SQL Query Optimization Part 2
Analyzing SQL Query Plans
With the Query Plan Viewer

A White Paper From

For more information, see our web site at
http://www.goldstarsoftware.com

SQL Query Optimization Part 2:

Analyzing SQL Query Plans with the Query Plan Viewer

Last Updated: 02/04/2022

Note #1: *This paper is part 2 of a multi-part series on SQL query optimization. You may wish to start with Part 1 before looking at the other parts, as each takes a specific area of query optimization and breaks it down into manageable steps.*

Note #2: *This paper includes screenshots from the Actian Zen v15 product specifically, but the information presented herein relates to all versions of the Actian Zen, Actian PSQL, and Pervasive PSQL – all the way back to the Pervasive.SQL 2000i, in fact!*

Introduction

One common complaint we hear about the Actian PSQL/Zen database environment is that SQL queries are running "slowly" for some reason. Now, it is known that the SQL Relational Database Engine (SRDE) is internally single-threaded, so working on large data sets won't see parallelization that other SQL engines may offer, but "slow" queries are usually the result of a SQL query that is either improperly designed, overly complicated, or simply a query that is requiring a lot of effort to complete.

Before a query can be optimized, we need to first understand the workload of our query so that we can understand exactly what the database engine is attempting to do when it runs the query. Once we know how much work was required to run the query in its current state, we can then analyze the query in detail to see exactly what decisions the database engine made about the data and how the data was eventually accessed at the Microkernel level. Finally, we can make small changes to the query in an attempt to optimize the query. As we make the changes, we can then observe the difference in the workload and decide if the change we made is good or bad.

If you have not yet done so, please review our white paper titled ***Determining the SQL Workload of a Query on Actian Zen*** so that you can properly ascertain the current workload required by the query. With this information, you won't just be stumbling around in the dark and making guesses, but you'll be able to see incremental improvements in your queries and know when you're going in the right direction.

This paper covers the second part of the process, capturing the SQL query plan and analyzing that plan to understand how the engine worked through the query.

What Is a Query Plan?

When the SRDE receives a query, it arrives as a simple string of text, such as "SELECT * FROM Person". The SQL engine is then responsible for tokenizing the query, which essentially picks apart the text into the various tokens, such as SELECT, *, FROM, and Person. The query is then parsed to validate syntax and field and table names (from the dictionary) and ensure that all required tokens are provided for each clause. Next, the query is optimized, which is a process that extracts key information from the query and from the data dictionary to determine the most efficient way of divining the proper result

set. The optimizer may shuffle the order that tables are accessed, and it has to make choices (sometimes difficuly ones) to find the most efficient way to calculate the result set for the query. This process results in an access plan, which is then executed to actually retrieve the data.

In essence, the *query plan* is the culmination of all of the preparatory work, and it indicates to the engine how the data will actually be extracted from the environment. It may be possible that the engine may generate multiple plans and then picks the best one, or the engine may simply plow forward with what it thinks is the best plan given the information known at the time. Thus the query plan becomes the blueprint for the execution phase, and will directly reflect how the data is ultimately accessed.

It is important to note that the query plan is built with imperfect information. While we know the data structures and indices involved (because these are in the data dictionary), we really have limited insight into the DATA inside the tables themselves. As such, the query plan chosen by the SRDE may not be the most efficient one available – this is where query hinting and forcing the order of tables becomes quite important.

Capturing the Query Plan

Capturing the query plan from the SQL engine for any given query is fairly simple. The engine supports a series of SET operations that control the capture into a query plan file (or QPF). Here is a basic sequence of events:

```
SET QryPlanOutput = 'C:\Data\QryPlans.qpf';
SET QryPlan = ON;
<Place your query here>;
SET QryPlan = OFF;
```

Let's look at each of these statements in turn.

The first statement here sets the output file for the query plan to a specific location. You can supply any file name you like here, but remember that the Query Plan Viewer isa going to look for files with a QPF extension, so sticking with that at the end is best. Also, remember that this statement is being executed by the engine, which may be on a remote server. The path provided here is an *engine-relative* path, so the above statement will actually create the query plan file on the server's C: drive, not on the workstation's C: drive. To make this as easy as possible, we recommend dropping the QPF file into a folder which is readily accessed by workstations, such as the same folder as the database or dictionary files. Otherwise, you'll find that you have to copy the file around in order to use it.

The second statement here turns the query plan capture ON, which means that all subsequent SQL queries executed within this session will have plans captured to the indicated file.

Now that query plan capture is enabled, you can run your own SQL statements and queries. If you want to work on a bunch of them at once, you can run each statement in a row and capture all of the statements into the same query plan file.

The last statement should be intuitive enough – it turns off the query plan capture. This allows you to run other SQL statements (such as COUNT() queries) while you are optimizing your target query without adding a bunch of extra "gunk" to the QPF file.

One other note: while the query plan capture doesn't slow things down very much, it is considered a bad idea to leave the query planner turned on for a long period of time. There have been instances in the distant past where the SQL engine has crashed when certain complicated queries were submitted with planning enabled.

Using the Query Plan Viewer

Now that you have a QPF file captured, you can open it with the **Query Plan Viewer**. The installer will not automatically link QPF files with the Query Plan Viewer, so you won't be able to just double-click on the query file. Instead, you will need to first start the Query Plan Viewer and then open the file.

To start the **Query Plan Viewer**, select the tool from the **Zen Control Center**'s *Tools* menu. Alternatively, you can click the Start menu and start typing "Zen Query Plan Viewer" in the search bar, which should find it as well. Another option is to run the program executable itself, which is "W3SQLQPV.EXE", from the Start menu or from any Command Prompt.

Once loaded, you'll see two blank panes. Go to the *File/Open* menu option and select the QPF file you wish to load up. (If you didn't use the QPF extension for your plan file, be sure to select "*All Files (*.*)*" in the lower right corner of the dialog or you'll never see it.) The last query from that query plan file should now be presented to you in the left side pane, and the graphical query plan will be showing on the right side.

If your QPF file includes multiple statements, then you may wish to open the *View* menu and select *First* (or press Ctrl-F) to jump to the first query captured. Other links and shortcuts are available for the *Next* (Ctrl-N), *Previous* (Ctrl-P), and the *Last* (Ctrl-L), as well as *GoTo* (Ctrl-G), which allows you to jump to any query by number.

It is unusual, but you can actually continue running the query capture while you are examining the plans. However, to load any query plans that were captured AFTER you loaded the file into the Query Plan Viewer, use the *File/Refresh* option. This will re-load the file and place you again on the last query.

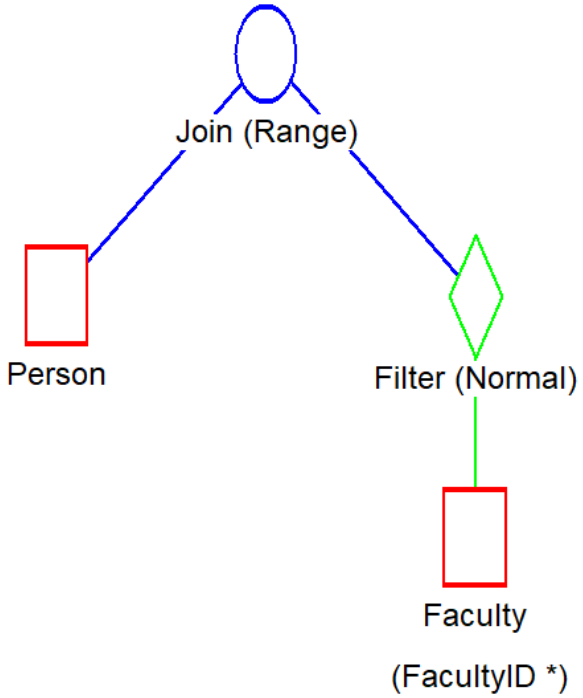
On the right-side pane, the graphical description of the query plan will be shown. If you have a complicated query that leverages one or more subqueries, the *SubQuery* menu option will allow you to view either the individual subqueries or the root query. Once you have the right query shown, the *View* menu offers sizing and scrolling options which should be fairly intuitive.

Interpreting the Query Plan

For the purpose of learning how to interpret the query plan, we are going to start with a very simple query from the **DEMODATA** database, so that you can play along on your own system, if you wish. The query we are going to start with is a simple join of the **Person** and **Faculty** table that extracts 3 fields:

```
SELECT Last_Name, First_Name, Salary
FROM Person INNER JOIN Faculty ON (Person.ID = Faculty.ID);
```

When we open this query plan, we get the following image:



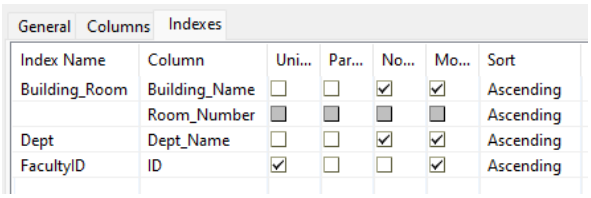
This looks pretty cryptic, but we'll go through each piece of the puzzle, and you'll see that reading this display is actually easier than it looks. When examining any plan, just remember to always read this diagram from the left to right, as this tells you the order of the table accesses chosen by the engine.

So, the first table listed here is the **Person** table. The absence of any other text below it means that the engine did not choose any specific index to access the data in this table. Instead, it is going to perform Step operations (i.e. StepFirst, StepNext, etc.) to read the data in physical order. [N.b. While reading only data records is usually faster than reading data by an index, files with a large percentage of deleted record slots can see a serious performance penalty as the empty slots have to be skipped as they are found. As such, this may be a place where an optimization can be found!]

If we do a quick `SELECT COUNT(*) FROM Person` query, we'll find that there are 1500 records in this table. This means that the engine will be reading each of the 1500 records in turn and passing them up to the Join layer, without any filtering.

Next, we look at the second table, **Faculty**. This icon differs in that the Index Name **FacultyID** is indicated below the table name. This tells me that this file will be accessed through this index, which also happens to be a unique key (as indicated by the *).

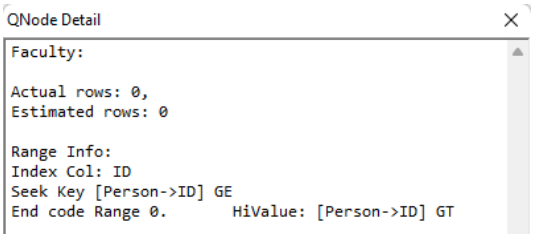
To see what field or fields comprise the **FacultyID** key, we go back to the **Control Center**, right-click the **Person** table, and select *Properties* and then go to the *Indexes* tab:



General Columns Indexes

Index Name	Column	Uni...	Par...	No...	Mo...	Sort
Building_Room	Building_Name	☐	☐	☑	☑	Ascending
	Room_Number	■	■	■	■	Ascending
Dept	Dept_Name	☐	☐	☑	☑	Ascending
FacultyID	ID	☑	☐	☐	☑	Ascending

From here, we can see that the **FacultyID** index is comprised of just one field (**ID**) and is a unique key stored in ascending order. In order to gain visibility into the logic of the **Faculty** table accesses, we then double-click the box in the graphic, and we get a dialog box containing the *query node detail*:



QNode Detail

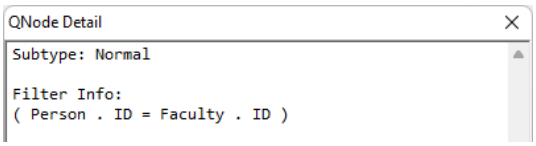
```
Faculty:

Actual rows: 0,
Estimated rows: 0

Range Info:
Index Col: ID
Seek Key [Person->ID] GE
End code Range 0.       HiValue: [Person->ID] GT
```

I don't know where the data for the *Actual rows* and *Estimated rows* comes from, but I've found that it is often incorrect, so don't put too much stock in those numbers. If you care how many records are available, you can issue another COUNT(*) query on this table to learn that there are 96 rows in the **Faculty** table within the **DEMODATA** database. However, since we are leveraging an index and not reading every record in a table-scan, this record count is less interesting at this time.

The real useful piece here is the *Range Info,* which tells us how the engine will be extracting data from the chosen key. This format is a bit strange, but it will eventually become easy enough to read with some practice.

In this case, our index is providing a column called "**ID**" (in the **Faculty** table). We are seeking on this index for all records that are greater or equal (*GE*) to a provided value – in this case the **ID** field that came from the **Person** table (**Person->ID**). We will stop (*End*) the search after the "*HiValue*" is exceeded, which in this case is any record that is greater than (*GT*) the **Person->ID** value. If you examine the logic carefully, you'll see that this will match all records with the same **ID** value, exactly as expected.

Double-clicking on the *Filter* node above the **Faculty** table, we see the filter that is handled here:



QNode Detail

```
Subtype: Normal

Filter Info:
( Person . ID = Faculty . ID )
```

This filter ensures that ONLY those combinations of records where **Person.ID** is equal to **Faculty.ID** will pass up to the Join above it, effectively creating an INNER JOIN, also known as an EQUAL JOIN.

One thing that you may have noticed is that the field list being returned by the query is essentially immaterial at this level, and it can be ignored. This is because when a record is accessed, the entire record is accessed as part of the engine workload, so all fields are

available with equal speed. (Of course, processing data, sorting data, and returning more fields will imply more work on the SQL engine and client, but this is unrelated to the query plan.)

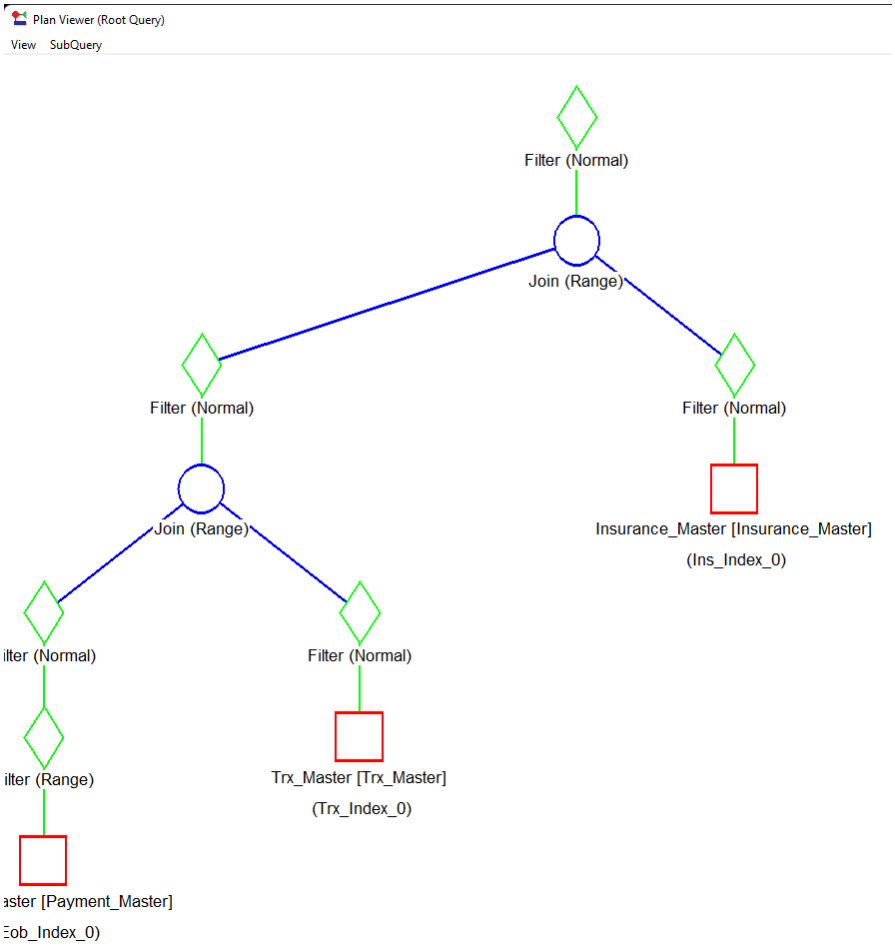
Of course, we have only just barely scratched the surface here in this section with a very simple query. As the query complexity increases (i.e. number of tables, types of joins, filters, ORDER BY clauses, etc.), so, too, will the complexity of the graphical image, as well as the complexity of the data hiding behind each node (when you double-click it). We recommend reviewing the Query Plan Viewer documentation in the Actian online help system, and beyond that, just play around with it! The more experience you have, the easier it will be to see and understand how your query is working.

A More Complex Example

In Part 1 of this series, we used the Monitor utility to watch a query (provided by a customer) that generated a huge workload – almost 100 million page accesses. This query took a REALLY long time to run (measured in hours) on a fast server. Here is the original query, which has been scrubbed to remove identifying data. We have also removed the field list, as this is immaterial to our discussion at this level to save space:

```
SELECT <field_list>
FROM Insurance_Master, Payment_Master, Trx_Master
WHERE Trx_Master.Trx_Pra_Id = Insurance_Master.Ins_Pra_Id AND
Trx_Master.Trx_Pri_Ins_Id = Insurance_Master.Ins_Id AND
Payment_Master.Eob_Pra_Id = Trx_Master.Trx_Pra_Id AND
Payment_Master.Eob_Pat_Id = Trx_Master.Trx_Pat_Id AND
Payment_Master.Eob_Trx_Seq = Trx_Master.Trx_Seq AND
((Trx_Master.Trx_Pra_Id='XXX') AND (Trx_Master.Trx_Date_From>={d
'2021-01-01'} And Trx_Master.Trx_Date_From<={d '2021-01-31'}) AND
(Trx_Master.Trx_Sec_Flag='Y') AND
(Trx_Master.Trx_Sec_Date_Xmit<={d '2021-11-01'}) AND
(Trx_Master.Trx_Charge_Amount>0) AND
(Insurance_Master.Ins_Pra_Id='XXX') AND
(Payment_Master.Eob_Pra_Id='XXX'))
```

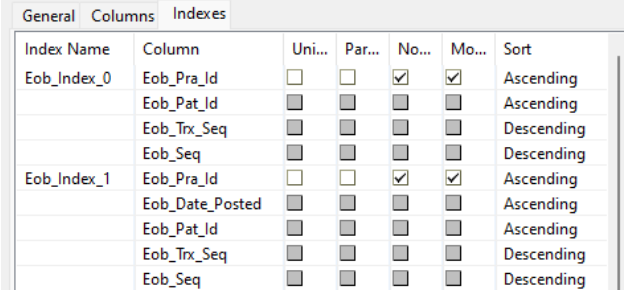
When we open up the QPF file, we get the following image in the viewer:



No, we didn't cut off the left side – sometimes the Query Plan Viewer doesn't scale the image properly in the viewport and ends up chopping data off on you, which can be REALLY annoying. In most cases, you can figure out the missing data without too much trouble, but sometimes you have to use the *File/Export XML* option to see what is hiding behind these graphics here.

So, what does this plan tell us? Remember that we interpret this graphic by starting in the lower left corner and working towards the upper right, so let's examine each of the tables in turn in detail.

The first table selected by the query optimizer is the **Payment_Master** table, which it accessed through the index called **Eob_Index_0**. Unfortunately, the QPV doesn't display the filter for this table properly, so we are not able to double-click on it to see exactly what filter is being used, but we can make some guesses. When we look at the table definition for this index, we see:



General Columns Indexes

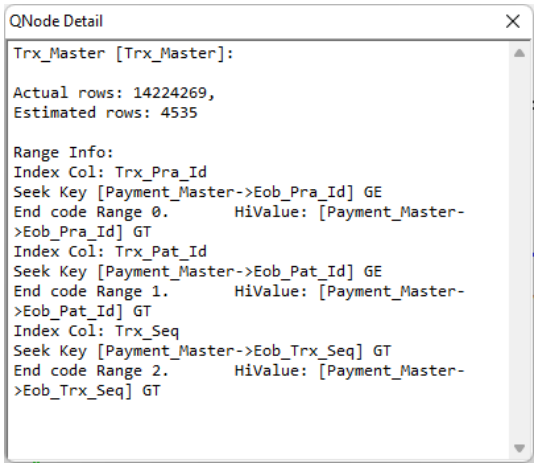
Index Name	Column	Uni...	Par...	No...	Mo...	Sort
Eob_Index_0	Eob_Pra_Id	☐	☐	☑	☑	Ascending
	Eob_Pat_Id	■	■	■	■	Ascending
	Eob_Trx_Seq	■	■	■	■	Descending
	Eob_Seq	■	■	■	■	Descending
Eob_Index_1	Eob_Pra_Id	☐	☐	☑	☑	Ascending
	Eob_Date_Posted	■	■	■	■	Ascending
	Eob_Pat_Id	■	■	■	■	Ascending
	Eob_Trx_Seq	■	■	■	■	Descending
	Eob_Seq	■	■	■	■	Descending

If we look at the WHERE clauses, there is only one WHERE clause that comes into play here, which is the last clause in the original query, so this must be what was optimized:

```
Payment_Master.Eob_Pra_Id='XXX'
```

Now, when we look at the underlying data (using our COUNT() tricks), we can see that the **Payment_Master** table currently has 12,094,355 records, and of those records, all 12,094,355 match this filter. This essentially means that 100% of the records pass the filter and are used to join in the next table – not much gain in terms of an optimization, if you think about it. This comes from a quirk in the data where EVERY record uses the same "**Pra_Id**" field. The net result is that we have 12 million rows passing the first filter, and therefore we have 12M lookups into the second table in the query to go with it.

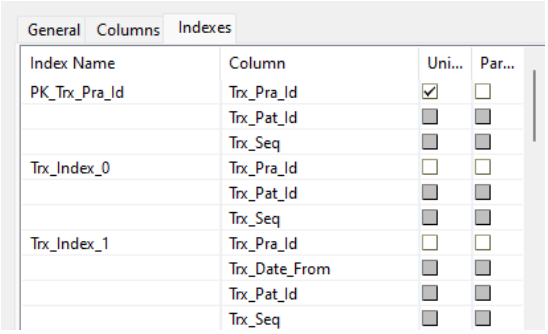
The second table in this query plan is the **Trx_Master** table. We are able to double-click on the table in the QPV, and it shows us the filtering that has been requested:



QNode Detail

```
Trx_Master [Trx_Master]:

Actual rows: 14224269,
Estimated rows: 4535

Range Info:
Index Col: Trx_Pra_Id
Seek Key [Payment_Master->Eob_Pra_Id] GE
End code Range 0.       HiValue: [Payment_Master-
>Eob_Pra_Id] GT
Index Col: Trx_Pat_Id
Seek Key [Payment_Master->Eob_Pat_Id] GE
End code Range 1.       HiValue: [Payment_Master-
>Eob_Pat_Id] GT
Index Col: Trx_Seq
Seek Key [Payment_Master->Eob_Trx_Seq] GT
End code Range 2.       HiValue: [Payment_Master-
>Eob_Trx_Seq] GT
```

As the index selected was **Trx_Index_0**, we can confirm that index definition:



General Columns Indexes

Index Name	Column	Uni...	Par...
PK_Trx_Pra_Id	Trx_Pra_Id	☑	☐
	Trx_Pat_Id	■	■
	Trx_Seq	■	■
Trx_Index_0	Trx_Pra_Id	☐	☐
	Trx_Pat_Id	■	■
	Trx_Seq	■	■
Trx_Index_1	Trx_Pra_Id	☐	☐
	Trx_Date_From	■	■
	Trx_Pat_Id	■	■
	Trx_Seq	■	■

Because we have the **Trx_Pra_Id** as an EQUAL lookup, and **Trx_Pat_Id** as an EQUAL lookup, and **Trx_Seq** is also an EQUAL lookup, this lookup degenerates into a single record request, which is nice and efficient. So, although there are 14 million rows in this table,

We then pass that data up through the filter, which includes the following:

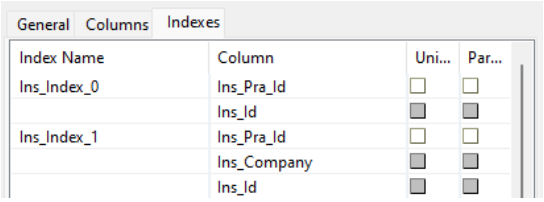
```
Subtype: Normal

Filter Info:

Trx_Master . Trx_Pra_Id = 'XXX' AND Trx_Master . Trx_Date_From >=
'2021-01-01 AND Trx_Master . Trx_Date_From <= '2021-01-31 AND
Trx_Master . Trx_Sec_Flag = 'Y' ) AND Trx_Master .
Trx_Sec_Date_Xmit <= '2021-11-01 ) AND Trx_Master .
Trx_Charge_Amount > 0 )
```

Here, we verify that every passing record ALSO matches the date range requested, along with the other restrictions listed in the query.

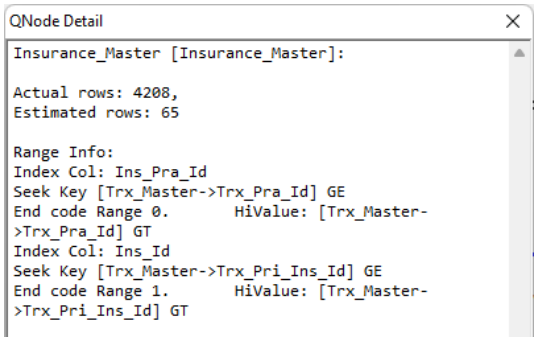
Once we have the data from the **Trx_Master** table, we can complete the first join, which verifies that the following filter criteria are met:

```
Subtype: Normal

Filter Info:

Payment_Master . Eob_Pra_Id = Trx_Master . Trx_Pra_Id AND
Payment_Master . Eob_Pat_Id = Trx_Master . Trx_Pat_Id AND
Payment_Master . Eob_Trx_Seq = Trx_Master . Trx_Seq
```

Finally, we bring the results of this combined table 1 and 2 to the third table, **Insurance_Master**, which is being accessed through the index called **Ins_Index_0**:



Looking at the query plan, we see that it is using the data from the **Trx_Master** table to select JUST the record we want.



The filter node above this table only adds one more restriction:

```
Subtype: Normal
```

```
Filter Info:

Insurance_Master . Ins_Pra_Id = 'XXX' )
```

Even though there are only 4208 rows in this particular file, remember that we have to perform a lookup into this file for EVERY combination that passes the first two filters, so we are likely going to be accessing the same records over and over again.

This brings us to the final filter after the third table is joined in:

```
Subtype: Normal

Filter Info:

Trx_Master . Trx_Pra_Id = Insurance_Master . Ins_Pra_Id AND
Trx_Master . Trx_Pri_Ins_Id = Insurance_Master . Ins_Id
```

With the joining and filtering done, we can now present data fields back to the caller.

Conclusion

The Query Plan Viewer exposes the underlying decisions that the SRDE made to run any given query. With some understanding of the choices the SQL optimizer is making, along with an accurate measurement of the SQL query workload (which we covered in Part 1 of this series), we are now poised to take the next leap towards query optimization. By changing the query ever-so-slightly, either through table order, restrictions, or hinting, we may be able to provide additional information to the optimizer so that it makes better choices, and thus produces results much more quickly. So, after you've played around in the QPV for a while, it's time to look up Part 3 of this series!

If you still can't get it to work, contact Goldstar Software and let us help!