

Ge Song, Frédéric Magoulès (Supervisor), Fabrice Huet  
(Co-Supervisor)

Lab MICS

CentraleSupélec

Université Paris-Saclay

# Parallel and Continuous Join Processing for Data Streams

Thèse pour l'obtention du grade de docteur

# Table of Contents

Introduction

Part I: Data Driven Stream Join (kNN)

Part II: Query Driven Stream Join (RDF)

Conclusion and Future Work

# Introduction

- Google: 24 PB / day
- Facebook: 10 million photos + 3 billion “likes” / day
- Youtube: 800 million visitors / month

## Issues

- The size of Data.
- The flip side of size is speed.
- Transfer cost.
- **Dynamic data – Data Stream**

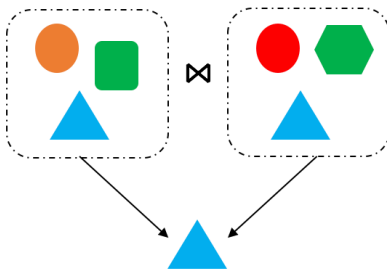
## Dynamic Data Stream

- Persistent **Static** Relations: **Batch-oriented** data processing
- Transient **Dynamic** Data Streams: Real-time **stream** processing
- **Architecture Level:** add or remove computational nodes based on the current load
- **Application Level:** withdraw old results and take new data into account

## Objective: parallel and continuous processing for Join operation

### Join:

- Find the common elements of several data sets under a specified condition.
- Popular and often used operation in the big data area.



## Type:

- Data Driven Join : kNN (Data Parallelism)
- Query Driven Join : Semantic Join on RDF data (Task Parallelism)

## Part I: Data Driven Stream Join (kNN)

- Introduction
- Survey about Parallel Solutions on MapReduce
  - Parallel Workflow
  - Theoretical Analysis
  - Experiment Result
- Continuous kNN
- Conclusion



# Outline

- Introduction
- Survey about Parallel Solutions
  - Parallel Workflow
  - Theoretical Analysis
  - Experiment Result
- Continuous kNN
- Conclusion

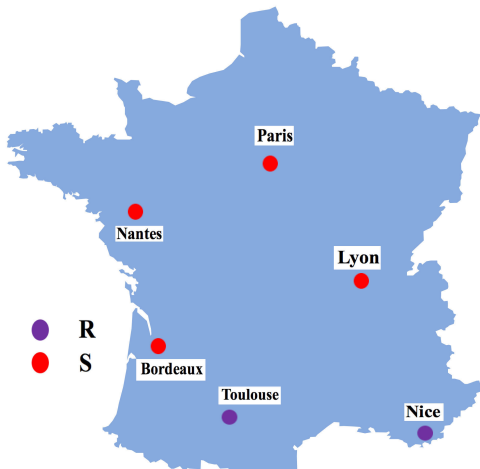
# Introduction

## Definition: kNN

Given a set of query points  $R$  and a set of reference points  $S$ , a  $k$  **nearest neighbor join** is an operation which, for each point in  $R$ , discovers the  $k$  nearest neighbors in  $S$ .

## An Example of kNN Join

For each city in R, find the nearest city in S. (1-NN)



- **R:** Query Set
- **S:** Reference Set

- Query never changes
- Data format changes: GPS (2 Dimensions), Twitter (77 Dimensions), Images (128 Dimensions) etc.

## Introduction: Basic Idea

- Nested Loop – Calculate the Distances

```
for(int r : R){  
    for(int s : S){  
        Distance(r, s);  
    }  
}
```

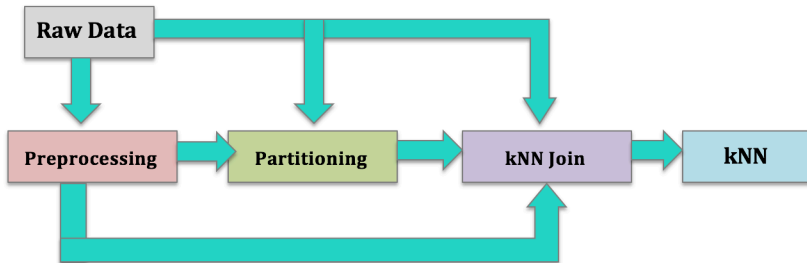
- Sort – Find the top k smallest distance for each element

**Problem: Data Intensive and Computation Intensive**

# Outline

- Introduction
- Survey about Parallel Solutions
  - Parallel Workflow
  - Theoretical Analysis
  - Experiment Result
- Continuous kNN
- Conclusion

## First Result: Parallel Workflow



## Data Preprocessing : Reduce the dimension of data

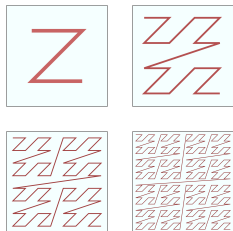
**The curse of dimensionality:** Too difficult to calculate in high dimension space.

**Solution:** Project data from high dimension space to a low dimension one while preserving the locality information

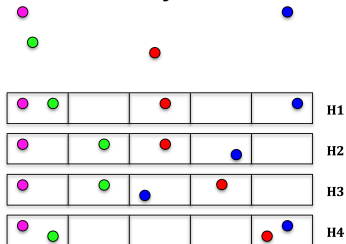


# Data Preprocessing – Reducing the dimension of data

## Space Filling Curve (Z-Value)



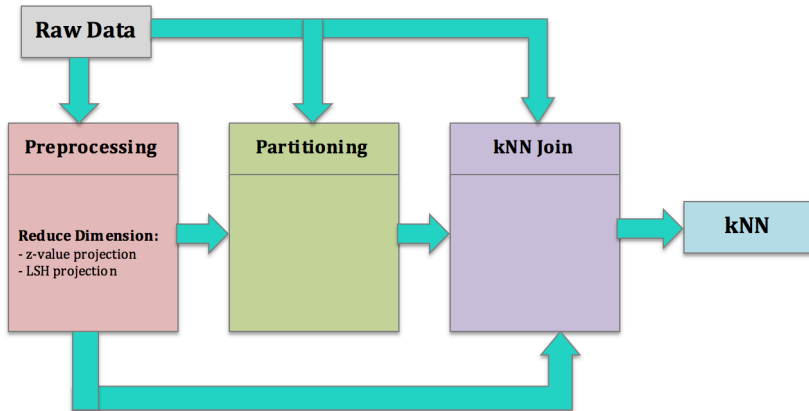
## LSH: Locality Sensitive Hashing



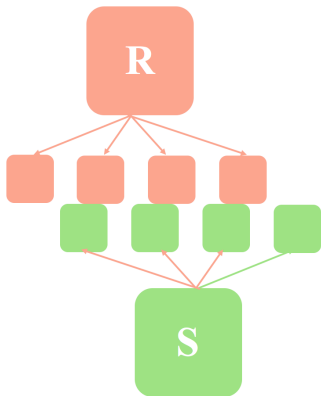
[z-value]: Efficient parallel kNN joins for large data in MapReduce, EDBT 2012, Chi Zhang et. al.

[LSH]: RankReduce - processing K-Nearest Neighbor queries on top of MapReduce, LSDS-IR 2010, Aleksandar Stupar et. al.

## Parallel Workflow

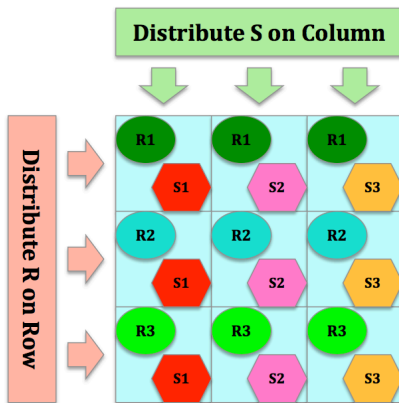


## Data Partitioning



**Purpose:** Cut big data sets into smaller ones

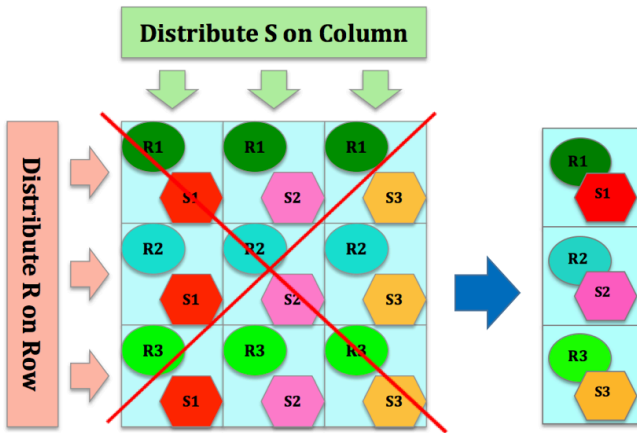
## Data Partitioning – Basic Idea (Block Nested Loop)



**Problem:**  $n^2$  tasks for calculating pairwise distances; wastes a lot of hardware resources, and ultimately leads to low efficiency.

## Data Partitioning – Motivation

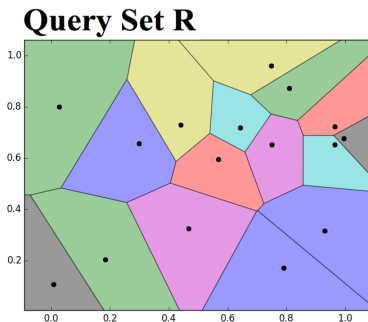
**Purpose:** Reduce the task number from  $n^2$  to  $n$



## Data Partitioning – Distance Based Partitioning Strategy – Voronoi Diagrams

This strategy wants to have the most relevant points in each partition.

- 1 Partition Query Set R
- 2 Use same diagrams to partition S

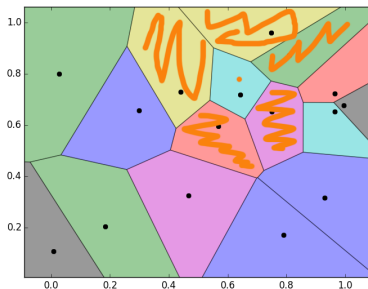


Take neighbor cells

## Data Partitioning – Distance Based Partitioning Strategy – Voronoi Diagrams

This strategy wants to have the most relevant points in each partition.

- 1 Partition Query Set R
- 2 Use same diagrams to partition S



Take neighbor cells

## Data Partitioning – Size Based Partitioning Strategy – Z-Value (or LSH)

This strategy wants to make each partition have equal size in order to achieve a good load balance.

- 1 A Sample of R
- 2 Partition the sample into equal sized partitions
- 3 Find corresponding partition in S for each R



Take 3 partitions



## Data Partitioning – Size Based Partitioning Strategy – Z-Value (or LSH)

This strategy wants to make each partition have equal size in order to achieve a good load balance.

- 1 A Sample of R
- 2 Partition the sample into equal sized partitions
- 3 Find corresponding partition in S for each R

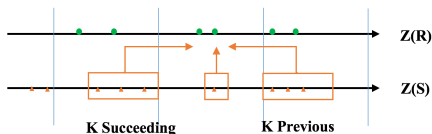


Take 3 partitions

## Data Partitioning – Size Based Partitioning Strategy – Z-Value (or LSH)

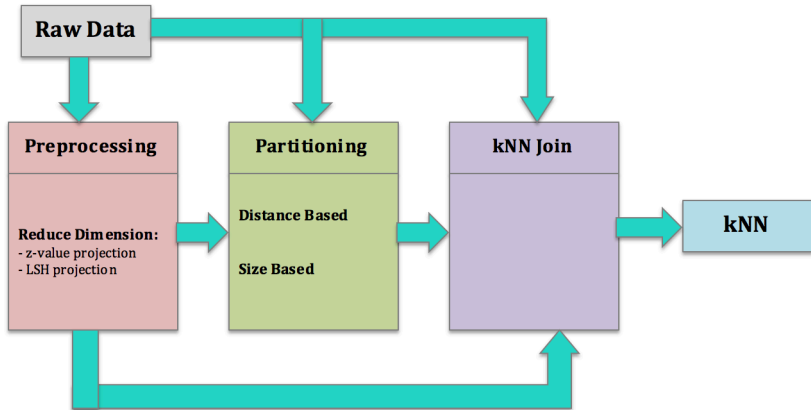
This strategy wants to make each partition have equal size in order to achieve a good load balance.

- 1 A Sample of R
- 2 Partition the sample into equal sized partitions
- 3 Find corresponding partition in S for each R



Take 3 partitions

## Parallel Workflow



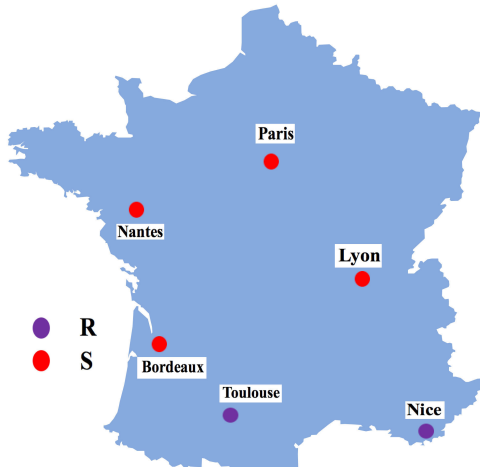
## Computation

- One job — Directly give the global results
- Two consecutive jobs — First give the local results, then merge the local results into the global results

**Purpose:** reduce the number of elements to be sorted.

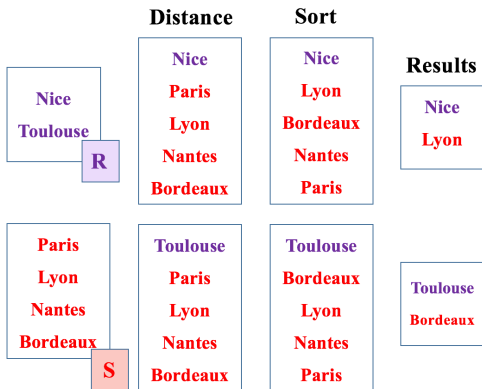
## Computation – Example

For each city in R, find the nearest city in S. (1-NN)



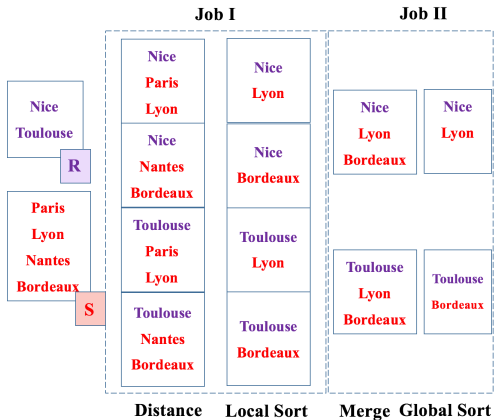
# Computation – Example

## One Job:

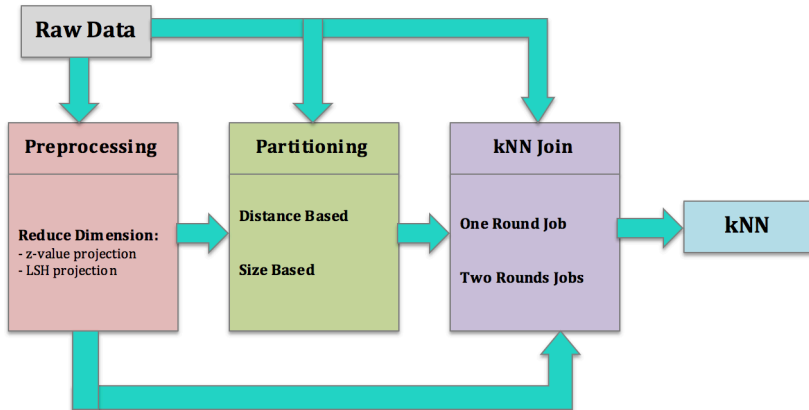


# Computation – Example

## Two Jobs:



## Parallel Workflow

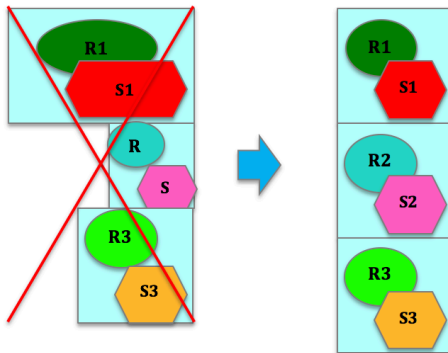




## Result 2: Theoretical Analysis

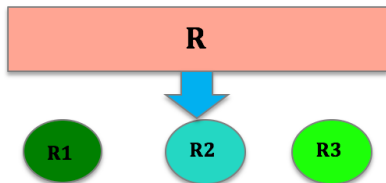
- Load Balance
- Accuracy
- Complexity

# Load Balance

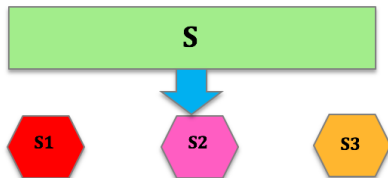


## Sub-Optimal Load Balance

Partition one, deduce the other:  
**(1) Partition R into equal-sized**



**(2) Partition S into equal-sized**



(1) is better in the worst case.

# Accuracy

Accuracy: the number of correct results we get in the end.

- **Z-Value**

- Depends on  $k$
- Shift of data — move data in the direction of a random vector
- Increase the number of shifts of data will decrease the error rate

- **LSH**

- Depends on parameter tuning
- Increase the number of hash functions will decrease the error rate

## Complexity

- **The number of MapReduce jobs:** starting a job requires some initial steps.
- **The number of Map tasks and Reduce tasks used to calculate  $kNN(R_i \bowtie S)$ :** the larger this number is, the more information is exchanged through the network.
- **The number of final candidates for each object  $r_i$ :** Finding the top  $k$  results is very time consuming.

## Result 3: Experimental Analysis

### Cluster Setting

- The experiments were run on two clusters of Grid'5000 with Hadoop 1.3 (3 replications, 1 slot per node)

### Datasets

- **OpenStreetMap** Geo dataset contains geographic XML data in two dimensions — Low Dimension
- **Caltech 101** It is a public set of images, which contains 101 categories of pictures of different objects. (Speeded Up Robust Features — 128 dimensions) — High Dimension

## Methods Evaluated

- **H-BkNNJ** Naive Method – Without preprocessing and partitioning – One Job
- **H-BNLJ** Block Nested Loop – Without preprocessing and partitioning – Two Jobs
- **PGBJ** Based on Voronoi – Preprocessing: Select Pivots – Distance Based Partitioning – One Job
- **H-zkNNJ** Based on Z-Value – Preprocessing: z-value – Size Based Partitioning – Two Jobs
- **RankReduce** Based on LSH – Preprocessing: LSH – Size Based Partitioning – Two Jobs

## Evaluation Result – Verify the theoretical Analysis

Execution Time for Geo dataset (2 dimensions):

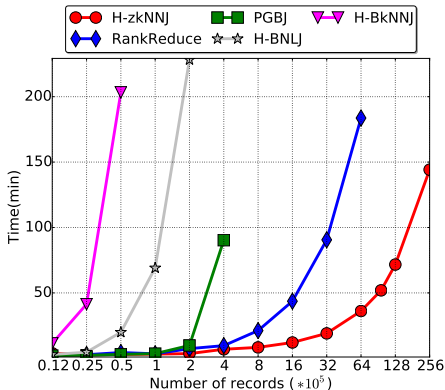
**H-BkNNJ: Naïve**

**H-BNLJ: Block Nested Loop**

**PGBJ: Voronoi**

**H-zkNNJ: z-value**

**RankReduce: LSH**





## Evaluation Result – Surprise

Execution Time for Image dataset (128 dimensions):

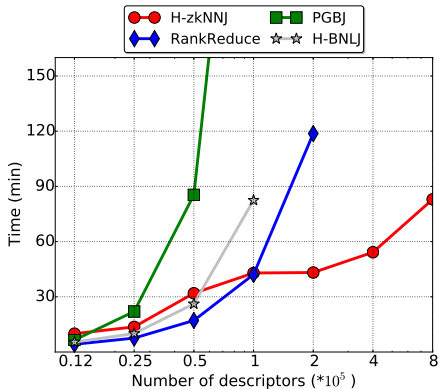
**H-BkNNJ: Naïve**

**H-BNLJ: Block Nested Loop**

**PGBJ: Voronoi**

**H-zkNNJ: z-value**

**RankReduce: LSH**



## Conclusions of the survey

- Clear and detailed view of the current algorithms for processing kNN on MapReduce
- A fine grained analysis both theoretical and experimental for each algorithm to obtain the best performance.

### **Limitation of the existing algorithms**

- Non of them can process data streams

# Outline

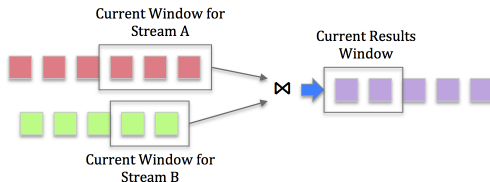
- Introduction
- Survey about Parallel Solutions
  - Parallel Workflow
  - Theoretical Analysis
  - Experiment Result
- Continuous kNN
- Conclusion

## Sliding Window Model – Motivation

- Unbounded sequence of elements which can not be wholly stored in bounded memory
- New items in a stream are more relevant than older ones.

### Sliding Window Model

Maintaining a moving window of the most recent elements in the stream



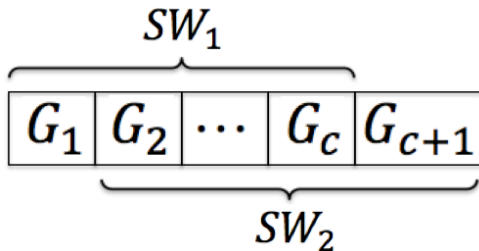
## Sliding Window – Two Strategies

- **Re-Execution Strategy**
  - **Eager Re-execution Strategies** — Generating new results right after each new data arrives
  - **Lazy Re-execution Strategies** — Re-Executing the query periodically
- **Data Invalidation Strategy**
  - **Eager Invalidation Strategies** — Scanning and moving forward the sliding window upon arrival of new data
  - **Lazy Re-execution Strategies** — Removing old data periodically and require more memory to store data waiting for expiration

## Sliding Window – Two Strategies

- **Re-Execution Strategy**
  - Eager Re-execution Strategies
  - **Lazy Re-execution Strategies**
- **Data Invalidation Strategy**
  - Eager Expiration Strategies
  - **Lazy Invalidation Strategies**

Re-Execution and Expiration Period – Generation



## Different types of dynamic kNN joins

- Static R and Dynamic S (SRDS)
  - Exists rarely in real applications.
  - Reuse the parallel methods
- Dynamic R and Static S (DRSS)
  - Most used scenario in real applications
  - Example: find restaurant for moving users
  - Reuse Random Partition method
- Dynamic R and Dynamic S (DRDS)
  - General situation
  - Example: find Pokémon for moving players
  - Basic Method + Advanced Method

## Dynamic R and Dynamic S – Basic Method (Sliding Block Nested Loop)

 $n^2$  tasks for each generation

		S in $i^{th}$ Generation			
		$S_1$	$S_2$	...	$S_n$
R in $i^{th}$ Generation	$R_1$	$G_i(R_1, S_1)$	$G_i(R_1, S_2)$	...	$G_i(R_1, S_n)$
	$R_2$	$G_i(R_2, S_1)$	$G_i(R_2, S_2)$	...	$G_i(R_2, S_n)$
	...	...	...	...	...
	$R_n$	$G_i(R_n, S_1)$	$G_i(R_n, S_2)$	...	$G_i(R_n, S_n)$



## Dynamic R and Dynamic S – Advanced Method (Naive Bayes Partitioning)

**Purpose:** n tasks for each generation:

- Partition new data items
- No time to repartition old ones

**Solution:** Classification for new data items based on Naive Bayes Theory

# Outline

- Introduction
- Parallel Workflow
- Theoretical Analysis
- Continuous kNN
- Experiment Result
- Conclusion

## Conclusion

- A detail survey for parallel kNN join on MapReduce
- Continuous kNN Join for Data Streams
- Theoretical and Experimental Analysis

## Part II: Query Driven Stream Join (RDF)

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Plan
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

# Outline

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

## Introduction – RDF Data Model

- **Resource Description Framework**
- Describe semantic relations among data.
- Triples in form of <subject, predicate, object> (e.g. <Sophie, hasSister, Ray>)

## Introduction – SPARQL Query Language

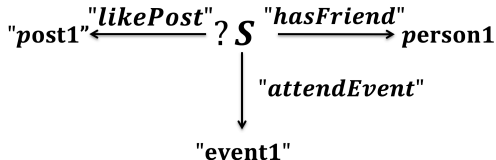
- SPARQL is a W3C recommendation query language for querying RDF data.
- The basic component of a SPARQL query is the triple patterns.
- A triple pattern is a special kind of triple where S, P and O can be either a literal or a variable.

### An Example (Triple Pattern Representation):

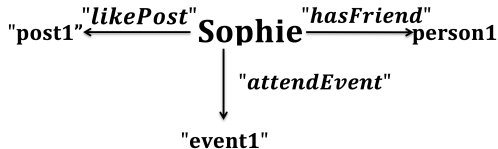
```
SELECT ?S
WHERE{
  Q1    ?S "hasFriend" person1 .
  Q2    ?S "likePost" "post1" .
  Q3    ?S "attendEvent" "event1"
}
```

## Introduction – SPARQL Query Example

### Graph Representation for SPARQL Query:



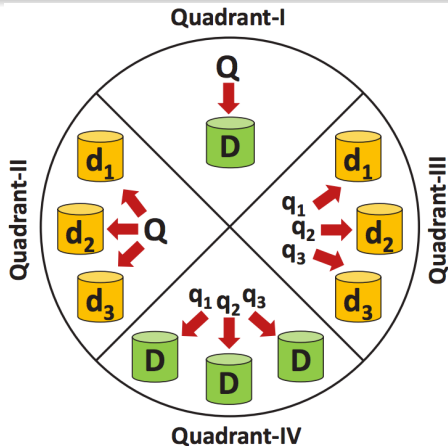
### Graph Representation for RDF Data:





## Related Works – 4 Types of Processing

- Q1: Centralize
- Q2 and Q4: Distribute either data or query
- Q3: Distribute both data and query (We use this manner)



DREAM: distributed RDF engine with adaptive query planner and minimal communication, PVLDB 2015, Mohammad Hammoud et. al.

## Related Work – Partitioning Strategies for RDF graphs

- Vertex Partitioning methods for graphs.
  - High overhead of loading big RDF graphs into the existing graph partitioner.
  - Requires the entire graph information in order to make decisions
  - Replication of the boundary of each partition in order to reduce the transmission of data
- Hash Partitioning based on indexes
  - Too many indexes ( up to 15)

## General Distributed Processing Steps

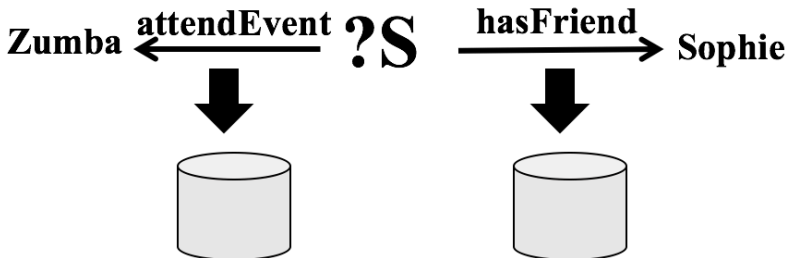
- Partition the RDF streams, and distribute these sub-streams to different nodes
- Decompose the queries into sub-queries and assign these sub-queries to the appropriate nodes
- Reply rapidly to the changes of data (the expiration of old data, and the update of new data), and return the results in real-time

# Outline

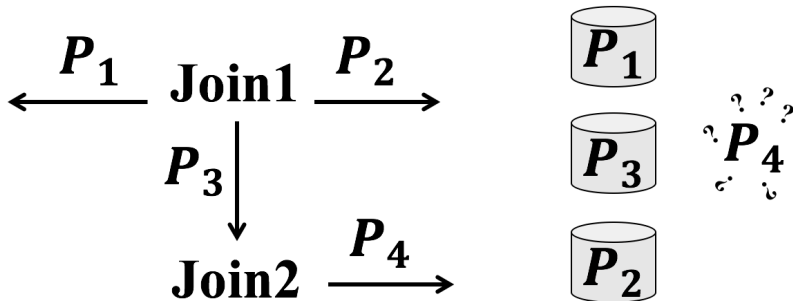
- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

## Query Decomposition

**Decomposition Strategy:** Divide the queries into triple patterns, send each triple pattern to different machines.

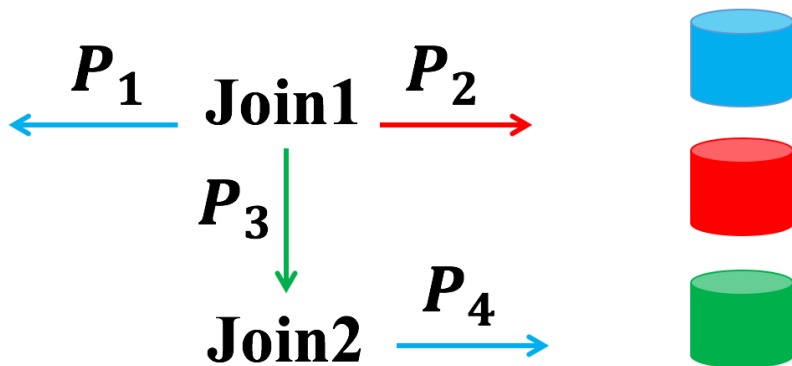


## Sub-Query Scheduling



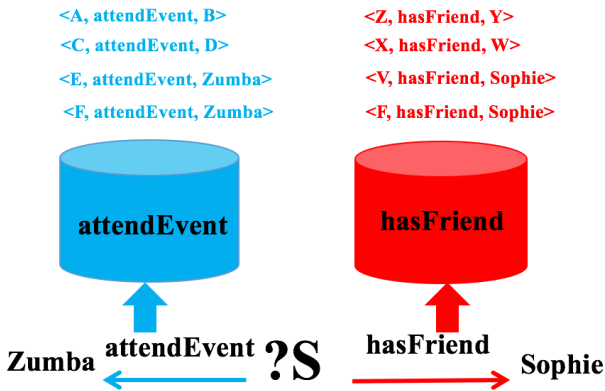
## Sub-Query Scheduling

### Edge Coloring Method: Maximize Parallelism



## Data Partitioning

**Partitioning Strategy:** The triples will be assigned to the nodes that hold the triple pattern with the same predicate.





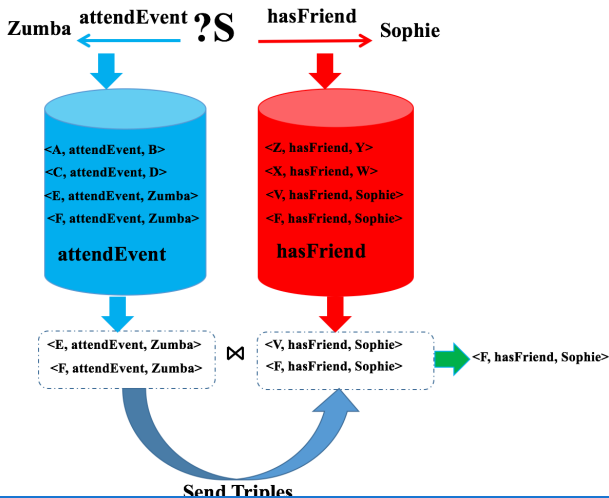
# Outline

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Analysis
- Implementation
- Experiment Result
- Conclusion

## Challenges

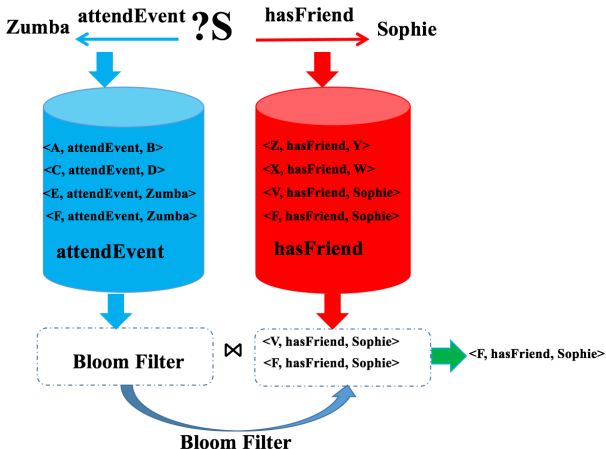
- Communication among nodes
- Join of the intermediate results produced by each triple pattern
- Order of sending and receiving information

# Communication



## Communication

Do not send triples, send a function saying that we already met these triples



## Communication among nodes – Bloom Filter

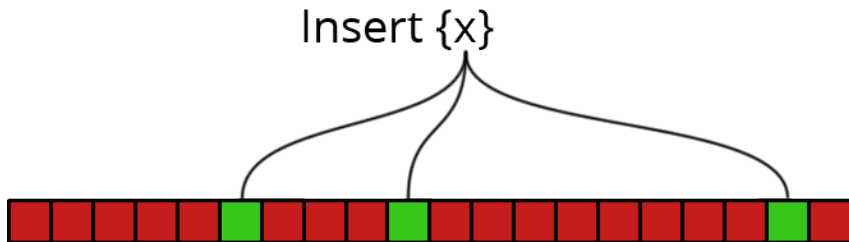


$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions

## Communication among nodes – Bloom Filter

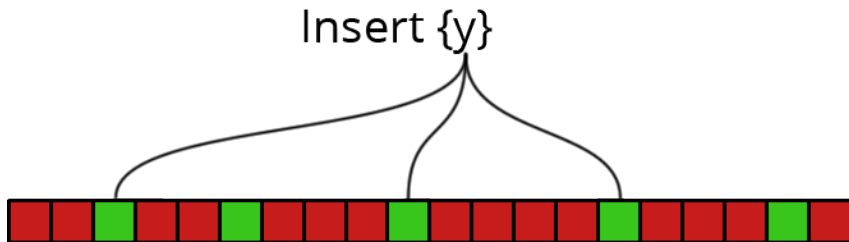


$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions

## Communication among nodes – Bloom Filter

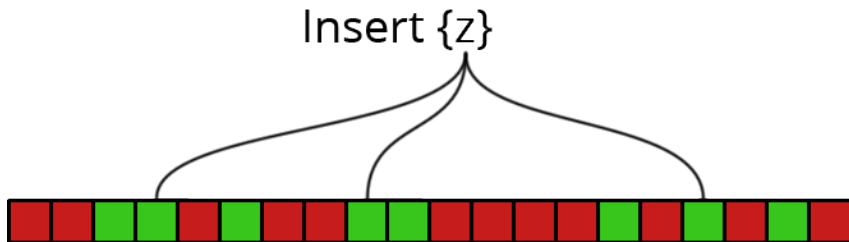


$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions

## Communication among nodes – Bloom Filter



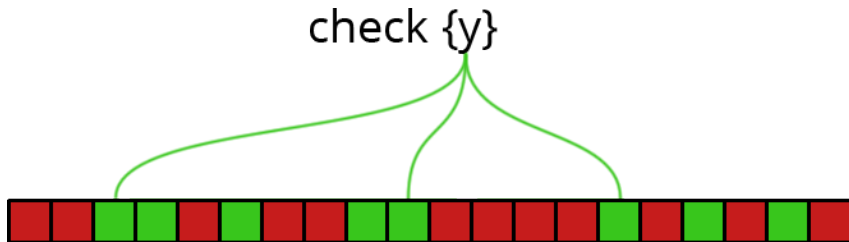
$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions



## Communication among nodes – Bloom Filter

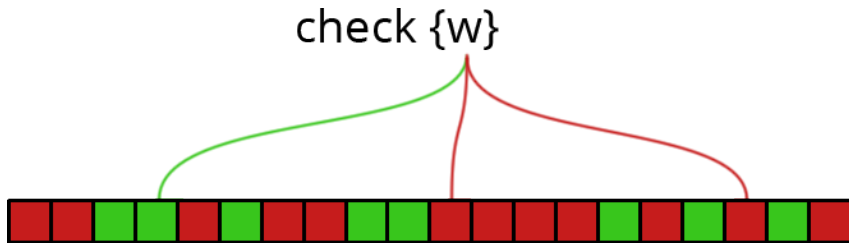


$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions

## Communication among nodes – Bloom Filter

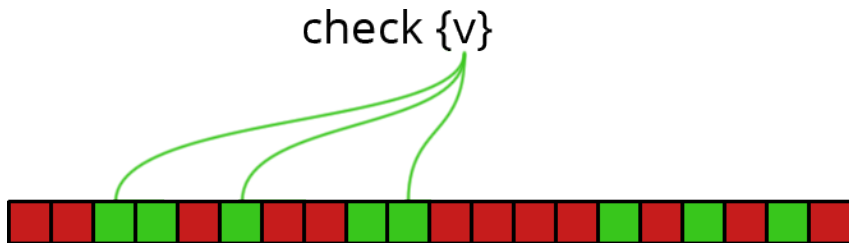


$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions

## Communication among nodes – Bloom Filter



$m = 20$  bits

$k = 3$  hash functions

$n = 3$  insertions

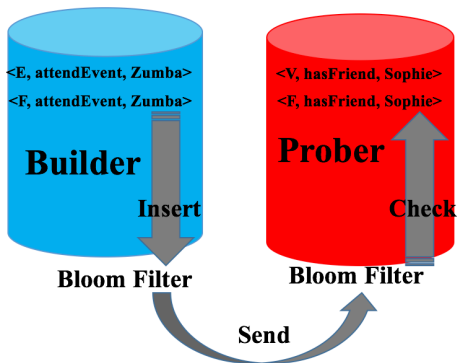
## Communication among nodes – Bloom Filter

### False Positive Rate

$$p = (1 - e^{-\frac{nk}{m}})^k$$

## Bloom Filter – Build and Probe

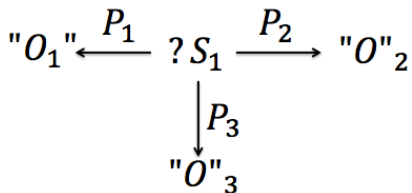
- **Builder:** create bloom filters.
- **Prober:** use bloom filters



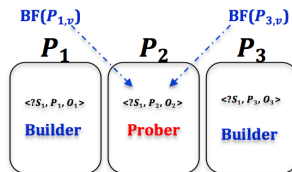
**Question:** Which triple patterns should be **Builders**, and which ones should be **Probers**?

# The join of the intermediate results – Structure Based Rules

## Rule 1: 1-Variable Join



Bloom Filter Send:

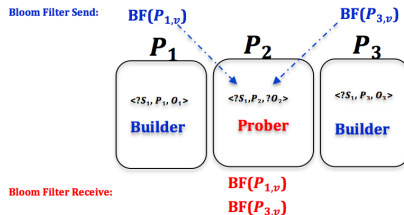
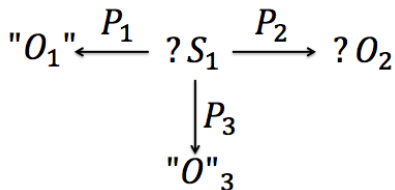


Bloom Filter Receive:

$BF(P_{1,v})$   
 $BF(P_{3,v})$

# The join of the intermediate results – Structure Based Rules

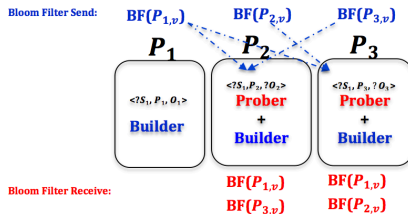
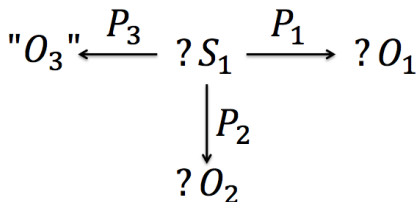
## Rule 2: 2-Variable Join





# The join of the intermediate results – Structure Based Rules

## Rule 3: Multiple-Variable Join



**Question:** what is the sending and receiving order?

## Sending and Receiving orders

### Rule 4: Query Topological Sort

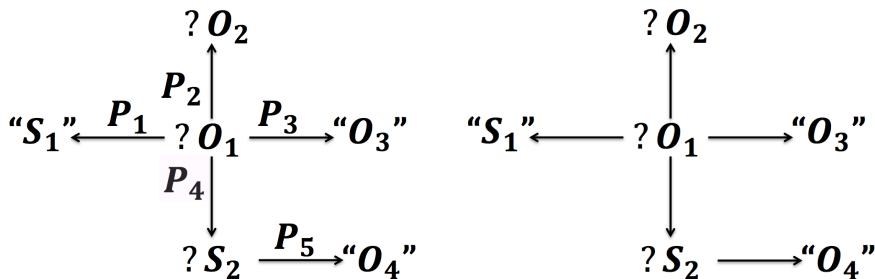
**Purpose:** Find the dependencies for each variable

#### Query Topological Sort

**Query Topological Sort** is a topological sort for the query graphs, where the constant nodes on the graph have higher priority than the variable nodes at the same level.

# The order of sending and receiving information

## Rule 4: Query Topological Sort

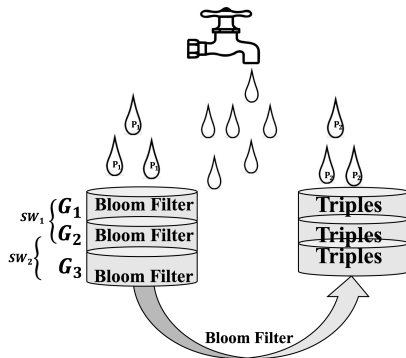
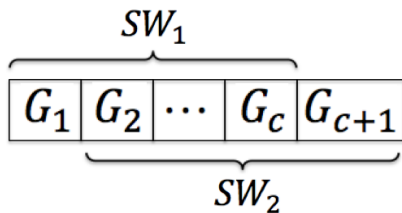


**Results:**  $\{ "O_4", ?S_2, "S_1", "O_3", ?O_2, ?O_1 \}$

# Outline

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

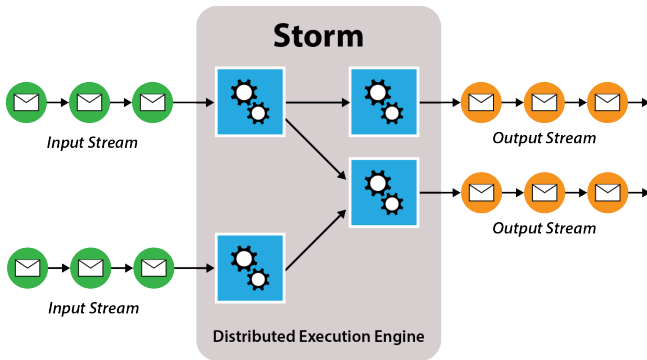
# Continuous Join: Sliding Window + Sliding Bloom Filter



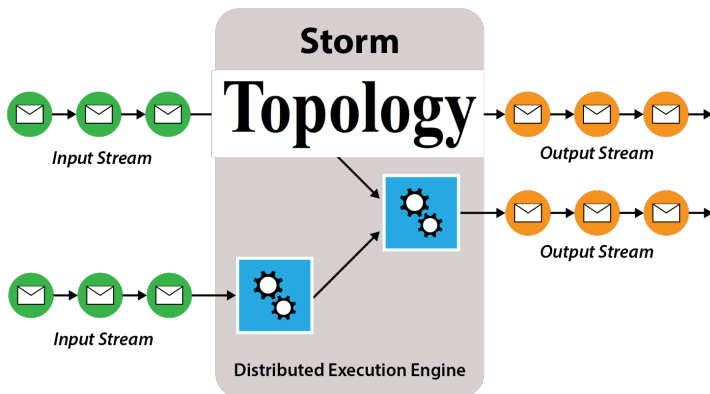
# Outline

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

# Apache Storm

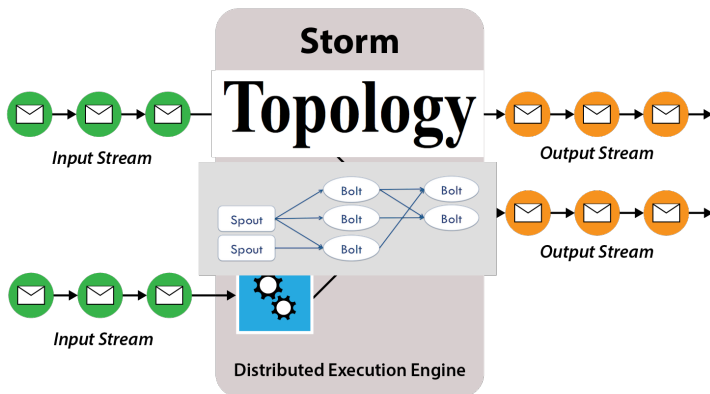


# Apache Storm

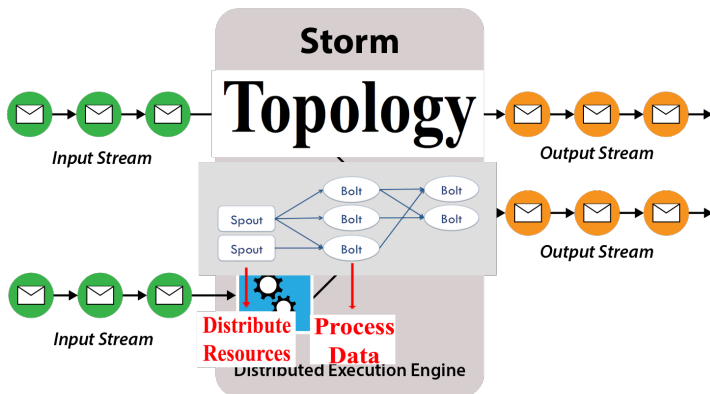




# Apache Storm

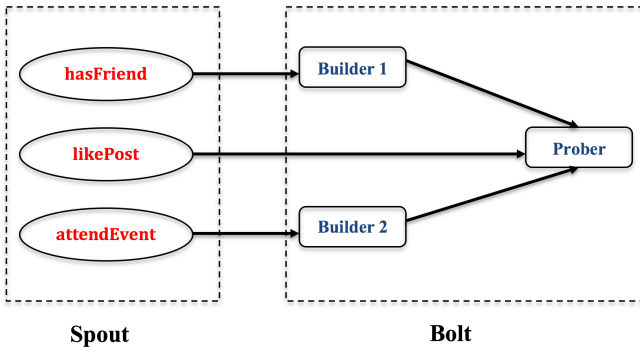


# Apache Storm



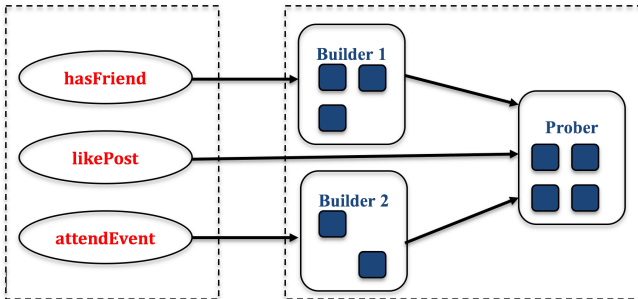
# Implementation

```
SELECT ?S  
WHERE{  
  Q1    ?S "hasFriend" person1 .  
  Q2    ?S "likePost" "post1" .  
  Q3    ?S "attendEvent" "event1"  
}
```



# Implementation

```
SELECT ?S  
WHERE{  
  Q1    ?S "hasFriend" person1 .  
  Q2    ?S "likePost" "post1" .  
  Q3    ?S "attendEvent" "event1"  
}
```



**Spout**

**Bolt**

# Outline

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

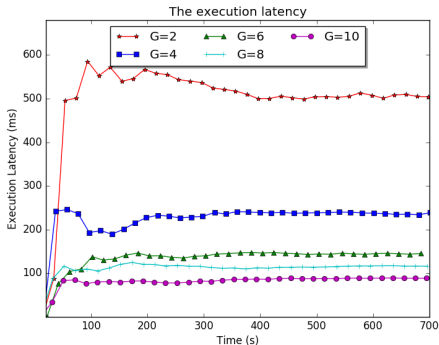
## Experiment Setting

- We evaluate the system on Grid 5000, with 11 nodes. 1 Master (Nimbus), 10 Slaves (Supervisor).
- The Storm version is 1.0, and we only use one slot on each machine.
- Apache Jena API is used for reading triples.
- Synthetic data
  - The RDF triples generated in Spouts are distributed to the nodes according to their predicate.

# Execution Latency

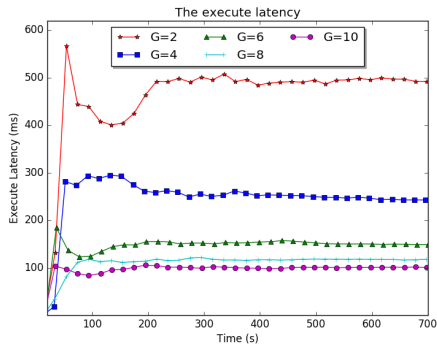
Sliding Window Size = 800 (1-Variable Join)

More Generations  $\Rightarrow$  More frequent updates  $\Rightarrow$  Faster

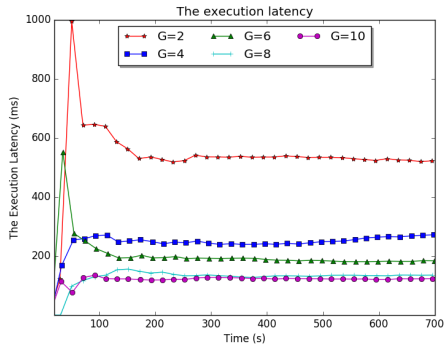


# Execution Latency

## 2-Variable Join



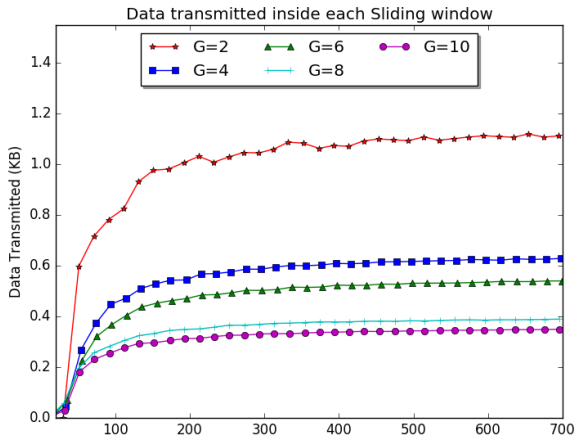
## Multiple-Variable Join





# Data Transmitted – Sliding Window Size = 800

## Multiple-Variable Join



## Accuracy

**We got 100% correct results – Surprise!**

Large Bloom Filters but very few matching elements.

# Outline

- Introduction
- Query Decomposition and Data Partition
- Parallel and Distributed Query Planner
- Continuous Join
- Implementation
- Experiment Result
- Conclusion

## Conclusion

- Parallel and distributed join processing on RDF streams
- Distribute both data and queries
- Efficient
  - Time
    - Execution Latency less than 600 ms
  - Space — 400 times more efficient than without using Bloom Filters

## Conclusion and Future Work



# Conclusion

- Data Driven Join
  - Data Preprocessing
  - Data Partitioning
  - Data Computation
- Query Driven Join
  - Query Decomposition
  - Communication among Sub-Queries
  - Combine results of Sub-Queries

## Future Work

- Enrich the Naive Bayes re-partitioning strategy for kNN stream join (theoretical and experimental analysis and proof – Accuracy, Load Balance )
- Enrich the SPARQL query engine by providing operations such as FILTER, OPTIONAL, etc.

## Future Applications

- Real Time Recommendation System Based on kNN
- Real Time Natural Language Processing System Based on RDF



## Publications:

- **K Nearest Neighbour Joins for Big Data on MapReduce: a Theoretical and Experimental Analysis** — TKDE 2016 — **Ge Song**, J. Rochas, L. Beze, F. Huet, F. Magoulès
- **Solutions for Processing K Nearest Neighbor Joins for Massive Data on MapReduce** — PDP 2015 — **Ge Song**, J. Rochas, F. Huet, F. Magoulès
- **A Hadoop MapReduce Performance Prediction Method** — HPCC 2013 — **Ge Song**, Z. Meng, F. Huet, F. Magoulès, L. Yu, X. Lin
- **A Game Theory Based MapReduce Scheduling Algorithm** — Springer New York — **Ge Song**, L. Yu, Z. Meng, X. Lin
- **Detecting topics and overlapping communities in question and answer sites** — SNAM 2014 — Z. Meng, F. Gandon, C. Zucker, **Ge Song**
- **Empirical study on overlapping community detection in question and answer sites** — ASONAM 2014 — Z. Meng, F. Gandon, C. Zucker, **Ge Song**

Thank You!