BENOCS

Self-repairing processing using ClickHouse for unreliable data sources

Ingmar Poese

05/12/2022

About BENOCS

Location: Berlin, Germany

Industry: Telecommunications, Network Telemetry

Mission: To improve network traffic conditions and visibility

2013

Founded as a Spin-off of Deutsche Telekom

2015

First live customers for BENOCS Director

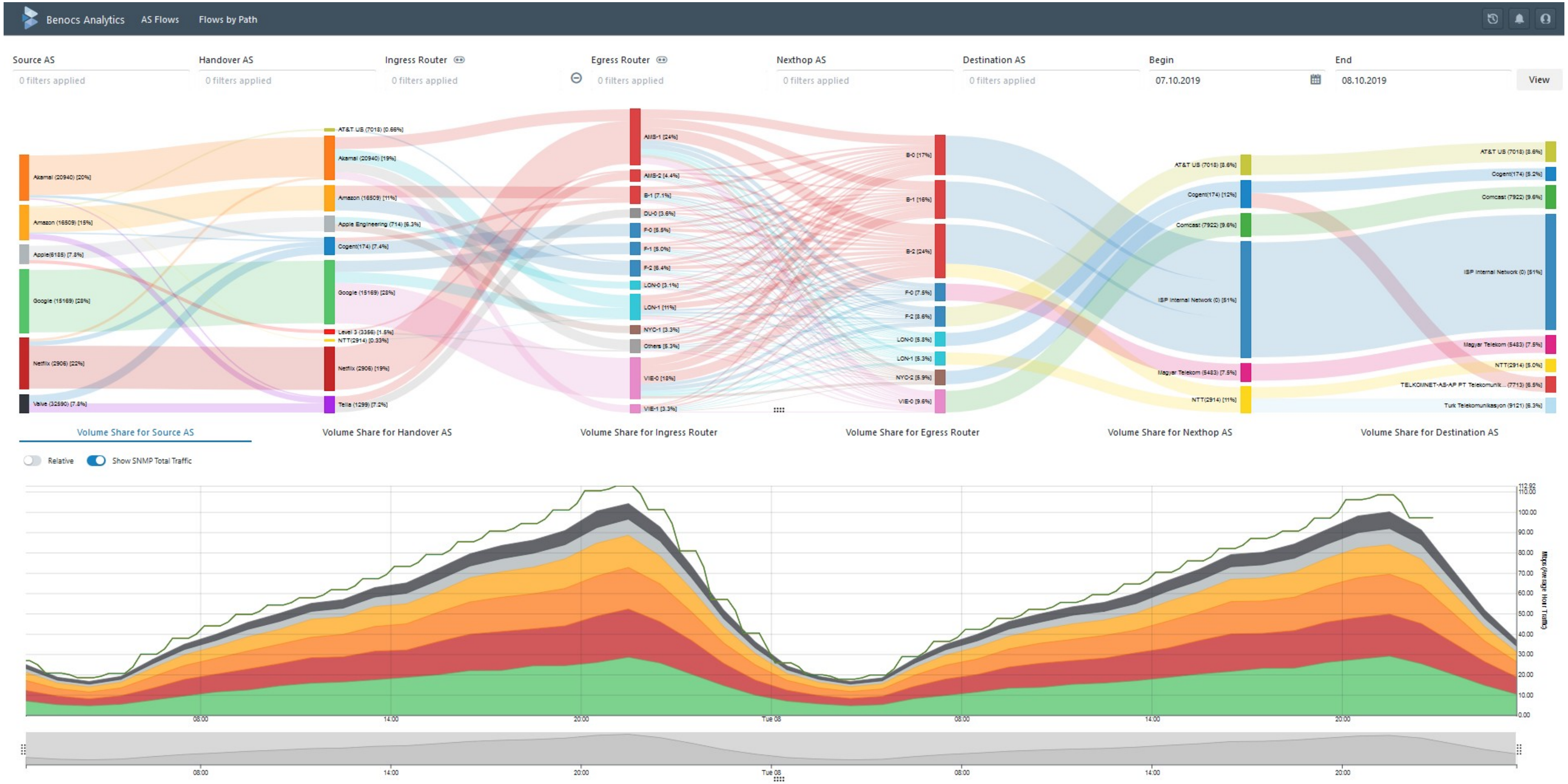
2018

Fully developed BENOCS Analytics enters the market

2020

Customer base expanded globally

Visualizes end-to-end traffic at one glance



Source AS: 0 filters applied

Handover AS: 0 filters applied

Ingress Router: 0 filters applied

Egress Router: 0 filters applied

Nexthop AS: 0 filters applied

Destination AS: 0 filters applied

Begin: 07.10.2019

End: 08.10.2019

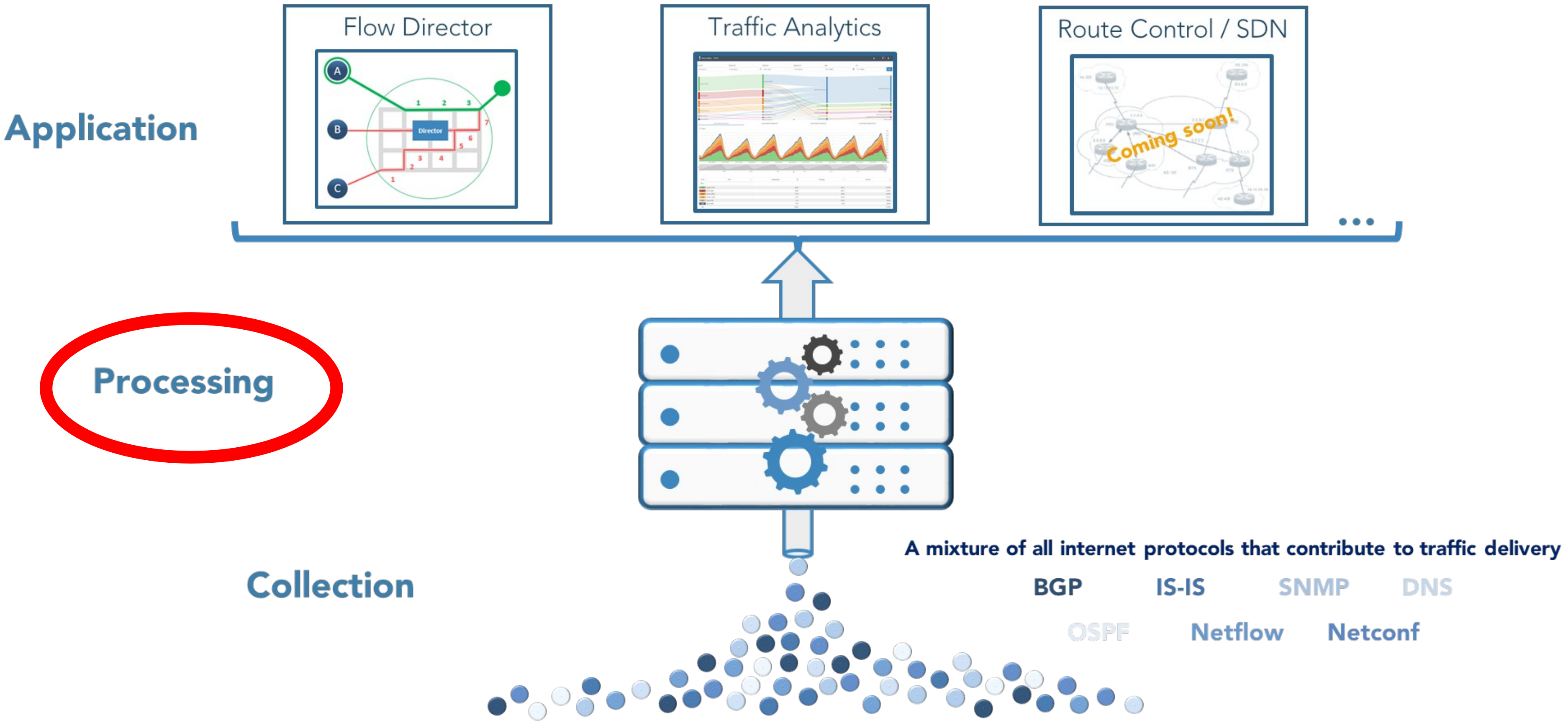
○ Relative

● Show SNMP Total Traffic

This is the BENOCS solution

Products:

Our products are made from our Core Engine meant to target efficient internet-traffic delivery.



Application

Flow Director

Traffic Analytics

Route Control / SDN

Coming soon!

...

Processing

Collection

A mixture of all internet protocols that contribute to traffic delivery

BGP IS-IS SNMP DNS

OSPF Netflow Netconf

The challenge in (our) data processing

No control over data sanity:
Expect data to be partially taited, useless, broken, wrong or (maybe) malicious.

No Control over delivery timing:
Data will arrive whenever (Bursts, constant delays, intermittent failures),

most of the time

without any guarantees what is happening when

Multiple data sources:
Each source at each customer may have its own delivery, timing and failure/recovery model

All analysis is interconnected, single sources are (almost) useless:
Data cannot be processed until multiple/all data sources are ready – or when there is evidence it will not come

Installations at customer premises (SaaS):
We manage everything, but run in a highly distributed, fully separated, partially internet-less environment.

Our Goals

Clear dependency graph

Define a graph of **Targets** where each table has a set of requirements (predecessors) and a set of dependencies (successors).

Easy adaption of intermediate processing steps

Each **Target** needs to be fully interchangeable without touching/modifying any of the other Targets.

Statelessness

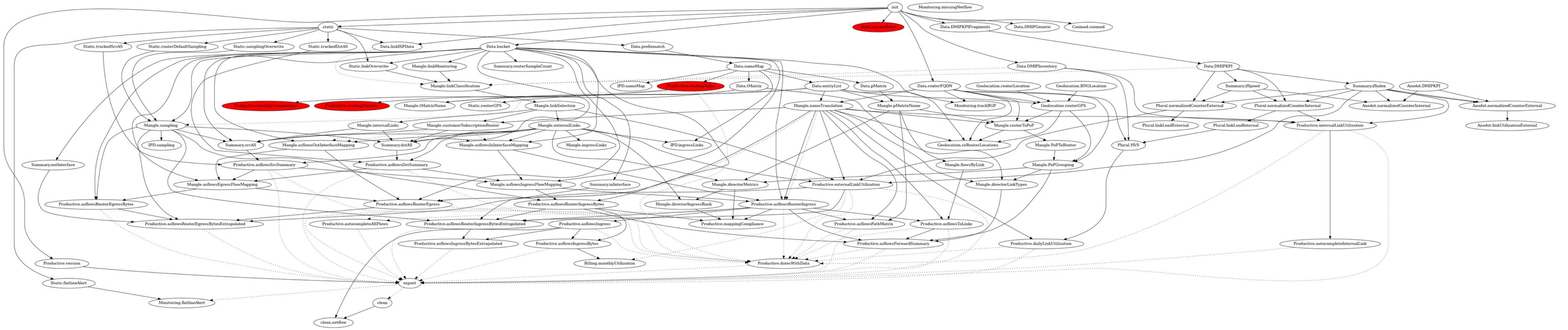
All state is kept in clickhouse. The Targets have no state and re-sychronize with the DB every time they run.

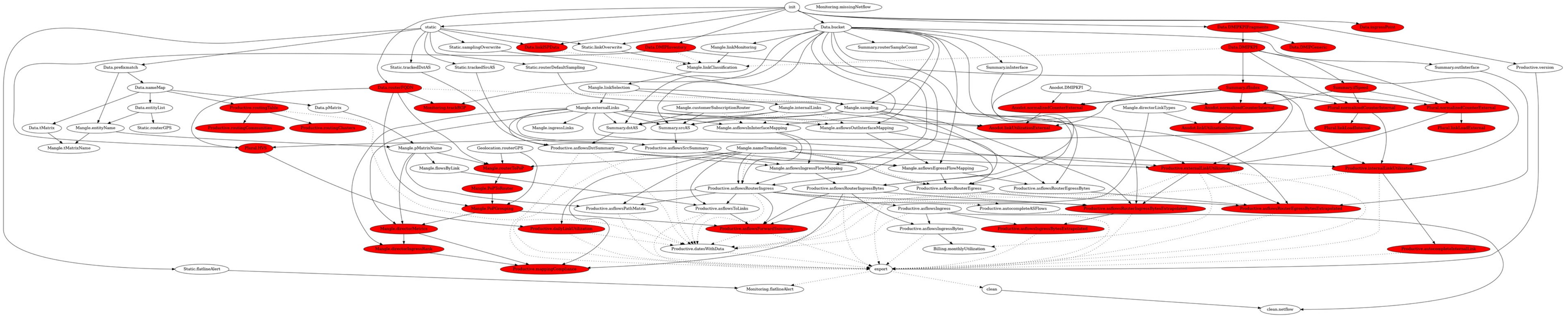
Easy re-processing of data or ***Stuff*** fails - all the time !

A **Target** will always process all data that is eligible for processing. No matter if new or old.

This allows for re-processing as well as normal time progression

Sample Target graphs


init
Monitoring.missingNetflow
static
Data.ingressPoint
Data.DMIPKPIFragments
Data.DMIPGeneric
Conmed.conmed
Static.trackedSrcAS
Static.routerDefaultSampling
Static.samplingOverwrite
Static.trackedDstAS
Data.linkISPData
Data.bucket
Data.prefixmatch
Static.linkOverwrite
Mangle.linkMonitoring
Summary.routerSampleCount
Data.nameMap
Data.DMIPInventory
Data.DMIPKPI
Mangle.linkClassification
IPD.nameMap
Productive.routingTable
Data.tMatrix
Data.entityList
Data.pMatrix
Data.routerFQDN
Geolocation.routerLocation
Geolocation.BNGLocation
Summary.ifSpeed
Summary.ifIndex
Anodot.DMIPKPI
Productive.routingCommunities
Productive.routingClusters
Mangle.tMatrixName
Static.routerGPS
Mangle.linkSelection
Mangle.nameTranslation
Mangle.pMatrixName
Monitoring.trackBGP
Geolocation.routerGPS
Plural.normalizedCounterExternal
Plural.normalizedCounterInternal
Anodot.normalizedCounterInternal
Anodot.normalizedCounterExternal
Mangle.sampling
Mangle.internalLinks
Mangle.customerSubscriptionRouter
Mangle.externalLinks
Mangle.routerToPoP
Plural.linkLoadExternal
Plural.linkLoadInternal
Productive.internalLinkUtilization
Anodot.linkUtilizationExternal
IPD.sampling
Summary.srcAS
Mangle.asflowsOutInterfaceMapping
Summary.dstAS
Mangle.asflowsInInterfaceMapping
Mangle.ingressLinks
IPD.ingressLinks
Geolocation.ceRouterLocations
Mangle.PoPToRouter
Plural.HVS
Summary.outInterface
Productive.asflowsSrcSummary
Productive.asflowsDstSummary
Mangle.flowsByLink
Mangle.PoPGrouping
Mangle.asflowsEgressFlowMapping
Mangle.asflowsIngressFlowMapping
Summary.inInterface
Mangle.directorMetrics
Productive.externalLinkUtilization
Mangle.directorLinkTypes
Productive.asflowsRouterEgressBytes
Productive.asflowsRouterEgress
Productive.asflowsRouterIngressBytes
Mangle.directorIngressRank
Productive.asflowsRouterIngress
Productive.asflowsRouterEgressBytesExtrapolated
Productive.autocompleteASFlows
Productive.asflowsRouterIngressBytesExtrapolated
Productive.asflowsIngress
Productive.mappingCompliance
Productive.asflowsPathMatrix
Productive.asflowsToLinks
Productive.asflowsIngressBytesExtrapolated
Productive.asflowsIngressBytes
Productive.asflowsForwardSummary
Productive.dailyLinkUtilization
Productive.autocompleteInternalLink
Productive.version
Billing.monthlyUtilization
Productive.datesWithData
Static.flatlineAlert
export
Monitoring.flatlineAlert
clean
clean.netflow


init
Monitoring.missingNetflow
static
Data.bucket
Data.DMIPKPIFragments
Data.ingressPoint
Static.samplingOverwrite
Data.linkISPData
Static.linkOverwrite
Data.DMIPInventory
Mangle.linkMonitoring
Summary.routerSampleCount
Data.DMIPKPI
Data.DMIPGeneric
Data.prefixmatch
Static.trackedDstAS
Static.trackedSrcAS
Static.routerDefaultSampling
Mangle.linkClassification
Summary.inInterface
Summary.outInterface
Productive.version
Data.nameMap
Data.routerFQDN
Mangle.linkSelection
Anodot.DMIPKPI
Summary.ifIndex
Summary.ifSpeed
Data.entityList
Productive.routingTable
Data.pMatrix
Monitoring.trackBGP
Mangle.externalLinks
Mangle.customerSubscriptionRouter
Mangle.internalLinks
Mangle.sampling
Anodot.normalizedCounterExternal
Mangle.directorLinkTypes
Anodot.normalizedCounterInternal
Plural.normalizedCounterInternal
Plural.normalizedCounterExternal
Data.tMatrix
Mangle.entityName
Static.routerGPS
Productive.routingCommunities
Productive.routingClusters
Mangle.ingressLinks
Summary.dstAS
Summary.srcAS
Mangle.asflowsInInterfaceMapping
Mangle.asflowsOutInterfaceMapping
Anodot.linkUtilizationExternal
Anodot.linkUtilizationInternal
Plural.linkLoadInternal
Plural.linkLoadExternal
Mangle.tMatrixName
Plural.HVS
Mangle.pMatrixName
Geolocation.routerGPS
Productive.asflowsDstSummary
Productive.asflowsSrcSummary
Mangle.nameTranslation
Mangle.flowsByLink
Mangle.routerToPoP
Mangle.asflowsIngressFlowMapping
Mangle.asflowsEgressFlowMapping
Productive.externalLinkUtilization
Productive.internalLinkUtilization
Mangle.PoPToRouter
Productive.asflowsRouterIngress
Productive.asflowsRouterIngressBytes
Productive.asflowsRouterEgress
Productive.asflowsRouterEgressBytes
Mangle.PoPGrouping
Productive.asflowsPathMatrix
Productive.asflowsToLinks
Productive.asflowsIngress
Productive.autocompleteASFlows
Productive.asflowsRouterIngressBytesExtrapolated
Productive.asflowsRouterEgressBytesExtrapolated
Mangle.directorMetrics
Productive.dailyLinkUtilization
Productive.asflowsForwardSummary
Productive.asflowsIngressBytes
Productive.asflowsIngressBytesExtrapolated
Mangle.directorIngressRank
Productive.datesWithData
Billing.monthlyUtilization
Productive.autocompleteInternalLink
Static.flatlineAlert
Productive.mappingCompliance
export
Monitoring.flatlineAlert
clean
clean.netflow

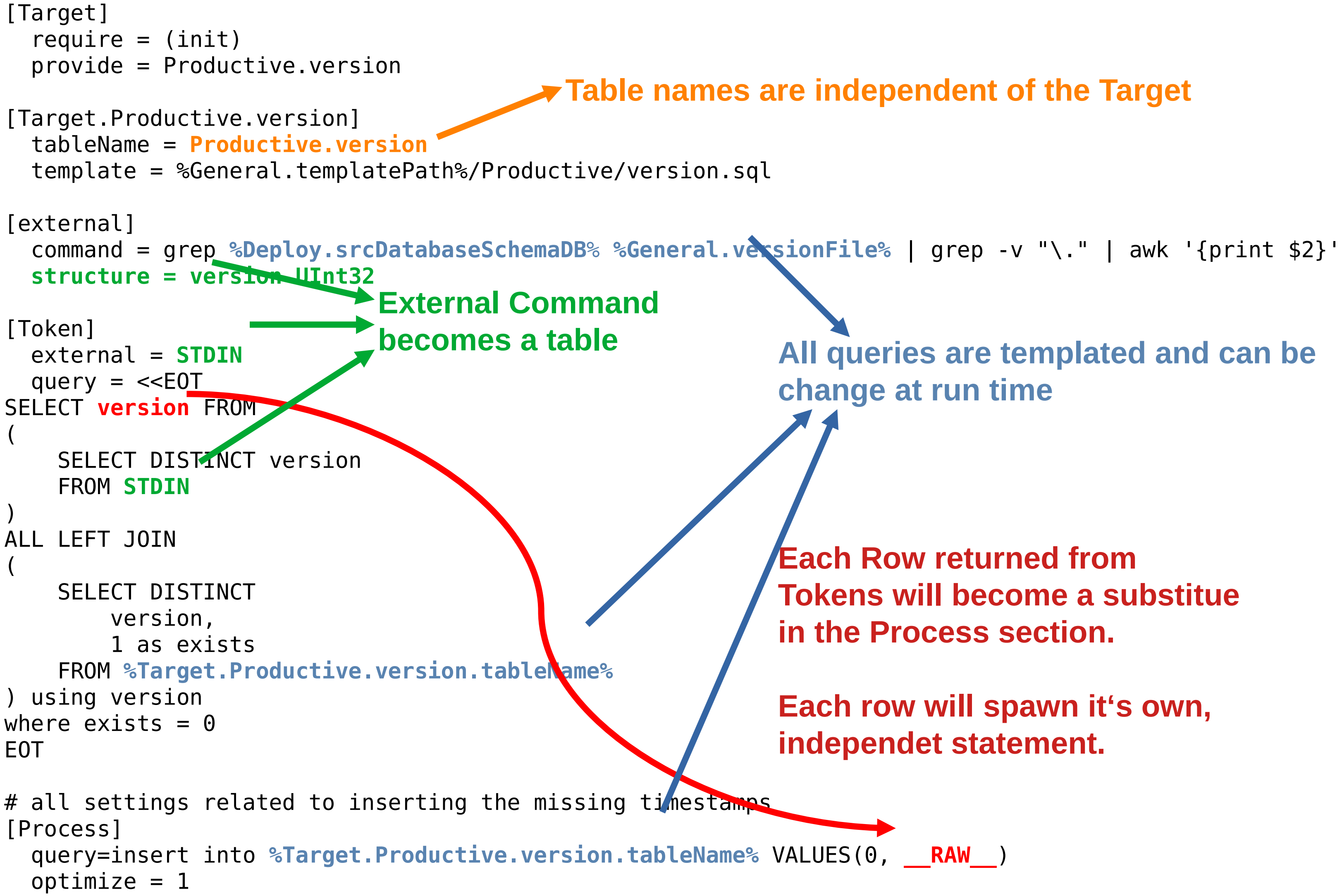
SyncClickhouse (working title)

Definition: Target

A **Target** is a specific processing step in the graph – usually bound to a single table. A **Target**

- can have none to multiple requirements (i.e. target that have to finish beforehand)
- defines exactly one **Name** that other **Targets** can depend on
- can be bound to one table in clickhouse. This table will be maintained through the **Target** (create, update, alter…)
- Defines two stages for processing
  1) Tokens
     1) Tokens uniquely identify non-overlapping parts of a table (usually time ranges/timestamps) that are eligible for processing. This is done through a single SQL statement.
     2) The SQL statement can consider data from an external source (i.e. files from a disk)
  2) Processing
     1) Each Token is processed independently and in non-determistic order (i.e. partially parallel) by spawning a templated query (usually an *INSERT* statement).
- There is a lot more Targets can do (multi-processing, external Commands, Query breakdown, etc)

Target Example:



```
[Target]
  require = (init)
  provide = Productive.version

[Target.Productive.version]
  tableName = Productive.version
  template = %General.templatePath%/Productive/version.sql

[external]
  command = grep %Deploy.srcDatabaseSchemaDB% %General.versionFile% | grep -v "\." | awk '{print $2}'
  structure = version UInt32

[Token]
  external = STDIN
  query = <<EOT
SELECT version FROM
(
    SELECT DISTINCT version
    FROM STDIN
)
ALL LEFT JOIN
(
    SELECT DISTINCT
        version,
        1 as exists
    FROM %Target.Productive.version.tableName%
) using version
where exists = 0
EOT

# all settings related to inserting the missing timestamps
[Process]
  query=insert into %Target.Productive.version.tableName% VALUES(0, __RAW__)
  optimize = 1
```

Table names are independent of the Target

External Command becomes a table

All queries are templated and can be change at run time

Each Row returned from Tokens will become a substitue in the Process section.

Each row will spawn it‘s own, independet statement.

Unintented work-around

Problem:

We have a set of customer specific RegEx and a table those RegEx are to be applied to.

If Desc

timestamp	Router	Interface	IfDesc
2022-12-05 14:53:46	R1	12	Customer C1: 1299
2022-12-05 14:53:46	R1	14	BB Links to R2
2022-12-05 14:53:46	R2	452	BB Link to R1
2022-12-05 14:53:46	R2	12	Upstream: 3320
2022-12-05 14:53:46	R2	25	Google GGC

Result
1299
remove
remove
3320
15169

Rules

- `Customer\s*.*?\s*:\s*(\d+)` → use match group as ASN
- `Upstream:\s*(\d+)` → use match group as ASN
- `Google\s+(GGC)` → map GGC to 15169

Token Query Example:

```
SELECT DISTINCT
    Timestamp,  regEx, index
FROM
(
    SELECT DISTINCT
        Timestamp, regEx, index
    FROM
    (
      SELECT DISTINCT
            timestamp,
            %Target.Mangle.linkVRFOverwrite.regEx% as originalRegExArr,
            arrayPushFront(originalRegExArr, '') as regExArr,
            arrayPushFront(arrayEnumerate(originalRegExArr),0 ) as indexArr
      FROM %Target.Data.DMIPKPI.tableName%
      WHERE timestamp >= '__MIN_TS__'
    )
    ALL INNER JOIN
    (
        SELECT DISTINCT
            timestamp
        FROM %Target.Data.DMIPGeneric.tableName%
        WHERE timestamp >= '__MIN_TS__'
    )
    USING timestamp
    ARRAY JOIN
        regExArr as regEx,
        indexArr as index
    UNION ALL
    SELECT DISTINCT
        Timestamp, '' as regEx, 0 as index
    FROM %Target.Data.DMIPKPI.tableName%
    WHERE timestamp >= '__MIN_TS__' AND KPI = 'VRF'
)
ALL LEFT JOIN
(
    SELECT DISTINCT
        timestamp,
        1 as exists
    FROM %Target.Mangle.linkVRFOverwrite.tableName%
) USING timestamp
where exists = 0 AND
      ((length(%Target.Mangle.linkVRFOverwrite.regEx%) > 0) or ((select count() from %Target.Summary.VRFASMapping.tableName%) > 0))
ORDER BY timestamp asc
```

Summary

Problem:

Work with unreliable, incomplete data from diverse data sources at distributed sites with limited internet connectivity.

Our Solution:

Build a pipeline around clickhouse that manages the data and uses the DBMS as its processing engine.

- Automatically takes care of dependecies, scheduling, DB consistency, (re)-processing, etc.
- Completly stateless – all state is stored implicitly in clickhouse
- Data Driven – process data when all sources have reported in. No external synchronization.

But… Is there a better way to do this ?

Thank you !

Backup

Unintented work-around

Problem:
We have a set of customer specific RegEx and a table those RegEx are to applied to.

Match an array of RegEx against a column with

```
multiMatchAny(haystack, [pattern1, pattern2, …, pattern])
```

Find which RegEx matched

```
multiMatchAnyIndex(haystack, [pattern1, pattern2, …, pattern])
multiMatchAllIndex(haystack, [pattern1, pattern2, …, pattern])
```

But – there is no function to extact the match groups from an dynmic/array of patterns

`extract*() functions` need constant patterns !

We can work around this by spawning one **Token** per timestamp/regEx and passing this via the config as a constant into clickhouse.

This spawns (potentially) a lot of queries (`O(nm), n=timestamps, m=RegExs`).