CS422 Project0 - Iterator Model(Volcano)

Liangze Jiang
École Polytechnique Fédérale de Lausanne, Switzerland

***Abstract*—In this project, we implement a volcano-style tuple-at-a-time engine, which includes six basic operators(Scan, Select, Project, Join, Aggregate and Sort). Finally, we manage to pass all the provided test without timeout, and the best speedup over baseline was 0.99 with a 4.59s average execution time.**

I. Scan

The Scan operator is very fundamental. In this project, we need to support the RowStore input layout, meaning that we output a single row every time the 'next' function is executed, given a scannable table. We choose to follow the skeleton code and use pattern matching to realize it. The pattern matching block matches the scannable table and check its structure(RowStore, ColumnStore or PAXStore) and return specific row. For now, we only match the RowStore structure.

II. Select

The Select operator plays a role similar to a filter, corresponding to the WHERE in SQL. Given a condition, it can check whether a specific row satisfies this condition. In 'next' function, we start from the head of input and check the condition iteratively, if the current record satisfies the condition, then we return this record, or we re-execute the 'next' function until we get a record or encounter EOF.

III. Project

The Project operator selects on the attributes(columns in the case of RowStore), corresponding to the SELECT in SQL. The skeleton code provide an evaluator which takes the original tuple and outputs the selected attributes. So, we only have to iteratively walk through the table and return an evaluated tuple when 'next' function is executed, provided that current tuple is not NilTuple or EOF.

IV. Join

The Join operator in this project only performs as equijoin, which corresponds to the FROM and some equality conditions in WHERE. Join operator takes two tables as input and outputs a joined table, each record in the joined table is a concatenation of two records in input tables who satisfies the equality conditions.

A. Nested Loop

A straightforward and simple method is to form the joined table in the initialization phase, then every time we call 'next' function, just return a single record. We firstly implement the simplest way which uses nested loop to check all the combinations of two input tables and check the equality condition for every single pair. This is not efficient because a nested loop results in a quadratic complexity(O(MN)), and our first implementation only reach a speedup over baseline by 0.6, with a 7.62s Average execution time.

B. Hash Join

Taking the advantage of the fact that all the joins are equi-joins in this project, we can implement hash join method to better the performance of join by a lot. In hash join, we first choose the ideally smaller table and group it by the condition key's hash code, called build phase. Then for each record in the second table, we compute its condition key's hash code and check whether the map in build phase contains this hash code. By this we can efficiently find the tuple pair that co-satisfiy the equality condition. Th e algorithm walk through two table separately, resulting in a O(M+N) complexity. Finally, it boosts our speedup over baseline from 0.6 to 0.86, with a 5.28s average execution time.

V. Aggregate

The aggregate operator performs functions on each group of the table. In the aggregate operator, we highly rely on the scala's functions, namely 'foldLeft'. For every aggregate calls, we get the argument of each tuples and reduce them by the reduce function provided by the skeleton code. And we put the workload on 'open' function to generate aggregated table.

VI. Sort

We may need sorted result in practical use, so the sort operator takes several attributes as sort key and sort them in possibly different order. In this project, we use the 'sortWith' function in scala and implement a custom comparison function that compare every single attribute and break the function as long as we get the comparison result in order to reduce execution time. And we put the workload on 'open' function to generate sorted table.

TABLE I
Improvements on First Commit

	Speedup	Time
First Commit	0.6	7.62s
Hash Join	0.86	5.28s
Hash Join + Code Cleaning	0.99	4.59s

VII. Optimization Strategies

Two main optimization strategies are used in this project, namely hash join and avoid unnecessary function calls. To avoid unnecessary function calls, we can check: if there are some intermediate results calculate again and again, then we should use a variable to record them. The hash join was introduced before. The comparison of these strategies are shown in Table I.

In addition, it is important to always be aware of the possibility of empty input or parameters, and one more thing is that inappropriate uses of iterator can cause a timeout.

VIII. Results and Further Works

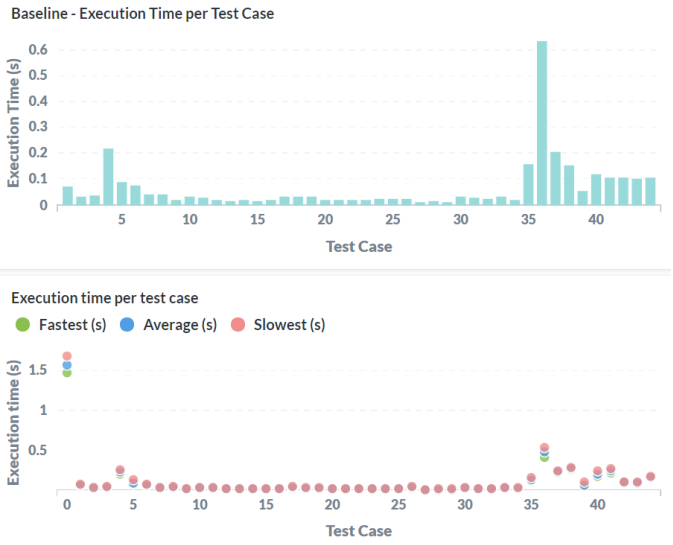

Fig. 1. Comparison of time between our results and baseline's

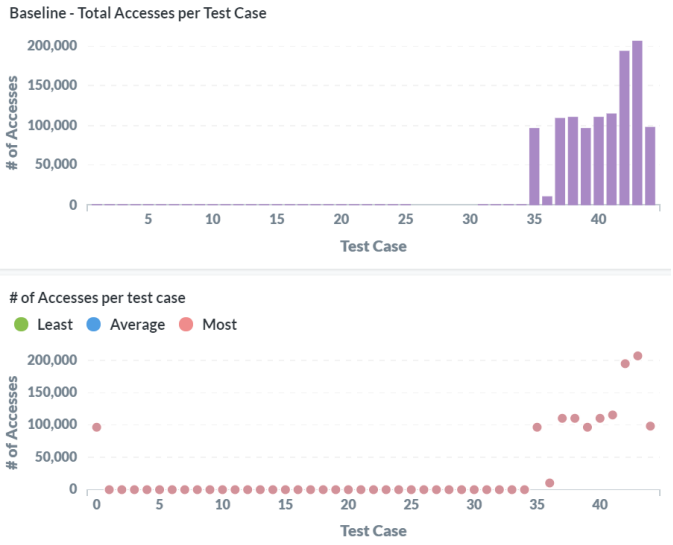

Fig. 2. Comparison of access between our results and baseline's

From Table I and Figure 1 we can see that our implementation is very close to the performance of baseline. Also, the access of our implementation is exactly the same as baseline's. And the best speedup over baseline by case is case 31, which reaches a 1.41 speedup.

As for further works, we think that more advanced sorting algorithms can further improve the performance. Based on the offset and fetch parameters, we may only want to extract part of sorted table. For example, the table has 1 million records and we only want the top 101-200 records. What we could do is implement a partial sorting algorithm based on min-heap. In this project, we checked the offset and fetch parameters in test case and found that mostly we need to retrieve the whole table, so we didn't implement it.

In addition, we think further steaming some input of operator can help to gain some performance too.