Query Optimizer

How the ClustrixDB Query Optimizer is Different

At the core of ClustrixDB Query Optimizer is the ability to execute one query with maximum parallelism and many simultaneous queries with maximum concurrency. This is achieved via a distributed query planner and compiler and a distributed shared-nothing execution engine.

What is a Query Optimizer?

SQL is a declarative language, i.e. a language that describes *what* is to be computed but not *how*. The job of the query optimizer is to determine how to do this computation, which ends up being critical to the performance of the entire system. For example, you might say in SQL that you want to join 3 tables and compute an aggregate operation. This leaves the following questions for the Query Optimizer:

- In what order should the tables be joined? This can be the difference between your query executing in 1ms or 10 minutes. Imagine if a predicate on one of the tables causes it to return no rows – starting the read from that table is likely optimal and fast.
- Which indexes should be used? Not using a proper index on a join constraint could be catastrophic, causing broadcast messages and full reads of the second table for each row of the first.
- Should we provide a non-blocking sort/aggregate? Should we do the sort/aggregate in stages, i.e. first on separate nodes and then re-aggregate /re-sort later?

The set of query plans that these permutations create is known as the Search Space. The job of the Query Optimizer is to explore the Search Space and determine which plan uses the least amount of database resources. Typically, this is done by assigning costs to each plan, then choosing the cheapest one.

The ClustrixDB Query Optimizer is also known as Sierra.

An Example

```
sql> -- Query
sql> SELECT Bar.a,
            Sum(Bar.b)
     FROM    Foo, Bar
     WHERE  Foo.pk = Bar.pk
     GROUP  BY Bar.a;

sql> -- Foo schema
sql> CREATE TABLE `Foo`
        ( `pk` INT(11) NOT NULL auto_increment,
          `a`  INT(11),
          `b`  INT(11),
          `c`  INT(11),
          PRIMARY KEY (`pk`) /*$ DISTRIBUTE=1 */,
          KEY `idx_ab` (`a`, `b`) /*$ DISTRIBUTE=2 */
        ) auto_increment=50002;

sql> -- Bar schema
sql> CREATE TABLE `Bar`
        ( `pk` INT(11) NOT NULL auto_increment,
          `a`  INT(11),
          `b`  INT(11),
           `c`  INT(11),
           PRIMARY KEY (`pk`) /*$ DISTRIBUTE=1 */,
           KEY `idx_ab` (`a`, `b`) /*$ DISTRIBUTE=2 */
         ) auto_increment=25001;
```

SQL Representation – What	Sierra Output – How

(display ((2 . "a") (0 . "expr0"))
  (aggregate ((2 . "a") ("expr0" sum (2 . "b")))
  (filter
  (inner_join
  (table_scan (1 : Foo ("pk" "a" "b" "c")))
  (table_scan (2 : Bar ("pk" "a" "b" "c")))
  (func eq
  (ref (1 . "pk"))
  (ref (2 . "pk"))))))
	(display ((2 . "a") (0 . "expr0"))
	  (stream_aggregate ((2 . "a") ("expr0" sum (2 . "b")))
	  (msjoin
	  (stream_merge
	  (index_scan (2 : Bar.idx_ab ("b" "pk" "a")))
	  (index_scan (1 : Foo.idx_PRIMARY ("pk"))
	  range equal (ref (2 . "pk"))))))

Diagram 1	

First, some explanations for these plans. We'll go from most indented to least indented. These descriptions should be read top to bottom for equally indented statements.

This SQL statement is performing the following:

- Read table Foo and join to Bar.
- Evaluate the join constraint.
- Compute the aggregate.
- Display the output to the user.

The Query Optimizer is doing the following:

1. Read table Bar, preserving the sort order via stream_merge.
2. Evaluate the join constraint while reading Foo and still preserving sort order via msjoin this time.
3. Compute the aggregate in a non-blocking way with stream_aggregate. This can be done because we preserved order.

How did the Query Optimizer find this plan? How did it make sure it even worked? Lets find out:

The ClustrixDB Query Optimizer

Sierra is modeled off of the Cascades Query optimization framework, which was chosen primarily because it provides the following:

- Cost-driven
- Extensible via a rule based mechanism
- Top-down approach
- General separation of logical vs. physical operators and properties
- Branch-and-bound pruning

Modern query optimizers are often split into two parts, the Model and Search Engine.

The Model lists the equivalence transformations (rules), which are used by the search engine to expand the search space.

The Search Engine defines the interfaces between the search engine and the model, and provides the code to expand the search space and to search for the optimal plan. This is implemented by the stack of tasks waiting to be computed. More on this below.

Terminology

Logical vs. Physical Model

In the Query Optimizer, the Logical model describes *what* is to be computed, and the Physical model describes *how* it is to be computed. In diagram 1 above, the SQL representation shows logically what to do and the Sierra output shows physically how to do it.

Operators and Expressions

An expression consists of: an Operator (required), Arguments (possibly none), and Inputs (possibly none). Arguments describe particular characteristics of the operator. There are both logical and physical operators and every logical operator maps to 1 or more physical operators. In the example the logical operator:

```
(table_scan (1 : Foo  ("pk" "a" "b" "c")))
```

maps to the physical expression:

```
(index_scan (1 : Foo  ("pk") (range equal (ref  (2 . "pk")))
```

These operators have arguments which describe their Namespace, Object Id, and Columns. Here, the table_scan has no inputs and index_scan has an input that represents the join constraint. (See List of Planner Operators.)

Physical Properties

Physical properties are related to intermediate results, or sub-plans. They describe things like how the data is ordered and how the data is partitioned. It is important to note that expressions (either logical or physical) and groups (see below) do not have physical properties. However, every physical expression has two descriptions related to physical properties:

1. What Physical properties can and can't I provide to my parent?
2. What Physical properties do I require of my input?

Here are some considerations Sierra takes while optimizing our query:

Physical Property Flow	
Invalid Plan	Valid Plan

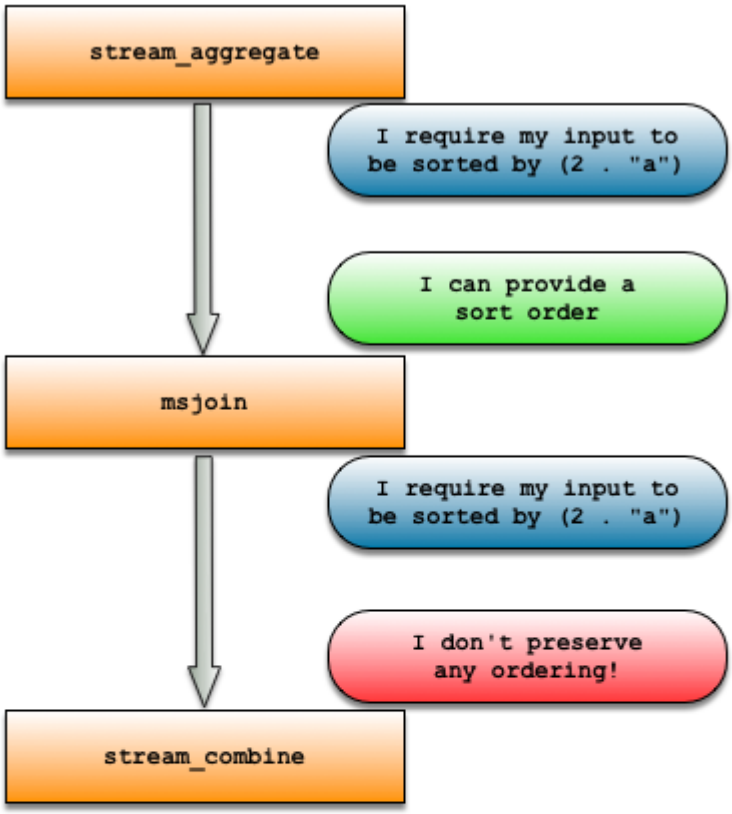
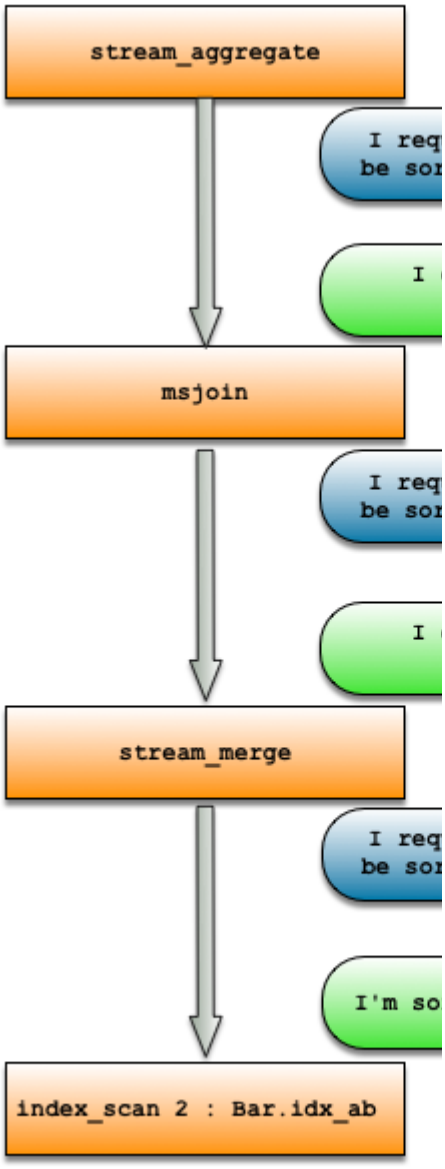

	

Diagram 2	

The invalid plan fails because the stream_combine operator is unable to preserve any kind of ordering that its inputs provide. However, in the valid plan, stream_merge is used, which can preserve the sort order of its child, and the index_scan itself does have sort order. In effect, plans which may or may not be valid are explored and the physical properties are used in order to validate whether they are possible. If any operator in the chain fails, the plan is invalidated.

Groups

Groups correspond to intermediate tables, or equivalently subplans of the query. Groups are logical and contain the following:

1. All the logically equivalent expressions that describe that intermediate table.
2. All the physical implementations of those logical expressions.
3. Winners: A physical expression that had the best cost given a set of physical properties.
4. Logical properties: Which columns it is required to produce as well as statistics about some of those columns.

Groups are the fundamental data structure in Sierra. The inputs to operators are always groups (indicated by group #'s), and every expression corresponds to some group.

The Group for "Read Table Bar"

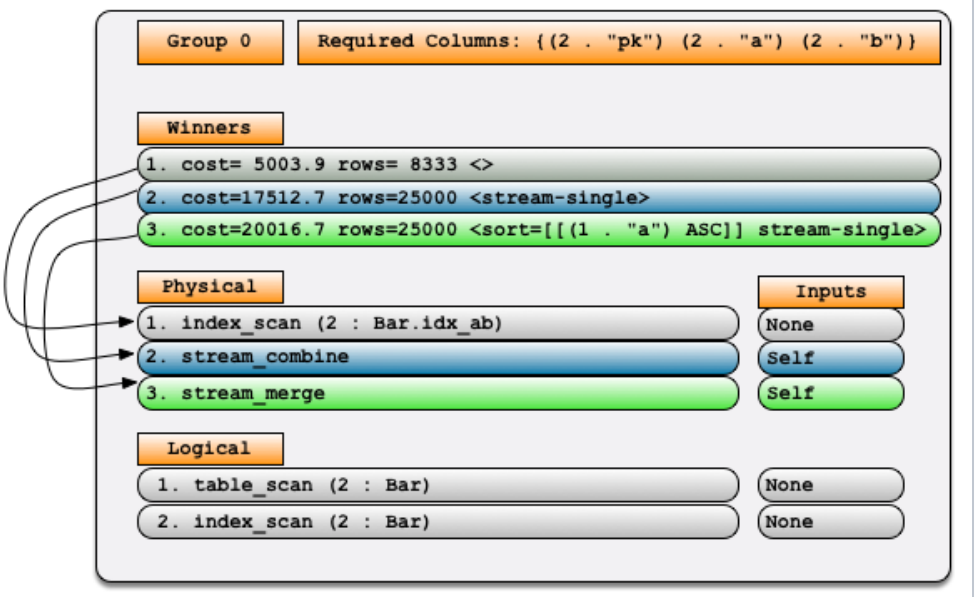


Diagram 3

The Memo

In the process of optimization, Sierra will keep track the intermediate tables that could be used in computing the final result table. Each of these corresponds to a group, and the set of all groups for a plan defines the memo. In Sierra, the memo is designed to represent all logical query trees and physical plans in the search space for a given initial query. The memo is a set of groups, with one group designated as the final (or top) group. This is the group which corresponds to the table of results from the evaluation of the initial query. Sierra has no explicit representation of all possible query trees, in part because there are simply too many. Instead, this memo represents a compact version of all possible query trees.

Rules (The Model)

The model's rule set can be thought of as defining the logical and physical search space of the optimizer. The memo is expanded to encompass the full logical and physical search space through the application of rules. The application of rules is a multi-step process of finding a binding in the memo, evaluating a rule condition (if the rule has one) and (if the condition is satisfied) firing the rule, which puts new expressions in the memo. When the optimizer is done applying rules, the memo structure will have been expanded to where it conceptually represents the entire search space. Here is an example of the swap_rule firing. It was responsible for exploring the join ordering picked in the final plan of the example.

Swap Join Order Rule	

```
(display
     ((1 . "pk") (2 . "pk")
     (inner_join
           (index_scan (1:Foo ("pk")))
           (index_scan (2:Bar ("pk"))
                (range_equal ref (1 .
               "pk"))))
```

```
(display
     ((1 . "pk") (2 . "pk")
     (inner_join
           (index_scan (2:Bar ("pk"))
           (index_scan (1:Foo ("pk")))
                (range_equal ref (2 .
               "pk"))))
```

Diagram 4

Tasks (The Search Engine)

Sierra's search engine is a series of tasks that are waiting to be computed. At any point in time during optimization, there are tasks waiting on a stack to be executed. Each task will likely push more tasks onto the stack in order to be able to achieve its goal. Sierra is done computing once the stack is empty. Sierra begins by taking an input tree and constructing the corresponding initial groups and expressions. Then, it starts off the search engine by pushing the task Optimize_group (top group). This starts off the chain of events that explores the entire search space, finds the cheapest winners for each group, and finally chooses the cheapest winner in the top group to be its output plan.

The Search Engine

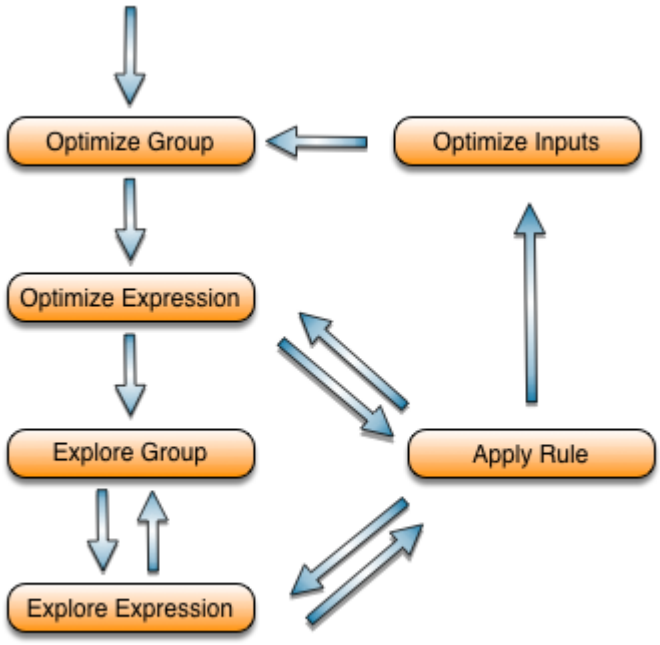


Diagram 5

The Cost Model

Sierra costs plans using a combination of I/O, CPU usage, and latency. Remember that ClustrixDB is distributed so total CPU usage and latency are not proportional. Every operator describes a function in order to compute its costs given its inputs. For example an index_scan uses the row estimation framework to compute how many rows it expects to read from the btree and then its computes its cost as such:

```
index_scan.rows = row_est()
index_scan.cost = cost_open + row_est() * cost_read_row
```

The operator above the index_scan would then use this cost and row estimate to estimate its own cost.

Row Estimation

The way Sierra chooses the optimal plan for a query is by finding the plan with the cheapest cost. Cost is strongly dependent on how many rows the optimizer thinks are going to be flowing through the system. The job of the row estimation subsystem is to take statistical information from our Probability Distributions and compute an estimated number of rows that will come out of a given expression.

The Explain

For the query above, we can get a succinct description of the plan, the row estimates, and the cost by prefacing the query with 'explain'. (For additional information, see Understanding the ClustrixDB Explain Output.)

```
sql> explain select bar.a, sum(bar.b) from bar,foo where foo.pk = bar.pk group by bar.a;

+-----------------------------------------------------------+-----------+-----------+
| Operation                                                 | Est. Cost | Est. Rows |
+-----------------------------------------------------------+-----------+-----------+
| stream_aggregate GROUPBY((1 . "a")) expr0 := sum((1 . "b")) | 142526.70 |  25000.00 |
|   msjoin KEYS=[((1 . "a")) GROUP]                         | 137521.70 |  25000.00 |
|     stream_merge KEYS=[(1 . "a") ASC]                     |  20016.70 |  25000.00 |
|       index_scan 1 := bar.idx_ab                          |   5003.90 |   8333.33 |
|     index_scan 2 := foo.__idx_foo__PRIMARY, pk = 1.pk     |      4.50 |      1.00 |
+-----------------------------------------------------------+-----------+-----------+
5 rows in set (0.01 sec)
```

Summary

To summarize how Sierra got the output in diagram 1, the following steps were performed:

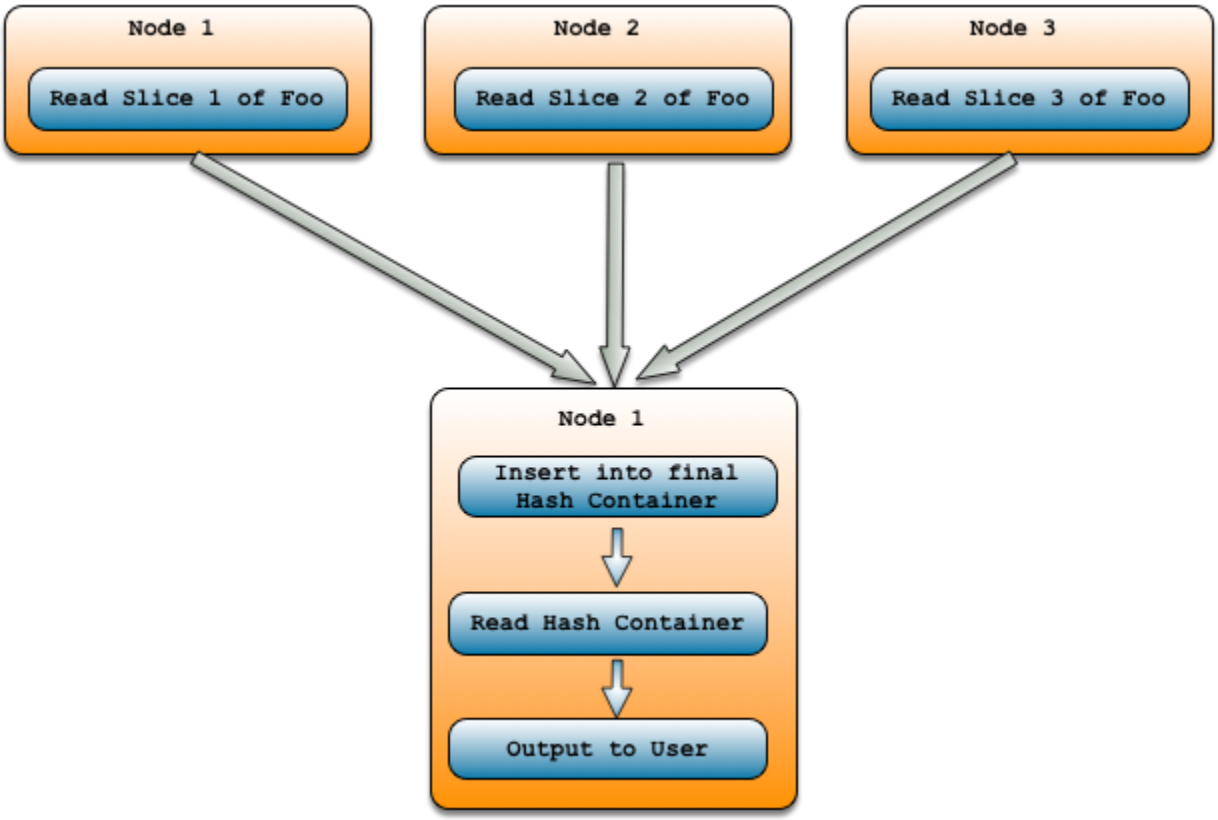
1. Sierra created many groups and expressions corresponding to the SQL Representation of diagram 1.
2. In an on-demand fashion via the Task mechanism Sierra then:
    a. Fired rules in order to expand the logical and physical search space of the query, creating additional groups and expressions, some of which might not correspond to any valid plan.
    b. Shrank the search space to the plans that were valid, with the use of physical properties.
    c. Used the cost model in order to determine the cheapest plans from a given set of physical properties for each group.
3. After all tasks had been processed, Sierra extracted the query plan corresponding to the winner of the top group, which is the output of diagram 1.

Distributed Considerations

We've talked a lot about how the query optimizers finds the best plan, but so far the concepts are not unique to ClustrixDB. One of the special things about Sierra is that it is able to reason about doing distributed operations. For example there are two ways to compute an aggregate. Let's understand the non-distributed way first:

a. Read table Foo which likely has slices on multiple nodes.
b. Forward all those rows to one node.
c. Insert all those rows into a hash container, computing the aggregate operation if necessary.
d. Read the container and output to the user.

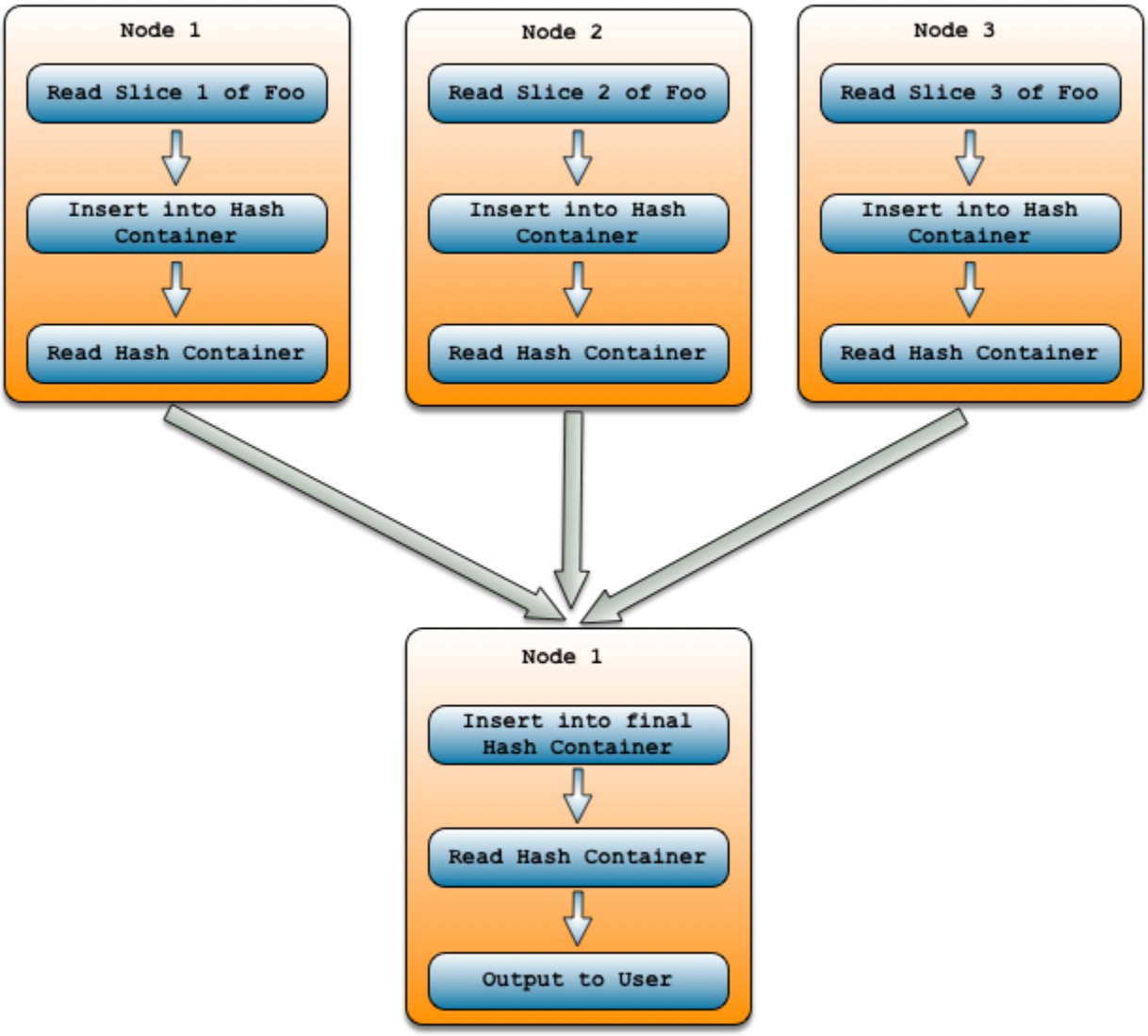
Here's what it would like:



But really we could distribute the aggregate and do this instead:

a. Compute the sub-plan (under the aggregate), which likely has result rows on multiple nodes.
b. Locally insert those rows into a local hash container, computing the aggregate operation if necessary.
c. Read the hash container on each node and forward to a single node.
d. If necessary:
    i. Insert all those rows into a new final hash container, computing the aggregate operation.
    ii. Read that hash container.
e. Output rows to the user.

Here's what this would look like:



The question for Sierra becomes which one is better and when? It turns out the gains from distributing the aggregate actually come from the fact that we are potentially sending a lot less data across the wire (between nodes), so the overhead of extra inserts and containers becomes worth it when the reduction factor of the aggregate operation is large. Sierra is able to reason about this with the cost model and determine the better approach for any query. For additional information, see Scaling Aggregates.