SQL Query Optimization Part 3
Optimizing SQL Queries
In Actian Zen

A White Paper From

GOLDSTAR
SOFTWARE
www.GoldstarSoftware.com

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

SQL Query Optimization Part 3:
Optimizing SQL Queries in Actian Zen

Last Updated: 02/04/2022

Note #1: *This paper is part 3 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 Workload of a SQL 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.

You'll then want to see the second paper in the series, titled ***Analyzing SQL Queries with the Query Plan Viewer***, as this paper explains how to capture the query plan and gain an understanding of exactly how the engine saw the query and what it wanted to do to find the data that was requested.

This paper covers the third part of the process, actually changing the SQL query based on the query plan and existing index information, and then updating the query so that the engine can be much more efficient and find the correct data with less effort.

Understanding MicroKernel Access

The Actian SQL Relational Database Engine (SRDE) does not directly access your data, but rather it relies on a transactional layer running underneath, called the MicroKernel Database Engine (MKDE) to access the binary data files and locate the needed data. The MKDE itself has a well-defined API layer that offers access to the data to the SRDE in the same way as it offers access to any other application, and this interface is known as the Btrieve API. (Newer versions also support the Btrieve 2 API, which has similar capabilities.) The Btrieve API offers a very fast and low-overhead method of accessing records.

The SRDE locates records by using the following MKDE functions:

- **Get Operations**: Allow the access of records based on an indexed lookup on a Btrieve "key", equivalent to a SQL index. Running an entire table by a specific key (i.e. in sorted order, ascending or descending) is possible using GetFirst, GetNext, GetPrevious, and GetLast. Additionally, several indexed Get operations are supported which provide indexed lookup against a key, including GetEqual, GetGreater, GetGreaterOrEqual, GetLess, GetLessOrEqual. These functions locate the first (or next) matching record on a given index value, and are crucial for effecting JOINs and WHERE clauses.
- **Step Operations**: When no index is available, the MKDE offers Step operations, including StepFirst, StepNext, StepLast, and StepPrevious. These operations can be used to read through an entire record set without using an index (i.e. in physical order) and are often more efficient that Get operations since the index pages are not accessed.
- **Extended Operations**: The MKDE offers a series of Extended operations that take the normal GetNext/StepNext and GetPrevious/StepPrevious operations and allow the caller to apply specific filtering criteria to rapidly scan through large numbers of records very rapidly and only return the relevant fields required for the query. Extended operations are heavily used by the SRDE in the normal course of operations to optimize the location of records matching a set of criteria. Additionally, the GetNextExtended and GetPreviousExtended operations support the retrieval of data through a specific key, which can provide data in the requested order at the same time. (Newer engines also support a DeleteExtended operation for efficiency in deleting records, as well as LIKE support in the filtering criteria.)

When the SRDE is able to leverage the MKDE to handle its filtering, this is known as a push-down filter (because it is pushed down to the lower MKDE level), and significant speed advantages are often seen as a result.

Optimizing WHERE Clauses

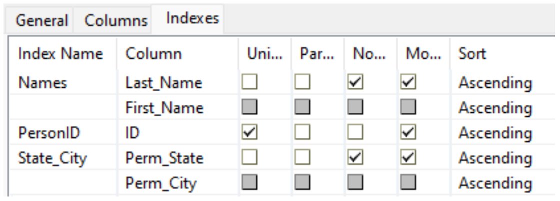
When you use a WHERE clause in your SQL query, the most obvious solution would be for the SQL engine to read every record from the MKDE source data set, decode the field you are filtering on, and look for matches. In fact, some queries have no choice but to do this, especially when you are trying to find a substring within a larger string, as in the example “`WHERE SUBSTRING(field,5,3) = ‘XXX’`”. Of course, this is also the least efficient method, as it must access every record and parse the returning data.

The next best choice would be to establish a position (either physically or on a key path) and leverage a push-down filter to build a series of Extended Operations calls to locate the needed data. Although each record still needs to be examined, if we can allow the MKDE to handle the details, we will see far less overhead, and thus a faster response time. A good example of this is “`WHERE LEFT(field,3) = ‘XXX’`”, because we can identify the exact starting byte of the field and therefore tell the MKDE where to filter for the ”XXX” string.

The best choice, by far, is to leverage an indexed lookup instead. Since all indexes are evaluated left to right, we can only do this if we are able to filter on the starting field(s) of a given index. For example, we have a clause like “`WHERE field = ‘XXX’`” and if “field” is indexed, we can jump directly to the first possible matching record with a GetEqual operation.

In reality, it is more common to see multi-field indexes, also known as “segmented keys” in a file. These keys would ONLY support a GetEqual operation if we knew the expected value for EVERY field in the index. Because of this, the engine doesn’t use a GetEqual, but rather builds a key buffer to search from the left and then uses a GetGreaterOrEqual operation to find the first matching record. Then, it can continue to issue GetNext operations for as long as the matching records are returned. When the first record is returned that does NOT match, the search is terminated.

Let’s look at a few more specific examples from the **DEMODATA.Person** table, which has the following index definitions:



Index Name	Column	Uni...	Par...	No...	Mo...	Sort
Names	Last_Name	☐	☐	☑	☑	Ascending
	First_Name	▨	▨	▨	▨	Ascending
PersonID	ID	☑	☐	☐	☑	Ascending
State_City	Perm_State	☐	☐	☑	☑	Ascending
	Perm_City	▨	▨	▨	▨	Ascending

Here’s the first query:

```
SELECT * FROM Person WHERE ID = 104101361;
```

Because we have an index start starts with the **ID** field, this query would generate an MKDE GetGreaterOrEqual call on the **PersonID** Index with the value of `104101361`. It would then issue a GetNextExtended to see if there are any more matches (there will be none), so this request will read just 2 records in total.

Here’s one on the **Last_Name** field:

```
SELECT * FROM Person WHERE Last_Name = ‘Ogelvie’;
```

Again, we have an index start starts with the **Last_Name** field, this query would generate an MKDE GetGreaterOrEqual call on the **Names** Index with the value of ‘Ogelvie” for the **Last_Name** field. Note that the MKDE expects a value for the **First_Name** field, which is part of the index, so the SRDE places the *smallest possible value* in the key buffer for those bytes. In this case, this is the ASCII 0 value, so a string of NULLs is added, and we then can match any records with the last name of Ogelvie.

What about the **First_Name** field?

```
SELECT * FROM Person WHERE First_Name = 'James';
```

In this case, we don't have any indexes that start with the **First_Name** field at all. Because there is no index available, we must perform a complete table-scan of the record set. Since no index will help here, we'll have to do a StepFirst operation followed by a StepNextExtended looking for any records where the bytes of the **First_Name** field are equal to 'James'. The MKDE has to look at every record in the data set, so this will be notably slower (and require more work) than the above examples.

If you examine the operations, you'll also see that some optimizations can also be used to support ">", ">=", "<", and "<=" criteria, but they may be only beneficial in certain circumstances. For example, let's say that our database has records from 2001 through 2022, and we are looking for records from 2022, so we build a clause of "`WHERE RecordDate > { '2022-01-01'}`". If we use a GetGreaterOrEqual request on the **RecordDate** index, we can exclude 95% of the data rather quickly. However, is the restriction is looking for records newer than 2002 instead, then this only eliminates 5% of the record set, and does not become a very good limiting factor for optimization.

By extension, any range (i.e. "`RecordDate BETWEEN {d '2020-01-01'} AND {d '2020-12-31'}`" can also be optimized by simply setting the starting point and looking for the ending point. The opposite (i.e. excluding a range) is not true, however, as we cannot optimize out "`RecordDate < {d '2020-01-01'} AND RecordDate > {d '2020-12-31'}`" with an index at all. Instead, we end up having to read every record from the file and filtering accordingly.

As SQL statements get more complicated, the ability to optimize WHERE clauses can get more complicated, too. Obviously, only ONE index can be used at a given time to access a table, so the optimizer must make the best possible choices. For example, what about an "OR" clause which is looking for two possible values? The engine may decide to search for one value and then search for the second separately, or it may simply decide that the effort is too great to keep track of multiple lookups, and it is simpler to read ALL records and only return those records that match. Clearly, the query plan (and the response time) will vary between these choices.

What happens if there are two WHERE clauses, and each has an index? The engine will attempt to estimate the number of matching records on each criterion (which may be based on limited information) and then pick one. Sometimes, it chooses wisely – but other times, not so much. This is when the use of *hints* comes into play, where you can ask the engine to only consider specific indexes for your query in the hopes to direct it to a more efficient solution.

Optimizing ORDER BY Clauses

In most cases, a WHERE clause is going to trump any ORDER BY, because the ORDER BY clause is always handled last in any query. For a query where the ORDER BY is not optimized, the initial data set is first constructed and retrieved by the engine, the fields are inserted into a temporary table built by the system for the express purpose of the sorting, and then an index is added to the data as requested. The data is then retrieved from the temporary table in that order and presented to the user. When the query is finished, the temporary table is destroyed.

It is possible to optimize an ORDER BY in a few cases, and therefore eliminate the need for any temp table. The first such case is when there is no WHERE clause, or when the ORDER BY and WHERE clauses are satisfied by the same index, as in "`WHERE RecordDate > {d '2022-01-01'} ORDER BY RecordDate`". In this example, the index on the **RecordDate** field is already satisfying the WHERE clause, so the engine is already returning the data in order and it takes no extra work to get the data in the right order for the ORDER BY clause. Another case that can be optimized is something like "`SELECT * FROM Person WHERE First_Name = 'Bill' ORDER BY ID`". In this example, the WHERE clause cannot be optimized, and the engine would normally pick a StepNextExtended operation loop to scan for records. It takes no extra time to switch this to a GetNextExtended loop using the index on the **PersonID** key, and thus effect the query order at no additional cost (and thus no temp table).

Optimizing Multi-Table (Join) Queries

When multiple tables are referenced in a SQL query, the database engine must effect the JOIN operation to link the tables together. Often, this is done through a primary key, but this is not a hard requirement – you can actually join on any fields you wish. Of course, if there is an index, the JOIN will be optimized down to a single set of lookups, but if there is no index available, then a linear table-scan operation may result in every record being read for each item in the JOIN. This makes optimizing the JOIN of paramount importance!

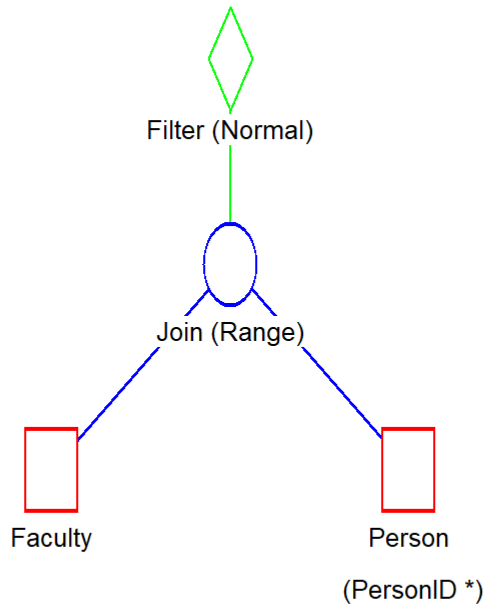
When we specify multiple tables, there are two ways to specify the JOIN, and we have named these for the way the system optimizes each:

1. Cost-based Optimizer: When a query contains multiple table names on the FROM line and you rely on the WHERE clauses to effect the JOIN, the database engine uses its own logic to make guesses as to which table may be the best to start with. In some cases, it makes good choices; in other cases, it may choose poorly.
2. Syntax-Based Optimizer: When a query contains multiple tables connected with the JOIN syntax and an ON clause to effect the JOIN, the database engine follows your lead and inherently uses the tables in the order in which you provide them. This allows you to specify the correct starting table for your environment and expect the engine to follow your lead. [N.b. There is a hidden engine registry setting that allows the engine to override your choices and optimize inner joins using its own algorithms, but this function is disabled by default.]

Let's look at a few simple examples of each, starting with the cost-based optimizer:

```
SELECT Last_Name, First_Name, Salary
FROM Faculty, Person
WHERE Person.ID = Faculty.ID;
```

Here, we see a query that lists the **Faculty** table joining to the **Person** table on the respective **ID** columns. Both of the **ID** fields are considered primary keys, so each has a defined unique index. When we examine the query plan, we see the following:



Which shows that we are doing a linear scan on all 96 records in the **Faculty** table, and then doing a lookup on the **Person** table based on the **ID** column, exactly as expected. Watching in the Monitor, we can confirm that the workload is approximately 305 pages accessed. (If you haven't gone through Part 1 and Part 2 of this series yet, you may wish to review this before continuing, so that the rest of the analysis makes sense.)

So, what happens when we change the query to the following?

```
SELECT Last_Name, First_Name, Salary
FROM Person, Faculty
WHERE Person.ID = Faculty.ID;
```

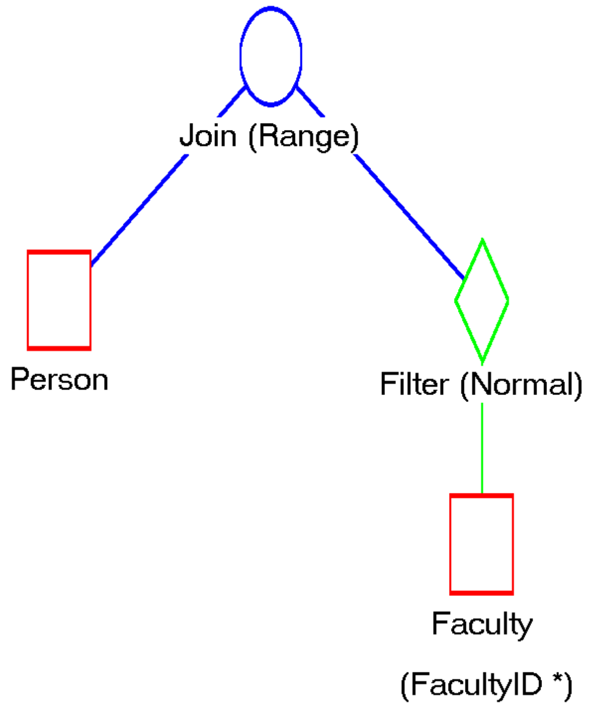
It should not be a surprise now that we get the EXACT SAME results! The query plan for this query is identical to the picture above, confirming that the cost-based optimizer has selected the **Faculty** table as our first table yet again, regardless of the order in which they were listed. And, checking the Monitor, the workload is the same 305 pages.

Next, let's rework this query to use the alternative syntax, invoking the syntax-based optimizer instead:

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

If you run this query, you will get exactly the same result set as the above queries, because they essentially mean the same thing. However, when we run this query, a very different process is executed internally within the engine. Instead of scanning 305 pages to complete the request, the engine is now looking at 4636 pages to complete the query –

the workload has gone up by a factor of 15x! Examining the query plan, we see why this change has occurred:

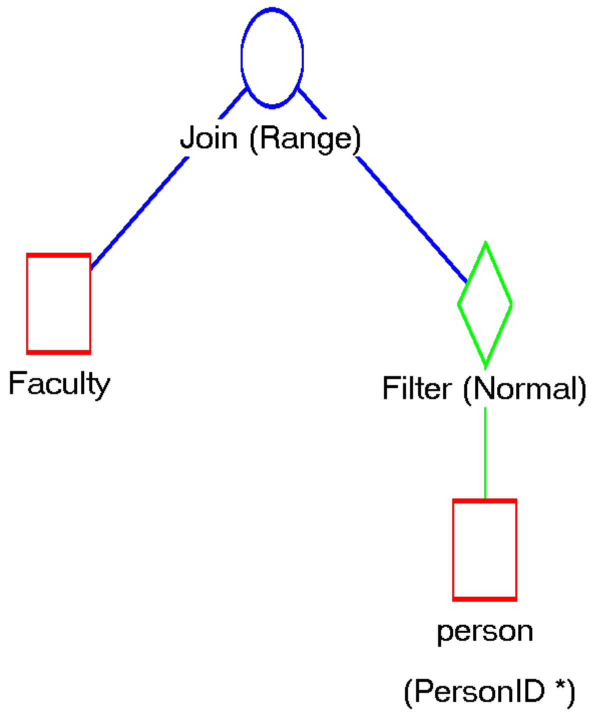


In this form of the query, the **Person** table is now the first one accessed, mainly because we asked the engine to start there. This forces a table-scan of the 1500 records in **Person**, and for each one, perform a lookup into the **Faculty** table. This results in substantially more effort to find the same data set.

Fixing this is easy. Since we know that the **Faculty** table is the best place to start, then we can use the order of the tables in the JOIN clause to tell the optimizer where to start:

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

When this query runs, it generates us a workload of 305 page accesses and a query plan that more closely matches the previous one, with the **Faculty** table on the left:



By manually selecting the “preferred” table order, we guide the engine to do exactly what we want, and we get an optimized query, reducing workload as a result.

Obviously, a small query like this won’t have a huge difference in a typical environment. When queries take less than a few milliseconds to execute, you might not think about optimizing it. However, what if this query is part of a screen refresh function that is executed every 10 seconds, by each of 100 concurrent users? The second form of the query would equate to 2.7 million page accesses per minute, while the second form would require only 183 thousand page accesses, and substantially less CPU overhead.

Based on all of this, why would you possibly want to ever use the syntax-based optimizer, if the cost-based optimizer can fix up your query for you? The answer is both simple and complex at the same time. In short, though, you may have more knowledge about your data set than the cost-based optimizer, and you may be better able to manually optimize a given query with that additional information.

A More Complex Example

We now take the same long-running query that we examined in parts 1 and 2 of this series and apply what we’ve learned. Let’s start by reviewing the query itself:

```
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'))
```

The first thing you'll notice is that the simple table list is used, thus allowing the cost-based optimizer to be used. However, if you look back at the query plan (in Part 2), you'll notice that the first table accessed was the **Payment_Master** table. Looking at the query, you can see that there is no WHERE clause that can limit the data here, and a complete table-scan is going to be generated! This is one of the main reasons why the query takes 99 million page accesses to complete.

If you look carefully at the WHERE clauses, you'll see that the major limiting factor (i.e. the way to reduce the data set) is likely to be the date range on the **Trx_Master** table, or perhaps one of the other 4 clauses from this same table. Remember, though, that we cannot just put the **Trx_Master** table first in the query, because the cost-based optimizer may re-order the tables anyway. Because of this, we want to eschew the cost-based optimizer in favor of the syntax-based optimizer, and therefore force the engine to start with the **Trx_Master** table instead.

At the same time, we are going to simplify the readability of the query by pulling the JOIN fields out of the WHERE clause and make them part of the JOIN criteria itself. Let's do this one step at a time by starting with the reformatting of the query to make it easier to read and removing the extraneous parentheses:

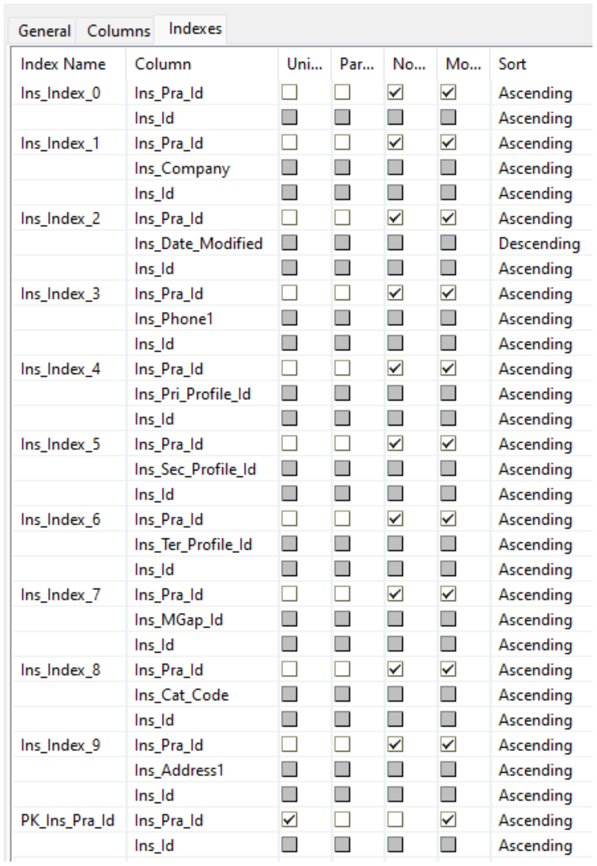
```
SELECT <field_list>
FROM Insurance_Master,
     Payment_Master,
     Trx_Master
WHERE Trx_Master.Trx_Pra_Id = Insurance_Master.Ins_Pra_Id
AND Trx_Master.Trx_Pri_Ins_Id = Insurance_Master.Ins_Id
AND Payment_Master.Eob_Pra_Id = Trx_Master.Trx_Pra_Id
AND Payment_Master.Eob_Pat_Id = Trx_Master.Trx_Pat_Id
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'
```

Next, we rearrange the tables to start with **Trx_Master** and start building our JOIN syntax:

```
SELECT <field_list>
FROM Trx_Master
  INNER JOIN Insurance_Master ON
    ()
  INNER JOIN Payment_Master ON
```

```
     ()
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'
```

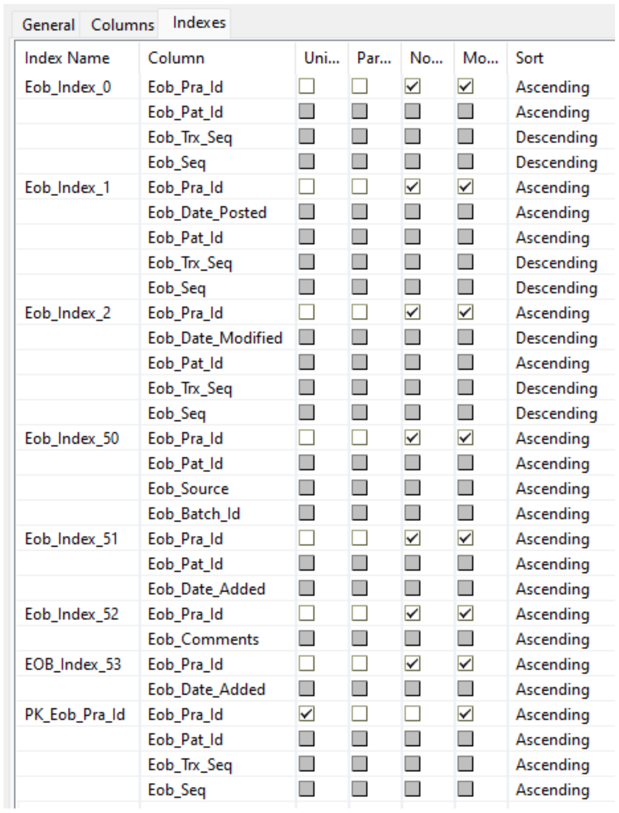
As discussed above, the most effective way to JOIN a table is through an index that limits us to a single matching record, or at least the smallest possible subset. To find the right index, we look at the index definitions for the **Insurance_Master** table, and we see:



Index Name	Column	Uni...	Par...	No...	Mo...	Sort
Ins_Index_0	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_1	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Company	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_2	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Date_Modified	■	■	■	■	Descending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_3	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Phone1	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_4	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Pri_Profile_Id	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_5	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Sec_Profile_Id	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_6	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Ter_Profile_Id	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_7	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_MGap_Id	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_8	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Cat_Code	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
Ins_Index_9	Ins_Pra_Id	☐	☐	✓	✓	Ascending
	Ins_Address1	■	■	■	■	Ascending
	Ins_Id	■	■	■	■	Ascending
PK_Ins_Pra_Id	Ins_Pra_Id	✓	☐	☐	✓	Ascending
	Ins_Id	■	■	■	■	Ascending

Our primary key for this table is **Ins_Index_0**, which uses a combination of the two fields **Ins_Pra_Id** and **Ins_Id**. These are the exact same two fields that are currently used in the WHERE clause, so we can leverage this unique key to perform a direct lookup.

What about the **Payment_Master** table? We find something similar in that index list:



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
Eob_Index_2	Eob_Pra_Id	☐	☐	✓	✓	Ascending
	Eob_Date_Modified	■	■	■	■	Descending
	Eob_Pat_Id	■	■	■	■	Ascending
	Eob_Trx_Seq	■	■	■	■	Descending
	Eob_Seq	■	■	■	■	Descending
Eob_Index_50	Eob_Pra_Id	☐	☐	✓	✓	Ascending
	Eob_Pat_Id	■	■	■	■	Ascending
	Eob_Source	■	■	■	■	Ascending
	Eob_Batch_Id	■	■	■	■	Ascending
Eob_Index_51	Eob_Pra_Id	☐	☐	✓	✓	Ascending
	Eob_Pat_Id	■	■	■	■	Ascending
	Eob_Date_Added	■	■	■	■	Ascending
Eob_Index_52	Eob_Pra_Id	☐	☐	✓	✓	Ascending
	Eob_Comments	■	■	■	■	Ascending
EOB_Index_53	Eob_Pra_Id	☐	☐	✓	✓	Ascending
	Eob_Date_Added	■	■	■	■	Ascending
PK_Eob_Pra_Id	Eob_Pra_Id	✓	☐	☐	✓	Ascending
	Eob_Pat_Id	■	■	■	■	Ascending
	Eob_Trx_Seq	■	■	■	■	Ascending
	Eob_Seq	■	■	■	■	Ascending

Again, we find a unique index called **Eob_Index_0** that includes the fields **Eob_Pra_Id**, **Eob_Pat_Id**, **Eob_Trx_Seq**, and **Eob_Seq**. Note that we are not going to get a perfect lookup here, but we will be able to use the first three of these four fields, which should limit our resulting data set considerably.

We can now remove the JOIN restrictions from the WHERE clause and move them to the JOIN section, where they can do the most good. Note that I also swapped the order of two JOIN criteria for **Insurance_Master** to provide an easier-to-read, left-to-right order:

```
SELECT <field_list>
FROM Trx_Master
  INNER JOIN Insurance_Master ON
    (   Insurance_Master.Ins_Pra_Id = 'XXX'
    AND Insurance_Master.Ins_Pra_Id = Trx_Master.Trx_Pra_Id
    AND Insurance_Master.Ins_Id  = Trx_Master.Trx_Pri_Ins_Id )
  INNER JOIN Payment_Master ON
    (   Payment_Master.Eob_Pra_Id = 'XXX'
    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 )
WHERE
    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
```

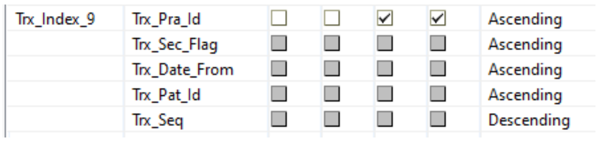
Finally, you'll see that the first two clauses in each JOIN are redundant. Because of that, we can simplify the query by removing these extra lines:

```
SELECT <field_list>
FROM Trx_Master
  INNER JOIN Insurance_Master ON
    (   Insurance_Master.Ins_Pra_Id = 'XXX'
    AND Insurance_Master.Ins_Id  = Trx_Master.Trx_Pri_Ins_Id )
  INNER JOIN Payment_Master ON
    (   Payment_Master.Eob_Pra_Id = 'XXX'
    AND Payment_Master.Eob_Pat_Id = Trx_Master.Trx_Pat_Id
    AND Payment_Master.Eob_Trx_Seq = Trx_Master.Trx_Seq )
WHERE
    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
```

Now that we have dealt with the more-important JOIN criteria, the WHERE criteria that are left certainly stand out quite clearly. We can now check to see if the **Trx_Master** table accesses can be optimized by looking at those index definitions, along with the optimizable criteria.

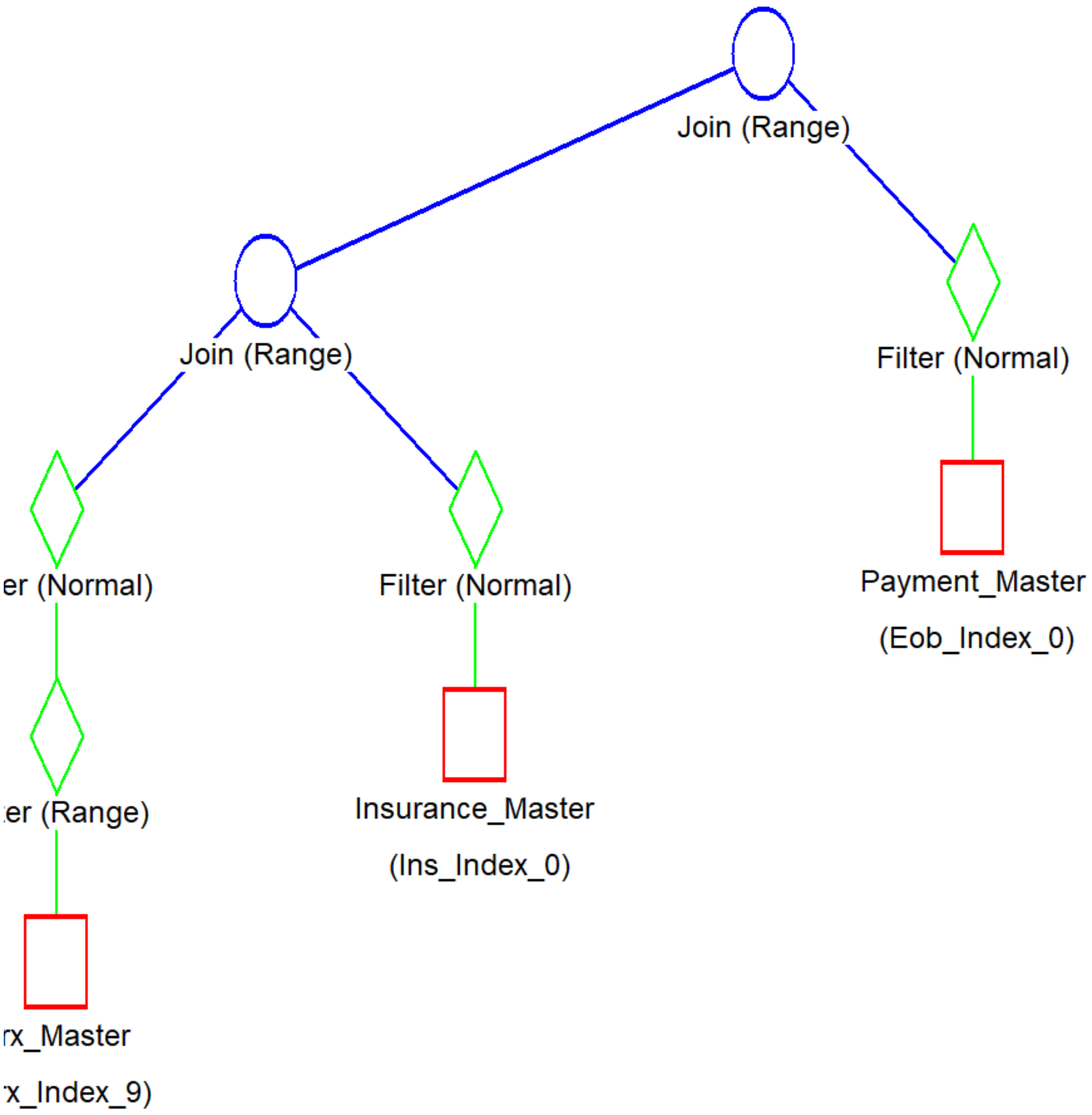
Recall that any "=" criteria can be easily handled by an MKDE lookup and thus optimized. Optimizing out a pure "<" or ">" may or may not be possible, as this will depend more on the data, which we don't see here. However, optimizing out the range IS possible with an index – especially when we look at the criteria and realize that this is restricting the data set to a one-month range only – so this likely will provide a HUGE benefit, and we want to include that!

There are 22 indexes in this table definition, so we are not providing the entire list here. However, we can see that one of the indexes, **Trx_Index_9**, provides a combination of **Trx_Pra_Id**, **Trx_Sec_Flag**, **Trx_Date_From**, **Trx_Pat_Id**, and **Trx_Seq**.



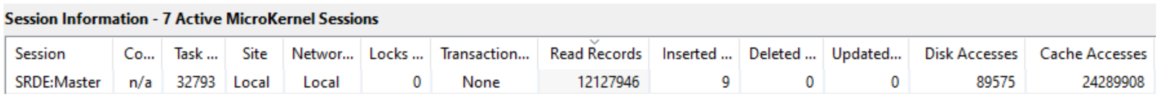
Trx_Index_9	Trx_Pra_Id	☐	☐	☑	☑	Ascending
	Trx_Sec_Flag	▨	▨	▨	▨	Ascending
	Trx_Date_From	▨	▨	▨	▨	Ascending
	Trx_Pat_Id	▨	▨	▨	▨	Ascending
	Trx_Seq	▨	▨	▨	▨	Descending

This index will handle my two "=" criteria AND the limiting date range, so that one is ideal for our purpose! Before we provide the hint, though, let's see if the engine will pick this index on its own. We run the query and generate a new query plan:



Indeed, we see that the engine did pick up **Trx_Index_9** all on its own, so we don't need to add the extra hint in this case. The query plan also confirms that **Ins_Index_0** and **Eob_Index_0** were picked up for the JOIN criteria as well, exactly as we expected.

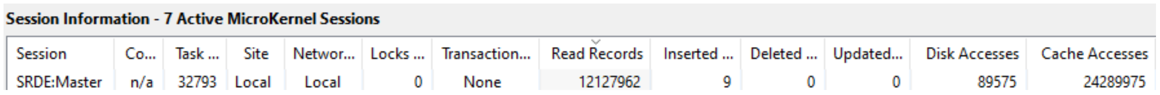
The other thing we notice is that the query doesn't take hours to run, but rather finishes almost before we can lift our finger from the mouse button. Of course, the real proof is what we did to the workload. We first grab a screenshot of the Monitor showing the database access counts for my SQL session:



Session Information - 7 Active MicroKernel Sessions

Session	Co...	Task ...	Site	Networ...	Locks ...	Transaction...	Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
SRDE:Master	n/a	32793	Local	Local	0	None	12127946	9	0	0	89575	24289908

Then, we run the query and grab the new number:


Session Information - 7 Active MicroKernel Sessions

Session	Co...	Task ...	Site	Networ...	Locks ...	Transaction...	Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
SRDE:Master	n/a	32793	Local	Local	0	None	12127962	9	0	0	89575	24289975

And with some simple subtraction,we can now compute the workload of this query – 24289975 – 24289908 = 67 pages! Now THAT is a successful optimization….

Conclusion

The MicroKernel Database Engine (MKDE) offers very fast access to the Actian Zen database files through a variety of simple functions. The SQL Relational Database Engine (SRDE) leverages the speed of the MKDE to access the data records navigationally, and uses that engine to execute SQL queries and return the requested data. Understanding how these two components work together is important for ensuring that your queries are properly formatted for the best possible performance overall.

Instead of just throwing your queries at the engine and dealing with the results, it is possible to spend a small bit of time understanding the workload and the query plan, and then using that understanding to optimize the query. By reducing the workload, your queries will finish more quickly and with less CPU load on the server, which means that user productivity will increase for the entire system.

Hopefully, this series of papers has provided you with the knowledge to optimize your own queries.

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