2004

Compressed positionally encoded record filters in distributed query processing.

Ying (Joy) Zhou
University of Windsor

Compressed Positionally Encoded Record Filters in Distributed Query Processing

By

Ying (Joy) Zhou

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2004

Your file Votre référence
ISBN: 0-612-92460-2
Our file Notre référence
ISBN: 0-612-92460-2

Abstract

Different from a centralized database system, distributed query processing involves data transmission among distributed sites, which makes reducing transmission cost a major goal for distributed query optimization. A Positionally Encoded Record Filter (PERF) has attracted research attention as a cost-effective operator to reduce transmission cost. A PERF is a bit array generated by relation tuple scan order instead of hashing, so that it inherits the same compact size benefit as a Bloom filter while suffering no loss of join information caused by hash collisions.

Our proposed algorithm PERF_C (Compressed PERF) further reduces the transmission cost in algorithm PERF by compressing both the join attributes and the corresponding PERF filters using arithmetic coding. We prove by time complexity analysis that compression is more efficient than sorting, which was proposed by earlier research to remove duplicates in algorithm PERF. Through the experiments on our synthetic testbed with 36 types of distributed queries, algorithm PERF_C effectively reduces the transmission cost with a cost reduction ratio of 62%-77% over IFS. And PERF_C outperforms PERF with a gain of 16%-36% in cost reduction ratio.

A new metric to measure the compression speed in bits per second, "compression bps", is defined as a guideline to decide when compression is beneficial. When compression overhead is considered, compression is beneficial only if compression bps is faster than data transfer speed. Tested on both randomly generated and specially designed distributed queries, number of join attributes, size of join attributes and relations, level of duplications are identified to be critical database factors affecting compression. Tested under three typical real computing platforms, compression bps is measured over a wide range of data size and falls in the range from 4M b/s to 9M b/s. Compared to the present relatively slow data transfer rate over Internet, compression is found to be an effective means of reducing transmission cost in distributed query processing.

Keywords: distributed query processing, PERF, compression, arithmetic coding, bloom filter, two-way semijoin, bps

Dedication

To Dr. J. Morrissey

To my parents

To my daughter

Acknowledgement

Endless thanks first go to my supervisor, Dr. Morrissey. Without her consistent trust, encouragement and support, both academically and mentally, I would not be able to finish my thesis work. Like a stream in the desert, her great personality, sympathy and wisdom always inspired and guided me through difficulties. I cannot thank her enough.

Thanks also go to my external reader Dr. Hlynka, who took so much effort to correct even trivial typos. Thanks my internal reader Dr. Jaekel for her suggestion on network issues and my committee Chair Dr. Aggarwal for his precious time.

Thanks go to Dr. Sodan for giving me access to the IBM cluster, which becomes an important computing platform for my thesis.

Thanks go to my friends and colleagues for their help.

Last thanks go to my loving parents, and my lovely daughter, who give me endless love and continuous source of strength and hope.

Table of Contents

List of Figures

Chapter 1 Introduction

One of the major goals in Distributed Query Processing (DQP) is to reduce the amount of transmission among the distributed sites. Since the dominant data transmission in DQP is created by distributed join operations, new operators such as semijoin, bloom join and PERF join had been proposed to replace the original join operation. Due to the compact size of bloom filters, algorithms using bloom filters consistently perform better than those employing semijoins [ML86] [MO98] [MO99] [MOL00]. But bloom join suffers from the loss of join information due to unavoidable hash collisions; therefore a relation may only be partially reduced after bloom joins.

Algorithms employing Positionally Encoded Record Filters (PERFs) [LR95] and variations such as Complete Reduction Filters (CRFs) [Zha03] and Composite Semijoin Filters (CSFs) [Zhu04] came into being with the desire to keep the advantage of bloom filters as well as to avoid the drawback of hash collisions. PERF bit arrays are generated and transmitted based on the relation tuple scan order instead of hashing. PERF join has been shown [LR95][HF00][Zha03] [Zhu04] to be a cost effective operator due to its dual benefits. A PERF join outperforms a bloom join since it inherits the advantage of a bloom filter without suffering the side effect of hash collisions. A PERF has the same storage and transmission efficiency as a bloom filter and preserves the complete join information by generating a filter based on the relation tuple scan order. Therefore, reduction methods that require the original join information to be retained can still be considered in PERF based algorithms. PERF join outperforms two-way semijoin due to 1) cheaper transmission cost due to the compact size of PERFs instead of joining attributes; 2) cheaper local operation to do the backward reduction as the compact PERF bit arrays might be able to fit into memory, so that the I/O cost to do semijoin on large-sized attribute can be avoided.

However, to proceed with the PERF reduction properly, it is required that the site receiving the PERFs keep the order of relation tuples the same once join attribute projections are sent to do the forward semijoin. This requirement becomes hard to fulfill when the duplicates in the join attribute projections are to be eliminated to reduce transmission cost. Sorting was proposed to eliminate duplications in former research

[LR95]. As duplicate removal is a general topic in DQP and sorting is time consuming, it is worthwhile to seek better solutions to remove duplicates efficiently. We propose algorithm PERF_C, which applies arithmetic coding to compress both the joining attributes and the PERF filters in PERF join to further reduce the transmission cost, based on the fact that arithmetic coding is more efficient than sorting. A new metric "compression bps" is defined to guide the decision on when compression is beneficial, when the cost of compression has to be addressed. Evaluations of algorithm PERF_C and compression bps are conducted based on 36 types of randomly generated distributed queries. The evaluation results of PERF_C are compared with PERF to investigate the compression effects. Specially designed tests on compression bps are also carried out to identify the critical database factors that will influence compression bps.

The rest of this thesis is organized in 6 chapters. After briefly reviewing the literature concerning DQP strategies and algorithms in chapter 2, the inefficiency of using sorting to eliminate duplicates in PERF is discussed and the motivation to use compression in PERF is stated in chapter 3. A compressed PERF algorithm is proposed in chapter 4 to demonstrate where and how compression can be applied in PERF. Compression bps is also defined in Chapter 4 with theorems 1 and 2 to discuss whether compression is still beneficial when the cost of compression has to be considered. Chapter 5 describes the implementation of algorithm PERF_C and the evaluations designed and carried on algorithms PERF_C and compression bps. The conclusion and future work are given in Chapter 6 before the thesis ends with the bibliography.

Chapter 2 Literature Review

A Distributed Database (DDB) is a collection of multiple, logically interrelated databases, which are dispersed geographically over a computer network and maintained by local computers [TC92][Ozs99]. Distributed Query Processing (DQP) is the process of retrieving data from different sites. The challenge of DQP is to design and develop a sequence of operations to minimize the cost of executing the query. The cost of distributed query processing can be grouped into local cost and transmission (communication) cost. Most research focused on minimizing the transmission cost, ignoring the local processing cost. It has been proven that finding such an optimal solution of DQP is NP-hard [CL90][WC96] [HK94][BR88][PV88] due to the complexity of the exhaustive enumeration of all possible query processing plans. Thus, heuristic algorithms have been proposed since the 1970's to quickly develop sub-optimal or near-optimal algorithms for distributed query processing.

2.1. Assumptions, definitions and models used in DQP

2.1.1. General assumptions

1. A point-to-point network is assumed and each distributed node has local processing and storage capabilities.
2. The relations are distributed amongst the nodes and all nodes can access all data.
3. Only select-project-join (SPJ) queries are considered.

2.1.2. Distributed query processing model

Typically the optimization of a distributed query is processed in three phases [KR87] [RK91][CL84][YC84][LR95][HF00]:

1. Initial local processing phase

   All local processing that requires no inter-site communication is performed here, which includes selection and projection on the joining and target attributes of relations.

2. Reduction processing phase

   A reducer, such as a semijoin or hash semijoin schedule is derived from the remaining join operations and executed to reduce the sizes of all joining relations in a cost-effective way.

3. Final query processing phase

All the preprocessed relations and intersite results are transmitted to a final site where the joins are performed and the answer to the query obtained.

2.1.3. Cost models and cost-effective

Two cost models are of interest in distributed query processing. The ***response time*** [SB82] calculates the elapsed time from the start of the query to the point when the final results are obtained, which can always be reduced if the queries can be processed in parallel. And the ***total time cost*** [ESW78] includes all the costs involved in the query processing. It is assumed that the cost involved in transmitting data from one site to another is linear, and the local processing cost is considered to be negligible in most cases. Therefore, as used in most research papers, the total time cost C (X) is represented as a linear function of the size of data transmitted, i.e. $C (X) = C_0 + C_1 * X$, Where X is the amount or size of data transmitted and C_0 and C_1 are system-dependent constants between distributed sites. The cost of data transmission is assumed to be the same between any two distributed computers for the same size of data transmitted. However, local processing cost can be significant for distributed systems under very high-speed network [RK91].

If applying an algorithm to the query processing results in the reduction of the transmitted data among distributed sites, this reduction is called the ***Benefit*** of this algorithm. However, the operation of this algorithm may necessarily involve ***Cost***, including local processing and intersite processing cost. Whether this algorithm is profitable or not depends on the ***Profit***, which is defined to be ***(Benefit - Cost)***. A query processing using a certain algorithm is called ***cost-effective*** or ***profitable*** if the ***cost*** introduced by executing this algorithm is less than the ***benefit***.

2.1.4. Selectivity model

Selectivity of an attribute is defined to be its cardinality divided by the size of its domain, under the assumption that the attribute value of each join attribute is uniformly and independently distributed over the domain of the corresponding attribute. The selectivity model is used to predict the reduction effect after the reduction process, as selectivity varies from 0 to 1. The selectivity of a join attribute projection dij, is $\rho(dij) = |dij|/|D (dij)|$, where |dij| is the cardinality of the projection of relation i over attribute j and |D (dij)| is the cardinality of the domain for attribute dij. |D (dij)| is the possible number of values (tuples) in the domain of dij, not a count of the real values occurring in the database. And it is assumed to be finite and known.

The selectivity of an attribute can be used to estimate the size of a relation Rk after reduction phase. For example, after the execution of a semijoin from Ri to Rk based on the joining attribute j, denoted by Ri$\rtimes$j Rk, the projection of attribute A at relation Rk is reduced to Rk'. Then S (Rk)' = S (Rk) * ρ(dij), where S (Ri) is the size of the relation in bytes.

2.2 DQP Strategies and algorithms

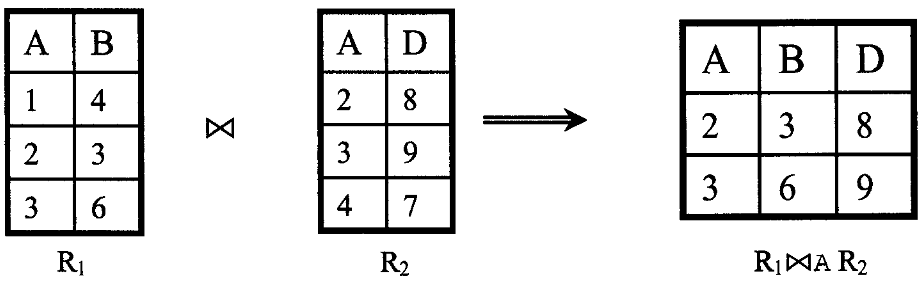
Various distributed query processing strategies and algorithms have been proposed in the past 20 years. They are grouped in this section to be join based, semijoin based, Bloom join based and PERF join based algorithms.

2.2.1 Initial feasible solution (IFS)

This is the most intuitive and basic strategy in DQP. All the relations are retrieved and shipped to the query site and the joins are performed there. Obviously, it is the simplest but rarely efficient. The cost will be associated with the size of relations that are transmitted.

2.2.2 Join and join based algorithms

The join operation is one of the fundamental relational database query operations to combine data stored at different sites [ME92][YC84]. R1 joins R2 on the same joining attribute A, denoted by $R_1 \bowtie_A R_2$, is shown in Figure 2.1.



A	B
1	4
2	3
3	6

R_1

$\bowtie$

A	D
2	8
3	9
4	7

R_2

$\Longrightarrow$

A	B	D
2	3	8
3	6	9

$R_1 \bowtie_A R_2$

Figure 2.1: Join operations

Join is performed by concatenating tuples of R_1 and R_2 where the value of attribute A is equal for both relations. In distributed systems, R1 and R2 may be located in different sites. So the join operation in DQP is performed by shipping the relation that has a smaller size to the bigger-sized one and performing the join there. The join result is then shipped to the final query site. The communication cost is the amount of data transferred over the network, and the benefit is the tuples eliminated by the join.

Since the join between two relations sometimes can eliminate tuples that do not match, it is natural that join be used to reduce the relations first before each of them is sent to the final site individually. However, the relation after join could be much larger than the participating ones, so join operations may expect greater cost of data transfer. Moreover, if there are a small percentage of tuples, which are of interest to the final site, it is obviously a waste to ship the entire relation. To summarize, although the join operation has the advantage of simplicity, it is still very expensive and effort has been made to optimize the join operators.

2.2.3 semijoin and semijoin based algorithms

Under the situation that sometimes shipping the whole relation to perform join is too expensive if only a few tuples are really needed, the semijoin operator was proposed [BC81][BGWR$^+$81] as an effective method to eliminate the transmission cost of traditional join. The basic idea is simple: instead of shipping the whole relation to perform the join, only the joining attributes are projected and shipped.

As illustrated in Figure 2.2, semijoin Ri $\rtimes$A Rj on joining attribute A is computed in the following steps:

1) Project the joining attribute A of Ri locally in site i.

2) Ship the projection Ri [A] to Rj.

3) Perform a join Rj $\bowtie$ Ri [A] of Rj with the attribute Ri [A]

4) Reduce Rj by eliminating tuples whose attribute A are not matching any value in Ri [A].

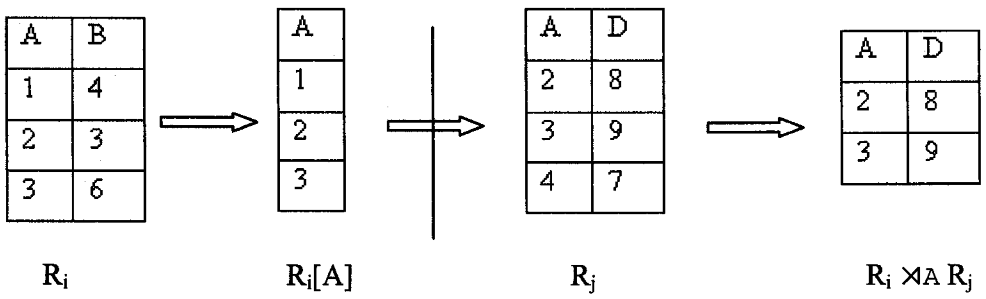


Figure 2.2: Semijoin operations

Step 2) is the cost of the semijoin. It is a leaner function of the size of Ri [A]. Step 4) contributes to the benefit of the semijoin. The benefit is "size of Rj before semijoin –size of Rj after semijoin". If the cost < benefit, this is a cost-effective semijoin.

Since semijoin was adopted in DQP, attention has been paid to improving the basic semijoin. As a result, new methods such as two-way semijoin [KR87], N-way pipelined semijoin [RK91], domain specific semijoin [CL90], composite semijoin [PC90], one-shot fixed precision semijoin [WLC91] have been proposed to enhance the performance of basic semijoin approach. Only two-way semijoin is mentioned below for the sake of relevance.

Kuang created a two-way semijoin operator [KR87] that can be used to replace semijoin to reduce relations in the backward direction as well. A two-way semijoin $R_i \longleftarrow A \longrightarrow R_j$ is illustrated in Figure 2.3 with 4 steps. It is called enhanced two-way semijoin [RK91], which further reduces the backward semijoin by shipping back to Rj the smaller size of the forward reduced relation Ri.

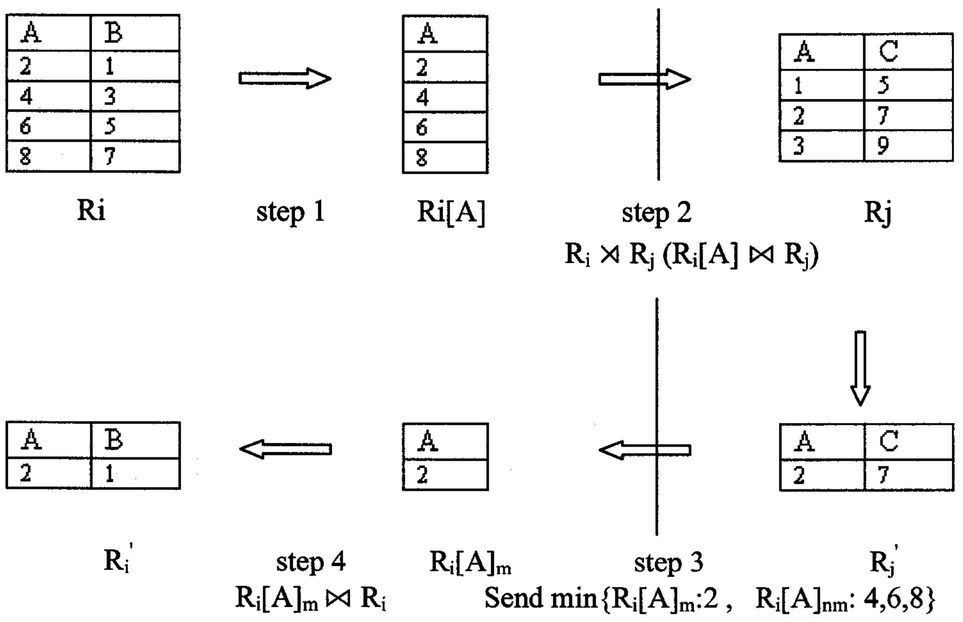


Figure 2.3: 2 – way semijoin operation

1. Send Ri [A] from site i to j.

2. Execute Ri [A] ⋉Rj to get the forward reduction of Rj. Then partition Ri [A] into Ri

[A] m and Ri [A] nm, where Ri [A] m contains the values of attribute A which match one of the values in Rj [A], and Ri [A] nm equals to (Ri [A]-Ri [A] m).

3. Send min {Ri [A] m, Ri [A] nm} from site j back to i.

4. Perform backward reduction. If Ri [A] m is smaller-sized, execute Ri [A] m ⋈ Ri. If Ri [A] nm is smaller, the tuples in Ri with attribute A that matches Rj are eliminated.

2.2.4 Bloom join and bloom Filter based algorithms

A bloom filter, as introduced by Bloom [Blo70], is a compact data structure for probabilistic representation of a set in order to support the membership queries (i.e., "Is element *X* in set *Y*?")[RI01]. A bloom filter was first applied in distributed query processing as Bloom join by Mullion [Mul83]. An array of bits is generated using a hash function on a join attribute and it is transmitted to the remote site instead of the join attribute itself. As the size of a bloom filter is normally smaller than the original joining attribute, the communication cost can be saved. As shown in Figure 2.4, a bloom filter is constructed at Ri using hash function f (x) = x mod 5, and it is shipped to site j as the reducer to reduce the relation Rj. A one-transform bloom filter based semijoin is called hash semijoin [CCY92][TC92] or bloom join [LR95]. Bloom join was proposed [Mul90][CCY92][TC92] as a new operator to replace semijoin and it can be described as follows:

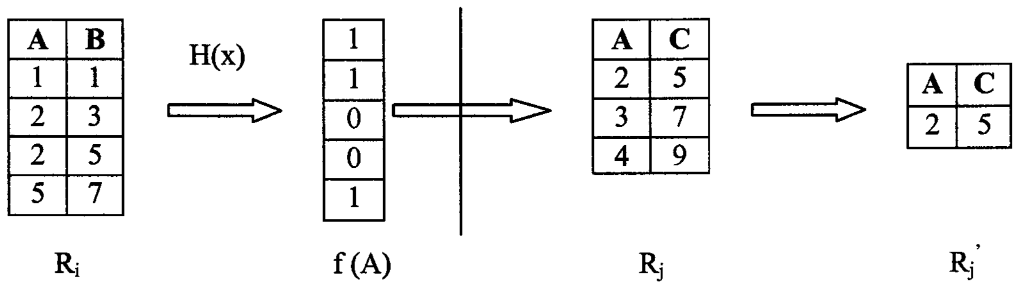


Figure 2.4: Bloom filter operation with hash function H(x) = x

1. Create a filter f (A) at site i.

Create a bit array as the filter and initiate all bits in the array into zero. Develop a hash function and use it to produce an address in the array for each value in the joining attribute. For each address produced, set the corresponding bit in the bit array to 1.

2. Reduce relation size

Send the filter f (A) from site i to site j. Hash on each value in join attribute A of Rj, using the same hash function to produce an address for Rj [A]. Check the bit value in the corresponding address in filter f (A); if the bit value is 1, the corresponding Rj tuple is kept. If the value is 0, discard this Rj tuple.

A bloom join or hash semijoin was shown to be beneficial due to flexibility and simplicity [CCY92]. It was suggested that a hash semijoin could be considered when a pure semjoin is not profitable due to the large width of the joining attribute [CCY92][TC92]. Through experiments, it was demonstrated that simply replacing semijoin with hash semijoin is almost always beneficial [MO98][MO99]. Closer attention was given to the effect of collisions on reduction filters and it was found that hash semijoin still achieved significant reductions even at a collision rate of 60% [MOL00]. Besides being applied to DQP in the form of hash semijoin, Bloom filters are also adopted in many applications when the space savings effect out-weights the collision drawbacks. Active research areas using bloom filters include web cache sharing [FCAB99], query filtering and routing [GBHC00][KBC+00], compact representation of a differential file [Mul83] and free text searching [Ram89].

However, a bloom filter may posit an element in a set when it is not, which in the literature is called a collision, a false position [Mit02], or a false drop. A collision happens in Figure 2.5. The tuple with the field of attribute A equals to 6 is posited in the final join since 6 has the same address as 1. However, it is an unwanted tuple.

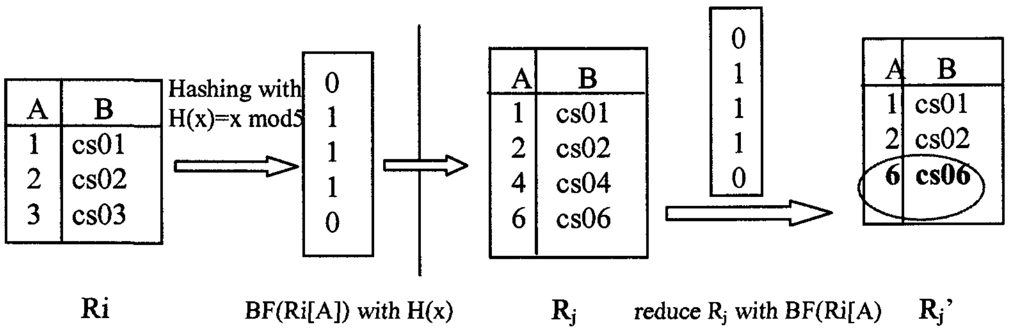


Figure 2.5: Collision happens in Bloom join with H(x) = x mod 5

A perfect hash function can be assumed to simplify the handling of collisions [Mor96][Ma97]. A perfect hash function is a time and space efficient implementation of static sets, which guarantees no collision. In [Sch90], a code generator called GPERF is developed, which can produce a perfect hash function, a mini hash function and a near-perfect hush function according to a given set of keys. However, a perfect hash function is hard to find and the bloom filter generated by a perfect hash function normally has huge size. So it is not practical to use a perfect hash function in DQP.

As far as collisions are concerned, there are 3 parameters that influence the construction of a m-bit bloom filter for a set with n members.

- The number of hash functions, denoted by k for h_1, h_2, …h_k. Parameter k corresponds to computational overhead;
- The size of the filter m, which counts for memory storage needed as it is preferred to put the filter in memory;
- The collision (error) rate ρerr

Bloom [Blo70] shows that for a given bloom filter of size m, the optimal number of hash functions k = ln2 (m/n) should be used to minimize collision rate ρerr. At this point, ρerr = (1/2) k, and the filter size m = (k * n)/(ln2). However, only a small number of hash functions are used in practice because of the adding of the computational overhead associated with each hash function. Research has shown that simply adding hash functions will not decrease the collision rate significantly, after the number of hash functions reaches a certain threshold [RI01].

Compressed bloom filters were introduced [Mit02] as another reasonable way to reduce collisions. The bloom filter is built with a very large size and is compressed to trade for a lower collision rate with a small number of hash functions. To maintain the same collision rate, having a much lager size bloom filter allows a smaller number of hash functions to be used, so that the computation cost to do the multi-transform of the hash functions can be reduced. The trade-off is the cost of compression and decompression, which happens only once before transmission and upon receipt of message. Simple arithmetic coding [MNW98] was suggested [Mit02], as its compression scheme is able to achieve near-optimal compression performance, with fast implementations for both compression and decompression.

Due to the unavoidable collisions in bloom filter based algorithms, the relation can only be partly reduced using bloom filter-based semijoins. So Bloom filter based algorithms are also referred to as lossy semijoins [Mit02][RI01], compared with the full reduction by using semijoins. A Bloom filter based reducer is also limited to handle only equality join and acyclic join due to the nature of bloom filters. Despite of these limitations, a bloom filter based semijoin is demonstrated to perform consistently better than purely using semijoin [MOL00][MO98][Mor96]. The advantages and the limitations of a bloom filter based semijoin have motivated a new operator called PERF (Positionally Encoded Record Filters) join.

2.2.5 PERF join and PERF-based algorithms

A new search filter PERF is proposed in [LR95] as a novel two-way join reduction implementation primitive. The basic idea of PERF join came from the desire to minimize the cost of the "backward" reduction in a two-way semijoin, which is done by sending a bit array PERF instead of the original joining attribute. Considering Ri $\bowtie$A Rj, after Rj is reduced to be Rj' by Rj $\bowtie$ Ri [A], a bit array called PERF is generated which contains one bit for every element in the join attribute project Ri [A]. The bit is set to be 1 if that element is in Rj' and 0 if not. The order of this PERF bit array is the same as the order of Ri [A] received from Ri. Instead of sending Rj' [A] back to reduce Ri, this PERF filter is sent back as a compact representation of Rj' [A]. The PERF join operation can be illustrated in Figure 2.7 with 3 steps.

1. Send join attribute projection: Project Ri on joining attribute A and the projection Ri [A] is sent from site i to site j
2. Forward semijoin reduction: Perform $R_i \rtimes R_j$ to reduce Rj to R_j' and construct PERF (Ri) at site j. PERF (Ri) is a bit array of |Ri| bits according to the tuple scan order of Ri. The k^{th} bit of PERF (Ri) is set if and only if the k^{th} tuple of Ri appears in the semijoin result $R_i \rtimes R_j$
3. Backward PERF join reduction: Send PERF (Ri) back to site i to reduce Ri to R_i'. Only the tuples that has matching PERF (Ri) set to 1 will appear in the final join and therefore kept.

PERF join outperforms two-way semijoin due to the following: 1) cheaper transmission cost, as the size of the PEFF array is normally smaller compared with the size of a joining attribute 2) cheaper local operation to do the backward reduction, as the compact PERF bit array might be able to fit into memory, so that the I/O cost to do semijoin on large-sized attributes can be avoided.

PERF join also outperforms Bloom filter based semijoins since it inherits the advantage of the Bloom filter without suffering the side effect of collisions. A PERF has the same storage and transmission efficiency as a Bloom filter, while having the advantage of being able to keep the relation tuple scan order. So it doesn't suffer any loss of join information incurred by hash collisions [LR95].

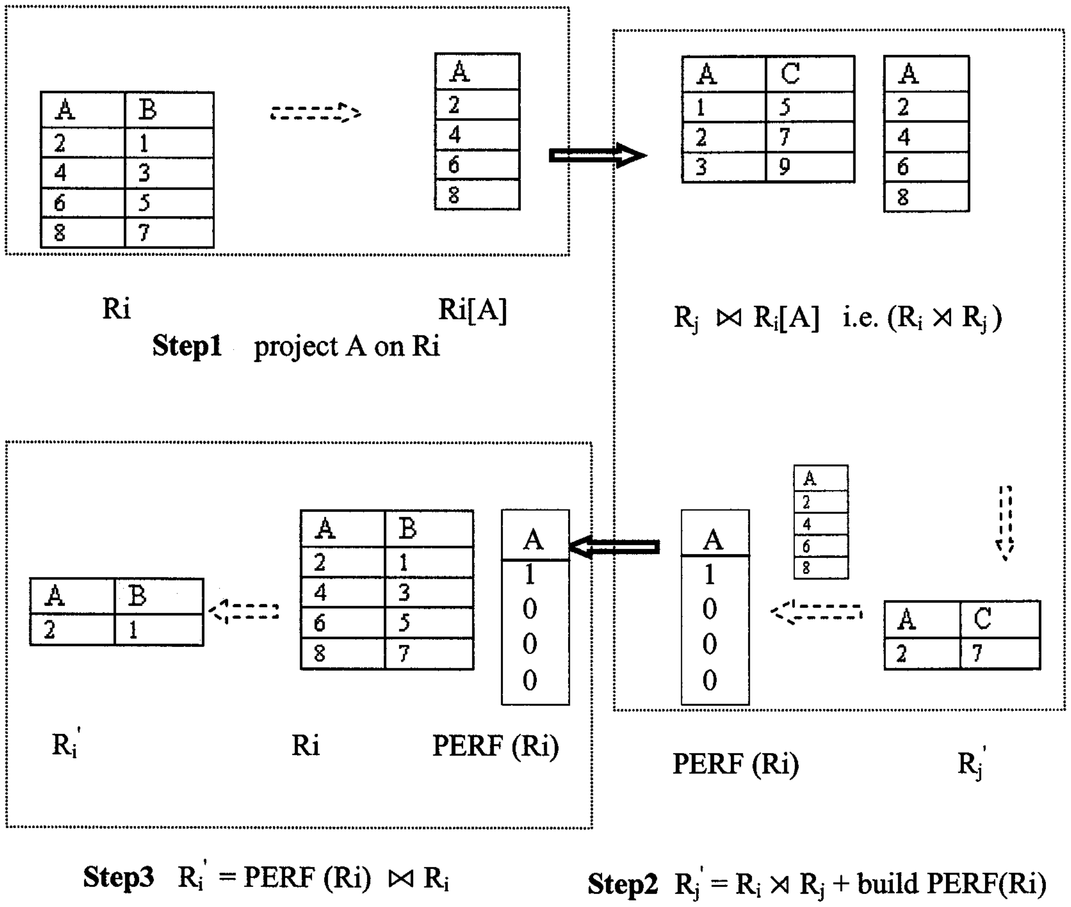

Ri
Ri[A]
Step1 project A on Ri
$R_j \bowtie R_i[A]$ i.e. $(R_i \ltimes R_j)$
R_i'
Ri
PERF (Ri)
PERF (Ri)
R_j'
Step3 $R_i' = \text{PERF (Ri)} \bowtie R_i$
Step2 $R_j' = R_i \ltimes R_j$ + build PERF(Ri)

Figure 2.6: PERF join operation

Research in [HF00] applied PERF joins to the classical semijoin algorithm AHY [AHY83] to generate an algorithm called AHYPERF. The complexity of AHYPERF is still O (σm^2), which is not increased compared with AHY, as data will be scanned in the same way, for the same number of times. Experiments had shown a considerable improvement of AHYPERF over AHY, and PERF join was recommended over both huge width joining attributes and ordinary joining attributes.

Chapter 3 Motivation

3.1 Eliminate duplicates in PERF based algorithms

Recent research has suggested PERF join as a beneficial operator in DQP [LR95][HF00] [Zha03][Zhu04] for its dual benefits: 1) use of bit array filters to save space and transmission cost 2) ability to keep complete join information by avoiding hash collisions. However, since the transmission of join attribute projections to the assembly site is still a necessary step in PERF join, it can be expensive when there are lots of duplicated elements in the join attributes. Intuitively, it will be beneficial if either the duplicates in join attributes or the duplicates in PERF filters can be eliminated efficiently. With the removal of duplicates to avoid sending the redundant data, the transmission time in the forward semijoin reduction phase and backward PERF join reduction phase can be reduced. Sorting is used as a traditional way to remove duplicates in DQP but it is very inefficient.

3.2 The inefficient way to eliminate duplicates via sorting

To eliminate the duplicates, sorting on the join attribute is normally needed, which implies that the join attribute projection sent to do the forward semijoin may not be in the original scan order of the relation. However, to proceed with the backward PERF reduction properly, it is necessary that the site receiving the PERF filters keep the order of relation tuples the same once a join attribute projection has been sent to do the forward semijoin, or be able to map from the "sent" order to the current order, if the order is changed after sending. When duplicates in joining attributes are eliminated for the purpose of reducing transmission time, effort has to be made to make sure that not a single tuple in the relation is lost and the original relation tuple order is maintained to correctly retrieve the tuples for the final join. This problem was first addressed in [LR95] as shown below:

Given a local relation Ri and remote relation Rj, and X is the join attribute on Ri without removing duplicates by its original scan order[1].

[1] The scan order of R is either the physical order of R, or the order according to some index used for the scan

The duplicated elements of X can be eliminated in 4 steps, shown in Figure 3.1:

1) Add to X an extra column Y, which represents the sequence number in projection X.

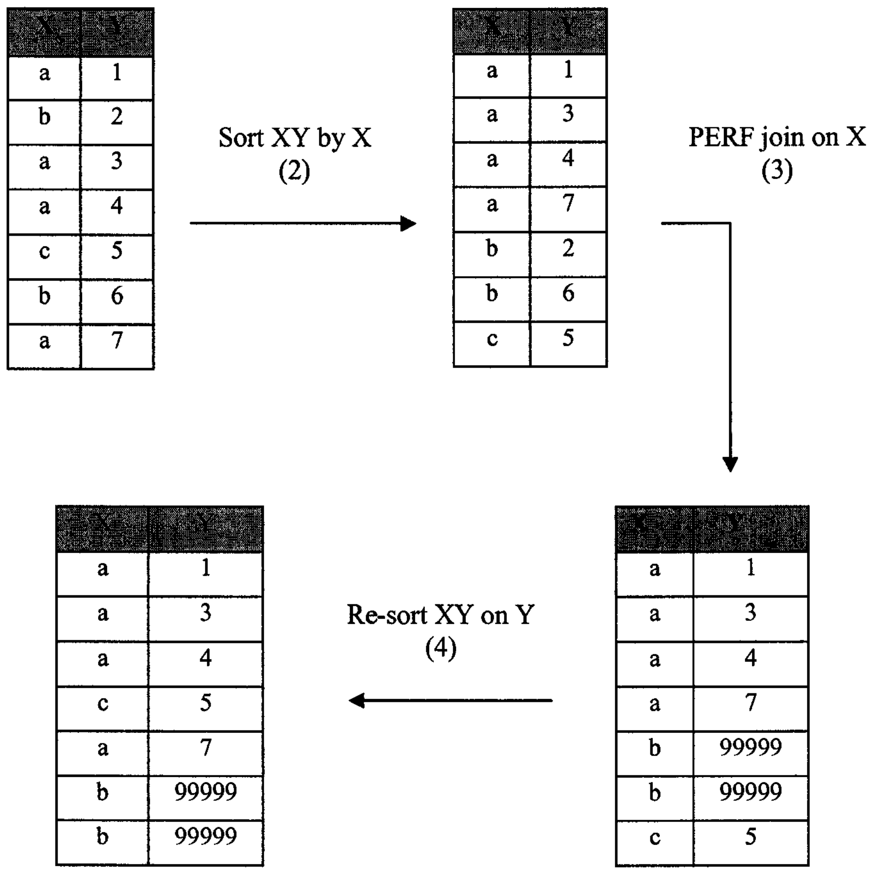
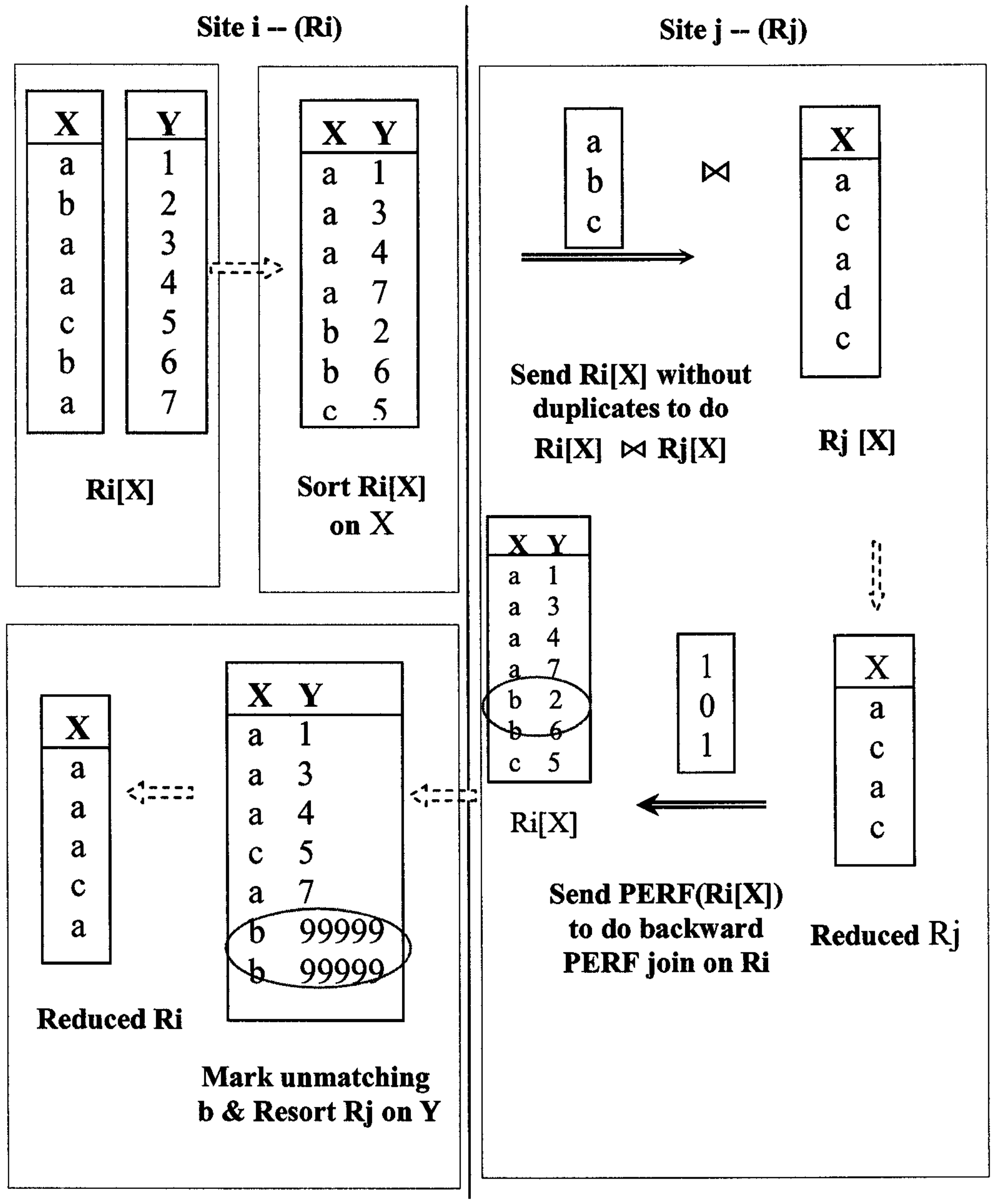


Figure 3.1: Sorting overhead involved when eliminating duplicates

2) Sort XY ordered by the join attribute X.
3) Transmit only the distinct values "a b c" of column X (without transmitting the duplicates and column Y) to Rj to do the forward semijoin. The PERF filter for column X can then be generated after the forward semijoin with Rj and sent back to where Ri resides. In our example, there is no "b" in Rj on the corresponding column X', so the PERF filter for Ri on column X, denoted by PERF (Ri [X]), is set to be "101". After PERF (Ri [X]) is received from Rj back to Ri, make a pass through the sorted XY to filter out all records with a non-matching value "b" in X. Mark these records by setting the corresponding Y value with a larger than any value of Y, for example 99999.

4) Re-sort the XY column on Y, and then the matching tuples are now in the original scan order[2]. Another pass through Ri is needed to retrieve these tuples to be used in the final join.

The whole process is further illustrated in Figure 3.2.



Site i -- (Ri)
Site j -- (Rj)
Ri[X]
Sort Ri[X] on X
Send Ri[X] without duplicates to do Ri[X] ⋈ Rj[X]
Rj [X]
Ri[X]
Send PERF(Ri[X]) to do backward PERF join on Ri
Reduced Rj
Reduced Ri
Mark unmatching b & Resort Rj on Y

Figure 3.2: Eliminate duplicated in PERF join via sorting

[2] The Y column can be looked on as the record # in relation R and it could have been used to retrieve tuples so that step 4 can be omitted. But record # is not always available and sometimes a query has to be processed according to certain key fields

From the above example, in order to eliminate the duplicates of each joining attribute in each relation R of the query, two sorting operations and two passes through R are needed when PERF filter based algorithms are used. It is well known that even the most efficient sorting algorithm has an average time complexity of O (n log n), and the worse case could be $O(n^2)$. Even if the sorting operation can be handled in parallel during the local processing phase, it is still time consuming when the number of joining attributes and the number of relations become very large.

3.3 Motivations for compressed PERF

Inspired by the work from [Mit02] where compressed Bloom filter was proposed to reduce transmission time with a fixed collision rate, we propose compression combined in PERF because:

1. Compression can always be beneficial when the aim is to reduce the amount of data transmitted among distributed sites, since the processing time of compression and decompression can be looked on as local processing time and is therefore treated as negligible.

2. A good compression algorithm is always more efficient than a sorting algorithm. The latter is normally used to eliminate the duplicated tuples. For example, the time complexity of arithmetic coding is O(n) [MNW98] while the most efficient sorting algorithm has the average time complexity of O (n log n).

3. Even if local processing time is considered, compression can be beneficial as long as the benefits of using compression are greater than the compression and decompression overhead.

To summarize, we have the questions and the goals of this thesis listed below:

1.To find out where and how compression can be used in PERF join with a new algorithm PERF_C designed and implemented.

2. To evaluate the cost reduction ratio of algorithm PERF_C, compared to algorithm PERF.

3. When the local cost of compression has to be considered, can we come up a metric to help decide when compression is beneficial?

4. To identify the factors that influence the compression effectiveness.

Chapter 4 Algorithm PERF_C and compression bps

4.1. Algorithm PERF_C: compressed PERF

The proposed algorithm PERF_C applies compression in PERF join (with the combination of composite semijoin) to further reduce the transmission cost, which is denoted to be *Tcosti* in the algorithm. Compression can be used both in the forward reduction phase and in the backward PERF reduction phase. Local Processing Cost involved in algorithm PERF_C is denoted as *Lcost i.*

Algorithm PERF_C:

1. Local processing

For each relation Ri, project on its joining attributes j, j = 1 ..number of join attributes. -- *Lcost1*

Feed joining attributes projections $P_{Ri\ (j)}$ into compressor to get the compressed projections $C\_ P_{Ri\ (j)}$. --*Lcost2*

2. Forward reduction

- Send the compressed joining attribute projections $C\_ P_{Ri\ (j)}$ in parallel to the assembly site; -- *Tcost1*
- Decompress $C\_ P_{Ri\ (j)}$ back into the original $P_{Ri\ (j)}$. -- *Lcost3*
- Perform forward reduction with semijoin and composite semijoin and generate a PERF filter $PERF_{Ri}$ for each relation Ri. -- *Lcost4*

3. Backward PERF join reduction

- Compress $PERF_{Ri}$ filter into C_PERF_{Ri} -- *Lcost5*
- Ship compressed filters $C\_ PERF_{Ri}$ back to Ri --*Tcost2*
- Decompress $C\_ PERF_{Ri}$ back into $PERF_{Ri}$ so that the backward PERF join can be performed to reduce Ri into Ri'. Ri' contains only the matching tuples needed for final join. -- *Lcost6*

4. Final join

Send the reduced Ri' in parallel to the assembling site to do the final join, excluding the joining attributes in Ri'. --*Tcost3*

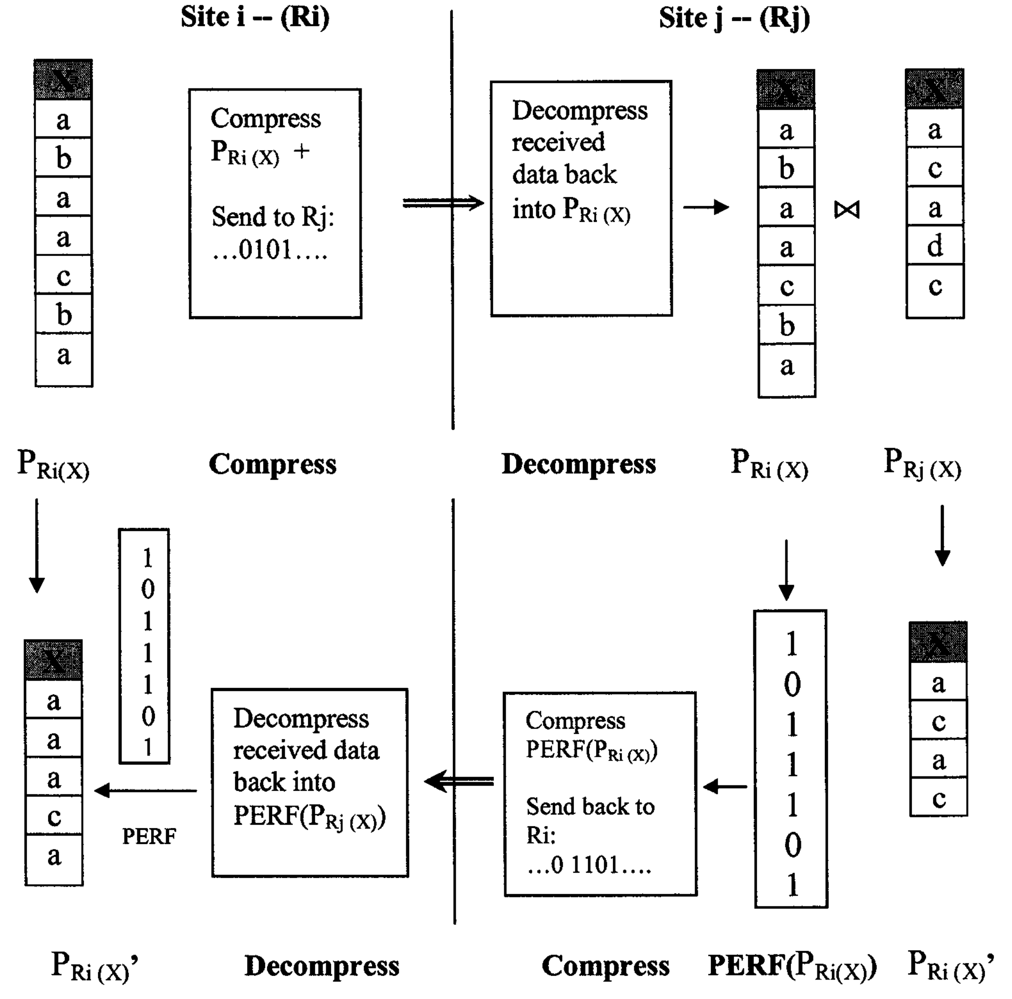
Compared to algorithm PERF, the transmission cost (Σ*Tcosti)* of algorithm PERF_C is reduced by compression with less *Tcost1, Tcost2.* However, local cost (Σ*Lcosti)* increases with *Lcost2, Lcost3, Lcost5* and *Lcost6 added* to do compression and decompression. Using the same example as in Figure 3.1, Algorithm PERF_C on one joining attribute is illustrated in Figure 4.1.

For a distributed query with n relations and m joining attributes, the PERF filters PERF $_{Ri}$ for all the relations Ri (i = 1..n) are built in 3 steps:

1. Generate a PERF filter for every join attribute projection $_X$ on every relation.

We denote these filters as PERF $_{Ri\ (X)}$, X= 1..m, i = 1..n. If composite semijoin is combined, m = (number of join attributes + number of composite semijoin attributes).

2. Create a PERF $_{Ri}$ for each relation Ri.



Site i -- (Ri)
Site j -- (Rj)
X
a
b
a
a
c
b
a
Compress PRi (X) + Send to Rj: ...0101....
Decompress received data back into PRi (X)
⋈
a
c
a
d
c
PRi(X)
Compress
Decompress
PRi (X)
PRj (X)
1 0 1 1 1 0 1
a
a
a
c
a
PERF
Decompress received data back into PERF(PRj (X))
Compress PERF(PRi (X)) Send back to Ri: ...0 1101....
PRi (X)'
Decompress
Compress
PERF(PRi(X))
PRi (X)'

Figure 4.1: Compressed PERF join

1) Start from the relation with the current maximum in-degree. If a relation Ri has more

than one PERF Ri (X) where X = 1..m, apply the "AND" logical operation on all these filters to get a final PERF filter for Ri. i.e. PERF Ri = Π PERF Ri (X) , X = 1 .. m.

2) Update the related PERF $_{Ri\,(X)}$ to refresh the changes made by the creation of PERF $_{Ri}$.

3) Repeat 1) and 2) until all relations have been processed.

The following example in Figure 4.2 demonstrates how algorithm PERF_C works on a distributed query with 3 relations and 4 join attributes. Relation R_1 has three attributes A, B and C with two join attributes B and C shaded. Similarly, there are four attributes B, D, E and F in relation R_2 with B and F as join attributes. F is the join attribute of Relation R_3 out of its two attributes F and G. Algorithm PERF_C chooses the relation with the maximum in-degree as the starting point.

R_1

A	B	C
0	▨	▨
1	▨	▨
2	▨	▨
3	▨	▨

R_2

B	C	D	E	F
▨	▨	▨	Mr	▨
▨	▨	▨	Ms.	▨
▨	▨	▨	Mr.	▨
▨	▨	▨	Ms.	▨

R_3

F	G
▨	F
▨	M
▨	M
▨	T
▨	F

Figure 4.2: Relations R1, R2, R3 (the shaded columns are join attributes)

Algorithm PERF_C proceeds in the following 6 steps in the above example query:

1. With every relation projected on its join attributes, all join attributes' projections are prepared: $P_{R1\,(B)}$, $P_{R1(C)}$, $P_{R2\,(B)}$, $P_{R2(C)}$, $P_{R2\,(F)}$ and $P_{R3\,(F)}$.

2. Apply compression on the above joining attribute projections and send the compressed join attribute projections $C\_P_{R1\,(B)}$, $C\_P_{R1(C)}$, $C\_P_{R2\,(B)}$, $C\_P_{R2(C)}$, $C\_P_{R2\,(F)}$ and $C\_P_{R3\,(F)}$ to the assembly site. Decompress the compressed join attribute projections back into $P_{R1\,(B)}$,

$P_{R1(C)}$, $P_{R2\ (B)}$, $P_{R2(C)}$, $P_{R2\ (F)}$ and $P_{R3\ (F)}$ after the assembly site receives the compressed joining attributes.

3. Build filter PERF $_{Ri}$ for each relation Ri, starting from the relation R_2 with the maximum in-degree 4. (Figure 4.3)

- Generate PERF $_{R2}$, where PERF $_{R2} = \Pi$ PERF $_{Ri}$ (X), $X \in \{B, C, (B, C), F\}$. (See left most frame of Figure 4.3)
- Update related PERF filters after the creation of PERF$_{R2}$. (Figure 4.4)
- Create PERF $_{R1}$ and PERF $_{R3}$ in the similar way, as shown in the middle and right frames of Figure 4.3.

4. Backward compression: compress PERF $_{Ri}$ and the compressed C_PERF Ri are sent back to Ri.

5. Backward reduction: decompress PERF $_{Ri}$ to reduce Ri into Ri'. (Figure 4.5)

6. Final query result obtains with the join operation on Ri'. (Figure 4.6)

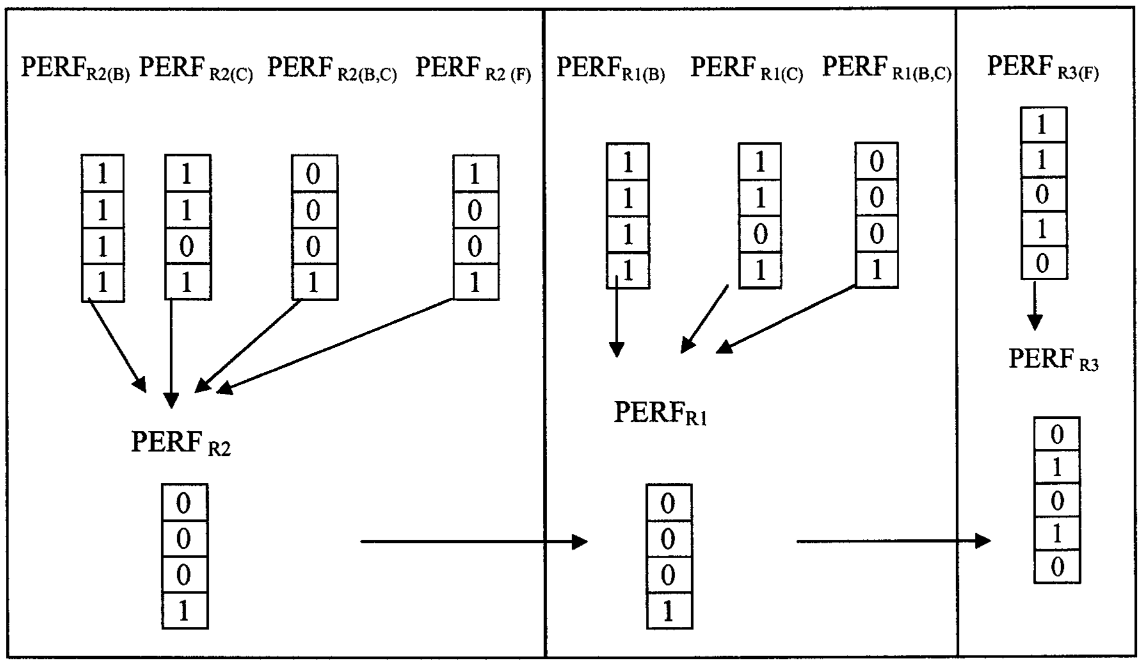


Figure 4.3: Final PERF filters for each relation

4.2 compression bps

When the cost of compressing and decompressing has to be considered, compression is beneficial only when the gain (reduced transmission cost) to use compression outweighs the associated overhead. As described above, the gain of Algorithm PERF_C is the reduced transmission cost Tcost1 and Tcost2 compared to original PERF. However, PERF_C adds the overhead of (Lcost2 + Lcost3 + Lcost5 +Lcost6) to do the compression and decompress. In this thesis, we define "compression bps" as a new metric to show when compression is beneficial.

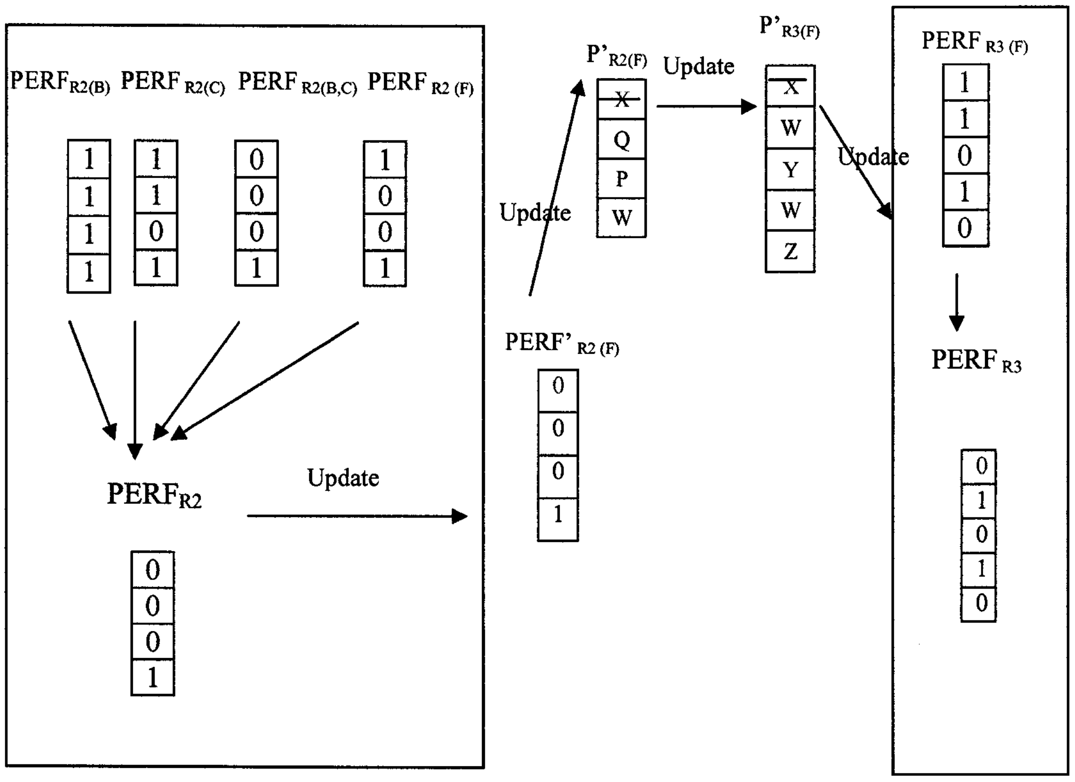


Figure 4.4: Update related PERF filters to refresh changes after PERF R2 is created

R_1'

3	9	6

R_2'

9	6	Y	Ms.	W

R_3'

W	M
W	T

Figure 4.5: Reduced relations Ri'

3	9	6	Y	Ms	W	M
3	9	6	Y	Ms	W	T

Figure 4.6: Final result

4.2.1. Definition 1

$$\text{Compression bps} = \frac{\text{(The number of bits saved by compression)}}{\text{(Time to compress + Time to decompress)}}$$

For example, the size of a join attribute projection P(R) is measured to be 52517 bytes. It takes 0.08 seconds to compress P(R) and decompress back to P(R). The compressed P(R) has 13044 bytes. Then Compression bps = (52517 –13044) *8 /0.08 = 3,947,300 (bit/s)

4.2.2. Theorem 1

Compression can be used as a cost effective operation in distributed query processing if compression bps is greater than data transfer speed. In other words, if compression bps > data transfer speed, then compression is considered beneficial.

Proof

Suppose a projection with size X is to be transmitted to a distributed site through network. By compression, X can be reduced to be X'. We denote the data transfer speed by n (b/s) and the compression bps by c (b/s). According to the condition, we have c > n. The total time spent on compression and decompression is:

$$\text{Tcompress} = (X-X')/c$$

And the time saved by compression due to less data transmitted over network is:

$$\text{Tnetwork} = (X-X')/n$$

Since c > n, we have Tcompress < Tnetwork. That is, the time spent to compress and decompress the same amount of data is less than the time to transmit these data over network, so compression is beneficial.

4.2.3. Theorem 2

Suppose transmission cost is linear with the amount of the transferred data X, i.e. Tnetwork = $C_0 + C_1*X$. As X approaches ∞ or X is big enough so that C_0 can be ignored, compress bps should be greater than $1/C_1$ in order to keep compression beneficial.

Proof

According to the assumption, we have the time spent to transfer X bits of data,

$$\text{Tnetwork} = C_0 + C_1*X \quad \text{------------------------------------}(4.1)$$

From the definition of compression bps, we have the time spent on compression and decompression

$$\text{Tcompress} = X/\text{compression bps} \quad \text{---------------------------------}(4.2)$$

To keep compression beneficial, we must have Tcompress < Tnetwork. According to (4.1) and (4.2), we have:

$$1/\text{compression bps} < C_0/X + C_1$$

As X $\rightarrow \infty$ or X is greater enough so that the startup C_0 can be ignored, we have compression bps > $1/C_1$.

Chapter 5 Implementation and Evaluation

The major goals in this chapter:

1. To implement algorithm PERF_C.
2. To evaluate the cost reduction resulting from compression and compare with algorithm PERF, neglecting compression overhead.
3. To find out if compression can be cost-effective in distributed query processing when compression overhead has to be considered.
4. To implement the measurement of compression bps.
5. To measure compression bps in a wide range of relation size and under different real computing platforms.
6. To compare compression bps measured above with the current distributed system data transfer speed to see if the compression bps is fast enough to be considered advantageous in distributed query processing.

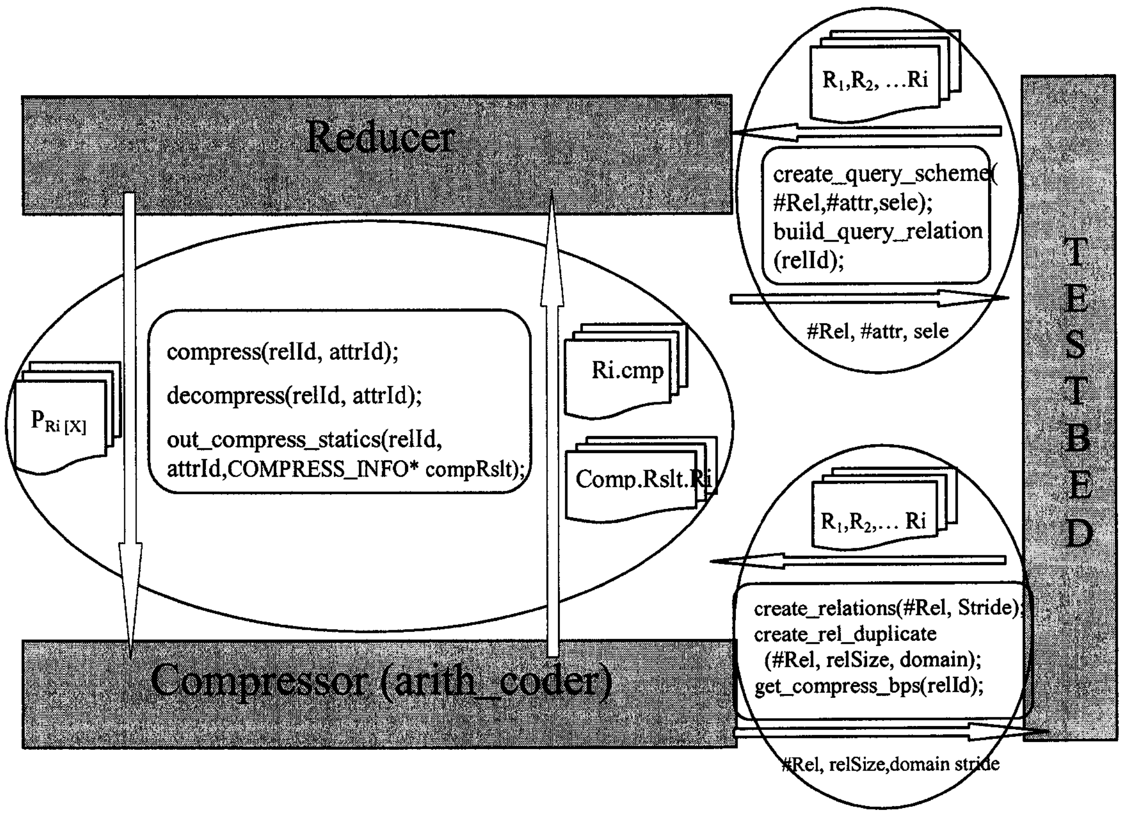
5.1. Implementation

5.1.1. Platform

- GNU C and VC++
- Sun Solaris 9, Linux, Windows

5.1.2. Infrastructure

The implementation of algorithm PERF_C consists of the testbed module, the reducer module, the compressor module and interface reducer-compressor, interface reducer-testbed and interface compressor-testbed, as shown in Figure 5.1.



Reducer
Compressor (arith_coder)
TESTBED
R1,R2, ...Ri
create_query_scheme(#Rel,#attr,sele);
build_query_relation(relId);
#Rel, #attr, sele
compress(relId, attrId);
decompress(relId, attrId);
out_compress_statics(relId, attrId,COMPRESS_INFO* compRslt);
PRi [X]
Ri.cmp
Comp.Rslt.Ri
R1,R2,... Ri
create_relations(#Rel, Stride);
create_rel_duplicate(#Rel, relSize, domain);
get_compress_bps(relId);
#Rel, relSize,domain stride

Figure 5.1: Implement PERF_C: key modules and interface

5.1.3. Testbed module

The testbed module randomly creates distributed queries and relations for the reducer and compressor modules. Large amounts of distributed queries of arbitrary number of relations (3-6), arbitrary number of join attributes (2-4) and selectivity varying from low, medium to high are generated to test the reduction ability of distributed query algorithms such as PERF and PERF_C. First, a query scheme is created to describe the feature of a query that is going to be generated, which describes the number of relations (#Rel), the number of join attributes (#attr) and the domain and selectivity of each join attribute in this query. Then the corresponding relations are generated with tuples randomly selected from the assigned domains according to the query scheme. The queries generated from the testbed are finally fed to selected distributed query algorithms for evaluation. New functionality to generate relations with specific relation size and selectivity is added to the testbed module to measure the compression bps and observe its behavior.

Key concepts

Size and selectivity

For each relation R_i, let $| R_i |$ denote the cardinality of R_i, $S(R_i)$ represent the size of the relation R_i in bytes, and $W(R_i)$ the width of a tuple in R_i in bytes. Then:

$$S(R_i) = | R_i | * W(R_i) \qquad (5.1)$$

The size and selectivity of each individual attribute d_{ij} are represented by $S(d_{ij})$ and $\rho(d_{ij})$ respectively. The width of the join attribute in bits is $W(d_{ij})$. Then:

$$S(d_{ij}) = | d_{ij} | * W(d_{ij}). \qquad (5.2)$$

$\rho(d_{ij})$ is the selectivity on joining attribute j of relation R_i. It is the number of different values occurring in the attributes divided by the number of all possible values of the attribute, also called the domain of d_{ij}. Let $| d_{ij} |$ denote the cardinality of the joining attribute and $D(d_{ij})$ the domain of d_{ij}, the selectivity is represented as

$$\rho(d_{ij}) = | d_{ij} | / D(d_{ij}) \qquad (5.3)$$

When $\rho(d_{ij})$ is small, d_{ij} is called to have a high selectivity. Selectivity in our implementation varies from low [0.7..0.9], medium [0.4..0.7] or high[0.1..0.4].

Figure 5.2 gives the statistical information of a query of four relations and 2 join attributes created in our testbed. In this example, we have $D(d_{i1}) = 990$ and $D(d_{i2}) = 610$. Note that only R_4 has both join attributes. The selectivity of the join attribute projection of R_1 is calculated as follows:

$$\begin{aligned}\rho(d_{12}) &= | d_{12} | / D(d_{12}) \\ &= 435/610 \\ &= 0.713115\end{aligned}$$

Relation	S (R_i)	S (d_{i1})	ρ (d_{i1})	S (d_{i2})	ρ (d_{i2})
R_1	4800	0	0.000000	435	0. 713115
R_2	1900	945	0. 954545	0	0.000000
R_3	1700	0	0.000000	525	0.860656
R_4	3300	825	0.833333	565	0.926230

Figure 5.2: Database statistical information

Cost and Benefit

For algorithm PERF, the total transmission cost is the sum of the reduced relations Ri' (after applying PERF to reduce Ri), the size of the join attribute projections and the size of PERF filters. The cost of PERF_C is smaller than PERF with the reduced size of compressed join attributes and compressed PERF filters.

$$C\ (PERF) = \sum S\ (R_i') + S\ (Projections) + S\ (Filters)\quad (i = 1\ldots n) \qquad (5.4)$$

$$C\ (PERF\_C) = \sum S\ (R_i') + S\ (compressed\ Projections) + S\ (compressed\ Filters)\quad (i = 1\ldots n) \qquad (5.4)'$$

We call distributed query processing without any reducer the IFS (Initial Feasible Solution), the cost of IFS is

$$C\ (IFS) = \sum S\ (R_i) \qquad (i = 1..\ n)$$

The benefit of PERF_C or PERF is the difference between the size of the original relations and the size of the reduced relations.

$$B\ (PERF\_C) = \sum (S\ (R_i) - S\ (R_i')) \qquad (i = 1\ldots n) \qquad (5.5)$$

$$B\ (PERF) = \sum (S\ (R_i) - S\ (R_i')) \qquad (i = 1\ldots n) \qquad (5.5)'$$

The benefit ratio (or reduction ratio) is the benefit over the size of the original relations.

$$BR\ (PERF) = B\ (PERF) / \sum S\ (R_i) \qquad (i = 1\ldots n) \qquad (5.6)$$

$$BR\ (PERF\_C) = B\ (PERF\_C) / \sum S\ (R_i) \qquad (i = 1\ldots n) \qquad (5.6)'$$

If the benefit exceeds the cost, the algorithm is called cost-effective.

The cost reduction ratio is the reduced cost (compared to CIF), over the size of the original relations.

$$\text{CRR (PERF)} = (\text{C (CIF)} - \text{C (PERF)}) / \text{C (CIF)}$$

$$= (\sum S(R_i) - \sum S(R_i') - S(\text{Projections}) - S(\text{Filters})) / \sum S(R_i) \qquad (i = 1\ldots n) \qquad (5.7)$$

$$\text{CRR (PERF\_C)} = (\text{C (CIF)} - \text{C (PERF\_C)}) / \text{C (CIF)}$$

$$= (\sum S(R_i) - \sum S(R_i') - S(\text{compressed Projections}) - S(\text{compressed Filters})) / \sum S(R_i)$$

$$(i = 1\ldots n) \qquad (5.7')$$

Interface to reducer module

1.create_query_scheme (#Rel, #attr, sele)

Create distributed query schemes according to the user desired number of relations *#Rel*, number of join attributes *#attr* and selectivity level *sele*. The query schemes, which include the size of each relation Ri, the domain and selectivity for each join attribute of each relation Ri in the query, are randomly picked within the legal range and stored in a temp file.

2. build_query_relation(relId)

Build each relation in the distributed query by generating its distinct relation tuples according to the query scheme for each relation passed from the above function. Relations created in this query are used to test the cost and benefit for both PERF and PERF_C algorithms.

Interface to compressor module

1. create_relations (#Rel, stride)

Generate a series of relations R_1, R_2, ---$R\#_{Rel}$ with the relation size increased a ***stride*** in between. These relations are fed into compressor to test the compression bps.

2. create_rel_duplicate(#Rel, relSize, domain);

Create one-join-attribute relations Ri, i = (0.. #Rel) with the same relation size of *relSize*, the same join attribute domain of *domain,* but a different selectivity of (1/*#Rel*) *i. The selectivity of Ri varies from 0 to 1 with a difference of (1/*#Rel*) in between. For example, create_rel_duplicate (7, 9000, 10000) will generate 7 relations with the same relation size of 9000, the same domain of 10000, but a selectivity of Ri to be (1/7 * i). In this way, Ri represents different levels of duplication with R0 the highest and R6 lowest.

3. get_compress_bps (relId);

Calculate and output the compression bps when compressing the relation of *relId.*

The compression bps is measured as follows (with an iteration of *loop* times):

1) Call create_relations () or create_rel_duplicate () to generate a series of relations Ri,

   i =1..*Rel#*

2) Call get_compress_bps () to compress Ri and obtain the compression bps c_bps [i] for $R_{i,}$ i = (1.. *#Rel*)
3) Repeat 1)—2) *loop* times
4) Calculate the average c_bps [i] after repeating *loop* times

5.1.4. Reducer module

Algorithm PERF_C and algorithm PERF are implemented here as the reducer. The reducer module will first call create_query_scheme() and build_query_relation() to generate numerous distributed queries for every query type, and then apply PERF join and compressed PERF join to reduce the transmission cost. The cost, reduced cost and cost reduction ratio are finally calculated for both algorithm PERF and algorithm PERF_C.

Key data structure

1. temp_max

To record the relation name and number of join attributes of the current maximum in-degree relation in the query. Distributed query processing by PERF_C chooses the maximum in-degree relation as the starting point to process, then the second maximum in-degree relation, and so on, until all the relations in the query have been processed.

```
typedef struct
 {
            int rel;  //relation Id
            int no_join_attr;  //number of join attributes
} MAX_IN_DEGREE;
MAX_IN_DEGREE temp_max;
```

2. projection[MAXREL][MAXROW][MAXATTR]

To store the projections for each join attributes of each relation in the query.

3. filters[MAXREL][MAXROW][MAXATTR]

To store the PERF filter PERF $_{Ri\,(X)}$ created for each join attribute projection P $_{Ri\,(X)}$.

4. filter[MAXREL][MAXROW]

To store the PERF filter PERF $_{Ri}$ created for each relation Ri.

Key Functions

1. scan_query_data ()

It belongs to the local processing phase of the distributed query processing. It reads in the query information such as #Rel, #attr and selectivity level. Then scans relations and projects on its join attributes.

2. build_filter ()

Creates PERF filters for join attribute projections.

3. find_max ()

Finds the current max in-degree relation as the starting point to process.

4. when_to_end ()

Terminates query processing when all the relations in the query have been processed.

5. filter_composite ()

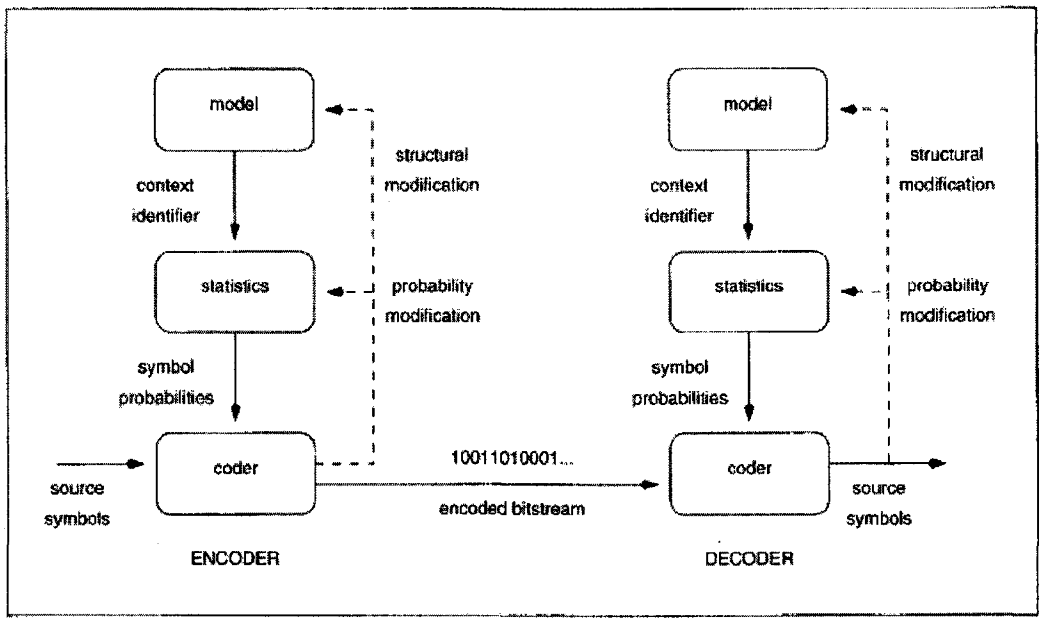
Creates PERF filter for each relation.

6. statistic()

Calculates the gain, cost and reduction ratio for algorithm PERF_C or PERF.

5.1.5. Compressor module

The compressor module is implemented based on Moffat, Neal and Witten's research of the compression method arithmetic coding [MNW98], where a modular structure was proposed to separate the coding, modeling and probability estimation components in a compression system, as shown in figure 5.3. We modified the input interface and the output interface of the coder module to suit our needs. To interface with the Reducer module, functions to compress and decompress join attribute projections for each relation in the query and to output the compressed projections and the compression statistic information to the Reducer module are added. To interface with the Testbed module, functions to compress the relations created in the testbed and to output the compression result are provided.



(Courtesy [MNW98] Fig.1.)

Figure 5.3: Modeling, statistics, and coder modules

5.1.6. Interface between PERF module and the compressor module

The join attributes projections P_{Ri} [X] in each relation are gathered together at the reducer module and passed to the compressor module, where P_{Ri} [X] are compressed into Ri.cmp with smaller size. Passed back to the reducer module, Ri.cmp is then sent to the assembly site instead of P_{Ri} [X] to reduce the transmission cost. The compression statistic information such as the compression ratio, the compression time and the compression bps are extracted from the compressor module and passed to the reducer module in the form of a structure "compressRslt.Ri", which is used in the reducer module to calculate the overall cost reduction ratio for algorithm PERF_C.

Key data structure

- COMPRESS_INFO:

This is the interface structure between the reducer module and the compressor module. It is created in the compressor module and passed on to the reducer module to calculate the cost reduction ratio of algorithm PERF_C.

```
typedef struct{
double  bytes_before_compress;
Double bytes_after_compress;
double  compress_ratio;
 //(bytes_before_compress-bytes_after_compress)/ bytes_before_compress
double compress_time;
//elapse time spent on compression
double compress_bps;
//(bytes_before_compress-bytes_after_compress)/ compress_time
}COMPRESS_INFO;
COMPRES_INFO compress_rslt;
```

Key functions

*1. **compress()-*** To compress the join attributes projections / or its corresponding PERF filters in each relation Ri, and to output the compression statistic information in "compRslt.Ri" and the compression result "Ri.cmp".

*2. **decompress()***- To decompress Ri.cmp back into the original join attribute projections / or PERF filters to be used to do PERF join.

*3. **out_compress_statics()-*** To output the statistics of the compression result such as "bytes before compression, bytes after compression, time spent on compression, compress ratio, and compress bps".

5.2. Evaluation of algorithm PERF_C

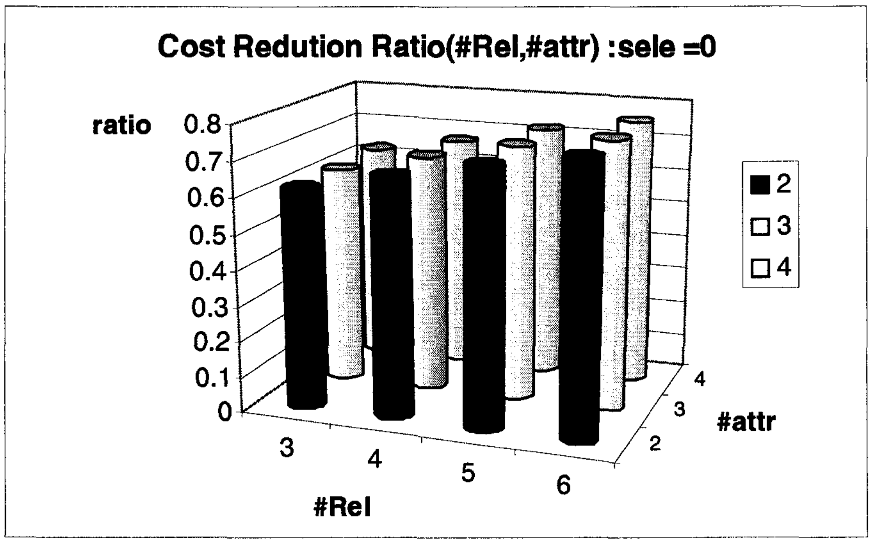
Algorithm PERF_C is implemented with arithmetic coding as the compressor. To be consistent with former research, the evaluation of algorithm PERF_C inherited the use of the synthetic generated distributed queries with 3-6 relations, 2-4 join attributes and 3 levels of selectivity (0 for low, 1 for medium and 2 for high). These distributed queries are generated based on every combination of {(3, 4, 5, 6), (2, 3, 4), (0, 1, 2)}, which results in 36 different types of distributed query, as shown in the left most column in Figure 5.4. To keep the variation of the experiment results acceptable, 500 test runs are performed on each query type to evaluate algorithm PERF_C and algorithm PERF, which results in a total (500 * 36) test runs each for algorithm PERF_C and algorithm PERF. CRR (PERF_C) and CRR (PERF) are calculated by averaging the cost reduction ratios of the 500 runs and they are compared based on the exact same set of distributed queries. The difference between CRR (PERF) and CRR (PERF_C) is looked on as the benefit of compression in DQP. The original experiment results on PERF and PERF_C, with 500 queries generated for every query type from the testbed, is listed in Figure 5.4.

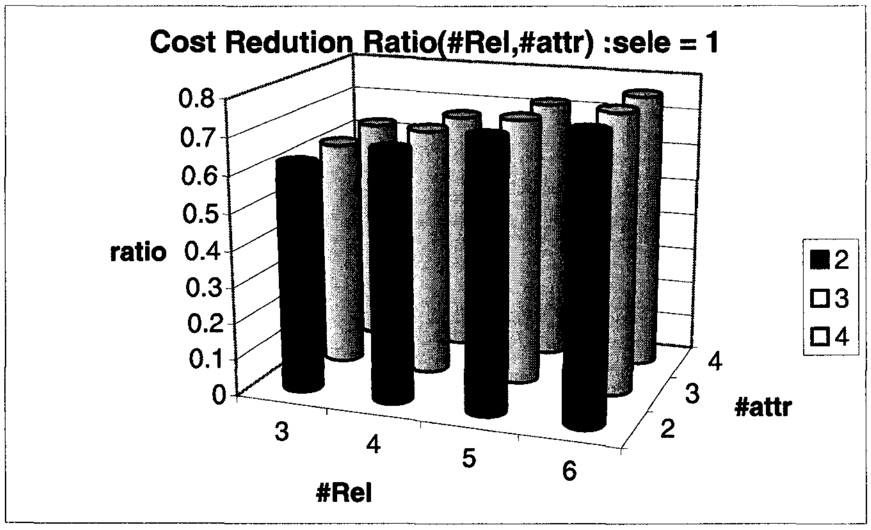
5.2.1. Characteristic of algorithm PERF_C

We evaluate the characteristic of the proposed algorithm PERF_C by observing how its cost reduction ratio (CRR) is related to #Rel, #attr and their selectivity level.

Query Type	# Rel	# attr	selectivity	RelSize (B)	CostReduced (PERF)	CostReduced (PERF_C)	CRR % (PERF)	CRR % (PERF_C)
3-2-0	3	2	0	7855.49	2923.42	4896.24	36.51	61.91
3-3-0	3	3	0	7429.16	2753.56	4623.8	36.26	61.75
3-4-0	3	4	0	7238.91	2783.89	4565.9	37.73	62.64
3-2-1	3	2	1	7954	3005.83	4985.1	37.42	62.45
3-3-1	3	3	1	7692.37	2894.85	4813.86	37.02	62.21
3-4-1	3	4	1	7456.16	2818.37	4673.49	37.74	62.64
3-2-2	3	2	2	8237.86	3169.75	5197	38.48	63.09
3-3-2	3	3	2	8303.91	3231.89	5260.7	38.56	63.14
3-4-2	3	4	2	7949.39	3191.43	5094.61	39.64	63.78
4-2-0	4	2	0	10588.2	4900.8	7175.76	45.53	67.32
4-3-0	4	3	0	10663.73	4979.36	7253.11	45.62	67.37
4-4-0	4	4	0	10465.8	4757.8	7041	44.74	66.85
4-2-1	4	2	1	10858.4	5113.8	7411.64	46.35	67.81
4-3-1	4	3	1	11185.6	5283.8	7644.52	46.48	67.89
4-4-1	4	4	1	10802.61	5152.51	7412.55	47.14	68.29
4-2-2	4	2	2	11283	5565.2	7852.32	48.84	69.3
4-3-2	4	3	2	11556.31	5654.71	8015.35	48.45	69.07
4-4-2	4	4	2	11621.4	5697.4	8067	48.62	69.17
5-2-0	5	2	0	12978.4	7009.6	9397.12	53.3	71.98
5-3-0	5	3	0	14071.4	7660.2	10224.68	53.68	72.21
5-4-0	5	4	0	14188	7699	10294.6	53.3	71.98
5-2-1	5	2	1	13052.2	7311.6	9607.84	55.35	73.21
5-3-1	5	3	1	14108.4	7788.8	10316.64	54.54	72.72
5-4-1	5	4	1	14452.8	8041.4	10605.96	55.08	73.05
5-2-2	5	2	2	14155.2	8175.6	10567.44	57.39	74.43
5-3-2	5	3	2	15179.8	8579.8	11219.8	56.23	73.74
5-4-2	5	4	2	15064.4	8736.4	11267.6	57.68	74.61
6-2-0	6	2	0	14976	9074.4	11435.04	59.94	75.96
6-3-0	6	3	0	16421.4	9860.2	12484.68	59.44	75.66
6-4-0	6	4	0	17090.2	10377.2	13062.4	59.96	75.97
6-2-1	6	2	1	15455.4	9432.6	11841.72	60.55	76.33
6-3-1	6	3	1	16903.8	10356.2	12975.24	60.7	76.42
6-4-1	6	4	1	17431.4	10662.8	13370.24	60.6	76.36
6-2-2	6	2	2	16714.2	10608.6	13050.84	63.43	78.06
6-3-2	6	3	2	17587.8	10913.8	13583.4	61.86	77.12
6-4-2	6	4	2	18204.4	11407	14125.96	62.29	77.38

Figure 5.4: Experiment result of PERF and PERF_C (500 runs)





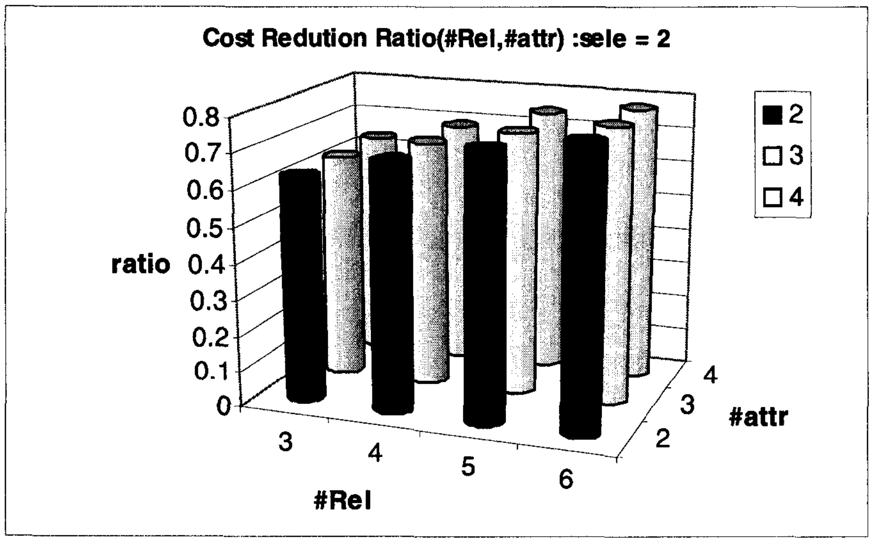


Figure 5.5: Cost reduction ratio of PERF_C

From the above graph, we can see that PERF_C effectively reduced transmission cost with a CRR more than 60%. Higher CRR achieves on queries with higher # Rel under every selectivity level. In general, CRR increases with the increasing of #attr, though the difference of CRR is not big. Selectivity level and #attr place less important roles on CRR than #Rel in PERF_C, as the maximum CRR difference of different selectivity level/#attr is less than 2%. It makes PERF_C extremely useful in distributed queries which consist of low-selectivity join attributes as low-selectivity normally leads to a poor cost reduction ratio in DQP.

5.2.2. PERF_C vs. PERF

Figure 5. 6 and Figure 5.7 summarize the comparison of PERF and PERF_C in terms of cost reduction ratio.

Cost Reduction Ratio (%)		Number of Join Relations											
		3						4					
		High		Med		Low		High		Med		Low	
		PERF	PERF_C	PERF	PERF_C	PERF	PERF_C	PERF	PERF_C	PERF	PERF_C	PERF	PERF_C
Number of Attributes	2	36.51	61.91	37.42	62.5	38.48	63.09	45.53	67.32	46.35	67.81	48.84	69.3
	3	36.26	61.75	37.02	62.2	38.56	63.14	45.62	67.37	46.48	67.89	48.45	69.07
	4	37.73	62.64	37.74	62.6	39.64	63.78	44.74	66.85	47.14	68.29	48.62	69.17
Average		36.83	62.10	37.39	62.43	38.89	63.34	45.30	67.18	46.66	68.00	48.64	69.18
Cost Reduction Ratio (%)		5						6					
		High		Med		Low		High		Med		Low	
		PERF	PERF_C	PERF	PERF_C	PERF	PERF_C	PERF	PERF_C	PERF	PERF_C	PERF	PERF_C
Number of Attributes	2	53.3	71.98	55.35	73.2	57.39	74.43	59.94	75.96	60.55	76.33	63.43	78.06
	3	53.68	72.21	54.54	72.7	56.23	73.74	59.44	75.66	60.7	76.42	61.86	77.12
	4	53.3	71.98	55.08	73.1	57.68	74.61	59.96	75.97	60.6	76.36	62.29	77.38
Average		53.43	72.06	54.99	72.99	57.10	74.26	59.78	75.86	60.62	76.37	62.53	77.52

Figure 5.6: Cost reduction ratio of PERF_C and PERF

We conclude our comparison of PERF and PERF_C as follows:

1. It is effective to use PERF join and compressed PERF join in DQP to reduce

transmission cost. The transmission cost is reduced significantly with both PERF and PERF_C reducer in contrast to IFS. PERF reducer comes up with the cost reduction ratio between 36%-62% and PERF_C between 61.9% and 78.06% (in comparison with IFS).

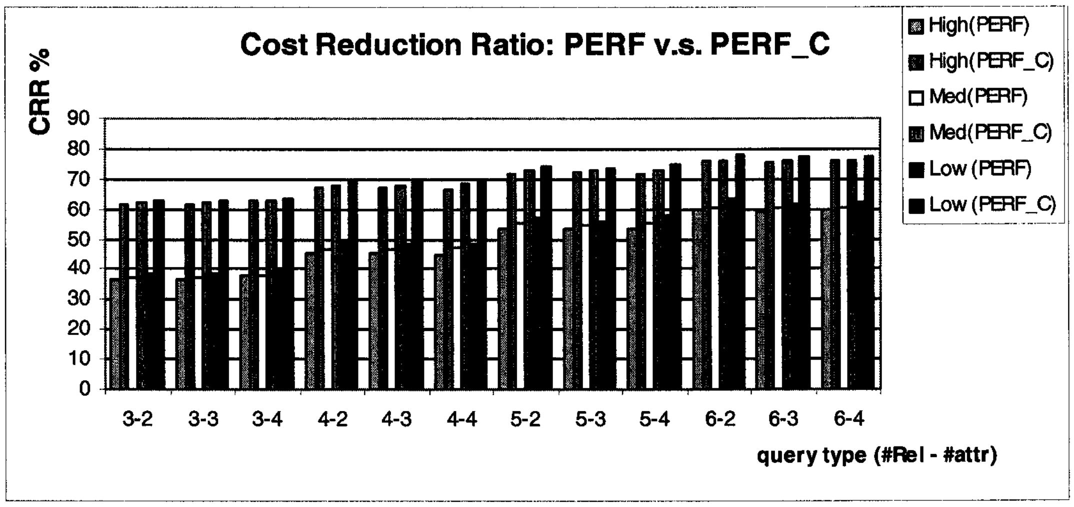


Figure 5.7: Compare the cost reduction ratio of PERF_C and PERF

2. Algorithm PERF_C outperforms algorithm PERF with an overall 16-36% more cost reduction brought to PERF via compression. The combination of PERF join and compression makes algorithm PERF_C a powerful method to reduce transmission cost.
3. For both PERF and PERF_C, the performance difference between low-selectivity join attributes and high-selectivity join attributes is not obvious, which makes PERF and PERF_C valuable reducers for queries with low or medium selectivity.
4. The cost reduction ratio of PERF and PERF_C, CRR (PERF_C) and CRR (PERF), both increase with the growth of the number of relation (#Rel) and the growth of the number of join attribute (#attr). However, the performance gap between CRR (PERF_C) and CRR (PERF) is getting smaller when #Rel and #attr grows bigger. This implies that the cost reduction via compression grows slower than the growth of the cost reduction via PERF join, when the #Rel and #attr increase. This feature also motivated the following evaluation on compression bps: What are the factors to influence compression bps? Does compression bps always grows when the data size to be compressed grows?

5.3. Evaluation of compression bps

The questions to be answered through the evaluation:

1. How does compression bps behave during the distributed query processing with 36 different types of queries generated in testbed?
2. How does compression bps change according to the database factors: #Rel, #attr, the selectivity level and duplication level of join attributes?
3. How much can we get for the compression bps in the real world? Is the compression bps fast enough that compression can still be used in DQP, even when the cost of compression cannot be neglected?

We designed the following tests to evaluate compression bps under different circumstances:

1. Test compression bps under 36 types of distributed queries created in our testbed to evaluate the relationship between compression bps and #Rel, #attr and selectivity.
2. Test the compression bps on specially created relations with different levels of duplication.
3. Calculate compression bps under real computing environment with different types of platforms to see if compression is still beneficial (or when compression is beneficial) in DQP even if the cost of compression is considered.

Each test is performed 500 times and the average of these 500 runs is reported as the result.

5.3.1. Compression bps under 36 types of Distributed Queries

Figure 5.8 details the output of our test on compression bps under 36 different types of queries. And Figure 5.9 illustrates how the database factors (#Rel, #attr and the selectivity level) in the distributed query influence compression bps.

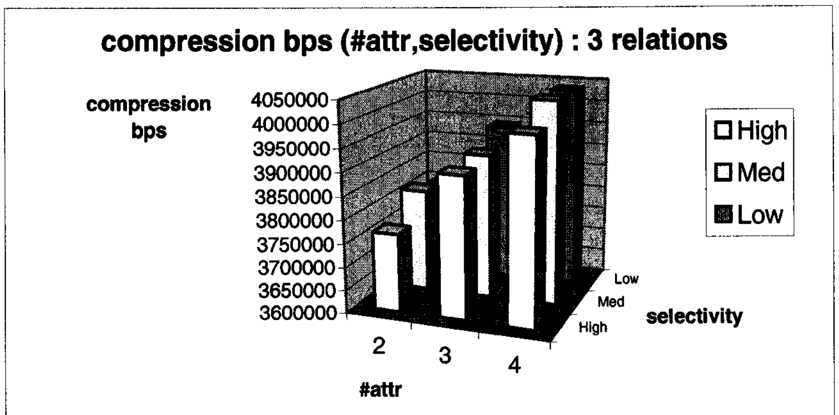
It is shown in Figure 5.9:

1. For queries with #Rel of 3, 4, 5 and 6, compression bps increases as #attr grows, regardless of the selectivity level in the query. It is understandable, since #attr is associated with the data size to be compressed. This suggests that the size of the data to be compressed is an important factor for compression bps.
2. The effect of selectivity on compression bps cannot be clearly identified with the test of 36 types of distributed queries, as the test result does not follow a regular

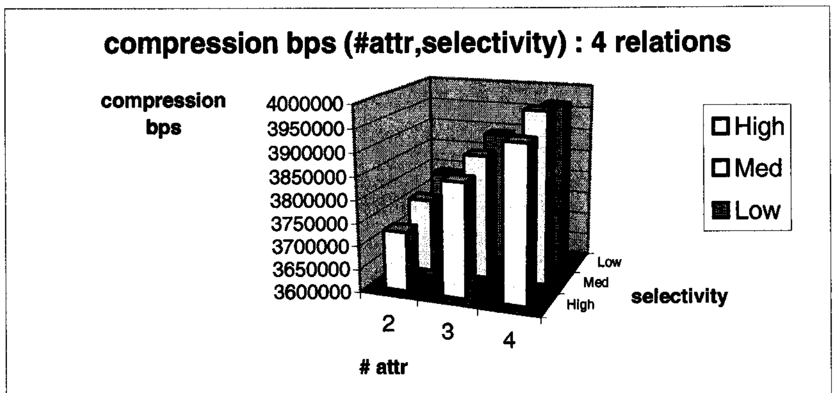
pattern. For example, for #Rel =3, medium selectivity gives the highest compression bps at #attr=2 or 4. However, the highest compression bps happens at lowest selectivity when #attr=3. Inconsistency happens at #Rel=4, #Rel=5 also. The evaluation on selectivity using the 36 types of queries is not sufficient, as the sizes of relations in each query are randomly picked every time a new query is generated.

Query Type	Before Compress(B)	After Compress(B)	compress Ratio	compress Time(s)	compression bps (b/s)
3-2-0	45674	16191	0.6455	0.062582	3769100.227
3-3-0	54876	18958	0.654	0.073209	3905210.37
3-4-0	65843	22313	0.6606	0.086333	3998254.917
3-2-1	46055	16299	0.6461	0.06248	3821381.572
3-3-1	57220	19755	0.6541	0.076233	3912634.265
3-4-1	67766	22962	0.6607	0.08842	4039398.839
3-2-2	47751	16923	0.6454	0.065	3800373.457
3-3-2	62095	21438	0.654	0.0821	3943669.204
3-4-2	73351	24830	0.6605	0.095693	4036128.215
4-2-0	44293	15779	0.6439	0.060755	3729324.363
4-3-0	53788	18770	0.6501	0.07207	3848969.223
4-4-0	62601	21488	0.6558	0.082605	3939531.386
4-2-1	45888	16334	0.6441	0.062655	3762248.729
4-3-1	56839	19829	0.6501	0.075975	3872830.027
4-4-1	65303	22393	0.6559	0.085825	3977494.642
4-2-2	47557	16924	0.644	0.06459	3789134.127
4-3-2	58263	20320	0.6501	0.077575	3891592.541
4-4-2	70507	24176	0.6556	0.09269	3966328.551
5-2-0	42847	15291	0.6434	0.058876	3727777.569
5-3-0	54675	19171	0.6484	0.07328	3837789.149
5-4-0	64342	22217	0.6534	0.08492	3927034.87
5-2-1	43272	15425	0.6437	0.059776	3724157.236
5-3-1	55053	19299	0.6486	0.073946	3847165.2
5-4-1	65362	22555	0.6537	0.086236	3935414.088
5-2-2	46999	16755	0.6433	0.064212	3767522.913
5-3-2	59078	20711	0.6483	0.07972	3829592.028
5-4-2	68872	23745	0.6535	0.091064	3930450.469
6-2-0	41338	14729	0.6438	0.05712	3716990.571
6-3-0	53238	18655	0.6487	0.071944	3797497.373
6-4-0	63889	22081	0.6533	0.084882	3889973.731
6-2-1	42480	15143	0.6436	0.058567	3726761.056
6-3-1	55036	19280	0.6487	0.074401	3815735.826
6-4-1	65121	22492	0.6532	0.08628	3916278.26
6-2-2	46422	16535	0.6436	0.063797	3752378.633
6-3-2	56532	19826	0.6482	0.076073	3835177.794
6-4-2	68936	23788	0.6533	0.090857	3946745.984

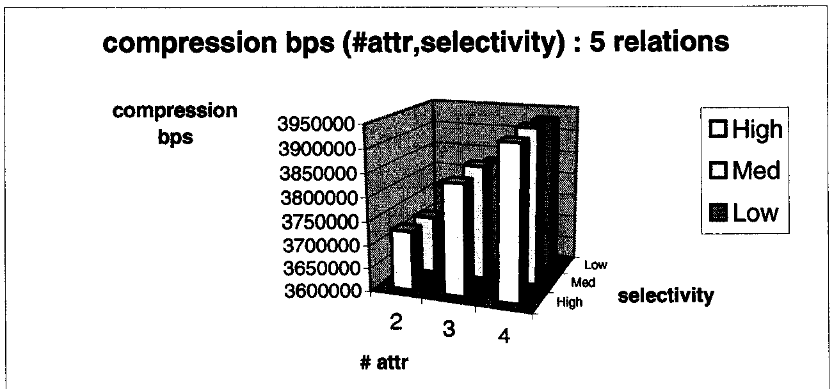
Figure 5.8: Compression bps on 36 types of distributed queried (500 runs)



	2	3	4
□High	3769100.227	3905210.37	3998254.917
□Med	3821381.572	3912634.265	4039398.839
■Low	3800373.457	3943669.204	4036128.215

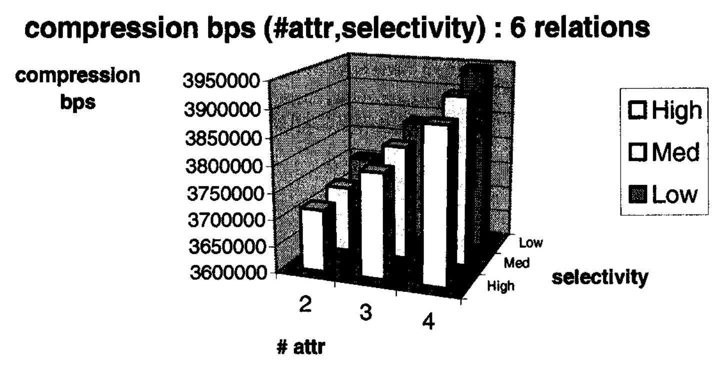


	2	3	4
□High	3729324.363	3848969.223	3939531.386
□Med	3762248.729	3872830.027	3977494.642
■Low	3789134.127	3891592.541	3966328.551



	2	3	4
□High	3727777.569	3837789.149	3927034.87
□Med	3724157.236	3847165.2	3935414.088
■Low	3767522.913	3829592.028	3930450.469

Figure 5.9: Compression bps: effects of Rel, attr and selectivity



	2	3	4
□ High	3716990.571	3797497.373	3889973.731
□ Med	3726761.056	3815735.826	3916278.26
■ Low	3752378.633	3835177.794	3946745.984

Figure 5.9: Compression bps: effects of #Rel, #attr and selectivity (Continued)

In response to the concerns in 1, 2 above, relations with controllable relation sizes and controllable selectivity levels should be generated in order to test how compression bps is influenced by the size of data to be compressed and the duplication of the data.

5.3.2. Compression bps with join attribute duplications

The duplication level of a join attribute can be represented by relation size, domain and selectivity of the join attribute altogether. For the same relation size and same domain, high selectivity level results in high duplication level and low selectivity low duplication level. Based on this analysis, a special test was designed to create a series of one-join-attribute relations with the same relation size and same join attribute domain but different levels of duplications by evenly varying the selectivity from 0 to 1. Under this circumstance, the higher the selectivity, the higher the duplication level. As in the example shown in Figure 5.10 with the column of # of tuples =2000, relations R0- R6 are created to have the same relation size 2000 and the same join attribute domain 8000. However, the selectivity of Ri will be (1/7* i), with a difference of 0.17 between Ri and R_{i+1}, i = 0..5. There are 2 extremes: R0 has the highest duplication level (or the highest selectivity level) with all its tuples are duplicated, and R6 has the lowest duplication level with no duplicates at all. In Figure 5.10, seven relations R0-R6 are created for each relation size, which can be picked up from {2000, 3000, 4000, 5000, 6000, 7000, 8000} with the domain of 8000. Compression bps is tested on R0-R6, and the effects of duplication and # of tuples are illustrated in Figure 5.11.

Duplicate Level	**# of tuples**						
- Selectivity	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**
R0 - 0	9307878	10530812	11271774	12034603	12273688	12392519	12598255
R1- 0.17	4683660	5375029	5901417	6281398	6624087	6613446	6867911
R2 - 0.33	4042043	4676084	5203301	5660991	5775503	6087033	6057909
R3 - 0.50	3529782	4255035	4716034	5135605	5376294	5568856	5778102
R4 - 0.67	3084163	3810601	4327149	4719396	4956336	5226763	5347795
R5 - 0.83	2759174	3487304	4044189	4462222	4578408	4891671	5073682
R6 - 1	2583843	3268305	3734724	4146440	4427159	4636466	4623043

Figure 5.10: Compression bps with different duplication levels and # of tuples

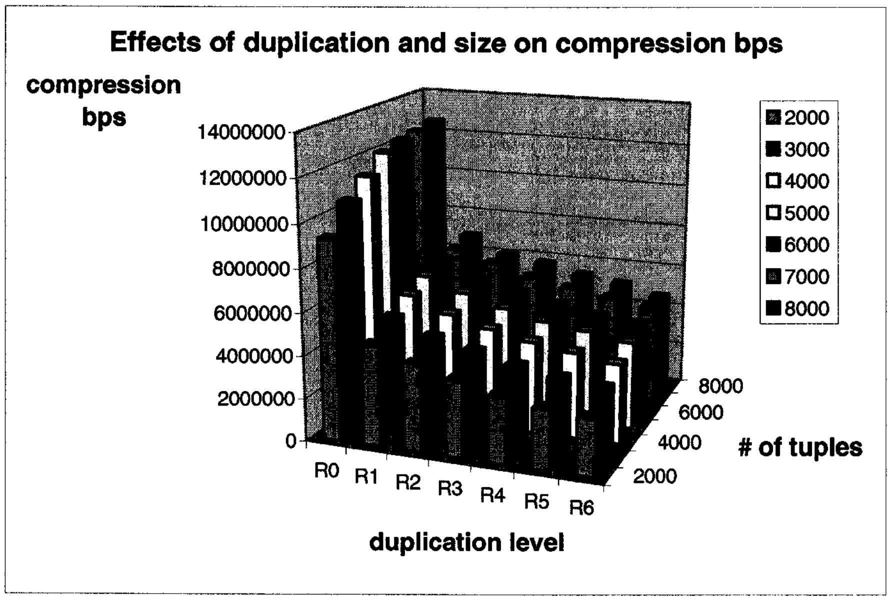


Figure 5.11: Effect of duplication level and # of tuples on compression bps

We conclude from above:

1. Higher compression bps is achieved at higher duplication level, which is consistent with our analysis in Chapter 4. So for the joining attributes in the same relation (these join attributes have the same # of tuples), the attributes with higher duplication (or higher selectivity) level compress better. As shown in Figure 5.11, for the same relation size, the highest compression bps is obtained at R0, which has all its tuples duplicated with the highest selectivity and the lowest compression bps happens at the lowest selectivity R6.
2. Although compression bps grows when the increases under the same duplication level, the growth of compression bps slows down when the tuples size has reached

a certain point. For example, compression bps for selectivity 0.17 slows down to increase after the # of tuples is greater than 6000. This again motivated our interest to measure compression bps for a wider range of data sizes.

5.3.3. Compression bps under 3 different platforms

In order to make further conclusions on how compression bps changes with the size of data to be compressed, we have designed another test to measure compression bps on a wide range of data sizes and carried this test on 3 different types of computing platforms with their system information listed in Figure 5.12.

	Davinci	Horus	PC
Machine	Sun Enterprise 6500 Server	IBM cluster	PC with dual CPUs
OS	Solaris	Debian Linux	Linux
Memory	8G Shared Memory	8G DSM in total (Master: 900M, node1-node14: 516M)	320MB SDRAM
Work Load	Heavy (load: 3.5 – 4)	Light (Load: 0.1 - 0.2)	Light (Load: 0.1)
CPU/node	14 UltraSparc processors	15 nodes with Xeon 2 Ghz Hyper thread processor on each node	2X Intel Celeron A 367.5MHz
NETWORK	Gigabit Ethernet	64 bit/66MHz Gigabit Myrinet	10/100M Ethernet

Figure 5.12: Three typical computing platforms in the real world

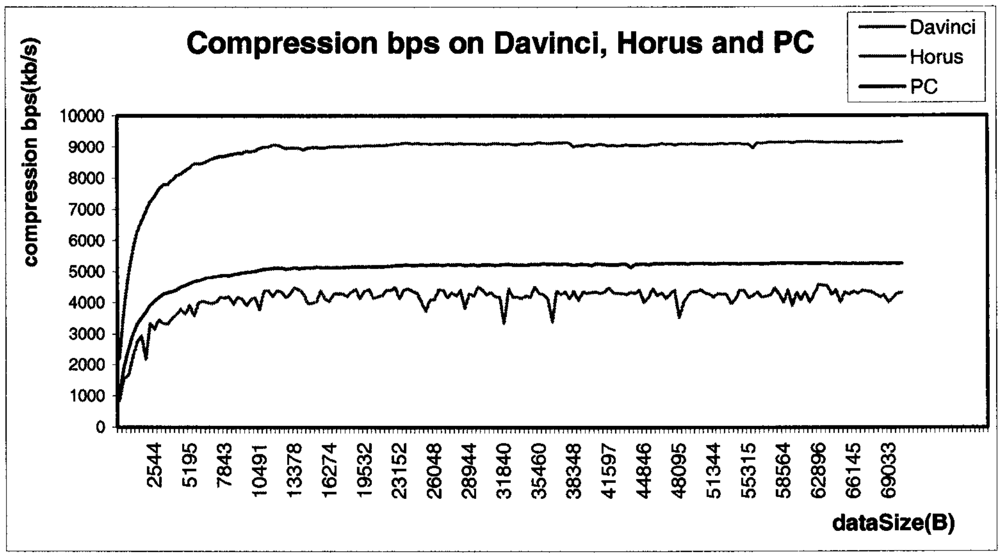


Figure 5.13: Compression bps for the above three typical computing platforms

The measurements under these different platforms have revealed that:

1. Compression bps converges after data size is greater than a certain point, which we name "converge point". Compression bps grows as the compression data size

grows until this converge point is reached. For example, compression bps measured under IBM cluster Horus increases from 2M b/s to 9M b/s when the data size grows from 3M Bytes to 10Mbytes, and compression bps remains 9M b/s after the data size reaches 10M bytes.

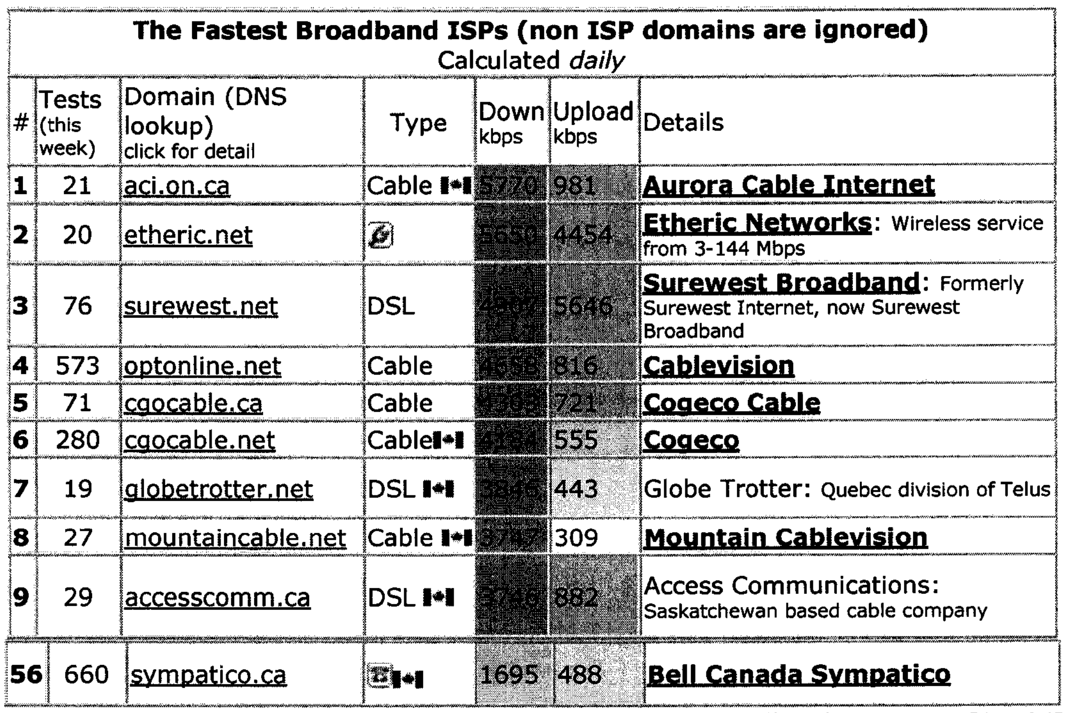
2. CPU computing power and the current machine workload are critical computing resources to influence compression bps. As a result, an off-shelf single-user PC produces a faster compression bps than a Server (Davinci) with heavy workload (3.5- 4), as shown in Figure 5.13. This makes compression available and suitable even for cheap and off-shelf machines.
3. As data size varies in a wide range, compression bps measured in this test can be used as hints to decide when compression is beneficial in DQP. For example, if current data transfer speed is 3M b/s on Davinci, compression may be considered useful when the relation to be compressed is larger than 4M bytes, since compression bps on Davinci is greater than 3M b/s only when data Size is greater than 4M bytes.

5.3.4. Compare compression bps with current data transfer speed

Compression bps measured under Davinci, Horus and the PC converges to 4M b/s, 9Mb/s and 5M b/s respectively, which is quite impressive compared with the current data transfer speed over Internet, the real distributed system. According to the data transfer speed published by broadband.com [Bro04] in May 7th, 2004, the fastest download speed to date was 5M b/s (Figure 5.5) and most end users received a data transfer speed in the range of 1M-1.5 M (Figure 5.6). With slow network still the main bottleneck for data transfer in distributed systems, compression can be considered an effective and reasonable way for distributed query optimization.

We summarize the findings of our tests on compression bps:

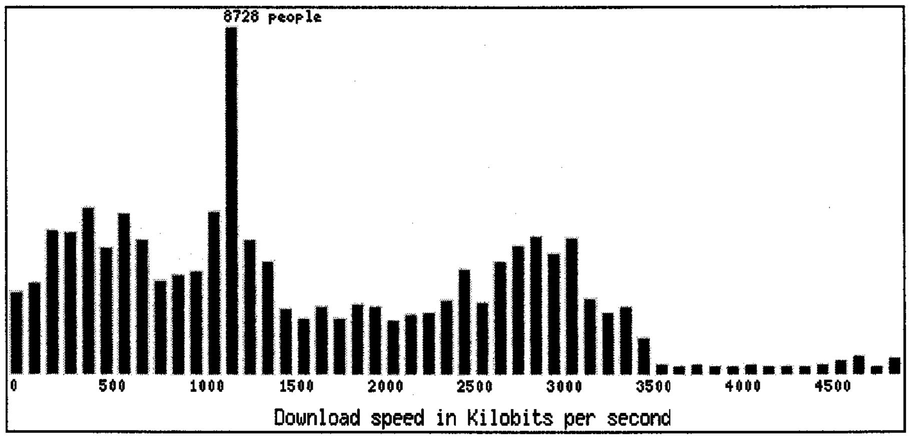
1. For the queries with the same #Rel, a higher compression bps is reached if a query has more join attributes.
2. Higher compression bps is reached if join attributes have higher duplications. i.e. more transmission cost can be reduced with compression on queries with highly duplicated join attributes.



The Fastest Broadband ISPs (non ISP domains are ignored) Calculated *daily*						
#	Tests (this week)	Domain (DNS lookup) click for detail	Type	Down kbps	Upload kbps	Details
1	21	aci.on.ca	Cable 🇨🇦	5770	981	**Aurora Cable Internet**
2	20	etheric.net		5650	4454	**Etheric Networks**: Wireless service from 3-144 Mbps
3	76	surewest.net	DSL	4907	5646	**Surewest Broadband**: Formerly Surewest Internet, now Surewest Broadband
4	573	optonline.net	Cable	4658	816	**Cablevision**
5	71	cgocable.ca	Cable	4303	721	**Cogeco Cable**
6	280	cgocable.net	Cable 🇨🇦	4184	555	**Cogeco**
7	19	globetrotter.net	DSL 🇨🇦	3846	443	Globe Trotter: Quebec division of Telus
8	27	mountaincable.net	Cable 🇨🇦	3747	309	**Mountain Cablevision**
9	29	accesscomm.ca	DSL 🇨🇦	3746	882	Access Communications: Saskatchewan based cable company
56	660	sympatico.ca	🇨🇦	1695	488	**Bell Canada Sympatico**

(Courtesy of broadbandreport.com [Bro04])

Figure 5.14: The download and upload speed for the fastest ISPs



(Courtesy of broadbandreport.com [Bro04])

Figure 5.15: Data transfer speed distribution

3. Compression bps increases as the size of data to be compressed grows, until it reaches the converge point.
4. Compression bps measured under three real computing environments is comparable with current Internet data transfer speed. This makes compression a practical and reasonable option to be used in DQP to reduce transmission cost, even when the overhead of compression has to be considered.

Chapter 6 Conclusions and future work

6.1. Conclusions

We address the goal of distributed query optimization in the approach to minimize the transmission cost by reducing the amount of data to be transmitted over network. PERF join, which optimizes the backward reduction phase of the 2-way semijoin with the transmission of PERF filters (bit arrays generated according to relation tuple scan order) instead of join attribute projections, has attracted recent research attention [LR95] [HF00] [Zhang03] [Zhu04] due to its use of space-efficient and collision-free PERF filters. The issue of eliminating duplicated tuples in join attributes has been brought up to further optimize PERF join in [LR95][Zhang03][Zhu04]. From our investigations to date, sorting is the only approach that has been proposed to eliminate duplications [LR95]. The inefficiency of sorting has motivated our algorithm PERF_C, where compression is applied to both join attributes projections (in forward reduction phase of PERF join) and PERF filters (in backward reduction phase of PERF join), to further reduce the transmission cost in algorithm PERF. Through theoretical analysis, compression has been proved more efficient than sorting to eliminate duplicates. After further research, we found that the benefit of compression goes beyond simply eliminating duplicates. Compression is an effective operation to reduce transmission cost for general queries of arbitrary number of relations, join attributes, and selectivity levels as well. Our experiments have shown that algorithm PERF_C achieves a cost reduction ratio of 62%-77% for randomly generated general queries, which outperforms algorithm PERF by about 16%-36%.

Since the transmission cost among distributed site is more critical in DQP, it is common that local cost is not considered in most researches in DQP. But with the increasing of network link speed, it is not always feasible to ignore expensive local processing [RK91]. In this thesis, a new term "compression bps" is defined to address the local cost for compression and decompression during algorithm PERF_C. We prove that compression should be used in DQP if compression bps is faster than current data transfer speed. In this sense, compression bps can be treated as a guide to decide when compression should be applied in DQP, even when the local cost of compression is also considered.

Experiments on compression bps with distributed queries have shown that the higher the number of join attributes and the higher duplication level, the higher compression bps will be. Compression bps is also measured based on three typical computing platforms to give an idea of compression bps on real computing environments. The measured compression bps falls into the range of 4 - 9 M b/s, which is fast enough compared with current data transfer speed published in May 7th 2004.

Summary of thesis contributions:

1. New algorithm PERF_C was proposed to apply compression to PERF join to further reduce transmission cost.
2. New concept of compression bps was defined to measure the compression speed. Theorem 1 and Theorem 2 give hints that compression should be beneficial in DQP if compression bps is greater than data transfer speed.
3. Designed, implemented algorithm PERF_C with arithmetic coder as the compressor. Algorithm PERF was also implemented to compare with algorithm PERF_C.
4. The old testbed was extended to allow bigger domain and relation size. New features to create user-controllable relations with user-defined relation size and user-defined selectivity level were added to the testbed.
5. Experiments were designed and implemented to evaluate both algorithm PERF_C and compression bps. All the experiments are carried out with 500 runs to avoid coincidence and to enhance robustness.
6. Compression was found to be an effective operation to reduce the transmission cost in DQP. Through experiments on 36 types of general queries, Algorithm PERF_C outperforms algorithms PERF with a gain of 16%-36% more cost reduction ratio achieved.
7. Compression bps was measured to be in the range of 4 - 9M b/s under the real computing platforms, which is quite impressive compared with current data transfer speed on Internet.

6.2. Future work

1. Interleave compression with transmission during DQP to reduce the compression overhead. Current implementation has a serial interface between the compressor

and the reducer, where the reducer has to wait for the compressor to finish. For example, join attributes projections are first compressed and then the compressed join attributes projections are transmitted to the distributed sites. If we can interleave compression with transmission in a pipelined manner, the time spent on compression can be overlapped with transmission.

2. Change the reducer and set up a reducer model: Apply compression to a wider range of distributed query optimization methods, for example, Bloom filter or semijoin, to see how compression performs. i.e. replace the reducer module in Figure 5.1. With other DQP reducer. It is ideal to be able to come up with a reducer model to describe how the cost reduction ratio is related to the database factors in distributed queries, such as number of relations (#Rel), number of join attributes (#attr), size of relations, selectivity, duplication level etc.
3. Change the compressor and set up a compressor model: Gather knowledge on how compression bps is influenced by the compressor. Re-evaluate PERF_C and calculate compression bps using different compression modes of arithmetic coder or even try different types of compressor. It is desirable that a quantified model for compressor can be set up too.
4. Establish an adaptive decision-making system based on the knowledge of the distributed queries, the reducers, the compressors available, and the current distributed system data transfer speed. This adaptive decision-making system will suggest a decision on whether compression should be used and help to choose the appropriate reducer, compression method or mode based on the work in 1 and 2.

BIBLIOGRAPHY

[AHY83] P. Apers, A. Hevner and S. Yao, "Optimization algorithms for distributed queries", IEEE Transactions on Software Engineering, 9(1), pp.51-60, 1983.

[AM91] J. Ahn, S. Moon, "Optimizing joins between two fragmented relations on a broadcast local network", Info. Syst., Vol. 16(2), pp.185-198, 1991

[BC81] P. Bernstein and D. Chiu, "Using semi-join to solve relational queries", Association for Computing Machinery Journal, vol.28, pp.25-40, Jan 1981.

[Bea95] W. T. Beal or, "Semi-join strategies for total cost minimization in distributed query processing", Master thesis, University of Windsor, 1995.

[BFS00] S. Bandyopadhyay, Q. Fu and A. Sengupta, "A Cyclic multi-relation semijoin operation for query optimization in distributed databases", Proc. 19th IEEE International Performance, Computing and Communications Conference - IPCCC 2000 PERFORMANCE, February, 2000.

[BG81] P. A. Bernstein, N. Goodman, "The power of natural semijoins", SIAMJ. Computer, Vol. 10. 4, pp.751-771, Nov. 1881.

[BG93] D. Bell, J. Grimson, "Distributed Database Systems", Addition-Wesley.

[BGW+81] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.Rothie, "Query processing in a system for distributed databases(SDD-1)", ACM Transactions on database Systems,Vol.6(4), pp.602-625, 1981.

[BKK+01] R. Braumandl, M. Keidl, A. Kemper, D.Kossmann, S.Seltzsam,K. Stocker, "Object Globe: Open Distributed Query Processing Services on the internet", IEEE Computer Society Technical Committee on data Engineering, pp.1-7, 2001.

[BL82] P. A. Black,W. S. Luk, "A new heuristic for generating semi-join programs for distributed query proceeding", In Proc. IEEE COMPSAC, pp.581-558, Dec. 1982

[Blo70] B. H. Bloom. “Space/time trade-offs in hash coding with allowable errors”, Communication of the ACM, vol.13(7), pp.422-426, 1970.

[BPR90] P. Bodorik, J. Pyra and J. S. Riordon, “Correcting execution of distributed queries”, In Proc. Of 2nd int. Symp. on Databases in Parallel and distributed Systems, pp.192-201, 1990.

[BR88] P. Bodorik and J. S. Riordon, “Heuristic Algorithms for Distributed Query Processing”, IEEE, pp.144-155,1988.

[BR88] P. Bodorik, J.S. Riordon, “Distributed query processing optimization objectives”, Fourth International Conference on Data Engineering, pp. 320 – 329, 1988.

[BRB+01] C. Badue, B. Ribeiro-Neto, R. Baeza-Yates, N. Ziviani, “Distributed query processing using partitioned inverted files”, String Processing and Information Retrieval, 2001. Proceedings. Eighth International Symposium on , 2001, pp. 10 - 20.

[BRJ89] Peter Bodorik, J. Spruce Riordon and C. Jacob, “Dynamic Distributed Query Processing Techniques”, ACM, pp.349-357, 1989.

[Bro04] Broadband report.com retrieved May 7th, 2004. http://www.broadbandreports.com/archive

[BRP92] Peter Bodorik,J. Spruce Riordon and James S. Pyra, “Deciding to Correct Distributed Query Processing”, IEEE Transactions on Knowledge and data Engineer in Vol.4 No.3, pp.253-265, Jun. 1992.

[CBH84] D. M. Chiu, P. A. Berstain and Y. C. Ho, “Optimizing chain queries in a distributed database system”, SIAMJ. Computer, Vol. 13. No.1, pp.116-134,1984.

[CCY92] Tung-Shou Chen, Arbee L.P. Chen and Wei Pang Yang, "Hash-semijoin: A new technique to minimizing Distributed Query time", IEEE, 1992.

[CH82] W.W. Chu, P. Hurley, "Optimal Query Processing for Distributed Database System", IEEE. Trans. On Comput., Vol. c-31,no.9, pp.135-150, Sep. 1982.

[CH84] D. M. Chiu, Y. C. Ho, "Optimizing star queries in a distributed database system: A method for interpreting tree queries into optimal semijoin expressions", VLDB, pp.959-967,1984.

[Cha82] Jo-Mei Chang, "A Heuristic Approach to Distributed Query Processing", Proceeding of the 8th VLDB Conference, pp.54-61, 1982.

[CL00] H. Chen and C. Liu, "An efficient algorithm for processing distributed queries using partition dependency", Parallel and Distributed Systems, pp. 339 –346, 2000.

[CL84] L. Chen and V. Li, "Improvement algorithms for semi-jion query processing programs in distributed database system", IEEE Transaction on computers, Vol. 33(11), pp.959-967, 1984.

[CL85] A. L. P. Chen, V. O. K. Li, "An optimal algorithm for distributed star queries", IEEE Trans. On Software Engineering, Vol.11 No. 10, pp.1097-1107,1985.

[CL90] L. chen and V.Li, "Domain-specific semi-join: A new operation for distributed query processing", Information science,Vol. 52, pp.165-183, 1990.

[CY90] M.-S. Chen, P.S. Yu, "Using combination of join and semijoin operations for distributed query processing", Distributed Computing Systems, Proceedings., 10th International Conference on, pp. 328 –335,1990.

[CY91] M.-S. Chen and P. S. Yu, "Determing Beneficial Semijoins for a join

Sequence in Distributed Query Processing", IEEE, pp.50-58, 1991.

[CY92] M.-S. Chen, P. S. Yu, "Interleaving a Join Sequence with Semijoins in Distributed Query", Parallel and Distributed Systems, IEEE Transactions on, Volume: 3 Issue: 5, pp.611 –621, Sept. 1992.

[CY93] M.-S. Chen, P. S. Yu, "Combining Jion and Semi-Jion Operations for Distributed Query processing", IEEE Transactions and Data Engineering Vol.5, No.3, pp.534-542, 1993.

[CY94] M.-S. Chen, P. S. Yu, "A graph theoretical approach to determine a join reducer sequence in distributed query processing", Knowledge and Data Engineering, IEEE Transactions on, Volume: 6 Issue: 1, 152 –165, Feb. 1994.

[CY96] M.-S. Chen, P. S. Yu, "Optimization of parallel execution for multi-join queries", IEEE, pp.416-428, 1996.

[ESW78] R. Epstein, M. StoneBraker, and E. Wong, "Query Processing in a Distributed Relational Database System", In Proc. ACM SIGMOD Int. Conf. On Management of Data, pp.169-180, May 1978.

[FCAB99] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol (Extended version)," Computer Sciences Dept., University Wisconsin–Madison, Tech. Rep. 1361, Feb. 1999.

[GBHC00] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed Data Structures for Internet Service Construction", In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), (San Diego, CA, 2000).

[Gra96] Jim Gray, "Data Management: Past, Present, and Future", Microsoft Research, Technical Report MSR-TR-96-18, Jun. 1996.

[Gre98] Michael Gregory, "Genetic Algorithm Optimization of Distributed Database Queries", IEEE, pp.271-267, 1998.

[GW89] G. Graefe and K. Ward, "Dynamic Query evaluation plans", ACM SIGMOD, pp.73-170, 1989.

[HCY97] H.-I. Hsiao, M.-S. Chen, P. S. Yu, "Parallel Execution of Hash joins in Parallel Database", IEEE, pp.872-883, 1997.

[HF00] R. Haraty, R. Fany. "Distributed query optimization using PERF joins", Symposium on Applied Computing Proceedings of the 2000 ACM symposium on Applied computing, Como, Italy, 2000. Pages: 284 – 288.

[HK94] Harris and Kotagiri. "Join Algorithm Costs Revisited", VLDB Journal Vol.5, 1994, pp. 64 – 84.

[HM00] A. Hameurlain and F. Morvan, "An Overview of Parallel Query Optimization in Relational Systems", IEEE,2000.

[HY79] A.R. Hevner and S. B. Yao, "Query Processing in Distributed Databases", IEEE Trans. on Software Eng., SE-5(3), pp.1770-187, 1979.

[JK84] M. Jarke and J. Koch, "Query Optimization in database system", ACM Computing Surveys Vol.16, No.2, pp.112-152,1984.

[KHY82] Y. Kambayashi, M. Yoshikawa and S. Yajima, "QUERY Processing for Distributed Databases Using Generalized Semijoins", ACM Proceedings of SIGMOD, pp.151-160,1982.

[Kos00] D. Kossmannn, "The State of the Art in Distributed Query Processing", ACM Computing Surveys (CSUR), v.32 n.4, pp.422-469, Dec. 2000.

[Kos98] D. Kossmann, "Iterative Dynamic Programming: A New Class of Query Optimization Algorithms", ACM Transactions on Database Systems,1998.

[KR87] H. Kang and N. Roussopoulos, "Using two-way semijoins in distributed query processing", In Proceedings of The 3rd International Conference on Data Engineering, pp.664-651, 1987.

[KS00] D. Kossman and K. Stoker, "Iterative Dynamic Programming A new Class of Query Optimization Algorithm," ACM Transaction on Database Systems, Vol.25, No.1, pp.43-82, 1986.

[LC01] C.-H. Le and M.-S. Chen, "Distributed query processing in the Internet: exploring relation replication and network characteristics", 21st International Conference on Distributed Computing Systems, , Apr 2001, pp. 439 - 446.

[Lia98] Yan Liang, "Reduction of collisions in bloom filters during distributed query optimization", M.sc Theses, Computer Science, University of Windsor,1998.

[LR95] Z. Li and K. A. Ross, "PERF join: an alternative to two-way semijoin and bloomjoin", In Proceedings of CIKM'95, pp.137-144, 1995.

[LY93] C. Liu and C. Yu, "Performance issues in distributed Query Processing. IEEE, pp.889-905,1993.

[Ma97] X. Ma. "The use of bloom filters to minimize response time in distributed query optimization". Master thesis, University of Windsor, 1997.

[MB95] J. Morrissey and S. Bandyophdhyay, "Computer Communication Technology and its effects on Distributed Query Optimization Strategies", IEEE, pp.598-601, 1995.

[MB96] J. Morrissey and W. T. Bealor, "Minimizing data transfers in distributed query optimization: A comparative study and evaluation",The Computer Journal, vol.39, no.8, pp.676-687, Dec 1996.

[MB96] J. Morrissey and W. T. Bealor, "Minimizing data transfers in distributed query processing: a comparative study and evaluation", The computer journal, Vol.39, No. 8, pp. 675 – 687, 1996.

[MBB95] J. Morrissey, S. Bandyopadhyay, and W. T. Bealor, “A comparison of static and dynamic strategies for query optimization”, In Proceedings of the 7th /IASTED/ISM International Conference on Parallel and Distributed Computing Systems, 1995.

[ME92] P. Mishra and M. Eich, “Join processing in relational databases”, ACM Computing Surveys, vol.24(1), pp.63-113, March 1992

[Mit02] M. Mitzenmacher, “Compressed Bloom filters”, IEEE/ACM Transactions on networking.Volume: 10 Issue: 5 , Oct 2002. Page(s): 604 –612

[MM98] J. Morrissey and X. Ma, “Investigating response time minimization in distributed query optimization”, Present at ICCI'98, pp.124-138, 1998.

[MNW98] A. Moffat, R. M. Neal and I. H. Witten, “Arithmetic Coding Revisited”, ACM Transactions on Information Systems, Vol.16, No.3, July 1998.

[MO97] J. Morrissey and W.K. Osborn, “Experiences with the use of reduction filters in distributed query optimization”, In proceedings of the 9th international Conference On Parallel and Distributed Computing and Systems (PDCS'97), pp.327-330, 1997.

[MO98] J. Morrissey and W.K. Osborn, “Distributed Query Optimization using reduction filter”, Electrical and Computer Engineering, 1998. IEEE Canadian Conference on, Volume: 2, pp.707-710,1998.

[MO99] J. Morrissey, W. K. Osborn, “The effect of collisions on the performance of reduction filter”, Proceedings of the 1999 IEEE Canadian Conference n electrical and computer Engineering, pp.215-219, 1999.

[MOL00] J. Morrissey, W. K. Osborn, and Y. Liang, “Collisions and reduction filters in distributed query processing”, In Proceedings of the 2000 IEEE Conference on Electrical and Computer Engineering, vol.1, pp.240-244, 2000.

[Mor96] J. Morrissey, "Reduction filters for minimizing data transfers in distributed query optimization", In Proceedings of the 1996 Canadian Conference on Electrical and Computer Engineering, vol.1, pp.198-201, 1996.

[Mor96] J. M. Morrissey, "Reduction filters for minimizing data transfers in distributed query optimization", IEEE, pp.198-201,1996.

[Mul83] J. K. Mullin, "A second look at bloom filters", Communication of the ACM, vol.26(8), pp.570-571, August 1983.

[Mul94] J.K. Mullin, "Optimal Semijoins for Distributed Database Systems", Software Engineering, IEEE Transactions on, Volume: 16 Issue: 5, pp. 558 – 560,1994.

[Mul96] Craig S. Mullins, "Distributed Query Optimization", Technical Enterprise, 1996.

[NS98] F. Najjar and Y. Slimani, "Distributed optimization of cyclic queries with parallel semijoins", IEEE, pp. 717 -722,1998.

[Osb98] W. Osborn, "The use of reduction filters in distributed query optimization", Master's Thesis, the university of Windsor,1998.

[Ozs99] M. T. Ozsu, "Principles of Distributed Database Systems", Printice Hall, Upper saddle River, New Jersey 07458, Second Edition (1999).

[PC90] W. Perrizo, C. Chen, " Composite Semijoins in Distributed Query Processing", Information Sciences Vol. 50,No.3, pp.197-218,1990.

[Per85] W. K. Perrizo, "Upper bound response time semijoin strategies", 1st international conference on super computer systems, pp.273-279,1985.

[PV88] S. Pramanik and D. Vineyard, "Optimizing join queries in distributed database", IEEE Transaction on software engineering Vol. 14, No. 9, pp1319-1326, Sept. 1988.

[Ram89] M. V. Ramakrishna. "Practical performance of Bloom filters and parallel free-text searching", Communications of the ACM,32 (10). 1237-1239, Oct.1989

[RI01] M. Ripeanu, A. Iamnitchi, "Bloom Filters – Short Tutorial", Retrieved 2003 http://people.cs.uchicago.edu/~matei/PAPERS/bf.doc

[RK91] N. Roussopoulos and H.Kang, "A pipeline n-way join algorithm based on the 2-way semi-jion program", IEEE Transactions on Knowledge and data Engineering Vol.3(4), pp.486-495, 1991.

[SB82] M. S. Sacco and S. B. Yao, "Query Optimization in Distributed Database System", In M. C. Yovits (ed.), Advances in Computers,Volum21,New York: Academic press, pp225-273,1982.

[Sch90] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator", In Proceedings of the 2 C++ Conference, pages 87--102, San Francisco, California, April 1990. USENIX.

[Seg86] A. Segev, "Global Heuristic for Distributed Query Optimization", Proceedings of IEEE INFOCOM, pp.388-394, 1986.

[TC02] P.S.M. Tsai, A.L.P. Chen, "Optimizing queries with foreign functions in a distributed environment", Knowledge and Data Engineering, IEEE Transactions on, Volume: 14 Issue: 4, pp. 809- 824, 2002.

[TC92] J. C. R. Tseng and A. L. P.Chen. "Improving distributed query processing by hash-semijoins", Journal of Information Science and Engineering, vol.8, pp525-540, 1992.

[Vla97] Richard Vlach, "Query Processing in Distributed Database System", 1997

[Wan90] Chihping Wang, "The complexity of processing tree queries in distributed databases", IEEE, pp.604-601,1990.

[WC96] C. Wang and M-S. Chen, “On the complexity of distributed query optimization”, IEEE Transactions on Knowledge and Data Engineering, vol. 8(4), pp.650-662, 1996.

[WC96] C. Wang and M-S Chen, “On the Complexity of Distributed Query Optimization”, Knowledge and Data Engineering, IEEE Transactions on, Volume: 8 Issue: 4, pp.650-662,1996.

[WCS92] C. Wang, A. L. P. Chen and S- C Shyu, “A parallel execution method for minimizing distributed Query response time”, IEEE transactions on Parallel and distributed Systems, 3(3), pp.325-333,1992.

[WLC91] C. Wang, V. Li and A. L. P Chen, “Distributed Query Optimization by One-Shot Fixed-Precision Semi-Join Execution”, Processing 7th International Conference on Data Engineering, pp.756-763,1991.

[Won77] E. Wong, “Retrieving dispersed Data from SDD-1”, Proc. 2nd Berkely Workshop on Distributed Data Management and Computer Networks, pp.217-235, 1977.

[YC84] C. T. Yu and C. C. Chang, “Distributed query processing”, ACM Computing Surveys,16(4), pp.399-433,1984.

[YL90] C. Yu and C. Liu, “Experiences with distributed query processing”, IEEE, pp. 192-199,1990.

[Zha03] Yue (Amber) Zhang, “Variation of Bloom Filters Applied in Distributed Query Optimization”, Master’s Thesis, University of Windsor, 2003

[Zhu04] Y.M. Zhu. “Implementation of Composite Semijoins Using A Variation of Bloom Filters” Master thesis, University of Windsor, 2004.

Vita Auctoris

Name:	**Ying (Joy) Zhou**
Education:	
	2004 M.Sc. Computer Science University of Windsor Windsor, Ontario, Canada
	1995 M. Eng., Electrical and Computer Engineering Nankai University Tianjin, P.R. China
	1992 B.Sc., Computer and System Science Nankai University Tianjin, P.R. China
Working Experience:	
	Web Developer & Computer Technician Liquidator Powerhouse, Windsor, ON
	Research Assistant University of Windsor 2001 — 2003
	Graduate Teaching Assistant University of Windsor 2001 — 2003
	Instructor/Lecturer Nankai University 1995 — 2000
	Software Engineer HuaXing System Integration and Electronic Engineering Tianjin, PRC 1996 — 2000