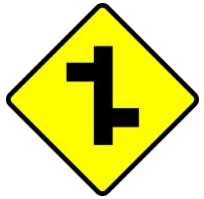
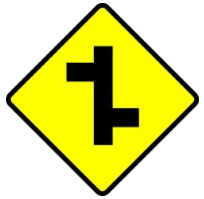


DataJunction

A metrics platform to store and manage metric definitions.



The history of DJ

Beto

Software Engineer @ Preset.io (an Apache Superset SaaS company)

- Apache Superset PMC member
- Not a DJ but plays music
  - Wrote 44 songs last month for FAWM
- beto@preset.io

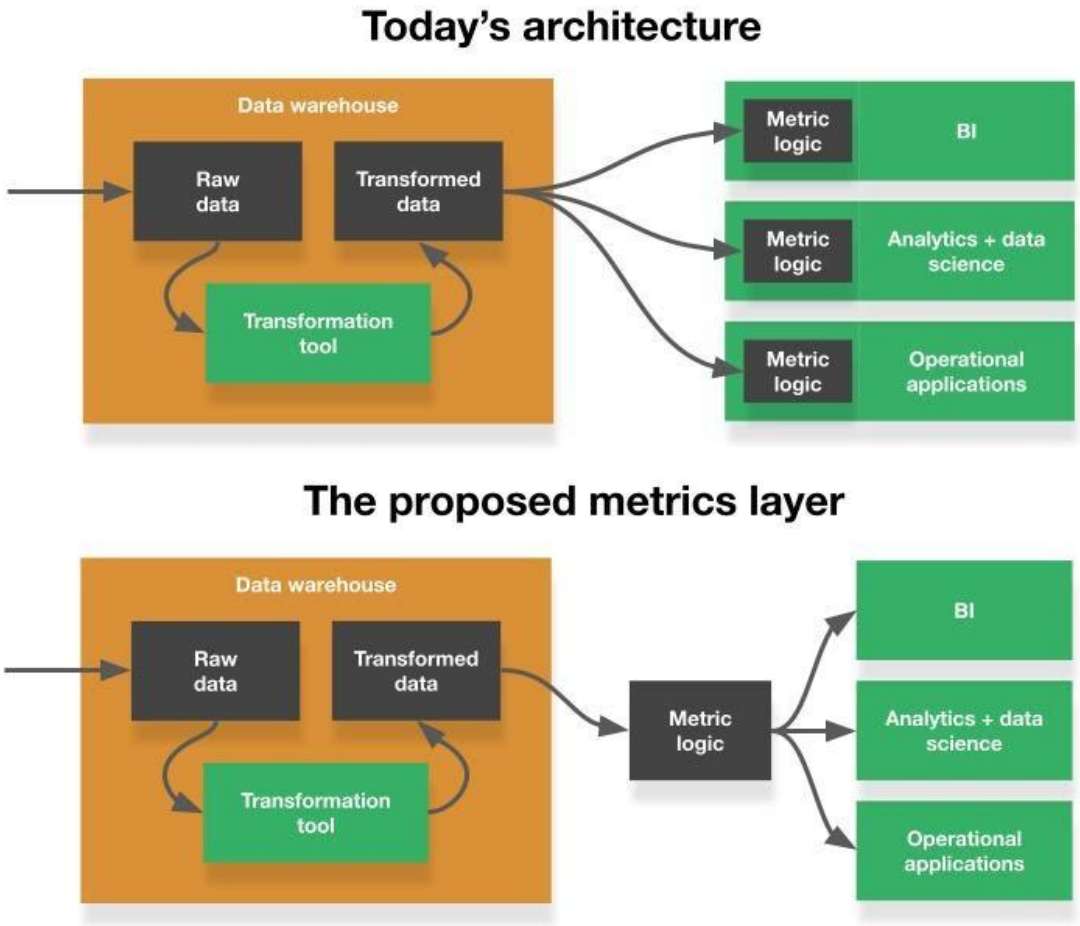
The history of DJ

———

- ➔ Facebook experimentation team (2014?)
  - A/B tests (Deltoid)
- ➔ Later
  - Materialization
    - Transforms and cubes
      - Eg, (country, gender)
      - Additive metrics
  - Anomaly detection
  - Dashboards
  - Growth accounting
  - ?

The history of DJ

- ➔ Netflix (2017) N
  - ◆ Let’s open source DJ!
- ➔ github.com/DataJunction (2018)
  - ◆ 0 commits…
- ➔ The missing piece of the modern data stack (2021)
  - ◆ https://benn.substack.com/p/metrics-layer



"A better architecture would do for metrics what dbt did for transformed data—make them globally accessible to every other tool in the data stack. Rather than each tool defining their own aggregations, the metrics layer is a centralized clearing house for how all metrics are calculated."

Why did it take so long?

➔ 2 strong requirements:
- A robust ANSI SQL parser
- Transpilers to generate DB-specific SQL (Hive, Presto, etc.)

Why did it take so long?

➔ 2 strong requirements:

- A robust ANSI SQL parser -> **sqloxide**
- Transpilers to generate DB-specific SQL (Hive, Presto, etc.)

Why did it take so long?

➔ 2 strong requirements:

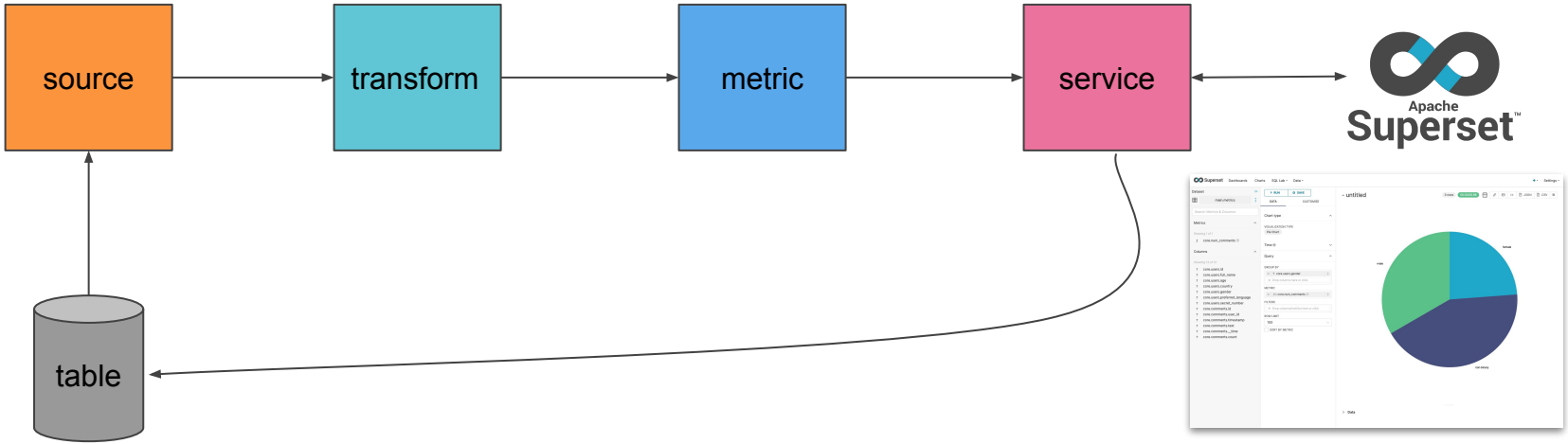
- A robust ANSI SQL parser -> **sqloxide**
- Transpilers to generate DB-specific SQL (Hive, Presto, etc.)
  i. Generate a **SQLAlchemy** query object
  ii. Use **dialects** to transpile to DB-specific SQL

Why SQL?

- ➔ SQL is like sharks 🦈
  - Sharks are older than trees
  - Sharks are older than the rings of Saturn, actually
  - (Sharks are also older than SQL)
  - Sharks have been around for 450 Million years because they’re good at what they do
- ➔ SQL has been around for almost 50 years for the same reason
  - Declarative
  - “Standardized”

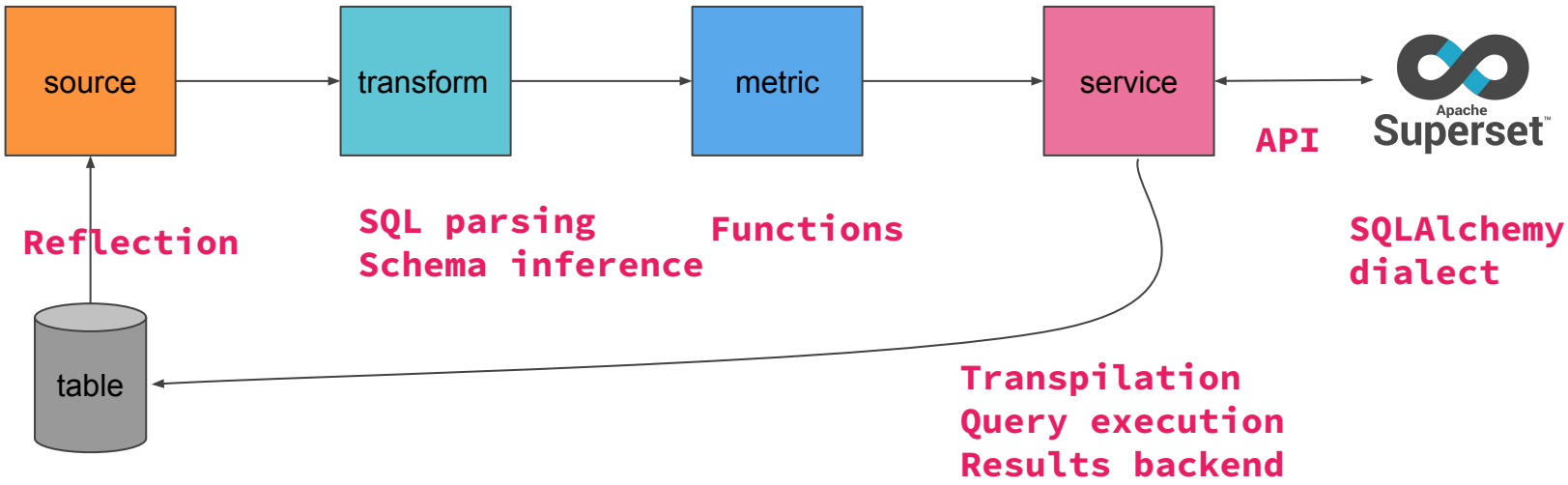
First proof of concept

➔ End-to-end demo: Superset consuming DJ metrics

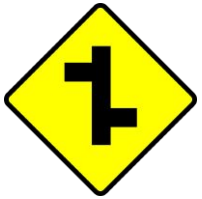


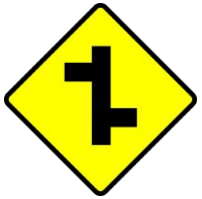
First proof of concept

➔ End-to-end demo: Superset consuming DJ metrics



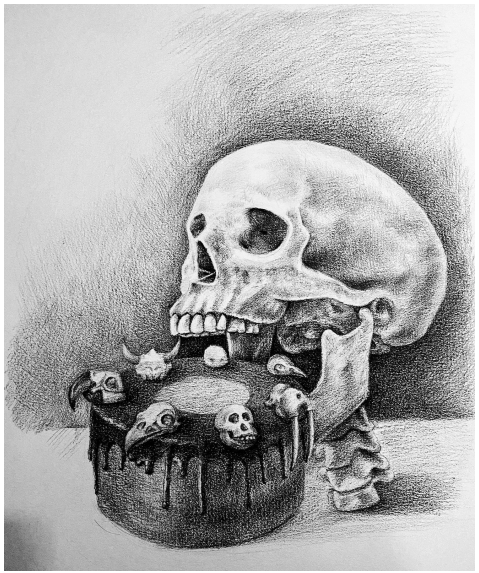
Demo!



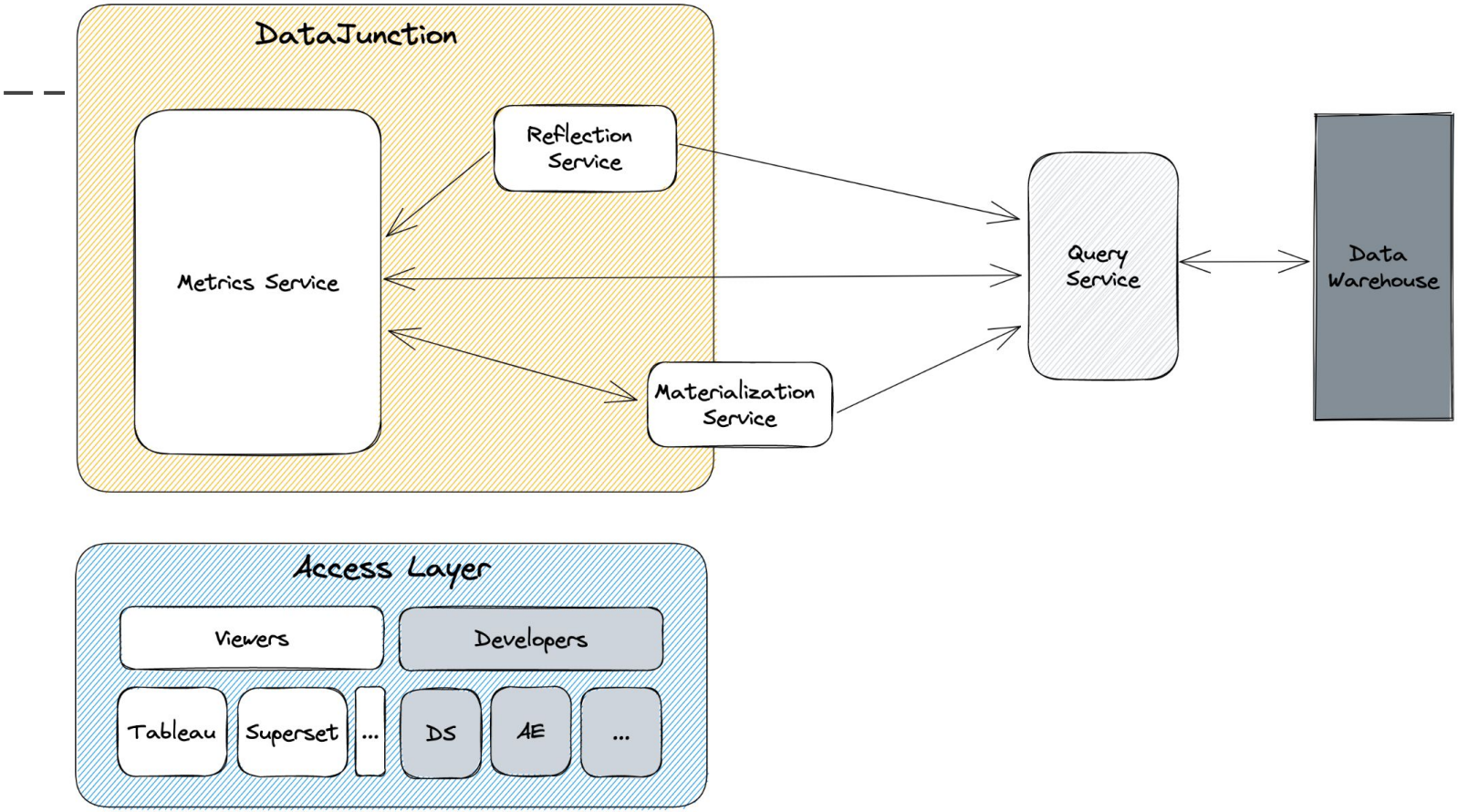


Yian

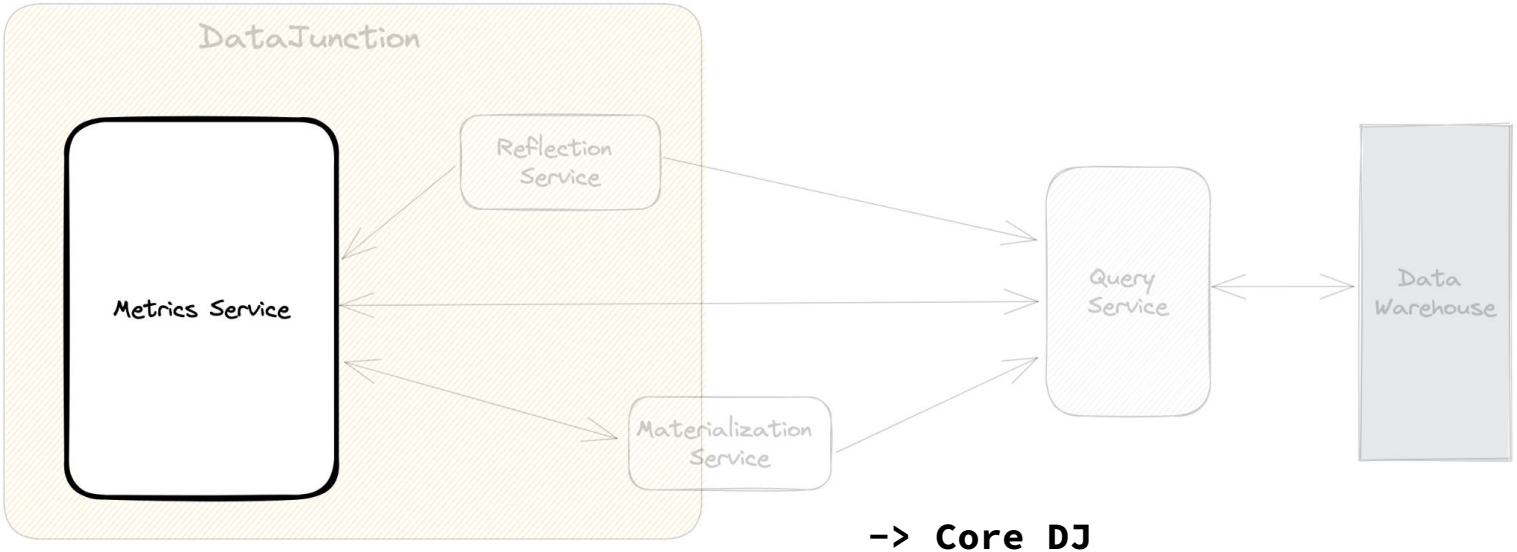
Software Engineer at Netflix (XP)

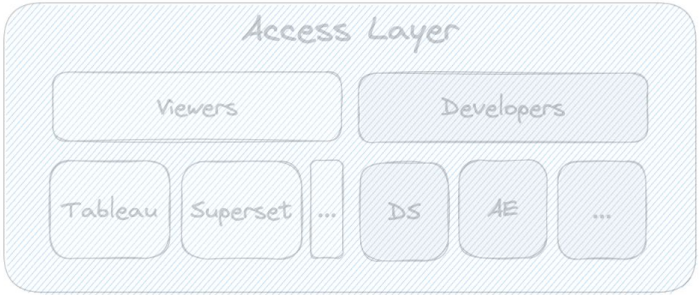


Architecture Overview



Architecture Overview: Metrics Service



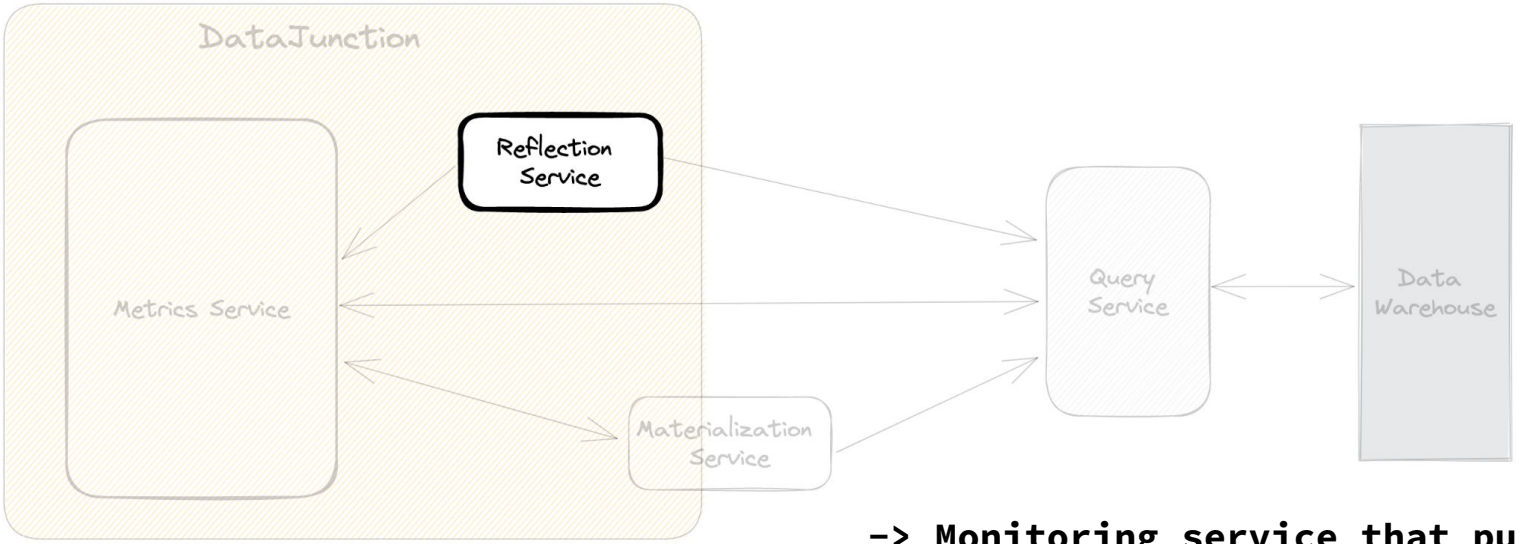


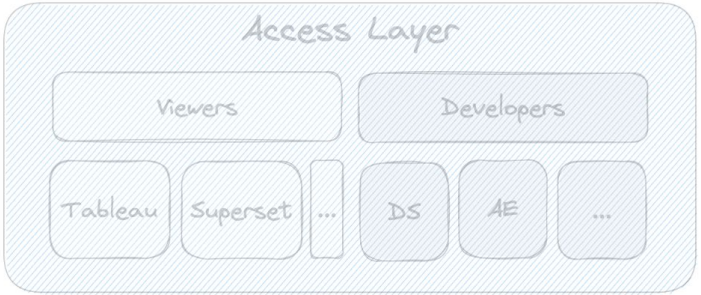
-> Core DJ

-> All nodes stored in a database with support for versioning and rollbacks.

-> Provides APIs for creating, updating and introspection of the node DAG

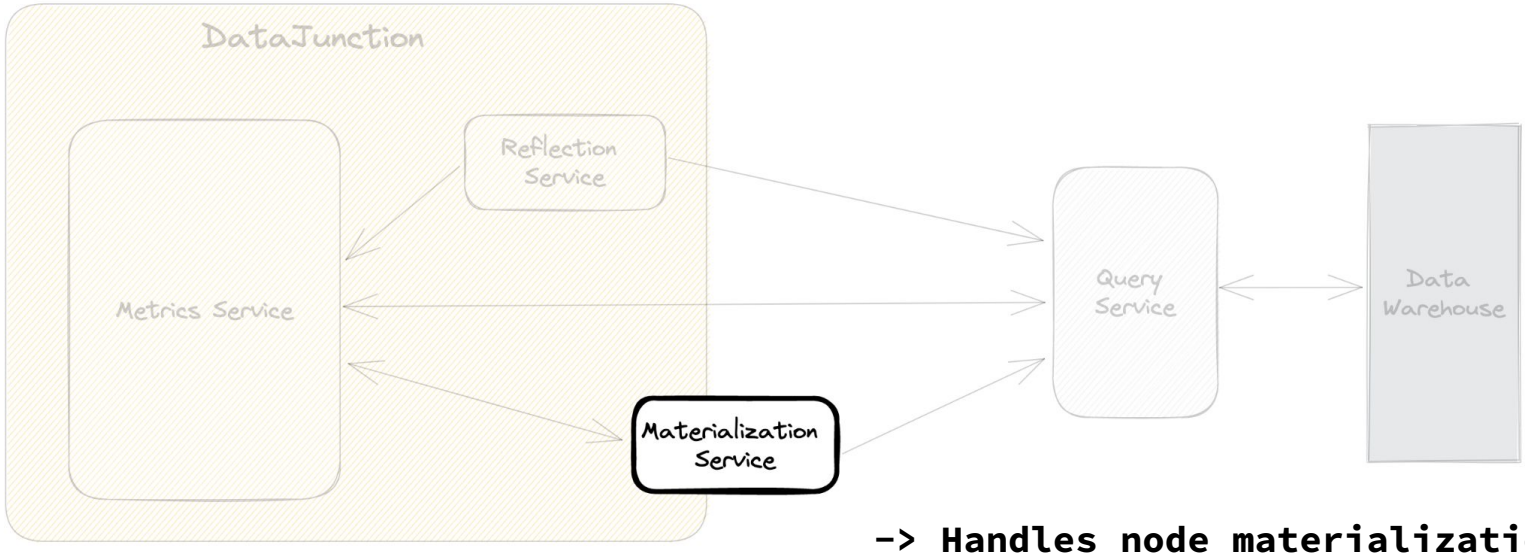
Architecture Overview: Reflection Service



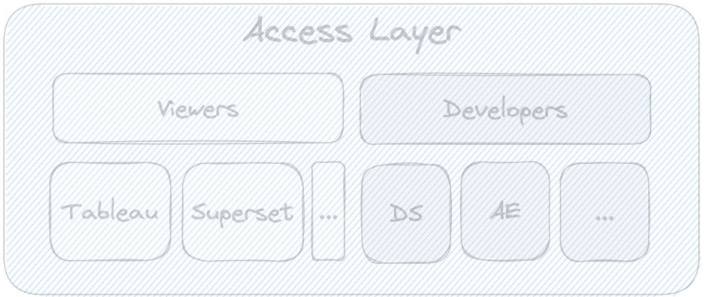


-> Monitoring service that pushes info about state of external tables to DJ.
-> Keeps schemas of source nodes in sync with actual tables.
-> Breaking changes are identified and communicated to the downstream nodes

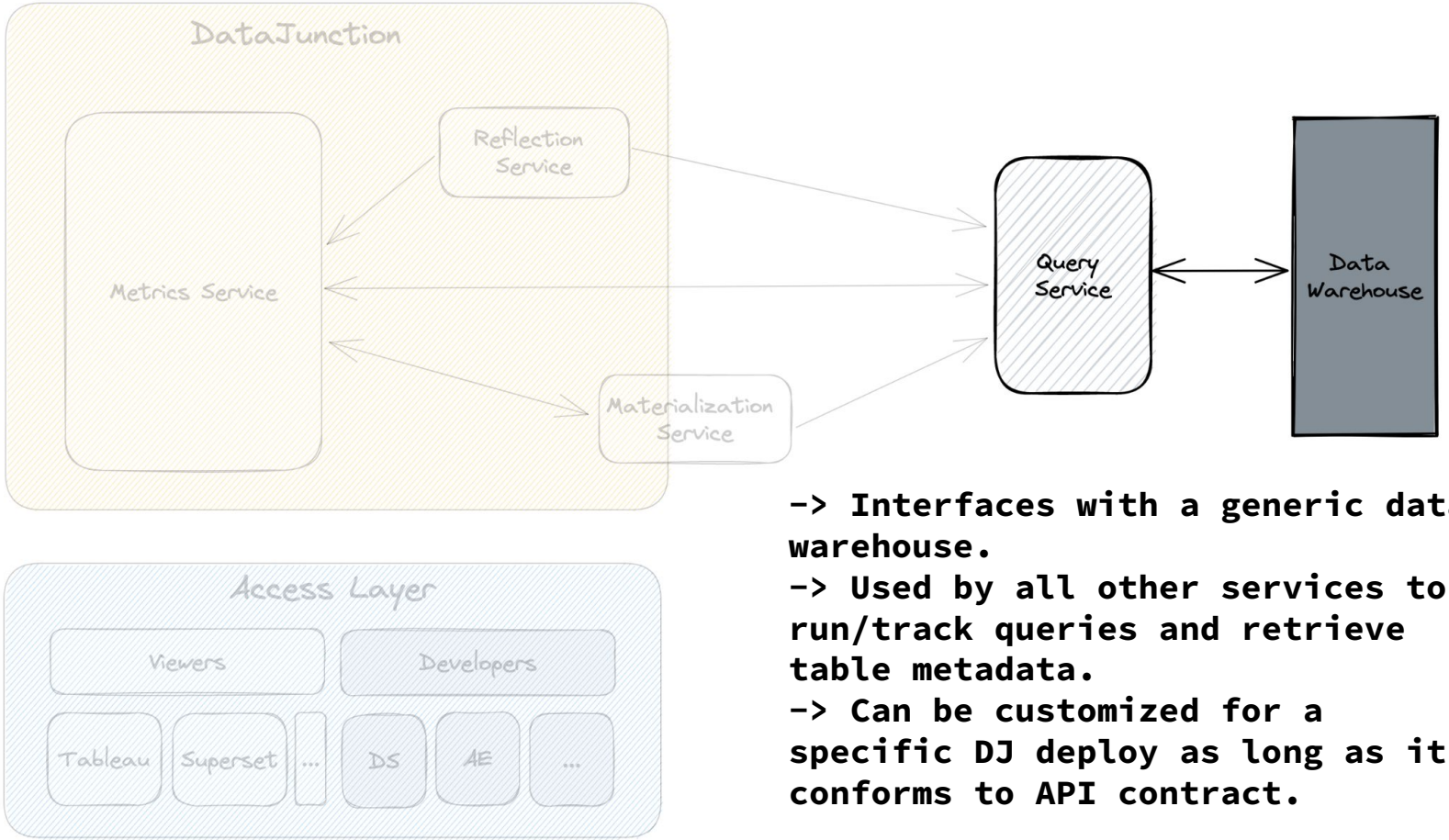
Architecture Overview: Materialization Service



-> Handles node materialization

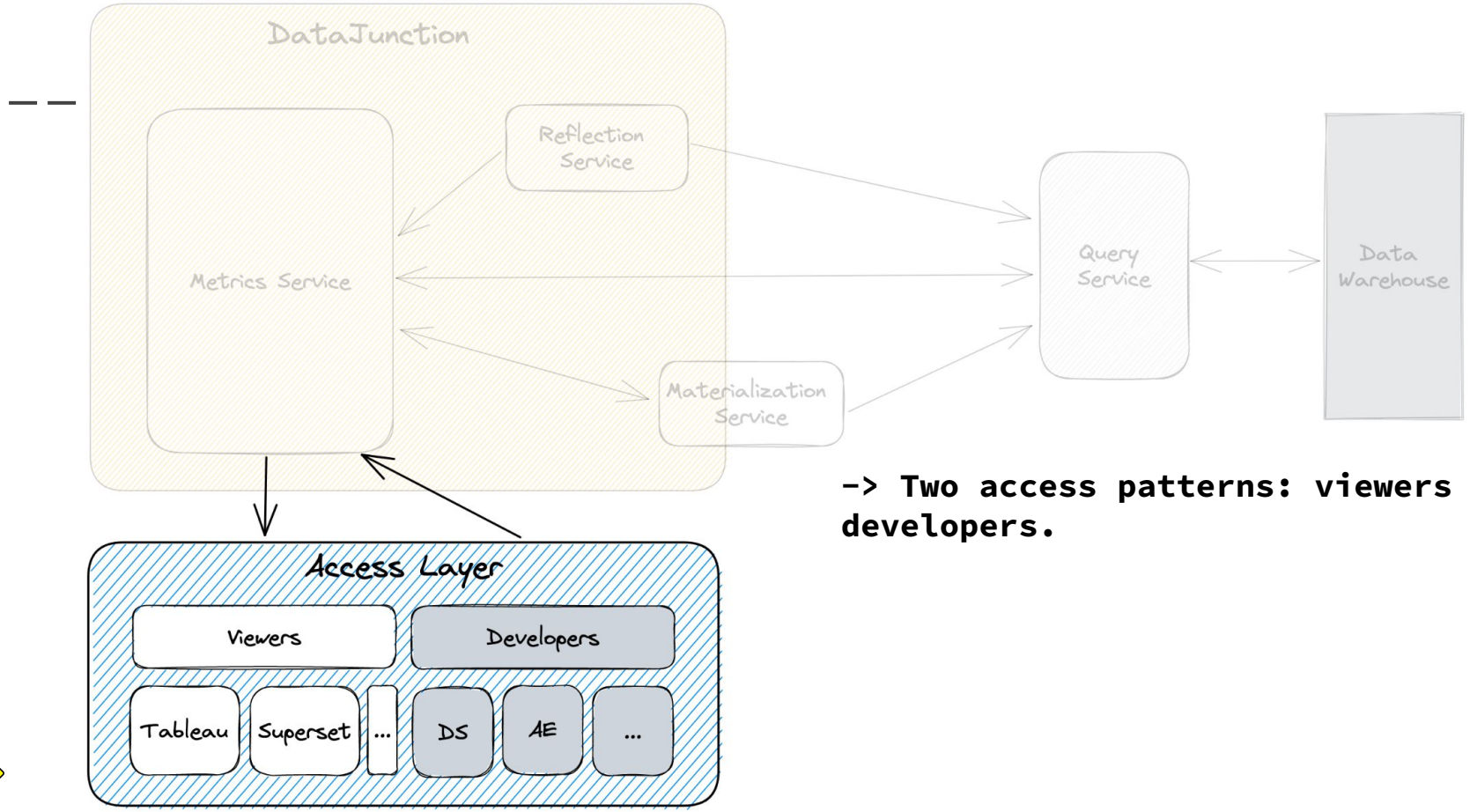


Architecture Overview: Query Service



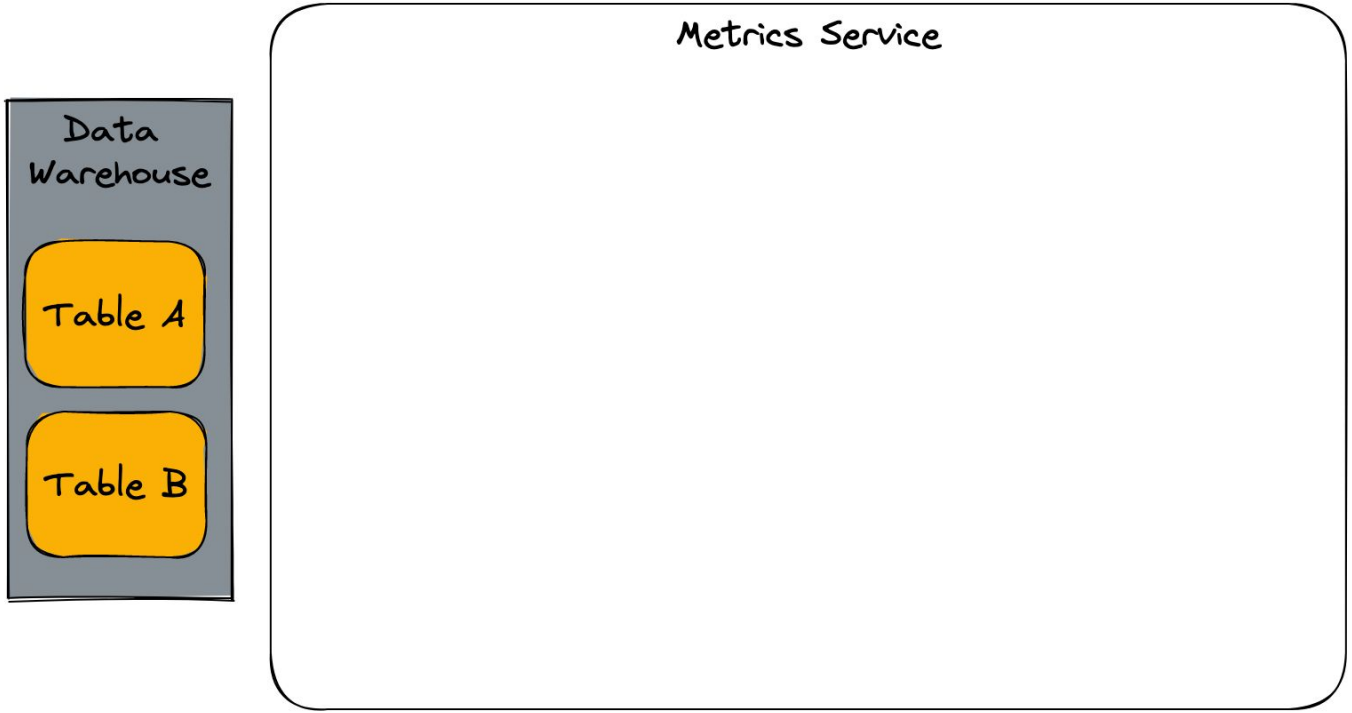
-> Interfaces with a generic data warehouse.
-> Used by all other services to run/track queries and retrieve table metadata.
-> Can be customized for a specific DJ deploy as long as it conforms to API contract.

Access Layers

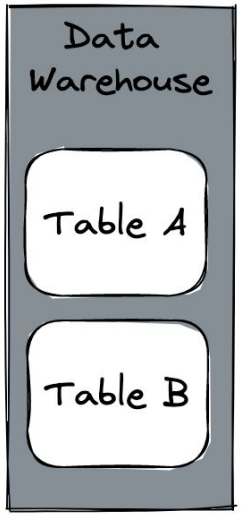


-> Two access patterns: viewers vs developers.

Contribution Flow: Source Nodes



Contribution Flow: Source Nodes



Two API calls

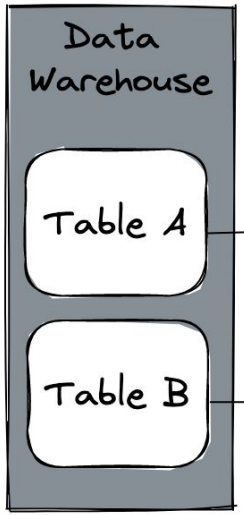
(1) Create the source node

Source Node A

Source Node B

```
curl -X 'POST' \
  'http://dj:8000/nodes/' \
  -H 'Content-Type: application/json' \
  -d '{
  "name": "views",
  "display_name": "Views",
  "description": "Views events",
  "columns": {},
  "mode": "published",
  "type": "source"
}'
```

Contribution Flow: Source Nodes



Two API calls

(2) Link it to a real table.

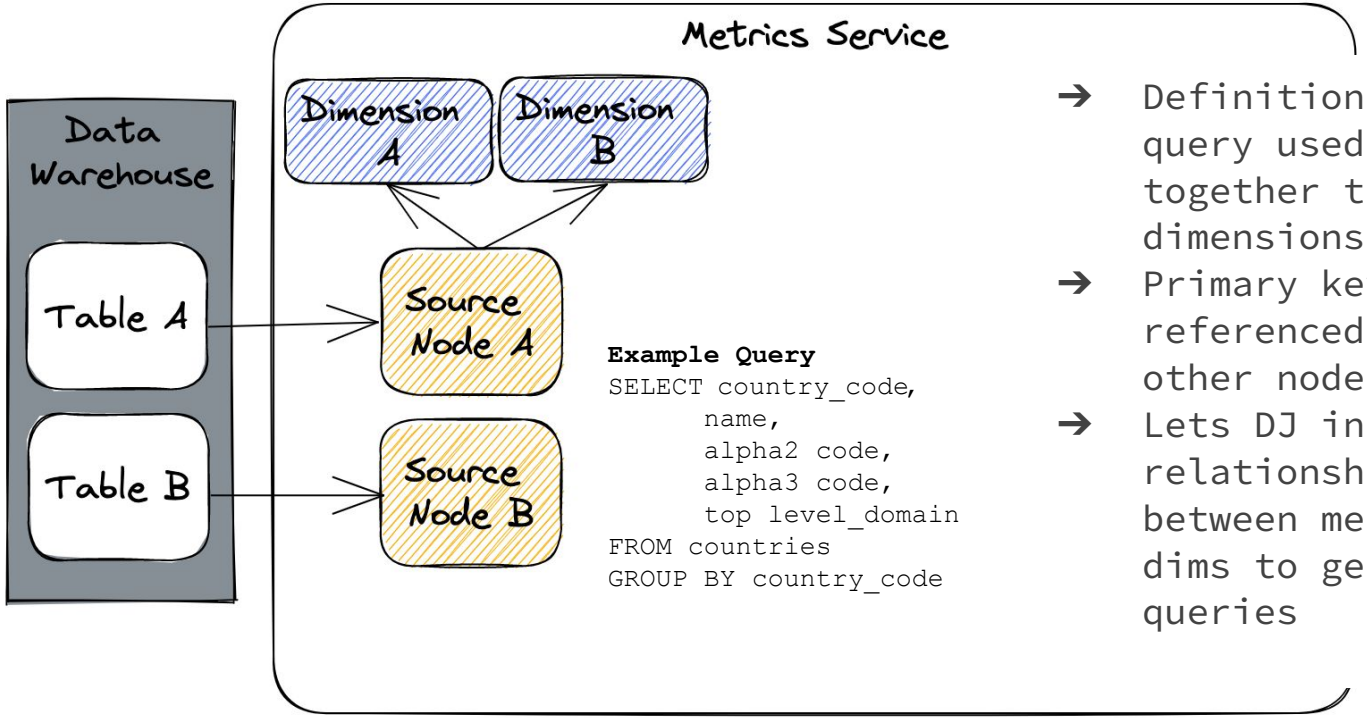
```
curl -X 'POST' \
  'http://dj:8000/nodes/views/table/' \
  -H 'Content-Type: application/json' \
  -d '{
  "catalog_name": "prod",
  "schema": "example",
  "table": "views",
  "columns": []
}'
```

Source Node A

Source Node B

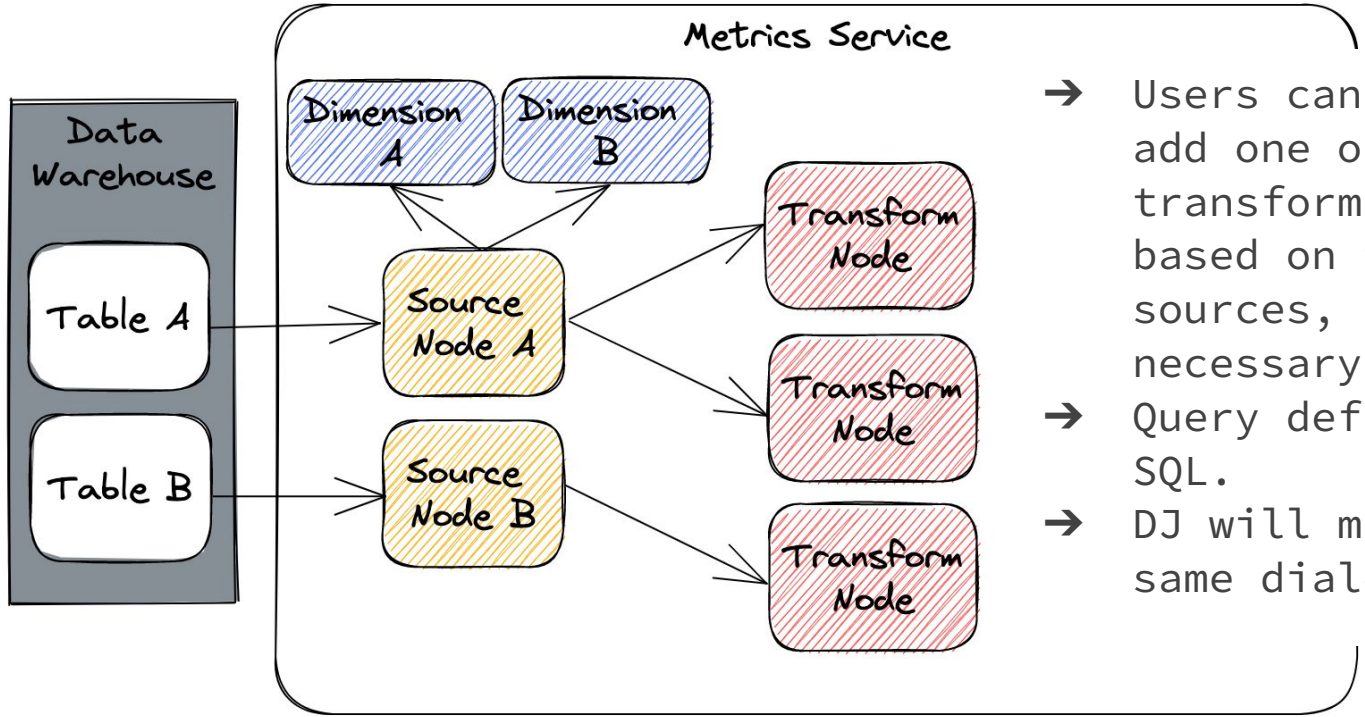
The source node's columns will be populated based on the actual table.

Contribution Flow: Dimension Nodes



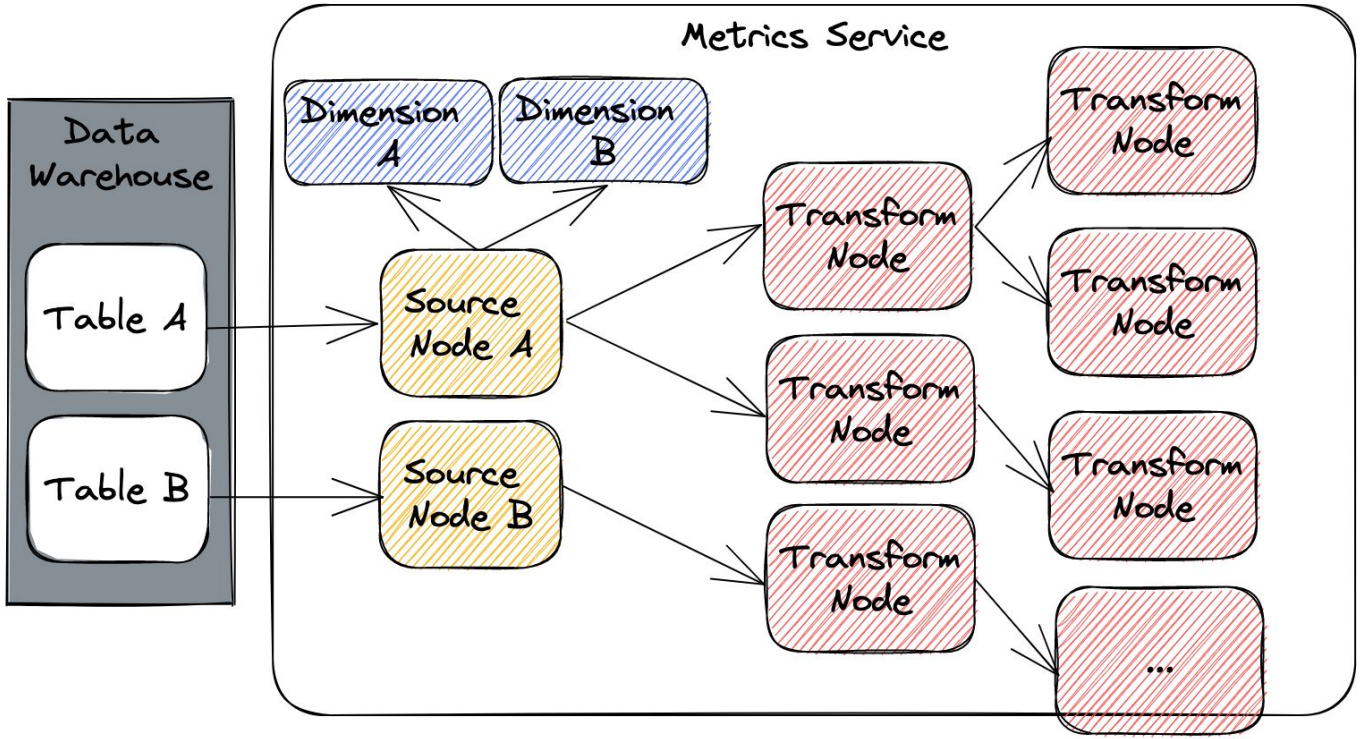
- Definition includes query used to bring together the dimensions dataset.
- Primary key (can be referenced from other nodes)
- Lets DJ infer relationships between metrics and dims to generate queries

Contribution Flow: Transform Nodes



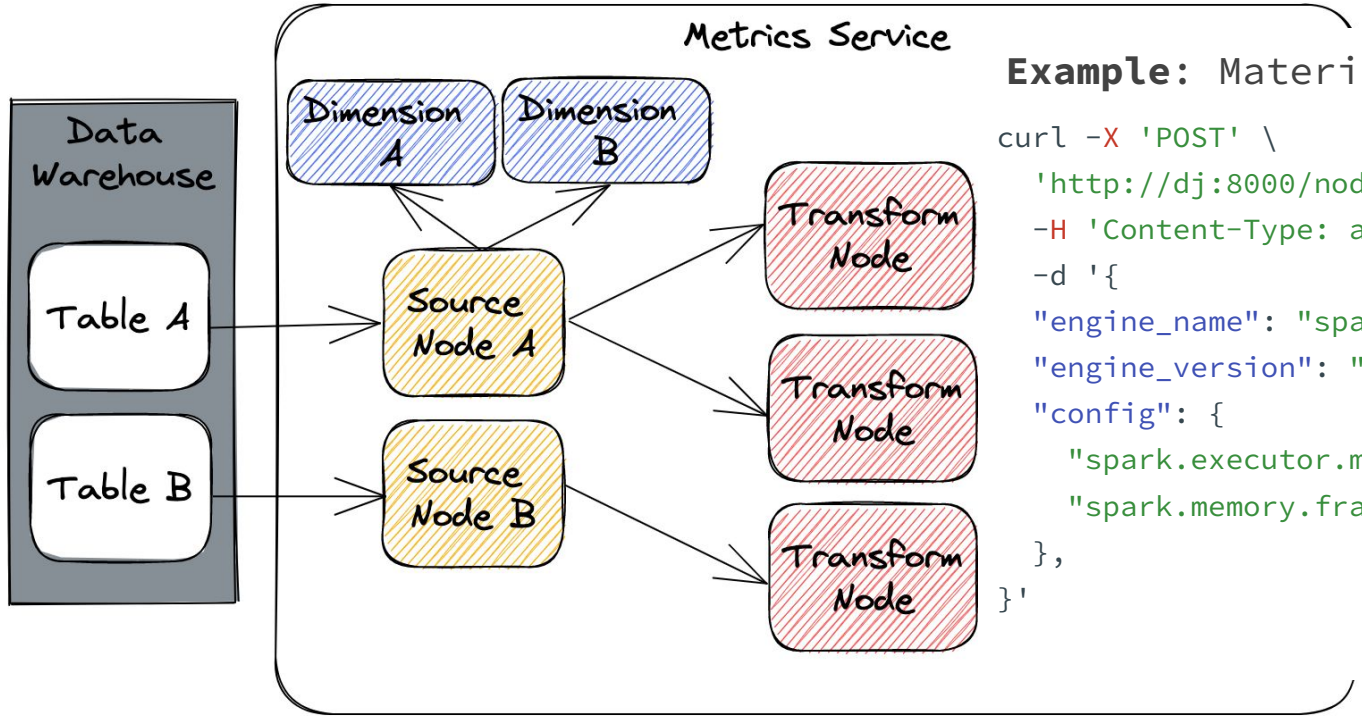
- ➔ Users can choose to add one or more transform nodes based on the sources, if necessary.
- ➔ Query defined in SQL.
- ➔ DJ will maintain same dialect.

Contribution Flow: Transform Nodes



- Provides flexibility.
- As many layers as needed.
- Can reuse others' transforms when convenient.
- Can control how the transform is materialized by setting the materialization configuration.

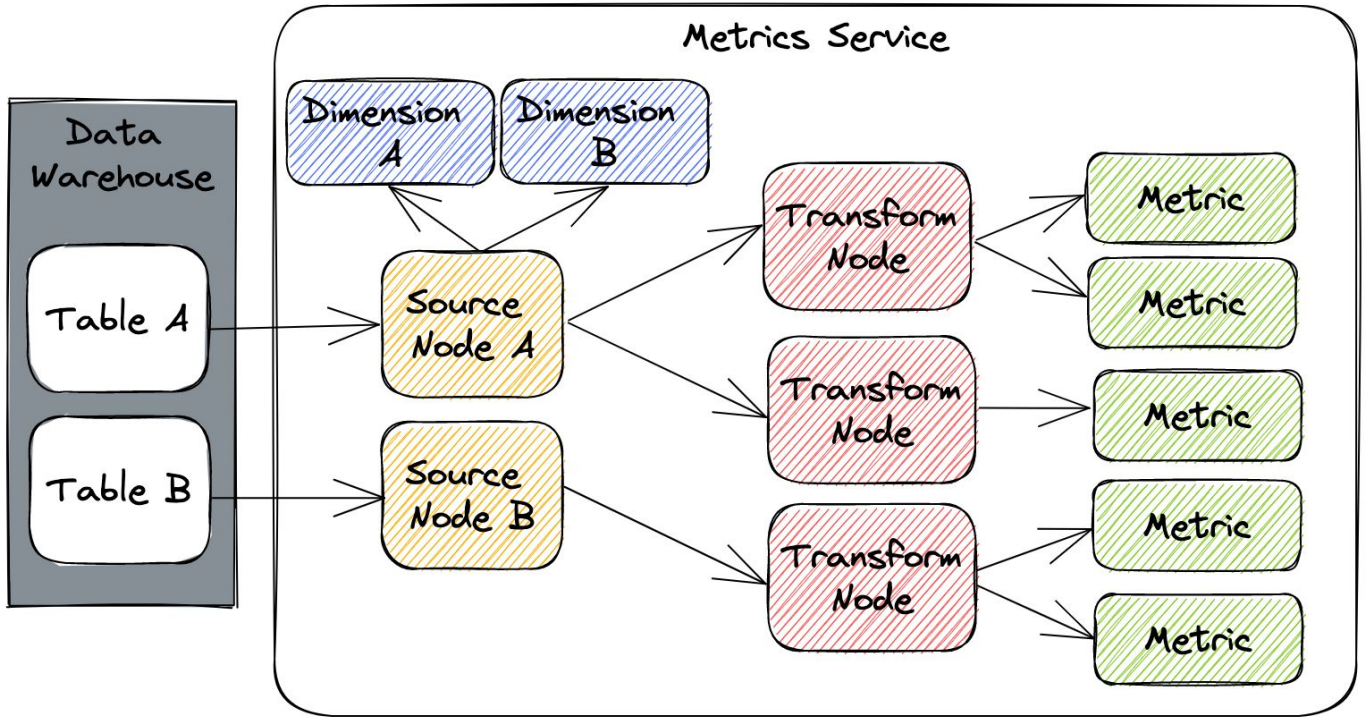
Contribution Flow: Transform Nodes



Example: Materialization Config

```
curl -X 'POST' \
  'http://dj:8000/nodes/views_agg/table/' \
  -H 'Content-Type: application/json' \
  -d '{
  "engine_name": "spark",
  "engine_version": "3.1.2",
  "config": {
    "spark.executor.memory": "4g",
    "spark.memory.fraction": "0.6", ...
  },
}'
```

Contribution Flow: Metrics



- An aggregation of columns on an existing upstream node
- Dimensions on the upstream nodes are used to infer available dims for metric

Consumption: Metrics

➔ In addition to the regular node CRUD API endpoints, some additional endpoints specific to metrics:

- **GET** `/metrics/{name}/sql/`
  - Generates SQL for a metric based on the selected filters and dimensions
- **GET** `/metrics/{name}/data/`
  - Runs the generated SQL and returns the metrics data

Consumption: Metrics

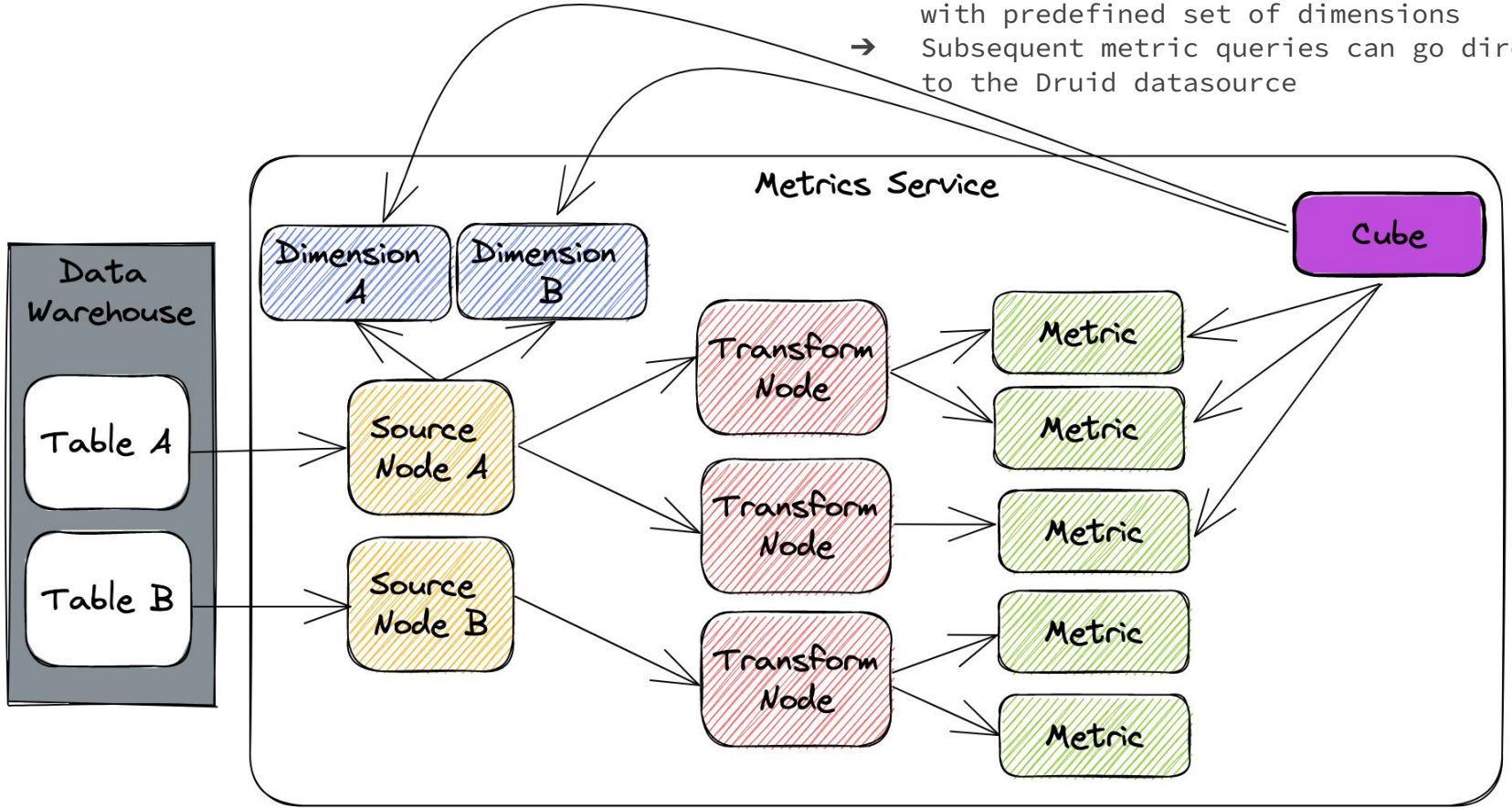
Generated SQL:

```
SELECT
   <dimension A>,
   <dimension B>, ...
   <measure A>,
   <measure B>, ...
FROM <upstream node>
WHERE <filter expression> AND ...
GROUP BY <dimension A>, <dimension B>, ...
```

Contribution Flow: Cubes

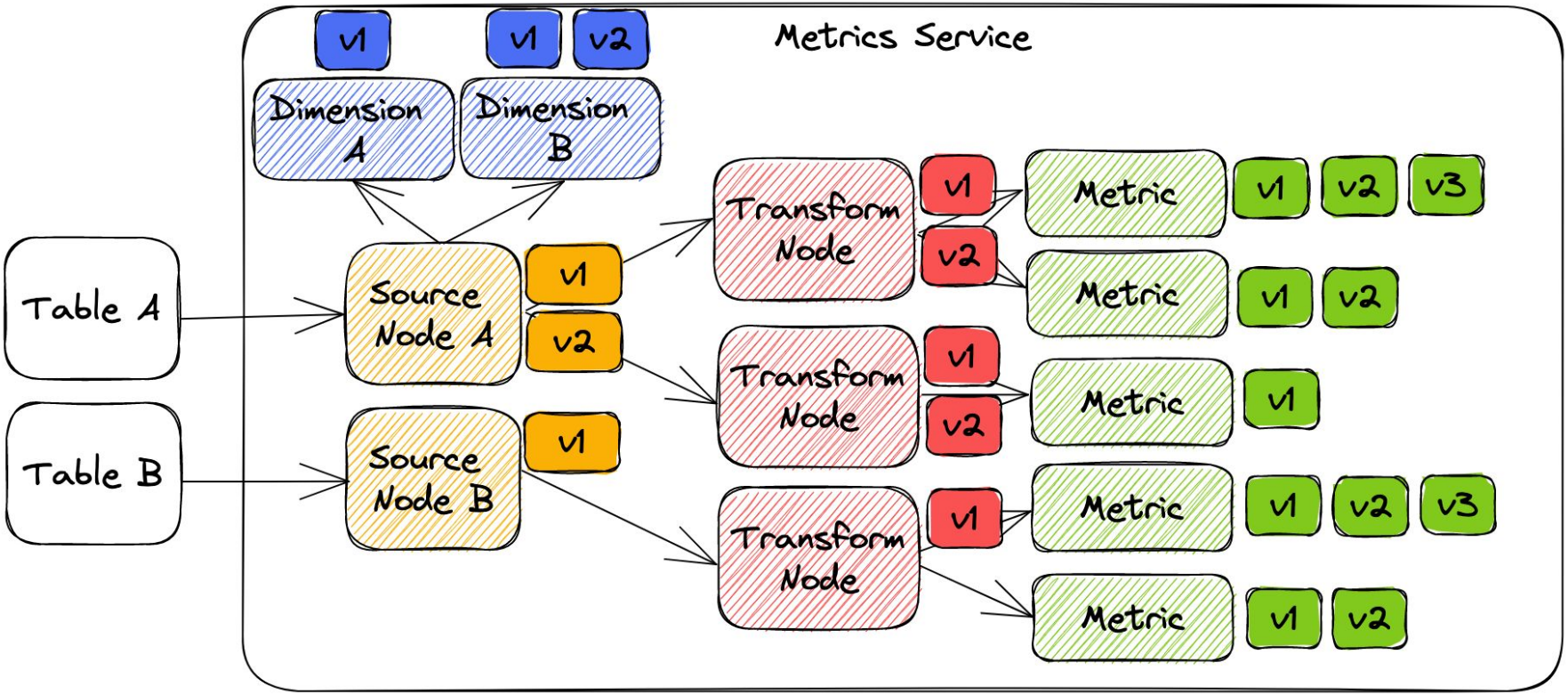
- A cube is a multi-dimensional set of metrics.
- Primary benefit: materialization to Druid with predefined set of dimensions
- Subsequent metric queries can go directly to the Druid datasource

.

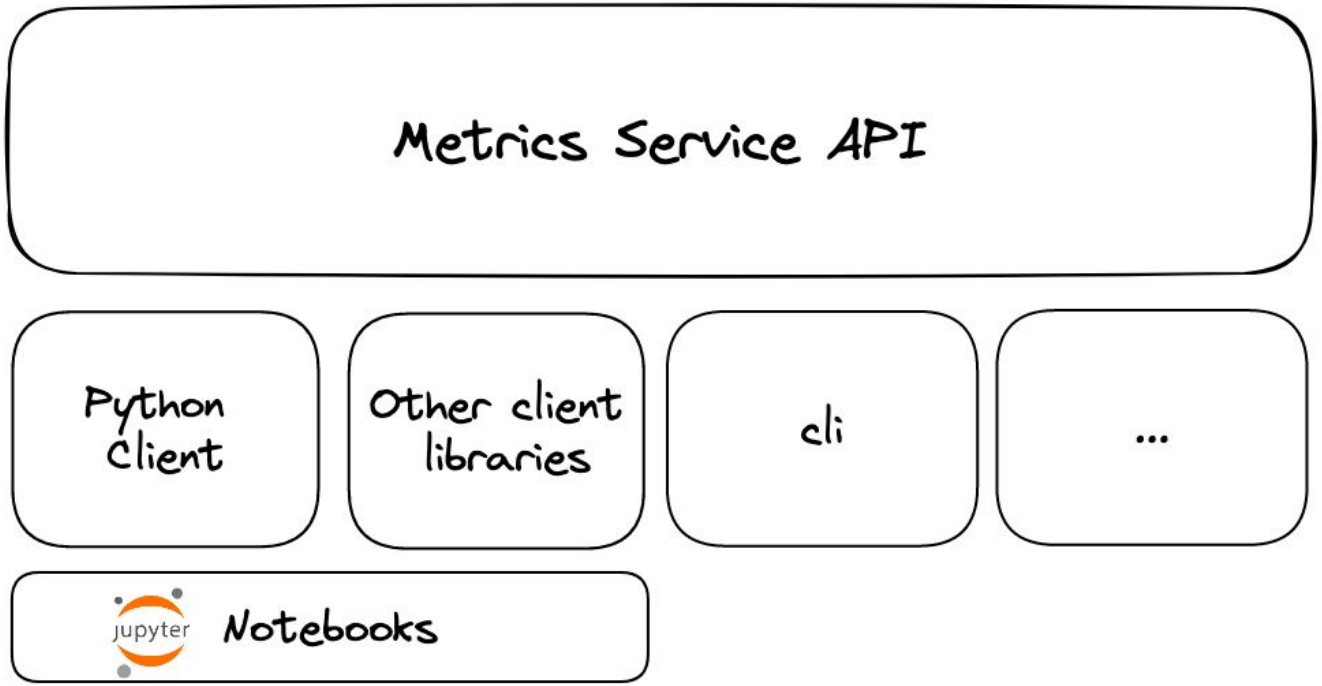


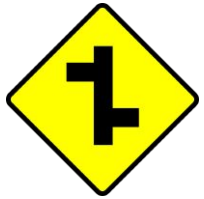
Consumption: Versioning

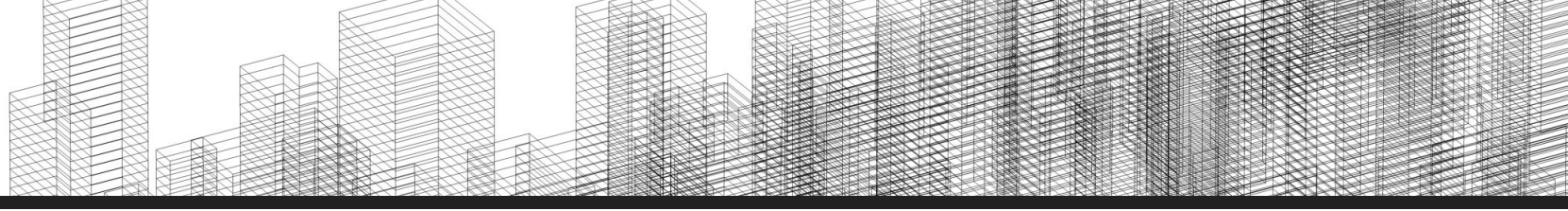
-



Client Support







DJ-SQL Parsing and Query Construction

How DataJunction Constructs Arbitrarily Complex Queries from your SQL definitions

Nick

Director, AI & ML PI Research @Travelers

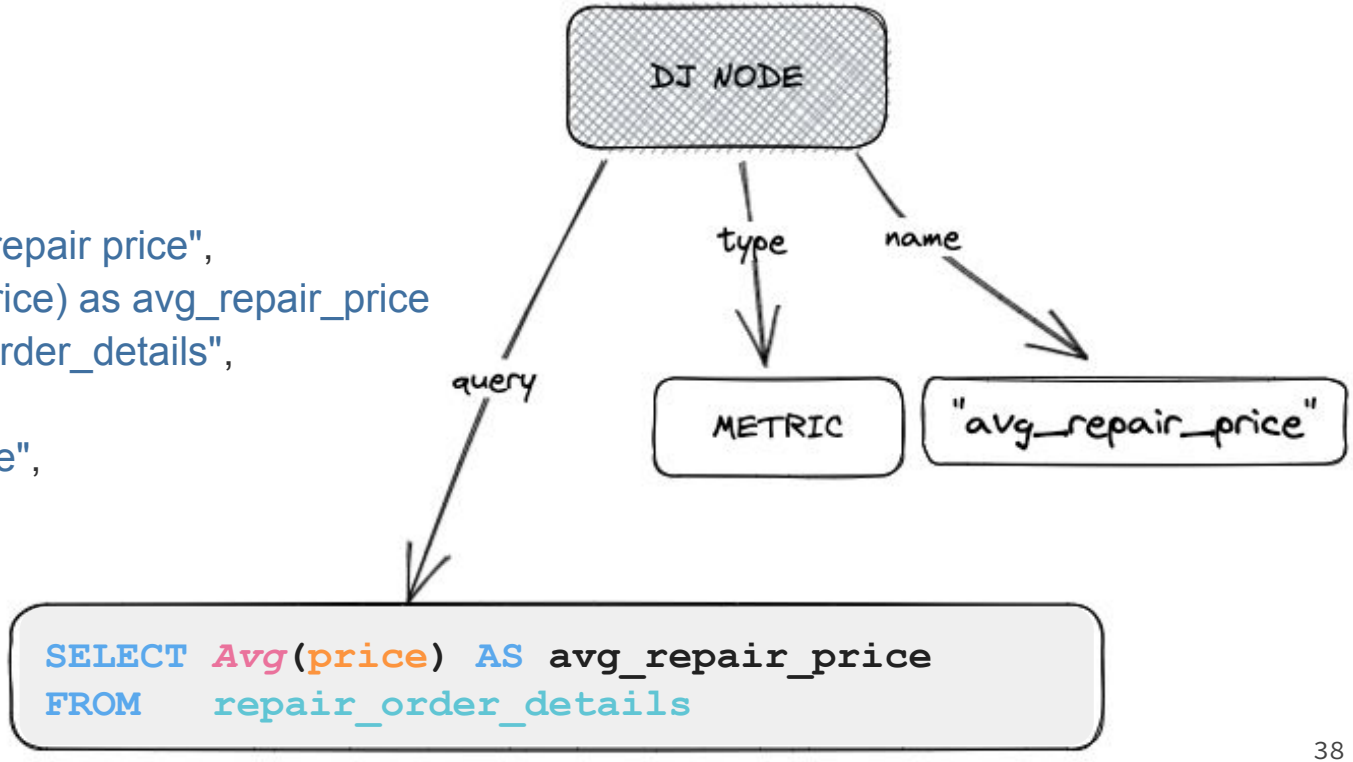
You give DJ SQL. DJ gives you… SQL?

- ➔ DJ gives you familiar and ergonomic ways to model your data
  - Tell DJ **SOURCE**s
    - **TRANSFORM** data with SQL
    - Define **DIMENSION**s with SQL
    - Tell DJ **METRIC**s with SQL
- ➔ DJ parses SQL *exactly* as written, representing it as its own AST. From the AST, DJ can…
  - Give back the queries as written
  - Incorporate DJ metadata into queries
  - Optimize queries given underlying data sources
  - Combine queries to compute metrics
  - Determine query similarity

Creating A Metric with SQL

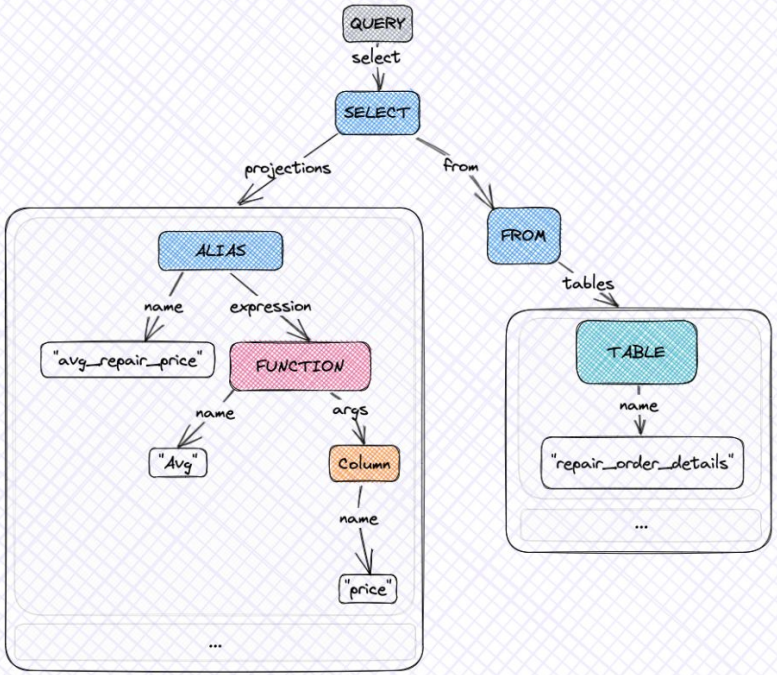
POST dj/nodes/

```
{
   "description": "Average repair price",
   "query": "SELECT Avg(price) as avg_repair_price
             FROM repair_order_details",
   "mode": "published",
   "name": "avg_repair_price",
   "type": "metric",
}
```

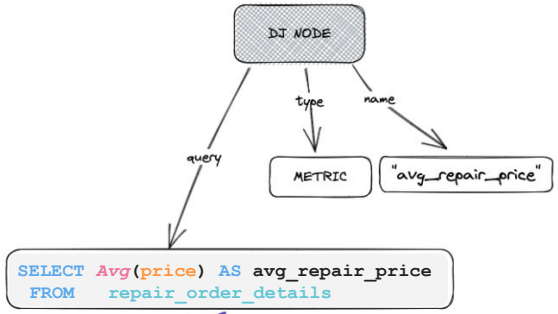


PARSING

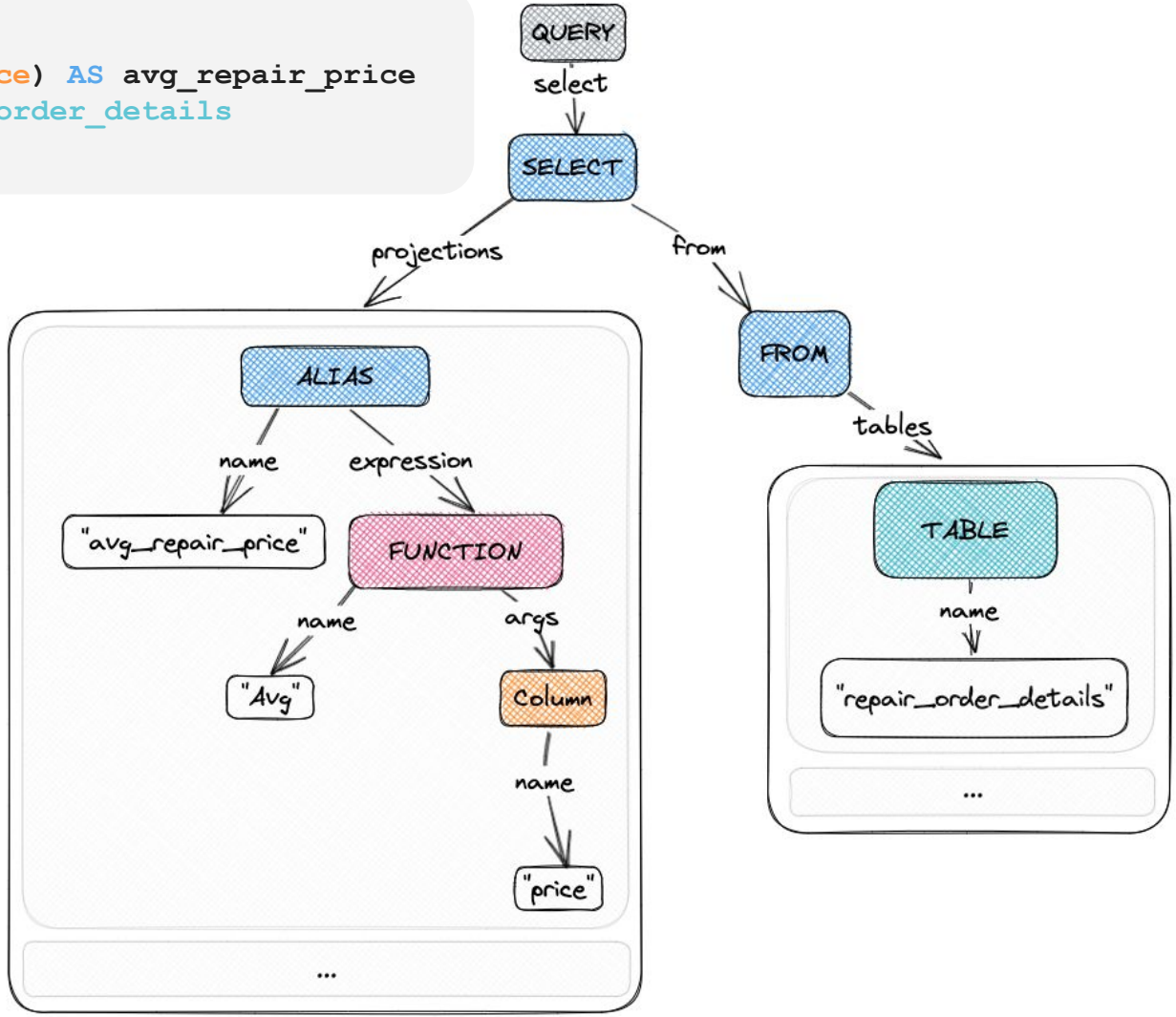
AST SPACE



METADATA SPACE



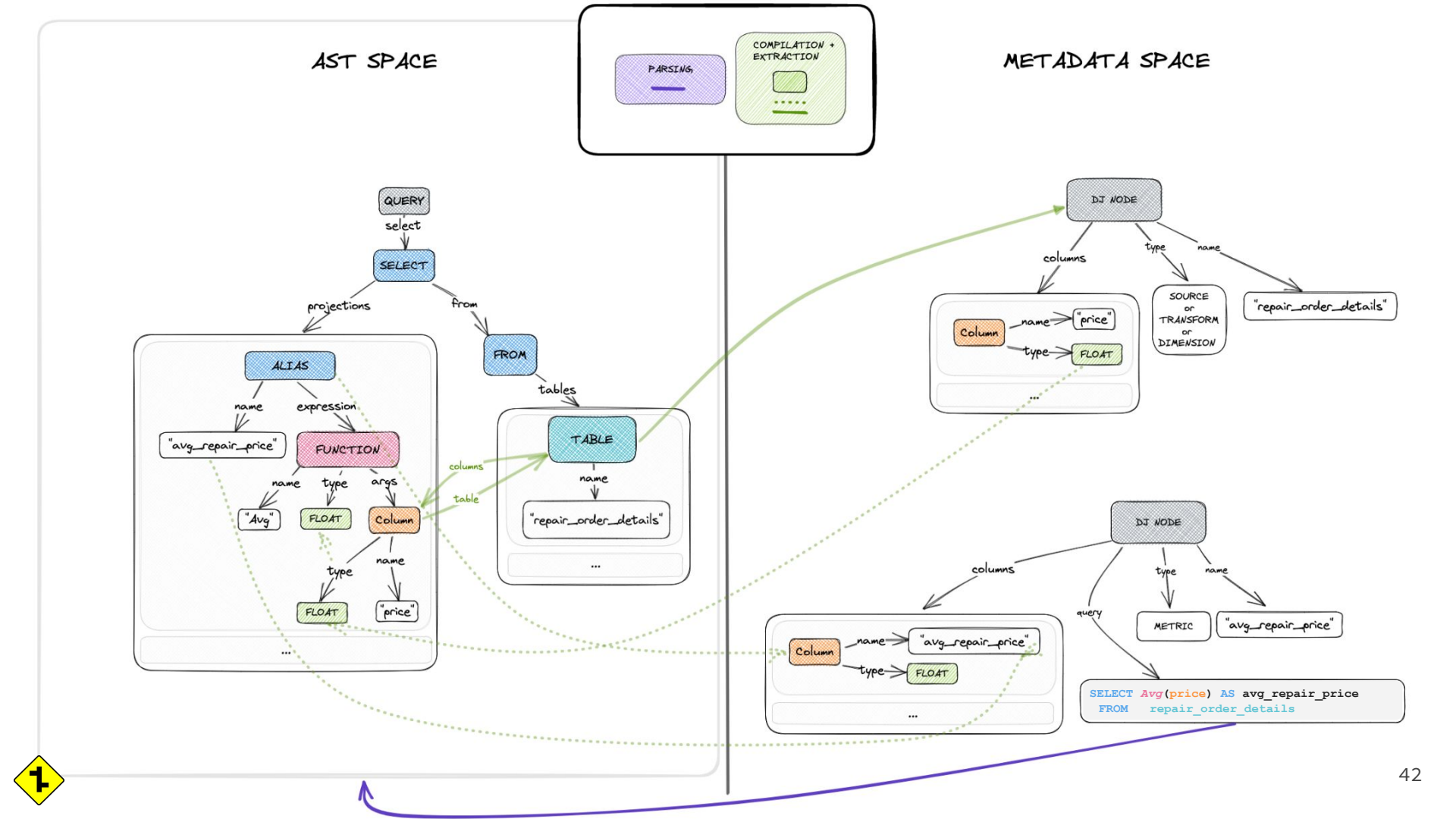
```sql
SELECT Avg(price) AS avg_repair_price
FROM   repair_order_details
```



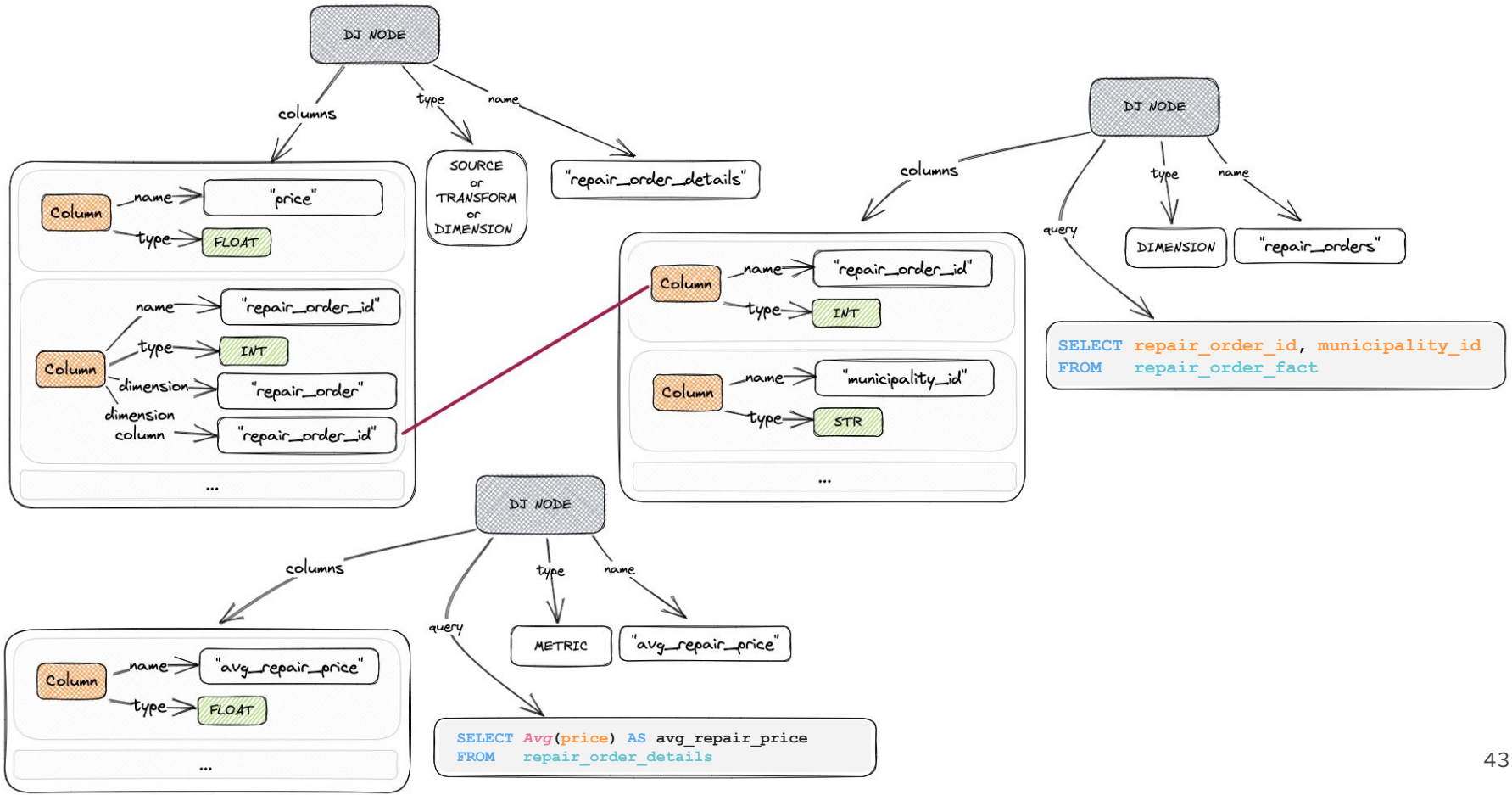
Compilation & Extraction: Incorporating DJ Metadata Into Queries

---

- ➔ Once DJ has parsed queries into ASTs it can take steps to validate and incorporate DJ metadata into them:
  - DJ finds the DJ Nodes that tables represent
  - DJ deduces what table expressions that column expressions originate from
  - So, DJ infers columns and their types - whether they exist unambiguously and if compound expressions are valid
- ➔ With all of this information baked into the AST, it is easy to see what dimensions are valid to be applied to any given query

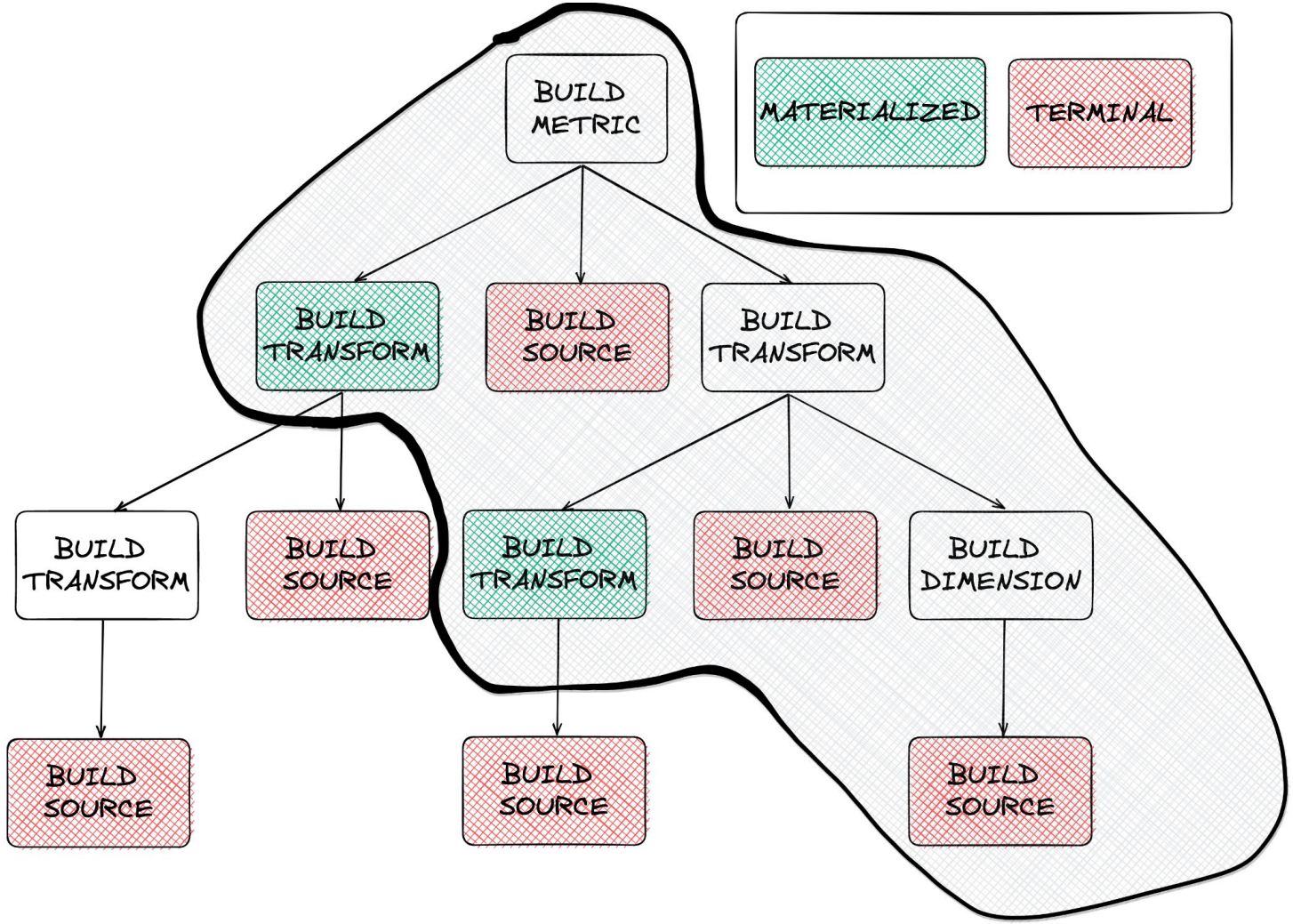

AST SPACE
PARSING
COMPILATION + EXTRACTION
METADATA SPACE
QUERY
select
SELECT
projections
from
FROM
ALIAS
name
expression
"avg_repair_price"
FUNCTION
name
type
args
"Avg"
FLOAT
Column
type
name
FLOAT
"price"
...
tables
TABLE
name
"repair_order_details"
...
columns
table
DJ NODE
columns
type
name
Column
name
"price"
type
FLOAT
...
SOURCE or TRANSFORM or DIMENSION
"repair_order_details"
DJ NODE
columns
type
name
query
Column
name
"avg_repair_price"
type
FLOAT
...
METRIC
"avg_repair_price"
SELECT Avg(price) AS avg_repair_price
FROM repair_order_details

METADATA SPACE



Build Planning: Optimizing Your Queries

- DJ tracks all of the valuable information derived in the compile and extraction.
- This information is used downstream while determining the optimal way to surface data for you.
- Since you can tell DJ of materializations of nodes - even intermediate nodes such as transforms and dimensions that might otherwise be derived in raw SQL from your sources - you can also specify parameters for how to leverage them while designing how your metric will be derived.
- This means understanding all of the routes that could be explored to create a viable query and choosing the optimal route that fits within given constraints

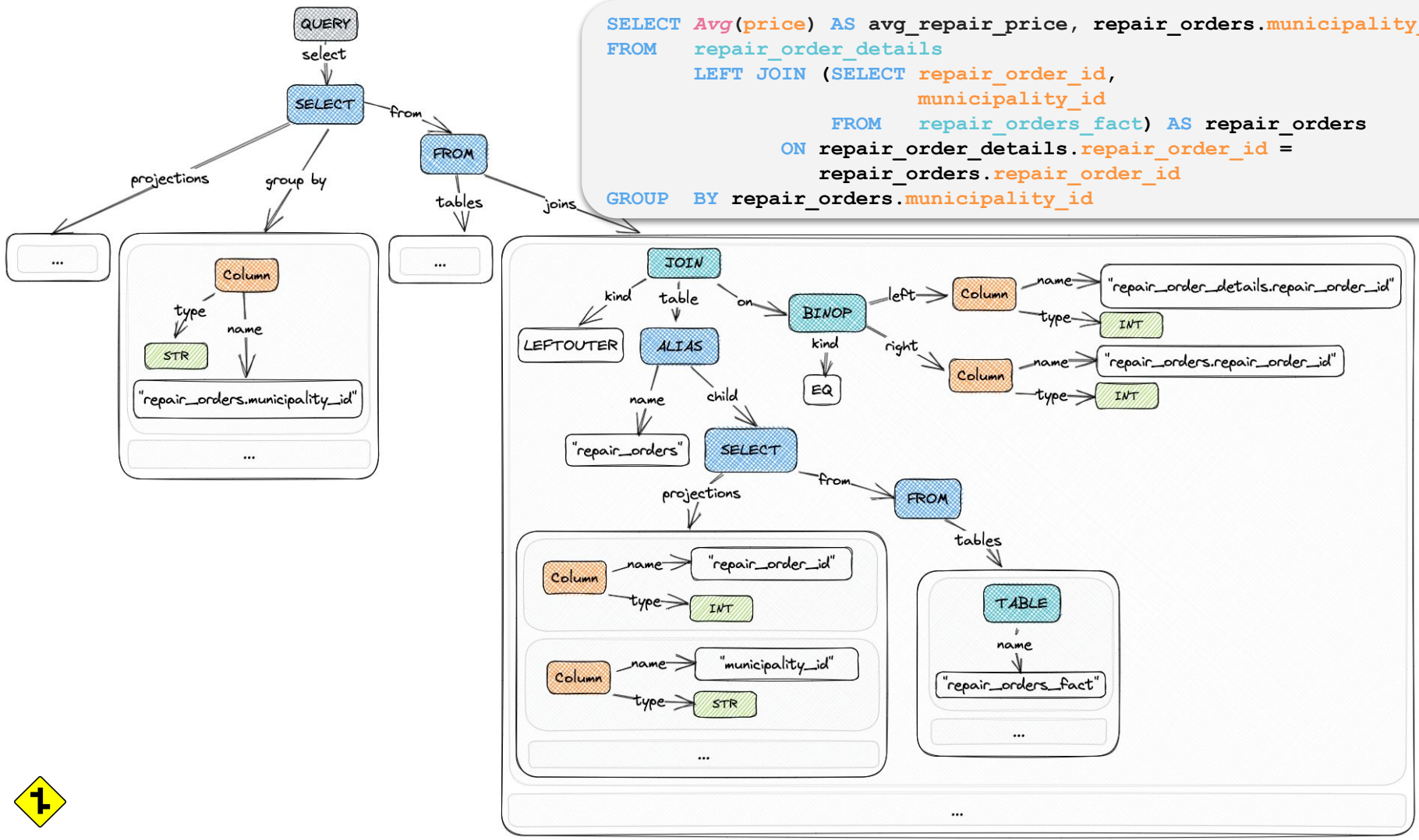

BUILD METRIC
MATERIALIZED
TERMINAL
BUILD TRANSFORM
BUILD SOURCE
BUILD TRANSFORM
BUILD TRANSFORM
BUILD SOURCE
BUILD TRANSFORM
BUILD SOURCE
BUILD DIMENSION
BUILD SOURCE
BUILD SOURCE
BUILD SOURCE

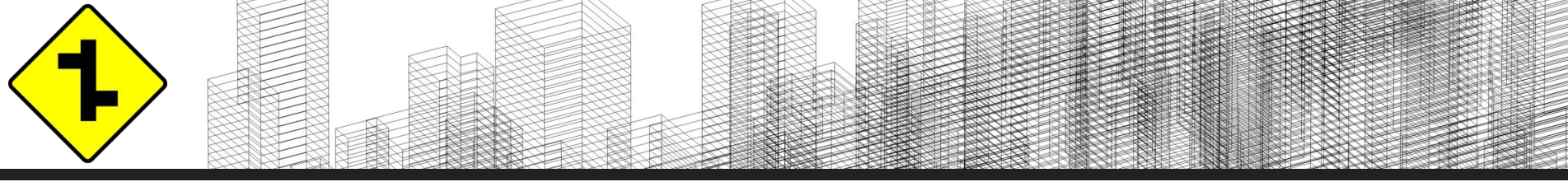
Build: Deriving Meaning From Your Queries

- Finally, once we have a route in mind to build a query, there is the matter of actually making a valid query.
- Following the build plan we design, we implement the appropriate joins to incorporate requested dimensions
- Ultimately, the SQL DJ create will be be what SQL was written combined in the expected ways

GET dj/nodes/metrics/avg_repair_price/sql/?dimensions=repair_order.municipality_id

```
SELECT Avg(price) AS avg_repair_price, repair_orders.municipality_id
FROM   repair_order_details
       LEFT JOIN (SELECT repair_order_id,
                         municipality_id
                  FROM   repair_orders_fact) AS repair_orders
              ON repair_order_details.repair_order_id =
                 repair_orders.repair_order_id
GROUP  BY repair_orders.municipality_id
```





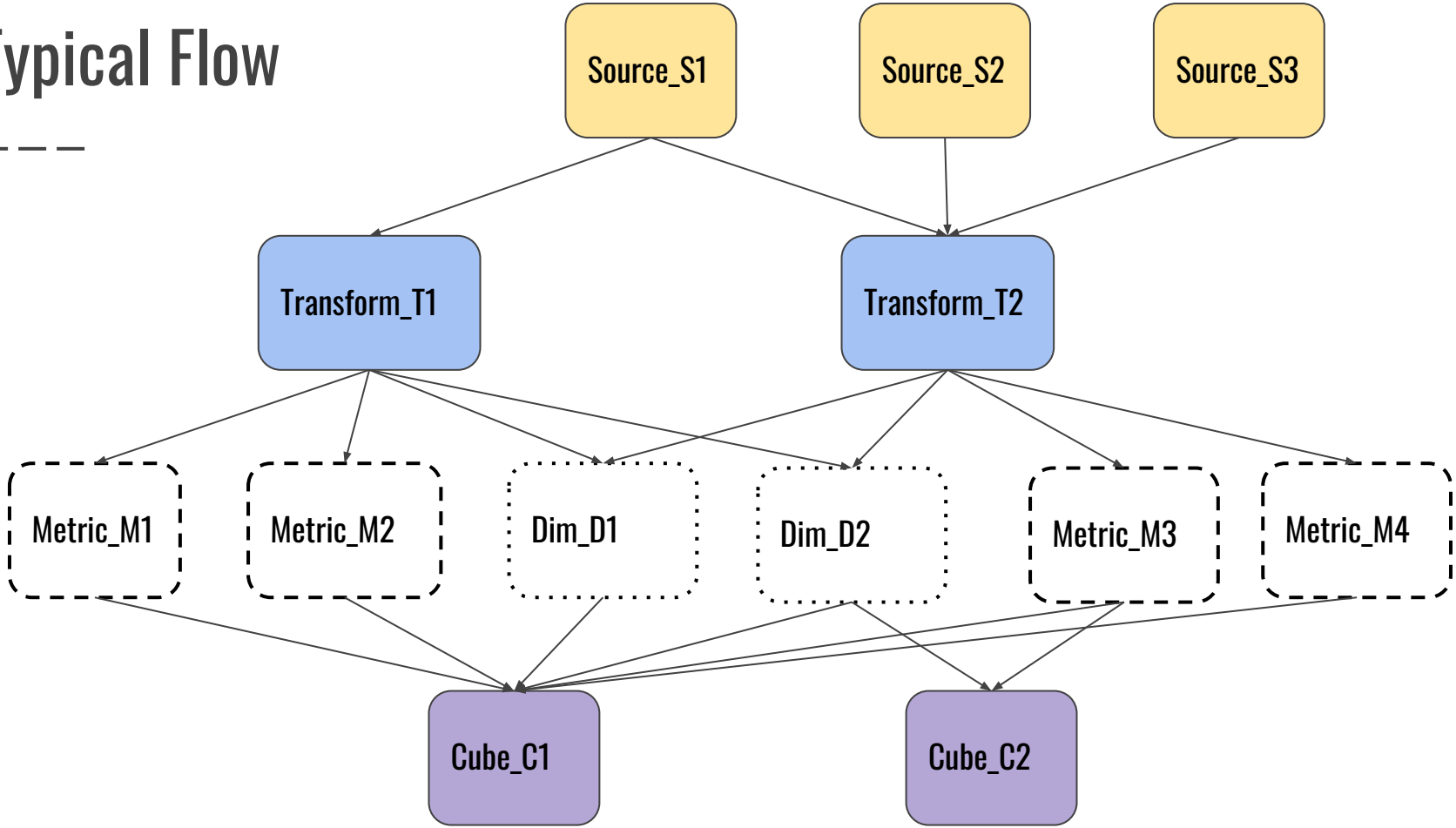
Data Materialization in DJ

How DJ makes what you define.

Olek

Data Engineer (Core DSE at Netflix)

Typical Flow



Source Nodes → view to the “outside”

prod.viewing_f

External metadata:
- Ownership / ACLs
- Table schema
- Partition keys
- Data availability
  - HW mark
  - min/max partitions

DJ metadata:
- Dimensions
- Measures

prod.geography_d

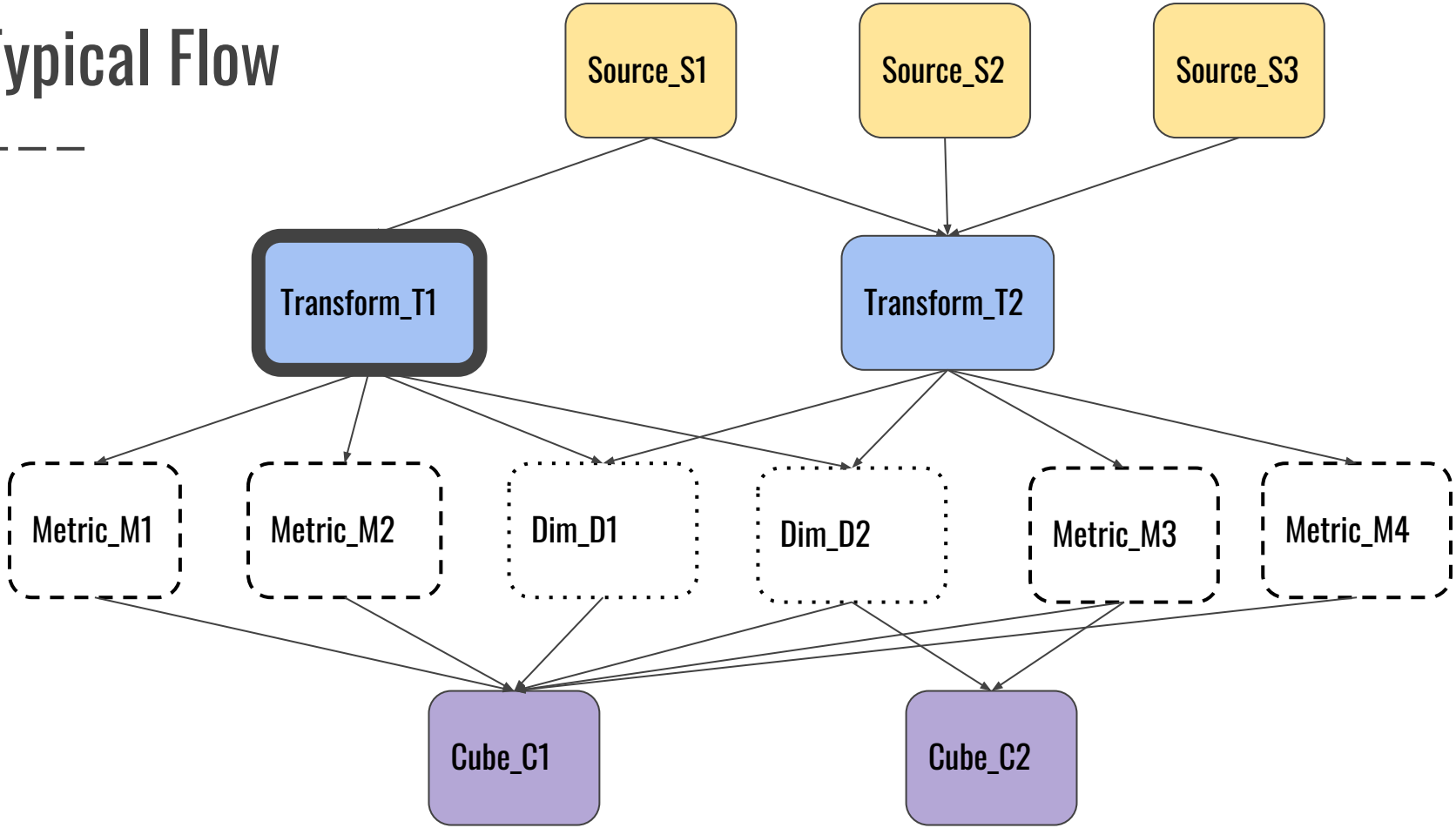
Table schema:
- **date** (int)
  dimension: time
- **country_code** (string)
  dimension: country
- **lang_code** (string)
  dimension: language
- **currency** (string)

...

Looking for a table that DJ doesn’t see yet?

→ Click “here” to add to DJ

Typical Flow



Transform Nodes → form of expression

Transform_T1

Requires:

- SparkSQL (other dialect later)

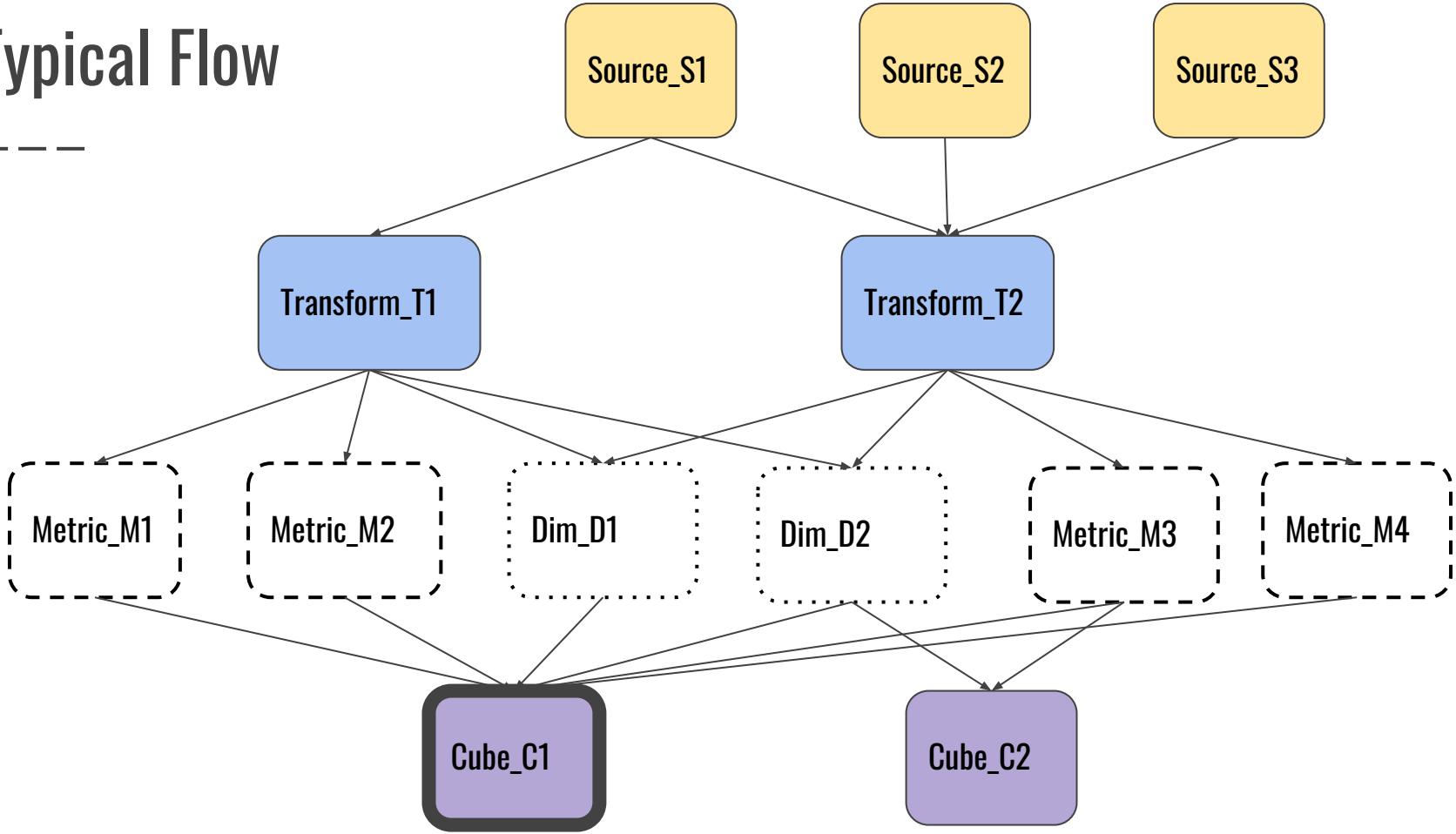
Optional:

- Materialization details
  - Catalog / schema / table
  - Cadence
  - Spark config
- Data audits
  - Business audit spec
- Caching targets:
  - e.g. Druid

Materialization Service will:

- Infer output schema
  - Columns and types
  - Measures and dimensions
  - Column level dependencies
- If materialization is defined
  - Set up materialization schedule
  - Monitor and
  - Reflect data availability in DJ

Typical Flow



Cube Nodes → metrics and dimensions

Cube_C1

Requires:

- List of metrics
- List of dimensions
  - Group By, Rollup, Cube, Grouping Sets

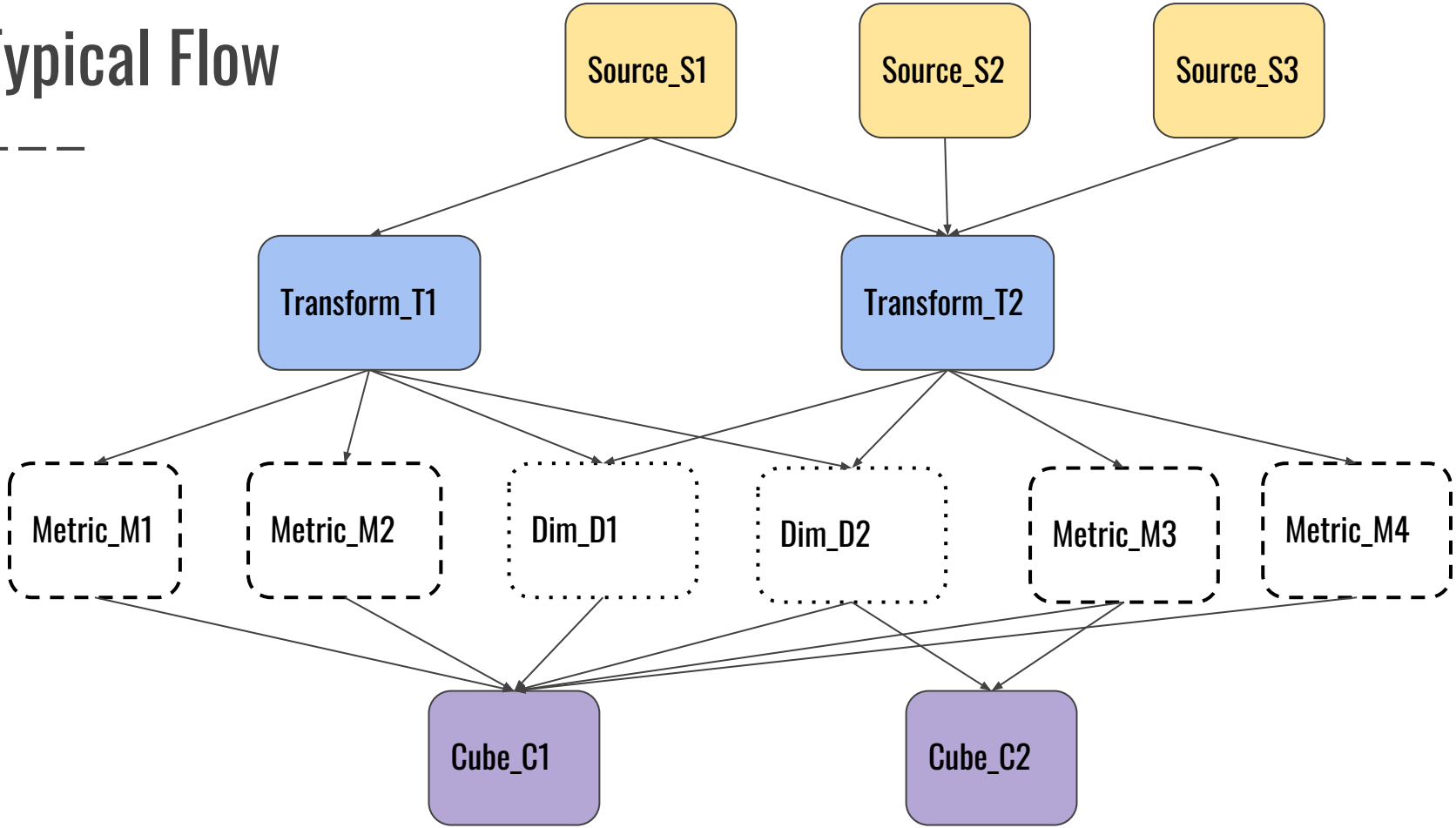
Optional:

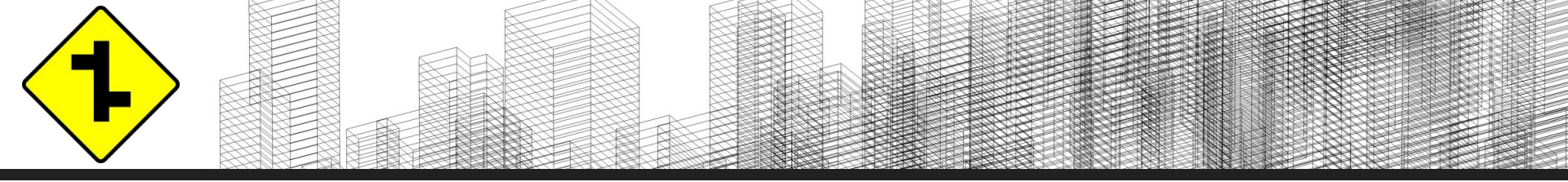
- Materialization details
  - Catalog / schema / table
  - Cadence
- Data audits
  - Business audit spec
- Caching targets:
  - Druid

Materialization service:

- Similarly to Transform Nodes
  - Materialize data
  - Cache it in Druid
  - Refresh on schedule
  - Mark the Data Availability

Typical Flow





Node Dependencies and Change Management

How DJ keeps track of the relationships between nodes

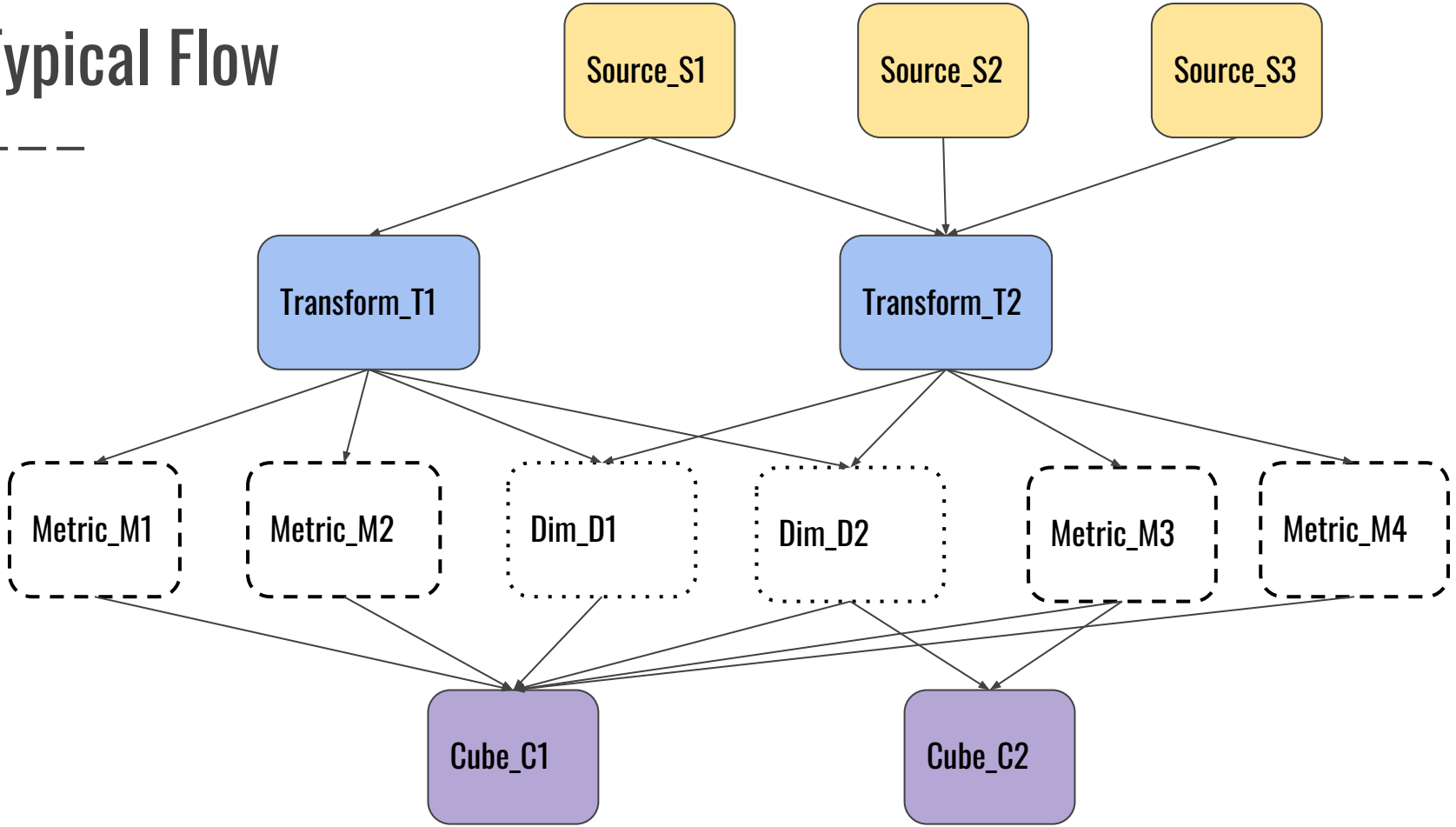
Sam

Software Engineer - Experimentation Platform (XP-Analysis @ Netflix)

Nodes are Dependent on Other Nodes

- ➔ Selecting columns
- ➔ Joining two nodes
- ➔ Linking a column to a dimension
- ➔ Filtering another node
- ➔ Feeding an aggregation in a metric
- ➔ *Many more…*

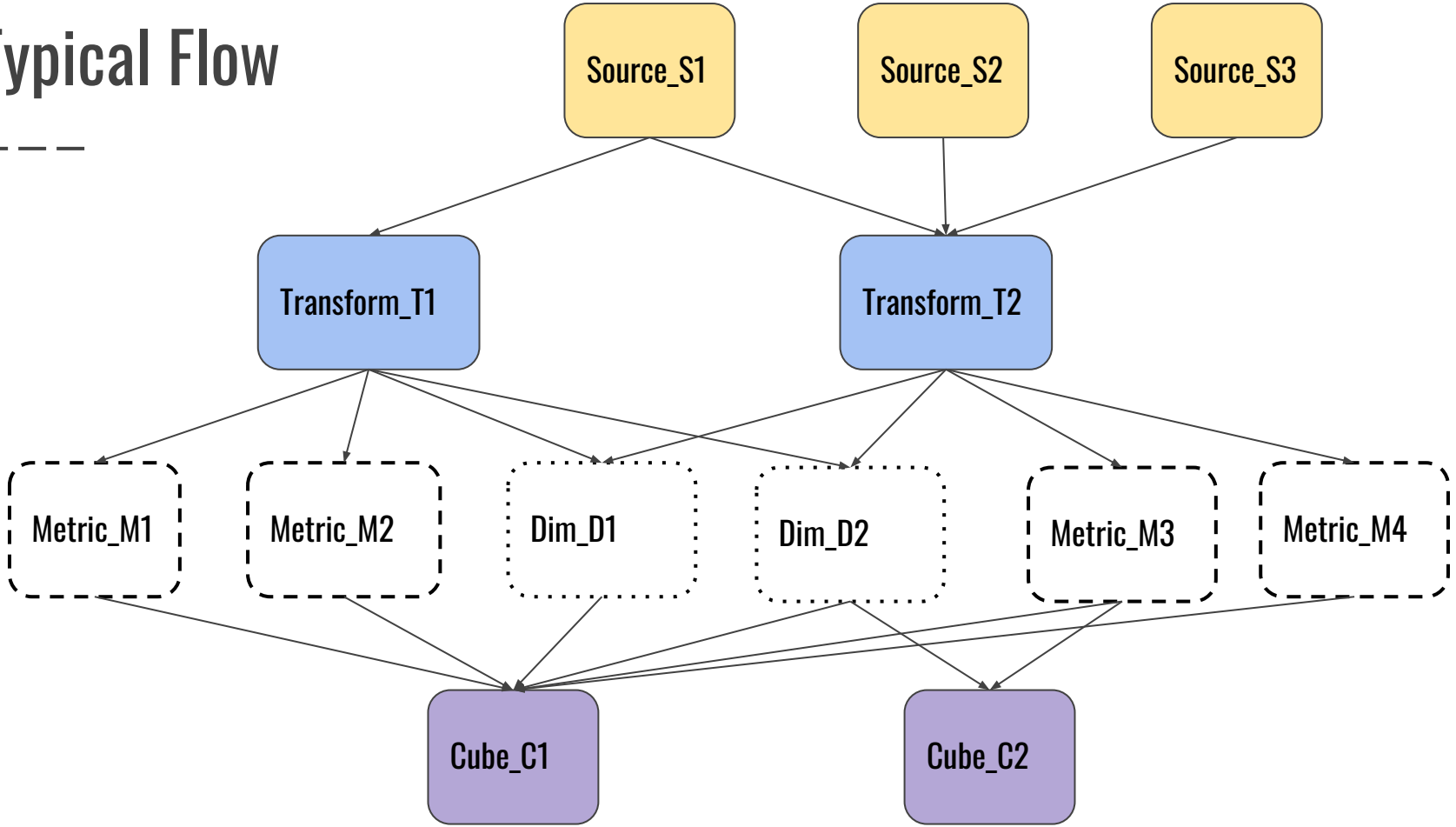
Typical Flow



Source Nodes vs. “Other” Nodes

- ➔ Source nodes are unique in that they are meant to be representatives of external tables, i.e. tables in your data warehouse.
- ➔ Therefore, whether source nodes are “valid” or “invalid” can’t be controlled by DJ - they simply reflect state in the data warehouse
- ➔ “Other” nodes however (transforms, metrics, dimensions) are always dependent on at least one node. In the simplest case, a single source node.

Typical Flow



Node Dependency Validation

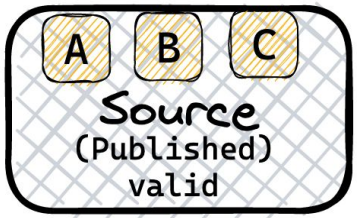
➔ Dependency validation happens on the node's query:

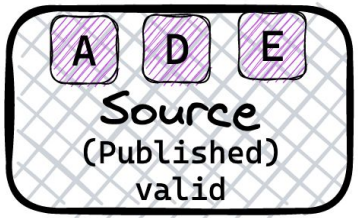
- **All** node names used in the query must exist in the DJ DAG and have a status of **valid**
- **All** columns used in projections must exist in the referenced node
- **All** columns used in functions or operations must match the type requirements

Node “Status” vs. Node “Mode”

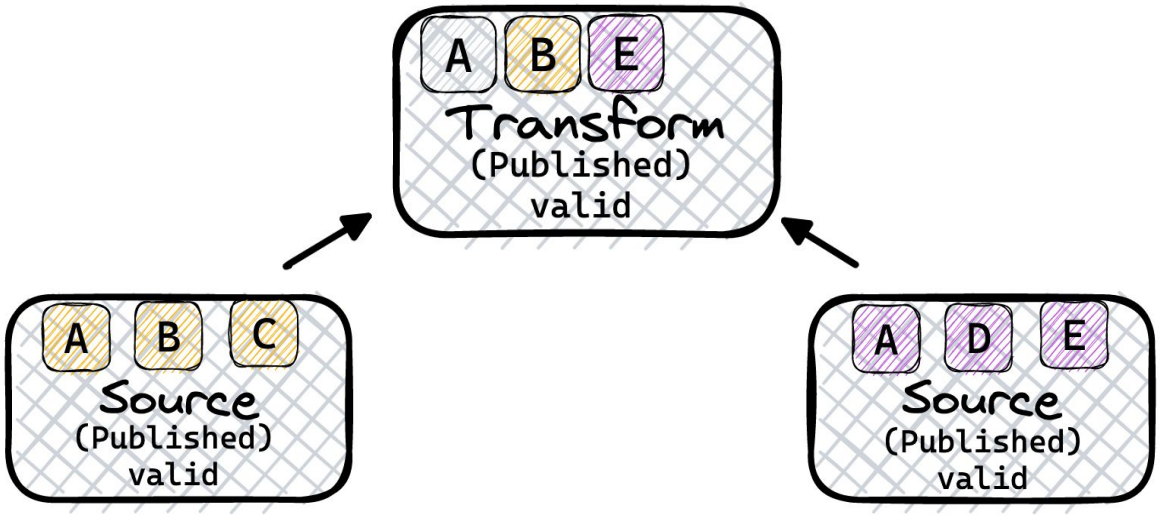
- ➔ **Status** - A system defined attribute of **valid** or **invalid**
- ➔ **Mode** - A user defined attribute of **draft** or **published**
- ➔ When a user specifies a node is in **draft** mode, while adding or updating a node, the system allows changes that may make a node **invalid**.
- ➔ However, when a user specifies that a node is in **published** mode, the system enforces a strict requirement that the node is **valid**

Example 1

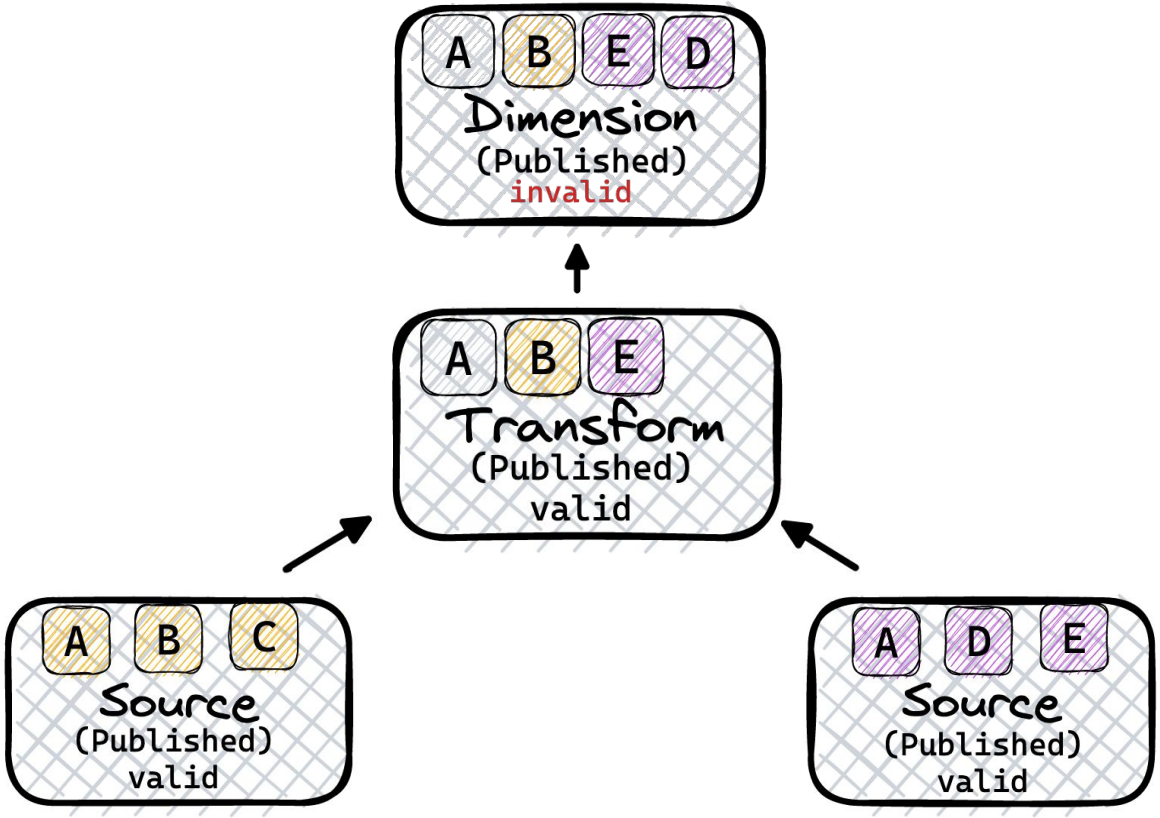




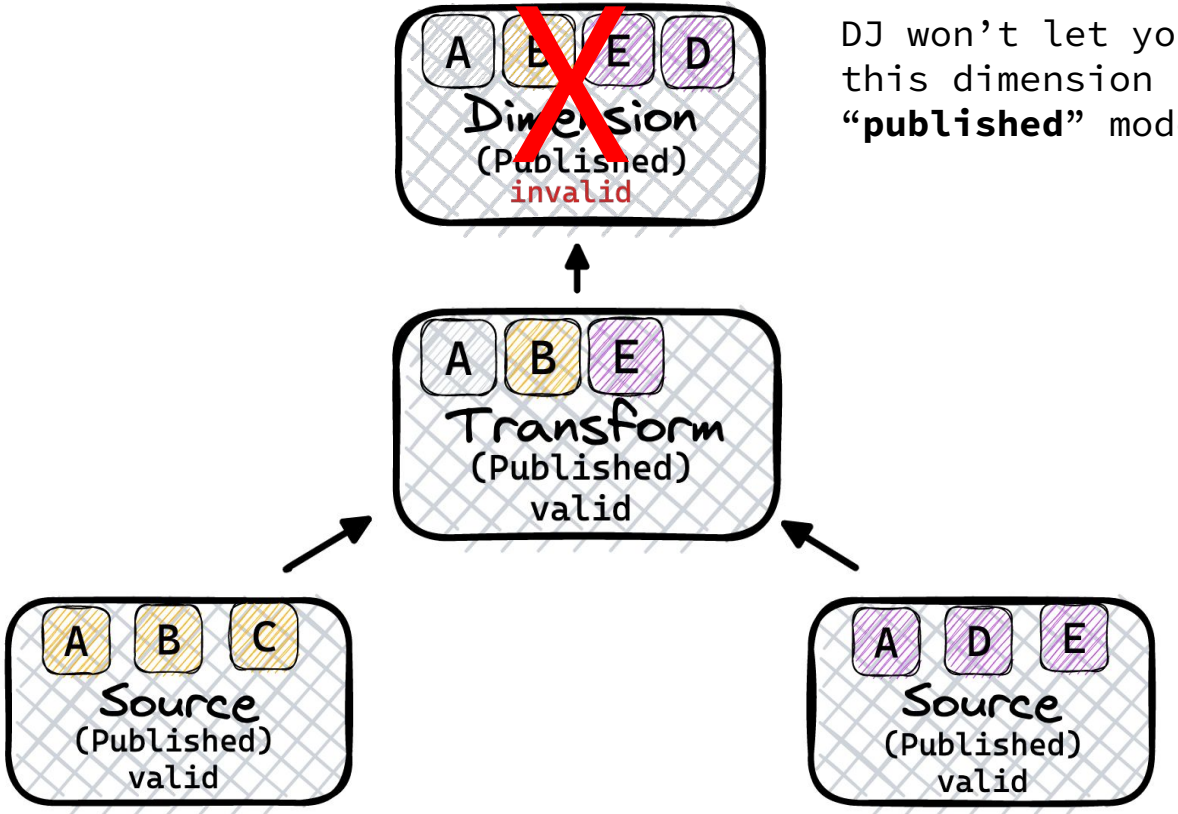
Example 1


A
B
E
Transform
(Published)
valid
A
B
C
Source
(Published)
valid
A
D
E
Source
(Published)
valid

Example 1

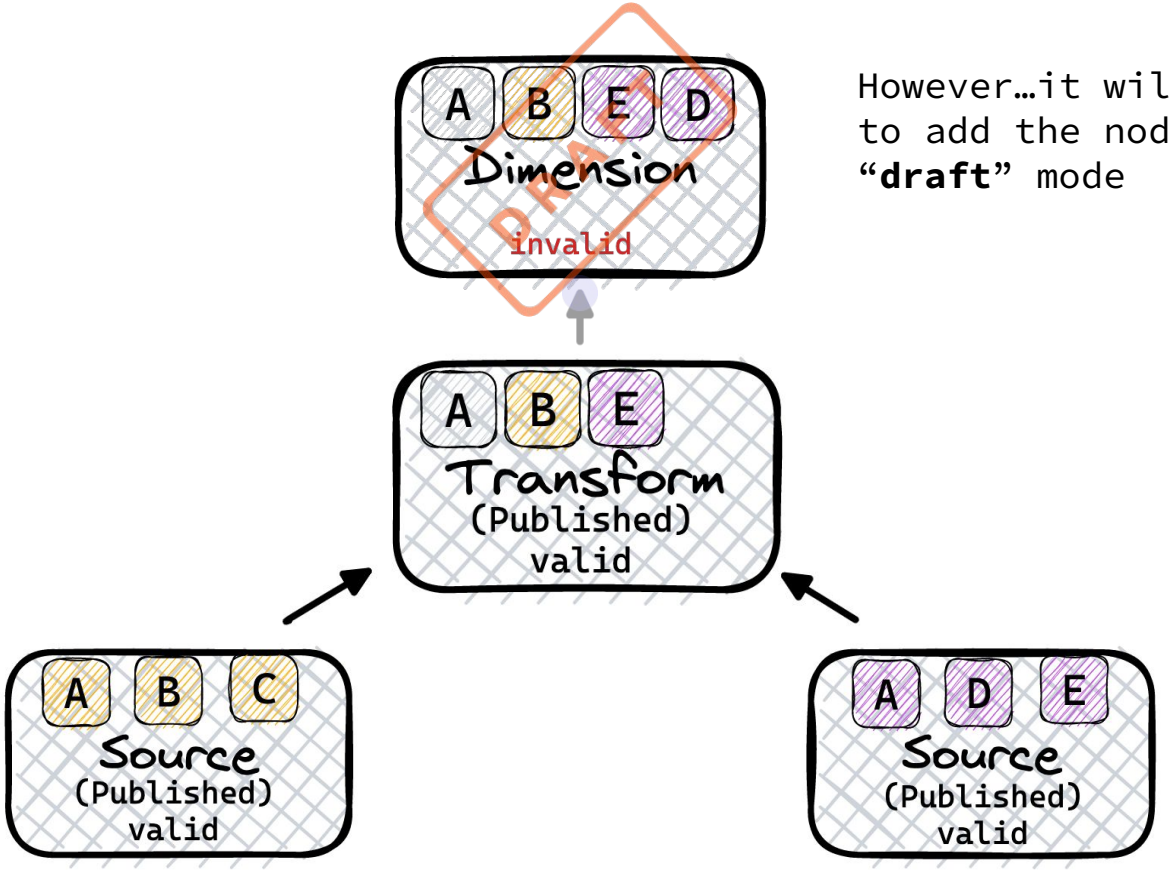

A B E D
Dimension
(Published)
invalid
A B E
Transform
(Published)
valid
A B C
Source
(Published)
valid
A D E
Source
(Published)
valid

Example 1



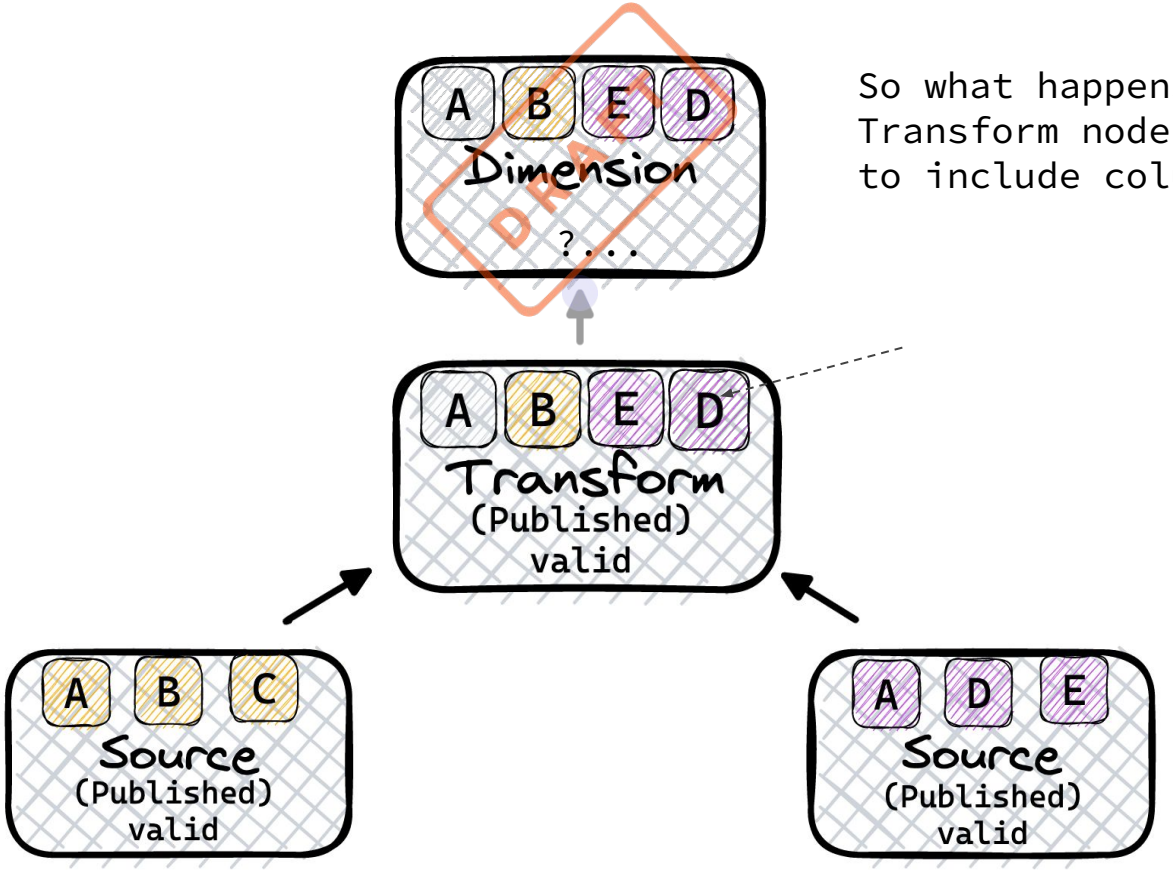
DJ won't let you create this dimension node in "**published**" mode

Example 1



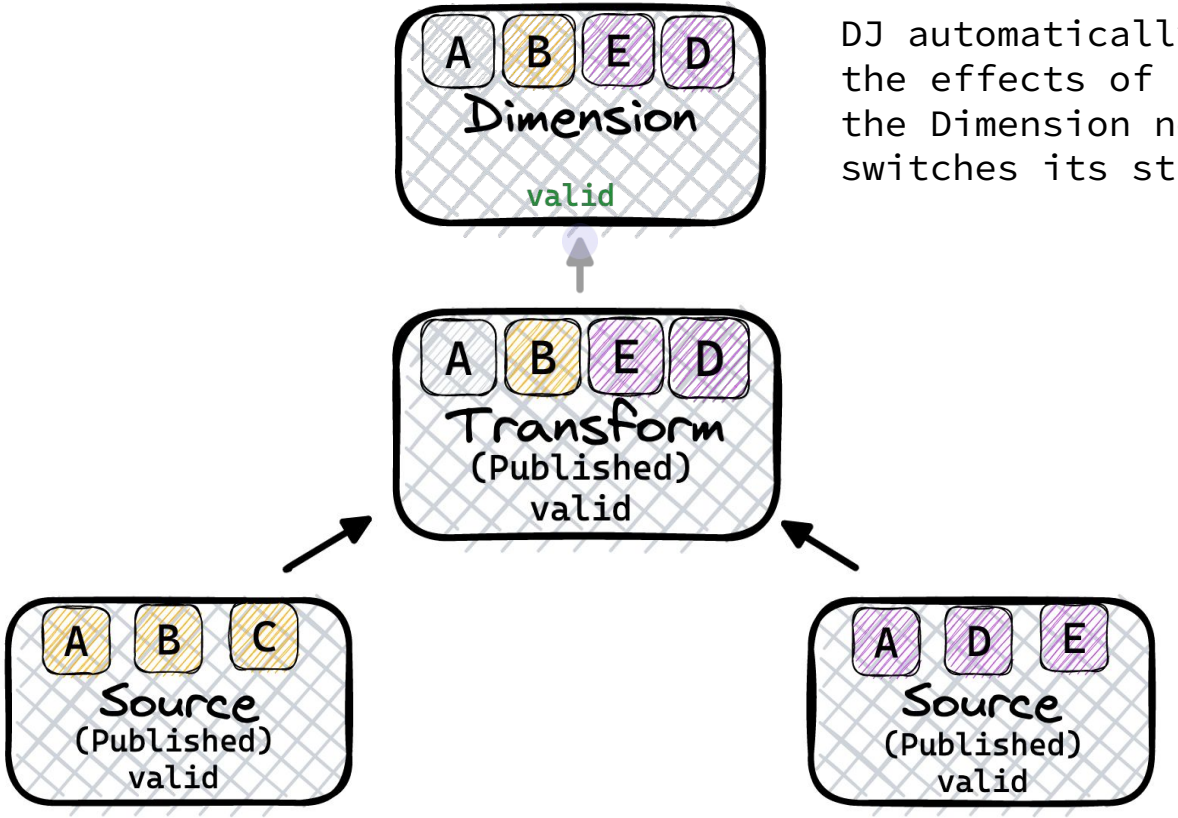
However…it will allow you to add the node in "**draft**" mode

Example 1



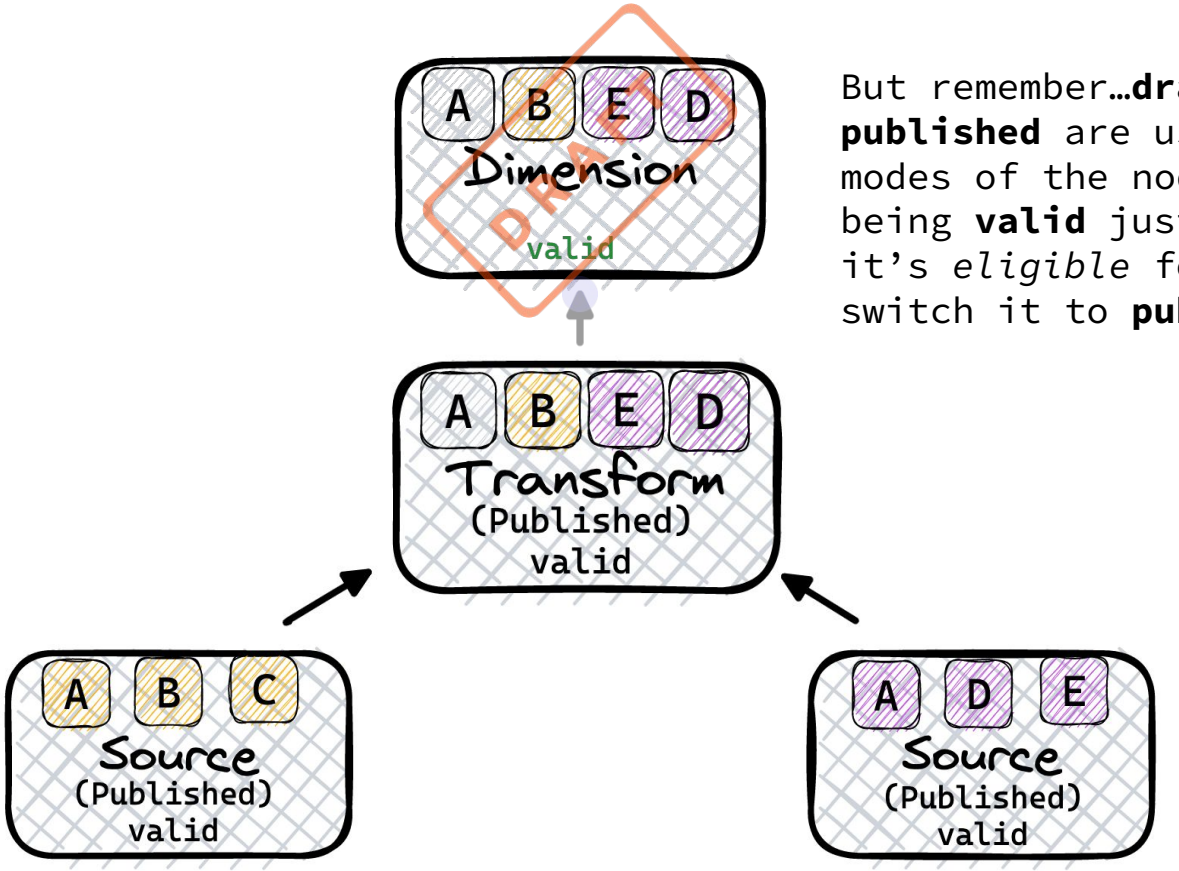
So what happens when the Transform node is updated to include column D?

Example 1



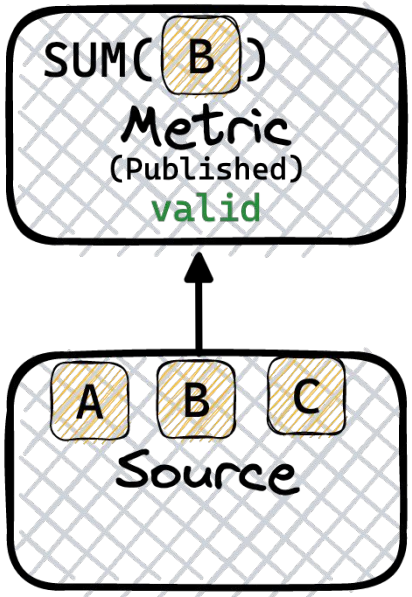
DJ automatically propagates the effects of the change to the Dimension node and switches its status to **valid**

Example 1

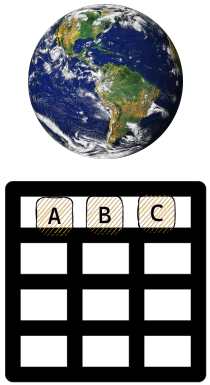


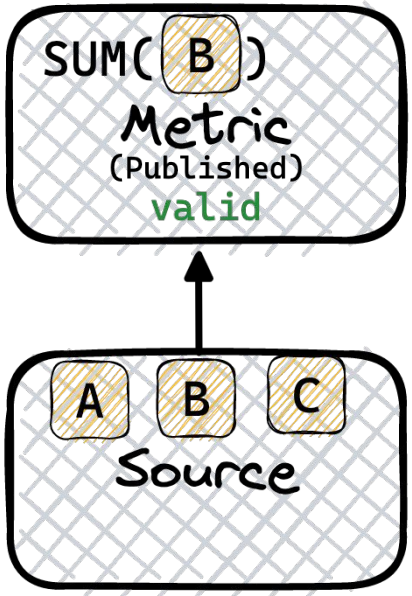
But remember…**draft** and **published** are user defined modes of the node. The node being **valid** just means that it's *eligible* for someone to switch it to **published**.

Example 2

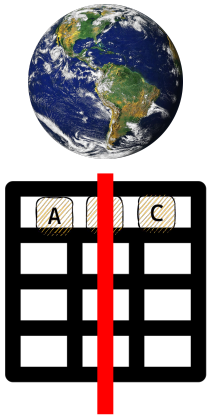


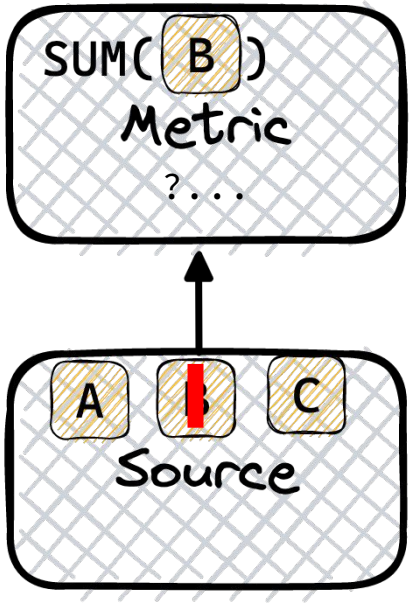
Example 2



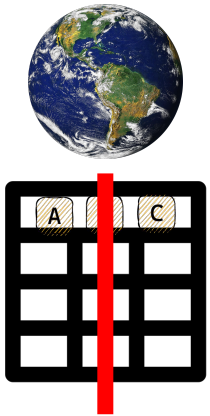


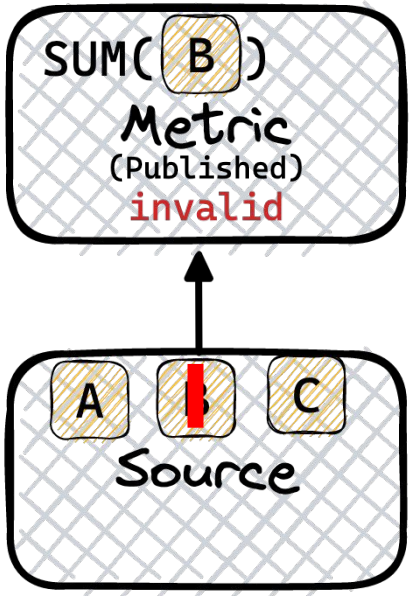
Example 2



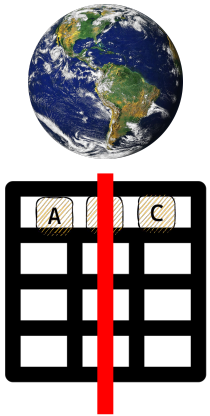


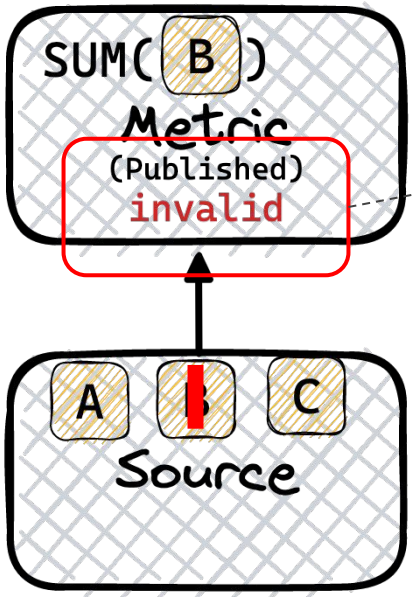
Example 2





Example 2





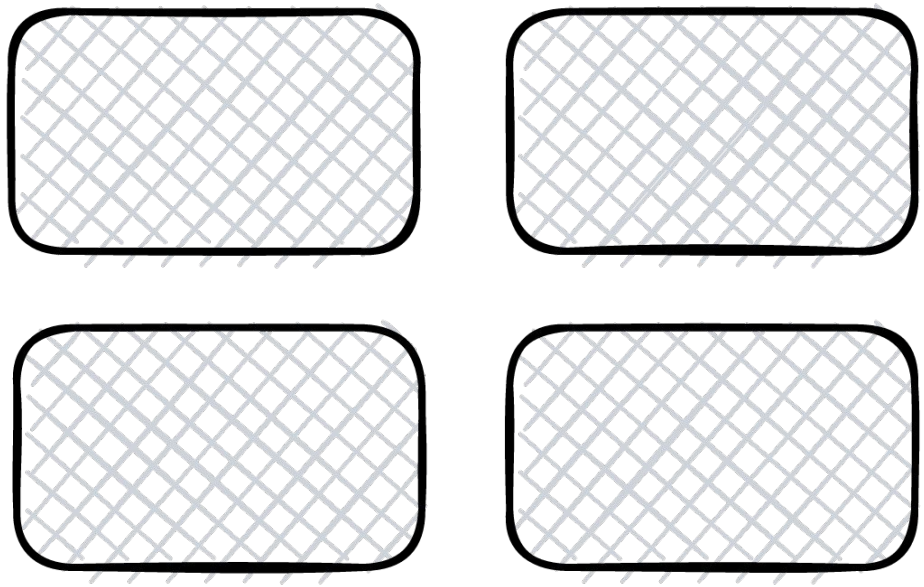
Nodes that are both **published** and **invalid** is a quadrant that's a rough representation of the health of the overall DJ graph

(Published)

DRAFT

valid

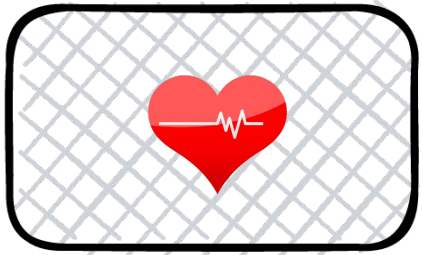
invalid



(Published)

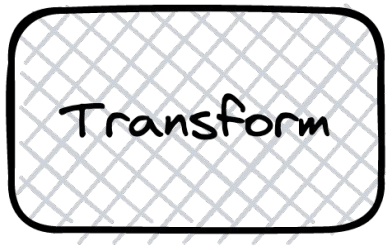
DRAFT

valid

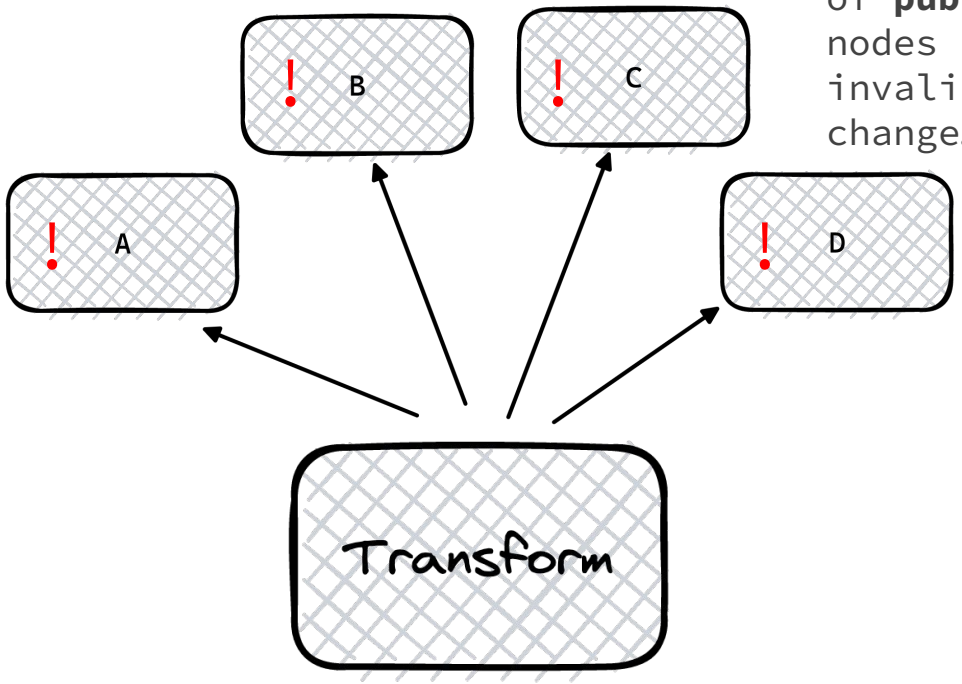


invalid

Example 3

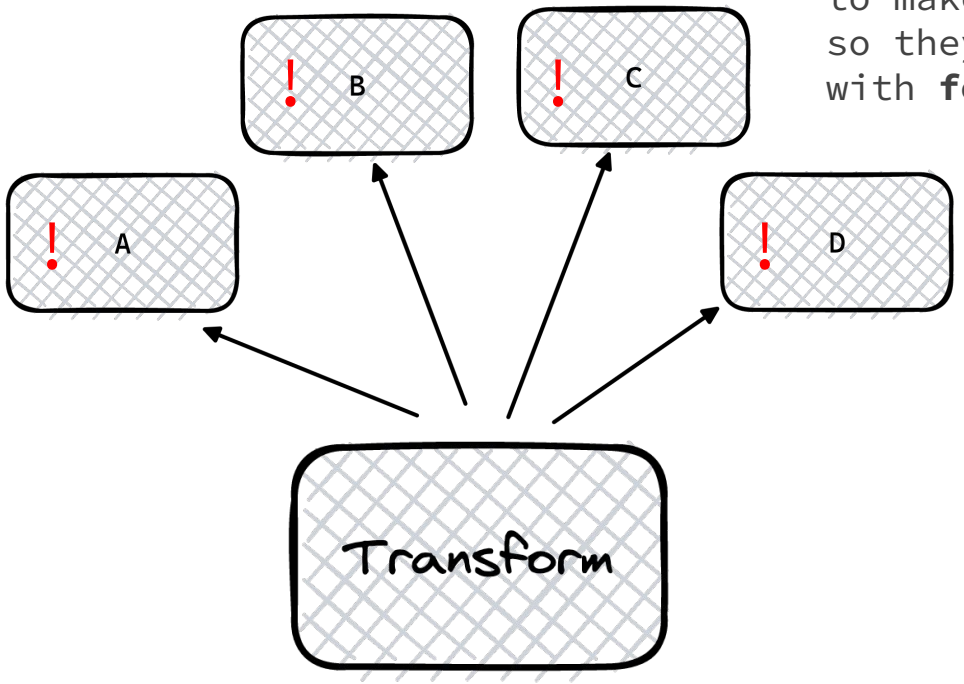


Example 3



DJ stops the change and returns an error with a list of **published** downstream nodes that will be made invalid by that specific change…

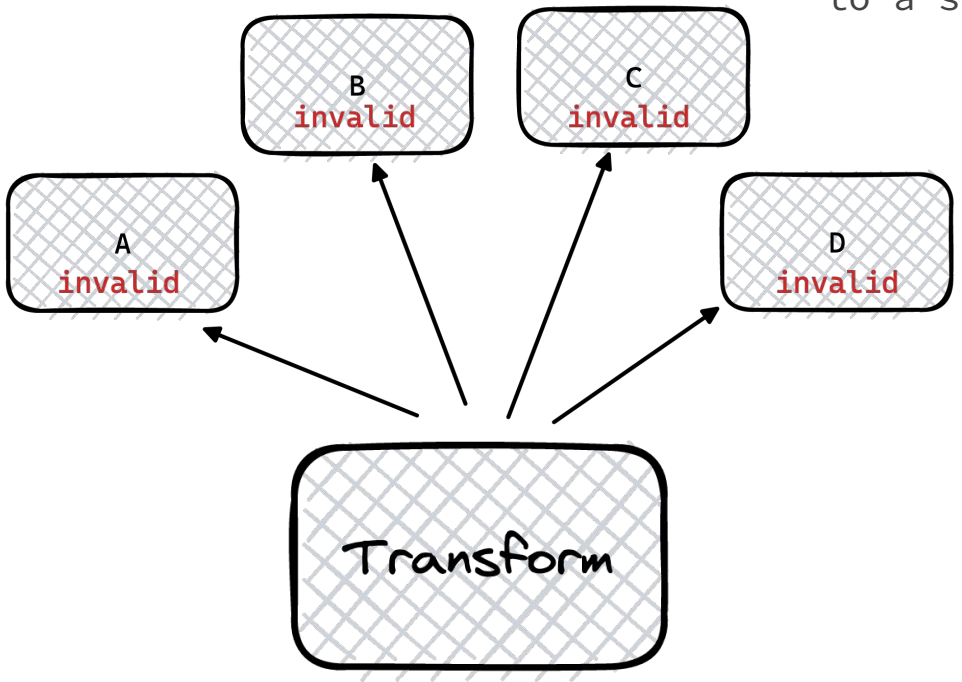
Example 3



Given this information, the user feels informed enough to make the change anyway, so they try again, this time with **force** enabled

Example 3

All of the affected nodes are automatically switched to a status of **invalid**



In summary

- ➔ **Mode** is set by the user and is meant to represent the user's intentions (**draft** or **published**)
- ➔ **Status** is set by the system and is meant to represent the current state of the node within the DAG (**valid** or **invalid**)

Questions?