Iron Vector Benchmarks [August 2025]

This document describes a series of benchmarks comparing vanilla Apache Flink and Apache Flink with the Iron Vector accelerator. The Iron Vector accelerator is installed by adding several files to the /opt/flink/lib folder. No configuration or code changes were performed (with a few exceptions mentioned below).

At this stage, the Iron Vector accelerator only supports “stateless” Flink pipelines: no joins, aggregations, etc.

PS: I’d love to run a proper Nexmark test, but I just didn’t have time to set it up. Iron Vector currently supports q0, q1, q2, q10 and q21 queries.

Default Test Setup

Each Flink job had 2 TaskManagers, each with 2 task slots (total parallelism of 4).
Each TaskManager was a Kubernetes pod with 6 GB RAM, 1 CPU core.
Used Flink 1.19.1 with default configuration except:

- RocksDB state backend, slightly tuned (not relevant for these tests).
- Checkpointing of 1 minute.
- When Iron Vector was enabled:
    - taskmanager.memory.segment-size: "128 kb" (increased from the default 32 kb).

Input: Kafka topic with 80 partitions, 1+ TB of storage. Avro format with the Confluent Schema Registry. WarpStream on the broker side.

Each run usually took ~35 minutes; the first 5 minutes are ignored.

The “vector-blackhole” sink used in the test is a simple discarding / blackhole sink, but with the ability to configure parallelism.

The Kafka source throughput (bytes/s) and the average TaskManager CPU are the key tracked metrics.

Transformation Logic

The same transformation logic is used in all test scenarios:

```sql
SELECT id, block_number, log_index * 2 as res, CONCAT('block_hash_',
REGEXP_REPLACE(block_hash, '0x', '')) as res2
FROM KafkaTable
WHERE transaction_index > 40
```

It demonstrates Iron Vector support for:

- Projections
- Filters
- Expressions
- System functions

There were no specific reasons to choose this logic or these system functions (CONCAT, REGEXP_REPLACE). As we'll see below, most of the CPU time is spent on data serialization / deserialization, rather than actually performing computations.

Test A: Completely Fused Pipeline

This pipeline was defined using the following Flink SQL:

```sql
CREATE TABLE KafkaTable (
  id STRING NOT NULL,
  block_number BIGINT,
  block_hash STRING,
  transaction_hash STRING,
  transaction_index BIGINT,
  log_index BIGINT,
  address STRING,
  data STRING,
  topics STRING,
  block_timestamp BIGINT
) WITH (
  'connector' = 'kafka',
  'value.format' = 'avro-confluent',
  ...
)

CREATE TABLE BlackholeTable (
  id STRING,
  block_number BIGINT,
  res BIGINT,
  res2 STRING
) WITH (
  'connector' = 'vector-blackhole',
  'sink.parallelism' = '4'
)

INSERT INTO BlackholeTable
```

```
SELECT id, block_number, log_index * 2 as res, CONCAT('block_hash_',
REGEXP_REPLACE(block_hash, '0x', '')) as res2
FROM KafkaTable
WHERE transaction_index > 40
```

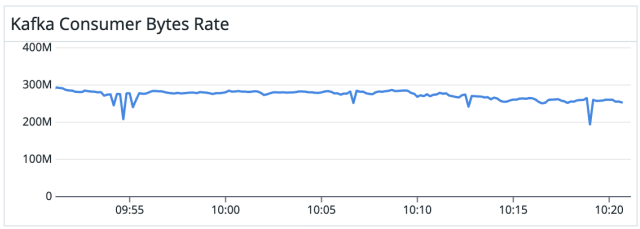
Since the sink parallelism is the same as the default parallelism, the resulting Flink pipeline is completely fused (represented as a single “box” in the Flink UI), and there is no network communication between the TaskManagers.

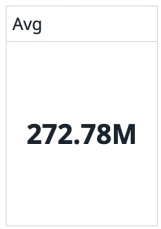
This is not a very realistic scenario: almost any real-world Flink pipeline involves an “exchange” between stages, e.g., when parallelism differs or a keyBy operator is used.

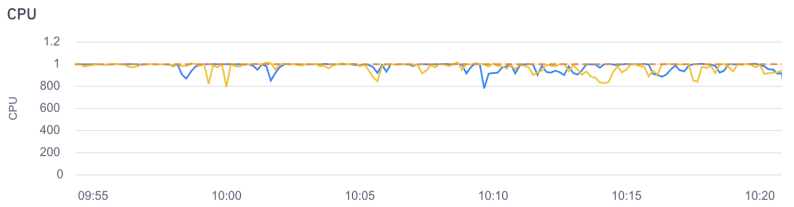
However, it’s still a useful scenario to understand what happens when the network exchange is not involved.

Frankly speaking, at this test, we primarily measure the source Avro deserialization and the impact that Iron Vector can have on it (and the overall pipeline that follows).

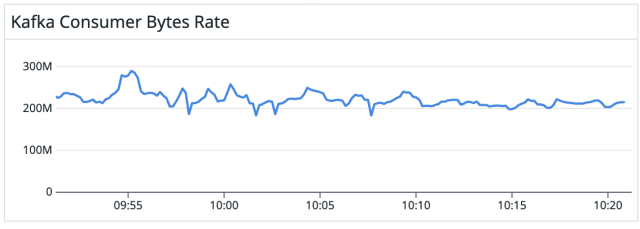
The initial baseline run **without** Iron Vector:

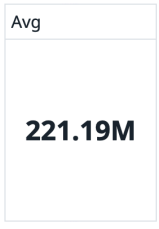


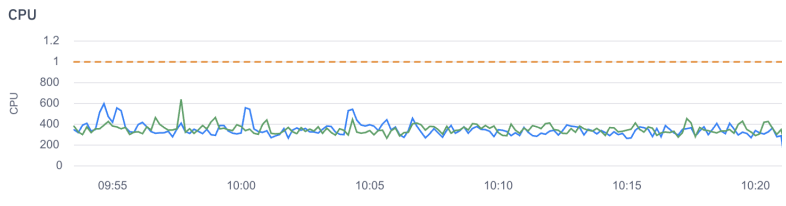




The initial run **with** Iron Vector (IV):







As you can see, the throughput with IV is significantly lower than the baseline (by ~23%), but you can also notice that the CPU is **much** lower: 30-40% instead of > 90%.

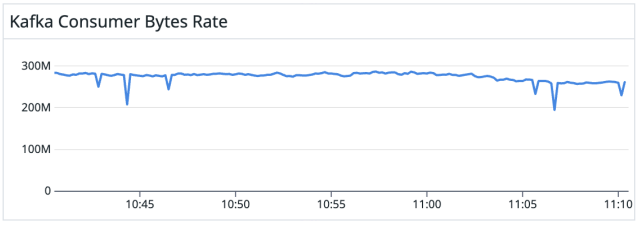
After some investigation, it was discovered that the Kafka consumer was not producing the data quickly enough. **The Iron Vector Avro to Arrow conversion is very efficient, and the pipeline becomes IO-constrained, not CPU-constrained.**

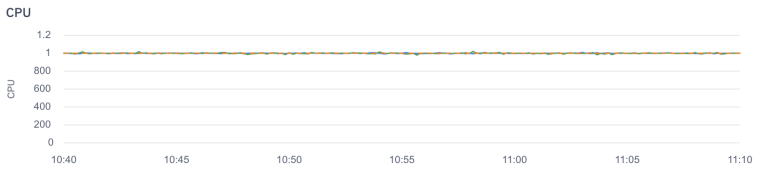
In order to fully leverage the CPUs, the Kafka consumer configuration was tuned in the following way:

- max.poll.records: 2048
- fetch.max.bytes: 104857600
- max.partition.fetch.bytes: 33554432
- fetch.max.wait.ms: 10000
- fetch.min.bytes: 1048576
- receive.buffer.bytes: -1

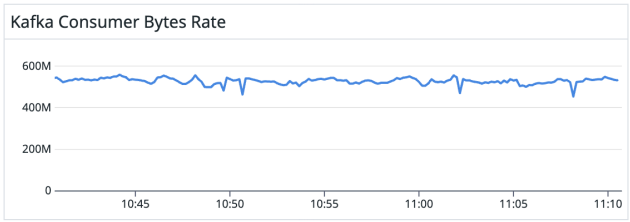
ALL of the following tests were performed with this configuration.

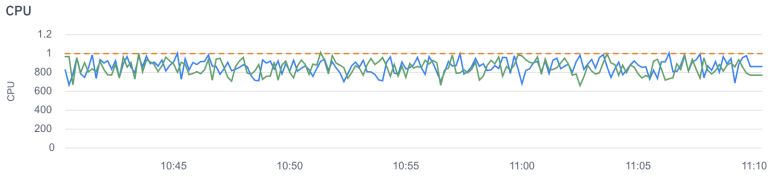
Now, the baseline run **without** Iron Vector:





The run **with** Iron Vector (IV):



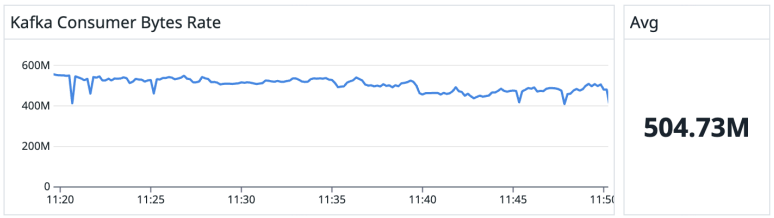


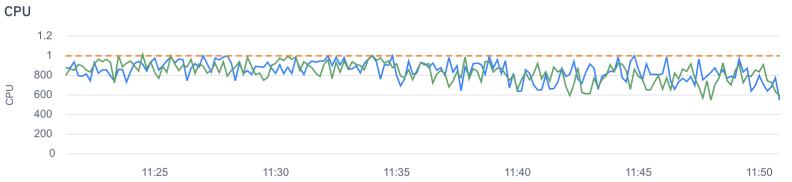
Now the throughput with IV is ~93% higher! The CPU is fully utilized in the case of the baseline run; the IV CPU fluctuates around 80%.

Flink object reuse configuration can be enabled to eliminate some intermediate serialization. It's generally safe to enable it in the case of the SQL and Table APIs (unless your UDF does something weird). Even though it's disabled by default, it's important to benchmark the pipelines with it, as it's a very common optimization.

The baseline run **without** Iron Vector **with** object reuse enabled:

The run **with** Iron Vector (IV) and **with** object reuse enabled:





As you can see, enabling object reuse increased the baseline throughput by ~40%, so the IV increase is “only” ~31%.

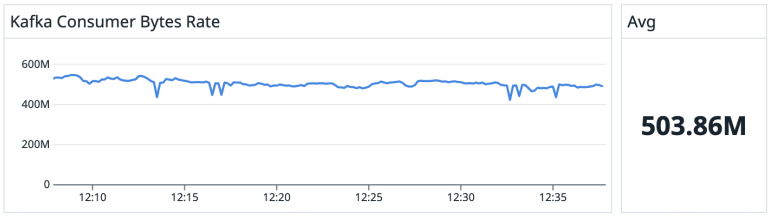
Predictably, it didn’t make any impact on the IV run since IV mostly uses Arrow batches in runtime.

Test B: Pipeline with Rebalancing

In this test scenario, we modified the sink parallelism to 1 to introduce rebalancing. Other than that, the Flink SQL to define the tables and computation is the same. We also leveraged the tuned Kafka consumer configuration from the previous test.

The baseline run **without** Iron Vector:

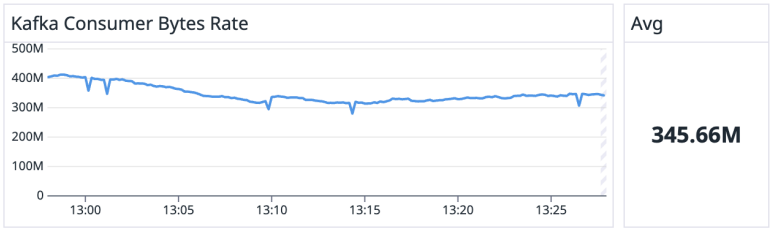
The run **with** Iron Vector (IV):



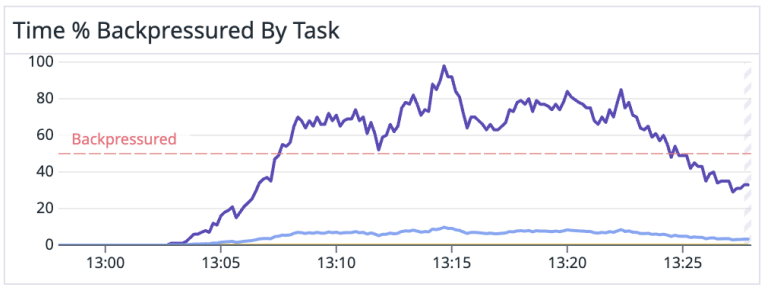
We see a slightly lower baseline throughput, but largely the same IV throughput. Still getting about ~97% improvement.

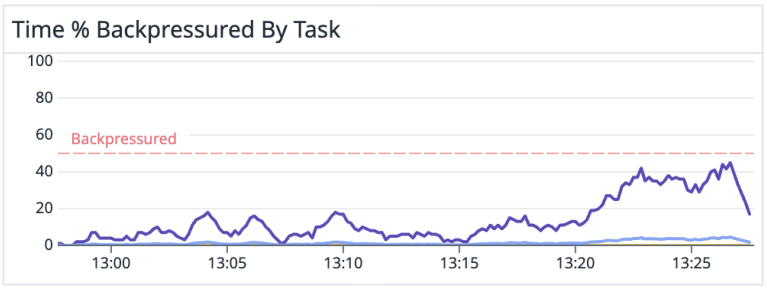
Now let's enable object reuse to see if we get the same performance impact.

The baseline run **without** Iron Vector **with** object reuse enabled:



Initially, the throughput increased significantly, but it quickly dropped. The reason is backpressure: even though we use the discarding / blackhole sink (which should theoretically create zero backpressure), the data still needs to be serialized, sent over the network, and deserialized first. Here's the backpressure comparison between the baseline (with object reuse) and IV runs:





This explains why, even when object reuse is enabled, we don't see a huge improvement.

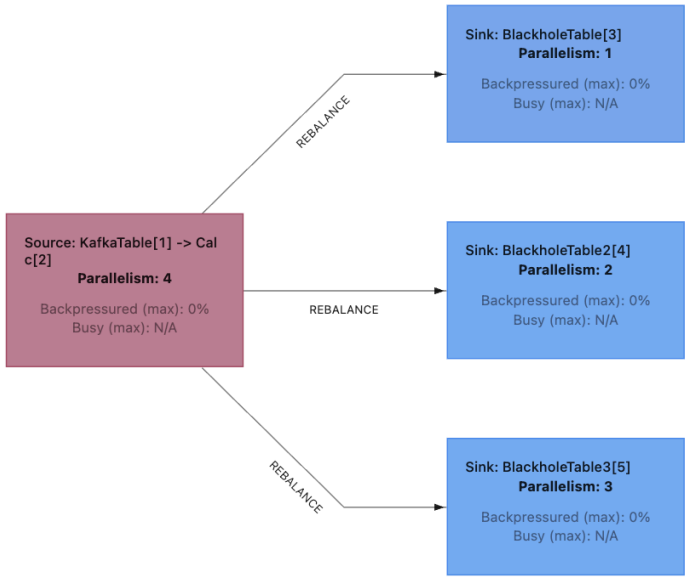
So, in this test, IV performs ~46% to ~97% better than the baseline, depending on the object reuse configuration.

Test C: Pipeline with Multiple Sinks

In this test scenario, we use the same source, but create three blackhole sinks with parallelism of 1, 2 and 3. Then we use the statement set to connect them to the same source:

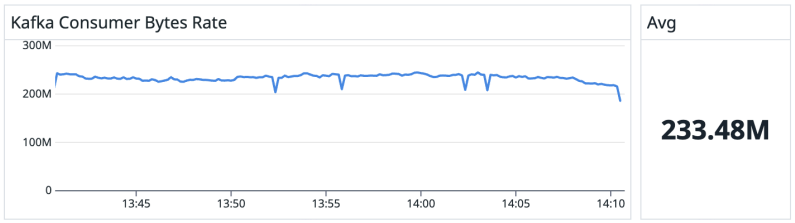
```sql
EXECUTE STATEMENT SET
BEGIN
  INSERT INTO BlackholeTable SELECT id, block_number, log_index * 2 as res,
CONCAT('block_hash_', REGEXP_REPLACE(block_hash, '0x', '')) as res2 FROM KafkaTable WHERE
transaction_index > 40;
  INSERT INTO BlackholeTable2 SELECT id, block_number, log_index * 2 as res,
CONCAT('block_hash_', REGEXP_REPLACE(block_hash, '0x', '')) as res2 FROM KafkaTable WHERE
transaction_index > 40;
  INSERT INTO BlackholeTable3 SELECT id, block_number, log_index * 2 as res,
CONCAT('block_hash_', REGEXP_REPLACE(block_hash, '0x', '')) as res2 FROM KafkaTable WHERE
transaction_index > 40;
END
```

So the pipeline looks like this in the Flink UI:

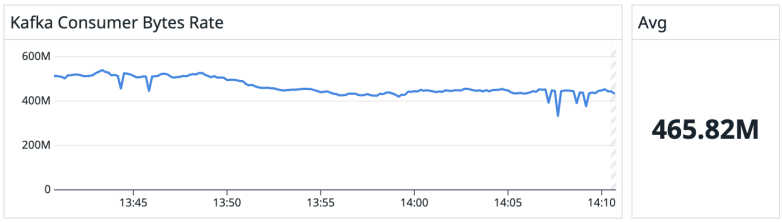


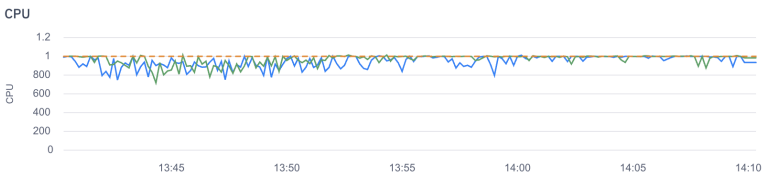
This test is quite helpful to understand how Flink object reuse, as well as Iron Vector's zero-copy Arrow exchange, react to the increase in the number of destinations.

The baseline run **without** Iron Vector:



The run **with** Iron Vector (IV):

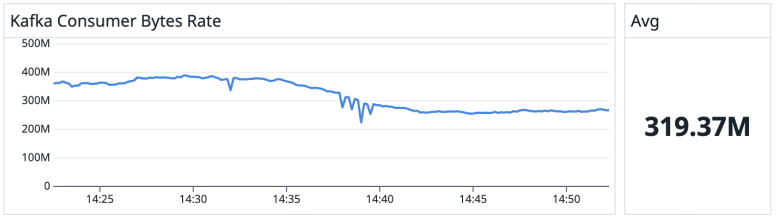




Flink is doing more work now, but the ratio stays the same, ~100%.

Now let's enable object reuse.

The baseline run **without** Iron Vector **with** object reuse enabled:

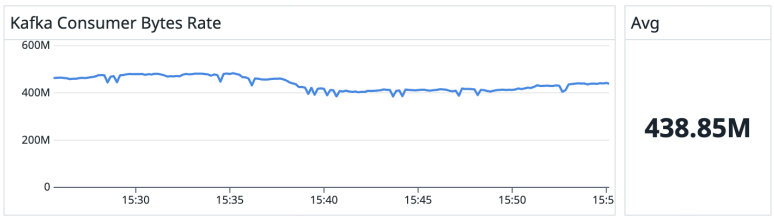


The baseline improved by ~37%, but IV throughput is still higher by ~46%. Again, backpressure is the reason for the moderate increase.

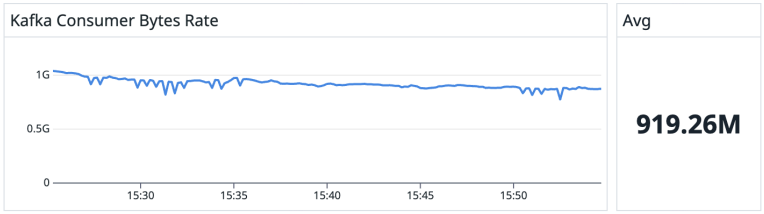
Test D. Increased Parallelism

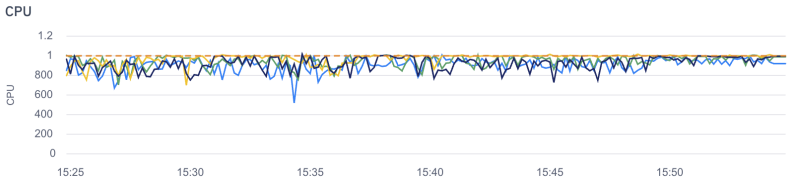
Now let's re-run Test B, but double the parallelism to 8 by adding two more TaskManagers. The goal is to make sure the improvement scales with the increased number of TaskManagers.

The baseline run **without** Iron Vector:



The run **with** Iron Vector (IV):





The ratio stays the same, we're still getting ~100% (even slightly more) throughput increase.

This proves that it would be possible to reduce the number of TaskManagers in half and still get the same throughput when the Iron Vector accelerator is enabled.

Summary

Test	Baseline	Iron Vector	Increase
Test A: Completely Fused Pipeline	274.05 MiB/s	527.79 MiB/s	93%
Same but with object reuse	383.91 MiB/s	504.73 MiB/s	31%
Test B: Pipeline with Rebalancing	255.76 MiB/s	503.86 MiB/s	97%
Same but with object reuse	345.66 MiB/s	503.86 MiB/s	46%
Test C: Pipeline with Multiple Sinks	233.48 MiB/s	465.82 MiB/s	100%
Same but with object reuse	319.37 MiB/s	465.82 MiB/s	46%
Test D. Increased Parallelism	438.85 MiB/s	919.26 MiB/s	109%

This series of benchmarks demonstrated that Iron Vector can improve Apache Flink throughput by 46% to 100%+, depending on the Flink configuration.

We can also see that the Flink pipelines with object reuse perform better. Even though it's not enabled by default and likely not used by many Flink users, it's important to compare against it because it's a relatively straightforward change. It could've been excluded from the benchmarks to make the numbers look better, but Irontools is not interested in benchmaxxing.

There is also likely room for further improvements. The CPU typically wasn't fully utilized during the Iron Vector runs. There are also some planned improvements for the network exchange and its data serialization.

It's also evident that the improvement scales with the number of TaskManagers, so it's possible to leverage Iron Vector to:

- Reduce the compute cost of running Flink pipelines by 46% to 100%.
- Or increase the throughput by the same amount.