Getting the Most from Hash Joins on DB2® Universal Database™

Adriana Zubiri
DB2 UDB Performance Team
IBM Toronto Lab
Email: zubiri@ca.ibm.com
August, 2002

Abstract

In DB2 UDB, the optimizer can choose between nested loop joins, merge joins, and, if enabled properly, hash joins. Hash joins can significantly improve the performance of certain queries when the system is tuned properly. The purpose of this paper is to explain how hash join works and how to properly tune your system to be able to exploit its potential.

Trademarks
The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries: AIX, DB2, DB2 Universal Database, IBM

Other company, product and service names may be trademarks or service marks of others.

© 2002 International Business Machines Corporation. All rights reserved.

Table of Contents

1. Introduction 1

2. Background 1

- **2.1 Join methods** 1
- **2.2 Which join method will be chosen?** 3

3. How to tune and monitor hash joins 3

4. The setup and configuration for our experiments 6

- **4.1 System configuration** 6
- **4.2 Workload description** 6
- **4.3 Database layout and tuning** 6

5. The results of our experiments 8

- **5.1 The impact of enabling hash join** 8
- **5.2 Sortheap experiments** 12
- **5.3 Sheapthres experiments** 14

6. Conclusions 17

1. Introduction

The DB2 UDB optimizer can choose between different methods when it performsa join: by default, it chooses between a nested loop join and a merge join. When a special environment variable is set, it can also choose a hash join. Hash joins can significantly improve the performance of certain queries, especially in Decision Support System (DSS) environments in which queries are complex. The purpose of this paper is to explain how hash join works and how to properly tune it to achieve the best performance.

We will start by giving the background necessary to read this paper. We will explain the different types of join methods, how they work, and how DB2 UDB chooses a particular method for a particular join. We will then explain the elements needed to tune and monitor hash join. After this, we will present several experiments we have conducted and many interesting results. We will finish the paper with our conclusions.

2. Background

A join between two tables is an operation in which the rows from one table are concatenated with the rows of the other table. In addition, you can specify a condition to define which rows are concatenated. To execute this operation, DB2 can choose different join methods. This section gives an overview on how the different join methods work and how DB2 chooses which join method to use for a particular join.

2.1 Join methods

When joining two tables, no matter which join method is being used, one table is selected as the outer table and the other one as the inner table. The optimizer decides which table would be the outer and which one would be the inner based on costs and the type of join method selected. The outer table is accessed first and is only scanned once. The inner table could be scanned multiple times depending on the type of join and which indexes are present. It is also important to remember that even though you are trying to join more than two tables, the optimizer will only join two tables at a time and keep intermediate results if necessary. To understand the advantages of the hash join method it is also important to understand how the other join methods work.

Nested loop join

As we mentioned before, the outer table is scanned only once. With nested loop joins, for each row in the outer table there are two ways to find the matching rows in the inner table:

- Scan the inner table. That is, read every row in the inner table and for each row determine if it should be joined or not with the row we are considering for the outer table.
- Do an index lookup of the joins column(s) on the inner table. This is possible when the predicate used for the join includes a column that is in an index for the inner table. This dramatically reduces the number of rows accessed in the inner table.

In nested loop join the decision on which is the outer table and which is the inner table is very important because the outer is only scanned once, and the inner is accessed once for every row in the outer table. As mentioned before, the optimizer uses a cost model to decide which table would play each role. Some factors taken into account by the optimizer for this decision are:

- Table size
- Buffering
- Predicates
- Ordering requirements
- The presence of indexes

The join columns cannot be LONG or LOB fields.

Merge join

Merge joins require an equality join predicate (i.e., a predicate of the form table1.column = table2.column). It also requires that the input tables be sorted on the join columns. This could be achieved by scanning an existing index or by sorting the tables before proceeding with the join. The join columns cannot be LONG or LOB fields.

Both tables are scanned at the same time looking for matching rows. Both the outer and inner table are scanned only once, unless there are repeated values in the outer table, in which case some parts of the inner table may be scanned again. Because the tables are generally scanned only once, the decision on which is the outer and which is the inner table is somewhat less important than with the other join methods. Still , because of the possibility of duplicate values, the optimizer usually chooses the table with fewer duplicate values as the outer table. Ultimately, however, it uses the cost model to decide which table should play each role.

Hash join

Hash joinsrequire one *or more* equality join predicates in which the column types for each predicate are the same. In the case of CHAR types, the lengths have to be the same. In the case of DECIMAL types, the precision and scale must be the same. Again, the join columns cannot be LONG or LOB fields. The fact that hash join can handle more than one equality predicate is a nice advantage over merge join, which can handle only one.

For hash joins, the inner table (called the *build* table) is scanned first, and the rows are copied into memory buffers. These buffers are divided into *partitions* based on a “hash code,” which is computed from the column(s) in the join predicate(s). If there is not enough space in memory to hold the entire table, some partitions are written into temporary tables on disk. Then the outer table (called the *probe* table) is scanned. For each row in the probe table the same hash algorithm is applied to the join column(s). If the hash code obtained matches the hash code of an build row, the actual join columns are compared. If the partition that matches the probe table row is in memory then the comparison takes place immediately. If the partition was written to a temporary table, then the probe row is also written into a temporary table. Finally, the temporary tables containing rows from the same partitions are processed to match the rows.

Because of the advantage to to keep the build table in memory, the optimizer usually chooses the smaller table as the build table to avoid having to spill that table to disk. But, once more, the cost model is the one deciding which table would play each role.

Let’s get into more detail on how hash join takes advantage of SMP systems. In SMP systems (with intra-parallel = ON and dft_degree > 1), a hash join may be executed, in parallel, by multiple tasks on the same or multiple CPUs. When executing in parallel, the build table is dynamically partitioned into multiple parallel tuple streams, and each stream is processed by a separate task to enter build tuples into memory. At the end of processing the build table streams, the hash join process adjusts the contents of memory and performs any needed movement of partitions into or out of memory. Next, multiple parallel tuple streams of the probe table are processed against the in-memory partitions and spilled, as needed, for tuples from hash join partitions that were spilled to

temp tables. Finally, the spilled partitions are processed in parallel, with each task processing one or more of the spilled partitions.

2.2 Which join method will be chosen?

So far we have discussed the different join methods available in DB2 UDB. As we can see there are certain methods that present, at first glance, a better choice than others. For example, merge join has the advantage of scanning the tables only once when compared with nested loop join that scans the inner table for every row of the outer. So, it seems as if merge join would be a better choice; however, if there is an index present, a nested loop could end up being a better choice.

Similarly, a hash join seems a better choice than a merge join because it does not need to sort the input tables before executing, but if we need to preserve the order of the rows in the outer table, then a merge or nested loop join might be a better choice -- a hash join cannot guarantee ordering because it might spill to disk and that would disrupt the order.

So how does the DB2 UDB optimizer decide which join method to use for a particular join? First, it has to take into account the types of predicates in the query. When the possible join methods are chosen, then the DB2 UDB optimizer decides which join method to use based on a cost model and on the selected optimization level. The optimization level is a configurable parameter in the database configuration file that tells the optimizer how much optimization to do. The higher the value, the more optimization occurs. The possible values for optimization level are: 0, 1, 2, 3, 5, 7 and 9. These values affect the possible join methods as follows:

- A nested loop join is a possibility for every optimization level.
- A merge join is a possibility for optimization levels 1 and above.
- A hash join is a possibility for optimization levels 5 and above.

3. How to tune and monitor hash joins

Hash joins can significantly improve the performance of certain queries, especially in Decision Support Systems (DSS) that have complex queries. One of the performance advantages that hash join has over merge join is that it does not require any sorts beforehand, which could be very costly. The key for hash join is to be able to fit all (or as many as possible) rows of the build table into memory without having to spill to disk. The memory used for these buffers comes from *sortheap*, so the tuning of this (and the *sheapthres*) parameters are very important.

The *sortheap* is a database configurable parameter that defines the maximum amount of memory that could be used for a sort or for a hash join. Each sort or hash join has a separate *sortheap* that is allocated as needed by the database manager. Not necessarily all sorts and hash joins allocate this amount of memory; a smaller *sortheap* could be allocated if not all the memory is needed. *Sortheap* can be allocated from shared or private memory depending of the requirements of the operation. A shared *sortheap* is used only for intra-parallel (SMP) queries where degree > 1; a private sortheap is used when only one agent will perform the sort or hash join, and there is no need to share the memory with other agents. It is also important to understand that one single query might have more than one hash join and/or sort and that multiple sortheaps might be required to be allocated at the same time depending on the nature of the plan.

The *sheapthres* is a database manager configurable parameter. This parameter is used differently for shared and private sorts:

- For private sorts or hash joins, sheapthres acts as an instance-wide *soft* limit on the total amount of memory that all concurrent private sorts can consume. When this consumption reaches the limit, then the memory allocated for additional incoming sortheap requests will be reduced significantly.
- For shared sorts or hash joins, sheapthres acts as an instance wide *hard* limit on the total amount of memory that all shared sorts can consume. When this consumption gets close to the limit, then the memory allocated for additional incoming sortheap requests will be reduced significantly, and eventually no more shared sort memory requests will be allowed.

In uniprocessor systems, hash join uses only private sortheaps. In SMP systems (with intra-parallel = ON and dft_degree > 1), hash join uses shared sortheaps for the first part of the operation and private sortheaps for the last phase. For more information on these parameters, see the *Administration Guide: Performance* that comes with DB2 UDB.

How do we tune these parameters? As in most tuning exercises, you need to have a starting point: a workload to test and tools to measure the results of your tuning. After that, it is an iterative process of changing one parameter at a time and measuring again. In most cases you would already have values set for sortheap and sheapthres in your existing system, so we suggest that you start with what your current settings are. If you have a brand new installation, you can follow the rules of thumb for DSS systems which are to allocate 50% of the memory you have to buffer pools and the other 50% for sheapthres. Then make your sortheap equal to the sheapthres divided by the number of complex concurrent queries that would be executing at one time, by the maximum number of concurrent sorts and hash joins that your average query has. (A good starting number for this is 5 or 6.) In summary:

sortheap = sheapthres / (complex concurrent queries * max number of concurrent sorts and hash joins in your average query)

Remember that “complex concurrent queries” is not equal to “concurrent users”. There are usually many more users than complex concurrent queries executing at a time. In most DSS TPC-H benchmarks, which try to push the databases to their limits, no more than 8 or 9 is used as the number of complex concurrent queries (a.k.a. *streams*) for databases up to 10 TB, so start conservatively and move up as necessary.

After you have set your starting sortheap and sheapthres values are, run an average workload, and collect database and database manager snapshots. DB2 UDB offers some monitor elements to be able to monitor hash join. You don’t need to turn any monitor switches on, because all hash join monitor elements are collected with the Basic monitor switch. Also, all elements are counters and are resettable. For details on how DB2 UDB snapshots work, see the *System Monitor and Reference Guide* that comes with DB2 UDB. Here is a description of each of the monitor elements:

Total Hash Joins: This monitor element counts the total number of hash joins executed.

This value can be collected with a database or application snapshot.

Hash Join Overflows: This monitor element counts the total number of times that hash join exceeded the available sortheap while trying to put the rows of the build table in memory.

This value can be collected with a database or application snapshot.

Hash Joins Small Overflows: This monitor element counts the total number of times that hash join exceeded the available sortheap by less than 10% while trying to put the rows of the build table

in memory. The presence of small overflows suggest that increasing sortheap can help performance.

This value can be collected with a database or application snapshot.

Total Hash Loops: This monitor element counts the total number of times that a single partition of a hash join was larger than the available sortheap.

Remember that if DB2 UDB cannot fit all the rows of the build table in memory, certain partitions are spilled into temporary tables for later processing. When processing these temporary tables, DB2 attempts to load each of the build partitions into a sortheap. Then the corresponding probe rows are read, and DB2 UDB tries to match them with the build table. If DB2cannot load some build partitions into a sortheap, then DB2 must resort to the hash loop algorithm. It is very important to monitor hash loops because hash loops indicate inefficient execution of hash joins and might be the cause of severe performance degradation. It might indicate that the sortheap size is too small for the workload, or, more likely, the sheapthres is too small, and the request for sortheap memory could not be obtained.

This value can be collected with a database or application snapshot.

Hash Join Threshold: This monitor element counts the total number of times that a hash join sortheap request was limited due to concurrent use of shared or private memory heap space. This means that hash join requested a certain amount of sortheap but got less than was requested. It might indicate that the sheapthres is too small for the workload.

This value can be collected with a database manager snapshot.

4. The setup and configuration for our experiments

The previous section described how to monitor and tune hash joins. In this section, we relay the results of our tuning experiments.

4.1 System configuration

Our tests were made on an IBM H80 with:

- 6 x 500 MHz processors
- 16 GB memory
- 8 x 18.2 GB SSA (external) drives, 2 x 18.2 GB SCSI (internal) drives

We used DB2 UDB Enterprise Edition v7.2 at FixPack 7 running on AIX® 4.3.3.

4.2 Workload description

The database schema and queries were taken from the industry-standard TPC-H benchmark. The TPC-H benchmark models a decision support system by executing ad-hoc queries under controlled conditions. It represents information analysis of an industry that must manage, sell, and distribute a product worldwide. The 22 queries answer questions in areas such as pricing, promotions, supply-and-demand management, profit and revenue management, customer satisfaction, market share, and shipping management. The TPC-H database size is 10 GB, which means it represents a database with 100,000 suppliers. The database is populated with a TPC-supplied data generation program, dbgen, which creates the synthetic data set. It is important to understand that these results are not meant to be compared with any published TPC-H benchmark as we did very little tuning, as we explain later. For more information on the TPC-H benchmark, see http://www.tpc.org

4.3 Database layout and tuning

We used a very basic database layout. We created four DMS tablespaces for the data: one for lineitem data, one for lineitem indexes, one for the rest of the data and one for the rest of the indexes; and an SMS temporary tablespace. All five tablespaces were spread over the same seven SSA disks and used 16 KB pages. No special transfer rates or overhead parameters were used.

As for memory, because we used the 32-bit engine version for DB2 UDB on AIX, we were only allowed to use 7 AIX 256 MB segments, so we followed the rules of thumb for DSS systems as described in the previous section and used 192000 4KB pages (750 Mb) for the sortheap threshold (*sheapthres*) and 48000 16KB pages (750 Mb) for the bufferpool. We could have used EEE to take advantage of all the memory in the box, but we decided not to so that we could keep the study simple. All of our results for EE are applicable to EEE as well.

We used the following db2set environment variables:

```
db2set DB2_HASH_JOIN=Y
db2set DB2_RR_TO_RS=ON
db2set DB2MEMMAXFREE=8000000
db2set DB2MEMDISCLAIM=Y
db2set DB2_MMAP_WRITE=NO
db2set DB2_MMAP_READ=NO
db2set DB2_EXTENDED_OPTIMIZATION=ON
db2set DB2_LIKE_VARCHAR=y,y
```

The database and database manager configuration files we used are shown in Table 1.

Table 1. The database and database manager configuration values

Parameter	**Configuration File**	**Value**	**Changed during experiments?**
numdb	dbm cfg	1	No
svcename	dbm cfg	xzubiri	No
dft_mon_sort	dbm cfg	on	No
dft_queryopt	dbm cfg	7	No
sheapthres	dbm cfg	192,000	Yes
intra_parallel	dbm cfg	on	Yes
max_querydegree	dbm cfg	-1	Yes
sortheap	db cfg	10,666	Yes
dft_degree	db cfg	6	Yes
num_freqvalues	db cfg	0	No
num_quantiles	db cfg	300	No
dbheap	db cfg	8,000	No
buffpage	db cfg	48,000	No
locklist	db cfg	8,192	No
num_iocleaners	db cfg	7	No
num_ioservers	db cfg	21	No
max_appls	db cfg	15	No
dft_extent_sz	db cfg	64	No
dft_prefetch_sz	db cfg	448	No
logbufsz	db cfg	64	No
logfilsz	db cfg	4,000	No
logprimary	db cfg	100	No
logsecond	db cfg	25	No
maxfilop	db cfg	1,024	No

5. The results of our experiments

For every result we present, we have done two runs to account for run-to-run variability. That means that the results here are the *average* of the two runs. In every case the difference for the two runs was less than 1%. We also used relative elapsed time in the presentation of the results to avoid comparisons with any published TPC-H benchmark. We will explain the details for each experiment.

5.1 The impact of enabling hash join

The main purpose of this experiment was to understand the impact of enabling hash join without doing any specific tuning.

We divided the experiment into an SMP (multiple CPUs) and a UNI (one CPU only) set of tests. For the SMP portion we tested 1, 3, 5, 7 and 10 streams. For the UNI portion we only tested 1, 3 and 5 streams, because having more streams with only one CPU was not realistic.

The parameters shown in Table 2 were the only differences e between the SMP and UNI sets of tests.

Table 2. Parameters that differed between UNI and SMP tests

Parameter	Configuration File	SMP	UNI
max_querydegree	dbm cfg	-1	1
intra_parallel	dbm cfg	on	off
dft_degree	db cfg	6	1

To reflect basic realistic tuning, we changed the sortheap values for different numbers of streams. For the SMP experiments, the sortheap value was determined as follows:

sortheap = sheapthres / (number of streams * 6)

For the UNI experiments, the sortheap value was determined as follows:

sortheap = 64000 / 6

The reason for this fixed number (10666) is that for uniprocessors, sortheap is allocated from private memory, and in DB2 UDB 32 bits on AIX, the maximum amount of memory we can allocate for a sortheap is one segment, which is equal to 256 MB.

To enable and disable hash join, we used the DB2_HASH_JOIN DB2 UDB registry variable.

The result times are relative to the time that it takes to run the workload for one stream with hash join = ON. So, we consider that running the workload for one stream with hash join = ON takes one unit of time. Every other result is expressed relative to that unit for this experiment. For example it takes 3.14 times unitsto run one stream with hash join = OFF.

The results are found in Figures 1 and 2. In every experiment for both SMP and UNI, enabling hash join gave better results than disabling it. It is also important to notice that the larger the number of streams, the larger the performance gain. Another interesting observation from these experiments is that the performance improves almost linearly with the number of streams, which shows how well DB2 UDB scales with minimum tuning. In fact, it scales slightly better with hash join on rather than off.

Figure 1 shows that scalability drops a little at 10 streams when hash join is enabled. What causes this smal drop? If you look closely into the hash monitor information shown in Figure 2, you can see that at 10 streams we started getting hash loops, because the size of the sortheap was decreased significantly to accommodate the increasing number of sortheaps that needed to be concurrently allocated. When you tune your system, keep in mind the impact of having hash loops. We will see more of this in the next experiment.

The effect of enabling hash join
(SMP)

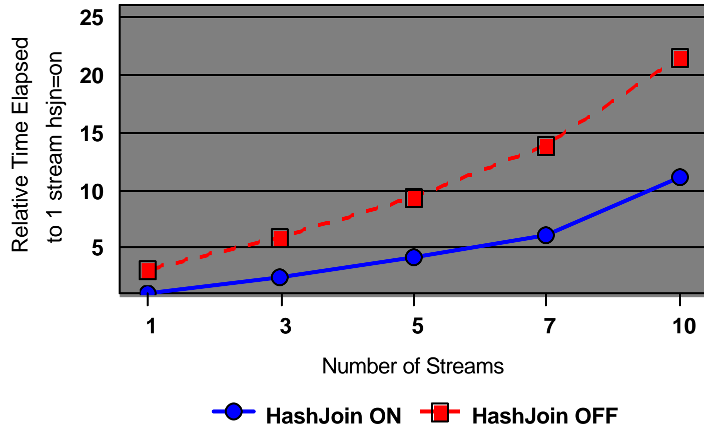

Figure 1. Scalability of hash joins in an SMP system

Relative Time to one stream hsjn=on

Streams	Hsjn ON	Hsjn OFF
1	1.00	3.14
3	2.46	5.87
5	4.23	9.40
7	6.18	13.89
10	11.11	21.59

Total Hash Joins

Streams	Hsjn ON	Hsjn OFF
1	38	n/a
3	114	n/a
5	190	n/a
7	266	n/a
10	380	n/a

Total Hash Loops

Streams	Hsjn ON	Hsjn OFF
1	0	n/a
3	0	n/a
5	0	n/a
7	0	n/a
10	1000	n/a

Hash Join Threshold

Streams	Hsjn ON	Hsjn OFF
1	0	n/a
3	0	n/a
5	0	n/a
7	0	n/a
10	0	n/a

Hash Join Overflows

Streams	Hsjn ON	Hsjn OFF
1	3	n/a
3	24	n/a
5	55	n/a
7	92	n/a

Hash Join Small Overflows

Streams	Hsjn ON	Hsjn OFF
1	0	n/a
3	3	n/a
5	8	n/a
7	15	n/a

Figure 2. Monitor information for SMP experiment

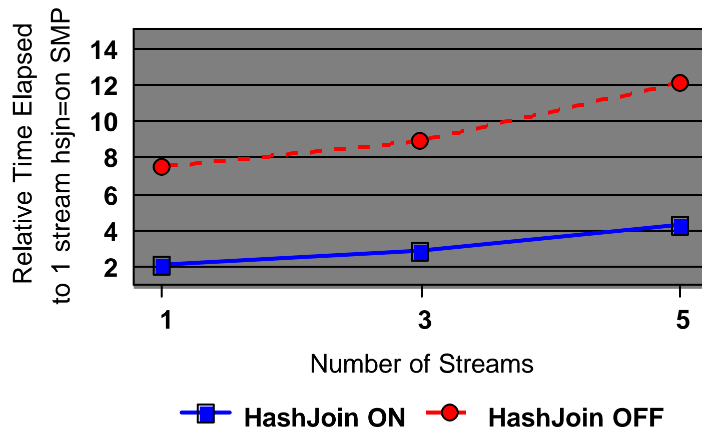

Figure 3. Scalability of hash joins in an uniprocessor system

Relative Time to 1 stream hsjn=on SMP

Streams	Hsjn ON	Hsjn OFF
1	2.20	7.51
3	2.96	8.96
5	4.31	12.11

Total Hash Joins

Streams	Hsjn ON	Hsjn OFF
1	38	n/a
3	114	n/a
5	190	n/a

Total Hash Loops

Streams	Hsjn ON	Hsjn OFF
1	0	n/a
3	0	n/a
5	0	n/a

Hash Join Threshold

Streams	Hsjn ON	Hsjn OFF
1	0	n/a
3	0	n/a
5	0	n/a

Hash Join Overflows

Streams	Hsjn ON	Hsjn OFF
1	8	n/a
3	24	n/a
5	40	n/a

Hash Join Small Overflows

Streams	Hsjn ON	Hsjn OFF
1	1	n/a
3	3	n/a
5	5	n/a

Figure 4. Monitor information for uniprocessor experiment

5.2 Sortheap experiments

As we mentioned before, the tuning of the sortheap and sheapthres parameters are the most important factors in the performance of hash join. The main purpose of this experiment was to understand how different sortheap sizes could affect the performance of hash join. In this test we used all the initial tuning as explained in the previous section and only varied the size of sortheap, while leaving sheapthres at its original value of 192,000. We only ran one stream for these tests. The results are found in Figure 5.

The times shown are relative to the time that it takes to run with a sortheap of 256 pages. We consider that running the workload for a sortheap of 256 pages takes one unit of time. Every other result is expressed relative to that unit for this experiment.

Let's analyze the results in detail. We can see that in general the bigger the sortheap, the less overflow we get and the better the performance. We get the best performance at 40000 pages in which we only see three overflows.

The first question is, why don't even bigger sortheaps give us better performance? If you look into the monitor information shown in Figure 6, you can see that at 50000 pages we started hitting the sheapthres. And the bigger the sortheap, the worse performance we got since we hit the sheapthres more times. The reason we get degradation when we hit sheapthres is that when a sortheap request hits the threshold, DB2 will try to satisfy it but by giving less memory than requested. As we can see in these cases, even though the sortheap we requested was bigger than before, we got more overflows. This is because the sortheap we were getting was in reality less than what we asked for, and this caused overflows, thus reducing the performance of the hash join.

Another thing we notice by looking at the chart is that we got much bigger improvements on the left side of the chart when going from 256 to 500 pages, from 500 to 1000 pages, and all the way to 4000 pages. Why? Looking once more into the monitor information we can see that the number of hash loops decreased significantly by increasing the sortheap until they disappeared at 4000 pages, after which the performance gains of increasing sortheap were much smaller than before.

From these results we can conclude that the worst enemy of the hash joins are the hash loops. Although hitting sheapthres causes degradation, as is seen for sortheap > 40000, the order of magnitude in performance degradation with having hash loops is far worse than hitting sheapthres, and it is important to keep this in mind when it is not possible to avoid both situations.

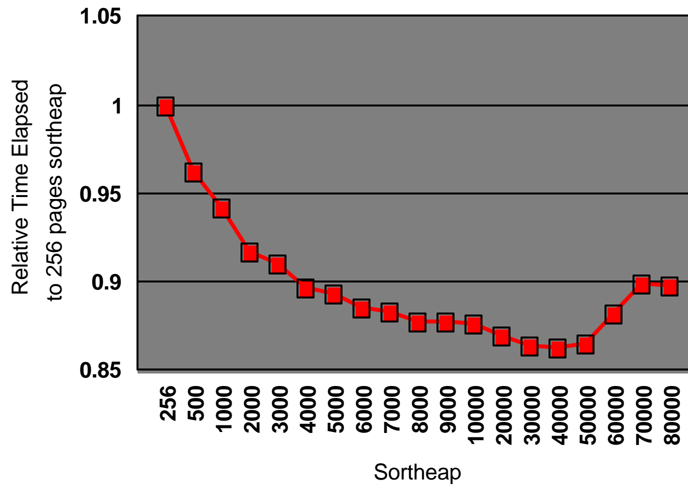

Figure 5. Results of changing the value of sortheap

Sortheap	Relative Time to 256 sortheap	Hash Joins	Hash Loops	Hash Overflows	Hash Small Overflows	Hash Join Threshold
256	1.000	37	750	26	0	46
500	0.962	37	400	25	1	18
1000	0.942	37	268	20	1	3
2000	0.917	37	100	17	0	0
3000	0.910	37	100	15	0	0
4000	0.897	37	0	15	1	0
5000	0.893	37	0	11	0	0
6000	0.886	37	0	11	0	0
7000	0.883	37	0	9	1	0
8000	0.878	37	0	8	0	0
9000	0.878	37	0	8	0	0
10000	0.877	37	0	8	0	0
20000	0.870	37	0	5	0	0
30000	0.864	37	0	4	1	0
40000	0.863	37	0	3	0	0
50000	0.865	37	0	4	1	1
60000	0.882	37	0	4	0	2
70000	0.899	37	0	5	0	3
80000	0.898	37	0	5	0	3

Figure 6. Monitor information for sortheap experiment

5.3 Sheapthres experiments

The main purpose of these experiments was to understand the tuning of sheapthres, the relationship between sortheap and sheapthres, and the impact of that relationship in hash join. For this purpose we divided this experiment in five sets of tests. In each set we fixed the size of the sortheap and changed the sheapthres so that it was twice, three times, four times, five times, six times and seven times the size the sortheap. The five sortheap sizes we used were 5000, 10000, 15000, 20000 and 30000 pages. The results are shown in Figures 7 through 9.

The times shown are relative to the time that it takes to run with a sortheap of 5000 pages and with a sheapthres of 10000 pages (double the sortheap). So, we consider that running the workload for a sortheap of 5000 pages and a sheapthres of 10000 pages takes one unit of time. Every other result is expressed relative to that unit for this experiment.

If we look at Figure 7, we can see that for all sortheaps the best performance for this particular workload is when the sheapthres value is 6 to 7 times the sortheap value. Looking at Figure 8 and at the monitor data in Figure 9, we can see that it is at that level that we get rid of all the hash join threshold hits. In all cases, the best improvement is found when going from double to triple the sortheap value because the hash join threshold decreases significantly at that time. In addition, when sortheap is 5000, hash loops decrease significantly when going from double to triple the size. As the DB2 UDB manuals indicate, the general recommendation is to have sheapthres at least three times the sortheap, but we wanted to see what the results of having sheapthres too small was for hash join. In your real system, we would never recommend having sheapthres lower than three times the sortheap.

Another important thing to observe is that the gains from increasing the sheapthres are in some cases very small; in the overall performance tuning picture of your database/instance, it might not be worth allocating the memory into sheapthres instead of, for example, bigger sortheaps or more bufferpools. Be aware of how you set sheapthres and its impact on the whole system. The most important thing we can conclude from these experiments is that tuning sheapthres has some impact in hash join performance, but that impact is not as significant as tuning sortheap.

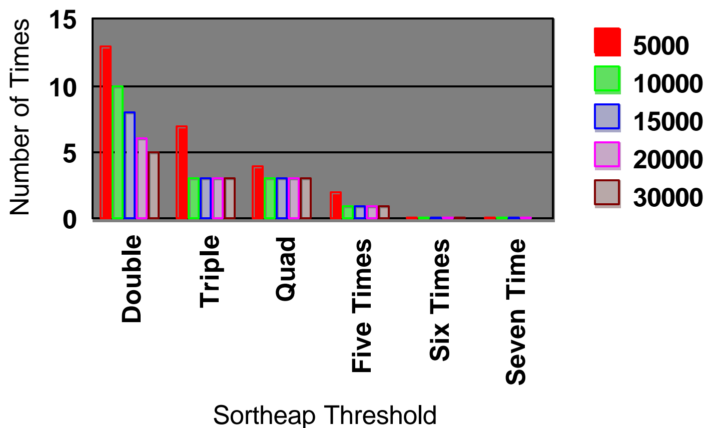

Figure 7. Results of changing the value of sheapthres on elapsed time

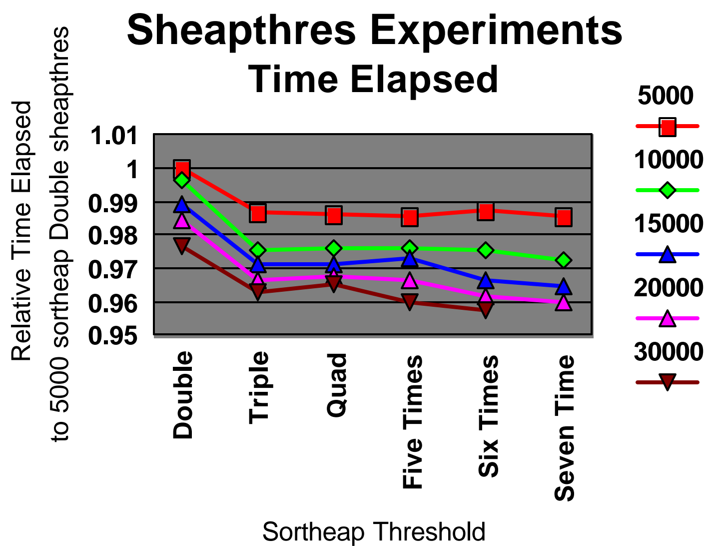

Figure 8. Results of changing the value of sheapthres on hitting the hash join thresholds

Sortheap = 5000

Sheapthres	Relative Time	Hash Joins	Hash Loops	Hash Overflows	Hash Small Overflows	Hash Join Threshold
Double	1.000	37	52	16	0	13
Triple	0.987	37	18	12	0	7
Quad	0.986	37	0	12	0	4
Five Times	0.986	37	0	12	0	2
Six Times	0.988	37	0	11	0	0
Seven Times	0.986	37	0	11	0	0

Sortheap = 10000

Sheapthres	Relative Time	Hash Joins	Hash Loops	Hash Overflows	Hash Small Overflows	Hash Join Threshold
Double	0.997	37	0	10	0	10
Triple	0.976	37	0	8	0	3
Quad	0.976	37	0	8	0	3
Five Times	0.976	37	0	8	0	1
Six Times	0.975	37	0	8	0	0
Seven Times	0.973	37	0	8	0	0

Sortheap = 15000

Sheapthres	Relative Time	Hash Joins	Hash Loops	Hash Overflows	Hash Small Overflows	Hash Join Threshold
Double	0.989	37	0	8	1	8
Triple	0.971	37	0	6	0	3
Quad	0.971	37	0	6	0	3
Five Times	0.973	37	0	6	0	1
Six Times	0.966	37	0	6	1	0
Seven Times	0.965	37	0	6	1	0

Sortheap = 20000

Sheapthres	Relative Time	Hash Joins	Hash Loops	Hash Overflows	Hash Small Overflows	Hash Join Threshold
Double	0.985	37	0	7	0	6
Triple	0.967	37	0	6	0	3
Quad	0.967	37	0	6	0	3
Five Times	0.967	37	0	6	0	1
Six Times	0.962	37	0	5	0	0
Seven Times	0.960	37	0	5	0	0

Sortheap = 30000

Sheapthres	Relative Time	Hash Joins	Hash Loops	Hash Overflows	Hash Small Overflows	Hash Join Threshold
Double	0.977	37	0	6	0	5
Triple	0.963	37	0	5	0	3
Quad	0.965	37	0	5	1	3
Five Times	0.960	37	0	5	2	1
Six Times	0.957	37	0	4	1	0
Seven Times	n/a	n/a	n/a	n/a	n/a	n/a

Figure 9. Monitor information for sheapthres experiment

6. Conclusions

As we have seen, hash join can provide a significant performance improvement with a small amount of tuning. As we have demonstrated in our first experiment, just by following some basic rules of thumb, we have obtained significant improvement in our DSS workload by enabling hash join. Also, we have shown we can improve the scalability when having many streams.

We have also learned that from the two main parameters that affect hash join: sortheap and sheapthres, sortheap seems to be the most important one. By increasing sortheap we can potentially avoid having hash loops, which are the worst enemies of hash join. Also, we saw that the larger the sortheap, the fewer the overflows and the better the performance. This does not mean we should ignore sheapthres, which can definitely affect performance, but in general sheapthres is determined by the amount of memory you have available in the box. We can conclude then that the best way to tune hash join is to determine how much you can afford for sheapthres and then tune the sortheap value accordingly.

The key for tuning hash join is: *Make the sortheap as large as possible to avoid overflows and hash loops, but not so large as to hit sheapthres*.