About the Author

Scott Pickett is the IBM Informix World Wide Technical Sales person as a thirty-five year exclusive user and consumer of Informix products. During this time, he has lived every single Informix version and release except maybe those that start with "0." (I forget) and experienced every single major position in the IT Industry that there is and has had his own business as well in the past. He approaches technical issues and problems from many different perspectives. When he's not doing Informix, he is happily married and father of one child, and still loves travel, 68 countries later and classical music.

Introduction

This article started out as the result of an IBM Informix User Community post I read last year that I answered on the IBM Informix Community. A user essentially wanted to know how to detect and resolve sequential scans. I decided I should elaborate more fully on this matter, as this is a problem which in my experience I have seen at many customer installations. This problem requires accessing several different parts of our Informix ecosystem for a database administrator, all of which you have at your disposal if you are user Informix. As an aside, the approach chosen herein is not the only one that is possible. There will be things not presented here for reasons of time and space; as such, I consider this article to be not comprehensive.

Sequential scans

A sequential scan in Informix is the search of table without the aid of an index being presently available to be used (a torturous set of words to be sure but accurate) and we have some circumstances that need to be looked at:

1) Any small table will get a sequential scan and it usually does not matter whether there is an index on the table that the query can use:
    a. It may be a small table that is perfectly intact and has had the usual maintenance performed upon it (update statistics). Here there is nothing more to do.
    b. The table is reporting small numbers or 0 but you think there could be more records in the table than it is currently reporting. Run update stats low for the table. Look again at the row counts, and if row counts are much larger, look at the table schema and see what indexes are available; if none ….. need to investigate. Normally, having completed this step and if the row counts are low, nothing more to do.

2) You have a huge production mix and know your system is not performing optimally. Occasionally you notice issues with performance and high cpu usage Your users call on the phone, send text or email, saying the system is slow at peak or near-peak usage or even randomly throughout the day and through learned experience (in other words, you have experienced enough uncomfortable internal calls about it to remember this), you think you know whom is running the query and where it is. The users, having done their part, hang up the phone with no further detail provided.

This issue can be multiple things, but I usually look at sequential scans for this problem first. Often, in my experience, there are many sequential scans in a given instance, but to prioritize in order to get some large immediate relief, you usually want to look at the worst ones first initially as they are often the ones that are causing the worst performance. The others are likely, **cumulatively**, actually causing much of the slowness as well (and robbing you of precious performance microseconds which matter because the query **runs in a lot of sessions** every second), but are harder for you to immediately impact as there so many of these and you can't fight all of your programming staff at the same time here (you can't make the changes and put them into production and chances are each one of these was written by someone else whom has long since departed and the current staff "won't" know what module executes the query); and as a result you won't achieve much in the time you have. Of course, selecting a unique source code snippet of the running query and then searching the file system source code repository for that unique snippet might prove to have immediate benefits.

Two benefits to prioritizing looking at the big stuff first is:

a. You can fix an immediate big problem that produces results with limited impact to the rest of the IT infrastructure (and people);
b. If you are right on the big stuff, then you gain cred with programming management to begin to make the programmers write better code which is the real source of this problem to yet further clean things up for all the rest of these issues.

It has been my professional experience at over 500 different customer accounts that most sql application and script code is hastily written and not always or ever optimized and not written properly. So a practice over the years has grown at companies to either 1) throw hardware at it, or 2) have the dba fix it and all other things as well in the IT infrastructure as well. While this practice is a real problem, it isn't going to be fixed in our lifetimes.

3) We need to look:

   a. At Informix query plan data on queries.
   b. Cumulative data on sequential scans by table.
   c. Methods to determine the worst queries to look at first for sequential scanning.

We will look at all of this in the article.

Things That Happen During Sequential Scans:

If the entire table in question is already in memory, you are lucky. It should scan it in memory and return an answer in more than reasonable time. And performance for that would be negligible. Large tables in particular, are often not fully cached unless there is a large surplus of memory available.

Since the optimizer cannot filter the rows and automatically eliminate the data you don't otherwise need to look at because there is no usable index, the entire table is read into memory.

And if all of the buffer pool memory allocated to the instance has been used, some of the buffer pool memory pages previously occupied by another table(s) will be forced out to make room for the data returned, which pages returned to memory are based on a most recently used to least recently used algorithm. Which pages get forced out is indeterminate in advance and changes each time due to dynamic usage. onstat -P can provide some clues run repeatedly in side by side sessions.

Pages are driven into memory from disk in a sequential scan scenario via increased cpu usage; this increase can be a lot depending on how many pages are involved. Users see this as slow and the system admin sees this as spikes on cpu usage. When the task is done and query completed; the system tries to heal itself and pages no longer needed in memory by the sequentially scanned table gradually/quickly disappear and are replaced in memory by other tables/indexes forced out previously or needed by other processing; the other processing can be the very same pages previously driven into memory at the start of this paragraph.

While this sounds ok, this explanation does not take into account of this happening on large scale systems with several hundreds or thousands of sessions and in particular when the sequential scans are happening in parallel, all or most of the time during normal operating hours and on multiple or single large tables constantly. Such servers are always sequentially scanning and cpu usage measured at the operating system is always near or at 100%. DBA's at these installations burn out quite easily as down time often today is not allowed and the user calls/tweets/emails are initially often relentless and eventually drop to none except for the most egregious of situations. The long-term phone call volume drop toward none when there ought to be calls often represents user loss of confidence in the overall system performance. And throwing hardware at it, such as new cpu or memory or both, typically the first choice of management, only puts a band aid over it, putting the unresolved query and/or database work off responsible, and as a result a much larger problem to the future, if it changes anything at all.

Since the database keeps growing, performance is worse with each passing day as the real cause, the query and/or its underlying table schemas, isn't fixed. Bouncing entire tables out of memory to fit an entire table(s) in memory in this manner is called *thrashing* to many. When we refer specifically to the buffer pool only being swapped in and out (similar to a yoyo) this condition is called Buffer Pool Turnover (BTR); typically you don't want to turn over the entire buffer pool for any page size more than 10 times an hour as performance will noticeably suffer. There is a way to detect the condition in Informix, thanks to Art Kagel:

select current as time, bufsize, nbuffs, round((((pagreads + bufwrites) /nbuffs) / (select (ROUND (((sh_curtime - sh_pfclrtime)/60)/60)) from sysshmvals)),1) BTR from sysbufpool;

It is useful to run this query periodically say every 5 minutes or less in the scheduler and unload its data to an appended file with a current date and time stamp to notice trends and patterns in processing. If you have more than one buffer pool size it will report for each buffer pool size.

So how do you detect/fix sequential scans when you are looking at the problem after the fact when the scans have already occurred; or may be occurring now?

Detection:

Part 1

These scans occur on tables. Use a big stick approach to data gathering; gather all available data on the user tables for all databases. Not just on sequential scans. There are many causes for slow performance. You want all tables and the basic data we collect on them as well as the database server *onstats,* to get the big picture. Put the data in a spreadsheet and so unload all table data and bring it into the spreadsheet. Sort within the spreadsheet the leading indicator columns in descending order one at a time on a first pass; remember your header columns are present. If the problem isn't a sequential scan from the data examination on a table then move on and look at other things. Here, we'll assume our issue is sequential scans; otherwise, this is a 500 page book and not done here.

If the configuration parameter *TBLSPACE_STATS* is set to 0, none of the output of the query below will produce any data, so make sure you set that configuration parameter to 1 to produce meaningful data. You will need to reboot the instance if this configuration parameter is not set to see results here.

One of the columns to look at is *seqscans*, which, as you might guess, is the Informix column name shorthand for sequential scans. As a rule of thumb, this column's value should be 0, unless the number of rows in the table is 4000 or less, 0 meaning no sequential scans have occurred on the table during the current run period where the profile counts have remained stable and have not been zeroed out either by executing *onstat -z* on the command line or by an instance shutdown and reboot. This data does not survive either of those operations. From a long-term database server performance historical perspective, saving data on sequential scans is a first step toward measuring query efficiency.

The data of interest is in the *sysmaster* database, in the view *sysptprof* and in the table *sysptnhdr*. The sysmaster schema for each Informix version is located in *$INFORMIXDIR/etc/sysmaster.sql* (the columns included in each of these objects differ from release to release of Informix so make sure you look at the sysmaster schema for your release first before running any queries). Here is a rather simple query, it is slightly long and wordy as column headers are provided in English for you to know what the data is once it gets into the spreadsheet:

```
Unload to clientdata.out
Select a.dbsname as database_name,
        a.tabname as table_name,
        a.partnum as partition_number,
        hex(a.partnum) as hex_partition_number,
        dbinfo ('dbspace', a.partnum) as Dbspace,
```

```
        a.lockreqs as lock_requests,
        a.lockwts as lock_waits,
        a.deadlks as deadlocks,
        a.lktouts as lock_timeouts,
        a.isreads as isreads,
        a.iswrites as iswrites,
        a.isrewrites as isrewrites,
        a.isdeletes as isdeletes,
        a.bufreads as buffer_reads,
        a.bufwrites as buffer_writes,
        a.seqscans as sequential_scans,
        a.pagreads as page_reads,
        a.pagwrites as page_writes,
        b.flags as flags,
        b.rowsize as row_size,
        b.ncols as number_of_columns,
        b.nkeys as number_of_indexes,
        b.nextns as extents,
        b.pagesize as page_size,
        b.created  as date_created,
        b.serialv as table_version,
        b.fextsiz as first_extent_size,
        b.nextsiz as next_extent_size,
        b.nptotal as total_pages_allocated,
        b.npused as total_pages_used,
        b.npdata as total_data_pages,
        b.octptnm as octal_partition_number,
        b.nrows as number_of_rows
  from sysmaster:sysptprof a, sysmaster:sysptnhdr b
where a.partnum = b.partnum
   and a.dbsname not in ("sysmaster","sysadmin","onpload","sysuser","sysutils")
```

Some of the above Informix 14 columns may not be present in your release. So check. Notice that the query isn't sorted. As hinted at above, this reduces impact on very busy systems, so let the spreadsheet do the sorting and obviously in numerically descending order when the data gets there. When instances get very busy, sometimes large queries or *onstats* can send off a "*Changing data structure forced command termination*" error which many of you may have seen. Removing the sort from the query <u>reduces</u> the possibility of this issue, but not always on busy systems. Run a *wc -l* on the file produced to see how many rows are unloaded; there is obviously a problem if the line count here exceeds my current Excel spreadsheet maximum limit of 1,048,576 rows (it's different on various spreadsheet product/edition/releases). Two+ sheets are used in that case, after sorting descending at the operating system level. Use the Linux or Unix *split* utility to make the math work to get it within a spreadsheet workbook on multiple sheets if need be although your system might have to be really huge and bad to look at 1,048,576+ objects of sequential scans.

Choose all of the columns above just in case there is something odd here. Some these columns can point to other problems based on what's seen, others are used as supporting data to a particular condition. A few columns in my experience never get looked at for any purpose, like *octptnm* and can be dropped from the output if you like. So why are we doing it this way?

- The offline spreadsheet becomes a starting point for examining issues at the database object layer.
- By sorting descending within the spreadsheet multiple ways and times over multiple columns, the load is kept off the server and this allows for quick and easy to read analysis. You know what tables are receiving the most sequential scans in descending order and the tables that have the most rows or pages or deadlocks or read the most buffers and which tables have indexes (but not necessarily whether the table has the correct indexes for a particular query). Normally, the query is composed in column order as found within the table, but it is probably useful to move the columns you desire to examine next to each other within the spreadsheet, and drop the ones within the query you think you do not need. The query also outputs index partitions, but *seqscans* is always 0 for these, as are row counts). One really useful thing is to sort again in query column order; you see the number of times all found query's run in.

How do you interpret this data? The column labels are deliberately obvious for each data point. A table with 2,000,000,000 sequential scans with an accurate row count between 0 and 4000 is usually not something to worry about. A table with 100,000 sequential scans and a row count of 50,000 or more is a problem. A table with 60,000 rows or more and no indexes (b.nkeys = 0) is always a problem for possible sequential scans. A table with 60 sequential scans and 500,000,000 rows is always problem, especially if there are indexes. Basically, if you have 50-60,000 rows or more in a table and have sequential scans then you should look at it. The ordering of your data can easily tell you where to start looking. There are those of you reading this that will disagree with this, but when it is really bad and for many dbas they know it is, you have to start somewhere. So start big and work the list downwards. This is not the only criteria. Read on.

Part 2

The next thing to do is look at your table schemas which we identified in Part 1 having sequential scanning activity to see. We do this to see what columns of a table are indexed. If there are indexes, and queries are doing a sequential scan, or if there are not indexes and you are getting sequential scans on a table, then you know you have a problem. So to see the table schema and its indexes and other related DDL:

dbschema -d database_name -t table_name -ss > table_name.sql

The *-ss* option is there so to see where the data storage is. Hopefully, you don't have any tables or indexes in *rootdbs* ……… if you do ….. move them.

As an aside, running dbschema during the middle of the production day on large systems can slow a system somewhat so run it over the weekend or at night if possible when things are slower as a matter of maintenance to a file and keep your history. Then you can run *diff* between your

current file and a previous file to see if any changes have occurred since your last run. Most of the time you may only need two files, the previous and current ones. If you want to keep a schema history, then compress and encrypt them if security is a concern.

Typically when there is a sequential scan, there is not an index with columns in the right places where the optimizer can take advantage of them; or there is not an index at all which can be used.

It is possible that a table will have an index and yet a query will still sequentially scan, as the filter column chosen in the query does not make use of an indexed column.

Put the table schema output aside for now but don't delete it …. We will come back to it later on.

Part 3

So far, we now know the tables to look at for our sequential scanning activity and the schemas to evaluate. The original spreadsheet has the row and page counts that we can use for supporting data. These activities may tell us whether we need to add an new index which may include columns not previously considered for indexing, or whether update statistics needs to be run because we have recently added a huge new table or a huge number of records to an existing table but haven't updated statistics just yet, which can cause the optimizer to err in query execution. None of this tells us what the queries are actually causing the issue(s).

Live running queries (and some not live as well, but its session is) can be found in the view sysmaster:syssqexplain.sqx_sqlstatement (this captures the first 32,000 bytes only of the query in memory as it is the size of the column) where sysmaster:syssqexplain.sqx_seqscan > 0 and sqx_iscurrent="Y". As you may guess, if the value for the column sqx_seqscan is > 0 then we have **some portion of the query** running a sequential scan. You might want to unload this data delimited to a file and then upload to a spreadsheet and perhaps sort on the sql statement column in the spreadsheet to see how often that particular sql statement is run to see its impact.

So here is the code, output for the statement, column names aliased are a bit long but we need to see the data retrieved, some of which is not really needed here:

```
Select a.sqx_sessionid as session_id,
        a.sqx_sdbno as sdblock_array_position,
        a.sqx_iscurrent as current_running_statement,
        a.sqx_executions as total_executions,
        a.sqx_cumtime as total_cumulative_execution_time,
        a.sqx_bufreads as total_buffer_reads,
        a.sqx_pagereads as total_diskread_pages_reads,
        a.sqx_bufwrites as total_buffer_writes,
        a.sqx_pagewrites as total_diskwrite_pages_reads,
        a.sqx_totsorts as total_sorts_performed,
        a.sqx_dsksorts as total_sorts_to_disk,
        a.sqx_sortspmax as max_sort_disk_space_required,
        a.sqx_conbno as stmt_conblock,
```

```
        a.sqx_ismain as statement_main_block,
        a.sqx_estcost as query_estimated_cost,
        a.sqx_estrows as query_estimated_returned_rows,
        a.sqx_seqscan as number_of_sequential_scans_run,
        a.sqx_srtscan as number_of_sort_scans_run,
        a.sqx_autoindex as number_of_auto_index_paths,
        a.sqx_index as number_of_index_paths,
        a.sqx_remsql as number_of_remote_sql_paths_run,
        a.sqx_mrgjoin as number_of_merge_joins_run,
        a.sqx_dynhashjoin as number_of_dynamic_hashjoin_run,
        a.sqx_keyonly as number_of_key_only_scans_run,
        a.sqx_tempfile as number_of_temp_file_used,
        a.sqx_tempview as number_of_temp_view_used,
        a.sqx_secthreads as number_of_secondary_threads_used,
        a.sqx_sqlstatement as sql_statement_executed
  from sysmaster:syssqexplain a
where a.sqx_seqscan > 0
   and a.sqx_iscurrent="Y"
```

To see data populated in all of the selected columns above, SQLTRACE should be turned on in the configuration file; otherwise the columns sqx_cumtime, sqx_bufreads, sqx_pagereads, sqx_bufwrites, sqx_pagewrites, sqx_totsorts, and sqx_dsksorts shown above will not be populated. Make sure SHMADD or SHMVIRTSIZE has spare memory available and are set large enough to handle this; 4 times the default SHMADD value at a minimum. Changes to SQLTRACE require a server reboot to take effect. NOTE: *cumtime*, the current amount of cumulated execution time is reported here, and you can sort on that in reverse order as well. Obviously, a really big number relative to the others might be a problem here as well.

Here there are some small problems for some of the columns chosen above to consider:

1) The Optimizer may run a query shown in the column *sqx_sqlstatement* with a multi-part where clause as a sequential scan, such as a join to a table with less than 2000 rows. It does not tell us which specific query part is running a sequential scan, **just that some part** of the query is running a sequential scan. The *sqexplain* output (a bit further ahead) does show each join in the query and how it is supposed to run that piece of the query.

2) As stated above, the query output in the column *sqx_sqlstatement* only captures the first 32,000 bytes of a running query; if your query is longer than that it is likely that the query output may not contain *from* or *where* clause information or the *order by*, *group by, having,* or *into temp* clause; and even some bottom parts of the *select* clause if the statement is that big. Most statements are not that long and hopefully we will get a longer length with a new data type for the column *sysmaster:syssqexplain.sqx_sqlstatement* in a future release and its underlying base table column *sysmaster:sysconblock.cbl_stmt*. You can get the query plan of these or any other running queries via a different method. See the link to Mike Walker's article at the end of this article.

3) If the query shows that the *sysmaster:syssqexplain.*remsql value is 1 in the query sqexplain data then this portion of the query can be your problem. Remote sql queries run to another instance to execute and without the benefit of knowing what statistics and database objects are there. Slowness can and often does result. Using the environment variable FET_BUF_SIZE on remote queries can speed these up, potentially in combination with the environment variable OPTOFC=1 and OPEN, FETCH, and CLOSE statements on cursors. Perhaps if copies of the statistics for all instances that are connected were stored and updated as normal procedure on all of the other connected instances these would not be an issue performance wise. May be we see this as a future enhancement in the product.

4) If Parallel Data Query (PDQ) is turned on (the environment variable PDQPRIORITY > 0 and configuration parameter MAX_PDQPRIORITY > 0) this can be a problem if the *sysmaster:syssqexplain.*sqx_dynhashjoin value is 1 in the query sqexplain data output and you are intending to run OLTP queries. Dynamic hash joins are usually associated with Data Warehouse queries and not OLTP. These types of queries tend to run with lots of CPU and memory utilization and without the benefit of an index, hence their similarity in one aspect with a sequential scan. Lower PDQPRIORITY values of 1 or 2 help here if needed, which will drastically reduce CPU spikes caused by setting PDQ much higher but still supply a bit more memory and cpu than PDQPRIORITY=0. It goes without saying that existing PDQ queries operating on tuned down settings will take more time to complete but everything else will take less time on average.

5) Sometimes the optimizer will try to run an auto index operation, where it creates an index automatically on the fly during query processing. In these cases the *sysmaster:syssqexplain.*sqx_autoindex with a value of 1. This operation can be a bit slow depending on how much data needs indexing. Obviously, creating a new permanent index can and likely should be the solution.

6) Sort scans, seen in *sysmaster:syssqexplain.*srtscan with a value of 1. are often found in conjunction with sequential scans when no index is available to use. Fixing the query or how it runs (such as with an index) fixes this issue.

7) Knowing in advance what kinds of statement uses and situations may generate or act like a sequential scan is very helpful (not a comprehensive list);
    a. No index on the table being queried.
    b. Select * from table_name;
    c. Select * from table_name where 1=1;
    d. Any where clause in a delete or update or insert …. select …. where statement of the form *where 1=1;*
    e. Composite Index on table is a,b,c and the table gets where clause on the second column b. No other index has column b in its definition.
    f. Use of the NVL function and some other aggregate functions.
    g. Use of Substrings in queries that don't start on the first character of the string
    h. Correlated subqueries

i. Random temporary tables filling up large with no index and then selected against …. look into making a permanent temp table and index. Update stats …
j. Remote queries to another instance act exactly like sequential scans in terms of performance but are not categorized as such. Informix calls these *remsql.*

Query capture steps:

Typically:

1) Capture all of the currently running sequential scan queries into an appended single file, query as shown above, scheduled multiple times at different times of the 24 hour week (because certain queries only run at certain times of a day in a lot of installations), with different names to indicate the interval. This data can be compressed and encrypted for storage and security purposes. I usually run this every 5 minutes when I have a problem and otherwise randomly for every production processing shift and only when something changes.

2) Separating out the individual queries out into separate files from the main file in the first step with .sql file extensions so that each file contains 1 query each for testing purposes (not shown herein), with a **set explain on avoid_execute;** statement as the first line of each file with the query directly below and possibly a **set explain off avoid_execute**; statement at the end of every file with the query in between. Informix will create the file *sqexplain.out* to store query execution data here if it does not already exist. Where it creates this file is as follows:
    a. If the client application and the database server are on the same computer, the output file is stored in your current directory.
        i. If you are using a version 5.x or earlier client application and the output file does not appear in the current directory, check your home directory for the file.
        ii. When the current database is on another computer, the output file is stored in your home directory on the remote host.
    b. For a mapped user without a home directory, the explain output file is stored in *$INFORMIXDIR/users/server_svrnum/*uid_*uid.*
    c. For a mapped user with a home directory, remote clients' explain output files are stored in the user's home directory, and local clients' explain output files are stored in the user's current working directory.
3) Run the query and look at the results.
4) Notice the estimated cost on how Informix sees the expense of running the query. If it is not much (< 1000) then it may not be worth your while to concentrate here if things are really bad and move on to some other query or something else entirely.

In general, knowing where the query's originating source code file(s)/module(s) is/are pays off big time here as well as your operating environment. I usually look at a given query in the file

from above, pick out a piece of the query that is looking pretty unique as a search string to query source code flat files, and query via *find* and *grep* (and their derivatives)

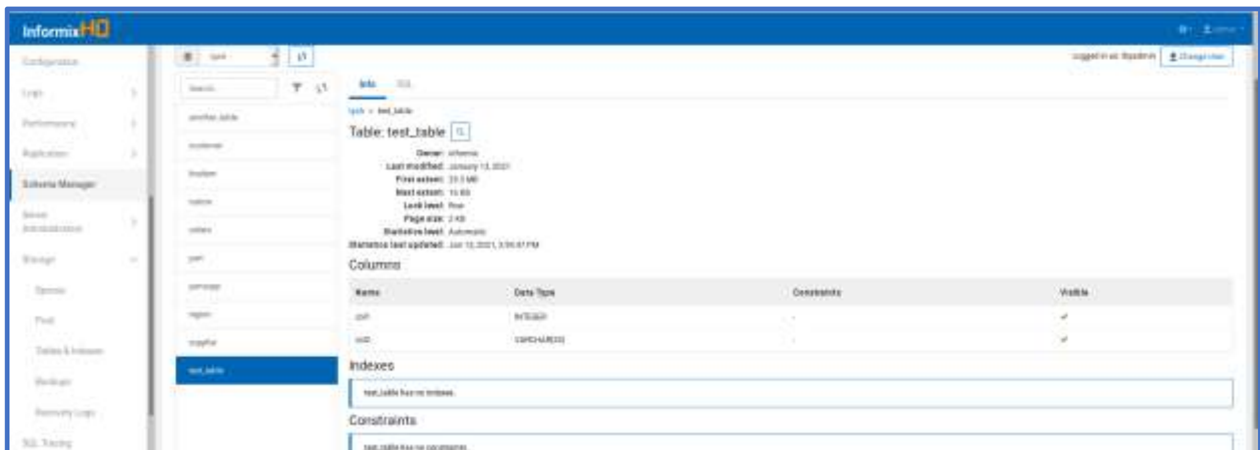
We will use a table called *test_table*, with 2 million rows and no index and 2 columns, so that any query run against it has to run a sequential scan:

```
File  Edit  View  Search  Terminal  Help
INFO - test_table:  Columns  Indexes  Privileges  References  Status  cOnstraints  triGgers  Table  Fragments  E
Display status information for a table.

----------------------- tpch@inst_1 ------------ Press CTRL-W for Help --------


Table Name           test_table
Owner                informix
Row Size             25
Number of Rows       2000000
Number of Columns    2
Date Created         01/13/2021
```

```
informix@ifmx-svr:~
File  Edit  View  Search  Terminal  Help
INFO - test_table:  Columns  Indexes  Privileges  References  Status  ...
Display column names and data types for a table.

----------------------- tpch@inst_1 ------------ Press CTRL-W for Help --------

Column name          Type                                    Nulls

col1                 integer                                 yes
col2                 varchar(20,0)                           yes
```

```
informix@ifmx-svr:~
File  Edit  View  Search  Terminal  Help
INFO - test_table:  Columns  Indexes  Privileges  References  Status  ...
Display information about indexes for the columns in a table.

----------------------- tpch@inst_1 ------------ Press CTRL-W for Help --------

Index_name         Owner     Type/Clstr Access_Method       Columns
```

Or in one view via InformixHQ:



To avoid query execution time and also to see the query plan from the optimizer, we run *set explain on avoid_execute;* just before the query runs:

```
MODIFY: ESC    = Done editing      CTRL-A = Typeover/Insert     CTRL-R = Redraw
        CTRL-X = Delete character  CTRL-D = Delete rest of line

----------------------- tpch@inst_1 ------------ Press CTRL-W for Help --------

set explain on avoid_execute;
select * from test_table;
```

```
File  Edit  View  Search  Terminal  Help
SQL:  New  Run  Modify  Use-editor  Output  Choose  Save  Info  Drop  Exit
Run the current SQL statements.

---------------------- tpch@inst_1 ----------- Press CTRL-W for Help -----

      col1 col2

 Warning! avoid execute has been set
 No rows found.
```

So what did the output of *set explain on avoid_execute;* look like? See above, it was directed to the *sqexplain.out* file in the directory that is presently used and here is what the output looks like on a singleton query for my query (what appears in the file is relevant to your running queries only here):

```
QUERY: (OPTIMIZATION TIMESTAMP: 01-15-2021 08:19:22)
------
select * from test_table

Estimated Cost: 551489
Estimated # of Rows Returned: 2000000

  1) informix.test_table: SEQUENTIAL SCAN
```

Notice it is running a sequential scan. There is no index. So it should run a sequential scan. But there is also another reason it will run a sequential scan. Any query to any table of the form "*select * from table_name;*" where there is no *where* clause will automatically run a sequential scan in Informix. Is this particular query necessary to pay attention to? In other words, is the optimizer's Estimated Cost of the query execution really detrimental to the running of the server? If as a singleton query, a one-off query, probably not. But if run throughout the day then the as

shown six-figure Estimated Cost of the query is most definitely worth fixing. Notice too that it estimates 2,000,000 rows will be returned in the query, which is in this case exactly right. One place I went to in my consulting days had 500+ queries of this form, with about 80 running simultaneously, only the coded table name was different. Ouch.

Estimated Cost should always be in low digits, say < 1000, and even that is possibly too high on a system with massive numbers of sessions and users using the same query. Commentary here is directed to the situation of operational concerns of real production systems and not the examples provided here on my Lenovo T470 laptop upon which the examples are prepared.

A Query's Estimated Cost:

The latter point from above, the estimated cost, needs explanation here before proceeding. Informix has something called an estimated cost of running a query. This number has no meaning other than how the optimizer thinks the expense of the cost of running the query is relative to the running operations. Properly executing queries can range from single digits to as many as lower 4 digits. These are not expensive queries and 99% of queries on many systems run with these cost numbers as long as your SQL is written correctly. And every sequential scan fixed means everything else will run a little bit faster on average. In testing, a single user can run a sequential scan with a large estimated cost in less than a second and that looks good; the same query running several hundred times in parallel is a different matter.

Some queries performing a sequential scan will have an abnormal estimated cost of 4 digits or higher backed by the query plan showing how the query is running which will say "SEQUENTIAL SCAN" as part of the query plan. Any query plan with an estimated cost > 1,000 is worth looking at, especially if it is running in a lot of sessions all day long, **cumulatively**, they are a problem; a lot of small micro seconds cumulatively add up to be a lot of time. Queries with multi-part *where* clauses will run each part of the query differently normally. And since many systems running over Informix make money for the company, time is money as well. Hence, no downtime allowed.

Any query with a sequential scan with an estimated cost > 100,000 <= 1,000,000 is a definite problem and needs fixing.

Those queries with an estimated cost > 1,000,000 are likely to take longer than anyone's lifetime to **never return an answer** and **therefore worth fixing immediately and before all else.** Symptoms of these queries can be extremely prolonged usage spikes in CPU usage close to or at 100% and lots of end user calls and system administrator calls as well. And if there are multiples of these queries running in parallel …… not much is happening on these systems.

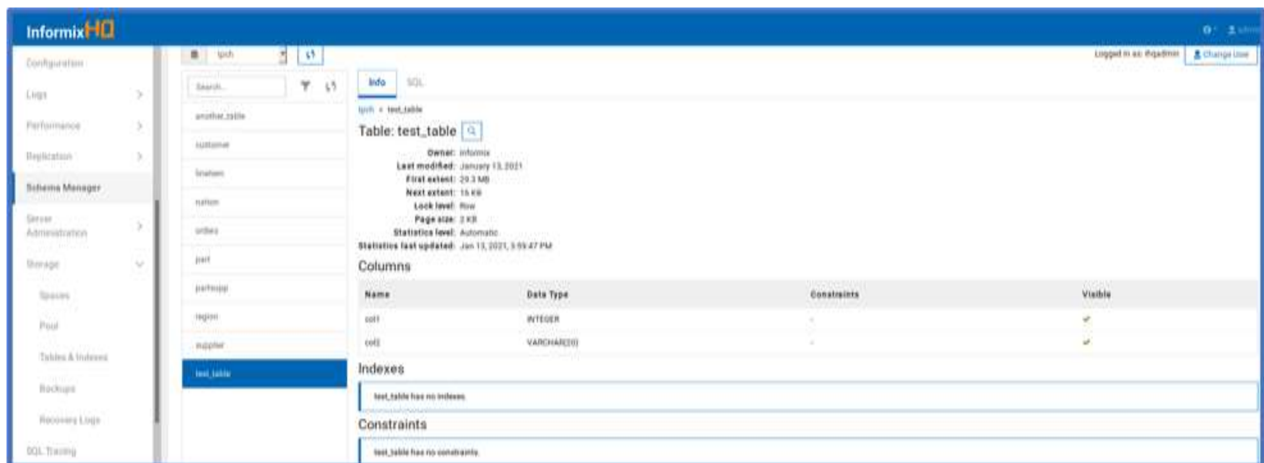
Informix 14.10.FC6, due for release at the end of Q1/beginning of Q2 2021, is scheduled to have as a feature the ability to see the current clock time of a running query in *onstat -g ses* or *onstat -g sql*, which will provide one more indicator here of slowness or the lack thereof as well.

So how do we fix the *test_table* query above? Well, first I haven't shown how much time this takes to run. We will use the time utility initially with no index first.

Here is the schema:

```
create raw table "informix".test_table
  (
    col1 integer,
    col2 varchar(20)
  ) in my_new_dbspace  extent size 30000 next size 16  compressed lock mode row;

revoke all on "informix".test_table from "public" as "informix";

{ TABLE "informix".another_table row size = 25 number of columns = 2 index size
= 0 }
```

And in InformixHQ the schema:



So the solution here is to find a way to build an index to filter the data and the table has two columns:

And here is what we are running below:

*select * from tpch test_table.sql;* (not including the set explain on statement which is removed)

time dbaccess tpch test_table.sql &

```
2000000 row(s) retrieved.

Database closed.

real    0m15.954s
user    0m2.739s
sys     0m3.619s

[1]+  Done                    time dbaccess tpch test_table.sql
Ifmx:
```

Execution time was 15.954 seconds as a sequential scan on my machine. The machine was not shutdown and rebooted prior to execution.

Next we'll create a unique index on the table and since the column *col1* has unique values thru prior inspection. *Test_table* is a raw table and a unique or regular index may be built over these kinds of tables:

create unique index test_table_1 on test_table(col1);

This index took 5 seconds or so to build on my laptop.

Now we will run *set explain on avoid_execute;* with the same query as previous and see how the optimizer intends to run this query.

```
Ifmx: cat sqexplain.out

QUERY: (OPTIMIZATION TIMESTAMP: 01-22-2021 15:02:40)
------
select * from tpch:Test_table

Estimated Cost: 551489
Estimated # of Rows Returned: 2000000

  1) informix.test_table: SEQUENTIAL SCAN

Ifmx:
```

Putting the index on the table had no effect on the query plan. Did I do something wrong? May be I forgot to run update statistics, to update the system catalog for the new index?

Update statistics low for table test_table;

```
Ifmx: cat sqexplain.out

QUERY: (OPTIMIZATION TIMESTAMP: 01-22-2021 15:02:40)
------
select * from tpch:Test_table

Estimated Cost: 551489
Estimated # of Rows Returned: 2000000

  1) informix.test_table: SEQUENTIAL SCAN

QUERY: (OPTIMIZATION TIMESTAMP: 01-22-2021 15:06:32)
------
select * from tpch:Test_table

Estimated Cost: 551489
Estimated # of Rows Returned: 2000000

  1) informix.test_table: SEQUENTIAL SCAN

Ifmx:
```

The query plan did not change. We know this will still run a sequential scan. The index and update statistics did nothing. Is the query any faster because of the index and update stats being run?

```
2000000 row(s) retrieved.

Database closed.

real    0m16.392s
user    0m3.121s
sys     0m3.601s

[1]+  Done                    time dbaccess tpch test_table.sql
Ifmx:
```

It's actually a bit slower, about 16.4 seconds. Again, the machine was not shutdown and rebooted prior to execution.

May be if we change the query to bring out just the first 500 records rather than all of them (assuming we only wanted those records originally when written):

*select first 500 * from tpch test_table.sql;* (set explain on included)

```
QUERY: (OPTIMIZATION TIMESTAMP: 01-25-2021 12:43:49)
------
select first 500 * from tpch:Test_table

Estimated Cost: 157
Estimated # of Rows Returned: 2000000

  1) informix.test_table: SEQUENTIAL SCAN

Ifmx:
```

So the Estimated Cost is way down, so that should mean we get the data back quicker, even though it is still a sequential scan. Is it quicker?

```
500 row(s) retrieved.


Database closed.


real    0m0.043s
user    0m0.005s
sys     0m0.016s

[1]+  Done                    time dbaccess tpch test_table.sql
Ifmx:
```

So it’s quicker, but may be not what you want.

Was it the index, or the sequential scan? The query plan says it was not the index and the stats were updated to reflect the presence of the new index. So this was with a sequential scan.

If the intent of the original query was to retrieve all of the data then the *first 500* clause would not be implemented in your code; <u>this specific situation arises</u> when customers migrate apps from other database products to Informix. There are other possibilities and solutions …..

A conclusion here:

1) The query as written will always run a sequential scan as that is how Informix will default run a query of this type:
    a. With no where clause to filter and
    b. A select clause asking for everything, all data, from a table
2) The query might therefore need to be rewritten if it’s a problem for performance, we might have to consider its usage,
    a. Is it a batch query or online,
    b. Was it written in a time when all that was required was the first 0-500 rows or so but Informix didn’t support the *first n rows* clause in the select statement;
    c. What happens if I modify the query to say the first 500 rows or so?

So it is a bit involved, but this is how you do it and gradually, you prove your case to the programming staff and yourself in this way by having the proof. Otherwise, your fellow IT Professionals will say it is the responsibility of the dba to do it. **Save all of your set explain outputs as evidence, and make sure you physically place the code “set explain on avoid_execute;” over the top of every query you run first**; so that you don’t contribute to the problem. Of course, if you are the dba **and** the programmer .... you have to fix it. You can,

once the statement has run, run **set explain off avoid_execute;** to return your session to normal functioning by the way.

In all of the analysis you should have the table schema available and your spreadsheet (sorted by sequential scans descending and number of rows descending as well) as you need that to determine whether you need to do one of three possibilities to fix that sequential scan:

a). Create a new index with some new columns, either single or composite
b). Modify the query (slight rewrite), and test using *set explain on avoid_execute*
c). Update statistics

There are several other possibilities for slow systems as well, but for sequential scans, this is a good start. Throughout all of this, we have been operating on compressed data, which makes the query select thru optimally organized pages with all of the rows pushed forward on each page, leaving empty space at the rear of the dbspace for new data. Had the data not been compressed it is likely our queries would have taken longer to execute.

Mike Walker of Advanced Data Tools has a good presentation on this subject with a different approach (finding the query plan of a running query, for example) to the problem here.

Thanks to Carlton Doe.

If you have questions you may reach me at: spickett@us.ibm.com