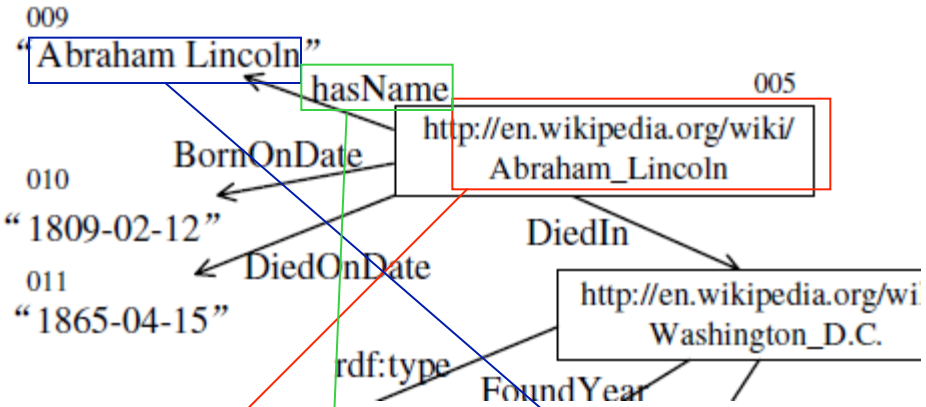
1. *If* v_i *is a literal vertex,* v_i *and* u_i *have the same literal value;*
2. *If* v_i *is an entity or class vertex,* v_i *and* u_i *have the same URI;*
3. *If* v_i *is a parameter vertex, there is no constraint over* u_i*;*
4. *If* v_i *is a wildcard vertex,* v_i *is a substring of* u_i *and* u_i *is a literal value.*
5. *If there is an edge from* v_i *to* v_j *in Q with the property p, there is also an edge from* u_i *to* u_j *in G with the same property p.*

gStore: Framework

- Offline process:
  - 1. represent an RDF dataset by an RDF graph *G*
  - 2. store it by its adjacency list table *T*
  - 3. encode each entity and class vertex into a bitstring (called *vertex signature*), link these vertex signatures to form a *data signature graph G**
  - 4. also encode query into a query signature graph *Q**
- Online process
  - 1. Indexing with VS*-Tree
  - 2. Perform query algorithm

gStore: Storage

- *Adjacent List:* {(eLabel, nLabel)+}
  - vertex *u* is represented by an adjacency list [*vID*, *vLabel*, *adjList*]



vLabel	adjList {(eLabel, nLabel)+}
y:Abraham_Lincoln	(hasName, "Abraham Lincoln") (BornOnDate, "1809-02-12"), (DiedOnDate, "1865-04-15") (DiedIn, y:Washington_D.C)
y:Washington_D.C	(hasName, "Washington D.C.") (FoundYear , "1790") (rdf:type, y:city)
y:United_States	(hasName, "United States") (hasCapital,y:Washington_D.C) (rdf:type, y:country)
y:Reese_Witherspoon	(hasName, "ReeseWitherspoon") (BornOnDate, "1976-03-22") (hasCapital, y:New_Orleans,_Louisiana) (rdf:type, y:Actor)
y:New_Orleans,_Louisiana	(FoundYear, "1718"), (locatedIn, y:United_States) (rdf:type, y:city)

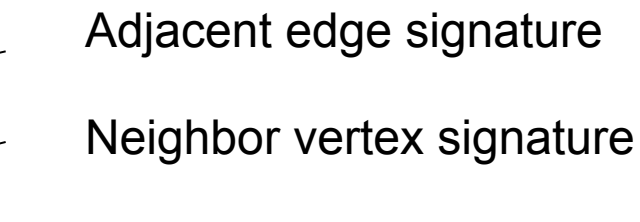
gStore: Encoding

- vertex to *vertex signature* (bitstring)
  - Straightforward idea: label -> bitstring
    - H("y:Abraham_Lincoln") -> "0000 0001"
  - But a vertex in the graph should not be simply characterized as its label
  - Their adjacent edges and neighbor vertices matter (if a vertex *v* (in query *Q*) can match a vertex *u* (in RDF graph *G*), each neighbor vertex and each adjacent edge of *v* should match to some neighbor vertex and some adjacent edge of *u*.)

1. Why the straightforward way does not work ?

gStore: Encoding

- vertex to data signature (bitstring)
  - encode each of its adjacent edge labels and the corresponding neighbor vertex to *edge signature*
  - Generate a vertex signature from all edge signatures of its adjacent edges to characterize a vertex, by performing bitwise OR operation



  - $e(eLabel, nLabel) \rightarrow eSig(e)$
  - $eSig(e) = eS\ ig(e).e, eS\ ig(e).n$
  - $\Rightarrow vS\ ig(v) = eS\ ig(e_1)|\ eS\ ig(e_2)|.....|eS\ ig(e_n)$

Why OR?

gStore: Encoding

- Eg: has_name "Abraham_Lincoln"
- Adjacent edge to its edge signature
  - H("has_name") -> "0010 0010 0000"
- Neighbor vertex to its signature
  - H("Abraham_Lincoln") -> "01000 0010 0010 0100
- Concatenate them to get an edge signature

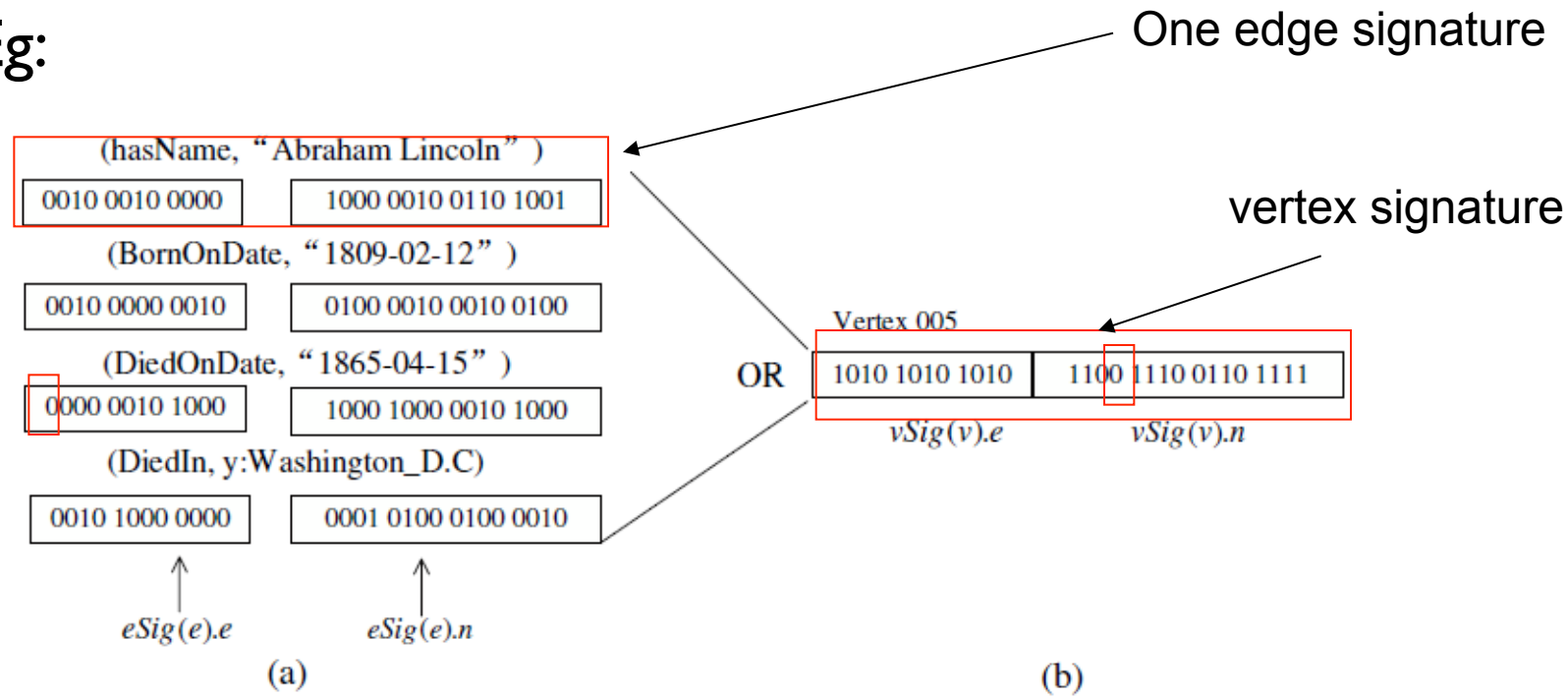
m of M bits are set to one

  - *n-gram:"Abraham_Lincoln" -> "Abr-bra-rah…col-oln"
  - Some other tricks, determine Parameter M, m, n

How to choose hash functions?
eg. BKDR hash, AP hash

gStore: Encoding

- Eg:



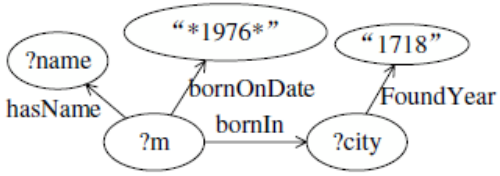
gStore: Encoding

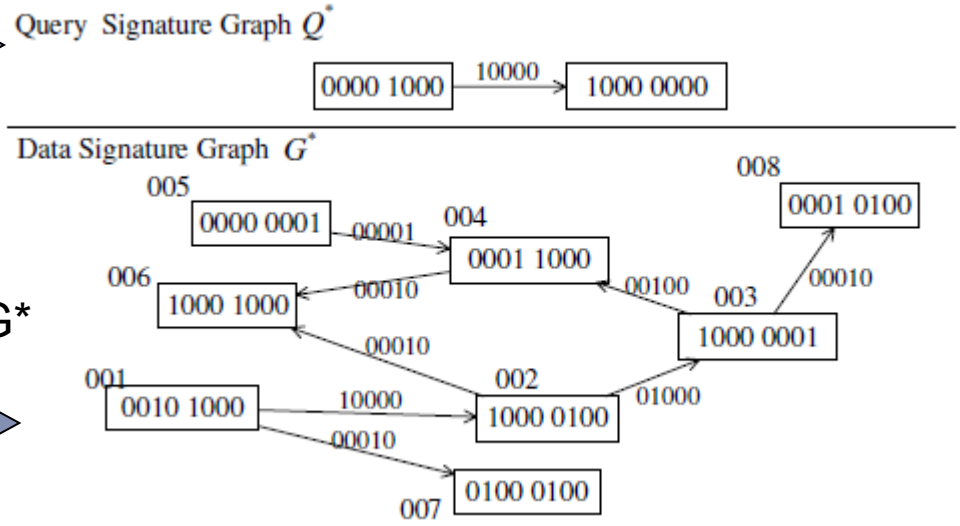
- Offline process:

Question becomes finding match of Q* over G*

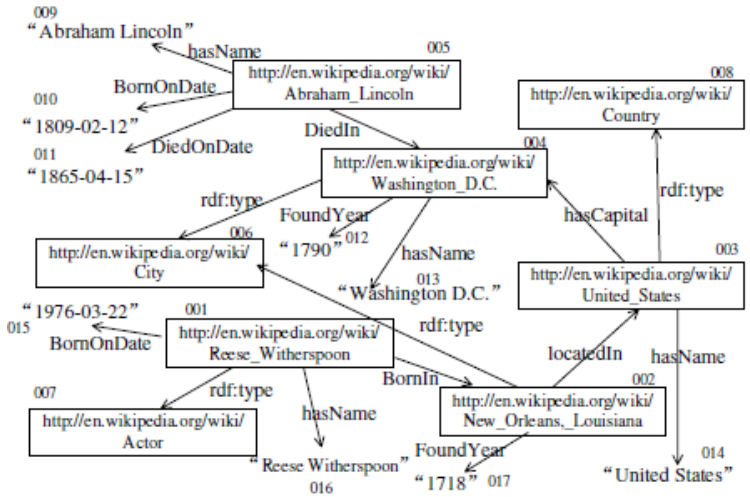
Q to Q*

Query Signature Graph





G to G*



Data Signature Graph

gStore: subgraph matching

- In this way, we can verify the match between *Q* and *G (RS)* by simply checking the match between corresponding encoded bitstrings
- Transfer the vertex edge matching problem to bitstring matching problem between Q* and G* (CL)
  - NP hard !!!

2. They claim (RS) ⊆ CL, Where is the difference comes form?

gStore: Indexing

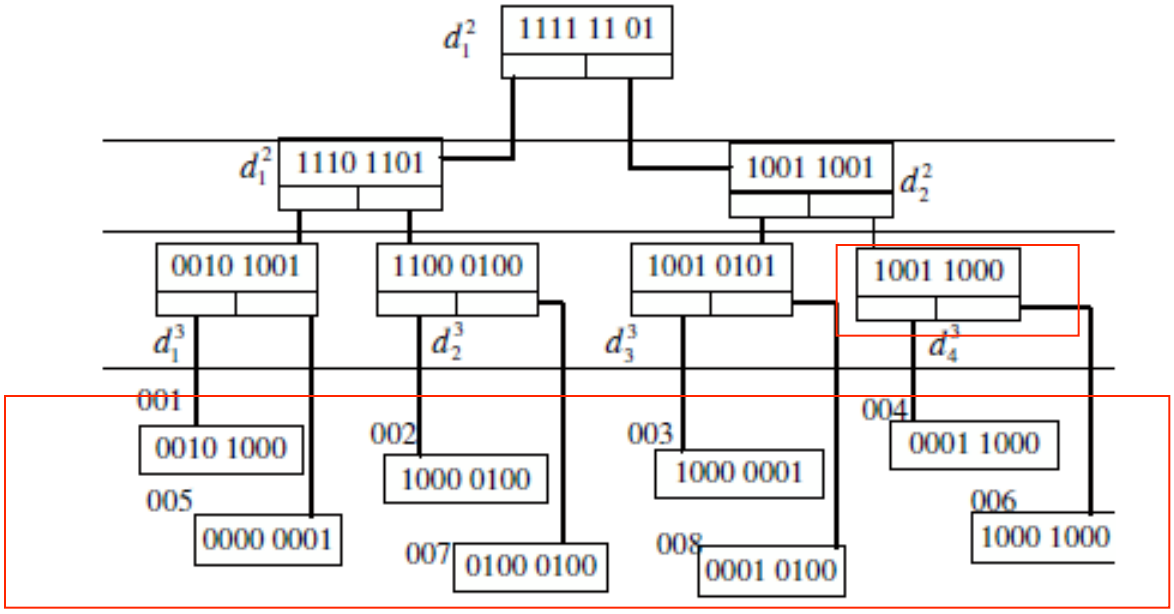
- S-Tree, VS-Tree, VS*-Tree

gStore: Indexing

- Problem
  - Eg: find “11” in {“00”,“01”,“10”,“11”}
  - Motivation: reduce the search space, find a match efficiently
  - Building index instead of enumeration

gStore: Indexing

- *S-Tree*



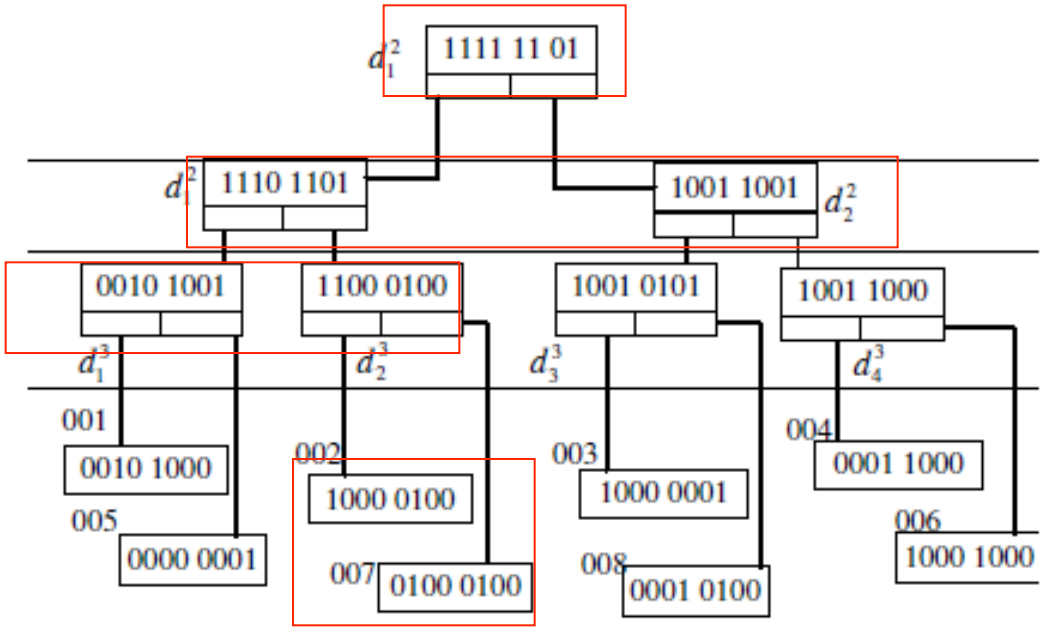
From child nodes by performing bitwise OR operation

Vertices in G*

Why OR ?

gStore: Indexing

- Search using S-Tree
  - Find a matching
    - k: Search key
    - n: label of current node



Go descending the tree iff (k AND n == k)

Eg: k = 0100 0100

1. n = 1111 1101

2.

n = 1110 1101 d^2_1

n = 1001 1001 (does't hold the test condition), subtree of d^2_2 is pruned

3. Similarly only d^3_2 holds
4. Finally only node 007 holds, find the key

Why AND ?

gStore: Indexing

- Why OR & AND operation
  - OR operation mains the *set ones* (the positions where bit '1' appears), which represents the character of the bitstring
  - And operation tests if current node contains the identical set ones of the search key
  - If test condition holds <=> current node contains the character of search key, which indicates the search space. It means a matching may be found among child nodes potentially
  - Eg:

Problem ?

Dab = 1111

Da = 0101 OR

Db = 1010

K=0101

K And Dab == K

=> go on searching the child nodes

it does not mean search key inevitably exists in his child nodes

K=0001

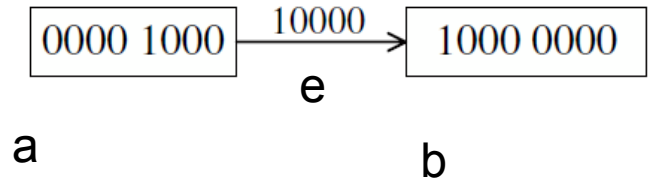
gStore: Indexing

- The key problem to be addressed is how to find matches of $Q*$ (query signature graph) over $G*$ (data signature graph) efficiently.
- A straightforward method
  - First, for each vertex $v_i \in V(Q*)$, we find a list $R_i = \{u_{i1}, u_{i2}, ..., u_{in}\}$, where $v_i \& u_{ij} = v_i$, $u_{ij} \in V(G*)$, and $u_{ij} \in R_i$.
  - Then, we perform a *multi-way join* over these lists R_i to find matches of $Q*$ over $G*$ the
  - Two S-Tree Join

gStore: Indexing

- Eg: Muilti-way join
  - We find node a in quert has a candidate list $R_a = \{u_{a1}, u_{a2}, ..., u_{an}\}$, in G*, node b has a candidate list $R_b = \{u_{b1}, u_{b2}, ..., u_{bn}\}$,
  - we test each element in Ra× Rb, whether there exists an edge from u_{ai} to u_{bi} qualified as edge e or not

Query Signature Graph Q^*



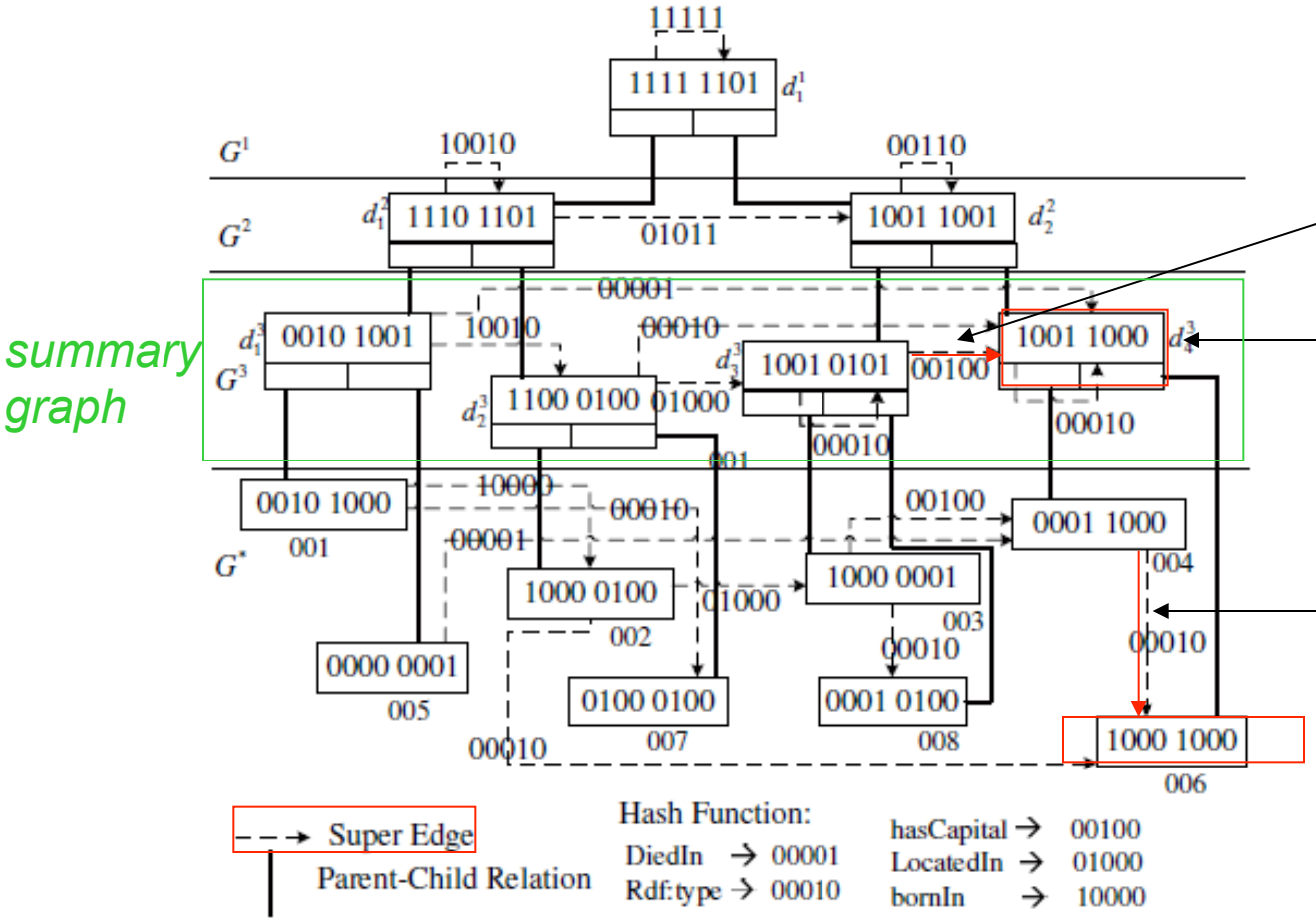
- first step (finding R_i) is a *classical inclusion query* can be solved by S-Tree, but S-tree cannot support the second step (multi-way join), which is NP-hard.

gStore: Indexing

- VS-Tree
  - a matching of graph is a combination of matching of edges and matching of vertices
  - Add *Super Edge* to S-Tree
  - => VS-Tree

gStore: Indexing

- VS-Tree



A super edge built from edges between child nodes

Built from child nodes by performing OR operation

edge in data signature graph G*

Vertex in data signature graph G*

gStore: Indexing

- VS-Tree
- Eg: Building super edge from low level super edges

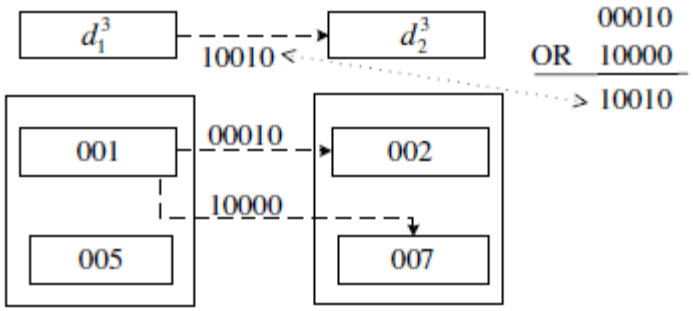


Figure 12: Building Super Edges

gStore: Indexing

- VS-Tree Building rules:
  - each leaf entry of S-tree corresponds to one vertex signature in $G*$.
  - link leaf entries according to $G*$'s structure: given two leaf entries d_1 and d_2 in a S-tree, we introduce an edge between them, if and only if there is an edge between u_1 and u_2 in $G*$, where d_1 (d_2) corresponds to u_1(u_2) in $G*$. We also introduce an edge signature $Sig(v_1v_2)$ as the edge label of d_1d_2 in a VS-tree
  - given two leaf nodes $d^l_1$and d^l_2 in the S-tree, we introduce a super edge from d^l1 to d^l_2, if and only if there is at least one edge from d^l_1's children to d^l_2's children

gStore: Algorithm

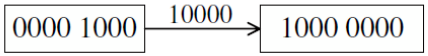
- *Summary Match*
  - *Consider a query signature graph Q∗ with n vertices v_i (i=1,...,n) and a summary graph G^l in the l-th level of VS-tree. A set of nodes $\{d^l_i\}$ (i = 1, ..., n) at G^l is called a* summary match *of Q∗ over G^l , if and only if the following conditions hold:*

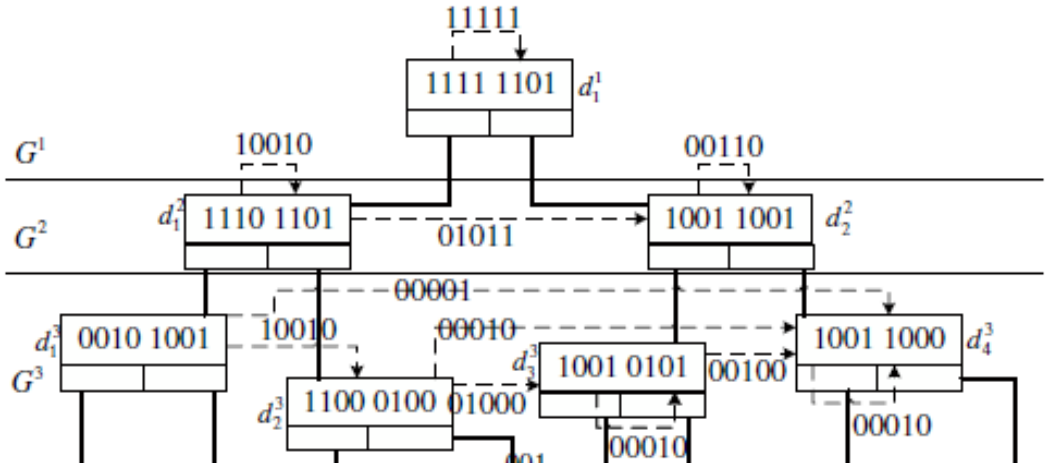
1. $vSig(v_i)\&d^I_i.Sig = vSig(v_i),\ i = 1, ..., n;$
2. *For any edge $\overrightarrow{v_1v_2}$ in Q^*, there must exist a super edge $\overrightarrow{d^I_1d^I_2}$ in G^I and $Sig(\overrightarrow{v_1v_2})\&Sig(\overrightarrow{d^I_1d^I_2}) = Sig(\overrightarrow{v_1v_2})$.*

gStore: Algorithm

- Eg: Summary Match
  - *0000 1000 AND d^l_1 == 0000 1000*
  - *1000 0000 AND d^l_1 == 1000 0000*
  - *10000 AND e(d^l_1 d^l_1) == 10000*
  - *=> (d^l_1, d^l_1) is a summary match of Q* in G^l*

Query Signature Graph Q^*





gStore: Algorithm

- Top-down search strategy over the VS-tree to find matches of $Q*$ over $G*$.
  - *Given a query signature graph $Q*$, a data signature graph $G*$ and VS-tree built over $G*$:*

1) Given a summary graph G^I in VS-tree, if there exists no summary match over G^I, there must exist no match of Q^* over G^*.

2) Assume that n vertices $\{u_1, ..., u_n\}$ forms a match (Definition 4.4) of Q^* over G^*. Given a summary graph G^I in VS-tree, u_i's ancestor in G^I is node d_i^I, $i = 1, ..., n$. $\{d_1^I, ..., d_n^I\}$ must form a summary match (Definition 5.1) of Q^* over G^I.

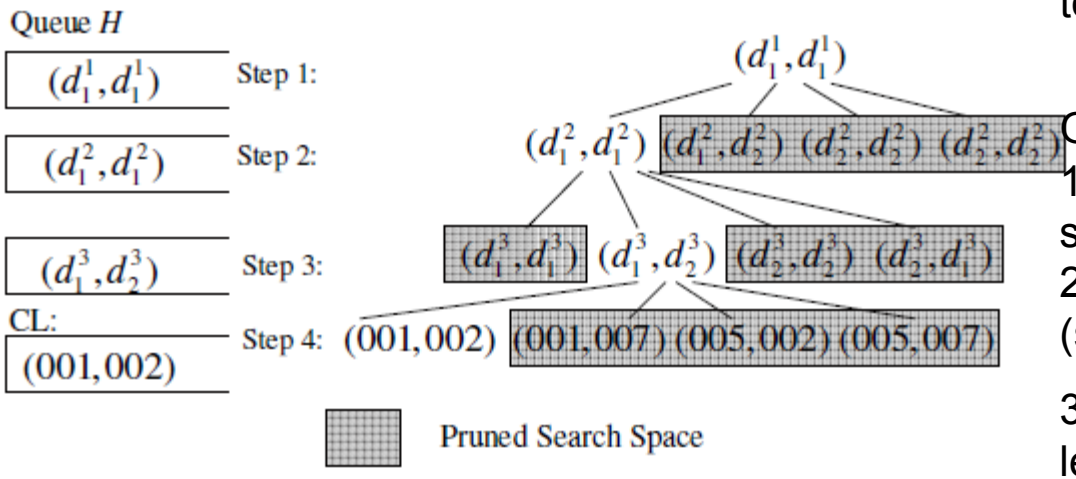
An apriori property

gStore: Algorithm

- *Child State*:
  - Given a summary match (d^l_1, d^l_1),its child states are formed by d^l_1.*children* × d^l_1.*children*
- *Valid Child State*:
  - For each child state, check whether it is a summary match of $Q*$. If so, we call it a *valid child state*.

gStore: Algorithm

- Broad First Search for summary match over VS-Tree (Traversing the VS-Tree by level order)



$G^{1:}$

(d^1_1, d^1_1), is a summary match, push it to the Queue

$G^{2:}$

1. Pop one element, test his child states;
2. (d^2_1, d^2_1) is a valid child state (summary match), push it to the Queue;
3. For each element in the Queue of level G^1, repeat two steps above, prune other child states which are not valid

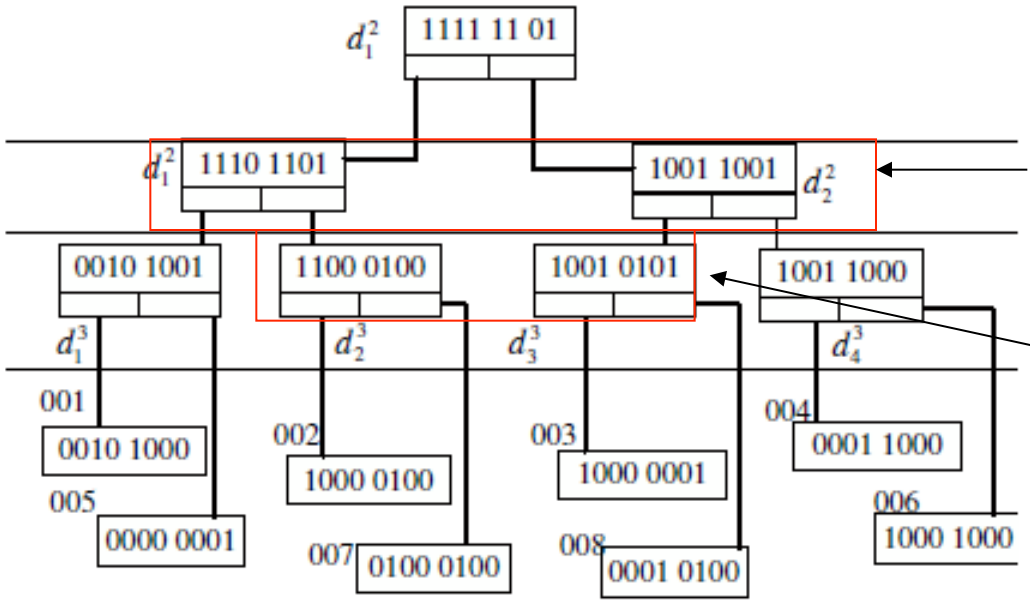
…

It applies the apriori property to prune

gStore: Improvement

- Consider another search key for the same S-Tree
  - *K = 1000 0100*

Problem: For some keys, the high level nodes in summary graph become less selective (low pruning power)



No pruning can be performed

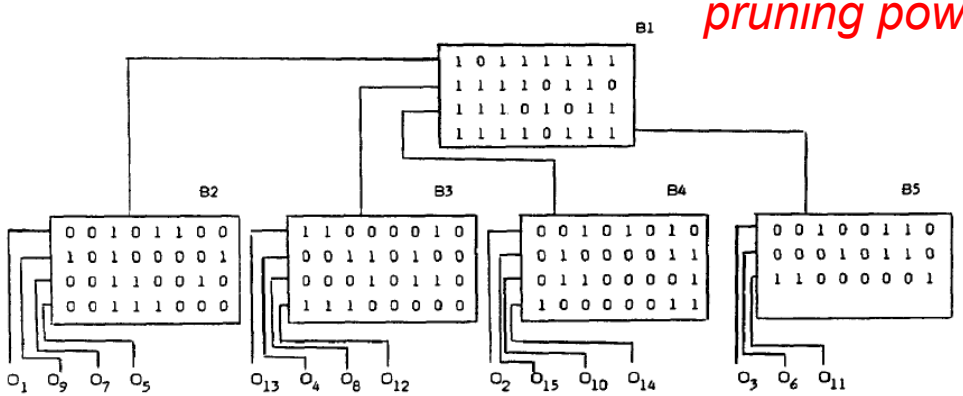
Still two nodes hold for test condition (k AND n == k)

What's the reason?

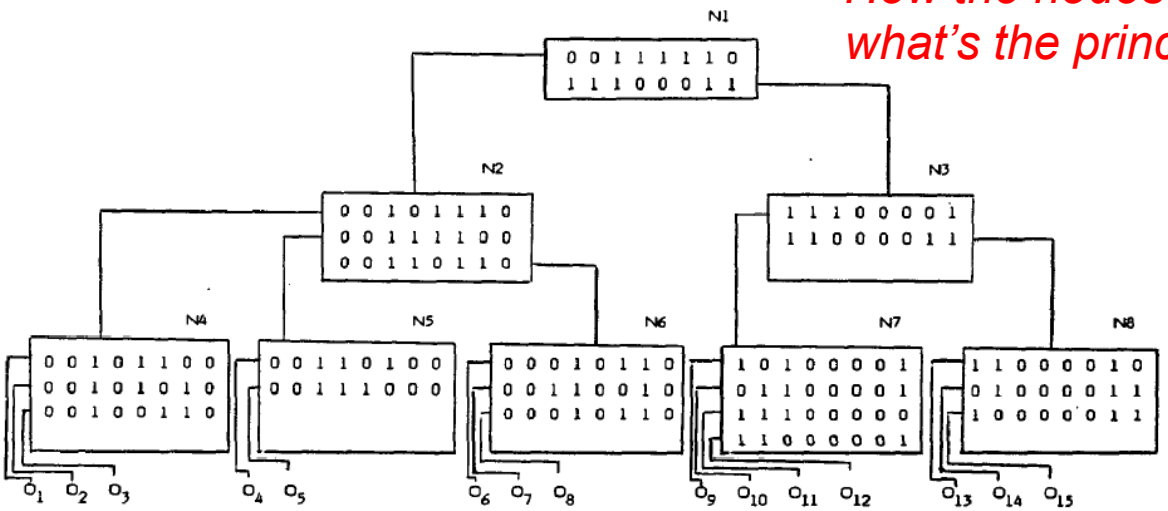
gStore: Improvement

- Which tree is a better index?
  - Eg: K = 1010 0010

How the nodes are organized affects the pruning power

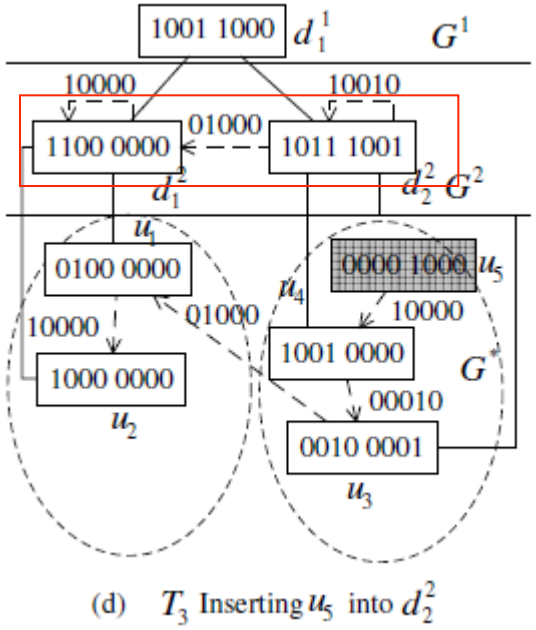


How the nodes should be organized, what's the principle to follow?

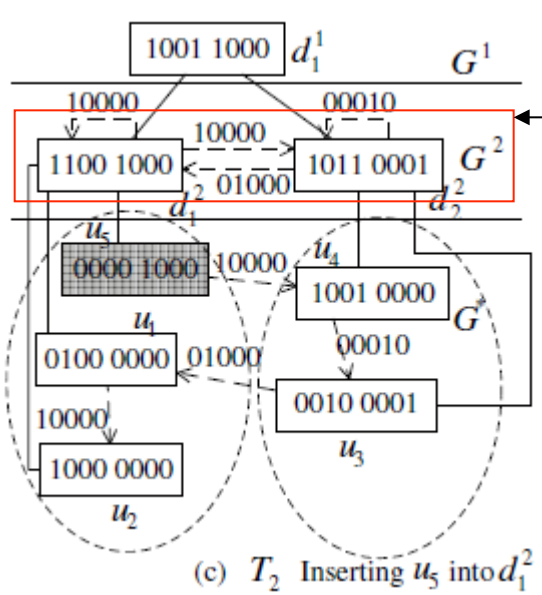


gStore: Improvement

- VS*-Tree
  - Motivation: In VS-Tree, not only the nodes, but also the super edge in higher level summary graph is affected when new child node is inserted



(d) T_3 Inserting u_5 into d_2^2



(c) T_2 Inserting u_5 into d_1^2

Cause additional super edge from d^2_2 to d^2_1

gStore: Improvement

- VS*-Tree: Insertion
- A metric to measure the distance to guide the new insertion
  - $\delta(u, d_i)$: the *Hamming distance* between u and d_i
  - $|u|$: is the length of the vertex signature (bitstring)
  - $\beta(u, d_i)$ is the number of newly introduced super edges adjacent to d_i , if u chooses node d_i.

$$Dist(u, d_i) = \frac{\delta(u, d_i)}{|u|} \times \frac{\beta(u, d_i)}{Max_{j=1}^{n}(\beta(u, d_j))}$$

- Find the node with smallest distance to new node to insert

gStore: Improvement

- VS*-Tree: Split
  - find two entities that have the maximal Hamming distance between them as two *seed* nodes
  - associate each left entry with the nearest seed node, according to Dist() metric
  - after node splitting, update the signatures and the super edges associated with the two new nodes
  - invokes insertions over the upper level of VS∗-tree, which also leads to the splitting that may be propagated to the root of the VS∗-tree.

gStore: Improvement

- VS*-Tree: Deletion
  - find the leaf node *d* where *u* is stored, the nodes along the path from the root down to *d* will be affected
  - bottom-up strategy to update the signature of and super edges associated with the nodes
  - After deletion, if some node *d* has less than the minimal fanout of node, then *d* is deleted and its entries are reinserted

gStore: Improvement

- VS*-Tree: Pruning Power of a summary graph
  - Determine which level of summary tree to begin the search for the least search space
  - $\prod_{j=1}^{j=m} |N(v_j, G^I)| - |N(Q^*, G^I)|$ denotes the search space that can be pruned

DEFINITION B.1. *Given a query signature graph Q^* with n edges e_i, $i = 1, ..., n$ and m vertices v_j, $j = 1, ..., m$, and summary graph G^I at the I-th level of VS*-tree, the* pruning power *of G^I with regard to Q^* is defined as follows:*

$$P(Q^*, G^I) = 1 - \frac{|N(Q^*, G^I)|}{\prod_{j=1}^{j=m} |N(v_j, G^I)|} \quad (2)$$

where $N(Q^*, G^I)$ denotes the set of summary matches of Q^* over G^I, and $N(v_j, G^I)$ denotes the set of nodes d (in G^I) and $(d \& v_j = d)$.

Remaining search space

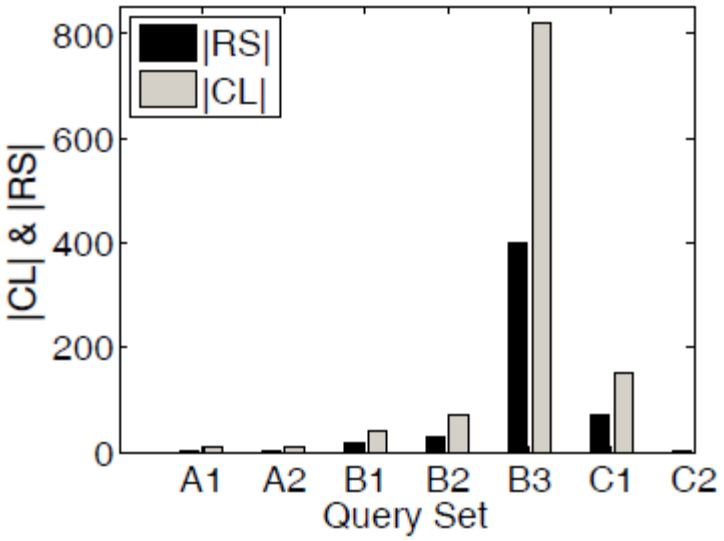
Total search space

gStore: Evaluation

- Offline Performance
  - Yago and DBLP
  - each dataset is first converted into a factorized form: one file *T* with RDF triples represented as integer triples, and one dictionary file *M* to map from ids to literals
  - All methods utilize the same input files and load them into their own systems.

gStore: Evaluation

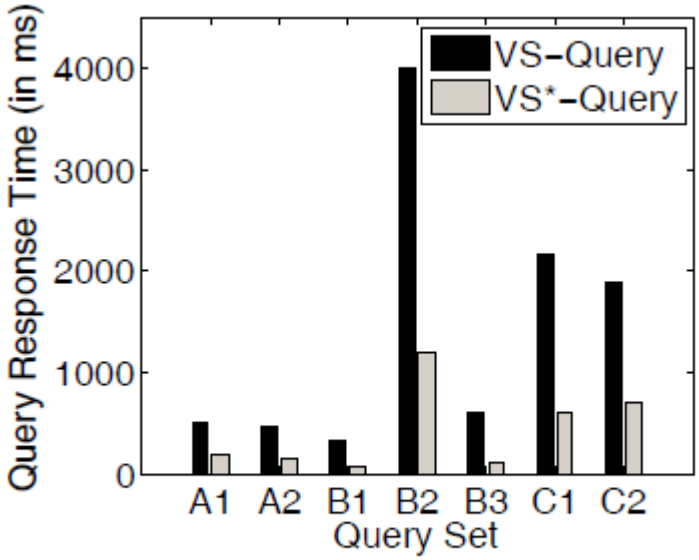
- Effect of Pruning Power
  - RS is the expected result data
  - CL is the candidate data pieces generated in gStore



(a) $|RS|$ VS. $|CL|$

gStore: Evaluation

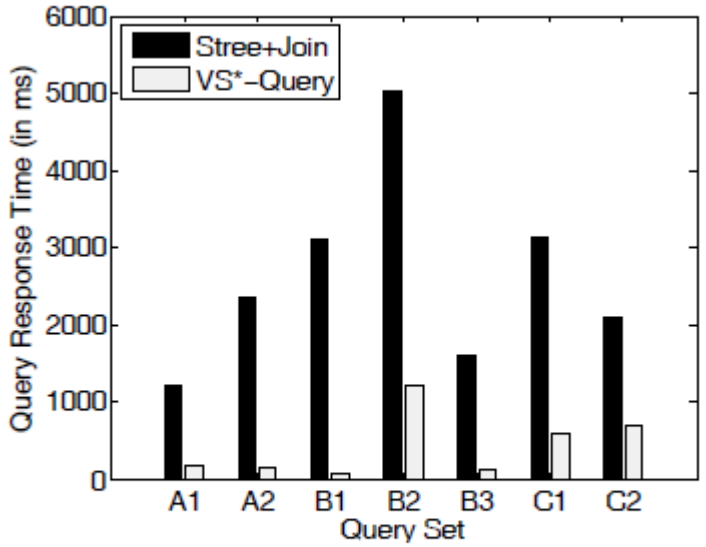
- VS-query Versus VS*-query



(b) Query Response Time

gStore: Evaluation

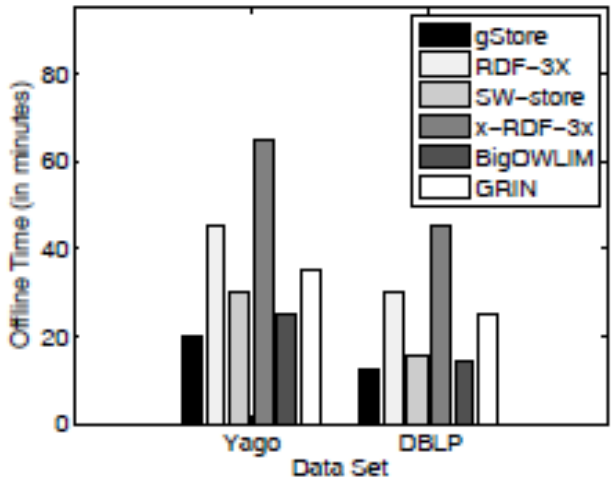
- S-Tree Join Versus VS*-query



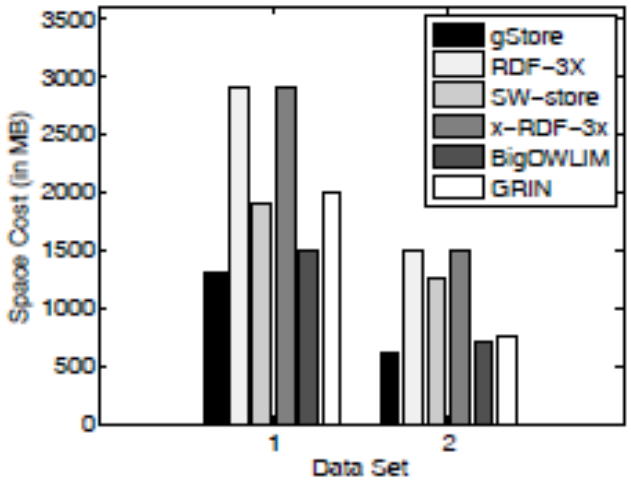
(a) Evaluating Stree+Join Method Over Yago

gStore: Evaluation

- Offline Performance
  - loading time: total offline processing time
  - total space cost: the size of the whole database including the corresponding indexes



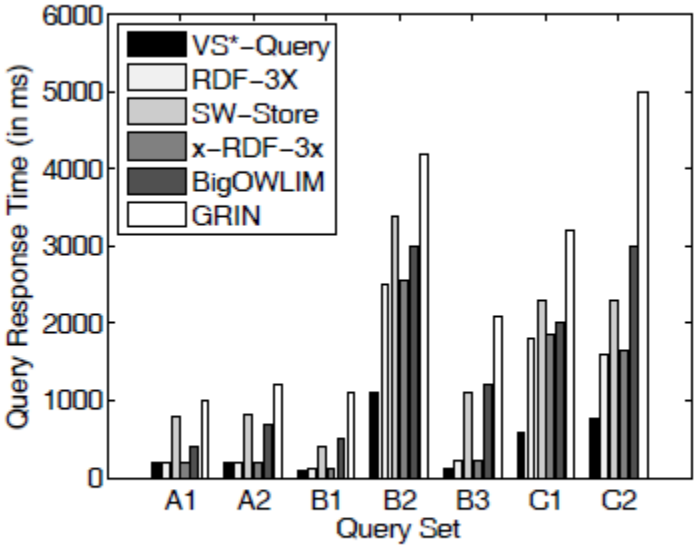
(a) Offline Processing Time



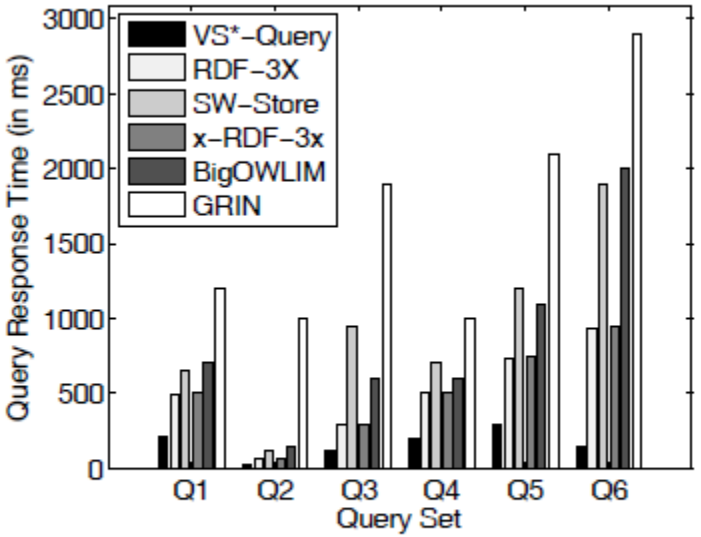
(b) DB Size

gStore: Evaluation

- Online Process
  - Query response time



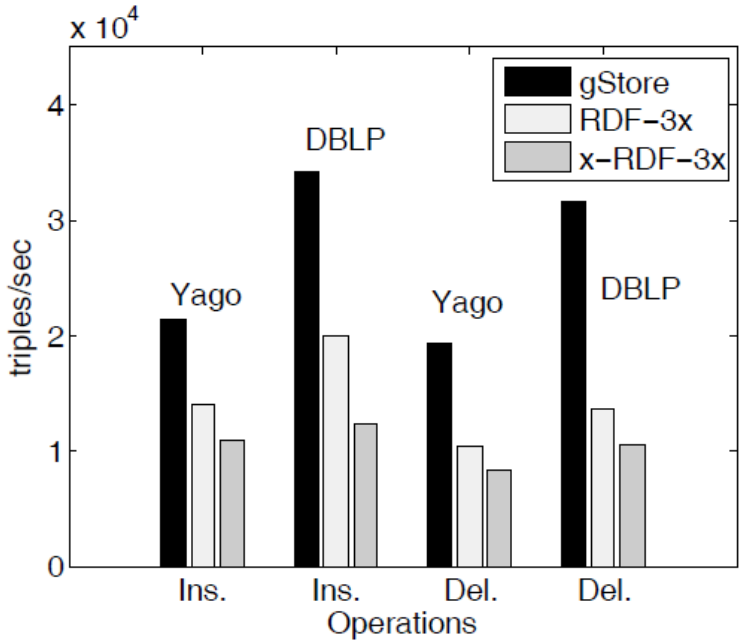
(a) Yago



(b) DBLP

gStore: Improvement

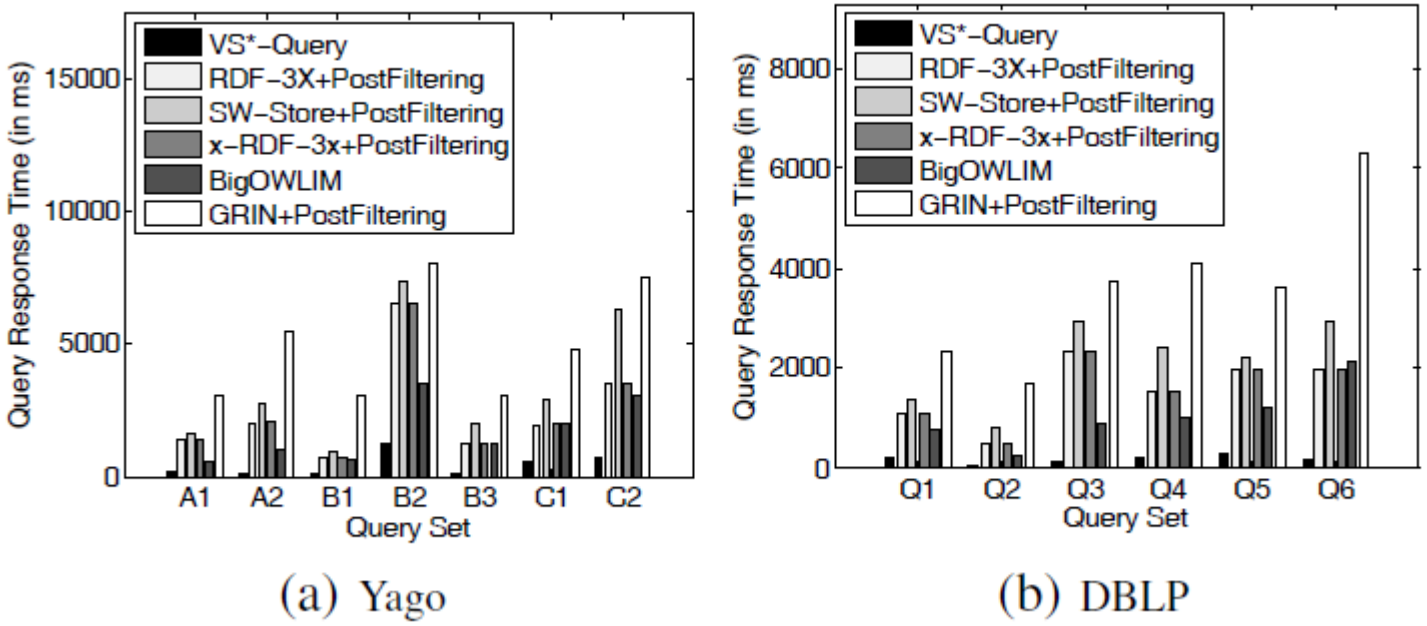
- Online Updates (insertion and deletion)



(b) Evaluating Online Updates

gStore: Evaluation

- Online Process
  - Query response time with wildcard



Summary

- Different perspective to RDF data
- SPARQL query process and difficulty of the problem
- related work and drawbacks
- gStore Appraoch: Subgraph matching
  - Storage (adjacent list) and encoding (bitstring)
  - Subgraph matching problem
  - Indexing with S-Tree, VS-Tree, VS*-Tree and search algorithm
  - Evaluation

Reference

- [1] gStore: Answering SPARQL Queries via Subgraph Matching
- [2] Storing and Indexing Massive RDF Data Sets
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.*Introduction to Algorithms*. The MIT Press, 2001.
- [4] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *SIGIR*, pages 77–87, 1986.
- [5] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas,S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a DBMS for approximate string processing. *IEEE Data Eng. Bull.*, 24(4):28–34, 2001.

Reference

- [6] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
- [7] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *WWW*, pages 90–101, 2003.
- [8] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, 2009.
- [9] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

Reference

- [10] T. Neumann and G. Weikum. X-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 1(1):256–263, 2010.