Efficient Processing of Distributed Iceberg Semi-joins

Mohammed Kasim Imthiyaz, Dong Xiaoan, and Panos Kalnis

Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{iscp1134, dongxiao, dcskp}@nus.edu.sg

Abstract. The Iceberg SemiJoin (ISJ) of two datasets $\mathcal{R}$ and $\mathcal{S}$ returns the tuples in $\mathcal{R}$ which join with at least k tuples of $\mathcal{S}$. The ISJ operator is essential in many practical applications including OLAP, Data Mining and Information Retrieval. In this paper we consider the distributed evaluation of Iceberg SemiJoins, where $\mathcal{R}$ and $\mathcal{S}$ reside on remote servers. We developed an efficient algorithm which employs Bloom filters. The novelty of our approach is that we interleave the evaluation of the Iceberg set in server $\mathcal{S}$ with the pruning of unmatched tuples in server $\mathcal{R}$. Therefore, we are able to (i) eliminate unnecessary tuples early, and (ii) extract accurate Bloom filters from the intermediate hash tables which are constructed during the generation of the Iceberg set. Compared to conventional two-phase approaches, our experiments demonstrate that our method transmits up to 80% less data through the network, while reducing the disk I/O cost.

1 Introduction

Many practical applications, including Data Warehousing [2], Market-basket Analysis [5] and Information Retrieval [6], rely on Iceberg queries. Such queries compute aggregate functions over a set of attributes and return the aggregate values which are above some threshold. They are called *Iceberg* queries because the result is usually very small (i.e., the tip of an Iceberg) compared to the input set.

Here, we deal with the evaluation of Distributed *Iceberg SemiJoins* (ISJ). Consider the following scenario: Server S stores the transactions of a supermarket's cashiers in a table $\mathcal{S}(pID, rest)$, where pID identifies the product and $rest$ is a set of attributes containing the details of the transaction (e.g., cashier ID, timestamp, etc). A remote server R contains a set $\mathcal{R}(pID)$ of "interesting" products. The user at site R wants to find which of the products in $\mathcal{R}$ sold at least T items in supermarket S. For the dataset of Figure 1 and assuming that $T = 2$, the only qualifying product is coffee (notice that some of the products in $\mathcal{R}$ may not appear in $\mathcal{S}$ and visa versa). Formally, the ISJ query corresponds to the following SQL statement (Query 1):

pID
coffee
toothpaste
cola

(a) Dataset $\mathcal{R}$

pID	rest
coffee	Cashier 2, 10:01
fruit	Cashier 1, 10:02
fruit	Cashier 1, 10:02
cola	Cashier 3, 10:03
coffee	Cashier 1, 10:30

(b) Dataset $\mathcal{S}$

Fig. 1. Two datasets to be joined

```
Query 1:    SELECT S.pID, COUNT(S.rest)
            FROM R, S
            WHERE R.pID = S.pID
            GROUP BY S.pID
            HAVING COUNT(S.rest) >= T
```

Since the datasets reside in remote servers, a straight-forward way to evaluate the ISJ is to employ a two-steps algorithm: (i) Execute Query 2 in server S to find the set of tuples $\mathcal{S}_{ICE}$ which appear at least T times[1] in $\mathcal{S}$.

```
Query 2:    SELECT S.pID, COUNT(S.rest)
            FROM S
            GROUP BY S.pID
            HAVING COUNT(S.rest) >= T
```

(ii) Transfer $\mathcal{S}_{ICE}$ to server R and evaluate the join $\mathcal{R} \bowtie \mathcal{S}_{ICE}$. We call this method *Naïve Iceberg SemiJoin* (nIS).

The simplest way to evaluate Query 2 is to maintain one counter per group in the main memory (i.e., 3 counters in our example). Using this method, we can compute the answer by reading $\mathcal{S}$ only once. However, this is inapplicable in practice, since the number of groups is usually larger than the available memory. Another approach is to sort $\mathcal{S}$ on the pID attribute by employing external sorting; subsequently, Query 2 can be answered by reading the sorted data from the disk. External sorting, however, may require several passes over the data if the available memory is limited. Moreover, in some applications (e.g., Information Retrieval [6]) $\mathcal{S}$ is computed on-the-fly from other relations and is impractical to be materialized. The implicit drawback of both methods is that they generate all possible groups, although (by the definition of Iceberg queries) only few of them are expected to satisfy the threshold. Fang et.al [8] solved this problem by designing a family of algorithms based on sampling and multiple hashing. We call their method *Efficient Iceberg Computation* (EIC).

In this paper we present the *Multiple Filter Iceberg SemiJoin* (MulFIS) algorithm. MulFIS exploits the intermediate steps of EIC in order to minimize the cost of Distributed ISJ. Specifically, instead of computing $\mathcal{S}_{ICE}$ and $\mathcal{R} \bowtie \mathcal{S}_{ICE}$

[1] In our examples we use COUNT for simplicity. However, our methods are also applicable to more complex criteria (e.g., HAVING SUM(S.price * S.itemsSold)>=T).

in two independent steps, MulFIS interleaves the execution in the remote servers. Therefore it can access the internal hash tables which are generated by EIC; these are used to construct accurate Bloom filters [3] (i.e., compact representations of the tables' contents) with minimal extra cost. The resulting Bloom filters are exchanged between the processing sites enabling the elimination of unnecessary tuples at the early stages. We developed a prototype and performed extensive experimentation with standard industry benchmarks. Our results revealed that MulFIS can reduce the network cost of the Distributed ISJ by 80% compared to nIS, while the Disk I/O cost also decreases. We also show that MulFIS is better than alternative implementations which employ Bloom filters at various stages of the query execution.

The rest of this paper is organized as follows: In Section 2 we present the related work together with some essential background; the EIC method is described in more details in Section 3. Next, in Section 4 we discuss several variations of our approach and explain the MulFIS algorithm. Our experimental results are presented in Section 5, while Section 6 concludes the paper.

2 Related Work

A common technique to reduce the network cost of joins in distributed databases is the *SemiJoin* [1]. Assume that relation $\mathcal{R}$ at site R is joined with relation $\mathcal{S}$ at site S on attribute a (i.e., $\mathcal{R} \bowtie_a \mathcal{S}$). SemiJoin reduces the amount of data transferred between the two sites as follows: (i) At site R, it computes $\mathcal{R}' = \pi_a(\mathcal{R})$ and sorts the result. (ii) $\mathcal{R}'$ is sent to site S, where it is used to compute $\mathcal{S}' = \pi_{\mathcal{S}.*}(\mathcal{R}' \bowtie_a \mathcal{S})$. (iii) $\mathcal{S}'$ is transferred back to site R. There, the final join $\mathcal{R} \bowtie_a \mathcal{S}'$ is executed and the results are presented to the user.

BloomJoin [4] is another approach which uses Bloom filters [3] to eliminate non-matching tuples. Bloom filters are bitmap vectors representing the contents of a relation in a compact way. BloomJoin works as follows: (i) A k-bit-vector (i.e., Bloom filter) is initialized to 0 at site R. Then, each tuple of $\pi_a(\mathcal{R})$ is hashed in the vector, setting the corresponding bit to 1. (ii) The vector is sent to site S. By using the same hash function, the values of $\pi_a(\mathcal{S})$ that correspond to 0-bits in the vector, are eliminated. Let $\mathcal{S}'$ be the subset of $\mathcal{S}$ which was not eliminated (i.e., the candidate tuples). (iii) $\mathcal{S}'$ is sent to site R. Finally the join $\mathcal{R} \bowtie_a \mathcal{S}'$ is evaluated at R. BloomJoin typically outperforms SemiJoin in terms of network cost since the filter is generally smaller than the projection on the join attribute. Observe that due to hash collisions in the filter, BloomJoin performs lossy compression. Therefore, the candidate set $\mathcal{S}'$ may contain non-matching tuples which would have otherwise been eliminated by SemiJoin.

Iceberg queries were introduced by Fang et.al. [8] who proposed a family of evaluation techniques based on sampling and multiple hashing. The advantage of these algorithms is that they are output sensitive; therefore, they avoid generating all possible groups of tuples. We will discuss the method further in the next section. Iceberg queries have also been studied in the context of Data Warehousing. The problem in this case is the efficient evaluation of an entire

Iceberg Cube (i.e., multiple Group-Bys). Beyer and Ramakrishnan [2] proposed a bottom-up computation strategy (BUC). BUC is based on the anti-monotonic property which assures that if a cell of a group-by does not satisfy the threshold T, then none of the descendent cells can satisfy T. This idea was extended by Han et.al. [9] for iceberg cubes with complex measures which do not satisfy the anti-monotonic property (e.g., AVERAGE).

Ng et.al [11] focus on the parallel computation of iceberg cubes. They assume a cluster of interconnected workstations and design parallel variations of BUC which aim at reducing the I/O cost and achieve good load balancing. This is different from our work, since they do not consider joins. Several papers also discuss the evaluation of *Top-K* queries in distributed environments. In the work of Fagin et.al [7] for instance, the measure function is evaluated by a distributed join involving several sites. Yu et.al. [12], on the other hand, focus on selecting the appropriate sites which contribute to the answer from a large set of remote sources. Notice, however, that the requirements of distributed Top-K queries are very different from the Distributed ISJ. To the best of our knowledge, Distributed ISJ is discussed only by Mamoulis et.al. [10]; the authors consider spatial datasets and propose a recursive partitioning algorithm to prune the 2-D space. Nevertheless, this paper focuses on spatial joins and requires a middleware evaluation site.

3 Efficient Iceberg Computation (EIC)

In this section we present in details the EIC method [8] since it is the basis of our algorithms. Recall from Section 1 that $\mathcal{S}_{ICE}$ is the answer set for the Iceberg query (i.e., Query 2). We call the tuples in $\mathcal{S}_{ICE}$ *heavy targets*, since they satisfy the threshold T. The aim of EIC is to select a set $\mathcal{F}$ of *potentially* heavy targets by eliminating fast many groups which cannot satisfy the threshold T. Observe that $\mathcal{F}$ does not necessarily contain the correct answer. There are two types of errors: (i) If $\mathcal{F} - \mathcal{S}_{ICE} \neq \emptyset$ then the algorithm generates *false positives*, meaning that $\mathcal{F}$ contains tuples which do not satisfy T. (ii) If $\mathcal{S}_{ICE} - \mathcal{F} \neq \emptyset$, there are *false negatives* meaning that some heavy targets are missed. EIC uses two techniques to compute $\mathcal{F}$:

- *Sampling*: A set $\mathcal{S}_{rnd}$ of random samples is selected from $\mathcal{S}$. Then we calculate the count of each target in the sample, scaled by $\frac{|\mathcal{S}|}{|\mathcal{S}_{rnd}|}$. If it satisfies the threshold, the target is added into $\mathcal{F}$. Obviously the result may contain both false positives and false negatives.
- *Course Count*: This technique uses an array $A[1..m]$ of m counters initialized to zero and a hash function h which maps the grouping attributes v of a tuple to a cell of A. The algorithm works as follows: (i) For each tuple in $\mathcal{S}$ with grouping attributes v, the counter $A[h(v)]$ is incremented by one. After scanning the entire $\mathcal{S}$ the algorithm generates a bitmap vector $BMAP[1..m]$. A bit $BMAP[i], 1 \leq i \leq m$ is set if $A[i] \geq T$. Intuitively, $BMAP$ indicates which hash values correspond to *potential* heavy targets. Notice that $BMAP$

is much smaller than A. (ii) *Candidate Selection*: $\mathcal{S}$ is scanned, and a target with grouping attributes v is added to $\mathcal{F}$ if $BMAP[h(v)] = 1$. Observe that $\mathcal{F}$ may contain false positives but no false negatives. (iii) $Count(\mathcal{F})$: $\mathcal{S}$ is scanned again to explicitly count the frequency of targets in $\mathcal{F}$. The targets that occur less than T times (i.e., false positives) are eliminated, while the remaining targets are the final answer $\mathcal{S}_{ICE}$.

Several hybrid approaches are possible by combining sampling and course count. In this paper we use the Multiscan Defer-Count variation with shared bitmaps. The algorithm first performs sampling. Then, it executes K hash scans using a different hash function $h_i, 1 \le i \le K$ each time and keeps in memory all resulting bitmap vectors $BMAP_i$. During the $(i+1)$ hashing scan, $A[h_{i+1}(v)]$ is incremented by one for the target v only if $BMAP_j[h_j(v)] = 1, \forall j : 1 \le j \le i$. However, the counter is not incremented if v belongs to the heavy targets discovered by sampling. Intuitively, this method decreases the number of false positives in $\mathcal{F}$. Notice that the $Count(\mathcal{F})$ step is still necessary to eliminate any remaining false positives.

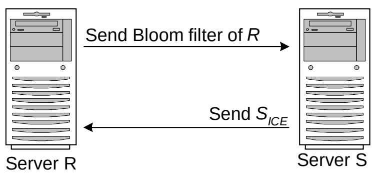
4 Distributed Iceberg SemiJoin

In this section we present our methods for computing the Distributed ISJ operator. Recall from Section 1 that the datasets $\mathcal{R}$ and $\mathcal{S}$ of Query 1 reside in remote servers. We already mentioned a naïve method (nIS) to evaluate the query: First, run EIC at server S to compute the iceberg result $\mathcal{S}_{ICE}$. Then, transfer $\mathcal{S}_{ICE}$ to server R and perform the join $\mathcal{R} \bowtie \mathcal{S}_{ICE}$. Below, we propose sophisticated methods which require fewer disk I/Os and achieve lower network cost than nIS.

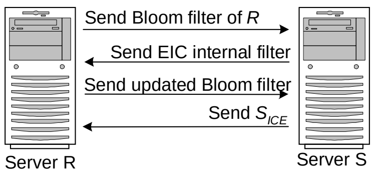
Our methods employ Bloom filters to minimize the amount of information transferred through the network. Since Bloom filters implement lossy compression, they may contain false positives. Several design parameters affect the number of false positives. These are: the number of hash functions k, the size m of the filter in bits and the number of keys n. The following formula expresses the probability of having false positives:

$$FP(k, m, n) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k} \qquad (1)$$

In our implementation, we used only one hash function for simplicity. In order for our methods to perform well, we must use an appropriate value for the size of the Bloom filter. If m is too small, there will be too many false positives and the pruning will not be effective; if m is too large, the Bloom filters will add excessive overhead. Since the value of m depends on n, we must estimate the number of unique keys in the relations. For $\mathcal{R}$ we can simply check the statistics of the DBMS. However, it is more difficult to estimate the unique keys of $\mathcal{S}_{ICE}$, since they depend on the threshold T and the selectivity of the filters. In our prototype, we selected m empirically; developing a formal cost model is outside



(a) ISIF



(b) MulFIS

Fig. 2. The ISIF and MulFIS evaluation methods

the scope of this paper. Next, we present three algorithms which evaluate the Distributed ISJ by using Bloom filters.

4.1 Naïve Filtered Iceberg SemiJoin (nFIS)

The first method, nFIS, is a straight-forward extension on the naïve algorithm nIS; we present it here for completeness. The algorithm starts by computing $\mathcal{S}_{ICE}$ at server S. Then it continues as a traditional BloomJoin: Server S receives a Bloom filter of $\mathcal{R}$ which is used to eliminate unnecessary tuples from $\mathcal{S}_{ICE}$. The remaining tuples $\mathcal{S}'_{ICE}$ are sent to server R where the join $\mathcal{R} \bowtie \mathcal{S}'_{ICE}$ is evaluated.

4.2 Iceberg SemiJoin with Internal Filtering (ISIF)

ISIF works as follows (Figure 2.a): Server R sends a Bloom filter of $\mathcal{R}$ to S. Then, S executes the EIC algorithm; in every step of EIC (i.e., sampling, candidate selection and counting) it uses the Bloom filter to prune the non-qualifying groups. When EIC terminates, it generates a set $\mathcal{S}'_{ICE}$ which is essentially the same as in nFIS. $\mathcal{S}'_{ICE}$ is sent to server R where the final join is evaluated. The advantage of ISIF over nFIS is that it prunes many groups earlier; therefore it saves some effort from EIC. We show in the next section that ISIF outperforms nFIS in terms of execution time.

4.3 Multiple Filter Iceberg SemiJoin (MulFIS)

The intuition of the MulFIS algorithm (Figure 2.b) is that the internal hash tables generated by EIC during the computation of the iceberg set, can be used as accurate Bloom filters to reduce further the network cost. Similar to ISIF, MulFIS starts by sending a Bloom filter BF_1 of $\mathcal{R}$ to site S. There, the execution of EIC starts and BF_1 is used to eliminate non-qualifying groups during the sampling phase and the construction of the first hash table. By using the intermediate hash table of EIC, a new Bloom filter BF_2 is generated and sent back to site R, where it is used to prune tuples from $\mathcal{R}$. From the remaining tuples of $\mathcal{R}$ we construct a third Bloom filter BF_3 which is transferred to site S. There, the evaluation of EIC resumes and for the rest of the steps (i.e., second hash scanning, candidate selection and counting), BF_3 is used to prune more tuples. The result $\mathcal{S}_{ICE}$ of EIC is sent to site R, where the final join $\mathcal{R} \bowtie \mathcal{S}_{ICE}$ is computed. The details are presented in Algorithm 1. Until now we have assumed that EIC performs only two hash scans. In practice more scans may be necessary if the available memory in site S is very small; in this case, steps 3b - 7 are repeated. By employing multiple filters MulFIS manages to reduce significantly the network cost, as we will see at the next section. Additionally, it reduces the false positive tuples in $\mathcal{F}$ and therefore requires less disk accesses and fewer CPU cycles during the evaluation of EIC.

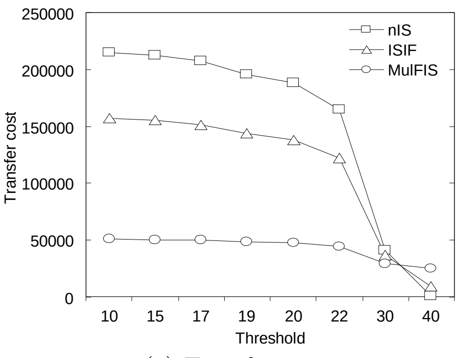
Algorithm 1 Multiple Filter Iceberg SemiJoin (MulFIS)

```
1.  Generate a Bloom filter $BF_1$ for $\mathcal{R}$
2.  Send the query and $BF_1$ to site S
3a. Perform sampling in $\mathcal{S}$. Use $BF_1$ to prune tuples during sampling
3b. Perform hash scanning $h_1$ in $\mathcal{S}$. Use $BF_1$ to prune tuples during scanning
4.  Use the hash table from 3b to generate a Bloom filter $BF_2$
5.  Send $BF_2$ to site R
6a. Scan $\mathcal{R}$ and use $BF_2$ to eliminate unmatching tuples
6b. Generate a Bloom filter $BF_3$ from the qualifying tuples of $\mathcal{R}$
7.  Send $BF_3$ to site S
8a. Perform hash scanning $h_2$ in $\mathcal{S}$. Use $BF_3$ to prune tuples during scanning
8b. Perform Candidate Selection. Use $BF_3$ to prune tuples while generating $\mathcal{F}$
8c. Generate $\mathcal{S}_{ICE}$ by executing Count($\mathcal{F}$). Use $BF_3$ to prune tuples
9.  Send $\mathcal{S}_{ICE}$ to site R
10. Evaluate the join $\mathcal{R} \bowtie \mathcal{S}_{ICE}$ in site R
```

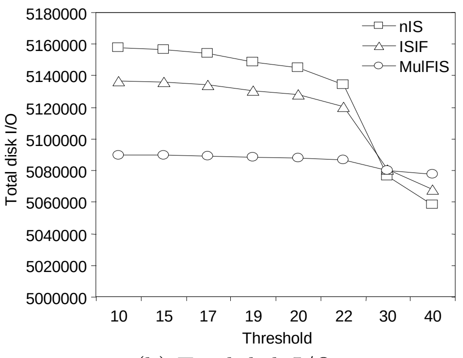
5 Experimental Evaluation

In order to evaluate the effectiveness of the proposed algorithms, we developed a prototype in C++ running on Sun UltraSparc III machines. The servers were physically connected to a 100Mbps LAN and the communication between them was achieved through Unix sockets. We extracted our datasets from the industry standard TPC-H benchmark. The $\mathcal{S}$ relation consists of around 1.3M tuples from the LineItem table. We generated three instances of $\mathcal{R}$ denoted as $\mathcal{R}_{10}, \mathcal{R}_{30}, \mathcal{R}_{60}$. All instances contain 10K tuples, but only a subset of them (i.e., 10%, 30% and 60%, respectively) joins with at least one tuple of $\mathcal{S}$. We executed Query 1 for several threshold values T. We selected T such that the cardinality of the iceberg result $\mathcal{S}_{ICE}$ varied from around 130 to 35,000.

We measured the network cost of each algorithm by counting the number of bytes transmitted through the network. The largest amount of data that can be transferred in one physical frame is referred to as MTU (Maximum Transmission Unit); for Ethernet, $MTU = 1500$ bytes. Each packet consists of a header and the actual data. The largest segment of TCP data that can be transmitted is called MSS (Maximum Segment Size). Essentially, $MTU = MSS + B_H$, where B_H is the size of the TCP/IP headers (typically, $B_H = 40$ bytes). Let D be a set of data. The size of D in bytes is $B_D = |D| \cdot B_{obj}$, where B_{obj} is the size of each object in bytes (e.g., 4 bytes for an integer attribute). Thus, when the



(a) Transfer cost



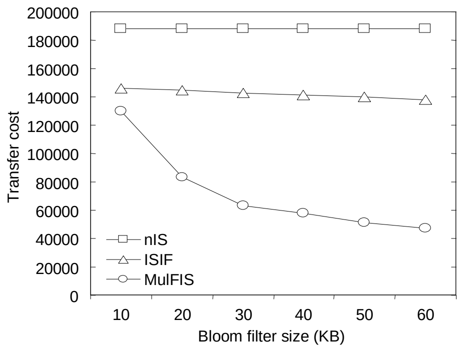
(b) Total disk I/O

Fig. 3. Performance vs. the threshold T for the $\mathcal{R}_{60}$ dataset.

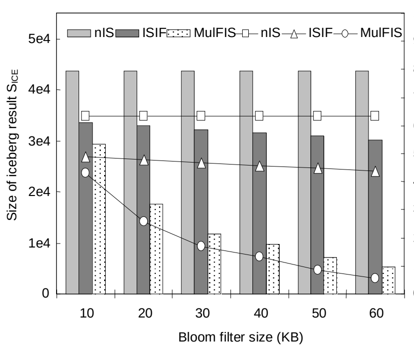
whole D is transmitted through the network, the number of transferred bytes is: $T_B(B_D) = B_D + B_H \cdot \lceil \frac{B_D}{MSS} \rceil$, where the second component of the equation is the overhead of the TCP/IP headers.

In the first set of experiments (Figure 3), we measured the performance of the algorithm for varying threshold T. For fairness, we used the $\mathcal{R}_{60}$ dataset, which represents the worst case for MulFIS, while we set the size of the Bloom filters to 60KB. MulFIS clearly outperforms the other algorithms in terms of transfer cost for most values of T. This is due to the multiple filters which manage to eliminate early most unnecessary tuples. Notice that while ISIF achieves some improvement over nIS, it is still much worse than MulFIS due to the lack of feedback from the intermediate hash tables in server S. Observe, however, that for large values of T, nIS performs slightly better. In this case, the cardinality of $\mathcal{S}_{ICE}$ is so small that does not justify the overhead of the Bloom filters. In practice, we can minimize the performance loss of MulFIS by using smaller filters, as we discuss below. In terms of disk I/O the trend is similar, although the relative difference is considerably smaller. This is due to the fact that the I/O cost is dominated by the scanning of $\mathcal{S}$. Improving this aspect of MulFIS is part of our on-going work. Notice, finally, that we do not present the results of nFIS because the transfer and I/O cost are the same as these of ISIF.

In the next experiment we set $T = 20$ and vary the size of the Bloom filter. In Figure 4.a we present the transfer cost. If the Bloom filter is very small (i.e., 10KB), there are many hash collisions; therefore MulFIS cannot prune enough tuples. For larger values, the collisions decrease fast; thus MulFIS performs similarly for 40-60KB large filters. Obviously, nIS is constant since it does not employ any filter. Notice that the performance of ISIF also does not vary significantly. This indicates that the most important factor for the performance gain of MulFIS is the feedback mechanism from the internal hash tables of S. To investigate this further, we show in Figure 4.b the size of the iceberg result $\mathcal{S}_{ICE}$ (bars) and the number of false positives (lines). As expected, both metrics decrease in MulFIS when we use more accurate (i.e., larger) filters due to the feedback, while ISIF is almost unaffected.



(a) Transfer cost



(b) $|\mathcal{S}_{ICE}|$ and #of false positives

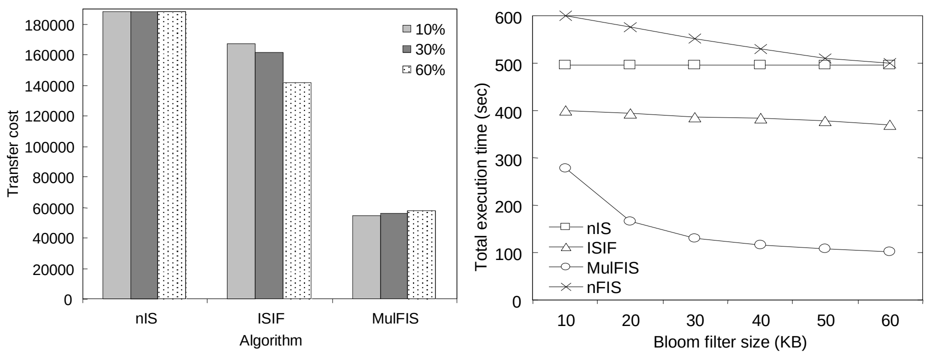
Fig. 4. Performance vs. the Bloom filter size ($T = 20$, $\mathcal{R}_{60}$ dataset).

Next (Figure 5.a) we draw the transfer cost for each of the three datasets $\mathcal{R}_{10}, \mathcal{R}_{30}, \mathcal{R}_{60}$; we used 60KB Bloom filters and T was set to 20. nIS is constant since it always transmits the same $\mathcal{S}_{ICE}$ result (i.e., no pruning). As expected, ISIF performs better when more tuples of $\mathcal{R}$ join with $\mathcal{S}$, since a larger percentage of the Bloom filter contains accurate information. Observe, however, that the trend for MulFIS is different. This happens because there is an overhead of the additional Bloom filters, while the feedback step of MulFIS does not depend on the number of matching tuples. Nevertheless, the overhead is very small and MulFIS still outperforms significantly the other algorithms.

In the final experiment (Figure 5.b) we show the actual running time of the algorithms. The trend is similar to this of the transfer cost (compare with Figure 4.a). Here we also present the nFIS algorithm. Recall that the transfer cost of nFIS is the same as ISIF, since both algorithms employ only one Bloom filter. The difference is that the filter is used at the intermediate steps of ISIF but only at the last step of nFIS. Observe that the actual running time is affected considerably, due to the larger set of intermediate results in nFIS. The performance is even worse than nIS, due to the additional overhead of the filter.

6 Conclusion

In this paper we dealt with the evaluation of the Distributed Iceberg Semi-Join operator. This operator is essential in numerous real-life applications. It is used, for instance, to analyze information from two independent datamarts, or to extract correlated documents from a remote digital library. We developed MulFIS, an efficient algorithm which interleaves the execution of the iceberg query and the join in the two servers and uses the internal hash tables to prune the non-qualifying groups. We developed a prototype and used an industry standard benchmark to validate that MulFIS provides significant advantages over its competitors. Currently, we are working towards two directions: (i) to improve further the disk I/O cost and (ii) to describe formally the behavior of our algorithm by developing an accurate cost model.



(a) Transfer cost for $\mathcal{R}_{10}, \mathcal{R}_{30}, \mathcal{R}_{60}$ (b) Total execution time (sec)

Fig. 5. Performance vs. $\mathcal{R}$ and total execution time ($T = 20$).

References

1. P.A. Bernstein and D.W. Chiu. Using semijoins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.
2. Kevin S. Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 359–370, 1999.
3. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
4. Kjell Bratbergsengen. Hanshing methods and relational algebra operations. In *Proc. of the 10th Int. Conf. on Very Large Data Bases (VLDB)*, pages 323–333, 1984.
5. S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 255–264, 1997.
6. A. Broder, S.C. Glassman, and M.S. Manasse. Syntactic clustering of the web. In *Proc. of the 6th Int. World Wide Web Conference*, 1997.
7. Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proc. of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001.
8. Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proc. of the 24th Int. Conf. on Very Large Data Bases (VLDB)*, pages 299–310, 1998.
9. Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, 2001.
10. Nikos Mamoulis, Panos Kalnis, Spiridon Bakiras, and Xiaochen Li. Optimization of spatial joins on mobile devices. In *Proc. of the 8th Int. Symposium on the Advances in Spatial and Temporal Databases (SSTD)*, pages 233–251, 2003.
11. Raymond T. Ng, Alan Wagner, and Yu Yin. Iceberg-cube computation with pc clusters. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 25–36, 2001.
12. Clement T. Yu, George Philip, and Weiyi Meng. Distributed top-n query processing with possibly uncooperative local systems. In *Proc. of the 29th Int. Conf. on Very Large Data Bases (VLDB)*, pages 117–128, 2003.