Tune your Informix database for top performance, Part 1: Tuning the data model and the application

A case study for Informix database performance using the "Fastest DBA" benchmark

Jack Parker
April 14, 2011

Author Jack Parker analyzes the "Fastest DBA benchmark" to explain some basic principles of how to tune your IBM® Informix® database application to get the best possible performance. In Part 1, of the series, he examines how the data model and the application, together with ensuring that you have updated statistics, play into performance.

View more content in this series

Introduction

Of late, I have found myself called in on a number of performance-related issues and thought I should teach a class on it. Since I'm too lazy to come up with my own benchmark, and since it would be more useful to dissect a widely used benchmark, I thought a good study would be the "Fastest DBA benchmark" used by Lester Knutsen and Advanced DataTools. For those of you unfamiliar with this, this is a periodic contest thrown by Advanced DataTools, where they come up with a slow-running application and challenge participants to tune it. I've included links to details about this contest in the Related topics section.

I also realized that classes are not as useful as the written word, so what you are now reading is an exploration of how to tune a database based on a dissection of the Fastest DBA benchmark. We will execute the benchmark, note performance issues, address each issue individually, and attempt the benchmark in a final pass with all of the issues addressed. Note that this is the second benchmark; there will be a follow-on article that dissects the third (and current) benchmark.

This article is not about installing the benchmark — the README file is fairly comprehensive. I had two issues putting it on my MacBook, the first being the SHMBASE for the engine. I had to edit the source $ONCONFIG file and set that properly (as it is re-copied over with each setup.sh execution). The second issue I had was that my Informix user defaults to the 'admin' group, so I had to manually change ownership on the dbspaces to Informix:Informix.
The execution of the benchmark as it is delivered will not complete in a timely manner. The benchmark itself is simple enough: it loads a customer table, then generates bills for those customers. The final kicker is an update to the bill table which takes quite some time. The work is not that complicated, but it provides an opportunity to look at how performance can be improved with tuning.

**Where to start?**

Typically, in any performance tuning project the first issue is one of problem statement. How do you know when you have tuned the problem away? How can you measure performance and determine that it has improved, and by how much? For the purposes of this article, I will contend that the benchmark takes too long to run and that we need it to execute in under an hour. Furthermore, we will capture database metrics at each stage to determine whether we made it better.

The second issue of database tuning is determining what to tune. We will usually see the best results from correcting:

1. The data model
2. The application
3. The disk
4. And finally the engine.

Since I am running on a single disk on a MacBook Pro, there will not be much that I can do in terms of disk tuning. Things to keep an eye on are CPU, disk, and memory usage.

**Running the benchmark**

Once you have the benchmark installed, execute an `onstat -z` to clear the engine metrics so you can be sure you are seeing only what the benchmark does to the engine.

You will note that the creators of the benchmark were kind enough to turn on the explain plan, so that is built in the current directory. There is a timer on the entire run.sh script, but it might be useful to have intermediate times for each step, so we will edit benchmark_run.sql and insert timestamps after each block of code with following statement:

```sql
select current hour to second from systables where tabid=1;
```

Next, start the benchmark with `run.sh`. In a separate window, start an `onstat -pr`. This will print out engine metrics every 5 seconds. You can then sit next to the benchmark and monitor it or go home for the evening or weekend.

Our first insert into the bills table (on my laptop) took 1m16s (the second took 1m17s) and the engine profile looks like this:

**Listing 1. Engine Performance Profile**

```
IBM Informix Dynamic Server Version 11.50.FC6DE -- On-Line --
```
This output is analogous to the oil dipstick in a car. In one brief page, it captures the state of the engine.

The top line (under Profile) covers disk and cache performance. Here, we are interested in the read and write cache hit rates. Read cache hits should be in excess of 99 percent, and we would like write cache hits to be above 95 percent. As you can no doubt see, we are not there. Since we haven't done much work yet and the cache numbers may not have settled, this may be OK. Although looking one line higher at the size of the engine in memory (118,812 KB), we aren't working with a lot of memory, let's keep cache on the table.

The second line (isamtot) indicates how much actual work has been performed (isamtot is total isam operations for example).

The third line (gp_*) describes how the engine is managing with Generic Pages or non-standard pages. Since we are not using any in this benchmark, the readings are all 0. I will remove this line from future dipstick readings.

The fourth line starts with the number of times we have exceeded locks, user threads or buffers (all 0 and should generally be so). The next two are the amount of User CPU and System CPU, the Number of Checkpoints and the number of Flushes.

The fifth line is of great interest:

### Table 1. Line 5 of engine profile

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufwaits</td>
<td>The number of times a process had to wait for a buffer, whether because there were not enough buffers or because the buffer in question was in use by someone else.</td>
</tr>
<tr>
<td>lokwaits</td>
<td>The number of times there was a wait for a lock to disengage.</td>
</tr>
<tr>
<td>lockreqs</td>
<td>The number of lock requests (number of rows or pages locked).</td>
</tr>
<tr>
<td>deadlks</td>
<td>The number of times deadlocks were detected.</td>
</tr>
</tbody>
</table>
Of interest here is a marginal number of buffer waits, and sequential scans, but a rather large number (at this point) of lock requests.

The sixth line tells us how read-ahead is working, and the number of waits for latches. Index and data read-aheads appear to all get consumed by the engine. Again, we see a rather large number (at this point) of latch requests.

Let's let the benchmark run a little longer and check again.

It's been two days. My laptop has gone to sleep a few times, so we haven't actually run for 48 hours (although the engine thinks it has been up all that time). Let's revisit our dipstick and see what it has to say.

**Listing 2. Performance after two days**

IBM Informix Dynamic Server Version 11.50.FC6DE -- On-Line --
Up 3 days 05:42:03 -- 127064 Kbytes

<table>
<thead>
<tr>
<th>dskreads</th>
<th>pagreads</th>
<th>bufreads</th>
<th>%cached</th>
<th>dskwrites</th>
<th>pagwrites</th>
<th>bufwrites</th>
<th>%cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>19966695480</td>
<td>19989379994</td>
<td>60891586765</td>
<td>67.21</td>
<td>660217</td>
<td>892639</td>
<td>3121469</td>
<td>78.85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>isamtot</th>
<th>open</th>
<th>start</th>
<th>read</th>
<th>write</th>
<th>rewrite</th>
<th>delete</th>
<th>commit</th>
<th>rollbk</th>
</tr>
</thead>
<tbody>
<tr>
<td>15495512</td>
<td>9115</td>
<td>1469626</td>
<td>6189708</td>
<td>759769</td>
<td>955295</td>
<td>1112</td>
<td>6097</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ovlock</th>
<th>ovusertread</th>
<th>ovbuff</th>
<th>usercpu</th>
<th>syscpu</th>
<th>numckpts</th>
<th>flushes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>221553.88</td>
<td>127876.85</td>
<td>1136</td>
<td>684</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bufwaits</th>
<th>llokwaits</th>
<th>lockreqs</th>
<th>deadlks</th>
<th>dltouts</th>
<th>ckpwaits</th>
<th>compress</th>
<th>seqscans</th>
</tr>
</thead>
<tbody>
<tr>
<td>364286149</td>
<td>0</td>
<td>59940823034</td>
<td>0</td>
<td>51</td>
<td>1707</td>
<td>102128</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ixda-RA</th>
<th>idx-RA</th>
<th>da-RA</th>
<th>RA-rgsused</th>
<th>lchwaits</th>
</tr>
</thead>
<tbody>
<tr>
<td>224274</td>
<td>0</td>
<td>19966058547</td>
<td>19966274629</td>
<td>4146547</td>
</tr>
</tbody>
</table>

The things we wanted to keep an eye on have ballooned out of control. (Note: Typically, the checkpoint wait number might give pause, but in this case, the benchmark is running a single SQL process, so that process is waiting on itself, and checkpoints are a more efficient way to write to disk anyway, so let's not get excited about that.)

Read cache rate is at 67 percent. Write cache rate is at 78 percent. While the amount of logical ISAM instructions we have performed is at 6 million reads, the number of lock requests (which is a physical indicator of what was read) is at 59 billion. The number of sequential scans is out of control as are the buffer waits and latch waits. We are doing a huge amount of read-ahead, which is getting consumed, and that is great, but why are we doing so much?

It looks like we are doing a sequential scan (of an entire table) to satisfy the SQL command, whatever it is. Let's take a closer look at that.
In the sysmaster database are two tables of current interest, sysptprof (Partition Profile) and syssesprof (Session Profile). Let's see what they offer us:

```sql
select * from sysptprof order by abs(seqscans) desc;
```

Note: If the number of sequential scans exceeds MAXINT (more than 2 billion), it will roll over to a negative number — hence, we ask for the absolute value. Here are the results from that select:

### Listing 3. Table performance profile

<table>
<thead>
<tr>
<th>dbname</th>
<th>benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabname</td>
<td>bills</td>
</tr>
<tr>
<td>partnum</td>
<td>3145799</td>
</tr>
<tr>
<td>lockreqs</td>
<td>136990885</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>2419974</td>
</tr>
<tr>
<td>iswrites</td>
<td>605280</td>
</tr>
<tr>
<td>isrewrites</td>
<td>853404</td>
</tr>
<tr>
<td>isdeletes</td>
<td>0</td>
</tr>
<tr>
<td>bufreads</td>
<td>1085720339</td>
</tr>
<tr>
<td>bufwrites</td>
<td>2071693</td>
</tr>
<tr>
<td>seqscans</td>
<td>99566</td>
</tr>
<tr>
<td>pagreads</td>
<td>-1398716020</td>
</tr>
<tr>
<td>pagwrites</td>
<td>534809</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dbname</th>
<th>benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabname</td>
<td>customer</td>
</tr>
<tr>
<td>partnum</td>
<td>3145795</td>
</tr>
<tr>
<td>lockreqs</td>
<td>727188</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>99566</td>
</tr>
<tr>
<td>iswrites</td>
<td>0</td>
</tr>
<tr>
<td>isrewrites</td>
<td>99562</td>
</tr>
<tr>
<td>isdeletes</td>
<td>0</td>
</tr>
<tr>
<td>bufreads</td>
<td>1898629</td>
</tr>
<tr>
<td>bufwrites</td>
<td>99562</td>
</tr>
<tr>
<td>seqscans</td>
<td>4</td>
</tr>
<tr>
<td>pagreads</td>
<td>727282</td>
</tr>
<tr>
<td>pagwrites</td>
<td>99562</td>
</tr>
</tbody>
</table>

Note how pagreads (page reads) has rolled over and gone negative. There is no way to tell how many times this has occurred during our run. Likewise, the lockreqs (lock requests) may have rolled over multiple times.

Out of 102,000 sequential scans (see onstat output above), 99,000 of those have been against this table. This table has ~600,000 rows in it. We have issued at least 136 million lock requests against it. (In fact, 20 minutes later, that number is at 934 million. This is a problem.) If we take 99,000*600,000, we get 59.4 billion, which, not coincidentally, makes up a large portion of our lock requests. In other words, we have sequentially scanned this table, locking every row, a lot of times. Since there are 100,000 customers, I must assume that we are close to the end of the current step. In fact, it just finished:
Unfortunately, we did not get a chance to look at the Session Profile table. This would have showed us similar information, but at a session level, so we could have seen that our session had done an inordinate number of sequential scans and lock requests.

**A common problem**

I will address this first problem separately because I saw it three times last week from colleagues. Our first major problem is that we are scanning the full bills table every time we want a row out of it and locking every row we read.

This sequential scan/lock nonsense has totally obscured real performance metrics. We cannot pay attention to read cache hit rate since we have been sequentially scanning a huge table over and over. Accordingly, let's address this before looking at what will be more realistic performance metrics.

The actual statement that took two days to run was:

```sql
update customer
    set balance_due = balance_due + ( select sum ( total_bill )
        from bills where bills.customer_number = customer.customer_number )
    where customer_number in ( select customer_number from bills );
```

If we look at the explain plan (located in sqexplain.out), we begin to see the problem (my notes in **bold**):

**Listing 4. Explain plan**

```sql
update customer
    set balance_due = balance_due + ( select sum ( total_bill )
        from bills where bills.customer_number = customer.customer_number )
    where customer_number in ( select customer_number from bills )
```

Estimated Cost: 3  -- The optimizer thought this would be cheap
Estimated # of Rows Returned: 10  With only 10 rows affected

1) informix.customer: INDEX PATH Excellent, an index was used to read this table

(1) Index Name: informix. 101_2
    Index Keys: customer_number  (Serial, fragments: ALL)
    Lower Index Filter: informix.customer.customer_number = ANY subquery

Subquery:

Estimated Cost: 2
Estimated # of Rows Returned: 1

1) informix.bills: SEQUENTIAL SCAN Here is our problem

    Filters: informix.bills.customer_number = informix.customer.customer_number

Subquery:
Estimated Cost: 2  The optimizer thinks this will be cheap. Why?
Estimated # of Rows Returned: 1

1) informix.bills: SEQUENTIAL SCAN

What is wrong with the bills table that the optimizer thinks it should sequentially scan the entire table? Let's have a look at the table.

Select * from benchmark:systables where tabname = 'bills';

<table>
<thead>
<tr>
<th>tabname</th>
<th>bills</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowsize</td>
<td>1266</td>
</tr>
<tr>
<td>nrows</td>
<td>0.00</td>
</tr>
<tr>
<td>created</td>
<td>08/23/2010</td>
</tr>
<tr>
<td>ustlowts</td>
<td>2010-08-25 20:52:22.00000</td>
</tr>
</tbody>
</table>

I've excluded a lot of excess information here to show you two key pieces of information. According to the system catalogs, the bills table is empty. "ustlowts" is null, which indicates that statistics have never been run against this table. In fact, if we go look at the benchmark code, we will see that the table is created, populated, and used, but statistics are never updated. This is Database 101 stuff. The optimizer uses system catalogs indicating how many rows there are and how well the indexes are formed to figure out what query path to use. Since the table is "empty," it is obviously easiest to read the entire table rather than trying to use an index.

The second key piece of information is that we are trying to read the bills table using a customer_number. There is no index on customer_number in the bills table. Even if we do update statistics for the table, the optimizer will have no choice but to sequentially scan it.

In our first improvement to the benchmark, let us add an index on the bills table and update statistics for the table. Since an index makes it more expensive to write to a table, create the index immediately after the bills table gets populated. Let us then update statistics after the index has been created. (Note, in V11.x statistics are generated at index creation time as part of the index creation process. Let's test that out.)

We add the following:

create index bills_idx1 on bills(customer_number);

Immediately, after the bills table is populated, we then clear the status counters with: onstat -z and restart the benchmark.

Our index has now been added. Let's check the table info:

<table>
<thead>
<tr>
<th>tabname</th>
<th>bills</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowsize</td>
<td>1266</td>
</tr>
<tr>
<td>nrows</td>
<td>605280.00000</td>
</tr>
<tr>
<td>created</td>
<td>2010-08-25 20:52:22.00000</td>
</tr>
</tbody>
</table>
Much better. Statistics have been updated, and not only is nrows populated but we also see when statistics were last updated. This run performed as follows:

**Listing 5. After updating statistics**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>36m5.033s</td>
</tr>
<tr>
<td>User</td>
<td>0m0.005s</td>
</tr>
<tr>
<td>Sys</td>
<td>0m1.479s</td>
</tr>
<tr>
<td>dskreads</td>
<td>1397183</td>
</tr>
<tr>
<td>bufreads</td>
<td>17971316</td>
</tr>
<tr>
<td>%cached</td>
<td>92.23</td>
</tr>
<tr>
<td>dskwrits</td>
<td>656896</td>
</tr>
<tr>
<td>pagwrits</td>
<td>885752</td>
</tr>
<tr>
<td>bufwrits</td>
<td>3087149</td>
</tr>
<tr>
<td>%cached</td>
<td>78.72</td>
</tr>
<tr>
<td>isamtot</td>
<td>14888305</td>
</tr>
<tr>
<td>open</td>
<td>284</td>
</tr>
<tr>
<td>start</td>
<td>1354389</td>
</tr>
<tr>
<td>read</td>
<td>6438035</td>
</tr>
<tr>
<td>write</td>
<td>747981</td>
</tr>
<tr>
<td>rewrite</td>
<td>956259</td>
</tr>
<tr>
<td>delete</td>
<td>30</td>
</tr>
<tr>
<td>commit</td>
<td>170</td>
</tr>
<tr>
<td>rollback</td>
<td>0</td>
</tr>
<tr>
<td>ovlock</td>
<td>0</td>
</tr>
<tr>
<td>ovusert</td>
<td>0</td>
</tr>
<tr>
<td>ovbuf</td>
<td>0</td>
</tr>
<tr>
<td>usercpu</td>
<td>2980.00</td>
</tr>
<tr>
<td>syscpu</td>
<td>119.77</td>
</tr>
<tr>
<td>numckpts</td>
<td>22</td>
</tr>
<tr>
<td>flushes</td>
<td>20</td>
</tr>
<tr>
<td>bufwaits</td>
<td>244886</td>
</tr>
<tr>
<td>lokwaits</td>
<td>0</td>
</tr>
<tr>
<td>lockreqs</td>
<td>10281980</td>
</tr>
<tr>
<td>deadlks</td>
<td>0</td>
</tr>
<tr>
<td>dltouts</td>
<td>0</td>
</tr>
<tr>
<td>ckpwaits</td>
<td>10</td>
</tr>
<tr>
<td>compress</td>
<td>12</td>
</tr>
<tr>
<td>seqscans</td>
<td>71</td>
</tr>
<tr>
<td>ixda-RA</td>
<td>479759</td>
</tr>
<tr>
<td>idx-RA</td>
<td>64</td>
</tr>
<tr>
<td>da-RA</td>
<td>901978</td>
</tr>
<tr>
<td>RA-pgsused</td>
<td>1381601</td>
</tr>
<tr>
<td>lchwait</td>
<td>12531</td>
</tr>
</tbody>
</table>

We are now within reason on a number of these indicators and have removed the noise, but it is still too early to tune the engine.

**Tuning the data model**

The most effective thing we can do to improve performance is correct any data model issues. Here is our current data model.

**Figure 1. Data model for the benchmark database**

Here we see some problems. For starters, the bills table has all sorts of de-normalized junk in it. All of the Customer, State, and Product information has been dumped in there.
All of the items in bold are extra junk to carry around that could be easily retrieved from the source tables. Sales_Tx is egregious because the one place where it could be used, it is instead read out of the state table. Product_Price does not belong in here, either, but if we look at how it is used, it is merely poorly named. This is really how much the customer was billed for the product on this bill. It should more properly be called Billed_Price.

If we look a little more deeply, we see static data mixed together with dynamic data. The customer name and address, for example, is unlikely to change. However, we certainly hope that their balance_due does. Furthermore, billing notes would not typically describe a customer. They should describe the relationship between a customer and his bill. Furthermore, the bill itself is broken. In this benchmark, we will generate six bills for each customer, instead of a single bill with six line items on it. Let's revisit this data model with these thoughts in mind.

Here is a revised model. It still has issues: Zipcode is enough to identify the city and state, so those attributes should migrate to a City table (instead of a state table), but that data is static, so we'll leave it alone.
If we were to reorganize the schema, that would take a little bit of work, and we would want to include that in the benchmark time. Note that executing the same application against the above data model took 16 seconds to execute (after the data had been reorganized).

Before we do that, we should size the tables. While Informix no longer has a performance issue with a large number of extents, there is still an issue in that each extent allocation takes time. Far better to size the table properly and not have to waste time extending it every time we insert 20 rows.

Sizing for these tables should be:

**Table 2. Table sizes**

<table>
<thead>
<tr>
<th>Table</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill_Items</td>
<td>20,416</td>
</tr>
<tr>
<td>Billing_Notes</td>
<td>202,048</td>
</tr>
<tr>
<td>Bills</td>
<td>14,976</td>
</tr>
<tr>
<td>Customer</td>
<td>14,976</td>
</tr>
<tr>
<td>Customer_Balance</td>
<td>1,728</td>
</tr>
</tbody>
</table>

Product and state will not change, so we won't bother sizing them. We will, however, update statistics for them. (Note: This became unnecessary and ineffective with changes further down this article).

**Tuning the application**

The next most effective thing we can do to improve performance is tune the application. This fits under the rubric of "Know thy data, Know thy queries." We should keep an eye on the data model when we do this as there may be some things we can assist the application with in the data model.
The first thing that the application does is generate three sets of bills for customers who have a product_code that starts with A. A different number of bills might be generated based on the customer start date (>= 1/1/2000, >=1/1/2005, >=1/1/1985). And variously for products 1 and 2, 4, and 7, and for 9 and 10. Looking at the data, all product codes start with A and the lowest start date in the customer table is 1/1/2007. In other words, we could combine these three insert statements into a single insert statement and only have to read the source tables once instead of three times.

Next, the application updates the Bills table, setting a product discount to 10 where the start_date is <= 1/1/2009 and their balance_due is > 50000. The WHERE clause is a repeat of this condition and adds a red herring of "product number in (1 through 10)." Since this encompasses all products, there is no point to that filter. There are about 40,000 customers whose start_date is < 1/1/2009 and whose balance_due is > 50000. This means that a sequential scan of the table will be faster than an indexed read, and we cannot play with the filter criteria.

Next, we update the bills again, setting the total bill to the price minus the discount plus the sales tax, which is read from the state table. Again, there is the red herring of product numbers in 1 through 10 (in other words, all of them). If only we had left the sales_tax in the Bills table we wouldn't have to reach out and grab it, especially since we now have to navigate through the customer table to get there. So let's put that back in.

Finally, we update the customer, incrementing their balance_due with the value of total_bill from the bills table. There is another red herring in that we only update customers who have bills. Since they all have bills this is moot. We have put Balance_due into its own table at present. If it were in the table with the Bills in it, we could avoid an extra update. Arguably, since there is now only one bill per customer, it might live there.

Then we read the totals to verify that the application ran correctly: Record count, sum(product_price), sum(product_discount), sum(total_bill). This information is now in the Bill_Items table. We could speed this read up if we made summary columns for these in the bill table. Of course, this would yield an incorrect record count, which we happen to know is the #Customers*6, but that is really departing from the straight and narrow).

And, we read Count(*) and sum(balance_due) from customers. Since we now have Balance_Due in the bills table, we should read directly from there now.

Our data model now looks like this:
Figure 4. Revised data model 2

The operations we are going to perform are:

- Scan customer table (6 seconds)
- Scan State table (0 seconds)
- Join Customer and State table (6 seconds)
- Insert into Bills (7 seconds)
- Create indexes (1 second)
- Load bill_items table (8 seconds)
- Create index on bill_items (3 seconds)
- Update bill_items (58 seconds); sequential scan of bill_items with a nested loop read of customer
- Update bills (5 seconds); sequential scan of bills with a nested loop read of bill_items
- Verification (4 seconds)

Listing 6. After revising the data model

<table>
<thead>
<tr>
<th>Profile</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dskreads</td>
<td>721155</td>
<td>734054</td>
<td>10036861</td>
<td>92.82</td>
<td>19724</td>
<td>60567</td>
<td>1450097</td>
<td>98.64</td>
</tr>
<tr>
<td>pagreads</td>
<td>10836861</td>
<td>796236</td>
<td>796197</td>
<td>65</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bufreads</td>
<td>1450097</td>
<td>98.64</td>
<td>10295768</td>
<td>447</td>
<td>807589</td>
<td>4558564</td>
<td>706197</td>
<td>65</td>
</tr>
<tr>
<td>%cached</td>
<td>82.7</td>
<td>92.82</td>
<td>19724</td>
<td>60567</td>
<td>1450097</td>
<td>98.64</td>
<td>10295768</td>
<td>447</td>
</tr>
</tbody>
</table>

If we look at the update of the bill_items, all we are doing is updating a discount based on information from the customer table, which we had when we first built the bill_items table. Let's
take advantage of that and set the discount while we are creating the table in the first place. The following use of a CASE statement will take care of that for us:

**Listing 7. CASE statement**

```
insert into bill_items
    (bill_number,
     customer_number,
     product_number,
     billed_price,
     product_discount)
select bill_number,
     bills.customer_number,
     product_number,
     product_price,
     case when (customer.start_date<="01/01/2009"
     and customer.balance_due>50000) then 10
     else 0
     end case
from bills, product, customer
where product.product_number in (1,2,4,7,9,10)
    and bills.customer_number=customer.customer_number;
```

And if we run again?

**Listing 8. Rerun CASE statement**

```
Profile
dskreads  pagreads  bufreads  %cached  dskwrits  pagwrits  bufwrits  %cached
209638     221593    8178955   97.44    25104      49481      844982     97.03
isamtot   open    start    read      write     rewrite   delete   commit   rollbk
7271368   376      908477   4242538   706230    100913      65        32        0
ovlock     ovuserthread ovbuff     usercpu  syscpu   numckpts   flushes
0          0            0          45.13    3.65     4          3
bufwaits   lokwaits   lockreqs   deadlks    dltouts    ckpwaits   compress   seqscans
1617       0          6056020    0          0          2          13         10
ixda-RA    idx-RA     da-RA      RA-pgsused lchwaits
1784       0          202678     204462     81
real 0m34.160s
user 0m0.006s
sys 0m0.050s
loaded bills in 3 sec
loaded bill_items in 19 sec
updated bills in 6 sec
verifications 4 sec
```

Look at all those locks. Since we are not manipulating locks ourselves, the engine is locking each row (or page) as it needs it. Let's lock the tables in exclusive mode. Note that this requires using transactions to surround our locks. We need to keep an eye on the logical logs and ensure that we have enough so that they don't fill and put us into a long transaction.

Locking the tables exclusively before beginning each insert our update operation yields:
Listing 9. Exclusive locks

<table>
<thead>
<tr>
<th>Exclusive locks</th>
<th>Kbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>78036 Kbytes</td>
<td></td>
</tr>
</tbody>
</table>

**Profile**

<table>
<thead>
<tr>
<th>dskreads</th>
<th>pagreads</th>
<th>bufreads</th>
<th>%cached</th>
<th>dskwrits</th>
<th>pagwrits</th>
<th>bufwrits</th>
<th>%cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>9943</td>
<td>217306</td>
<td>7344046</td>
<td>99.88</td>
<td>5757</td>
<td>38980</td>
<td>836059</td>
<td>99.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>isamtot</th>
<th>open</th>
<th>start</th>
<th>read</th>
<th>write</th>
<th>rewrite</th>
<th>delete</th>
<th>commit</th>
<th>rollbk</th>
</tr>
</thead>
<tbody>
<tr>
<td>7270986</td>
<td>496</td>
<td>908610</td>
<td>4241799</td>
<td>706213</td>
<td>100995</td>
<td>54</td>
<td>28</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ovlock</th>
<th>ovuserthread</th>
<th>ovbuff</th>
<th>usercpu</th>
<th>syscpu</th>
<th>numckpts</th>
<th>flushes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>35.80</td>
<td>3.90</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bufwaits</th>
<th>lokwaits</th>
<th>lockreqs</th>
<th>deadlks</th>
<th>dltouts</th>
<th>ckpwaits</th>
<th>compress</th>
<th>seqscans</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>3252</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>idx-RA</th>
<th>idx-RA</th>
<th>da-RA</th>
<th>RA-pgsused</th>
<th>lchwaits</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>152</td>
</tr>
</tbody>
</table>

**Profile**

<table>
<thead>
<tr>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0m31.829s</td>
<td>0m0.006s</td>
<td>0m0.048s</td>
</tr>
</tbody>
</table>

Something else I forgot at the application layer: Informix does very well when reading and writing large volumes of data. But with a logging database, we are going to log every operation we perform. We can improve large batch inserts by setting the table type to raw when we load it and then reset it to a standard table so we can build indexes on it. Changing the Bills and Bill_Items tables to raw and then loading yields:

Listing 10. Use raw tables to load

<table>
<thead>
<tr>
<th>raw tables to load</th>
</tr>
</thead>
<tbody>
<tr>
<td>849</td>
</tr>
<tr>
<td>208192</td>
</tr>
<tr>
<td>6732674</td>
</tr>
<tr>
<td>100.00</td>
</tr>
<tr>
<td>10031</td>
</tr>
<tr>
<td>13202</td>
</tr>
<tr>
<td>836054</td>
</tr>
<tr>
<td>98.80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>isamtot</th>
<th>open</th>
<th>start</th>
<th>read</th>
<th>write</th>
<th>rewrite</th>
<th>delete</th>
<th>commit</th>
<th>rollbk</th>
</tr>
</thead>
<tbody>
<tr>
<td>7474373</td>
<td>370</td>
<td>807441</td>
<td>4042766</td>
<td>706210</td>
<td>100993</td>
<td>49</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ovlock</th>
<th>ovuserthread</th>
<th>ovbuff</th>
<th>usercpu</th>
<th>syscpu</th>
<th>numckpts</th>
<th>flushes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31.98</td>
<td>2.32</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bufwaits</th>
<th>lokwaits</th>
<th>lockreqs</th>
<th>deadlks</th>
<th>dltouts</th>
<th>ckpwaits</th>
<th>compress</th>
<th>seqscans</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2664</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>idx-RA</th>
<th>idx-RA</th>
<th>da-RA</th>
<th>RA-pgsused</th>
<th>lchwaits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

**Profile**

<table>
<thead>
<tr>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0m21.686s</td>
<td>0m0.005s</td>
<td>0m0.019s</td>
</tr>
</tbody>
</table>

**Summary**

We have exhausted what we can demonstrate with this particular example and will have to dig into the engine in a following article with an example more geared to engine exploration. Of interest in this example is that we executed the benchmark without touching the engine. We "corrected" the data model and application and updated statistics. The tools we used to monitor what the engine was doing are applicable across any tuning exercise with Informix.
Appendix

Below is the original benchmark SQL script and the modifications we made in this article. Additions are marked in **bold** while deletions are marked in *italics*.

```sql
-- File:        benchmark_run.sql
-- Date:        4/20/2009
-- Author:        Lester Knutsen
-- Description:
--    There are 4 tables in this process to generate bills
--    customer table with 100100 records,
--    states table with sales tax info
--    product table with 10 products and the
--    bills table where the results get written to.
-- Log:  Created 4/20/2009

set explain on;
update statistics high for table product(product_number);
update statistics high for table state(state);
update statistics high for table customer(customer_number);
drop table bill_items;
set isolation dirty read;

-- drop and re-create the bills table

drop table "informix".bills ;

-- Note: Type raw and not carrying useless columns around
-- Also got rid of Primary key since we are now a raw table.

create raw table "informix".bills
    (    bill_number serial not null ,
        customer_number integer,
        last_name char(30),
        first_name char(30),
        address char(50),
        city char(20),
        state char(2),
        state_name char(20),
        zip integer,
        start_date date,
        bill_date date,
        bill_notes char(1000),
        product_code char(4),
        product_number integer,
        product_name char(50),
        product_price decimal(16,2),
        product_discount decimal(16,2),
        sales_tx decimal(16,2),
        total_bill decimal(16,2),
        balance_due decimal(16,2),primary key (bill_number )    );

-- Note: Bill_items was not part of original data model

create raw table bill_items
    (    bill_number integer not null ,
        customer_number integer,
        product_number integer,
        billed_price decimal(16,2),
        product_discount decimal(16,2)
    );
```

-- Note: Loading Bills in one single statement.

begin work;
lock table bills in exclusive mode;
insert into bills
  (bill_number,
customer_number,
bill_date,
sales_tax,
total_bill,
balance_due)
select 0,
customer_number,
today, -- bill_date
state.sales_tax,
0, -- total bill
balance_due
from customer, state
where customer.state=state.state;

alter table bills type(standard);
-- Note: And putting the index back on, effectively updates statistics
create index bill_idx1 on bills(customer_number);
commit;-- Note: Replaced the next 4 blocks of code with the single block above

-- Create bills for product numbers 1 and 2
insert into bills
  (customer_number,
last_name,
first_name,
address,
city,
state,
state_name,
zip,
start_date,
bill_date,
product_code,
product_number,
product_name,
product_price,
product_discount,
sales.tax,
total_bill)
select
customer.customer_number,
customer.last_name,
customer.first_name,
customer.address,
customer.city,
customer.state,
state.state_name,
customer.zip,
customer.start_date,
today, -- bill_date,
customer.product_code,
customer.product_number,
customer.product_name,
customer.product_price,
0, -- product_discount,
state.sales_tax,
0 -- total_bill
from customer, state, product
where customer.state = state.state
and customer.start_date >= "01/01/2000"
and product.product_number in ( 1, 2 );

Create bills for product number 4 and 7

insert into bills
(
customer_number,
lst_name,
first_name,
address,
city,
state,
state_name,
zip,
start_date,
bill_date,
product_code,
product_number,
product_name,
product_price,
product_discount,
sales_tax,
total_bill
)
select
customer.customer_number,
customer.last_name,
customer.first_name,
customer.address,
customer.city,
customer.state,
state.state_name,
customer.zip,
customer.start_date,
today, -- bill_date,
customer.product_code,
product.product_number,
product.product_name,
product.product_price,
0, -- product_discount,
state.sales_tax,
0 -- total_bill
from customer, state, product
where customer.state = state.state
and customer.start_date >= "01/01/2005"
and product.product_number in ( 4, 7 );

-- Create bills for product number 9 and 10

insert into bills
(
customer_number,
lst_name,
first_name,
address,
city,
state,
state_name,
zip,
start_date,
bill_date, 
product_code, 
product_number, 
product_name, 
product_price, 
product_discount, 
sales_tx, 
total_bill )

select  
customer.customer_number,  
customer.last_name,  
customer.first_name,  
customer.address,  
customer.city,  
customer.state,  
state.state_name,  
customer.zip,  
customer.start_date,  
today, -- bill_date,  
customer.product_code,  
product.product_number,  
product.product_name,  
product.product_price,  
0, -- product_discount,  
state.sales_tax,  
0 -- total_bill  
from customer, state, product  
where customer.state = state.state  
and customer.start_date >= "01/01/1985"  
and product.product_number in ( 9, 10 );Note: Loading Bill_Items with info that used to be in Bills

begin work;  
lock table bill_items in exclusive mode;  
insert into bill_items  
(bill_number,  
customer_number,  
product_number,  
billed_price,  
product_discount)  
select bill_number,  
bills.customer_number,  
product_number,  
product_price,  
case when (customer.start_date<="01/01/2009" and  
customer.balance_due>50000) then 10  
else 0  
end case  
from bills, product, customer  
where product.product_number in (1,2,4,7,9,10)  
and bills.customer_number=customer.customer_number;  
alter table bill_items type(standard);  
create index bill_items_idx2 on bill_items(bill_number);  
commit;Note: We took care of this by loading the bill_items table.

-- Update bills - give the best customers a $10 discount

update bills  
set product_discount = ( select 10 from customer  
where customer.customer_number = bills.customer_number  
and customer.start_date <= "01/01/2009"  
and customer.balance_due > 50000 )  

where bills.customer_number in ( select customer_number
from customer where customer.start_date <= "01/01/2009"
and customer.balance_due > 50000 )
and product_number in ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );

-- Update bills - calculate the total with sales tax and minus the discount

-- Note: Similar, but taking advantage of our new data model.
begin work;
lock table bills in exclusive mode;
update bills
set total_bill = (( ( select sum(bill_items.billed_price-product_discount)
from bill_items
where bills.bill_number=bill_items.bill_number))
* ( 1 + sales_tax ) ),
balance_due=balance_due + (( ( select sum(bill_items.billed_price-product_discount)
from bill_items
where bills.bill_number=bill_items.bill_number))
* ( 1 + sales_tax ));
commit;
update bills
set total_bill = (( product_price - product_discount )
* ( 1 + ( select sales_tax from state where bills.state = state.state )))
where product_number in ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );

Note: Not needed since we have balance in the bills table.

-- Update the customer balance for customers who just got a bill
update customer
set balance_due = balance_due + ( select sum ( total_bill )
from bills where bills.customer_number = customer.customer_number )
where customer_number in ( select customer_number from bills );

-- Calculate the totals - This must match the expected results
-- Note: Modified to read values out of new structures.
select count(*) record_count,
sum ( billed_price ) sum_product_price,
sum ( product_discount ) sum_product_discount
from bill_items;
select count(*) record_count,
sum ( product_price ) sum_product_price,
sum ( product_discount ) sum_product_discount,
sum ( total_bill ) sum_total_bill
from bills;

Note: We had to move the total bill down to this section since that data is now at the bill level.

Note also: Because there are 4 states in the original customer table which do not exist in the State table, the number of bills will not match the number of customers (off by 120) and the balance_due as computed from bills will not include the $044509.99 balance due from these customers. This could of course be corrected, but that is beyond the scope of this effort.

select count(*) customer_count,
sum(total_bill) sum_total_bill,
sum ( bills.balance_due )
from bills;

-- Perform a checkpoint to make sure all data is written to disk -
-- This is required ;}
execute function sysadmin:task ( 'onmode', 'c' );

-- End of SQL script
Related topic

- Learn more about the Fastest Informix DBA Contest, sponsored by Advanced DataTools.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)