1 Motivation

2 Requirements

2.1 Use Cases

3 Background and Research

The goal of this document is to present design for supporting LATERAL and UNNEST in Drill. The motivation, requirements and background for why we want to do it is best described in High Level Design [1] document.

4 Design Overview

The design for the support of LATERAL and UNNEST is divided into 2 broad categories:

4.1 Planning Phase:

At a high level some of the planning side changes which will be required are as below. The detailed Planning phase changes are captured in document located here [4]

- UNNEST will always have a Project on top of it. This is to project columns from unnested data when columnToUnnest has array of structs or map.
- UNNEST and its corresponding LATERAL will have reference to each other.
- Planner to generate correct plan for cases when there is multiple UNNEST at each level. Need to further see if Calcite supports multiple UNNEST at same level or we should change the PLAN such that this case is handled by multiple levels of LATERAL and UNNEST in the pipeline.
- Planner to generate correct plan for cases when there is LATERAL at multiple levels. Example being when UNNESTing is required for *company.department.employee* where department and employee both are of complex types and employee is nested within the department. Here first level LATERAL will UNNEST department data and second level will UNNEST employee data for each department row.
- Planner should restrict the reference of any other table on right side of LATERAL other than from left side input of LATERAL.

- Planner should be able to differentiate between JOIN's such that when LATERAL keyword is present in subquery then it's replaced by LATERAL JOIN operator.

4.2 Execution Phase:

The execution phase design deals with the changes that are needed at runtime model of Drill. Drill follows an iterative model where each operator in the operator tree calls next() on it's upstream operator starting from root operator. Eventually when the next() of leaf operator is called which is usually Scan it performs the task of reading the data from its source, creates an output Record Batch and sends it to the downstream operator. The downstream operators on receiving this Record Batch performs the operation it is supposed to and create an output batch out of it to further send it downstream. Finally after going through each operator the final output Record Batch from root operator (i.e. Screen) is sent back to the client over the network channel.

Drill's operator can be broadly divided into 2 categories:

- **Blocking Operators:** Operators which block and don't produce any output batch until they see all the input from at least one of their children.
  Example: Hash-Join, Sort, etc
- **Unblocking Operators:** Operators which don't block to see all the input from at least one of its children before producing an output batch. It works on a batch by batch basis i.e. for every input record batch it can produce an output record batch.
  Example: Filter, Project, etc

Rule 1: *LATERAL and UNNEST both will fall in the category of* ***Unblocking operators.***

For LATERAL right side of the sub-graph will always have an UNNEST, such that the right side will always work on the left side of the input. This makes the Drill support of LATERAL and UNNEST different from what Postgres can support. Also right side of LATERAL will work on multiple rows at a time from LEFT side incoming of LATERAL instead of one row at a time. To achieve this the right side leaf operator of LATERAL which is UNNEST will produce an extra column containing the RowId of each left row. RowId starts from 1. This gives rise to a couple of new rules:

Rule 2: *LATERAL and UNNEST will always* ***coexist*** *with each other.*

Rule 3: *Right side of LATERAL will only refer to input on the left side of LATERAL. This means there cannot be any other data source which is referenced on the right side. Right side will always work on multiple rows from left and will have an implicit RowId column for each left row.*

Further in Drill there is concept of IterOutcome's which is used to indicate the state of one operator to another in the operator tree. Currently there are 6 states in use:

- **OK_NEW_SCHEMA**
- **OK**
- **NONE**
- **NOT_YET**
- **STOP**
- **OUT_OF_MEMORY**

To support both LATERAL and UNNEST which behaves like streaming operator we need to introduce a new state for IterOutcome which will be called **EMIT.** It is needed to make blocking operators in Drill like HashAgg / Sort, etc to emit an output without seeing all the data. This is like making these blocking operators behave like streaming operators. UNNEST will be the sole generator of **EMIT** outcome and LATERAL will be the only operator to consume it and not let it pass to downstream depending upon from which child branch it's received. All the other operators also need to be aware about how to handle this new state. Since LATERAL and UNNEST work on the same input they should be co-located in the same fragment and hence cannot be separated by an exchange. Also based on Rule 3 since right side of LATERAL cannot reference any other data source this also restricts the existence of any exchange operator on right branch. The rule which each operator should follow for EMIT IterOutcome is defined in section 4.2.1

Rule 4: *EMIT will be only be created or generated from UNNEST and consumed by LATERAL. Exchange operators will never see EMIT from upstream operators.*

4.2.1 EMIT Processing Rule:

EMIT is an indication from upstream operator to a downstream operator to unblock and generate an output batch based on data it currently has. Operators will resetup their state machine as needed and process the next set of data as a fresh set of inputs. It marks the output boundary for each left side batch processed by UNNEST. That is, it guarantees that the following record batch will not have any record from UNNEST output of current left side batch. In case of single level LATERAL, **EMIT** is only expected from right branch with UNNEST. Whereas in case of multi-level LATERAL, **EMIT** can be received from both the left and right branch of lower level LATERAL (Lateral_2 in figure below). General rule with **EMIT** in case of LATERAL will be that it will consume the **EMIT** outcome received from right branch and pass

the **EMIT** outcome received from left branch after processing all the records. This means that for each batch received from left side along with **EMIT** outcome, LATERAL will return OK with Each output except the last output for current left batch. For last output out of left current batch it will send **EMIT** along with it.

> ***Rule 5:*** *LATERAL will pass thru EMIT from left branch input, along with last output batch for that input. For all other output batches it will send OK. But LATERAL will consume EMIT from right branch and convert it to OK before passing it downstream.*

In figure below Lateral_2 will consume EMIT from UNNEST_2 but will pass through EMIT from UNNEST_1 to LATERAL_1. UNNEST_1 will send EMIT with last output batch for each input batch from left. LATERAL_2 will send EMIT to LATERAL_1 only with last output batch produced after joining left and right input. All previous output batches from LATERAL_2 for current left input will be sent with OK.

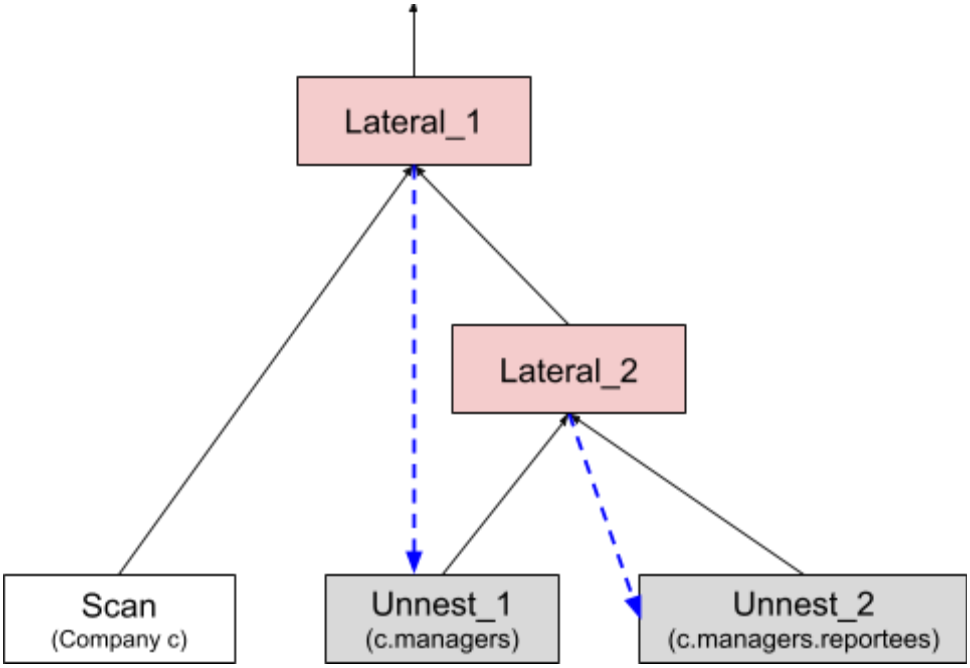


Figure 1: Example of Multilevel Lateral. Example data for this can be: Company having list of employees {emp_id, emp_name, salary}, list of managers {emp_id, dept_id, which in turn has list of reportees {emp_id}}

Let's consider a few examples to see how operator will populate output batch and treat EMIT outcome:

1. ***Single LATERAL and UNNEST with no other operators in right side branch:***

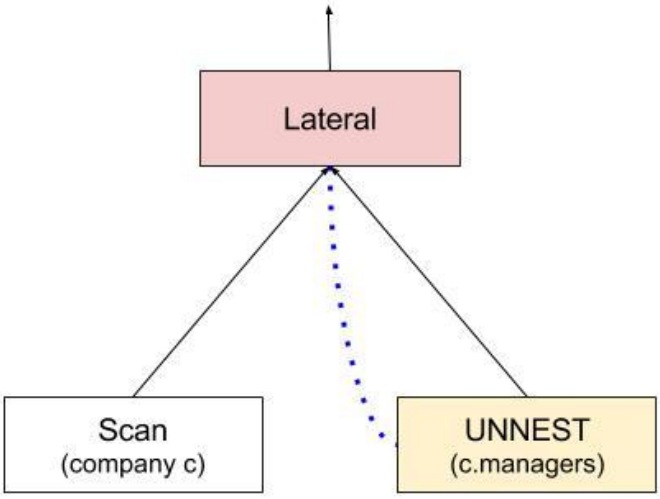


Figure 2: Plan for single lateral and UNNEST. Data is same as for figure 1, but we are unnesting only managers array for each customer record.

In above figure LATERAL will never see EMIT from left side. Whereas on the right side it will see EMIT outcome with last output batch of left incoming from UNNEST. All the other output batches produced from right side will come with OK outcome. On getting each output batch from right side, LATERAL will perform the join between **left row** and it's **right output batches** and populate the final output batch. It will keep populating the output batch until it's full and then send it downstream.

- In the event (*with combination of Filter on right branch and cross-join*) when entire left and right side batches are consumed in single output batch but still there is space left for fitting more records, in that case LATERAL will call next() on the left side until it sees another batch with at least 1 record and with OK outcome. Then it will call next() on the right side for this left side batch and populate the output batches after joining left and right. Once the output batch is filled it will send it downstream with OK outcome.
  - If it sees OK_NEW_SCHEMA from left, then it will pass previous output batch downstream and process new batch with OK_NEW_SCHEMA in subsequent next() call from above.
  - If it sees NONE from left, then it will pass previous output batch downstream with OK outcome and will return NONE in the subsequent next() call.
  - If it sees STOP/OutOfMemory then it will just return those outcomes.

- For UNNEST each output batch can contain fewer elements than the memory allocated for the batch. Since our batch size may be bigger than output produced by UNNEST.

2. ***Multilevel LATERAL with no other operator in right side branch:***

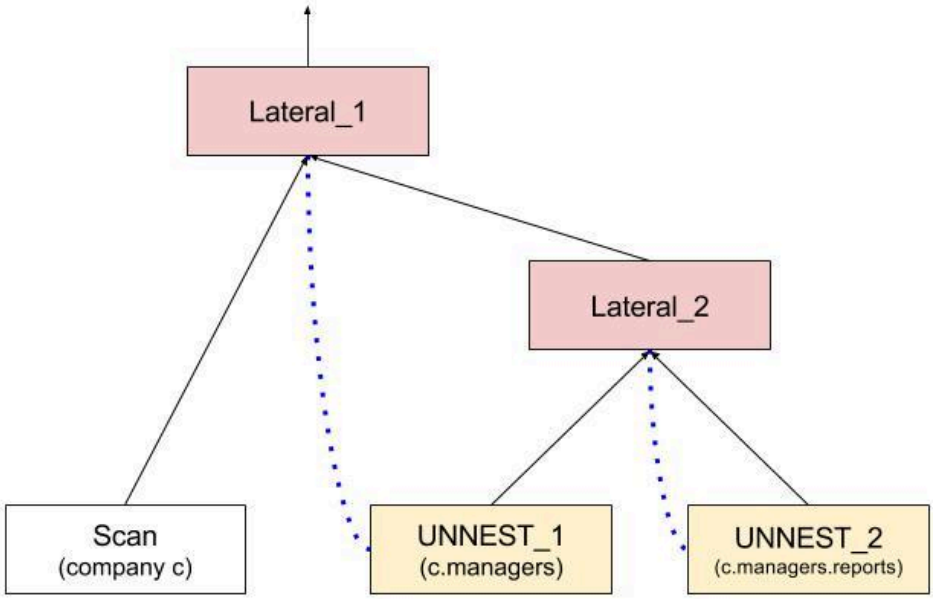


Figure 3: Plan for multilevel lateral and UNNEST. Same as Figure 1

In above figure LATERAL_1 will never see EMIT from left side. Whereas on the right side it will see EMIT outcome pass down by LATERAL_2. For LATERAL_2 it can see EMIT outcome from both left and right side. For each batch on the left side of LATERAL_1, it will call next() on right side which is on LATERAL_2. Then LATERAL_2 will call next on left() side on UNNEST_1. UNNEST_1 will produce one or more output batch along with EMIT / OK outcome.

- Let's consider the case when LATERAL_2 receives 2 batches from UNNEST_1, first will come with OK outcome and second batch will come with EMIT outcome. For the first batch it will call next() on UNNEST_2 and for each output batch received it will perform the join and populate the output batch. Let's consider the output batch has consumed entire output from join of first left batch from UNNEST_1 and all the corresponding output batches from UNNEST_2. There is still space left to populate more rows. Then LATERAL_2 will call next() on UNNEST_1 like before since the previous outcome was OK. On calling left.next() it will receive another batch but with EMIT outcome. Then LATERAL_2 will again call next() on right side and get all the output batches one by one. It will again perform the join and populate the output batch.
    - If in this case output batch still has some space left after consuming second left batch and corresponding right output batches, then it will not

call next() on left side. This will be decided based on last outcome from left side which was EMIT. With this output batch LATERAL_2 will also return outcome of EMIT to LATERAL_1.

- If in case when output batch gets filled in between before consuming all the rows on left side and corresponding output batches on right side, then LATERAL_2 will return the output batch but with outcome OK. On the subsequent next() call on LATERAL_2 it will process the leftover rows on left batch and join with corresponding output batches on right. Populate the output batch with join result, and if all the records fit's in one batch then it will send that downstream with outcome EMIT.

LATERAL_1 will call next() on LATERAL_2 until it see's EMIT outcome from right side. For all the batches with OK outcome from right, LATERAL_1 will perform the join and populate the output batches. On seeing EMIT from right side, it will mark current left batch as processed and will work on next left batch and keep populating the output batch until it gets filled.

3. ***Multiple UNNEST at same level:***

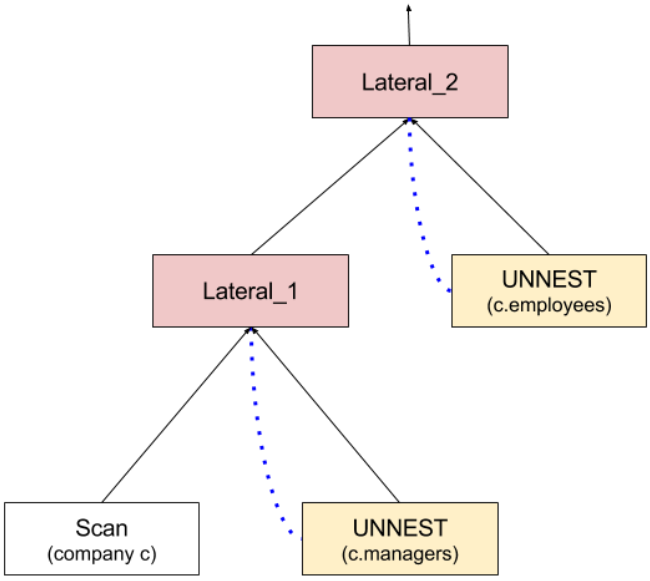


Figure 4: Plan for multiple arrays being UNNESTed. Data is same as for figure 1, but we are unnesting managers array and then employees array for each customer record.

In the case of multiple unnest for 2 different arrays in a record like managers and Employees, the plan will look like above. In this case also the behavior of populating output batches along with EMIT outcome will be similar to the case with single Lateral and UNNEST since LATERAL_2 will never see EMIT from it's left side.

4. ***Single Lateral and UNNEST but with Agg operator***

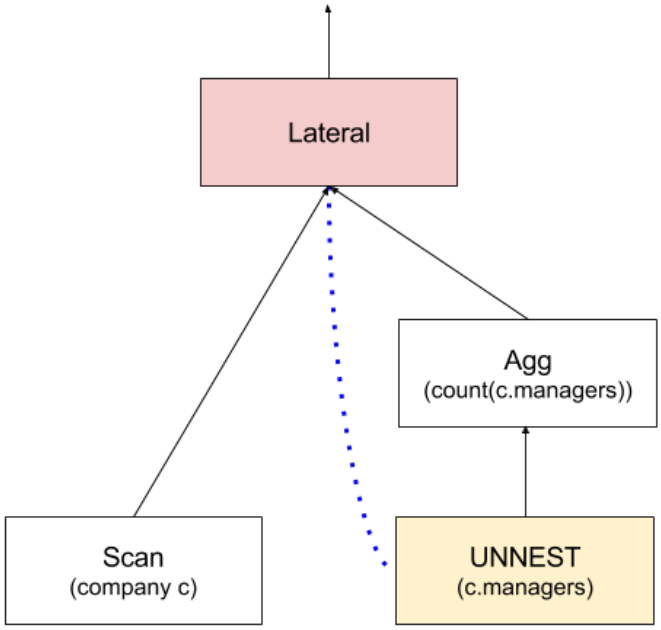


Figure 5: Plan for multiple arrays being UNNESTed. Data is same as for figure 1, but we are unnesting managers array and then employees array for each customer record.

This case is also similar to case 1. The difference is that with Agg in the right branch between LATERAL and UNNEST, for each output batch produced by UNNEST with OK outcome Agg will keep buffering and applying Agg function on it. The Agg key will contain the RowId column along with the Agg key chosen in the query. When UNNEST produce a batch with EMIT outcome, the Agg will produce an output batch with EMIT outcome. LATERAL on receiving the batch will produce output record and start filling the output batch. LATERAL will keep buffering all the output's until the batch is filled and will send it downstream after that along with OK outcome. So for the operator's between LATERAL and UNNEST on right branch, their Agg/Sort or Filter condition will operate on key which will have RowId as the first expression in it alongwith query defined key.

4.2.2 Memory Layout of Value vector's in repeated type and primitive type:

For the sake of discussion let's consider only 1 complex type i.e. array. In Drill's Value Vector terminology Array<T> is defined as RepeatedValueVector<T> where T can be primitive, complex type or repeated type. Again for discussion in this document consider T as a primitive type INT4. The memory layout of actual data inside RepeatedValueVector<INT4> is same as in ValueVector<INT4> (i.e. Value Vector of primitive type INT4). Actual representations are as below:

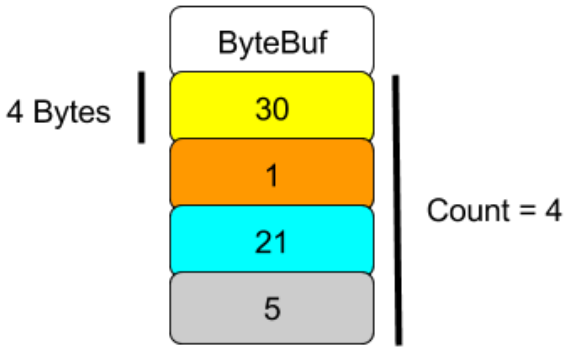


Figure 6: Diagram representing Bytebuf representation inside ValueVector for data column of form: {30, 1, 21, 5}

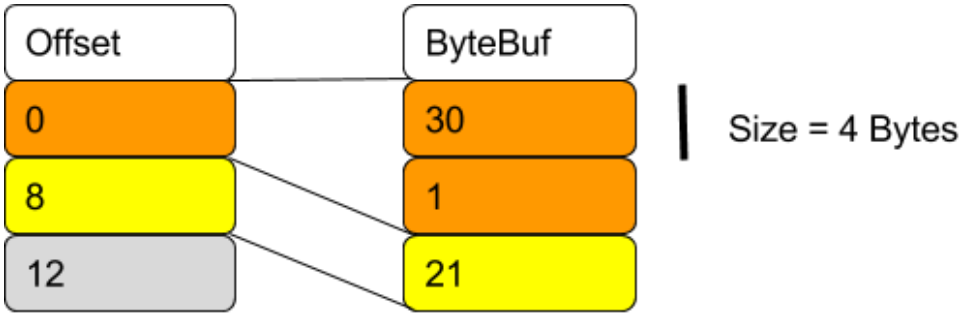


Figure 7: Diagram representing Bytebuf representation inside ValueVector for data column of form: {[31,20], [11]}

From both the diagram above we can see that actual data is stored serially in contiguous memory location referenced by DrillBuf inside the value vector. UNNEST is the operation of converting 1 row of input array type column to multiple rows of output column. Each row in the output column is an element of the array in input row. Unnest will also produce an implicit column for RowId. For example: If the first row of array column has 2 elements, then UNNEST output batch will produce 2 output elements for first row and RowId of 1 will be associated with both the output elements in the output batch. Thus the actual memory used by a column after UNNEST is greater than or equal to memory consumed before Unnest in terms of Value Vector. The offset vector of array type column will not be needed and replaced by additional RowId ValueVector after Unnest.

Rule 6: *Memory consumed by output Value Vectors of UNNEST will be greater than or equal to input Value Vector of UNNEST.*

4.2.3 Execution Flow:

Based on the above identified rules and description let's dive into the detailed design on how the execution flow will work.

- Operator's like Filter, Project, AGG, Sort and other possible ones will support **EMIT.** On seeing **EMIT** all the operators will unblock (if at all) and produce an output batch with the data they have so far and pass this IterOutcome downstream.
- Since processing is done at batch level by LATERAL and UNNEST, there is an implicit column for RowId which provides mapping between left batch row and all it's corresponding right side output rows so that cross-join can be performed correctly. Earlier the implementation was done by processing one row at a time which didn't meet the expected performance benefits.
- Let's say we have an index in LATERAL named **joinIndex** which is initialized to -1. This index is set to 0 for each new batch which UNNEST has to process. Also assume the column which UNNEST will process is named **columnToUnnest**.
- Let's say left side incoming of Lateral is set as incoming of UNNEST. Then **joinIndex** is incremented to 0. When **next()** is called on right branch of LATERAL which eventually call's **next()** of UNNEST, then UNNEST will call **getRecordIndex() on LATERAL** which will provide the record index of current left batch. If joinIndex is 0 that means Unnest is working on a new batch and checks for schema change, otherwise it means Unnest is working on same left batch as previously. Inside getRecordIndex() it does following:

```
int getRecordIndex() {
        assert (joinIndex < incoming.getRecordCount())
        return joinIndex;
}
```

- UNNEST will process the record in **columnToUnnest** and generate the output batch. It will return the **IterOutcome.EMIT** with the last output batch, which marks the end of all output generated for the current left batch and guarantees there won't be any record for current left batch inside next output record batch.
- Based on restriction on number of records allowed per Record Batch, the output of UNNEST for current left batch can span multiple batches. But **EMIT** will still determine the end of output from UNNEST for current left batch. If there are multiple output

batches produced from UNNEST for current left batch, then only last output batch will be sent with EMIT outcome. All others will be sent with OK outcome.

- Depending upon the operator's present downstream of UNNEST graph, different things can happen. Let's consider all of them individually:
    - If there is Project then it will process the Output Record Batch from UNNEST and let it pass to the LATERAL.
    - If there is Filter then it will filter out certain or all the rows in the output Record Batch and pass it to the LATERAL. In this case LATERAL should take care of handling JOIN with empty batches.
        - If UNNEST produces at least one record then LATERAL will JOIN left and right with that record and pass the output downstream. It will set the ***joinIndex*** with the last RowId processed from right side batch. If ***joinIndex*** < ***incoming.valueCount*** then it calls next() on the right side. If ***joinIndex == incoming.valueCount*** that means it has processed all the rows in incoming record batch and will call next() on the left side to get more incoming records.
        - Otherwise if UNNEST produces an Empty Batch then current batch in incoming of LATERAL will be marked as processed and will set the ***joinIndex*** to -1 again. Left side will be populated in the output batch iff join type is Left Join. Then next() will be called again on the left side branch.
    - If there is Streaming Agg then it will aggregate output from UNNEST for current left batch and on seeing IterOutcome.EMIT it will produce an output Batch and pass this IterOutcome downstream. LATERAL sees IterOutcome.EMIT it does join with record in current right incoming (if any) and increments its ***joinIndex*** based on RowId processed from right side**.**
    - If there is SORT then it will block unless it has seen EMIT from UNNEST and will start sending the Output Batches downstream along with EMIT outcome. Then upstream will behave similar as in the case of Streaming Agg.
- Based on the input from the left and right side, LATERAL will perform the implicit join and send the output downstream when batch is filled. There can be 2 cases to consider here:
    - LATERAL with UNNEST only on right branch and no UNNEST on left branch. In this case LATERAL will consume the EMIT outcome and not let it flow downstream along with output batch.

    - In case of multilevel LATERAL join there will be UNNEST on both left and right side of the operator tree. LATERAL will consume the EMIT from right side but will pass-through the EMIT from left side, which will eventually be consumed by top-level LATERAL.

  So EMIT outcome will never past top level LATERAL in the call graph which helps us to avoid making exchange operators aware about this outcome and handle it.
- For cases where there are UNNESTing required for multiple arrays at the same level in a record, then the plan will be generated with multiple LATERAL and UNNEST pair stacked on top of each other.

4.2.4 Handling OK_NEW_SCHEMA with UNNEST and LATERAL:

> ***Rule 7:*** *Schema change happens at Batch boundary NOT at Record boundary for UNNEST and LATERAL.*

All the operators in Drill currently participate in Schema exploration phase as part of first **next()** call in the operator tree. Also if during execution phase anytime a new Schema is observed compared to what is already known, then **OK_NEW_SCHEMA** outcome is used to notify downstream operators. There are few operators like Agg which doesn't handle schema change during execution phase and fails the query. For **UNNEST and LATERAL** we have to handle this case. UNNEST and LATERAL both needs some special handling for this case too which are described as below:

- **Build Schema Phase:**

  LATERAL like other Binary operators needs to participate in **BuildSchema** phase for fast schema return and for UNNEST it will handle the BatchState.FIRST state like other Unary operators. In BuildSchema phase all the Blocking operators just return the schema related information downstream with empty batch. Below are the rules for BuildSchema or very FIRST record phase, considering example in ***Figure 1*****.**
    - **LATERAL** starts with BatchState.BUILD_SCHEMA and **UNNEST** starts with BatchState.FIRST as other binary and unary operator in Drill.
    - When LATERAL_1 get's the first incoming from left, it can be empty or batch with some rows and with IterOutcome.OK_NEW_SCHEMA. It then calls **next()** on right branch.
    - Ultimately UNNEST_1 next() gets called via LATERAL_2.left.next(). Seeing the state as BatchState.FIRST, UNNEST_1 just creates an empty batch with correct schema for its **columnToUnnest** and sends that downstream with OK_NEW_SCHEMA.

- Then LATERAL_2 calls right.next() which call's next() of UNNEST_2. UNNEST_2 does the same thing as UNNEST_1 seeing it as BatchState.FIRST.
- LATERAL_2 combines the OK_NEW_SCHEMA from both left() and right() branches and creates an empty output batch with just schema and send downstream along with OK_NEW_SCHEMA.
- Finally LATERAL_1 receives OK_NEW_SCHEMA from both left and right branch and creates an empty output batch with new schema information and sends it downstream with OK_NEW_SCHEMA.
- Note that if there was any data which was received by LATERAL_1 from left side it buffers it and sends empty batch in previous step. As part of subsequent next() calls this buffered data is processed and pushed downstream. After every operator processes this fast schema path their state's are changed from BUILD_SCHEMA to FIRST and from FIRST to NOT_FIRST.

- **OK_NEW_SCHEMA after BuildSchema phase:**

  After the fast schema path there still can be cases when OK_NEW_SCHEMA is received by left side of LATERAL with zero or more records in the batch. Details on how to handle this case is described in Section 5.

  Similarly for UNNEST the rule to determine the change in SCHEMA during execution phase is to actually check for SCHEMA change for it's **columnToUnnest**, for 1st record of every new incoming batch from LATERAL **NOT** just on the basis of OK_NEW_SCHEMA outcome seen by LATERAL with it's left incoming.

  - UNNEST will send empty batch along with **OK_NEW_SCHEMA** only with 1st output record batch for **all incoming** from LATERAL in **which it sees a schema change** for **columnToUnnest**.
  - To determine OK_NEW_SCHEMA phase, UNNEST checks IterOutcome when getRecordIndex() == 0. Something like below:

```
inputRecordIndex = LateralJoin.getRecordIndex();
if(inputRecordIndex == 0) {
    // Check for SCHEMA change in columnToUnnest.
    // If schema changed then UNNEST will return OK_NEW_SCHEMA with
    // empty o/p batch
}
```

5 Implementation Details

From implementation perspective changes needs to be done both in planner and in execution phase. Based on the Design overview following things are determined:

5.1 Algorithms (and Data Structures)

- **UNNEST** will behave mostly like FLATTEN. It will need to have reference to corresponding **LATERAL** operator from which it will get it's incoming. UNNEST always process one batch at a time from incoming and sends the output downstream followed by **OK** outcome. Only the last output batch will be sent downstream with **EMIT** outcome. UNNEST along with the flattened array column also has the RowId column in its output batch. With respect to other IterOutcomes so far we have not encountered a scenario where UNNEST will produce IterOutcome.NOT_YET / OUT_OF_MEMORY / STOP. However it can still produce OK / NONE / OK_NEW_SCHEMA other than EMIT. For handling OK_NEW_SCHEMA please refer Section 4.2.4. Upon seeing NONE, UNNEST should return this state to the downstream operator's indicating there is no more data. Although UNNEST will never see NONE since LATERAL on processing the last record in last incoming will not call next() on right branch of UNNEST and can just send NONE downstream, but this will also help to unit test UNNEST.

- For **LATERAL** the pseudo algorithm will look like below:

```
rightOutput: VectorContainer
outgoing: VectorContainer

while(true) {
        Incoming: left.next();
        foreach row in Incoming:
                rightOutput: unnest.next()
                outgoing: crossJoin row with rightOutput
}
```

Few things that needs to be considered in detail for LATERAL:

1) Lateral keeps calling next() on left side until it receives a record batch with some records or it see's either of OK_NEW_SCHEMA / EMIT / NONE / STOP / OOM.

Rule 8: *LATERAL will call next() on left side branch, unless it receives a record batch with >0 records or see outcomes such as OK_NEW_SCHEMA, EMIT, NONE, STOP or OOM.*

2) If left IterOutcome is NONE, that means there is no batch associated with it. In this case LATERAL will just pass NONE to the downstream operator without calling next() on right branch

3) LATERAL should also be able to handle the empty Record Batch from right side correctly. There are cases when empty record batch is received by LATERAL with IterOutcome of **EMIT** and **OK_NEW_SCHEMA**, it has to differentiate these 2 cases and increment the **joinIndex** correctly. Since empty batch with EMIT can be result of Filter or other operators in the tree between UNNEST and LATERAL whereas **OK_NEW_SCHEMA** always produces an empty record batch as first output followed by actual data for the current row along with EMIT. So the rule for LATERAL will be such that for each right side it will call next() on it unless it see's EMIT before increasing the **joinIndex**.

Rule 9: *LATERAL will call next() on right side branch, unless it see's EMIT, before increasing the* ***joinIndex*** *which mark's that current row is fully processed by the right side child.*

4) LATERAL can get left batch with IterOutcome of OK_NEW_SCHEMA with zero or more records.
    a) If it's with zero records, then it will not call next() either on left and right side. It will create container with new schema information for left and existing known schema information from right side and then send empty batch with new schema downstream along with OK_NEW_SCHEMA outcome.
    b) If it's with >0 records then it will call next() on right side batch. Then in right side there can again be a LATERAL as in figure 1 in which case let's discuss how that will work. When called next() on LATERAL_2, it calls next() on UNNEST_1. UNNEST_1 check for schema change:
        i) If there is schema change for UNNEST_1 then it will send empty batch with OK_NEW_SCHEMA outcome downstream. LATERAL_2 on receiving OK_NEW_SCHEMA from UNNEST_1 with empty batch will behave as discussed in 4)a
        ii) If there is no schema change for UNNEST_1 then it will send a batch with zero or more records with outcome OK/EMIT. LATERAL_2 on receiving batch with zero record and OK, follows step 1. If it sees batch with zero record but with EMIT then it will just return empty batch with EMIT outcome to LATERAL_1. If there is a batch with some records with either OK/EMIT, then it calls next() on right side.

iii) UNNEST_2 then check for schema change and produce OK_NEW_SCHEMA with empty batch in case there is schema change for it's column. Then LATERAL_2 on receiving OK_NEW_SCHEMA from right side just re-creates the container with empty batch and send downstream with OK_NEW_SCHEMA. LATERAL_1 also does the same thing.

Rule 10: *LATERAL will never call right side if left batch is empty and with any outcome. If left batch outcome is OK_NEW_SCHEMA and empty batch it just recreates schema for output batch based on last known right side schema and new left schema and send OK_NEW_SCHEMA downstream.*

Rule 11: *LATERAL on receiving OK_NEW_SCHEMA from either of left and right side with zero records re-creates the output batch schema and send empty batch downstream with OK_NEW_SCHEMA.*

5.2 Multilevel LATERAL and Multiple UNNEST at same level:

With multiple UNNEST and LATERAL there can be 2 basic cases:

- **Multiple LATERAL at different levels:** This is generated in plan when we have repeated type nested at multiple levels. Like we have Customer table, each row of which has array of Orders and each record of Order has array of Items. Something like *Customer{cust_id, cust_name, array<Order{order_id, cust_id, order_price, array<Items{item_id, order_id, item_price}>}>}*. If we want to denormalize this data then the query plan will look something like below:

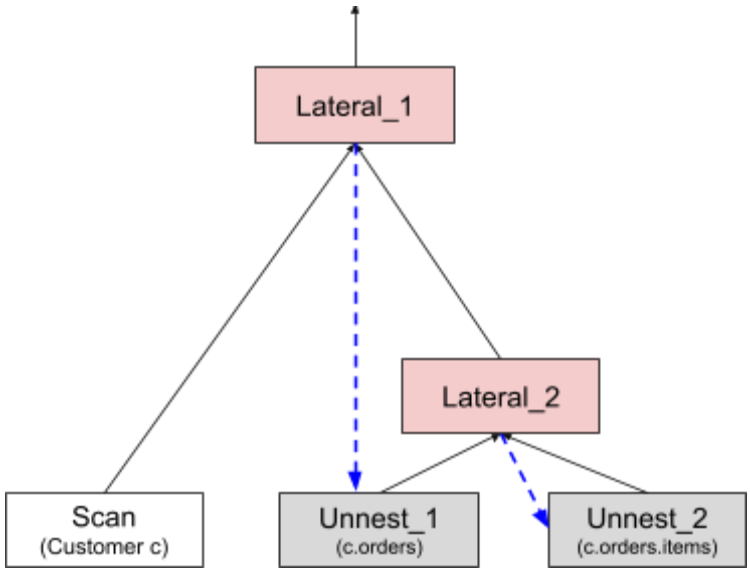
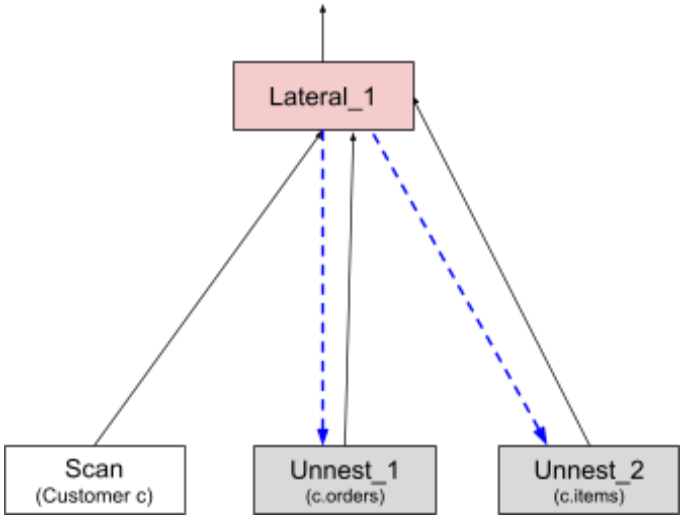


Figure 8: Showing example for Multilevel lateral

- **Multiple UNNEST at same level:** This is generated in plan when we have multiple repeated type at same level. Like we have Customer table, each row of which has array of Orders and array of Items. Something like *Customer{cust_id, cust_name, array<Order{order_id, cust_id, order_price}>, array<Items{item_id, order_id, item_price}>}*. If we want to denormalize this data then we have 2 options to perform this, the query plan's will look something like below:

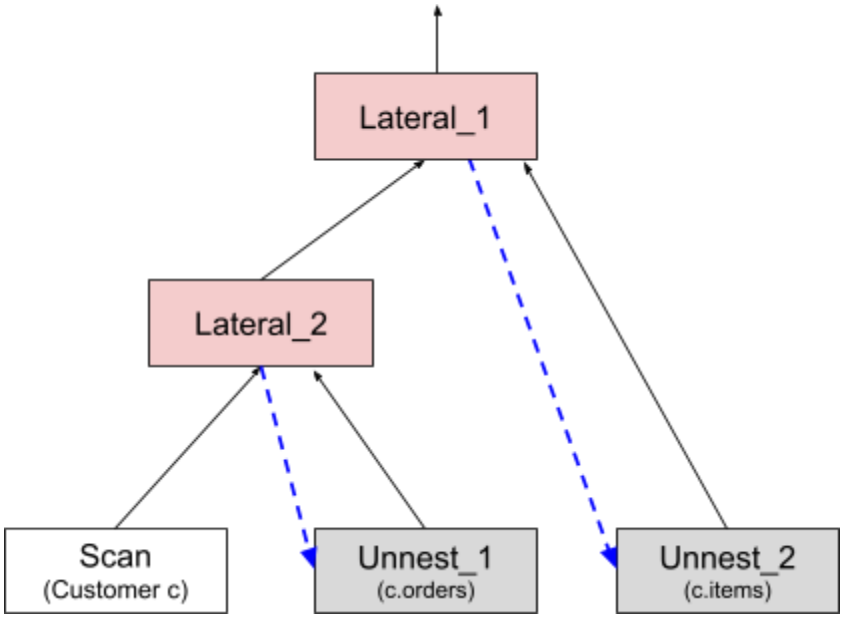
    - **Option 1:** LATERAL with N right inputs, one for each UNNEST.
        - **Advantages**:
            - Less number of copy for columns which doesn't require UNNESTing (let's call them C_L), since there is only 1 LATERAL. There will be only 1 copy for those columns in the final output batch. If there are 20 elements in orders array and 10 elements in item arrays then total copy for C_L columns are 200 per input row.
        - **Disadvantages**:
            - For cross-product all the input from N right sides needs to be held in memory. This can be difficult in low memory cases.
            - Spilling can really affect the performance since it involves the cost of writing data to disk and then back to memory. Depending upon the data and available memory, number of spilling will vary and hence can be very slow and also means more copy for UNNESted data.

- Cross-product can be implemented in complex ways to do product of 2 input from right, store the result (spill if need be) and then perform cross-product with 3rd input and so on. So if there are N right side inputs - {N1, N2, N3,.., Nn} and left side being L1 then
  $Output_{Lateral}$ = *LateralJoin(L1, N1, N2, N3,.., Nn}*
  = *LateralJoin(L1, LateralJoin(N1,N2,N3..,Nn))*
  = *LateralJoin(L1, LateralJoin(N1,N2),N3,...,Nn)*
  = *LateralJoin(L1, LateralJoin(O_{N1N2},N3),N4,...,Nn)*
  Where O_{N1N2} = *LateralJoin(N1,N2).*
- May need extra changes on Calcite side to support N-ary operator type.



- **Option 2:** Multiple UNNEST at same level replaced by consecutive pairs of LATERAL and UNNEST.
  - **Advantages**:
    - No need to worry about memory constraint w.r.t cartesian product. Since at each level LATERAL will perform cartesian product between input from left and right, copy it to output and send it downstream once output batch is full. It will work in an streaming manner.
    - Easier to implement since complexity to deal with spilling during cartesian product is not there anymore. This prevent writing data to disk and back to memory, which means less copy of UNNESted data.
    - Already supported by Calcite.
  - **Disadvantages**:
    - There is increase in number of copies from input to output vector for columns in LATERAL incoming which

are not being UNNESted. Based on example in option 1 there will total of (20 + 200) copies of C_L columns per row across 2 levels of LATERAL.



5.3 APIs and Protocols

Currently no change in the public API's and RPC protocol of Drill.

5.4 Performance

5.3.1: Performance versus relational

The effect of Lateral Join and Unnest is to produce denormalized output from a nested record. As pointed out in the High Level Design, Lateral join and Unnest have improved performance compared to Flatten because their use reduces the number of scans needed and network transfer of huge amount of data.

In addition, memory consumed by Unnest is much smaller than that consumed by Flatten. To compare the performance benefit of LATERAL and UNNEST, based on the TPCH benchmark, we need to compare the relational representation of TPCH data with a nested representation of the data. For instance, the customer - order - lineitem relations in the TPCH data can be represented as a nested type :

Customer{cust_id, cust_name, ***array[order]****}*
Where order {order_id, order_price, order_desc, cust_id, ***array[lineitem]****}*

Where lineitem {item_id, item_name, order_id, item_price}

Now, we can rewrite the TPCH queries and compare the relative performance.

5.3.2: Performance versus Flatten

Given the nested representation we can also compare the performance of the same TPCH queries written using Flatten versus queries written using LATERAL and UNNEST.

5.5 Error and Failure handling

TBD

5.6 Deployment

TBD

5.7 Memory management

With the nested data representation, we will probably see an increase in the size of the data compared to the relational model. In addition, denormalizing the data would increase the amount of memory used, again compared to the relational model. However, because the processing does not need to maintain a table in memory (as in the left side of a join), we would expect to see better memory usage than the relational case. We should see little to no increase in memory compared to queries using Flatten.

There can be cases where multiple UNNEST are present at same level in which case LATERAL has to perform the cross-product of all the output's from UNNEST. This cross-product can lead to increase in memory usage and needs some thinking from spilling perspective. This also depends upon if n-ary case is supported by Calcite or not.

5.8 Availability Implications

TBD

5.9 Scalability Issues

This design improves the scalability for the queries which are written using FLATTEN and can be rewritten using LATERAL and UNNEST.

5.10 Backward Compatibility

With introduction of new IterOutcome EMIT most of the existing operators will need to be changed to handle this new state. We need to ensure after change all the operators work as before in absence of LATERAL and UNNEST in the query.

5.11 Security and Authentication impact

None

5.12 UI changes

Potential UI changes includes:

- Parsing and display of new plan including LATERAL and UNNEST keyword.
- Showing plan with multiple UNNEST at same level.
- Showing plan with multi-level LATERAL.
- In physical plan display, correctly showing links between UNNEST and its corresponding LATERAL operator.

5.13 Options and metrics

TBD

5.14 Debugging

TBD

5.15 Testing implications

Adding corner cases in this section which are determined during design discussions. We should consider testing the following cases -

- LATERAL with one UNNEST; with no other operators between LATERAL and UNNEST
- LATERAL with multiple UNNEST; with no other operators between LATERAL and UNNEST
- LATERAL with one UNNEST;
    - with filter between LATERAL and UNNEST. Consider case where filter removes all records and produces empty batch.
    - with blocking operators between LATERAL and UNNEST. For blocking operators include test cases for aggregator and join separately.
- Multiple levels of LATERAL
- Test with data such that LATERAL incoming only contains 1 record and later after UNNESTING this fills entire output batch of UNNEST. Test for correctness and memory pressure on LATERAL Join in this case.
- Test with data such that when LATERAL join is doing cross-product **of all the UNNEST at same level** and output of Cross-Product cannot stay within a single Batch boundary. Is LATERAL able to perform the JOIN based on Cross-Product output obtained so far and on future **next()** call to LATERAL it can perform the remaining Cross-Product properly

before processing any new incoming. [NOT VALID WITH NEW OPTION FOR MULTIPLE UNNEST AT SAME LEVEL]

5.16 Tradeoffs and Limitations

5.16.1 Tradeoffs

The main issue with implementing LATERAL Join and UNNEST is that of coordinating between the two operators. In order for output records to be sync'd, LATERAL needs to know that the pipeline on the RHS containing the UNNEST operator has completed processing of the current row. Additionally, the RHS can contain an aggregator or some other blocking operator that needs to know when to unblock and pass data downstream to the LATERAL Join. To resolve this, we considered two approaches. For details please refer to Lateral Join Working Notes [2].

- Introduce an implicit Row_Id for the record being processed by the LATERAL. The Row_Id is duplicated by UNNEST for every record it outputs. The Row_Id would be considered a group by column and blocking operators can unblock when the Row_Id changes.
    - **Advantages**: No changes to existing operators
    - **Disadvantages**:
        - a) Additional memory consumed by the Row_Id.
        - b) Can lead to a deadlock.
- Introduce a new state to signify that the unnest operation has consumed the row entirely. Downstream operators will unblock when this state occurs, and LATERAL will proceed.
    - **Advantages** :
        - a) Uses no additional memory
        - b) Cannot deadlock
        - c) Allows Unnest to process more than one row at a time (not considered in this design).
    - **Disadvantages** : Requires a new state that must be handled by all operators.

5.16.2 Limitations:

- With first milestone we will not support couple of functionalities:
    - Only Lateral Join existing in the query without UNNEST.
    - LATERAL will not take any join conditions.
    - Any other table function other than UNNEST on right side of LATERAL.
    - Capability to perform join between 2 column in left incoming of LATERAL without reading data twice. Example:

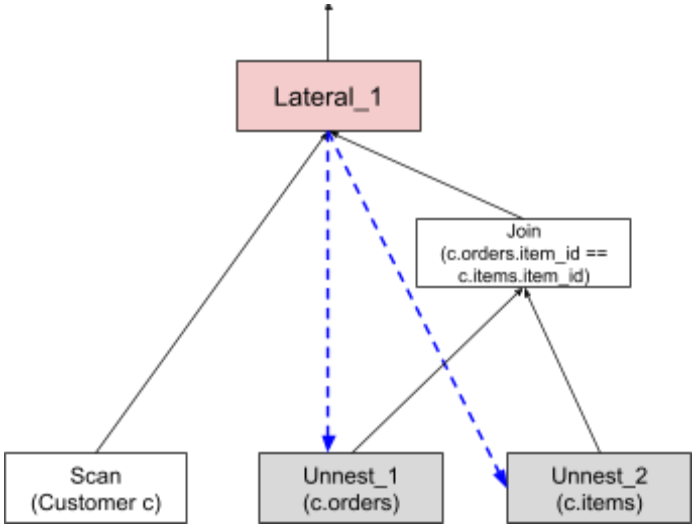
Cases when regular JOIN exist's in the right side of LATERAL. When JOIN can exist for 2 UNNEST at same level. Let's say customer data has *cust_id, List<Orders>, List<Items>* as the schema. Order internally has *order_id, order_amount and item_id* whereas Items internally has *item_id, item_name*. The subquery to LATERAL can have JOIN on *item_id* from Orders unnesting and *item_id* from Item unnesting. The plan will look like below:

SELECT c.cust_name, orderCost, itemName FROM customer c

LATERAL *(SELECT O.order_amount as orderCost, I.item_name as itemName*

FROM ***UNNEST(c.orders)*** *O,* ***UNNEST(c.items) I***

JOIN O.item_id == I.item_id)*;*



7 Open items

TBD

8 Notes

See [2].

9 References

[1] “Lateral Joins and Unnest.” *Google Docs*, Google,

Google Doc Link

[2] “Working Notes Lateral & Unnest.” *Google Docs*, Google,

Google Doc Link

[3] “7.2. Table Expressions.” *PostgreSQL*,

www.postgresql.org/docs/9.4/static/queries-table-expressions.html.

“9.18. Array Functions and Operators.” *PostgreSQL*,

www.postgresql.org/docs/9.2/static/functions-array.html.

[4] “Complex Type Planning Detailed Design.” *Google Docs*, Google,
Google Doc Link

10 Document History

Date	Author	Version	Description
2018-01-25	Chunhui Shi, Parth S. Chandra, Sorabh Hamirwasia	0.1	Initial Draft
2018-02-08	Sorabh Hamirwasia	0.2	Updated section ***4.2.1 EMIT Processing Rule***, ***5.1 Algorithms (and Data Structures)*** ***Lateral Section.*** Also updated the document to reflect the changes in multiple UNNEST at same level, since it will be handled using binary operator instead of n-ary.