Query Engines: Push vs. Pull

26 Apr 2021

People talk a lot about "pull" vs. "push" based query engines, and it's pretty obvious what that means colloquially, but some of the details can be a bit hard to figure out.

Important people clearly have thought hard about this distinction, judging by this paragraph from Snowflake's Sigmod paper:

> Push-based execution refers to the fact that relational operators push their results to their downstream operators, rather than waiting for these operators to pull data (classic Volcano-style model). Push-based execution improves cache efficiency, because it removes control flow logic from tight loops. It also enables Snowflake to efficiently process DAG-shaped plans, as opposed to just trees, creating additional opportunities for sharing and pipelining of intermediate results.

And...that's all they really have to say on the matter. It leaves me with two major unanswered questions:

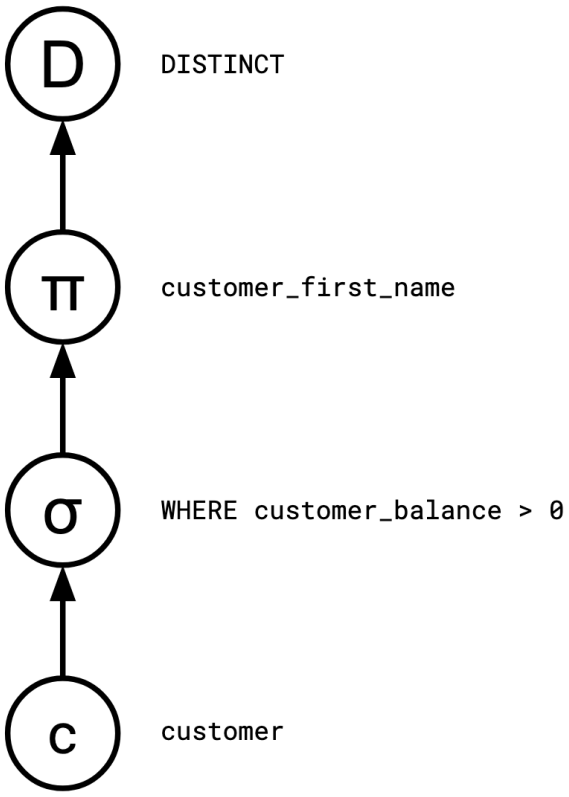
1. Why does a push-based system "enable Snowflake to efficiently process DAG-shaped plans" in a way not supported by a pull-based system, and who cares? (DAG stands for directed, acyclic graph.)
2. Why does this improve cache efficiency, what does it mean to "remove control flow logic from tight loops?"

In this post, we're going to talk about some of the philosophical differences between how pull and push based query engines work, and then talk about the practical differences of why you might prefer one over the other, guided by these questions we're trying to answer.

Consider this SQL query.

```
SELECT DISTINCT customer_first_name FROM customer WHERE customer_balance
```

Query planners typically compile a SQL query like this into a sequence of discrete *operators*:




```
Distinct
<- Map(customer_first_name)
<- Select(customer_balance > 0)
<- customer
```

In a *pull based* system, *consumers* drive the system. Each operator produces a row when asked for it: the user will ask the root node (`Distinct`) for a row, which will ask `Map` for a row, which will ask `Select` for a row, and so on.

In a *push based* system, the *producers* drive the system. Each operator, when it has some data, will tell its downstream operators about it. `customer`, being a base table in this query, will tell `Select` about all of its rows, which will cause it to tell `Map` about of *its* rows, and so on.

Let's start by building a super simple implementation of each kind of query engine.

A basic pull-based query engine

A pull-based query engine is also generally said to use the *Volcano* or *Iterator* model. This is the oldest and most well-known query execution model, and is named for the paper which standardized its conventions in 1994.

First, we'll start with a relation and a way to turn that into an *iterator*:

```
let customer = [
  { id: 1, firstName: "justin", balance: 10 },
  { id: 2, firstName: "sissel", balance: 0 },
  { id: 3, firstName: "justin", balance: -3 },
  { id: 4, firstName: "smudge", balance: 2 },
  { id: 5, firstName: "smudge", balance: 0 },
];

function* Scan(coll) {
  for (let x of coll) {
    yield x;
  }
}
```

Once we have our hands on an iterator, we can repeatedly ask it for its `next` element.

```
let iterator = Scan(customer);

console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

This outputs:

```
{ value: { id: 1, firstName: 'justin', balance: 10 }, done: false }
{ value: { id: 2, firstName: 'sissel', balance: 0 }, done: false }
{ value: { id: 3, firstName: 'justin', balance: -3 }, done: false }
{ value: { id: 4, firstName: 'smudge', balance: 2 }, done: false }
{ value: { id: 5, firstName: 'smudge', balance: 0 }, done: false }
{ value: undefined, done: true }
```

We can then create some operators to transform an iterator into another form.

```
function* Select(p, iter) {
  for (let x of iter) {
    if (p(x)) {
      yield x;
    }
  }
}

function* Map(f, iter) {
  for (let x of iter) {
    yield f(x);
  }
}

function* Distinct(iter) {
  let seen = new Set();
  for (let x of iter) {
    if (!seen.has(x)) {
      yield x;
      seen.add(x);
    }
  }
}
```

Then we can translate our original query:

```
SELECT DISTINCT customer_first_name FROM customer WHERE customer_balance
```

into this:

```
console.log([
  ...Distinct(
    Map(
      (c) => c.firstName,
      Select((c) => c.balance > 0, Scan(customer))
    )
  ),
]);
```

which outputs, as expected:

```
[ 'justin', 'smudge' ]
```

A basic push-based query engine

A push based query engine, sometimes known as the *Reactive*, *Observer*, *Stream*, or *callback hell* model, as you might expect, is like our previous example, but turned on its head.

Let's start by defining an appropriate `Scan` operator.

```
let customer = [
  { id: 1, firstName: "justin", balance: 10 },
  { id: 2, firstName: "sissel", balance: 0 },
  { id: 3, firstName: "justin", balance: -3 },
  { id: 4, firstName: "smudge", balance: 2 },
  { id: 5, firstName: "smudge", balance: 0 },
];

function Scan(relation, out) {
  for (r of relation) {
    out(r);
  }
}
```

We model "this operator tells a downstream operator" as a closure that it calls.

```
Scan(customer, (r) => console.log("row:", r));
```

Which outputs:

```
row: { id: 1, firstName: 'justin', balance: 10 }
row: { id: 2, firstName: 'sissel', balance: 0 }
row: { id: 3, firstName: 'justin', balance: -3 }
row: { id: 4, firstName: 'smudge', balance: 2 }
row: { id: 5, firstName: 'smudge', balance: 0 }
```

We can define the rest of our operators similarly:

```
function Select(p, out) {
  return (x) => {
    if (p(x)) out(x);
  };
```

```
}

function Map(f, out) {
  return (x) => {
    out(f(x));
  };
}

function Distinct(out) {
  let seen = new Set();
  return (x) => {
    if (!seen.has(x)) {
      seen.add(x);
      out(x);
    }
  };
}
```

Our query is now written:

```
let result = [];
Scan(
  customer,
  Select(
    (c) => c.balance > 0,
    Map(
      (c) => c.firstName,
      Distinct((r) => result.push(r))
    )
  )
);

console.log(result);
```

Outputting, as expected,

```
[ 'justin', 'smudge' ]
```

Comparison

In a pull-based system, the operators sit idle until someone asks them for a row. This means it's obvious how to get results out of the system: you

ask it for a row and it gives it to you. However, this also means that the behaviour of the system is very tightly coupled to its consumers; you do work if asked to and not otherwise.

In the push-based system, the system sits idle until someone tells it about a row. Thus, the work the system is doing and its consumption are decoupled.

You might have noticed that compared to our pull-based system, in our push-based system above we had to do a strange dance with creating a buffer (`result`) which we instructed the query to shove its results into. This is how push-based systems wind up feeling, they don't exist in relation to their designated consumer, they kind of just exist, and when things happen, they do stuff in response.

DAG, yo

Let's go back to our first major question:

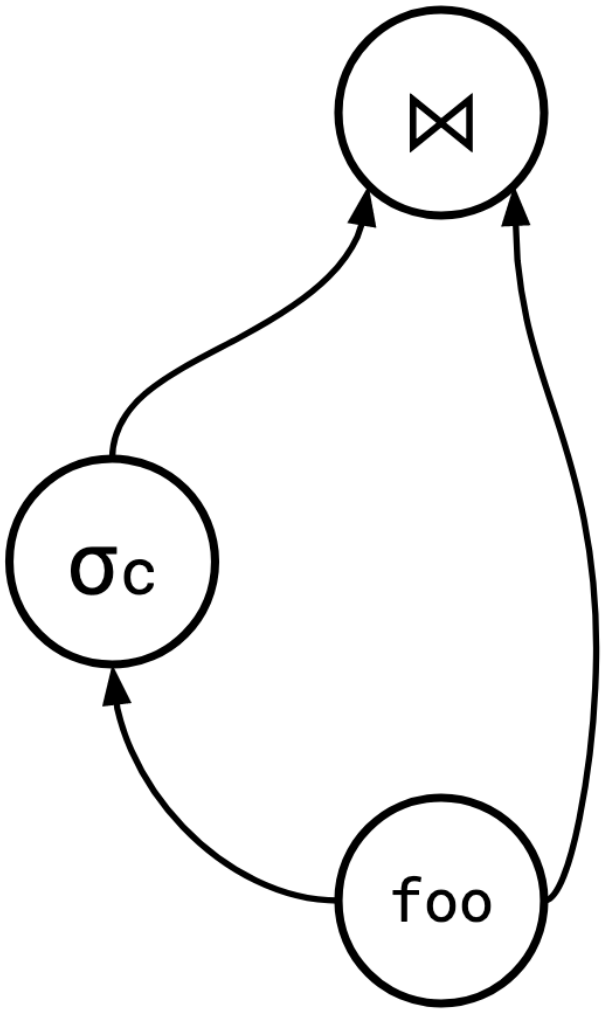
> Why does a push-based system "enable Snowflake to efficiently process DAG-shaped plans" in a way not supported by a pull-based system, and who cares?

By "DAG-shaped plans" they mean operators which output their data to multiple downstream operators. It turns out this is more useful than it sounds, even in the context of SQL, which we often think of as inherently tree-structured.

SQL has a construct called `WITH` that allows users to reference a result set multiple times in a query. This means the following query is valid SQL:

```
WITH foo as (<some complex query>)
SELECT * FROM
    (SELECT * FROM foo WHERE c) AS foo1
  JOIN
    foo AS foo2
  ON foo1.a = foo2.b
```

Which has a query plan that looks something like this:



Outside of this explicit example, a smart query planner can often make use of DAG-ness to reuse results. For instance, Jamie Brandon has a post describing a general method for decorrelating subqueries in SQL that makes extensive use of DAG query plans in order to be efficient. Because of all this, it's valuable to be able to handle these cases without simply duplicating a branch of the plan tree.

There are two main things that make this hard in a pull model: scheduling and lifetimes.

Scheduling

In a setting where every operator has exactly one output, when to run an operator to produce some output is obvious: when your consumer needs it. This becomes, at the very least, messier with multiple outputs, since "requests for rows" and "computations to produce rows" are no longer one-to-one.

By comparison, in a push-based system, scheduling of operators was never tied to their outputs in the first place, so losing that information makes no difference.

Lifetime

The other tricky thing with DAGs in a pull-based model is that an operator in such a system is at the mercy of its downstream operators: a row that might be read in the future by any of its consumers must be kept around (or must be able to be re-computed). One general solution to this is for an operator to buffer all of its rows that get output so you can re-hand them out, but introducing potentially unbounded buffering at every operator boundary is undesirable (but is, by necessity, what Postgres and CockroachDB do for `WITH` having multiple consumers).

This is not as much of a problem in a push-based system, because operators now drive when their consumers process a row, they can effectively *force* them to take ownership of a row and deal with it. In most cases, this will either result in some kind of essential buffering that would have been needed even in the absence of a DAG (say, for a `Distinct` or hash join operation), or will simply be processed and passed on immediately.

Cache Efficiency

Now let's talk about the second question.

> Why does this improve cache efficiency, what does it mean to "remove control flow logic from tight loops?"

First of all, the Snowflake paper cites a paper by Thomas Neumann in support of this claim. I don't really think this paper supports the claim in isolation though, if I had to sum up the paper, it's more like, "we would like to compile queries to machine code in service of improved cache efficiency, and to that end, a push-based paradigm is preferable." The paper is very interesting and I recommend you give it a read, but it seems to me that its conclusions don't really apply unless you're starting from a position of wanting to compile your queries using something like LLVM

(which, from some cursory research, it's not clear to me if Snowflake does).

In doing research for this section I found this paper by Shaikhha, Dashti, and Koch, that does a great job of highlighting some of the strengths and weaknesses of each model. In fact, they reference the Neumann paper:

> More recently, an operator chaining model has been proposed that shares the advantage of avoiding materialisation of intermediate results but which reverses the control flow; tuples are pushed forward from the source relations to the operator producing the final result. Recent papers seem to suggest that this push-model consistently leads to better query processing performance than the pull model, even though no direct, fair comparisons are provided.
>
> One of the main contributions of this paper is to debunk this myth. As we show, if compared fairly, push and pull based engines have very similar performance, with individual strengths and weaknesses, and neither is a clear winner. Push engines have in essence only been considered in the context of query compilation, conflating the potential advantages of the push paradigm with those of code inlining. To compare them fairly, one has to decouple these aspects.

They conclude that there's no clear winner here but observe that compiling a push-based query makes for simpler code. The main idea is that it turns out it's actually extremely easy to unroll a synchronous, push-based query into the equivalent code you'd write by hand. Take our query from before:

```
let result = [];
Scan(
  customer,
  Select(
    (c) => c.balance > 0,
    Map(
      (c) => c.firstName,
      Distinct((r) => result.push(r))
    )
  )
```

```
);

console.log(result);
```

This very naturally unrolls to:

```
let result = [];
let seen = new Set();
for (let c of customer) {
  if (c.balance > 0) {
    if (!seen.has(c.firstName)) {
      seen.add(c.firstName);
      result.push(c.firstName);
    }
  }
}

console.log(result);
```

If you try to unroll the equivalent pull-based query you'll find the resulting code is much less natural.

I think it's hard to come to any real conclusions about which is "better" based on this, and I think the most sensible thing is to make choices based on the needs of any particular query engine.

Considerations

Impedance Mismatch

One thing that can come up with these systems is a mismatch at their boundaries. Crossing a boundary from a pull system to a push system requires *polling* its state, and crossing a boundary from a push system to a pull system requires *materialization* of its state. Neither of these are dealbreakers, but both incur some cost.

This is why in a streaming system like Flink or Materialize you'll typically see push-based systems used: the inputs to such a system are *inherently* push-based, since you're listening to incoming Kafka streams, or something similar.

In a streaming setting, if you want your end consumer to actually be able to interact with the system in a pull-based way (say, by running queries against it when it needs to), you need to introduce some kind of materialization layer where you build an index out of the results.

Conversely, in a system that doesn't expose some kind of streaming/tailing mechanism, if you want to know when some data has changed, your only option will be to poll it periodically.

Algorithms

Some algorithms are simply not appropriate for use in a push system. As discussed in the Shaikhha paper: the merge join algorithm working is fundamentally based around the ability to traverse two iterators in lockstep, which is not practical in a push system where the consumer has little-to-no control.

Similarly, `LIMIT` operators can be problematic in the push model. Short of introducing bidirectional communication, or fusing the `LIMIT` to the underlying operator (which is not always possible), the producing operators cannot know they can stop doing work once their consumer has been satisfied. In a pull system this is not a problem, since the consumer can just stop asking for more results when it doesn't need any more.

Cycles

Without going into too much detail, having not just DAGs but full on cyclic graphs in either of these models is nontrivial, but the most well-known system that solved this is Naiad, a Timely Dataflow System, whose modern incarnation is Timely Dataflow. Both of these systems are push systems, and as with DAGs, many things just work better in a push model here.

Conclusion

Overwhelmingly introductory database materials focus on the iterator model, but modern systems, especially analytic ones, are starting to explore the push model more. As noted in the Shaikhha paper, it's hard to

find apples-to-apples comparisons, since a lot of the migration to push models are motivated by a desire to compile queries to lower level code and the benefits that come from that cloud the results.

Despite that, there are some quantitative differences that make each model appropriate in different scenarios and if you're interested in databases it's worth having a general idea of how they both work. In the future I'd like to go into more detail about how these systems are constructed and try to expose some of the magic that makes them work.