Sensors **2015**, *15*, 6105-6132; doi:10.3390/s150306105

OPEN ACCESS

sensors

ISSN 1424-8220

www.mdpi.com/journal/sensors

Article

In-Network Processing of an Iceberg Join Query in Wireless Sensor Networks Based on 2-Way Fragment Semijoins

Hyunchul Kang

School of Computer Science and Engineering, Chung-Ang University, Seoul 156-756, Korea;
E-Mail: hckang@cau.ac.kr; Tel.: +82-2-820-5306; Fax: +82-2-826-0853

Academic Editor: Leonhard M. Reindl

Received: 11 October 2014 / Accepted: 2 March 2015 / Published: 12 March 2015

Abstract: We investigate the in-network processing of an iceberg join query in wireless sensor networks (WSNs). An iceberg join is a special type of join where only those joined tuples whose cardinality exceeds a certain threshold (called iceberg threshold) are qualified for the result. Processing such a join involves the value matching for the join predicate as well as the checking of the cardinality constraint for the iceberg threshold. In the previous scheme, the value matching is carried out as the main task for filtering non-joinable tuples while the iceberg threshold is treated as an additional constraint. We take an alternative approach, meeting the cardinality constraint first and matching values next. In this approach, with a logical fragmentation of the join operand relations on the aggregate counts of the joining attribute values, the optimal sequence of 2-way fragment semijoins is generated, where each fragment semijoin employs a Bloom filter as a synopsis of the joining attribute values. This sequence filters non-joinable tuples in an energy-efficient way in WSNs. Through implementation and a set of detailed experiments, we show that our alternative approach considerably outperforms the previous one.

Keywords: iceberg join query; wireless sensor networks; sensor network database; 2-way fragment semijoin; optimal sequence of 2-way fragment semijoins

1. Introduction

In wireless sensor networks (WSNs), the values sampled by a senor node can be modeled as a relational tuple that consists of the sensor readings as its main attributes and often of the node ID, the timestamp of the sampling, the location of the node, etc. as its auxiliary attributes [1]. Thus, for a region

of WSNs, the sensor readings of the nodes deployed in the region can be modeled as a *virtual* relation physically distributed across the senor nodes in the region. In WSN applications, which include vehicle surveillance, environment monitoring, animal habitat monitoring, and climate research to name just a few, a relational *join* query can be issued against two virtual relations. For example, based on the scenario in vehicle surveillance presented in [2], let us consider the identification of the moving objects that have passed two particular regions in WSNs. In each region, the ID of a passing object is sampled and stored with the time of passage. Then, the sets of sensor readings stored in the two regions are modeled as virtual relations denoted as $\mathcal{R}_0$ and $\mathcal{R}_1$. The following join query is to retrieve the ID and time of the objects that have passed both of the two regions:

SELECT $\mathcal{R}_0$.*ID*, $\mathcal{R}_0$.*timestamp*, $\mathcal{R}_1$.*timestamp*
FROM $\mathcal{R}_0$, $\mathcal{R}_1$
WHERE $\mathcal{R}_0$.*ID* = $\mathcal{R}_1$.*ID*

Since a join is an important type of query in WSNs to monitor the *correlations* among the senor readings, processing of joins in WSNs has received much attention. A survey of the state-of-the-art techniques is presented in [3]. A naïve method to answer a join query, $\mathcal{R}_0 \bowtie \mathcal{R}_1$, in WSNs is the *external* join, whereby all the tuples of $\mathcal{R}_0$ and $\mathcal{R}_1$ are sent to the base station where the result of the join is produced. In WSNs, the power in a node is consumed the most when the node transmits data [4]. Thus, the external join is not energy-efficient, and the state-of-the-art techniques conduct *in-network* processing of joins.

In this paper, we investigate the in-network processing of a special type of equijoin query called *iceberg join* in WSNs. It is to retrieve the *frequent* patterns of correlation among the sensor readings. For a joining attribute value *v*, it contributes to the join result only if the number of joined tuples for *v* exceeds some given threshold. This join frequency threshold is called *iceberg threshold* and denoted as α throughout this paper. Figure 1a shows an example of iceberg join of two relations $\mathcal{R}_0$ and $\mathcal{R}_1$ with $\alpha = 2$, denoted as $\mathcal{R}_0 \bowtie^{\alpha=2}_{A=A} \mathcal{R}_1$. A query retrieving attribute A from this iceberg join can be expressed in SQL as in Figure 1b.

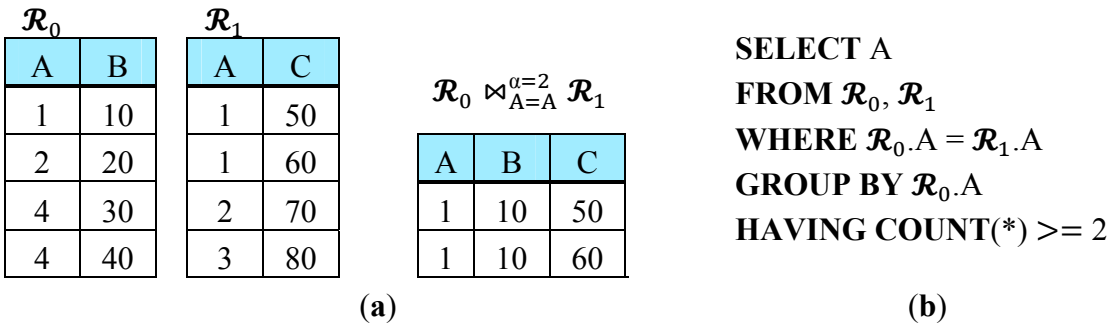


Figure 1. An Example of an Iceberg Join Query. (**a**) Join operand relations $\mathcal{R}_0$ and $\mathcal{R}_1$, and the result of their iceberg join with $\alpha = 2$; (**b**) SQL expression retrieving attribute A from the iceberg join in (**a**).

In bird habitat monitoring, sensor nodes can be deployed to sample bird songs when birds are singing. From these audio samples, their fingerprints are generated, stored, and later used to recognize the bird species and to estimate their population size in certain regions [5]. For two regions of interest in WSNs,

an iceberg join query can be issued to retrieve the fingerprints that are frequently sampled in both regions in studying regional correlations in bird population. Let $\mathcal{R}_0$ and $\mathcal{R}_1$ denote the virtual relations storing the fingerprints in the two regions. Then, this query can be expressed as:

SELECT $\mathcal{R}_0$.*fingerprint*
FROM $\mathcal{R}_0$, $\mathcal{R}_1$
WHERE $\mathcal{R}_0$.*fingerprint* = $\mathcal{R}_1$.*fingerprint*
GROUP BY $\mathcal{R}_0$.*fingerprint*
HAVING COUNT(*) >= α

The iceberg join is an important type of query in WSNs because the frequent or prominent phenomena of interest in terms of correlations among sensor readings could be detected in a more energy-efficient way than with the conventional joins. Considering the resource constraints in WSNs and the cross-references required in join processing, the processing of the conventional types of join queries in WSNs could be too expensive [3]. The efficient processing of iceberg joins in WSNs deserves attention but so far little work has been reported. The most relevant ones include the schemes proposed in [6–8].

An iceberg join operation involves the checking of the *join predicate* and of the *cardinality constraint* between the tuples of the join operand relations. There could be two approaches depending on which condition of the two is the primary one to check. The primary condition is checked first, and for the tuples that satisfy it, the remaining condition is checked next. It would be reasonable to regard the join predicate as an intrinsic requirement of a join operation while treating the cardinality constraint as an additional one. In [6], such a view was taken, and a scheme called *SRJA* (Synopsis Refinement iceberg-Join Algorithm) was proposed, where a histogram-based synopsis of the joining attribute value ranges is transmitted for filtering non-joinable tuples. In [6], it was shown that SRJA significantly outperformed the baseline schemes. In this paper, we investigate an *alternative* approach where the cardinality constraint is checked first as the primary condition and then for those tuples that satisfy it, the join predicate is checked. We show that this approach is substantially superior to the other one. The contributions of this paper are as follows:

- We consider a logical *fragmentation* of join operand relations based on the aggregate counts of the joining attribute values, proposing a *2-way fragment semijoin* operation using a Bloom filter as a synopsis of the joining attribute values. In the *backward reduction* of the 2-way fragment semijoin, the false positives inherent with the Bloom filter are efficiently handled.
- We take advantage of the *Highest Count First* strategy with which efficient reduction of the join operand relation (called *Low Count Cut*) occurs, developing a dynamic programming algorithm that generates the *optimal* sequence of 2-way fragment semijoins. The Highest Count First strategy is shown to be more effective in filtering non-joinable tuples than the transmissions of the value ranges widely used in WSNs.
- Through implementation and a set of detailed experiments, we show that our approach considerably outperforms the previous one.

The rest of this paper is organized as follows: in Section 2, the problem statement is given. In Section 3, the background for our scheme is presented. In Section 4, an overview of our approach is given. In Section 5, the optimization with a dynamic programming algorithm is described. In Section 6, the performance of our scheme is compared with that of SRJA. In Section 7, related work is presented.

Finally, in Section 8, the conclusions are drawn and the future work is given. Notations used in this paper are summarized in Table 1 of Section 4.

2. Problem Statement

An iceberg join query Q for two virtual relations $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ on attribute A in WSNs is assumed to be submitted to the base station as a continuous query modeled as a sliding window join [9]. Each evaluation of the query is conducted against a window of $\boldsymbol{\mathcal{R}}_0$ and that of $\boldsymbol{\mathcal{R}}_1$, and the query result is returned to the base station. Initially, the base station forwards Q to three sensor nodes $\tilde{n}_0$, $\tilde{n}_1$, and $\tilde{m}$. $\tilde{n}_i$ is the coordinator node of the region for $\boldsymbol{\mathcal{R}}_i$, $i = 0, 1$. $\tilde{m}$ is the node located at the midpoint between $\tilde{n}_0$ and $\tilde{n}_1$, which is to take part in in-network query optimization and processing. In each region, a *routing tree* whose root is $\tilde{n}_i$ is constructed with the standard routing tree construction algorithm of [1]. $\tilde{n}_i$ disseminates Q to all the sensor nodes in the region. In each region, *preprocessing* is carried out to collect the aggregate count of each joining attribute value that is sampled in the window. Each sensor node in a region generates a *node histogram*, which is a set of (value, count) pairs in the node. Then, it sends the node histogram to its parent node in the routing tree. Eventually, $\tilde{n}_i$ obtains the *region histogram*. It is a binary relation with the joining attribute A and the *count* attribute. In [6], this relation is called the *base histogram*. Let us denote the base histogram of region $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ as R_0 and R_1, respectively. Now the problem is to fully reduce R_0 and R_1 such that only those tuples of them that are qualified for the iceberg join remain. Let R'_0 and R'_1 respectively denote the full reduction of R_0 and R_1. Then, the final result of Q can be obtained by two semijoins followed by a final join: $(\boldsymbol{\mathcal{R}}_0 \ltimes_{A=A} R'_0) \bowtie_{A=A} (\boldsymbol{\mathcal{R}}_1 \ltimes_{A=A} R'_1)$. Once R'_0 and R'_1 are obtained, the operations to produce the final result of Q are straightforward. Thus, in this paper, we deal only with the problem of optimally obtaining R'_0 and R'_1.

3. Background

Our scheme employs the Bloom filter [10] and a variation of the 2-way semijoin [11,12]. In Section 3.1, semijoin and 2-way semijoin operations are briefly described. In Section 3.2, an overview of Bloom filter and its theory are given. In Section 3.3, the technique of SRJA [6] is described.

3.1. Semijoin and 2-Way Semijoin

The semijoin was proposed in [13] as a means to reduce join operand relations in distributed databases. Given a join $R_0 \bowtie_{A=A} R_1$, semijoin $R_0 \rtimes_{A=A} R_1$ reduces R_1 such that only those tuples of R_1 joinable with R_0 remain. Assuming that R_0 and R_1 reside at different sites in distributed databases, the semijoin $R_0 \rtimes_{A=A} R_1$ is executed by sending R_0[A] to R_1 and joining the two. That is, $R_0 \rtimes_{A=A} R_1 = R_0[\mathrm{A}] \bowtie_{A=A} R_1$. R_1 is said to be fully reduced by the semijoin. Let us call R_0 the *reducer relation*, and R_1 the *reduced relation*. The size of the result of semijoin $R_0 \rtimes_{A=A} R_1$ is $|R_1| \cdot s$, where s is called the semijoin selectivity and it is estimated as $|R[\mathrm{A}]|/|D_A|$, where D_A is the domain of A [14].

The 2-way semijoin was investigated in [11,12,15] as an extension of the semijoin. It includes the *backward reduction* phase in addition to the forward reduction phase of the original semijoin such that both of the two join operand relations are fully reduced. Suppose semijoin $R_0 \rtimes_{A=A} R_1$ reduces R_1 to R'_1. In the backward reduction phase of 2-way semijoin $R_0 \rtimes\ltimes_{A=A} R_1$, R'_1[A] or its complement (*i.e.*,

$R_1[\mathrm{A}] - R'_1[\mathrm{A}]$) or a bit vector indicating which value of $R_1[\mathrm{A}]$ is joinable and which is not is sent back. In distributed query processing, this backward reduction is often effective and can be applied to a pipelined *n*-way join [12]. In our scheme proposed in this paper, the backward reduction is efficiently merged with the handling of the false positives inherent with the Bloom filter employed in implementing a semijoin.

3.2. Bloom Filter

A *k*-transform Bloom filter is a bit vector of length *m* that probabilistically represents a set *S* with *k* hash functions $h_1(),\cdots, h_k()$ for $k \geq 1$ [10]. Initially, all the *m* bits are set to 0. For each value *x* in *S*, $h_1(x)$-th, $\cdots$, $h_k(x)$-th bits of the Bloom filter are set. For a given value *y*, *y* is not in *S* if any of the $h_1(y)$-th, $\cdots$, $h_k(y)$-th bits of the Bloom filter is not 1, whereas *y* is *probably* in *S* if all those *k* bits are 1. In the latter case, a *false positive* is possible due to the possibility of the collisions in hashing. However, it is guaranteed that false negatives are not possible.

For two relations R_0 and R_1 that reside at different locations, a Bloom filter can be employed in implementing semijoin $R_0 \rtimes_{A=A} R_1$ with possible errors. First, R_0 is scanned to construct a Bloom filter $bf\{R_0, A\}$ that represents $R_0[\mathrm{A}]$. $bf\{R_0, A\}$ is sent to R_1. Then, for each value *v* of R_1.A, the membership test of $v \in bf\{R_0, A\}$ is done to filter non-joinable tuples of R_1. None of the joinable tuples of R_1 is filtered out but some of the non-joinable ones could survive because of the false positives (*i.e.*, $R_0 \rtimes_{A=A} R_1 \subseteq R_0 \rtimes_{A=A}^{bf} R_1 \subseteq R_1$, where $\rtimes_{A=A}^{bf}$ denotes the semijoin implemented using a Bloom filter). Since $bf\{R_0, A\}$ is a bit vector, it is usually much smaller than $R_0[\mathrm{A}]$. In WSNs, it would be more energy-efficient to send $bf\{R_0, A\}$ than to send $R_0[\mathrm{A}]$, provided that the false positives could be properly handled.

When the length of a *k*-transform Bloom filter is *m*, and the number of values in *S* is *n*, the probability of a false positive is approximately:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (1)$$

and it is minimized to $1/2^k$ when $k = (m/n) \cdot ln\, 2$ [16,17]. In this case, the number of bits used to represent a value in *S* is $m/n = k/ln\, 2$.

3.3. SRJA

Given an iceberg join of $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ on attribute *A* in WSNs, SRJA works as follows [6]: The preprocessing as described in Section 2 is carried out to obtain R_0 and R_1. At $\tilde{n}_0$ and $\tilde{n}_1$, the values in $R_0[\mathrm{A}]$ and $R_1[\mathrm{A}]$ is respectively divided into a sequence of *value ranges* with the information on the *count* attribute associated with each range. A range is defined as a 4-tuple (*minval*, *maxval*, *mincount*, *maxcount*). *minval* and *maxval* are respectively the minimum and maximum value of *A* in the range, and the *mincount* and *maxcount* are respectively the minimum and maximum value of *count* among all the counts associated with the values of *A* in the range. The sequence of these ranges constitutes the synopsis of the values of *A* and *count* in R_i (i = 0, 1). A value range represented as an interval would be much smaller than the list of all the values in the range. In WSNs, it is a common practice to send a value range (*i.e.*, an interval [*minval*, *maxval*]) instead of the full list of values in the range to reduce data transmission cost though accuracy is compromised [18].

Protocol Executed by $\tilde{n}_0$ for SRJA

```
1: begin
2:    process the values with a high count;  // optimization 1
3:    initialize the synopsis of R0;
4:    process sparse subranges;  // optimization 2
5:    loop
6:       J-flag = D-flag = FALSE;
7:       send the synopsis to m̃;
8:       receive the tagged synopsis from m̃;
9:       if( no subrange to process in the received synopsis ) break;
10:      if( exist(JOIN-tagged subrange) ) J-flag = TRUE;
11:      if( exist(DIVIDE-tagged subrange) ) D-flag = TRUE;
12:      process PRUNE-tagged subranges;
13:      if( J-flag ) process JOIN-tagged subranges;
14:      if( D-flag ) then
15:         divide DIVIDE-tagged subranges;
16:         process sparse subranges;
17:      endif
18:   endloop
19: end
```

Figure 2. Protocol Executed by $\tilde{n}_0$ for SRJA.

$\tilde{n}_0$ and $\tilde{n}_1$ send the synopses of R_0 and R_1 to the sensor node $\tilde{m}$ located at the midpoint between $\tilde{n}_0$ and $\tilde{n}_1$. At $\tilde{m}$, the value range matching is conducted first with the *minval* and *maxval* of the ranges in the synopses. In this matching process, the original synopses are modified. Some ranges of a synopsis are deleted because there is no matched counterpart in the other synopsis or further divided into subranges so that the two synopses have exactly the same set of ranges. Then, for each pair of matched ranges of R_0 and R_1, the *mincount*'s and *maxcount*'s are checked, and the ranges are tagged as PRUNE, JOIN, or DIVIDE. The pair for which $maxcount(R_0) \times maxcount(R_1) < \alpha$ is tagged as PRUNE. The pair for which $mincount(R_0) \times mincount(R_1) \geq \alpha$ is tagged as JOIN. The remaining pairs are tagged as DIVIDE. The tagged synopses are sent back to $\tilde{n}_0$ and $\tilde{n}_1$. The tuples in a PRUNE range are deleted from R_i (i = 0, 1). The tuples in a JOIN range not qualified for the query are filtered out by 2-way semijoins, and the qualified ones are finally joined for the result. A DIVIDE range is further divided into subranges. The synopsis of R_i is reconstructed with them. This process is repeated until the final query result is obtained. The optimization techniques for SRJA are as follows [6]:

- Optimization 1: Before the above process begins, the tuples whose value of *count* exceeds α is sent separately for checking their joinability because they are likely to be qualified for the query.
- Optimization 2: The tuples whose value of A belongs to a sparse range are also separately sent. Eliminating a sparse range would make the synopsis more selective.
- Optimization 3: For a range to be tagged as DIVIDE, the *maxcount* of the opposite range (*opp_maxcount*) is also sent when the tagged synopses are sent back. When the range is divided into subranges, the tuples which turn out not to be qualified for the query ($count \times opp\_maxcount < \alpha$) are deleted.

The skeleton of the protocol executed by $\tilde{n}_0$ for SRJA is described in Figure 2. The one for $\tilde{n}_1$ is symmetrical. In [6], it was shown that SRJA significantly outperformed the following baseline schemes:

- NAÏVE: The external join where all the tuples of $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ are sent to the base station.
- SIJ: The synopsis join of [2] extended for iceberg joins where R_0 and R_1 are sent to $\tilde{m}$ and fully reduced there.

4. Overview of Our Approach

Given an iceberg join of $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ on attribute A in WSNs, suppose the preprocessing described in Section 2 has been carried out to obtain R_0 and R_1. In this section, the main components of our scheme in fully reducing R_0 and R_1 are described. They include *2-way fragment semijoin*, *Low Count Cut*, *Highest Count First strategy*, and the *sequence* of 2-way fragment semijoins. The issue of optimization is dealt with in Section 5. The notations used in this section and Section 5 are summarized in Table 1.

Table 1. Notations.

Notation	Description
R_i, R_{1-i}	R_0 and R_1 when i = 0; R_1 and R_0 when i = 1
α	Iceberg threshold
$\tilde{n}_i$	The coordinator sensor node in the region for R_i (i = 0, 1)
$\tilde{m}$	The sensor node located at the midpoint between $\tilde{n}_0$ and $\tilde{n}_1$.
A	The joining attribute
D_A	The domain of A
$\|\|A\|\|$	The size of a value of A
$\overline{\ltimes}$, $\overline{\rtimes}$	2-way fragment semijoin operator
R_i^n	A logical fragment of R_i defined as $\sigma_{count\,=\,n}\, R_i$ (i = 0, 1)
M_i	The initial maximum value of the attribute count in R_i (i = 0, 1)
H_i	The current highest value of the attribute count in R_i (i = 0, 1)
L_i	The current lowest value of the attribute count in R_i (i = 0, 1)
$R_i(H_i, H_{1-i})$	The current state of R_i after reduced by a sequence of fragment semijoins (i = 0, 1)
$R_i^n(H_i, H_{1-i})$	A logical fragment of $R_i(H_i, H_{1-i})$ defined as $\sigma_{count\,=\,n}\, R_i(H_i, H_{1-i})$ (i = 0, 1)
$\overline{\ltimes}_i\,(H_i, H_{1-i})$	A fragment semijoin for which $R_i^{H_i}(H_i, H_{1-i})$ is the reducer relation
$S^i_{\overline{\ltimes}*}(H_i, H_{1-i})$	Given $R_i(H_i, H_{1-i})$ and $R_{1-i}(H_{1-i}, H_i)$, the optimal sequence of fragment semijoins that fully reduces R_0 and R_1 provided that the first semijoin is $\overline{\ltimes}_i\,(H_i, H_{1-i})$ (i = 0, 1)
$C^i_{\overline{\ltimes}*}(H_i, H_{1-i})$	The cost of $S^i_{\overline{\ltimes}*}(H_i, H_{1-i})$
$\hat{c}^i_{\overline{\ltimes}}(H_i, H_{1-i})$	The cost of $\overline{\ltimes}_i\,(H_i, H_{1-i})$
$n^i_{\overline{\ltimes}*}(H_i, H_{1-i})$	The value of attribute count in R_i with which (Equation (2)) is minimized (i = 0, 1). Let $n = n^i_{\overline{\ltimes}*}(H_i, H_{1-i})$. Then, the following sequence of fragment semijoins is the prefix of $S^i_{\overline{\ltimes}*}(H_i, H_{1-i})$: $\overline{\ltimes}_i\,(H_i, H_{1-i}) \cdot \overline{\ltimes}_i\,(H_i - 1, H_{1-i}) \cdot \overline{\ltimes}_i\,(H_i - 2, H_{1-i}) \cdots \overline{\ltimes}_i\,(n, H_{1-i})$.
$BF_i(H_i, H_{1-i})$	The Bloom filter sent for $\overline{\ltimes}_i\,(H_i, H_{1-i})$
$\|\|BF_i(H_i, H_{1-i})\|\|$	The size of $BF_i(H_i, H_{1-i})$
$k_i\,(H_i, H_{1-i})$	The optimal number of hash functions used for $BF_i(H_i, H_{1-i})$
$C^i_{fp}(H_i, H_{1-i})$	The cost of handling false positives with $BF_i(H_i, H_{1-i})$ in executing $\overline{\ltimes}_i\,(H_i, H_{1-i})$

4.1. 2-Way Fragment Semijoin

As described in Section 2, R_0 and R_1 are the base histograms of $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ with attribute *count*. For relation R_i ($i = 0, 1$), let R_i^n denote the result of $\sigma_{count\,=\,n}\, R_i$. R_i^n is a horizontal *fragment* of R_i on count, and thus, a horizontal subset of the base histogram of $\boldsymbol{\mathcal{R}}_i$. For example, given an iceberg join between $\boldsymbol{\mathcal{R}}_0$ and $\boldsymbol{\mathcal{R}}_1$ in Figure 1 on attribute A, $R_0 = \{(\mathrm{A}, \mathrm{count})\,|(1,1),(2,1),(4,2)\}$, and $R_1 = \{(\mathrm{A}, \mathrm{count})\,|(1,2),(2,1),(3,1)\}$. Thus, $R_0^1 = \{(1,1),(2,1)\}$, $R_0^2 = \{(4,2)\}$, $R_1^1 = \{(2,1),(3,1)\}$, $R_1^2 = \{(1,2)\}$.

Let us consider a *logical* fragmentation of R_0 and R_1 on count. Suppose the lowest and highest value of count in R_0 is 1 and 7, respectively. Then, R_0 is logically partitioned into 7 fragments: R_0^1, R_0^2, …, R_0^7. Similarly, suppose R_1 is logically fragmented into R_1^1, …, R_1^8. Now let us consider an iceberg join with $\alpha = 30$. Since R_0 and R_1 are fragmented, the semijoin where a fragment is the reducer relation can be used. For example, semijoin $R_0^7 \rtimes_{A=A} R_1$ is executed in the following way: $R_0^7[\mathrm{A}]$ is sent to R_1. Since $\lceil \alpha/7 \rceil = 5$, only those tuples of R_1 that belong to the fragments R_1^j where $5 \le j \le 8$ could be joinable with the tuples in R_0^7. Thus, only $\bigcup_{j=5}^{8} R_1^j$ could be considered as the reduced relation. The tuples in those fragments not joinable with $R_0^7[\mathrm{A}]$ are deleted.

Now let us define a new type of operation called *fragment semijoin* by modifying the semijoin. In the semijoin, the joinable tuples of the reduced relation remain whereas the non-joinable ones are deleted. In our fragment semijoin, the tuple filtering is done in the other way around with a side-effect. The joinable ones are deleted whereas the non-joinable ones remain. In other words, the result of a fragment semijoin is the complement of the conventional semijoin (in some commercial DBMSs, this variation of the semijoin is called an *anti join*; in fact, the term *fragment anti join* might be more exact one than fragment semijoin, however, we keep the term semijoin with a modifier "fragment" because it is widely known). Let us denote a fragment semijoin operator as $\overline{\rtimes}$. With $\alpha = 30$, a fragment semijoin $R_0^7 \,\overline{\rtimes}_{A=A}\, R_1$ is executed in the following way: $R_0^7[\mathrm{A}]$ is sent to R_1. The tuples in $\bigcup_{j=5}^{8} R_1^j$ not joinable with $R_0^7[\mathrm{A}]$ are intact (*i.e.*, not deleted). Instead, the joinable ones are deleted and inserted to a separate relation R'_1. The reason why the unmatched tuples remain is that they might be joined with the tuples in other fragments of R_0. The insertion of matched tuples to R'_1 is a necessary side-effect of a fragment semijoin. Initially, R'_1 is empty. Every time a fragment semijoin to reduce R_1 is executed, the matched tuples, if any, are inserted to R'_1. When *all* the tuples of R_1 that are joinable with R_0 have been inserted to R'_1, R_1 is said to be fully reduced to R'_1. Until then, R_1 is said to be reduced to R''_1, which keeps the tuples of R_1 yet to be checked for joinability with other fragments of R_0. Note that $R'_1 \cup R''_1 \subseteq R_1$. The difference, $R_1 - (R'_1 \cup R''_1)$, is the set of tuples of R_1 that have been finally confirmed not joinable with R_0. How this difference is computed will be explained in the next two subsections. In a *2-way* fragment semijoin, the backward reduction phase is added where the matched joining attribute values are sent back.

So far, we have assumed that the joining attribute values are sent in the forward reduction phase. In our scheme, we employ the Bloom filter as a synopsis of the joining attribute values in executing a fragment semijoin to reduce the amount of data transmission in WSNs. The fragment semijoin with a Bloom filter is the same as above except (1) the Bloom filter constructed from the joining attribute values are sent; and (2) the false positives need to be handled. With $\alpha = 30$, the fragment semijoin $R_0^7 \,\overline{\rtimes}_{A=A}\, R_1$ using a Bloom filter is executed in the following way: The values in $R_0^7.\mathrm{A}$ are represented in a Bloom filter, $bf\{R_0^7, A\}$, which is sent to R_1. For each value of A in $\bigcup_{j=5}^{8} R_1^j$, the membership test is done with

$bf\{R_0^7, A\}$. The matched tuples (including those due to false positives) are moved to R'_1. Their values of A are sent back to R_0. The ones that turn out to have been sent due to false positives are sent back to R_1, and their corresponding tuples are moved from R'_1 back to their original fragments. Note that the backward reduction phase is mandatory with the fragment semijoin using a Bloom filter to sort out false positives. In the rest of this paper, what we mean by a fragment semijoin denoted with $\overline{\rtimes}$ is a 2-way fragment semijoin using a Bloom filter unless stated otherwise.

4.2. Low Count Cut (LCC)

In our approach, the cardinality constraint is the primary condition to check. One of the advantages we gain by checking the cardinality constraint first is that we can delete those tuples of R_0 and R_1 with low counts without checking the join predicate. Let M_0 and M_1 respectively denote the maximum count in R_0 and R_1. Then, the tuples in the fragments R_0^j ($j \leq \lceil \alpha / M_1 \rceil$) and R_1^j ($j \leq \lceil \alpha / M_0 \rceil$) cannot meet the cardinality constraint, and they need not be considered at all. For example, suppose R_0 is fragmented into R_0^1,…, R_0^7 whereas R_1 is fragmented into R_1^1,…, R_1^{10}, and α = 20. R_0^1 is ignored for the join because M_1 = 10 ($1 \times 10 < \alpha$). Similarly, so is $\bigcup_{j=1}^{2} R_1^j$ because $M_0 = 7$ ($7 \times 1 < \alpha$ and $7 \times 2 < \alpha$). In general, the maximum count of one relation determines the minimum count of the candidate tuples for the join in the other relation. The tuples with the count less than this minimum can be deleted without checking the join predicate. Let us call this reduction effect as *Low Count Cut* (*LCC*). The LCCs in the above example are called *initial* LCCs. The initial LCCs for R_0 would be possible after $\tilde{n}_0$ is notified of M_1 as a part of the optimization process, which will be described in Section 5.5. Other than initial ones, LCC could occur after a fragment semijoin is executed. It will be explained in the next subsection.

4.3. Highest Count First (HCF)

It would be efficient to take advantage of LCCs in reducing R_0 and R_1 with a sequence of fragment semijoins. When a fragment of one relation is to be selected as the reducer relation for a fragment semijoin, it is desirable to select the fragment with the *highest* count in that relation, for it would result in LCC in the reduced relation. For example, suppose R_0 is fragmented into R_0^3, ⋯, R_0^7 whereas R_1 is fragmented into R_1^4, ⋯, R_1^9, and α = 25. Suppose the fragment semijoin $R_0 \,\overline{\ltimes}_{A=A}\, R_1^9$ is executed, and R_0 is reduced to R''_0. In R_1, R_1^4, ⋯, R_1^8 remain. In R''_0, R''^4_0, ⋯, R''^7_0 remain. Note that R''_0 does not contain R_0^3. It is deleted due to LCC. Note that the LCC due to a fragment semijoin occurs only when the fragment with the highest count is the reducer relation. If $R_0 \,\overline{\ltimes}_{A=A}\, R_1^8$ is executed to reduce R_0 to R''_0 with R_1^9 remaining, R''^3_0 is still contained in R''_0 because some of the tuples in R''^3_0 might be joinable with those in R_1^9.

Let us call the strategy of selecting the fragment with the highest count as the reducer relation for a fragment semijoin as *Highest Count First* (*HCF*). Figure 3 shows how R_0 and R_1 are reduced after a fragment semijoin is executed with the HCF strategy. In Figure 3a, each box with a count value in R_0 and R_1 denotes a fragment. For example, the box at the top of R_0 with count = p denotes the fragment R_0^p. Figure 3a shows how R_0 and R_1 are logically fragmented. As shown, $R_0 = \bigcup_{j=p}^{q} R_0^j$ ($p \leq q$) and $R_1 = \bigcup_{j=r}^{s} R_1^j$ ($r \leq s$). Figure 3b shows the fragment semijoin $R_0^q \,\overline{\rtimes}\, R_1$. Figure 3c shows which

fragments of R_0 and R_1 remain. In R''_0, R''^{q-1}_0 is now the fragment with the highest count. In R''_1, the LCC has occurred, and $n = \lceil \alpha/(q-1) \rceil$ is now the lowest count in the remaining fragments.

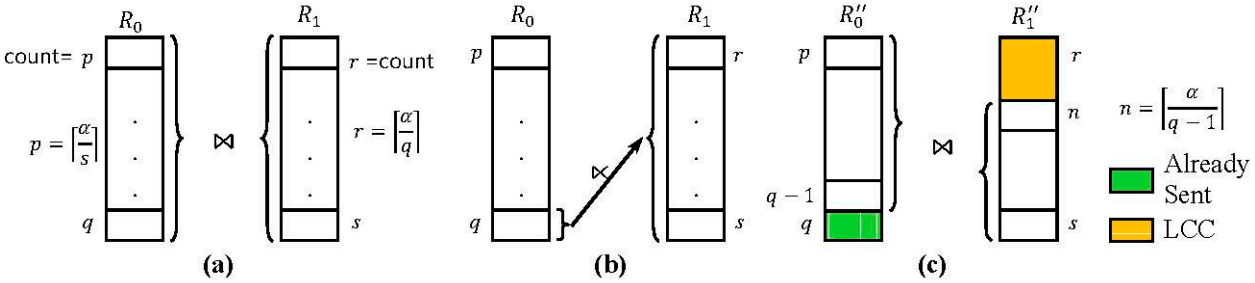


Figure 3. Low Count Cut with Highest Count First Strategy. (**a**) Logical fragmentation of R_0 and R_1, where $R_0 = \cup_{j=p}^{q} R_0^j$ $(p \le q)$ and $R_1 = \cup_{j=r}^{s} R_1^j$ $(r \le s)$; (**b**) fragment semijoin $R_0^q \overline{\ltimes} R_1$; (**c**) Remaining fragments of R_0 and R_1 after the fragment semijoin in (**b**).

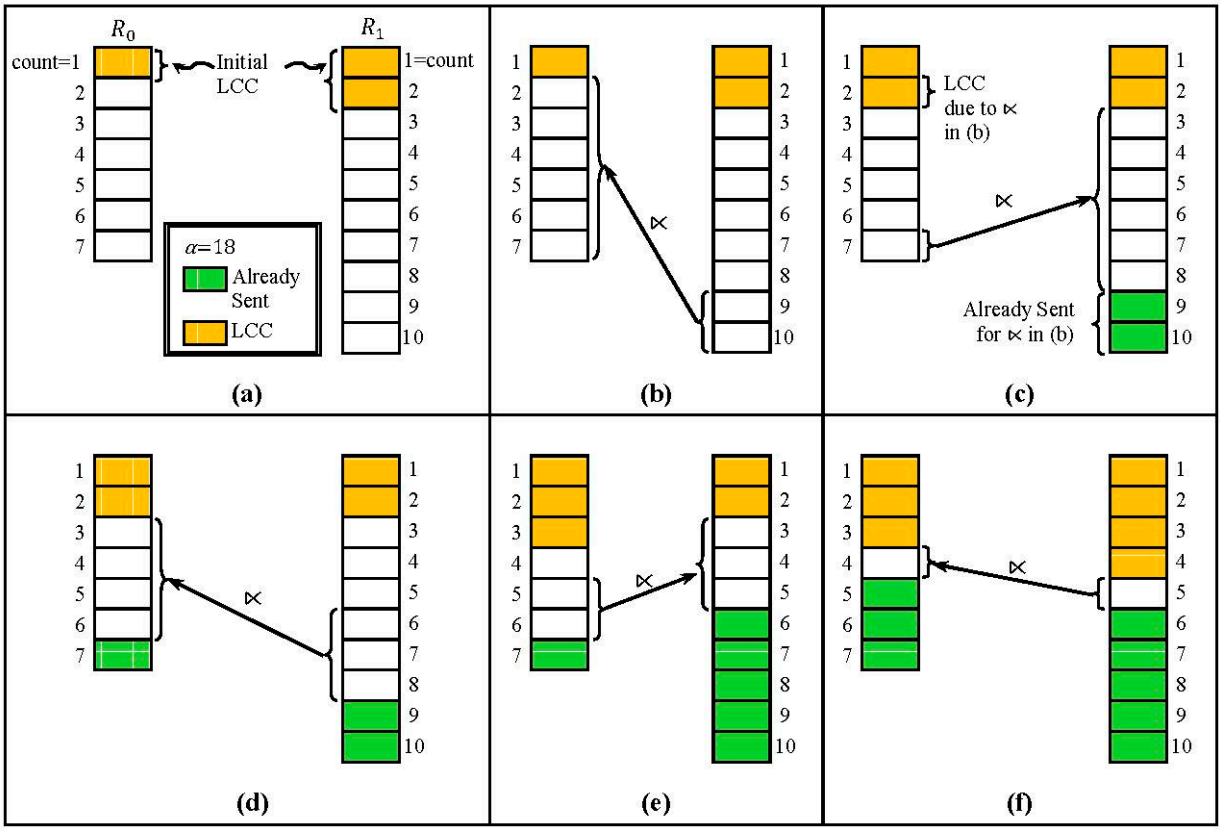


Figure 4. A Sequence of Fragment Semijoins. (**a**) Given $R_0 = \cup_{j=1}^{7} R_0^j$ and $R_1 = \cup_{j=1}^{10} R_1^j$, the initial LCCs remove the fragments R_0^1, R_1^1, and R_1^2; (**b**–**f**) A sequence of fragment semijoins are executed where the fragments of R_0 (or R_1) being the reducer relations, and those of R_1 (or R_0) being the reduced relations. After each fragment semijoin, LCCs occur.

4.4. Fragment Semijoin Sequence

R_0 and R_1 could be fully reduced with a sequence of fragment semijoins which interleaves two types of fragment semijoins: one with a fragment of R_0 being the reducer relation, and the other with a

fragment of R_1 being the reducer relation. Figure 4 shows an example where $R_0 = \bigcup_{j=1}^{7} R_0^j$ and $R_1 = \bigcup_{j=1}^{10} R_1^j$ are reduced by a sequence of fragment semijoins with $\alpha = 18$. After the initial LCCs (Figure 4a), a sequence of fragment semijoins are executed after which LCCs occur (Figure 4b–f). Throughout the sequence, either the fragment with the highest count in R_0 or that in R_1 is the reducer relation. In Figure 4b,d,e, more than one fragment is depicted as being the reducer relation for fragment semijoins. Since every fragment semijoin selects one fragment at a time as the reducer relation with the HCF strategy, the number of fragment semijoins executed in Figure 4b,d,e, is respectively equal to the number of fragments marked as selected. For example, in Figure 4b, two fragment semijoins are sequentially executed; $R_0 \ \bar{\ltimes} \ R_1^{10}$ first, and then $R''_0 \ \bar{\ltimes} \ R_1^9$ where R''_0 is the result of $R_0 \ \bar{\ltimes} \ R_1^{10}$.

5. Optimal Sequence of Fragment Semijoins

In our approach, a sequence of fragment semijoins are executed to fully reduce R_0 and R_1, and the Bloom filter is employed as a synopsis of the joining attribute values. Data transmission for a fragment semijoin occurs to send the Bloom filter and to handle the false positives. According to the Bloom filter theory presented in Section 3.2, the length of a Bloom filter and the number of transformations used could be optimally set to minimize the probability of false positives. In this section, we present an algorithm that generates the optimal sequence of 2-way fragment semijoins with the HCF strategy whereby the total amount of data transmission in fully reducing R_0 and R_1 is minimized.

5.1. Formulation of Optimization Problem

We develop a *dynamic programming algorithm* to generate the optimal sequence of fragment semijoins that fully reduces R_0 and R_1. In describing the algorithm, it is convenient to denote R_0 and R_1 as R_i and R_{1-i} (i = 0, 1). For example, if we need to mention both $R_0 \rtimes R_1$ and $R_1 \rtimes R_0$ to state something that is applied to both (Note that $R_0 \rtimes R_1 \neq R_1 \rtimes R_0$ because the semijoin operation is not commutative.), $R_i \rtimes R_{1-i}$ (i = 0, 1) will do. In the rest of this paper, the subscripts i and $1 - i$ (e.g., R_i, R_{1-i}) are used with (i = 0, 1) omitted if they are related to R_0 or R_1 and the context is clear. Other notations used in this section are summarized in Table 1.

The current state of R_i after a certain sequence of fragment semijoins has been executed can be represented with the two current highest counts in the remaining fragments of R_i and R_{1-i}. Let H_i denote the current highest count of R_i as depicted in Figure 5. The following two statements hold:

- The current lowest count of R_i is $\lceil \alpha / H_{1-i} \rceil$ (Figure 5a).
- If the initial maximum count of R_i is M_i, each fragment in $\bigcup_{j=H_i+1}^{M_i} R_i^j$ has been selected as the reducer relation for the fragment semijoins executed thus far before or after reduced by some fragments of R_{1-i} (Figure 5b).

Let $R_i(H_i, H_{1-i})$ denote R_i reduced thus far, and $R_i^n(H_i, H_{1-i})$ denote the fragment of $R_i(H_i, H_{1-i})$ defined by $\sigma_{count\,=\,n} R_i(H_i, H_{1-i})$. Given $R_i(H_i, H_{1-i})$ and $R_{1-i}(H_{1-i}, H_i)$, let $\bar{\ltimes}_i \ (H_i, H_{1-i})$ denote the fragment semijoin $R_i^{H_i}(H_i, H_{1-i}) \ \bar{\ltimes} \ R_{1-i}(H_{1-i}, H_i)$, and $S_{\bar{\ltimes}*}^i(H_i, H_{1-i})$ denote the optimal sequence of fragment semijoins provided that the first one is $\bar{\ltimes}_i \ (H_i, H_{1-i})$. Let $C_{\bar{\ltimes}*}^i(H_i, H_{1-i})$ denote the cost of $S_{\bar{\ltimes}*}^i(H_i, H_{1-i})$, where the cost is defined to be the total amount of data transmission in bits. $C_{\bar{\ltimes}*}^i(H_i, H_{1-i})$ is given in the following *recurrence relation*:

$$C^{i}_{\overline{\ltimes}*}(H_i, H_{1-i}) = \min_{\left\lceil \frac{\alpha}{H_{1-i}} \right\rceil \le n \le H_i} \left\{ \sum_{j=H_i}^{n} \hat{c}^{i}_{\overline{\ltimes}}(j, H_{1-i}) + C^{1-i}_{\overline{\ltimes}*}(H_{1-i}, n-1) \right\}, i = 0, 1 \quad (2)$$

$\hat{c}^{i}_{\overline{\ltimes}}(j, H_{1-i})$ denotes the cost of $\overline{\ltimes}_i\ (j, H_{1-i})$. Thus, the term $\sum_{j=H_i}^{n} \hat{c}^{i}_{\overline{\ltimes}}(j, H_{1-i})$ is the total cost of executing the following sequence of fragment semijoins: $\overline{\ltimes}_i\ (H_i, H_{1-i}) \cdot \overline{\ltimes}_i\ (H_i - 1, H_{1-i}) \cdot \overline{\ltimes}_i\ (H_i - 2, H_{1-i}) \cdots \overline{\ltimes}_i\ (n, H_{1-i})$. With this sequence, the fragments $R_i^j(H_i, H_{1-i})$ where $j = H_i, H_i - 1, H_i - 2, \cdots, n$ are to be selected as the reducer relations in the descending order of j according to the HCF strategy.

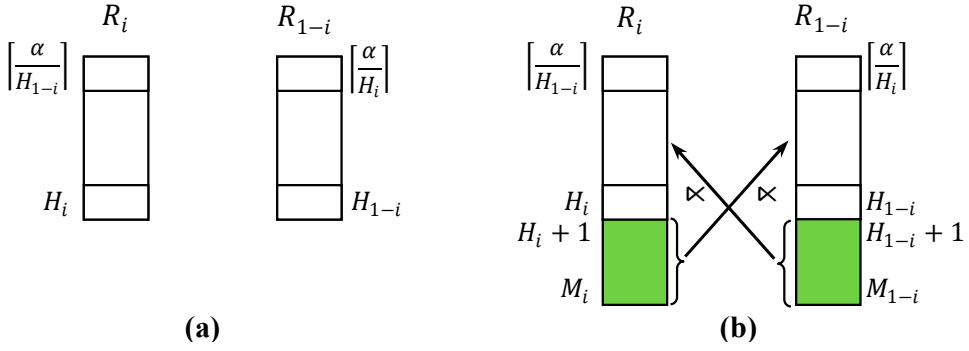


Figure 5. State of $\boldsymbol{R_i}$ and $\boldsymbol{R_{1-i}}$ after a certain sequence of fragment semijoins has been executed. (**a**) The current lowest count of R_i is $\lceil \alpha / H_{1-i} \rceil$, where H_i denotes the current highest count of R_i; (**b**) Let the initial maximum count of R_i be M_i. Then, each fragment in $\bigcup_{j=H_i+1}^{M_i} R_i^j$ has been selected as the reducer relation for the fragment semijoins executed thus far before or after reduced by some fragments of R_{1-i}.

The *termination condition* of this recurrence relation is $\lceil \alpha / H_{1-i} \rceil < H_i$ or $\lceil \alpha / H_i \rceil < H_{1-i}$, which means either one of the two relations gets empty. In such a case, no further fragment semijoin is needed. Thus:

$$C^{i}_{\overline{\ltimes}*}(H_i, H_{1-i}) = 0 \text{ if } \left\lceil \frac{\alpha}{H_{1-i}} \right\rceil < H_i \text{ or } \left\lceil \frac{\alpha}{H_i} \right\rceil < H_{1-i} \quad (3)$$

Let $n^{i}_{\overline{\ltimes}*}(H_i, H_{1-i})$ denote the value n for which $C^{i}_{\overline{\ltimes}*}(H_i, H_{1-i})$ in (Equation (2)) is minimized. That is,

$$n^{i}_{\ltimes*}(H_i, H_{1-i}) = \min\text{arg}_{\left\lceil \frac{\alpha}{H_{1-i}} \right\rceil \le n \le H_i} \left\{ \sum_{j=H_i}^{n} \hat{c}^{i}_{\overline{\ltimes}}(j, H_{1-i}) + C^{1-i}_{\overline{\ltimes}*}(H_{1-i}, n-1) \right\}, i = 0, 1 \quad (4)$$

Let M_i be the maximum count in R_i. Then, the optimal sequence of fragment semijoins for R_0 and R_1 is either $S^{0}_{\overline{\ltimes}*}(M_0, M_1)$ or $S^{1}_{\overline{\ltimes}*}(M_1, M_0)$. Its cost is $min\{C^{0}_{\overline{\ltimes}*}(M_0, M_1), C^{1}_{\overline{\ltimes}*}(M_1, M_0)\}$ or $\min_{i \in \{0,1\}} \{C^{i}_{\overline{\ltimes}*}(M_i, M_{1-i})\}$. Let $j = \min\text{arg}_{i \in \{0,1\}} \{C^{i}_{\overline{\ltimes}*}(M_i, M_{1-i})\}$. The value of j, which is 0 or 1, indicates that the first fragment semijoin in the optimal sequence is $\overline{\ltimes}_j\ (M_j, M_{1-j})$ where $R_j^{M_j}$ is the reducer relation. Thus, the optimal sequence can be represented as i ($i \in \{0,1\}$) followed by a sequence of counts in R_i and R_{1-i} as follows:

$$i;\ n_i^1, n_{1-i}^1, n_i^2, n_{1-i}^2, \cdots, n_i^k, n_{1-i}^k, \cdots,$$

where

$$n_i^1 = n_{\ltimes *}^{i}(M_i,\ M_{1-i})$$
$$n_{1-i}^1 = n_{\ltimes *}^{1-i}(M_{1-i},\ n_i^1 - 1)$$
$$n_i^2 = n_{\ltimes *}^{i}(n_i^1 - 1,\ n_{1-i}^1 - 1)$$
$$n_{1-i}^2 = n_{\ltimes *}^{1-i}(n_{1-i}^1 - 1,\ n_i^2 - 1)$$
$$\ldots\ldots$$

For example, if the sequence of fragment semijoins in Figure 4 is the optimal one, then it is represented as 1, 9, 7, 6, 5, 5.

5.2. Cost of 2-Way Fragment Semijoin

The cost of a fragment semijoin $\hat{c}_{\ltimes}^{i}(x,y)$ where i = 0, 1 is defined as the total amount of data transmission in bits, and given as follows:

$$\hat{c}_{\ltimes}^{i}(x,y) = \|BF_i(x,y)\| + C_{fp}^{i}(x,y) \tag{5}$$

The first term denotes the size of the Bloom filter, and the second is for the cost of handling false positives. Since the total cost of sending all the joinable attribute values (excluding those from false positives) in the backward reduction phases is the same for all the possible sequences of fragment semijoins, it is omitted. The multiplication of the number of hops between the two regions of R_0 and R_1 is also omitted because it is the same for all the sequences.

5.2.1. Size of Bloom Filter

$\|BF_i(x,y)\|$ denotes the size of the Bloom filter $BF_i(x,y)$, which represents the set of joining attribute values in $R_i^x(x,y)$ to be sent in executing $\overline{\ltimes}_i\ (x,y)$. When this fragment semijoin is to be executed, the reduced state of R_i and R_{1-i} are respectively $R_i(x,y)$ and $R_{1-i}(y,x)$. That is, $H_i = x$, $H_{1-i} = y$, and $BF_i(x,y)$ is to represent the set of joining attribute values in $R_i^x(x,y)$. The joining attribute values in R_i are unique among all the tuples. Thus, the number of distinct values in the joining attribute A of $R_i^x(x,y)$ is $|R_i^x(x,y)|$. This number can be estimated as follows using the semijoin selectivity estimation described in Section 3.1:

$$|R_i^x(x,y)| = |R_i^x| \cdot \left(1 - \frac{\left|\bigcup_{j=y+1}^{M_{1-i}} R_{1-i}^{j}\right|}{|D_A|}\right) \tag{6}$$

R_i and R_{1-i} have been reduced by each other with a fragment semijoin sequence until (H_i, H_{1-i}) becomes (x, y). However, the above estimation is valid since the joining attribute values are unique in R_i, that is, $R_i^p \cap R_i^q = \emptyset$ $(p \neq q)$, and thus, the following holds: $R_i^p \ltimes R_{1-i} = R_i^p \ltimes (R_{1-i} \ltimes R_i^q)$ where $p \neq q$.

Now $\|BF_i(x,y)\|$ can be estimated as follows according to the Bloom filter theory described in Section 3.2:

$$\|BF_i(x,y)\| = \frac{|R_i^x(x,y)| \cdot k_i(x,y)}{ln\,2} \tag{7}$$

where $k_i(x,y)$ denotes the *optimal* number of hash functions for $BF_i(x,y)$, which will be explained shortly. In some cases, the number of tuples of $R_i^x(x,y)$ is so small that the size of the list of joining attribute values might be smaller than $\|BF_i(x,y)\|$. In such a case, the list is sent instead of $BF_i(x,y)$. To cover such exceptions, we modify the above equation as follows:

$$\|BF_i(x,y)\| = min\left\{\frac{|R_i^x(x,y)|\cdot k_i(x,y)}{ln\,2}, |R_i^x(x,y)|\cdot\|A\|\right\}+1 \quad (8)$$

where $\|A\|$ denotes the number of bits to represent a value in the joining attribute *A*, and the one added is for a flag bit (for distinguishing a Bloom filter from a value list).

5.2.2. Cost of Handling False Positives

The cost of handling the false positives is estimated as follows: Let *D*, *E*, and *F* respectively denote the number of joining attribute values in R_{1-i} for which the membership tests with $BF_i(x,y)$ are to be done, the number of non-joinable tuples among those *D* values, and the number of false positives out of those *E* values. Then:

$$D = \left|\bigcup_{j=\lceil\frac{\alpha}{x}\rceil}^{y} R_{1-i}^j(y,x)\right|,\ E = D\cdot\left(1-\frac{|R_i^x(x,y)|}{|D_A|}\right),\ \text{and}\ F = E\cdot\frac{1}{2^{k_i(x,y)}} \quad (9)$$

The estimation of *E* is according to the semijoin selectivity estimation as described in Section 3.1. In the equation of *F*, $1/2^{k_i(x,y)}$ is the probability of a false positive as summarized in Section 3.2. The values of the false positives are sent back to R_i in the backward reduction phase, and then sent back to R_{1-i} after the false positives are confirmed. Thus, the cost of handling the false positives for a fragment semijoin is $2\cdot F\cdot\|A\|$. That is:

$$C_{fp}^i(x,y) = \frac{1}{2^{k_i(x,y)-1}}\cdot\left|\bigcup_{j=\lceil\frac{\alpha}{x}\rceil}^{y} R_{1-i}^j(y,x)\right|\cdot\left(1-\frac{|R_i^x(x,y)|}{|D_A|}\right)\cdot\|A\| \quad (10)$$

In case that the list of joining attribute values is sent instead of $BF_i(x,y)$, $C_{fp}^i(x,y)=0$.

5.2.3. Optimal Number of Hash Functions.

$k_i(x,y)$, the optimal number of hash functions for $BF_i(x,y)$, is determined as follows: as summarized in Section 3.2, the probability of false positives is affected by the number of hash functions. As the number of hash functions increases, the length of the Bloom filter increases with the probability of false positives decreased. In (Equation (5)), the cost of a fragment semijoin, $\hat{c}_{\ltimes}^i(x,y)$, is given as a sum of two terms, $\|BF_i(x,y)\|$ and $C_{fp}^i(x,y)$. There exists a tradeoff between the two terms, and thus, $k_i(x,y)$ should be determined to be a positive integer such that $\hat{c}_{\ltimes}^i(x,y)$ is minimized. From Equations (8) and (10), the cost of a fragment semijoin $\overline{\ltimes}_i(x,y)$ with *k* hash functions is given as follows:

$$\frac{|R_i^x(x,y)|\cdot k}{ln\,2} + \frac{1}{2^{k-1}}\cdot\left|\bigcup_{j=\lceil\frac{\alpha}{x}\rceil}^{y} R_{1-i}^j(y,x)\right|\cdot\left(1-\frac{|R_i^x(x,y)|}{|D_A|}\right)\cdot\|A\|+1 \quad (11)$$

This equation can be given as a function of *k*: $f(k) = a\cdot k + b/2^k + c$, where *a*, *b*, and *c* are constants. The differentiation of $f(k)$ with respect to *k* reveals that $k_i(x,y) = \lceil log_2((b/a)\cdot ln\,2)\rceil$,

where $a = |R_i^x(x,y)|/ln\,2$, $b = 2 \cdot \left|\cup_{j=\lceil \alpha/x \rceil}^{y} R_{1-i}^{j}(y,x)\right| \cdot \left(1 - (|R_i^x(x,y)|/|D_A|)\right) \cdot \|A\|$. If this value is less than 1, $k_i(x,y)$ is set to 1.

5.3. Dynamic Programming Algorithm

$C_{\ltimes *}^{i}(x,y)$ and $n_{\ltimes *}^{i}(x,y)$ can be obtained with a dynamic programming algorithm. From (Equation (2)) through (Equation (4)), $C_{\ltimes *}^{i}(x,y)$ and $n_{\ltimes *}^{i}(x,y)$ are initialized as follows: If there is no fragment remaining in either one of R_0 and R_1, that is, if $x < \lceil \alpha/y \rceil$ or $y < \lceil \alpha/x \rceil$, then $C_{\ltimes *}^{i}(x,y)$ is set to 0, and $n_{\ltimes *}^{i}(x,y)$ is undefined. If none of R_0 and R_1 is empty but there remains only one fragment in R_i (*i.e.*, $x = \lceil \alpha/y \rceil$), the only possible fragment semijoin is the one where that fragment is the reducer relation. Thus, $C_{\ltimes *}^{i}(x,y) = \hat{c}_{\ltimes}^{i}(x,y)$ and $n_{\ltimes *}^{i}(x,y) = x$. Starting from these initializations, the optimal solution for the cases where the number of remaining fragments of R_0 and R_1 is greater than 1 can be obtained. For example, suppose there are two fragments remaining in R_0 and one in R_1 (Figure 6a). $S_{\ltimes *}^{0}(H_0, H_1)$ can be obtained as follows: Because of the HCF strategy, the fragment with the highest count is the reducer relation in the first semijoin (Figure 6b). The remaining fragment of R_0 might be the reducer relation of the next semijoin (Figure 6c). That is, Figure 6b,c show all the possible cases. Figure 6d,e respectively show the subsequent fragment semijoin where the fragment of R_1 is the reducer relation after the semijoins in Figure 6b,c assuming that no LCC has occurred. The optimal solutions (*i.e.*, $S_{\ltimes *}^{1}(H_1, H_0)$) in Figure 6d,e are already known, because the number of remaining fragment in R_0 is 1 (Figure 6d) or 0 (Figure 6e), and that in R_1 is 1. The optimal solutions here are already given from the initialization. In this way, $S_{\ltimes *}^{i}(H_i, H_{1-i})$ can be obtained when there are 3, 4,…, $M_i - \lceil \alpha/M_{1-i} \rceil + 1$ fragments remaining in R_i (i = 0, 1). Figure 7 describes the algorithm that carries out this process.

In the algorithm, procedure *opt*(i, L_0, H_0, L_1, H_1) is invoked. As shown in Figure 8a, the arguments L_0 and H_0 are respectively the current lowest and highest count in R_0. L_1 and H_1 are those in R_1. Procedure *opt*(i, L_0, H_0, L_1, H_1) vacuously returns unless $L_0 = \lceil \alpha/H_1 \rceil$ and $L_1 = \lceil \alpha/H_0 \rceil$. If the arguments are valid ones, it finds $S_{\ltimes *}^{i}(H_i, H_{1-i})$. When i = 0, opt(0, L_0, H_0, L_1, H_1) finds out a value n in the range $[L_0, H_0]$ that minimizes $\sum_{j=H_0}^{n} \hat{c}_{\ltimes}^{0}(j, H_1) + C_{\ltimes *}^{1}(H_1, n-1)$. (Figure 8a corresponds to the first term, and Figure 8b to the second one.) In doing so, $C_{\ltimes *}^{0}(H_0, H_1)$ and $n_{\ltimes *}^{0}(H_0, H_1)$ are obtained. Similarly, when i = 1, opt(1, L_0, H_0, L_1, H_1) finds out a value n in the range $[L_1, H_1]$ that minimizes $\sum_{j=H_1}^{n} \hat{c}_{\ltimes}^{1}(j, H_0) + C_{\ltimes *}^{0}(H_0, n-1)$. $C_{\ltimes *}^{1}(H_1, H_0)$ and $n_{\ltimes *}^{1}(H_1, H_0)$ are obtained.

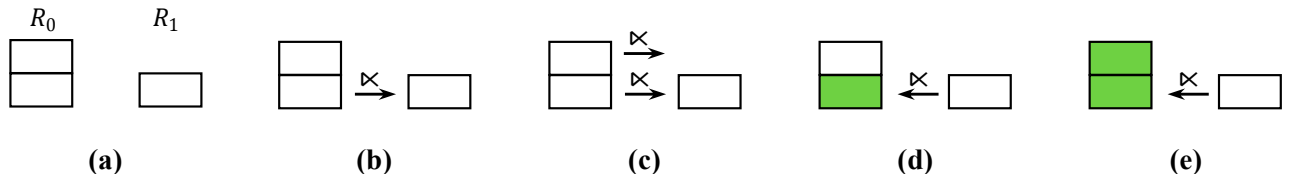


Figure 6. Optimal Solution from Initialization. (**a**) Two fragments remaining in R_0 and one in R_1. There are only two possible cases for $S_{\ltimes *}^{0}(H_0, H_1)$, (**b**) One case, (**c**) The other case, (**d**) The optimal solution (*i.e.*, $S_{\ltimes *}^{1}(H_1, H_0)$) for the case of (b) is given from the initialization, (**e**) The same for the case of (**c**).

Algorithm: Optimal Sequence of Fragment Semijoins

```
1: procedure opt_fsj_seq()
2: begin
3:   max_num_frag = bigger(M0 − ⌈α/M1⌉ +1, M1 − ⌈α/M0⌉ +1);
4:   for num_frag = 2 to max_num_frag do
5:     for opp_num_frag=1 to num_frag do
6:       for H0 = M0 to ⌈α/M1⌉ by -1 do
7:         for H1 = M1 to ⌈α/M0⌉ by -1 do
8:             opt(0, H0 − num_frag + 1, H0, H1 − opp_num_frag + 1, H1);
9:             opt(1, H0 − opp_num_frag + 1, H0, H1 − num_frag + 1, H1);
10:            if( 1 < opp_num_frag < num_frag) then
11:                opt(0, H0 − opp_num_frag + 1, H0, H1 − num_frag + 1, H1);
12:                opt(1, H0 − num_frag + 1, H0, H1 − opp_num_frag + 1, H1);
13:            endif
14:         endfor
15:       endfor
16:     endfor
17:   endfor
18: end procedure
```

Figure 7. Algorithm for Generating Optimal Sequence of Fragment Semijoins.

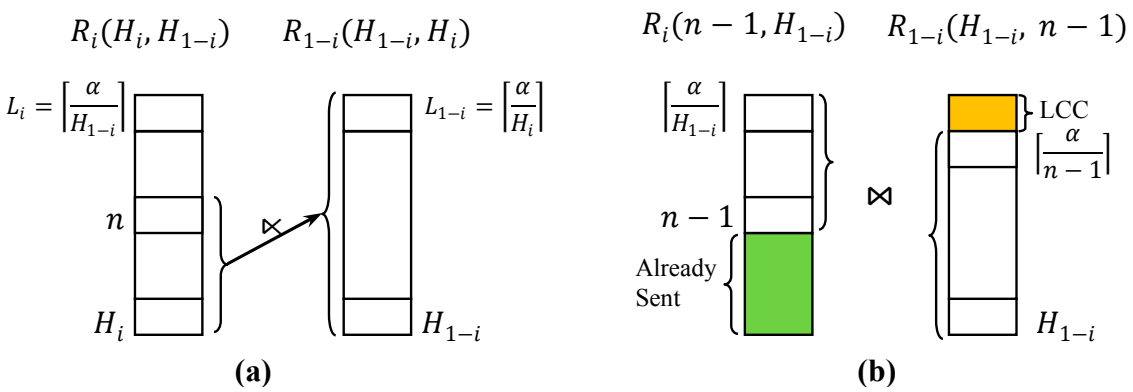


Figure 8. Optimal Solution Obtained with Procedure *opt*(i, L_0, H_0, L_1, H_1), where L_i and H_i are respectively the current lowest and highest count in R_i. (**a**) The optimal value of n for $S^{i}_{\ltimes *}(H_i, H_{1-i})$ is obtained; (**b**) using the optimal solution, $S^{1-i}_{\ltimes *}(H_{1-i}, n-1)$, for each value of n.

The variable *num_frag* denotes the number of remaining fragments in R_i, and *opp_num_frag* denotes that in R_{1-i}. In the two outermost loops, the values of these variables increase. In the two innermost loops, pairings of H_0 and H_1 are provided such that the difference between H_i and L_i is set by *num_frag* and *opp_num_frag*, i = 0, 1. The invocations of *opt*() in Line 8 and 9 find the optimal solution for the case where the number of fragments in the reducer relation is greater than that in the reduced relation. Those in Line 11 and 12 find the optimal solution for the inverse case.

For example, suppose four and three fragments remain in R_0 and R_1, respectively (Figure 9a). Let us consider $S^{0}_{\ltimes *}(H_0, H_1)$ for them. Figure 9b–e show that there could be four cases possible with the HCF strategy. After one, ⋯, four fragments of R_0 are used as the reducer relations in succession, the remaining fragments are shown at the bottom of Figure 9b–e, respectively. To find out which case of the four would

lead to $S^0_{\ltimes *}(H_0, H_1)$, $S^1_{\ltimes *}(H_1, H_0)$ for the fragments in each of the bottom of Figure 9b–e should be already known (see (Equation (2))). In other words, before $S^0_{\ltimes *}(H_0, H_1)$ for the fragments in Figure 9a is obtained, $S^1_{\ltimes *}(H_1, H_0)$ for the fragments in each of the bottom of Figure 9b–e should have been obtained. The algorithm in Figure 7 guarantees this by calling procedure *opt*() in proper order through the nested loops.

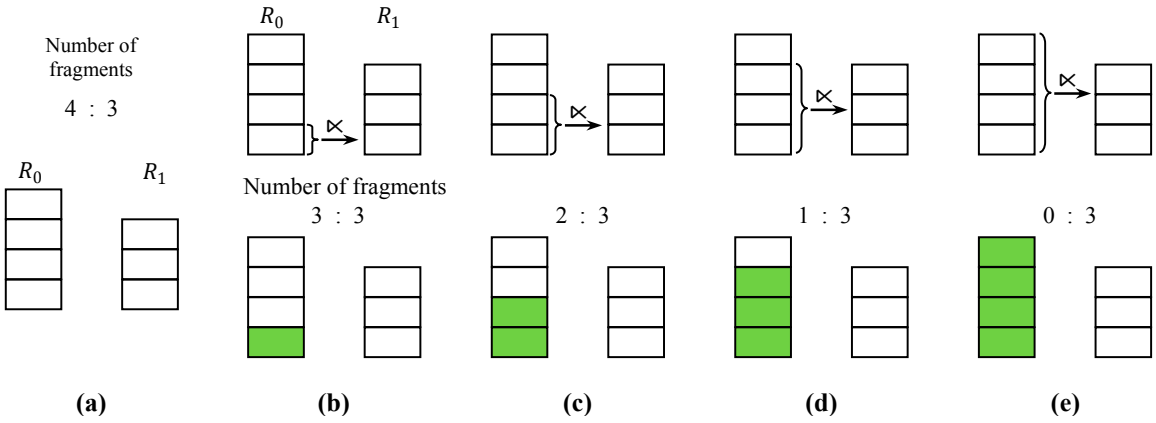


Figure 9. An Example of Optimization with Procedure *opt*(). (**a**) 4 and 3 fragments remaining in R_0 and R_1, respectively, (**b**–**e**) For the remaining fragments in (**a**), 4 cases that could result in the optimal solution, $S^0_{\ltimes *}(H_0, H_1)$, are shown at the top. Determining which case leads to the optimal solution is possible, because the optimal solution, $S^1_{\ltimes *}(H_1, H_0)$, for the remaining fragments in each of the bottom of (**b**–**e**) is already known.

The complexity of the algorithm in Figure 7 is $O\left(f_0 \cdot f_1 \cdot \left(\frac{f(f+1)}{2} - 1\right)\right)$ where $f_i = M_i - \lceil \alpha / M_{1-i} \rceil + 1$, which is the number of fragments in R_i remaining after initial LCCs (i = 0, 1), and $f = max(f_0, f_1)$. The innermost two for loops (line 6 and 7 in Figure 7) combined are repeated $f_0 \cdot f_1$ times. The second outermost for loop (line 5) is repeated i times when num_frag = i at the outermost for loop. Since num_frag at the outermost for loop varies from 2 to f, the computation in the nested for loops (lines 8–13) would be repeated $f_0 \cdot f_1 \cdot (2 + \cdots + f)$ times. The complexity would be $O(f^4)$ when $f_0 = f_1 = f$. Due to the characteristics of iceberg queries which are supposed to produce very selective result, M_i's and α would grow or shrink together. Otherwise, the query would not be so selective. It means that the number of fragments remaining after initial LCCs would not be high. When $M_0 = M_1 = 15$, and $\alpha = 150$, for example, $f_0 = f_1 = f = 6$. When $M_0 = M_1 = 15$, and $\alpha = 200$, $f_0 = f_1 = f = 2$.

5.4. Post Optimization

So far, we have assumed that the backward reduction phase is carried out for each fragment semijoin in the optimal sequence. As described in Section 4.1, the values for which false positives are confirmed need to be sent back again and their corresponding tuples need to be inserted back to the reduced relation. In the backward reduction, if the matched joining attribute values are sent back with their counts, re-insertion would not be necessary at all. In doing so, the concern is the overhead of sending the count for each individual matched value. An efficient way of handling this problem is to delay the backward reduction of each fragment semijoin until the execution of all the fragment semijoins in the optimal

sequence is completed, and to conduct the backward reductions all at once. For this, R_i sends R_{1-i} a message of the form: $n_j, n_{j+1}, \ldots, n_{M_i}, V_j, V_{j+1}, \ldots, V_{M_i}$ where $j = \lceil \alpha / M_{1-i} \rceil$, V_j is the set of matched joining attribute values (possibly including those from false positives) whose count is equal to j, and $n_k = |V_k|$ ($k = j,\ j+1, \cdots, M_i$).

5.5. Query Optimization and Processing

In-network optimization of an iceberg join query Q is carried out in the sensor node $\tilde{m}$ located at the midpoint between $\tilde{n}_0$ and $\tilde{n}_1$. Thus, it is required for $\tilde{n}_0$ and $\tilde{n}_1$ to send the information necessary for the optimization to $\tilde{m}$. First, $\tilde{n}_i$ sends the count information of R_i (*i.e.*, $\left|R_i^1\right|, \cdots, \left|R_i^{M_i}\right|$) to $\tilde{m}$. $\|A\|$ and $|D_A|$ are assumed to have already been sent to $\tilde{m}$ when Q was initially forwarded to $\tilde{n}_0$, $\tilde{n}_1$, and $\tilde{m}$. All the equations in Sections 5 and 3.2 can be evaluated if the aforementioned information is available. The node $\tilde{m}$ generates the optimal sequence of fragment semijoins and sends it to $\tilde{n}_i$ with the count information of R_{1-i} (i.e., $\left|R_{1-i}^1\right|, \cdots, \left|R_{1-i}^{M_{1-i}}\right|$). The count information of R_i as well as the optimal sequence as described in Section 5.1 are represented as a short sequence of integers. Thus, the communication overhead for optimization could be very small. Now $\tilde{n}_0$ and $\tilde{n}_1$ are ready to execute the optimal sequence to fully reduce R_0 and R_1.The skeleton of the protocol executed by $\tilde{n}_0$ for our scheme is described in Figure 10. The one for $\tilde{n}_1$ is symmetrical.

```
1: begin
2:     send the count information of R_0 to m̃;
3:     receive the optimal sequence of fragment semijoins OPT and
                the count information of R_1 from m̃;
4:     process initial low count cut for R_0 with M_1;
5:     if( ñ_0 is to start forward reduction in OPT ) then S-flag = TRUE;
6:                                                    else S-flag = FALSE;
7:     endif
8:     // forward reduction phase
9:     if( S-flag ) make and send Bloom filters to ñ_1 according to OPT;
10:    loop
11:        if( no fragment semijoin to execute is remaining in OPT ) break;
12:        receive Bloom filters from ñ_1 and process forward reductions;
13:        if( no fragment semijoin to execute is remaining in OPT ) break;
14:        make and send Bloom filters to ñ_1 according to OPT;
15:    endloop
16:    // backward reduction phase
17:    send the matched values of R_0 with their count information to ñ_1;
18:    receive the values sent due to false positives from ñ_1;
19:    receive the matched values of R_1 with their count information from ñ_1;
20:    send the values received due to false positives to ñ_1;
21: end
```

Figure 10. Protocol Executed by $\tilde{n}_0$ for Our Scheme.

6. Performance Evaluation of Our Scheme and SRJA

In this section, we compare the performance of our scheme and that of SRJA. We have implemented both of our scheme and SRJA, measuring the total number of packets transmitted among the sensor

nodes and the total number of transmissions among the sensor nodes while the two schemes are executed. These two performance metrics are employed to compare the energy-efficiency of the two schemes. The number of packets transmitted is measured assuming that the network is IEEE 802.15.4-compliant. We also measured the ratio of joinable values transmitted over all the values transmitted. This metric is to compare the effectiveness of data filtering as well as the energy-efficiency of the two schemes. Finally, we present an analytical comparison of the total number of transmissions in the two schemes. All the procedures for the experiments were implemented in C and the experiments were conducted in a system of Windows 7 with an AMD Phenom II X4 945 Processor (3.0 GHz) and 4 GB memory.

6.1. Parameters

The network and query parameters in the experiments are summarized in Table 2. We have considered WSNs where sensor nodes are uniformly deployed. Each of the two regions for an inter-region iceberg join where a join operand relation resides is assumed to be a square consisting of $n \times n$ nodes. The distance between the coordinator nodes $\tilde{n}_0$ and $\tilde{n}_1$ of the two regions is set to 30 hops.

Table 2. Parameter Settings in the Experiments.

Parameter	Value
Size of a region ($n \times n$ nodes)	n = 5, 10, 15
The distance between $\tilde{n}_0$ and $\tilde{n}_1$	30 hops
The range of joining attribute values	1..10,000
The range of count	1..15
Epoch (*i.e.*, sampling interval)	10, 20, 30, 40, 50, 60 s (default: 30 s)
Query window size	3 h
Iceberg threshold (α)	50, 100, 150, 200

The default epoch (*i.e.*, sampling interval) of every sensor node is set to 30 s, and the size of the sliding window of the join query is set to 3 h as in the experiments with SRJA in [5]. After setting the join selectivity between R_0 and R_1, the values of the joining attribute in the iceberg join operand relations are randomly generated as an integer in the range [1, 10,000] with a random distribution of their counts under the constraint that the highest count for a value could be 15. The iceberg thresholds considered are 50, 100, 150 and 200.

6.2. Experimental Results

Each of the reported measurements in this subsection is the average out of 50 runs against different data values. The experimental results reveal that our scheme considerably outperforms SRJA. Figure 11a,b respectively compare the total number of packets transmitted and the total number of transmissions of the two schemes as the size of each region varies from 5 × 5 nodes to 15 × 15 nodes while α is set to 150. Figure 11c,d compare the same while α is set to 200. As the size of each region gets bigger, more data is collected at the sensor nodes and more tuples need to be processed in both schemes. Thus, more packets are transmitted. As for the number of transmissions, it also increases in SRJA. In our scheme, it is not so sensitive to the data volume because for a given α, the optimal sequence

of fragment semijoins generated could include the similar number of semijoins. For the total number of packets transmitted, the average performance improvement with our scheme over SRJA is 63.58% ($\alpha = 150$) and 71.97% ($\alpha = 200$). For the total number of transmissions, the average performance improvement with our scheme over SRJA is 59.66% ($\alpha = 150$) and 57.27% ($\alpha = 200$).

Figure 12a,b respectively compare the total number of packets transmitted and the total number of transmissions of the two schemes as α varies from 50 to 200 while the size of each region is set to 10 × 10 nodes. Figure 12c,d compare the same while the size of each region is set to 15 × 15 nodes. As α increases, the iceberg join query gets more selective. Thus, in both schemes, more tuples are filtered and the number of packets transmitted decreases. The number of transmissions turns out not so sensitive to the increase of α except for the case of $\alpha = 200$. In SRJA, the effect of subrange pruning results in the decrease of transmissions for larger α's. In our scheme, on the other hand, the number of transmissions slightly increases from $\alpha = 50$ to 150, then decreases for $\alpha = 200$. These changes depend on the optimal sequence of fragment semijoins generated. The more semijoins are to be executed in the optimal sequence, the more transmissions would occur. For the total number of packets transmitted, the average performance improvement with our scheme over SRJA is 59.71% (regions of 10 × 10 nodes) and 64.07% (regions of 15 × 15 nodes). For the total number of transmissions, the average performance improvement with our scheme over SRJA is 63.98% (regions of 10 × 10 nodes) and 66.13% (regions of 15 × 15 nodes).

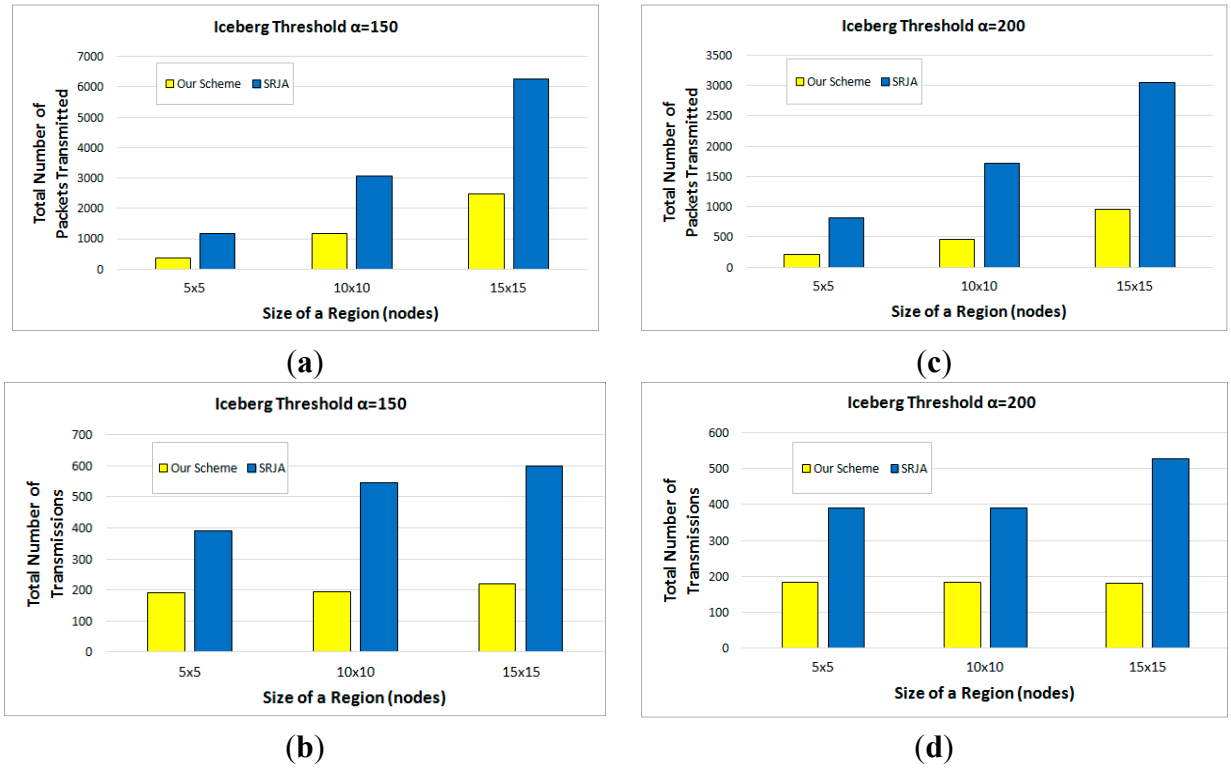


Figure 11. Performance Comparison with respect to Varying Sizes of a Region. (**a**) Total number of packets transmitted ($\alpha = 150$); (**b**) Total number of transmissions ($\alpha = 150$); (**c**) Total number of packets transmitted ($\alpha = 200$); (**d**) Total number of transmissions ($\alpha = 200$).

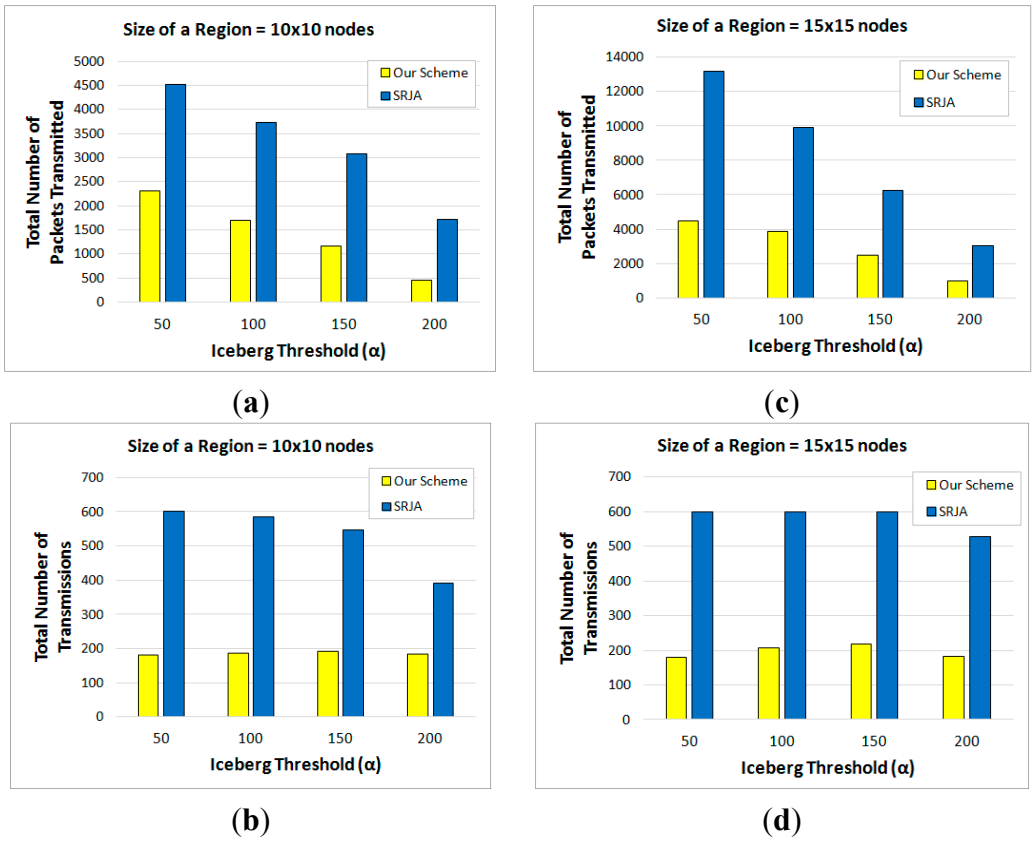


Figure 12. Performance Comparison with respect to Varying Iceberg Thresholds. (**a**) Total number of packets transmitted (10 × 10 nodes); (**b**) Total number of transmissions (10 × 10 nodes); (**c**) Total number of packets transmitted (15 × 15 nodes); (**d**) Total number of transmissions (15 × 15 nodes).

Figure 13a,b respectively compare the total number of packets transmitted and the total number of transmissions of the two schemes as the epoch (*i.e.*, sampling interval) at each node varies from 10 s to 60 s while the size of each region is set to 10 × 10 nodes and α is set to 150. As the sampling rate gets higher, more data is collected at the sensor nodes and both schemes are supposed to transmit more packets. The number of transmissions also increases. However, SRJA turns out to suffer much more with higher sampling rates. For the total number of packets transmitted, the average performance improvement with our scheme over SRJA is 62.97%. For the total number of transmissions, the average performance improvement with our scheme over SRJA is 58.66%.

Figure 14a compares the ratio of joinable values transmitted in the two schemes as the size of each region varies from 5 × 5 nodes to 15 × 15 nodes while α is set to 150. This ratio is defined as:

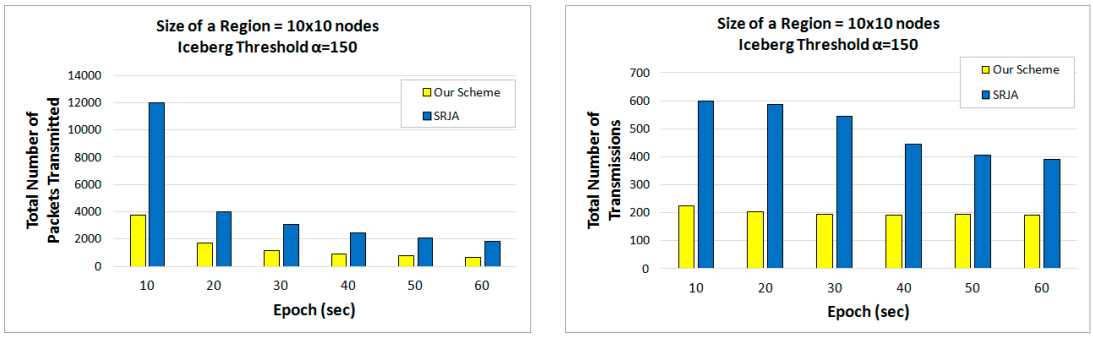
$$\frac{\text{Total number of joinable attribute values}}{\text{Total number of join attribute values transmitted}} \quad (12)$$

Figure 14b compares the same as α varies from 50 to 200 while the size of each region is set to 10 × 10 nodes. This ratio gets lower as the query gets more selective or as more non-joinable values are transmitted. As the size of each region gets bigger, the number of tuples collected at the sensor nodes increases and the size of the iceberg join result also increases with a given α. Thus, this ratio increases in both schemes (Figure 14a). As α increases, on the other hand, this ratio decreases in both schemes

because the iceberg join query gets more selective (Figure 14b). In Figure 14, this ratio in our scheme turns out to be significantly higher than that in SRJA. This means that the effectiveness of filtering out non-joinable values and the energy-efficiency in our scheme is much higher than that in SRJA.
The major reasons for the improvements are two-fold:

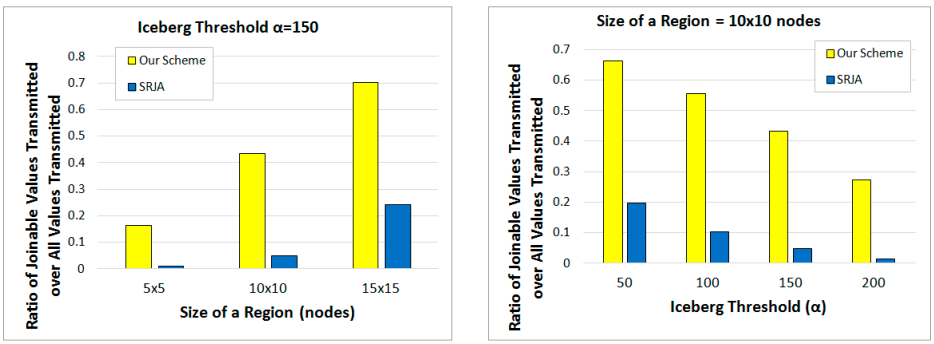
- In SRJA, the histogram-based value ranges are sent as a synopsis of the joining attribute values. In our scheme, a Bloom filter constructed from a count-based fragment is sent as a synopsis of the joining attribute values. The Bloom filter is more compact than the value ranges. Besides, the false positives are efficiently handled in the backward reduction phase of the 2-way fragment semijoins in our scheme.

SRJA is centered around checking the join predicate with the cardinality constraint as an additional condition. For each pair of matched ranges, if there exists at least one pair of tuples $t_0 \in R_0$ and $t_1 \in R_1$ such that t_0.count $\times$ t_1.count $\geq \alpha$, the non-joinable tuples in either range cannot be filtered out, and recursive divisions of each of the two ranges into subranges are required. In contrast, our scheme is centered around the cardinality constraint with the join predicate as the secondary condition. Only the fragment semijoins between the fragments satisfying the cardinality constraint are carried out to filter non-joinable tuples. Besides, the reductions from the LCCs that result with the HCF strategy are effective.



(**a**) (**b**)

Figure 13. Performance Comparison with respect to Varying Epochs. (10 × 10 nodes, $\alpha = 150$). (**a**) Total number of packets transmitted; (**b**) Total number of transmissions.



(**a**) (**b**)

Figure 14. Comparison of the Ratio of Joinable Values Transmitted with respect to (**a**) varying sizes of a region ($\alpha = 150$); (**b**) varying iceberg thresholds (10 × 10 nodes).

6.3. Analytical Comparison of the Total Number of Transmissions

In this subsection, we present an analytical comparison of the total number of transmissions among the sensor nodes throughout the execution of SRJA and our scheme. In Figures 2 and 10, the protocols executed by the coordinator node of a region for SRJA and our scheme are described. In each of the two schemes, let $X_{i,1-i}$ be the number of messages that $\tilde{n}_i$ sends to $\tilde{n}_{1-i}$ plus the number of messages that $\tilde{n}_i$ receives from $\tilde{n}_{1-i}$. Let Y_i be the number of messages that $\tilde{n}_i$ sends to $\tilde{m}$ plus the number of messages that $\tilde{n}_i$ receives from $\tilde{m}$. Let $M_x = X_{01} + X_{10}$ and $M_y = Y_0 + Y_1$. Then, the total number of transmissions in each scheme is $d \cdot M_x + \frac{1}{2} d \cdot M_y$ where d is the distance between $\tilde{n}_0$ and $\tilde{n}_1$ in hops.

6.3.1. SRJA

Considering the protocol in Figure 2, we have:

$$X_{i,1-i} = H_{i,1-i} + S_{i,1-i} + \sum_{k=1}^{N_r} (f_k^J \cdot J_{i,1-i} + f_k^D \cdot S_{i,1-i}) \quad (13)$$

$H_{i,1-i}$ denotes the number of messages $\tilde{n}_i$ sends to $\tilde{n}_{1-i}$ plus the number of messages $\tilde{n}_i$ receives from $\tilde{n}_{1-i}$ in processing the values with a high count (line 2 in Figure 2), $S_{i,1-i}$ denotes that in processing sparse subranges in the synopsis (line 4 or line 16), and $J_{i,1-i}$ denotes that in processing JOIN-tagged subranges (line 13). f_k^J is 1 if the tagged synopsis received from $\tilde{m}$ includes at least one JOIN-tagged subrange. It is 0 otherwise. f_k^D is 1 if the tagged synopsis includes at least one DIVIDE-tagged subrange. It is 0 otherwise. The sparse subranges could be generated after the synopsis is initialized (line 3) or after the DIVIDE-tagged subranges are divided (line 15). Finally, N_r denotes the number of rounds needed until SRJA is terminated. It means how many times the loop (line 5 through line 18) is repeated either fully or partially (up to line 9 for breaking the loop). Thus, $N_r \geq 1$. Now M_x is:

$$(H_{01} + H_{10}) + (S_{01} + S_{10}) + \sum_{k=1}^{N_r} \left(f_k^J \cdot (J_{01} + J_{10}) + f_k^D \cdot (S_{01} + S_{10})\right) \quad (14)$$

Normally, $H_{01} = 2$ because a 2-way semijoin is executed. $\tilde{n}_0$ first sends the values with a high count with their count information to $\tilde{n}_1$, and then receives from $\tilde{n}_1$ the list of joinable values as a result (line 2). $\tilde{n}_1$ does the same symmetrically. If these two symmetrical processes are conducted as an asymmetrical one, one message can be saved, and thus, the number of transmissions could be significantly reduced when the distance between $\tilde{n}_0$ and $\tilde{n}_1$ is long. We can let $\tilde{n}_0$ start the process by sending its data to $\tilde{n}_1$. When $\tilde{n}_1$ returns the result, it can send its data as well. Then, $\tilde{n}_0$ finally returns the result. In this way, $H_{01} + H_{10} = 3$. Similarly, $S_{01} + S_{10} = 3$. Meanwhile, $J_{01} + J_{10} = 2$ because one 2-way semijoin in either direction but not the symmetrical two is enough in processing the JOIN-tagged subranges. Thus:

$$M_x = 6 + \sum_{k=1}^{N_r} \left(2f_k^J + 3f_k^D\right) \quad (15)$$

Meanwhile, $Y_i = 2N_r$ because $\tilde{n}_i$ is to send the synopsis to $\tilde{m}$ (line 7) and receive the tagged synopsis from $\tilde{m}$ (line 8) in each round. Thus, $M_y = 4N_r$. Let T_S be the total number of transmissions in SRJA. Then:

$$T_S = d \cdot (6 + \sum_{k=1}^{N_r} \left(2f_k^J + 3f_k^D\right) + 2N_r) \quad (16)$$

6.3.2. Our Scheme

Considering the protocol in Figure 10, we have $X_{i,1-i} = F_{i,1-i} + B_{i,1-i}$, where $F_{i,1-i}$ denotes the number of messages $\tilde{n}_i$ sends to $\tilde{n}_{1-i}$ in the forward reduction phase (line 9 through line 15 in Figure 10), while $B_{i,1-i}$ denotes the number of messages $\tilde{n}_i$ sends to $\tilde{n}_{1-i}$ plus the number of messages $\tilde{n}_i$ receives from $\tilde{n}_{1-i}$ in the backward reduction phase (line 17 through line 20). Now M_x is $(F_{01} + F_{10}) + (B_{01} + B_{10})$.

In executing the optimal sequence of fragment semijoins, R_0 and R_1 are supposed to take turns to play the role of reducer relation. Thus, the number of messages including the Bloom filters for a subsequence of fragment semijoins in the forward reduction phase is equal to the number of this turn overs denoted as N_t. For example, in the sequence of fragment semijoins in Figure 4, there are 5 turn overs with R_1 being the reducer relation at first in Figure 4b. Thus, $F_{01} + F_{10} = N_t$. With the same argument for why $H_{01} + H_{10} = 3$ in SRJA, $B_{01} + B_{10} = 3$. Thus, $M_x = N_t + 3$.

Meanwhile, $Y_i = 2$ because $\tilde{n}_i$ sends the count information of R_i to $\tilde{m}$ once (line 2) and receive from $\tilde{m}$ the optimal sequence and the count information of R_{1-i} once (line 3). Let T_O be the total number of transmissions in our scheme. Then:

$$T_O = d \cdot (N_t + 5) \tag{17}$$

From (Equations (16) and (17)), $T_O < T_S$ if $N_t < \sum_{k=1}^{N_r}\left(2f_k^J + 3f_k^D\right) + 2N_r + 1$. For the comparison, let us not consider the restricted case of $N_r = 1$. This case happens in SRJA only when at least one of R_0 and R_1 has very small number of tuples and their join attribute values are sparsely distributed. Such a case is not of much interest. Now since $N_r \geq 2$, we have $\sum_{k=1}^{N_r}\left(2f_k^J + 3f_k^D\right) \cong 5(N_r - 1)$. The reason for this is as follows: in Figure 10, we note that $f_{N_r}^J = f_{N_r}^D = 0$ for the final round. In each of the interim rounds, we note that $f_k^D = 1$ because without any DIVIDE-tagged subrange the next round except the final one would not be necessary. In each of the interim rounds, it is not necessarily that $f_k^J = 1$. Assuming that normally it holds, $T_O < T_S$ iff $N_t < 7N_r - 4$. The values of N_t and N_r depend on several parameters such as the number of tuples, distribution of joining attribute values, their count distribution, iceberg threshold, effectiveness of non-joinable value filtering, and so on. In the experiments in the previous subsection, it turns out that $N_t < 3$ while $N_r \geq 2$ on average. For such values, $T_O < T_S$ as shown by the measurements in the experiments.

7. Related Work

In-network processing of joins in WSNs has received much attention [3]. The state-of-the-art techniques deal with various types of join queries with different types of spatio-temporal characteristics. Spatially, the type of join that has received so far the most attention is the inter-region join where each of the join operand relation is stored at a region of WSNs [2,6,19–24]. The pair-wise joins where pairing of sensor nodes for join could be determined by predicates are investigated in [25,26]. Temporally, one-shot join against a static join operand relations are dealt with in [2,23,27], while the join submitted as a continuous query against the streaming join operand relations are dealt with in [19–21,24–26,28,29]. As for the type of join predicates, equijoin queries are considered in [2,19,20], while theta-joins are considered in [20–30].

These techniques mostly adapt the conventional join implementations to WSNs (nested-loop join [20–23], hash join [2,19,21], and the sort-merge join [19]). The semijoins and 2-way semijoins are

also employed and adapted to WSNs for filtering of non-joinable tuples [2,23,24,27,29]. In [28,30], on the other hand, a filtering approach is proposed rather than adapting the conventional join algorithms to WSNs. The cost-based optimizations are investigated in [21–26,28].

In [21], a technique called *Distribute-Broadcast Join* adapted from the nested-loop join is proposed. The main contribution of this work is the cost-based selection of optimal join region in WSNs. In [22,23], *Mediated Join* also adapted from the nested-loop join is proposed, where cost-based selection of inner and outer relation for nested-loop join is investigated. In [20], distributed algorithms for indexed nested-loop join and hash join are proposed. In the former, a technique of dynamically creating and using a distributed B+ tree in WSNs is developed. In the latter, a technique of partitioning and joining tuples with geographic hashing is investigated. In [25,26], a *pair-wise join* between any two sensor nodes is investigated. Multiple routing trees are employed and cost-based join initiation for long-running join query is proposed. Also, the issue of adaptive and cost-based re-optimization against the changes of sampled data is dealt with. In [2], *Synopsis Join* adapted from hash join as well as semijoin is proposed. It employs geographic hashing for partitioning and filtering of the joining attribute values and optimally determines the nodes of the final joins for each matched value. In [29], *Two-Phase Self Join* where one join operand relation is fully reduced with a semijoin is proposed. To process a join query, it employs a query decomposition technique assuming that the selection predicate on one relation is highly selective. In [27], *SENS-Join* based on a 2-way semijoin is proposed. It handles a general type of join predicates on multiple attributes, using a quad-tree as a multi-dimensional join filter based on Z-ordering. In [19], *PEJA* adapted from hash join as well as sort-merge join is proposed. In WSNs, physical sorting of the tuples distributed over the sensor nodes is infeasible. Thus, it conducts logical sort of tuples through division of joining attribute value range, partitioning and filtering tuples with geographic hashing. In [28,30], algorithms using the join filters are proposed for continuous queries on multiple attributes. At each sensor node involved in the join query processing, a join filter is installed for each joining attribute and only those tuples whose join attribute values pass the relevant filters are sent to the base station.

Other issues addressed by the state-of-the-art includes routing protocols, query dissemination, join initiation, involvement of the base station in in-network processing, collection of metadata for continuous joins [3].

In [18,24,31], in-network join processing where the Bloom filter is employed in WSNs are addressed. In [18], for a join between an external relation and the virtual relation in WSNs, a technique where a Bloom filter constructed with the joining attribute values of the external relation is injected into the network is investigated. In [24], a technique where the Bloom filter is transmitted instead of the joining attribute values in in-network computation of the two semijoins $R_0 \ltimes R_1$ and $R_0 \rtimes R_1$ to answer a join $R_0 \bowtie R_1$, is proposed. In this technique, when the Bloom filter constructed for R_0 is disseminated through the routing tree of R_1, it is selectively forwarded to the subtrees of the routing tree such that the cost of sending it is always compensated in reduction. The optimal solution in such a selective forwarding is proposed. In [31], an extension of the Bloom filter called Window Bloom filter (WBF) is devised to support a general join query with a time window. The proposed technique represents join attribute values in a compact way with the WBF, and saves the energy consumption by not sending the redundant join attribute values.

As described in Section 3.3, in SRJA [6], the value range instead of the joining attribute values is sent as a histogram-based synopsis. Transmitting the value ranges instead of the joining attribute values is

considered for in-network join processing [18]. In [2], a technique using a histogram-based synopsis of the join operand relations coupled with geographic hashing is proposed. However, the considered join in these techniques is not an iceberg join.

The iceberg query was first introduced in [32]. It is defined as a query that retrieves aggregate values above some specified threshold in the applications such as data warehousing, data mining, information retrieval, and so on. Iceberg query processing over distributed data was also investigated [33–35]. However, the join query was not dealt with in these work. In [7], an iceberg distant join in spatial databases was investigated. In this work, the type of the join query is different from the one we consider in this paper in that the cardinality constraint is only on one join operand relation. That is, when the iceberg threshold is α, the tuple of one relation that is joined with more than α tuples of the other relation is qualified for the query.

8. Conclusions

In this paper, we investigated an alternative approach to processing an iceberg join in WSNs, and described an optimized scheme. In the previous approach, the join predicate is checked first, and the cardinality constraint is checked next. In our approach, the order is reversed. Our scheme refers to the aggregate count of each of the joining attribute values, logically fragmenting the relations on the counts. Based on the fragmentation, it generates the optimal sequence of 2-way fragment semijoins using a Bloom filter as a synopsis of joining attribute values in filtering non-joinable tuples. A detailed set of experiments showed that our approach is substantially superior to the previous one.

As a future work, we plan to extend our scheme for more complicated cases. First, the two regions producing the virtual relations to be joined could partially or fully overlap with each other. Secondly, more than two regions are involved in multiple joins. Thirdly, the correlations among the sensor readings along the neighboring n regions are to be monitored with an n-way join.

Conflicts of Interest

The author declares no conflict of interest.

References

1. Madden, S.; Franklin, M.; Hellerstein, J.; Hong, W. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.* **2005**, *1*, 122–173.
2. Yu, H.; Lim, E.; Zhang, J. On In-Network Synopsis Join Processing for Sensor Networks. In Proceedings of the 7th International Conference on Mobile Data Management, Nara, Japan, 9–13 May 2006; pp. 32–39.
3. Kang, H. In-network processing of joins in wireless sensor networks. *Sensors* **2013**, *13*, 3358–3393.
4. Zhao, F.; Guibas, L. *Wireless Sensor Networks: An Information Processing Approach*; Morgan Kaufmann: San Francisco, CA, USA, 2004.
5. Stattner, E.; Vidot, N.; Hunel, P.; Collard, M. Wireless Sensor Network for Habitat Monitoring: A Counting Heuristic. In Proceedings of the 37th Annual IEEE Conference on Local Computer Networks Workshops, Clearwater, FL, USA, 22–25 October 2012; pp. 753–760.

6. Lai, Y.; Lin, Z.; Gao, X. SRJA: Iceberg Join Processing in Wireless Sensor Networks. In Proceedings of the International Workshop on Database Technology and Applications, Wuhan, China, 27–28 November 2010; pp. 1–4.
7. Shou, Y.; Mamoulis, N.; Cao, H.; Papadias, D.; Cheung, D. Evaluation of Iceberg Distance Joins. In Proceedings of the 8th International Symposium on Advances in Spatial and Temporal Databases, Santorini Island, Greece, 24–27 July 2003; pp. 270–288.
8. Cohen, S.; Matias, Y. Spectral Bloom Filters. In Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003; pp. 241–252.
9. Arasu, A.; Babu, S.; Widom, J. The CQL continuous query language: Semantic foundations and query execution. *VLDB J.* **2006**, *2*, 121–142.
10. Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM* **1970**, *7*, 422–426.
11. Kang, H.; Roussopoulos, N. Using 2-way Semijoins in Distributed Query Processing. In Proceedings of the 3rd IEEE International Conference on Data Engineering, Los Angeles, CA, USA, 3–5 February 1987; pp. 644–651.
12. Roussopoulos, N.; Kang, H. A pipeline n-way join algorithm based on the 2-way semijoin program. *IEEE Trans. Knowl. Data Eng.* **1991**, *4*, 486–495.
13. Bernstein, P.A.; Chiu, D.W. Using semi-joins to solve relational queries. *J. ACM* **1981**, *1*, 25–40.
14. Özsu, M.T.; Valduriez, P. *Principles of Distributed Database Systems*, 3rd ed.; Springer: New York, NY, USA, 2011.
15. Li, Z.; Ross, K.A. PERF Join: An Alternative to Two-way Semijoin and Bloomjoin. In Proceedings of the 4th International Conference on Information and Knowledge Management, Baltimore, MD, USA, 28 November–2 December 1995; pp. 137–144.
16. Andrei, B.; Mitzenmacher, M. Network applications of Bloom filters: A survey. *Internet Math.* **2004**, *4*, 485–509.
17. Mullin, J.K. Optimal semijoins for distributed database systems. *IEEE Trans. Softw. Eng.* **1990**, *5*, 558–560.
18. Abadi, D.; Madden, S.; Lindner, W. REED: Robust, Efficient Filtering and Event Detection in sensor networks. In Proceedings of the 31st International Conference on very Large Data Bases, Trondheim, Norway, 30 August–2 September 2005; pp. 769–780.
19. Lai, Y.; Chen, Y.; Chen, H. PEJA: Progressive energy-efficient join processing for sensor networks. *J. Comput. Sci. Technol.* **2008**, *6*, 957–972.
20. Pandit, A.; Gupta, H. Communication-Efficient Implementation of Range-Join in Sensor Networks. In Proceedings of 11th International Conference on Database Systems for Advanced Applications, Singapore, 12–15 April 2006; pp. 859–869.
21. Chowdhary, V.; Gupta, H. Communication-efficient Implementation of Join in Sensor Networks. In Proceedings of the 10th International Conference on Database Systems for Advanced Applications, Beijing, China, 17–20 April 2005; pp. 447–460.
22. Coman, A.; Nascimento, M. A Distributed Algorithm for Joins in Sensor Networks. In Proceedings of the 19th International Conference on Scientific and Statistical Database Management, Banff, Canada, 9–11 July 2007.

23. Coman, A.; Nascimento, M.; Sander, J. On Join Location in Sensor Networks. In Proceedings of the 8th International Conference on Mobile Data Management, Mannheim, Germany, 7–11 May 2007; pp. 190–197.
24. Min, J.; Yang, H.; Chung, C. Cost based in-network join strategy in tree routing sensor networks. *Inf. Sci.* **2011**, *16*, 3443–3458.
25. Mihaylov, S.; Jacob, M.; Ives, Z.; Guha, S. Dynamic Join Optimization in Multi-Hop Wireless Sensor Networks. In Proceedings of the 36th International Conference on very Large Data Bases, Singapore, 13–17 September 2010; pp. 1279–1290.
26. Mihaylov, S.; Jacob, M.; Ives, Z.; Guha, S. A Substrate for In-Network Sensor Data Integration. In Proceedings of the 5th Workshop on Data Management for Sensor Networks, Auckland, New Zealand, 24 August 2008; pp. 35–41.
27. Stern, M.; Buchmann, E.; Böhm, K. Towards Efficient Processing of General-Purpose Joins in Sensor Networks. In Proceedings of the 25th IEEE International Conference on Data Engineering, Shanghai, China, 29 March–2 April 2009; pp. 126–137.
28. Stern, M.; Böhm, K.; Buchmann, E. Processing Continuous Join Queries in Sensor Networks: A Filtering Approach. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–11 June 2010; pp. 267–278.
29. Yang, X.; Lim, H.; Özsu, M.; Tan, K. In-Network Execution of Monitoring Queries in Sensor Networks. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, 12–14 June 2007; pp. 521–532.
30. Mo, S.; Fan, Y.; Li, Y.; Wang, X. Multi-Attribute Join Query Processing in Sensor Networks. *J. Netw.* **2014**, *10*, 2702–2712.
31. Min, J.; Kim, J.; Shim, K. TWINS: Efficient time-windowed in-network joins for sensor networks. *Inf. Sci.* **2014**, *1*, 87–109.
32. Fang, M.; Shivakumar, N.; Garcia-Molina, H.; Motwani, R.; Ullman, J.D. Computing Iceberg Queries Efficiently. In Proceedings of the 24th International Conference on Very Large Data Bases, New York, NY, USA, 24–27 August 1998; pp. 299–310.
33. Zhao, H.; Lall, A.; Ogihara, M.; Xu, J. Global Iceberg Detection over Distributed Data Streams. In Proceedings of the 26th International Conference on Data Engineering, Long Beach, CA, USA, 1–6 March 2010; pp. 557–568.
34. Manjhi, A.; Shkapenyuk, V.; Dhamdhere, K.; Olston, C. Finding (Recently) Frequent Items in Distributed Data Streams. In Proceedings of the 21st International Conference on Data Engineering, Tokyo, Japan, 5–8 April 2005; pp. 767–778.
35. Zhao, Q.; Ogihara, M.; Wang, H.; Xu, J. Finding Global Icebergs over Distributed Data Sets. In Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Chicago, IL, USA, 26–28 June 2006; pp. 298–307.