Improving Hash Join Performance By Exploiting Intrinsic Data Skew

by

Bryce Cutt

BSc(Hons), University of British Columbia, 2007

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The College of Graduate Studies

(Interdisciplinary Studies)

THE UNIVERSITY OF BRITISH COLUMBIA (Okanagan)

March, 2009

© Bryce Cutt 2009

Abstract

Large relational databases are a part of all of our lives. The government uses them and almost any store you visit uses them to help process your purchases. Real-world data sets are not uniformly distributed and often contain significant skew. Skew is present in commercial databases where, for example, some items are purchased far more often than others. A relational database must be able to efficiently find related information that it stores. In large databases the most common method used to find related information is a hash join algorithm. Although mitigating the negative effects of skew on hash joins has been studied, no prior work has examined how the statistics present in modern database systems can allow skew to be exploited and used as an advantage to improve the performance of hash joins. This thesis presents *Histojoin*: a join algorithm that uses statistics to identify data skew and improve the performance of hash join operations. Experimental results show that for skewed data sets *Histojoin* performs significantly fewer I/O operations and is faster by 10 to 60% than standard hash join algorithms.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables iv

List of Figures v

Acknowledgements vi

Dedication vii

1 Introduction 1

2 Background 4
- 2.1 Relational Databases 4
  - 2.1.1 Joins 6
  - 2.1.2 Keys 6
  - 2.1.3 Cardinality 7
- 2.2 Hash Join 8
  - 2.2.1 In-Memory Hash Join 8
  - 2.2.2 Hash Partitioning 11
  - 2.2.3 Grace Hash Join 13
  - 2.2.4 Hybrid Hash Join 14
  - 2.2.5 Dynamic Hash Join 16
  - 2.2.6 Hash Join Performance Enhancements 17
- 2.3 Skew 17
- 2.4 Statistics and Histograms 18
  - 2.4.1 Histograms and Hash Joins 20
- 2.5 Example Database 20
- 2.6 Relational Algebra Query Plan Diagrams 20

3 Histojoin 24
- 3.1 General Approach 24
  - 3.1.1 Theoretical Performance Analysis 25
- 3.2 Histojoin Algorithm 26

3.2.1 Algorithm Overview . . . 26
3.2.2 Selecting In-Memory Tuples . . . 29
3.2.3 Partitioning . . . 31
3.3 Using Histojoin . . . 32
3.3.1 Join Cardinality . . . 32
3.3.2 Histogram Inaccuracies . . . 33
3.3.3 Query Optimizer Modifications . . . 36

4 Experimental Results . . . 38
4.1 Stand-Alone Evaluation . . . 38
4.1.1 Primary-to-Foreign Key Joins . . . 38
4.1.2 Many-to-Many Joins . . . 40
4.1.3 Histogram Inaccuracies . . . 41
4.1.4 Joins on String Keys . . . 42
4.1.5 Multi-Way Joins . . . 42
4.2 PostgreSQL Implementation . . . 43
4.2.1 Primary-to-Foreign Key Joins . . . 45
4.2.2 Multi-Way Joins . . . 45
4.2.3 Effect of Number of MCVs . . . 45
4.3 Results Summary . . . 46

5 Discussion and Conclusion . . . 49

Bibliography . . . 51

List of Tables

2.1 Part Relation 5
2.2 Purchase Relation 5
2.3 Part-Purchase Join Result 7
2.4 Part-Purchase Hash Join Result 14
2.5 An Accurate Histogram on the partid Attribute of the Purchase Relation . . 18
2.6 An Aggregate Histogram on the partid Attribute of the Purchase Relation . 19

3.1 Absolute Reduction in Total I/Os of Skew-Aware Partitioning versus Random Partitioning for Various Values of f and g and $|R| = |S| = 1000$ 26
3.2 Histogram Partitioning Example 30
3.3 Join Cardinality Cases 33

List of Figures

2.1 The Parts of a Relation 4
2.2 Chained Hash Table Example (mod5 hash function) 10
2.3 Partition 0 of Part and Purchase Relations 12
2.4 Partition 1 of Part and Purchase Relations 12
2.5 Partition 2 of Part and Purchase Relations 12
2.6 Partition 3 of Part and Purchase Relations 13
2.7 Partition 4 of Part and Purchase Relations 13
2.8 DHJ Partitioning Example Part 1 21
2.9 DHJ Partitioning Example Part 2 22
2.10 TPC-H Schema from [1] 23
2.11 Example Relational Algebra Diagram 23

3.1 Two Level Partitioning 24
3.2 Total I/Os Percent Difference 27
3.3 Partkey Histogram for Lineitem Relation TPC-H 1 GB Zipf Distribution (z=1) 27
3.4 Histojoin Flowchart 28
3.5 Example Multiple Join Plans 36

4.1 Lineitem-Part Join (1GB, z=1) 39
4.2 Lineitem-Part Join (1GB, z=2) 40
4.3 Percentage Improvement in Total I/Os of Histojoin vs. Hash Join (1GB) . . 40
4.4 Total I/Os for Wisconsin Many-to-Many Join (1GB, z=1) 41
4.5 Total I/Os for Lineitem-Part Join with Histogram Inaccuracies (1GB) 42
4.6 Total I/Os for Lineitem-Supplier Join on String key (1GB, z=1) 43
4.7 Total I/Os for Lineitem-Supplier-Part Join (1GB) 43
4.8 PostgreSQL Lineitem-Part Join (10GB, z=1) 45
4.9 PostgreSQL Lineitem-Part Join (10GB, z=2) 46
4.10 PostgreSQL Percentage Improvement in Total I/Os of Histojoin vs. Hash Join (10GB) 46
4.11 Total I/Os for PostgreSQL Lineitem-Supplier-Part Join (10GB) 47
4.12 PostgreSQL Lineitem-Part Join With Various Amounts of MCVs (10GB, z=1) 48

Acknowledgements

I would like to thank my supervisor Dr. Ramon Lawrence for providing me with the opportunity to do a Master's degree under his supervision. Dr. Lawrence has been a valuable instructor, mentor, and most of all friend during this journey. His feedback, insight, and passion for the subject matter has been invaluable in re-igniting my own obsession with data management and in bringing my thesis to its current form.

I am forever greatful for the patience my wife and family have had for me when my mind is engrossed in a subject. Angela has always provided love and support no matter what mental state I was in. My parents Ken and Sue have always provided a safe harbour when nothing seems to be going right.

I would also like to thank Dr. Patricia Lasserre and Dr. Yves Lucet for mentoring me throughout my previous degree and providing many opportunities for me to build the confidence and skills needed to pursue a Master's degree.

Many people have provided feedback and support regarding my thesis and I would like to express my appreciation to them and acknowledge their contribution to my success.

Dedication

To my family

1. Introduction

The world contains enormous stores of information for various uses, be they academic, government, financial, commercial, etc. For this information to be useful it must be stored and retrieved efficiently. For many years the method of choice for storing large amounts of information has been database systems. Most modern commercial database systems are relational database systems. A relational database consists of tables of information (called relations) that are related to each other according to various rules (Section 2.1).

A relational database system provides methods for storing, retrieving, sorting, searching, and comparing information. It does so while limiting a user's need to understand the underlying system. The operations performed by a database are largely automated and the user interacts with the database primarily through queries in a very natural English-like language called SQL (Structured Query Language) [6].

When a user queries a database the actual operations performed and the algorithms used are hidden from the user. The database must carefully manage its memory usage as most large database systems contain far more information than will fit in the memory of a computer at one time. Databases choose how to search relations for information and how to compare information in multiple relations so that related information can be returned to the user. Returning related information from multiple relations requires that the database compare parts of the different relations and join the matching parts together. This is done by specialized algorithms called join algorithms (Section 2.1.1).

Hash join is the standard join algorithm used in database systems to process large join queries (Section 2.2). Any performance improvement for hash joins is significant due to the cost and prevalence of hash-based joins, especially in the large queries present in data warehouses and decision-support systems. We are increasingly dependent on large governmental, educational, and commercial database systems that are queried regarding our tax information, our student records, and whenever we purchase an item.

With a centralized database system the primary concern of a join algorithm (other than producing an accurate join) is to produce a result quickly by efficiently using the limited memory available and only using disk resources when absolutely necessary as the speed of a disk is an order of magnitude slower than the speed of memory. Parallel and distributed database systems attempt to load balance the work of a join across many nodes. Although parallel databases attempt to avoid partition skew for load balancing, no parallel join algorithm has examined maximizing the in-memory results produced by keeping frequent data memory-resident.

Real data sets often contain skew. *Skew* occurs in data when certain values occur more frequently than others (Section 2.3). Partition skew is when a data set is split into multiple

partitions and some partitions contain more information than the others. Many data sets follow the "80/20 rule" where a small subset of the data items occur much more frequently. For example, consider a company that sells many products and has a database that stores information on customer purchases. This information is valuable for determining which products to re-stock, which sales will be most beneficial, and which products need to be featured in advertising. If a few products are sold far more often than other products then the database will contain far more entries related to those products. This is a very common example of skew.

Traditionally skew has been seen as a negative for join algorithms. Prior work has focused on how to maximize the performance of join algorithms by avoiding the negative effects of skew and using memory more efficiently. Avoiding and mitigating partition skew has been considered in centralized and distributed databases for hash joins [10]. More recently in [19] various approaches were used to lower the negative effect of skew on the performance of sort-merge join algorithms. In each of these cases skew was seen as a problem to be avoided. It is a major issue for the largest databases used by corporations and government where data sizes are in terabytes and queries may take hours.

Modern relational databases contain statistics on the underlying data set that can be used to detect skew before a join algorithm is used. This research examines how skew can be detected and exploited by a modified hash join algorithm to improve performance. There has been no prior work that detects data skew in the relations and uses that skew as an advantage to maximize the number of in-memory results produced by a hash join. By using the features of a modern database system skew can finally be seen as an advantage and used to greatly increase database performance when properly exploited.

By detecting intrinsic data skew in the data set and preferentially buffering the most useful data, memory is used more efficiently, I/O operations are decreased, and join and database performance is improved. This thesis is confirmed by research that produced the *Histojoin* algorithm. *Histojoin* is a modification to hash join that exploits data skew to improve hash join performance. The *Histojoin* algorithm implementation uses statistics to detect skew in the input relations. Statistics such as histograms [13] are commonly produced by a database system for query optimization and can be exploited at no cost by the join algorithm. The algorithm has better performance than standard hash joins for skewed data. The improvements made to hash join allow it to finally take advantage of statistics that have been available in commercial database systems for years.

The contributions are as follows.

- An analysis of the advantage of exploiting data skew to improve hash join performance.
- A modification of hash join called *Histojoin* that uses statistics to detect data skew and adapt its memory allocation to maximize its performance.
- An implementation of *Histojoin* in a stand-alone Java database system and an implementation of *Histojoin* in the popular PostgreSQL open source database system.

- An experimental evaluation that demonstrates the benefits of *Histojoin* for large data warehouse queries using the TPC-H data set.

This thesis expands on the presentation in [4, 5]. The PostgreSQL implementation is currently being evaluated for inclusion in the main production branch of PostgreSQL where it will have an impact on many real world databases and millions of users.

The organization of this thesis is as follows. In Chapter 2 the database and many of its operations are described in enough detail to provide a base of understanding necessary to appreciate the purpose and benefits of *Histojoin*. In Chapter 3 the *Histojoin* algorithm's operation and functionality are explained in detail. In Chapter 4 *Histojoin* is compared to a standard hash join in many experiments. The results demonstrate significant performance improvements with *Histojoin* vs. hash join that increase as the data skew increases. In Chapter 5 the content of this thesis is summarized and conclusions are drawn from the experimental results.

2. Background

2.1 Relational Databases

A modern relational database consists of tables of information that are related to each other according to various rules. These tables are referred to as *relations*.

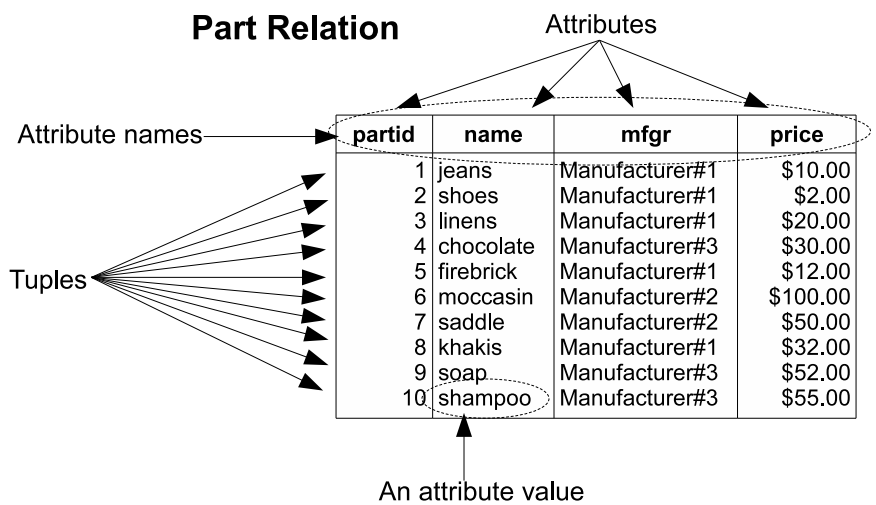


Figure 2.1: The Parts of a Relation

A relation consists of columns and rows where each *row* is an entry in the relation and each *column* specifies a piece of information that each row contains. A row is referred to as a *tuple* and a column is referred to as an *attribute*. In the example *Part* relation given in Figure 2.1 and Table 2.1, the attributes are (*partid, name, mfgr, and price*) and one of the tuples is (1, jeans, Manufacturer#1, $10.00). As can be seen the tuple contains a piece of information (a value) for each attribute in the relation and if multiple tuples are examined it is apparent that values in the same attribute of different tuples are of a similar type. The *Purchase* relation given in Table 2.2 follows a similar format. These relations will be used as examples throughout the thesis.

partid	name	mfgr	price
1	jeans	Manufacturer#1	$10.00
2	shoes	Manufacturer#1	$2.00
3	linens	Manufacturer#1	$20.00
4	chocolate	Manufacturer#3	$30.00
5	firebrick	Manufacturer#1	$12.00
6	moccasin	Manufacturer#2	$100.00
7	saddle	Manufacturer#2	$50.00
8	khakis	Manufacturer#1	$32.00
9	soap	Manufacturer#3	$52.00
10	shampoo	Manufacturer#3	$55.00

Table 2.1: Part Relation

purchaseid	partid	quantity	tax	shipdate	shipmode
1	1	10000	$0.02	1993-10-10	MAIL
2	2	50000	$0.08	1995-10-28	RAIL
3	2	5000	$0.03	2001-04-19	TRUCK
4	2	3300	$0.02	1998-07-24	AIR
5	2	8300	$2.00	2004-01-14	MAIL
6	2	1000	$0.08	1993-10-11	RAIL
7	2	2000	$0.02	1995-10-29	TRUCK
8	2	3100	$0.03	2001-04-20	AIR
9	2	1900	$0.03	1998-07-25	MAIL
10	3	1800	$0.02	2004-01-15	RAIL
11	3	1500	$0.08	1993-10-12	TRUCK
12	3	1100	$0.08	1995-10-30	AIR
13	4	500	$0.02	2001-04-21	MAIL
14	4	1500	$0.03	1998-07-26	RAIL
15	5	100000	$0.02	2004-01-16	TRUCK
16	6	200000	$0.05	1993-10-13	AIR
17	7	1300	$0.08	1995-10-31	MAIL
18	8	10000	$0.02	2001-04-22	RAIL
19	9	5000	$0.02	1998-07-27	TRUCK
20	10	100000	$0.02	2004-01-17	AIR

Table 2.2: Purchase Relation

A relational database also models the relationships between the data the relations represent. The example database containing the *Part* and *Purchase* relations models types of parts that are for sale and individual purchases made of those parts. Each type of part can be purchased one or more times and each purchase is a purchase of one and only one type of part. This relationship can be seen by examining the *partid* attribute in the *Part* and *Purchase* relations. Each value in the *partid* attribute of the *Purchase* relation exists in one and only one tuple of the *partid* attribute in the *Part* relation. This is commonly referred to as a one-to-many relationship (see Section 2.1.3). The *partid* attribute contains unique values in the *Part* relation in that every tuple has its own value for this attribute and none are the same.

2.1.1 Joins

When a user retrieves information about a part they may want to also know all the individual purchases that have been made for that part. Also when they retrieve information about an individual purchase they may want to know about the part that was purchased. Using the relationship between *Part* and *Purchase* we can find the part that was sold by taking the *partid* of the purchase and looking it up in the *Part* relation. We can also find all the purchases made of a part by taking the part's *partid* and looking for all matching tuples in the *Purchase* relation.

If the database completed this process as described above the user would do the first query, make a note of the *partid*, and then do another query. As the data in a real database is usually far more complex and abundant than this example (perhaps millions of tuples) it is more efficient to allow the database to do this second lookup using specifically designed algorithms called join algorithms wherein the database joins the tuples of one relation with the tuples of another relation according to their relationship and some join condition.

The result of joining two relations is a collection of tuples where for each tuple in the first relation and each matching tuple in the second relation we have a result tuple whose values are a concatenation of the values from the first relation tuple and the second relation tuple. If the first relation has the attributes (*partid, name, mfgr, and price*) and the second relation has the attributes (*purchaseid, partid, quantity, tax, shipdate, and shipmode*) then each result tuple has the attributes (*partid, name, mfgr, price, purchaseid, partid, quantity, tax, shipdate, and shipmode*). In this simple database example it contains a duplicate of the join attribute (*partid*) because it is a simple concatenation.

The SQL statement for this join would be "*SELECT * FROM Part, Purchase WHERE Part.partid = Purchase.partid*" and the result returned by the query is shown in Table 2.3.

2.1.2 Keys

In a database system it is important to be able to find an individual tuple in a relation. Keys are a means for a database to find and compare individual tuples in its relations. Usually when joins are performed on relations those joins are performed using key attributes

partid	name	mfgr	price	purchaseid	partid	quantity	tax	shipdate	shipmode
1	jeans	Manufacturer#1	$10.00	1	1	10000	$0.02	1993-10-10	MAIL
2	shoes	Manufacturer#1	$2.00	2	2	50000	$0.08	1995-10-28	RAIL
2	shoes	Manufacturer#1	$2.00	3	2	5000	$0.03	2001-04-19	TRUCK
2	shoes	Manufacturer#1	$2.00	4	2	3300	$0.02	1998-07-24	AIR
2	shoes	Manufacturer#1	$2.00	5	2	8300	$2.00	2004-01-14	MAIL
2	shoes	Manufacturer#1	$2.00	6	2	1000	$0.08	1993-10-11	RAIL
2	shoes	Manufacturer#1	$2.00	7	2	2000	$0.02	1995-10-29	TRUCK
2	shoes	Manufacturer#1	$2.00	8	2	3100	$0.03	2001-04-20	AIR
2	shoes	Manufacturer#1	$2.00	9	2	1900	$0.03	1998-07-25	MAIL
3	linens	Manufacturer#1	$20.00	10	3	1800	$0.02	2004-01-15	RAIL
3	linens	Manufacturer#1	$20.00	11	3	1500	$0.08	1993-10-12	TRUCK
3	linens	Manufacturer#1	$20.00	12	3	1100	$0.08	1995-10-30	AIR
4	chocolate	Manufacturer#3	$30.00	13	4	500	$0.02	2001-04-21	MAIL
4	chocolate	Manufacturer#3	$30.00	14	4	1500	$0.03	1998-07-26	RAIL
5	firebrick	Manufacturer#1	$12.00	15	5	100000	$0.02	2004-01-16	TRUCK
6	moccasin	Manufacturer#2	$100.00	16	6	200000	$0.05	1993-10-13	AIR
7	saddle	Manufacturer#2	$50.00	17	7	1300	$0.08	1995-10-31	MAIL
8	khakis	Manufacturer#1	$32.00	18	8	10000	$0.02	2001-04-22	RAIL
9	soap	Manufacturer#3	$52.00	19	9	5000	$0.02	1998-07-27	TRUCK
10	shampoo	Manufacturer#3	$55.00	20	10	100000	$0.02	2004-01-17	AIR

Table 2.3: Part-Purchase Join Result

as the join attributes. Many database systems also automatically generate statistics for key attributes which will be important in later sections of this thesis.

Primary Keys

A tuple contains values for each of its attributes. A tuple can be uniquely identified by finding the tuple that has exactly these values. A *primary key* (PK) is a minimal set of attributes that uniquely identifies a tuple in a relation. For example, in the *Part* relation the PK is *partid*. Inspection of Table 2.1 shows that every tuple in the *Part* relation has a different value in this attribute. In the *Purchase* relation the PK is *purchaseid*.

Foreign Keys

When two relations are related the database must store some information in these relations so that for each tuple in one relation all of the related tuples in the other relation can be found. A *foreign key* (FK) is a set of attributes in a relation whose values can be used to find related tuples in another relation. For example, in the case of the *Purchase* relation the *partid* attribute for each *Purchase* tuple contains a value that is equal to the *partid* value for the related *Part* tuple. In the *Part* relation (Table 2.1) there is a tuple with the value 5 in its *partid* attribute. In the *Purchase* relation (Table 2.2) all tuples with the value 5 in their *partid* attribute are related to that single tuple from the *Part* relation.

Often a database system will enforce that a foreign key value is not valid unless that value exists in the primary key attribute for at least one tuple in the relation that the foreign key references.

2.1.3 Cardinality

If there is a relationship between two relations in a database then that relationship has a cardinality. The possible cardinalities are *one-to-one* (1:1), *one-to-many* (1:M), and *many-*

to-many (M:N).

In a one-to-one relationship each tuple in one relation is related to (shares attribute values with) one and only one tuple in the related relation. For this relationship to exist the values in the key attributes must be unique and therefore those attributes can be a primary key of the relation.

In a one-to-many relationship each tuple in the "one" side relation can share the same key values as many tuples in the "many" side relation while each tuple in the "many" side contains the same value as one and only one tuple in the "one" side relation. Usually this type of relationship is implemented as a primary-to-foreign key relationship where the joins are perfomed using the values in the primary key of the "one" relation and a foreign key of the "many" relation. The key attributes of the "one" side relation must contain unique values and can therefore be a primary key of the relation. The key attributes of the "many" side relation are not unique and are foreign key attributes of the relation. The *Part* and *Purchase* relations follow this type of relationship. It is possible that a tuple in the "one" side relation may not be related to any tuples in the "many" relation.

A many-to-many relationship exists when each tuple in one of the relations can share the same attribute values as many tuples in the other relation and vice versa. The set of join attributes in each relation cannot be a primary key simply because it cannot be required to be unique.

2.2 Hash Join

There are numerous join algorithms [11]. The three general types are nested loop join, sort-based join, and hash-based join. In a *nested loop join* each tuple in the first relation is compared linearly to each tuple in the second relation and any matching tuples generate a result tuple. A *sort-based join* first sorts each of the input relations on the join attributes and then linearly scans through both relations simultaneously, generating a result tuple each time the same join attribute values are encountered in both underlying relations. A *hash-based join* is one that uses a hash function (Section 2.2.1) on the join attributes of the input relations and only compares the tuples of the first relation with the tuples of the second relation that hash to the same value as only those tuples have a chance of matching and generating result tuples.

In a hash join the smaller relation of the two (in a one-to-many join it is usually the "one" side of the join) is called the *build* relation and the other relation is called the *probe* relation. The build relation is the relation whose tuples are used to build and fill the in-memory data structures while the probe relation is the relation whose tuples are used to probe and search the in-memory data structures.

2.2.1 In-Memory Hash Join

An in-memory hash join is a join algorithm that uses a hash function and in-memory hash table to simplify the comparison of join attribute values and limit the number of tuple

comparisons necessary to find all the result tuples of a join.

Hash Function

A *hash function* is a function ($y = f(x)$) that takes an input value and returns an output value that falls within an acceptable range of possible values. The input values could be lines of text and the output values could be all valid 8 digit hexadecimal numbers. The input values could be all positive integer numbers and the output values could be all integers between (and including) 0 and 4.

The process of calling a hash function on a value is often called hashing, and the output value is often referred to as the *hash* of the input value. Often a hash function is used to take a set of input values and map those into locations where they are to be stored.

As a function, if two input values are the same, then the output values generated must be the same. For instance if the hash of 5 is 0 the first time the hash function is called then the hash of 5 must be 0 the second time. It is not required that different input values produce different output values.

One use of a hash function would be to separate a large unordered set of unsigned integers (positive whole numbers) into five groups of roughly equal size making sure that all numbers that are equal fall in the same group without first sorting those numbers. If we use a hash function that outputs five possible values we can have a group for each value the hash function produces.

A very simple hash function that can be performed on unsigned integers is an arithmetic modulo operation. If there are five groups then performing a modulo 5 (mod5) on the input values would produce output values in the range 0 to 4 which is exactly 5 possible values. Assign these possible values to the groups much like the indexes of an array. Starting at 0 we have group 0, group 1, group 2, group 3, and group 4. The hash of the number is the group to place the number in.

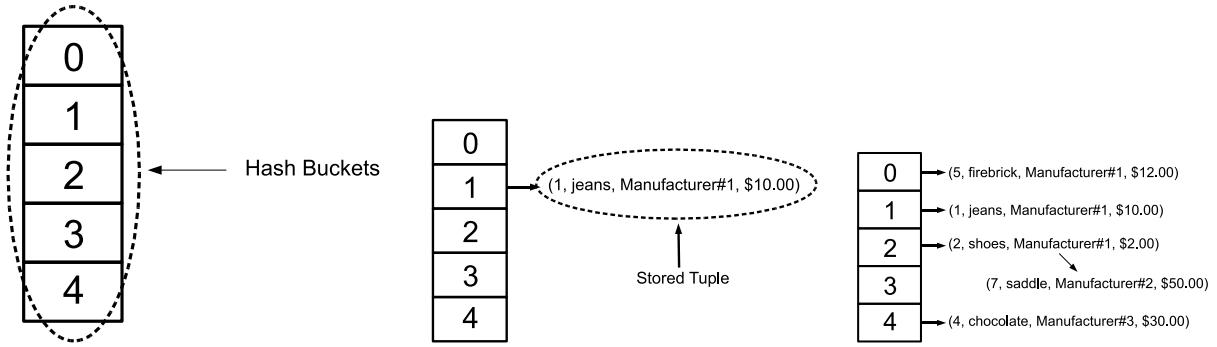
A data structure that uses a hash function to determine the location that each value is to be stored and uses that same hash function to find values that are already stored in the data structure is called a *hash table*. In the example above each location in the hash table stores multiple values. To store a new value is an O(1) operation as it requires only the hash function to be computed on the value and then the value can be stored directly in the appropriate location. To find and retrieve a value in this hash table is not O(1) however as once the correct location is found from the hash value all values in that location must be searched to find the correct one. A perfect hash table is one that maps each input value to a unique location which would make retrieval an O(1) operation. Finding the perfect hash function that creates a perfect hash table is often impossible. Database hash functions must be fast but not perfect. Further discussion of hashing as well as how to find a good hash function is discussed in Section 6.4 of [17].

Separate Chaining Hash Table

One common form of hash table is a separate chaining hash table (or chained hash table). In a chained hash table you start by determining a rough estimate of the number of entries the table needs to store. Then a hash bucket is created for each value the hash table is to store and an array of pointers to those hash buckets is also created. The index of the array is the hash value that will be placed in that bucket so any input value that hashes to 3 will be placed in bucket 3. For example, if the hash table must store 5 values then 5 hash buckets are necessary and indexed from 0 to 4 as shown in Figure 2.2a.

As we assume our hash function is not perfect, a hash bucket must be able to accomodate multiple input values that hash to the same hash value. It is also possible that an input value could occur multiple times. A hash bucket can be as simple as a linked list. Each bucket in Figure 2.2a initially contains an empty linked list.

To insert an input tuple into the chained hash table the hash value of the input tuple is calculated and then the input tuple is appended to the corresponding hash bucket. This is an O(1) operation as long as appending to the hash bucket is O(1), and it is for any well built linked list. For example, the *Part* tuple with *partid* 1 hashes to bucket 1 (with a mod5 hash function) and is appended to the linked list for bucket 1 as shown in Figure 2.2b. If the tuples with *partid* 2, 4, 5, and 7 are also inserted, the hash table would look like Figure 2.2c. Both 2 and 7 hash to the same bucket but because the bucket can accomodate multiple tuples this situation is handled.



(a) Initial Table (b) After Inserting Tuple With partid 1 (c) After Inserting All Tuples

Figure 2.2: Chained Hash Table Example (mod5 hash function)

To retrieve or find an entry in the chained hash table, the hash value of the input value is calculated and then the corresponding hash bucket is linearly searched for any tuples that match the input value. For example, to find a tuple with *partid* 7 in the hash table, 7 is hashed and found to match with bucket 2. The tuples in that bucket are then linearly scanned and the tuple that contains *partid* 7 is returned. If the hash function generates relatively unique hash values for input tuples and there are no duplicate *partid* values then most hash buckets will contain only one tuple and this operation is in practice close to O(1). However this operation could be O(N) in the worst case where all input tuples happen to

hash to the same bucket. Hash tables usually implement a feature that detects and corrects this problem by adapting the hash function and rehashing the input tuples. For a chained hash table this could be as simple as creating a new hash table that is multiple times larger than the current hash table. Rehashing is an expensive operation. Additional information on hashing and hash tables is in [11].

Probing A Hash Table

Probing a hash table of tuples is the process of searching the hash table for any tuples that will join with a given tuple and producing result tuples from any matches. It is possible for one probe to return multiple result tuples if multiple tuples in the hash table match with the given tuple. Finding matching tuples in a hash table can be done while comparing very few tuples which is far more efficient than comparing each probe tuple to every build tuple as in a nested loop join.

If the hash table in Figure 2.2c is probed with the *Purchase* tuple (17, 7, 1300, $0.08, 1995-10-31, MAIL) the *partid* value 7 is hashed and found to match to bucket 2. Bucket 2 is then linearly scanned and any tuples that have the *partid* 7 are joined with the *Purchase* tuple to produce a result tuple. The first tuple in bucket 2 does not match but the second one does which produces the result tuple (7, saddle, Manufacturer#2, $50.00, 17, 7, 1300, $0.08, 1995-10-31, MAIL). If multiple tuples in bucket 2 had matched with the probe tuple then multiple result tuples would have been generated.

Hash Join Algoritm

An in-memory hash join is performed by storing the build relation tuples in an in-memory hash table and then probing those tuples with all probe relation tuples. A hash table is created in memory and then filled with build relation tuples. For each probe relation tuple the hash table is probed and any matches are returned as result tuples. If there are more build tuples than will fit in memory at one time then an in-memory hash join can not be used. To solve this problem hash join algorithms that partition the build relation into smaller chunks have been developed.

2.2.2 Hash Partitioning

If there are more build relation tuples than can fit in available memory, one approach is to partition the relations into smaller partitions of tuples so that all of the build relation tuples in a partition can fit in memory. The number of partitions necessary is determined by calculating what percentage of the build relation will fit in available memory. For example, if 20% of the build relation will fit, then 5 partitions are necessary.

When splitting the input relations into partitions the build relation is partitioned first. The hash value of the join attribute value of each tuple is calculated and used to determine which partition the tuple will be placed in. If a join of the *Part* and *Purchase* relations is being performed, the join attribute in each relation is the *partid* attribute. Note this

attribute is an unsigned integer (always positive whole number) so a hash function could be an arithmetic modulo operation as described in Section 2.2.1. If the database chooses to partition the join into five partitions then the hash function could be modulo five (mod5).

The build and probe relations have their own partitions, but these partitions are related. To see this take a look at the hash function and the attribute values being hashed on. If the hash function is mod5 then an attribute value of 5 would hash to 0 for both relations. Therefore any tuple in the build relation with a *partid* attribute value of 5 would end up in partition 0 for the build relation and any tuple in the probe relation with a *partid* attribute value of 5 would end up in partition 0 for the probe relation. Other tuple values (such as 10) will hash to this same partiton, but we can at least guarantee that all tuples with the same value will be in corresponding partitions.

Figures 2.3a, 2.4a, 2.5a, 2.6a, and 2.7a represent the contents of the five partitions of the *Part* relation when partitioned using mod5 as the hash function. Notice how all tuples in partition 0 (Figure 2.3a) have values in the *partid* attribute that hash to 0 with the given hash function.

Similarly Figures 2.3b, 2.4b, 2.5b, 2.6b, and 2.7b represent the contents of the five partitions of the *Purchase* relation using the given hash function. All values of the *partid* attribute in partition 0 (Figure 2.3b) hash to 0.

partid	name	mfgr	price
5	firebrick	Manufacturer#1	$12.00
10	shampoo	Manufacturer#3	$55.00

(a) Part Partition 0

purchaseid	partid	quantity	tax	shipdate	shipmode
15	5	100000	$0.02	2004-01-16	TRUCK
20	10	100000	$0.02	2004-01-17	AIR

(b) Purchase Partition 0

Figure 2.3: Partition 0 of Part and Purchase Relations

partid	name	mfgr	price
1	jeans	Manufacturer#1	$10.00
6	moccasin	Manufacturer#2	$100.00

(a) Part Partition 1

purchaseid	partid	quantity	tax	shipdate	shipmode
1	1	10000	$0.02	1993-10-10	MAIL
16	6	200000	$0.05	1993-10-13	AIR

(b) Purchase Partition 1

Figure 2.4: Partition 1 of Part and Purchase Relations

partid	name	mfgr	price
2	shoes	Manufacturer#1	$2.00
7	saddle	Manufacturer#2	$50.00

(a) Part Partition 2

purchaseid	partid	quantity	tax	shipdate	shipmode
2	2	50000	$0.08	1995-10-28	RAIL
3	2	5000	$0.03	2001-04-19	TRUCK
4	2	3300	$0.02	1998-07-24	AIR
5	2	8300	$2.00	2004-01-14	MAIL
6	2	1000	$0.08	1993-10-11	RAIL
7	2	2000	$0.02	1995-10-29	TRUCK
8	2	3100	$0.03	2001-04-20	AIR
9	2	1900	$0.03	1998-07-25	MAIL
17	7	1300	$0.08	1995-10-31	MAIL

(b) Purchase Partition 2

Figure 2.5: Partition 2 of Part and Purchase Relations

partid	name	mfgr	price
3	linens	Manufacturer#1	$20.00
8	khakis	Manufacturer#1	$32.00

(a) Part Partition 3

purchaseid	partid	quantity	tax	shipdate	shipmode
10	3	1800	$0.02	2004-01-15	RAIL
11	3	1500	$0.08	1993-10-12	TRUCK
12	3	1100	$0.08	1995-10-30	AIR
18	8	10000	$0.02	2001-04-22	RAIL

(b) Purchase Partition 3

Figure 2.6: Partition 3 of Part and Purchase Relations

partid	name	mfgr	price
4	chocolate	Manufacturer#3	$30.00
9	soap	Manufacturer#3	$52.00

(a) Part Partition 4

purchaseid	partid	quantity	tax	shipdate	shipmode
13	4	500	$0.02	2001-04-21	MAIL
14	4	1500	$0.03	1998-07-26	RAIL
19	9	5000	$0.02	1998-07-27	TRUCK

(b) Purchase Partition 4

Figure 2.7: Partition 4 of Part and Purchase Relations

2.2.3 Grace Hash Join

The Grace Hash Join (GHJ) [16] is a hash join algorithm that partitions the input relations into multiple temporary disk files using hash partitioning as described in Section 2.2.2 and then performs an in-memory hash join (Section 2.2.1) on each build and probe pair of disk files.

The first step of a GHJ is to partition the build and probe relations as described in Section 2.2.2. A file is created on disk for each build relation partition, and another file is created on disk for each probe relation partition.

The tuples of the build relation (given in Table 2.1) are read one at a time, their hash value is calculated from the join attribute value (*partid*), and they are appended to the corresponding build relation file on disk. The order that tuples are read from the underlying relations cannot be guaranteed but for convenience we will assume they are read in the order that they appear in Tables 2.1 and 2.2 for this example. Looking at Table 2.1 the first build tuple read is (1, jeans, Manufacturer#1, $10.00) and it is placed in partition 1 (Table 2.4a). This process is continued for all other build relation tuples.

Once the build relation is partitioned a tuple is read from the probe partition and the hash value of its join attribute value is calculated. The tuple is then appended to the corresponding probe relation file on disk. Looking at Table 2.2 the first probe tuple that would be retrieved from the *Purchase* relation is (1, 1, 10000, $0.02, 1993-10-10, MAIL) and its hash value is 1 so it would be placed in partition 1 (Table 2.4b).

After partitioning, both relations there are pairs of related partitions on disk. The build relation partition 0 is related to probe relation partition 0 because they were partitioned using the same hash function on the common join attribute. Any tuples in probe partition 0 will join with only tuples from build partition 0.

The next step is to load tuples from the first build relation partition (partition 0) into memory and then probe them with tuples from the first probe partition. An efficient data structure must be used for it to be efficient to store the build tuples in memory and search them for tuples that match a given probe tuple. A hash table such as a separate chaining

hash table (as discussed in Section 2.2.1) is created in memory and then filled with tuples from the build relation partition. Once this is complete a tuple is read from the corresponding probe relation partition. The hash value of the tuples join attributes is calculated, and then the probe tuple is compared to all build tuples in the corresponding hash table bucket. If any build tuples are found in the hash bucket that share the same join attribute value (not hash value) as the probe tuple then a result tuple is generated for each match. The first tuple in probe partition 0 is (15, 5, 100000, $0.02, 2004-01-16, TRUCK). It matches with the build relation tuple (5, firebrick, Manufacturer#1, $12.00) because they both have *partid* equal to 5. The result tuple is a concatenation of the values of both tuples and is exactly (5, firebrick, Manufacturer#1, $12.00, 15, 5, 100000, $0.02, 2004-01-16, TRUCK). This process continues for each tuple in probe relation partition 0. The disk files for build partition 0 and probe partition 0 are then deleted, and the in-memory hash table is removed from memory. The above process is continued for each build/probe partition pair on disk until all files have been deleted.

The result of joining these two relations can be seen in Table 2.4.

partid	name	mfgr	price	purchaseid	partid	quantity	tax	shipdate	shipmode
5	firebrick	Manufacturer#1	$12.00	15	5	100000	$0.02	2004-01-16	TRUCK
10	shampoo	Manufacturer#3	$55.00	20	10	100000	$0.02	2004-01-17	AIR
1	jeans	Manufacturer#1	$10.00	1	1	10000	$0.02	1993-10-10	MAIL
6	moccasin	Manufacturer#2	$100.00	16	6	200000	$0.05	1993-10-13	AIR
2	shoes	Manufacturer#1	$2.00	2	2	50000	$0.08	1995-10-28	RAIL
2	shoes	Manufacturer#1	$2.00	3	2	5000	$0.03	2001-04-19	TRUCK
2	shoes	Manufacturer#1	$2.00	4	2	3300	$0.02	1998-07-24	AIR
2	shoes	Manufacturer#1	$2.00	5	2	8300	$2.00	2004-01-14	MAIL
2	shoes	Manufacturer#1	$2.00	6	2	1000	$0.08	1993-10-11	RAIL
2	shoes	Manufacturer#1	$2.00	7	2	2000	$0.02	1995-10-29	TRUCK
2	shoes	Manufacturer#1	$2.00	8	2	3100	$0.03	2001-04-20	AIR
2	shoes	Manufacturer#1	$2.00	9	2	1900	$0.03	1998-07-25	MAIL
7	saddle	Manufacturer#2	$50.00	17	7	1300	$0.08	1995-10-31	MAIL
3	linens	Manufacturer#1	$20.00	10	3	1800	$0.02	2004-01-15	RAIL
3	linens	Manufacturer#1	$20.00	11	3	1500	$0.08	1993-10-12	TRUCK
3	linens	Manufacturer#1	$20.00	12	3	1100	$0.08	1995-10-30	AIR
8	khakis	Manufacturer#1	$32.00	18	8	10000	$0.02	2001-04-22	RAIL
4	chocolate	Manufacturer#3	$30.00	13	4	500	$0.02	2001-04-21	MAIL
4	chocolate	Manufacturer#3	$30.00	14	4	1500	$0.03	1998-07-26	RAIL
9	soap	Manufacturer#3	$52.00	19	9	5000	$0.02	1998-07-27	TRUCK

Table 2.4: Part-Purchase Hash Join Result

2.2.4 Hybrid Hash Join

In the previous hash join example, none of the build partitions were more important than the others. All partitions were initially written to disk and then loaded back into memory to be joined a partition at a time. For any significantly large join in a real database it is likely that there is not enough memory to store all of the build relation tuples in memory at one time, but there is often enough memory to store some of the build tuples in memory while partitioning the build relation. Thus, less tuples must be written to disk while partitioning and then read back while probing. The speed of storing data on and retrieving data from disk is on the order of a million times slower than storing and retrieving data from a computer's main memory. It is immediately apparent that limiting how much data must be stored on disk during the join is vitally important to the performance of a join algorithm. If some

build tuples can be maintained in memory while partitioning then those tuples do not need to be written to disk and then re-read later during the join.

Hybrid hash join (HHJ) [7] is a standard hash join algorithm used in database systems. HHJ works by partitioning the build relation as described in Section 2.2.2 except the first partition (partition 0) is stored in memory and all other partitions are stored on disk. The number of partitions is determined just like in hash join by calculating what percentage of the build relation will fit in memory. If 20% of the build relation will fit then one fifth of the relation will fit in memory and 5 partitions are used.

At the start of the join a file is created on disk for each partition other than the first one. A hash table is created in memory to store the build tuples of build relation partition 0. When tuples are read during partitioning of the build relation any tuples that do not fall into partition 0 are appended to the end of the on-disk file for the corresponding partition. Any tuples that fall into partition 0 are placed in the in-memory hash table.

After the build relation tuples are partitioned a file is created on disk for each of the probe relation partitions except the first one. Probe tuples are then read one at a time from the probe relation and the hash value of their join attribute value is calculated. If the probe tuple falls into the first partition then the in-memory hash table is probed with that probe tuple and any matches are returned as result tuples exactly as in the previous example. The probe tuple can then be discarded. It is not necessary to write it to disk as all result tuples it would create have been generated. If the tuple falls in any other partition then it is appended to the end of the corresponding partition file on disk for the probe relation.

When all of the probe tuples have been read from the probe relation the cleanup phase begins. All tuples and the hash table are deleted from memory. An in-memory hash table is created and the tuples in the disk file for the second build relation partition (partition 1) are then loaded into it. Probe tuples are then read one at a time from the disk file for the second probe relation partition (partition 1) and used to probe the hash table. Any matches are returned as result tuples as above. All tuples and the hash table are then removed from memory and the two files that were just read are deleted from disk. This process is repeated for each partition that has a file on disk (partitions 2, 3, and 4). After this process the join is finished.

The partitioning process assumes that an equal amount of tuples will fall into each build partition. This assumption makes sense if the data set is uniform. If, for example, it turns out that more than 20% of the build tuples hash to partition 0 then too much memory will be used to store these tuples. Various database systems handle this issue differently. The HHJ implementation in the PostgreSQL database system repartitions with twice as many partitions until it is no longer using too much memory.

If almost no build tuples hash to partition 0 then the algorithm is not making proper use of its memory and could perform faster if it kept more build tuples in memory. The solution to this problem is an algorithm called Dynamic Hash Join.

2.2.5 Dynamic Hash Join

Dynamic hash join (DHJ) [8, 21] is similar to hybrid hash join except that it dynamically selects memory-resident partitions during execution. Instead of picking only one partition to remain memory-resident before the join begins, DHJ allows all partitions to be memory-resident initially and then flushes partitions to disk as required when memory is full.

Although DHJ adapts to changing memory conditions, there has been no research on determining what is the best partition to flush to maximize performance. Various approaches select the largest or smallest partition, a random partition, or use a deterministic ordering. No approach has considered using data distributions to determine the optimal partition to flush. In the examples a deterministic ordering will be assumed because it is the simplest. The highest numbered partitions will be flushed first.

The DHJ algorithm is similar to HHJ. DHJ determines what percentage of tuples can fit in memory and then creates a number of partitions. If 20% of the tuples can fit in memory then assuming a uniform distribution of tuples (an equal amount will hash to each partition) the algorithm will need at least 5 partitions. Often because a database system knows a uniform distribution of tuples is unlikely it will create more partitions than the above estimation in an effort to compensate for data that is not uniform. In this example, 10 partitions may be used. It is also possible that the tuples are not evenly divided among the partitions due to the hash function used (partition skew). When joining *Part* and *Purchase* the build tuples from *Part* are uniformly distributed which is acceptable for this example: 20% of the *Part* relation (the maximum amount that fits in memory) is equivalent to 2 tuples.

While partitioning the build relation DHJ starts with all build partitions in memory. A hash table is stored in memory for each build partition. It reads the tuples in order from the *Part* relation and flushes a partition to disk only when more than 2 tuples will be stored in memory. When a partition is flushed to disk its hash table is removed from memory as well.

Tuple 1 (the tuple with *partid* 1) is read from the *Part* relation and placed in partition 1 (Figure 2.8a). The build partitions are now using 10% of memory. Tuple 2 is read and placed in partition 2 (Figure 2.8b); 20% of memory is now used. Tuple 3 is read and placed in partition 3 (Figure 2.8c). Because memory is over-full, partition 4 is flushed and its tuples written to disk (Figure 2.8d). This has been referred to as a *frozen* [8, 18] partition in previous work. It is marked as frozen so that any further tuples that would fall into that partition are immediately placed on disk. This did not free up any memory so partition 3 is also flushed. The disk file for partition 3 now contains tuple 3 and memory is again only 20% full (Figure 2.8e). Tuple 4 is read and written to disk because partition 4 was previously frozen (Figure 2.8f). Tuple 5 is read and placed in partition 0 (Figure 2.9a). Memory is over-full again so partition 2 is flushed to disk which puts the partitions back at the memory limit (Figure 2.9b). Tuple 6 is read and placed in partition 1 (Figure 2.9c) which causes partition 1 to be flushed to disk (Figure 2.9d). Only 10% of memory is now in use. Tuples 7, 8, and 9 are all read and written to disk (Figure 2.9e). Tuple 10 is read and placed in partition 0 in memory (Figure 2.9f). All build tuples have now been partitioned and the build partitions are not over-using memory.

Reading tuples from the probe relation is handled exactly as for HHJ, except if multiple

partitions had been kept in memory after partitioning the build relation then any probe tuples that hashed to an in-memory partition could have generated result tuples before the cleanup phase. The cleanup phase is handled exactly as for HHJ.

2.2.6 Hash Join Performance Enhancements

It is possible for the database optimizer to poorly estimate the size of the underlying relations and choose the larger one to be the build relation. After partitioning the relations, a hash join knows the exact size of its inputs. If the build input is larger than the probe input hash join can swap them and use the probe input as the build input instead. This is called role reversal [12].

An optimization that can be used during the cleanup phase after the input relations have been partitioned is to process multiple partitions at the same time. In this optimization, instead of performing one cleanup iteration per partition, each cleanup iteration operates on as many partitions that can fit in memory.

When a single partitioning step is not sufficient to construct partitions that fit in memory, multiple rounds of recursive partitioning are used. Recursive partitioning avoids having many (hundreds) of open files during partitioning which increases the impact of random I/Os while writing to those files.

2.3 Skew

Skew can be classified [23] as either *partition skew* or *intrinsic data skew*. Partition skew is when the partitioning algorithm constructs partitions of non-equal size (often due to intrinsic data skew but also due to the hash function itself). Minimizing partition skew has been considered for distributed databases [10] and DHJ [15, 21]. Partition skew can be partially mitigated by using many more partitions than required, as in DHJ, and by producing histograms on the data when recursive partitioning is required.

Consider two relations $R(\underline{A})$ and $S(\underline{B}, A)$ where attribute A is the join attribute between R and S. The underlined attributes are the primary key attributes of the relations. Assume that the number of tuples of R, denoted as $|R|$, is smaller than the number of tuples of S (i.e. $|R| < |S|$). Assume $S.A$ is a foreign key to $R.A$. In a hash join of R and S the build relation is R and the probe relation is S.

When performing a hash join most systems have no intelligent way of selecting which partition remains memory-resident. PostgreSQL simply selects the first partition. This assumption makes sense if the data set is uniform. In that case, each tuple in R is equally likely to join to tuples in S, so it does not matter what tuples in R are left in memory. If the data is skewed such that certain tuples in R join to many more tuples in S than the average, it is preferable that those tuples of R remain in memory.

Intrinsic data skew is when data values are not distributed uniformly. Intrinsic data skew may cause partition skew for hash joins when the join attribute on the build relation is not the primary key attribute. Data skew causes values to occur with different frequencies in

the relations. In the example that joins $R.A = S.A$ (primary-to-foreign key join), data skew may cause the distribution of values of $S.A$ (probe relation) to vary dramatically.

Histograms have been previously used during recursive partitioning [12] to detect data skew and minimize partition skew. However, no previous work has discussed using existing histograms in the database system to estimate and exploit skew in the probe relation while partitioning the build relation.

2.4 Statistics and Histograms

Most commercial database systems create and maintain some form of aggregate statistics on their relations to be used for optimizing query processing. A database will keep an accurate count of how many tuples are in each relation and the average size of a tuple so that when a query is performed it can correctly calculate the amount of memory necessary at each step in the query and which order of operations will perform the fastest.

When a selection or filter is performed on a relation as part of a query, a database can make educated guesses as to how many tuples will make it past the filter based on the filter being used. If the database knows that there are roughly 10 distinct values in the attribute being filtered on and the filter is something of the form "*attribute* = *value*" then it could assume that one tenth of the relation tuples will match the filter.

partid	**Frequency**
1	1
2	8
3	3
4	2
5	1
6	1
7	1
8	1
9	1
10	1

Table 2.5: An Accurate Histogram on the partid Attribute of the Purchase Relation

When there is a uniform distribution of values in an attribute this is a safe estimate, but this is not always the case. The *partid* attribute of the *Purchase* relation in Table 2.2 is a perfect example of a case when this estimate would be quite incorrect. There are 10 distinct values in the *partid* attribute, but they are not uniformly distributed. While the average frequency of each value is 2 the value 2 occurs 4 times as often. A histogram of the *partid* attribute of the *Purchase* relation is provided in Table 2.5. Generally a histogram bucket is defined by the range of values that fall into the bucket and the aggregate frequency of all those values (how many times those values occur). The first column of this histogram is the attribute value and the second column is how frequently that value occurs in all the tuples of the relation. Not all histograms are organized this way (see [13]).

With a pre-generated histogram on the *partid* attribute the query optimizer can more accurately estimate the result of a filter. If the filter is "*partid* = *2*" the database would estimate that 8 tuples would pass the filter. The histogram in Table 2.5 is ideal because every distict value in the *partid* attribute has its frequency stored.

In a large relation with millions of tuples a histogram aggregates attribute values so that one bucket contains many values. A simple example of how this affects the accuracy of the histogram can be seen in Table 2.6 where the bucket value ranges now contain 2 values each. The frequency column of the histogram specifies the sum of all the frequencies of the individual values in the bucket. Since no individual frequencies are maintained a uniform assumption must be made where it is assumed that each value in the bucket corresponds to an equal portion of the frequency of the bucket. With the filter "*partid* = 2" and this second histogram the database would estimate that 5 (rounded up from 4.5) tuples would pass the filter because the frequency of the bucket is 9 and the number of distinct values in the range of the bucket is 2.

partid Range	**Frequency**
1-2	9
3-4	5
5-6	2
7-8	2
9-10	2

Table 2.6: An Aggregate Histogram on the partid Attribute of the Purchase Relation

To compensate for the bad estimates caused by aggregate histograms many specialized histogram types are used by various database systems. For example, an end-biased histogram maintains the most frequent values individually from the rest of the buckets. It keeps track of the frequency of those individual values as well as the frequency of each bucket. End-biased histograms take up only slighly more space than our example histograms but allow the most frequent values to be known and exploited. Equi-depth histograms arrange the buckets so that each contains the same number of values and consequently the frequency of all buckets is the same but the width of the bucket (number of values in its range) varies.

When the input to a join is not a base relation (it could be another join or a selection for example) the base relation statistics may not accurately reflect the distribution of tuples on that input. One method to compensate for this is SITs (Statistics on Intermediate Tables) [2]. When a database system supports SITs it occasionally runs common queries that involve joins and other operators and stores the statistics of the intermediate results of these queries. The query optimizer would then use these statistics to improve any estimates for queries that are equivalent to the common queries or when one of the common queries is a sub query of the current query being estimated.

For a further discussion of histograms see [13, 14, 22].

2.4.1 Histograms and Hash Joins

In the preceding discussion of HHJ and DHJ it was apparent that keeping some of the build tuples in memory while partitioning the build relation not only kept those build tuples from having to be written to and re-read from disk, it also kept the related probe tuples from having to be written to and re-read from disk. The histograms in Tables 2.5 and 2.6 show that some of the build tuples are related to more probe tuples than the other build tuples are. If the hash join knew this information when it was choosing which build tuples to keep in memory during the build phase, it could save many more tuple disk writes and reads.

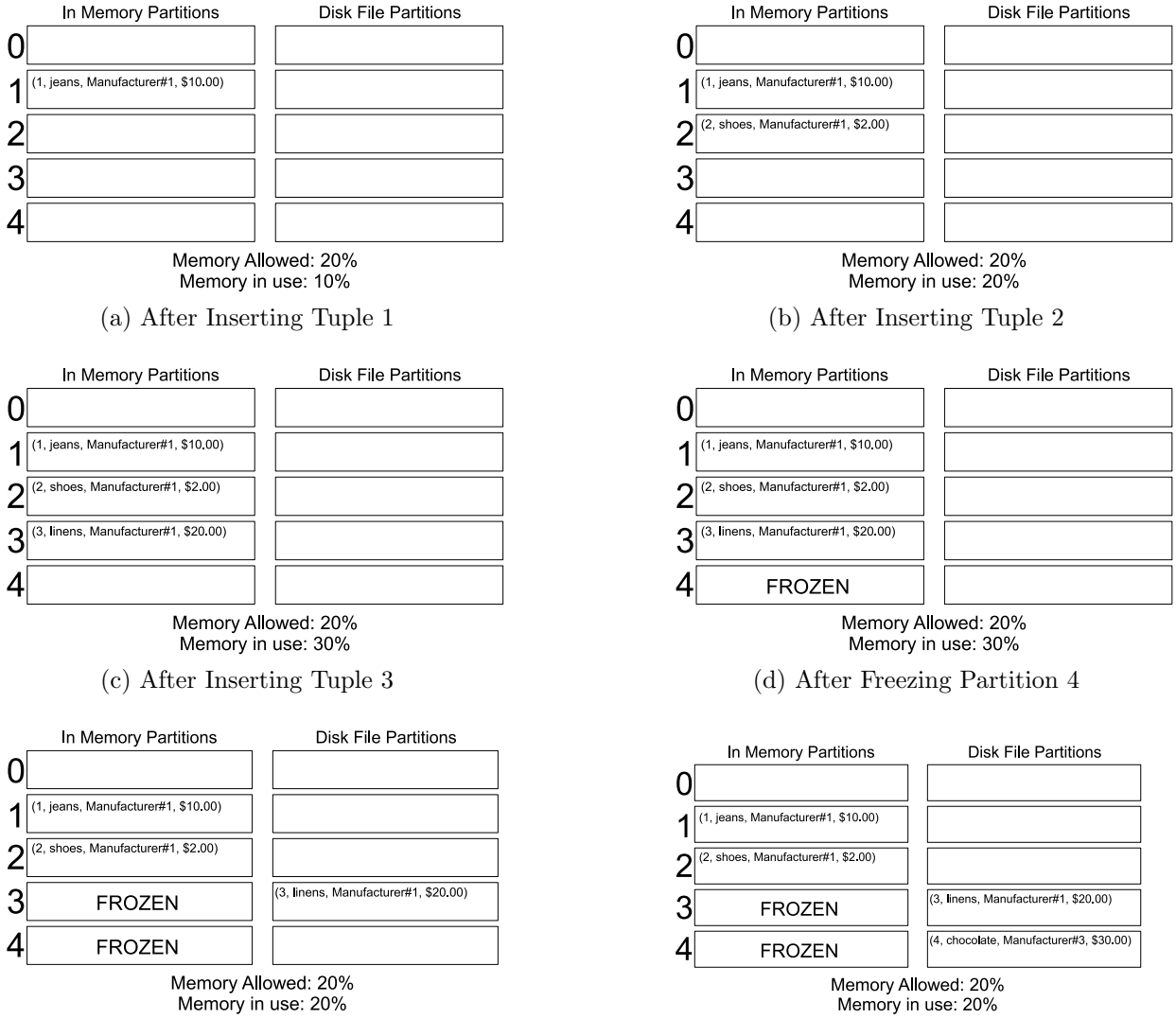
If the histogram for the *partid* attribute of the *Purchase* relation was examined before partitioning the *Part* relation, the order that DHJ freezes partitions could be changed so that partition 1 stays in memory and partition 0 is flushed to disk. If this could be done the number of result tuples generated during the probe phase would increase significantly and the number of probe tuples written to disk during the probe phase and re-read during the cleanup phase would decrease.

2.5 Example Database

In this thesis, the TPC-H database benchmark standard is used as an example database. TPC-H is a decision-support and data warehouse benchmark developed by the Transaction Processing Performance Council (TPC). More information about the TPC-H benchmark can be found at [1]. The TPC-H schema diagram is in Figure 2.10. The *SF* in the diagram represents the scale factor of the relations. Scale factors 1 and 10 will be used which produce total database sizes of approximately 1 and 10 GB respectively. The largest relation, *LineItem*, has just over 6 million tuples for SF=1 and 60 million tuples for SF=10.

2.6 Relational Algebra Query Plan Diagrams

In this thesis relational algebra (RA) diagrams are used to describe the order in which multiple relations are joined in a single database query. RA diagrams can contain many operators but in this thesis only joins and relations are represented in these diagrams. In Figure 2.11 the example RA diagram *LI* represents the *LineItem* relation, *S* represents the *Supplier* relation, and *P* represents the *Part* relation of the TPC-H database benchmark described in Section 2.5. The transposed hourglass symbol represents a join of the relations connected below it. In Figure 2.11 *Supplier* is joined with *LineItem* and then the result of this join is joined with *Part*.



(a) After Inserting Tuple 1

(b) After Inserting Tuple 2

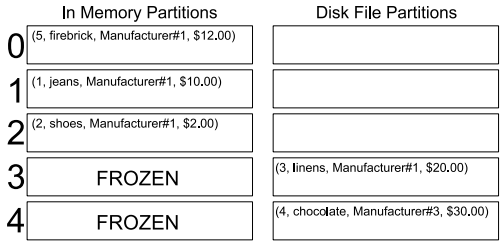
(c) After Inserting Tuple 3

(d) After Freezing Partition 4

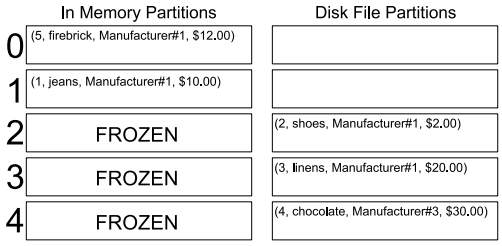
(e) After Freezing Partition 3

(f) After Inserting Tuple 4

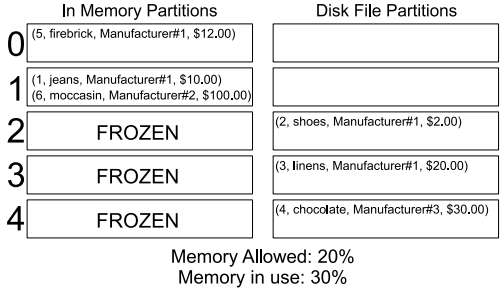
Figure 2.8: DHJ Partitioning Example Part 1



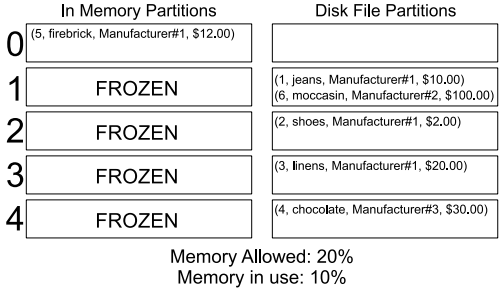
(a) After Inserting Tuple 5



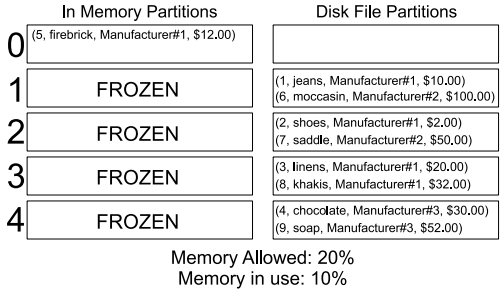
(b) After Freezing Partition 2



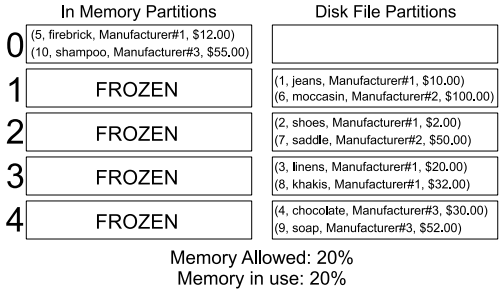
(c) After Inserting Tuple 6



(d) After Freezing Partition 1



(e) After Inserting Tuples 7, 8, and 9



(f) After Inserting Tuple 10

Figure 2.9: DHJ Partitioning Example Part 2

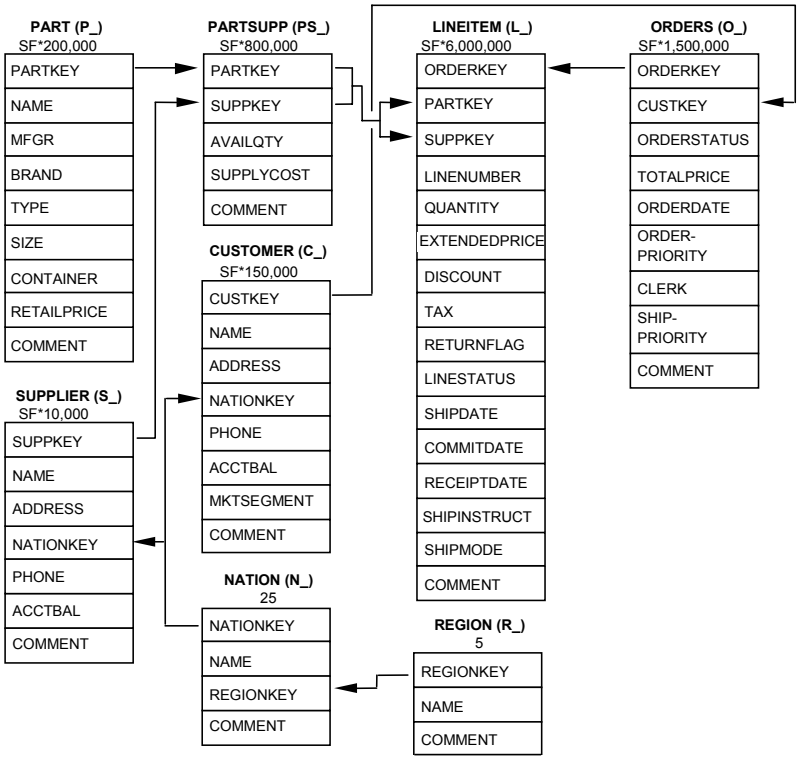


Figure 2.10: TPC-H Schema from [1]

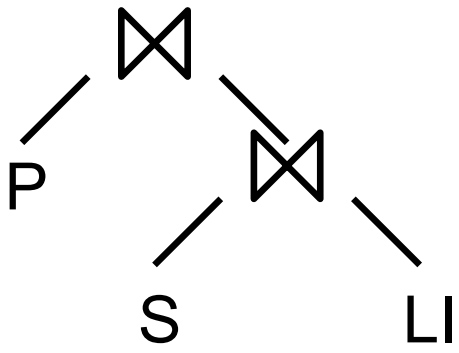


Figure 2.11: Example Relational Algebra Diagram

3. Histojoin

3.1 General Approach

The *Histojoin* algorithm is designed to use statistics and histograms as currently implemented in the database system. The algorithm does not assume any histogram method and will work with any method. Commercial systems typically implement equi-depth [20] or maxdiff (Microsoft SQL server) histograms. An overview of histograms can be found in [13, 14, 22]. The actual construction of the histograms is orthogonal to this work.

The general approach is to use the extra memory available to the hash join to buffer the tuples that participate in the most join results. Consider a primary-to-foreign key join between $R(\underline{A})$ and $S(\underline{B}, A)$ on A, where R is the smaller relation and some subset of its tuples are buffered in memory. Unlike hybrid hash join that selects a random subset of the tuples of R to buffer in memory, the tuples buffered in memory will be chosen based on the values of A that are the most frequently occurring in relation S.

For example, let R represent a $Part$ relation, and S represent a $Purchase$ relation. Every company has certain parts that are more commonly sold than others. A common part may be associated with thousands of purchases and a rare part only a handful. If a single part tuple ordered thousands of times is kept in memory when performing the join, every matching tuple in $Purchase$ does not need to be written to disk and re-read during the cleanup phase.

Hash partitioning randomizes tuples in partitions. This is desirable to minimize the effect of partition skew, but data skew is also randomized. Traditional hash joins have no ability to detect data skew in the probe relation or exploit it by intelligent selection of in-memory partitions.

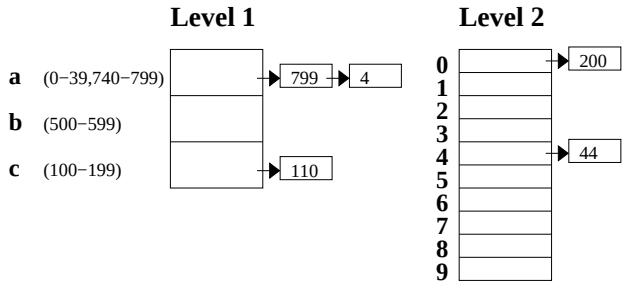


Figure 3.1: Two Level Partitioning

This approach uses two levels of partitioning. The first level performs range partitioning where ranges of values of $R.A$ are selected to be memory-resident. Tuples that do not fall

into the ranges are partitioned using a hash function as usual. The data structures used are shown in Figure 3.1.

In Figure 3.1 there are 3 in-memory partitions (a, b, c) and 10 hash partitions numbered 0 to 9. For Level 1 partitions, each partition is defined by one or more join attribute value ranges. For example, partition a consists of values from 0 to 39 and 740-799. Ideally, these attribute values are the most frequently occurring in S. The maximum partition size is bounded by the memory size available to the join. The Level 2 partitions are regular hash partitions. If a tuple does not fall into any of the Level 1 partitions, it is placed in a Level 2 partition by hashing the join attribute value. In general, there may be multiple Level 1 memory-resident partitions each defined by multiple ranges of values. The only constraints are that each partition must fit in the available memory during a cleanup phase at the end of the join, and the total memory used by in-memory partitions is always below the memory available.

3.1.1 Theoretical Performance Analysis

This section provides the theoretical maximum improvement of skew-aware partitioning using data distributions versus random partitioning (dynamic hash join). Let f represent the fraction of the smaller relation (R) that is memory-resident: $f = M/|R|$ (approximately), where M is the memory size. The number of tuple I/O operations performed by dynamic hash join is $2 * (1 - f) * (|R| + |S|)$. The factor 2 represents the two I/Os performed for each non-memory-resident tuple: one to flush to disk if not memory-resident and then one to read again during the cleanup phase of the join. Note that this does not count the cost to read the tuple initially.

Let g represent the fraction of the *larger* relation (S) that joins with the in-memory fraction f of R. If the distribution of the join values in S is uniform, then $f = g$. Data skew allows $g > f$ if memory-resident tuples are chosen properly. The number of I/Os performed by *Histojoin* is $2 * (1 - f) * |R| + 2 * (1 - g) * |S|$. The absolute difference in I/Os performed between DHJ and *Histojoin* is $2 * (1 - f) * (|R| + |S|) - (2 * (1 - f) * |R| + 2 * (1 - g) * |S|)$ which simplifies to $2 * (g - f) * |S|$. A negative number indicates DHJ is outperforming *Histojoin*, while a positive number indicates *Histojoin* performs better than DHJ. The percentage difference in I/Os is $\frac{(g-f)*|S|}{(1-f)*(|R|+|S|)}$.

The absolute difference in total I/Os performed given selected values of f and g is given in Table 3.1. The percentage difference in total I/Os performed is given in Figure 3.2. The absolute difference is directly proportional to the difference between f and g. The table shown is for a 1:1 ratio of R and S where $|R| = |S| = 1000$. The difference between f and g is bounded above by the intrinsic skew in the data set and is limited by how we exploit that skew during partitioning.

Properly exploiting data skew allows $g > f$, but if the in-memory tuples are chosen poorly, it is possible for $g < f$. This is worse than the theoretical average of the uniform case for dynamic hash join. Tuples may be chosen improperly if the statistics used for deciding which tuples to buffer in memory are incorrect causing the algorithm to buffer worse than average

tuples.

As an example, consider a data set following the "80/20 rule". If we can keep 20% of tuples of R in-memory ($f = 20\%$) that join with 80% of the tuples in S ($g = 80\%$), then a skew-aware join will perform 68% fewer tuple I/Os than hybrid hash join. However, if the data had "80/20 skew", but the 20% of tuples of R buffered in-memory were the least frequently occurring in S, then it may be possible for $g = 5\%$, resulting in skew-aware join performing 17% more I/Os than hash join.

	g						
f	**5%**	**10%**	**20%**	**50%**	**80%**	**90%**	**100%**
5%	0	100	300	900	1500	1700	1900
10%	-100	0	200	800	1400	1600	1800
20%	-300	-200	0	600	1200	1400	1600
50%	-900	-800	-600	0	600	800	1000
80%	-1500	-1400	-1200	-600	0	200	400
90%	-1700	-1600	-1400	-800	-200	0	200

Table 3.1: Absolute Reduction in Total I/Os of Skew-Aware Partitioning versus Random Partitioning for Various Values of f and g and $|R| = |S| = 1000$

3.2 Histojoin Algorithm

A low cost technique for performing skew-aware partitioning is by using histograms. Histograms [13] are used in all commercial databases for query optimization and provide an approximation of the data distribution of an attribute. A histogram divides the domain into ranges and calculates the frequency of the values in each range. An example histogram produced by Microsoft SQL Server 2005 for the TPC-H relation *Lineitem* on attribute *partkey* is provided in Figure 3.3.

The advantage of using histograms is that they are readily available, calculated and maintained external to the join algorithm, and require no modification to the query optimizer or join algorithm to use. On examination of the histogram, the query optimizer can determine if the *Histojoin* algorithm will be beneficial. An imprecise or out-of-date histogram limits *Histojoin*'s ability to exploit the data skew.

3.2.1 Algorithm Overview

The *Histojoin* algorithm works by implementing a set of privileged partitions in addition to the partitions dynamic hash join would normally use. These privileged partitions are the last partitions to be flushed from memory to disk and are arranged so that they are flushed in a specific order, whereas the non-privileged partitions are flushed in a randomized order. The privileged partitions correspond to the Level 1 partitions shown in Figure 3.1.

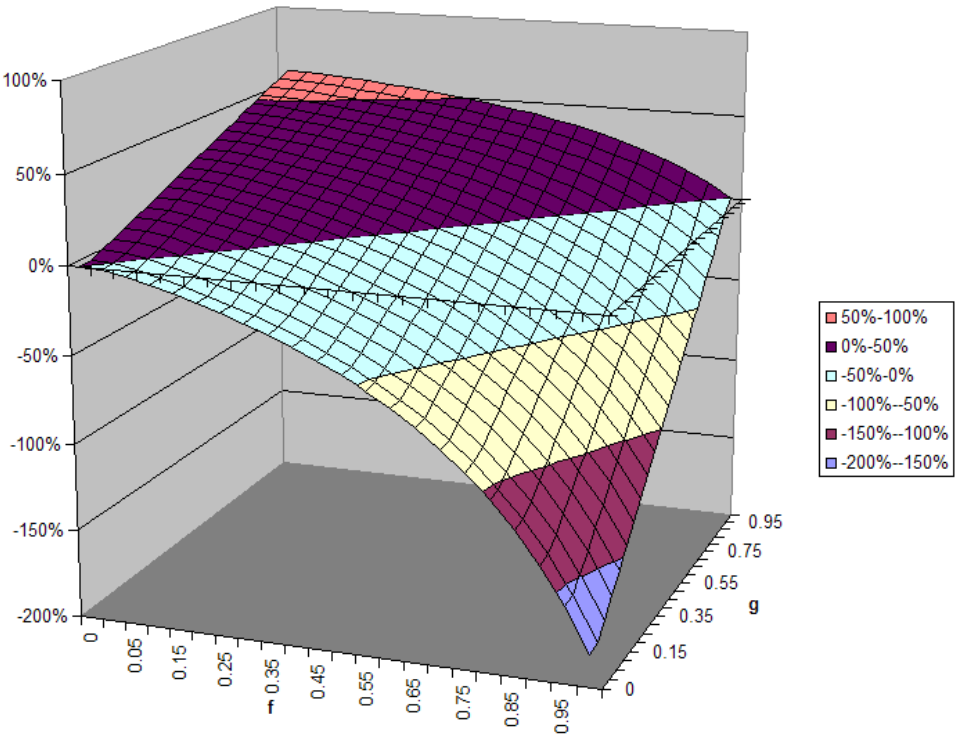


Figure 3.2: Total I/Os Percent Difference

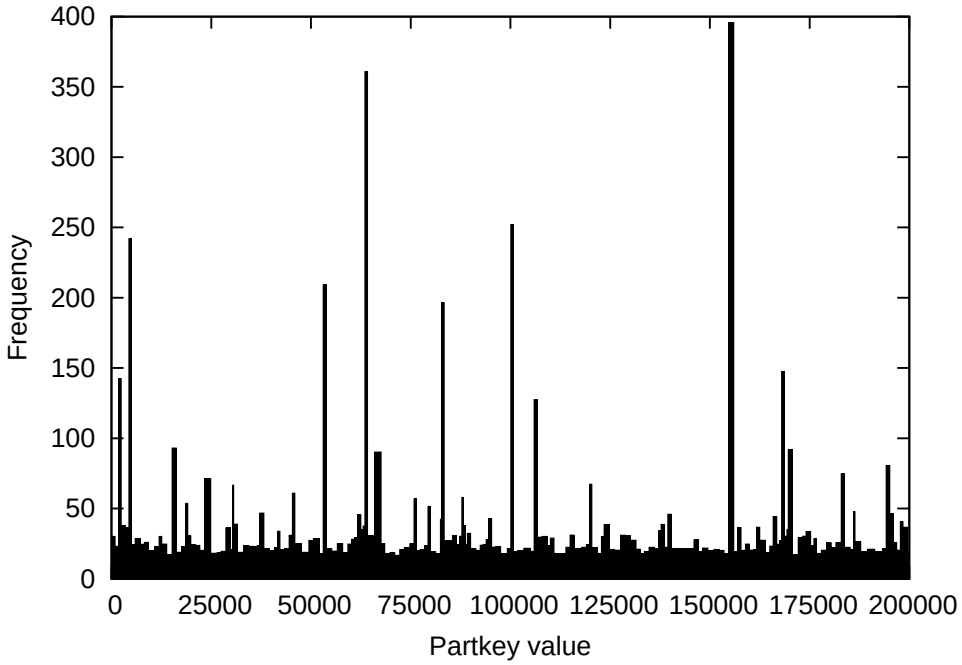


Figure 3.3: Partkey Histogram for Lineitem Relation TPC-H 1 GB Zipf Distribution (z=1)

The differences between *Histojoin* and dynamic hash join are isolated in the hash table. *Histojoin*'s hash table is a two layered table that is aware of which partitions are privileged, how to determine if tuples fall in the privileged partitions, and how to optimally flush the partitions by first randomly flushing non-privileged partitions and then flushing the privileged partitions in order of worst to best.

The key difference between *Histojoin* and dynamic hash join is that *Histojoin* attempts to isolate frequently occurring tuples in the privileged partitions which are the last ones flushed. Dynamic hash join spreads out frequently occurring tuples across all partitions and provides no special handling for frequent tuples.

A flowchart describing the *Histojoin* algorithm is in Figure 3.4. The first step is to load the histogram and determine which tuple value ranges are in the privileged partitions. At this point, if insufficient skew is detected or there is limited confidence in the histogram, a decision is made on the maximum size of the privileged partitions. If no skew is detected, *Histojoin* will allocate no memory to the privileged partitions, and the algorithm behaves identically to dynamic hash join. Determining the privileged partitions is discussed in Section 4.2.

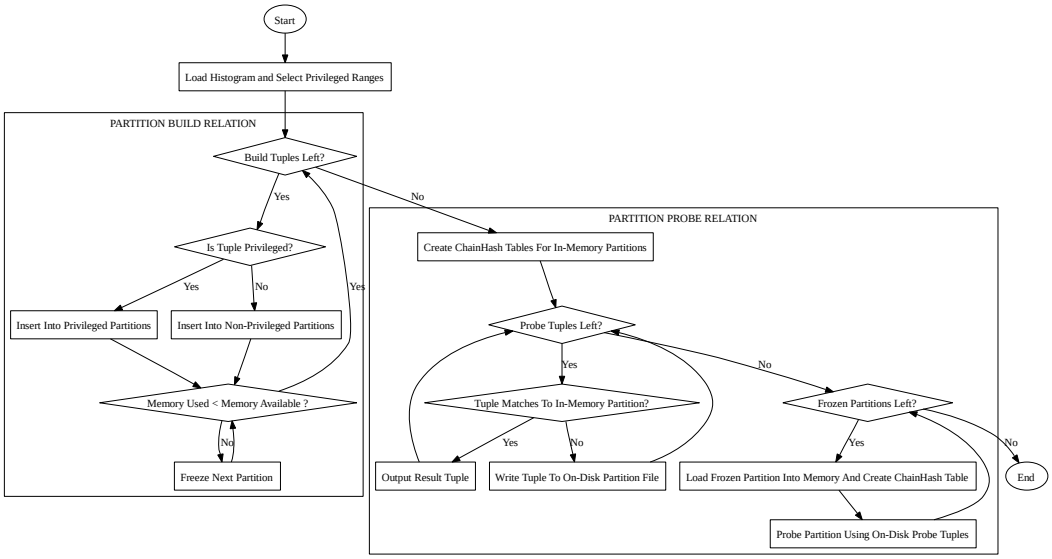


Figure 3.4: Histojoin Flowchart

Given sufficient detected skew, the privileged partition ranges are organized into efficient data structures to allow rapid determination of privileged tuples. Each build and probe tuple requires a range check to determine if they belong in a privileged partition. The range check operation is discussed in Section 4.3.

Histojoin processes the join in a similar manner to dynamic hash join. Tuples are read from the build relation. When a tuple is read, the range check is performed. If the tuple falls into a privileged partition, it is placed there. Otherwise, the tuple's partition is determined using a hash function similar to DHJ. Whenever memory is full while the build relation is

being read, a partition flush is performed. Non-privileged partitions are flushed first. If all non-privileged partitions are flushed, the privileged partitions are flushed in reverse order of benefit. When flushing the non-privileged partitions, we flush in random order. Once a partition is flushed, a single disk buffer is allocated to the partition to make writing tuples to the disk file more efficient. A flushed partition cannot receive any new tuples in memory.

Once the build relation is completely read, there will be some build partitions in memory and others in disk files. Partitions that are memory-resident have main memory (chained) hash tables constructed to store their tuples. These hash tables will be probed using tuples from the probe relation.

The probe relation tuples are then read. The range check is performed on each tuple. If the tuple corresponds to an in-memory build partition (privileged or not), it probes the chained hash table for that partition to potentially generate results. If the corresponding build partition is not in-memory, the probe tuple is written to the probe partition file on disk. Once all probe relation tuples are read, there will be pairs of build and probe partitions on-disk. Main memory is cleared, and each partition pair is read from disk and processed. Typically, the build relation partition is read, a chained hash table produced, and then results are generated by probing using probe relation tuples. However, common practices such as those described in Section 2.2.6 can be applied.

In summary, *Histojoin* behaves like dynamic hash join except that its hash table structure allows for the identification and prioritization of frequently occurring tuples in the probe relation. The differences between DHJ and *Histojoin* are embedded in the distribution of tuples between the two layers of the hash table and the order in which partitions are frozen to disk to free up memory. All other hash join techniques are unaffected by these modifications.

3.2.2 Selecting In-Memory Tuples

Given a histogram that demonstrates skew, *Histojoin* must determine a set of join attribute range(s) that constitute the most frequently occurring values in S. Tuples of R with these frequently occurring values are the ones in the privileged partitions. For instance in Figure 3.3 there are several ranges of part keys that occur frequently in *LineItem*. The challenge is that the join partition size is determined independently from histogram partitioning. For example, let $|R| = 1000$ and $M = 100$. Thus, at least 10 partitions of R are required. The in-memory partition can have 100 tuples. It may require multiple independent ranges in the histogram to define a set of attribute ranges that contain up to 100 distinct values of R and have high frequency in S.

The greedy algorithm reads the histogram for S on the join attribute, sorts its buckets by frequency, and selects as many buckets that fit into memory in the order of highest frequency first. The detailed steps are:

- Assume each histogram bucket entry is a 6-tuple of the form *(MINVAL, MAXVAL, ROWCOUNT, EQROWS, DISTINCT_ROWS, ROWS_R)*. *MINVAL* and *MAXVAL* are the lower and upper (inclusive) values defining the bucket range. *ROWCOUNT* is the number of rows in S with a value in the range of *[MINVAL,MAXVAL)*. *EQROWS* is

the number of rows in S whose value is exactly equal to $MAXVAL$. $DISTINCT_ROWS$ is the distinct number of values in S in the bucket range. $ROWS_R$ is a derived value (not present in the histogram) that is the estimate of the number of rows in R that have a value in the histogram bucket range. The estimation of $ROWS_R$ is given in Section 4.2.1.

- A bucket *frequency* is calculated as:
  $(ROWCOUNT\ +\ EQROWS)\ /\ ROWS_R$.
- Sort the buckets in decreasing order by frequency.
- The sorted list is traversed in order. Assume the size of memory in tuples is M, and *count* is the number of tuples currently in the in-memory partition. A histogram bucket range is added to the in-memory partition if $count\ +\ ROWS_R\ <=\ M$.
- The previous step is repeated until the histogram is exhausted, there is no memory left to allocate, or the current bucket does not fit entirely in memory.

Consider the histogram in Figure 3.2, and a join memory size of 400 tuples. The first histogram bucket added has range 751-1000 (250 tuples) as its frequency is 4.6. The second histogram bucket added has range 101-200 with frequency 3.1. The remaining memory available can be allocated in various ways: leave as overflow, find next bucket that fits, or divide a bucket. With integer values, it is possible to take the next best bucket and split the range. In this case, the range from 1-100 can be divided into a subrange of 1-50.

MINVAL	MAXVAL	ROWCOUNT	EQROWS	DISTINCT_ROWS	ROWS_R	FREQ
1	100	300	5	100	100	3.05
101	200	300	10	100	100	3.1
201	350	150	100	150	150	1.67
351	500	200	40	150	150	1.6
501	750	244	6	250	250	1
751	1000	650	500	250	250	4.6

Table 3.2: Histogram Partitioning Example

Not all histograms will separate out the frequency of a boundary value (such as $EQROWS$) in that case the frequency is calculated as $ROWCOUNT/ROWS_R$. When a histogram does separate out the frequency of a boundary value (such as with maxdiff histograms), these values can be used as separate bucket ranges as they typically have very high frequencies. These single, high-frequency values are referred to as *premium values*. Premium values have a high payoff as they occupy little memory (one tuple each) and match with many rows in the probe relation. Premium values tend to be good values to keep in-memory even when the accuracy in the histogram is low, especially when they are significantly more common than the average value.

Note in the example that value 350 occurs 100 times even though on average the other values in the range of 201-349 only occur once. Tuple with key 350 should be memory-resident. The algorithm creates separate one value ranges for each separation value. When

sorted, these ranges may be selected independently of the rest of their histogram bucket. For example, with a memory size of 400 tuples, the algorithm selects the following ranges: 1000, 350, 500, 200, 750, 100, 751-999, 101-199, and 1-46. (The last range is a partial range of 1 to 100.) A tuple is in the in-memory partition if it falls in one of these ranges.

Estimating Cardinality Of Value Ranges

A histogram *estimates* the number of distinct values and number of tuples in a histogram bucket (*DISTINCT_ROWS* and *ROWCOUNT* respectively). If the histogram is on the probe relation S, then the histogram provides the number of tuples in each bucket for relation S. However, *Histojoin* also requires an estimate of the number of tuples in R that have a value in a histogram bucket. This estimate is used to determine approximately how many histogram buckets can be memory-resident for the build relation R. This value is also used to determine the relative value of each bucket. Histogram buckets with few rows in R and numerous rows in S are prime candidates for privileged partitions.

Given the number of distinct values in S, $DISTINCT_ROWS$, the estimate of the number of rows in R with values in that range, $ROWS_R$, is determined as follows:

- For integer values, it is calculated using the bucket low and high range values. That is, $ROWS_R = MAXVAL - MINVAL + 1$.
- For non-integer values, it is estimated as $ROWS_R = DISTINCT_ROWS$.

There will be inaccuracy in estimating $ROWS_R$ for non-integer values. For one-to-many joins, $ROWS_R$ will be underestimated due to primary key values not appearing in the foreign key relation. For many-to-many joins, it is impossible to determine exactly how many rows in R will have values in the range without having a histogram on R as well. Some heuristics can be used based on the size of relation R, but in general, the estimate may be imprecise. Thus, it is critical that *Histojoin* adapts its memory management to flush even privileged partitions in case of inaccurate estimates. This is discussed further in Section 3.2.3.

3.2.3 Partitioning

Histojoin partitions tuples in two layers. The first layer contains privileged partitions with join attribute ranges that are defined as described in Section 4.2. A tuple is placed in a privileged partition if its join attribute value falls into one of the privileged partition ranges. This is performed using a range check function. A *range check* is performed for each tuple by comparing its join attribute value with the ranges calculated in Section 3.2.2. In this example, the ranges are 1000, 350, 500, 200, 750, 100, 751-999, 101-199, 1-46.

For efficiency, the range check is implemented in two steps. The first step uses a hash table to record all ranges of size one. Each hash table entry maps the join attribute to a partition number. This step is used for very frequently occurring values such as premium values. The size of this hash table is very small, usually less than 200 entries, as the number

of premium values is limited based on the number of histogram buckets. The second step processes all ranges of size greater than one by storing them in a sorted array. This sorted array is searched using a binary search to detect if a value is in one of the ranges.

When a range check is performed the value is first tested against the hash table. If it is in the hash table then the mapped partition is returned. If not then a binary search is performed on the sorted array, and if the correct range is found the related partition is returned. If the value is not found in either of these two structures then it does not fall in a privileged partition, and the value is hashed to find which non-privileged partition it belongs in. For tuples with join values that do not fall into privileged partition ranges, the tuples are placed in hash partitions using a hash function. This hash partitioning works exactly the same as in dynamic hash join.

This hash table and search array method works for all types and combinations of values. A further speed optimization for integer values is to enter every value that falls in a range into the hash table and not use the binary search. This works for integer values because the possible values in a range can be discretely enumerated.

3.3 Using Histojoin

This section contains a discussion of some of the issues in using *Histojoin*. These issues include handling different join cardinalities, tolerating inaccuracy in histograms, and supporting joins where the input relations are from selection or other join operators.

3.3.1 Join Cardinality

Although the previous examples considered primary-to-foreign key joins, *Histojoin* works for all join cardinalities. *Histojoin* is useful when there is a histogram on the join attribute of the probe relation, and the probe relation has skew. If due to filtering the foreign key relation is the smaller (build) relation, *Histojoin* is not usable because the probe (primary key) relation is uniform, and there is no skew to exploit. However, it may be possible to reverse the roles and still make the larger foreign key relation the probe relation if there is skew to exploit that improves performance.

Histojoin adds no benefit over dynamic hash join for one-to-one joins due to the uniform distribution of the probe relation. In this case, *Histojoin* behaves exactly as dynamic hash join and allocates no privileged partitions.

For many-to-many joins, *Histojoin* only requires the histogram on the probe relation. The algorithm behaves exactly as in the one-to-many case, but execution of the algorithm may result in flushing privileged partitions as the size estimates of the privileged build partitions are less accurate. For example, a histogram may indicate that the values from 5 to 10 have high frequency in the probe relation. *Histojoin* will estimate that there are 6 tuples in the build relation in that range. However, there may be multiple occurrences of each value such that there are actually 30 tuples in the build relation with values in that range. This may force *Histojoin* to flush some privileged partitions to compensate for the

over-allocation of memory. A histogram on the build relation may mitigate some of these estimation concerns, but may be hard to exploit as independently produced histograms may have widely differing bucket ranges. Even when *Histojoin* over-allocates privileged partition ranges, dynamic flushing based on frequency improves performance over dynamic hash join while avoiding memory overflows.

The join cardinality cases are enumerated in Table 3.3.

Type	Larger Side	Approach	Special Notes
1-1	Either	behave like DHJ	No skew in relations.
1-M	1	behave like DHJ	No skew in probe.
			Evaluate role reversal if skew on many-side.
1-M	M	use probe histogram	Skew can be exploited.
M-N	M or N	use probe histogram	Skew can be exploited.

Table 3.3: Join Cardinality Cases

3.3.2 Histogram Inaccuracies

In the ideal case, the join algorithm would know the exact distribution of the probe relation and be able to determine exactly the skew and the frequently occurring values. Without pre-sampling the inputs, this requires a pre-existing summary of the distribution as provided by histograms. Histograms are not perfect because they summarize the information, which results in lack of precision. Also, the histogram may be inaccurate as it may be constructed by only using a sample of the data or was constructed before some modifications occurred on the table.

Note that skew-aware partitioning, as implemented by *Histojoin*, can be used with a sampling approach as well as with pre-defined histograms. The advantage of using histograms is that there is no overhead during join processing as compared to sampling. The disadvantage is the accuracy of the distribution estimation may be lower. Histograms are valuable because they require no pre-processing for the join and are kept reasonably up-to-date for other purposes by the optimizer. Non-random sampling has been experimented with by examining the first few thousand tuples of the probe relation before processing the build relation. Although it is sometimes possible to determine very frequent values using this approach, in general, most relational operators produce a set of initial tuples that is far from a random sample. True random sampling incurs cost that is too high for the potential benefit.

There are two key histogram issues. First, the histogram may not be a precise summary of a base relation distribution due to issues in its construction and maintenance in the presence of updates. Second, if the join is performed on relations that are derived from other relational operators (selection, other joins), then a histogram on the base relation may poorly reflect the distribution of the derived relation. Without an approach to derive histograms through relational operators, we must decide on our confidence in the histogram when allocating memory in the operator.

This approach assigns a *confidence value* to the histogram. The confidence value reflects the confidence in the accuracy of the histogram in relation to the data it is designed to represent. Histograms derived after selections have lower confidence than those recently built on the base relation.

The confidence value is used to determine how many privileged partitions are used. With a high confidence value, privileged partition ranges are defined such that almost all of the memory is allocated to the privileged partitions, as we are reasonably certain that the best tuples in the histogram are actually the best tuples in the relation. For a low confidence value, only the absolute best values as determined by the histogram are used as the range partitions. The result is that the algorithm can control its benefit or penalty as compared to DHJ based on the confidence of the estimates. This improves the stability, robustness, and overall performance of the algorithm.

For example, consider M=1000 (1000 tuples can fit into memory). Let the join attribute value range be 1 to 2000. With a high confidence histogram, the algorithm would define privileged partition ranges to occupy all 1000 tuples of memory available. For instance, it may allocate 4 ranges 100-199, 300-599, 1000-1199, and 1500-1899 that would correspond to 1000 tuples in the build relation. With a low confidence histogram, the algorithm only allocates the very best ranges, which may result in only 2 ranges such as 100-199 and 1000-1199 (300 total tuples). The algorithm determines the ranges to allocate based on the frequency of occurrence and the confidence value. With the low confidence histogram, the range 100-199 must have been significantly more common than average. The number of privileged partitions is reduced with a low confidence histogram to reduce the penalty of error. For instance, the range of values 100-199 may turn out to be very infrequently occurring in the probe relation. Buffering build tuples in the range 100-199 then would produce fewer results than buffering random tuples.

There are multiple possibilities for determining how many tuples to put in the privileged partition based on the histogram confidence level. One approach is to select ranges whose frequencies are one or more standard deviations better than the mean frequency of all ranges. The amount that the ranges must be better than the mean is increased for lower confidence histograms. A high confidence histogram will fill up memory with histogram buckets that are above the mean. A low confidence histogram will only accept buckets that are multiple standard deviations better than the mean.

The approach chosen to measure the quality of the histogram depends on the database system and its optimizer. The two experimental implementations (see Chapter 4) use different approaches to selecting ranges based on the confidence level. The stand-alone Java implementation that only performs the joins and does not have an optimizer operates in two modes. Histograms on base relations with or without a selection operator are considered high confidence and all privileged ranges better than the average are selected. A low confidence histogram is when a base relation histogram is used to estimate the distribution of a relation produced by an intermediate join operator. In this case, only single premium values are used and no ranges. The PostgreSQL implementation exploits PostgreSQL's statistics that capture the most common values (MCVs) of an attribute. All MCVs are kept regardless of

the histogram confidence and the equi-depth histogram is not used to select ranges. This is an effective approach as the penalty for being incorrect with MCVs is minimal, the payoff is potentially very high, and there is a high probability that MCVs of a base relation remain MCVs in derived relations. More details are in Section 6.

There are two potential "costs" in using this approach. The first is a *lost opportunity cost* that occurs when due to low confidence in the histogram we do not select ranges with frequently occurring values as privileged partitions. In this case, the performance of the algorithm could have been improved had it been more aggressive on selecting privileged partition ranges. However, the performance would be no worse than dynamic hash join as any tuples that are not privileged get flushed randomly as in DHJ. The second cost, *inaccuracy cost*, is much more important. Inaccuracy cost occurs when a value range is selected as privileged and turns out to be less frequently occurring than average. For example, if the 100 build tuple values in the range 100-199 map to 2 tuples on average in the probe relation, and the average build tuple maps to 3 tuples on average, then skew-aware partitioning will have worse average performance than dynamic hash join. For low confidence histograms, it is better to be conservative in selecting privileged ranges, as there is a penalty for being too aggressive. By selecting no privileged ranges, *Histojoin* behaves exactly as DHJ.

Handling Selections

The discussion so far has considered joins where both inputs are base relations. It is common that a selection is performed before a join. A selection on the probe relation may change the distribution of join values and result in lower confidence in the histogram. The confidence can be changed based on the attribute correlation. If the selection attributes are highly correlated with the join attribute, then the histogram will most likely be very inaccurate. If there is low correlation, then the histogram is more usable and the uniform assumption can be applied. For example, the uniform assumption assumes that if a selection reduces the cardinality of the entire relation by 90%, then the cardinality of each histogram bucket is also reduced by 90%. If present, multi-dimensional histograms on both the selection and join attributes may be used to estimate the distribution after selection. SITs may be used as well. See Section 2.4 for more information about database statistics.

Selections on the build relation are less problematic. A selection on the build relation may affect the number of build tuples in a privileged partition range. For instance, if the algorithm determines that the range 100-199 is valuable, it expects 100 unique values in the build relation. However, a selection may cause the actual number of build tuples to be 50. This is another example of a lost opportunity cost because given this knowledge, the algorithm may have been able to select more privileged partitions (since memory is available) or select different ones because the value of the partition range may be lowered since not all of its build tuples participate in the join. Note that since we do not allocate a static amount of memory to privileged partitions, the extra memory for the 50 tuples is available for other partitions (most likely non-privileged hash partitions) to use. The algorithm will still outperform dynamic hash join if the build tuples actually in the privileged partition range join with more probe tuples than the average build tuple.

Multiple Join Plans

When a query consists of multiple joins, *Histojoin* can be used with each join as long as a histogram is available or can be estimated for the probe relations. *Histojoin* can be used for star joins, where multiple relations are joined to one common relation, which are very common in data warehouses.

For example, consider a star join of the tables *Part*, *Supplier*, and *LineItem* as shown in Figure 3.5a. With histograms on *LineItem.partkey* and *LineItem.suppkey* and no selection operations, *Histojoin* will have high confidence histograms for both joins. The bottom join of *LineItem* and *Supplier* will use the histogram on *LineItem.suppkey*. The second join will use the histogram on *LineItem.partkey* which will accurately reflect the distribution of *LineItem.partkey* in the intermediate join result *LineItem-Supplier* as the intermediate result was produced using a primary-to-foreign key join. In general, star joins with no selections and histograms on all join attributes of the fact table are accurately estimated and result in large performance improvements for skewed data.

(a) Part Supplier LineItem

(b) Customer Order LineItem

Figure 3.5: Example Multiple Join Plans

In contrast, consider a join of the tables *Customer*, *Orders*, and *LineItem* as shown in Figure 3.5b. The join of *LineItem* and *Orders* can exploit the histogram on *LineItem.orderkey*. However, the top join has no base histogram that can model the distribution of *custkey* in the intermediate relation *LineItem-Orders*. It is possible to estimate the histogram from one on *Orders.custkey*, but it would be a low confidence histogram. When selections are added with joins, the confidence of the histograms decreases, especially with selections on the probe relations.

3.3.3 Query Optimizer Modifications

There are minimal query optimizer modifications required to use *Histojoin*. *Histojoin* can be used as a drop-in replacement to hybrid or dynamic hash join, or the concepts used to modify an existing hash join implementation. If *Histojoin* is implemented separately from hybrid hash join, then when costing a potential join, *Histojoin* will return a high, not-applicable cost for joins where a histogram does not exist or cannot be estimated for the probe relation.

As a drop in replacement for hybrid hash join the cost for *Histojoin* when it cannot make use of histograms would be exactly equal to that of hybrid hash join. The cost of *Histojoin* will be the same formula as given in Section 3.1. In estimating the term g in the formula from the histogram we first calculate which histogram buckets will be in the privileged partitions. The histogram tells us how many probe tuples are related to each histogram bucket so using this we can estimate the number of results we will get from the in-memory build tuples. In practice this can be done in one step as the estimate of result tuples can be calculated while choosing privileged build tuple ranges.

Given the list of privileged partitions, g is estimated by summing up the $FREQ*ROW_R$ or alternatively $ROWCOUNT + EQ_ROWS$ (see Section 4.2) then dividing by S. That is, $g = \frac{\sum ROWCOUNT+EQ_ROWS}{|S|}$.

Using the example histogram given in Figure 3.2, without separating out the max values, the ranges in sorted order are 751-1000 (W=4.6), 101-200 (W=3.1), 1-100 (W=3.05), 201-350 (W=1.67), 351-500 (W=1.6), and 501-750 (W=1). In the example, the build relation R has 1000 tuples, and the probe relation S has 2505 tuples. With 350 tuples of memory ($f = 35\%$) the first two ranges 751-1000 and 101-200 would be selected as privileged and $E = 250 * 4.6 + 100 * 3.1 = 1460$. $g = E/|S| = 1460/2505 = 58\%$. Using the formulas in Section 3.1.1 we expect DHJ to perform 4556 I/Os during this join and *Histojoin* to perform 3405 I/Os (25% less).

Using *Histojoin* this way will allow a query optimizer to only use the algorithm when *Histojoin* indicates that it will have a performance benefit (by exploiting the skew it potentially sees). A major benefit is that no major changes to the optimizer are required. The only issue is the DBMS must make the histograms available to the *Histojoin* operator when costing and initializing.

Histojoin's performance and applicability are increased according to the database system support for statistics collection. For instance, *Histojoin* works best when provided with a list of the most frequent values and their frequencies. It is this list of values and their associated tuples that must remain memory-resident. Some statistics systems collect this data explicitly either separate from the histogram (PostgreSQL's most common values) or as part of the histogram (end-biased histograms). Note that histogram bucket ranges are a less accurate approximation to the most frequent value list.

The challenge of using *Histojoin* on derived operators (selections, joins, etc.) can also be mitigated by better statistics collection. For example SITs [2] and statistics collection on views allow the optimizer to have improved distribution estimates for relations of intermediate operators. Instead of base histograms and uniform assumptions, these approaches can provide *Histojoin* with more accurate data when deciding on privileged ranges/values. Any technique to improve the histogram accuracy will improve *Histojoin*'s performance. In summary, *Histojoin* will always produce a correct result that is robust and optimal according to the distribution estimate given. The more accurately the histogram reflects the actual data distribution, the better actual performance *Histojoin* will have.

4. Experimental Results

This chapter presents two separate experimental evaluations for *Histojoin*. The first evaluation is a stand-alone Java application performing the joins. The second evaluation is an implementation of the algorithm in PostgreSQL. The *Histojoin* algorithm was tested with the TPC-H data set. The TPC-H generator produced by Microsoft Research [3] was used to generate skewed TPC-H relations. Skewed TPC-H relations have their attribute values generated using a Zipf distribution, where the Zipf value (z) controls the degree of skew. The data sets tested were of scale 1 GB and 10 GB and labeled as skewed (z=1) and high skew (z=2).

4.1 Stand-Alone Evaluation

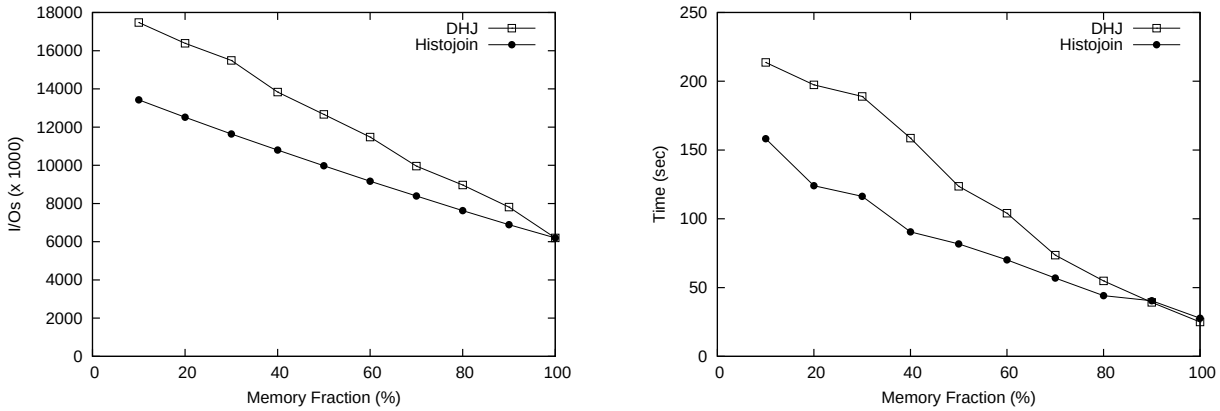
The dynamic version [8] of hybrid hash join [7] (DHJ) was compared to *Histojoin*. Both algorithms were implemented in Java and used the same data structures and hash algorithms. The only difference between the implementations is that *Histojoin* allocated its in-memory partitions using a histogram and DHJ flushed partitions to free memory without regard to the data distribution. DHJ typically flushes partitions in a deterministic ordering, but this implementation flushes randomly such that a more accurate average case is found. For instance, a deterministic ordering may always flush the exact worst partition for a join first. A random ordering will flush the worst partition first with probability $1/P$ (where P is the number of partitions).

The data files were loaded into Microsoft SQL Server 2005, and histograms were generated. The histograms were exported to disk, and the data files converted to binary form. Data files were loaded from one hard drive and a second hard drive was used for temporary files produced during partitioning. The experimental machine was an Athlon 64 3700+ (2.2Ghz) with 1.5GB RAM running Windows XP Pro and Java 1.6. All results are the average of 10 runs. These results use TPC-H scale 1 GB and demonstrate the applicability of the algorithm in various scenarios.

4.1.1 Primary-to-Foreign Key Joins

The joins tested were *LineItem-Part* on *partkey* and *LineItem-Supplier* on *suppkey* for z=1 and z=2. Memory fractions, f, were tested ranging from 10% to 100%.

Histojoin performs approximately 20% fewer I/O operations with the z=1 data set which results in it being about 20% faster overall. This is a major improvement for a standard



(a) Total I/Os (b) Total Time

Figure 4.1: Lineitem-Part Join (1GB, z=1)

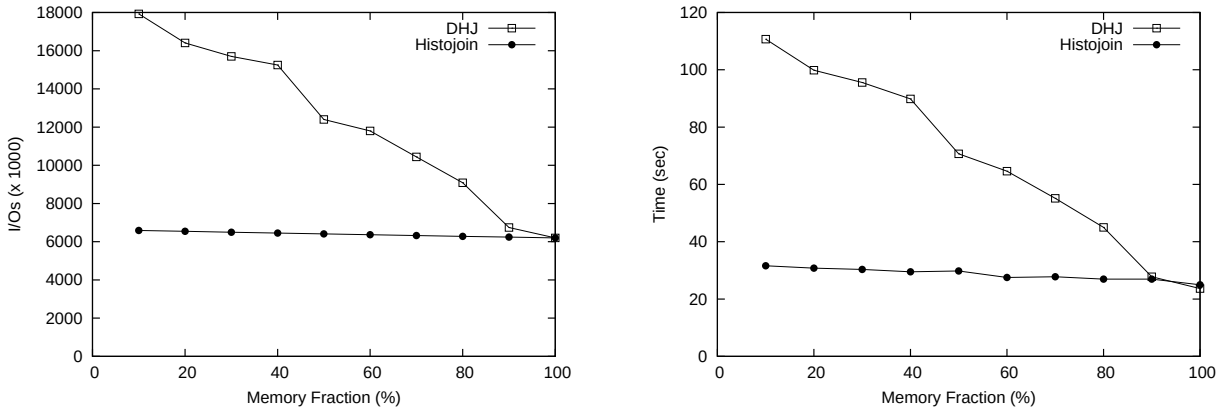
operation like hash join. An improvement occurs over all memory sizes until full memory is available for both joins.

For the z=2 data set, the performance difference is even more dramatic. In the 10% memory case *Histojoin* performs 60% fewer I/Os resulting in 60% faster execution. The results by total I/Os and by time are in Figures 4.2a and 4.2b respectively.

DHJ is slower because random partitioning causes the most important tuples to be distributed across all partitions. Regardless what partitions are flushed (or conversely what partition(s) remain in memory), hash join is guaranteed to not keep in memory all of the most beneficial tuples. Even worse, for highly skewed data sets, it is very likely that it will evict the absolute best partition. For instance, with 10% memory and 10 partitions, hash join has only a 10% probability of keeping the partition in memory with the key value that is most frequently occurring. The performance of dynamic hash join is *unpredictable* for skewed relations and is highly dependent on the partition flushing policy. For highly skewed relations and low memory percentages the likelihood of DHJ flushing the best values is *very* high.

For the z=1 data set and *LineItem-Supplier*, *Histojoin* performs about 10-20% fewer total I/Os and executes 10-20% faster. For the z=2 data set, *Histojoin* performs between 20-60% fewer total I/Os and executes 20-60% faster. A summary of the percentage total I/O savings of *Histojoin* versus dynamic hash join for all joins is in Figure 4.3.

Experiments with uniform data show that the performance of *Histojoin* and hash join is identical, as there are no tuples that occur more frequently than any other, and the performance is independent of the tuples buffered in memory. For totally uniform data, *Histojoin* selects no privileged partitions. For mostly uniform data, such as generated by the standard TPC-H generator, there are still some join attribute ranges that are marginally better and are used by *Histojoin* to improve performance slightly.



(a) Total I/Os (b) Total Time

Figure 4.2: Lineitem-Part Join (1GB, z=2)

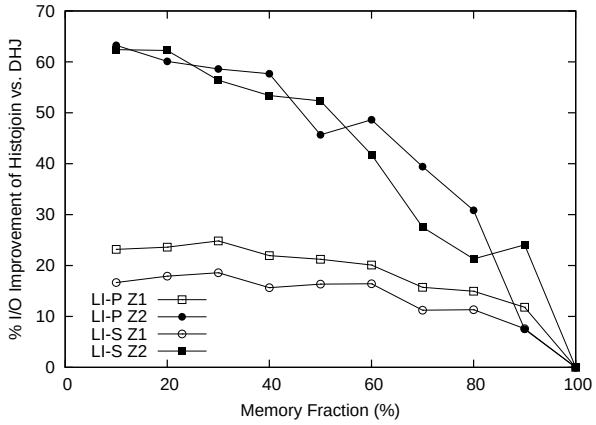


Figure 4.3: Percentage Improvement in Total I/Os of Histojoin vs. Hash Join (1GB)

4.1.2 Many-to-Many Joins

The many-to-many join tested combined a randomized version of the *Wisconsin* relation [9] with a randomized and Zipfian skewed (z=1) version of the Wisconsin relation on the *tenPercent* attribute. Both relations contained 1,000,000 tuples. The *tenPercent* attribute has a domain that is 10% the size of the relation. For 1,000,000 tuple relations, the domain of *tenPercent* is 100,000. Memory fractions, f, were tested ranging from 10% to 100%.

For this test, the build relation (randomized *Wisconsin*) contains 1,000,000 tuples, and the *tenPercent* attribute contains values in the range 0 to 99,999, each value being shared by 10 tuples. The probe relation has a domain of 0 to 99,999 as well with a Zipfian distribution of the values. In the generated Zipfian relation, the top 2 values occur in 127,380 of the 1,000,000 tuples (12.7%) and 31,266 of the 1,000,000 tuples (3.7%) respectively. Beyond the top 200 values, the average value occurs in approximately 5.7 of the 1,000,000 tuples. A

histogram on the probe relation attribute is misleading because it shows an integer domain of 100,000 tuples which underestimates the size of each privileged relation partition by a factor of 10.

This underestimation causes *Histojoin* to allocate too much memory for privileged partitions because it thinks the partitions contain far fewer tuples than they really do. However, these privileged partitions are dynamically flushed as required with no harm to the performance. The *Wisconsin* results by total I/Os (includes cost of reading each relation) for this join are in Figure 4.4. For all memory sizes, *Histojoin* performs approximately 10% fewer I/O operations than DHJ.

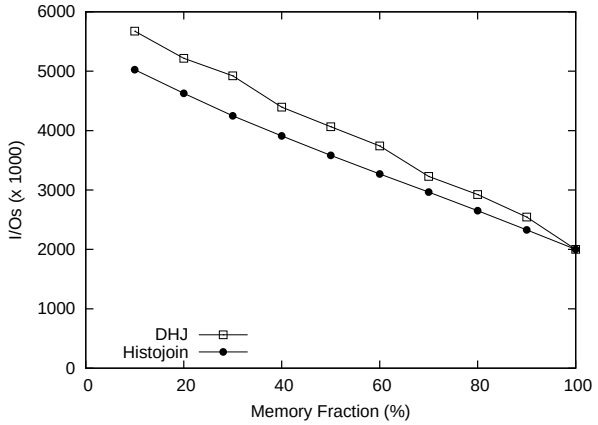


Figure 4.4: Total I/Os for Wisconsin Many-to-Many Join (1GB, z=1)

4.1.3 Histogram Inaccuracies

To demonstrate the effect of histogram inaccuracies on join performance, a modified TPC-H *LineItem* relation was created to show the worst case scenario for *Histojoin* and how the use of histogram confidence mitigates this scenario. The new *LineItem* relation contains only every 10000th *partkey* (1, 10000, 20000, ..., 200000) and each of these values occurs as often as the others. A histogram was created that indicates to *Histojoin* that these 10000th values never occur in *LineItem* so that in all cases except the 100% memory case *Histojoin* will not store any of the corresponding build tuples from the *Part* relation in memory but will instead fill memory with build tuples whose partkey values never occur within *LineItem*.

Histojoin was compared to DHJ using the join *LineItem-Part* on *partkey*. Memory fractions, f, were tested ranging from 10% to 100%. *Histojoin* executed the join under two confidence levels. In the high confidence level, it assumed the histogram was very accurate and fully allocated privileged partitions to memory. In Figure 4.5, this corresponds to the HJ Bad Histogram plot. *Histojoin* does considerably worse than DHJ by trusting a totally wrong histogram. In comparison, when executed under a low confidence level, *Histojoin* only selects premium values from the histogram (in the diagram as HJ Bad Premium Values).

Since the histogram is completely inaccurate, the premium values give no performance improvement, but also result in little cost compared to DHJ due to the minimal amount of memory occupied. If the histogram is only 10% correct (10% of the premium values are good), *Histojoin* in low confidence mode outperforms DHJ. In summary, executing *Histojoin* in low confidence mode has little risk and considerable reward if the histogram is even marginally accurate.

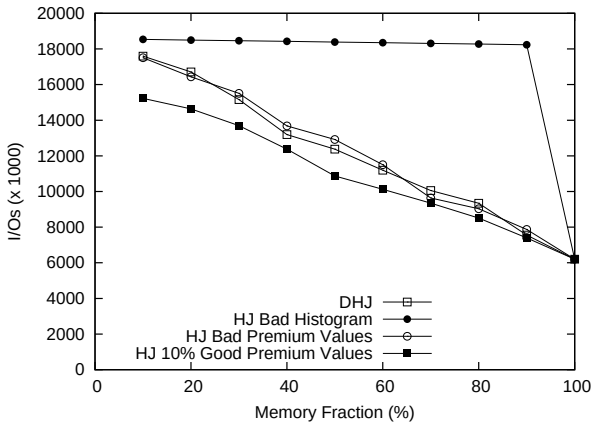


Figure 4.5: Total I/Os for Lineitem-Part Join with Histogram Inaccuracies (1GB)

4.1.4 Joins on String Keys

With string keys *Histojoin* is less accurate in predicting the size of build partition ranges for privileged partitions. To test joining on string keys, versions of the *LineItem* and *Supplier* relations were generated with the *suppkey* replaced by randomly generated strings. Once again memory fractions, f, were tested ranging from 10% to 100%. Much like the join of *LineItem-Part* on *partkey* using integer keys *Histojoin* performs around 20% fewer Total I/Os than DHJ for the z=1 dataset. The results are in Figure 4.6.

4.1.5 Multi-Way Joins

As described in Section 3.3.2, *Histojoin* can be used on multi-way star joins when a histogram exists on the join attributes of the fact relation. A star join of the tables *Part*, *Supplier*, and *LineItem* as shown in Figure 3.5a falls into this category.

If the memory for the entire query is split evenly between the two joins then for a memory percentage above 10% the first join of *LineItem-Supplier* would be done completely in memory as *Supplier* is quite small in comparison to *Part*. For this reason *Histojoin* was compared to DHJ using memory fractions ranging from 3% to 10%. The total I/Os for the entire join using a z=1 dataset are shown in Figure 4.7a. For all memory sizes *Histojoin* performs about 20% fewer I/Os than DHJ. For memory sizes above 10%, *Histojoin* is faster than DHJ but only one join requires disk I/Os. Results for the z=2 dataset are shown in

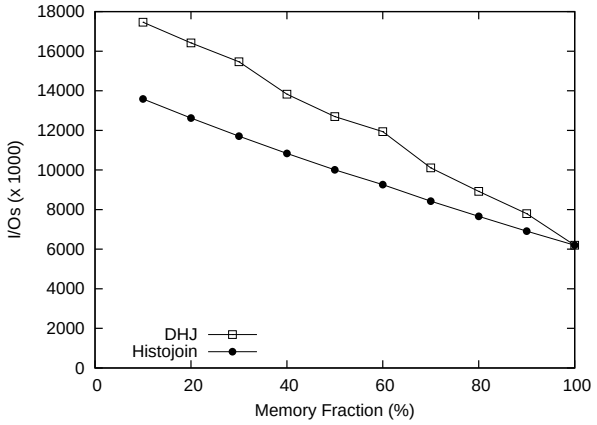
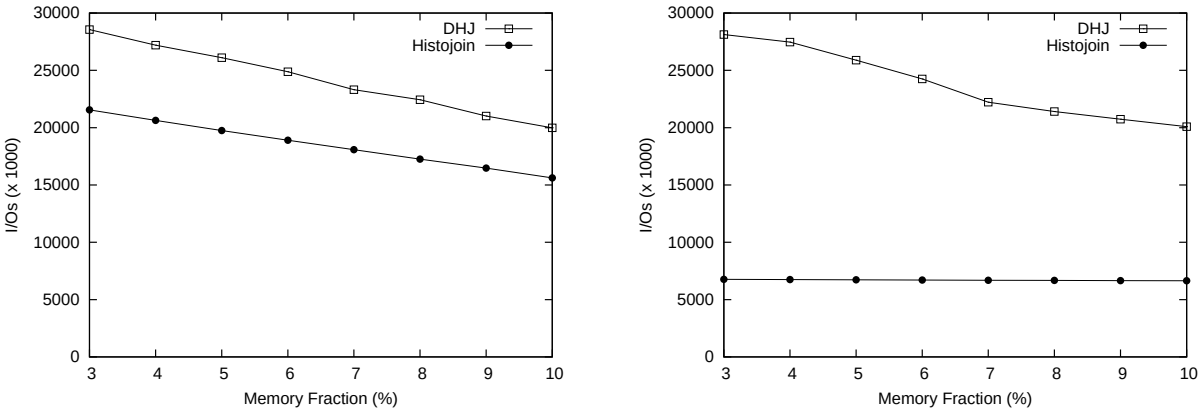


Figure 4.6: Total I/Os for Lineitem-Supplier Join on String key (1GB, z=1)

Figure 4.7b. Due to the high skew, *Histojoin* dramatically improves on the performance of DHJ.



(a) z=1 (b) z=2

Figure 4.7: Total I/Os for Lineitem-Supplier-Part Join (1GB)

4.2 PostgreSQL Implementation

Histojoin was implemented in PostgreSQL 8.4 to test its performance for large-scale data sets in a production system. PostgreSQL implements hybrid hash join (HHJ). Its HHJ implementation requires that the number of partitions be a power of two, and it always keeps the first partition in memory. Thus, experimental data was only collected for memory fractions: 3.1% (1/32), 6.2% (1/16), 12.5% (1/8), 25% (1/4), 50% (1/2), and 100%.

PostgreSQL collects statistics on its tables. Statistics on an attribute of a table include the most common values (MCVs) and an equi-depth histogram. The user is able to control

on a per table basis the number of MCVs. The user can also initiate statistics re-calculations. The query optimizer has access to the histograms and a list of most common values (MCVs) that are automatically generated for foreign key attributes.

Histojoin was added to the PostgreSQL HHJ implementation. Using environment flags that PostgreSQL uses to control which joins are available, *Histojoin* can be turned on and off from the standard HHJ implementation. Thus, the existing HHJ implementation was altered instead of having two hash join algorithms for the optimizer to choose between.

Histojoin requires the ability to use the existing statistics which were available in the planner. The code uses the join attributes of the probe relation to find statistics for that attribute. If no statistics were available, *Histojoin* would not be used. If the optimizer determines that the build relation will completely fit in memory then *Histojoin* is not used as it would have no positive effect and add unnecessary overhead. If statistics are available, *Histojoin* only uses the MCVs (not the histogram) as the MCVs are more precise. However, this means that the privileged partitions do not occupy very much of the memory available for build relation tuples. The MCVs were determined and allocated into an in-memory hash table when the join operator was initialized. The default number of MCVs is 10, which produces a small hash table (less than 1KB), however the database administrator can increase this number to as many as 10,000 MCVs. The hash table size is at least 4 times the number of MCVs (load factor is less than or equal to 25%) to make collisions unlikely.

During the partitioning of the build relation, a build tuple's join attributes are hashed according to the small MCV hash table to determine if its value is one of the MCVs. If it is, then the tuple is put into the hash table, otherwise it is processed using the regular hash function as usual. While partitioning the probe relation, a probe tuple's join attribute is hashed and a lookup performed in the MCV table. If there is a match, the tuple is joined immediately, otherwise it proceeds through the hash join as normal. In effect, the MCV lookup table is a small mini-hash table for the most frequent values.

The equi-depth histograms are not used as it is preferable to increase the number of MCVs rather than allocate ranges from the histogram. The experiments all use the default of 10 MCVs unless otherwise specified. Results are improved when MCVs are set to 100 or more.

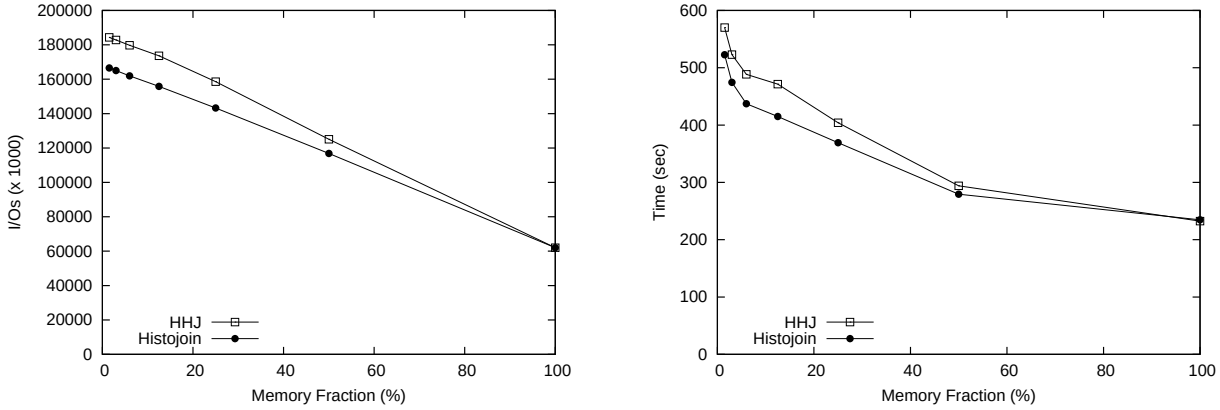
The experimental machine for the PostgreSQL implementation was an Intel Core 2 Quad Q6600 (2.4Ghz) with 8GB RAM running 64bit Debian Linux with a 2.6.25 kernel. These results use TPC-H scale 10 GB. Note that even for machines with large main memories, a join operator is allocated only a small fraction of the memory available as it must compete with other operators and other queries for system resources. The default join memory size for PostgreSQL is 1 MB. The experiments alter that memory size to produce the desired memory fraction based on the build table size.

There are a couple of differences from the Java experiments that should be noted. First, the execution times more accurately reflect the number of I/Os performed. This is due to increased stability and performance of PostgreSQL on Linux versus a Java implementation on Windows. The Java I/O counts are exact, but the execution times are more variable. There is less I/O performance improvement for the PostgreSQL implementation compared to

the Java implementation because the PostgreSQL implementation has very small privileged partitions (just the MCVs) where the Java implementation uses all available memory for privileged partitions by filling them with valuable histogram bucket ranges.

4.2.1 Primary-to-Foreign Key Joins

The *LineItem-Part* results by total I/Os (includes cost of reading each relation) and by time for the z=1 data set are in Figures 4.8a and 4.8b respectively. *Histojoin* is around 10% faster and performs 10% less I/Os than HHJ. With the z=2 dataset, *Histojoin* performs approximately 50% faster (Figures 4.9a and 4.9b). The percentage improvement of *Histojoin* is shown in Figure 4.10. Note that the sudden improvement of HHJ for the z=2 50% memory case is because HHJ manages to get the best tuples from the build partition in its in-memory partition by chance.



(a) Total I/Os (b) Total Time

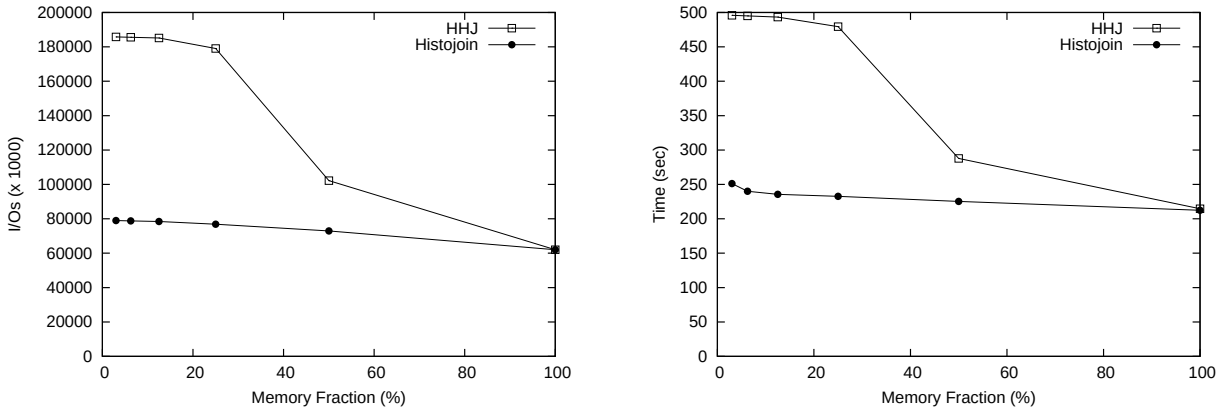
Figure 4.8: PostgreSQL Lineitem-Part Join (10GB, z=1)

4.2.2 Multi-Way Joins

When performing a star join of the tables *Part*, *Supplier*, and *LineItem* any memory size above 10% of the size of the *Part* table will run the smaller join of *LineItem* and *Supplier* completely in memory. This multi-way join was tested with memory fractions (sizes) of 0.78% (2770KB), 1.56% (5440KB), and 3.12% (10880KB). Figure 4.11a shows that for the z=1 dataset *Histojoin* performs around 6% fewer IOs than HHJ and for the z=2 dataset *Histojoin* performs around 40% fewer IOs than HHJ.

4.2.3 Effect of Number of MCVs

By increasing the number of MCVs from the default 10, the performance of *Histojoin* increases as *Histojoin* is able to capture more of the most valuable tuples. The join of *LineItem*



(a) Total I/Os (b) Total Time

Figure 4.9: PostgreSQL Lineitem-Part Join (10GB, z=2)

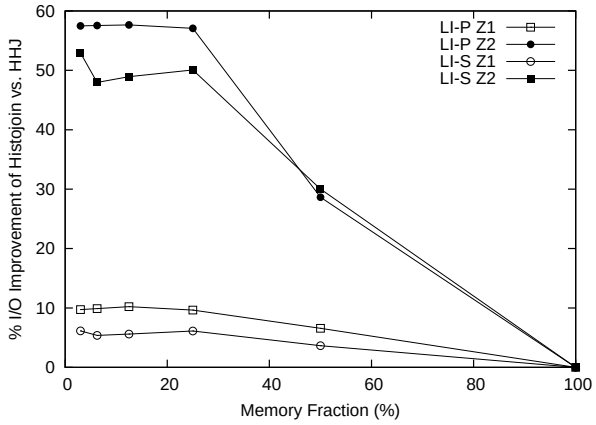
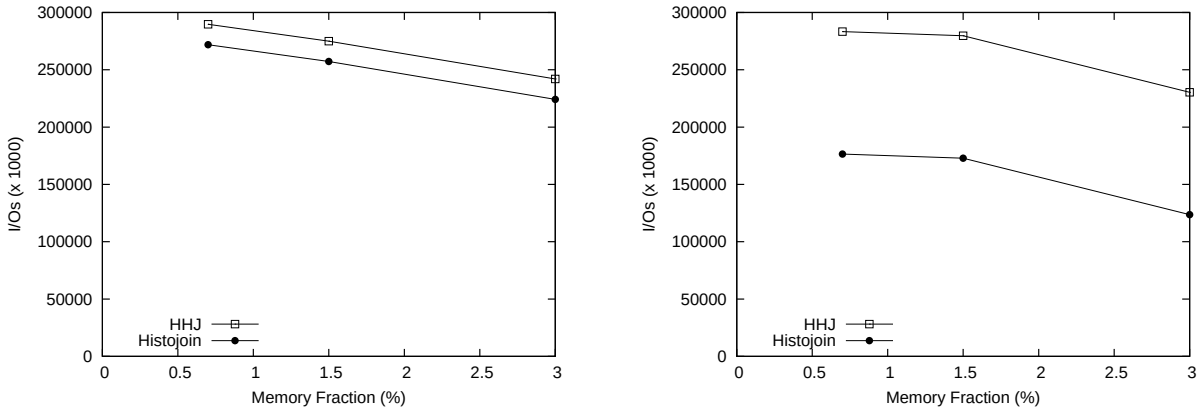


Figure 4.10: PostgreSQL Percentage Improvement in Total I/Os of Histojoin vs. Hash Join (10GB)

and *Part* was performed with a memory size of 6.2% and various amounts of MCVs. The results by total I/Os and by time are in Figures 4.12a and 4.12b respectively. The query was run with 10, 100, 300, 500, 700, and 1000 MCVs on *partkey*. *Histojoin*'s performance with the z=1 dataset can be increased by adding more MCV statistics as this dataset has many relatively good MCVs. As more MCVs are added the benefit per new MCV is much less.

4.3 Results Summary

For skewed data sets, *Histojoin* dramatically outperforms traditional hash joins by 10% to 60%. This is significant because hash join is a very common operator used for processing the largest queries. As the amount of skew increases, the relative performance improvement



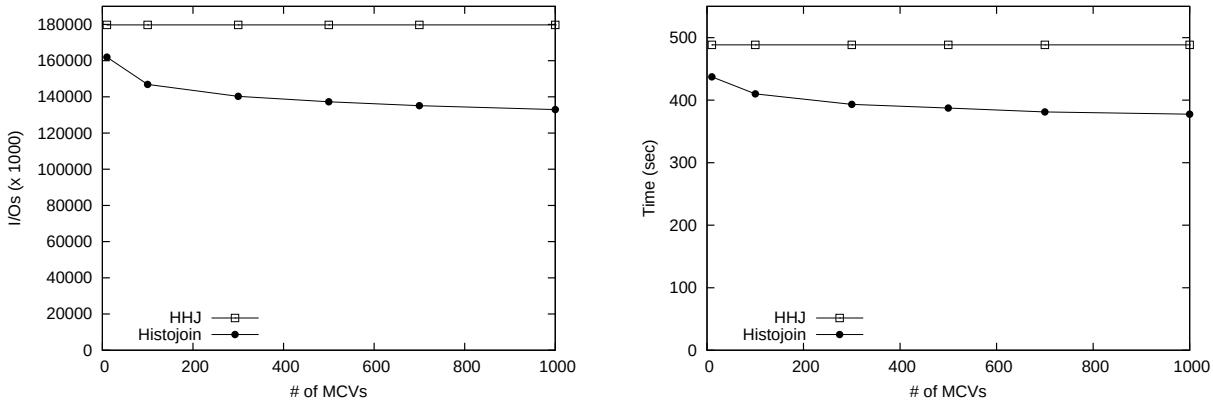
(a) z=1 (b) z=2

Figure 4.11: Total I/Os for PostgreSQL Lineitem-Supplier-Part Join (10GB)

of *Histojoin* increases.

Histojoin introduces no performance penalty compared to hash join for uniform data sets or data sets where the skew is undetected due to selection conditions or stale histograms. *Histojoin*'s performance improvement depends on the amount of skew detected (as given by the formula in Section 3.1). *Histojoin* has better performance with a more accurate estimate of the distribution of the probe relation. When the confidence in the histogram approximation of the distribution is low, *Histojoin* allocates fewer privileged partitions which must be significantly better than the average. Thus, *Histojoin* will exploit whatever skew is detectable and fall back to dynamic hash join behavior otherwise. Even with low accuracy histograms, *Histojoin* will improve join performance over hash join for skewed data sets.

The implementation of *Histojoin* in PostgreSQL uses only premium values determined from pre-generated MCV lists to determine its privileged partitions. *Histojoin* is minimally affected by bad estimates as the MCV lists are small and represent only a minimal memory overhead. In the experiments this implementation shows a large improvement over the standard hybrid hash join operator used for all large unsorted joins in PostgreSQL while adding no noticeable overhead when skew cannot be exploited. *Histojoin* is especially valuable for smaller memory fractions as its relative benefit over HHJ is higher.



(a) Total IOs

(b) Total Time

Figure 4.12: PostgreSQL Lineitem-Part Join With Various Amounts of MCVs (10GB, z=1)

5. Discussion and Conclusion

This thesis began by describing the general function of a database system (Section 2.1) and how join algorithms fit into that system (Section 2.2). Hash join algorithms are used in many of the large, important, and costly queries that a database performs especially in the large database systems used by governments and commercial organizations.

The original in-memory hash join algorithm (Section 2.2.1) increased the speed of joining large unordered data sets significantly over previous methods by reducing the number of tuple comparisons that needed to be made but was limited to data sets that could fit in memory. The Grace Hash Join (Section 2.2.3) handled very large relations by partitioning them into smaller memory sized chunks that could be joined one at a time but required writing these partitions to disk and then re-reading them during a cleanup phase which incurred extra I/O operations per tuple. Grace Hash Join also failed to make use of all available memory during the initial partitioning phase.

Hybrid Hash Join (Section 2.2.4) improved on the Grace Hash Join by keeping one of the partitions in memory after partitioning the build relation and producing results while partitioning the probe relation. Recognizing that partitions sizes could vary due to partition skew in the build relation and that memory conditions could change the Dynamic Hash Join algorithm was created (Section 2.2.5). Dynamic Hash Join is similar to Hybrid Hash Join except it initially keeps all partitions in memory and then dynamically flushes them to disk whenever memory must be freed. Since many partitions can remain in memory the targetted size of a partition does not necessarily have to be equivalent to the size of memory and Dynamic Hash Join can adapt its memory usage much easier if the partitions are targetted at being smaller than the amount of available memory.

In a hash join the partitions that remain memory-resident after the build relation is partitioned are chosen before any probe relation tuples have been read. Consequently there is no guarantee that the memory-resident build tuples will join with many of the probe tuples. Due to skew (Section 2.3) the memory-resident build tuples may join with far more probe tuples than the frozen partitions or far fewer. Statistics such as histograms already exist in most commercial database systems and when present can give a good indication of the number of probe tuples that will join with each build tuple (Section 2.4).

The effect of skew on hash joins is well known and modern hash join algorithms try to minimize the negative effect that skew has on their performance. Previously skew has not been exploited by a hash join to improve algorithm performance. The theoretical benefits of exploiting skew (Section 3.1.1) are significant, however the practical benefits rely on the statistics accurately representing skew and the algorithms ability to compensate when the statistics are incorrect.

The generic *Histojoin* algorithm described in Section 3.2 does not rely on any particular database system or type of statistics although the maxdiff histograms used by Microsoft SQL Server are used in the examples. The ability of a *Histojoin* implementation to exploit skew is directly proportional to how good the statistics are. Two actual implementations of *Histojoin* have been created, one as a stand-alone Java algorithm and the other as an addition to the Hybrid Hash Join algorithm in the PostgreSQL open source database system. The statistics available to the Java implementation are more comprehensive than those available in PostgreSQL and consequently that implementation is better able to detect and exploit skew in the underlying relations being joined. If *Histojoin* was implemented in a system that used SITs (Statistics on Intermediate Tables) [2] or other advanced statistics the amount of instances when skew could be exploited would increase. It is a distinct benefit of *Histojoin* that it can adapt to the database system it is implemented in.

Empirical experimental results (Section 4) have shown that *Histojoin* can have significant practical benefits over conventional hash join algorithms. Demonstrating an algorithm in only the ideal cases, however, does not prove that it is generally beneficial. *Histojoin* is able to adapt to cases where exploiting skew is not possible or necessary, such as when performing a one-to-one join or when the entire join can be performed in memory, without noticeable runtime overhead. *Histojoin* has also been designed to handle cases where statistics are inaccurate and attempting to exploit skew would reduce performance (although this ability is implementation specific (Sections 3.3.2 and 4.1.3)).

Through discussion and collaboration with core PostgreSQL developers the PostgreSQL implementation of *Histojoin* has been modified to handle many potentially negative situations gracefully while still providing a significant benefit when skew can be exploited.

This research is important as it makes improvements to a core database algorithm that is used in all major database systems by corporations and individuals alike. The algorithm is generic in that it is not limited to one database system and does not require significant changes to the underlying system to gain its benefits.

Since the benefits of *Histojoin* depend heavily on the statistics available it would be useful to examine more types of statistics used in current database systems (especially SITs) and how they can be exploited by *Histojoin*. As defined, *Histojoin* is a join algorithm for centralized database systems but the concept should adapt to distributed and grid databases.

This thesis expands on the presentation in [4, 5]. The PostgreSQL implementation is currently being evaluated for inclusion in the main production branch of PostgreSQL where it will have an impact on many real world databases and millions of users. In conclusion, by exploiting statistics already existing in database systems hash join algorithms can gain significantly increased performance.

Bibliography

[1] TPC-H Benchmark. Technical report, Transaction Processing Performance Council, Available at: *http://www.tpc.org/tpch/*.

[2] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *ACM SIGMOD*, pages 263–274, 2002.

[3] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew. Technical report, Microsoft Research, Available at: *ftp.research.microsoft.com/users/viveknar/tpcdskew*, 1999.

[4] B. Cutt and R. Lawrence. Using Intrinsic Data Skew to Improve Hash Join Performance. *Information Systems, To appear*, 2008.

[5] B. Cutt and R. Lawrence. Histojoin: A Hash Join that Exploits Skew. In *IEEE Canadian Conference of Electrical and Computer Engineering*, May 2008.

[6] C. Date. *The SQL standard*. Addison Wesley, Reading, US, third edition, 1994.

[7] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD*, pages 1–8, 1984.

[8] D. DeWitt and J. Naughton. Dynamic Memory Hybrid Hash Join. Technical report, University of Wisconsin, 1995.

[9] D. J. DeWitt. *The Wisconsin Benchmark: Past, Present, and Future*. 1993.

[10] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson Education, Inc., 2002.

[12] G. Graefe. Five Performance Enhancements for Hybrid Hash Join. Technical Report CU-CS-606-92, University of Colorado at Boulder, 1992.

[13] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[14] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *ACM SIGMOD*, pages 233–244. ACM, 1995.

[15] M. Kitsuregawa, M. Nakayama, and M. Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *VLDB*, pages 257–266, 1989.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1), 1983.

[17] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Series in computer-science and information processing. Addison-Wesley, 1973.

[18] R. Lawrence. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In *VLDB 2005*, pages 841–842, 2005.

[19] W. Li, D. Gao, and R. T. Snodgrass. Skew handling techniques in sort-merge join. In *SIGMOD*, pages 169–180, 2002.

[20] M. Muralikrishna and D. J. DeWitt. Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. In H. Boral and P.-Å. Larson, editors, *ACM SIGMOD*, pages 28–36. ACM Press, 1988.

[21] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *VLDB*, pages 468–478, 1988.

[22] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 294–305, New York, NY, USA, 1996. ACM.

[23] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, pages 537–548, 1991.