

**COMPLETE OVERFLOW MANAGEMENT
ALGORITHM TO JOIN
LARGE RELATIONAL TABLES**

**A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science**

By

Venkata Nagarjuna Rao Goli

**In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY**

**Major Department:
Computer Science**

June 2002

Fargo, North Dakota

UMI Number: 3103619



UMI Microform 3103619

Copyright 2003 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

North Dakota State University
Graduate School

Title

Complete Overflow Management Algorithm

to Join Large Relational Tables

By

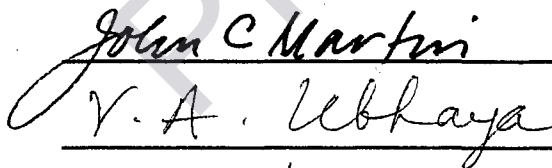
Venkata Nagarjuna Rao Goli

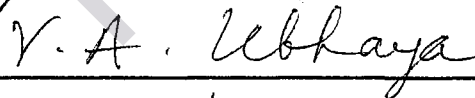
The Supervisory Committee certifies that this *disquisition* complies with North Dakota State University's regulations and meets the accepted standards for the degree of

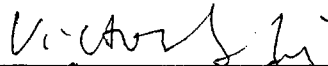
DOCTOR OF PHILOSOPHY

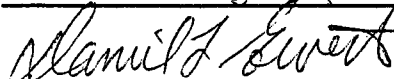
SUPERVISORY COMMITTEE:


Chair


John C. Martini


V. A. Ubhaya

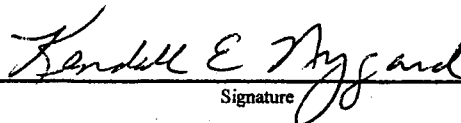

Victor Li


Manish Grewal

Approved by Department Chair:

6/27/02

Date


Signature

ABSTRACT

Goli, Venkata Nagarjuna Rao, Ph.D., Department of Computer Science, College of Science and Mathematics, North Dakota State University, June 2002. Complete Overflow Management Algorithm to Join Large Relational Tables. Major Professor: Dr. William Perrizo.

A new hash based algorithm, the Complete Overflow Management join Algorithm, (COMA), was proposed as a resolution for collisions in hash table and main memory bucket overflows during hash join processing. Collisions and bucket overflows are caused by skew in the data. Changes in the available main memory size cause bucket overflows. Collisions and bucket overflows during hash join result in excessive disk I/O. Traditional join algorithms, such as Join Indices, Hybrid hash, and Domain Vector hash, assume that the data distribution is normal. They collapse when the hashing algorithm employed affects data distribution. COMA sets aside a page of memory to copy the overflowing records. In addition, COMA employs orthogonal hashing functions to avoid collisions and rehashes the records to avoid bucket overflows. In order to accommodate fluctuations in the main memory size, COMA chooses the smaller of the build or probe buckets. This selection process, in turn, facilitates merging of multiple buckets. A cost model of the algorithm was presented. Implementation aspects of the algorithm were discussed. COMA was implemented, and the performance was compared with other similar join algorithms using the Wisconsin benchmark for varying parameters. Analysis results show that, when skew and kurtosis are high, COMA is the best. Specific proposals were made for extending COMA to applying with configurators, in data warehouses, multi-level join queries, mobile databases, and statistical databases to achieve better performance.

ACKNOWLEDGMENTS

I thank God; my father, the late Sri. G. Subbaiah, Bachelor of Commerce; and my mother, Smt. G. Vimala Devi, Bachelor of Science for helping me reach one of my life's ambitions. I would like to express my sincere love and affection to my wife, Dr. Krishna Jayanthi, and my son, SaiHarsha, who put up with my irregular hours and were instrumental in keeping me motivated. My special thanks to Professor Perrizo, my major adviser and my mentor, for being a constant source of inspiration and for guiding me through my stay at North Dakota State University and through my dissertation. I could not have hoped for a better adviser. My sincere thanks to the other members of my committee: Dr. Vasant Ubhaya, Dr. John Martin, Dr. Victor Shi, and Dr. Dan Ewert for their encouragement. My special thanks to all past and current members of Dr. Perrizo's database research group. I thank the Department of Computer Science as well as The Graduate School for their concerted support in remotely submitting this disquisition. I thank Qualcomm, Inc., and its management and system administration staff for letting me use its resources for performance evaluation. Finally, I would like to thank my alma mater, Sri Venkateswara University, India, where the motto is "Knowledge, the most valuable asset, is to be absorbed from all directions, by all means" (Sanskrit: *Gyanam Samyagavekshanam*.)

TABLE OF CONTENTS

ABSTRACT	iii
ACKNOWLEDGMENTS.....	iv
LIST OF TABLES	viii
LIST OF FIGURES.....	ix
CHAPTER 1. INTRODUCTION	1
1.1. Database Systems.....	1
1.2. Data Warehouse Systems	2
1.3. Configurators.....	3
1.4. Join Operation	4
1.5. Outline.....	9
CHAPTER 2. LITERATURE REVIEW AND FOUNDATION.....	11
2.1. Join Processing Algorithms.....	12
2.1.1. Evolution	12
2.1.2. Join Acceleration Schemes.....	123
2.2. Domain Vector Join Algorithms	14
2.3. Overflow Handling in Hash Join Processing	16
CHAPTER 3. COMPLETE OVERFLOW MANAGEMENT JOIN ALGORITHM.....	18
3.1. Domain Vectors.....	18
3.2. Domain Vectors and Join Query Processing.....	20
3.3. Domain Vector Perfect Hash Join Process.....	22
3.3.1. DVPH Algorithm	26
3.4. Complete Overflow Management Algorithm	29

3.4.1. Orthogonal Hash Functions.....	30
3.4.2. Orthogonal Hash Functions in COMA.....	34
CHAPTER 4. ALGORITHM AND ANALYTICAL COST MODEL.....	36
4.1. Complete Overflow Management Join Algorithm.....	36
4.1.1. Elements Required for Complete Overflow Management Join Algorithm.....	36
4.1.2. Hashing Functions.....	36
4.1.3. Algorithm	37
4.2. Analytical Cost Model of Complete Overflow Management Algorithm.....	41
4.2.1. Notations	42
4.2.2. Cost Functions for Individual Steps in the Algorithm	44
4.2.3. Total Cost Function of COMA.....	53
CHAPTER 5. IMPLEMENTATION AND PERFORMANCE ANALYSIS.....	56
5.1. Hardware and Software Environment.....	57
5.1.1. Hardware	57
5.1.2. Database Initialization.....	58
5.1.3. Query Execution.....	58
5.1.4. Algorithms Compared.....	58
5.1.5. Database Engine	58
5.1.6. Software Implementation	59
5.1.7. Skewed Distribution in Attribute Values	59
5.2. Performance Parameters.....	59
5.3. Key Assumptions in Performance Evaluation	63
5.4. Performance Analysis	63

5.4.1. COMA General Performance	64
5.4.2. Bucket Overflows	74
5.4.3. Collisions	77
CHAPTER 6. FUTURE RESEARCH	80
6.1. Future COMA Theoretical Research	80
6.2. Future COMA Application-Oriented Research	83
CHAPTER 7. CONCLUSION	88
REFERENCES	90

PREVIEW

LIST OF TABLES

<u>Table</u>	<u>Page</u>
4.1. COMA cost function notations used.	42
4.2. Cost-related symbol notations	43
4.3. Function notations used.....	44
5.1. COMA performance test machine specifications.....	57
5.2. Values of fixed parameters.....	60
5.3. Variable parameters that influence query performance	62

PREVIEW

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1.1. The join operation	5
3.1. Join query processing using domain vectors.....	21
3.2. Data structures for join query processing.....	21
3.3. Patients and Tenants table	23
3.4. Build phase of the domain vector perfect hash	24
3.5. Join query data structures	25
5.1. Memory (pages) vs. query execution time	65
5.2. PK table participation vs. query execution time (skew = 0)	66
5.3. PK table participation vs. query execution time (skew = 3)	67
5.4. FK table participation vs. query execution time (skew = 0)	68
5.5. FK table participation vs. query execution time (skew = 3)	68
5.6. Number of records in PK table vs. query execution time (skew = 0)	69
5.7. Number of records in FK table vs. query execution time (skew = 0)	70
5.8. Primary key size vs. query execution time (fixed PK table record size) (skew = 0, main memory = 150 pages, and PK table record size=200 bytes)	71
5.9. Primary key size vs. query execution time (skew = 0, main memory = 150 pages, and variable PK table record size)	72
5.10. PK table record size vs. query execution time (skew = 0 and main memory = 150 pages)	73
5.11. FK table record size vs. query execution time (skew = 0 and main memory = 150 pages).....	74
5.12. Kurtosis vs. bucket overflows (main memory = 150 pages and skew = 0).....	75

5.13. Kurtosis in primary key values vs. query execution time (main memory = 150 pages and skew = 0)	76
5.14. Kurtosis in foreign key values vs. query execution time (main memory = 150 pages and skew = 0)	77
5.15. Skew vs. query execution time (main memory = 150 pages and kurtosis = 0).....	78
5.16. Skew in foreign key values vs. query execution time (main memory = 150 pages and kurtosis = 0).....	79

PREVIEW

CHAPTER 1. INTRODUCTION

In this introductory chapter, databases and their importance in applications are discussed in Section 1.1. Sections 1.2 and 1.3 discuss the latest applications of databases in data warehouses and configurators, respectively. Section 1.4 presents the most critical operation of databases, the Join process, and provides an example. The last section in this chapter, Section 1.5, outlines all chapters in this study.

1.1. Database Systems

The database is the heart of information processing systems. Databases pump data to the information processing systems. Relational databases are the primitive and common database management systems. Current relational databases are in the form of data warehouse systems. Data warehousing is a vital enterprise technology that is revolutionizing the way companies are accessing information and using it to create strategic business solutions. Data warehouse systems are centralized database systems with a collection of decision support technologies aimed at enabling the knowledge worker (executive, manager, or analyst) to make better and faster decisions [CHOU97]. These systems require effective mechanism for storage, retrieval, and processing complex objects.

Database systems provide a mechanism to define data structures that represent objects, and they provide an interface to manipulate (read and write) that data. Standard software systems requests data in the form of queries. Databases respond by processing the query and returning the result data. Relational databases apply the “entity-relationship model”

and store data items representing the “real-world model” in relations (or tables). Relations storing different information are often related to each other. Often, these relations share a common property or attribute(s). The three basic relational operators are project, select, and join. Each relational operator produces a new relation (table) based on one or more input tables. The SQL PROJECT operator discards some columns; SELECT discards some rows based on a predicate; and JOIN combines records from two tables to form a third table. The focus of this research is on optimizing the join operation.

A brief description of two types of information systems, data warehouse systems and configurators, elicits the importance of join processing and thus the importance of improving the performance of join algorithms.

1.2. Data Warehouse Systems

Large relational tables exist in data warehouse systems. Development of data warehouses has brought with it the need for corresponding meta-data to be available to the business community. To assemble valuable data for decision-making, a warehouse system takes snapshots of operational data repeatedly at different times. These snapshots are then used for comparison, evaluation, projection, pattern recognition, and other decision-support system (DSS) related activities. Warehouse databases can easily expand into the terabyte-size systems spoken of and implemented today. Time is the killer dimension that leads to mammoth database sizes. In operational applications, such as order entry or purchasing, on-line querying is usually limited to very few rows at a time. True data warehouse queries are data-intensive.

The nature of a data warehouse includes the following characteristics [INMO96]:

1. Integrated data,
2. Detailed and summarized data,
3. Historical data, and
4. Meta-data

Unlike end users of operational environments, who have minimal need to access meta-data, warehouse knowledge workers use meta-data as the primary mechanism for developing their queries and determining warehouse data quality. During the development of the data-warehouse, existing meta-data about data sources are used to create new meta-data. These meta-data are used to evaluate and format data for the target warehouse environment. For many organizations, developing a data warehouse provides an impetus for integrating data from various sources into a consistent format and structure. Both processing the meta-data as well as integration involve the join operation.

1.3. Configurators

Complexity and the amount of data eligible for query processing are proportional to the diversity of information in the database. Attributes with large cardinalities require efficient data handling mechanisms. Configurators are used to resolve complexity and to explore data. Configurators are programmable expert systems. They are useful for data mining. A configurator is the preferable choice when relational, object, and rule models need to be integrated. Business knowledge is preserved, processed, and retrieved at a later date by modeling the processing lines through objects, rules, and relations. Currently, configurators are being extended to on-line analytical processing (OLAP). They are useful to interpret the query and to identify data structures necessary for query processing.

Configurators contain a rule set and an action set. This information is then read by the configurator to identify appropriate structures (such as domain vectors) to be used in queries.

The success and performance of data warehouses and configurators depend on their performance. During transactions, the performance is tied to how fast configurators or data warehouses can “mix and match” or “join” data in different entities.

1.4. Join Operation

The basis for the success of relational databases and the most sought-after database operation is the join. The join operation, in its simplest form, produces the result of a Cartesian product of records from two related tables. During transaction processing, databases match related rows present in two separate, but related, tables. Related attributes of the two relations play pivotal roles while matching. This operation is the join operation. Within the same database, many such relations representing real-world relationships can exist among tables. In this process, the relational information may be duplicated. *Normalizing* the database guarantees that most of the problems, such as redundancy, potential for inconsistency (update anomalies), insertion anomalies, and deletion anomalies, do not occur.

The success of normalizing a database depends on the performance of a join operation in the database management system. Normalization requires maintenance of referential integrity. Join enables retrieval of information from a normalized database while maintaining referential integrity. The attribute that plays a crucial role in matching the rows is called the joining attribute. The response time to process a join query depends on

the number of values in the joining attribute. In general, an equijoin concatenates together the records (or some subset of attributes from them) that match from two related tables. The example in Figure 1.1 makes this description clearer and illustrates the processing problems involved.

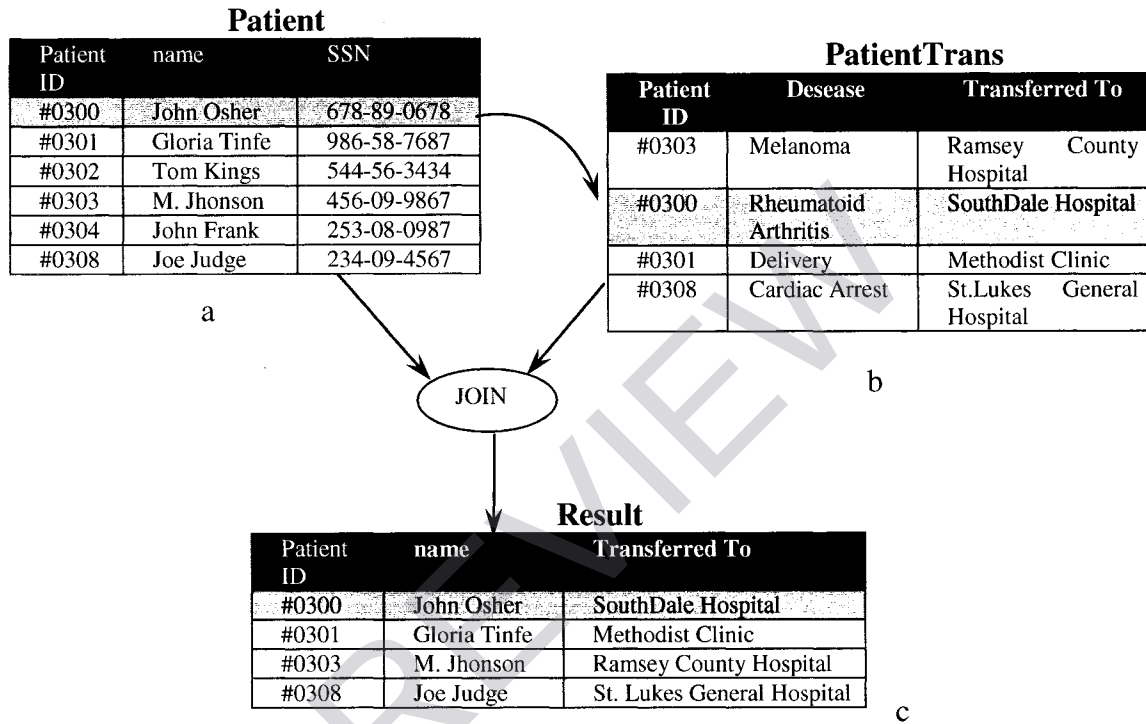


Figure 1.1. The join operation.

Figure 1.1 describes a hospital database containing “Patient” table (Figure 1.1a) and “PatientTrans” table (Figure 1.1b) representing those patients who were transferred to other hospitals. PatientID is the common attribute between these two tables. The first record in the “Patient” table, for example, has PatientID #0300. The related record (with PatientID attribute value #0300) in the “PatientTrans” table reveals that this patient has rheumatoid arthritis. In general, it is desirable to generate a report showing the current location of patients and the reason for their transfer. This report would be the result of

doing an equijoin between Patient and PatientTrans tables on PatientID attribute. Leaving out undesired attributes (i.e., doing a projection), the equijoin gives the Result shown in Figure 1.1c.

The result table facilitates the information that Patient ID #0300 (John Osher) is at SouthDale Hospital, Patient ID #0301 is at Methodist Clinic, etc. Because of the way relational databases are designed, much information inherent in the data can be extracted or organized conveniently by the join operation. Join is a significant aspect of contemporary data processing because hospitals and other service related corporations typically maintain data in relational database format.

During join processing, records of a relation must be identified efficiently using the values in a set of unique attributes. This set is called the *primary key*. This primary key can be achieved by integrating the access path mechanism with the storage allocation at the record level. Key sequenced tables are common in databases. In the above example, attribute “PatientID” in the “Patient” table is a good candidate for the primary key. The same attribute values are represented in a second table (“PatientTrans” table) and are used to match rows from the two tables. The PatientID attribute in the “PatientTrans” table is the *foreign key*. It is common that the join between two tables is performed based on the primary key and foreign key values.

Initially, two fundamental joining algorithms, nested loop and sort merge, were laid down by Blasgen and Eswaran [BLAS77]. In Figure 1.1, “Patient” table is sorted on PatientID while the “PatientTrans” table is not. The nested loop algorithm is the simplest join algorithm to form the complete join result. In the nested loop algorithm for each tuple of the first relation, all tuples of the other relation are searched for matches. If cardinalities

of the two relations are m and n , the order of complexity of the algorithm is $O(m \times n)$. In the above example, joining “Patient” and “PatientTrans” tables require 6×4 , or 24, record accesses, and the cost of join processing depends on the number of record accesses. The sort-merge join method requires that both relations be sorted on the primary key (or foreign key) attributes. If the “PatientTrans” table is also sorted on PatientID, it would be possible to form the join result by sequentially stepping through each of the relations and merging the matching records. Essentially, sorting reduces the number of records accessed during a join.

Due to the critical nature of the equijoin in relational database management systems, much research has been devoted to developing algorithms and techniques to improve performance above that of the fundamental algorithms. Researchers, in their quest to improve the performance of join, suggested surrogate structures to accelerate the join operation. The first such attempt was made through join indices by Valduriez [VALD87]. The idea is to store the information of the matching key values from two tables (required to maintain referential integrity) in a separate index and to use this index efficiently to process the join.

Sorting a large table is difficult and time consuming. With the advent of large main memory computers, the number of records that can be saved into the main memory increased. Large main memory facilitated hashing more records into the main memory. Hashing is done based on a hash value generated uniquely from a key attribute value. When a hash algorithm is applied on join participating key values of records from two different tables, their hash values match, thus sorting is avoided. Hash-based join algorithms are applied when the selectivity ratio is low and the number of records that

need to be matched is many times more than the main memory capacity. Hash join algorithms are attractive when no indices are available on joining column(s) and the join result need not be sorted. Hash-based techniques use a hashing function to hash the records to hash-file-buckets. Each bucket is then examined for records from joining tables that have matching join attribute values to produce the result of the join operation. Hybrid Hash join, a flavor of hash join, is adaptive to work with dynamic changes in the available main memory. More about these techniques is discussed in Chapter 2. When the selectivity ratio is very low, hashing all records in the main memory is a waste of main memory space. Domain vectors built on the primary-key-value domain facilitate hashing of only the qualifying records into main memory, increasing the capacity of records in hash partitions and thus improving the throughput.

Because of the variations in data distributions and selection parameters involved, the data are always skewed, and the selectivity ratio is low. Traditional join techniques assume that key value distribution is normal, and these techniques often collapse when the key value distribution is uncertain (skewed or kurtotic.)

When hash tables are used, overflow of a hash table is imminent. Overflow of a hash table is the result of an incorrect estimate of the number of participating join records and depends on the key value distributions. Hash table overflows makes hash based join algorithms ineligible for multi-user environments where available main memory capacity cannot be constant. Developing an algorithm to avoid hash bucket overflows during hash join is the goal of this study.

1.5. Outline

This research work centers around the well-known data structure, the domain vectors, which accelerates join operations by making use of bit vectors defined on a set of domain values. Join acceleration algorithms are developed for the use of domain vectors. This research work continues that effort, extending it to Complete Overflow Management join Algorithm for join acceleration. An analysis which explains how factors, such as selectivity, main memory size, distribution criteria of primary key values, etc., influence the join query performance is done. The performance of an implementation of Complete Overflow Management join Algorithm is compared with the performance of other related algorithms.

This thesis is divided into seven chapters as follows:

1. **Introduction:** Chapter 1 provides an overview of databases and data warehouse systems, introduces joining and its importance, and outlines the rest of the work.
2. **Literature Review and Foundation:** Chapter 2 summarizes the previous research on join acceleration and explains how Complete Overflow Management Algorithm (COMA) arises out of it.
3. **Complete Overflow Management join Algorithm:** Chapter 3 presents the data structures and an algorithm for domain vector processing of a binary join, and shows how domain vectors and orthogonal hashing are used to process Complete Overflow Management join Algorithm.

4. **Algorithm and Analytical Cost Model:** Chapter 4 gives the general algorithm for Complete Overflow Management join Algorithm using domain vectors and orthogonal hashing in sufficient detail to estimate where the costs come from, and analyzes the effectiveness of the algorithm.
5. **Implementation and Performance Analysis:** Chapter 5 explains different aspects of an implementation of domain vectors and orthogonal hash joins, and gives a comparison of their performance with other related algorithms, presenting numerical results and a statistical analysis of algorithms.
6. **Future Research:** Chapter 6 discusses areas of research suggested by this work, improvements, select-project-join queries, data warehouse environments, configurators, application of join in mobile grid computers and related databases, etc.
7. **Conclusion:** Chapter 7 reviews the content and conclusion of the research.

CHAPTER 2. LITERATURE REVIEW AND FOUNDATION

In this chapter, join processing, the importance of bit vector for join processing, application of such techniques for data warehouse, and the previous research in this direction are discussed.

Database systems provide powerful features for manipulating large quantities of data. Database systems receive user requests in the form of queries. The query is then converted into a query execution plan. A query plan is usually compiled into a tree of operators, called a join sequence tree, where a leaf node represents an input relation. Internal nodes represent the result relation from joining the two relations associated with its two child nodes. Improving the performance of join is the scope of this research.

Domain Vector Hash Join Method (DVH) [PERR91] and the Domain Vector Perfect Hash Join Method [GOLI93] exploit a special bit vector, the Domain Vector (DV), to identify records participating in the join. The domain vector is a bit vector associated with a joining attribute of a relation. Domain vector represents all values present in the database for a particular attribute, plus room for expansion. Presence or absence of a value in the relation's joining attribute is signified by presence or absence of a 1-bit in a related position of the vector. For example, the domain vector of the primary key of a relation may have 1-bits in all positions (so it is not necessary to maintain such domain vector). Domain vectors facilitate semi-join reduction (elimination of non-participants) of both joining relations. They are space efficient, and the update overhead is low [PERR94a]. Domain vectors have been implemented successfully on the IBM AS/400 database machine [IBM88], Intel Pentium based PC machines, as well as SUN Ultra 1 machines. This

technique is applicable to mine data from a data warehouse. Diverse data of data warehouse require domain vectors on a number of domain sets. In such situations, identifying the appropriate domain vector is a major task. For this reason, configurators are used along with Domain vectors. In the following section, the history of various join algorithms is discussed.

2.1. Join Processing Algorithms

2.1.1. Evolution

Join is the most important basic operation of database systems. Many researchers proposed techniques to improve the join query performance. Important join methods are sort merge join, nested loops, materialized views, hybrid hash join, join indices, and hit lists. All these processes suggest ways to join two relations in a database. The sort merge technique involves sorting two relations followed by a merge process. The nested loop process iteratively matches tuples from the two relations. Performance costs for these two methods are high for relations with a very large number of records. Materialized view [ADIB80, SHMU84, BLAK90] and join indices [VALD87] require a pre-computed join result. This information is maintained on the disk and accessed each time a join is required. The join index method requires the projected result of a materialized view in the form of a tree-based data structure called the join index. The join index identifies joining tuples by the record identifier. Join indexing avoids reading non-participant records. However, join indices require maintenance of the completed join in a surrogated form [VALD87] of the join result.

Among various other join methods, hash join has been the focus of much research effort and is reported to have performance superior to that of others. Grace hash join [GOOD81, KITS83] partitions joining tables prior to the start of the join. Hybrid hash join simultaneously joins the first partition records while reading the joining tables. Since the introduction of hybrid hash join [DEWI84, SHAP86, PATE94], many hash join methods have been developed [DEWI84, SHAP86, NAKA88, KITS89, BLAK90, ZELL90, PERR91, PANG93, LO96, GOPA01]. In [GOPA01], a criss-crossed hash join strategy, a blend of both join indices and hash join, was proposed. The algorithm uses page maps for join acceleration. Almost all hash join methods rely on partitioning and creating temporary files for large joins. In [SHAN01], the authors explored the use of new scalar and aggregate functions in SQL for constructing complex XML documents directly in the relational engine. They proposed an "outer union plan" to generate the content of an XML document.

2.1.2. Join Acceleration Schemes

Many researchers suggested different techniques to improve the join performance. Bit vector filtering was suggested to improve efficiency in the hash join [DEWI85, SCHN89, GRAE94]. Mackert and Lohman [MACK86] suggested ways of using Bloom filters [BLOO70, BABB79] to find matching join values. Dewitt *et. al.* suggested a non-persistent bit vector filtering technique in [DEWI85]. Goetz Graefe [GRAE94] suggested issues pertinent to comparing hash-based query evaluation techniques. One of the new issues discussed in [GRAE94] is bit vector filtering. In [CHAN98, CHAN99], bitmap encoding schemes for relational queries are discussed.