Push-Based Execution in DuckDB

Mark Raasveldt
DBMS transform SQL into query plans

Query plans contain operators

Operators need to be executed

How?

SELECT SUM(l_extendedprice) FROM lineitem
JOIN orders ON (l_orderkey=o_orderkey)
GROUP BY l_returnflag;
Two paradigms: Pull-based and push-based

**Pull-based**
- Pull data from other operators when required

**Push-based**
- Push data into operator when data is available
DuckDB initially used a pull-based execution model: "Vector Volcano". Every operator implements `GetChunk`. Query starts by calling `GetChunk` on the root. Nodes recursively call `GetChunk` on children.
void Projection::GetChunk(DataChunk &result) {
    // get the next chunk from the child
    child->GetChunk(child_chunk);
    if (child_chunk.size() == 0) {
        return;
    }
    // execute expressions
    executor.Execute(child_chunk, result);
}
In this model:

- Single-threaded execution is straightforward
- Multi-threaded not so much...

How do you make operators parallelism-aware?
Pipeline Parallelism

- Exchange operator
- Optimiser splits query plan into partitions
- Partitions can be executed independently

**Problems:**
- Load imbalance issues
- Plan explosion
- Added materialization costs
Pipeline Parallelism

- Morsel-Driven Parallelism
- Individual operators are parallelism-aware
- Query is divided into pipelines
- Pipelines are executed in parallel

Viktor Leis et al.
SELECT SUM(l_extendedprice) 
FROM lineitem 
JOIN orders 
ON (l_orderkey=o_orderkey) 
GROUP BY l_returnflag;
Contestation happens **at endpoints**

**Source:** Scan of orders

**Sink:** HT build of join

Use **parallelism-aware** operators at endpoints

Other operators (HT probe, projection, filter, etc...) don't need to be aware
- **Sink Interface**

- Sinks can define **global** and **local** states

- **Sink** is called until all data is exhausted

- **Combine** is called (once per thread)

- **Finalize** is called (once)

```c
void Sink(
    ExecutionContext &context,
    GlobalSinkState &gstate,
    LocalSinkState &lstate,
    DataChunk &input);

void Combine(
    ExecutionContext &context,
    GlobalSinkState &gstate,
    LocalSinkState &lstate);

void Finalize(
    ClientContext &context,
    GlobalSinkState &gstate);
```
void HashJoin::Sink(DataChunk &input) {
    // build the hash table
    BuildHashTable(input);
}

void HashJoin::GetChunk(DataChunk &result) {
    // probe the hash table
    left_child->GetChunk(child_chunk);
    ProbeHashTable(child_chunk, result);
}
Pipelines are run by pulling from child of sink

After child is exhausted, call Combine/Finalize

Mix of push/pull: sink is push, rest is pull...

```c
void RunPipeline() {
    // fetch data from child of sink
    while (sink->child->GetChunk(child_chunk)) {
        sink->Sink(child_chunk, ...);
    }
    // finished: combine
    sink->Combine(...);
    if (all_threads_finished) {
        // all threads are finished: finalize sink
        sink->Finalize(...)
        ScheduleNextPipeline();
    }
}
```
How do we partition *Sources*?

Not as straightforward...

Sources are located at the bottom of the pipeline.
Pipeline Parallelism

- Set up a **tasks** in thread context
- Tasks define how the scan is partitioned
- Read those tasks in the `GetChunk`

```cpp
void TableScan::GetChunk() {
    // check if there is a task scheduled for this operator
    table.ScanTask(thread_context.tasks.find(this));
}
```
This mostly works

Problems:

- Data flow duplicated in every operator
- No clean interface for source parallelism
- How to parallelize UNION nodes?
- How to parallelize FULL/RIGHT outer joins?
- Scan Sharing?
- Async I/O?
Push-Based Execution
What is push-based execution?

Our previous model was **pull-based:**
- **GetChunk** called when an operator requires data

**Push-based** is the other way around
- Push data into operators

**Sink interface** is already push-based!
Push-Based moves data flow \textbf{out of operators}

- Data flow is handled in central location
- Simplifies implementation of operators
- But reduces flexibility!
Define **Operator** and **Source** interface

**Operator** processes data
- Projection, Filter, Hash Probe, ...

**Source** emits data
- Table scan, aggregate HT scan, ORDER BY scan, etc
Operator Interface

```cpp
OperatorResultType Execute(
    ExecutionContext &context,
    DataChunk &input,
    DataChunk &chunk,
    OperatorState &state);
```

**Execute** takes an input chunk, and outputs another chunk
Projection is straightforward

```cpp
void Projection::Execute(DataChunk &input, DataChunk &result) {
    executor.Execute(input, result);
}
```
Hash Probe seems straightforward…

```cpp
void HashJoin::Execute(DataChunk &input, DataChunk &result) {
    Probe(input, result);
}
```

How do we handle multiple matches per tuple?

1 input entry can lead to many output entries...

Operators need a way of signalling they are not done processing the input
**OperatorResultType** is used for this

```cpp
enum OperatorResultType {
    NEED_MORE_INPUT,
    HAVE_MORE_OUTPUT,
    FINISHED
};
```

- **NEED_MORE_INPUT**: Operator will be called with a new input chunk
- **HAVE_MORE_OUTPUT**: Operator will be called with the same input chunk
- **FINISHED**: The operator will not be called again, terminates the pipeline
enum OperatorResultType {
    NEED_MORE_INPUT,
    HAVE_MORE_OUTPUT,
    FINISHED
};

- **FINISHED** required to interrupt execution
- Happens naturally in a pull-based model
- e.g. **LIMIT** in pull-based simply stops pulling
- In push-based, we need to signal to the execution loop that we finished early
### Source Interface

- Similar to **Sink** interface
- **Global** and **local** states
- **GetData** is called until no more data remains
- Or pipeline is cancelled earlier

```c
void GetData(
    ExecutionContext &context,
    DataChunk &chunk,
    GlobalSourceState &gstate,
    LocalSourceState &lstate);
```
Pipeline Events
Pipeline Events

**UNION nodes**

**How do we execute unions?**
Pull-Based: Easy, we control the flow

```c
void Union::GetChunk(DataChunk &result) {
    if (!left_done) {
        left_child->GetChunk(result);
        if (result.size() > 0) {
            return;
        }
    }
    left_done = true;
}
right_child->GetChunk(result);
```

How do we do it push-based?
SELECT SUM(l_orderkey) FROM (SELECT * FROM lineitem UNION ALL SELECT * FROM lineitem)

- **Push-Based Union**
- Create two pipelines with same sink
  - Or more, if there are more unions
- **Sink::Finalize** only after all pipelines are done!
How do we handle the **Union** case here?
Split up **Pipeline** into **Events**

- **Pipeline Events**
  - Split up **Pipeline** into **Events**
  - Schedule those **Events**

- **Pipeline Events Diagram**
  - PIPELINE
  - PIPELINE EXECUTE
    - Executes main pipeline
  - PIPELINE FINISH
    - Call Sink::Finalize
  - PIPELINE COMPLETE
    - Mark pipeline as completed
SELECT SUM(l_orderkey) FROM
(
    SELECT *
    FROM lineitem
    UNION ALL
    SELECT *
    FROM lineitem
)

Now we can schedule multiple unions that will call Finalize once.
SELECT SUM(l_orderkey) FROM

( SELECT * FROM lineitem UNION ALL
SELECT * FROM lineitem UNION ALL
SELECT * FROM lineitem )

Can stack multiple unions
Full/Right Outer Joins have similar challenge

Three phases:

- Build HT
- Probe HT
- Scan HT (after ALL probing is finished)
SELECT \( \text{sum}(l\text{\_orderkey}) \) FROM lineitem
FULL OUTER JOIN orders ON (l\text{\_orderkey}=o\text{\_orderkey});
Pipeline Events

SELECT sum(l_orderkey) FROM lineitem
FULL OUTER JOIN orders ON (l_orderkey=o_orderkey)
FULL OUTER JOIN part ON (l_partkey=p_partkey);

Can stack multiple full/right outer joins
Sinks often have an expensive Finalize step

- e.g. **order by** - merging sorted segments

Need to be executed in parallel
SELECT * FROM lineitem ORDER BY l_orderkey;
Future Work
Scan Sharing (TODO)

- Detect pipelines that have the same source
- Scan once, sink into multiple pipelines

```
SELECT SUM(l_orderkey)
FROM lineitem
UNION ALL
SELECT AVG(l_orderkey)
FROM lineitem;
```

- Complicated by **projection & filter pushdown**

- Disjoint projections -> scan sharing not useful*

* Unless we are scanning a row-store
Async I/O (TODO)

- Current scans are still pull-based
- Fine for in-memory data
- Reading from disk/http/etc -> stall on read

Async I/O solves this by pushing I/O to background threads

When I/O completes, push data into pipeline
• **Hybrid Early/Late Materialization**

• Async I/O prefetches all required columns

• Early materialization

• Late materialization at times preferable

  • e.g. query with selective predicate on one column
This query selects a few rows

But reads all columns of entire lineitem table

**Early materialization**: read entire lineitem table

**Late materialization**: read l_orderkey column and few rows from other columns
Hybrid Early/Late Materialization

- Lazy vectors enable hybrid of early/late materialization
- When a vector is first used, fetch data from disk

- Conflicts with Async I/O!

**Potential solution**: Hybrid Async I/O

- Prefetch with async I/O
- Stop prefetching for a column if we detect column data is not required
That’s all folks!
Thanks for listening!