Understanding the ClustrixDB Explain Output

The EXPLAIN statement is used to reveal how the ClustrixDB Query Optimizer also known as Sierra, will execute INSERT, SELECT, UPDATE, and DELETE statements. The output from EXPLAIN presents three columns:

- Operation - An internal operator to accomplish a task
- Est. Cost - The estimated cost is a measurement proportional to how much wall-clock time will be required to perform the operation
- Est. Rows - The estimated number of rows the Sierra believes will be output by the operator

These describe the execution plan as a physical plan to implement the declarative SQL statement. In most instances, each row in the output of EXPLAIN represents a single operation collecting input or working with the input which is indented one level in following rows. In other words, most explain statements can be read as most indented is executed first and the total execution follows toward least indented.

The Data

To illustrate the EXPLAIN output, we'll go through an exercise defining and using a database to track customers and products that are sold to them. This example is for illustrative purposes and is not necessarily a good way to design your data for your application -- a good data model for your application will depend on your business needs and usage patterns. This data model focuses on relations rather than a complete data consistency model.

We will start with this basic data model. (Download the script used here.)

```
sql> CREATE TABLE customers (
         c_id INTEGER AUTO_INCREMENT
       , name VARCHAR(100)
       , address VARCHAR(100)
       , city VARCHAR(50)
       , state CHAR(2)
       , zip CHAR(10)
       , PRIMARY KEY c_pk (c_id)
      ) /*$ SLICES=3 */;
sql> CREATE TABLE products (
         p_id INTEGER AUTO_INCREMENT
       , name VARCHAR(100)
       , price DECIMAL(5,2)
       , PRIMARY KEY p_pk (p_id)
      ) /*$ SLICES=3 */;
sql> CREATE TABLE orders (
         o_id INTEGER AUTO_INCREMENT
       , c_id INTEGER
       , created_on DATETIME
       , PRIMARY KEY o_pk (o_id)
       , KEY c_fk (c_id)
       , CONSTRAINT FOREIGN KEY c_fk (c_id) REFERENCES customers (c_id)
      ) /*$ SLICES=3 */;
sql> CREATE TABLE order_items (
         oi_id INTEGER AUTO_INCREMENT
       , o_id INTEGER
       , p_id INTEGER
       , PRIMARY KEY oi_pk (oi_id)
       , KEY o_fk (o_id)
       , KEY p_fk (p_id)
       , CONSTRAINT FOREIGN KEY order_fk (o_id) REFERENCES orders (o_id)
       , CONSTRAINT FOREIGN KEY product_fk (p_id) REFERENCES products (p_id)
      )  /*$ SLICES=3 */;
```

After populating the database, there are 1,000 customers, 100 products, 4,000 orders and around 10,000 order_items.

Getting Our Toes Wet

Let's start with a simple query which gives us all the information about all of the customers.

```
sql> EXPLAIN SELECT * FROM customers;
+-------------------------------------------------------+-----------+-----------+
| Operation                                             | Est. Cost | Est. Rows |
+-------------------------------------------------------+-----------+-----------+
| stream_combine                                        |    712.70 |   1000.00 |
|   index_scan 1 := customers.__idx_customers__PRIMARY |    203.90 |    333.33 |
+-------------------------------------------------------+-----------+-----------+
```

In general, the explain output can be read from innermost indentation first and working your way to lower indentation eventually winding up at the first line in the output. Reading the explain output above, the first thing that happens is the operation index_scan on the index customers. __idx_customers__PRIMARY, the primary key index, and assigns the name "1" to the results of the read. In this case, that name is not used again. Note that the estimated rows are approximately 333 even though there are 1,000 customers in the relation. This is because each index_scan is reading a subset of the data distributed in the cluster which we call a slice. With this relation's schema where there are three slices, so three index_scan operations are running in parallel to gather the customer information. The output from the three index_scan operations is passed to the stream_combine operator which will, as the name implies, combines the streams into one so that it can be delivered to the client. The stream_combine operator works by simply copying the entire contents of the first input to arrive into the single stream output and continuing on until all streams have been combined.

Let's see what happens if we add a limit to our query.

```
sql>  EXPLAIN SELECT * FROM customers LIMIT 10;
+-----------------------------------------------------------+-----------+-----------+
| Operation                                                 | Est. Cost | Est. Rows |
+-----------------------------------------------------------+-----------+-----------+
| row_limit LIMIT := param(0)                               |    615.70 |     10.00 |
|   stream_combine                                          |    615.70 |     30.00 |
|     row_limit LIMIT := param(0)                           |    203.90 |     10.00 |
|       index_scan 1 := customers.__idx_customers__PRIMARY |    203.90 |    333.33 |
+-----------------------------------------------------------+-----------+-----------+
```

Here we have pretty much the same execution plan as before with the addition of the row_limit operator. A row_limit operator takes an incoming stream and closes the input stream once the limit (and offset) are satisfied. Since there are three parallel streams, Sierra "pushes down" a copy of the row_limit operator to each index_scan stream since there is no need to read more than 10 rows from each slice. After the streams are combined, we limit the output again so that the client gets the requested 10 rows.

Let's say we want to have an ordering for the results.

```
sql> EXPLAIN SELECT * FROM customers ORDER BY c_id;
+-------------------------------------------------------+-----------+-----------+
| Operation                                             | Est. Cost | Est. Rows |
+-------------------------------------------------------+-----------+-----------+
| stream_merge KEYS=[(1 . "c_id") ASC]                  |    816.70 |   1000.00 |
|   index_scan 1 := customers.__idx_customers__PRIMARY |    203.90 |    333.33 |
+-------------------------------------------------------+-----------+-----------+
```

This plan is similar to the unsorted version except that this time there is a stream_merge to combine the results rather than stream_combine. The stream_ merge operator works by pulling the next row from all incoming streams into the output stream based on the ordering provided. In this case, the order is on the c_id column ascending so stream_merge will pop the row which compares smallest on all streams.

In a ClustrixDB cluster, data is typically hash distributed across the nodes and since stream_combine returns whatever arrives first, the results may look different than databases which are not distributed and always read the data in order. For example:

```
sql> SELECT * FROM customers LIMIT 10;
+------+---------------------+--------------+-------------+-------+-------+
| c_id | name                | address      | city        | state | zip   |
+------+---------------------+--------------+-------------+-------+-------+
|    1 | Chanda Nordahl      | 4280 Maple   | Greenville  | WA    | 98903 |
|    2 | Dorinda Tomaselli   | 8491 Oak     | Centerville | OR    | 97520 |
|    9 | Minerva Donnell     | 4644 1st St. | Springfield | WA    | 98613 |
|   21 | Chanda Nordahl      | 5090 1st St. | Fairview    | OR    | 97520 |
|    4 | Dorinda Hougland    | 8511 Pine    | Springfield | OR    | 97477 |
|    6 | Zackary Velasco     | 6296 Oak     | Springfield | OR    | 97286 |
|   11 | Tennie Soden        | 7924 Maple   | Centerville | OR    | 97477 |
|    3 | Shawnee Soden       | 4096 Maple   | Ashland     | WA    | 98035 |
|   24 | Riley Soden         | 7470 1st St. | Greenville  | WA    | 98613 |
|   12 | Kathaleen Tomaselli | 8926 Maple   | Centerville | OR    | 97477 |
+------+---------------------+--------------+-------------+-------+-------+
```

Repeating this query may get different results. By adding an ORDER BY clause to the statement, we can ensure that we get consistent results. To make things more interesting, we'll also change the ordering from ascending to descending.

```
sql> EXPLAIN SELECT * FROM customers ORDER BY c_id DESC LIMIT 10;
+------------------------------------------------------------------+-----------+-----------+
| Operation                                                        | Est. Cost | Est. Rows |
+------------------------------------------------------------------+-----------+-----------+
| row_limit LIMIT := param(0)                                      |    622.70 |     10.00 |
|   stream_merge KEYS=[(1 . "c_id") DSC]                           |    622.70 |     30.00 |
|     row_limit LIMIT := param(0)                                  |    203.90 |     10.00 |
|       index_scan 1 := customers.__idx_customers__PRIMARY REVERSE |    203.90 |    333.33 |
+------------------------------------------------------------------+-----------+-----------+
```

We can see from this execution plan, the database will first read the primary index in parallel and in reverse on all available slices with the index_scan operator, stop reading after 10 rows are found using the row_limit operator, merge those streams by selecting the greatest value for c_id from each stream with the stream_merge operator, and finally limit that to 10 rows via a repeated application of the row_limit operator.

Explaining Joins

So far, we have been looking at single relation reads. One of the Sierra's jobs is to compare the cost of different join orderings and choose the plan with the lowest cost. This query will yield the order id, product name, and price for every row in order_items.

```
1.   | sql> EXPLAIN SELECT o_id, name, price FROM orders o NATURAL JOIN order_items NATURAL JOIN
products;
2.   +-------------------------------------------------------------------------------+-----------+-----------+
3.   | Operation                                                                     | Est. Cost | Est. Rows |
4.   +-------------------------------------------------------------------------------+-----------+-----------+
5.   | nljoin                                                                        |  95339.90 |   9882.00 |
6.   |   nljoin                                                                      |  50870.90 |   9882.00 |
7.   |     stream_combine                                                            |     82.70 |    100.00 |
8.   |       index_scan 3 := products.__idx_products__PRIMARY                        |     23.90 |     33.33 |
9.   |     nljoin                                                                    |    507.88 |     98.82 |
10. |       index_scan 2 := order_items.p_fk, p_id = 3.p_id                         |     63.19 |     98.82 |
11. |       index_scan 2 := order_items.__idx_order_items__PRIMARY, oi_id = 2.oi_id |      4.50 |      1.00 |
12. |   index_scan 1 := orders.__idx_orders__PRIMARY, o_id = 2.o_id                 |      4.50 |      1.00 |
13. +-------------------------------------------------------------------------------+-----------+-----------+
```

This plan is a little more complex and will require a little more explanation to see what is happening.

1. Given the indentation, we can infer that an index_scan will happen first. In the output of the explain, we can see that the p_id found in the index_scan of the primary key of products on line 8 is used when reading the p_fk index of order_items on line 10 and the oi_id is used when reading the order_items primary key index on line 11. At essentially the same time, the products are collected with the stream_combine operator and the order_items information is collected by doing an nljoin of order_items.p_fk and the order_items primary key index.
2. The nljoin operator is a nested loop join which implements a relational equi-join.
3. The output of the products stream_combine and order_items nljoin are then joined in another nljoin.
4. The order_items.o_id is used to read orders and the results are all put together in a final nljoin.

Looking at the estimated rows in the final nljoin let's us know that in this particular data set, Sierra thinks there are approximately 9882 order_items rows.

Stage	Operation	Lookup/Scan representation	Lookup/Scan Key	Run on Node
1	Lookup and Forward	__idx_products__PRIMARY	none (all nodes with slices)	The node where the query begins
2.1	Index Scan	__idx_products__PRIMARY	None, all rows	Nodes with slices of __idx_products__PRIMARY
2.2	Lookup and Forward	p_fk	p_id = 3.p_id	same
3.1	Index Scan	p_fk	p_id = 3.p_id	Nodes with slices of p_fk
3.2	Join			same
3.3	Lookup and Forward	__idx_order_items__PRIMARY	oi_id = 2.oi_id	same
4.1	Index Scan	__idx_order_items__PRIMARY	oi_id = 2.oi_id	Nodes with slices of __idx_order_items__PRIMARY
4.2	Join			same
4.3	Lookup and Forward	__idx_orders__PRIMARY	o_id = 2.o_id	same
5.1	Index Scan	__idx_orders__PRIMARY	o_id = 2.o_id	Nodes with slices of __idx_orders__PRIMARY

5.2	Join			
5.3	Lookup and Forward	GTM	none - single GTM node	
6	Return to user			The node where the query began

Locks

ClustrixDB uses Two-phase locking (2PL) as a concurrency control to guarantee serializability. Sierra will plan locks for writes as well as reads for update in a transaction. First, we'll examine a simple update which increases the value of price by 1 when greater than 10.

```
sql> EXPLAIN UPDATE products SET price = price + 1 WHERE price > 10;
+--------------------------------------------------------------------------+-----------+-----------+
| Operation                                                                | Est. Cost | Est. Rows |
+--------------------------------------------------------------------------+-----------+-----------+
| table_update products                                                    |   1211.58 |     81.00 |
|   compute expr0 := (1.price + param(0))                                  |   1211.58 |     81.00 |
|     filter (1.price > param(1))                                          |   1210.50 |     81.00 |
|       nljoin                                                             |   1208.70 |     90.00 |
|         pk_lock "products" exclusive                                     |    803.70 |     90.00 |
|           stream_combine                                                 |     83.70 |     90.00 |
|             filter (1.price > param(1))                                  |     24.57 |     30.00 |
|               index_scan 1 := products.__idx_products__PRIMARY           |     23.90 |     33.33 |
|         index_scan 1 := products.__idx_products__PRIMARY, p_id = 1.p_id |      4.50 |      1.00 |
+--------------------------------------------------------------------------+-----------+-----------+
```

In this query plan:

1. We read the products primary key index with an index_scan and send the output to a "pushed down" filter which discards rows with a price not greater than 10 at each slice.
2. Those outputs are then combined with a stream_combine and that stream is distributed across the cluster to acquire an exclusive primary key lock with the pk_lock operator on the rows found.
3. We can then use the p_id found and read the primary key index with another index_scan.
4. The filter is applied again since the row found in the first index_scan may have had a price change since reading the row and acquiring the lock.
5. Matching rows are sent to a compute operator which calculates the new value for price and the new row is sent to the table_update operator which writes the new value.

At some point, taking individual row locks for every row to modify is more expensive than simply acquiring a single table lock and modifying all qualifying rows. The Sierra optimizer will consider using table locks instead of row locks during plan exploration and choose the plan with the lowest cost. In this example, 100 rows would normally be too small to bother with a table lock though if Sierra chose to take a table lock, the plan would look like the following explain.

```
sql> EXPLAIN UPDATE products SET price = price + 1;
+-------------------------------------------------------------+-----------+-----------+
| Operation                                                   | Est. Cost | Est. Rows |
+-------------------------------------------------------------+-----------+-----------+
| table_locks 1                                               |   8084.03 |    100.00 |
|   table_update products                                     |     84.03 |    100.00 |
|     stream_combine                                          |     84.03 |    100.00 |
|       compute expr0 := (1.price + param(0))                 |     24.34 |     33.33 |
|         table_lock "products" exclusive                     |     23.90 |     33.33 |
|           index_scan 1 := products.__idx_products__PRIMARY |     23.90 |     33.33 |
+-------------------------------------------------------------+-----------+-----------+
```

What is interesting here is that it looks like index_scan is an input to table_lock. This not the case since the table lock will be acquired prior to any read. With this in mind, we can see the plan:

1. Reads all rows in the relation with index_scan.
2. Adds 1 to the price with compute.
3. Combines those results into a single stream with stream_combine.
4. Sends the output to table_update to write the new value.

The table_lock operator is a helper operator for Sierra which has a heuristic for balancing the relatively inexpensive single lock with the fact that other updates are blocked, and thus consuming wall-clock time, during this transaction.

Using Indexes to Improve Performance

So far we have only examined reading the primary key index to get results. We can change this by adding indexes which make sense for a given workload. For example, let's imagine we have a business process that works better if we have our customer information sorted by zip code and coalesced into small chunks. To get this information:

```
sql> EXPLAIN SELECT name, address, city, state, zip FROM customers ORDER BY zip LIMIT 10
OFFSET 0;
+-----------------------------------------------------------+-----------+-----------+
| Operation                                                 | Est. Cost | Est. Rows |
+-----------------------------------------------------------+-----------+-----------+
| row_limit LIMIT := param(0)                               |   2632.70 |     10.00 |
|   sigma_sort KEYS=[(1 . "zip") ASC]                       |   2632.70 |   1000.00 |
|     stream_combine                                        |    712.70 |   1000.00 |
|       index_scan 1 := customers.__idx_customers__PRIMARY |    203.90 |    333.33 |
+-----------------------------------------------------------+-----------+-----------+
```

This reads the primary key index, combines the results and then sends those rows to a sigma_sort operator. The sigma_sort operator builds a temporary container in memory or in storage as necessary to sort the rows found by zip code. Once all of the results are sorted, they are passed along to the row_limit operator to enforce the limit and offset.

We can significantly improve the performance here if we read the zip code in order rather than reading all the rows, sort on zip code, and then return the next batch of 10 rows. To do this, we add an index on customers.zip and see how Sierra changes the execution plan.

```
sql> ALTER TABLE customers ADD INDEX (zip);
sql> EXPLAIN SELECT name, address, city, state, zip FROM customers ORDER BY zip LIMIT 10 OFFSET 0;
+----------------------------------------------------------------------+-----------+-----------+
| Operation                                                            | Est. Cost | Est. Rows |
+----------------------------------------------------------------------+-----------+-----------+
| msjoin KEYS=[(1 . "zip") ASC]                                        |    674.70 |     10.00 |
|   row_limit LIMIT := param(0)                                        |    622.70 |     10.00 |
|     stream_merge KEYS=[(1 . "zip") ASC]                              |    622.70 |     30.00 |
|       row_limit LIMIT := param(0)                                    |    203.90 |     10.00 |
|         index_scan 1 := customers.zip                                |    203.90 |    333.33 |
|   index_scan 1 := customers.__idx_customers__PRIMARY, c_id = 1.c_id |      4.50 |      1.00 |
+----------------------------------------------------------------------+-----------+-----------+
```

Here, the query optimizer chooses to:

1. Read the customers.zip index in order with the index_scan operator on all slices in parallel.
2. Limit the results with the "pushed down" row_limit operator.
3. Merge those results and preserve order with the stream_merge operator.
4. Limit the merged results with another row_limit.
5. Use the c_id found in the zip index to read the rest of the row.
6. Use the msjoin operator to perform the equi-join.

The msjoin operator is a "merge sort nested-loop join" which is similar to nljoin, but preserves sort order during a join. Notice that in this plan, the sort order is read for the zip index and preserved all the way through the plan which eliminates the need to create a sigma container to sort the results. In other words, this plan streams all results as it goes which can be an important consideration when reading millions of rows.

Aggregates

Another common task when working with a relational database is to sift through big data to calculate SUM, AVERAGE, MINIMUM, or MAXIMUM. These queries are executed by adding a GROUP BY clause to your statement which declares how you want the data to be aggregated. ClustrixDB also implements the MySQL extension to GROUP BY to allow inclusion of non-aggregated columns in the output columns. If there is not a one-to-one relation between the GROUP BY columns and the non-aggregated columns, the value of the non-aggregated columns will be one of the values in the row though which value is returned is not defined. Since we have a one-to-one mapping between zip and state in our data, we can generate a result set which generates that mapping for us.

```
sql> EXPLAIN SELECT zip, state FROM customers GROUP BY zip;
+---------------------------------------------------------+-----------+-----------+
| Operation                                               | Est. Cost | Est. Rows |
+---------------------------------------------------------+-----------+-----------+
| sigma_distinct_combine KEYS=((1 . "zip"))               |   1303.90 |   1000.00 |
|   sigma_distinct_partial KEYS=((1 . "zip"))             |    203.90 |   1000.00 |
|     index_scan 1 := customers.__idx_customers__PRIMARY |    203.90 |    333.33 |
+---------------------------------------------------------+-----------+-----------+
```

This query will:

1. First, perform an index_scan and send the output the sigma_distinct_partial operator.
2. The sigma_distinct_partial operator which produces an output of one row per distinct value of KEYS on the same node as the read.
3. Those distinct values are then sent to the sigma_distinct_combine operator which will do the same distinct operations on KEYS on the node where the query was initiated.

For a more realistic aggregation, let's presume we want to find how many orders each customer has placed and that customer's name.

```
sql> EXPLAIN SELECT c.name, COUNT(*) FROM orders o NATURAL JOIN customers c GROUP BY o.c_id;
+--------------------------------------------------------------------------------+-----------+-----------+
| Operation                                                                      | Est. Cost | Est. Rows |
+--------------------------------------------------------------------------------+-----------+-----------+
| hash_aggregate_combine GROUPBY((1 . "c_id")) expr1 := countsum((0 . "expr1")) |  12780.38 |   4056.80 |
|   hash_aggregate_partial GROUPBY((1 . "c_id")) expr1 := count((0 . "expr0"))  |   7100.87 |   4056.80 |
|     compute expr0 := param(0)                                                  |   7100.87 |   4056.80 |
|       nljoin                                                                   |   7046.78 |   4056.80 |
|         stream_combine                                                         |    712.70 |   1000.00 |
|           index_scan 2 := customers.__idx_customers__PRIMARY                   |    203.90 |    333.33 |
|         index_scan 1 := orders.c_fk, c_id = 2.c_id                             |      6.33 |      4.06 |
+--------------------------------------------------------------------------------+-----------+-----------+
```

In this plan:

1. The index_scan of the customer's primary key is first and combined with stream_combine and the c_id is used to read the orders.c_fk index with another index_scan.
2. Those results are joined on the node where we read the orders.c_fk index with the nljoin operator and grouped and counted with the hash_aggregate_partial operator on the same node.
3. The results are then sent to the hash_aggregate_combine operator on the originating node for a final group and count before returning rows to the user.

Summary

Hopefully, this is a sufficient introduction to the EXPLAIN output for the ClustrixDB Sierra Query Optimizer that you use to examine your own queries. For a complete list of the operators which might show up in the EXPLAIN, consult the List of Planner Operators. For more information on how Sierra does query optimization, see Query Optimizer in Distributed Database Architecture.