



Hash Joins
Implementation and Tuning
Release 7.3

Prabhaker "GP" Gongloor
Sameer Patkar

Center of Expertise
Worldwide Customer Support
Oracle Corporation

March 1997

TABLE OF CONTENTS

INTRODUCTION	3
Audience.....	3
Document Overview	3
Concepts	3
Conventions Used in this Paper	4
NESTED LOOPS JOIN (<i>NLJ</i>)	4
SORT MERGE JOIN (<i>SMJ</i>).....	5
PARALLELISM IN JOIN EXECUTION	6
HASH JOINS OVERVIEW	7
Parallel Hash Joins.....	8
Anti-Joins and Outer Joins.....	8
HASH JOIN PRINCIPLES	8
HASH JOINS ALGORITHM.....	12
COSTING OF HASH JOINS	13
In-Memory Hash Join	13
On-Disk Hash Join.....	13
HASH JOINS PERFORMANCE TUNING	14
Minimum Setting for HASH_AREA_SIZE.....	14
HASH_AREA_SIZE and Tablespace I/O	16
HASH_MULTIBLOCK_IO_COUNT.....	17
Size of Join Inputs.....	17
Reliability of Predicting Relative Sizes of Two Join Inputs.....	17
Exploiting Interesting Orderings.....	18
Number of Distinct Values	18
Skew in Build and Probe Inputs.....	18
Other Performance Considerations	18
TROUBLESHOOTING	19
APPENDICES	20
Appendix A – Citations	20
Appendix B – Enabling Hash Joins.....	20

Copyright © Oracle Corporation 1997 All rights reserved. Printed in the U.S.A.

ATTENTION READERS INCLUDING ORACLE EMPLOYEES: NO PART OF THIS DOCUMENT MAY BE REPRODUCED, IN ANY FORM, WITHOUT THE PERMISSION OF THE AUTHORS.

Authors: Prabhaker “GP” Gongloor, Sameer Patkar

Oracle is a registered trademark of Oracle Corporation. Oracle7 is a trademark of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

INTRODUCTION

Audience

You are an Oracle professional interested in the underlying internal mechanisms used in sorts and high performance query processing. This report is written with the assumption that you have a solid understanding of Release 7.1 concepts. The following reports are recommended preparatory reading: *Tuning Large Sorts* and *Asynchronous Reads* in *Technical Reports, Compendium Volume I, 1996*, published by the Center of Expertise.

Document Overview

Extensive academic research indicates that a hash join algorithm is often the best choice for many join queries. In our preliminary research, we observed hash joins performing two to three times faster than sort merge joins in an idealized test environment.

In our preliminary research we tested hash joins using a “small” 10 MB table and a “large” table, which was at least ten times greater than the small table. Our research is ongoing and may cover the following: table size thresholds when hash joins perform better than sort merge joins, scalability of hash joins compared to sort merge joins or indexed sort merge joins, anti-joins, and so on. Research findings will be published at a later date.

We caution readers about applying results observed in a test environment to their own environment. Different configurations with varying workloads will have different behaviors and performance profiles. Nonetheless, we are confident that appropriate use of information in this paper will help users significantly improve sort runtimes. Furthermore, we encourage readers to use their own development environments to test hash joins.

This document is organized to facilitate understanding hash joins and is meant to be read from start to finish. After this introductory section, we briefly describe existing join methods as preparation for understanding the newer hash joins method. The hash join part of the report start with an overview, followed by several more detailed descriptive sections. To close out the report, we provide tips on performance tuning and troubleshooting.

Concepts

A *join method* defines the algorithm used to perform a join. For Release 7.3 and earlier, the *nested loop join* and *sort-merge join* methods are available. These join methods are described in this paper as preparation for understanding the *hash joins* method introduced in Release 7.3, which is a significant enhancement in query processing capabilities. We also describe implementation, usage best practices, performance characteristics, tuning, and, guidelines for problem diagnosis.

A *join* returns rows created using columns from more than one table. A query with multiple tables in the FROM clause will perform one or more join operations. The Oracle server only performs one join operation at a time. A join operation reads a *left* and *right* row source, known as a *pair*, and combines row pairs based on a *join condition*. If more than two tables must be joined, then *pairing* occurs successively until all tables are joined.

Normally, a *join condition* is specified in the WHERE clause. (Absence of a WHERE clause will produce a Cartesian product of all rows from all tables joined.) Joins are classified into different topologies depending on the join predicates in the query; for example, star joins, chain, cycle, Cartesian product, and so on.

Note that the cost based optimizer must be enabled to use the hash joins method. The *Cost Based Optimizer (CBO)* chooses a query execution plan by determining which plan is most efficient. Costs are determined based on an estimation of the resources consumed in terms of disk I/O, CPU, and network I/O. Each object accessed in the plan is costed using a model that is based on access methods, statistics, and initialization parameters. Object statistics include object size, cardinality, clustering factor of indexes, and so on. Initialization parameters include sort area size, hash area size, multi-block read factor, and so on. The optimizer also factors in any hints supplied by the user. When costing joins, the join cardinality and join method costs are used for each join permutation considered.

Conventions Used in this Paper

In this paper, we use a convention as shown in the following example:

```
SELECT S.a, B.a FROM S,B WHERE S.a = B.a
```

Note the use of the letter *a* to denote the join column, and, letters *S* and *B* to denote the join tables.

Let the size of the tables be $r_s = \text{ROWS}(S)$ and $r_b = \text{ROWS}(B)$, such that, $r_s \leq r_b$.

If valid statistics were collected for the join tables using ANALYZE commands, then the cost based optimizer will correctly choose the join order to be (*S,B*), and **not** (*B,S*). The bigger table, *B*, is known as the *inner table*. The smaller table, *S*, is known as the *outer table*. *S* is also known as the *driving table*, the reason for which will become clear later in the paper.

Initialization parameters are denoted by small upper case letters; for example, DB_BLOCK_SIZE.

NESTED LOOPS JOIN (NLJ)

The nested loops join (*NLJ*) algorithm matches column *a* for every row in table *S* to column *a* for every row in table *B*. This requires $(r_s \times r_b)$ key comparisons with the following cost:

$$\text{Cost}(\text{NLJ}) \propto \text{Read}(S) + [r_s \times \text{Read}(B)]$$

The cost of a nested loops join can be prohibitive for large tables.

The number of comparisons can be significantly reduced when a B*tree index exists on the bigger inner table. This variation is known as an *indexed nested loops join*. An indexed nested loops join can be used for any join condition – equality, less than, greater than, etc. – including no join condition (resulting in a Cartesian product).

The performance of an indexed nested loops join is reasonably good for resultant sets of low cardinality.

An indexed-NLJ performs well when columns are retrieved by an index look-up without accessing the big table. However, if the B*tree index is several levels deep, or possibly fragmented, then the indexed-NLJ is usually not efficient due to the need to access large number of intermediate nodes in the B*tree.

In Oracle, the cost of accessing intermediate nodes of the index may be somewhat alleviated because of the buffer cache. If the buffer cache is maintaining recent versions of index pages, this reduces the I/O on intermediate nodes. Another issue worth consideration is the space required for the index. The space needed by an index may be very large, and sometimes is comparable to the size of the table.

Finally, the indexed nested loops join method may not be possible because only the leading columns in an index can be referenced in a query.

SORT MERGE JOIN (*SMJ*)

The sort-merge join (*SMJ*) algorithm must read columns of *S* and *B* from disk, sort the join keys, and perform a join by reading two sorted sets.

An in-memory sort occurs when the data to be sorted from both tables fits in the sort area. Obviously, this is desirable, but will not happen for large tables.

A *disk sort* occurs when a sort operation must use disk storage. This is the case when all the data to be sorted, known as the *row source*, will not fit in the sort area. A disk sort has two phases. During the first phase, data is read into the sort area, sorted, and then written out as a *sort run* to a sort segment. The sort runs phase continues until the row source is consumed. During the second phase, the *merge phase*, the sort runs are read back into the sort area and merged. A merge phase may require several *merge passes* to complete the entire sort operation. (A special case of sort merge join is a cartesian product or when join tables are not connected. For Release 7.2 and later, Oracle does not perform a sort on the two inputs, but merges and buffers the output, so that no re-computation is required for further processing within the query.)

After both the inputs are sorted, the sort runs need to be merged using optimal merge width to yield the final output. The cost of performing a sort merge join can be described as follows:

$$\begin{aligned} \text{Cost}(\text{SMJ}) \propto & \text{Read}(S) + \text{Write}(\text{SortRuns}(S)) \\ & + \text{Read}(B) + \text{Write}(\text{SortRuns}(B)) \\ & + \text{Merge}(S, B) \\ & + \text{CPUSortCost}(S + B) \end{aligned}$$

The efficiency of *Merge(S,B)* depends on memory allocated – `SORT_AREA_RETAINED_SIZE` in a non-MTS environment – involves reading-writing sort runs from-to temporary segments, and, the cost of comparing the records. The sort merge join method does not require an index on the joined columns. Thus, the dominant costs in the sort merge join are sorting, reading the tables, and the I/O reading-writing sort runs from-to temporary tablespace. The sort merge join can be used for any join condition – equality, less than, greater than, etc. – including no join condition (resulting in a Cartesian product).

The sort merge join performs reasonably well for large data sets and the performance is largely governed by size of the sort required in relation to available memory, `SORT_AREA_SIZE`. Sort performance is non-linear with respect to sort set sizes ^[7].

PARALLELISM IN JOIN EXECUTION

Oracle introduced the scaleable parallel query architecture in Release 7.1. This architecture supports ^[1] the execution of *parallel nested loops join* and *parallel sort merge join* methods. Parallelism is achieved by dividing the workload between a paired set of query slave processes executing a *parallel query execution plan*.

A *parallel query execution plan* is represented by a tree of operators called the Data Flow Operator tree (*DFO*). Each DFO node in this tree is a SQL operation and is assigned to a set of query slaves. The interaction between the slaves is known as intra-operator parallelism. Operations of these nodes are organized as a pipeline to establish a producer-consumer relationship. The producer-consumer interaction between SQL operations is known as inter-operator parallelism. The pipeline is an inter-process communication channel where rows are transmitted between slave sets. This mechanism supports partitioning or re-partitioning of rows based on hash, round-robin, broadcast or key ranges. There are only two adjacent DFO operators pipe-lined at any one time. Slave sets move from a certain node to its grandparent after completion and continue traversing the DFO tree until all parallel operations are accomplished. SQL operations parallelized in Release 7.3 include full table scan, nested loop joins, sort merge joins, hash joins, ORDER BY, GROUP BY, DISTINCT, UNION, and UNION ALL.

A *parallel nested loops join* is achieved by dividing the join task across slaves of a single slave set. Because a nested loops join has compatible left and right input sources on the adjacent nodes of the DFO tree, the two operators can be combined into single compound DFO, also known as a *table view*, which is represented by a single SQL statement with nested SQL in the FROM clause. A parallel nested loops join has two variations in implementation. The first implementation partitions the left input source (table scan) and broadcasts the right input. The right input is usually a small table or an index look up. The second implementation broadcasts the left input source and partitions the right input. The right is a large table that can be scanned in parallel and the left input is a relatively small table.

A *parallel sort merge join* requires both input sources (left , right) to be consumed before producing the output and hence the left and right input sources are incompatible. The first set of slave processes (table readers) scan the tables involved in the join and the second set of slave processes perform the join based on a hash function of the join columns. Thus, the second slave set is required to sort merge join only the data set that is hashed to it.

The parallel query architecture for sort merge joins and nested loops joins works reasonably well for large data sets, but are still expensive when large data sets must be processed.

HASH JOINS OVERVIEW

Release 7.3 supports the hash join method for equi-joins^[2]. Appropriately tuned hash joins should perform better than sort merge joins and indexed nested loop joins when indexes are several levels deep. In the worst case, hash joins should perform as good as sort merge joins, or, indexed nested loop joins when indexes are several levels deep. A hash join does not require an index to exist on the driving table, *S*.

The basic premise of any hash join algorithm is to build an in-memory hash table from the smaller of input row sources, the *build* input, and use the larger input to *probe* the hash table.

Hash join algorithms work well for simple hash joins when the available hash area memory is large enough to hold the build input. It is not necessary to fit the probe input in memory. Typically, the cost optimizer organizes the join order keeping the smaller of the two inputs on the left. If the available memory is insufficient to hold the hash table for the entire build input, then hash table overflow occurs. To deal with hash table overflow for build inputs larger than hash memory^[3,4], the build and the probe input are both split into disjoint partitions on disk in the initial pass over both the join inputs (called the *partitioning* phase). The row data is partitioned using a hash function on the join keys. Partitioning effectively divides the problem of a large input into multiple smaller inputs that can be independently processed. Partition pairs of the build and probe inputs with the same key values are then joined (called the *join* phase). This algorithm, known as the *grace join*, dramatically reduces the search space and key comparisons required for doing the join. A number of variations and optimizations of the grace join algorithm exist in the literature^[3,4] and a few relevant ones are mentioned below.

A limitation of the hash join algorithm is that it is based on the assumptions that the distribution of join values in the tables is not skewed such that each partition receives approximately the same number of rows. This is generally not true and partition skew is a reality that needs to be dealt with.

The grace join algorithm has to read and write back all build and probe inputs to temporary space on disk for the partitioning phase if hash table overflow occurs.

The *hybrid hash join* is based on the grace join algorithm, but uses memory more efficiently and is implemented in Oracle 7.3. Instead of writing all the build and probe partitions to disk in the partitioning phase, rows belonging to build input are used to build a memory resident hash table as they are read into the memory. Then, the probe input is read to probe the hash table. Thus, the hybrid hash join algorithm tries to minimize disk I/O by joining the initial build and probe partitions directly in memory without writing and re-reading them from disk.

However, the problem of partitioning the inputs is not trivial. It is difficult to have a partitioning scheme which will split any data distribution into equal partitions without any skew. Thus, different techniques like *bit-vector filtering*, *role reversal*, and *histograms* are applied to the *hybrid hash join* to minimize the effects of partition skew. These topics are described in subsequent sections.

If after partitioning, the smaller of the two inputs is larger than the size of the memory available to build the hash table, the hash table overflow is dealt with by performing a *nested-loops hash join*. The hash table is built with part of the build input partition and all of probe input is read and joined. Then, the remainder of the build input is iteratively retrieved, hash table built and joined with all the probe partitions until all of the build input is consumed. Again, this is described in subsequent sections.

Parallel Hash Joins

Parallel Hash Join is implemented like the Parallel Sort-Merge Join. Parallel Sort Merge Join requires both input sources to be consumed before producing the output and hence the input sources are incompatible. Parallel Hash Joins is currently restricted like the Parallel Sort Merge Join to consume both the left and right inputs before producing any output. The first set of slave processes (table readers or producers) in the Parallel Hash Join scan the left input and use key-value based hash partitioning to pipe the data to the appropriate second slave set processes which perform the join (consumers). The second slave set builds the hash table in memory in parallel with the scanners. After the left input is consumed and the hash table built, the producers now scan the right input and pipe the data to the appropriate consumer based on key-values. The consumer slaves now use the data from the right input to probe the hash table previously built to perform an in-memory join and return matched rows to the Query Coordinator.

Anti-Joins and Outer Joins

Outer joins and anti-joins can benefit from the new hash join implementation. An example of an anti-join query is as follows:

```
SELECT * FROM S WHERE S.a NOT IN (SELECT B.a FROM B WHERE B.b = value and ...);
```

Before Release 7.3, the above query would use a nested loops join. As discussed earlier in this paper, the performance of a nested loops join can be bad for large datasets when compared to other join algorithms. Oracle Release 7.3 supports hash joins and sort merge joins for the anti-join queries. The initialization parameter is ALWAYS_ANTI_JOIN and the available hints are MERGE_AJ or HASH_AJ.

HASH JOIN PRINCIPLES

In this section, we use a limited example to introduce the principle workings of the Oracle hash join algorithm, including partitioning, fan-out, bit-vector filtering, dynamic role reversal, and histograms driven hash joins. Loosely defined concepts are described in more detail in subsequent sections.

Consider the following two datasets (or tables) involved in a hash join:

$$S = \{ 1, 1, 1, 3, 3, 4, 4, 4, 4, 5, 8, 8, 8, 8, 10 \}$$

$$B = \{ 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 8, 9, 9, 9, 10, 10, 11 \}$$

The first step in a hash join is to determine whether the smaller table, the *build input*, can fit into the hash area memory of size HASH_AREA_SIZE.

If the build input fits in the available hash area memory, then an in-memory *hash table* is built by reading the build input. This is the simplest case of a hash join. Following the hash table build, the big table is scanned and joined with the left.

If the build input cannot fit in the available hash area memory, then the build input must be *partitioned*. The number of partitions is known as the *fan-out*, which is determined from HASH_AREA_SIZE and *cluster size*. Cluster size is a function of HASH_MULTIBLOCK_IO_COUNT. Cluster size is the number of contiguous blocks of a partition that must be filled before an I/O is scheduled to write out to the temporary tablespace on disk. During the partitioning phase, each partition is allocated a cluster worth of blocks from the hash area memory for buffering the partition data.

High fan-out can produce a large number of small partitions, resulting in inefficient I/O. At the other extreme, small fan-out can produce a small number of large partitions, which will not fit in hash memory. Finding the optimum fan-out and partition size is the key to optimum performance. Writing partitions during the partitioning phase and reading partitions during the join phase constitute a significant cost for the whole hash join operation. There is an optimal cluster size that balances efficient I/O against large fan-out within the constraints of available memory.

A hash function on the join column uniquely separates the rows from tables *S* and *B* into disjoint buckets, or *partitions*. Oracle uses an internal hash function that tries to minimize data skew among different buckets. For simplicity in this example, we use *modulo* as the hash function:

```
MOD( join_column_value, 10)
```

Using the above hash function produces ten disjoint partitions for the two join tables as shown in Figure 1.

For the join, the keys must be compared which fall in the partitions S_i and B_j where $i = j$.

PARTITION		S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
	VALUES	0,0,10, 10	1,1,1, 1,11	2,2,2, 2,2,2	3	NULL	NULL	NULL	NULL	8	9,9,9
R0	10	√									
R1	1,1,1		√								
R2	NULL										
R3	3,3				√						
R4	4,4,4,4										
R5	5										
R6	NULL										
R7	NULL										
R8	8,8,8,8									√	
R9	NULL										

Figure 1: Reduced Search Space through partitioning the build and probe inputs in Hash Joins

Furthermore, if either of the partitions S_i and B_j is NULL, then the other corresponding partition can be ignored. As shown, in Figure 1 below, it is only necessary to join those buckets marked “√”. Hence, the number of key comparisons is minimized through value-based partitioning. In contrast, nested loop joins and sort-merge joins must compare all tuples from both tables, even tuples that are not in the final output. Obviously, this can be very expensive for some queries.

A *bitmap vector* of the unique column values of the build input is created as the build input is read into hash area memory for partitioning. The bitmap memory comes from the hash area and can take up to five percent of HASH_AREA_SIZE. In this example, the following bitmap vector will be built:

```
{ 1, 3, 4, 4, 5, 8, 10 }
```

The bitmap vector is used during the partitioning phase of larger input, B , to determine whether the row read is needed for the join, and, if not, it is discarded.

If any of the partition for S_i is filled during the table scan, then that partition is scheduled to be written out as temporary segments to disk. The I/O is performed asynchronously. At the end of the table scan of S , a hash table is built from the maximum partitions of S that can be accommodated in the available memory. Note that available memory must allow for 15 to 20 percent overhead for managing the bitmap vector, hash table, partition table, and so on.

After the full table scan of S , the big table, B , is read. (Table B will be used the *probe input* as described below.) As table B is read, the join column value is compared with the bitmap vector. If the bitmap vector contains the join value, then the row from B is preserved and partitioned into the appropriate bucket. Otherwise, the row from B is discarded. In this example, the following rows in B are discarded when being read:

```
{ 0, 0, 2, 2, 2, 2, 2, 2, 9, 9, 9, 9, 9 }
```

The method of using a bitmap to eliminate rows during the partitioning phase is known as *bit-vector filtering*.

The overall cost of the hash join method may be greatly reduced by the initial eliminations. This is particularly true when the join cardinality is low relative to the sizes of the join tables, and, when the big input has a large number of rows to be joined with the small input.

If the join value from B is in the bit-vector and falls into any of the partitions of B already in memory, then the join is performed and the results is returned. Otherwise, the relevant select-list columns are written out to temporary segments.

After the first pass of B , the maximum number of unprocessed partitions of S are read from temporary segments into memory and a hash table is built. The previously read partitions of B are re-read to perform an in-memory join.

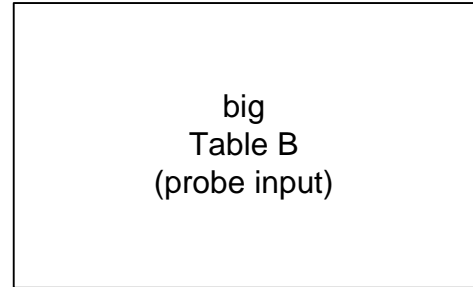
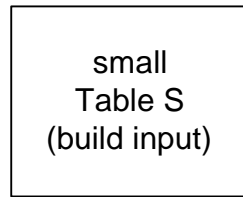
After the first pass of S and B is completed, when the next set of partitions S_i and B_i must be read from disk, the sizes of the S_i and B_i partitions are considered before determining the join order. The smaller of the two is used as the build input because it has a better chance of fitting into the available memory. This is known as *dynamic role reversal* when the build inputs are swapped. The hash join algorithm terminates when all partitions S_i and B_i are processed.

A *histogram driven hybrid hash join* keeps tracks of the partition sizes, the data distribution within the partitions at runtime, and, chooses partitions as build input that would minimize the I/O in the later stages of the algorithm.

The Oracle hash join algorithm uses the hybrid hash join method with bit-vector filtering and dynamic role reversal.

Figure 2 on the next page illustrates the various steps in the hash join algorithm.

1. Determine Fanout as M / C
Tune Fanout to account for bit-vector memory, PQ buffer, asynch I/O, etc.
2. Read S, hash into partition using hash function on join key, and, generate hash value for next level and store.
3. Build bitmap for unique join keys.
4. Put row into empty slot. If no space in current slot, or, no more empty slots, schedule writes to disk.
5. If end of S, try to find smallest N partitions in B to perform probe with B. Write (Fanout - N) partitions to disk.
6. Build hash table from N partitions of B in memory.



	P1	P2	P3	P4	P5
C11	C ₁₁	C ₂₁			
C12	C ₁₂	C ₂₂			
C13	C ₁₃	C ₂₃			
C14	C ₁₄	C ₂₄			

7. Read probe B, filter probe through bit-vector, if join value not present, discard row.

8. For filtered probe, use same hash function as before and find the partition. If row falls into partition in memory, perform join, else, put on disk.

9. If end of B, read(S,B) partition pairs from disk.
Use smaller input for creating hash table. (Dynamic role reversal.)

10. Iterate over probe. If build probe is 10 times memory, use nested loop hash join.

LEGEND

M : hash area memory
 P1 ... P5 : Partitions
 C11 C54 : Cluster in partition.
 An in-memory cluster is **slot**.
 The following condition must be met:
 $(\#slots) * Cluster\ size > M - overhead$
 Worst case scenario is no less than 6 slots.

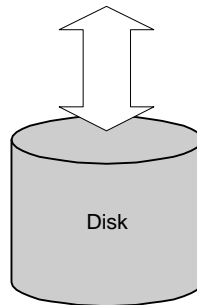


Figure 2: Hash Joins Algorithm

HASH JOINS ALGORITHM

In this section, we present a simplified step-by-step outline of the Oracle hash join algorithm.

Step 1 – Determine fan-out

Such that:

$$(\text{Number of Partitions}) \times C \leq F_{avm} \times M$$

where F_{avm} is the usable fraction of hash area and C is the cluster size. (See page 14.)

Fan-out Calculation takes into account overhead required for partition, hash tables, bitmap vector filter, some extra buffers required for asynchronous I/O and parallel query. (Calculating optimal fan-out is described in section “On-Disk Hash Join” on page 13.)

WHILE (ROWS IN S) LOOP

Step 2 – Read the join column value and select-list items from S .

Use an internal hash function, say HASH_FUN_1, and map the join column value to a partition.

Also generate a hash value for the next level using another hash function, say HASH_FUN_2, and store it with the join key. The second hash function value will be used to build a hash table in the later part of the algorithm

Step 3 – Build bitmap vector for all unique keys in build input.

Step 4 – If no space in the partition, write the partition to disk.

END LOOP

Step 5 – Try to fit the maximum number of possible partitions in memory from which a Hash table can be built and flush the rest to disk

Step 5.1 – Sort partitions by size.

Step 5.2 – Choose smallest number of partitions that fit in memory.

Step 5.3 – Schedule disk writes on other (Fan-out - X) number of partitions.

Step 6 – Build hash table from X partitions of S , using hash value already calculated for next level using HASH_FUN_2.

WHILE (ROWS TO BE READ FROM B) LOOP

Step 7.1 – Filter rows using bit-vector.

Step 7.2 – Hash the rows that pass the bit-vector filter into the appropriate partition using join key and internal HASH_FUN_1.

Step 7.3 –

IF hashed row falls into partitions in memory perform join by applying internal HASH_FUN_2 value and traversing the appropriate hash bucket

ELSE write to disk to the appropriate partition of S (we do keep track of R_i and S_i pairs on disk through the partition table) the HASH FN2 value, the join keys, and select list items

END LOOP

Step 8 – Read (S, B) unprocessed partition pairs from disk.

Use smaller input to build hash table and larger one for probe. In building the hash table we use internal HASH_FUN_2 value. This results in *dynamic role reversal* leading to zigzag execution trees. On the first

iteration, the optimizer should select the smaller table to be the build input and the larger table to be the probe. Role reversal only takes place after the first iteration.

Step 9 – If the smaller of the two inputs is too large and does not fit in memory then read the smaller build input into memory in chunks and iterate over the probe. This is called the *nested hash loops join*.

END OF HASH JOIN ALGORITHM

COSTING OF HASH JOINS

In-Memory Hash Join

Let us consider the cost of a hash join in the simplest case. That is, when the available hash area memory is large enough to hold the build input, S , after it is partitioned. The cost of the hash join is primarily dominated by reading the build input, building the hash table, and, reading the probe input to perform an in-memory join.

$$\text{Cost(HJ)} = \text{Read}(S) + \text{Build Hash Table in Memory (cpu)} \\ + \text{Read}(B) + \text{Perform In memory Join(cpu)}$$

Eliminating the CPU costs, which is orders of magnitude cheaper compared to disk I/O in terms of time, the dominant cost of hash joins is:

$$\text{Cost(HJ)} \propto \text{Read}(S) + \text{Read}(B)$$

On-Disk Hash Join

When the available hash area memory, M , is insufficient to hold the build input, S , it must be written to disk. Obviously, the bigger table, B , will also be written to disk.

$$\text{Total Cost(HJ}_{12}) \propto \text{Cost(HJ}_{\text{iteration 1}}) + \text{Cost(HJ}_{\text{iteration 2}})$$

Fan-out, F , the number of partitions, is dependent on the I/O efficiency of cluster size, C . (Equation 1 on page 14.)

F is computed as $(F_{\text{avm}} \times M) / C$

where F_{avm} is the fraction of available hash memory. Generally, $F_{\text{avm}} = 0.8$ is a good estimate.

If $S > M$ then the I/O cost for hash joins at the end of first iteration is given by:

$$\text{Cost(HJ}_{\text{ITERATION 1}}) \propto \text{Read}(S) + \text{Read}(B) + \text{Write}((S - M) + (B - B * M / S))$$

That is, left and right inputs must be scanned, and, the partitions that do not fit in memory must be written to disk.

Iteration 2 in the hash join algorithm involves processing S_i and B_i partition pairs on disk using a nested hash loops join when the partitioned build input does not fit in memory. For each chunk of build input read, the probe input is read so this requires multiple reads of the probe input

$$\text{Cost(HJ}_{\text{ITERATION 2}}) \propto \text{Read}((S - M) + n \times (B - B * M / S))$$

The dominant cost of a hash join is I/O on the probe input. If n is the number of steps required to complete iteration 2, then the I/O cost is approximately proportional to:

$$(S - M) + n (B - B * M / S)$$

Because the probe input may be reread more than once, n is used as a multiplier to calculate the re-reading cost.

This multiplier n may be computed as the ratio of the size of each build partition (S / F) over the size of memory, M . Therefore:

$$n = (S / F) / M$$

When n is large – typically, greater than 10 – the hash join algorithm will incur a lot of I/O writing and re-reading the data in the partitions.

For smaller values of n – typically, less than 10 – the hash-join should perform better than sort-merge join for normally distributed data.

The value of n is inversely proportional to fanout, F . Increasing fanout will reduce n and when memory is fixed, fanout can be increased by reducing cluster size, C .

HASH JOINS PERFORMANCE TUNING

Generally, when a smaller table, S , in a join fits in memory, then hash joins should perform better than SMJ and NLJ. The performance difference is due to the reduced search space and optimizations like bit-vector filtering and dynamic role reversal.

Unlike SORT_AREA_SIZE, all hash join parameters are dynamic and can be set for the session. This provides a lot of flexibility when tuning individual queries.

Minimum Setting for HASH_AREA_SIZE

We recommend that a minimum HASH_AREA_SIZE be determined and used for hash joins, otherwise performance will be unacceptable. We name this minimum value, $M_{critical}$. A proof is provided below, which is corroborated by research findings.

Let $M_{critical}$ be the amount of allocated memory for HASH_AREA_SIZE below which hash join performance degrades drastically.

Let *cluster size*, C , the unit of I/O be:

$$C = DB_BLOCK_SIZE \times HASH_MULTIBLOCK_IO_COUNT \leq 64 \text{ K} \quad (1)$$

Generally, $C \leq 64 \text{ K}$ because this is the limit set by most operating systems for the unit of I/O.

Let F_{avm} , the *fraction of available memory* usable from the hash area memory, be:

$$F_{avm} = 0.8$$

We assume an overhead of 10 to 20 percent for storing partition tables, bitmap of unique left input S , and the hash table. A few buffers are also set aside for performing parallel query and asynchronous I/O.

Let *fan-out*, the maximum number of partitions into which S can be split be:

$$\text{Fan-out} = (F_{avm} \times M_{critical}) / C$$

Hash join performance degrades when the size of each of the partitions from the build input are not small enough to fit in available memory. The degradation occurs because the probe input must be re-read multiple times to perform a nested hash loops join. (See equation 4.)

When the size of each partition of build input, $S \geq \text{AVAILABLE MEMORY}$, we get:

$$\begin{aligned}
 \Rightarrow & \quad (S / \text{Fanout}) \geq F_{\text{avm}} \times M_{\text{critical}} \\
 \Rightarrow & \quad (S / (F_{\text{avm}} \times M_{\text{critical}}) / C) \geq \text{AVAILABLE MEMORY} \\
 \Rightarrow & \quad (S \times C) \geq M_{\text{critical}} \times M_{\text{critical}} \times F_{\text{avm}} \times F_{\text{avm}} \\
 \Rightarrow & \quad (M_{\text{critical}})^2 \leq (S \times C) / (F_{\text{avm}})^2 \\
 \Rightarrow & \quad M_{\text{critical}} \leq \text{SQRT}(S \times C) / F_{\text{avm}} \tag{2}
 \end{aligned}$$

Equation 2 is corroborated by our research findings. We observed that for any HASH_MULTIBLOCK_IO_COUNT, reducing hash area memory below M_{critical} resulted in drastic performance degradation.

A worst case scenario, where all rows are returned from B , was used. The test query used was as follows:

```

SELECT /*+ USE_HASH(B) ORDERED */ S.col1, S.col2, ... ,S.coln
FROM S, B where S.col1 = B.col1 ;
    
```

Average row sizes of S and B were 100 bytes each. Variables in the research tests were as follows:

Table sizes: $S = 10 \text{ MB}$ or $S = 100 \text{ MB}$, and, $B = 1000 \text{ MB}$.

$\text{HASH_AREA_SIZE} = m \times M_{\text{critical}}$ where $m = \{0.2, 0.5, 2, 5\}$

$\text{HASH_MULTIBLOCK_IO_COUNT} = h$ where $h = \{1,2,4,8\}$

The size of S in Equation 3 was computed as the data going into the equi-join query; that is:

```

Cardinality(S) x [ [ Average Size(select list item 1 )
                   +[ Average Size(select list item 2 )
                   +[ Average Size(select list item 3 )
                   + ...
                   +[ Average Size(select list item n )
                   +[ Average size of join keys if not already in select list items]
    
```

Chart 1 on the next page shows some of our research results when HASH_MULTIBLOCK_IO_COUNT and HASH_AREA_SIZE were varied. The results are for $S = 100 \text{ MB}$ and $B = 1000 \text{ MB}$. Table S contained 1 million rows and table B contained 10 million rows. The join column was defined as NUMBER(10) in both tables. Degradation in some test runtimes was so bad, that we were forced to terminate tests. (In the chart below, the data point at 16350 seconds, or 4.5 hours, was terminated before completion.)

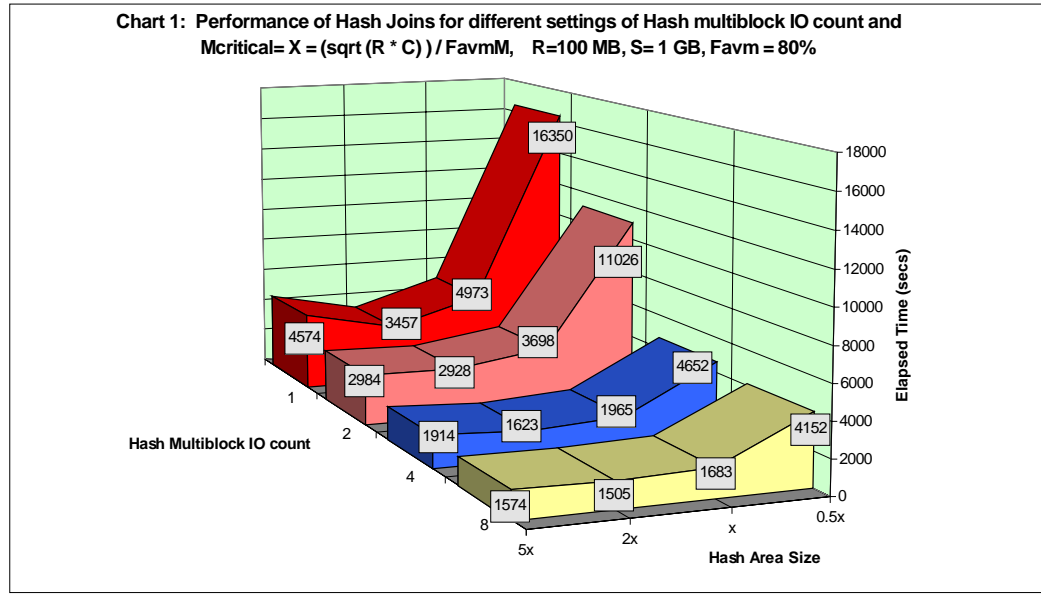


Chart 1 above shows that a HASH_AREA_SIZE below $M_{critical}$ causes a drastic degradation in performance.

Generally, hash area sizes greater than $M_{critical}$ resulted in better performance.

However, a hash area size greater than $M_{critical}$ does not necessarily provide better performance. At $M_{critical}$, HASH_MULTIBLOCK_IO_COUNT begins to impact performance. (I/O versus fanout.) For example, HASH_AREA_SIZE = $5 \times M_{critical}$ and HASH_MULTIBLOCK_IO_COUNT = 1 resulted in sub-optimal fanout calculation.

Furthermore, in this scenario comparison of performance of hash joins across HASH_MULTIBLOCK_IO_COUNT for a given HASH_AREA_SIZE can be misleading because a lower HASH_MULTIBLOCK_IO_COUNT utilizes less hash area memory.

The amount of hash memory, $M_{critical}$, was calculated by substituting values of HASH_MULTIBLOCK_IO_COUNT in the Equation 2, where S was 10 MB or 100 MB, and, $F_{avm} = 0.8$.

HASH_AREA_SIZE and Tablespace I/O

The bigger the hash area memory, HASH_AREA_SIZE, the better the chances of caching the build input in memory. However, caution is required because very large memory allocation can cause swapping on some systems. The default value for HASH_AREA_SIZE is $2 \times \text{SORT_AREA_SIZE}$, with a maximum value that is operating system dependent. Hash area memory is allocated from the user global area (UGA).

Our preliminary research findings show that *no* temporary tablespace I/O occurred when:

$$\text{HASH_AREA_SIZE} = n \times \text{SIZE}(S) \quad \text{where } n \text{ is } 1.4 \text{ to } 1.6$$

The above can be exploited when there is plenty of system memory, but a shortage of disk space for the temporary tablespace. The idea is to increase HASH_AREA_SIZE until the small table fits in the hash area memory. In such cases, assuming that there is enough inexpensive CPU power compared to disk I/O, then the cost would be as follows:

$$\text{Cost(HJ)} \propto \text{Read}(S) + \text{Read}(B)$$

HASH_MULTIBLOCK_IO_COUNT

Higher values of `HASH_MULTIBLOCK_IO_COUNT` make I/O efficient because large chunks of size cluster factor, C , can be read/written with a single I/O. This causes a smaller fan-out, $F = F_{avm} \times M / C$.

However, this means that there will be fewer large partitions and may not fit in memory!

`HASH_MULTIBLOCK_IO_COUNT` should set to 1 when the amount of data in the smaller table is orders of magnitude larger than available hash memory. This will result in the maximum number of partitions possible, each of which has a higher chance of fitting into memory.

Because Oracle performs I/O in 64 K chunks and `DB_BLOCK_SIZE` is fixed, `HASH_MULTIBLOCK_IO_COUNT` can be varied such that the following holds true:

$$\text{DB_BLOCK_SIZE} \times \text{HASH_MULTIBLOCK_IO_COUNT} \leq 64 \text{ K}$$

Using the above formula, `HASH_MULTIBLOCK_IO_COUNT` can be varied until the optimum value is found giving best performance for a query. If it is not possible to fine-tune individual queries, it is relatively safe to set `HASH_MULTIBLOCK_IO_COUNT` to a moderate value, such as 4 or 8, for `DB_BLOCK_SIZE = 4K`.

An inappropriate hash function can lead to excessive collisions in the hash table, and, to partitions of uneven sizes. In the worst case, all key values fall into a single partition and one complete iteration is wasted^[3]. It might also be the case that fan-out yields too many or too few partitions. In these cases, a different setting for `HASH_MULTIBLOCK_IO_COUNT` may help.

Note that having skew in partitions is not necessarily bad, as long as the smallest of the two partition pairs is small enough relative to available memory. It does not matter which of the tables is smaller, because of dynamic role reversal. Furthermore, data skew may have little effect when a large number of rows are eliminated through bit-vector filtering

Size of Join Inputs

The hash join algorithm^[3] terminates when the smaller of the inputs fits completely in memory, regardless of the size of the probe input. In contrast, when sorting two inputs of a sort merge join, the size of each input – sort area size and number of sort runs – determines the number of merge levels to be used in the sort.

A hash join is the recommended join method when the tables are different sizes and the smaller table is close to the available memory.

Reliability of Predicting Relative Sizes of Two Join Inputs

If the cost based optimizer can accurately determine the sizes of the two join inputs for the first pass on S and B , then it can choose the smaller input to be the build input and the larger one to be the probe input.

If there are more partition passes on S and B , the hash join operation will dynamically pick the smaller of the two inputs. Predicting the relative size of join inputs is not a problem because the sizes of the partition files are maintained.

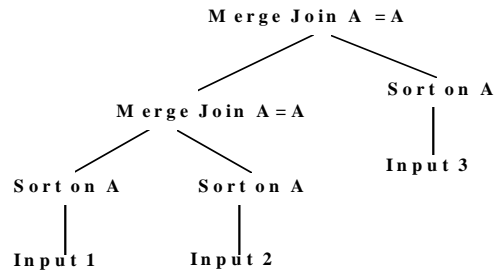
For the first pass of S and B to be effective, the data in the columns being equi-joined must be appropriately analyzed.

The columns in the join might need to be analyzed with appropriate sampling in `ESTIMATE` or `COMPUTE` mode and with appropriate number of buckets (histograms) for non-uniform distributions.

Exploiting Interesting Orderings

Interesting orderings^[4] are multiple join clauses on the same predicate in a n-way join. The figure below illustrates the scenario where an intermediate sort on *A* can be avoided because the output of merging *Input1* and *Input2* are already sorted.

A sort merge join will be more efficient than a hash join when data can be pipe-lined between multiple operations to avoid intermediate sorts, as shown in the example below.



Interesting Orderings in the Joins

Number of Distinct Values

You should consider the impact of the number of distinct value in the build input and probe input. Note that the number of distinct values in the build inputs are independent of each other.

Bit-vector filtering is effective when the number of distinct values in the columns being joined are few and most of the items in the probe are filtered out by the bit-vector. Bit-vector filtering is not very effective when the cardinality of the join output is very large and most input rows match. Furthermore, bit-vector filtering is not very effective when there are too many distinct values and most or all bits in the bit-vector filter will be set, such that most of the probe rows pass through the bit-vector filter.

Skew in Build and Probe Inputs

You should consider skews in the build and probe input. Note that the skews are independent of each other.

A skewed distribution (as opposed to an uniform distribution) in the build input should make bit-vector filtering more effective than a skewed distribution of the probe input. This is because the hash join algorithm terminates when the build input fits in memory, regardless of the size or skew of the probe input. Furthermore, it is the build input which determines the effectiveness of the bit-vector. Effectiveness of bit-vector filtering is more pronounced when skew is in the build input rather than the probe input.

Other Performance Considerations

A query that has an ORDER BY or GROUP BY operation may benefit from using an index access path, if the index yields the resultant data in the right order, then the cost of sorting can be eliminated. In such cases, the EXPLAIN plan output is indicated by a GROUP BY NOSORT or ORDER BY NOSORT.

Some queries might benefit from an INDEX_FFS hint. An INDEX_FFS hint will force a fast full scan of the index. This new row source was implemented using multi-block I/O and can use parallelism. However the resulting rows are not guaranteed to be sorted.

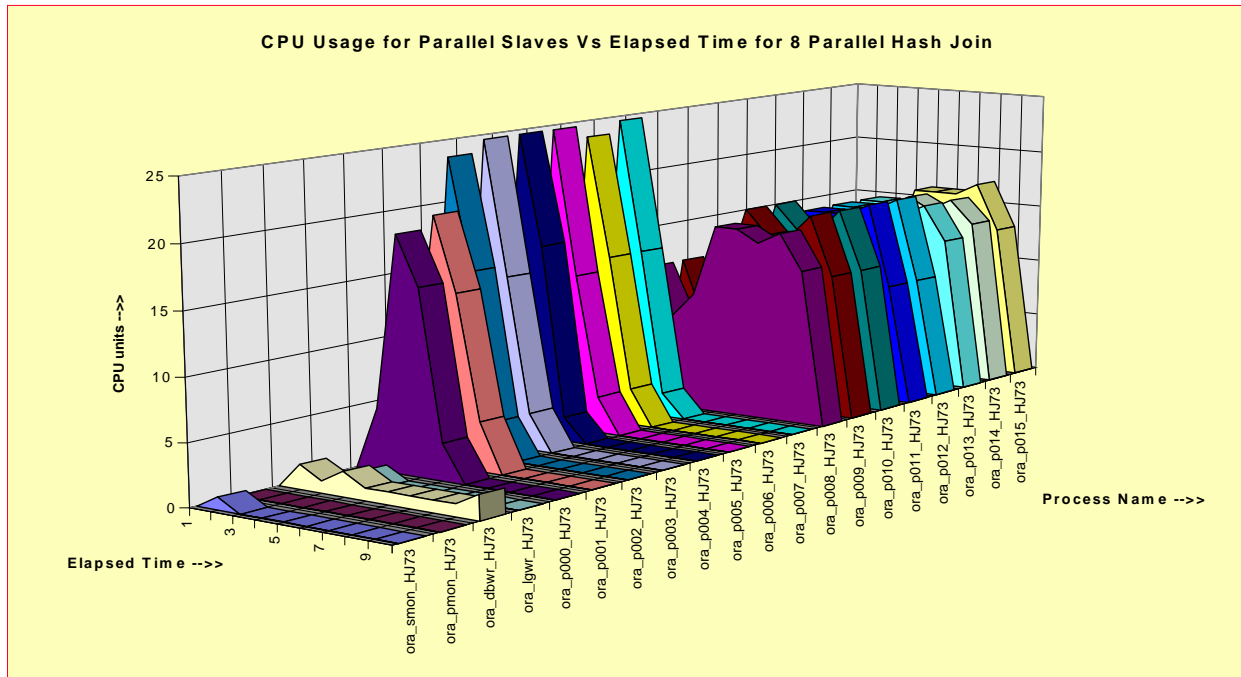
One other consideration against choosing a sort merge join^[4] is when the query result must be sorted on a join key and the join result is so large that sorting the output is expensive when compared to joining the two inputs.

TROUBLESHOOTING

The following tips should be useful in trouble shooting problems with hash joins.

1. Make sure that the smaller table is the driving table (build input).
2. Check that tables and/or columns of the join tables are appropriately analyzed.
3. Histograms are recommended only for non-uniform column distributions. If needed, you can override the join-order chosen by the cost based optimizer using the ORDERED hint.
4. Check that hash area size, M , allocated for hash joins is at least $M_{critical}$. (See Equation 2).
5. To eliminate or minimize I/O to the temporary tablespace, try setting hash area size, M , such that $M \geq M_{critical}$ and $M = \lceil 1.6 \times SIZE(S) \rceil$.
6. For parallel hash joins, make sure there is no skew in the slave processes workloads by monitoring CPU usage, and, messages sent and received using VSPQSTAT. Monitoring these statistics over the elapsed time of the operation will show whether there is slave workload skew. Skews could also occur because there are very few distinct values in the column being equi-joined.

The figure below illustrates two sets of slave processes in a parallel hash join of degree 8; table scanners are ora_p000 to ora_p007 and hashers are ora_p008 to ora_p015. These two slave sets have different CPU profiles and show approximately uniform work load among the slaves. This figure also illustrates the inter-operator and intra-operator parallelism used in Oracle Parallel Query architecture. The CPU profile was monitored using the *ps* Unix command over the elapsed time of the hash join query.



APPENDICES

APPENDIX A – Citations

- [1] *Oracle7 Server Documentation, Addendum, Release 7.1*, Part No: A12042-3.
- [2] *Oracle Release 7.3 Server Addendum*, and, *Oracle Release 7.3 Readme File*
- [3] *A Performance Evaluation of Histogram-Driven Recursive Hybrid Hash Join*, Ph.D dissertation, Graefe Goetz, Portland State University.
- [4] *Sort-Merge Join: An Idea Whose Time Has(H) Passed?*, IEEE International Conference on Data Engineering., February 1994, Graefe Goetz.
- [5] September 1996 to January 1997 Interviews: Cetin Ozbutun, Sameer Patkar, Linda Willis, Gary Hallmark.
- [6] *Query Processing in Parallel Relational Database Systems*, section *Parallel Processing of Joins*, IEEE Computer Society Press, 199?, Hongjun Lu, et al.
- [7] *Tuning Large Sorts*, Technical Reports Compendium, Volume I, 1996, Center of Expertise, Oracle Worldwide Customer Support.
- [8] Oracle confidential development documents, by Cetin Ozbutin.

APPENDIX B – Enabling Hash Joins

COMPATIBLE must be set to 7.3 or higher.

HASH_JOIN_ENABLED must be set to TRUE.

OPTIMIZER_MODE must be set to CHOOSE. Hash joins are only invoked if the cost based optimizer is used.

A hash join can be forced using a hint as shown in the following example:

```
SELECT /*+ USE_HASH(S) */ S.a, B.a , ... FROM S,B WHERE S.a = B.a ;
```

Unlike SORT_AREA_SIZE, which requires re-starting the database, dynamic hash join parameters allow individual queries to be tuned per session. Dynamic parameters affecting hash joins are: HASH_MULTIBLOCK_IO_COUNT and HASH_AREA_SIZE.

HASH_AREA_SIZE determines maximum amount of memory to be used for hash join.

HASH_MULTIBLOCK_IO_COUNT determines how many blocks should be read and written at a time to temporary space. This parameter is similar in functionality to DB_MULTIBLOCK_IO_COUNT.

Because HASH_JOIN_ENABLED=TRUE by default, customers might want to look out for queries whose access plans might have changed after upgrading to Release 7.3.

Alternatively, disable this feature and selectively turn it on per session by using the ALTER SESSION command, but only enable the feature in production if you feel the Hash Joins feature performs adequately well in your setting.