DEVELOPMENT AND EVALUATION OF INTER-QUERY OPTIMIZATION HEURISTICS IN DATABASE SYSTEMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENC

By

Yiğit Kulabaş

January, 1996

DEVELOPMENT AND EVALUATION OF INTER-QUERY OPTIMIZATION HEURISTICS IN DATABASE SYSTEMS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Yiğit Kulabaş

January, 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Özgür Ulusoy (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Cevdet Aykanat

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Erol Arkun

Approved for the Institute of Engineering and Science:

[illegible]

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

DEVELOPMENT AND EVALUATION OF INTER-QUERY OPTIMIZATION HEURISTICS IN DATABASE SYSTEMS

Yiğit Kulabaş
M.S. in Computer Engineering and Information Science
Advisor: Asst. Prof. Özgür Ulusoy
January, 1996

In a multi-user database system multiple queries can be issued by different users at about the same time. These queries may have some common operations and/or common relations to process. In our work, we have developed some inter-query optimization heuristics for improving the performance by exploiting the common relations within the queries. We have focused mostly on the join operation, with the build and probe phases. Some of the proposed heuristics are for the build phase, some for the probe phase, and finally some for the memory flush operation. The performance of the proposed heuristics is studied using a simple simulation model. We show that the heuristics can provide significant performance improvements compared to conventional scheduling methods for different workloads.

Keywords: Inter-query optimization, join operation with build and probe phases, memory flush operation.

ÖZET

VERİTABANI SİSTEMLERİNDE SORGULARARASI OPTİMİZASYON METODLARININ GELİŞTİRİLMESİ VE İNCELENMESİ

Yiğit Kulabaş
Bilgisayar ve Enformatik Mühendisliği Bölümü - Yüksek Lisans
Danışman : Yrd. Doç.Özgür Ulusoy
Ocak 1996

Çok kullanıcılı veri tabanı sistemlerinde, yakın zamanlı olarak birden fazla sorgunun, değişik kullanıcılar tarafından sisteme yüklenmesi sıkça rastlanan bir durumdur. Bu sorgular ortak işlem ve/veya verilere sahip olabilirler. Çalışmamızda, bu ortak işlem ya da verilerin, sorgular tarafından paylaşılması amacıyla bazı sorgulararası en iyileme (optimizasyon) teknikleri geliştirmiş bulunmaktayız. Bu çalışmalar sırasında daha çok birleştirme (join) işlemine odaklandık. Birleştirme işleminde ise şekil olarak dağıtım kodlaması (hash) tekniğini kullandık. Geliştirilen metodların bir kısmı dağıtım kod tablolarının oluşturulması, bir kısmı bu tabloların sorgulanması, bir kısmı ise belleğin verimli olarak kullanılmasını sağlamaktadır. Çalışma kapsamında ayrıca bu metodların performansları bir benzetim modelinin üzerinde karşılaştırılmıştır. Karşılaştırmalar sonucunda en çok dikkat çeken nokta ise, geliştirilen bütün tekniklerin, sorguların birer birer ele alınması tekniğine göre oldukça önemli üstünlükler göstermesi olmuştur.

Anahtar Kelimeler: Sorgulararası en iyileme, birleştirme işlemi, inşa-etme aşaması, sorgulama aşaması, bellek temizleme aşaması.

ACKNOWLEDGEMENTS

I am very grateful to my advisor Asst. Prof. Özgür Ulusoy for his guidance, suggestions, and encouragement throughout the development of this thesis. I would like to thank Prof. Erol Arkun and Assoc. Prof. Cevdet Aykanat for reading and commenting on the thesis.

Contents

List of Figures

List of Tables

1. Introduction

We start this chapter with a brief introduction to the concepts in Multi-Query Environments, Query Scheduling Optimization, Intra-Query Parallelism, Inter-Query Parallelism, and Sharing Relational Operations. Following this general introduction, we provide a brief overview of the thesis topic, along with the related work and the general organization of the thesis.

1.1. Overview of Multi-Query Environments

During the last decade, there have been significant enhancements in the performance of computer systems. The processor speed and the capacity of main memory and secondary storage devices have steadily increased. Symmetric multiprocessing technology, clustering technology, and parallel processing technology have continuously been improved. These enhancements have also enforced improvements in the database technology. The leading database vendors have developed uniprocessor, multiprocessor, clustered, and parallel versions of their DBMS's. These versions have continuously been enhanced to have better performance.

An important part of the workload for the database systems consists of resource-intensive read-only queries. For this reason, one of the main factors that the performance depends on, is query scheduling. To optimize the scheduling of queries in terms of the response time, various algorithms have been developed. These algorithms mainly focused on intra-query parallelism. In other words, research on query scheduling has so far concentrated mainly on efficient processing of single queries.

However, an important property of DBMS's, regardless of whether they are uniprocesor, multiprocessor, clustered, or parallel systems, in general they are multi-user systems. In a multi-user system, multiple queries can be delivered by different users at the same time. A multi-user environment leads to "multi-query" executions, and to optimize query scheduling in such environments, the focus should not be only on "single-query" execution but also on "multi-query" execution.

In a multi-query environment, quite often the same relation is a part of several queries. When the queries are optimized only by the single-query manner, such relations are processed multiple times, since the queries are scheduled independently. If these relations could be shared between concurrently executing queries, they would have to be processed only once.

A multi-query scheduling method can optimize a set of queries together, by identifying common relations in the set and scheduling them in such a way that the repeated processing of the common relations is avoided. Therefore, the total execution time of queries will decrease.

This thesis is focused on efficient scheduling of "multi-queries". Several scheduling algorithms are presented which can be used to take advantage of sharing of relations among a set of queries. The aim is to optimize query scheduling in multi-user environments. The general information about the thesis can be found in Section 1.2. Before the overview, let us go over the parallelism concepts used in query scheduling.

1.1.1. Intra-Query Parallelism vs. Inter-Query Parallelism

Various query scheduling techniques based on parallelising query executions, have been developed. The focus in these techniques have basically been on the parallel execution of a single query at a time. This type of parallel execution is called "intra-query" parallelism.

1.1.1.1. Intra-Query Parallelism:

This type of parallelism is used in most of the parallel machines, and parallel DBMS's. There are two types of intra-query parallelism:

- Pipelined Intra-Query Parallelism
- Partitioned Intra-Query Parallelism

1.1.1.1.1. Pipelined Intra-Query Parallelism:

Queries that require multiple relational operations can be parallelised by the "pipeline" version. In this type of scheduling, the operations are classified on the basis of whether they are dependent or independent. The dependent operations use the output of the other operations (e.g., sorts, aggregation).

This type of parallelism benefits in OLCP (On-line Complex Processing) and DSS (Decision Support Systems) response times and can be used on any type of multi-processor machines [1].

1.1.1.1.2. Partitioned Intra-Query Parallelism:

Any type of query can be parallelised by this method since the parallelism is achieved by data partitioning. In this type of scheduling, database tables are partitioned and spread across multiple disks.

This type of parallelism provides a high degree of scalability when searching database tables, benefits DSS processing of large tables and is only available on parallel (shared-nothing) machines [1].

1.1.1.2 Inter-Query Parallelism:

This method deals with the scheduling of more than one query at a time. In this type of scheduling, multiple threads are used for execution of each query. Multiple threads may execute on multiple processors. There is a significant benefit in reducing OLTP response times.

In the literature, the term "inter-query parallelism" is mostly used to refer to the "independent" processing of multi-queries as multi-threads. In this thesis, we deal with the "dependency" of the multi-queries, which is another side of the inter-query parallelism [1].

1.1.2. Inter-Query Parallelism and Relational Operations

Inter-query parallelism can be applied on any type of relational operations. It tries to achieve the following goal: get the information in the relations into the main memory and complete the operation as quickly as possible. Therefore, efficient memory management is one of the most important requirements in query scheduling. Inter-query parallelism can optimize the memory usage by sharing common tables between different operations of different queries.

Among all relational operations, the join operation is distinguished by its complexity. It is the most expensive relational operation in terms of the amount of system resources required. Many algorithms have been developed to eliminate this complexity; namely traditional nested loops, hashing techniques, sort-merge techniques, and their mixtures. All these algorithms have both serial and parallel versions. It has been shown that hash-based join algorithms outperform the others in most environments. There are also different versions of the hash join algorithms. One of the most commonly used one is the hybrid-hash join algorithm. In this thesis our discussions will basically involve this algorithm.

The hybrid-hash join algorithm executes in two phases, called "Build" and "Probe". In the first phase, the smaller relation is scanned and an in-memory hash table is constructed by hashing each tuple on the join attribute. In the second phase, the outer relation is scanned and hash values of the tuples are used to probe the hash table to test for matches. The matched tuples are joined. There is a partial order defined on these two phases. The probe phase cannot begin until the build phase is completed [2].

Although sharing in multi-query environments is applicable for any type of query operations, the focus in this thesis will be on the join operation.

1.2. Overview of the Thesis

This thesis presents several scheduling algorithms that can be used to exploit the sharing of relations for batches of queries.

In a multi-user database system, the queries delivered to the same system, by different users at about the same time, can include the same relations, operations, or even the same total query. As an example consider an Airline Database. There can be lots of agencies, and other users entering exactly the same query: "Flights from New York to Miami on Dec 27, 1995" at about the same time. Or, think of a similar example, this time the queries entered about the same time are not exactly the same but they use the same relations: user-1 "Flights from New York to Miami on Dec 27, 1995", user-2 "Flights from New York to San Francisco on Dec 27, 1995".

If the queries delivered within a certain time period are not considered as a batch, the potential of sharing some operations and relations can not be exploited. This is the point that shows the importance of inter-query optimization for the performance enhancements, and resource utilization. However, this topic has been considered in just a few works as explained in Section 1.3. The significance of inter-query optimization and the lack of enough research works in this topic have been the basic motivations of the thesis.

An ideal multi-query scheduling method would optimize a set of queries together and minimize the total execution time. The conventional multi-query scheduling corresponds to scheduling queries independently. The scheduler allows the maximum possible number of queries to run together with no consideration for the sharing of relations.

The optimum exhaustive sharing algorithm has a huge time complexity: $n!.2^n.m!$, where n is the total number of relations, and m is the number of relations in the memory. Here n! is the complexity of selecting an ordering of the queries, 2^n is the number of choices to select probe/build relations, and m! is the complexity of determining the order in which relations are flushed from the memory [3]. Therefore, using the exhaustive algorithm is not feasible. Heuristic algorithms need to be developed to provide efficient sharing of relations in a batch of queries. The thesis provides such heuristics and presents their performance results obtained in a simulation environment.

For the sake of simplicity, some assumptions have been made in developing and testing the heuristics. A uniprocessor, multi-user database system has been chosen as the query execution environment. Due to its large time complexity, the join operation has been chosen to apply the inter-query optimization, although it is applicable to any kind of operations. For the implementation of the join operation, the Hybrid-Hash Join Algorithm has been used. All the algorithms have been designed to optimize the execution time of the build and probe phases of the hybrid hash join, and the memory flush operation, by exploiting the share of the common relations. All the queries are assumed to be single-join queries; i.e. each query consists of a join of two relations.

A full query optimization will be possible when the intra-query and inter-query optimization techniques are used together. In this thesis, we ignore the application of intra-query techniques as our focus is on inter-query processing.

The remainder of the thesis is organized as follows. In the second chapter, inter-query optimization will be discussed. The ideas behind developing the heuristics will be explained in detail. In the third chapter, the scheduling algorithms based on the proposed heuristics will be demonstrated. Each algorithm will be explained in detail, and related examples will be given. The fourth chapter is devoted to the simulation results. The performance results of the algorithms will be compared in this chapter. The last chapter concludes the thesis and provides some suggestions for the future work.

1.3 Related Work

The work on the optimization of queries have mostly been provided for single query scheduling. The optimization techniques have first begun with the simple, basic queries |4| and extended to larger and more complex queries [5] [6] [7] [8] [9].

For multi-query scheduling, we noticed that the number of related works is very few. One of these works is [10], which examines the effect of different memory allocation schemes. But this is a very preliminary study that can not answer important scheduling policy questions. There are some query-language-level optimization research conducted for multi-query environment [11] [12] [13] [14].

One of the studies that deals with multi-query optimization at the individual query operator level is the inspiration of this thesis [3]. In this study, the hybrid-hash join with the build and probe phases, is the operation that is focused on. In this paper, there are several heuristics proposed for improving the average response time of multiple queries. Our work considers a similar execution environment and has the same scheduling goal. We have developed some more heuristics for the build and flush phases of the join operation. We have also developed different probe heuristics. In [3] no alternatives are provided for the probe phase.

When developing new techniques for the join operation, we have had an extensive research on the join optimization [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27]. Some of these papers deal with different join techniques, some with hashing details in the join environment, and some with CPU, disk and memory usage of the join operation.

2. Inter-Query Optimization

In a set of queries, if several joins involve the same relation, the hash table for this relation can be shared. Here is a simple set as an example:

JOIN(X,Y), JOIN(X,Z).

For both joins, we have two phases: build the hash table of one of the relations (in most cases, the small one), probe the hash table with the other relation. Here the relation to be shared is X, and relations Y and Z will share the hash table of X to probe.

The sharing we just considered has two significant impacts on the efficiency of query execution. Since the built relation is scanned once, the total execution time is reduced. (If the two operations were handled separately, hash tables of two relations would have to be built, and the scan time would be doubled.) Another impact on the performance is that, since only one table is built, there will be a considerable saving in the memory usage.

As could easily be noticed, in the case of two queries it is straightforward to determine which relation should be selected to build and which one to probe. This decision does not require the involvement of complex algorithms.

Consider another set of queries:

JOIN(X,Y), JOIN(X,Z), JOIN(X,T), JOIN(T,Z), JOIN(T,A), JOIN(Y,Z), JOIN(A,Y)

In this case, the queries involve relation Y 3 times, relation X 3 times, relation Z 3 times, relation T 3 times, and finally relation A 2 times. Which relations should be built? Relation Y, X, Z, or T? Or even relation A? Let Y be the built relation and X, Z and A make the probes. After this processing (i.e. the probes to the hash table of relation Y are finalized), we are left with the requirement to process X 2 times, Z 2 times, T 3 times, and A 1 time. Let T be selected to be built and the probes are made. We are left with the processing requirements of X and Z 1 time. Is this a good solution? If Y and T had been built at the same time, and the probes would have followed them, would the total execution time be shorter? Would Y and T fit into memory at the same time? If T and A together fit into memory, would it be more efficient to build them both? After T and A are built, we still have to build X, Y, or Z, but none of them fits into memory; so, which relation should we flush from the memory? In selecting a relation to build, what criteria need to be considered? Many more similar questions can be asked. This simple example clearly illustrates the requirement for the optimization algorithms for the shared relations in processing the join operations.

Three basic questions are involved in inter-query optimization:

- Which relation to build?
- How and when to probe?
- Which relation to flush?

2.1. Which Relation to Build?

There can be many different considerations in detecting the relation to be built. Again, let us consider an example:

Set of Queries:

JOIN(A,B), JOIN(A,C), JOIN(B,C), JOIN(B,D)

with the following sizes

Relation A: 20 pages, Relation B: 40 pages,
Relation C: 10 pages, Relation D: 20 pages.

A number of heuristics can be considered in determining the built relation:

Heuristic 1 - Highest Number of Relations Interacted with:

In the example, A and C join with 2 relations, B with 3 relations, while D joins with only one relation. According to this heuristic, for this example, relation B is selected to be built.

	Phase 1 Number of Relations Interacted with		*Phase 2* Number of Relations Interacted with	
Relation A:	2 - (B , C)	probe	1 - (C)	build
Relation B:	3 - (A , C , D)	build	0	
Relation C:	2 - (A, B)	probe	1 - (A)	probe
Relation D:	1 - (B)	probe	0	

When the hash table of relation B is formed, the three relations that B joins, namely A, C and D can probe the table at the same time. As a result of these probes, the queries JOIN(A,B), JOIN(B,C), JOIN(B,D) are processed *(End of Phase 1).* Only the query JOIN(A,C) remains unprocessed. At this stage (Phase 2), B and D will have the value 0 (i.e., there is no join operation that uses these relations left), while A and C have 1. One of the relations will be built, the other one will probe and the remaining query, JOIN(A,C) will be processed.

The aim of using this heuristic is to build the relation that will be probed with the maximum number of relations. If we assume that each join query is assumed to be submitted by a different user, by this heuristic, after the build and probe phases, the highest number of users will get the results for their queries.

Heuristic 2 - Largest Number of Tuples that will Probe[3]:

Let's define the Weight of a Relation X as the total size of the relations that Relation X joins with. In the example given above, we have the following weights: A 50 (40+10), B 50 (20+20+10), C 60 (40+20), and D 40. As the first step, C is chosen to be built as its weight is larger than the others. If we use *Heuristic 2*, C will be built as shown below.

	Phase 1 Weight	
Relation A:	50 - (Size(B) + Size(C), 40+10)	probe
Relation B:	50 - (Size(A) + Size(C) + Size(D), 20+10+20)	probe
Relation C:	60 - (Size(A) + Size(B), 40+20)	build
Relation D:	40 - (Size(B), 40)	N/A

A and B will make the probes and the queries JOIN(A,C) and JOIN(B,C) will be processed first. In the second phase, we have:

	Phase 2 Weight - Heuristic 2	Heuristic 1
Relation A:	40 - (Size(B) , 40)	1 - (B)
Relation B:	40 - (Size(A) + Size(D), 20+20)	2 - (A,D)
Relation C:	0	0
Relation D:	40 - (Size(B), 40)	1 - (B)

All the joins of C are processed and C has a weight 0, while A, B, and D has the weight 40. If we apply only *Heuristic 2*, any of three relations A, B and D is selected to be built. But, if we also consider *Heuristic 1*, B will be the one to be built since it interacts with 2 relations while the others interact with only one. After B is built, the remaining queries will be processed.

The aim of using this parameter is similar to that of *Heuristic 1*, but in this case, the relative sizes of relations are also important. There is a subtle memory utilization: the larger relations are enforced to make probes and leave the memory as soon as possible. Therefore besides response times, the memory utilization is also considered.

Heuristic 3 - Smallest Size:

This heuristic is directly related with the size of the relations. The relation with the smallest size is selected to build. In the given example the sizes of A, B, C, and D are respectively 20, 40, 10, and 20. Thus, C will be the built relation. A and B will make the probes. Then A or D is built. The built one will be probed by B. Finally, the relation A or D that was not built in the previous step, will be built and be probed by B. Here, the only performance consideration is the savings in memory utilization.

It is also possible that these three heuristics can be used together. Consider the following types of algorithms:

Mixed Algorithms:

Various combinations of heuristics could be considered in scheduling algorithms. Some examples can be:

weight / size, (*Heuristic 2 / Heuristic 3*)
number of relation probes / size, (*Heuristic 1 / Heuristic 3*)
number of relation probes x weight,
c1 x number of relation probes + c2 x weight + c3 x size
(for some constant c1,c2,c3 values)

etc.

Cascaded Algorithms:

A possible scheduling algorithm can be:

If the weights of two relations are the same, use the number of relations heuristic. If they are also the same, build the small sized relation.

2.2. How and When to Probe?

The hash tables of the built relations are probed by the other relations. But there can also be different considerations about the probing.

Heuristic 4 - Immediate Probing:

The probing can be done immediately after the relation is built. While explaining the Building Heuristics, this type of probing has been assumed.

Let us consider the same example with a limitation in the memory size:

Set of Queries:

JOIN(A,B), JOIN(A,C), JOIN(B,C), JOIN(B,D)

with the following sizes

Relation A: 20 pages, Relation B: 40 pages,
Relation C: 10 pages, Relation D: 20 pages,

and the memory size is 50 pages.

If we use *Heuristic 1* for choosing the relation to build, as you could remember the order of the built relations will be B and A (or C) . If we use *Heuristic 4* for probing after the hash table of B is built, it will be immediately probed by A, C, and D. And as a result, JOIN(A,B), JOIN(B,C) and JOIN(B,D) will be processed. After probes are completed, the hash table of B will not be used again, so it is flushed from the memory. Then, the hash table for A is formed and JOIN(A,C) is processed by probing the hash table of A with C.

This type of probing has some advantages as well as some disadvantages. One of the advantages arises from the timely response given to some of the users. The user gets his/her answer immediately after one of the relations in his/her query is built. Another advantage is related to the memory management; the memory contains at most one hash table and is always available for other operations.

The disadvantage arises from the optimization point of view. For the above case, B is probed by A, C, and D. After this, the hash table for A is built and is probed by C. If the hash tables of both A and B were available at the same time, the hash values for C would be calculated only once and C would probe both tables at the same time. (Here we assume that the join attributes of C are the same in both queries.) With *Heuristic 4*, we will not be able to take this advantage and the processing time will increase. Another disadvantage arises again from the user side. It has been mentioned as an advantage that, some of the users get their results as soon as a hash table related to their query is obtained. But, the owner of the last handled query will wait until all the previous queries are processed.

Heuristic 5 - Probe with No Need for a Flush:

The probing phase can wait until all the relations are built. This can be achieved if the memory is large enough to contain all the hash tables of the built relations.

Let us use *Heuristic 3* for choosing which relation to build. As you could remember the order of the relations to build will be C, A and D. And, let us use *Heuristic 5* for probing. C will use 10 pages of the memory, while A and D use 20 pages each. And as a result, the memory will be totally occupied. After this, all the probes will be done.

This heuristic is not complete, because most of the time the memory is not large enough to contain all the hash tables of the built relations. The *Heuristics 7*, *9* and *10* are alternatives to handle this case, and they all make use of *Heuristic 5* for the case that the memory is large enough to handle all the hash tables. Before discussing those heuristics, let us focus on another issue:

Heuristic 6 - Probe while Building:

When there are more than one relation built in the memory, we can have the following situation : the relation that will be built next can also be able to probe one or more hash tables residing in the memory.

Consider again the same example. By using Heuristic 3 for choosing which relation to build, we will have the order of the built relations as C, A, and D. First C is built. After that, it is time for A to be built. While building the hash table of A, the hash values of A can also probe C and finalize JOIN(A,C). If this is not done, following the build phase of all relations, the hash values of A will again be computed to probe C.

After A is built, the hash table for D is built. And finally B will probe all the built relations A, C, and D; and JOIN(A,B), JOIN(B,C), JOIN(B,D) are processed.

So far, there was no need for a flush to occur, as it was assumed that the amount of memory is large enough. Let us have a more complex example for the rest of heuristics with flush operation:

Example 2:

JOIN(A,B), JOIN(A,C), JOIN(B,C), JOIN(B,D), JOIN(A,E), JOIN(C, E)

with the following sizes

Relation A: 20 pages, Relation B: 40 pages, Relation C: 10 pages,
Relation D: 20 pages, Relation E: 30 pages
Total Memory Capacity : 40 pages

Heuristic 7 - Partial Probe:

The probing phase can wait until there is no place in the memory for the hash table of the next relation to be built. To build the next relation, we need to flush at least one hash table of a relation from the memory.

Let us use a cascading algorithm for choosing the order of the relations to build: Apply *Heuristic 1*; if the values obtained for some relations are the same, apply *Heuristic 2*; if some relations still have the same values for *Heuristic 2* , apply *Heuristic 3*.

We have the following values for each of the relations for the three heuristics:

	Heuristic 1	Heuristic 2	Heuristic 3
Relation A:	3 - (B, C, E)	80 - (Size(B)+Size(C)+Size(D),40+10+30)	20
Relation B:	3 - (A, C, D)	50 - (Size(A)+Size(C)+Size(D),20+10+20)	40
Relation C:	3 - (A, B, E)	90 - (Size(A)+Size(B)+Size(E), 20+40+30)	10
Relation D:	1 -(B)	40 - (Size(B), 40)	20
Relation E:	2 -(A, C)	30 - (Size(A)+Size(C), 20+10)	30

And the order of relations for being built will be: C, A and D. We have a memory of 40 pages. When C is build, 30 pages of memory is left. The next relation A is 20 pages. Therefore, we have enough space for A. A will be built using *Heuristic 6*: it will probe C while being built and finalize the query JOIN(A,C). After A is built, the amount of left memory is 10 pages. When it is time to build D, there is no enough space for this relation as its size is 20 pages and there is only 10 pages of available memory. Therefore, we have to flush relations till there is enough space for Relation D. If A is flushed, the available memory size will increase to 30 pages which is enough for Relation D. So, let us flush A.

Before flushing A, relations B and E will probe the hash table of A. Both B and E will also need to probe the hash table of C. To probe A, the hash values of B and E will be computed and to probe C at a later time, recalculating these values will only be a waste of time. So, the probing of A and C by B and E will be done at the same time. This can be specified as another heuristic:

Heuristic 8 - Probe More than one Hash Table: If the probing of different hash tables at the same time is possible, finalize the probes to avoid recalculating the hash values at a later time.

By using this heuristic, the queries JOIN(A,B), JOIN(B,C), JOIN(A,E), JOIN(C,E) will be finalized at the same time. After this, A will be flushed and then D will be built. C can also be flushed since there is no more relations to probe it. As the next step, B will probe D.

Heuristic 9 - Flush All:

This heuristic works the same as *Heuristic 7* until the flush time. When the flush of a relation is needed, all the relations in the memory are flushed.

Let the order of relations for being built be: C, D, and A. We have a memory of 40 pages. When C is built, 30 pages of memory is left. The next relation D is 20 pages. That is to say we have enough space for D. D will be built, and since there is no join operation on D and C, *Heuristic 6* is not applicable at this step. After D is built, the amount of available memory is 10 pages. When it is time to build A, there is no enough space for this relation as its size is 20 pages. So we have to flush some relations. By this heuristic all the relations are flushed. Before the flush of D, it is probed by B. B probes C at the same time using *Heuristic 8*. Before the flush of C it is also probed by A and E. After the flush operations, JOIN(A,C), JOIN(B,C), JOIN(B,D), JOIN(C,E) queries are completed and the whole memory is available. As the next step, A is built and it is probed by B and E. As a result, all the queries are completed.

Let us now consider the sequence of operations that uses the same build order and performs probing by using *Heuristic 7*. The workflow is exactly the same until the need for a flush. According to *Heuristic 7*, flushing only D is enough. When D is flushed, the available memory size will increase to 30 pages and 20 paged A can fit into the memory. Before the flush, D is probed by B. B probes C at the same time by *Heuristic 8*. The queries JOIN(B,C), JOIN(B,D) are processed. After the probe of B, D is flushed. C still remains in the memory, since the other probes of C, namely probing by A and E are not completed. The next step is building relation A. A probes C during the building phase using *Heuristic 6*. Then E probes A and C at the same time (*Heuristic 8*), and A is also probed by B. All the queries are processed.

Heuristic 9 has a disadvantage over *Heuristic 7*. The disadvantage arises from not being able to use *Heuristic 8*. For the above example, using *Heuristic 9* causes the hash values of E to be processed twice. In very complex query environments, i.e., real life examples with hundreds of relations, this re-processing can cause an important add-on to the query time. Therefore, *Heuristic 7* can be expected to perform better than *Heuristic 9*.

2.3. Which Relation to Flush?

After deciding which relation to build, the next step is to build the hash table of the selected relation, if of course there is enough space in the memory for the hash table. If there is a memory limitation, then some of the relations should be flushed out of the memory.

Here emerges another question. If there are more than one relation in the memory, which relation should we flush? Flushing all the relations in the memory vs. flushing until there is enough space for the next relation to be built was discussed in Section 2.3. If the second alternative is used, what should be done to detect which relation to flush? This new issue is also another important criteria related to query optimization.

Let us again use Example 2 with the build criteria used in Heuristic 7. The flow of the query scheduling was as follows:

The order of relations for being built are: C, A and D. We have a memory of 40 pages. When C is build, 30 pages of memory is left. The next relation A is 20 pages. That is to say we have enough space for A. A will be built as in Heuristic 6: it will probe C while being built and finalize the query JOIN(A,C). After A is built, the amount of memory left is 10 pages. When it is time to build D, there is no enough space for this relation with its size of 20 pages and 10 pages of available memory. So, we have to flush relations till there is enough space for Relation D.

Here comes the question: should we flush C or A?

Heuristic 10: Flush the Larger Sized Relation:

Relation A has a size of 20 while C has 10. According to this heuristic A should be flushed first. When A is flushed, the available memory will increase to 30 pages which is enough for Relation D. So there is no need to flush C to build D.

Heuristic 11: Flush According to the Join Set of the Next Relation to be Built:

Let us have another example. We have the relations with the following properties:

	Number of relations Interacted	Join Set (Relations Interacted with)	Size
Relation A:	4	{B, C, E, G}	20
Relation B:	5	{A, C, D, F, G}	30
Relation C:	2	{A, B}	20
Relation D:	1	{B}	20
Relation E:	1	{A}	20
Relation F:	2	{B, G}	30
Relation G:	3	{A, B, F}	20

Memory Size: 50

Let the order of building the relations be : B, A, G.

Let us have the relations A and B in the memory. And it is time to build G. If we use *Heuristic 10*, we will flush B. By this way, C and G will probe both A and B. D and F will probe B. After this, G will be built. Since there is no relation need to be built, the probes are completed. A will be probed by E, and G will be probed by F.

As you can notice, the hash values of F are computed twice, once for probing B before its flush, and once for probing G. Instead of flushing B, if we flush A all the hash values will be completed only once. Before the flush of A, C and G will probe both A and B. A will also be probed by E. After this, G will be built. Then the probes are completed and F probes both B and G, and E probes B.

In *Heuristic 11* , before selecting the relation to flush, the relations that are in the join set of the next relation in the build queue, and the relations that are in the join set of the relations in the memory are compared.

In the example, the next relation to build is G, with the join set {F}. The relations in the memory are A and B with the join sets of {C, E, G} and {C, D, F, G}, respectively. The intersection of the join sets of A and G is empty set, while the intersection of the join sets of B and G is {F}. Since the number of elements in the intersection set of A and G, is less then the intersection set of B and G, A will be flushed out of the memory.

Before using this technique, some investigation should be done. For the above example, flushing either A or B is enough for G to be built. If we do not make any pre-investigation, and directly use *Heuristic 11*, there won't be any advantage. For the above example, let the join sets be the same but the sizes be different. Let B be 40, A be 10 and G be 20. Here flushing only A, is not enough for G to be built. But if we directly use *Heuristic 11*, A will be flushed and after this since the memory is not enough for G, B will also be flushed before building G.

Heuristic 12 - Consider also the Next Batch

While processing the last queries in a batch, the hash tables in the memory can also be used by the next batch. At the end of the processing of a batch of queries, the probes to the in-memory relations will be completed, but the relations will not be flushed from the memory. When it is the time for the next batch, the in-memory relations will be assumed to be the first built relations.

Let us consider the first batch as the above group, we have A and G in the memory as the final step. Let us assume that without using this heuristic, the build queue of the second batch is as follows: B, E, G. But by *Heuristic 12*, since G and A are already in the memory, the build queue will change according to the new weights without considering G and A.

3. Algorithms

Before discussing the algorithms, let us first go over some important data structures used. The queries are formed randomly. Each query is a row in the ***query-table***. An ***inter-relation table*** is formed by using *query-table*. Before its usage let us give an example:

Rel-1	Rel-2
R1	R2
R3	R1
R4	R1
R7	R5
R5	R6
R6	R5
R5	R7
R1	R7
R2	R3
R7	R4

query-table

R1	**0**						
R2	1	**0**					
R3	1	1	**0**				
R4	1	0	0	**0**			
R5	0	0	0	0	**0**		
R6	0	0	0	0	1	**0**	
R7	1	0	0	1	1	0	**0**
	R1	R2	R3	R4	R5	R6	R7

inter-relation table

Inter-relation table is used in all the algorithms. At the initial state, this table shows whether there is a join operation between two relations. In the example, since there is a query as JOIN(R1,R2), the value in the inter-relation table for the cell R1-R2 is 1. In the same manner, since there is no join between R2-R4, the corresponding value in the inter-relation table is 0. Join is a commutative operation. For both JOIN(R1,R2) and JOIN(R2,R1), the operation is completed by building either R1 or R2, and probing with the other one. For this reason, the corresponding value for R1-R2 in the inter-relation table is set by one of the mentioned joins.

Besides these two tables, we have some more tables that are algorithm specific. These tables are:

size-table: This table shows the size of each relation. It is formed at the initialization part and is not changed by any algorithm (i.e., it is a static table).

weight-table: This table shows the sum of the sizes of relations that a relation joins with. It is formed according to the join-table and continuously modified as the joins are completed.

number-of-relations-table: This table shows the number of relations that a relation joins with. This table is a dynamic table as weight-table and modified as the queries are completed.

3.1. Main Algorithm

The main algorithm for the join operation is directly related with the probe phase. All the algorithms start with an ***initialize*** session. In this session, the tables are initialized. Then the query set is formed randomly by ***getbatch***. Then the other parts of the main algorithm ***build-criteria***, ***build***, ***flush-criteria***, ***probe*** and ***flush*** take place according to the probe workflow. Let us go into procedural details.

Build-criteria: This part is directly related with the build heuristic.

Build: The main build phase.

Flush-Criteria: This part is directly related with the flush heuristic.

Probe: The main probe phase. As mentioned before, the probe heuristic forms the main outline. This algorithm only deals with the probe details, not the heuristics.

Flush: The flush job is done in this phase.

The main algorithm without inter-relation optimization is as follows:

```
initialize
getbatch
for each join query
    begin
        build the hash table of the first relation
        probe the hash table with the hash values of the other relation
        flush the hash table of the relation from the memory
    end
```

Since there is no optimization, the inter-relation table, weight-table, number-of-relations-table, build-criteria algorithm, and flush-criteria algorithm are not used. For each query one of the relations is built, the other relation probes the hash table of the built relation, and after the probe the hash table is flushed from the memory.

To see the outline of the optimized version, let us first focus on the probe heuristics.

3.2. Main Outline According to Probe Heuristics

In this part only join heuristics will be discussed. The real join process will be described in detail in Section 3.5.

3.2.1. Immediate Probe

The probing can be done immediately after the relation is built as discussed in detail with *Heuristic 4*. The main algorithm now takes the following form:

```
initialize
getbatch
while there exist join queries to process
    begin
        select the relation to be built by build-criteria
        build the hash table of the relation selected by build-criteria
        complete all the probes to the relation
        flush the relation from the memory
    end
```

The algorithm resembles the algorithm with no inter-query optimization. One of the relations is built, then this relation is probed immediately and flushed. But here, there is an important difference, the built relation is probed by more than one relation. And finally the relation is flushed. This loop continues until all the joins are processed.

The relation to be built is detected by the help of build-criteria. The details about the phases build and build-criteria can be found in Section 3.3. The probe phase is explained in Section 3.4. In this algorithm, since there is only one relation in the memory, there is no need for an algorithm for deciding which relation to flush. So flush-criteria is not used. The details about the flush procedure can be found in Section 3.5.

3.2.2. Total Probe

This algorithm applies *Heuristic 8*, i.e., when there is need for a flush, all the relations in the memory are flushed.

The algorithm is as follows:

```
initialize
getbatch
while there exist join queries to process
    begin
        select the relation to be built by build-criteria
        while the memory is not totally occupied
                and there are still queries left to be processed
                begin
                    build the hash table of the relation selected by built-criteria
                    select the relation to be built by build-criteria
                end
        complete all the probes to all the relations in the memory
        flush all the relations from the memory
    end
```

Here the maximum number of relations are built as long as the memory is available, by the condition *while the memory is not totally occupied.* When the memory is totally occupied, to continue to the operation, (i.e., to be able to build the other relations), all of the relations should be flushed from the memory. The aim of the flush is to leave space for the next relation to be built. Before the flush, all the probes that will be made to the relations are completed. And finally all the relations in the memory are flushed. As in immediate flush, there is no need for flush-criteria. This loop continues until all the joins are processed.

3.2.3. Partial Probe

This algorithm applies *Heuristic 7*, i.e., the probing phase can wait until there is no space in the memory for the hash table of the next relation to be built. To build the next relation, we need to flush at least one relation from the hash table.

```
initialize
getbatch
while there exist join queries to process
    begin
        select the relation to be built by build-criteria
        while the memory is not totally occupied
                and there are still queries left to be processed
                  begin
                        build the hash table of the relation selected by built-criteria
                        select the relation to be built by build-criteria
                  end
        select the relation to be flushed from the memory by flush-criteria
        complete all the probes to the selected relation which will be flushed
        flush the relation from the memory
    end
```

To select which relation to flush, the function flush-criteria is used. After the relation to be flushed is detected, all the probes that will be made to this relation are completed. And finally the relation is flushed. This loop continues until all the joins are processed.

3.3. Initialization Session

Before going into details of build, probe and flush operations, let us first focus on two initialization procedures: **initialize** and **getbatch**. Here are the algorithms for *Heuristic 2 - Build by Largest Number of Tuples that will Probe.*

Initialize performs the classical initialization job. The algorithm is as follows:

```
For all relations
    Set the size of the relation, in the size-table
    Set the weight value of the relation to 0, in the weight-table
Reset the query-table
Reset all the variables
```

Following the execution of this function the contents of the main tables, namely query-table, inter-relation table, size-table, and weight-table will be:

Rel-1	Rel-2
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0

query-table

R1	**0**						
R2	0	**0**					
R3	0	0	**0**				
R4	0	0	0	**0**			
R5	0	0	0	0	**0**		
R6	0	0	0	0	0	**0**	
R7	0	0	0	0	0	0	**0**
	R1	R2	R3	R4	R5	R6	R7

inter-relation table

Relation	Size
R1	37
R2	43
R3	73
R4	24
R5	40
R6	45
R7	53

size-table

Relation	Weight
R1	0
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0

weight-table

Getbatch forms the query set to be processed. And the algorithm is as follows:

```
For each query (number of queries times)
    Form Relation1 of this query randomly
    Form Relation2 of this query randomly
    Update the inter-relation table
        (Set the value of inter-relation table entry for Relation1-Relation2 to 1)
    Update the weight-table
        (Add the size of Relation1 to the weight value of Relation2
         Add the size of Relation2 to the weight value of Relation1)
```

First the query-table is formed randomly, and the other tables are rebuilt as follows:

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	1	1	1	**0**			
R5	0	1	0	0	**0**		
R6	1	0	1	0	1	**0**	
R7	0	0	0	0	1	1	**0**
	R1	R2	R3	R4	R5	R6	R7

inter-relation table

Relation	Weight
R1	142
R2	64
R3	106
R4	153
R5	141
R6	203
R7	85

weight-table

weight of R1 = size(R3) + size(R4) + size(R6) = 73 + 24 + 45 = **142**
weight of R2 = size(R4) + size(R5) = 24 + 40 = **64**
weight of R3 = size(R1) + size(R4) + size(R6) = 37 + 24 + 45 = **106**
weight of R4 = size(R1) + size(R2) + size(R3) = 37 + 43 + 73 = **153**
weight of R5 = size(R2) + size(R6) + size(R7) = 43 + 45 + 53 = **141**
weight of R6 = size(R1) + size(R3) + size(R5) + size(R7) = 37 + 73 + 40 + 53 = **203**
weight of R7 = size(R5) + size(R6) = 40 + 45 = **85**

These two procedures are used by all the build/probe and flush algorithms except for consecutive flush. The related algorithms for this flush type are explained in Section 3.6.3.

3.4. Build Algorithms

The build phase of the join operation is composed of two main parts: **build-criteria** and **build**. By **build-criteria** one of the relations is chosen to be built. Let us begin with the *Heuristic 2 - Largest Number of Tuples that will Probe*.

3.4.1. Build by Largest Number of Tuples that will Probe

In this algorithm, the build-criteria chooses the relation with the largest weight. The algorithm is as follows:

```
Choose the relation with the largest weight
if the chosen relation is Relation0 then
    Set Completed Flag to True
else
    Increase the value of occupied by the size of the chosen relation
    Increase the value of hashno by 1.
```

Here *occupied* is the variable showing the total amount of occupied memory and *hashno* is the total number of hash tables built until that time. The relation with the largest value is chosen by the function **largest**.

```
Set the largest weight to 0
Set the relation to build to Relation0
for each relation
    if the weight of the relation is larger than largest weight
        set largest weight to the weight of the relation
        set relation to build, to the relation
```

In largest, the relation to build is initially set to 0. At the end of the function if the highest value is still 0, this means that all the relations have the weight zero, i.e., all the weight values were modified since all the queries have been processed.

For the above example first Relation6 is chosen since it has the largest weight (203). The hashno becomes 1, and the size of the occupied memory becomes 45, which is the size of Relation6.

Following the selection of a relation (say RelationX), the relation is built by the following **build** algorithm.

```
Set the value of inter-relation table for RelationX-RelationX to 2
if RelationY and RelationX have the value 1 in the inter-relation table
  For all relations - RelationY
    Change this value to 3
    Modify the weight value of RelationY by subtracting the size value of RelationX
 else if RelationY and RelationX have the value 3 in the inter-relation table
    Change this value to 4
Set the weight of RelationX to -hashno
Read the data related to RelationX from the disk
For each tuple
    Form the hash value of the key attribute
    Insert the tuple into the hash table
    For all relations - RelationZ
          if the RelationX has value 4 with RelationZ in the inter-relation table
             probe the hash table of RelationZ with the hash value of the tuple
```

After the execution of this procedure the inter-relation table is reformed. The value for RelationX-RelationX in the table becomes 2. Value 2 means that RelationX is built. Then for all the relations that join with RelationX, the value in the table becomes 3. Value 3 means that these relations can probe the hash table of RelationX. If the value is already 3 in the table, this means that the other relation is already in the memory, so RelationX can probe it. RelationX probes the hash table of the other relation, with its hash values. This is Probe while building and was explained as *Heuristic 6*. After this phase the tables are like:

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	1	1	1	**0**			
R5	0	1	0	0	**0**		
R6	**3**	0	**3**	0	**3**	**2**	
R7	0	0	0	0	1	**3**	**0**

inter-relation table

Relation	Size
R1	**97**
R2	64
R3	**61**
R4	153
R5	**96**
R6	**-1**
R7	**40**

weight-table

Weight(R1)	*142 - 45*	***97***	*Weight(R3)*	*106 - 45*	***61***
Weight(R5)	*141 - 45*	***96***	*Weight(R7)*	*85 - 45*	***40***
Weight(R6)	*-hashno*	***-1***			

3.4.2. Highest Number of Relations Interacted with

The algorithms here resemble the algorithms given in Section 3.4.1. In this heuristic, instead of using the weight table, we use the number-of-relations table. Here are the modified lines in each algorithm:

initialize:

```
For all relations
    ...
    Set the value of the relation to 0, in the number-of-relations-table
...
```

For the above example

relation	R1	R2	R3	R4	R5	R6	R7
# relations	0	0	0	0	0	0	0

getbatch:

```
For each query (number of queries times)
    ...
    Update the number-of-relations-table
        (Increment the value of Relation1 for the number-of-relations-table
         Increment the value of Relation2 for the number-of-relations-table)
```

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	R4	R5	R6	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	1	1	1	**0**			
R5	0	1	0	0	**0**		
R6	1	0	1	0	1	**0**	
R7	0	0	0	0	1	1	**0**

inter-relation table

relation	*#relations*
R1	3
R2	2
R3	3
R4	3
R5	3
R6	4
R7	2

#relations-table

largest:

```
Set the largest number-of-relations to 0
...
for every relation - RelationX
    if number-of-relations value of RelationX is larger than largest number-of-relations
        set largest number-of-relations to the number-of-relations value of RelationX
```

For the example Relation6 is chosen as in the first heuristic, since it has the largest value, 4 as the number of relations interacted with.

build-criteria:

```
Choose the relation with the largest number of relations
...
```

The value of the occupied memory is 45, the size of Relation6. The parameter hashno is set to 1.

build:

```
...
For all relations - RelationY
    if the RelationY and RelationX have the value 1 in the inter-relation table
        ...
        Decrement the value of RelationY in the #relations-table
    ...
Set the value of RelationX in #relations-table to -hashno
```

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	1	1	1	**0**			
R5	0	1	0	0	**0**		
R6	**3**	0	**3**	0	**3**	**2**	
R7	0	0	0	0	1	**3**	**0**

inter-relation table

Relation	#relations
R1	**2**
R2	2
R3	**2**
R4	3
R5	**2**
R6	**-1**
R7	**1**

#relations-table

3.4.3. Smallest Size

In this heuristic, there is no change in the **initialize**, **getbatch**, and **build** functions of Section 3.3.1. Instead of using **largest**, we use a function called **smallest**. By this function, the relation with the smallest size is chosen. Because of its similarity, the algorithm will not be presented here in detail. In this case, both the weight and size tables are used. After the weights are computed, the smallest sized relation is chosen to be built. The build operation is followed by the modification of the weight table exactly as in Section 3.3.1.

For the same example, Relation4 is chosen with its smallest size, 24.

relation	R1	R2	R3	R4	R5	R6	R7
size	37	43	73	24	40	45	53

And as a result, the tables will be as follows:

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	**3**	**3**	**3**	**2**			
R5	0	1	0	0	**0**		
R6	1	0	1	0	1	**0**	
R7	0	0	0	0	1	1	**0**
	R1	R2	R3	R4	R5	R6	R7

inter-relation table

Relation	Weight
R1	**118**
R2	**40**
R3	**82**
R4	**-1**
R5	141
R6	203
R7	85

weight-table

3.5. Probe Phase

As explained in Section 3.1., the main algorithm is directly related with the probe heuristics. This section explains the probe phase of the main algorithm.

```
Select the relation to be probed - RelationX
For all relations - RelationY
   if the value corresponding to RelationX and RelationY in inter-relation table is 3
      Read the data of RelationY from the disk
      Form the hash value for each tuple of RelationY
      For all relations - RelationZ (including RelationX)
         if the value in the inter-relation table for RelationY and RelationZ is 3
            join RelationY and RelationZ
               (probe the hash table of RelationZ with the values of RelationY
                write the matching tuples to the output buffer)
      For all relations - RelationZ
         Update the inter-relation table by setting the RelationY-RelationZ to 0
```

The relation to be probed, namely RelationX is chosen based on the probe heuristic. Then, first the Relations that RelationX joins with, namely RelationY's are detected. This means that RelationY will probe the hash table of RelationX. As mentioned by Heuristic 8 - Probe More than one Hash Table, if there are more than one hash table in the memory, then probing all tables at the same time is a factor that can increase the performance. So all the relations that have a join with RelationY and have hash tables already built in the memory, namely RelationZ's, are probed by RelationY. RelationX is also a member of RelationZ's.

For the above examples let us consider the two probe heuristic alternatives and complete the probe phase:

1- Immediate probing

For this case, let us use the number-of-relations heuristic for build, and use the final tables of Section 3.4.2. By using the number-of relations heuristic, Relation6 is chosen and its hash table is built. According to the immediate probing technique, Relation6 should be probed immediately after it has been built.

After the probe, the tables' contents will be:

Rel-1	Rel-2
R5	**R6**
R7	R5
R3	**R6**
R3	R4
R1	R3
R2	R5
R6	**R1**
R6	**R7**
R1	R4
R4	R2

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	1	1	1	**0**			
R5	0	1	0	0	**0**		
R6	**0**	0	**0**	0	**0**	**2**	
R7	0	0	0	0	1	**0**	**0**

inter-relation table

Relation	#relations
R1	**2**
R2	2
R3	**2**
R4	3
R5	**2**
R6	**-1**
R7	**1**

#relations-table

Relation1, Relation3, Relation5 and Relation7 probe Relation6. And the joins

JOIN(R6,R1), JOIN(R3,R6), JOIN(R5,R6), JOIN(R6,R7)

are completed. This procedure continues until all the queries are processed. In the second turn of the loop, Relation4 is chosen and built. Then Relation1, Relation2, and Relation3 will probe Relation4. After these operations, the #relations-table is as follows:

relation	R1	R2	R3	R4	R5	R6	R7
#relations	1	1	1	-2	2	-1	1

and the completed joins are:

JOIN(R1,R4), JOIN(R4,R2), JOIN(R3,R4)

As the next step, Relation5 is built since it has the maximum value in the table. Then Relation2 and Relation7 probe Relation5. Then, the table is like:

relation	R1	R2	R3	R4	R5	R6	R7
#relations	1	0	1	-2	-3	-1	0

and the completed joins are:

JOIN(R2,R5), JOIN(R7,R5)

Finally, Relation1 is built and probed by Relation3. JOIN(R1,R3) is processed. By this way all the values in the table are less than or equal to 0, and this shows that all the queries have been processed.

2- Probe All:

This alternative is more complex than Immediate Probing. In this case, the relations are built as long as the memory is available, and when there exists no enough memory space, all the probes to the built relations are completed. Let us use the *Maximum Number of Tuples that will Probe Heuristic* for build.

With this heuristic, Relation6 is chosen to be built. By building Relation6, the amount of occupied memory becomes 45. After this step, the maximum weighed relation becomes Relation4. Therefore, Relation4 is built. The occupied value is increased to 69, by the size of Relation4.

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	**R4**	R5	R6	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	**3**	**3**	**3**	**2**			
R5	0	1	0	0	**0**		
R6	3	0	3	0	3	**2**	
R7	0	0	0	0	1	3	**0**

inter-relation table

Relation	Size
R1	**73**
R2	**40**
R3	**37**
R4	**-2**
R5	96
R6	-1
R7	40

weight-table

The next relation to be built is Relation5. But its size is 40 and cannot fit into memory. So it is the probe time. The queries JOIN(R1,R4), JOIN(R4,R2), JOIN(R3,R4), JOIN(R6,R1), JOIN(R3,R6), JOIN(R5,R6), JOIN(R6,R7) are processed.

The tables are as follows:

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	**R4**	R5	R6	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	**0**	**0**	**0**	**2**			
R5	0	1	0	0	**0**		
R6	**0**	0	**0**	0	**0**	**2**	
R7	0	0	0	0	1	**0**	**0**

inter-relation table

Relation	Size
R1	**73**
R2	**40**
R3	**37**
R4	**-2**
R5	96
R6	-1
R7	40

weight-table

After this operation memory is totally flushed. Relation5 and Relation1 are built and again probed completely. The other queries JOIN(R2,R5), JOIN(R7,R5), JOIN(R1,R3) are processed.

In both alternatives there is no need for a criteria in detecting which relation to flush. But before going into details of *Heuristic 7- Probe After Flush,* we should first investigate the flush algorithms.

3.6. Flush Algorithms:

When there is no place for the hash table of the next relation, at least one of the relations in the memory should be flushed from the memory. Flush operation is composed of two main parts: **flush-criteria** and **flush**. Between these two phases **probe** operation is completed.

Before the flush-criteria, let us focus on the **flush** phase.

```
Move the chosen RelationX from the memory
Decrease the occupied value by the size of RelationX
Set the value of inter-relation table for RelationX-RelationX to 0
Set completed to 0
For all relations- RelationY-RelationZ
        if the value of inter-relation table for RelationY-RelationZ is more than 0
                increment the value for completed
If completed is larger than 0
    set the completed value to 0
else
    set the completed value to 1
```

As easily seen, the flush function not only flushes the RelationX , but also detects whether there are more queries to process by setting the completed value. If completed is equal to 1 at the end, this means that all the queries have been processed. Now, let us focus on flush algorithms.

3.6.1. Largest Sized Relation

The procedural description of this heuristic is:

```
Set RelationX to Relation0
for each relation - RelationY
        if weight of RelationY is less than 0 and size of RelationY is larger than RelationX
                Set RelationX to RelationY
```

Let us consider the example of Section 3.3.1., and use weight-table for build, partial probe for probe, and largest size for flush. In 3.3.1., as the first step, Relation6 was the built relation, so the occupied memory value was 45. After this, we choose Relation4 with the highest weight value. By this operation the total occupied memory becomes 69.

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	**3**	**3**	**3**	**2**			
R5	0	1	0	0	**0**		
R6	**3**	0	**3**	0	**3**	**2**	
R7	0	0	0	0	1	**3**	**0**

inter-relation table

Relation	Weight
R1	**73**
R2	**40**
R3	**37**
R4	**-2**
R5	96
R6	**-1**
R7	40

weight-table

The next relation to build is Relation5. But since Relation5 cannot fit into memory, one of the relations in the memory should be flushed. To detect which relation to flush, the relations with the negative weight values are compared according to their sizes. Here we have Relation4 and Relation6 in the memory. Relation6 will be the flushed relation with its larger size.

Then, by the probe operation the 3 values related to Relation6 is selected from the inter-relation table: Relation1, Relation3, Relation5, and Relation7. For all these relations, it is detected whether they also join with Relation4 and it is found that Relation1 and Relation3 also probe Relation4. So all the related probes are completed, and then Relation6 is flushed from the memory. By this way the queries JOIN(R1,R4), JOIN(R6,R1), JOIN(R3,R4), JOIN(R3,R6), JOIN(R5,R6), JOIN(R6,R7) are completed.

Rel-1	Rel-2
R5	**R6**
R7	R5
R3	**R6**
R3	**R4**
R1	R3
R2	R5
R6	**R1**
R6	**R7**
R1	**R4**
R4	R2

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	0						
R2	0	0					
R3	1	0	0				
R4	**0**	3	**0**	**2**			
R5	0	1	0	0	0		
R6	**0**	0	**0**	0	**0**	**0**	
R7	0	0	0	0	1	**0**	0

inter-relation table

Relation	Weight
R1	73
R2	40
R3	37
R4	-2
R5	96
R6	**0**
R7	40

weight-table

After the flush, the occupied value is decreased to 24. Now only the hash table of Relation4 is in the memory. Relation5 is able to be built. When Relation5 is built the occupied value increases to 64, and the tables are like:

Rel-1	Rel-2
R5	*R6*
R7	R5
R3	*R6*
R3	*R4*
R1	R3
R2	R5
R6	*R1*
R6	*R7*
R1	*R4*
R4	R2

query-table

	R1	R2	R3	R4	**R5**	R6	R7
R1	0						
R2	0	0					
R3	1	0	0				
R4	0	3	0	2			
R5	**0**	**3**	**0**	**0**	**2**		
R6	0	0	0	0	**0**	0	
R7	0	0	0	0	**3**	0	0

inter-relation table

Relation	Weight
R1	73
R2	**0**
R3	37
R4	-2
R5	-3
R6	0
R7	**0**

weight-table

The next relation to build is Relation1, but it cannot fit into the memory. Again one of the relations in the memory should be chosen. The relations that are in the memory are Relation4 and Relation5. Since Relation5 is larger than Relation4, it is the flushed one.

Relation5 will be probed by Relation2 and Relation7. From these relations Relation2 can also probe the other relation in the memory, namely Relation4.

Rel-1	Rel-2
R5	*R6*
R7	**R5**
R3	*R6*
R3	*R4*
R1	R3
R2	**R5**
R6	*R1*
R6	*R7*
R1	*R4*
R4	**R2**

query-table

	R1	R2	R3	R4	**R5**	R6	R7
R1	0						
R2	0	0					
R3	1	0	0				
R4	0	**0**	0	2			
R5	0	**0**	0	0	**0**		
R6	0	0	0	0	**0**	0	
R7	0	0	0	0	**0**	0	0

inter-relation table

Relation	Weight
R1	73
R2	**0**
R3	37
R4	-2
R5	-3
R6	0
R7	**0**

weight-table

As a result queries, JOIN(R4,R2), JOIN(R2,R5), JOIN(R7,R5) are processed. Finally, Relation1 is built and is probed by Relation3. JOIN(R1,R3) is processed by this way. All the values in the inter-relation table and weight-table turn to 0.

3.6.2. Flush by Join-Set

Flush by Join-Set uses *Heuristic 12* for the flush operation. When there is a need for flush we have two main items: relations that are in the memory and a relation that cannot fit into memory (say, Relation**X**). This algorithm compares whether the relations that will probe Relation**X**, will also probe in-memory relations. For this algorithm we use a new table named common-table. Here is the algorithm:

```
set min-common to 10
set flag to 0
if RelationX is Relation0 then
   flush all the relations from the memory
else
  for all relations - RelationY
      set the common-table for RelationY to -1
  for all relations - RelationY
     if weight of RelationY is less than 0
         set the common-table value for RelationY to 0
         if the value in the inter-relation table for RelationX-RelationY is equal to 3
            increment the common-table value for RelationY
            increment flag
         for all relations - RelationZ
            if the value in inter-relation table for RelationY-RelationZ is equal to 3
            and the value in inter-relation table for RelationX-RelationZ is equal to 1
                 increment the common-table value for RelationY
                 increment flag
  if flag is larger than 0
     for all relations - RelationY
     if common-table value for RelationY is larger than 0
        and smaller than min-common
         set min-common to the common-table value of RelationY
         set the relation to flush, to RelationY
  else
    flush-largest
```

Here by the common-table which relation to flush is detected. Let us turn back to the example of Section 3.3.1. The tables are as follows:

Rel-1	Rel-2
R5	R6
R7	R5
R3	R6
R3	R4
R1	R3
R2	R5
R6	R1
R6	R7
R1	R4
R4	R2

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	**3**	**3**	**3**	**2**			
R5	0	1	0	0	**0**		
R6	**3**	**0**	**3**	**0**	**3**	**2**	
R7	0	0	0	0	1	**3**	**0**

inter-relation table

Relation	Weight
R1	**73**
R2	**40**
R3	**37**
R4	**-2**
R5	96
R6	**-1**
R7	40

weight-table

As explained in Section 3.6.1, after Relation6, Relation4 is built. The next relation to build is Relation5, but it cannot fit into memory. So flush by join-set takes place. The common table is first totally set to -1. The common table is as follows.

relation	R1	R2	R3	R4	R5	R6	R7
common	-1	-1	-1	-1	-1	-1	-1

Then the loops begin and first the value for Relation4 is changed to 1, since both Relation4 and Relation5 join with Relation2. Then the common-value of the other in-memory relation, Relation6 is changed to 2. First the value is increased to 1, since we have JOIN(R5,R6) as a query. Then the value is increased to 2, since both Relation5 and Relation6 join with Relation7. And the table contents become:

relation	R1	R2	R3	R4	R5	R6	R7
common	-1	-1	-1	**1**	-1	**2**	-1

Relation4 is chosen for flush, since it has the smallest value greater than 0. The probes to Relation4 are completed. Then, Relation5 is built. Relation5 will also probe Relation6 during the build phase. And the inter-relation table value will turn to 4 as explained in Section 3.3. By this way, JOIN(R5,R6), JOIN(R3,R4), JOIN(R1,R4), JOIN(R4,R2) will be processed.

Rel-1	Rel-2
R5	**R6**
R7	R5
R3	R6
R3	**R4**
R1	R3
R2	R5
R6	R1
R6	R7
R1	**R4**
R4	**R2**

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**0**						
R2	0	**0**					
R3	1	0	**0**				
R4	**0**	**0**	**0**	**0**			
R5	0	**3**	0	0	**2**		
R6	3	0	3	0	**4**	**2**	
R7	0	0	0	0	**3**	3	**0**

inter-relation table

Relation	Weight
R1	73
R2	**0**
R3	37
R4	**0**
R5	**-3**
R6	**-1**
R7	0

weight-table

3.6.3. Consecutive Flush

The consecutive flush uses two sets of query batches and it is fundamentally different from the other two heuristics. This heuristic is used in processing the last queries in a batch and has no effect in deciding which relation to flush during the execution. For this decision, it has to use one of the two alternatives explained above. Most of the steps related to this algorithm are performed in the initialization and getbatch sessions. The basic aim is to make use of the in-memory hash tables of one batch in the following batch. For this reason, significant modifications are needed in initialization, getbatch, flush, probe, and build sessions. The difference in the flush session is simply as follows:

```
Move the chosen RelationX from the memory
Decrease the occupied value by the size of RelationX
Set the value of inter-relation table for RelationX-RelationX to 0
Set completed to 0
...
```

changes to:

```
if the next relation is not the last relation to build
    Move the chosen RelationX from the memory
    Decrease the occupied value by the size of RelationX
    Set the value of inter-relation table for RelationX-RelationX to 0
Set completed to 0
```

By this way, at the end of a batch we still have relations that have hash tables in the memory. When it is time to process the next batch, what we do is **not** initializing the inter-relation table to all zeros and modify the table according to the new batch received by getbatch. The getbatch is described below:

```
For each query (number of queries times)
    Form Relation1 randomly
    Form Relation2 randomly
    if none of Relation1 and Relation2 is in the memory
        Set the value of inter-relation table for Relation1-Relation2 to 1
        Add the size of Relation1 to the weight value of Relation2
        Add the size of Relation2 to the weight value of Relation1
    else if at least one of Relation1 and Relation2 is in the memory
        Increment hashno
        Set the value of inter-relation table for Relation1-Relation2 to 3
        if Relation1 is in the memory
            Set the weight of Relation1 to -hashno
        if Relation2 is in the memory
            Set the weight of Relation2 to -hashno
```

After these tables are reformed according to this heuristic, the functions that use these tables should also be modified. The modification for build function is as follows:

```
if the value for RelationX-RelationX is equal to 0
    do the build job
```

The build job explained in Section 3.3. There is no need for the modification of the probe function. Let us turn back to the last phase of the above example which was nearly finalized in Section 3.6.2. If we continue with the example, the next relation to build would be Relation1 and by join-set algorithm the relation to flush would be Relation5. And the tables would be like:

Rel-1	Rel-2
R5	*R6*
R7	*R5*
R3	R6
R3	*R4*
R1	R3
R2	*R5*
R6	**R1**
R6	R7
R1	*R4*
R4	*R2*

query-table

	R1	R2	R3	R4	R5	**R6**	R7
R1	**2**						
R2	0	**0**					
R3	**3**	0	**0**				
R4	0	0	0	**0**			
R5	0	**0**	0	0	**0**		
R6	**4**	0	3	0	**0**	**2**	
R7	0	0	0	0	**0**	3	**0**

inter-relation table

Relation	Weight
R1	**-4**
R2	**0**
R3	**0**
R4	0
R5	**0**
R6	-1
R7	0

weight-table

After this step, since all the relations have weights less than or equal to 0, the probes will be computed. But by this consecutive flush technique, the relations will not be flushed from the memory.

Rel-1	Rel-2
R5	*R6*
R7	*R5*
R3	**R6**
R3	*R4*
R1	**R3**
R2	*R5*
R6	*R1*
R6	**R7**
R1	*R4*
R4	*R2*

query-table

	R1	R2	R3	R4	**R5**	R6	R7
R1	**2**						
R2	0	**0**					
R3	0	0	**0**				
R4	0	0	0	**0**			
R5	0	0	0	0	**2**		
R6	0	0	0	0	0	**0**	
R7	0	0	0	0	0	0	**0**

inter-relation table

Rel-1	Rel-2
R1	R2
R2	R3
R3	R1
R4	R5
R6	R1
R6	R7
R7	R1
R5	R4
R6	R4
R7	R6

next query-table

Now, it is time to begin processing the next batch shown with the ***next query-table*** above. Since we have Relation1 and Relation2 already in the memory, the calculations are made according to the consecutive flush - getbatch algorithm.

Rel-1	Rel-2
R1	R2
R2	R3
R3	R1
R4	R5
R6	R1
R6	R7
R7	R1
R5	R4
R6	R4
R7	R6

query-table

R1	**2**						
R2	3	**0**					
R3	3	1	**0**				
R4	0	0	0	**0**			
R5	0	0	0	3	**2**		
R6	3	0	0	1	0	**0**	
R7	3	0	0	0	0	1	**0**
	R1	R2	R3	R4	**R5**	R6	R7

inter-relation table

Relation	Weight
R1	**-1**
R2	**73**
R3	**43**
R4	**45**
R5	**-2**
R6	**77**
R7	**45**

weight-table

If we did not use this method and go forward as in the beginning, the memory would be totally available. First Relation1 would be built, and Relation4 would follow it. But, by this technique we do not lose time by rebuilding Relation1 and Relation5.

4. Simulation Model

This chapter briefly presents the database system model that we used to evaluate the performance of the algorithms. The model is based on a uniprocessor database system. It contains two physical resources shared by the queries: CPU and Disk. Table 1 provides the set of parameters used in specifying the configuration of the database system. The disk characteristics are those of the Fujitsu Model M2266 (1GB, 5.25") disk drive [3].

Number of CPU's	1	Disk Seek Factor	16 msec
CPU Speed	30 MIPS	Disk Rotation Time	16.667 msec
Memory	100 pages	Disk Settle Time	2 msec
Page Size	8 KB	Disk Transfer Rate	3.09 MB/sec
Tuple Size	400 Bytes		
Number of Disks	1		

Table 1 - Simulation Model Parameters

To evaluate the algorithms fairly, the CPU overhead of performing various operations during query processing should be taken into account. The CPU costs of the operations considered in our model are presented in Table 2. The parameter values are based on instruction counts taken from the Gamma prototype [3].

		instruction	ms
t-initselect	Initiate Select	20,000	0.66667
t-initjoin	Initiate Join	40,000	1.33333
t-term-j	Terminate Join	10,000	0.33333
t-term-s	Terminate Select	5,000	0.16667
t-apply	Apply a Predicate	100	0.00333
t-read	Time to Read a Tuple	300	0.01000
t-output	Time to Write a Tuple into Output Buffer	100	0.00333
t-probe	Probe Hash Table	200	0.00667
t-insert	Insert Tuple in Hash Table	100	0.00333
t-hash	Hash Tuple Using Split Table	500	0.01667
t-sio	Start an I/O	1,000	0.03333
t-cb	Copy a Byte into Memory	1	0.00003
t-ct	Copy a Tuple into Memory (a tuple = 400 bytes)	400	0.01333
t-cp	Copy a Page into Memory (a page = 8 KBs)	8192	0.27307

Table 2 - CPU costs of some operations

The time required for processing each join operation can be computed by using the parameter values. The time for reading a file with N pages can be calculated as follows:

read-file(x)	cpu-time	disk-time
start an I/O	0.03333	
seek the disk		16
rotational latency		16.667
for all pages of x (N pages)		
transfer the page to memory		2.52832*N
copy the page into memory	0.27307*N	
t-readfile() =	0.033+0.27307*N	32.667+2.52832*N

The time required for a build operation can also be computed for a relation of N pages. First, the relation to be built is read, then the build operation is performed for every tuple of the relation.

build(x)	cpu-time	disk-time
read-file(build) t-readfile() =	0.033+0.27307*N	32.667+2.52832*N
for all tuples of x (N * 8KB/400B)		
read tuple	0.01*N*8KB/400	
compute hash-value	0.01667*N*8KB/400	
copy the page into memory	0.00333*N*8KB/400	
t-build() =	0.033+0.88747*N	32.667+2.52832*N

The probe operation is the final step of the join. Again, the relations to probe are first read, then for every tuple in the relation the processing is completed. In the calculation of the time associated with the line "if join write to output buffer", we assume a join probability of 50%.

probe(x)	cpu-time	disk-time
for each relation to be probed		
read-file(probe)	0.033+0.27307*N	32.667+2.52832*N
for all tuples of x (N * 8KB/400B)		
initiate-join	1.3333	
read tuple	0.01*N*8KB/400	
compute hash-value	0.01667*N*8KB/400	
probe hash-table	0.00667*N*8KB/400	
if join write to output buffer	0.00167*N*8KB/400	
terminate-join	0.3333	
t-probe() =	1.6996+0.99007*N	32.667+2.52832*N

This probe time is calculated according to the Partial Probe algorithm.

The details of the described model were captured in a simulation program. This program was written in CSIM/C [28], which is a process-oriented simulation language based on the C programming language.

During the simulation of the algorithms we made some assumptions about our workload. The simplified workload consists purely of single hash-join queries. The reason of choosing single hash-join queries is for the sake of simplicity. For a more complex multi-join query, there are different execution strategies like bushy, left-deep and right-deep. Using such queries would not have allowed the separation of the effects of sharing from other query scheduling issues. The join selectivity in our queries is assumed to be 50%. Also, when we talk about two join operations with a common relation (e.g., JOIN(R1,R2) and JOIN(R1,R3)), we assume that the joins are performed on the same attribute of the common relation (R1).

These assumptions also took place in [3]. Having the same assumptions enabled us to compare the performance results of our heuristics against the results of theirs.

5. Performance Results

In this section we present the simulation results for the proposed heuristics. The values used in simulation experiments are given in Table 1. Ten different relations are considered in the experiments. These relations have a size of 20 - 70 pages, while the memory is composed of 100 pages. The sizes of the batches differ from 10 to 50. We calculate the mean performance results for a batch by using 10 consecutive batches. The query batches are formed randomly.

5.1. Comparison of the Probe Heuristics

We first present the comparative results of the three probe heuristics. Figure 1 shows the performance of the heuristics for batches of 10, 20, 30, 40, and 50 queries. All of the probe heuristics below use weight heuristic for build. Partial Probe uses consecutive flush with join-set as the flush heuristic.

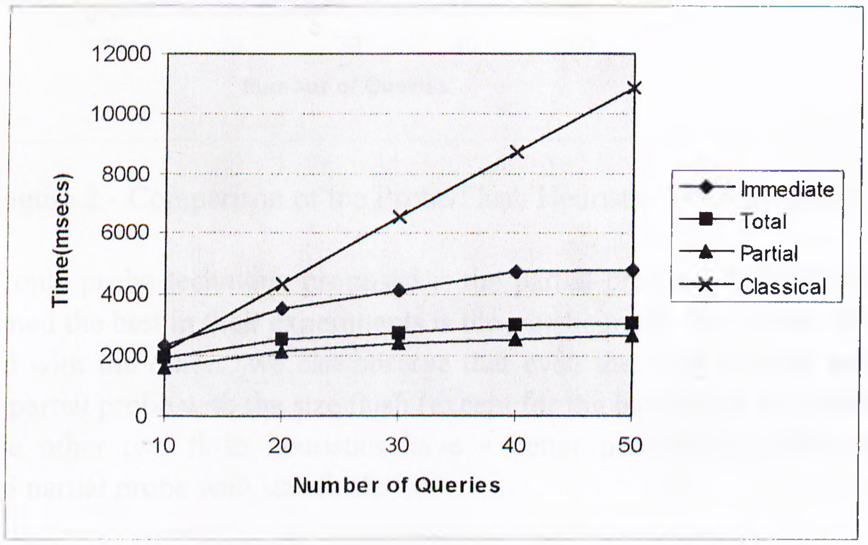

Figure 1 - Comparison of the Probe Heuristics (Performance)

As you can see in Figure 1, all three techniques perform much better than the classical method, which uses no inter-query optimization. The processing time for the classical method increases linearly with the increasing number of queries.

As mentioned before, the main outline of the join algorithm is directly related to the probe phase. For this reason, the classical method is compared to the probe heuristics. Classical method will not take place in the following figures. Here, all the probe heuristics use the weight heuristic for the build phase. Immediate probing and Total probing do not require any flush algorithm. In the figure the flush heuristic used with the partial probe is the consecutive heuristic, which is better than the other flush heuristics.

To see the performance differences between the probe heuristics better, let us present the results in a different scale. In Figure 2 you will also be able to find partial probe with the other flush heuristics.

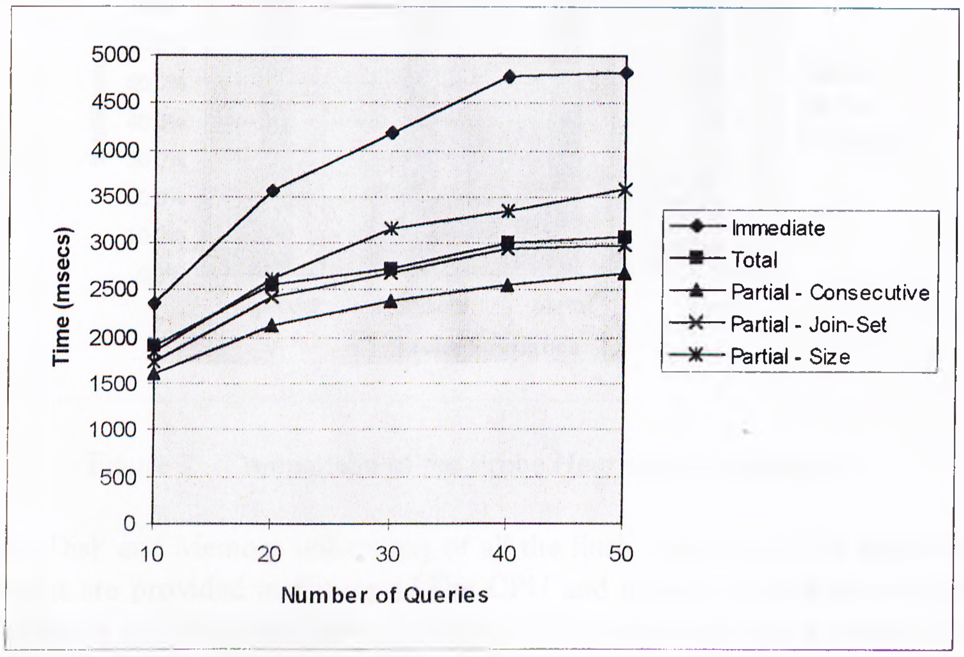

Figure 2 - Comparison of the Probe/Flush Heuristics (Performance)

In [3], the only probe technique proposed is the partial probing. The flush technique that performed the best in their experiments is the size heuristic. But when we compare this method with the others, we can observe that even the total probing technique is better than partial probe with the size flush (except for the batch with 10 queries). In all batches, the other two flush heuristics have a better performance than both total probing and partial probe with size flush.

Immediate probing has a worse performance than both total and partial probe techniques. The worse performance can be explained by the lack of application of *Heuristic 8*. As you can remember, *Heuristic 8* was probing more than one hash table at the same time. Immediate probing does not use this technique. Partial probe performs a little bit better than total probe as it applies *Heuristic 8* more often.

We also collected some statistics about the resource utilization by the heuristic. The best CPU utilization is observed with the partial and total probing. The memory utilization on the other hand is better with partial probing. The disk utilization corresponds to the read operations for the relations stored on the disk. Higher disk utilization means more frequent disk I/O, and thus increased response times for the queries. Therefore, the method that leads to the highest disk utilization (i.e. , classical) has the worst performance. The details can be found in Figure 3. The heuristics used with the probe algorithms are the same as the ones in Figure 1. The flush heuristic used with the partial probe is the size heuristic.

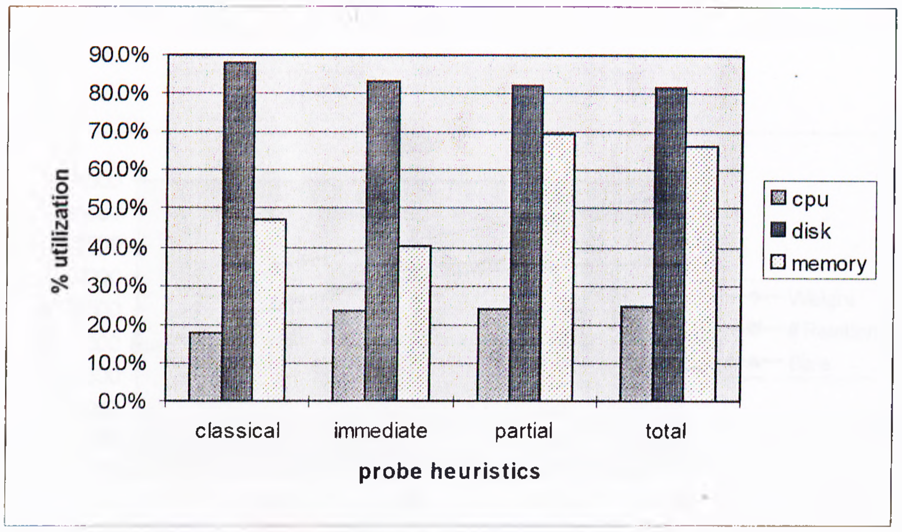

Figure 3 - Comparison of the Probe Heuristics (Utilization)

The CPU, Disk and Memory utilizations of all the flush heuristics in the partial probe environment are provided in Figure 4. The CPU and memory utilizations of join-set and consecutive are better than the size heuristic. This is because both heuristics use the in-memory relations more efficiently than the size heuristic. The disk utilizations seem to be about the same for all three heuristics.

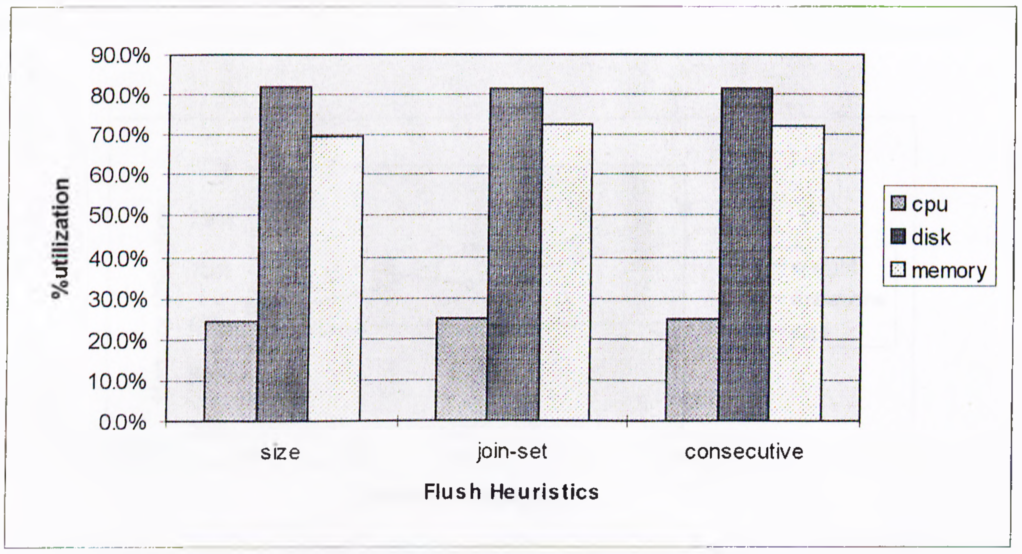

Figure 4 - Comparison of the Flush Heuristics (Utilization)

5.2. Comparison of the Build Heuristics

Figure 5 provides the comparative performance of different build heuristics. Although the performances of all three heuristics are close to each other, the #relations heuristic provides a little bit better performance than the others in large number of batches. The better performance can be contributed to the fact that by this heuristic minimum number of relations is needed to be built.

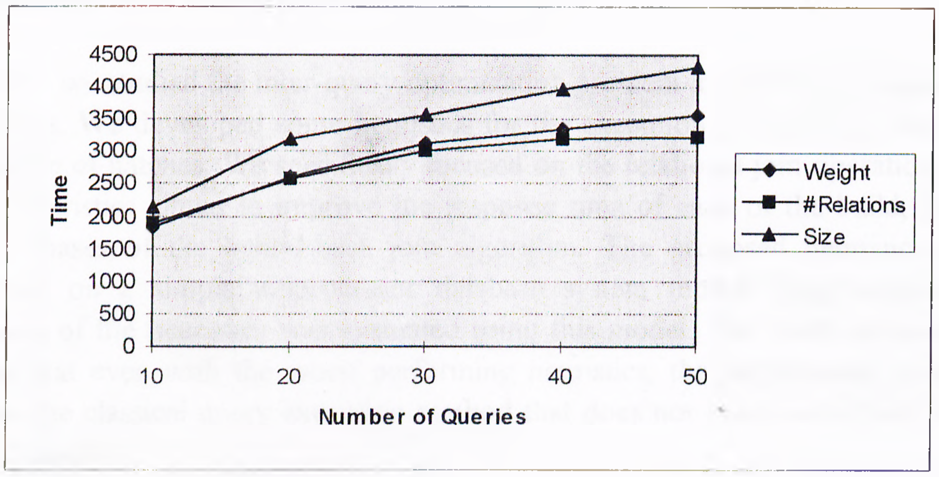

Figure 5 - Comparison of the Build Heuristics (Performance)

The memory utilizations of build heuristics are provided in Figure 6. The CPU and disk utilizations are nearly the same for all the three heuristics. The #relations heuristic has a better memory utilization than the others when the number of queries in the batch increases. Size is the worst method in terms of both the response time and the memory utilization.

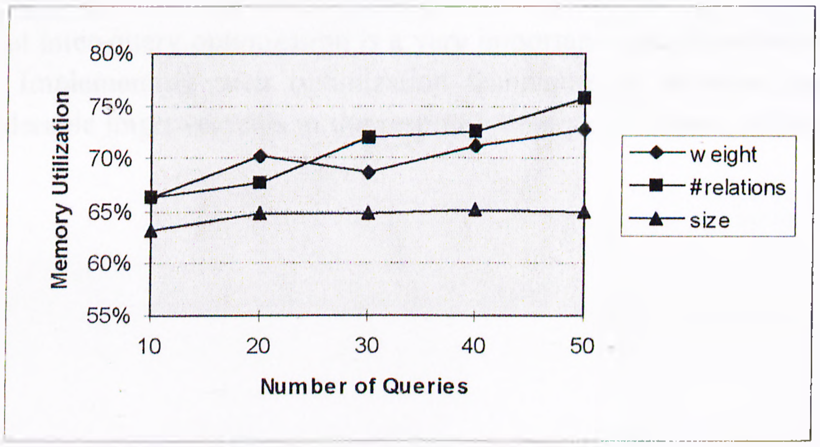

Figure 6 - Comparison of the Build Heuristics (Memory Utilization)

6. Conclusion and Future Work

In this thesis, we studied the inter-query optimization problem in multi-query execution environments. We developed some heuristics for the execution of queries in the form of a sequence of batches. We specifically focused on the relational join operation. The proposed heuristics aimed to improve the response time of each of the build, probe, and flush phases of the hybrid-hash join algorithm. The proposed heuristics were implemented on a simple uniprocessor database system model. The comparative performance of the heuristics was evaluated using this model. The most considerable result was that even with the worst performing heuristics, the performance is much better than the classical query execution method that does not make use of any inter-query optimization.

In developing the heuristics and implementing the simulation model, we made some simplifying assumptions to make our results comparable to others' and also to concentrate on certain steps of the query execution. We assumed all the queries consist of join operations and hybrid-hash method is used in processing joins. As a future work, queries with multi-operations, queries that have operations other than join, and join implementation techniques other than the hybrid-hash can be considered.

We studied the heuristics on a uniprocessor database system environment. The heuristics can also be implemented on multi-processor systems and parallel machines.

We believe that inter-query optimization is a very important topic that deserves further investigation. Implementing such optimization techniques in database systems can provide considerable improvements in the response time performance of the processed queries.

References

[1] Jones, J., "Parallel Database Concepts Presentation", AT&T, 1995

[2] Lu, H., Tan, K., and Shan, M., "Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory", Proceedings of the 16th VLDB Conf., Brisbane, Australia, 1990

[3] Mehta, M., Soloviev, V., and DeWitt, D., "Batch Scheduling in Parallel Database Systems", University of Wisconsin-Madison, 1993

[4] Selinger, P. G. et. al., "Access Path Selection in a Relational Database Management System", Proc. ACM, SIGMOD Conf., 1979

[5] Krishnamurthy, R., Boral., H., and C. Zaniolo, "Optimization of Nonrecursive Queries", Proceedings 12th VLDB Conf., August 1986

[6] Ioannidis, Y. and Kang, Y. C., "Randomized Algorithms for Optimizing Large Join Queries", Proc. ACM SIGMOD Conf., Atlantic City, NJ, May 1990

[7] Swami. a. and A. Gupta, "Optimization of Large Join Queries", Proc. ACM SIGMOD Conf., June 1988

[8] Schneider, D. and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines.", Proc. 16th VLDB Conf., Melbourne, Australia, Aug. 1990

[9] Chen, Ming-syan et. al., "Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries", Proc. 8th IEEE Data Engineering Conf., Phoenix, Az, Feb. 1992

[10] Brown, K., et. al., "Resource Allocation and Scheduling Issues for Mixed Database Workloads", Comp. Sc. Tech. Rep. TR 1095, University of Wisconsin-Madison, July, 1992

[11] Chakravarthy, U. S. et. al., "Semantic Query Optimization in Expert systems and Database Systems", Expert Database Systems: Proc. of 1st International Workshop, Melno Park, California, 1986

[12] Finkelstein, F., "Common Expression Analysis in Database Applications", Proc. ACM SIGMOD Conf., Orlando, FL, June 1982

[13] Hall, P.V., "Common Subexpression Identification in General Algebraic Systems", Tech. Rep. UKSC 0060, IBM United Kingdom Scientific Centre, Nov. 1974

[14] Sellis, T., "Multiple Query Optimization", ACM TODS 13(1), March 1988

[15] DeWitt, D., Naughton, J., and Schneider, D., "An Evaluation of Non-Equijoin Algorithms", University of Wisconsin, Madison, Feb 1991

[16] Kitsuregawa, M., and Ogawa, Y., "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer", Proc. of the 16th VLDB Conf., 1990

[17] Stathal, A., and Naughton, J., "Using Shared Virtual Memory for Parallel Join Processing", University of Wisconsin, Madison, 1993

[18] Wilschut, A., and Apers, P., "Dataflow Query Execution in a Parallel Main-Memory Environment", University of Twente, The Netherlands, 1993

[19] Pang, H., Carey, M. and Livny, M., "Partially Preemptible Hash Joins", SIGMOD Washington, DC, May 1993

[20] Lee, C., and Chang, Z., "Workload Balance and Page Access Scheduling For Parallel Joins In Shared-Nothing Systems", 9th Intern. Conf. on Data Eng., 1993

[21] Kitsuregawa, M., Nakano, M., and Takagi, M., "Query Execution for Large Relations on Functional disk System", 5th Intern. Conf. on Data Engr., 1989

[22] Severance, C., Pramanik, S., and Wolberg, P., "Distributed Linear Hashing and Parallel Projection in Main Memory Databases", Proc. of 16th VLDB Conf., 1990

[23] Lieuwen, D., Dewitt, D., and Mehta, M., "Parallel Pointer-Based Techniques for Object-Oriented Databases", 2nd Intern. Conf. on Parallel and Distr. Information Systems, 1993

[24] Soloviev, V., "A Truncating Algorithm for Processing Band-Join Queries", 9th Intern. Conf. on Data Eng., 1993

[25] Mikkilineni, K., and Su, S., "An evaluation of Relational Join Algorithms in Pipelined Query Processing Environment", IEEE, 1988

[26] Bittan, D., Boral, H., Dewitt, D., and Wilkinson., K., "Parallel Algorithms for the Execution of Relational Database Operations", University of Wisconsin, Madison, 1983

[27] Pang, H., Carey, M. and Livny, M., "Managing Memory For Real-Time Queries", University of Wisconsin, Madison, 1994

[28] CSIM User Manual