TRD: Intelligence - Benchmarking Fulfillment and Transit Time

Key Stakeholders

Author(s): @Julia Le @Elliott Feng

Squad Engineering Manager: @diptamay.sanyal (Deactivated)

- ☐ **Product signoff:**
- ☐ **Pod Lead:** @Ketan Rajesh Kumar

[Notes from review meeting]

Background

Retailers are not always aware of carrier delays that happen across the network. This is valuable information to have so they can address issues with carriers or prepare for extended TNTs. If the time between order date and delivery date is outside a preferred window, retailers need to investigate whether it's an issue with a carrier or their fulfillment process. Retailers cannot tell if carrier delays are isolated or are also impacting other retailers/carriers

The purpose of this initiative is to bring awareness to longer fulfillment and carrier transit times, broken down by carrier and region, so that a retailer can take action on either the fulfillment or carrier side (better staff or train for fulfillment, address issues with carriers, prepare consumers for longer wait times). With this information, retailers can alert customers to potential delays/longer transit times than normal/risk of items arriving after Christmas during high volume periods such as peak.

Design Goals

Functional Requirements

- Visualize the stage(s) in the order/shipment progress affected by delays (pre-ship versus post-ship).
- Demonstrate historical trends in fulfillment and transit time, with merchandise category benchmark and carrier network comparisons
- Flag carrier and regions experiencing increased fulfillment time and TNT
- List shipments likely affected by longer transit times based on their carrier, region and/or other characteristics

Non-Functional Requirements

- Scalability - We are looking to onboard 800+ retailers. The solution needs to be scalable for all
- Reliability - Uptime 99.95% (industry standard).
- Latency - Predictions and visualizations must not exceed Metabase's maximum runtime threshold
- Maintainability - We need to be able to improve or modify predictive models without negatively affecting performance for existing retailers
- Metrics and Alerts - We need to have proactive monitoring and alerts per retailer on the following:
  a. Resource, quotas and limits usage per retailer
  b. Performance metrics for shipment delay predictions, such as sensitivity & specificity

Solution Summary

Benchmarking Fulfillment Time and Time in Transit

Data Sources:

- Actual *Fulfillment Time* (FT) and *Time in Transit* (TNT) will be computed for each shipment, using carrier scan event data logged by Atlas.
- For development and testing, we will continue using the `monitor.shipments` table currently populated by `monitor_etl` on composer.

Our goal state is to consume from tables logged by **Monitor Analytics**, which should provide local ship date, local delivery date and their respective local timezones to assure the accuracy of our calculation.

The definition for FT and TNT are listed as below:

> `fulfillment time in calendar days = date_diff(utc ship date, utc order date, days)`
>
> `time in transit in calendar days = date_diff(local delivery date, local ship date, days)`

This is the steps how we get utc/local ship_date and delivery_date. For order date, we always use utc(passed by upstream).

First we use a regex function to convert the `event_ts_raw` into a datetime, and we take it as a local time.

Then we use the following function to get the raw/utc ship_date, basically the first 300 event code in the event list:

```
first_value(case when event_code = '300'
            then struct(safe_cast(coalesce(event_derived_ts, event_ts) as datetime) as utc, event_ts_local as local, event_ts_raw as raw)
else null end ignore nulls)
            over (shipment_window_asc) as ship_date,
```

and for raw/utc delivery_date, the first 50X event code in the event list:

```
first_value(case when event_code in ('500', '501', '502', '503')
            then struct(safe_cast(coalesce(event_derived_ts, event_ts) as datetime) as utc, event_ts_local as local, event_ts_raw as raw)
else null end ignore nulls)
            over (shipment_window_asc) as delivery_date,
```

Then we use the following to calculate TNT/FT:

```
datetime_diff(delivery_date.utc, t.order_date.utc, day) as order_to_delivery_days,
datetime_diff(ship_date.utc, t.order_date.utc, day) as order_to_ship_days,(FT)
datetime_diff(delivery_date.local, ship_date.local, day) as ship_to_delivery_days,(TNT)
```

Note: This calculation is done in calendar days because FT relies on retailer and distribution center business days (which we don't store), and TNT relies on carrier service business days (which the carrier team is working on). When data on working days and hours become available, we should be prepared to recompute using business days.

Benchmarking with Narvar Network Data:

For each retailer, we allow FT and TNT comparisons to (1) retailer category benchmark, (2) carrier network benchmark. Drilldowns by geographical location must be available, and will be visualized on a heatmap.

We also target to provide near-real time benchmarks, an hourly update is acceptable.

Hence we need to offer the granularity at above levels, as the following fields would be required to be grouped by:

1. retailer_moniker
2. carrier_moniker
3. timestamp_trunc on partition_ts/event_ts(on hour level)
4. dest_country
5. origin_country
6. dest_state
7. origin_state
8. *dest_zip/Destination Location Grid {longitude, latitude}
9. *origin_zip/Origin Location Grid {longitude, latitude}

*Note**: We will be providing benchmarks at the state-level geo granularity for GA (US-only). 8 and 9 are for future planning.

The following example would explain what we will be offering:

For a retailer ***R***, with merchandise category ***X***, with carriers ***{c1, c2, ...}***, and within any timeframe:

- View median and 95th percentile TNT and FT for retailer ***R*** for all carriers,
- View median and 95th percentile TNT and FT for merchandise category ***X*** (spanning multiple retailers)
- View median and 95th percentile TNT and FT for carrier ***c1*** and retailer ***R***
- View median and 95th percentile TNT and FT for carrier ***c1*** and merchandise category ***X*** (spanning multiple retailers)
- Allow drill down by state, and preferably with all the above breakdown levels (US-only)
- Benchmarks should be as near-real time as possible, especially for Active Shipments dashboard.

Solution Details

Approaches for Getting FT/TNT Benchmarks:

1. **Compute median/p95 at dashboard runtime, using BigQuery median/percentile function**
   a. Pros:
      i. No need to maintain a view/rollup table. Query is easy to build and maintain.
      ii. We will have the flexibility for different widgets and breakdown level.
      iii. Easy to add new requests if needed.
   b. Cons:
      i. Median and percentile function are using sorting logic. It is an **O(n log n)** algorithm so very time consuming.
      ii. Very inefficient and unacceptably long query runtimes if the user chooses to query over a wide date range.
2. **Precompute counts of every TNT/FT value** on timestamp trunc granularity into a **materialized view, using Javascript UDF function to calculate median/p95 at dashboard runtime**
   a. Pros:
      i. Counts are precomputed. At query level, we can get the median/p95 in constant time, as sorting is not required.
      ii. For historical data, we would be able to get the trivial solution instead very quick, since materialized view will store the data for historical.
      iii. Comparatively easy to maintain, we can simply recreate view to accommodate new
   b. Cons:
      i. Materialized view will store the data physically, and its size grows rapidly with increasing granularity (e.g. zip code-level benchmarking versus state-level).
      ii. Dashboard query to the view will need pivot https://cloud.google.com/bigquery/docs/reference/standard-sql/query-syntax#pivot_operator and udf. Dashboard queries will be complex and need to be in SQL, or this logic will need to be in an additional non-materialized view.
3. **Precompute counts of every TNT/FT value at batch-level using a rollup table and update through an hourly/daily ETL, using Javascript UDF function to calculate median/p95 at dashboard runtime**
   a. Pros:
      i. Counts are precomputed. At query level we can get the median/p95 in **O(n)** time, as sorting is not required.
      ii. For historical data, we would be able to get the trivial solution very quickly.
   b. Cons:
      i. Maintaining a rollup table on BigQuery would require additional setups. We need to maintain an etl on airflow. This query run itself would be time consuming. Not cost effective as well.
      ii. We will need to maintain a two step query when calculating real time benchmark. Need to fetch records from the rollup table for historical, as long as the latest data from the source table for most recent records that are not fetched by the ETL yet.
      iii. Table size grows rapidly.
      iv. Changes are hard to apply if we have a rollup table. Accommodating new requests can be painful as changes will be applied on all elements: rollup tables, ETL and dashboard queries.
      v. Backfill will be needed for any logic changes.

We will be proving why we chose approach 2 in POC for Computing Median & Percentiles of TNT/FT: Run-Time vs Precomputed.

Schema for TNT (Almost Identical for FT):

Column Name	Data Type	Decription
retailer_partition	INTEGER	An integer: `CAST(ABS(MOD(FARM_FINGERPRINT(retailer_moniker),4000)) as INT64)`, used as the partition for the view. Inherited from the base table `monitor.shipments`.
retailer_moniker	STRING	Retailer moniker.
carrier_moniker	STRING	Carrier moniker.
dest_country	STRING	Dest country.
dest_state	STRING	Dest state.
origin_country	STRING	Origin country.
origin_state	STRING	Dest country.
tnt	INTEGER	Ship to delivery dates.
event_batch_ts	TIMESTAMP	Time frame granularity: `timestamp_trunc(partition_ts, hour)`.
tnt_count	INTEGER	Count of shipments with a particular `tnt` value. Used to efficiently calculate median and p95 in dashboard queries at runtime.

Additional Joins:

Materialized views do **not** allow any outer joins during view creation. Hence there are two outer joins we need to add afterwards:

1. Left join with `analytics.v_retailer_category`, in order to get the Retailer Merchandise category for benchmarking.
2. *Left join with `ort.zipcode_dataset_for_eddapi`, in order to get the geo information.

Number 1 will be live in GA. Number 2 is for future plan.

Those two joins could potentially impact the performance of query.

Tentative Dashboard Deliverables

Metabase mockup of dashboard widgets involving data intelligence, under "Dashboard Deliverables":

Monitor V2: Intelligence & Predictive Analytics - Scoping

Sign in to your Google Account

You must sign in to access this content

Sign in

Code

Where is the code going to go? Does this touch any common repos? Any new repos being created?

https://github.com/narvar/monitor-analytics

Testing

Materialized view testing for dashboard queries:

We are using the following steps to test the performance of the proposed materialized view.

Creation of view:

We created the view for demo using the data from `monitor.shipments`.

```
create or replace materialized view `narvar-data-lake.tmp.monitor_tnt_test8`
PARTITION BY range_bucket(retailer_partition, generate_array(0,4000,1))
CLUSTER BY event_batch_ts, carrier_moniker

as
SELECT
retailer_partition,
retailer_moniker,
carrier_moniker,
dest_country,
dest_state,
origin_country,
origin_state,
ship_to_deliver_days,
TIMESTAMP_TRUNC(partition_ts, day) event_batch_ts,
count(1) as tnt_count
FROM `narvar-data-lake.tmp.monitor_tnt_test_base`
WHERE DATE(partition_ts) >= "2021-01-01"
group by retailer_partition, retailer_moniker, carrier_moniker, dest_country, dest_state, origin_country, origin_state, event_batch_ts,
ship_to_deliver_days
```

We tested the performance for the materialized view, leveraging the run time based on the retailer, carrier and geo.

Unit testing for UDF:

We are using the following UDF to calculate median and percentile:

```
CREATE TEMP FUNCTION percentile(r STRUCT<_0 INT64,_1 INT64,_2 INT64,_3 INT64,_4 INT64,_5 INT64,_6 INT64,_7 INT64,_8 INT64,_9 INT64,_10
INT64,_11 INT64,_12 INT64,_13 INT64,_14 INT64,_15 INT64,_16 INT64,_17 INT64,_18 INT64,_19 INT64,_20 INT64,_21 INT64,_22 INT64,_23
INT64,_24 INT64,_25 INT64,_26 INT64,_27 INT64,_28 INT64,_29 INT64,_30 INT64>, p FLOAT64)
  RETURNS INT64
  LANGUAGE js AS r"""
  arr =
Array(Number(r['_0']),Number(r['_1']),Number(r['_2']),Number(r['_3']),Number(r['_4']),Number(r['_5']),Number(r['_6']),Number(r['_7']),Nu
mber(r['_8']),Number(r['_9']),Number(r['_10']),Number(r['_11']),Number(r['_12']),Number(r['_13']),Number(r['_14']),Number(r['_15']),Numb
er(r['_16']),Number(r['_17']),Number(r['_18']),Number(r['_19']),Number(r['_20']),Number(r['_21']),Number(r['_22']),Number(r['_23']),Numb
er(r['_24']),Number(r['_25']),Number(r['_26']),Number(r['_27']),Number(r['_28']),Number(r['_29']),Number(r['_30']))
  sum = arr.reduce((a, b) => a + b, 0)
  pct_of_sum = Math.floor(sum*p) - 1;
  cumsum = 0
  for (var i = 0; i <= arr.length; i++) {
      cumsum += arr[i]
      if (cumsum >= pct_of_sum) {
          return i
      }
   }
  return null
  """;
```

We will perform unit testing on the udf, and further optimize the udf to increase query performance at runtime.

QA Test Plans (Functional, Automation and Performance)

We will cover the functional test on QA once monitor analytics has data flowing into QA. In QA env, Tasks includes:

1. Define the granularity
2. Build up materialized view from the base table.

3. Set up dashboard query to get data for dashboard widgets
4. Build up dashboard and make sure data comes in plate

We will cover the query performance testing directly on the prod data as QA might not have the same volume of data as prod and could not suffice the needs for query performance.

Scaling and Rate Limiting

Our materialized view will grow rapidly because we are keep writing data into the base table, and breakdown into a very small granularity - combination of {retailer_moniker, carrier_moniker, geo}.

As introduced in the doc, https://cloud.google.com/bigquery/docs/materialized-views-intro, we will have option to partition and cluster the materialized view to save query cost.

The here are the solutions:

1. The partition column for materialized view should match the source table, hence we will be using retailer_partition, a mapped integer value from retailer_moniker as the partition. This is matching our product feature.
2. We will use the dest/origin_state, event_batch_ts and carrier_moniker as cluster to further optimize the query.

The above two will make sure when a specific retailer is querying through dashboard, they will be querying only their assigned bucket, and required clusters.

By doing this, query cost and run time will reduce a lot.

Operation Details | Runbook

To be filled.

Tech Debt

To be filled.