Efficient Parallel kNN Joins for Large Data in MapReduce

Chi Zhang[1] Feifei Li[2] Jeffrey Jestes[2]

[1] Dept of Computer Science
Florida State University

[2] School of Computing
University of Utah

April 4, 2012

Outline

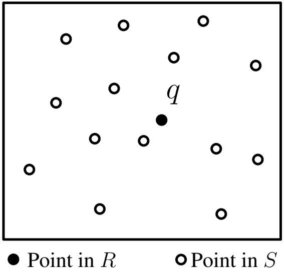
1. Introduction
2. Background: kNN Join
3. Parallel kNN Join for Multi-dimensional Data Using MapReduce
    - Exact kNN Join
    - Approximate kNN Join
4. Experiments
5. Conclusions

Outline

1. Introduction
2. Background: *k*NN Join
3. Parallel *k*NN Join for Multi-dimensional Data Using MapReduce
    - Exact *k*NN Join
    - Approximate *k*NN Join
4. Experiments
5. Conclusions

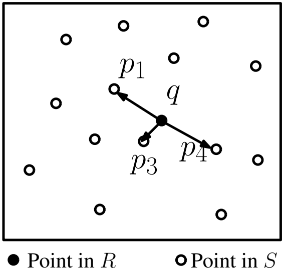
k Nearest Neighbor Join

- k nearest neighbor join (*k*NN join)
    - Given two data sets R and S, for every point q in R, *k*NN join returns k nearest points of q from S.



k Nearest Neighbor Join

- k nearest neighbor join (*k*NN join)
  - Given two data sets *R* and *S*, for every point *q* in *R*, *k*NN join returns *k* nearest points of *q* from *S*.

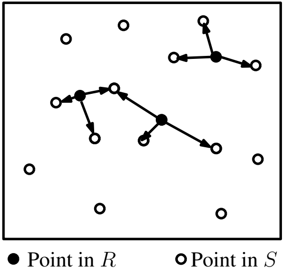


3-NN join for q

(q, p_1)
(q, p_3)
(q, p_4)

k Nearest Neighbor Join

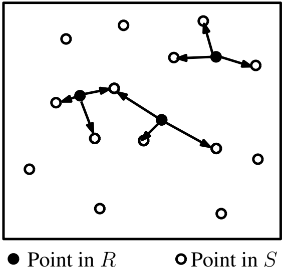
- k nearest neighbor join (kNN join)
  - Given two data sets R and S, for every point q in R, kNN join returns k nearest points of q from S.



Find kNN in S for all points in R

k Nearest Neighbor Join

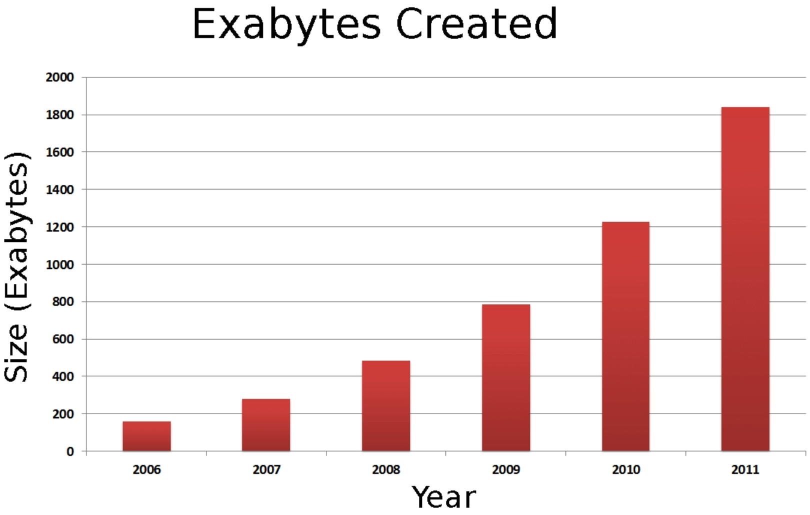
- k nearest neighbor join (kNN join)
  - Given two data sets R and S, for every point q in R, kNN join returns k nearest points of q from S.



Find kNN in S for all points in R

- Numerous applications: knowledge discovery, data mining, spatial databases, multimedia databases, etc.

Data Growth

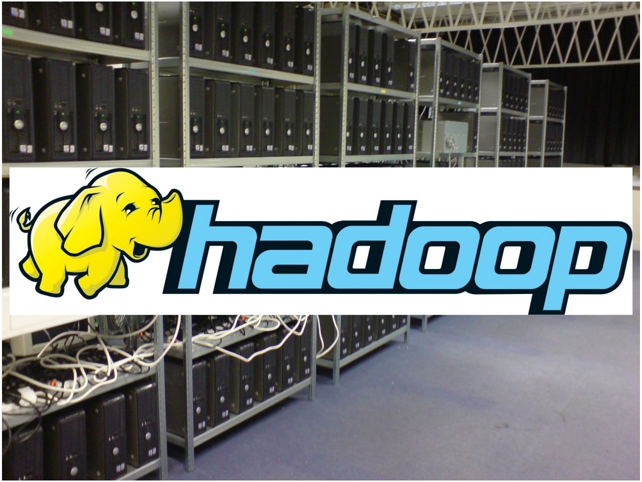


Source: IDC

Rise of Distributed and Parallel Computing

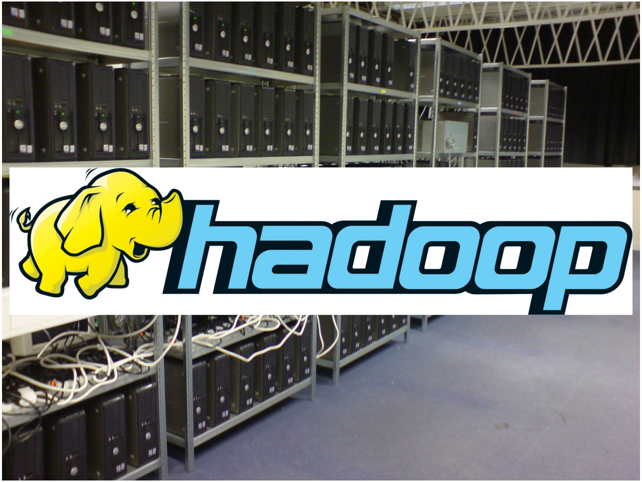
- Data sets are growing at an exponential rate.
    - A single machine cannot handle large data efficiently.
    - Parallel and distributed computing is the trend.

Rise of Distributed and Parallel Computing



- Data sets are growing at an exponential rate.
    - A single machine cannot handle large data efficiently.
    - Parallel and distributed computing is the trend.

Rise of Distributed and Parallel Computing



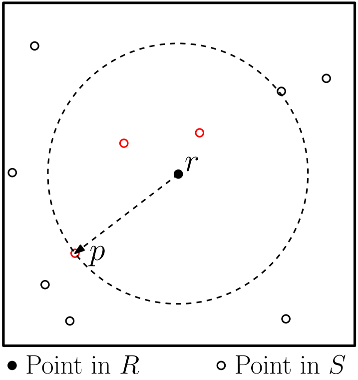
- Challenges:
  - Minimize communication and computation.
  - Achieve good load balance.

Outline

1. Introduction
2. Background: *k*NN Join
3. Parallel *k*NN Join for Multi-dimensional Data Using MapReduce
    - Exact *k*NN Join
    - Approximate *k*NN Join
4. Experiments
5. Conclusions

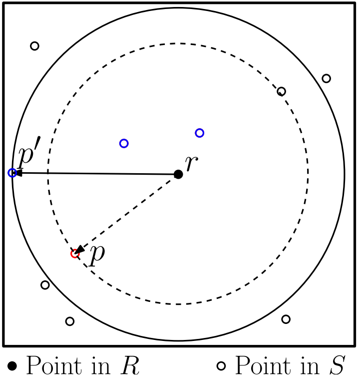
kNN Join

- Exact kNN Join
  - $knn(r, S)$ = set of kNN of r from S.
  - $knnJ(R, S) = \{(r, knn(r, S)) | \text{ for all } r \in R\}$.



kNN Join

- Exact kNN Join
  - $knn(r, S)$ = set of kNN of r from S.
  - $knnJ(R, S) = \{(r, knn(r, S)) | \text{ for all } r \in R\}$.
- Approximate kNN Join
  - $aknn(r, S)$ = approximate kNN of r from S.
    - p = kth NN of r in $knn(r, S)$.
    - p' = kth NN for r in $aknn(r, S)$
    - $aknn(r, S)$ is a *c-approximation* of $knn(r, S)$: $d(r, p) \leq d(r, p') \leq c \cdot d(r, p)$.
  - $aknnJ(R, S) = \{(r, aknn(r, S)) | \forall r \in R\}$.



Outline

1. Introduction

2. Background: *k*NN Join

3. Parallel *k*NN Join for Multi-dimensional Data Using MapReduce
    - Exact *k*NN Join
    - Approximate *k*NN Join

4. Experiments

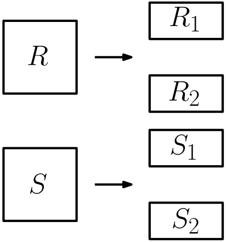
5. Conclusions

Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method

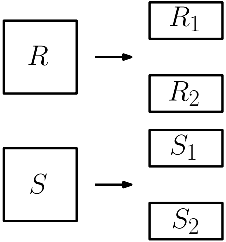
Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method
  1. Partition R and S, each into n equal-sized disjoint blocks.



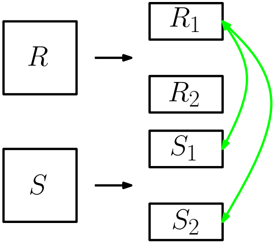
Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method
  1. Partition R and S, each into n equal-sized disjoint blocks.
  2. Perform (BNLJ) for each possible R_i,S_j pairs of blocks



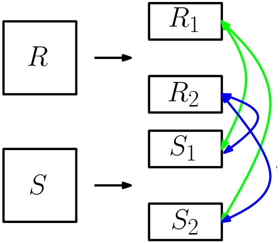
Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method
  1. Partition R and S, each into n equal-sized disjoint blocks.
  2. Perform (BNLJ) for each possible R_i,S_j pairs of blocks



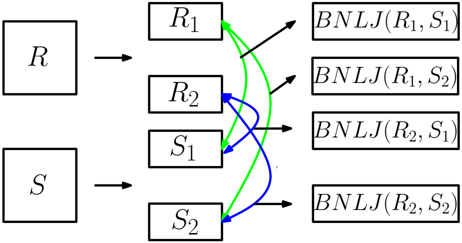
Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method
  1. Partition R and S, each into n equal-sized disjoint blocks.
  2. Perform (BNLJ) for each possible R_i,S_j pairs of blocks



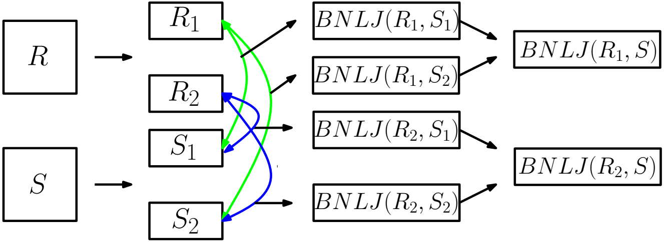
Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method
    1. Partition *R* and *S*, each into *n* equal-sized disjoint blocks.
    2. Perform (BNLJ) for each possible R_i,S_j pairs of blocks



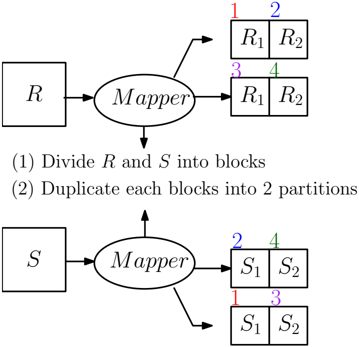
Exact *k*NN join: Block Nested Loop Join

- Block nested loop join (BNLJ) based method
    1. Partition R and S, each into n equal-sized disjoint blocks.
    2. Perform (BNLJ) for each possible R_i,S_j pairs of blocks
    3. Get global *k*NN results from n local *k*NN results for every record in R



Exact *k*NN join: Block Nested Loop Join

Two-round MapReduce algorithm: Round 1

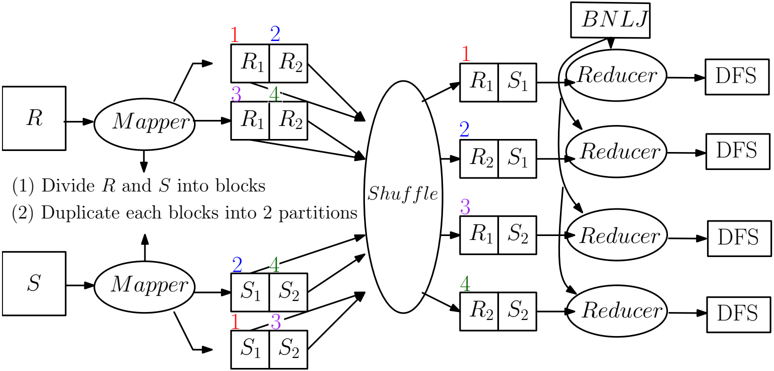


(1) Divide R and S into blocks

(2) Duplicate each blocks into 2 partitions

Exact *k*NN join: Block Nested Loop Join

Two-round MapReduce algorithm: Round 1

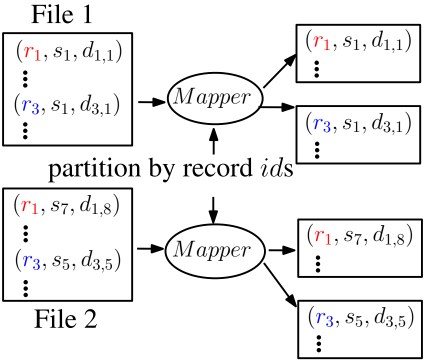


(1) Divide R and S into blocks

(2) Duplicate each blocks into 2 partitions

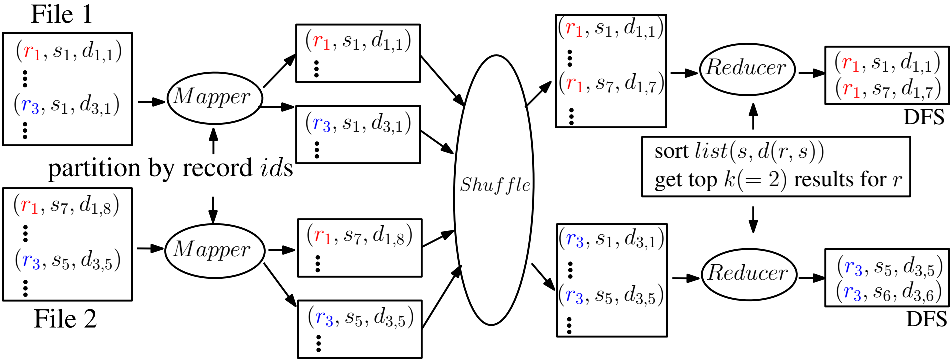
Exact *k*NN join: Block Nested Loop Join

Two-round MapReduce algorithm: Round 2



Exact *k*NN join: Block Nested Loop Join

Two-round MapReduce algorithm: Round 2



Exact *k*NN join: Block R-tree Join

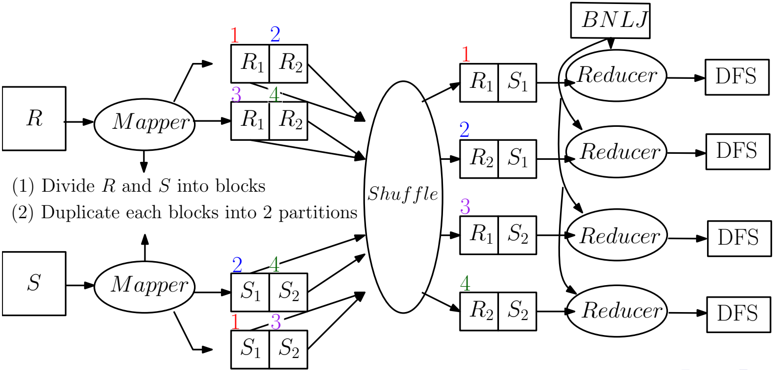
- Use spatial index (R-tree) to improve performance

Exact *k*NN join: Block R-tree Join

- Use spatial index (R-tree) to improve performance
    - Build R-tree index for a block of S in a bucket to speed up kNN computations.
    - Similar to BNLJ algorithm, only need to replace BNLJ with block R-tree join (BRJ) in the first round.

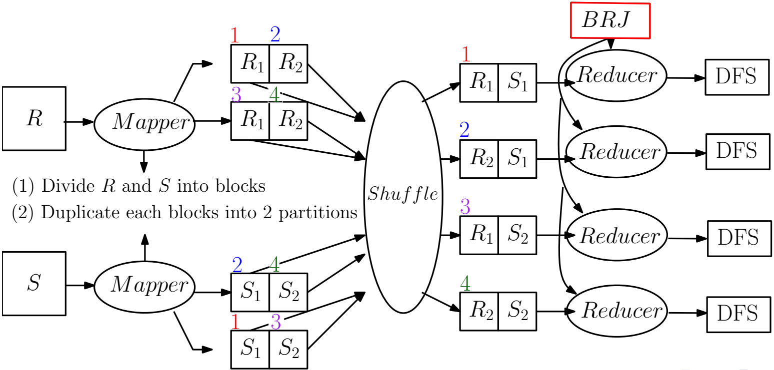
Exact *k*NN join: Block R-tree Join

- Use spatial index (R-tree) to improve performance
  - Build R-tree index for a block of *S* in a bucket to speed up *k*NN computations.
  - Similar to BNLJ algorithm, only need to replace BNLJ with block R-tree join (BRJ) in the first round.



Exact *k*NN join: Block R-tree Join

- Use spatial index (R-tree) to improve performance
  - Build R-tree index for a block of *S* in a bucket to speed up *k*NN computations.
  - Similar to BNLJ algorithm, only need to replace BNLJ with block R-tree join (BRJ) in the first round.



Outline

1. Introduction

2. Background: kNN Join

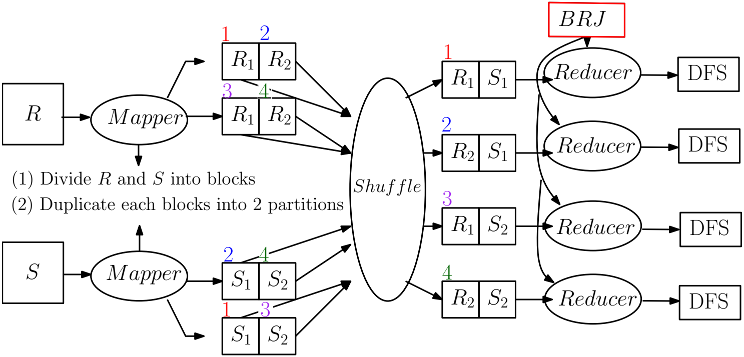
3. Parallel kNN Join for Multi-dimensional Data Using MapReduce
    - Exact kNN Join
    - Approximate kNN Join

4. Experiments

5. Conclusions

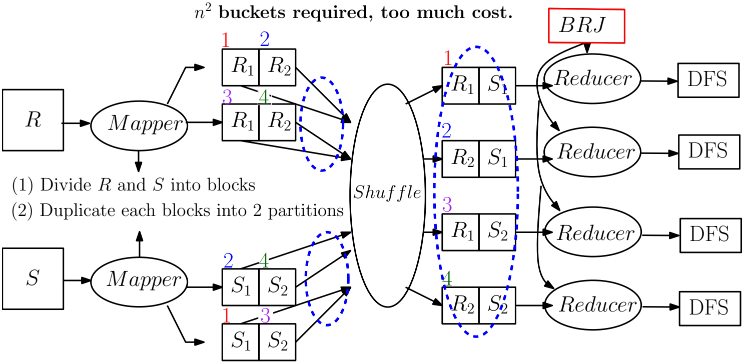
Approximate *k*NN join

- Problems with exact *k*NN join solution



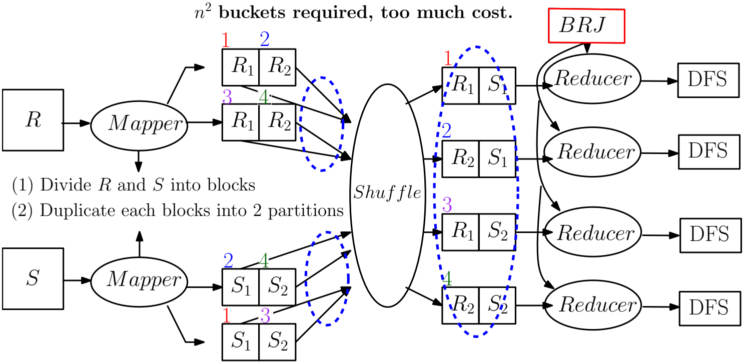
Approximate *k*NN join

- Problems with exact *k*NN join solution
  - Too much communication and computation (n^2 buckets required)



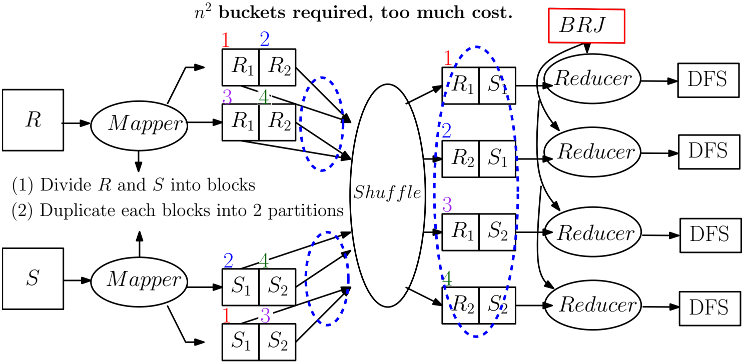
Approximate *k*NN join

- Problems with exact *k*NN join solution
    - Too much communication and computation (n^2 buckets required)
- Find solution requiring $O(n)$ buckets.



Approximate *k*NN join

- Problems with exact *k*NN join solution
  - Too much communication and computation (n^2 buckets required)
- Find solution requiring $O(n)$ buckets.
  - We search for approximate solutions.
  - Space-filling curve based methods ([YLK10], dubbed zkNN)



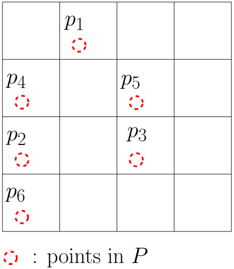
[YLK10] B. Yao, F. Li, P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. *ICDE*, 2010.

Approximate *k*NN join: *Z*-order *k*NN join

- The idea of zkNN
    - Transform *d*-dimensional points to 1-D values using *Z*-value.
    - Map *d*-dimensional *k*NN join query to to 1-D range queries.
    - Multiple random shift copies are used to improve spatial locality.
        - In practice 2 copies is arleady good enough.

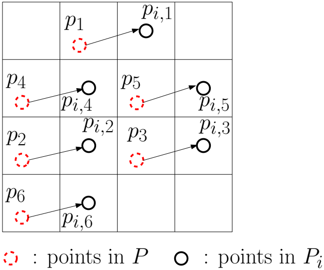
Approximate *k*NN join: *Z*-order *k*NN join

- The idea of zkNN
  - Transform *d*-dimensional points to 1-D values using *Z*-value.
  - Map *d*-dimensional *k*NN join query to to 1-D range queries.
  - Multiple random shift copies are used to improve spatial locality.
    - In practice 2 copies is arleady good enough.



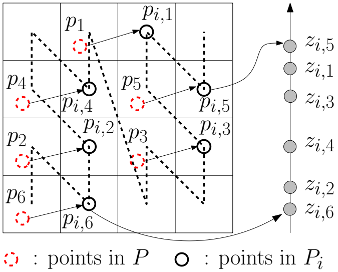
Approximate *k*NN join: *Z*-order *k*NN join

- The idea of zkNN
    - Transform *d*-dimensional points to 1-D values using *Z*-value.
    - Map *d*-dimensional *k*NN join query to to 1-D range queries.
    - Multiple random shift copies are used to improve spatial locality.
        - In practice 2 copies is arleady good enough.



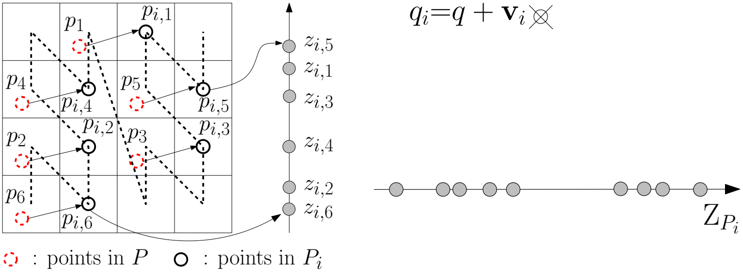
Approximate *k*NN join: *Z*-order *k*NN join

- The idea of zkNN
  - Transform *d*-dimensional points to 1-D values using *Z*-value.
  - Map *d*-dimensional *k*NN join query to to 1-D range queries.
  - Multiple random shift copies are used to improve spatial locality.
    - In practice 2 copies is arleady good enough.



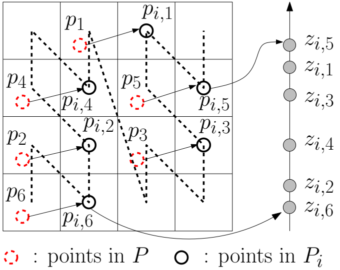
Approximate *k*NN join: *Z*-order *k*NN join

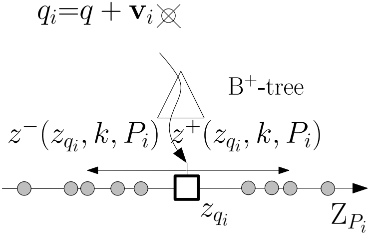
- The idea of zkNN
  - Transform *d*-dimensional points to 1-D values using *Z*-value.
  - Map *d*-dimensional *k*NN join query to to 1-D range queries.
  - Multiple random shift copies are used to improve spatial locality.
    - In practice 2 copies is arleady good enough.



Approximate *k*NN join: *Z*-order *k*NN join

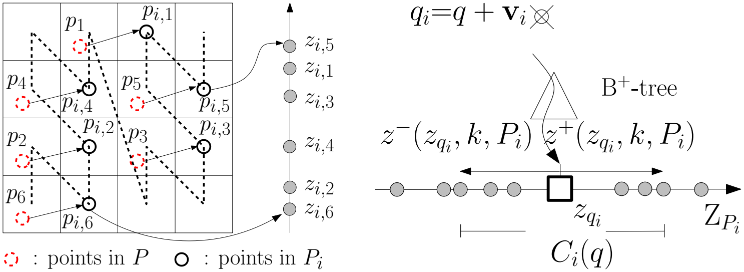
- The idea of zkNN
  - Transform *d*-dimensional points to 1-D values using *Z*-value.
  - Map *d*-dimensional *k*NN join query to to 1-D range queries.
  - Multiple random shift copies are used to improve spatial locality.
    - In practice 2 copies is arleady good enough.





Approximate *k*NN join: *Z*-order *k*NN join

- The idea of zkNN
  - Transform *d*-dimensional points to 1-D values using *Z*-value.
  - Map *d*-dimensional *k*NN join query to to 1-D range queries.
  - Multiple random shift copies are used to improve spatial locality.
    - In practice 2 copies is arleady good enough.



Approximate *k*NN join: *Z*-order *k*NN join

- In our group's previous work we derive the following guarantee for the z*k*NN join:

Theorem

Given a query point $q \in \mathbb{R}^d$*, a data set* $P \subset \mathbb{R}^d$*, and a small constant* $\alpha \in \mathbb{Z}^+$*. We generate* $(\alpha - 1)$ *random vectors* $\{\mathbf{v}_2, \ldots, \mathbf{v}_\alpha\}$*, such that for any* i*,* $\mathbf{v}_i \in \mathbb{R}^d$*, and shift* P *by these vectors to obtain* $\{P_1, \ldots, P_\alpha\}$ $(P_1 = P)$*. Then, the zkNN join returns a constant approximation in any fixed dimension for* $\mathrm{knn}(q, P)$ *in expectation.*

Approximate kNN join: H-zkNNJ

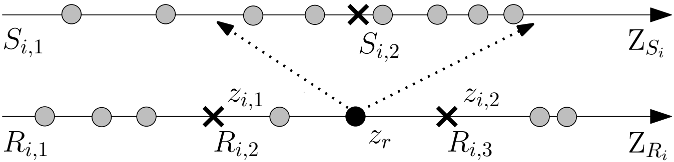
- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
    - Partitioning policy:
        - To achieve linear communication and computation costs (to the number of blocks n in each input data set)

Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
  - Partitioning policy:
    - To achieve linear communication and computation costs (to the number of blocks n in each input data set)
  - Partitioning by z-values:

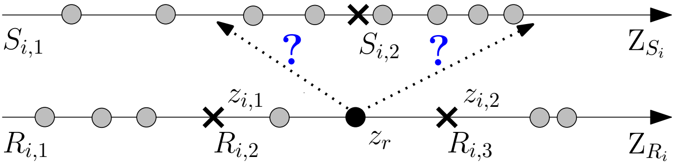
Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
  - Partitioning policy:
    - To achieve linear communication and computation costs (to the number of blocks *n* in each input data set)
  - Partitioning by *z*-values:



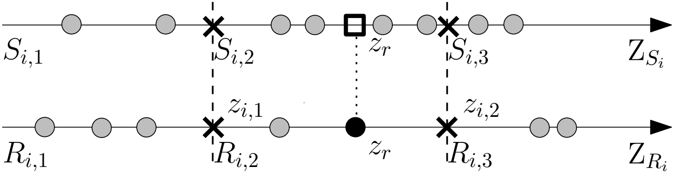
Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
  - Partitioning policy:
    - To achieve linear communication and computation costs (to the number of blocks *n* in each input data set)
  - Partitioning by *z*-values:



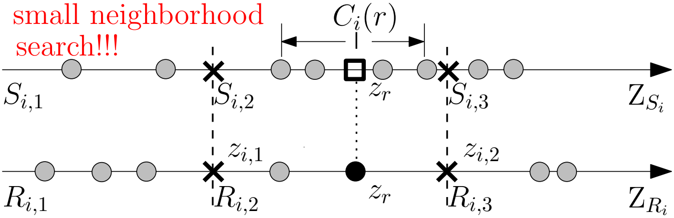
Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
  - Partitioning policy:
    - To achieve linear communication and computation costs (to the number of blocks n in each input data set)
  - Partitioning by z-values:
    - Partition input data sets R_i and S_i into $\{R_{i,1}, ..., R_{i,n}\}$ and $\{S_{i,1}, ..., S_{i,n}\}$ using $(n-1)$ z-values $\{z_{i,1}, ..., z_{i,n}\}$



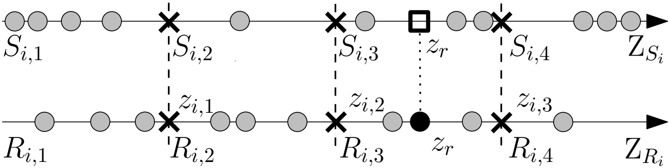
Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
    - Partitioning policy:
        - To achieve linear communication and computation costs (to the number of blocks *n* in each input data set)
    - Partitioning by *z*-values:
        - Partition input data sets R_i and S_i into $\{R_{i,1}, ..., R_{i,n}\}$ and $\{S_{i,1}, ..., S_{i,n}\}$ using $(n-1)$ *z*-values $\{z_{i,1}, ..., z_{i,n}\}$



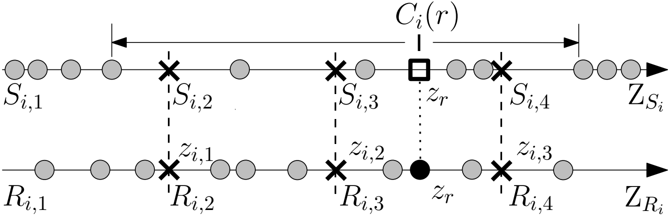
Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
  - Partitioning policy:
    - To achieve linear communication and computation costs (to the number of blocks n in each input data set)
  - Partitioning by z-values:
    - Partition input data sets R_i and S_i into $\{R_{i,1}, ..., R_{i,n}\}$ and $\{S_{i,1}, ..., S_{i,n}\}$ using $(n-1)$ z-values $\{z_{i,1}, ..., z_{i,n}\}$



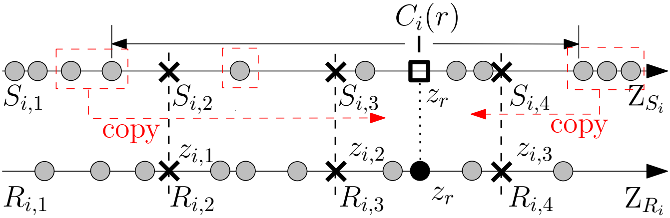
Approximate *k*NN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
    - Partitioning policy:
        - To achieve linear communication and computation costs (to the number of blocks *n* in each input data set)
    - Partitioning by *z*-values:
        - Partition input data sets R_i and S_i into $\{R_{i,1}, ..., R_{i,n}\}$ and $\{S_{i,1}, ..., S_{i,n}\}$ using $(n-1)$ *z*-values $\{z_{i,1}, ..., z_{i,n}\}$



Approximate kNN join: H-zkNNJ

- Apply zkNN for join in MapReduce (H-zkNNJ)
- Partition based algorithm
  - Partitioning policy:
    - To achieve linear communication and computation costs (to the number of blocks n in each input data set)
  - Partitioning by z-values:
    - Partition input data sets R_i and S_i into $\{R_{i,1}, \dots, R_{i,n}\}$ and $\{S_{i,1}, \dots, S_{i,n}\}$ using $(n-1)$ z-values $\{z_{i,1}, \dots, z_{i,n}\}$



Approximate kNN join: H-zkNNJ

- Choice of partitioning values.
  - Each block of R_i and S_i shares the same boundary so we only search a small neighborhood and minimize communication.
  - Goal: load balance.

Approximate *k*NN join: H-zkNNJ

- Choice of partitioning values.
  - Each block of R_i and S_i shares the same boundary so we only search a small neighborhood and minimize communication.
  - Goal: load balance.
  - Evenly partition R_i or S_i.

Approximate *k*NN join: H-zkNNJ

- Choice of partitioning values.
  - Each block of R_i and S_i shares the same boundary so we only search a small neighborhood and minimize communication.
  - Goal: load balance.
  - Evenly partition R_i or S_i.
    - Evenly partition $R_i \rightarrow O(\frac{|R_i|}{n} \log |S_i|)$
    - Evenly partition $S_i \rightarrow O(|R_i| log |S_i|)$

Approximate *k*NN join: H-zkNNJ

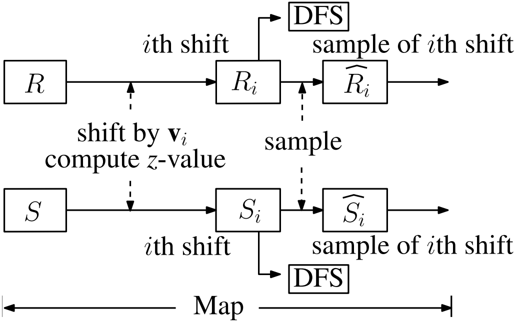
- Computation of partitioning values.
    - Quantiles can be used for evenly partitioning a data set D.
    - Sort a data set D and retrieve its $(n-1)$ quantiles (expensive).

Approximate *k*NN join: H-zkNNJ

- Computation of partitioning values.
  - Quantiles can be used for evenly partitioning a data set D.
  - Sort a data set D and retrieve its $(n-1)$ quantiles (expensive).
- We propose sampling based method to estimate quantiles.
  - We proved that both estimations are close enough (within ϵN) to the original ranks with a high probability (1-$e^{-2/\epsilon}$).

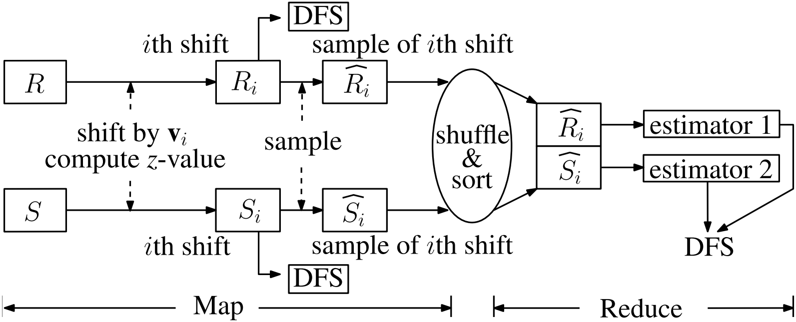
Approximate *k*NN join: H-zkNNJ

- $H-zkNNJ$ algorithm can be implemented in 3 rounds of MapReduce.
  - Round 1: construct random shift copies for R and S, R_i and S_i, $i \in [1, \alpha]$, and generate partitioning values for R_i and S_i



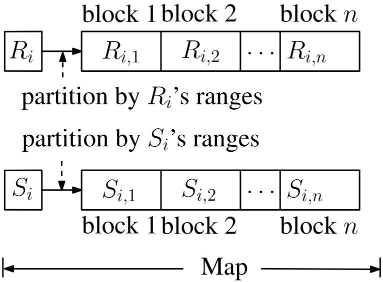
Approximate *k*NN join: H-zkNNJ

- $H-zkNNJ$ algorithm can be implemented in 3 rounds of MapReduce.
    - Round 1: construct random shift copies for R and S, R_i and S_i, $i \in [1, \alpha]$, and generate partitioning values for R_i and S_i



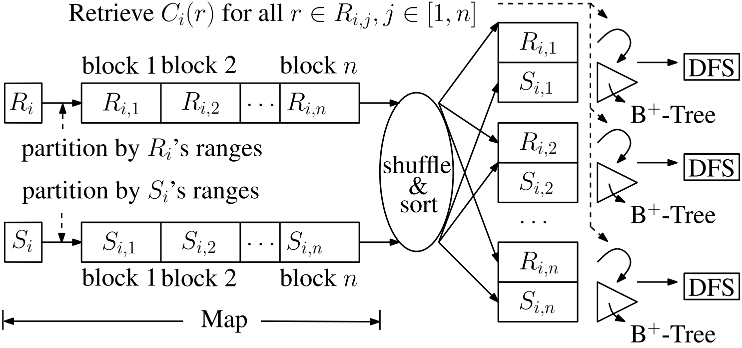
Approximate kNN join: H-zkNNJ

- $H-zkNNJ$ algorithm can be implemented in 3 rounds of MapReduce.
    - Round 2: partition R_i and S_i into blocks and compute the candidate points for $knn(r, S)$ for any $r \in R$.



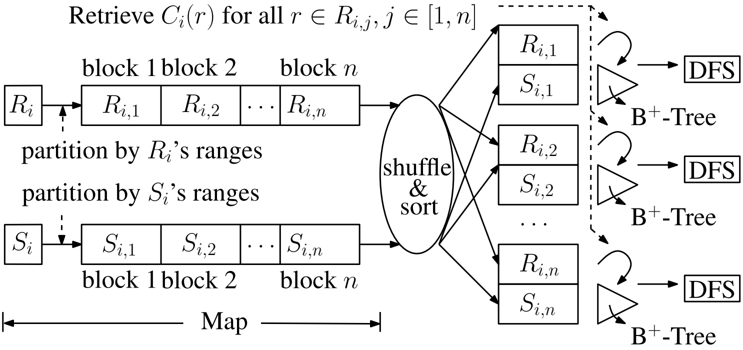
Approximate kNN join: H-zkNNJ

- $H-zkNNJ$ algorithm can be implemented in 3 rounds of MapReduce.
    - Round 2: partition R_i and S_i into blocks and compute the candidate points for $knn(r, S)$ for any $r \in R$.



Approximate *k*NN join: H-zkNNJ

- $H-zkNNJ$ algorithm can be implemented in 3 rounds of MapReduce.
  - Round 3: determine knn(r, $C(r)$) of any $r \in R$ from the (r, $C_i(r)$) emitted by round 2.



Outline

1. Introduction

2. Background: kNN Join

3. Parallel kNN Join for Multi-dimensional Data Using MapReduce
   - Exact kNN Join
   - Approximate kNN Join

4. Experiments

5. Conclusions

Experiments: algorithms

- We implement the following methods in Hadoop 0.20.2:
    - Exact Methods:
        - The baseline solution is denoted *H-BNLJ*,
        - The improvement to the baseline solution is denoted *H-BRJ*.
    - Approximate Methods:
        - Our three-round solution is denoted by *H-zkNNJ*, (meaning "Hadoop *z*-value *k*NN Join").

Experiments: setup

- Experiments are performed in a heterogeneous Hadoop cluster with 17 machines:
  1. 9 machines with 2GB of RAM and an Intel Xeon 1.86GHz CPU
  2. 6 machines with 4GB of RAM and an Intel Xeon 2GHz CPU
     - One is reserved for the master (running JobTracker and NameNode).
  3. 2 machines with 6GB of RAM and an Intel Xeon 2.13GHz CPU

Experiments: setup

- Experiments are performed in a heterogeneous Hadoop cluster with 17 machines:
  1. 9 machines with 2GB of RAM and an Intel Xeon 1.86GHz CPU
  2. 6 machines with 4GB of RAM and an Intel Xeon 2GHz CPU
     - One is reserved for the master (running JobTracker and NameNode).
  3. 2 machines with 6GB of RAM and an Intel Xeon 2.13GHz CPU
- All machines are directly connected to a 1000Mbps switch.
- Each slave node has 300GB hard drive space and 1GB of RAM for Hadoop daemon.
- The chunk size of DFS is set to 128MB.

Experiments: datasets

- *OpenStreet* Map dataset:
    - the road-networks for 50 states in U.S.
    - 160 million records.
    - preprocessed to remove duplications
    - each record consists of a 4 bytes integer id, two 4 bytes real type coordinates representing latitude and longitude, and a description information.
    - the coordinates has a positive real domain (0,100000).
    - stored in text format, 6.6GB.

Experiments: datasets

- *OpenStreet* Map dataset:
    - the road-networks for 50 states in U.S.
    - 160 million records.
    - preprocessed to remove duplications
    - each record consists of a 4 bytes integer id, two 4 bytes real type coordinates representing latitude and longitude, and a description information.
    - the coordinates has a positive real domain (0,100000).
    - stored in text format, 6.6GB.
- Large synthetic Random-Cluster datasets:
    - data sets have varying dimensionality (up to 30).
    - each record has a 4-byte id and float type d-dimensional coordinates.

Experiments: configurations and defaults

- Data set configurations
    - (MXN) represents a data set configuration containing M records of R and N record of S (in 10*s of millions*).

Experiments: configurations and defaults

- Data set configurations
  - (MXN) represents a data set configuration containing M records of R and N record of S (in 10*s of millions*).
- Default values for OpenStreet dataset:

Symbol	Definition	Default
(MXN)	data set configuration	(4x4)
k	# of nearest neighbor	10
α	# of shift copies	2
ϵ	the error rate of sampling	0.003
γ	the physical number of machines	16

Experiments: configurations and defaults

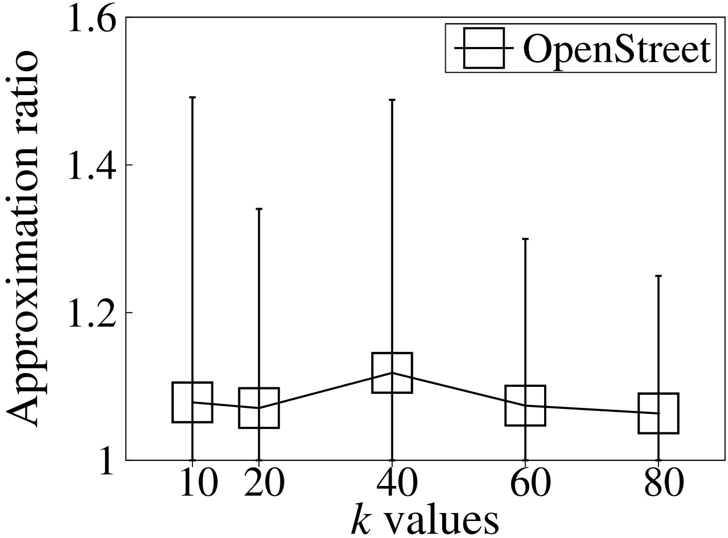
- Data set configurations
  - (MXN) represents a data set configuration containing M records of R and N record of S (in 10*s of millions*).
- Default values for OpenStreet dataset:

Symbol	Definition	Default
(MXN)	data set configuration	$(4x4)$
k	# of nearest neighbor	10
α	# of shift copies	2
ϵ	the error rate of sampling	0.003
γ	the physical number of machines	16

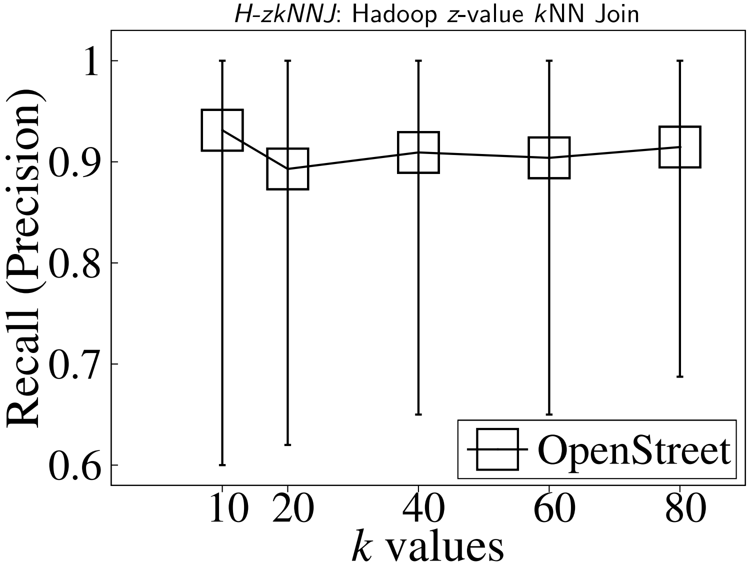
- Values for R-Cluster dataset:
  - (2x2) is set to be the default data set configuration.

Experiments: Approximation quality

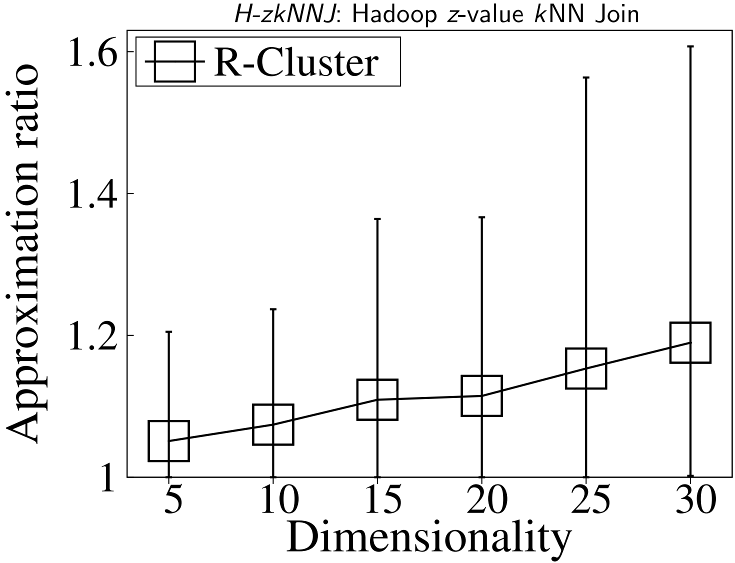
H-zkNNJ: Hadoop *z*-value *k*NN Join



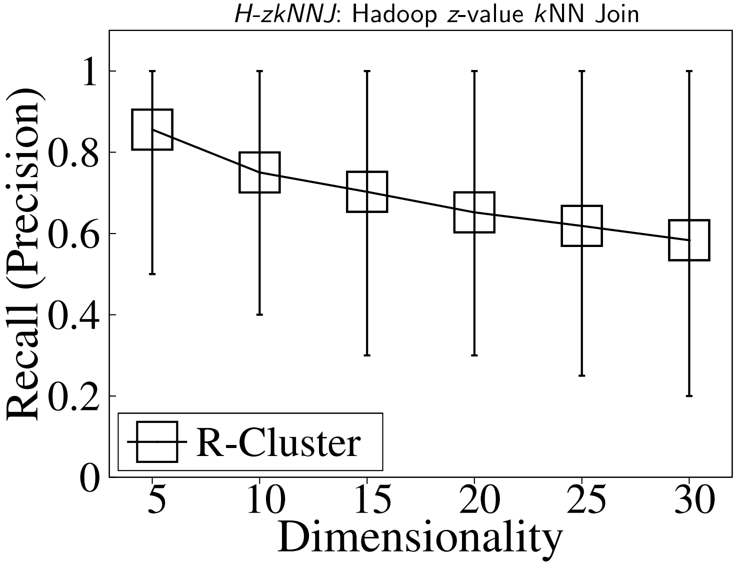
Experiments: Approximation quality



Experiments: Approximation quality

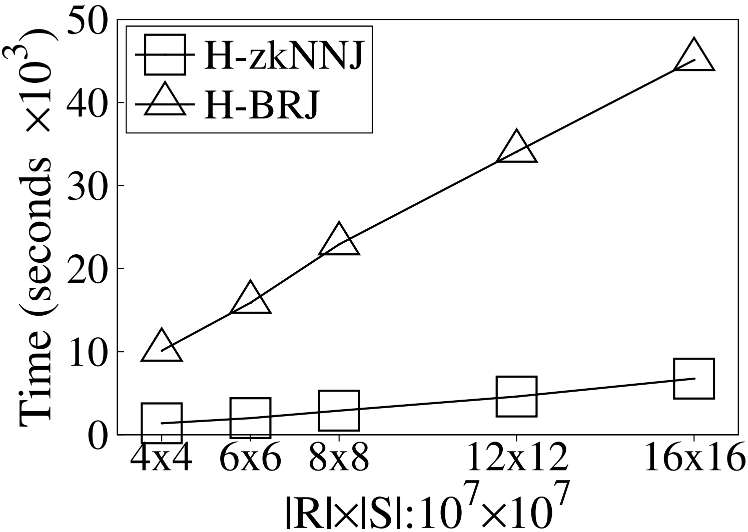


Experiments: Approximation quality



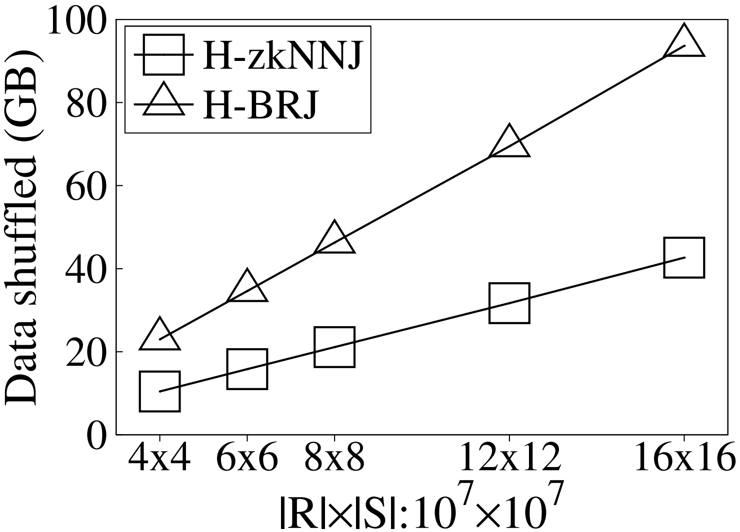
Experiments: Running time and communication cost

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



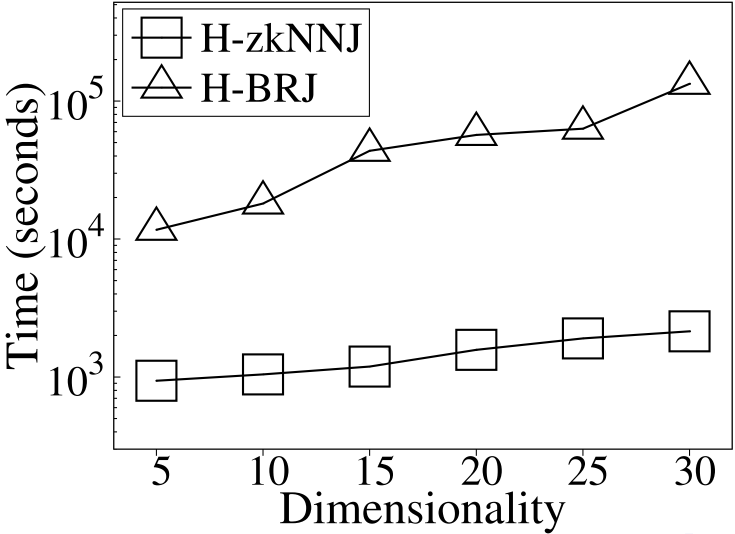
Experiments: Running time and communication cost

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



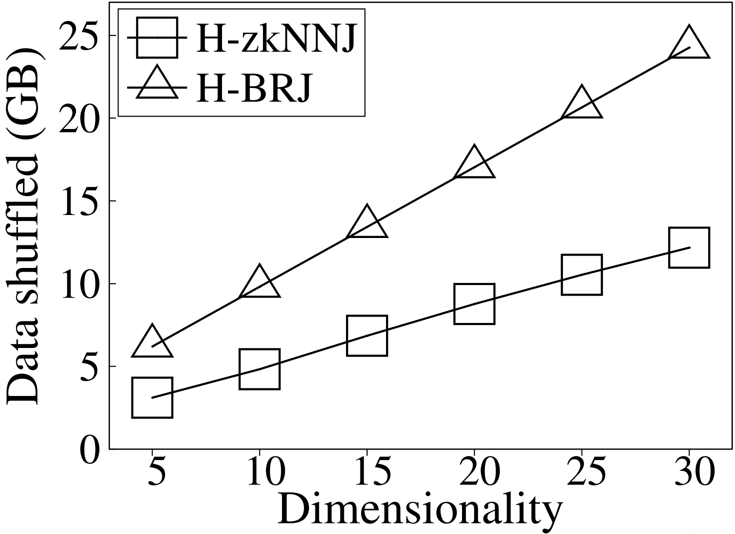
Experiments: Effect of d

H-zkNNJ: Hadoop z-value kNN Join
H-BRJ: Hadoop Block R-tree Join



Experiments: Effect of *d*

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



Conclusions

- We study efficient methods to perform kNN joins in MapReduce.
  - Exact (H-BRJ) and approximate (H-zkNNJ) algorithms are proposed.
  - H-zkNNJ performs orders of magnitude better than other methods with excellent approximation quality.
- We plan to investigate kNN joins on very high dimensions in the future.

The End

Thank You

Q and A

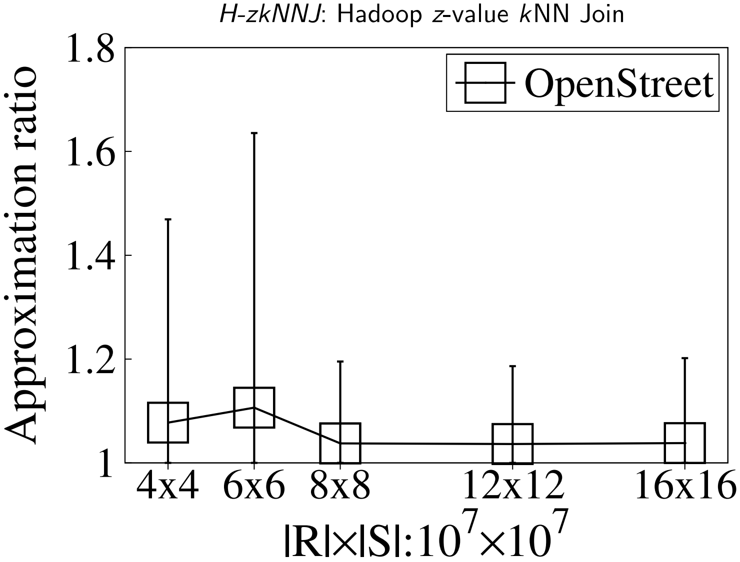
Approximate *k*NN join: *Z*-order *k*NN join

- zkNN algorithm

Algorithm 1: zkNN(q, P, k, α)

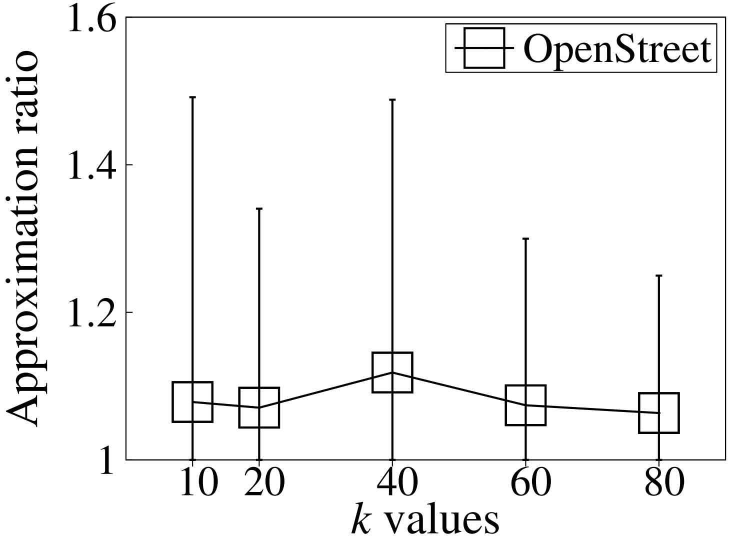
```
generate $\{\mathbf{v}_2, \dots, \mathbf{v}_\alpha\}$, $\mathbf{v}_1 = \overrightarrow{0}$, $\mathbf{v}_i$ is a random vector in $\mathbb{R}^d$;
$P_i = P + \mathbf{v}_i$ ($i \in [1, \alpha]$; $\forall p \in P$, insert $p + \mathbf{v}_i$ in $P_i$);
for $i = 1, \dots, \alpha$ do
    let $q_i = q + \mathbf{v}_i$, $C_i(q) = \emptyset$, and $z_{q_i}$ be $q_i$'s $z$-value;
    insert $z^-(z_{q_i}, k, P_i)$ into $C_i(q)$;
    insert $z^+(z_{q_i}, k, P_i)$ into $C_i(q)$;
    for any $p \in C_i(q)$, update $p = p - \mathbf{v}_i$;
$C(q) = \bigcup_{i=1}^{\alpha} C_i(q) = C_1(q) \cup \cdots \cup C_\alpha(q)$;
return knn($q$, $C(q)$).
```

Experiments: Approximation quality



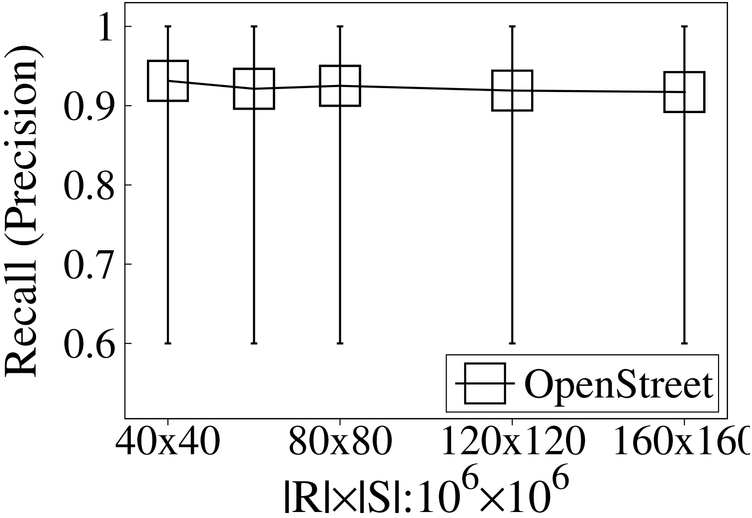
Experiments: Approximation quality

H-zkNNJ: Hadoop *z*-value *k*NN Join

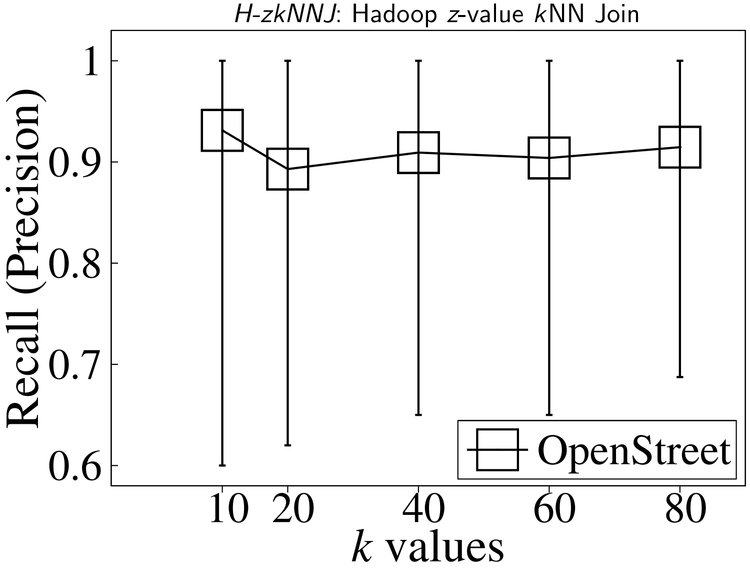


Experiments: Approximation quality

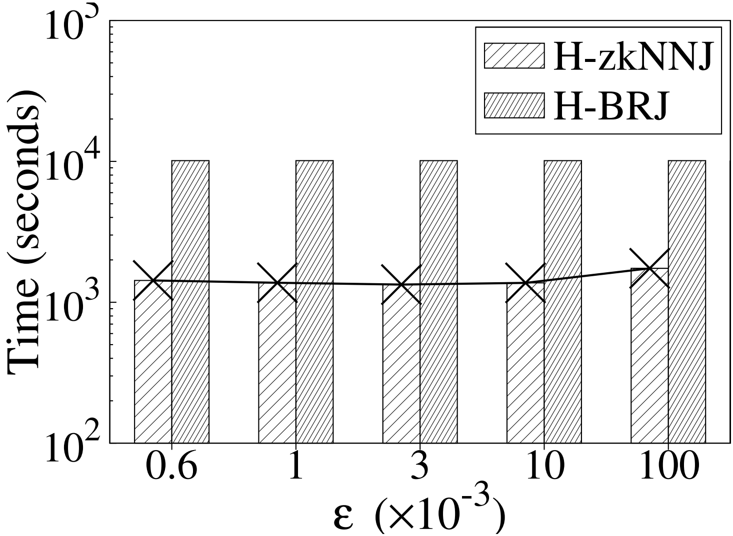
H-zkNNJ: Hadoop *z*-value *k*NN Join



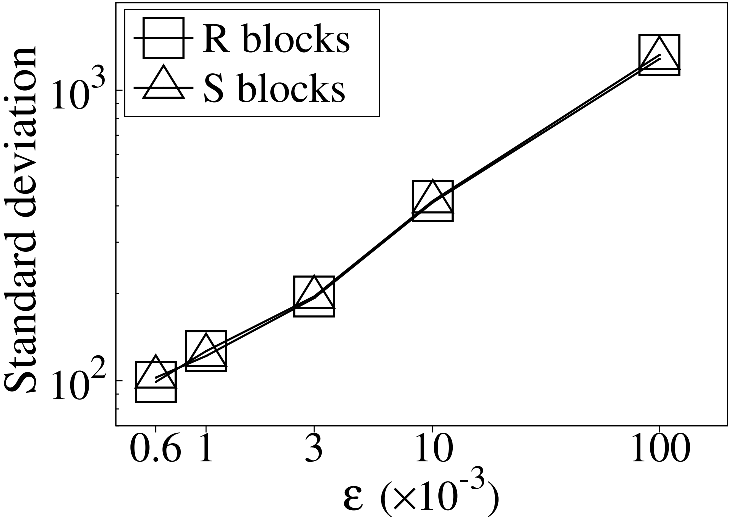
Experiments: Approximation quality



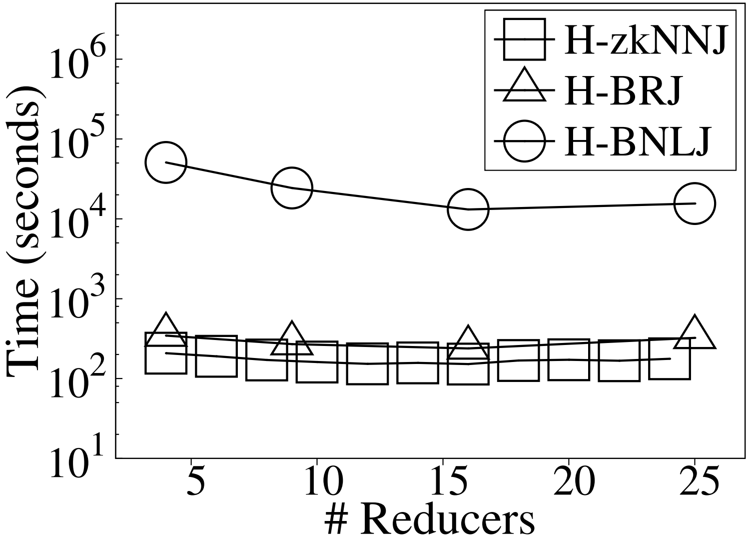
Experiments: Effect of ε



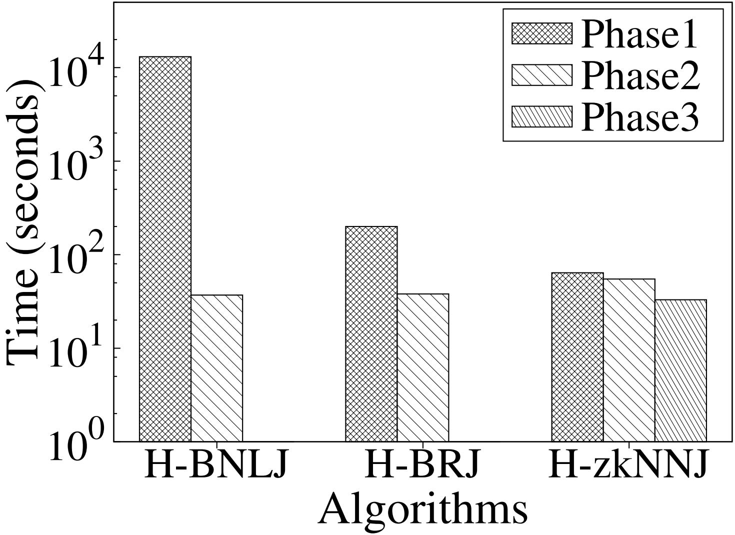
Experiments: Effect of ε



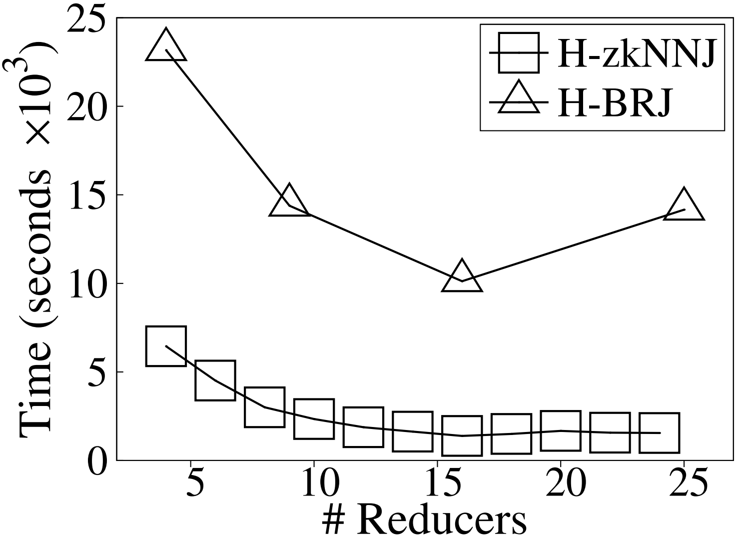
Experiments: Evaluation of H-BNLJ



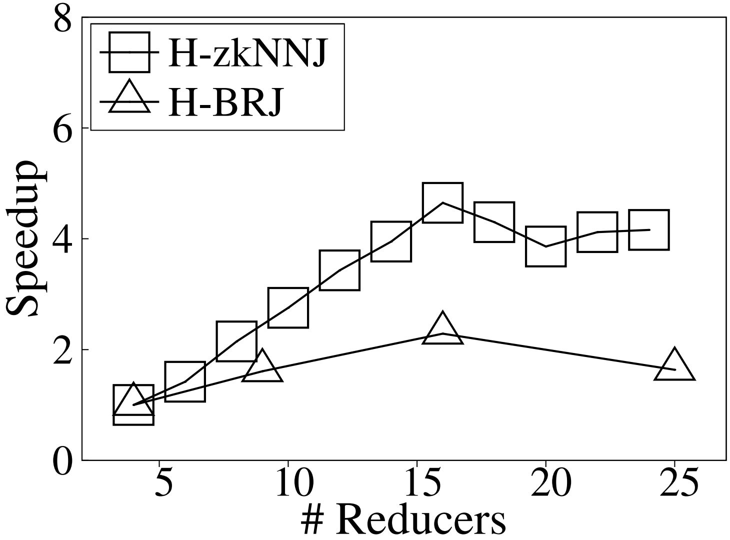
Experiments: Evaluation of H-BNLJ



Experiments: Speedup

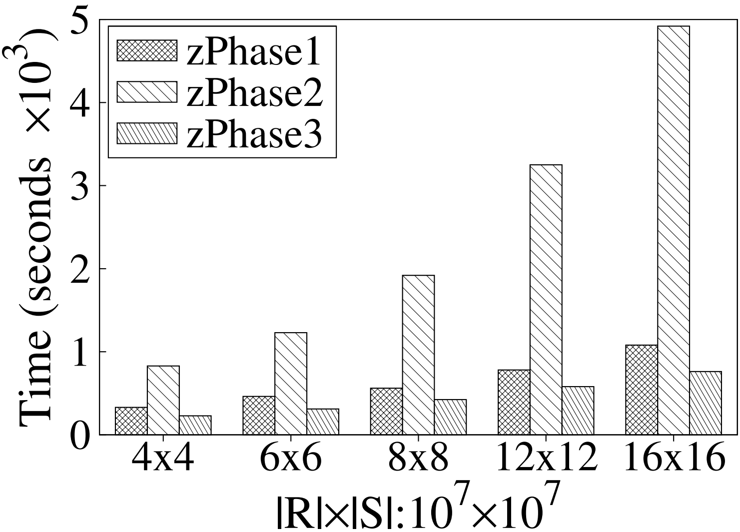


Experiments: Speedup



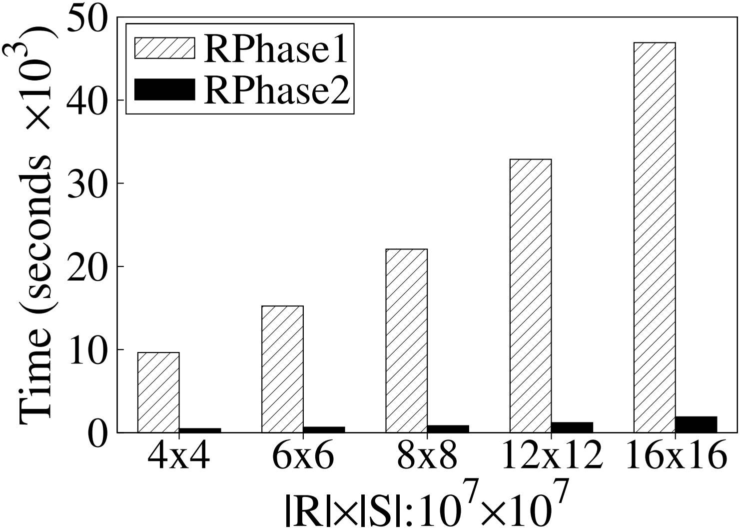
Experiments: Running time and communication cost

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



Experiments: Running time and communication cost

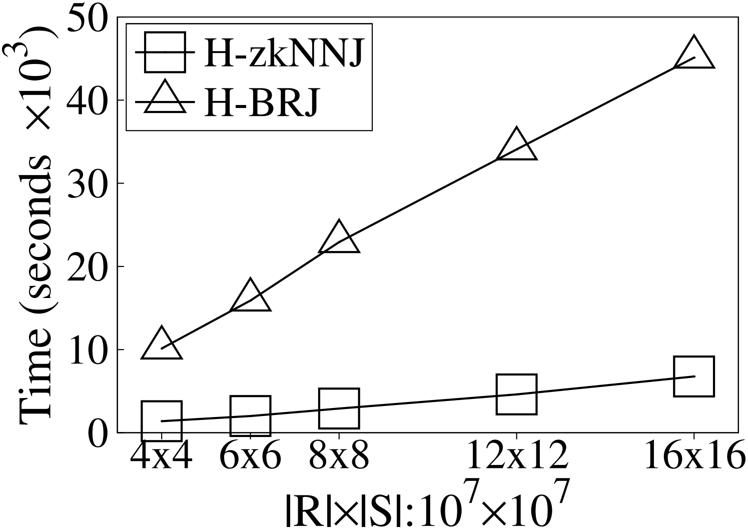
H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



Experiments: Running time and communication cost

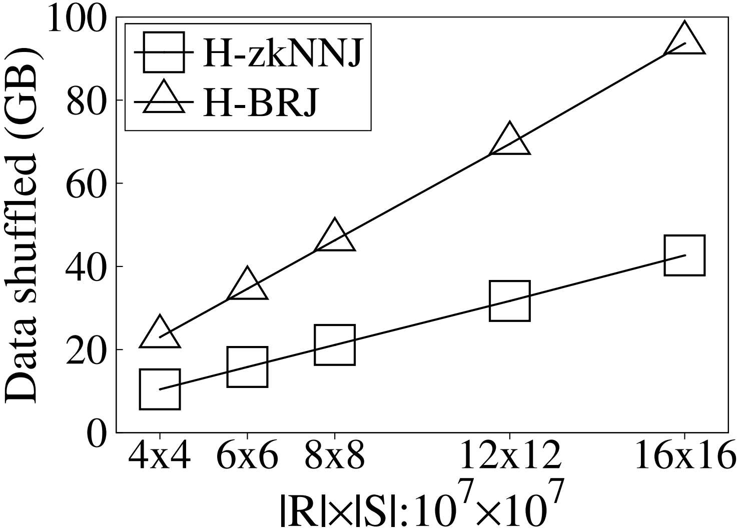
H-zkNNJ: Hadoop *z*-value *k*NN Join

H-BRJ: Hadoop Block R-tree Join



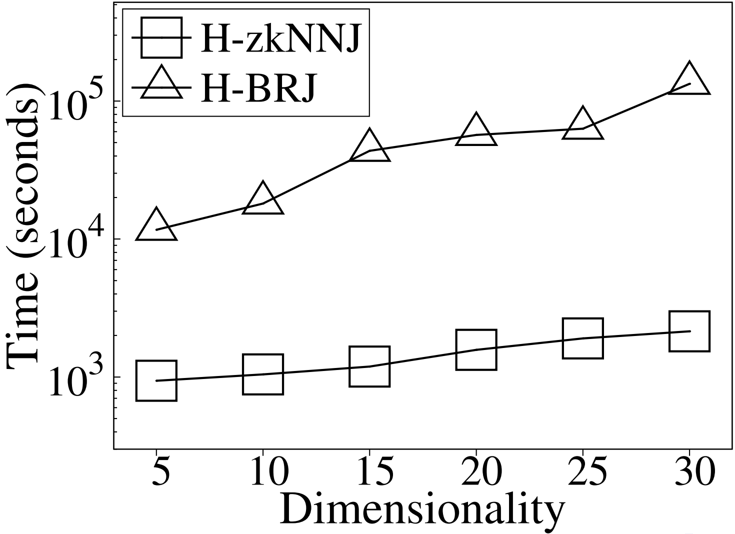
Experiments: Running time and communication cost

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



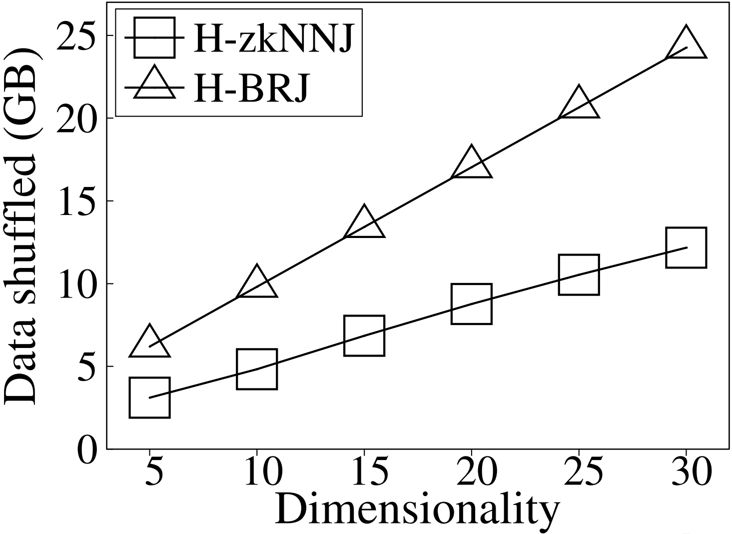
Experiments: Effect of d

H-zkNNJ: Hadoop z-value kNN Join
H-BRJ: Hadoop Block R-tree Join



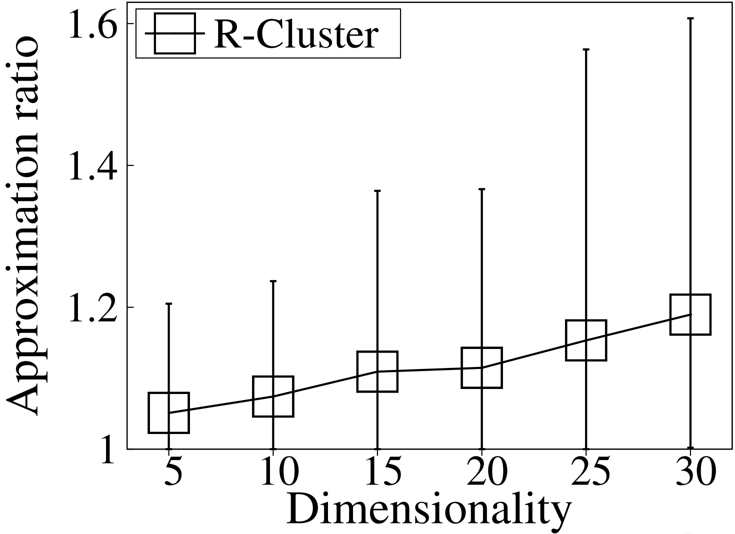
Experiments: Effect of *d*

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



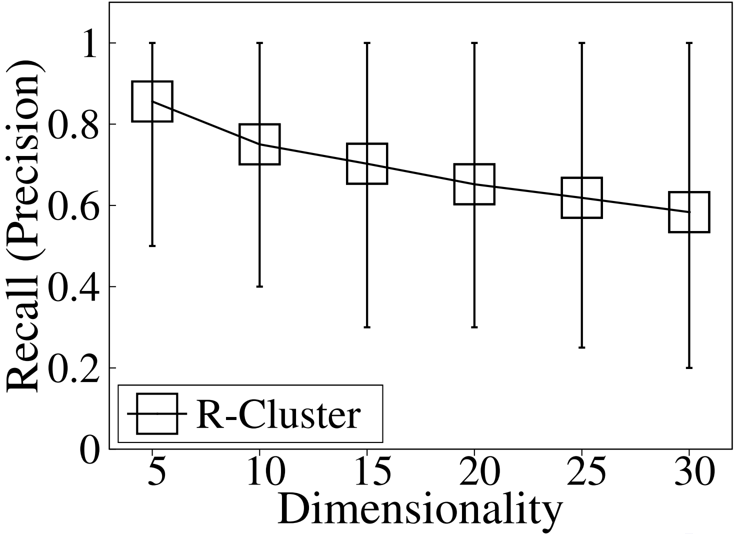
Experiments: Effect of *d*

H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join

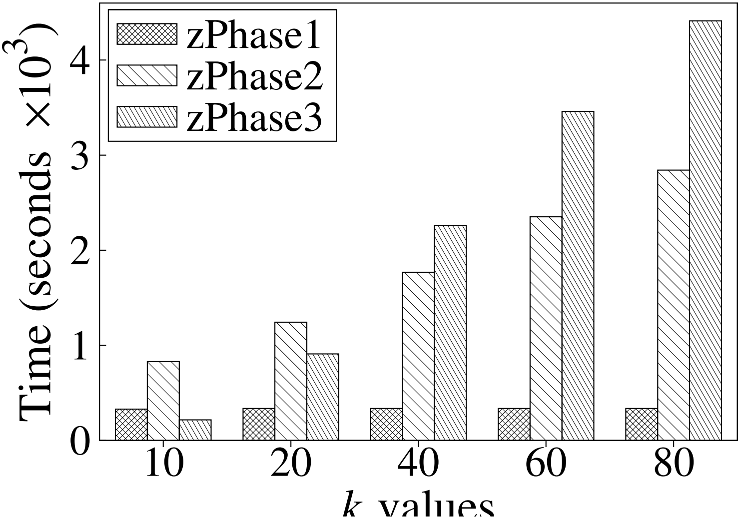


Experiments: Effect of d

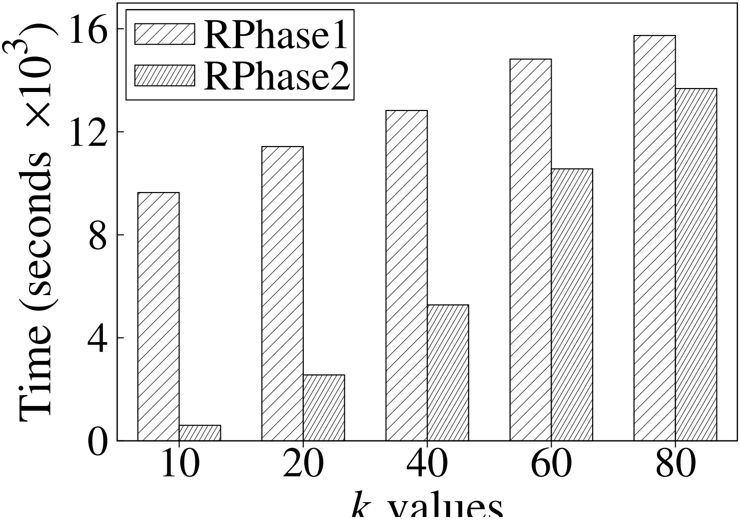
H-zkNNJ: Hadoop *z*-value *k*NN Join
H-BRJ: Hadoop Block R-tree Join



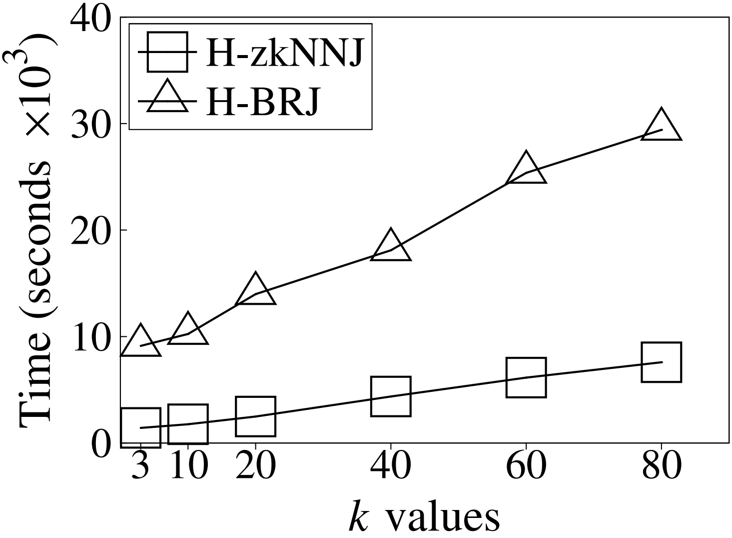
Experiments: Effect of k



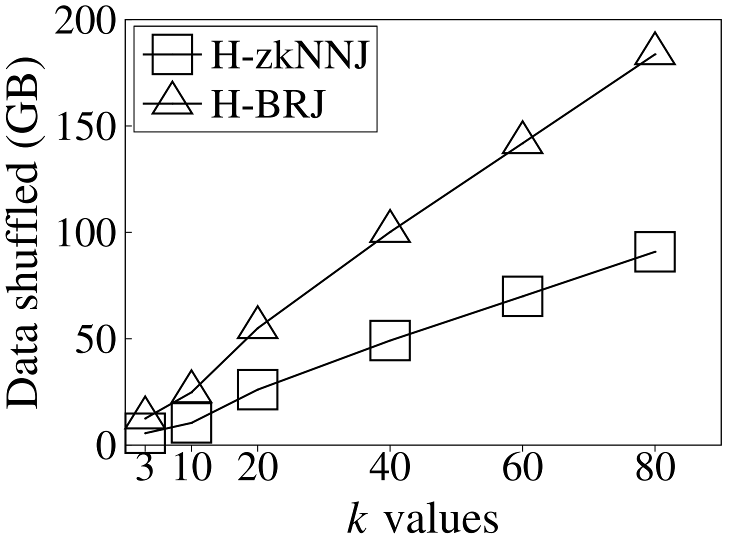
Experiments: Effect of *k*



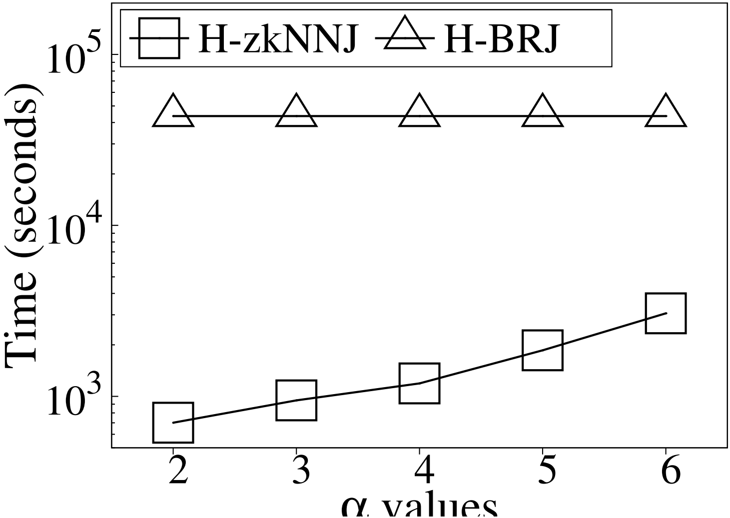
Experiments: Effect of *k*



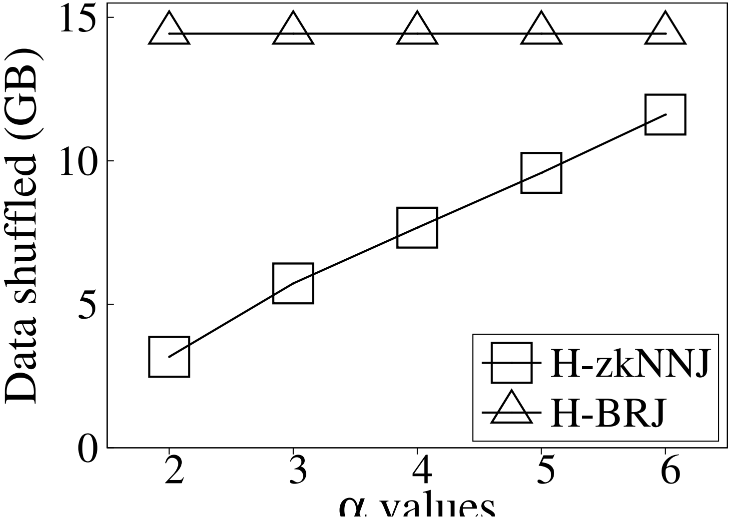
Experiments: Effect of *k*



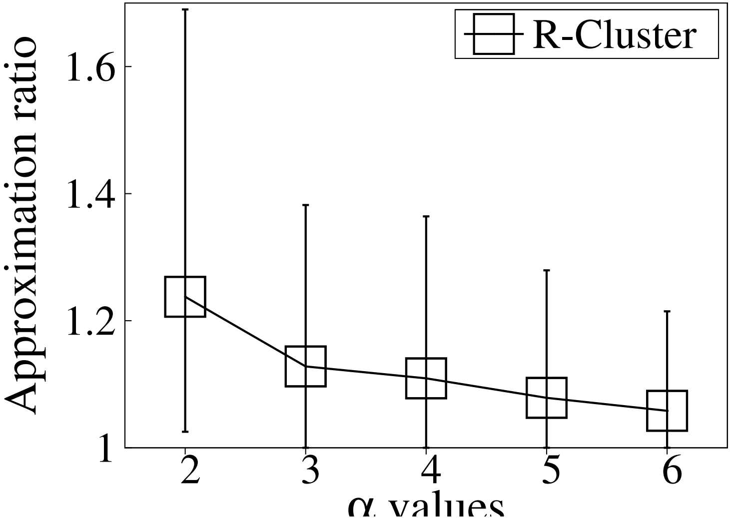
Experiments: Effect of number of shifts α



Experiments: Effect of number of shifts α



Experiments: Effect of number of shifts α



Experiments: Effect of number of shifts α