Best Practices: Informix Query Performance Tuning Basics

Mike Walker
Advanced DataTools
mike@advanceddatatools.com
Mike Walker

Mike Walker has been using Informix databases for over 20 years, as a developer and as a database administrator.

Mike heads up the Remote DBA Support for Advanced DataTools Corporation.

Contact Info:

mike@advanceddatatools.com
www.advanceddatatools.com
Office: 703-256-0267
Cell: 303-909-4265
Webcast Guidelines

• The Webcast is being recorded. The Webcast replay and slides may be available in a few days.
• Please Mute your Line. Background sounds will distract everyone.
• Use the Chat Button in the upper right to ask questions.
Performance Tuning Basics

- Identifying long running queries
- Explain plans – Query Tuning
- Optimizer directives
- Monitoring the buffer pool usage
- Finding busy tables
- Checking statistics
Long Running Queries – What’s Running?

• How to tell if a session is doing anything
• Start with “onstat -u”
• First position in the Flags column indicates what’s going on
  - B - Waiting for a buffer
  - C - Waiting for a checkpoint
  - G - Waiting for a write of the logical-log buffer
  - L - Waiting for a lock
  - S - Waiting for mutex
  - T - Waiting for a transaction
  - Y - Waiting for condition
  - X - Waiting for a transaction cleanup (rollback)

Interested in anything that is NOT a “Y” – and last position is a “-”
Long Running Queries – What’s Running?

onstat -u

<table>
<thead>
<tr>
<th>address</th>
<th>flags</th>
<th>sessid</th>
<th>user</th>
<th>tty</th>
<th>wait</th>
<th>tout</th>
<th>locks</th>
<th>nreads</th>
<th>nwrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>70000174751a028</td>
<td>---P--D 1</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>47702</td>
<td>3024353</td>
</tr>
<tr>
<td>70000174751a850</td>
<td>---P--F 0</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5744623</td>
<td></td>
</tr>
<tr>
<td>70000174751b078</td>
<td>---P--F 0</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2744394</td>
<td></td>
</tr>
<tr>
<td>70000174a2615c8</td>
<td>Y--P---</td>
<td>240522</td>
<td>admin</td>
<td>DX-ALCV</td>
<td>7000018cc1dc7c0</td>
<td>0</td>
<td>3</td>
<td>149935</td>
<td>0</td>
</tr>
<tr>
<td>70000174a261df0</td>
<td>---PR--</td>
<td>1555094</td>
<td>webuser</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4989784</td>
<td>512</td>
</tr>
<tr>
<td>70000174a262e40</td>
<td>Y--P---</td>
<td>1565579</td>
<td>webuser</td>
<td>-</td>
<td>700001855e069b8</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>700001753360ca8</td>
<td>Y--P---</td>
<td>1567350</td>
<td>webuser</td>
<td>-</td>
<td>70000185997dd18</td>
<td>0</td>
<td>0</td>
<td>3905</td>
<td>0</td>
</tr>
<tr>
<td>7000017533614d0</td>
<td>--BPX--</td>
<td>1567353</td>
<td>webuser</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>38</td>
<td>128</td>
</tr>
<tr>
<td>7000017533645c0</td>
<td>Y--P---</td>
<td>1562970</td>
<td>webuser</td>
<td>-</td>
<td>700001856de09b8</td>
<td>0</td>
<td>3</td>
<td>13332</td>
<td>192</td>
</tr>
<tr>
<td>700001753365e38</td>
<td>Y--P---</td>
<td>869782</td>
<td>bob</td>
<td>PROD-SRV</td>
<td>700001762fb5628</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>700001753366660</td>
<td>---PR--</td>
<td>1543869</td>
<td>webuser</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>11186388</td>
<td>834</td>
</tr>
<tr>
<td>700001753366e88</td>
<td>Y--P---</td>
<td>1496985</td>
<td>webuser</td>
<td>-</td>
<td>700001787828898</td>
<td>0</td>
<td>3</td>
<td>39872</td>
<td>0</td>
</tr>
</tbody>
</table>
Long Running Queries – How Long?

• Use "onstat -g ntt" to find out when the connection was established and when the last SQL was submitted

<table>
<thead>
<tr>
<th>netscb</th>
<th>thread name</th>
<th>sid</th>
<th>open</th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000007030e7cd0</td>
<td>sqlexec</td>
<td>951521</td>
<td>11:59:18</td>
<td>13:52:05</td>
<td>13:52:05</td>
</tr>
<tr>
<td>70000070284e600</td>
<td>sqlexec</td>
<td>951510</td>
<td>11:57:23</td>
<td>13:52:05</td>
<td>13:52:05</td>
</tr>
</tbody>
</table>
Long Running Queries

• Use onstat -u to tell you what is running
• Use onstat -g ntt to tell you the last time SQL was submitted
• Use the results of both to see what’s running *and* for how long
  – How long = difference between last SQL time and current time
Long Running Queries

Combine the info from the onstats together, repeating at regular intervals:

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Start Time</th>
<th>End Time 1</th>
<th>End Time 2</th>
<th>End Time 3</th>
<th>Wait Time</th>
<th>CPU Time</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Long Running Queries

For OLTP, would like to see the SQL time change every second or so

SQL time is changing for most sessions approximately every second

Sessions 59022843 and 59024816 have been running for 20 minutes and 10 minutes respectively
Long Running Queries

• For a session where SQL is being submitted regularly, view the SQL at regular intervals:
  
  onstat -g sql <SID> -r 1

• If the SQL is changing, then will soon get a good idea of what is going on:
  – Lots of different statements?
  – Repeated statements? Maybe with different literal values?

• If different statements are executed, but see one SQL repeated frequently, then this may be the first statement to begin investigating
Long Running Queries

- Use the same approach with a background process that periodically checks how long all SQLs have been running and dump session information to a file when exceeds a threshold.

- Also use **Informix SQL Trace** to record queries and then retrieve those that ran the longest.

- SQL Trace is covered extensively elsewhere.
Reviewing Query Plans

• Now we have found some slow SQL, find out what it’s actually doing by obtaining a Query Plan

• Turn on Dynamic Explain to get plan for a session:

  onmode -Y <sid> <0|1|2> [filename]
  0=off
  1=plan + statistics on
  2=only plan on
**Dynamic Query Plans**

```
onmode -Y 10563 1

onstat -g ses
```

IBM Informix Dynamic Server Version 12.10.FC5AEE -- On-Line -- Up 1 days 12:01:36 -- 2947104 Kbytes

<table>
<thead>
<tr>
<th>session</th>
<th>id</th>
<th>user</th>
<th>tty</th>
<th>pid</th>
<th>hostname</th>
<th>threads</th>
<th>memory</th>
<th>used</th>
<th>#RSAM</th>
<th>total</th>
<th>used</th>
<th>dynamic explain</th>
</tr>
</thead>
<tbody>
<tr>
<td>session</td>
<td>10657</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>16384</td>
<td>12480</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>10653</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>16384</td>
<td>12480</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td><strong>10563</strong></td>
<td>informix</td>
<td>2</td>
<td>4243</td>
<td>apollo</td>
<td>1</td>
<td><strong>73728</strong></td>
<td><strong>64480</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>on</strong></td>
</tr>
<tr>
<td></td>
<td>10028</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>apollo</td>
<td>1</td>
<td>335872</td>
<td>321728</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>10011</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>apollo</td>
<td>1</td>
<td>241664</td>
<td>100072</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>626688</td>
<td>472280</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>43</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>626688</td>
<td>471576</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>618496</td>
<td>494080</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>informix</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>102400</td>
<td>86784</td>
<td></td>
<td></td>
<td></td>
<td>off</td>
</tr>
</tbody>
</table>

Set Dynamic Explain for Session 10563
Dynamic Query Plans

Explain plan written to a file in the user’s home directory with the SID in the name:

```
-rw-rw-rw- 1 informix informix  573 Apr  7 11:17 sqexplain.out.10563
```

```
cat sqexplain.out.10563
```

```
QUERY: (OPTIMIZATION TIMESTAMP: 04-07-2017 11:17:33)
      select * from snapshot
```

Estimated Cost: 79971
Estimated # of Rows Returned: 1199409

```
1) informix.snapshot: SEQUENTIAL SCAN
```

Query statistics:

```
Table map :
-----------------------------------
Internal name | Table name
-----------------------------------
t1 | snapshot

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>1203641</td>
<td>1199409</td>
<td>1203641</td>
<td>00:00.00</td>
<td>79971</td>
</tr>
</tbody>
</table>
```

Query Statistics shown because used:

```
onmode -Y <sid> 1
```
Dynamic Query Plans

• Using “onmode -Y” will not produce anything until the next statement runs – so no good for getting the explain plan for a single, long running statement

• Limited value if prepared SQL is being executed

• For a closer look and to start tuning, capture the SQL to a file, and get the explain plan for that…
SET EXPLAIN

• SET EXPLAIN ON / SET EXPLAIN OFF:

```
SET EXPLAIN ON;
SELECT * FROM x WHERE y = 10;
SET EXPLAIN OFF;
```

• By default, the query plan is written to the file: sqexplain.out

• File is created in the current directory (UNIX)

• If use client app, the file will be in home directory of the user that SQL was executed as

• File will be appended to each time more SQL is executed
SET EXPLAIN

slow1.sql:

set explain file to "slow1.exp";
set explain on;
output to /dev/null
select c.customer_num, o.order_num
from customer c, orders o
where c.customer_num = o.customer_num
  and c.company = "Play Ball!"
order by 2;

timex dbaccess -e stores_demo slow1.sql > slow1.out 2>&1 &
SET EXPLAIN

QUERY: (OPTIMIZATION TIMESTAMP: 04-09-2017 07:50:47)
------
select c.customer_num, o.order_num
from customer c, orders o
where c.customer_num = o.customer_num
  and c.company = "Play Ball!"
order by 2

Estimated Cost: 6
Estimated # of Rows Returned: 2
Temporary Files Required For: Order By

1) informix.c: SEQUENTIAL SCAN

  Filters: informix.c.company = 'Play Ball!

2) informix.o: INDEX PATH

  (1) Index Name: informix.102_4
    Index Keys: customer_num  (Serial, fragments: ALL)
    Lower Index Filter: informix.c.customer_num = informix.o.customer_num

NESTED LOOP JOIN
### Query statistics:

---

**Table map:**

<table>
<thead>
<tr>
<th>Internal name</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>c</td>
</tr>
<tr>
<td>t2</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>1</td>
<td>3</td>
<td>28</td>
<td>00:00.00</td>
<td>4</td>
</tr>
<tr>
<td>scan</td>
<td>t2</td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>00:00.00</td>
<td>0</td>
</tr>
<tr>
<td>nljoin</td>
<td></td>
<td>4</td>
<td>3</td>
<td></td>
<td>00:00.00</td>
<td>6</td>
</tr>
<tr>
<td>sort</td>
<td></td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>00:00.00</td>
<td>0</td>
</tr>
</tbody>
</table>

As long as query is allowed to complete, Query Statistics will be shown at the end of the plan

[ EXPLAIN_STAT=1 in ONCONFIG ]
SET EXPLAIN

For long running SQL or for Insert, Update or Delete operations, use "AVOID_EXECUTE" to get the explain plan without running the SQL:

```sql
set explain file to "slow2.exp";
set explain on avoid_execute;
update orders
set ship_instruct = null
where customer_num = 104;
```
set explain file to "slow2.exp";
Explain set.

set explain on avoid_execute;
Explain set.

update orders
set ship_instruct = null
where customer_num = 104;
0 row(s) updated.

Warning! avoid_execute has been set

Database closed.

If use AVOID_EXECUTE will NOT see the Query Statistics in the Explain Plan
Anatomy of a Query Plan

- Query SQL
- Cost/Rows Returned/Temp Files/Directives
  - Table 1: Name & Access Method
    - Table 1: Filters
    - Table 1: Index Info
  - Table 2: Name & Access Method
    - Table 2: Filters
    - Table 2: Index Info
  - Table 1 & 2: Join Method
- Subqueries
- Query Statistics (if enabled)
QUERY: (OPTIMIZATION TIMESTAMP: 04-09-2017 07:50:47)
------
select c.customer_num, o.order_num
from customer c, orders o
where c.customer_num = o.customer_num
  and c.company = "Play Ball!"
order by 2

Estimated Cost: 6
Estimated # of Rows Returned: 2
Temporary Files Required For: Order By

1) informix.c: SEQUENTIAL SCAN
   Filters: informix.c.company = 'Play Ball!'

2) informix.o: INDEX PATH
   (1) Index Name: informix. 102_4
       Index Keys: customer_num  (Serial, fragments: ALL)
       Lower Index Filter: informix.c.customer_num = informix.o.customer_num
       NESTED LOOP JOIN
select c.cust_id, c.cust_name, o.order_id
from customer c, order o
where c.cust_id = o.cust_id
and c.cust_type = 'PREF'
and o.pay_type != 'CREDCARD'
and o.ship_method = 'FEDEX'

Estimated Cost: 51207
Estimated # of Rows Returned: 9000

1) informix.c: SEQUENTIAL SCAN
   
   Filters: informix.c.cust_type = 'PREF'

2) informix.o: INDEX PATH

   Filters: (informix.o.ship_method = 'FEDEX' AND informix.o.pay_type != 'CREDCARD')

   (1) Index Name: informix.order_ix2
   Index Keys: cust_id  (Serial, fragments: ALL)
   Lower Index Filter: informix.c.cust_id = informix.o.cust_id

   NESTED LOOP JOIN
Query Plans – Breaking it Down

1) `informix.c`: SEQUENTIAL SCAN

   Filters: `informix.c.cust_type = 'PREF'`

2) `informix.o`: INDEX PATH

   Filters: (`informix.o.ship_method = 'FEDEX' AND informix.o.pay_type != 'CREDCARD'`)

(1) Index Name: `informix.order_ix2`
   Index Keys: cust_id  (Serial, fragments: ALL)
   Lower Index Filter: `informix.c.cust_id = informix.o.cust_id`

   NESTED LOOP JOIN

---

**customer**

Read ALL Rows

**order**

Read rows for cust_id

Reject those that don’t match filters

cust_type = ‘PREF’
Query Plans – Breaking it Down

**customer**
Read ALL Rows

**order**
Read rows for cust_id
Reject those that don’t match filters

```sql
SELECT count(*)
FROM customer
WHERE cust_type = 'PREF';
```

(Count(*))
100000

```sql
SELECT count(*)
FROM customer;
```

(Count(*))
25000

```sql
SELECT count(*)
FROM customer c, order o
WHERE c.cust_id = o.cust_id
AND c.cust_type = 'PREF';
```

(Count(*))
250000

**Results of query...**

6040 row(s) retrieved.
Query Plans

customer (all rows) : 100000 rows
customer.cust_type = ‘PREF’ : 25000 rows
Joins to order table : 250000 rows
Rows returned from query : 6040 rows

Number of Rows Read

vs

Number of Rows Returned

Lots of rows read...and then discarded!
Try and make the number of rows read as close as possible to those that are needed
### Query Statistics

#### Table map:

<table>
<thead>
<tr>
<th>Internal name</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>c</td>
</tr>
<tr>
<td>t2</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>25000</td>
<td>10000</td>
<td>100000</td>
<td>00:00.41</td>
<td>15501</td>
</tr>
<tr>
<td>scan</td>
<td>t2</td>
<td>6040</td>
<td>90000</td>
<td>250000</td>
<td>00:14.76</td>
<td>4</td>
</tr>
<tr>
<td>nljoin</td>
<td></td>
<td>6040</td>
<td>9000</td>
<td></td>
<td>00:15.24</td>
<td>51207</td>
</tr>
</tbody>
</table>

**Rows Read**

**Rows Needed (after filtering)**

**Table Scan, but fast!**

This is where the time was spent

---

**Review the “Filters”**

Including the filter columns in the index will reduce the rows scanned and the rows discarded.
Query Statistics

```
Query statistics:-----------------
Table map:----------------------------------------
<table>
<thead>
<tr>
<th>Internal name</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>c</td>
</tr>
<tr>
<td>t2</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>25000</td>
<td>10000</td>
<td>100000</td>
<td>00:00.40</td>
<td>15501</td>
</tr>
<tr>
<td>scan</td>
<td>t2</td>
<td>6040</td>
<td>128571</td>
<td>6100</td>
<td>00:01.15</td>
<td>1</td>
</tr>
<tr>
<td>nljoin</td>
<td></td>
<td>6040</td>
<td>12858</td>
<td>00:01.61</td>
<td>25312</td>
<td></td>
</tr>
</tbody>
</table>
```

New index includes ship_method
create index order_ix3 on order(cust_id, ship_method)

Rows Read is much closer to the Rows Needed – fewer rows discarded

Faster!
Sequential Scans

• If a Query Plan contains a Sequential Scan, all rows of the table are read (before any filter is applied)

• Don’t freak out!
  – If most of the rows read from the table are needed, then it may be okay
  – Consider that many indexed reads of data can be costly because of the read of the index, plus the read of the data page
Sequential Scans

A Scan of all Data Pages *may* be faster than lots of Indexed Reads

But it depends on how many rows are actually needed
A scan of a large table can trash the cache
Sequential Scans

1) informix.order: SEQUENTIAL SCAN

   Filters: (informix.order.ship_method = 'FEDEX' AND informix.order.pay_type = 'CREDCARD')

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>1000000</td>
<td>71429</td>
<td>1000000</td>
<td>00:04.55</td>
<td>280001</td>
</tr>
</tbody>
</table>

1) informix.order: INDEX PATH

   Filters: informix.order.ship_method = 'FEDEX'

   (1) Index Name: informix.order_ix4
   Index Keys: pay_type order_id  (Serial, fragments: ALL)
   Lower Index Filter: informix.order.pay_type = 'CREDCARD'

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>12200</td>
<td>71429</td>
<td>500000</td>
<td>00:06.16</td>
<td>2304336</td>
</tr>
</tbody>
</table>

**Rows Read**

**Fewer Rows**

**Slower!**
SELECT inventory.week_nr, <snip>, inventory.quantity
FROM product, inventory
WHERE inventory.prod_num = product.prod_num
    AND inventory.year_num = 2016
    AND inventory.quantity > 0
    AND TRIM(product.department || '-' || product.prod_type) IN ('A-1')
    AND inventory.store IN (201)

Estimated Cost: 6828412
Estimated # of Rows Returned: 2349350

1) informix.product: SEQUENTIAL SCAN

   Filters: TRIM ( BOTH ' ' FROM (((informix.product.department || '-' ) ||
informix.product.prod_type ))) = 'A-1'

2) informix.inventory: INDEX PATH

   Filters: informix.inventory.quantity > 0

(1) Index Name: informix.inventory_idx3
   Index Keys: prod_num store week_nr year_num  (Key-First)  (Serial, fragments: ALL)
   Lower Index Filter: (informix.inventory.prod_num =
informix.product.prod_num AND informix.inventory.store = prod_type )
   Index Key Filters:  (informix.inventory.year_num = 2016 )
NESTED LOOP JOIN
### Query statistics:
---

**Table map:**

<table>
<thead>
<tr>
<th>Internal name</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>product</td>
</tr>
<tr>
<td>t2</td>
<td>inventory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>737</td>
<td>67311</td>
<td>673240</td>
<td>00:01.61</td>
<td>49460</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scan</td>
<td>t2</td>
<td>242321</td>
<td>23493496</td>
<td>977563</td>
<td>02:26.49</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nljoin</td>
<td></td>
<td>242321</td>
<td>2349350</td>
<td>02:28.19</td>
<td></td>
<td>6828412</td>
</tr>
</tbody>
</table>

Despite the ugly scan of the product table, it doesn’t take long.

Time is spent reading inventory.

75% of the rows read are discarded.
Query Tuning Example

2) informix.inventory: INDEX PATH

Filters: informix.inventory.quantity > 0

(1) Index Name: informix.inventory_idx3
Index Keys: prod_num store week_nr year_num
(Serial, fragments: ALL)

Lower Index Filter: (informix.inventory.prod_num = informix.product.prod_num AND informix.inventory.store = 201 )
Index Key Filters: (informix.inventory.year_num = 2016 )

NESTED LOOP JOIN

- prod_num is supplied from table 1 (good)
- store has a literal value (good)
- year_num has a literal value (good), BUT it’s position in the index is after another column (week_nr), so it is used as a filter, but not for drilling into the index (Key-First). Index pages will be read and discarded (bad)
- quantity is not in the index. A jump to the data page is needed to read the value to apply the filter. Rows will be read and discarded (bad)
Query Tuning Example

• New Index:
  inventory( prod_num, store, year_num, quantity, week_nr)

• `year_num` can now be used for drilling down in the index

• `quantity` is now in the index. Query uses a “>” operator, but can be used for scanning the index leaf nodes, PLUS there is no need to check the data page

• `week_nr` is not needed to filter records, but is used in the select clause. Now no need to go to the data page at all!
Query Tuning Example – Key-Only

Key-Only
With a Key-only read, all the columns needed to satisfy the query are in the index. There is no need to read the data page.
Index must also include columns used by the select clause and order by to get a key-only
Key-only reads are very fast!
Query Tuning Example – Key-Only

Query statistics:
-----------------

Table map:
---------------------------------------------
| Internal name | Table name |
---------------------------------------------
| t1            | product    |
| t2            | inventory  |

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>737</td>
<td>67311</td>
<td>673240</td>
<td>00:01.66</td>
<td>49460</td>
</tr>
<tr>
<td>scan</td>
<td>t2</td>
<td>242321</td>
<td>23496424</td>
<td>242321</td>
<td>00:00.82</td>
<td>2</td>
</tr>
<tr>
<td>nljoin</td>
<td></td>
<td>242321</td>
<td>2349645</td>
<td>00:02.54</td>
<td>199938</td>
<td></td>
</tr>
</tbody>
</table>

With the new index, the query execution dropped from 2.5 minutes to 2.5 seconds.

Previously 2m 26s

All rows read are rows that are needed.
Correlated Sub-Queries

**Correlated**

```sql
select c.*
from customer c
where exists (
    select "X"
    from order o
    where o.custid = c.custid
    and o.stat = "OPEN"
)
```

Outer query referenced in Inner query...
Inner query must be repeated for each row returned by the outer query

**Non-Correlated**

```sql
select unique c.*
from customer c,
    order o
where c.custid = o.custid
    and o.stat = "OPEN"
```

```sql
select c.*
from customer c
where custid in (
    select custid
    from order o
    where o.stat = "OPEN"
)
```
Correlated Sub-Queries

... AND NOT EXISTS
(SELECT C.cust_key FROM customer C
 WHERE C.flag_1 = 0 AND C.flag_2 = 0
 AND C.cust_key = ST.cust_key)

1) informix.st: INDEX PATH
   Filters: (informix.st.cust_status IN ('F', 'Q', 'H')
   AND NOT EXISTS <subquery> )
   (1) Index Name: informix.site_x02
       Index Keys: ssn site  (Key-First)  (Serial, fragments: ALL)
       Lower Index Filter: informix.st.ssn > 'A'
       Index Key Filters: (informix.st.site = 210 )

Subquery:
     _________
     Estimated Cost: 9236
     Estimated # of Rows Returned: 33

1) informix.sm: SEQUENTIAL SCAN
   Filters: ((informix.c.cust_key = informix.st.cust_key AND
   informix.c.flag_2 = 0 ) AND informix.c.flag_1 = 0 )
Correlated Sub-Queries

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>1</td>
<td>85</td>
<td>3265281</td>
<td>04:05.61</td>
<td>816703</td>
</tr>
</tbody>
</table>

Subquery statistics:

Table map:

<table>
<thead>
<tr>
<th>Internal name</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>c</td>
</tr>
</tbody>
</table>

Cumulative values

<table>
<thead>
<tr>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>270015447</td>
<td>04:04.08</td>
<td>9236</td>
</tr>
</tbody>
</table>
Correlated Sub-Queries

... AND NOT EXISTS
(SELECT C.cust_key FROM customer C
 WHERE C.flag_1 = 0 AND C.flag_2 = 0
 AND C.cust_key = ST.cust_key)

AND ST.cust_key NOT IN
(SELECT C.cust_key FROM customer C
 WHERE C.flag_1 = 0 AND C.flag_2 = 0)

Change EXISTS to a NOT IN and remove the join

No longer a Correlated Subquery
Subquery will execute one time only
Correlated Sub-Queries

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>2</td>
<td>171</td>
<td>3265356</td>
<td>00:01.79</td>
<td>36899</td>
</tr>
</tbody>
</table>

Subquery statistics:

Table map:

<table>
<thead>
<tr>
<th>Internal name</th>
<th>Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>c</td>
</tr>
</tbody>
</table>

Table: t1

<table>
<thead>
<tr>
<th>type</th>
<th>table</th>
<th>rows_prod</th>
<th>est_rows</th>
<th>rows_scan</th>
<th>time</th>
<th>est_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>t1</td>
<td>8193</td>
<td>330</td>
<td>32981</td>
<td>00:00.04</td>
<td>9236</td>
</tr>
</tbody>
</table>

Fraction of records read & much faster!

With the change, the query execution dropped from 4m 5s to less than 2 seconds
Correlated Sub-Queries

- Sometimes a CSQ can be a good thing
- Adding a join to a subquery can reduce the data set returned
- Efficiencies made to a subquery will be compounded when executed repeatedly
Optimizer Directives

• Change the generated query plan by removing paths from consideration
  – Will not be ignored (as long as they are valid)
  – Negative directives (Don’t do something)

• Great tool for tuning queries
Include the directive as a comment in the SQL, followed by a “+”:

SELECT --+ directive text
SELECT {+ directive text }  
SELECT /*+ directive text*/
Optimizer Directives – 4GL

Can be used in 4GL, but must PREPARE the SQL with the Directives so that it is submitted to the database

```plaintext
let sql_string =
    'select {+ USE_HASH(o) } c.state, max(order_date) ',
    'from order o, cust c ',
    'where o.cust_id = c.cust_id ',
    'group by 1 ',
    'order by 1'

prepare sql_do_unl from sql_string
declare curs_do_unl cursor for sql_do_unl
```

Advanced DataTools
Types of Directives

- Access Methods
- Join Order
- Join Methods
- Star Join
- Optimization Goal
- Explain Plan
- Statement Cache

Not all directives are available in all engine versions.
The following are for 12.10.
# Types of Directives: Access Methods

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FULL</strong></td>
<td>Performs a full-table scan</td>
</tr>
<tr>
<td><strong>INDEX</strong></td>
<td>Uses the index specified to access the table</td>
</tr>
<tr>
<td><strong>INDEX_ALL or MULTI_INDEX</strong></td>
<td>Access the table using the specified indexes (Multi-index scan)</td>
</tr>
<tr>
<td><strong>INDEX_SJ</strong></td>
<td>Use the specified index to scan the table in an index self-join path.</td>
</tr>
<tr>
<td><strong>AVOID_FULL</strong></td>
<td>No full-table scan on the listed table</td>
</tr>
<tr>
<td><strong>AVOID_INDEX</strong></td>
<td>Does not use any of the specified indexes</td>
</tr>
<tr>
<td><strong>AVOID_INDEX_SJ</strong></td>
<td>Does not use an index self-join path for the specified indexes</td>
</tr>
<tr>
<td><strong>AVOID_MULTI_INDEX</strong></td>
<td>Does not use a multi-index scan path for the specified table</td>
</tr>
</tbody>
</table>
Types of Directives: Join Order

**ORDERED** Join tables or views in the order in which they are referenced in the FROM clause of the query
Types of Directives: Join Methods

**USE_NL**
Forces nested loop join on specified tables

**USE_HASH**
Forces hash join on specified tables

**AVOID_NL**
Avoids nested loop join on specified tables

**AVOID_HASH**
Avoids hash join on specified tables
# Types of Directives: Star Join

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AVOID_FACT</strong></td>
<td>At least one table must be specified. Do not use the table (or any table in the list of tables) as a fact table in star-join optimization.</td>
</tr>
<tr>
<td><strong>AVOID_STAR_JOIN</strong></td>
<td>The optimizer does not consider a star-join execution plan.</td>
</tr>
<tr>
<td><strong>FACT</strong></td>
<td>Exactly one table must be specified. Only consider the specified table as a fact table in the star-join execution plan.</td>
</tr>
<tr>
<td><strong>STAR_JOIN</strong></td>
<td>Favor a star-join plan, if one is possible.</td>
</tr>
</tbody>
</table>
Types of Directives: Optimization Goal

**FIRST_ROWS \((N)\)**  
Tells the optimizer to choose a plan optimized to return the first \(N\) rows of the result set

**ALL_ROWS**  
Tells the optimizer to choose a plan optimized to return all of the results

“Query level” equivalent of:
- OPT_GOAL configuration parameter (instance level)
  - 0=First Rows, -1=All Rows (default)
- OPT_GOAL environment variable (environment level)
- SET OPTIMIZATION statement (session level)
  - FIRST_ROWS, ALL_ROWS
Types of Directives: Explain Plan

**EXPLAIN**
Turns SET EXPLAIN ON for the specified query

**EXPLAIN AVOID_EXECUTE**
Prevents the data manipulation statement from executing; instead, the query plan is printed to the explain output file
Types of Directives: Statement Cache

AVOID_STMT_CACHE

Prevent the statement from being stored in the statement cache. Forces the optimizer to reoptimize the statement every time that the statement is run.
Directives Examples: ORDERED

```
select /*+ ORDERED */
customer.lname, orders.order_num, items.total_price
from customer, orders, items
where customer.customer_num = orders.customer_num
    and orders.order_num = items.order_num
    and items.stock_num = 6 and items.manu_code = "SMT"
```

DIRECTIVES FOLLOWED:
ORDERED

DIRECTIVES NOT FOLLOWED:

Estimated Cost: 15
Estimated # of Rows Returned: 1

1) `informix.customer`: SEQUENTIAL SCAN

2) `informix.orders`: INDEX PATH
   (1) Index Name: `informix.102_4`
       Index Keys: `customer_num` (Serial, fragments: ALL)
       Lower Index Filter: `informix.customer.customer_num = informix.orders.customer_num`
       NESTED LOOP JOIN

3) `informix.items`: INDEX PATH
   (1) Index Name: `informix.105_12`
       Index Keys: `stock_num manu_code` (Serial, fragments: ALL)
       Lower Index Filter: `(informix.items.manu_code = 'SMT' AND informix.items.stock_num = 6 )

DYNAMIC HASH JOIN
  Dynamic Hash Filters: `informix.orders.order_num = informix.items.order_num`
Directives Examples: Combine Directives

```sql
select /*+ ordered index(customer, zip_ix)
    avoid_index(orders," 102_4") */
c.lname, o.order_num, i.total_price
from customer c, orders o, items i
where c.customer_num = o.customer_num
    and o.order_num = i.order_num
    and stock_num = 6
    and manu_code = "SMT"
```
Directives Examples : Errors

Check the Explain Plan to make sure that the directives were followed as expected:

```

DIRECTIVES FOLLOWED:
ORDERED
INDEX ( customer zip_ix )
```

```
DIRECTIVES NOT FOLLOWED:
AVOID_INDEX ( orders 101_4 ) Invalid Index Name Specified.
```

The query will still be executed even with invalid directives
Optimizer Directives: Pros & Cons

Pros:

• Forces the engine to execute the SQL the way that we want
• Sometimes we know better!!
• Great for testing different plans.  What if..?  

Cons:

• Forces the engine to execute the SQL the way that we want
• Sometimes the engine knows better!!
• If new indexes are added, number of rows changes significantly, or data distributions change…then a better execution plan may be available
Bufferpool Usage

- Obvious performance benefit when queries are satisfied from the cache instead of disk
- Monitoring the bufferpool can give insight into proper sizing and direct you to problem tables, problem queries and missing indexes
Bufferpool Turnover

• Measure the turnover of pages in the bufferpool
  – Art Kagel’s calculation of Buffer Turnover
    $\text{Buffer Turnover Ratio} = \frac{((\text{bufwrits} + \text{pagreads}) / \text{number of buffers})}{\text{time\_since\_profile\_reset}}$
  – Aim for below 10 - Lower is better
  – See ratios script on IIUG site for this and other performance ratios
Bufferpool Turnover

BTR Calculation:

```
select bufsize,nbuffs, 
    round(((( pagreads + bufwrites ) 
    /nbuffs ) / ( 
        select (ROUND (((
            sh_curtime - sh_pfclrttime)/60)/60) ) 
    from syssshmvals ) ),1) BTR 
from sysbufpool;
```
### Bufferpool Turnover

<table>
<thead>
<tr>
<th>bufsize</th>
<th>nbuffs</th>
<th>btr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>250000</td>
<td>4.9</td>
</tr>
<tr>
<td>4096</td>
<td>50000</td>
<td>820.6</td>
</tr>
<tr>
<td>16384</td>
<td>10000</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Way too high!
Bufferpool Turnover

Useful to measure BTR *over time* to identify peak periods or look for changes
Bufferpool Turnover

Measure the turnover hourly using the delta of values to see when periods of higher turnover happen.

The above showed that the bufferpool turnover spikes in the early morning hours during report generation.
Bufferpool Use

• Find out what objects are using the bufferpool
• Look for tables/indexes that dominate the bufferpool
• Watch over time to see what swaps in/out
• Identify the troublemakers

Don’t assume that increasing the size of the bufferpool is the answer
onstat -P : Print partition buffer summary

<table>
<thead>
<tr>
<th>partnum</th>
<th>total</th>
<th>btree</th>
<th>data</th>
<th>other</th>
<th>dirty</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28</td>
<td>0</td>
<td>12</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>1048577</td>
<td>600</td>
<td>0</td>
<td>586</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>1048578</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1048579</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51380228</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>51380229</td>
<td>7</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Totals:</td>
<td>28674</td>
<td>915</td>
<td>3280</td>
<td>24479</td>
<td>83</td>
</tr>
</tbody>
</table>

Percentages:
- Data 11.44
- Btree 3.19
- Other 85.37
Bufferpool Use

Once have identified the tables that frequently consume a large part of the cache, or appear and disappear from the top-10, then can start tuning:

– Size the bufferpool to accommodate the commonly used tables to keep them in the cache
– Can tables be isolated to a different bufferpool?
– For tables that frequently trash the cache, identify queries that may be performing scans of the table
– Look at table level counters to get a better picture of what is happening
Table Counters

• Get information at a table fragment/index level
• `sysmaster:sysptprof` (view between systabnames & sysptntab)
• Useful way to identify tables that may be performance bottlenecks
• Reset counters with `onstat -z` or restart instance
• Need `TBLSPACE_STATS 1` set in `ONCONFIG`
### Table Counters - sysmaster:sysptprof

<table>
<thead>
<tr>
<th>dbsname</th>
<th>adtc_monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabname</td>
<td>snapshot</td>
</tr>
<tr>
<td>partnum</td>
<td>9437506</td>
</tr>
<tr>
<td>lockreqs</td>
<td>197049665</td>
</tr>
<tr>
<td>lockwts</td>
<td>0</td>
</tr>
<tr>
<td>deadlks</td>
<td>0</td>
</tr>
<tr>
<td>lktouts</td>
<td>0</td>
</tr>
<tr>
<td>isreads</td>
<td>180714794</td>
</tr>
<tr>
<td>iswrites</td>
<td>9271</td>
</tr>
<tr>
<td>isrewrites</td>
<td>4087</td>
</tr>
<tr>
<td>isdelete</td>
<td>17</td>
</tr>
<tr>
<td>bufreads</td>
<td>268432796</td>
</tr>
<tr>
<td>bufwrites</td>
<td>8661</td>
</tr>
<tr>
<td>seqscans</td>
<td>11459</td>
</tr>
<tr>
<td>pagreads</td>
<td>1273469</td>
</tr>
<tr>
<td>pagwrites</td>
<td>709</td>
</tr>
</tbody>
</table>

#### Identification

#### Locks

#### I/O
Table Counters – sysptprof – I/O

• Sort the results by pagreads/pagwrites to find those tables which result in high disk i/o

  – Use dbinfo("DBSPACE", partnum) to get the dbspace. Useful for fragmented tables.

```
select dbsname,
      tabname,
      bufreads,
      bufwrites,
      pagreads,
      pagwrites,
      dbinfo("DBSPACE", partnum) dbspace
from sysptprof
order by (pagreads+pagwrites) desc;
```
Table Counters – Sequential Scans

**seqscans** number of scans performed against this table

– Sorting by the number of scans alone will also include those small tables for which a scan is more efficient

– Combine with the **size** of the table to identify scans of large tables. Divide by the length of time since the counters were reset to get KB scanned/hour
  
  • Use systabinfo to get the size of a table - does not rely on update statistics
Sequential Scans - History

As with other metrics, keeping a history can help diagnose problems and determine if something is still a problem.

<table>
<thead>
<tr>
<th>date</th>
<th>rows</th>
<th>scans</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/03/2015</td>
<td>210970</td>
<td>625</td>
</tr>
<tr>
<td>01/04/2015</td>
<td>211192</td>
<td>625</td>
</tr>
<tr>
<td>01/05/2015</td>
<td>211218</td>
<td>625</td>
</tr>
<tr>
<td>01/06/2015</td>
<td>220488</td>
<td>918</td>
</tr>
<tr>
<td>01/07/2015</td>
<td>230141</td>
<td>1110</td>
</tr>
<tr>
<td>01/08/2015</td>
<td>239930</td>
<td>1438</td>
</tr>
<tr>
<td>01/09/2015</td>
<td>249377</td>
<td>38353</td>
</tr>
<tr>
<td>01/10/2015</td>
<td>258954</td>
<td>251260</td>
</tr>
<tr>
<td>01/11/2015</td>
<td>259301</td>
<td>251261</td>
</tr>
<tr>
<td>01/12/2015</td>
<td>259451</td>
<td>251261</td>
</tr>
<tr>
<td>01/13/2015</td>
<td>269028</td>
<td>251512</td>
</tr>
<tr>
<td>01/14/2015</td>
<td>278812</td>
<td>251682</td>
</tr>
<tr>
<td>01/15/2015</td>
<td>287936</td>
<td>251939</td>
</tr>
</tbody>
</table>

Sudden jump, then very few scans.
Use `sysptprof` to review counters for a specific table/index, while monitoring or capturing SQL to see values changing over time.

<table>
<thead>
<tr>
<th>Table/Idx</th>
<th>isreads</th>
<th>iswrites</th>
<th>isrewrt</th>
<th>isdel</th>
<th>bufreads</th>
<th>bufwr</th>
<th>pagreads</th>
<th>pagwr</th>
<th>scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>contract</td>
<td>889739</td>
<td>1940</td>
<td>289</td>
<td>176</td>
<td>1218475899</td>
<td>1129</td>
<td>152563</td>
<td>964</td>
<td>869301</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>contract</td>
<td>889741</td>
<td>1940</td>
<td>289</td>
<td>176</td>
<td>1218478853</td>
<td>1129</td>
<td>152563</td>
<td>964</td>
<td>869303</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>contract</td>
<td>889743</td>
<td>1940</td>
<td>289</td>
<td>176</td>
<td>1218481807</td>
<td>1129</td>
<td>152563</td>
<td>964</td>
<td>869305</td>
</tr>
</tbody>
</table>

Read operations are increasing, reads are satisfied from buffers and are the result of table scans.
Update Statistics

• The Database Optimizer uses statistics gathered on the tables, their columns and indexes to determine which is the best query plan to use.

• If the statistics don’t exist or are inaccurate, then Informix may choose a poor query plan.

• Make sure that statistics are rerun each time substantial changes are made to a table – including the number of rows and values in a column.
  
  – This includes populating temp tables inside of applications.
Update Statistics

• The basic (and most important) statistic is the number of rows in a table
  – Update statistics LOW for the table to update this statistic (stored in systables.nrows)

• Data Distributions denotes how unique or how common particular values are within a field
  – Update statistics MEDIUM performs a sampling of the records to determine distributions
  – Update statistics HIGH scans all records to determine distributions
How Fresh are your Stats?

• Basic stat of how many rows in a table is stored in <database>.systables.nrows

• The time that LOW stats were last updated is now recorded in the column ustlowts
How Fresh are your Stats?

select
    tabname[1,20],
    ustlowts,
    round(nrows) nrows
from systables
where tabtype = "T";

<table>
<thead>
<tr>
<th>tabname</th>
<th>ustlowts</th>
<th>nrows</th>
</tr>
</thead>
<tbody>
<tr>
<td>systables</td>
<td>2017-04-09 01:12:08.00000</td>
<td>91</td>
</tr>
<tr>
<td>syscolumns</td>
<td>2017-04-09 01:12:08.00000</td>
<td>629</td>
</tr>
<tr>
<td>sysindices</td>
<td>2017-04-09 01:12:28.00000</td>
<td>130</td>
</tr>
<tr>
<td>systabauth</td>
<td>2017-04-09 01:12:08.00000</td>
<td>91</td>
</tr>
<tr>
<td>syscolauth</td>
<td>2017-04-09 01:12:09.00000</td>
<td>44</td>
</tr>
<tr>
<td>sysviews</td>
<td>2017-04-09 01:12:09.00000</td>
<td>5</td>
</tr>
<tr>
<td>sysusers</td>
<td>2017-04-09 01:12:26.00000</td>
<td>1</td>
</tr>
<tr>
<td>sysdepend</td>
<td>2017-04-09 01:12:09.00000</td>
<td>2</td>
</tr>
<tr>
<td>syssynonymys</td>
<td>2017-04-09 01:12:09.00000</td>
<td>0</td>
</tr>
<tr>
<td>syssyntable</td>
<td>2017-04-09 01:12:09.00000</td>
<td>0</td>
</tr>
</tbody>
</table>
How Accurate are your (LOW) Stats?

Compare the **estimated** row counts in systables.nrows with the **actual** row counts:

```sql
select first 25 dbsname[1,12], tabname[1,20], est_nrows::int est, act_nrows::int act,
  case
    when est_nrows = 0 and act_nrows > 0 then 100::smallint
    when est_nrows = act_nrows then 0
    else round(abs(act_nrows - est_nrows)/est_nrows*100)
  end diff
from ((
  select n.dbsname, t.tabname, round(t.nrows) est_nrows, sum(i.ti_nrows) act_nrows
  from systables t, sysmaster:systabnames n, sysmaster:systabinfo i
  where t.tabname = n.tabname
    and n.dbsname = "<database name>"
    and n.partnum = i.ti_partnum
    and t.tabtype = "T"
  group by 1,2,3))
order by diff desc, act desc;
```

<table>
<thead>
<tr>
<th>dbname</th>
<th>tablename</th>
<th>est</th>
<th>act</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>stores_demo</td>
<td>sysattrtatypes</td>
<td>5</td>
<td>16</td>
<td>220</td>
</tr>
<tr>
<td>stores_demo</td>
<td>sysxtddesc</td>
<td>3</td>
<td>4</td>
<td>33</td>
</tr>
<tr>
<td>stores_demo</td>
<td>sysxtdtdtypes</td>
<td>24</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>stores_demo</td>
<td>sysprocbody</td>
<td>3763</td>
<td>3763</td>
<td>0</td>
</tr>
</tbody>
</table>

If actual and estimates are significantly different, then update low stats (at least)
How Fresh are your Stats?

Checking the column distributions

select t.tabname[1,18], c.colname[1,18],
  d.constr_time::datetime year to minute constructed,
  d.mode,
  d.resolution::decimal(5,2) res ,
  d.confidence::decimal(5,2) conf
from sysdistrib d, systables t, syscolumns c
where d.tabid = t.tabid
  and t.tabid = c.tabid
  and d.colno = c.colno
  and d.segno = 1
order by t.tabid, c.colno;

<table>
<thead>
<tr>
<th>tabname</th>
<th>colname</th>
<th>constructed</th>
<th>mode</th>
<th>res</th>
<th>conf</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>item_num</td>
<td>2017-03-18 01:11 H</td>
<td>0.50</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>items</td>
<td>order_num</td>
<td>2017-03-18 01:11 H</td>
<td>0.50</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>items</td>
<td>stock_num</td>
<td>2017-03-18 01:11 H</td>
<td>0.50</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>items</td>
<td>manu_code</td>
<td>2017-03-18 01:11 M</td>
<td>2.00</td>
<td>0.95</td>
<td></td>
</tr>
</tbody>
</table>
Next Webcast
Informix Best Practices

• Schema Design Tips - by Art Kagel
  – Thursday, November 30, 2017 2:00pm EST

• Getting Started with Informix Enterprise Replication – by Tom Beebe
  – Thursday, December 14, 2017 2:00pm EST

Please register for each webcast here at:
http://advanceddatatools.com/Informix/NextWebcast.html
Informix Training 2018

– Advanced Informix Performance Tuning
  • February 5-8, 2018

– Informix for Database Administrators
  • April 23-26, 2018

• All courses can be taken online on the web from your desk or at our training center in Virginia.
• We guarantee to NEVER cancel a course and will teach a course as long as one student is registered!
• Please register early as the last two courses have filled up and we have not been able to accommodate everyone.

http://advanceddatatools.com/Training/InformixTraining.html
Coming Soon…

New Training Servers

Each Student in class will have a server running Informix with:
- 8 Cores
- 8 – 62 GB RAM
- 1 SSD Disk
- 1-4 more disks
Questions?

Send follow-up questions to mike@advanceddatatools.com
Informix Support and Training from the Informix Champions!

Advanced DataTools is an Advanced Level IBM Informix Data Management Partner, and has been an authorized Informix partner since 1993. We have a long-term relationship with IBM, we have priority access to high-level support staff, technical information, and Beta programs. Our team has been working with Informix since its inception, and includes 8 Senior Informix Database Consultants, 4 IBM Champions, 2 IIUG Director’s Award winners, and an IBM Gold Consultant. We have Informix specialists Lester Knutsen and Art Kagel available to support your Informix performance tuning and monitoring requirements!

- Informix Remote DBA Support Monitoring
- Informix Performance Tuning
- Informix Training
- Informix Consulting
- Informix Development

Free Informix Performance Tuning Webcast replays at:
http://advanceddatatools.com/Informix/Webcasts.html
Email: info@advanceddatatools.com
Web: http://www.advanceddatatools.com
Thank You

Mike Walker
Advanced DataTools Corporation
mike@advancedatatools.com

For more information:
http://www.advancedatatools.com