sunshinesoftware
Architects of Internet Applications

design. develop. deliver.

Case Study: A SQL Server DBA Improves Customer Application Service

This paper documents how a DBA used a performance tool, Confio Ignite 8, to find and resolve database problems that directly impacted customer service.

What is the final cost of product development, if in the end your customers can't get their work out of your web based product fast enough? This report explains the pain points of a company whose database was designed 'on the fly' by the application developers. This isn't such a bad thing, unless these developers don't have a solid understanding of how databases work and how to get a fair level of performance out of them.

Case Study: A SQL Server DBA Improves Customer Application Service

This paper documents how a DBA used a performance tool, Confio Ignite 8, to find and resolve database problems that directly impacted customer service.

Customers Were Calling for Product Support

The Product Development manager decided that something had to be done. Customers were concerned and the business unit leaders were getting nervous at the reports about how certain features of the companies web application was taking too long to process their reports and disconnecting them from their active web sessions. Why are these problems coming to light now after the application has been online for over three (3) years?

The company interviewed me twice as to what I could do help them. After a thoroughly impressive display of SQL Server knowledge I was brought in to assist them to get this situation under control if possible. The system administrators were out of options. They'd thrown enterprise level SAN hardware at the problem. Still, they were having trouble with customers being dropped from their web sessions. Then it happened. Developers were tasked to discover why the application was behaving this way. The developers had exhausted their research into their code

How Proper Database Indexes Saved the Day!

By examining the index structure of a table that was involved in a multi-join query, we were able to trim 126 seconds off of the processing time of a query in a production database! In a one hour period this query came through the production database over ten times. That's seventeen (17) plus minutes of customers waiting to get their data back from the database. These days it's hard to keep a web-user on the page for more than a few seconds. It's needless to say that customers will detect a substantial speed increase in getting their data processed and be delighted to continue to use the company's online product offerings.

and decided that all these dynamically generated SQL statements were necessary for the product to function. Then they opened up Microsoft's SQL Server Management Studio (SSMS) for 2005 and turned on the Database Tuning Wizard. Right away the reports started to be generated about what indexes were missing from their tables based on the statistics kept by the server. Well, a production change request (CR) was approved and the create button was pressed to generate all these database index objects. Oops.

Index Evaluation and Creation

Being able to adjust any of the application code was out of the question for financial and time reasons. The application was live and there wasn't time or money or recode the application. So we were left with optimizing the database to process these queries more efficiently as our only recourse. So, now that the customer base had grown to a point where these performance issues were presenting themselves, it was time to examine the performance of a subset of all the production SQL queries. We needed to determine what index configuration would best help the end user (customer) experience. This section will illustrate the issues that led to a bad user experience. There were many variables to consider when deciding how to address these database performance issues.

1) How do we discover where to start?
2) What methods and tools can be used to detect the problems and document the performance observations?
3) How can the proposed solutions prove that it made a difference in the end user experience?
4) Is there a way to test the solutions using a production load level?

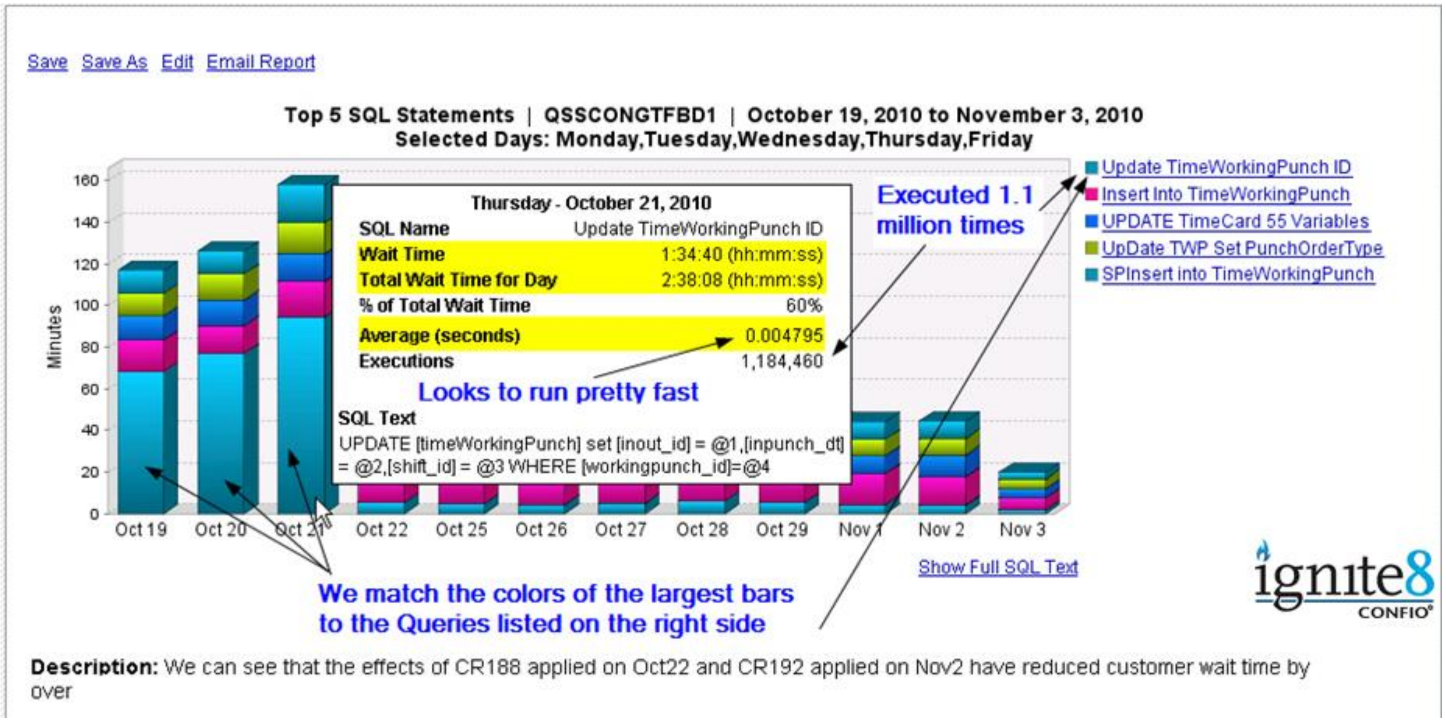
I recalled that a couple years ago that I was at an SQL Saturday event and talked with the sales reps for a software product that may help in this situation. It was a database wait time software package that was able to display which queries were taking the longest to process. This would be the starting point of the investigation phase.

Being able to use this software tool and translate that data into an action plan proved to be the point where the tires met the road. Having once read that being a Database Administrator (DBA) was to be a scientist. Document how the database is running now (getting a baseline), develop a hypothesis to improve that performance using proven techniques, test out that hypothesis and document if your database is running better than before. It was time to go to work.

There have been many books written by authors more knowledgeable than myself when it comes to database indexing and how SQL Server 2005 and 2008 process queries. My personal library is filled with many of them and I owe a debt of gratitude to those fine people for sharing their knowledge and experience with me through their works. There are some basic index creation rules that have been documented in many of these publications. We'll touch on some of these as we move through this performance tuning experience.

Getting Pointed in the Right Direction

The client created a workstation in the production area to be used as an SQL Administrators box. After installing an instance of SQL Server 2008 and verified connectivity to all the production database servers the environment was ready for the discovery process. Installing a trial version of Confio's SQL Ignite 8 and set it up to collect data from all my production servers allowed for the comparison of current performance against historical performance data, per query. After running for a few hours the gathered data began to point to where the query bottlenecks were. In allowing the tool to run for a day and collect data on the SQL Server performance, a report on the top 5 SQL Queries that were taking the longest to run was presented. The figure below illustrates the overall wait time in minutes during the initial phase of the database tuning efforts. Examining October 19 thru 21 there are large values (118 to 158) for the total wait time on this server. The light blue bars represent the query labeled as UPDATE TimeWorkingPunch ID and had an average running time of 0.004795. That's a nice value to have available to me. On October 21 the system executed this query 1.1 million times. Since this query is shown as having the largest bars in my chart (from October 19 - 21) as having the most wait time we'll explore this query further.



Thursday - October 21, 2010

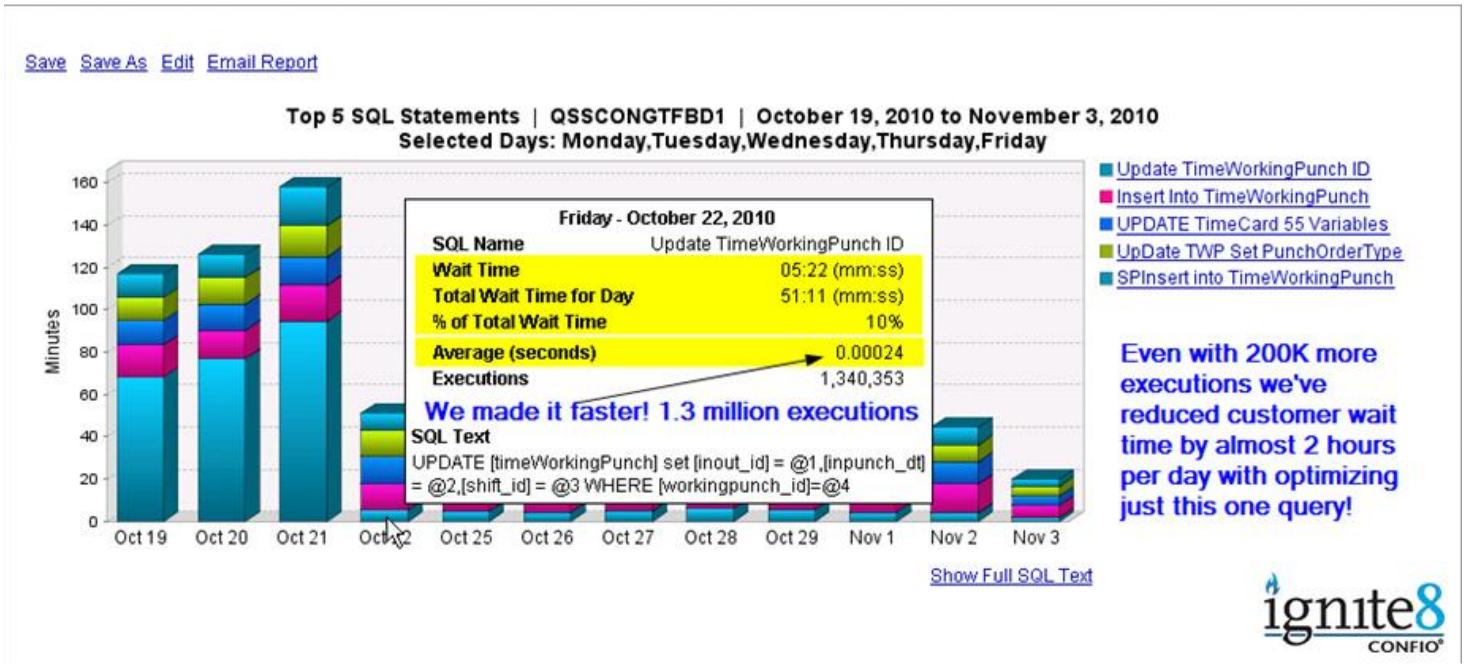
SQL Name	Update TimeWorkingPunch ID
Wait Time	1:34:40 (hh:mm:ss)
Total Wait Time for Day	2:38:08 (hh:mm:ss)
% of Total Wait Time	60%
Average (seconds)	0.004795
Executions	1,184,460

SQL Text

```
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt]
= @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

Description: We can see that the effects of CR188 applied on Oct22 and CR192 applied on Nov2 have reduced customer wait time by over

The goal of this work is to improve the end users experience, basically to make the database run faster. After following a process of table design discovery, index creation trial and error we were able to achieve an index structure that eliminated all of the Database Tuning Advisor (DTA seen as _dta_Objectname in SSMS) objects in favor of this new structure on a single table in the database. That code was implemented as a Change Request (CR) on the evening of October 21. The chart below will document the overall database performance from October 22 on. Also please note that the next day had a higher volume of this query hitting the database for a total of 1.3 million times as compared to the 11 million the previous day. By tuning this one table we took the overall database wait time from 2:38:08 per day down to 51:11. Fifty-one minutes from over two (2) hours. That one query had reduced the end users wait time from one hour and thirty-four minutes down to five minutes - for the whole day - running 1.3 million times! This was a significant improvement that needed to be illustrated to the product development manager.



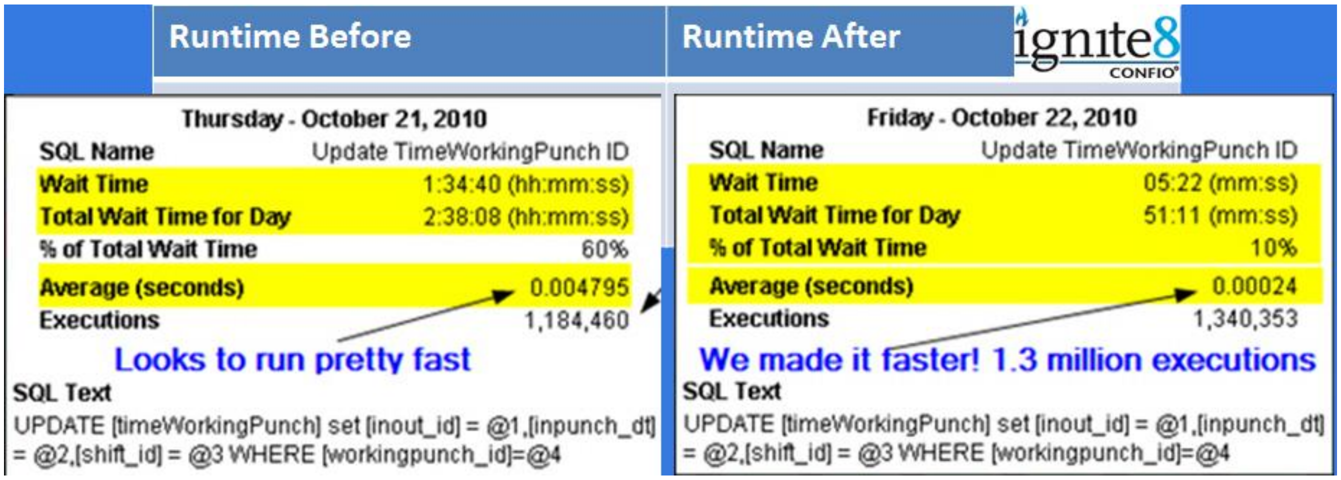
Friday - October 22, 2010

SQL Name	Update TimeWorkingPunch ID
Wait Time	05:22 (mm:ss)
Total Wait Time for Day	51:11 (mm:ss)
% of Total Wait Time	10%
Average (seconds)	0.00024
Executions	1,340,353

SQL Text

```
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt]
= @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

Included here is an overview of the results obtained by altering the index structure of the table that this query was executing against. Please also note that there was a total reduction of 50% in overall end user wait times on this query.



Thursday - October 21, 2010

SQL Name	Update TimeWorkingPunch ID
Wait Time	1:34:40 (hh:mm:ss)
Total Wait Time for Day	2:38:08 (hh:mm:ss)
% of Total Wait Time	60%
Average (seconds)	0.004795
Executions	1,184,460

SQL Text

```
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt]
= @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

Friday - October 22, 2010

SQL Name	Update TimeWorkingPunch ID
Wait Time	05:22 (mm:ss)
Total Wait Time for Day	51:11 (mm:ss)
% of Total Wait Time	10%
Average (seconds)	0.00024
Executions	1,340,353

SQL Text

```
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt]
= @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

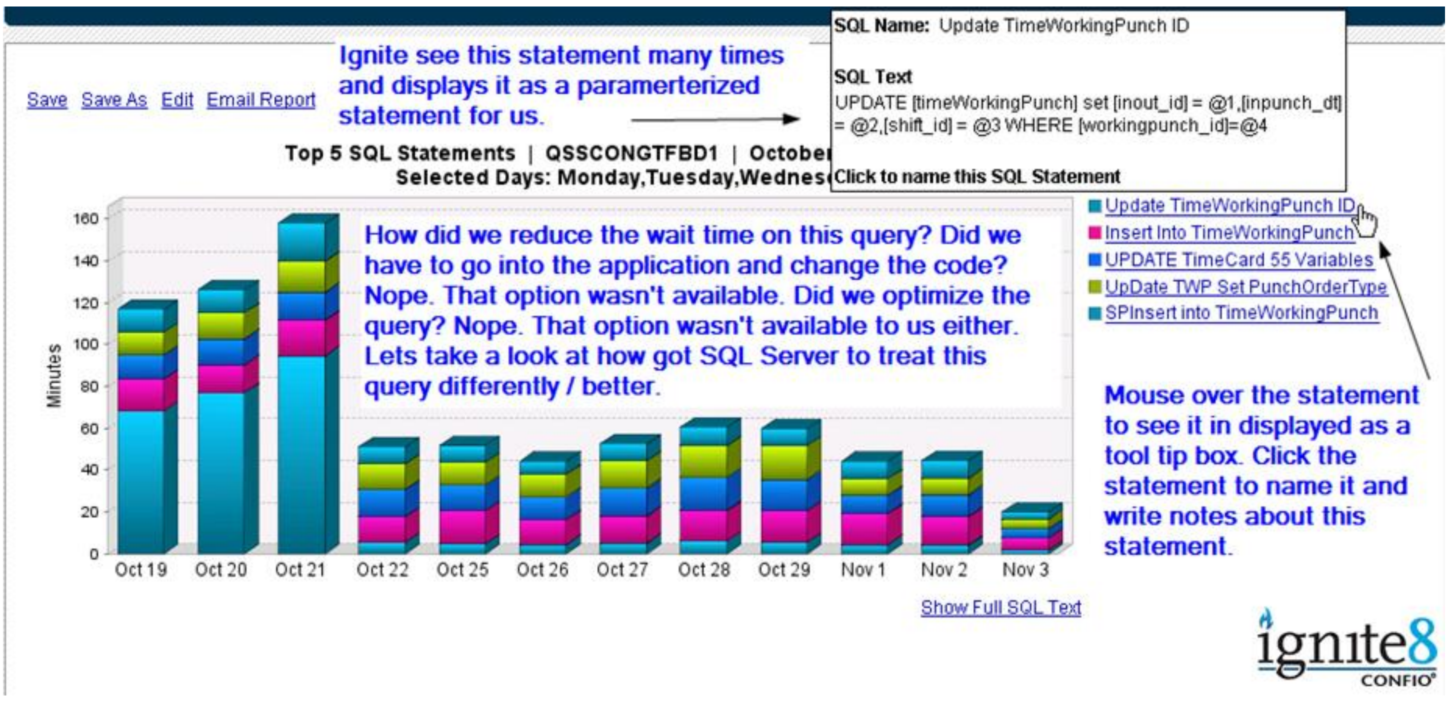
sunshinesoftware

Success Comes with a Price

The following Monday product support gets a call from a customer who states that they can't login to the web application. After much research it was discovered that the memory available to SQL Server had been completely taken up by the Buffer Manager. There were no free pages in memory for new queries to be processed. Queries were being processed much faster into the server, new query plans were being created and the server had not been configured properly for proper memory management. Needless to say that this memory issue was just hiding in the background masked by the slow executing queries keeping the database bogged down. Now that the database had new indexes to process the queries on this table data throughput was tripled (documented with an SQL load testing tool).

The Discovery Process

The Ignite tool allows me to give readable names to queries that I want to work with. The charting ability allowed me to document the performance of the database both before and after the CR was implemented. This one chart alone made the management executives extremely happy.



Ignite see this statement many times and displays it as a paramerterized statement for us.

SQL Name: Update TimeWorkingPunch ID

SQL Text

```
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt]
= @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

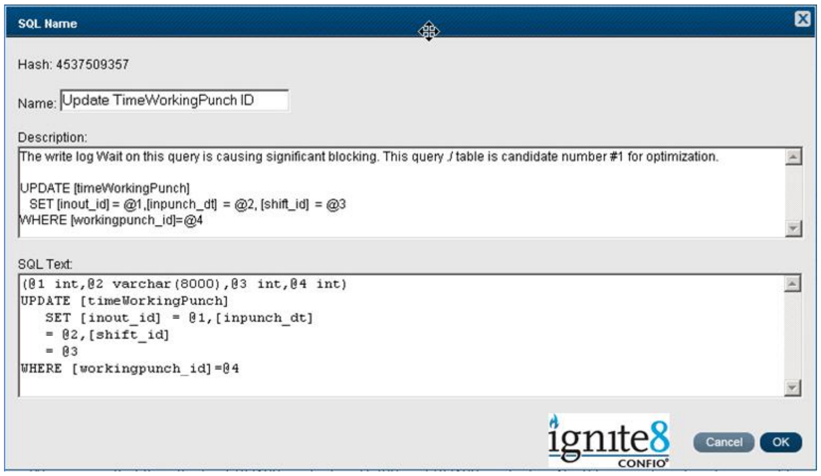
Click to name this SQL Statement

How did we reduce the wait time on this query? Did we have to go into the application and change the code? Nope. That option wasn't available. Did we optimize the query? Nope. That option wasn't available to us either. Lets take a look at how got SQL Server to treat this query differently / better.

Mouse over the statement to see it in displayed as a tool tip box. Click the statement to name it and write notes about this statement.

Save
Save As
Edit
Email Report
Top 5 SQL Statements | QSSCONGTFBD1 | October
Selected Days: Monday,Tuesday,Wednes
Minutes
0
20
40
60
80
100
120
140
160
Oct 19
Oct 20
Oct 21
Oct 22
Oct 25
Oct 26
Oct 27
Oct 28
Oct 29
Nov 1
Nov 2
Nov 3
Update TimeWorkingPunch ID
Insert Into TimeWorkingPunch
UPDATE TimeCard 55 Variables
UpDate TWP Set PunchOrderType
SPInsert into TimeWorkingPunch
Show Full SQL Text
ignite8
CONFIO

Let's look at the specifics of the top offending query. Ignite allows me to write notes about the query to help me remember specifics about it. See the image below.



Hash: 4537509357

Name: Update TimeWorkingPunch ID

Description:

The write log Wait on this query is causing significant blocking. This query / table is candidate number #1 for optimization.

```
UPDATE [timeWorkingPunch]
  SET [inout_id] = @1,[inpunch_dt] = @2, [shift_id] = @3
WHERE [workingpunch_id]=@4
```

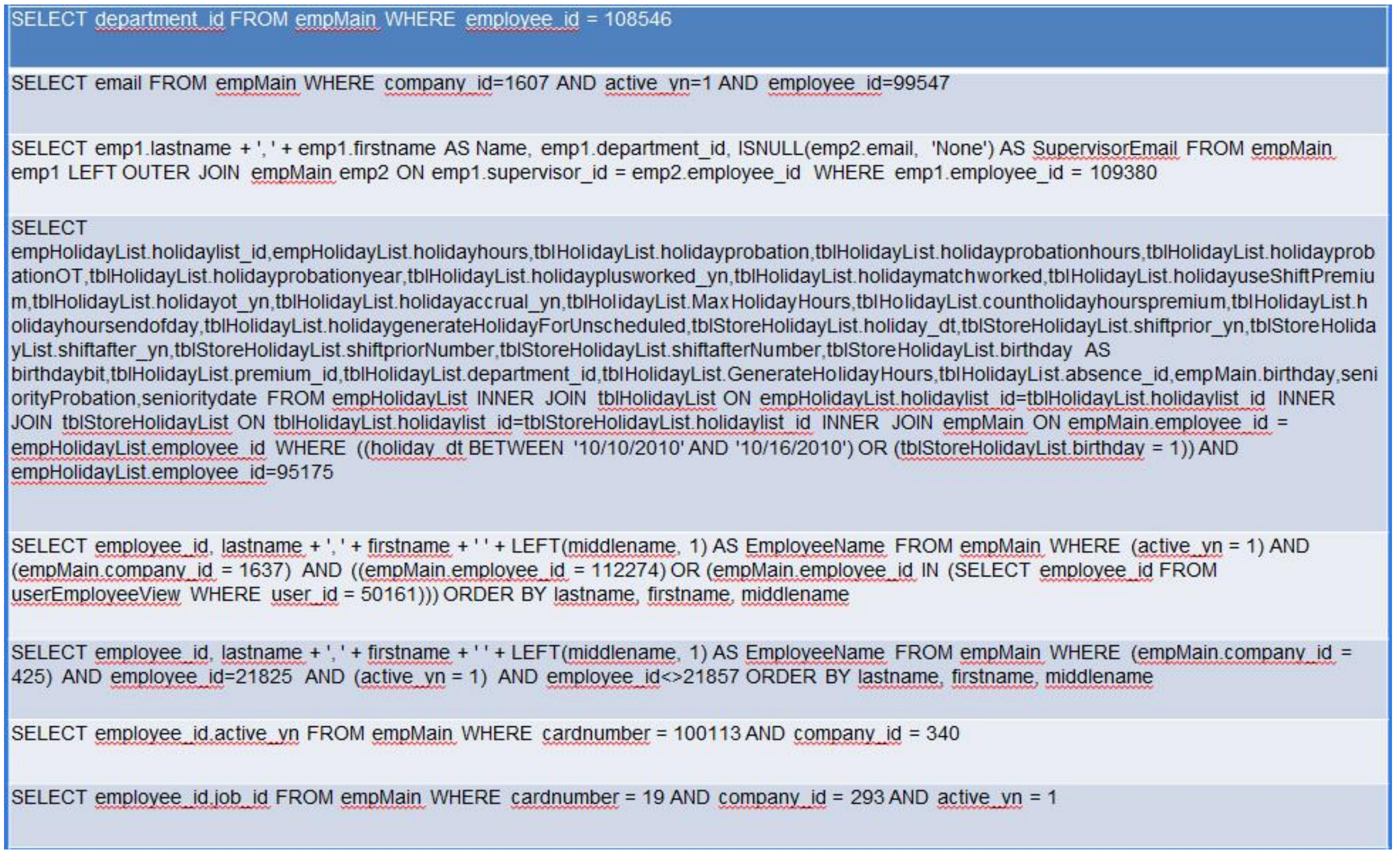
SQL Text:

```
(@1 int,@2 varchar(8000),@3 int,@4 int)
UPDATE [timeWorkingPunch]
   SET [inout_id] = @1,[inpunch_dt]
   = @2,[shift_id]
   = @3
WHERE [workingpunch_id]=@4
```

Here is a top level view of the process used to discover a unique set of queries that we'll use to rebuild the index structure on the effected table:

1) Because the SQL Server Profiler application can impact performance – run a few different traces throughout the day for 10 to 15 seconds each.
2) Save the trace data inside many different SQL tables on a remote admin server.
3) SELECT out from the TEXTDATA field the queries that contain the table name you want to work on.
4) Use a UNION command to aggregate all the queries into a single result set.
5) Save the result set as a CSV file (right clicking inside the result set will give you this option).
6) Import the CSV file into an MS Excel spreadsheet, and then sort it.
7) Remove the queries that are duplicated (the data values that are entered as Where clause filters will be different but the fields of the query will be in the same order. That's why we sorted the input).

You now have a list/set of unique queries that you'll take one at a time and evaluate. This is your set of queries to use while building a new Index structure.



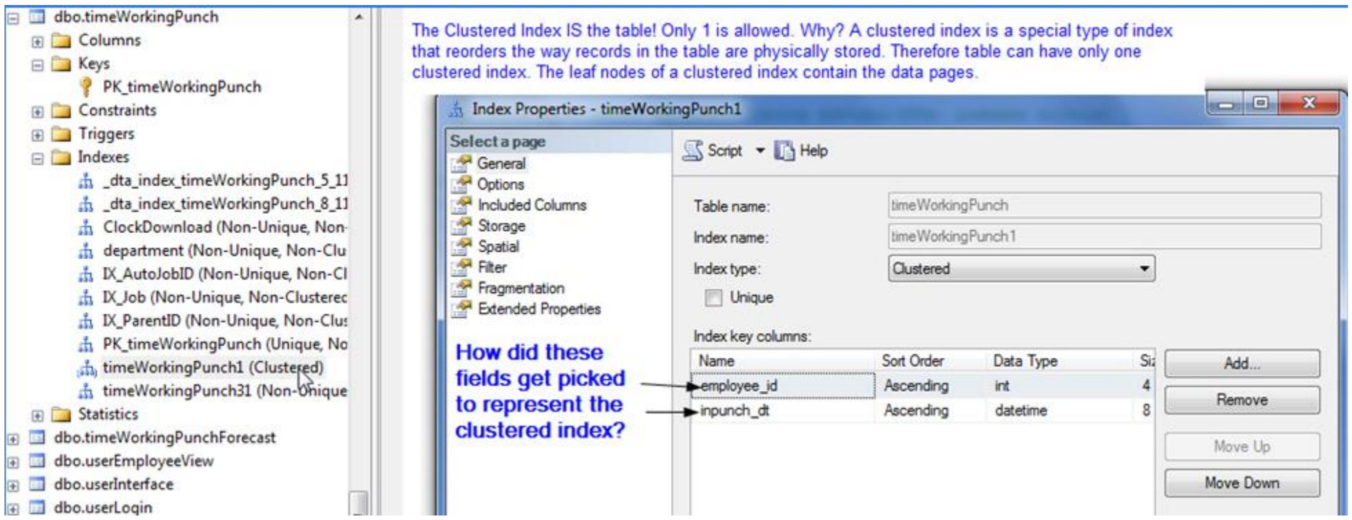
```
SELECT department_id FROM empMain WHERE employee_id = 108546

SELECT email FROM empMain WHERE company_id=1607 AND active_yn=1 AND employee_id=99547

SELECT emp1.lastname + ',' + emp1.firstname AS Name, emp1.department_id, ISNULL(emp2.email, 'None') AS SupervisorEmail FROM empMain
emp1 LEFT OUTER JOIN empMain emp2 ON emp1.supervisor_id = emp2.employee_id WHERE emp1.employee_id = 109380

SELECT
empHolidayList.holidaylist_id,empHolidayList.holidayhours,tblHolidayList.holidayprobation,tblHolidayList.holidayprobationhours,tblHolidayList.holidayprob
ationOT,tblHolidayList.holidayprobationyear,tblHolidayList.holidayplusworked_yn,tblHolidayList.holidaymatchworked,tblHolidayList.holidayuseShiftPremiu
m,tblHolidayList.holidayot_yn,tblHolidayList.holidayaccrual_yn,tblHolidayList.MaxHolidayHours,tblHolidayList.countholidayhourspremium,tblHolidayList.h
olidayhoursendofday,tblHolidayList.holidaygenerateHolidayForUnscheduled,tblStoreHolidayList.holiday_dt,tblStoreHolidayList.shiftprior_yn,tblStoreHolida
yList.shiftafter_yn,tblStoreHolidayList.shiftpriorNumber,tblStoreHolidayList.shiftafterNumber,tblStoreHolidayList.birthday AS
birthdaybit,tblHolidayList.premium_id,tblHolidayList.department_id,tblHolidayList.GenerateHolidayHours,tblHolidayList.absence_id,empMain.birthday,seni
orityProbation,senioritydate FROM empHolidayList INNER JOIN tblHolidayList ON empHolidayList.holidaylist_id=tblHolidayList.holidaylist_id INNER
JOIN tblStoreHolidayList ON tblHolidayList.holidaylist_id=tblStoreHolidayList.holidaylist_id INNER JOIN empMain ON empMain.employee_id =
empHolidayList.employee_id WHERE ((holiday_dt BETWEEN '10/10/2010' AND '10/16/2010') OR (tblStoreHolidayList.birthday = 1)) AND
empHolidayList.employee_id=95175

SELECT employee_id, lastname + ', ' + firstname + ' ' + LEFT(middlename, 1) AS EmployeeName FROM empMain WHERE (active_yn = 1) AND
(empMain.company_id = 1637) AND ((empMain.employee_id = 112274) OR (empMain.employee_id IN (SELECT employee_id FROM
userEmployeeView WHERE user_id = 50161))) ORDER BY lastname, firstname, middlename

SELECT employee_id, lastname + ', ' + firstname + ' ' + LEFT(middlename, 1) AS EmployeeName FROM empMain WHERE (empMain.company_id =
425) AND employee_id=21825 AND (active_yn = 1) AND employee_id<>21857 ORDER BY lastname, firstname, middlename

SELECT employee_id,active_yn FROM empMain WHERE cardnumber = 100113 AND company_id = 340

SELECT employee_id,job_id FROM empMain WHERE cardnumber = 19 AND company_id = 293 AND active_yn = 1
```

An interesting side effect of this process is that as you progress further down the list of queries, building the new index structures, you'll find that the execution times of the queries near the beginning of this process has changed. With the introduction of each new Index we've added overhead to the processes that SQL Server uses to keep all the indexes current. Each Update, Insert and Delete T-SQL statement will affect all the indexes that have been built on those fields. Best practices for how many indexes to be built, per table; in an Online Transactional Processing (OLTP) is not a hard and fast number. Generally, a single table should have no more than ten (10) indexes. That figure is a matter for a discussion on a different paper, and can be researched by a quick online search of the topic. Needless to say that the more indexes you introduce into a table structure, the more overhead you incur for SQL Server to handle.

Here is a sample of the code that was used to pull in all trace data from numerous traces taken at different times of the day.

```
select TextData , Duration from dbo.TRC1 where TextData like '%TimeWorkingPunch%'

union all

select TextData  , Duration from dbo. TRC2  where TextData like '% TimeWorkingPunch%'
union all

select TextData  , Duration from dbo. TRC3 where TextData like '% TimeWorkingPunch%'
```

You can also perform a sort by the Duration field to get an ordered list of the longest running queries first. The query was found in the Excel result set and now we start looking at this query to determine how to improve its performance. We start by looking at the existing table structure and questioning the decisions that were made to establish a clustered index.


dbo.timeWorkingPunch
Columns
Keys
PK_timeWorkingPunch
Constraints
Triggers
Indexes
_dta_index_timeWorkingPunch_5_11
_dta_index_timeWorkingPunch_8_11
ClockDownload (Non-Unique, Non-
department (Non-Unique, Non-Clu
IX_AutoJobID (Non-Unique, Non-Cl
IX_Job (Non-Unique, Non-Clusterec
IX_ParentID (Non-Unique, Non-Clus
PK_timeWorkingPunch (Unique, No
timeWorkingPunch1 (Clustered)
timeWorkingPunch31 (Non-Unique
Statistics
dbo.timeWorkingPunchForecast
dbo.userEmployeeView
dbo.userInterface
dbo.userLogin
Index Properties - timeWorkingPunch1
Select a page
General
Options
Included Columns
Storage
Spatial
Filter
Fragmentation
Extended Properties
Script
Help
Add...
Remove
Move Up
Move Down

The Clustered Index IS the table! Only 1 is allowed. Why? A clustered index is a special type of index that reorders the way records in the table are physically stored. Therefore table can have only one clustered index. The leaf nodes of a clustered index contain the data pages.

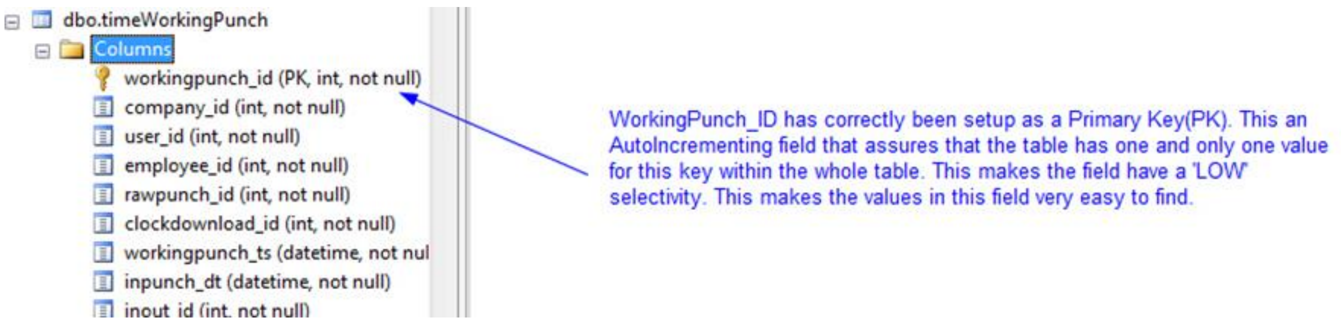
Table name: timeWorkingPunch
Index name: timeWorkingPunch1
Index type: Clustered
Unique: ☐

Index key columns:

Name	Sort Order	Data Type	Si
employee_id	Ascending	int	4
inpunch_dt	Ascending	datetime	8

How did these fields get picked to represent the clustered index?

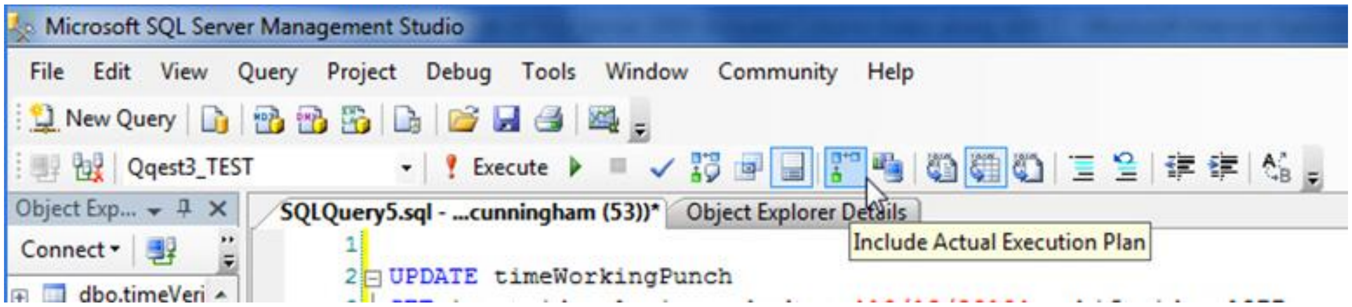
After reviewing the query list it can be seen that many of the queries are being filtered on the WorkingPunchID field in the WHERE clause. This leads to an exploratory mode and testing of how a new clustered index on that field would perform. Now the table will be ordered in a way that will make this field easier for the database to process. We'll test that theory soon.


dbo.timeWorkingPunch
Columns
workingpunch_id (PK, int, not null)
company_id (int, not null)
user_id (int, not null)
employee_id (int, not null)
rawpunch_id (int, not null)
clockdownload_id (int, not null)
workingpunch_ts (datetime, not nul
inpunch_dt (datetime, not null)
inout_id (int, not null)

WorkingPunch_ID has correctly been setup as a Primary Key(PK). This an AutoIncrementing field that assures that the table has one and only one value for this key within the whole table. This makes the field have a 'LOW' selectivity. This makes the values in this field very easy to find.

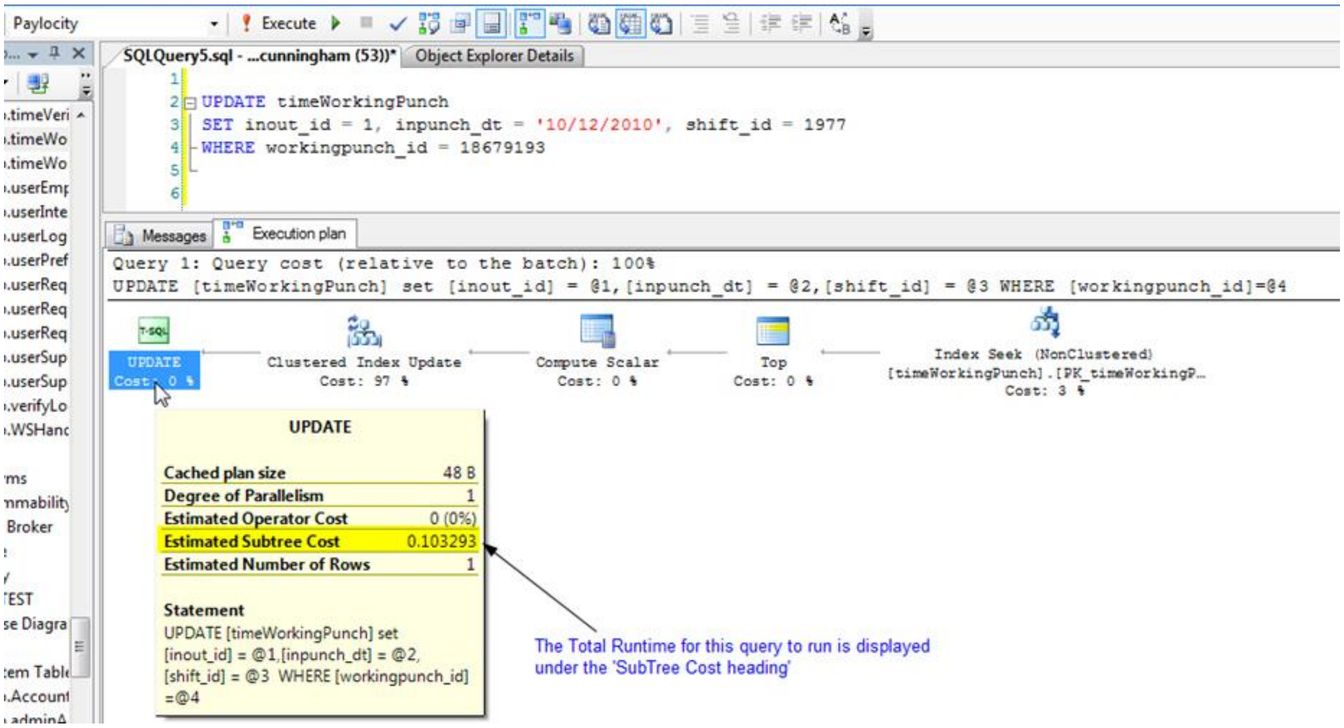
Turn on the SQL Server Display Execution Plan Option

Now we're ready to start running queries and evaluating runtimes against our new structures as compared to the version currently in production. We'll turn on the Display Execution Plan option so we can do our comparisons. Select this button inside SSMS to get a tab that shows the Execution Plan of your query.


Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
Qqest3_TEST
Execute
Object Exp...
Connect
dbo.timeVeri
SQLQuery5.sql - ...cunningham (53))*
Object Explorer Details
Include Actual Execution Plan

```
UPDATE timeWorkingPunch
```

Run your query and pay attention to the Subtree cost entry at the beginning of the execution plan as you read it from left to right.


Paylocity
Execute
SQLQuery5.sql - ...cunningham (53))*
Object Explorer Details
Messages
Execution plan
UPDATE
Cost: 0 %
Clustered Index Update
Cost: 97 %
Compute Scalar
Cost: 0 %
Top
Cost: 0 %
Index Seek (NonClustered)
[timeWorkingPunch].[PK_timeWorkingP...
Cost: 3 %
The Total Runtime for this query to run is displayed under the 'SubTree Cost heading'

```
UPDATE timeWorkingPunch
SET inout_id = 1, inpunch_dt = '10/12/2010', shift_id = 1977
WHERE workingpunch_id = 18679193
```

```
Query 1: Query cost (relative to the batch): 100%
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt] = @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

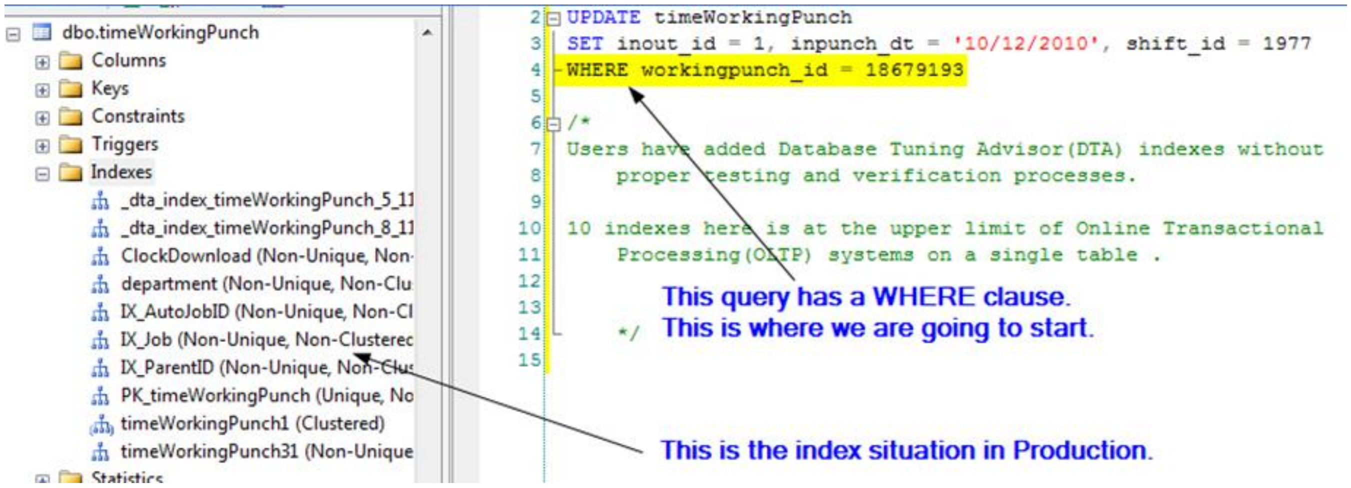
UPDATE

Cached plan size	48 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.103293
Estimated Number of Rows	1

Statement
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt] = @2, [shift_id] = @3 WHERE [workingpunch_id] =@4

Dissecting the Query

Now we get to compare how the production copy of the database table will perform as compared to the new table structure with its new indexes. We want to start at the WHERE clause of the query. The tables listed in any joins are also to be evaluated, but for this example UPDATE statement we'll stick to the WHERE clause.

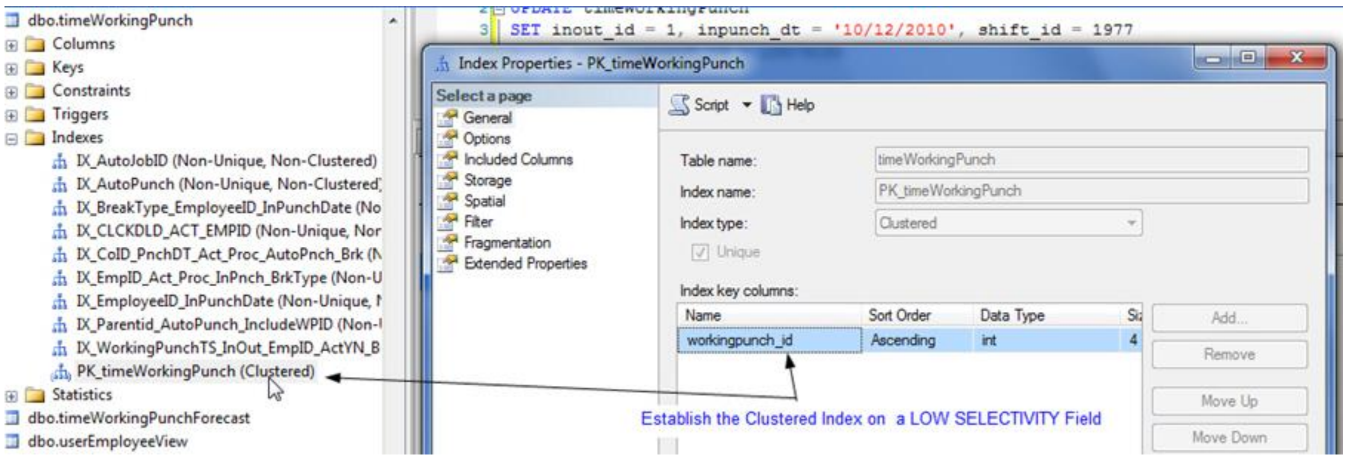


```
dbo.timeWorkingPunch
  Columns
  Keys
  Constraints
  Triggers
  Indexes
    _dta_index_timeWorkingPunch_5_11
    _dta_index_timeWorkingPunch_8_11
    ClockDownload (Non-Unique, Non-
    department (Non-Unique, Non-Clu
    IX_AutoJobID (Non-Unique, Non-Cl
    IX_Job (Non-Unique, Non-Clusterec
    IX_ParentID (Non-Unique, Non-Clus
    PK_timeWorkingPunch (Unique, No
    timeWorkingPunch1 (Clustered)
    timeWorkingPunch31 (Non-Unique
  Statistics
```

```
UPDATE timeWorkingPunch
SET inout_id = 1, inpunch_dt = '10/12/2010', shift_id = 1977
WHERE workingpunch_id = 18679193

/*
Users have added Database Tuning Advisor(DTA) indexes without
    proper testing and verification processes.

10 indexes here is at the upper limit of Online Transactional
    Processing(OLTP) systems on a single table .

    */
```

This query has a WHERE clause.
This is where we are going to start.
This is the index situation in Production.

The Primary Key (PK) that we established to build our clustered index on in our testing database is being used to identify the row to be updated.



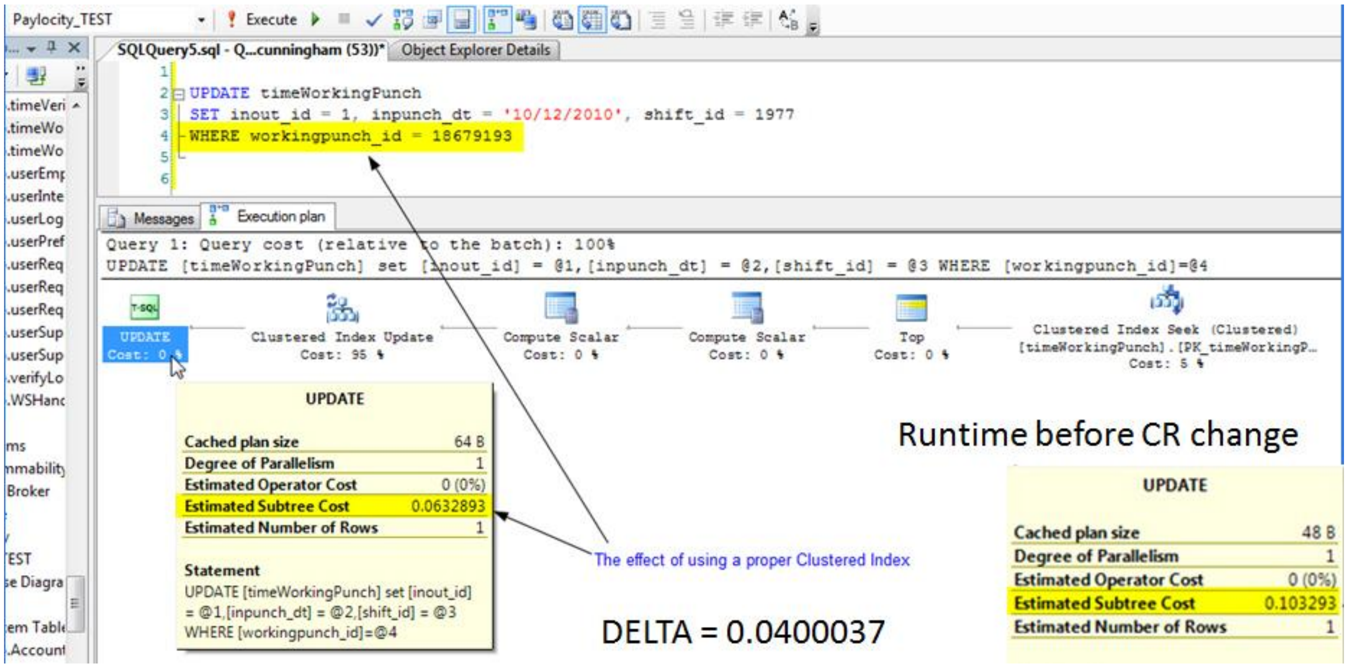
```
dbo.timeWorkingPunch
  Columns
  Keys
  Constraints
  Triggers
  Indexes
    IX_AutoJobID (Non-Unique, Non-Clustered)
    IX_AutoPunch (Non-Unique, Non-Clustered)
    IX_BreakType_EmployeeID_InPunchDate (No
    IX_CLCKDLD_ACT_EMPID (Non-Unique, Nor
    IX_CoID_PnchDT_Act_Proc_AutoPnch_Brk (N
    IX_EmpID_Act_Proc_InPnch_BrkType (Non-U
    IX_EmployeeID_InPunchDate (Non-Unique, N
    IX_Parentid_AutoPunch_IncludeWPID (Non-I
    IX_WorkingPunchTS_InOut_EmpID_ActYN_B
    PK_timeWorkingPunch (Clustered)
  Statistics
dbo.timeWorkingPunchForecast
dbo.userEmployeeView
```

```
SET inout_id = 1, inpunch_dt = '10/12/2010', shift_id = 1977
```

Table name: timeWorkingPunch
Index name: PK_timeWorkingPunch
Index type: Clustered
Unique [✓]
Index key columns:

Name	Sort Order	Data Type	Si
workingpunch_id	Ascending	int	4

The new structure generates a run time
of:



```
UPDATE timeWorkingPunch
SET inout_id = 1, inpunch_dt = '10/12/2010', shift_id = 1977
WHERE workingpunch_id = 18679193
```

```
Query 1: Query cost (relative to the batch): 100%
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt] = @2,[shift_id] = @3 WHERE [workingpunch_id]=@4
```

UPDATE

Cached plan size	64 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0632893
Estimated Number of Rows	1

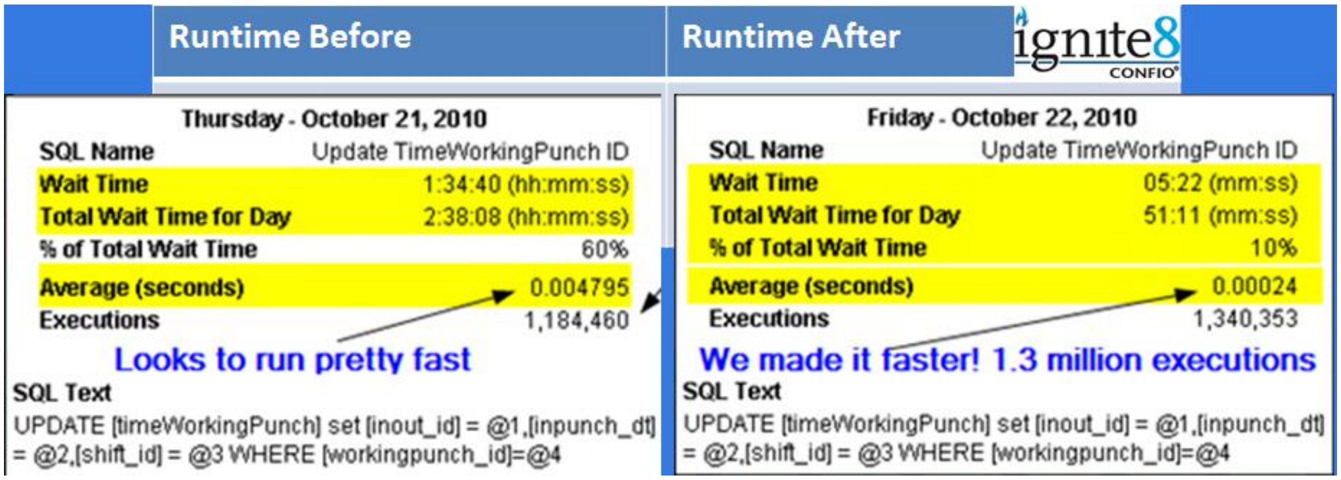
Statement
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt] = @2,[shift_id] = @3 WHERE [workingpunch_id]=@4

UPDATE

Cached plan size	48 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.103293
Estimated Number of Rows	1

DELTA = 0.0400037

This small change in runtime (the Delta value above), when executed over a million times resulted in a significant reduction in SQL server wait time for the end user. This became the template that was used to further detect and tune the database index structures.



Runtime Before

Thursday - October 21, 2010

SQL Name	Update TimeWorkingPunch ID
Wait Time	1:34:40 (hh:mm:ss)
Total Wait Time for Day	2:38:08 (hh:mm:ss)
% of Total Wait Time	60%
Average (seconds)	0.004795
Executions	1,184,460

SQL Text
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt] = @2,[shift_id] = @3 WHERE [workingpunch_id]=@4

Runtime After

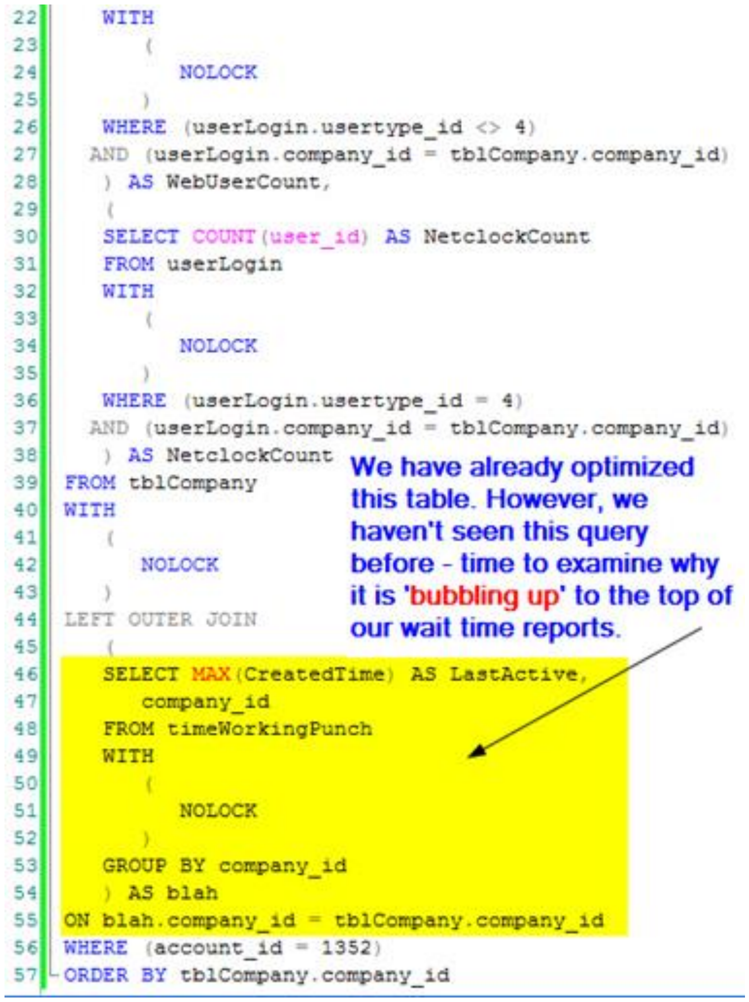
Friday - October 22, 2010

SQL Name	Update TimeWorkingPunch ID
Wait Time	05:22 (mm:ss)
Total Wait Time for Day	51:11 (mm:ss)
% of Total Wait Time	10%
Average (seconds)	0.00024
Executions	1,340,353

SQL Text
UPDATE [timeWorkingPunch] set [inout_id] = @1,[inpunch_dt] = @2,[shift_id] = @3 WHERE [workingpunch_id]=@4

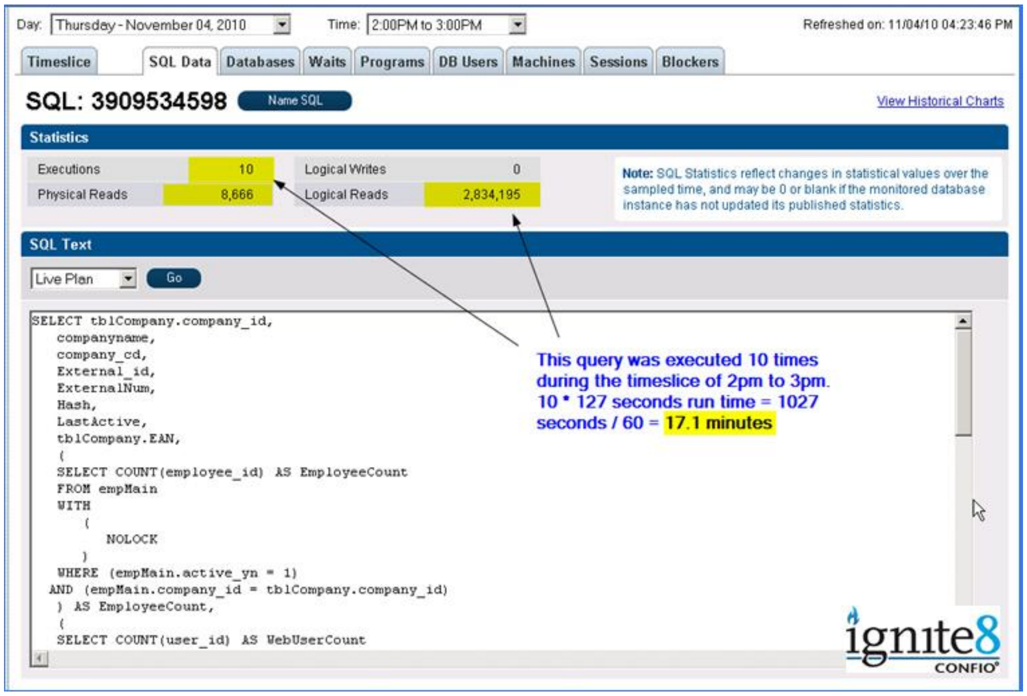
What happens now? Repeat!

After documenting the success of this process to upper management, it was time to revisit the Ignite tool to see what new queries were 'bubbling up' towards the top of the wait time matrix. A query became exposed that had somehow evaded the initial trace capture of the queries hitting this table.



```
    WITH
        (
            NOLOCK
        )
    WHERE (userLogin.usertype_id <> 4)
   AND (userLogin.company_id = tblCompany.company_id)
    ) AS WebUserCount,
    (
    SELECT COUNT(user_id) AS NetclockCount
    FROM userLogin
    WITH
        (
            NOLOCK
        )
    WHERE (userLogin.usertype_id = 4)
   AND (userLogin.company_id = tblCompany.company_id)
    ) AS NetclockCount
FROM tblCompany
WITH
    (
        NOLOCK
    )
LEFT OUTER JOIN
    (
    SELECT MAX(CreatedTime) AS LastActive,
        company_id
    FROM timeWorkingPunch
    WITH
        (
            NOLOCK
        )
    GROUP BY company_id
    ) AS blah
ON blah.company_id = tblCompany.company_id
WHERE (account_id = 1352)
ORDER BY tblCompany.company_id
```

We have already optimized this table. However, we haven't seen this query before - time to examine why it is 'bubbling up' to the top of our wait time reports.



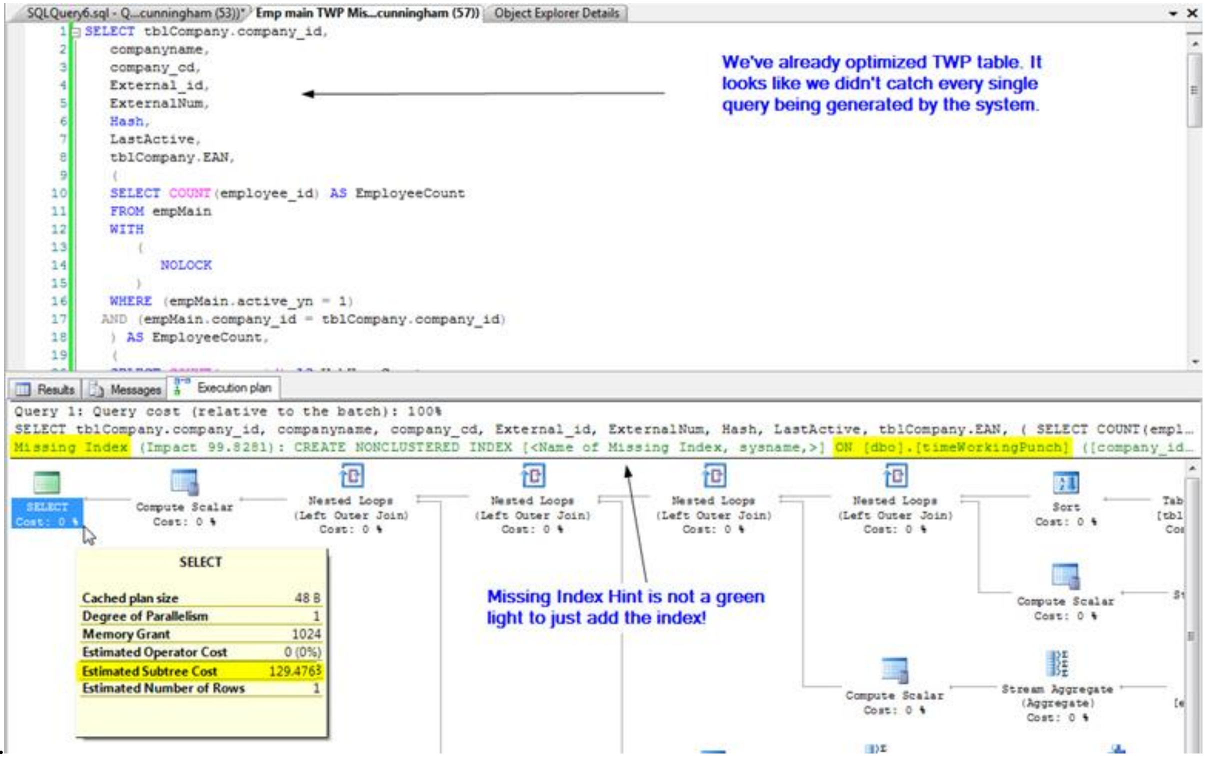
Executions	10	Logical Writes	0
Physical Reads	8,666	Logical Reads	2,834,195

Note: SQL Statistics reflect changes in statistical values over the sampled time, and may be 0 or blank if the monitored database instance has not updated its published statistics.

```
SELECT tblCompany.company_id,
   companyname,
   company_cd,
   External_id,
   ExternalNum,
   Hash,
   LastActive,
   tblCompany.EAN,
   (
   SELECT COUNT(employee_id) AS EmployeeCount
   FROM empMain
   WITH
      (
         NOLOCK
      )
   WHERE (empMain.active_yn = 1)
  AND (empMain.company_id = tblCompany.company_id)
   ) AS EmployeeCount,
   (
   SELECT COUNT(user_id) AS WebUserCount
```

This query was executed 10 times during the timeslice of 2pm to 3pm. 10 * 127 seconds run time = 1027 seconds / 60 = 17.1 minutes

By this time the new index structures have been in production for a week or so and response time is improving. The lack of customer support calls meant that our changes had performed as expected. The new query that was now to be evaluated had an astonishing result when it's runtime was tested in the old production database. This query took 129.476 seconds to execute. Also, please make a note of the Missing Index hint (in green) that SSMS is showing. Let's first take a look at that.


SQLQuery6.sql - Q...cunningham (53))*
Emp main TWP Mis...cunningham (57))
Object Explorer Details
We've already optimized TWP table. It looks like we didn't catch every single query being generated by the system.
Results
Messages
Execution plan
SELECT Cost: 0 %
Compute Scalar Cost: 0 %
Nested Loops (Left Outer Join) Cost: 0 %
Nested Loops (Left Outer Join) Cost: 0 %
Nested Loops (Left Outer Join) Cost: 0 %
Nested Loops (Left Outer Join) Cost: 0 %
Sort Cost: 0 %
Compute Scalar Cost: 0 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Missing Index Hint is not a green light to just add the index!

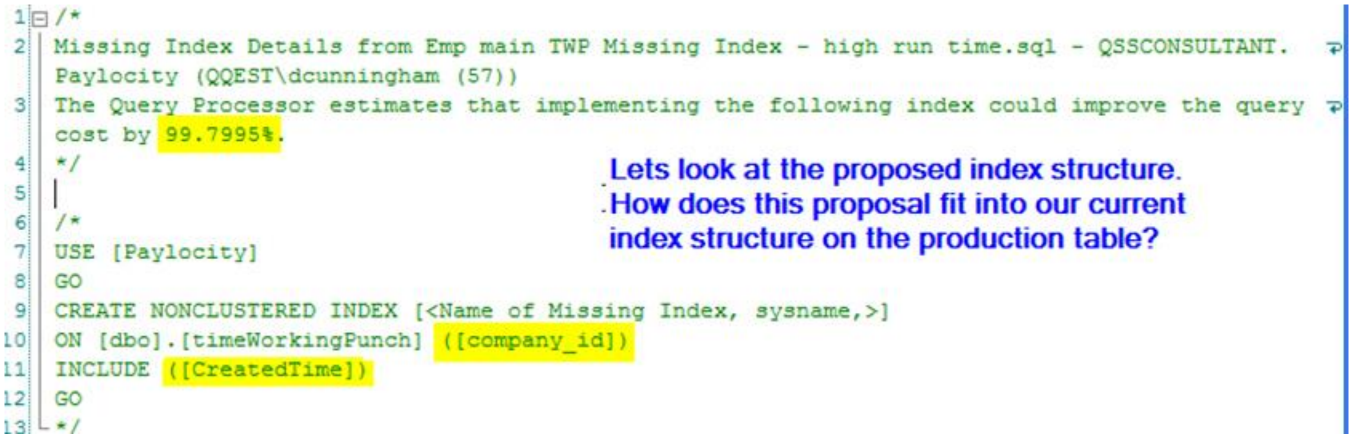
```
SELECT tblCompany.company_id,
    companyname,
    company_cd,
    External_id,
    ExternalNum,
    Hash,
    LastActive,
    tblCompany.EAN,
    (
    SELECT COUNT(employee_id) AS EmployeeCount
    FROM empMain
    WITH
        (
            NOLOCK
        )
    WHERE (empMain.active_yn = 1)
   AND (empMain.company_id = tblCompany.company_id)
    ) AS EmployeeCount,
    (
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT tblCompany.company_id, companyname, company_cd, External_id, ExternalNum, Hash, LastActive, tblCompany.EAN, ( SELECT COUNT(empl…
Missing Index (Impact 99.8281): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[timeWorkingPunch] ([company_id…
```

SELECT	
Cached plan size	48 B
Degree of Parallelism	1
Memory Grant	1024
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	129.4763
Estimated Number of Rows	1

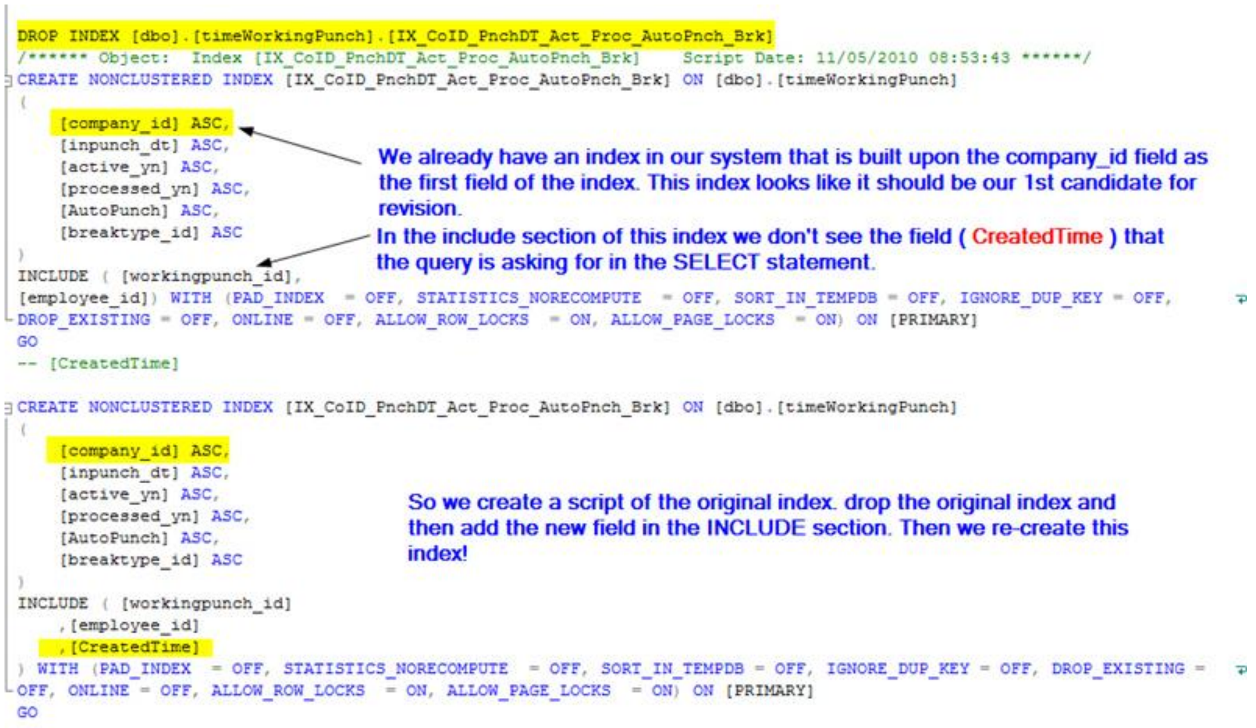
SSMS Offers Us Some Help

We know that we've already optimized this table during our initial investigation and CR changes. Is it possible that this index hint can be integrated into the new Index structure without adding a new index? Here's the full index hint.


Lets look at the proposed index structure.
How does this proposal fit into our current index structure on the production table?

```
/*
Missing Index Details from Emp main TWP Missing Index - high run time.sql - QSSCONSULTANT.
Paylocity (QQEST\dcunningham (57))
The Query Processor estimates that implementing the following index could improve the query
cost by 99.7995%.
*/

/*
USE [Paylocity]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [dbo].[timeWorkingPunch] ([company_id])
INCLUDE ([CreatedTime])
GO
*/
```

Here is our current index that is built on the filtered field company_id.



```
DROP INDEX [dbo].[timeWorkingPunch].[IX_CoID_PnchDT_Act_Proc_AutoPnch_Brk]
/****** Object:  Index [IX_CoID_PnchDT_Act_Proc_AutoPnch_Brk]    Script Date: 11/05/2010 08:53:43 ******/
CREATE NONCLUSTERED INDEX [IX_CoID_PnchDT_Act_Proc_AutoPnch_Brk] ON [dbo].[timeWorkingPunch]
(
    [company_id] ASC,
    [inpunch_dt] ASC,
    [active_yn] ASC,
    [processed_yn] ASC,
    [AutoPunch] ASC,
    [breaktype_id] ASC
)
INCLUDE ( [workingpunch_id],
[employee_id]) WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF,
DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
GO
-- [CreatedTime]
```

We already have an index in our system that is built upon the company_id field as the first field of the index. This index looks like it should be our 1st candidate for revision.

In the include section of this index we don't see the field (CreatedTime) that the query is asking for in the SELECT statement.

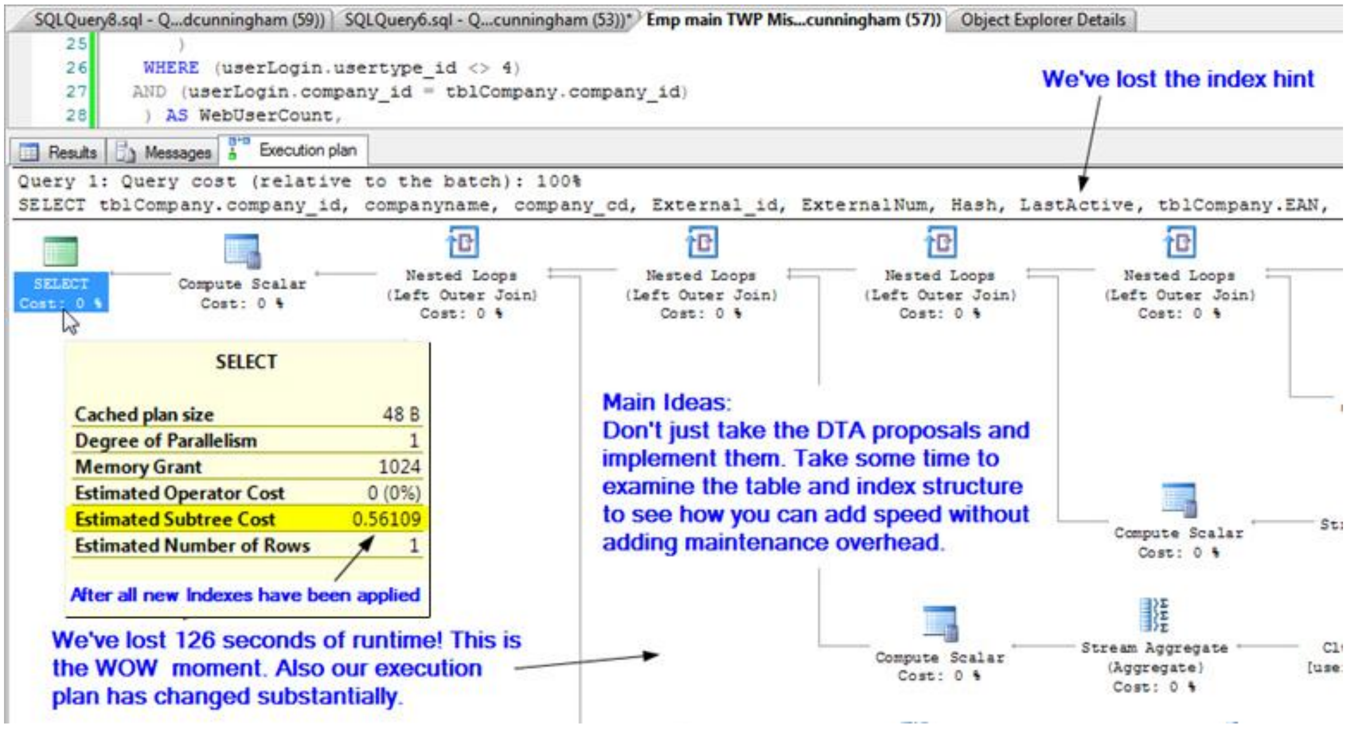
```
CREATE NONCLUSTERED INDEX [IX_CoID_PnchDT_Act_Proc_AutoPnch_Brk] ON [dbo].[timeWorkingPunch]
(
    [company_id] ASC,
    [inpunch_dt] ASC,
    [active_yn] ASC,
    [processed_yn] ASC,
    [AutoPunch] ASC,
    [breaktype_id] ASC
)
INCLUDE ( [workingpunch_id]
    ,[employee_id]
    ,[CreatedTime]
) WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF, DROP_EXISTING =
OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
GO
```

So we create a script of the original index. drop the original index and then add the new field in the INCLUDE section. Then we re-create this index!

Since we already have an index on the company_id field we move our attention to the SELECT statement. Here we see the requested field as: SELECT MAX(CreatedTime) as LastActive. The field CreatedTime is not listed in our INCLUDE part of the index. So we add this field into the index creation script. After dropping the original index and rebuilding the index with the new field it's time to test our performance.

The End Game

The test run gives me back an astounding result as a run time for this new index structure:



SQLQuery8.sql - Q...dcunningham (59)) | SQLQuery6.sql - Q...cunningham (53))* | Emp main TWP Mis...cunningham (57)) | Object Explorer Details
Results | Messages | Execution plan
We've lost the index hint
SELECT Cost: 0 %
Compute Scalar Cost: 0 %
Nested Loops (Left Outer Join) Cost: 0 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %

```
         )
    WHERE (userLogin.usertype_id <> 4)
   AND (userLogin.company_id = tblCompany.company_id)
    ) AS WebUserCount,
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT tblCompany.company_id, companyname, company_cd, External_id, ExternalNum, Hash, LastActive, tblCompany.EAN,
```

SELECT	
Cached plan size	48 B
Degree of Parallelism	1
Memory Grant	1024
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.56109
Estimated Number of Rows	1

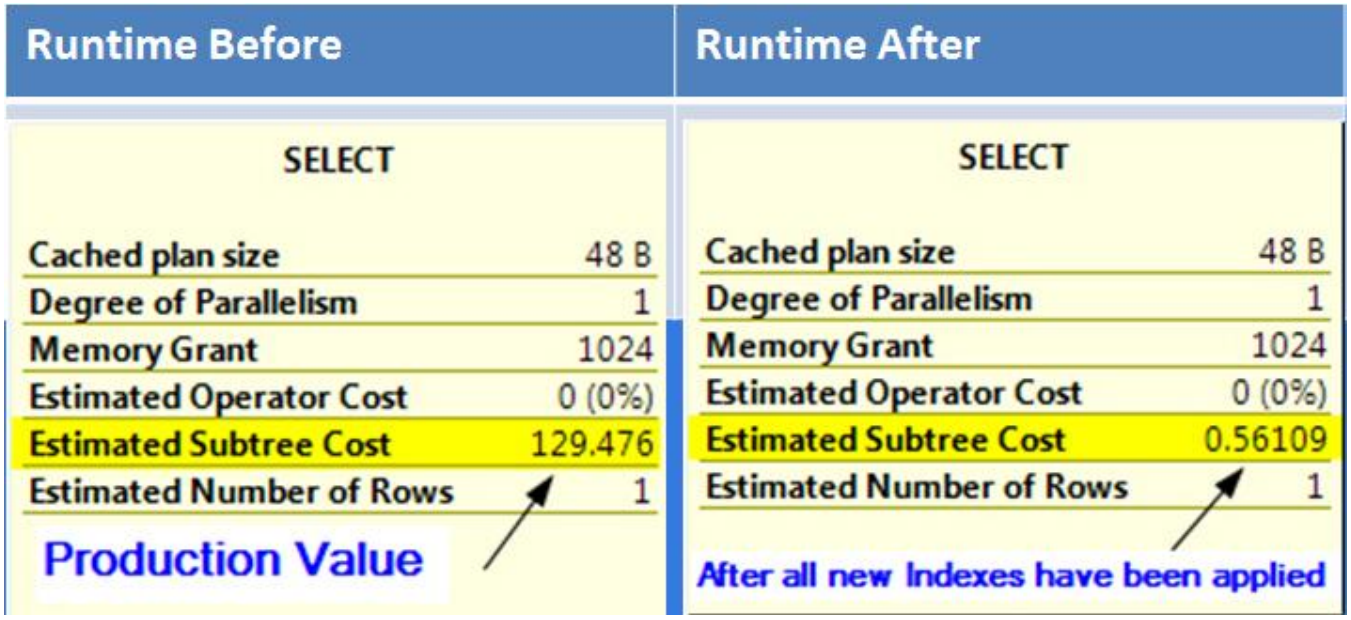
After all new Indexes have been applied

We've lost 126 seconds of runtime! This is the WOW moment. Also our execution plan has changed substantially.

Main Ideas:
Don't just take the DTA proposals and implement them. Take some time to examine the table and index structure to see how you can add speed without adding maintenance overhead.

Conclusion

After spending many hours of testing and tuning the index structures the final evaluation of this queries runtime offers a positive result.



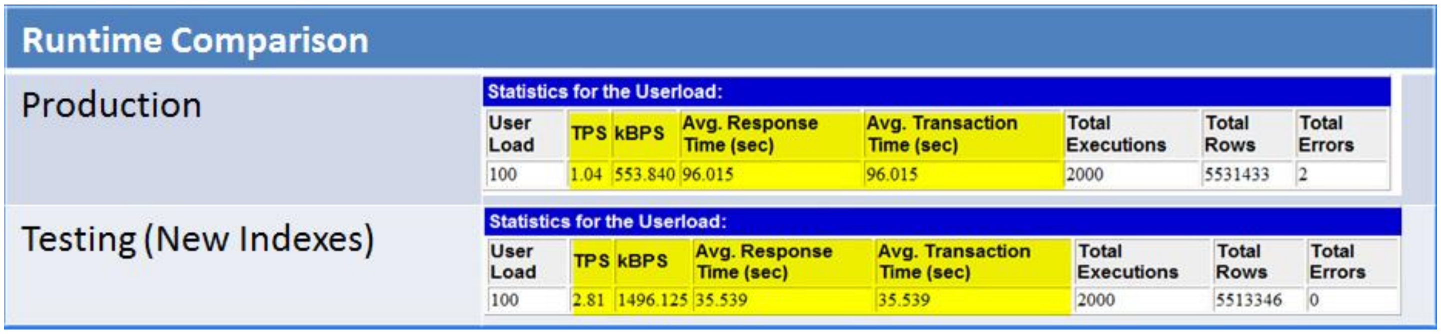
Runtime Before		Runtime After	
SELECT		SELECT	
Cached plan size	48 B	Cached plan size	48 B
Degree of Parallelism	1	Degree of Parallelism	1
Memory Grant	1024	Memory Grant	1024
Estimated Operator Cost	0 (0%)	Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	129.476	Estimated Subtree Cost	0.56109
Estimated Number of Rows	1	Estimated Number of Rows	1

Production Value

After all new Indexes have been applied

We then wanted to see this complete set of queries (in the Excel CSV file) run under a simulated load, to see how the proposed indexes would perform. Below is shown two reports from this tool. The first image reflects many important performance metrics against a copy of our production database. The second image is a report when the tool was pointed with the same workload files against the modified 'Testing' database.. All of the unique queries (over 260 of them) were executed in a load simulating 100 users. The final results were rewarding and left a warm fuzzy feeling inside.

This image displays a final verification that the proposed index optimizations will be an improvement in database performance and in query response time



Runtime Comparison

Production

Statistics for the Userload:

User Load	TPS	kBPS	Avg. Response Time (sec)	Avg. Transaction Time (sec)	Total Executions	Total Rows	Total Errors
100	1.04	553.840	96.015	96.015	2000	5531433	2

Testing (New Indexes)

Statistics for the Userload:

User Load	TPS	kBPS	Avg. Response Time (sec)	Avg. Transaction Time (sec)	Total Executions	Total Rows	Total Errors
100	2.81	1496.125	35.539	35.539	2000	5513346	0

About the Author

Dale Cunningham is a Sr. Database Administrator and has worked for enterprises both large and small. As a sole DBA for Intuit's CustomerCentral databases which supported the products of Quicken Online (now consolidated with the Mint application that was acquired by Intuit), QuickBooks Online, Quicken Desktop and Finance Works he was provided an environment of very large OLTP transactions per second and a 1.6 TB database to manage in a clustered environment. Many other companies have provided experience in getting the most performance as possible from SQL Server 2000, 2005 and 2008.

Dale has been involved with computers since 1984. The IBM XT model was the hook and the IT bug bite never healed. Beginning a degree in computer science at FIU with an AA degree from Broward Community College in hand, Data structures, C++ and Java became mandatory languages to learn. Then it all changed when a database class ignited the fire of what this wonderful invention from Sybase and eventually Microsoft could do. SQL server became integrated into every website he built in classic ASP 3, VBScript and JavaScript.

Along the way many people have participated in the development of Dale as a DBA, mainly the family members that endured the numerous hours he spent reading, testing and developing SQL scripts and ASP code. Dale achieved the MCDBA certification from Microsoft for SQL Server 2000 and

sunshinesoftware

is looking forward to becoming certified in SQL Server 2008. Thanks to Jack Martin, Steven Benjamin, Rafael van Dyke, TJay Belt and Rick Morelan of the Joes to Pros series, for jobs, friendship and additional training.

Thanks to Confio Software and Microsoft for supplying the software tools to make this project report possible – and easier!

Download a free trial of Confio Ignite from www.confio.com .