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

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.

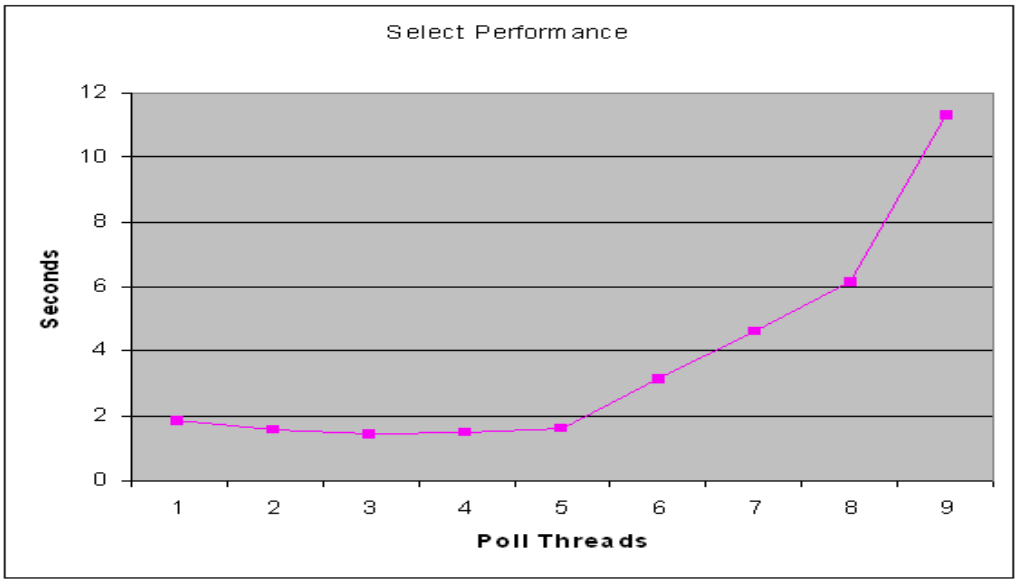
With this in mind, here are some initial ONCONFIG ($INFORMIXDIR/etc/$ONCONFIG) settings for OLTP.

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

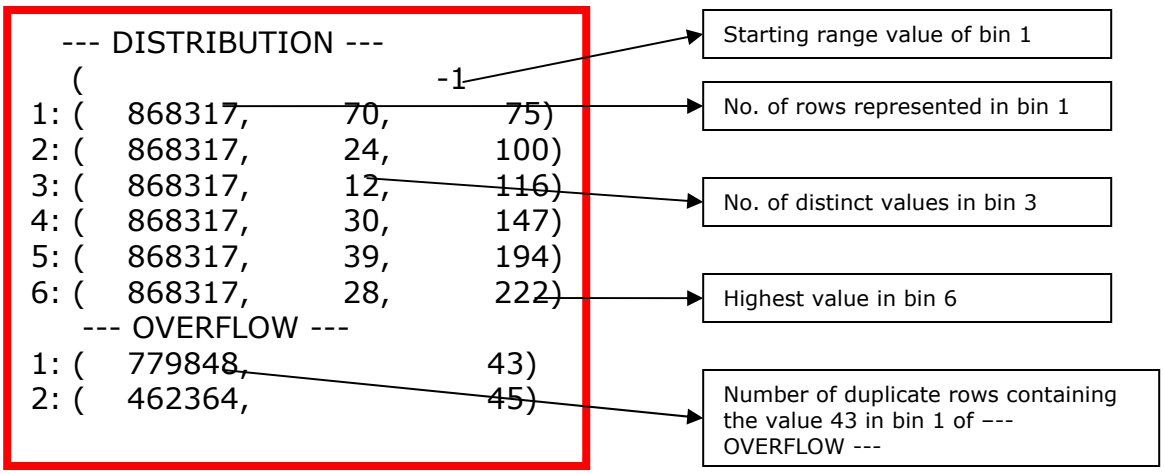
The UPDATE STATISTICS statement primarily collects information for the optimizer. Additionally, depending on usage, the UPDATE STATISTICS statement can determine the distribution of column values, force re-optimization of stored procedures and convert existing indices during a database server upgrade.

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.

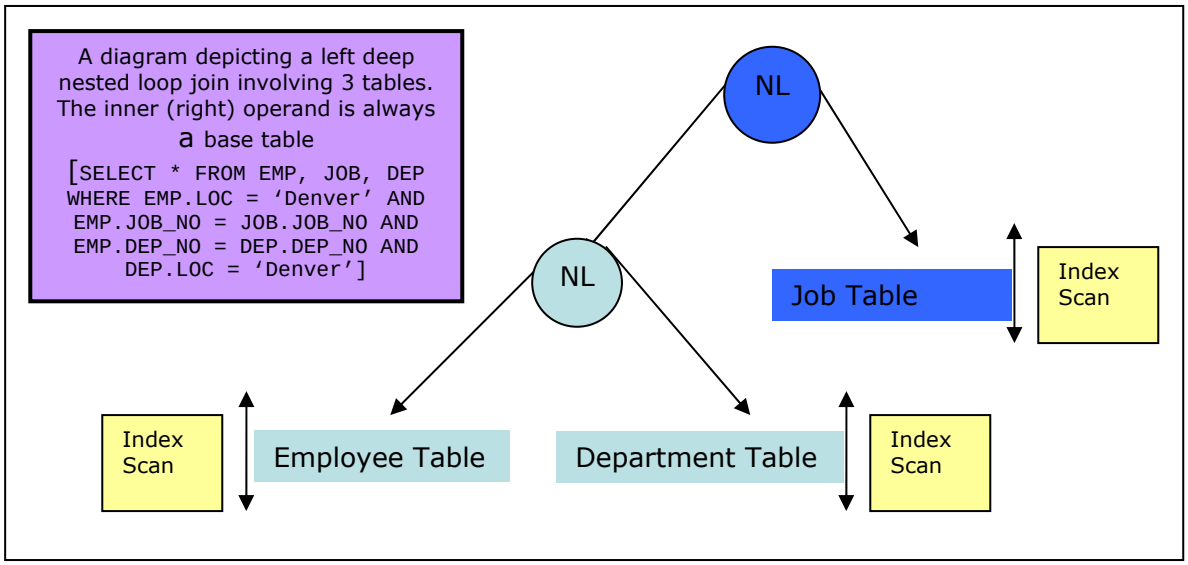
These names are derived from the appearance of the query plan drawn as a tree, with the outer join relation on the left and the inner relation on the right.

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.

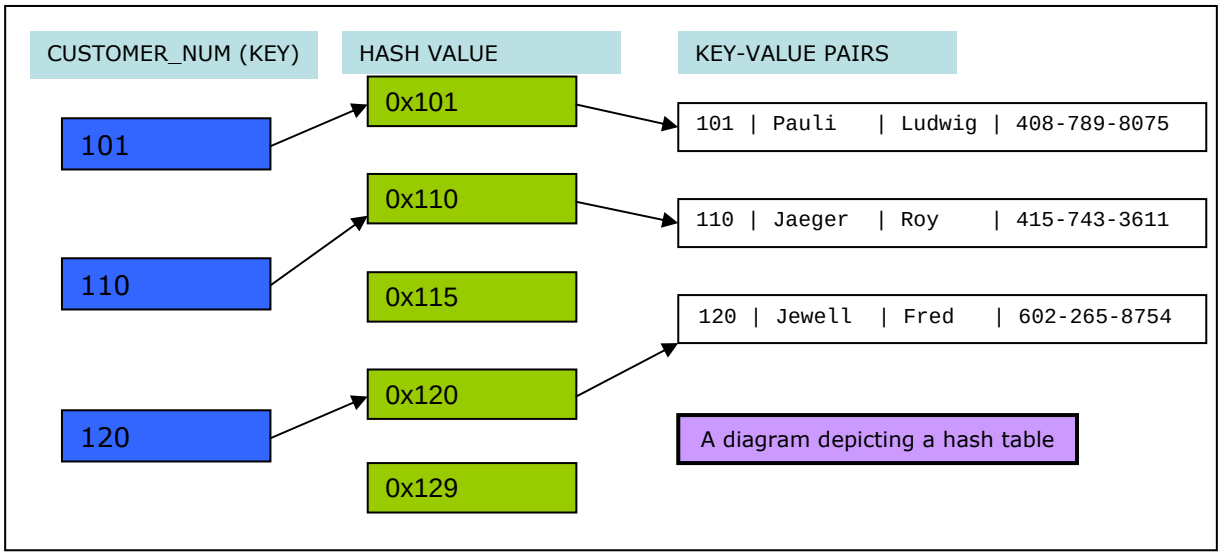
This can reduce the size of the intermediate results and is therefore a useful optimization technique in a distributed environment, where tables from multiple remote database servers are involved.

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*)

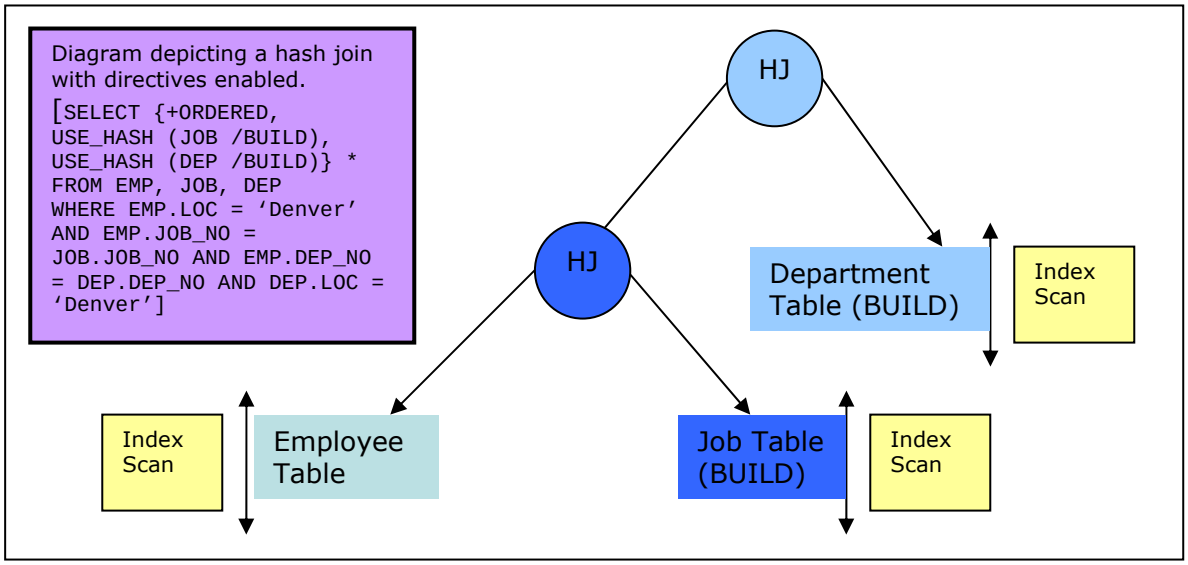
   The *hash table*, a data structure that associates keys with values enabling efficient lookup, is built on the smaller of the two tables, by applying a *hash function* to the join key of each filtered row. Hash joins require an "equi-join" predicate (predicates that use the "=" operator). The hash table 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.

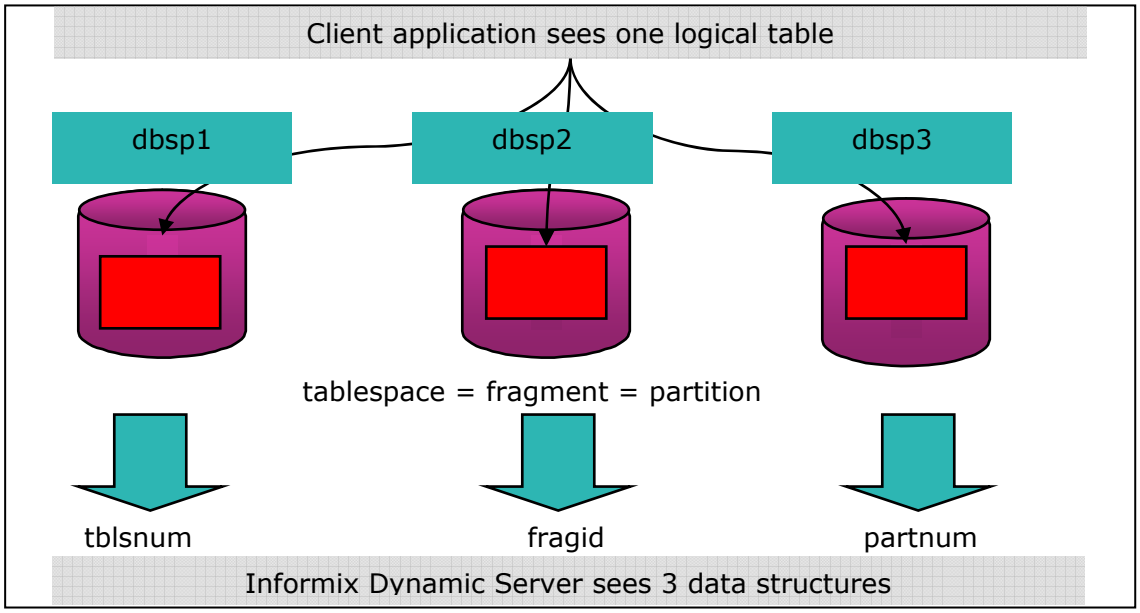
An index consumes space proportional to the width of the key values and the number of rows. Also the database engine must update the index whenever rows are inserted or deleted (or if key values are updated).

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

- Whether or not the application tends to delete many rows
- Whether or not the data is cycled through the table

A round-robin scheme is only useful when the following conditions apply:

- The queries need to scan the entire table
- The distribution of data is not known in advance
- The application does not delete many rows

If these conditions do not apply, you should decide on an expression-based scheme

In order to delete large amounts of data periodically based on a column such as month of the year, use that column in the distribution scheme. Then perform an ALTER FRAGMENT DETACH/ATTACH operation to cycle the data through the table. ALTER FRAGMENT DETACH and ATTACH statements provide the following advantages over bulk deletes and loads:

- The rest of the table fragments, other than the fragment being detached or attached are available to users
- The execution of a DETACH or ATTACH only takes a few seconds, making execution much faster than a bulk load or a mass delete

In an expression, avoid using columns that are subject to frequent updates, because this causes rows to move across fragments (for example, to be deleted from one fragment, and inserted in another). This type of activity increases CPU usage and adds I/O overhead.

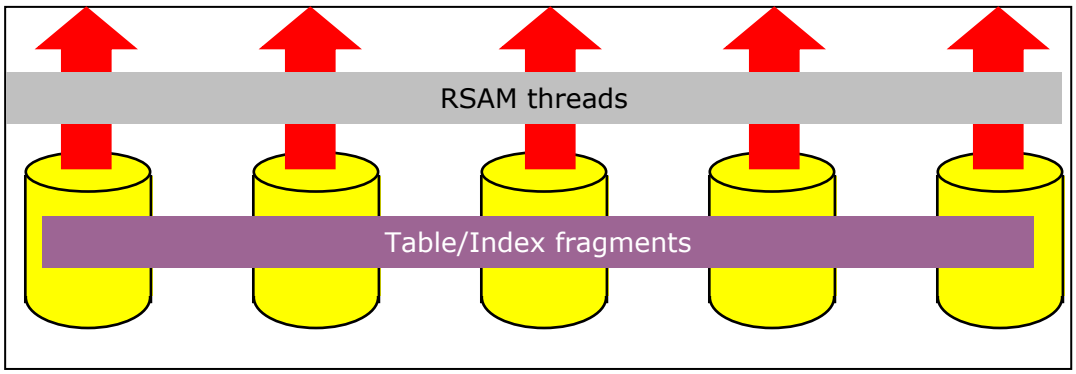
Note: Each table fragment has its own partition page in the table space (internally known as PARTITION-PARTITION) located in the initial chunk of its dbspace. Therefore each fragment can potentially reach the maximum number of allowable extents or table size.

A fragment has a partition number (partnum) of zero recorded in *systables* system catalog table, with the actual partnum recorded in the *sysfragments* catalog table. Each partnum recorded in the *sysfragments* table is linked to systables by the table identifier (tabid) column.

Parallelism

Fragments are accessed in parallel, decreasing scan time. Note however, for this to be efficient, the expression should generate an even distribution of rows across fragments.

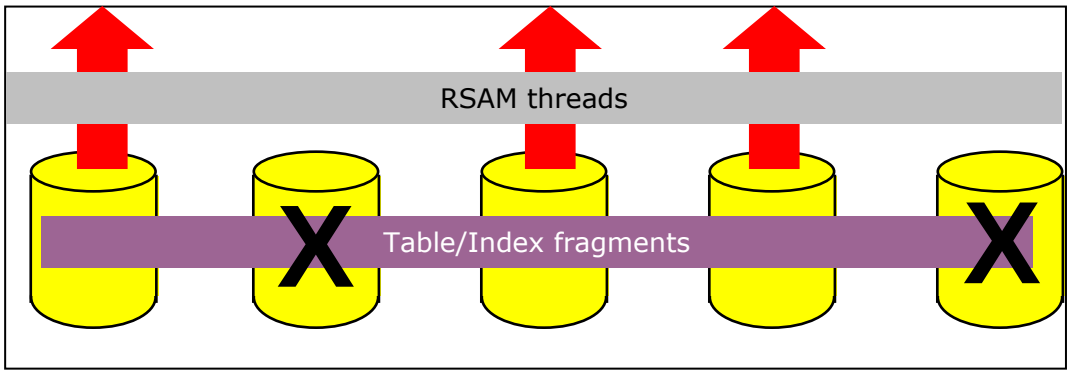
Sometimes business rules might dictate that some rows are accessed more frequently than others, in which case you can create a deliberate uneven distribution across fragments.



Fragment Elimination

Unnecessary fragments are skipped, improving concurrency and response time. I/O for the query is reduced, as is activity in the LRU queues. In a nested loop join, based on the key value from the outer table, inner table fragments can be eliminated if they fail the join condition.

It is important to note what kind of operators allow fragment elimination to occur and not occur. Expressions containing operators such as "IS NULL", "IS NOT NULL" and "!=" do not allow fragment elimination to occur. A fragmentation policy that employs a round-robin distribution scheme also does not allow fragment elimination to occur.



An enhancement in Version 10 of Informix Dynamic Server allows multiple table fragments belonging to the same table to reside in one or more dbspaces. Prior IDS versions required that each table fragment be created in a separate dbspace. The SQL statement uses the PARTITION keyword in its statement syntax to achieve this, as shown in the following example:

```
CREATE TABLE ACCOUNT_BALANCE (amount decimal (10, 2))
FRAGMENT BY EXPRESSION (
PARTITION acc_bal_ptn1 IN (amount < 100000) IN dbsp1,
PARTITION acc_bal_ptn2 IN (amount >= 100000 AND < 500000) IN dbsp2,
PARTITION acc_bal_ptn3 IN (amount >= 500000 AND < 1000000) IN dbsp3,
PARTITION acc_bal_ptn4 IN (amount > 1000000) IN dbsp3);
```

Guidelines for fragmenting tables and indexes

Below are guidelines for fragmenting tables and indexes:

- Fragment indexes to reduce contention between sessions. A query must only lookup one "table fragment" to locate the row. If the key value does not reduce contention (for example, when every user looks at the same set of key values), consider fragmenting the index on another value used in the WHERE clause. If a good expression is impossible to find, do not fragment the index; instead, detach it and place it in a separate dbspace.

- Keep expressions simple. Complex expressions might prevent the database server from eliminating fragments.

- Create an uneven distribution scheme. If the majority of the queries access only a portion of the data all the time, frequently accessed data is spread out over more fragments than the non-frequently accessed data.

- When the database server tests a value against the criteria for a given fragment, evaluation stops when the test is false. So, if the condition that is most likely to be false is placed first, the database server needs to evaluate fewer conditions before moving to the next fragment.

  For example, to SELECT a column value of amount = 2500 from the fragmented table ACCOUNT_BALANCE that was FRAGMENTED using the following expression, the database server needs to test all six of the inequality conditions:
  (amount >= 100 AND amount <= 1000) IN dbsp1,
  (amount > 1000 AND amount <= 2000) IN dbsp2,
  (amount > 2000 AND amount <= 3000) IN dbsp3;
  where as for the following expression, the database server only tests four inequality conditions:
  (amount <= 1000 AND amount >= 100) IN dbsp1
  (amount <= 2000 AND amount > 1000) IN dbsp2
  (amount <= 3000 AND amount > 2000) IN dbsp3
  The four tests are (amount <= 1000), (amount <= 2000), (amount <= 3000) and (amount > 2000)

- Avoid using an expression that requires a data type conversion. For example, a DATE data type is implicitly converted to an INTEGER for comparison purposes.

- Balance the number of fragments with the number of physical CPUs on the system.

- Do not fragment small tables, because it might not be worth the overhead of starting scan threads to access the fragments.

Guidelines for improving ATTACH/DETACH performance

ALTER FRAGMENT ATTACH/DETACH statements are primarily used to add or remove a large amount of data rapidly. These statements can slow down when the database server rebuilds indexes on the surviving table. However, the database server provides in built functionality to reuse indexes on the surviving table, which in turn can eliminate the need to perform an index build during the ATTACH or DETACH operation.

Here are some guidelines for improving ALTER TABLE ATTACH performance:

- Use non-overlapping expressions to ensure that no data movement occurs between the resultant partitions due to fragment expressions.
- Update statistics for all participant tables.
- Make indexes on the attached table unique if the index on the surviving table is unique.
- Ensure that database logging is enabled.
- Fragment the index the same way using the same fragment expression as the table to allow reuse of surviving index fragments
- When performing the ATTACH, ensure that the table being attached is non-fragmented in order to form the resultant fragmented table.
- Enable a dummy check constraint on the table being attached, with the same fragmentation expression as the resultant fragmented table. This will doubly ensure that no data movement occurs once the ATTACH statement is processed.

Here are some guidelines for improving ALTER FRAGMENT DETACH performance:

- Fragment the index in the same way as the table.
- Fragment the index with the same distribution scheme as the table.

Monitoring I/O activity across fragments

You can monitor and track the effectiveness of the fragmentation policy using the "onstat" command line utility. Using this utility enables you to determine if I/O is balanced across fragments.

The onstat –g ppf command displays the number of read/write requests sent to each fragment that is currently open. These requests generally do not indicate how many individual disk I/O operations occur, but they can give you an idea of the collective I/O activity.

The following statement will help you understand which partnums displayed by the onstat command, belong to which table in the SYSMASTER database:

SELECT T.TABNAME FROM SYSTABLES T, SYSFRAGMENTS F
WHERE T.TABID = F.TABID
AND PARTN = <PARTNUM column from onstat –g ppf output, converted to decimal>

7. Tuning the B-Tree Scanner

Prior to Version 9.3 of Informix Dynamic Server, the single B-Tree cleaner thread cleaned index pages of committed deleted items. For each request, it would take the following actions:

- Start a transaction
- Open the partition
- Verify it can get an IS lock on the partition
- Get the key
- Enter the critical section
- Clean the b-tree page
- Return its resources
- Commit
- Move to the next request

The B-Tree Scanner functionality evolved from the need to have greater performance and more reliability when cleaning index pages of deleted items.

Here is an overview of the B-Tree Scanner design goals:

- The workload for cleaning indexes is not from fully qualified requests submitted during a delete. Instead, it occurs by keeping track of how many times items in an index caused the server to do extra work. The index which caused the server to do the most work is the next index cleaned by the B-Tree scanner thread.

- The scanner searches the leaf level of an index for deleted items. After finding a deleted index item, the scanner thread test locks the item and performs a foreground remove of the item, possibly compressing the page.

- You can dynamically configure the scanner threads to allow for different workloads.

Some new terminology has been introduced by the B-Tree scanner. These are:

- Dirty Count, Hits – The number of times a user or an administrative thread has encountered an uncommitted deleted item while performing work
- Hot List – The list of indexes which need to be cleaned
- B-Tree Scanner – The new threads which are responsible for cleaning indexes

B-Tree Scanner Commands

The command used to start additional B-Tree cleaner threads is:

onmode –C start {count}

There can be a maximum of 32 threads running at any one time. If count is not specified, then count is 1.

The command user to stop B-Tree cleaner threads is:

onmode –C stop {count}

This command will not work immediately, but takes place on the assignment of the next unit of work. If count is not specified, then count is 1.

The command used to set the minimum number of deleted items an index must encounter before an index will be placed on the hot list is:

onmode –C threshold {size}

The command that determines the size of a table before index range cleaning is enabled is:

onmode –C range {size}

By storing the maximum and minimum dirty key values along with the index key statistics, it is possible to restrict the scan range of an index.

The commands that allows allow the priority of all B-Tree scanner threads to be set equal to that of normal database user threads, or to a priority lower than that of normal database user threads respectively, are:

onmode –C high

onmode –C low

User Interfaces

```
% onstat –C | onstat –C profile
BT scanner profile Information
==============================
Active Threads                              1
Global Commands                             0
Number of partition scans                16434
Main Block                  0xc0000001acdcc800
BTC Admin                   0x0000000000000000

BTS info             id    Prio    Partnum        Key          Cmd
0xc0000001acf148a0    0    Low     0x0C700009     1   100000   Scan index
    Number of leaves pages scanned                  131193826
    Number of leaves with deleted items               5135437
    Time spent cleaning (sec)                          546495
    Number of index compresses                         922463
    Number of deleted items                          80088769
    Number of index range scans                             0
    Number of index leaf scans                          21418
    Scan Type                                            Leaf

% onstat –C hot
```

```
Index Hot List
==============
    Current Item 28      List Created    08:40:39
    List Size    32      List expires in    0 sec
   Hit Threshold 500     Range Scan Threshold -1

Partnum        Key            Hits
0x02D00005       1           49613 *
0x02D00004       1           37344 *
0x02D00006       1           13853 *
0x02D00003       1            7545 *
0x0C600008       1            1080 *
0x0C700008       1            1042 *
0x0C700007       1             533 *
0x13D00003       1             511 *
0x13C00003       1             509
0x06400003       1             503
0x0C900006       1             502
::::
::::
0x0C80000D       1             500

% onstat –C clean
Index Cleaned Statistics
========================
 Partnum   Key   Dirty Hits Clean Time Pg Examined Items Del Pages/Sec
0x02600004   1       10           0           0           0      0.00
0x02600005   1       12           0           0           0      0.00
0x02600007   1       12           0           0           0      0.00
0x02700003   1        7           0           0           0      0.00
0x02700004   1        2           0           0           0      0.00
0x02700005   1        2           0           0           0      0.00
0x02700007   1       12           0           0           0      0.00
0x02d00003   1    10521       11493     8644646    11415444    752.17
0x0cc00007   1 C      1         282       77690       81137    275.50
“C” flag indicates cleaning in progress for this index partition

% onstat –C range
Cleaning Range Statistics
=========================
 Partnum   Key         Low         High       Size    Saving
0x00100002   1           1            1          4    100.0 %
0x01000030   1           7           11         24     83.3 %
0x01000102   1           1            1         40    100.0 %
0x02600003   1      611058      1060383    1187950     62.2 %
0x02600004   1      295045       295045     506245    100.0 %
0x02600005   1      293281       402135     414200     73.7 %
0x02600007   1       33037        82255     119657     58.9 %
0x02700003   1       66360       389302    1187950     72.8 %
0x02700004   1      211763       211763     506245    100.0 %
0x0cc00007   1         108         1996      26130     92.8 %
```

B-Tree Scanner Tuning Tips

By default B-Tree scanner threads run at a lower priority than user threads, so when the system becomes busy, index cleaning may not be able to keep up. In this case, you should set the priority of the cleaner threads to high rather than adding additional

threads. Adding additional threads that are already running on low priority will not increase the amount of deleted items that are cleaned.

8.Concurrency and Performance

Concurrency is a property of systems, which consist of computations that execute overlapped in time, and which can permit the sharing of common resources between the overlapped computations. In other words, *concurrency* occurs when two or more execution flows are able to run simultaneously.

There are several issues you need to consider when working in a multi-user environment. When one user thread is reading from a table, another user thread is reading or modifying the same table. Concurrency is crucial to performance in a multi-user environment.

To control concurrent events, the database engine employs locking and isolation levels.

Lock Type	Description
Shared	A shared lock reserves an object for reading only. This prevents the object from being changed while the lock remains in place. Multiple shared locks can be placed on the same object, allowing the same object to be read simultaneously by different *readers*[1].
Exclusive	An exclusive lock reserves an object for the use of a single reader or writer. This lock is used when the user thread modifies the object by changing its contents. Once an exclusive lock is placed on an object, no other type of lock can be placed on that object.
Promotable/Updateable	A promotable (or updateable) lock establishes the intent to update. This lock can be placed on an object that has already been locked with a shared lock. It cannot be placed on an object that has an exclusive lock or another promotable lock. However, once this type of lock changes its state from shared to exclusive, any existing shared locks should be dropped.

[1] A reader or writer is a user thread started by Informix Dynamic Server.

Locking and Isolation Levels

A lock is a software mechanism that prevents others from using a resource. A lock can be placed on an individual row or index key, a page of data, a table or the database itself. The maximum number of rows or pages that can be locked in a single transaction (with database logging enabled) is controlled by the total number of LOCKS configured in the ONCONFIG file. This configuration controls the initial size of the lock table[2] whose space is allocated in the resident portion of shared memory along with buffers, log files and locations of DBSPACES, CHUNKS and TABLESPACES. If lock allocations exceed the configured value, the database server dynamically increases the size of the table.

[2] The size of a single lock structure is 44 bytes. The maximum value of the LOCKS parameter is 8 million. The size of the memory allocated to the lock table is LOCKS * 44 bytes. Every new dynamic lock allocation adds 100,000 locks. The dynamic lock allocation occur a maximum of 15 times. The absolute maximum number of

locks that can be added this way is 8 million + 15 dynamic allocations of 100,000 locks (8,000,000 + 1,500,000). The absolute maximum is therefore 9.5 million locks, after which the engine runs out of locks. Any lock allocation that exceeds the value of LOCKS in a single transaction, is considered a "lock overflow". You can view this by running onstat –p and examining the "ovlock" column or by examining the database server message log.

The size of the object being locked is referred to as the *scope* of the lock or the *lock granularity*. On general the wider the scope the more concurrency is reduced. Locks are implicitly released when the transaction ends or when the database is closed. If a database does not use transactions and the transaction is not committed or rolled back, the lock can only be released by an explicit SQL statement such as UNLOCK TABLE <table-name>.

Locking granularity affects performance. When a user thread cannot access a row or a key value, the user thread can wait for another thread to release this resource. But if an entire page is locked by a userthread, a higher probability exists that a lot more threads are waiting to access a row on this page.

Page locking is the default behavior when a table is created without the LOCK MODE clause. With page locking, the entire page that contains the row is locked. The advantage of page locking is that when several rows on a data page are updated or selected, only a single lock is required. Conversely, a page lock on an index (especially during inserts, updates, or deletes of a row and hence the insert/update/delete of an index key) decreases concurrency because the index page is more densely packed and could potentially contain keys that would be available to other user threads.

Row and key locking are not default behaviors. Row locking is enabled with the LOCK MODE ROW clause using a CREATE/ALTER table DDL statement. Row and key locks generally provide the best overall performance for a relatively small number of rows due to the increased concurrency.

When a user deletes a row from within a transaction (database logging mode is enabled), the row cannot be exclusively locked because it will soon cease to exist. However the database server must somehow record that a row existed until the end of the transaction. To delete a row, the database server employs before/after page images, with the changes recorded in the physical log and the operation recorded in the logical log. The free map for the page is also updated and the space in the page's *slot table*[1] is reserved. Upon transaction roll back, the changes are undone by restoring the before page images from the physical log. Upon transaction commit, the space is released, and the specific slot that describes the row is deleted. If the deleted slot happens to be the last slot on the page, the page is freed for reuse as well.

[1] A slot table enables the database server to find data on a page. It is a series of four byte entries (or slots) which begin at the page ending time stamp and grows towards the beginning of the page. Each slot in the table describes a single row that is stored on that page. The data structure representing a slot comprises of two parts; the location of the row's first byte, and its length. The slot table also has its own "address" called a "row-id" that comprises of a page number and a slot number. A row-id is used to uniquely identify a row of data.

When a row in a table with a B-Tree index is to be deleted, the database server employs a technique called *key value locking* (KVL). Unlike row locking for data, key value locking is employed for index information. When a row is deleted, any key values in the corresponding index are not removed immediately. Instead, each key value is marked with a flag as deleted.

One of the important uses of key value locking is to assure that a unique key remains unique through the end of the transaction that deleted it. When other user threads encounter the deleted flag, the database server determines the presence of a lock. If a lock exists then the transaction has not committed or rolled back. This either sends a lock error or waits for the lock to be released depending on the status of the SET LOCK MODE value set by the session.

Here is a summary of locking semantics used in key value locking and the meaning of each semantic term:

- FETCH – Given a key value or a partial key value (its prefix), check if the key value is in the index and fetch the full key. A starting condition (=, >, or >=) is also given.
- FETCH NEXT – Having opened a range scan with a Fetch call, fetch the next key satisfying the key range specification (for example, a stopping key and a comparison operator (<, =, or <=))
- INSERT – Insert the given key (key value, row-id pair). For a unique index only the key value is searched. For a non unique index, the whole new key is provided as the search key.
- DELETE – Delete the given key (key value, row-id pair).

		Next Key Value	**Current Key Value**
FETCH and FETCH NEXT			**S** for commit duration
INSERT	Unique Index	**IX** for instant duration	**IX** for commit duration, if next key value was not previously locked in **S**, **X** or **IX** mode **X** for commit duration, if next key value is previously locked in **S**, **X** or **IX** mode
INSERT	Non Unique Index	**IX** for instant duration, if insert key value does not already exist. No lock if insert key value already exists.	**IX** for commit duration, if (1) next key is not locked during this call, or (2) next key is locked now, but next key not previously locked in **S**, **X** or **IX** mode **X** for commit duration, if next key is locked now and if it was previously locked in **S**, **X** or **IX** mode
DELETE	Unique Index	**X** for commit duration	**X** for instant duration
DELETE	Non Unique Index	**X** for commit duration, if delete key value will no longer exist No lock if value will definitely continue to exist	**X** for instant duration, if delete key value will not definitely exist after the delete **X** for commit duration, if delete key value might or will still exist after the delete

The following table estimates the number of locks for a given SQL statement that does not include smart, large object (BLOB or CLOB) columns[1].

[1] If BLOB or CLOB data types are present, add one lock for each non-partition (BLOBSPACE) BLOB or CLOB value processed by the DML statement. If byte-range locking is present, also add one lock for every range. For TEXT data types, no locks are placed if the DML statement is a SELECT statement.

If the DML statement is an INSERT or DELETE statement, then one lock for every page of TEXT or BYTE data that is inserted or deleted. If the DML is an UPDATE statement, then one lock for every page of old TEXT or BYTE data (before image) and one lock for every page of new TEXT or BYTE data (after image).

Isolation Level	DML Statement	Number of locks
Dirty Read	SELECT	0
Committed Read	SELECT	1
Cursor Stability	SELECT	2
Repeatable Read	SELECT (indexed scan)	1 + number of rows that satisfy the where clause + number of index key values that satisfy the where clause
Repeatable Read	SELECT (sequential scan)	1
N/A	INSERT	2 + number of indexes present
N/A	UPDATE	2 + 2 per changed key value
N/A	DELETE	2 + number of indexes present

The isolation level is the degree to which user threads are isolated from each other's concurrent actions. The number and duration of locks placed on data when a SELECT statement is executed, depends on the isolation level that you set. The type of isolation level that you set affects performance greatly, because of its effect on concurrency.

The following table shows various isolation levels supported by Dynamic Server:

Isolation Level	Value displayed by "onstat –g sql"
Dirty Read	DR
Committed Read	CR
Cursor Stability	CS
Repeatable Read	RR
Dirty Read (with retain update locks)	DRU
Committed Read (with retain update locks)	CRU
Cursor Stability (with retain update locks)	CSU

Monitoring and Administering locks

When the engine reads a page, it checks if the row in the page or the page itself (or the table, database to which the page/row belongs) is listed in the lock table. Once the corresponding lock table is established, the lock type is checked.

The following table summarizes the lock types that can be present in the lock table:

Lock Type	Description	Statement responsible
S	Shared Lock	SELECT
X	Exclusive Lock	INSERT/UPDATE/DELETE
U	Update Lock	SELECT (Update Cursor)

B	Byte Lock	Any DML statement that updates VARCHAR columns
I	Intent To Lock	Any DML statement

You use the onstat command "onstat –k" is used to view the lock table output. In the following example of onstat –k output, a user thread is inserting one row into a table in a logging database. The user thread (whose address is denoted by the column "owner" in the output) holds the following locks (denoted by "type") in the order shown:

- A shared lock (S) on the database
- An intent exclusive (IX) lock on the table displaying the intent to lock
- An exclusive lock (X) on the index created for this table
- An exclusive lock (X) on the unique index of the primary key constraint
- An exclusive lock (X) on the table row

```
Locks
address  wtlist  owner      lklist   type     tblsnum  rowid  key#/bsiz
440a43ec 0       55bf0354   0        HDR+S    100002   205          0
440a4444 0       55bf0354   440a43ec HDR+IX   100083   0            0
440a44f4 0       55bf0354   440a45fc HDR+X    100099   100          1
440a45fc 0       55bf0354   440a4704 HDR+X    100084   100          1
440a4704 0       55bf0354   440a4444 HDR+X    100083   300          0
5 active, 2000 total, 2048 hash buckets, 0 lock table overflows
```

User Thread

Indicates table lock

To determine the session responsible, issue the onstat –u command and match the "owner" column from the onstat –k output to the "address" column of the onstat –u output.

```
Userthreads
address  flags   sessid user     tty wait     tout locks nreads   nwrites
55bec018 ---P--D   1    informix -   0        0    0     30       104
55bec544 ---P--F   0    informix -   0        0    0     0        699
55beca70 ---P--F   0    informix -   0        0    0     0        0
55becf9c ---P--F   0    informix -   0        0    0     0        0
55bed4c8 ---P--F   0    informix -   0        0    0     0        0
55bed9f4 ---P--F   0    informix -   0        0    0     0        0
55bedf20 ---P--F   0    informix -   0        0    0     0        0
55bee44c ---P--F   0    informix -   0        0    0     0        0
55bee978 ---P--F   0    informix -   0        0    0     0        0
55beeea4 ---P--F   0    informix -   0        0    0     0        0
55bef3d0 ---P--F   0    informix -   0        0    0     0        0
55bef8fc ---P---   5    informix -   0        0    0     0        0
55befe28 ---P--B   6    informix -   0        0    0     0        0
55bf0880 L--PR--   17   informix 4   440a4704 -1   1     24       0
55bf0354 Y--BP---  39   informix 3   569b2a10 0    5     63       0
55bf0880 Y--P--D   12   informix -   4407f574 0    0     0        0
55bf0dac ---P--D   9    informix -   0        0    0     0        0
 17 active, 128 total, 27 maximum concurrent
```

Wait forever

Lock flag | "owner" from onstat -k | Session | Lock Address | No. of locks

To determine the table to which the lock applies, execute the following SQL statement on a system catalog table in the current database.
SELECT TABNAME FROM SYSTABLES WHERE PARTNUM = tblsnum (in decimal).

Additionally a query on the SYSMASTER: SYSLOCKS table returns the following columns, dbsname (*database name*), tabname (*table name*), rowidlk (*row id on which lock is placed*), keynum (*key number of the row*), type (*type of lock*), owner (session *id of the lock owner*), waiter (*session id of the first waiter on the lock*).

If an application session issues a "SET LOCK MODE TO WAIT" statement, the database engine waits for a lock to be released, instead of returning an error. An unusually long wait for a lock can give you the impression that the application is hanging. If the application should return immediately with an error, then it should issue the statement "SET LOCK MODE TO NOT WAIT" or it should wait for a specific number of seconds by executing "SET LOCK MODE TO WAIT <interval in seconds>".

9. Real cases from ATLAS/RETAIN systems

The following problems and their resolutions are real customer problems and resolutions taken from Informix ATLAS/RETAIN systems.

Operating Environment – HP-UX 11i
PMR Number – 15124,000,000
Informix Database Server – IDS 7.3/9.3/9.4/10.0
Problem
Every Saturday, around 3:00 pm users reported that the Informix instance was hung. They were unable to connect to the instance via dbaccess or via their application. It was not possible to shutdown the engine using onmode –ky.

Diagnosis
Various onstat dumps were captured, but nothing pointed to a problem. Running the HP-UX glance and top utilities showed that system was bottlenecked on CPU. The CPU usage in user mode was at 100 percent. All the oninit processes were on the top of the list, showing up as top CPU users. Users said this is the first time they are seeing this problem since they switched to a Veritas Cluster File system (CFS). HP and Veritas got involved in the discussion, but could not come up with any answers. Customer wanted to upgrade to IDS 9.4, but IBM support did not think it would solve the problem, since there was no real evidence of an Informix problem. Customer upgraded to 9.4 anyway. The instance hung again the following Saturday, but this time glance and top showed the CPU usage was 100 percent in system mode. During this time IBM support found that there was a problem accessing files on /usr/Informix which is created on CFS.

Resolution
When Veritas was told about this, they went back and suggested HP-UX kernel patch - **PHKL 33988** which eventually resolved the problem. Once this patch was installed, CPU usage never hit 100 percent and engine ran fine without "hanging".

Operating Environment – HP-UX 11.00
PMR Number – 399148 (Atlas Case ID)
Informix Database Server – IDS 7.3/9.3/9.4/10.0
Problem

About twice a week, customer reported that the Informix instance was hung.

Diagnosis
Onstat outputs collected showed that there were more than 1000 users connected to this instance. Lots of connections were made using I-STAR (srvinfx threads) from other instances. Operating system diagnostics were also collected. Total physical memory on the machine was 2GB. Glance and perfview analysis showed that physical memory was only a few megabytes when this problem occurred.

Resolution
IBM Support discovered that the memory starvation was induced by a customer application (written in ESQL/C) which ran as a daemon process. This processes leaked approximately 1MB of memory every 4-5 minutes. Customer had multiple copies of this process running, but they also had a cron job to stop/start these processes (as the memory leak bug was well known). Since they did not have the source code available to fix this memory leak, Informix support suggested they stop/start these processes at much shorter intervals than in the past. Once they did this, the Informix instance ran fine. Eventually they upgraded the hardware to a machine with 4GB of memory.

Operating Environment – HP-UX
PMR Number – 02108,999,778
Informix Database Server – IDS 7.3
Problem
Customer encounters slow performance from their IDS engine.

Diagnosis
Here is the excerpt from the Informix msglog

```
Informix Dynamic Server Version 7.31.UD7 – On-Line (LONGTX) – Up 21:37:47 – 547592 Kbytes
Blocked:LONGTX

16:12:13 Aborting Long Transaction: tx 0xd6bbbb18 username: normaliz uid: 259
16:12:13 Aborting Long Transaction: tx 0xd8840b78 username: normaliz uid: 259
```

Here is the excerpt from onstat –x

```
Transactions
address         flags   userthread      locks   log begin       isolation retrys coordinator
d6bbbb18        A-B--   d8a5285c        6       28738           DIRTY          0
d8840b78        A-B--   d8a6acf8        6       28738           DIRTY          0
Note: * B in flag means begin work
```

Resolution
The problem was due to the two transactions (denoted in the onstat –x output) consuming logical logs that exceeded the long-transaction high water-mark (LTXHWM) percentage of 50 percent that is set in the onconfig parameters. The current logical log when the transaction began was 28738 (denoted under "log begin"), and when the long transaction message appeared in the Informix msglog, the current logical log was 28762. Thus 24 (28762-28738) logical logs out of a total of 40 or 60 percent of the logical logs were consumed by user-threads d8a5285c and d8a6acf8. Increasing the number of logical logs will alleviate the problem. In addition to that, users could set the corresponding long-transaction exclusive access high water-mark (LTXEHWM) to 100 percent to allow other transactions access to the logical logs, while this long transaction is rolling back. However if the transactions were consuming all the available logical logs, then LTXEHWM should be set to approx. 50 percent and not more.

Operating Environment – Solaris 10
PMR Number – None (Sun Alert ID: 102576)
Informix Database Server – IDS 7.3/9.3/9.4/10.0

Problem

Dynamic Server hangs. All subsequent connection attempts will also yield ERRNO 13 (permission denied) from the listener thread. Database server restart is inevitable.

Diagnosis

Here is the excerpt from the Informix msglog

```
17:07:53 listener-thread: err = -27001: oserr = 0: errstr =: Read error occurred during
connection attempt.
17:07:54 listener: poll return with non T_LISTEN/T_DISCONNECT event
17:07:54 listener-thread: err = -25573: oserr = 13: errstr =: Network driver cannot accept a
connection on the port. System error = 13.
17:07:55 listener: poll return with non T_LISTEN/T_DISCONNECT event
17:07:55 listener-thread: err = -25573: oserr = 13: errstr =: Network driver cannot accept a
connection on the port. System error = 13.
```

Resolution

The following workaround should be used until the fix for the bug can be implemented.
Disable TCP fusion by adding the following line to the "/etc/system" file:
set ip:do_tcp_fusion = 0x0
"TCP fusion" seems to be a Solaris 10 TCP optimization feature applicable to many logical network interfaces mapped to one physical interface, which can be the case if running multiple zones on one machine.

Operating Environment – Red Hat Linux 7.3
PMR Number – 382794 (Atlas Case ID)
Informix Database Server – 9.40.UC2

Problem

This is a benchmark at a customer location on a 3 CPU Linux box containing 2GB of memory. It involves loading 150 million rows from 99 pipe delimited ASCII text files into a 2 column table using HPL. The Geodetic Data Blade is also involved.
After 50 million rows are loaded at the rate of 2,700 rows/sec and 80 percent CPU utilization, the load rate dropped 550 rows/sec and 15 percent CPU utilization.

Diagnosis

Memory leak due to bug 164144 (A GLS bug associated with the conversion of ASCII date-time values to Geo Time Range format). A workaround to this required each HPL load job handles only 3 data files at a time for a total of 33 load jobs. Onmode –F was run at the end of each load job.
In addition the load jobs were getting halted by long checkpoint durations. Bug 121346 [Deluxe Mode HPL load jobs cause engine to block at checkpoint requests] seems to be a suspect. They can't use express mode since table contains an opaque data type. The problem was worked around by adding onmode –c after each load job.

The database engine was also bounced after every load since at least 3 load jobs (job #7, job #8 & job #15) complained of SQL error -271, ISAM error 172.

Once after almost all of the 150 million rows were loaded, a single RTREE index build was started. [CREATE INDEX <INDX_NAME> ON TABLE <TABNAME> (COL_NAME OPAQUE_COL_TYPE) USING RTREE (BOTTOM_UP_BUILD='NO', FILLFACTOR=95, SORT_MEMORY=600000, BOUNDING_BOX_INDEX='NO')]. The build was proceeding as normal, gradually filling up temp space (23 GB available). Then it fails with the error

(RTRB5) – RTREE ERROR – Function 'srtclose' failed with error: 116. File rtlsort.c, line 182. [ISAM error 116 – Cannot allocate memory].
Resolution
The RTREE index-build error points to insufficient memory for the sort. So an additional (4^{th}) temporary space of 6 GB in size was added, bringing the total temp space to 23 GB. Earlier it was 17GB in 3 temp spaces. Now the index was built successfully.

Operating Environment – HP-UX 10.20
PMR Number – 360108 (Atlas Case ID)
Informix Database Server – 7.31
Problem
Dynamic server hung, blocked on checkpoint. At first it resembles a mutex deadlock.

Diagnosis
Everything was waiting on one thread that was holding a dbs_partn mutex. This in turn was waiting on a pt_# condition, which was waiting for the checkpoint to complete, which was waiting on the thread holding dbs_partn mutex. Further analysis shows that kernel asynchronous I/O (KAIO) was enabled, and this seemed to have stopped working too.

Resolution
Turned off KAIO by setting the environment variable KAIOOFF=1, and re-started dynamic server.

Operating Environment – Red Hat Linux
PMR Number – 88046,019,866
Informix Database Server – 7.31
Problem
Customer experiences a session hang when connecting to the database server using dbaccess. The users are connecting to the database server remotely via an x-terminal. Even after two minutes, the connection could not be established. Then user does a Ctrl-C to terminate the session.

Diagnosis
Here is the excerpt from the message log:
03:55:14 Checkpoint loguniq 51146, logpos 0x6e7728
03:56:50 listener-thread: err = -956: oserr = 2: errstr = (adani@dynamic-161-144-165-166): Client host or user (adani@dynamic-161-144-165-166) is not trusted by the server. System error = 2.
04:00:18 Checkpoint Completed: duration was 2 seconds.

However, as per the customer, this was not the user who was facing the connection problem.

Resolution
The customer set the following environment variables on the client:
INFORMIXCONTIME=600
INFORMIXCONRETRY=10
This will cause dbaccess to retry the connection up to ten times at 60 second intervals.

Operating Environment – AIX 5.3 (72 CPUs; 128 GB memory)
PMR Number – 74714,820,820
Informix Database Server – 9.40.FC7
Problem

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

IBM Canada
8200 Warden Avenue
Markham, ON
L6G 1C7
Canada

Printed in United States of America
11-05