What's a timeron worth to you? Who knows? DB2® SQL statement cost is measured in timerons, not dollars. This tutorial will discuss an approach to translating a timeron cost, and the associated administrative and computing resource costs into a cost metric that matters: dollars! You will learn the features and resources we considered for this calculation. We will discuss the resulting statistical monitors and measures available in DB2 and AIX® that monitor I/O, memory, and CPU, which were used in determining the actual dollar-cost calculations, jointly created by IBM and a partner. We will then show the calculation for assigning a dollar cost to an SQL statement by way of examples. Using this data, you will learn about our formulas for predicting a dollar cost for a system. Steps used to derive the formula will also be provided. Lastly, we will cover some of the items that caused skew.

Overview

Learn how to translate a timeron cost and the associated administrative and computing resource costs into a dollar figure.

Products

DB2 9.7, 10.1 enterprise server edition, for Linux, UNIX, and Windows
AIX 6.1 Technology Level 8 (TL8)
Not considered:
DB2 BLU Acceleration, Database Partitioning, pureScale®, HADR standby processing

Contributors and considerations for monetary cost:

CPU, disk, and memory contribute the highest percentage of dollar cost of a system. Tools to gather statistics for CPU, disk, and memory were available and were easy to set up. The factors not considered required statistics that were not readily available to us mostly due in part to supporting infrastructure teams that did not collect this data. While increasing data collection improved accuracy of the calculation, so did the complexity. Finally, some of the items introduced variables that were hard to calculate a dollar cost for. For example, to accurately calculate the support staff cost, we would have to understand the salary ranges for the support staff involved. This is not information that would be shared with us, so we would have to use industry averages. However, each support person is responsible for multiple items (DBA is responsible for more than just this database, UNIX engineer is responsible for more than just the Logical Partitioning (LPAR) the database runs on, etc.). The complexity comes with having to figure out a way to divide the cost of the person across what they support in a way that accurately represents the charge for the individual systems.

Statistics gathering (AIX and DB2)

AIX:
We started by creating a test environment with DB2 9.7 Fix Pack 8 and AIX 6.1 TL8, several tools were used initially, and as we formulated the equations we will use, we were able to narrow the data collections and tooling for this collection. NetApp Perstat utility provides a lot of information. It requires a C/C++ compiler and linker to set up, which required some coding. We used the free GNU C/C++ compiler/linker. We created tables that matched the C/C++ structs and constructed a C/C++ program that inserted into those tables via ODBC. Our intention is to use this statistics for future tuning and capacity planning measures. The perfstat_partition_total gives us enough information to calculate the average frequency of a core when running in power-saving mode. Therefore, we plan to use it to help with calculations of cost and savings by utilizing power-saving mode. The statistics gathered required some calculations. We set up advanced accounting in one of our development environments, and while it did provide helpful statistics due to it not being used across our landscape, we stopped its use. Tools used that are not part of our calculations due to inability to assign resource consumption to individual database instances: tprof, iostat, vmstat fcstat, lparstat, nmon, mpstat.

AIX Workload Manager:
We used AIX Workload Manager (WLM) in the passive mode. Starting WLM in passive mode means WLM will only gather statistics. WLM will not throttle workloads in passive mode, a workload for each database instance user ID was created. We initially used WLM stat with the verbose setting, but found we had everything we needed without verbose. Therefore, wlmstat -t was used to get CPU time. The CPU time reported is the total CPU time the workload class has consumed since system startup. This CPU time is already translated from ticks to milliseconds. The CPU time reported by wlmstat also reports external processing, such as the processing done by our Java™ FileMaker Pro scripts. We originally looked at the total_cpu_time for an
activity reported by the activity monitor and decided to use the CPU time reported for each SQL statement. However, there were some discrepancies in CPU time we were not able to explain, so we chose wlmstat for CPU time.

We chose to monitor at 10-minute intervals. All statistics that required interval monitoring ran at these same 10-minute intervals. This way, the statistics were comparable with one another since they were gathered at pretty close to the same time. We output the wlmstat data into an XML file, as it allowed us to add other data items that weren't reported by wlmstat (the time the statistics were gathered, for example). We also chose XML because we were familiar with the parsers provided in the Java libraries. We did have to write a parser for the raw wlmstat output, but we kept it fairly simple by assuming the output would remain static.

**DB2 event monitors**

**Units of Work (UOW) event monitor:**
The UOW event monitor provides lots of good statistics. However, since it is focused on units of work, it doesn't keep track of the timeron cost. Secondly, this event monitor recorded data as XML. A stored procedure then had to be called to read the XML and put the data into tables. There was too much data to do this at a regular interval. If we waited too long to run the stored procedure, it would take a while to translate the data due to the size of the data.

**Statistics event monitor:**
The statistics event monitor supplied aggregated statistics over the monitored time frame (average timeron cost, average execution time, and maximum execution time). It also provided a histogram that gave ranges of the number of SQL statements with costs in that range. We set up a cron entry that called the wlm_collect_stats at the same interval as wlmstat. This event monitor has a small footprint of data collected, so it is inexpensive to run. We are still working to figure out a way to use this data to extrapolate the average timeron cost over a 10-minute time frame. We ran into problems with high-cost SQL statements spanning multiple ranges. These statistics only reported this large cost during the time frame that the execution of the SQL statement started. We had high-cost SQL statements consuming resources during time frames their cost were not reported. This skewed our data because it led to lower average-cost SQL statements being assigned the resources actually consumed by the larger timeron cost SQL had previously started.

**Activity event monitor:**
The Activity event monitor gave us the timeron cost of every SQL statement. This does imply a large amount of data. The activity monitor records three rows for each SQL statement. It can also record 1 row for each parameter passed to the SQL statement. The size of the data can be controlled by settings (that is, we can eliminate the recording of the SQL text, eliminate the recording of the explain syntax, and eliminate the recording of the parameters). The data recorded is controlled by settings on workloads and service classes. This will be discussed more in the upcoming sections. We were able to get total_cpu_time, IOS (poocl_data_l_reads, direct_read_reqs, etc.) from this data. However, we were not able to explain some of the skew in these items, so we only used the timeron cost from this data.

**Set up activity event monitor:**

Translating the monetary cost of SQL statements, Part 1: SQL cost in DB2 9.7 and 10.1
We do not set up separate WLM classes. However, we do isolate applications via workloads for performance tests. This does allow us to isolate activity data by a workload. We do not throttle anything, so it is essentially set up the same way as AIX WLM in the passive mode.

Create any additional workloads if statistic isolation is desired.

```
create service class SC_TEST SUPER~
create service class SC_TEST under SC_TEST_SUPER~
create workload WL_TEST
SYSTEM_USER ( 'USER1', 'USER2', 'USER3' )
enable
service class SC_TEST under SC_TEST_SUPER~
grant usage on workload WL_TEST to group USER1~
grant usage on workload WL_TEST to group USER2~
grant usage on workload WL_TEST to group USER3~
```

This is an example of altering a workload to collect the activity data. We did not collect details, sections, or values because we wanted to minimize the data recorded. This eliminated the large `explain` and statement text lobs. It also eliminated collection of the parameters. The aggregate settings control what the statistics event monitor collects. The `collect activity metrics` item controls statistics the activity event monitor collects.

This is an example of creating an activity event monitor. Once the activity event monitor was created, we copied the DDL for the tables via one of our tools. We then dropped all the tables and re-created them. We did this to enable compression and to put data, indices, and LOBs in their own tablespaces. The activity event monitor doesn't have to be re-created. The activity event monitor can't be running when these changes are made.

```
CREATE EVENT MONITOR DBA_EVM_ACTIVITIES
FOR ACTIVITIES
WRITE TO TABLE
CONTROL (TABLE DBA.ACTIVITY_CONTROL IN DBA_D01),
ACTIVITY (TABLE DBA.ACTIVITY IN DBA_D01 ),
ACTIVITYMETRICS (TABLE DBA.ACTIVITYMETRICS IN DBA_D01),
ACTIVITY_STMT (TABLE DBA.ACTIVITY_STMT IN DBA_D01),
ACTIVITYVALS (TABLE DBA.ACTIVITYVALS IN DBA_D01)
MANUALSTART~
```

Check the state of activity event monitor.
select event_mon_state('DBA_EVM_ACTIVITIES') from sysibm.sysdummy1

Or

select
EVMONNAME,
    case when event_mon_state(EVMONNAME) = 1 then 'Executing'
    else 'Stopped'
end
from
SYSCAT.EVENTMONITORS

DB2 Statistics (Monitor Routines/Views):
We used the MON_GET_BUFFERPOOL and MON_GET_TABLESPACE to determine IOs (poocl_data_l_reads, direct_read_reqs, etc.). The statistics are the totals since the database was started. We created a script to write the data to a table and created a cron entry to execute this script at the same 10-minute intervals as the other monitors. We originally recorded both to compare the statistics and ended up only using the data recorded from MON_GET_BUFFERPOOL. At times, we also needed to drop and recreate tablespaces, which resulted in loss of statistics. When we subtracted the IOs from these two time frames to get the IOs over that time frame, we got a negative number. In our environment, bufferpools are much less likely to be dropped and re-created, so we chose to use the bufferpool statistics. See below for shell script and crontab entry mentioned above.

• Korn shell script to collect data
<call any profiles here>

db2 -td~ -v <<!
connect to $DB2INSTANCE~ insert into dba.mon_get_bp ( <columns>) select <columns> from table (mon_get_bufferpool(null, -2) ) - terminate~
!  
• Crontab entry to collect every 10 minutes
Collected at same interval as wlmstat statistics

00,10,20,30,40,50 * * * $HOME/db2work/monitor/db2snap.ksh > $HOME/db2work/monitor/db2snap.log 2> &1

Calculations
Dollar cost:
To calculate the dollar cost of an SQL statement, we have to define the cost of the resources involved in the calculations. We used the Internet to come up with list prices for each resource. Prices will more than likely be cheaper than this due to negotiations. These prices are not what the customer pays. We did not use any customer's pricing to derive these numbers. To calculate the dollar cost of the CPU used by an SQL statement, we first calculate the dollar cost for a millisecond of CPU time. This is done by dividing the yearly cost by the number of milliseconds
in a year. An example of the calculation follows. Milliseconds were used because a millisecond is
the unit of measure that `wlmstat` reports CPU consumption. We only took 80 percent of the total
CPU time because systems averaging CPU consumption greater than 80 percent tend to have
a CPU bottleneck. This generally means that they will probably need more CPU to alleviate the
bottleneck. Also, it doesn't matter how many cores a system has. The time will always be the same
because increasing the number of cores increases the total CPU time available by the same factor
as the price is increased.

• Define charges / time frame
  • CPU / core / year = $55,000
  • Memory / GB / year = $300
  • Disk / GB / year = $4.00
  • All pricing was based on list pricing. Pepsico pricing was not used.
• Calculate Dollars / Millisecond of CPU Time
  • .8 * Milliseconds / Year (Averaged to include leap year)
    • Assuming average CPU utilization of 80% or less.
    • Averaging above 80% CPU utilization can result in CPU bottlenecks
  • Divide Yearly Core cost by Milliseconds per year to get $ / millisecond of CPU time.
  • $ / millisecond cost will always be the same no matter how many cores.

The $/core/year was based on the cost of the hardware, the cost of AIX, and the cost of DB2. The
price assumed a four-year lease, so any one-time costs were divided across that term.

• Calculate Dollars / Millisecond of CPU Time
  • $ / Core / Year - $55,000
    • AIX $ Maintenance
    • PowerPC
    • Amortized DB2 one time cost
    • DB2 Maintenance
    • Assumes 4 year lease
  • Milliseconds / Year = 31,556,952,000
  • 80% of Milliseconds / Year
    = .8 * 31,556,952,000
    = 25,245,561,600 Milliseconds / Year
  • $ / Millisecond of CPU Consumption
    = $55,000 / 25,245,561,600
    = $ 2.1786E-6 / Millisecond of CPU consumed
Calculating memory:
We used the sum of the max size of all bufferpools to find the total GB of memory we would charge for. This information was pulled from the `SYSIBMADM.SNAPDB_MEMORY_POOL` view. For memory and disk, we needed a value we could use to assign a dollar cost, based on the amount of that resource an SQL statement used. The value needed to be a maximum so the cost could not exceed what was actually paid. Simply dividing the yearly cost by the number of milliseconds in a year assumes that each byte is only used once in a time frame. Therefore, we had situations where the dollar cost of IOs exceeded the price actually paid for the memory. Memory and disk are priced based on size, but the price includes more than just the size, including other items that impact throughput. For example, the frequency of the memory or the rotational speed of the disk. Throughput is what we used to assign cost. Specifically, we used $/IO for both memory and disk. We established a max bandwidth based on the `MON_GET_BUFFERPOOL` data. We found the maximum number of IOs for a 10-minute time frame. Based on this maximum, we estimated the total number of IOs that would occur in one year, given the maximum rate. We divided the cost of memory by this number to get the dollar cost per IO.

- Get maximum bufferpool size
  
  ```sql
  select sum(pool_watermark) from SYSIBMADM.SNAPDB_MEMORY_POOL where pool_id='BP' with ur
  = 2,717,188,096
  ```

- Convert to GB and multiply by $ / GB
  
  - $ / GB / Year = $300
    - = 300 * 2,717,188,096 / 1024 / 1024 / 1024
  - = $ 759

- Find maximum mem_ios and sample time frame for maximum IOs
  
  - MAX_IOS = 342,659,525
  - SAMPLE_TIME = 600,652,727

- Calculate number of sample time frames in year and hypothesize maximum throughput
  
  - Microseconds / Year = 3.1557E13
  - Number of Samples / Year
  - =3.1557E13 / 600,652,727
  - = 52537.85

- Hypothesize Maximum Possible IOs in 1 year
  
  - = 342,659,525 * 52537.85
  - = 1.80026E13

- Calculate $ / IO
  
  - = $759 / 1.80026E13
  - =4.21606E-11 $ / Memory IO

Calculate disk:
The procedure to determine cost was also the same as for memory. We used the disk allocated for all tablespaces to calculate a total cost. The space was used as part of the cost whether or not it was actually used because it was being paid for. Also, there was the potential for the disk to be used at any time. We also used bandwidth, $/IO, to assign a cost to an SQL statement based on what resources it consumed.

- Get Total GB of Disk (all tablespaces)
  
  \[ \text{df} -g \mid \text{grep} \ $\text{DB2INSTANCE} \]
  
  \[
  \begin{align*}
  /dev/hd01b\_lv04 & \ 235.00 \ 234.46 \ 1\% \ 770 \ 1\% \ /db2/db2hd01b/db2temp \\
  /dev/hd01b\_lv09 & \ 396.00 \ 34.89 \ 92\% \ 170 \ 1\% \ /db2/db2hd01b/db2appl
  \end{align*}
  \]
  
  \[
  = 235 \ \text{GB} + 396 \ \text{GB}
  \]
  
  \[
  = 631 \ \text{GB}
  \]

- Multiply by $ / GB
  
  - \[ \frac{4}{\text{GB}} \times 631 \times 12 = 30,288 \]
  
  - Find maximum disk_ios and sample time frame for maximum I/Os
    
    \[
    \text{MAX}_{\text{IOS}} = 342,786,630
    \]

  \[
  \text{SAMPLE}_{\text{TIME}} = 600,719,111
  \]

- Get Total GB of Disk (all tablespaces)
  
  \[ \text{df} -g \mid \text{grep} \ $\text{DB2INSTANCE} \]
  
  \[
  \begin{align*}
  /dev/hd01b\_lv04 & \ 235.00 \ 234.46 \ 1\% \ 770 \ 1\% \ /db2/db2hd01b/db2temp \\
  /dev/hd01b\_lv09 & \ 396.00 \ 34.89 \ 92\% \ 170 \ 1\% \ /db2/db2hd01b/db2appl
  \end{align*}
  \]
  
  \[
  = 235 \ \text{GB} + 396 \ \text{GB}
  \]
  
  \[
  = 631 \ \text{GB}
  \]

- Multiply by $ / GB
  
  - \[ \frac{4}{\text{GB}} \times 631 \times 12 = 30,288 \]
  
  - Find maximum disk_ios and sample time frame for maximum I/Os
    
    \[
    \text{MAX}_{\text{IOS}} = 342,786,630
    \]

  \[
  \text{SAMPLE}_{\text{TIME}} = 600,719,111
  \]

- Get Total GB of Disk (all tablespaces)
  
  \[ \text{df} -g \mid \text{grep} \ $\text{DB2INSTANCE} \]
  
  \[
  \begin{align*}
  /dev/hd01b\_lv04 & \ 235.00 \ 234.46 \ 1\% \ 770 \ 1\% \ /db2/db2hd01b/db2temp \\
  /dev/hd01b\_lv09 & \ 396.00 \ 34.89 \ 92\% \ 170 \ 1\% \ /db2/db2hd01b/db2appl
  \end{align*}
  \]
  
  \[
  = 235 \ \text{GB} + 396 \ \text{GB}
  \]
  
  \[
  = 631 \ \text{GB}
  \]
• Multiply by $ / GB
  • $4 / GB / Month
  • $4 * 631 * 12 = $30,288

• Find maximum disk_ios and sample time frame for maximum IOs
  • MAX_IOS = 342,786,630
  • SAMPLE_TIME = 600,719,111

• Calculate number of sample time frames in year and hypothesize maximum throughput
  • Microseconds / Year = 3.1557E13
  • Number of Samples / Year
    • = 3.1557E13 / 600,719,111
    • = 52532.04
  • Hypothesize Maximum Possible IOs in 1 year
    • = 342,659,525 * 52532.04
    • = 1.80006E13

• Calculate $ / IO
  • = $30,288 / 1.80006E13
  • = 1.68261E-9 $ / Disk IO

Activity data:
The activity and activity metrics tables have several columns. We only needed a fraction of those columns (time_started, time_completed, query_cost_estimate, etc.). We also included total_cpu_time, and all the bufferpool statistics (pool_data_l_read, pool_index_l_reads, etc.). These were kept to compare with the mon_get_bufferpool and wlmstat statistics. Therefore, to reduce the size of data, eliminate the join, and speed up the retrieval of data, we created a new table with just the columns we needed. This is the minimized table. We exported the activity data on a daily basis and loaded it into the minimized table. This allowed us to remove the larger activity data on a daily basis, and helped reduce the time it took to load the activity data into the minimized data. Our initial load covered four days and took eight hours to complete. Systems with lots of SQL being executed may require more frequent loads.

We also used non-recoverable data on our loads to prevent the table from going into a backup pending state. This can't be done in an HADR system. The table will not be usable when the failover occurs.

We used the wlmstat and mon_get_bufferpool data to calculate the cost of an SQL statement. The scripts used to collect this data ran independently of one another. Therefore, we had similar timestamps in wlmstat and mon_get_bufferpool, but they were not the same timestamps. Therefore, when we joined statistics to one another, we had to compare based on functions that translated the timestamps into YEAR, MONTH, DAY, HOUR, and MINUTE +/- 5 minutes. The +/-
5 minutes was used because the wlmstat statistics timestamp could differ by up to 5 minutes from the mon_get_bufferpool statistics timestamp for the same time range. This was a generous margin of error. It turned out that we never saw that big of a difference, but the padding helped to ensure we matched the correct time frames to one another. To simplify the SQL statement and assign the same time ranges to one another, we created a table that defined the values for the time range from each set (wlmstat and mon_get_bufferpool). We could join on the ID column now instead of translating timestamps. We also chose a range from one of the data sets to define the actual time range. We chose the wlmstat range because the CPU had the most impact on cost.

Table 1. WLM stats

<table>
<thead>
<tr>
<th>wlm_start_ts</th>
<th>wlm_end_ts</th>
<th>stats</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-08-01 19:00:17</td>
<td>2014-08-01 19:10:11</td>
<td>cpu_time, disk_ios, etc.</td>
</tr>
<tr>
<td>2014-08-01 19:10:11</td>
<td>2014-08-01 19:20:06</td>
<td>cpu_time, disk_ios, etc.</td>
</tr>
</tbody>
</table>

Table 2. Bufferpool stats

<table>
<thead>
<tr>
<th>bp_start_ts</th>
<th>bp_end_ts</th>
<th>stats</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-08-01 19:00:16</td>
<td>2014-08-01 19:10:09</td>
<td>Logical data reads, logical index reads, etc.</td>
</tr>
<tr>
<td>2014-08-01 19:10:10</td>
<td>2014-08-01 19:20:05</td>
<td>Logical data reads, logical index reads, etc.</td>
</tr>
</tbody>
</table>

Table 3. Stat ranges

<table>
<thead>
<tr>
<th>id</th>
<th>range_start_ts</th>
<th>range_end_ts</th>
<th>wlm_start_ts</th>
<th>wlm_end_ts</th>
<th>bp_start_ts</th>
<th>bp_end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2014-08-01 19:00:17</td>
<td>2014-08-01 19:10:11</td>
<td>2014-08-01 19:00:17</td>
<td>2014-08-01 19:10:11</td>
<td>2014-08-01 19:00:16</td>
<td>2014-08-01 19:10:09</td>
</tr>
</tbody>
</table>

Each value in the statistics column below was actually a column in the table. It is portrayed this way because there was not enough space to show all columns.

Since wlmstat is the total of each statistic (disk_ios, cpu_time, etc.), since system startup, we have to subtract the current statistics from the prior statistics to get the total of that statistics over a 10-minute time frame. For example, if we have a sample taken at 10 a.m. with cpu_time = 3000 and a second sample taken at 10:10 a.m. with cpu_time 3100, we have 100 ms assigned to the time frame of 10-10:10 a.m.

- wlmstat raw data to consumption per time frame.
- Each entry is the total number since system startup.
- Subtract each wlmstat entry by prior entry to get statistics per sampling period and put row in database.
- Assign each row to time range defined in the STAT_RANGES.

Table 4.
Preparation steps (bufferpool):
Each value in the statistics column in the table below was actually a column in the table. These columns are portrayed under the statistics column because there was not enough space to expand all columns. Since `mon_get_bufferpool` is the total of each statistic (`pool_data_l_reads`, `direct_read_reqas`) since system startup, we have to subtract the current statistics from the prior statistics to get the total of that statistic over a 10-minute time frame. For example, if we have a sample taken at 10 a.m. with `cpu_time` = 3000 and a second sample taken at 10:10 a.m. with `cpu_time` 3100, then we have 100 ms assigned to the time frame of 10-10:10 a.m.

Table 5.

<table>
<thead>
<tr>
<th>stats_range_id</th>
<th>bp_name</th>
<th>statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bp0</td>
<td><code>pool_data_l_reads</code>, <code>pool_index_l_reads</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>direct_read_req</code>, <code>pool_data_writes</code>, etc.</td>
</tr>
<tr>
<td>1</td>
<td>bp1</td>
<td><code>pool_data_l_reads</code>, <code>pool_index_l_reads</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>direct_read_req</code>, <code>pool_data_writes</code>, etc.</td>
</tr>
<tr>
<td>2</td>
<td>bp0</td>
<td><code>pool_data_l_reads</code>, <code>pool_index_l_reads</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>direct_read_req</code>, <code>pool_data_writes</code>, etc.</td>
</tr>
<tr>
<td>2</td>
<td>bp1</td>
<td><code>pool_data_l_reads</code>, <code>pool_index_l_reads</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>direct_read_req</code>, <code>pool_data_writes</code>, etc.</td>
</tr>
</tbody>
</table>

Preparation steps (activity weights):
Weight the cost of the SQL statement across all time frames it runs. The number of timerons assigned to each time frame depends on the amount of total execution time spent in that time frame. Store this information in a table. For example, if an SQL statement cost 100 timerons and it starts and ends in the 10–10:10 time frame, then all 100 timerons are assigned to the 10-10:10 time frame. If an SQL statement costs 100 timerons, the SQL statement runs for 10 minutes, and the SQL statement starts at 10:05 and ends at 10:15, then it spans two time frames (10–10:10 and 10:10–10:20). 50 timerons is assigned to the 10-10:10 time frame, and 50 timerons is assigned to the 10:10-10:20 time range.

Table 6.

<table>
<thead>
<tr>
<th>stat_range_id</th>
<th>sql_stmt</th>
<th>cost</th>
<th>weight</th>
<th>range_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQL1</td>
<td>2000</td>
<td>1</td>
<td>2000</td>
</tr>
<tr>
<td>1</td>
<td>SQL2</td>
<td>1000</td>
<td>.25</td>
<td>250</td>
</tr>
<tr>
<td>2</td>
<td>SQL2</td>
<td>1000</td>
<td>.5</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>SQL3</td>
<td>1000</td>
<td>.25</td>
<td>250</td>
</tr>
<tr>
<td>3</td>
<td>SQL4</td>
<td>100</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
Table 7.

<table>
<thead>
<tr>
<th>stats_range_id</th>
<th>range_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2250.5</td>
</tr>
<tr>
<td>2</td>
<td>500.5</td>
</tr>
<tr>
<td>3</td>
<td>350.5</td>
</tr>
</tbody>
</table>

For each time frame, add the total `mem_ios` and total `disk_ios` for all bufferpools in that time range. We used the weighted timerons created in Table 6 to create these summations for each time frame. This gives us the total `disk_ios` and total `mem_ios` for each 10-minute time frame. We can then assign these resources to individual SQL statements based on the SQL statement's fraction of the total cost.

- Aggregmem iosate bufferpool data over each statistics time range
- Assign sum of all memory io for a range to a STAT_RANGE
- \( \text{POOL\_DATA\_L\_READS + POOL\_TEMP\_DATA\_L\_READS + POOL\_XDA\_L\_READS + POOL\_TEMP\_XDA\_L\_READS + POOL\_INDEX\_L\_READS + POOL\_TEMP\_INDEX\_L\_READS + POOL\_DATA\_WRITES + POOL\_XDA\_WRITES + POOL\_INDEX\_WRITES} \)
- Assign sum of all disk io for a range to a STAT_RANGE
- \( \text{DIRECT\_READ\_REQS + DIRECT\_WRITE\_REQS + POOL\_DATA\_L\_READS + POOL\_TEMP\_DATA\_L\_READS + POOL\_XDA\_L\_READS + POOL\_TEMP\_XDA\_L\_READS + POOL\_INDEX\_L\_READS + POOL\_TEMP\_INDEX\_L\_READS + POOL\_DATA\_WRITES + POOL\_XDA\_WRITES + POOL\_INDEX\_WRITES} \)

For each time frame, add up the total `mem_ios` and total `disk_ios` for all bufferpools in that time range. We used the weighted timerons created in Table 6 to create these summations for each time frame. This gives us the total `disk_ios` and total `mem_ios` for each 10-minute time frame. We can then assign these resources to individual SQL statements based on the SQL statement's fraction of total cost.

Table 8.

<table>
<thead>
<tr>
<th>stats_range_id</th>
<th>bp</th>
<th>mem_ios</th>
<th>disk_ios</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bp0</td>
<td>1000000</td>
<td>1000000</td>
</tr>
<tr>
<td>1</td>
<td>bp1</td>
<td>500000</td>
<td>600000</td>
</tr>
<tr>
<td>2</td>
<td>bp0</td>
<td>800000</td>
<td>825000</td>
</tr>
<tr>
<td>2</td>
<td>bp1</td>
<td>100000</td>
<td>100000</td>
</tr>
</tbody>
</table>

Table 9.

<table>
<thead>
<tr>
<th>stats_range_id</th>
<th>mem_ios</th>
<th>disk_ios</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1500000</td>
<td>1600000</td>
</tr>
<tr>
<td>2</td>
<td>900000</td>
<td>925000</td>
</tr>
</tbody>
</table>

Preparation steps (SQL weights):
For each SQL statement, find the fraction of the total cost each SQL statement contributed to that time frame. For example, if the total cost of an SQL statement is 5,000 and the total timeron costs for all SQL statements for that same time frame is 10,000, the fraction of the cost is .5. Since this SQL statement accounted for half the total cost over that time frame, we are going to assign it half of the resources consumed during that time frame.

### Table 10.

<table>
<thead>
<tr>
<th>sql statement</th>
<th>stat range</th>
<th>Timeron cost for SQL</th>
<th>Total timerons for this stat range</th>
<th>Fraction of time SQL executed in this time range</th>
<th>Timerons in time frame</th>
<th>Cost Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>1</td>
<td>5000</td>
<td>10000</td>
<td>1</td>
<td>5000</td>
<td>.5</td>
</tr>
<tr>
<td>SQL2</td>
<td>1</td>
<td>2000</td>
<td>10000</td>
<td>.25</td>
<td>500</td>
<td>.05</td>
</tr>
<tr>
<td>SQL3</td>
<td>1</td>
<td>9000</td>
<td>10000</td>
<td>.5</td>
<td>4500</td>
<td>.45</td>
</tr>
</tbody>
</table>

### Preparation (resource consumption):

Use the fraction of total cost to divide the resources consumed across each of the SQL statements. For example, if an SQL statement accounts for half of the total timeron cost for a 10-minute time frame, it is assigned half of the resources consumed for that time frame.

### Table 11.

<table>
<thead>
<tr>
<th>sql statement</th>
<th>stat range</th>
<th>cost fraction</th>
<th>mem_ois</th>
<th>disk_ios</th>
<th>SQL mem_ios</th>
<th>SQL disk_ios</th>
<th>SQL CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>1</td>
<td>.5</td>
<td>100000</td>
<td>200000</td>
<td>10000</td>
<td>50000</td>
<td>100000</td>
</tr>
<tr>
<td>SQL2</td>
<td>1</td>
<td>.05</td>
<td>100000</td>
<td>200000</td>
<td>10000</td>
<td>5000</td>
<td>10000</td>
</tr>
<tr>
<td>SQL3</td>
<td>1</td>
<td>.45</td>
<td>100000</td>
<td>200000</td>
<td>10000</td>
<td>45000</td>
<td>90000</td>
</tr>
</tbody>
</table>

### Preparation (cost):

Multiply the total resources used with the cost for each resource to find the cost. Adding all these costs yields the total cost of the SQL statement.

### Table 12.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>timeron</th>
<th>SQL mem_ios</th>
<th>$/ IO</th>
<th>Total memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>5000</td>
<td>50000</td>
<td>4.21606E-11</td>
<td>$2.11E-06</td>
</tr>
<tr>
<td>SQL2</td>
<td>2000</td>
<td>5000</td>
<td>4.21606E-11</td>
<td>$2.11E-07</td>
</tr>
<tr>
<td>SQL3</td>
<td>9000</td>
<td>45000</td>
<td>4.21606E-11</td>
<td>$1.90E-06</td>
</tr>
</tbody>
</table>

### Table 13.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>timeron</th>
<th>SQL mem_ios</th>
<th>$/ IO</th>
<th>Total memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>5000</td>
<td>100000</td>
<td>1.68261E-9</td>
<td>$1.68E-04</td>
</tr>
<tr>
<td>SQL2</td>
<td>2000</td>
<td>10000</td>
<td>1.68261E-9</td>
<td>$1.68E-05</td>
</tr>
<tr>
<td>SQL3</td>
<td>9000</td>
<td>90000</td>
<td>1.68261E-9</td>
<td>$1.51E-04</td>
</tr>
</tbody>
</table>
Table 14.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>timeron</th>
<th>SQL CPU time</th>
<th>$/ ms cpu</th>
<th>$ CPU cost per execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>5000</td>
<td>5000</td>
<td>2.1786E-6</td>
<td>$1.09E-02</td>
</tr>
<tr>
<td>SQL2</td>
<td>2000</td>
<td>500</td>
<td>2.1786E-6</td>
<td>$1.09E-03</td>
</tr>
<tr>
<td>SQL3</td>
<td>9000</td>
<td>4500</td>
<td>2.1786E-6</td>
<td>$9.80E-03</td>
</tr>
</tbody>
</table>

Using Microsoft Excel, we created a scatter plot where the x-axis is the timeron cost, and the y-axis is the cost per execution. We expected this graph to show a linear relationship with a positive slope. Therefore, as the timeron cost grew, the cost per execution also grew.

The below scatter plot has some outliers removed. We had 966 data points. We considered 160 of those data points outliers. The red line shows the result of a linear regression solved using normal equations. This can also be done using the trend line feature for a graph. You can also select format trend line in Excel and choose whether to show the equation and r-squared value.

**Figure 1. Making predictions scatter plot diagram**

Making predictions:
We used normal equations to solve the linear regression. Since it is a linear regression, we had to solve for $m$ and $b$ in the equation $y = mx + b$. We used the data we had and fed it to the normal equation formula as a training set. Once $m$ and $b$ were found, we could plug any timeron value in for $x$ to get the dollars per execution cost of an SQL statement. We used the R-squared values to determine how well the line fit the data. When we removed some of the outliers, we noticed a large increase in accuracy of fit for the line.
• Our values (no outliers)
  • m = 9.84653762301420e-009
  • b = 2.14289640346172e-004
  • R-squared values with outliers: .09
  • With some outliers removed: .57

Before we removed any outliers, we would like to explain why those data points seemed to deviate from the regression. Researching the SQL statements required us to turn on the activity event monitoring with SQL text and parameters. This created a lot of data in a short time frame (75 GB in one day). Upon looking at the SQL, we found a statement with a cost of 399, which returned zero rows on some executions and 10,000 rows on other executions. We also found that we did not have statistics on our catalogs. The activity event monitor data showed some SQL statements with longer lock wait time and little CPU time. We have had instances where large amount of data was inserted or deleted, and applications attempted to use the data before the runstats was executed. This resulted in timeron costs that were too high or too low.

**Conclusion**

With the above step-by-step statistical monitors, formulas, and calculations, you can translate DB2 SQL statement cost from timerons to dollars.
Resources

- The Information Management area on developerWorks provides resources for architects, developers, and engineers.
- Stay current with developer technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Follow developerWorks on Twitter.
- Watch developerWorks demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Get involved in the developerWorks Community. Connect with other developerWorks users while you explore developer-driven blogs, forums, groups, and wikis.
About the authors

Mike Faltys

Mike Faltys is a DBA at Pepsico focusing on new application development and performance. He has presented at IDUG North America, IDUG Europe, and IBM Insight.

Christopher Godfrey

Christopher Godfrey is an Accelerated Value Leader at IBM working directly with PepsiCo to deliver proactive, cost-reducing, and productivity-enhancing advisory service with focus on DB2 database products. In his prior role, he worked with a range of customers, assisting them with resolving complex memory, performance, optimization issues ensuring health and reliability of their database servers. He has written and presented several IBM Knowledge Know presentations on DB2 LUW. He has a bachelor's degree in computer science from the University of Toronto.

Ian Finlay

Ian Finlay is an advisory development analyst and Master Inventor, who was been working on the DB2 for LUW Query Optimizer for more than 20 years. His areas of expertise include runtime engine modelling, and query plan tuning and analysis. He has spent the past eight years in DB2 for LUW L3 Optimizer Support, and currently works in the DB2 for LUW Optimizer New Development group.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)