DB2 Information Management Software

Performance Tuning Tips for IBM Informix Dynamic Server

Author: Bobby Sukumar
Email: bobbys@sg.ibm.com

Date: Dec 18, 2006
Revision: 1.1

Table of Contents

1. General guidelines for tuning new systems 3
2. Understanding and tuning update statistics 7
3. Understanding the SET EXPLAIN output 10
4. Join methods explained 12
5. SELECT statement explained 20
6. Fragmentation considerations 24
7. Tuning the B-Tree Scanner 28
8. Concurrency and Performance 31
9. Real cases from ATLAS/RETAIN systems 36
10. References 41

Target audience for this paper

- Database administrators
- Database SQL developers
- Database users

1.General guidelines for tuning new systems

Because performance tuning is iterative in nature, completing too many changes at one time can be a challenge. As time goes by, you should re-analyze these values and re-configure your settings on a regular basis to maintain the best performance.

In addition, tuning the system in a controlled environment[1] without interference from external influences[2] that could skew I/O and network performance is the key to knowing where you stand and where you have succeeded.

[1] Ideally you would want to have an environment that simulates a production system as realistically as possible, where the required queries are executed with the most number of expected concurrent users.

[2] Multiple database server instances that run on the same host computer perform poorly when compared with a single database server instance that manages multiple databases.

Online Transaction Processing (OLTP)

A typical OLTP workload is characterized by a large number of users performing a high volume of short transactions that INSERT, UPDATE, and DELETE data. When tuning Informix Dynamic Server in an OLTP environment, it is important to spend time in areas that will have the greatest impact on performance.

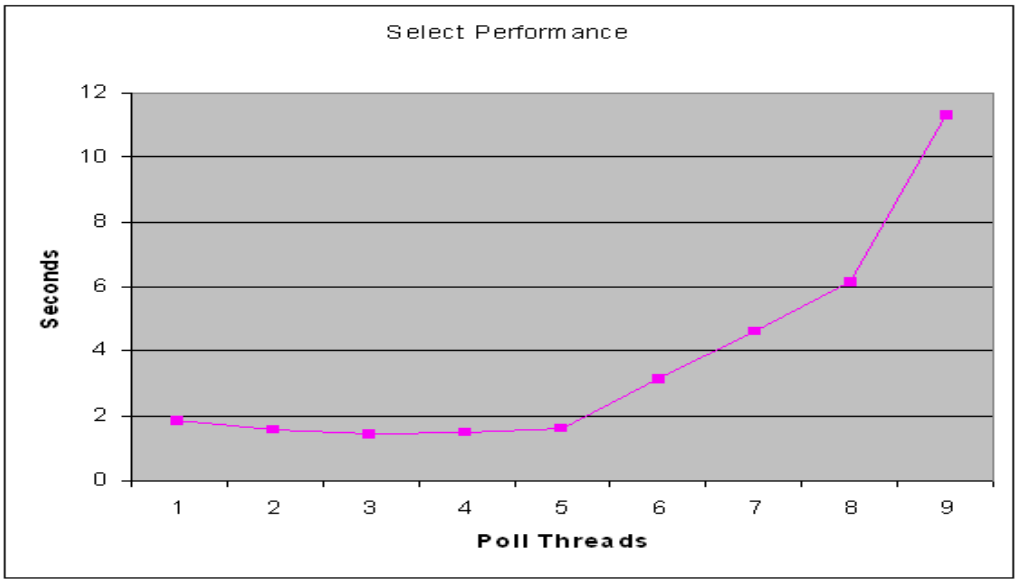
With this in mind, here are some initial ONCONFIG ($INFORMIXDIR/etc/$ONCONFIG) settings for OLTP.

Configuration Parameter	Initial Setting
BUFFERS[1]	Set to between 50 and 75 percent of available free memory. Set to an even greater percentage if you are not using the memory in the virtual portion (after testing to see if the memory is needed). When tuning this parameter and SHMVIRTSIZE, understand that both need to be changed if the combined total is 75 percent of the operating system physical memory.
LOCKS	1000 * number of users
PHYSDBS	Separate from root DBSPACE and place on separate high speed device. Logical log files are to be placed on a separate high speed device as well.
PHYSBUFF	Pages per I/O should be about 75 percent of the physical log buffer size (database uses buffered logging). Monitor with onstat –l
LOGBUFF	Pages per I/O should be about 75 percent of the logical log buffer size (database uses buffered logging). Monitor with onstat -l
LRUS[1]	Four LRU pairs per CPU VP.
CLEANERS	One page cleaner thread per LRU pair.

SHMVIRTSIZE	32000 + expected number of users * 800
CKPTINTVL	Set to 9999. Let the physical log initiate checkpoints. This can be contrary to popular belief, but the reasoning is that when LRU cleaning keeps dirty buffers to a minimum and even with a long interval, you can minimize the checkpoint waits.
LRU_MAX_DIRTY[1]	Set to 10. With many systems, the final settings can be as low as 1. If high transactional throughput is desired, chunk writes are preferred over LRU writes. The percentage of dirty buffer pages in LRU queues at checkpoint time should not be consistently greater than the maximum duration of a full checkpoint. Also with the 9.4 release, you can use fractional percentages when transactional volume is low and the efficiency of the I/O interface high (low latency).
LRU_MIN_DIRTY[1]	Set to 5. With many systems, the final setting can be as low as 0. Also with the 9.4 release, you can use fractional percentages when transaction volume is low and the efficiency of the I/O interface high (low latency).
RA_PAGES	32
RA_THRESHOLD	30
RESIDENT	-1 (Lock all resident and virtual segments on supported operating environments)
NETTYPE	Optimum number of connections per poll thread is 300 users[2] (See graph below)
OPTCOMPIND	0
VPCLASS (AIO)	Number of chunks that can be accessed during peak usage

[1] With the 10.00 release and above, use the BUFFERPOOL configuration parameter in ONCONFIG. It specifies values for LRUS, BUFFERS, LRU_MIN_DIRTY and LRU_MAX_DIRTY, thus eliminating the need to set these individually.

[2] The following graph shows 3 to 4 poll threads are optimal for 900 dynamic server users. The test of the select system call was conducted in a HP-UX 11i operating environment with 8 processors.
It takes one poll thread 1.83s to monitor 900 endpoints and 2 poll threads only 1.55s to do the same workload, each monitoring 450 endpoints. Beyond 4 poll threads, the division of workload becomes a bottleneck.



When tuning the online system for an OLTP environment, focus on the specific areas that will have the greatest impact on performance. What you want to achieve is:

1. High read and write buffer cache rates
2. Fast checkpoints
3. Maximum I/O throughput

From a disk I/O perspective, it is important to place the chunks on raw devices. Preferably a RAID environment which offers maximum performance coupled with data integrity and a battery powered write cache of significant size.

Certain operating system parameters also require to be tuned as they affect CPU utilization. These are:

Semaphore parameters

Semaphores are kernel objects with a typical size of one byte/semaphore. Dynamic server requires one set for each group of up to 150 VPs, one set for each additional VP that is added dynamically, one set for each group of 100 or fewer sessions connected through a shared memory interface.

In addition to the above, for shared memory connections allocate enough semaphores for 2 times the expected shared memory connections. SEMMNI specifies the number of semaphore sets and SEMMSL specifies the maximum number of semaphores per set. SEMMSL should be set to at least 100. Some environments require a maximum number of semaphores across all sets, specified by SEMMNS. This can be calculated using the formula:

SEMMNS = Number of VPs initialized with dynamic server + dynamically added VPs + 2 times the shared memory connections allowed + 6. The last value is the number of database server utilities such as ONSTAT and ONCHECK that may connect concurrently.

Parameters that set the number of open file descriptors

Some operating systems specify this parameter as NFILE, NFILES etc. This kernel object directs affects the growth in chunks and/or connections on the system. This is calculated using the formula:

NFILES = (CHUNKS * NUMAIOVPS) + NUMCPUVPS + Network Connections as specified by NETTYPE (or SQLHOSTS file). These include all connections except IPCSHM connections.

Memory configuration parameters

The configuration of memory in the operating system can affect other resources, including CPU and I/O. Insufficient physical memory leads to excessive paging and buffer management activity.

The third area is the most important of all. Usually this is achieved by eliminating I/O bottlenecks, optimizing the fragmentation strategy (fragmenting indexes pages as well as data pages), and obtaining an optimal index scan in the query plan.

Important note on the OPTCOMPIND Configuration Parameter

For join plans, the ONCONFIG setting of OPTCOMPIND influences the access plan for a specific ordered pair of tables. If you set OPTCOMPIND to 0 (zero) ensures the database server selects a join method as it did in previous versions of the database server, ensuring backward compatibility.

Setting OPTCOMPIND to 0 (zero) also ensures that the optimizer does not choose a Hash Join method over a Nested Loop join method for a multi-table query.

Version 10.00 of Informix Dynamic Server allows you to dynamically set OPTCOMPIND for a session using the following syntax:

SET ENVIRONMENT OPTCOMPIND 0

This setting takes precedence over the ONCONFIG file setting.

Decision Support Systems (DSS)

When tuning for a DSS environment, focus on specific areas that will have the greatest impact on performance. These are:

1. Optimum memory utilization
2. Parallel data queries (PDQ)
3. Light scans
4. Maximum I/O throughput

In general, DSS queries:

- Examine large volumes of data
- Execute with a high degree of complexity
- Give answers to critical business questions
- Are far more complex than most OLTP transactions
- Include a rich breadth of operators and selectivity constraints
- Generate intense activity on the part of the database server
- Are implemented with constraints derived from staying closely synchronized with an on-line production database

For optimum memory utilization, the area that will have the greatest impact is shared memory. Due to the nature of DSS queries, large amounts of shared memory located in the virtual segment are required to perform a variety of operations, such as light scans, hash joins, and sorts. It is critical to properly configure and tune the shared memory and PDQ parameters in the ONCONFIG file.

To increase performance of DSS queries, increase the amount of available virtual shared memory. With this in mind, here are some initial ONCONFIG ($INFORMIXDIR/etc/$ONCONFIG) settings for DSS.

Configuration Parameters	Initial Setting
BUFFERS	2000 (Minimize)
SHMVIRTSIZE	75 percent of available memory (maximize) or higher if memory is not needed elsewhere
SHMADD	32000
SHMTOTAL	Set to available memory for the Informix engine and not to the entire memory on the Unix system
RA_PAGES	128
RA_THRESHOLD	120
DS_TOTAL_MEMORY	90 percent of SHMVIRTSIZE

2. Understanding and tuning update statistics

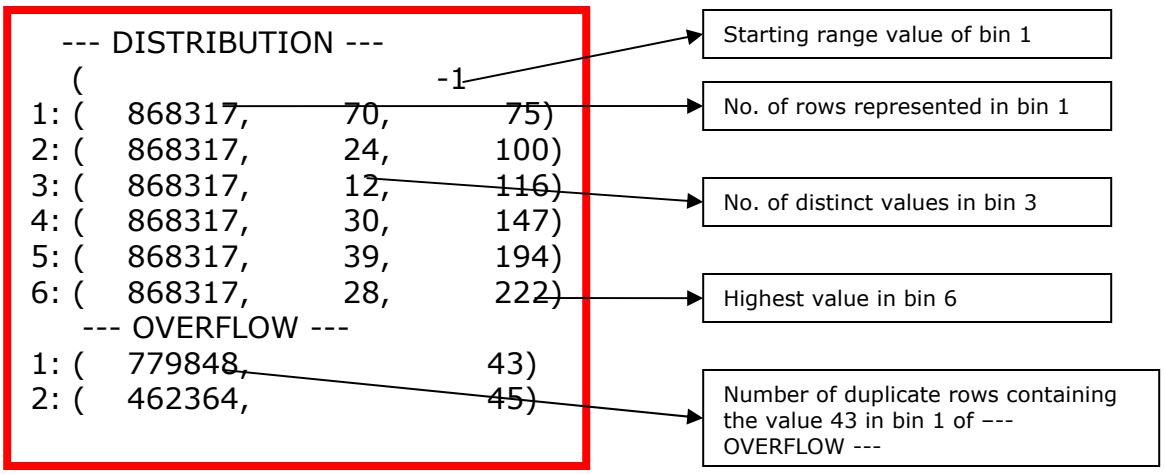
The UPDATE STATISTICS statement primarily collects information for the optimizer. Additionally, depending on usage, the UPDATE STATISTICS statement can determine the distribution of column values, force re-optimization of stored procedures and convert existing indices during a database server upgrade.

What is a distribution?

A distribution is a mapping of the data in a table's (or synonym's) column into a set of column values, *ordered by magnitude* or by collation. The range of these values is partitioned into disjunctive (non-overlapping) intervals called bins. Each bin contains an approximate equal portion of the sample of column values. For example, if one bin holds 2 percent of the data, 50 such intervals hold the entire sample (2 percent resolution).

The following diagram illustrates the histogram generated by the dbschema utility for a table which contains distributions. Distributions are generated by running the update statistics statement in medium or high mode but not low mode[1]. The overflow portion of the output shows the duplicate values that might skew the distribution data.

[1] Low mode does not generate distributions; it only updates systables, sysindexes and syscolumns with relevant values of number of rows, number of pages, number of distinct values of lead index key, number of b-tree index levels, index leaf pages, column min/max (second lowest and second highest value in the table) etc.



The optimizer can use data distributions to calculate how *selective*[1] a given filter in a query is. In the absence of distributions, this information is gathered from table indexes. However selectivity of a filter calculated using data distributions is far more accurate.

[1]Selectivity of a filter is a value between 0 and 1. Selectivity indicates the proportion of rows within the table that the filter can pass. A highly selective filter (one that passes the fewest rows) has a selectivity value close to 0, while a filter with low selectivity (one that passes almost all rows) has a selectivity value close to 1.

Example of how the optimizer approximates a value

Suppose there are 868,317 rows containing a value between -1 (lowest) and 75 (highest) in the first bin (see diagram above), of which only 70 are distinct (unique). By dividing 868317 by 70 (868317 / 70 = 12404), the optimizer deduces there are 12,404 rows containing a value between -1 and 75.

Scope

The scope of the UPDATE STATISTICS statement when not explicitly stated for a particular table or stored procedure covers every single table and SPL routine in the current database, including in the system catalog tables. If the UPDATE STATISTICS statement is executed using the FOR TABLE clause without a table name, distributions are also compiled for all temporary tables in that session. Similarly, if the UPDATE STATISTICS statement is executed using the FOR PROCEDURE/FOR FUNCTION/FOR ROUTINE clause without a procedure/function/routine name, then execution plans are re-optimized for all procedures, functions, and routines in the current database.

Basic algorithm for distributions

- Develop a scan plan based on available resources
- Scan the table
  For UPDATE STATISTICS HIGH – All rows in table
  For UPDATE STATISTICS MEDIUM – Sampling of rows in table based on confidence (number of samples) and resolution (percent of data represented in a bin)
- Sort each column
- Build distributions
- Begin the transaction
  - Delete old column values
  - Insert new column values
- Commit the transaction

Improving the update statistics run performance (version 9.40, 10.00 and above)

- Turn on PDQ (minimum 10) but only when running UPDATE STATISTICS for tables.
- Enable parallel sorting by enabling the environment variable PSORT_NPROCS.
- When running in HIGH or MEDIUM mode, increase the default sort memory by setting the environment variable DBUPSPACE. For example, specify DBUPSPACE=0:35 (35MB memory).
- Change the resolution to 1.5 for MEDIUM mode (increases bins and sample size).
- Run UPDATE STATISTICS for all columns of the table after allocating more memory.

What update statistics statement to run?

The following table summarizes under what scenarios different UPDATE STATISTICS statements are typically run:

When to execute	Update Statistics Statement
The number of rows has changed significantly	UPDATE STATISTICS LOW
You migrated from a previous version of Dynamic Server	UPDATE STATISTICS LOW DROP DISTRIBUTIONS then UPDATE STATISTICS MEDIUM/HIGH FOR TABLE
The columns in a table are not represented by an index or is not the leading column of an index	UPDATE STATISTICS LOW
The columns are used in a join or as a filter in the where clause of a query not represented by an index	UPDATE STATISTICS MEDIUM FOR TABLE
The columns are used in a join or as a filter in the where clause of a query represented by a single-column index	UPDATE STATISTICS HIGH FOR TABLE (index column)
The columns are used in a join or as a filter in the where clause of a query represented by a multi-column index	UPDATE STATISTICS HIGH FOR TABLE (first differing index column)
The columns are used in a join or as a filter in the where clause of a query represented by a multi-column index	UPDATE STATISTICS LOW FOR TABLE (all index columns)
The columns are in a small table (dozen pages)	UPDATE STATISTICS HIGH FOR TABLE
Stored Procedure Routines	UPDATE STATISTICS FOR PROCEDURE[1]

[1] When a stored procedure's statistics are updated, the database server stores the optimized query execution plan and the dependency list in the SYSPROCPLAN system catalog table, for use by other processes. A dependency list keeps track of changes that would cause re-optimization the next time that an SPL routine executes. In addition to this, running UPDATE STATISTICS for a procedure can be used to display the query execution plan for an SPL routine using "SET EXPLAIN ON" prior to running the UPDATE STATISTICS statement.

3. Understanding the SET EXPLAIN output

The SET EXPLAIN statement is executed to:

- Display the query execution plan (or query plan) generated by the cost based optimizer.
- Estimate the number of rows returned.
- Estimate the relative cost of the query.

The explain output file name, by default, is "sqexplain.out". Dynamic server writes this file to the current directory where the statement was executed.

Since SQL is declarative, you typically have a large number of alternate ways to execute a given query with widely varying performance. The optimizer evaluates some of the different, correct possible plans for executing the query and returns what the optimizer considers as the best alternative.

Query execution plans are very important tools in tuning the performance of a given query. If you review the query plan, you might be able to see where new indexes might fit or where indexes should be changed. You can also determine if the database server is not fully using existing indexes.

A sample SQEXPLAIN.OUT file

```
QUERY:
------
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C, ORDERS O
WHERE C.CUSTOMER_NUM = O.CUSTOMER_NUM
AND C.LNAME = 'Watson'


Estimated Cost: 5
Estimated # of Rows Returned: 2

 1) informix.c: SEQUENTIAL SCAN
      Filters: informix.c.lname = 'Watson'

 2) informix.o: INDEX PATH
  (1) Index Keys: customer_num (Serial, fragments: ALL)
      Lower Index Filter: informix.c.customer_num = informix.o.customer_num
NESTED LOOP JOIN
```

By examining the SET EXPLAIN output file, you can take steps to improve the performance of the query. The following table shows the information that appears in the output.

Output Field	Description
QUERY	Displays the executed query.
DIRECTIVES FOLLOWED	Lists the optimizer directives set for this query.
ESTIMATED COST	An estimate of the amount of work for the query. The optimizer uses this value to compare the cost of one path with another. The value is assigned based on the selected access method. This number does not directly translate into time, but it can be used to compare

	changes made to the same query. When data distributions are used, a query with a higher estimate generally takes longer to run.
ESTIMATED # OF ROWS RETURNED	This value is derived from information stored in the system catalog tables. For singleton selects, usually this value matches the actual rows returned by the query.
TEMPORARY FILES REQUIRED FOR	If temporary tables are required for an ORDER BY or GROUP BY clause in the query.
NUMBERED LIST OF TABLES	The order in which tables are accessed, followed by the access method used – Index path or Sequential scan.
NUMBERED LIST OF INDEX KEYS	The columns used as filters or indexes – indicated by column name. The notation (*Key Only*) indicates that all desired elements are part of the index key, so a read of the table is unnecessary. The notation (*Key First*) indicates usage of keys other than those listed as *Lower/Upper Index Filter*s. *Lower Index Filter* shows the key value where the index read begins. If the filter condition contains more than one value, an *Upper Index Filter* is shown for the key value where the index read stops.
JOIN PLAN (Algorithm)	When the query involves a join between two tables, the join method the optimizer used (*Nested Loop or Dynamic Hash*) is shown at the bottom of the output for that query. During a nested loop join, the outer table is listed first for each join-table pair. During a dynamic hash join, if the output contains the words *Build Outer*, the hash table is built on the first table listed (build table), otherwise the hash table is built on the second table listed. A *Semi Join* is a variation of the Nested Loop join where the inner table scan is halted when the first match is found.
FRAGMENT NUMBERS/ALL/NONE	The fragment numbers are the same as those recorded in the "partn" column of sysfragments. When *ALL* (meaning scan all fragments) is displayed, no fragment elimination has occurred. When *NONE* (meaning none of the fragments contain the queried information) is displayed, all fragments were eliminated.
ACCESS PLAN	SEQUENTIAL SCAN: Read rows in sequence. INDEX PATH: Scans one more indexes. AUTOINDEX PATH: Creates a temporary index. REMOTE PATH: Accesses another table (distributed join). A *First Row* scan is a variation of a table scan. The table scan is halted when the first match is found. A *Skip Duplicate Index* scan is a variation of an index scan to skip duplicate values.

4.Join methods explained

A join combines rows from two or more tables (relations). *Mathematically, a join is a relational composition*. The three possible types of joins are:

- Inner
- Outer
- Cross

Outer joins are further classified into:

- Left outer
- Right outer
- Full outer[1]

The inputs to a join are referred to as the *outer* and *inner* join operands, or *left* and *right* join operands respectively.

[1]Join syntax that uses the key words right outer, full outer and cross join in a SQL statement are only available in Informix Dynamic Server Versions 9.40 and 10.00 and above.

For performance reasons, you should avoid cross joins because they return the Cartesian product of the rows from the joined tables (all rows in table 1 x all rows in table 2).
In an example of a *Cross Join*, there is 1 row in the CUSTOMER table (using the filter C.LNAME = 'Watson') and there are 23 rows in the ORDERS table.

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C, ORDERS O
WHERE C.LNAME = 'Watson';
```

The resulting join is a Cartesian product that returns 23 rows (1 x 23) due to the absence of a join filter[2].

[2]A join filter is a conditional expression used in the WHERE clause of a query (non-standard syntax) or the ON clause of a query (standard ANSI-92 syntax).

```
CUSTOMER_NUM LNAME           FNAME          PHONE          ORDER_DATE
         106 Watson          George         415-389-8789   05/20/1998
         106 Watson          George         415-389-8789   05/21/1998
         106 Watson          George         415-389-8789   05/22/1998
         106 Watson          George         415-389-8789   05/22/1998
         106 Watson          George         415-389-8789   05/24/1998
         106 Watson          George         415-389-8789   05/30/1998
         106 Watson          George         415-389-8789   05/31/1998
         106 Watson          George         415-389-8789   06/07/1998
         106 Watson          George         415-389-8789   06/14/1998
         106 Watson          George         415-389-8789   06/17/1998
         106 Watson          George         415-389-8789   06/18/1998
         106 Watson          George         415-389-8789   06/18/1998
         106 Watson          George         415-389-8789   06/22/1998
         106 Watson          George         415-389-8789   06/25/1998
         106 Watson          George         415-389-8789   06/27/1998
         106 Watson          George         415-389-8789   06/29/1998
         106 Watson          George         415-389-8789   07/09/1998
         106 Watson          George         415-389-8789   07/10/1998
         106 Watson          George         415-389-8789   07/11/1998
         106 Watson          George         415-389-8789   07/11/1998
         106 Watson          George         415-389-8789   07/23/1998
         106 Watson          George         415-389-8789   07/24/1998
         106 Watson          George         415-389-8789   07/24/1998

23 row(s) retrieved.
```

Now consider the following example of an *Inner Join* (the database server default):

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C, ORDERS O
WHERE C.CUSTOMER_NUM = O.CUSTOMER_NUM
AND C.LNAME = 'Watson';
```

Or its equivalent in ANSI-92 syntax:

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C INNER JOIN ORDERS O
ON C.CUSTOMER_NUM = O.CUSTOMER_NUM
AND C.LNAME = 'Watson';
```

If the CUSTOMER is joined with the ORDERS table as in the example above, the single scan on the ORDERS table results in two rows (after the join filter is applied) for the 1 row from the CUSTOMER table where C.LNAME = 'Watson'. Thus, a total of two rows would result from the Inner Join (1 x 2).

```
CUSTOMER_NUM LNAME           FNAME          PHONE          ORDER_DATE
         106 Watson          George         415-389-8789   05/22/1998
         106 Watson          George         415-389-8789   06/25/1998

2 row(s) retrieved.
```

A *left outer join* is very different from an inner join. Instead of limiting results to those in both tables, it limits results to those in the *left* table (CUSTOMER table). This means that if the ON clause matches zero rows in the ORDERS table, a row in the result will still be

returned, but with NULL values for each column from the ORDERS table. For example in the SQL statement in ANSI-92 syntax below, CUSTOMER_NUM = 106 matches two rows in the ORDERS table, but the result set contains all of the customers from the CUSTOMER table (and the two matched rows) with NULL values for each unmatched column from ORDERS.

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C LEFT OUTER JOIN ORDERS O
ON C.CUSTOMER_NUM = O.CUSTOMER_NUM
AND C.LNAME = 'Watson';
```

CUSTOMER_NUM	LNAME	FNAME	PHONE	ORDER_DATE
101	Pauli	Ludwig	408-789-8075	<NULL>
102	Sadler	Carole	415-822-1289	<NULL>
103	Currie	Philip	415-328-4543	<NULL>
104	Higgins	Anthony	415-368-1100	<NULL>
105	Vector	Raymond	415-776-3249	<NULL>
106	**Watson**	**George**	**415-389-8789**	**05/22/1998**
106	**Watson**	**George**	**415-389-8789**	**06/25/1998**
107	Ream	Charles	415-356-9876	<NULL>
108	Quinn	Donald	415-544-8729	<NULL>
109	Miller	Jane	408-723-8789	<NULL>
110	Jaeger	Roy	415-743-3611	<NULL>
111	Keyes	Frances	408-277-7245	<NULL>
112	Lawson	Margaret	415-887-7235	<NULL>
113	Beatty	Lana	415-356-9982	<NULL>
114	Albertson	Frank	415-886-6677	<NULL>
115	Grant	Alfred	415-356-1123	<NULL>
116	Parmelee	Jean	415-534-8822	<NULL>
117	Sipes	Arnold	415-245-4578	<NULL>
118	Baxter	Dick	415-655-0011	<NULL>
119	Shorter	Bob	609-663-6079	<NULL>
120	Jewell	Fred	602-265-8754	<NULL>
121	Wallack	Jason	302-366-7511	<NULL>
122	O'Brian	Cathy	609-342-0054	<NULL>
123	Hanlon	Marvin	904-823-4239	<NULL>
124	Putnum	Chris	918-355-2074	<NULL>
125	Henry	James	617-232-4159	<NULL>
126	Neelie	Eileen	303-936-7731	<NULL>
127	Satifer	Kim	312-944-5691	<NULL>
128	Lessor	Frank	602-533-1817	<NULL>

29 row(s) retrieved.

In the following example, a NOT IN sub-query can be re-written to perform a more efficient join by testing for unmatched columns (NULLS) in the ORDERS table:

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE
FROM CUSTOMER C
WHERE C.CUSTOMER_NUM NOT IN (SELECT O.CUSTOMER_NUM FROM ORDERS O);
```

Can be re-written to use a left outer join and a test for NULL on the ORDERS table

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE
FROM CUSTOMER C LEFT OUTER JOIN ORDERS O ON C.CUSTOMER_NUM =
O.CUSTOMER_NUM WHERE O.CUSTOMER_NUM IS NULL;
```

A *right outer join* is similar to a left outer join, but the tables are reversed. Here, the results are limited to the rows in the *right* table (ORDERS table). If the ON clause matches zero (0) records in the CUSTOMER table, a row in the result will still be returned, but with NULL values for each column from the CUSTOMER table. For example, in the SQL statement in ANSI-92 syntax below, CUSTOMER_NUM = 106 matches two rows in the ORDERS table, but the result set contains all of the orders from the ORDERS table (and the two matched rows) with NULL values for each unmatched column from the CUSTOMER table.

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C RIGHT OUTER JOIN ORDERS O
ON C.CUSTOMER_NUM = O.CUSTOMER_NUM
AND C.LNAME = 'Watson';
```

CUSTOMER_NUM	LNAME	FNAME	PHONE	ORDER_DATE
<NULL>	<NULL>	<NULL>	<NULL>	05/20/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/21/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/22/1998
106	**Watson**	**George**	**415-389-8789**	**05/22/1998**
<NULL>	<NULL>	<NULL>	<NULL>	05/24/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/30/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/31/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/07/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/14/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/17/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/18/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/18/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/22/1998
106	**Watson**	**George**	**415-389-8789**	**06/25/1998**
<NULL>	<NULL>	<NULL>	<NULL>	06/27/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/29/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/09/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/10/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/11/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/11/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/23/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/24/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/24/1998

```
23 row(s) retrieved.
```

A *full outer join* combines the results of both left and right outer joins. The results are not limited to either the *left table* (CUSTOMER table) or the *right table* (ORDERS table). This means that if the ON clause matches zero (0) records in the CUSTOMER table, a row in the result will still be returned, but with NULL values for each column from the CUSTOMER table as well as from the ORDERS table. For example, in the SQL statement in ANSI-92 syntax below, CUSTOMER_NUM = 106 matches two rows in the ORDERS table, but the result set contains all the orders from the ORDERS table (and the two matched rows) with NULL values for each unmatched column from the CUSTOMER and ORDERS tables.

```
SELECT C.CUSTOMER_NUM, C.LNAME, C.FNAME, C.PHONE, O.ORDER_DATE
FROM CUSTOMER C FULL OUTER JOIN ORDERS O
ON C.CUSTOMER_NUM = O.CUSTOMER_NUM AND C.LNAME = 'Watson';
```

CUSTOMER_NUM	LNAME	FNAME	PHONE	ORDER_DATE
101	Pauli	Ludwig	408-789-8075	<NULL>
102	Sadler	Carole	415-822-1289	<NULL>
103	Currie	Philip	415-328-4543	<NULL>
104	Higgins	Anthony	415-368-1100	<NULL>
105	Vector	Raymond	415-776-3249	<NULL>
106	**Watson**	**George**	**415-389-8789**	**05/22/1998**
106	**Watson**	**George**	**415-389-8789**	**06/25/1998**
107	Ream	Charles	415-356-9876	<NULL>
108	Quinn	Donald	415-544-8729	<NULL>
109	Miller	Jane	408-723-8789	<NULL>
110	Jaeger	Roy	415-743-3611	<NULL>
111	Keyes	Frances	408-277-7245	<NULL>
112	Lawson	Margaret	415-887-7235	<NULL>
113	Beatty	Lana	415-356-9982	<NULL>
114	Albertson	Frank	415-886-6677	<NULL>
115	Grant	Alfred	415-356-1123	<NULL>
116	Parmelee	Jean	415-534-8822	<NULL>
117	Sipes	Arnold	415-245-4578	<NULL>
118	Baxter	Dick	415-655-0011	<NULL>
119	Shorter	Bob	609-663-6079	<NULL>
120	Jewell	Fred	602-265-8754	<NULL>
121	Wallack	Jason	302-366-7511	<NULL>
122	O'Brian	Cathy	609-342-0054	<NULL>
123	Hanlon	Marvin	904-823-4239	<NULL>
124	Putnum	Chris	918-355-2074	<NULL>
125	Henry	James	617-232-4159	<NULL>
126	Neelie	Eileen	303-936-7731	<NULL>
127	Satifer	Kim	312-944-5691	<NULL>
128	Lessor	Frank	602-533-1817	<NULL>
<NULL>	<NULL>	<NULL>	<NULL>	05/20/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/21/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/22/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/24/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/30/1998
<NULL>	<NULL>	<NULL>	<NULL>	05/31/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/07/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/14/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/17/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/18/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/18/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/22/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/27/1998
<NULL>	<NULL>	<NULL>	<NULL>	06/29/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/09/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/10/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/11/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/11/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/23/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/24/1998
<NULL>	<NULL>	<NULL>	<NULL>	07/24/1998

50 row(s) retrieved.

Because joins are both *commutative* as well as *associative*, the order in which the tables are joined does not change the final result of the query. However join order does have an enormous impact on the cost of the join operation. So choosing the best join order is very important.

In mathematical terms a binary operation f: A × A —› B is said to be commutative when for any y and any z in A, f (y, z) = f (z, y) where y and z are real numbers. For example, 2 x 3 = 3 x 2 since both expressions evaluate to 6.

Similarly a binary operation is termed associative if the order of evaluation is immaterial even if the operation appears more than once. For example, 2 + (3 + 1) = (2 + 3) + 1 since both expressions evaluate to 6. It is represented mathematically as (x * y) * z = x * (y * z) where x, y, z are real numbers.

Query plans involving joins can be classified as:

- Left Deep – The inner (right) operand of each join in the plan is a base table (rather than another join).
- Right Deep – The outer (left) operand of each join in the plan is a base table.
- Bushy – neither left-deep nor right-deep; both inputs to a join might be joins themselves.

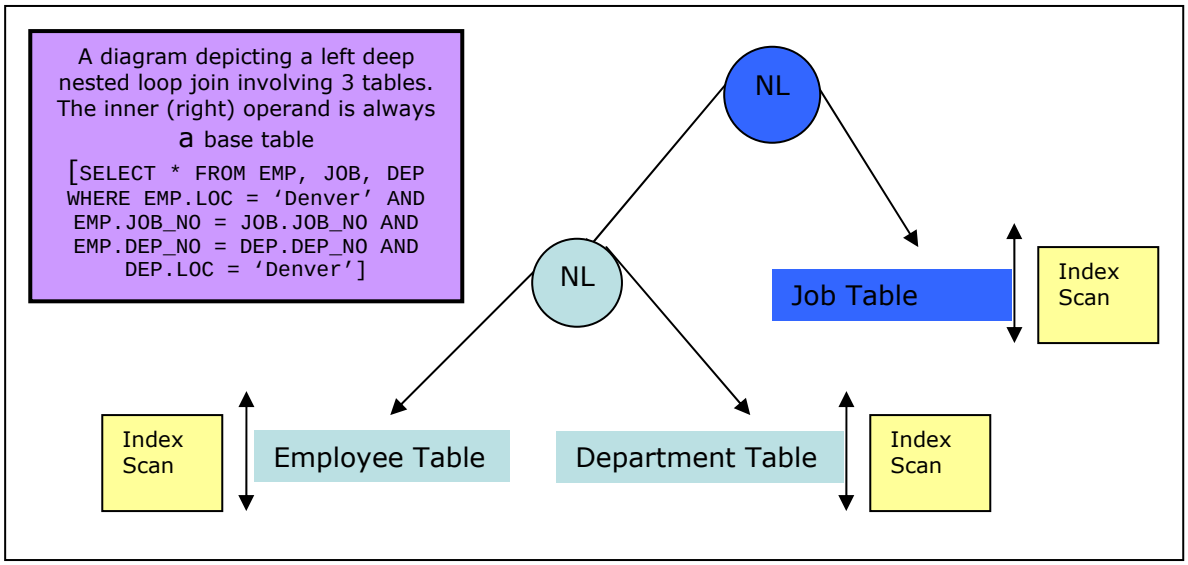
These names are derived from the appearance of the query plan drawn as a tree, with the outer join relation on the left and the inner relation on the right.

The basic problem of a join algorithm is to find, for each distinct value of the join filter or join key (which is C.CUSTOMER_NUM = O.CUSTOMER_NUM), the set of rows from each table that contains that value. The way the optimizer chooses to join the two tables is called a *Join Plan*. The fundamental algorithm used in the join plan can be a *Nested Loop Join*, *Hash Join*, *Semi Join or a Sort-Merge Join*.

Nested Loop Join

In a *Nested Loop Join* (performing the default Inner Join) the database server scans the left most (outer) table and then joins each of the rows that pass the filter, to the rows scanned from the right most (inner) table. From the above example, the outer table is CUSTOMER and the inner table is ORDERS. The steps taken are:

1. The database server scans the outer table (CUSTOMER) using a full table scan or an index scan and the following additional filter (C.LNAME = 'Watson').
2. Then for each row that satisfies the filter (C.CUSTOMER_NUM = O.CUSTOMER_NUM) on the outer table, the server reads the inner table (ORDERS) to find a match. This access is usually accomplished using an index lookup due to the potentially large number of times this action needs to be executed. If the inner table does not have an index, the database server might construct an auto-index at the time of query execution (if the cost to construct an auto-index is cheaper than the cost to scan the inner table for every qualifying row in the outer table).



Semi Join

If the optimizer changes a sub-query to a Nested Loop Join, the optimizer might use a variation of the Nested Loop Join called a *Semi Join*. In this situation, the server stops the inner table scan when a match is obtained. So for every row in the outer table, the inner table contributes at most one row.

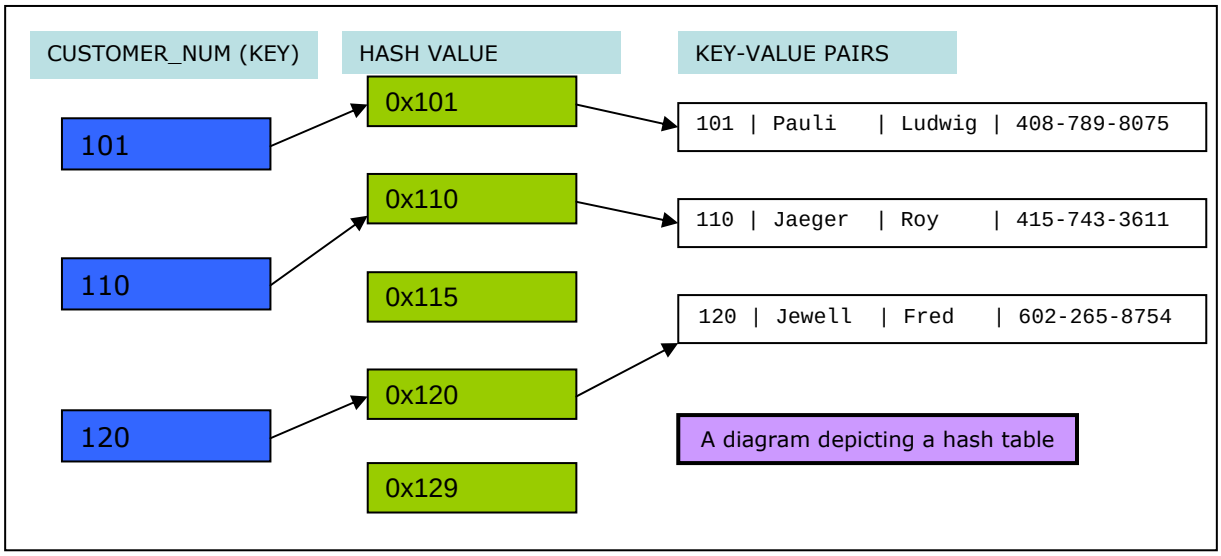
This can reduce the size of the intermediate results and is therefore a useful optimization technique in a distributed environment, where tables from multiple remote database servers are involved.

Hash Join

The optimizer usually employs a *Hash Join* algorithm (performing the Inner Join) only when at least one of the two join tables do not have an index on the join column, or when the database server must read a large number of rows from both tables. During a Hash Join, no indexing or sorting is required. A Hash Join consists of two activities:

1. Building the hash table (*Build Phase*)

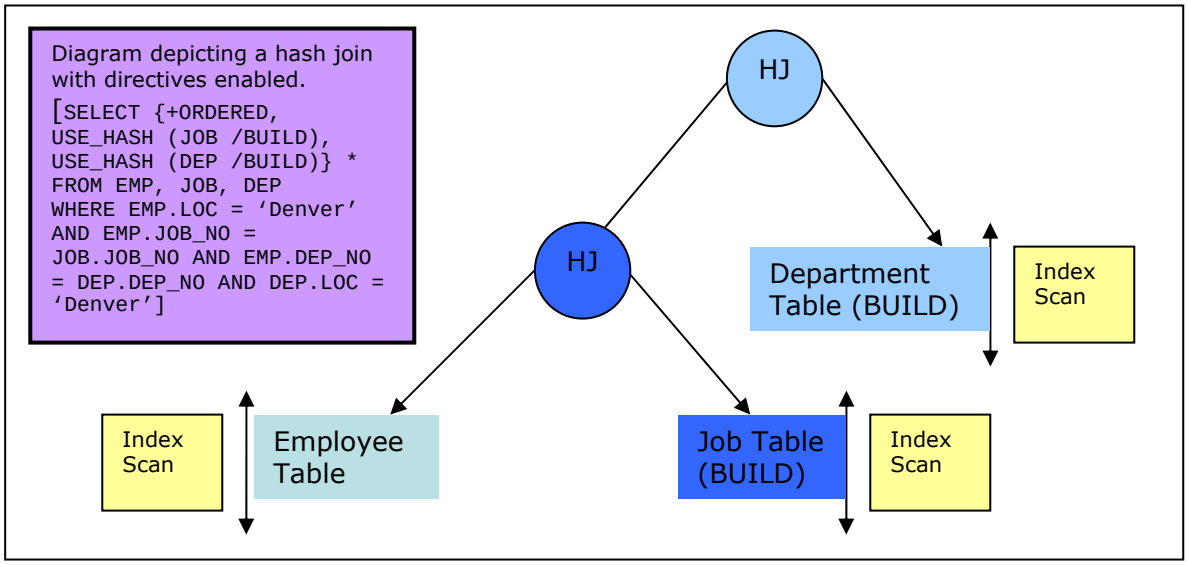
   The *hash table*, a data structure that associates keys with values enabling efficient lookup, is built on the smaller of the two tables, by applying a *hash function* to the join key of each filtered row. Hash joins require an "equi-join" predicate (predicates that use the "=" operator). The hash table is conceptually a series of buckets with each bucket assigned an address or *hash value* (binary data written in hexadecimal notation) that is derived from the join key of each row that satisfies the filter condition.



2. Scanning the hash table (*Probe Phase*)

   In the probe phase, the database server reads the other table (the larger of the two tables) in the join and applies any filters. For each row that satisfies the filter condition, the server scans (probes) the hash table for the matching rows.

Note: It is possible to employ a hash join, where the two tables being joined have a filter condition which does not produce relative rows (for example, using the NOT IN operator). This is achieved by first building a hash table with the NOT IN side of the join. Then scan the other table only selecting rows in which the filter condition hashes to an empty entry in the hash table.



Sort-Merge Join

The key idea of the Sort-Merge algorithm is to first sort the tables by the join attribute, so that interleaved linear scans will encounter these groups (sets) at the same time. Once sorting is complete, the join can be performed.

For each row in the outer table, consider the current group of rows from the inner table. (A group consists of a set of contiguous rows in the inner table with the same value in the join filter).

For each matching row in the current inner group, add a row to the join result. Once the inner group has been exhausted, both the inner and outer scans can be advanced to the next group.

5. SELECT statement explained

A SELECT statement in SQL, which is collectively termed a DML (Data Manipulation Language) statement, returns a result set of rows from one or more tables. In this statement, you specify a description of the desired result set, but do not specify what physical operations must be executed to produce the result set. Translating the query into optimal query execution plans is left to the query optimizer.

Commonly available keywords that must appear in the following order, related to SELECT include:

- FROM – Used to identify a table, view or synonym
- WHERE – Used to identify the rows to be retrieved or applied to GROUP BY
- GROUP BY – Used to combine rows with related values into distinct elements of a smaller set
- HAVING – Used to identify which rows followed by GROUP BY are to be retrieved
- ORDER BY – Used to identify which columns are used to sort the resulting data
- INTO TEMP – Used to temporarily save the results of a multi-table query in a separate table which can be queried or manipulated without modifications to the database schema. This table is termed *temporary* because it is automatically dropped when the session or the program terminates.

If you have a table called CUSTOMER, the query SELECT * FROM CUSTOMER will show all the columns of all the table rows.

With the same table, the query SELECT LNAME, FNAME FROM CUSTOMER will show the elements from the column LNAME and FNAME of all the table rows — in relational algebraic terms, the query will perform a *projection*.

With the same table, the query SELECT * FROM CUSTOMER WHERE CUSTOMER_NUM = 101 will result in all the elements of all the rows where the value of column CUSTOMER_NUM is equal to '101' — in relational algebraic terms, a *selection* will be performed, because of the WHERE keyword.

With the same table, the query SELECT * FROM CUSTOMER ORDER BY CUSTOMER_NUM DESC will show the same rows as the first query; however the results will be in reverse sort order because the ORDER BY keyword uses CUSTOMER_NUM as a sorting point. This query does not have a WHERE keyword, so anything and everything will be returned. Multiple ORDER BY items can be specified separated by comma (for example, ORDER BY CUSTOMER_NUM ASC, LNAME DESC) to further refine sorting.

Response Time Effects

Sometimes you can improve the performance of a SELECT statement by modifying the query construction. The following time costs can be reduced by optimizing query

construction and/or adding or removing appropriate indexes or tweaking certain configuration parameters.

Sort Time

A sort requires in-memory work as well as disk work. The in-memory work depends on the number of columns that are sorted, the width of the combined sort key and the number of rows that pass the filter in the where clause. The disk work depends on the number of disk pages where rows are placed, the number of rows that pass the filter in the where clause, the number of rows that can be placed on a sorted page, and the number of merge operations that must be performed. Hence the more specific the filter, the fewer the rows that are sorted. To reduce the cost of sorts, the filters must be as selective (restrictive) as possible. Also the list of columns in the select list (columns that are projected) should be made as relevant as possible, for example, by discarding unnecessary columns.

If repeated sorts on a large table is required, the optimizer avoids a sort step whenever it can use an index to produce the output in the proper order. However the following factors prevent the optimizer from using an index:

a) One or more of the ordered columns is not included in the index key.
b) The columns are named in a different sequence in the ORDER BY / GROUP BY clause when compared to the index key.
c) The ordered columns are taken from different tables.

In addition, temporary tables can be used to reduce the sorting scope. Building a temporary, ordered subset of a large table can speed up a SELECT statement.

Multiple sort threads (using the PSORT_NPROCS environment variable) should be started, so that sorting can proceed in parallel.

Data Mismatches

A SELECT statement can encounter additional costs when the data type of a column that is used in a condition differs from the definition of the column in the CREATE TABLE statement. This is most severe when the query compares a character value with a non-character value and the length of the non-character value is not equal to that of the character value. For example, a column in a table contains character values '001', ' 1' and '1'. If this is compared against integer 1 [for example, WHERE col_name = 1], all three rows having values '001', ' 1' and '1' are returned, resulting in a sequential scan, even if an index is present. Conversely, if the column is defined as an integer in the CREATE TABLE statement, and the filter condition of the WHERE clause employs a character value, then the engine needs to rewrite this query to perform a data conversion from character to integer, even if this type conversion has no noticeable overhead.

View Costs

Views are normally written to hide the complexities of the underlying query or to limit the data that a user can access. However a SELECT statement that employs a view rather than the base table might run more slowly than expected, especially if the complexity of the query (due to the use of ANSI joins) requires the creation of a temporary table. This temporary table is referred to as a *materialized view*.

Index Lookup Costs

Additional costs are incurred when an index is looked up in order to fetch the data rows. Since the index is stored on disk, the server must read its pages into memory

along with the corresponding data rows. The index traversal starts from the root node downwards to the leaf node.

The pages that hold the root node are almost always contained in page buffers, but the odds of finding a leaf node in page buffers depend on the size of the index, the structure of the query, and the frequency of column value duplication. If many duplicate values for an associated index key value exist, then multiple random accesses (hence more disk I/O operations) must be made to fetch the associated data row. In this scenario, removing the index on such a column might mean a lower time cost for the query.

Non-Sequential Access Cost

Depending on the relative ordering of the table data with respect to the index, sometimes pages containing several needed rows can be retrieved. The degree to which the physical ordering of a table's data rows on disk correspond to the table's index entries is called *clustering*[1]. A highly clustered table is one whose physical ordering on disk corresponds closely to that of its index. Sorting costs can be avoided when a table is clustered this way.

[1]Clustering a table involves the creation of a an index and the physical re-ordering of data rows by copying all the rows to a new table in the order specified by the column(s) on which the cluster index was created. This new table then replaces the old table. Even after this, any new rows inserted into a clustered table are not automatically re-ordered, but are stored physically at the end of the table regardless of their contents. Only upon re-clustering are the rows re-ordered again.

Sequential Access Cost

Disk costs as well as *access time*[2] are lowest when the database engine reads the rows of a table in physical order. When the first row on a page is requested, the disk page is read into a buffer page. Once the page is read, the server does not need to read it again. The server fulfills requests for subsequent rows on that page by reading from the buffer until all the rows on that page are processed. To make sure that the next page is ready in memory a read-ahead method can be employed. You can use the RA_PAGES and RA_THRESHOLD configuration parameters to specify criteria for this method.

Usually when unbuffered devices or raw devices are used to locate individual dbspaces and the table is organized properly with very few table extents, the disk pages of consecutive rows are also placed contiguously on disk. This arrangement ensures that the access arm does not move very much during a sequential read operation.

[2]Access time is the time taken for the access arm of a disk drive to reach the desired track, including the delay in the rotation of the disk (also called latency) to bring the required sector under the read-write mechanism. The average values for this rotational delay is a few milliseconds.

Network Access Cost

Moving data over a network imposes delays in addition to those encountered with direct disk access. Network delays can occur if an application (client) executes a select statement (or another DML statement) across the network to a database server on a different computer. Data sent over the network consists of command messages and buffer sized blocks of row data from one or more tables. If the network is busy, the client must wait its turn to transmit. Such delays are usually in milliseconds. If the network is congested, transmission delays rise exponentially to tenths of a second and more in both directions. When more than one client is involved, these delays can increase from milliseconds to seconds. In addition a delay between multiple SELECT

statements decreases the likelihood of a data page remaining in the page buffer. This makes network access costs highly variable.

For a distributed query, the optimizer estimates the cost of a SELECT statement to be higher because the estimate includes the cost of retrieving the row from disk and transmitting it across the network.

Network access costs in general cannot be lowered, but you can improve query performance:

- by multiplexing concurrent client connections using Max Connect, or
- in the case of distributed queries by reducing the size and number of data buffers using the FET_BUF_SIZE environment variable on the client.

The following example shows how the environment variables FET_BUF_SIZE, OPTOFC, and IFX_AUTOFREE influence the message traffic over a network:

```
PREPARE mystmt FROM "SELECT * FROM EMPLOYEE WHERE EMP_NUM > ?"
DECLARE mycur CURSOR FOR mystmt
OPEN mycur USING :myvar
        FETCH mycur INTO :mylname, :myfname
CLOSE mycur
FREE mycur
FREE mystmt
```

OPTIONS	SIZE OF OUPUT TUPLE	SIZE OF TUPLE BUFFER [1]	NUMBER OF NETWORK MESSAGES
NONE	60 KB	4096	40
FET_BUF_SIZE=16384	60 KB	16384	16
FET_BUF_SIZE=16384 OPTOFC=1	60 KB	16384	12
FET_BUF_SIZE=16384 OPTOFC=1 IFX_AUTOFREE=1	60 KB	16384	10

[1] The benefit of having a large tuple buffer diminishes, as size of the tuple buffer exceeds the average result set size.

Small Table Costs

A table is small if it occupies so few pages that it can be retained entirely in page buffers. Therefore operations on small tables are smaller than operations on large tables. For example, suppose a table has only 100 rows and each row is about 15 bytes. A table of this size can fit into a single 2K page, and can therefore be included into any SELECT statement with very little cost. No matter how this table is used, it costs no more than a single disk access to initially retrieve this page from disk.

Improving sequential scans

Sequential access to a table other than the first table in a plan involving a join between two tables of a select statement is ominous because such an access threatens to read every row in the table once for every row selected from the preceding table.

If the table is small, it is harmless to read it repeatedly because the data pages reside completely in page buffers. A sequential scan of such a table can be faster than searching the same table via an index especially if maintaining those index pages in memory pushes other useful pages back to disk. However if the table contains a large number of rows, repeated sequential access produces poor performance. You can overcome this, for example, by creating an index on the column that might be used to join two tables.

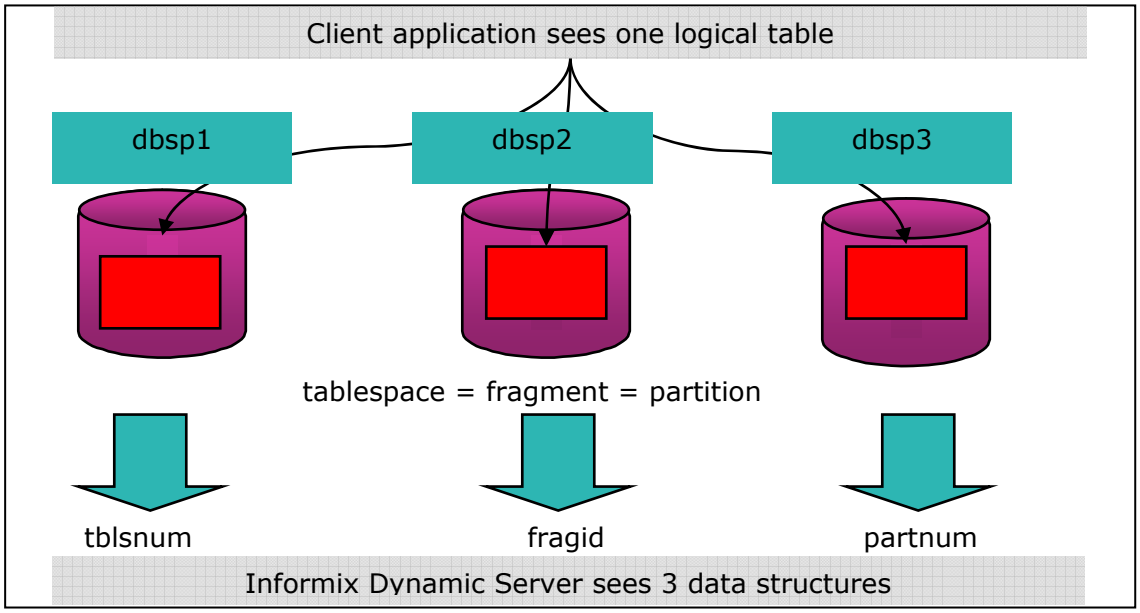
An index consumes space proportional to the width of the key values and the number of rows. Also the database engine must update the index whenever rows are inserted or deleted (or if key values are updated).

6.Fragmentation considerations

The primary consideration for when to fragment a table is not determined when the table reaches a certain size. While table size is important, the first two considerations should be:

- Query behavior and characteristics, for example, fixed or ad-hoc
- Knowledge of the data, for example, well known or unknown

These two considerations will determine the fragmentation policy. The key objectives of a fragmentation policy are parallelism and fragment elimination.



Planning a fragmentation policy

To a large extent, fragmentation policy is driven by the type of applications that access a table. If an expression based distribution scheme is chosen, a suitable expression must be constructed. With a round-robin distribution scheme, the database server decides which fragment it places the data rows.

In general the distribution scheme depends on the following factors:

- Whether or not the queries need to scan the entire table
- Whether the distribution of data is known in advance

SQL statements execute very slowly. There are approximately 400-450 ready threads in the queue.

Diagnosis

There are 1400-1500 concurrent sessions. Average CPU load is only 20 percent. The disk sub-system is also not heavily loaded. The storage area network is HP (XP 1200). In 5 minutes only 363,000 pages were read from disk, which equates to an average transfer rate of 4.7 MB/s (363000 * 4 KB / 300 s / 1024).

From the queue statistics (onstat -g qst), it seems that the most heavily loaded queues are the pt_* queues:

```
NAME      NWAITS   AVG_TIME  MAX_TIME  AVGQ MAXQ    NSERVS  AVG_TIME
...
hash PN_ID  3453    12869       75224    26   52   1383932        0
...
ps_6      476944    11925     1142849  3445   -1  17911952        0
ps_1      525149    12025     1318029  3871   -1  17940702        0
ps_5      390221    11816     1144661  2860 5711  17898469        0
ps_5      366101    11765     1202758  2626   -1  17864339        0
ps_1           2     2475        4602     1    1     77858        0
ps_6      466580    11925     1315310  3404   -1  17884618        0
ps_6      569252    11954     1316968  4129   -1  17933199        0
ps_6      483322    11913     1141628  3684   -1  17885366        0
ps_1          10    14584       24568     1    1    143787        0
ps_2           1    11916       11916     1    1     38962        0
ps_7      476593    11898     1217404  3501   -1  17884608        0
ps_5        1537    11548       63397    28   83   1742649        0
...
```

Customer then upgraded the instance to 9.40.FC8, which contains the fix to Bug 175470 [PERCEIVED MEMORY LEAK IN RSAM POOL AFTER FIX TO 166893]. But this caused a new problem, relating to the dbs_partn_1 MUTEX, which is waited on by about 600 user threads.

Resolution

This turned out to be a regression of the fix made to bug 175470. A new patch port, 9.40.FC8X2, was made available to this customer. Customer tested their application with more than 3000 users, and no performance problems were observed.

10. References

Reference	Author	Location
TCP-H Benchmark Overview	Informix/HP	www.tpc.org
Performance tuning IDS with Web Sphere	-	www.redbooks.ibm.com
Unlocking the mysteries of Update Statistics	John Miller	Chat with the labs series
Informix Performance Guide	-	publib.boulder.ibm.com
Fragmentation and Indexing Strategies	Mark Scranton	Chat with the labs series
ATLAS/RETAIN System	-	-
The Art of Computer Programming, Volume 2	Donald Knuth	-
Tuning the B-Tree Scanner	John Miller	nart.beaverton.ibm.com
ARIES/KVL: A Key Value Locking Method for Concurrency Control of Multi-Action Transactions on B-Tree Indexes	C. Mohan	-
Optimizing IDS Applications	Guy Bowerman	nart.beaverton.ibm.com

© Copyright IBM Corporation 2005
All Rights Reserved.
IBM Canada
8200 Warden Avenue
Markham, ON
L6G 1C7
Canada

Printed in United States of America
11-05

Neither this documentation nor any part of it may be copied or reproduced in any form or by any means or translated into another language, without the prior consent of all of the above mentioned copyright owners.

IBM makes no warranties or representations with respect to the content here of and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. IBM assumes no responsibility for any errors that may appear in this document. The information contained in this document is subject to change without any notice. IBM reserves the right to make any such changes without obligation to notify any person of such revision or changes. IBM makes no commitment to keep the information contained herein up to date.

The information in this document concerning non-IBM products was obtained from the supplier(s) of those products. IBM has not tested such products and cannot confirm the accuracy of the performance, compatibility or any other claims related to non-IBM products. Questions about the capabilities of non-IBM products should be addressed to the supplier(s) of those products.

IBM, the IBM logo, DB2, and DB2 Universal Database are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.