Unleashing the Power of SQL EXPLAIN

Take Query Optimization to the Next Level

By Jeff Ash *BASIS Advantage*

In the dynamic world of database management, optimizing queries is a constant pursuit. One indispensable tool in this quest is the SQL EXPLAIN statement (available in BBj® version 23.00+), a feature that provides a detailed roadmap of how BBj's SQL engine executes queries. In this article, we'll explore the nuances of the EXPLAIN statement and unravel its potential to unveil bottlenecks, enhance query efficiency, and empower developers to make informed decisions.

Understanding the Basics

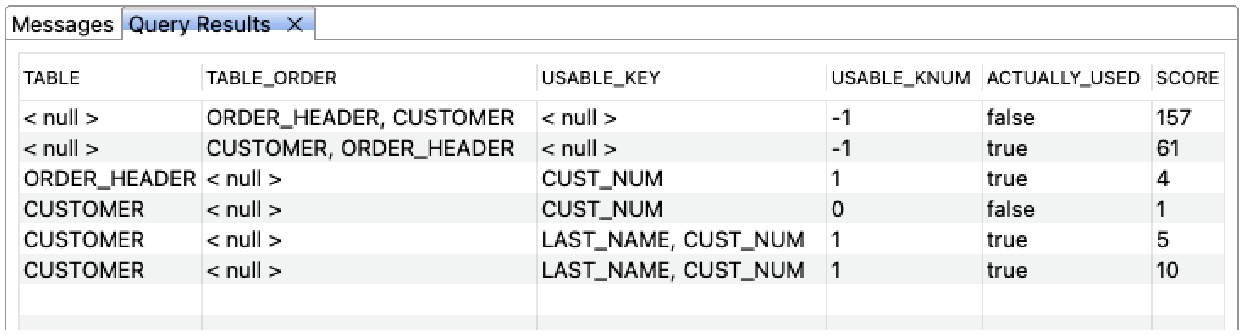
At its core, the EXPLAIN statement sheds light on the execution plan of a query. By delving into the output of this statement, developers can identify areas where improvements can be made, such as adding or modifying indexes or optimizing queries for better performance.

Let’s take a look at a practical example:

```
EXPLAIN SELECT
   cust_num, order_num, order_date, first_name, last_name
FROM order_header oh INNER JOIN customer c
   ON oh.cust_num = c.cust_num
WHERE last_name = 'Baldrake'
```

Breaking Down the Results

The output of the EXPLAIN statement resembles a standard ResultSet, with each row offering valuable insights into the execution plan. Let's decipher the critical column components found in **Figure 1**:



Messages | Query Results

TABLE	TABLE_ORDER	USABLE_KEY	USABLE_KNUM	ACTUALLY_USED	SCORE
< null >	ORDER_HEADER, CUSTOMER	< null >	-1	false	157
< null >	CUSTOMER, ORDER_HEADER	< null >	-1	true	61
ORDER_HEADER	< null >	CUST_NUM	1	true	4
CUSTOMER	< null >	CUST_NUM	0	false	1
CUSTOMER	< null >	LAST_NAME, CUST_NUM	1	true	5
CUSTOMER	< null >	LAST_NAME, CUST_NUM	1	true	10

Figure 1. *Results of EXPLAIN SELECT from the Practical Example*

TABLE

Each row has data referring to a single table or table order. Rows 3-6 reference a single table, shown in the TABLE column. Table order does not apply to these rows, so they have `<null>` in the TABLE_ORDER column.

TABLE_ORDER

Rows 1 and 2 refer to table order, so they have non-null data in the TABLE_ORDER column and null data in the TABLE column. Rows referring to table order provide information about the automatic reordering of tables that the SQL engine does during optimization (when possible). Note that when there is a non-null value in this column, the two relevant columns are ACTUALLY_USED and SCORE. The other columns in the result data do not apply.

USABLE_KEY

The SQL engine will consider using the index/key in the USABLE_KEY column during optimization. Each key in a BBx data file, such as an MKEYED, XKEYED, or VKEYED file, corresponds to an SQL index. Note that the value in USABLE_KNUM represents the key number on the file.

The order plays an important role during optimization for keys with multiple columns. For example, let's say you have an index with the segments defined as LAST_NAME, FIRST_NAME and you use a WHERE clause that refers to a singular segment. The SQL engine can only use the index/key for optimization if the index/key's first segment (LAST_NAME) is the referenced segment in the WHERE clause.

So, when you see an entry in this column, it indicates that this key meets the criteria necessary to be considered for optimization. However, the SCORE will determine which one the SQL engine ultimately chooses.

USABLE_KNUM

This column is simply the key number on the underlying data file for the table that corresponds to the segments shown in the USABLE_KEY column. See USABLE_KEY for an explanation of how the SQL engine uses this information.

ACTUALLY_USED

This column contains true or false, indicating whether or not the SQL engine determined to use the key number shown in USABLE_KNUM for optimization. When optimizing queries, the SQL engine must choose a single key for iteration. The primary role of the optimization strategy is to determine which key (index in SQL terminology) will result in the fewest file-read operations. This determination is made based on the value in the SCORE column. The lower the score, the better.

SCORE

The score is a rough estimate of the number of file-read operations required for processing the query.

Decoding the Score in Detail

The SCORE column is pivotal in determining the optimization strategy. A lower score indicates fewer file operations, translating to better performance. To illustrate, let's consider our earlier example:

```
Row 1: Score - 157 (ORDER_HEADER first, CUSTOMER second)
Row 2: Score - 61 (CUSTOMER first, ORDER_HEADER second)
```

Regardless of which table order the optimizer chooses, the join will be fast.

However, the WHERE clause provides additional information, indicating that we're only interested in customers with the LAST_NAME of Baldrake, with no limiting criteria on the table. So if the ORDER_HEADER table is first, it would need to look over a record in the ORDER_HEADER table, look up its corresponding record in the CUSTOMER table, check to see if the LAST_NAME matches, then iterate this for every record in the ORDER_HEADER table.

The sample database is small, so we would not see any performance differences. However, imagine a real-world scenario where the ORDER_HEADER table contains 1,000,000 records. That would involve a minimum of 2,000,000 reads to return the results (i.e., one read from ORDER_HEADER, then one read from CUSTOMER).

With the simple change of placing the CUSTOMER table on the left, you will see that the optimizer's change can dramatically increase performance. Due to the indexing of the CUSTOMER table, the SQL engine can limit the number of records it reads from the CUSTOMER table to only those that match the WHERE clause.

Returning to our real-world scenario, suppose only 1,000 records in the customer table match a LAST_NAME of Baldrake. Placing the CUSTOMER table first means that the SQL engine can iterate using KNUM 1 (LAST_NAME and CUST_NUM combined) only looking at 1,000 records in the table on the left, thus only looking up matches for 1,000 records in the right table. This results in a total of 2,000 read operations instead of 2,000,000.

Conclusion

While the SQL EXPLAIN statement doesn't provide explicit recommendations, it equips developers and administrators with invaluable insights into query optimization. By leveraging this information, one can identify the reasons behind sluggish queries and explore avenues for performance enhancement.

Consideration of additional filtering criteria and judicious indexing becomes the key to unlocking the full potential of your database queries. In the ever-evolving landscape of database optimization, SQL EXPLAIN emerges as a powerful ally, empowering you to navigate the complexities of query execution with confidence.

To begin exploring the power of the SQL EXPLAIN statement, install the latest release of BBj®.