Containerfile

```
# Use a Python 3.11 image from a trusted source
FROM python:3.11-slim

# Set the working directory inside the container
WORKDIR /app

# Install system dependencies for spatialite and gdal
# This is crucial for handling PostGIS-like operations with SQLite
RUN apt-get update && apt-get install -y \
    libspatialite-dev \
    libsqlite3-mod-spatialite \
    gdal-bin \
    git \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements file and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# This command will be run when the container starts
CMD ["/bin/bash"]
```

alias_dictionary.yaml


```yaml
users:
- account
- accounts
- client
- clients
- customer
- customers
- people
- user
- users
user_id:
- account number
- client id
- customer id
- user id
- user identifier
- user ids
- user number
- user_id
username:
- account name
- handle
- login
- user name
- user names
- username
- usernames
age:
- age
- ages
- years old
balance:
- account balance
- amount
- balance
- balances
- credit
- funds
is_active:
- account status
- active status
- is active
- is actives
- is_active
- status
last_login:
- last access
- last active date
- last login
- last logins
- last signed in
```

- last_login
- most recent login
location:
- coordinates
- geo location
- geographic coordinates
- location
- locations
- place
- position
sales:
- order
- orders
- purchase
- purchases
- sale
- sales
- transaction
- transactions
sale_id:
- order id
- order number
- purchase id
- sale id
- sale ids
- sale_id
- transaction id
product_name:
- item
- item name
- item sold
- product
- product name
- product names
- product_name
sale_date:
- order date
- purchase date
- sale date
- sale dates
- sale_date
- transaction date
- when it was sold
quantity:
- amount sold
- item count
- number of items
- number sold
- quantities
- quantity
price:
- amount
- charge
- cost

- how much
- price
- prices
- value
regions:
- area
- areas
- district
- region
- regions
- territory
- zone
- zones
region_id:
- area id
- region id
- region ids
- region number
- region_id
- zone id
name:
- identifier
- label
- name
- names
- title
boundaries:
- area shape
- border
- boundaries
- boundary
- geometry
- outline
- perimeter
- shape

binder.yaml


```
templates: []
catalogs:
  functions:
    count:
      returns_type: numeric
      class: aggregate
      clause: select
      args:
      - name: column
        types:
        - any
      surfaces:
      - pattern:
        - COUNT
        - (
        - '{column}'
        - )
        commutative: false
      - pattern:
        - count
        - (
        - '{column}'
        - )
        commutative: false
      - pattern:
        - how
        - many
        - (
        - '{column}'
        - )
        commutative: false
      - pattern:
        - number
        - (
        - '{column}'
        - )
        commutative: false
      - pattern:
        - number
        - of
        - (
        - '{column}'
        - )
        commutative: false
      - pattern:
        - quantity
        - of
        - (
        - '{column}'
        - )
        commutative: false
```

```
  - pattern:
    - total
    - number
    - of
    - (
    - '{column}'
    - )
    commutative: false
  applicable_types:
    column:
    - any
  label_rules:
  - not id
  aliases:
  - count
  - how many
  - number
  - number of
  - quantity of
  - total number of
sum:
  returns_type: numeric
  class: aggregate
  clause: select
  args:
  - name: column
    types:
    - numeric
  surfaces:
  - pattern:
    - SUM
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - aggregate
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - sum
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - sum
    - of
    - (
    - '{column}'
    - )
    commutative: false
```

```yaml
  - pattern:
    - total
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - total
    - amount
    - of
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - total
    - of
    - (
    - '{column}'
    - )
    commutative: false
  applicable_types:
    column:
    - numeric
  label_rules:
  - not id
  aliases:
  - aggregate
  - sum
  - sum of
  - total
  - total amount of
  - total of
avg:
  returns_type: numeric
  class: aggregate
  clause: select
  args:
  - name: column
    types:
    - numeric
  surfaces:
  - pattern:
    - AVG
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - average
    - (
    - '{column}'
    - )
    commutative: false
```

```
  - pattern:
    - average
    - of
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - avg
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - mean
    - (
    - '{column}'
    - )
    commutative: false
  applicable_types:
    column:
    - numeric
  label_rules:
  - not id
  aliases:
  - average
  - average of
  - avg
  - mean
min:
  returns_type: any
  class: aggregate
  clause: select
  args:
  - name: column
    types:
    - numeric
    - text
    - date
    - timestamp
  surfaces:
  - pattern:
    - MIN
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - bottom
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
```

```yaml
      - least
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - lowest
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - min
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - minimum
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - smallest
      - (
      - '{column}'
      - )
      commutative: false
    applicable_types:
      column:
      - numeric
      - text
      - date
      - timestamp
    label_rules: []
    aliases:
    - bottom
    - least
    - lowest
    - min
    - minimum
    - smallest
  max:
    returns_type: any
    class: aggregate
    clause: select
    args:
    - name: column
      types:
      - numeric
      - text
      - date
      - timestamp
```

```yaml
surfaces:
- pattern:
  - MAX
  - (
  - '{column}'
  - )
  commutative: false
- pattern:
  - greatest
  - (
  - '{column}'
  - )
  commutative: false
- pattern:
  - highest
  - (
  - '{column}'
  - )
  commutative: false
- pattern:
  - largest
  - (
  - '{column}'
  - )
  commutative: false
- pattern:
  - max
  - (
  - '{column}'
  - )
  commutative: false
- pattern:
  - maximum
  - (
  - '{column}'
  - )
  commutative: false
- pattern:
  - most
  - (
  - '{column}'
  - )
  commutative: false
applicable_types:
  column:
  - numeric
  - text
  - date
  - timestamp
label_rules: []
aliases:
- greatest
- highest
- largest
```

```
    - max
    - maximum
    - most
  distinct:
    returns_type: any
    class: scalar
    clause: select
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - DISTINCT
      - '{column}'
      commutative: false
    - pattern:
      - distinct
      - '{column}'
      commutative: false
    - pattern:
      - unique
      - '{column}'
      commutative: false
    - pattern:
      - unique
      - values
      - of
      - '{column}'
      commutative: false
    applicable_types:
      column:
      - any
    label_rules:
    - not id
    aliases:
    - distinct
    - unique
    - unique values of
  order_by_asc:
    returns_type: none
    class: ordering
    clause: order_by
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - ORDER
      - BY
      - '{column}'
      - ASC
      commutative: false
```

```
- pattern:
  - in
  - ascending
  - order
  - BY
  - '{column}'
  - ASC
  commutative: false
- pattern:
  - order
  - by
  - BY
  - '{column}'
  - ASC
  commutative: false
- pattern:
  - order
  - by
  - ascending
  - BY
  - '{column}'
  - ASC
  commutative: false
- pattern:
  - order_by_asc
  - BY
  - '{column}'
  - ASC
  commutative: false
- pattern:
  - sort
  - by
  - BY
  - '{column}'
  - ASC
  commutative: false
- pattern:
  - sort
  - by
  - ascending
  - BY
  - '{column}'
  - ASC
  commutative: false
applicable_types:
  column:
  - any
label_rules: []
aliases:
- in ascending order
- order by
- order by ascending
- order_by_asc
- sort by
```

```yaml
    - sort by ascending
  order_by_desc:
    returns_type: none
    class: ordering
    clause: order_by
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - ORDER
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - in
      - descending
      - order
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - order
      - by
      - descending
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - order_by_desc
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - sort
      - by
      - descending
      - BY
      - '{column}'
      - DESC
      commutative: false
    applicable_types:
      column:
      - any
    label_rules: []
    aliases:
    - in descending order
    - order by descending
    - order_by_desc
```

```yaml
  - sort by descending
group_by:
  returns_type: none
  class: grouping
  clause: group_by
  args:
  - name: column
    types:
    - any
  surfaces:
  - pattern:
    - GROUP
    - BY
    - '{column}'
    commutative: false
  - pattern:
    - group
    - by
    - BY
    - '{column}'
    commutative: false
  - pattern:
    - group_by
    - BY
    - '{column}'
    commutative: false
  - pattern:
    - grouped
    - by
    - BY
    - '{column}'
    commutative: false
  applicable_types:
    column:
    - any
  label_rules: []
  aliases:
  - group by
  - group_by
  - grouped by
having:
  returns_type: boolean
  class: predicate
  clause: having
  args:
  - name: condition
    types:
    - any
  surfaces:
  - pattern:
    - HAVING
    - '{condition}'
    commutative: false
  - pattern:
```

```yaml
    - having
    - '{condition}'
    commutative: false
  - pattern:
    - with
    - '{condition}'
    commutative: false
  applicable_types:
    condition:
    - any
  label_rules: []
  aliases:
  - having
  - with
limit:
  returns_type: none
  class: limit
  clause: limit
  args:
  - name: value
    types:
    - numeric
  surfaces:
  - pattern:
    - LIMIT
    - '{value}'
    commutative: false
  - pattern:
    - first
    - '{value}'
    commutative: false
  - pattern:
    - limit
    - '{value}'
    commutative: false
  - pattern:
    - only
    - '{value}'
    commutative: false
  - pattern:
    - top
    - '{value}'
    commutative: false
  applicable_types:
    value:
    - numeric
  label_rules: []
  aliases:
  - first
  - limit
  - only
  - top
extract:
  returns_type: numeric
```

```yaml
  class: scalar
  clause: select
  args:
  - name: part
    types:
    - any
  - name: column
    types:
    - date
    - timestamp
  surfaces:
  - pattern:
    - '{part}'
    - from
    - '{column}'
    commutative: false
  applicable_types:
    part:
    - any
    column:
    - date
    - timestamp
  label_rules: []
  aliases:
  - extract
  - get the
length:
  returns_type: numeric
  class: scalar
  clause: select
  args:
  - name: column
    types:
    - text
  surfaces:
  - pattern:
    - LENGTH
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - character
    - count
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - length
    - (
    - '{column}'
    - )
    commutative: false
```

```
    - pattern:
      - length
      - in
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - length
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - string
      - length
      - (
      - '{column}'
      - )
      commutative: false
    applicable_types:
      column:
      - text
    label_rules: []
    aliases:
    - character count
    - length
    - length in
    - length of
    - string length
  concat:
    returns_type: text
    class: scalar
    clause: select
    args:
    - name: column1
      types:
      - text
    - name: column2
      types:
      - text
    surfaces:
    - pattern:
      - '{column1}'
      - and
      - '{column2}'
      commutative: false
    applicable_types:
      column1:
      - text
      column2:
      - text
    label_rules: []
```

```
  aliases:
  - combine
  - concat
  - concatenate
  - join
cast:
  returns_type: any
  class: scalar
  clause: select
  args:
  - name: column
    types:
    - any
  - name: to_type
    types:
    - any
  surfaces:
  - pattern:
    - '{column}'
    - to
    - '{to_type}'
    commutative: false
  applicable_types:
    column:
    - any
    to_type:
    - any
  label_rules: []
  aliases:
  - cast
  - change type
  - convert
st_perimeter:
  returns_type: any
  class: scalar
  clause: select
  args:
  - name: geom
    types:
    - geometry_polygon
    - geography_polygon
  surfaces:
  - pattern:
    - ST_Perimeter
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - boundary
    - length
    - (
    - '{geom}'
    - )
```

```
    commutative: false
  - pattern:
    - length
    - of
    - boundary
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - outline
    - length
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - perimeter
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - st_perimeter
    - (
    - '{geom}'
    - )
    commutative: false
  applicable_types:
    geom:
    - geometry_polygon
    - geography_polygon
  label_rules:
  - postgis
  aliases:
  - boundary length
  - length of boundary
  - outline length
  - perimeter
  - st_perimeter
st_distance:
  returns_type: numeric
  class: spatial
  clause: select
  args:
  - name: geom1
    types:
    - geometry_point
    - geography_point
    - geometry_linestring
    - geography_linestring
    - geometry_polygon
    - geography_polygon
  - name: geom2
```

```yaml
  types:
  - geometry_point
  - geography_point
  - geometry_linestring
  - geography_linestring
  - geometry_polygon
  - geography_polygon
surfaces:
- pattern:
  - of
  - '{geom1}'
  - from
  - '{geom2}'
  commutative: false
- pattern:
  - distance
  - '{geom1}'
  - from
  - '{geom2}'
  commutative: false
- pattern:
  - distance
  - between
  - '{geom1}'
  - from
  - '{geom2}'
  commutative: false
- pattern:
  - how
  - far
  - '{geom1}'
  - from
  - '{geom2}'
  commutative: false
- pattern:
  - separation
  - of
  - '{geom1}'
  - from
  - '{geom2}'
  commutative: false
- pattern:
  - st_distance
  - '{geom1}'
  - from
  - '{geom2}'
  commutative: false
applicable_types:
  geom1:
  - geometry_point
  - geography_point
  - geometry_linestring
  - geography_linestring
  - geometry_polygon
```

```
      - geography_polygon
      geom2:
      - geometry_point
      - geography_point
      - geometry_linestring
      - geography_linestring
      - geometry_polygon
      - geography_polygon
    label_rules:
    - postgis
    aliases:
    - distance
    - distance between
    - how far
    - separation of
    - st_distance
  st_intersects:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - with
      - '{geom2}'
      commutative: false
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
```

```
    label_rules:
    - postgis
    aliases:
    - intersects
    - overlaps with
    - st_intersects
  st_area:
    returns_type: numeric
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry_polygon
      - geography_polygon
    surfaces:
    - pattern:
      - ST_Area
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - area
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - area
      - of
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - size
      - of
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - st_area
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - surface
      - area
      - (
      - '{geom}'
      - )
```

```yaml
      commutative: false
    applicable_types:
      geom:
      - geometry_polygon
      - geography_polygon
    label_rules:
    - postgis
    aliases:
    - area
    - area of
    - size of
    - st_area
    - surface area
  st_length:
    returns_type: numeric
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry_linestring
      - geography_linestring
    surfaces:
    - pattern:
      - ST_Length
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - distance
      - along
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - distance
      - of
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - length
      - along
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - length
      - of
      - (
```

```
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - line
    - length
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - path
    - length
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - st_length
    - (
    - '{geom}'
    - )
    commutative: false
  applicable_types:
    geom:
    - geometry_linestring
    - geography_linestring
  label_rules:
  - postgis
  aliases:
  - distance along
  - distance of
  - length along
  - length of
  - line length
  - path length
  - st_length
st_x:
  returns_type: numeric
  class: spatial
  clause: select
  args:
  - name: point
    types:
    - geometry_point
    - geography_point
  surfaces:
  - pattern:
    - ST_X
    - (
    - '{point}'
    - )
    commutative: false
  - pattern:
```

```
    - lon
    - (
    - '{point}'
    - )
    commutative: false
  - pattern:
    - longitude
    - (
    - '{point}'
    - )
    commutative: false
  - pattern:
    - st_x
    - (
    - '{point}'
    - )
    commutative: false
  - pattern:
    - x
    - coordinate
    - (
    - '{point}'
    - )
    commutative: false
  - pattern:
    - x
    - pos
    - (
    - '{point}'
    - )
    commutative: false
  applicable_types:
    point:
    - geometry_point
    - geography_point
  label_rules:
  - postgis
  aliases:
  - lon
  - longitude
  - st_x
  - x coordinate
  - x pos
st_y:
  returns_type: numeric
  class: spatial
  clause: select
  args:
  - name: point
    types:
    - geometry_point
    - geography_point
  surfaces:
  - pattern:
```

```
      - ST_Y
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - lat
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - latitude
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - st_y
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - y
      - coordinate
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - y
      - pos
      - (
      - '{point}'
      - )
      commutative: false
    applicable_types:
      point:
      - geometry_point
      - geography_point
    label_rules:
    - postgis
    aliases:
    - lat
    - latitude
    - st_y
    - y coordinate
    - y pos
  st_within:
    returns_type: boolean
    class: predicate
    clause: where
    args:
```

```
  - name: geom1
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  - name: geom2
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  surfaces:
  - pattern:
    - '{geom1}'
    - in
    - '{geom2}'
    commutative: false
  applicable_types:
    geom1:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
    geom2:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  label_rules:
  - postgis
  aliases:
  - contained in
  - inside
  - is inside
  - is within
  - st_within
  - within
st_contains:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: geom1
    types:
    - geometry
    - geography
    - geometry_polygon
  - name: geom2
    types:
```

```yaml
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - and
      - '{geom2}'
      commutative: false
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    aliases:
    - contains
    - encloses
    - st_contains
    - surrounds
  st_geometrytype:
    returns_type: text
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - ST_GeometryType
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - geometry
      - type
      - (
      - '{geom}'
      - )
```

```
    commutative: false
  - pattern:
    - shape
    - type
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - st_geometrytype
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - type
    - of
    - geometry
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - what
    - kind
    - of
    - shape
    - (
    - '{geom}'
    - )
    commutative: false
  applicable_types:
    geom:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  label_rules:
  - postgis
  aliases:
  - geometry type
  - shape type
  - st_geometrytype
  - type of geometry
  - what kind of shape
st_buffer:
  returns_type: geometry_polygon
  class: spatial
  clause: select
  args:
  - name: geom
    types:
    - geometry
```

```yaml
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  - name: radius
    types:
    - numeric
  surfaces:
  - pattern:
    - '{geom}'
    - by
    - '{radius}'
    commutative: false
  applicable_types:
    geom:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
    radius:
    - numeric
  label_rules:
  - postgis
  aliases:
  - area around
  - buffer
  - buffer around
  - expand by
  - st_buffer
st_union:
  returns_type: geometry
  class: spatial
  clause: select
  args:
  - name: geom_collection
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  surfaces:
  - pattern:
    - ST_Union
    - (
    - '{geom_collection}'
    - )
    commutative: false
  - pattern:
    - combine
    - (
    - '{geom_collection}'
    - )
```

```yaml
      commutative: false
    - pattern:
      - merge
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - st_union
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - union
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - union
      - of
      - (
      - '{geom_collection}'
      - )
      commutative: false
    applicable_types:
      geom_collection:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    aliases:
    - combine
    - merge
    - st_union
    - union
    - union of
  st_centroid:
    returns_type: geometry_point
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
```

```
  - pattern:
    - ST_Centroid
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - center
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - center
    - point
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - centroid
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - geometric
    - center
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - st_centroid
    - (
    - '{geom}'
    - )
    commutative: false
  applicable_types:
    geom:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  label_rules:
  - postgis
  aliases:
  - center
  - center point
  - centroid
  - geometric center
  - st_centroid
st_simplify:
```

```
  returns_type: any
  class: scalar
  clause: select
  args:
  - name: geom
    types:
    - geometry_linestring
    - geometry_polygon
    - geography_linestring
    - geography_polygon
  - name: tolerance
    types:
    - numeric
  surfaces:
  - pattern:
    - '{geom}'
    - by
    - '{tolerance}'
    commutative: false
  applicable_types:
    geom:
    - geometry_linestring
    - geometry_polygon
    - geography_linestring
    - geography_polygon
    tolerance:
    - numeric
  label_rules:
  - postgis
  aliases:
  - generalize
  - simplify
  - simplify shape
  - st_simplify
st_touches:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: geom1
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  - name: geom2
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  surfaces:
```

```yaml
  - pattern:
    - '{geom1}'
    - and
    - '{geom2}'
    commutative: false
  applicable_types:
    geom1:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
    geom2:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  label_rules:
  - postgis
  aliases:
  - borders
  - is adjacent to
  - st_touches
  - touches
st_crosses:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: geom1
    types:
    - geometry
    - geography
    - geometry_linestring
    - geometry_polygon
  - name: geom2
    types:
    - geometry
    - geography
    - geometry_linestring
  surfaces:
  - pattern:
    - '{geom1}'
    - and
    - '{geom2}'
    commutative: false
  applicable_types:
    geom1:
    - geometry
    - geography
    - geometry_linestring
    - geometry_polygon
    geom2:
```

```yaml
    - geometry
    - geography
    - geometry_linestring
  label_rules:
  - postgis
  aliases:
  - crosses
  - goes across
  - st_crosses
st_spatial_index:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: geom1
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  - name: geom2
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  surfaces:
  - pattern:
    - '{geom1}'
    - with
    - '{geom2}'
    commutative: false
  applicable_types:
    geom1:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
    geom2:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  label_rules:
  - postgis
  aliases:
  - spatial index
  - st_spatial_index
st_distance_operator:
  returns_type: numeric
```

```
    class: spatial
    clause: order_by
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - to
      - '{geom2}'
      commutative: false
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    aliases:
    - closest
    - closest to
    - nearest
    - st_distance_operator
  st_transform:
    returns_type: geometry
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
```

```yaml
        - geometry_polygon
      - name: srid
        types:
        - numeric
      surfaces:
      - pattern:
        - '{geom}'
        - to
        - '{srid}'
        commutative: false
      applicable_types:
        geom:
        - geometry
        - geography
        - geometry_point
        - geometry_linestring
        - geometry_polygon
        srid:
        - numeric
      label_rules:
      - postgis
      aliases:
      - convert coordinate system
      - reproject
      - st_transform
      - transform
columns:
  user_id:
    type: INTEGER
    type_category: numeric
    labels:
    - id
    table: sales
  username:
    type: VARCHAR(50)
    type_category: text
    labels: []
    table: users
  age:
    type: INT
    type_category: numeric
    labels: []
    table: users
  balance:
    type: DECIMAL(10, 2)
    type_category: numeric
    labels: []
    table: users
  is_active:
    type: BOOLEAN
    type_category: boolean
    labels: []
    table: users
  last_login:
```

```yaml
    type: TIMESTAMP
    type_category: timestamp
    labels: []
    table: users
  location:
    type: geography_point
    type_category: geography_point
    labels:
    - postgis
    table: users
  sale_id:
    type: INTEGER
    type_category: numeric
    labels:
    - id
    table: sales
  product_name:
    type: TEXT
    type_category: text
    labels: []
    table: sales
  sale_date:
    type: DATE
    type_category: date
    labels: []
    table: sales
  quantity:
    type: INT
    type_category: numeric
    labels: []
    table: sales
  price:
    type: FLOAT
    type_category: numeric
    labels: []
    table: sales
  region_id:
    type: INTEGER
    type_category: numeric
    labels:
    - id
    table: regions
  name:
    type: VARCHAR(50)
    type_category: text
    labels: []
    table: regions
  boundaries:
    type: geography_polygon
    type_category: geography_polygon
    labels:
    - postgis
    table: regions
tables:
```

```
  - regions
  - sales
  - users
  connectors:
    COMMA: ','
    AND: and
    AT: at
    BETWEEN: between
    FROM: from
    IN: in
    OF: of
    'ON': 'on'
    TO: to
    WITH: with
  punctuation:
    ',': ','
comparison_operators:
  equal:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    - name: value
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - '='
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - equals
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - equal
      - to
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - '='
      - '{value}'
```

```yaml
      commutative: false
    - pattern:
      - '{column}'
      - ==
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - exactly
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - exactly
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - the
      - same
      - as
      - '{value}'
      commutative: false
    applicable_types:
      column:
      - any
      value:
      - any
    label_rules: []
    aliases:
    - '='
    - ==
    - equal
    - equal to
    - equals
    - exactly
    - is
    - is exactly
    - is the same as
  not_equal:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    - name: value
      types:
      - any
    surfaces:
    - pattern:
```

```
  - '{column}'
  - '!='
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - is
  - not
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - '!='
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - <>
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - is
  - not
  - equal
  - to
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - does
  - not
  - equal
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - isn't
  - '{value}'
  commutative: false
applicable_types:
  column:
  - any
  value:
  - any
label_rules: []
aliases:
- '!='
- <>
- does not equal
- is not
- is not equal to
- isn't
- not_equal
```

```yaml
greater_than:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - numeric
    - date
    - timestamp
  - name: value
    types:
    - numeric
    - date
    - timestamp
  surfaces:
  - pattern:
    - '{column}'
    - '>'
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - is
    - greater
    - than
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - '>'
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - more
    - than
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - over
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - above
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - exceeds
    - '{value}'
    commutative: false
```

```
    applicable_types:
      column:
      - numeric
      - date
      - timestamp
      value:
      - numeric
      - date
      - timestamp
    label_rules: []
    aliases:
    - '>'
    - above
    - exceeds
    - greater_than
    - is greater than
    - more than
    - over
  less_than:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - numeric
      - date
      - timestamp
    - name: value
      types:
      - numeric
      - date
      - timestamp
    surfaces:
    - pattern:
      - '{column}'
      - <
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - less
      - than
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - <
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - smaller
```

```
    - than
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - under
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - below
    - '{value}'
    commutative: false
  applicable_types:
    column:
    - numeric
    - date
    - timestamp
    value:
    - numeric
    - date
    - timestamp
  label_rules: []
  aliases:
  - <
  - below
  - is less than
  - less_than
  - smaller than
  - under
greater_than_or_equal:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - numeric
    - date
    - timestamp
  - name: value
    types:
    - numeric
    - date
    - timestamp
  surfaces:
  - pattern:
    - '{column}'
    - '>='
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - is
```

```
      - greater
      - than
      - or
      - equal
      - to
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - '>='
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - at
      - least
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - at
      - least
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - not
      - less
      - than
      - '{value}'
      commutative: false
    applicable_types:
      column:
      - numeric
      - date
      - timestamp
      value:
      - numeric
      - date
      - timestamp
    label_rules: []
    aliases:
    - '>='
    - at least
    - greater_than_or_equal
    - is at least
    - is greater than or equal to
    - not less than
  less_than_or_equal:
    returns_type: boolean
    class: predicate
    clause: where
    args:
```

```yaml
- name: column
  types:
  - numeric
  - date
  - timestamp
- name: value
  types:
  - numeric
  - date
  - timestamp
surfaces:
- pattern:
  - '{column}'
  - <=
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - is
  - less
  - than
  - or
  - equal
  - to
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - <=
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - at
  - most
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - is
  - at
  - most
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - not
  - more
  - than
  - '{value}'
  commutative: false
- pattern:
  - '{column}'
  - up
```

```yaml
    - to
    - '{value}'
    commutative: false
  applicable_types:
    column:
    - numeric
    - date
    - timestamp
    value:
    - numeric
    - date
    - timestamp
  label_rules: []
  aliases:
  - <=
  - at most
  - is at most
  - is less than or equal to
  - less_than_or_equal
  - not more than
  - up to
between:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - numeric
    - date
    - timestamp
  - name: value1
    types:
    - numeric
    - date
    - timestamp
  - name: value2
    types:
    - numeric
    - date
    - timestamp
  surfaces:
  - pattern:
    - '{column}'
    - BETWEEN
    - '{value1}'
    - AND
    - '{value2}'
    commutative: false
  - pattern:
    - '{column}'
    - between
    - '{value}'
    commutative: false
```

```
  - pattern:
    - '{column}'
    - is
    - between
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - in
    - the
    - range
    - of
    - '{value}'
    commutative: false
  applicable_types:
    column:
    - numeric
    - date
    - timestamp
    value1:
    - numeric
    - date
    - timestamp
    value2:
    - numeric
    - date
    - timestamp
  label_rules: []
  aliases:
  - between
  - in the range of
  - is between
in:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - any
  - name: values
    types:
    - any
  surfaces:
  - pattern:
    - '{column}'
    - IN
    - (
    - '{values}'
    - )
    commutative: false
  - pattern:
    - '{column}'
    - in
```

```yaml
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - is
    - in
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - is
    - one
    - of
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - one
    - of
    - '{value}'
    commutative: false
  applicable_types:
    column:
    - any
    values:
    - any
  label_rules: []
  aliases:
  - in
  - is in
  - is one of
  - one of
like:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - text
  - name: value
    types:
    - text
  surfaces:
  - pattern:
    - '{column}'
    - LIKE
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - like
    - '{value}'
    commutative: false
```

```yaml
    - pattern:
      - '{column}'
      - matches
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - starts
      - with
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - ends
      - with
      - '{value}'
      commutative: false
    applicable_types:
      column:
      - text
      value:
      - text
    label_rules: []
    aliases:
    - ends with
    - like
    - matches
    - starts with
  is_null:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - IS
      - 'NULL'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - 'null'
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - empty
      - '{value}'
      commutative: false
```

```
    - pattern:
      - '{column}'
      - has
      - 'no'
      - value
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - missing
      - '{value}'
      commutative: false
    applicable_types:
      column:
      - any
    label_rules: []
    aliases:
    - has no value
    - is empty
    - is missing
    - is null
    - is_null
  is_not_null:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - IS
      - NOT
      - 'NULL'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - not
      - 'null'
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - not
      - empty
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
```

```
      - has
      - a
      - value
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - exists
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - present
      - '{value}'
      commutative: false
    applicable_types:
      column:
      - any
    label_rules: []
    aliases:
    - exists
    - has a value
    - is not empty
    - is not null
    - is present
    - is_not_null
_diagnostics:
  entity_type: _meta
  metadata:
    alias_collisions:
    - alias: amount
      meanings:
      - canonical: balance
        type: column
      - canonical: price
        type: column
    - alias: area
      meanings:
      - canonical: regions
        type: table
      - canonical: st_area
        type: postgis_actions
    - alias: combine
      meanings:
      - canonical: concat
        type: sql_actions
      - canonical: st_union
        type: postgis_actions
    - alias: length of
      meanings:
      - canonical: length
        type: sql_actions
      - canonical: st_length
```

```yaml
    type: postgis_actions
- alias: perimeter
  meanings:
  - canonical: boundaries
    type: column
  - canonical: st_perimeter
    type: postgis_actions
prefix_collisions:
- alias: amount
  longer_keys:
  - amount sold
- alias: account
  longer_keys:
  - account balance
  - account name
  - account number
  - account status
- alias: client
  longer_keys:
  - client id
- alias: customer
  longer_keys:
  - customer id
- alias: user
  longer_keys:
  - user id
  - user identifier
  - user ids
  - user name
  - user names
  - user number
- alias: item
  longer_keys:
  - item count
  - item name
  - item sold
- alias: product
  longer_keys:
  - product name
  - product names
- alias: quantity
  longer_keys:
  - quantity of
- alias: order
  longer_keys:
  - order by
  - order by ascending
  - order by descending
  - order date
  - order id
  - order number
- alias: purchase
  longer_keys:
  - purchase date
```

```
    - purchase id
  - alias: sale
    longer_keys:
    - sale date
    - sale dates
    - sale id
    - sale ids
  - alias: transaction
    longer_keys:
    - transaction date
    - transaction id
  - alias: boundary
    longer_keys:
    - boundary length
  - alias: geometry
    longer_keys:
    - geometry type
  - alias: outline
    longer_keys:
    - outline length
  - alias: shape
    longer_keys:
    - shape type
  - alias: area
    longer_keys:
    - area around
    - area id
    - area of
    - area shape
  - alias: region
    longer_keys:
    - region id
    - region ids
    - region number
  - alias: zone
    longer_keys:
    - zone id
  - alias: get
    longer_keys:
    - get the
  - alias: what
    longer_keys:
    - what kind of shape
  - alias: at
    longer_keys:
    - at least
    - at most
  - alias: not
    longer_keys:
    - not less than
    - not more than
  - alias: equal
    longer_keys:
    - equal to
```

```
- alias: is
  longer_keys:
  - is active
  - is actives
  - is adjacent to
  - is at least
  - is at most
  - is between
  - is empty
  - is exactly
  - is greater than
  - is greater than or equal to
  - is in
  - is inside
  - is less than
  - is less than or equal to
  - is missing
  - is not
  - is not empty
  - is not equal to
  - is not null
  - is null
  - is one of
  - is present
  - is the same as
  - is within
- alias: in
  longer_keys:
  - in ascending order
  - in descending order
  - in the range of
- alias: number
  longer_keys:
  - number of
  - number of items
  - number sold
- alias: sum
  longer_keys:
  - sum of
- alias: total
  longer_keys:
  - total amount of
  - total number of
  - total of
- alias: average
  longer_keys:
  - average of
- alias: most
  longer_keys:
  - most recent login
- alias: unique
  longer_keys:
  - unique values of
- alias: length
```

```yaml
  longer_keys:
  - length along
  - length in
  - length of
  - length of boundary
- alias: convert
  longer_keys:
  - convert coordinate system
- alias: distance
  longer_keys:
  - distance along
  - distance between
  - distance of
- alias: buffer
  longer_keys:
  - buffer around
- alias: union
  longer_keys:
  - union of
- alias: center
  longer_keys:
  - center point
- alias: simplify
  longer_keys:
  - simplify shape
- alias: closest
  longer_keys:
  - closest to
surface_warnings: []
inferred_args: []
surfaces_args_mismatch:
- canonical: between
  args:
  - column
  - value1
  - value2
  placeholders:
  - column
  - value
  - value1
  - value2
- canonical: in
  args:
  - column
  - values
  placeholders:
  - column
  - value
  - values
- canonical: is_null
  args:
  - column
  placeholders:
  - column
```

```
    - value
  - canonical: is_not_null
    args:
    - column
    placeholders:
    - column
    - value
  missing_applicable_types: []
  connectors:
  - _on
  - and
  - at
  - belonging to
  - between
  - from
  - in
  - of
  - 'on'
  - to
  - with
_binder_builder:
  connectors_debug:
    _connectors_from_graph: []
    _connectors_final_map:
      COMMA: ','
      AND: and
      AT: at
      BETWEEN: between
      FROM: from
      IN: in
      OF: of
      'ON': 'on'
      TO: to
      WITH: with
    _punctuation_final_map:
      ',': ','
_table_meta:
  users:
    aliases:
    - account
    - accounts
    - client
    - clients
    - customer
    - customers
    - people
    - user
    - users
    columns:
    - user_id
    - username
    - age
    - balance
    - is_active
```

```
  - last_login
  - location
sales:
  aliases:
  - order
  - orders
  - purchase
  - purchases
  - sale
  - sales
  - transaction
  - transactions
  columns:
  - sale_id
  - user_id
  - product_name
  - sale_date
  - quantity
  - price
regions:
  aliases:
  - area
  - areas
  - district
  - region
  - regions
  - territory
  - zone
  - zones
  columns:
  - region_id
  - name
  - boundaries
```

build_query_dynamics.txt

\#\# The Refined Architecture: The "Canonical Space"

The workflow is now split between a "Build-Time" phase that compiles knowledge and a "Run-Time" phase that intelligently processes queries.

`User Query  Tokenizer & Tagger  Multi-Stage Normalizer  Canonical Token Stream  Canonical Parser  Transformer  SQL`

\#\# 1. The "Build-Time" Phase: The Knowledge Compiler

Your `generate_graph_and_grammar.py` script evolves into a "Knowledge Compiler." It analyzes the `relationship_graph.yaml` to make intelligent decisions and produces two critical artifacts for the run-time engine.

\#\#\# Conflict-Aware Decision Logic

Without a `handling_strategy` key, the engine must "figure it out." It does this by performing **conflict detection**:

1. It builds a master vocabulary of every alias, keyword, and entity name from the graph.
2. It identifies any term that appears in multiple roles. For example, `"of"` is a `preposition` and also appears in the `pattern` for `st_distance`. This is a conflict. `"users"` might be a `table_name` and an alias for a `user_id` column. This is also a conflict.

Based on this analysis, it generates its two outputs:

\#\#\# Output 1: The Lean, Canonical Grammar

The `generated_grammar.lark` file becomes extremely small, fast, and stable. It is defined **only** in terms of canonical tokens, not the hundreds of aliases.

```lark
// A small, fast grammar that only understands canonical terms
start: query
query: select_statement

// These are now abstract tokens, not long lists of strings
column_name: CANONICAL_COLUMN
table_name: CANONICAL_TABLE
function_call: CANONICAL_FUNCTION

// The rules enforce the *relationships* between canonical terms
select_statement: SELECT_VERB column_list from_clause
column_list: column_name ("," column_name)* ("and" column_name)?
from_clause: "from" table_name
```

\#\#\# Output 2: The Normalization Map

This new file, perhaps `normalization_map.yaml`, is the brain of the Normalizer. It contains the instructions for mapping user input to the canonical space.

```yaml
# normalization_map.yaml
deterministic_aliases:
  # Maps all unique aliases to their one true canonical form
  "user name": "username"
  "customer": "users"
  "avg": "average"
  "how far": "distance"

contextual_rules:
  # Rules for resolving ambiguous keywords based on context
  "of":
    # If "of" appears after a distance function, it's part of that function call
    - context_before: [FUNCTION_ALIAS(st_distance)]
      action: "MERGE_INTO_FUNCTION"
    # A default rule might be to treat it as a filler word to be ignored
    - context_before: [ANY]
      action: "DISCARD"
```

\#\# 2. The "Run-Time" Phase: The Intelligent Interpreter

Your `process_query.py` script now executes this intelligent, multi-stage pipeline.

\#\#\# Step 2a: Tokenization and Tagging

Before normalization, the input string is broken into tokens, and each token is "tagged" with all its possible meanings from the graph.

  * **Input:** `"show average of sales price from customers"`
      * **Tagged Output:** `[TOKEN("show", type=SELECT_VERB), TOKEN("average", type=FUNCTION_ALIAS, canonical=avg), TOKEN("of", type=PREPOSITION), TOKEN("sales", type=TABLE_ALIAS, canonical=sales), TOKEN("price", type=COLUMN_ALIAS, canonical=price), TOKEN("from", type=PREPOSITION), TOKEN("customers", type=TABLE_ALIAS, canonical=users)]`

\#\#\# Step 2b: The Multi-Stage Normalizer

This component works on the tagged token stream.

1. **Deterministic Alias Pass:** This is the workhorse. It iterates through the tokens and replaces any alias with a single, unambiguous meaning with its canonical form. This step handles the vast majority of language variations.

2. **Contextual Disambiguation Pass:** This is the "pre-grammar gating" you described. It applies the `contextual_rules`. For the ambiguous word `"of"`, it looks at the preceding token. If it was `TOKEN(..., canonical=st_distance)`, it knows this `"of"` is part of the distance function and can be handled accordingly. This intelligently prunes

invalid paths early.

3. **Non-Deterministic Pass:** This is the final resort for any remaining ambiguity. As you said, it can try multiple options. If the token `"sales"` was tagged as both a `TABLE_ALIAS` and a `COLUMN_ALIAS`, this stage could generate two candidate canonical streams for the parser to try.

\#\#\# Step 2c: The Canonical Grammar as Gatekeeper

You are correct, the grammar's new role is to be a **gatekeeper**. It receives a clean stream of canonical tokens from the Normalizer. Its job is no longer to understand messy language, but to validate the final structure.

  * It ensures the query has all the necessary parts (`SELECT` and `FROM`).
  * It validates the relationships between canonical terms (e.g., ensuring a function is in the `SELECT` clause, not the `FROM` clause).
  * As you noted, it acts as a filter. If the Normalizer produces three candidate phrases, but only one is structurally valid according to the canonical grammar, that's the one that succeeds. The bad paths are simply discarded.

This architecture perfectly captures your refinements. It's an intelligent system that separates concerns, using a powerful normalization pipeline to handle linguistic variety and a lean, simple grammar to enforce structural correctness, all while being driven by the knowledge compiled from your YAML files.

canonical_grammar.lark

```
start: query
query: select_statement
// --- High-Priority Keyword Terminals ---
AND: "and"
AT: "at"
BELONGING_TO: "belonging to"
FROM: "from"
NOT: "not"
OF: "of"
OR: "or"
SELECT: "select"
COMMA: ","

// --- CANONICAL ENTITY TERMINALS ---
CANONICAL_TABLE: "regions" | "sales" | "users"
CANONICAL_COLUMN: "age" | "balance" | "boundaries" | "is_active" | "last_login" | "location" | "name" | "price" | "product_name" | "quantity" | "region_id" | "sale_date" | "sale_id" | "user_id" | "username"
CANONICAL_FUNCTION: "avg" | "cast" | "concat" | "count" | "distinct" | "extract" | "group_by" | "having" | "length" | "limit" | "max" | "min" | "order_by_asc" | "order_by_desc" | "st_area" | "st_buffer" | "st_centroid" | "st_contains" | "st_crosses" | "st_distance" | "st_distance_operator" | "st_geometrytype" | "st_intersects" | "st_length" | "st_perimeter" | "st_simplify" | "st_spatial_index" | "st_touches" | "st_transform" | "st_union" | "st_within" | "st_x" | "st_y" | "sum"

// --- MAIN PARSER RULES ---
selectable: column_name | function_call
select_statement: SELECT column_list from_clause
column_name: CANONICAL_COLUMN
table_name: CANONICAL_TABLE
function_call: CANONICAL_FUNCTION (OF column_list)?
column_list: selectable (COMMA selectable)* (COMMA? AND selectable)?
from_clause: (FROM | OF) table_name

%import common.WS
%ignore WS
```

config.py

```python
# File: config.py

# --- Input Files ---
SCHEMA_PATH = "schema.yaml"
KEYWORDS_PATH = "keywords_and_functions.yaml"

# --- Generated Artifacts ---
GRAPH_PATH = "relationship_graph.yaml"
NORMALIZATION_MAP_PATH = "normalization_map.yaml"
GRAMMAR_PATH = "canonical_grammar.lark"

# --- Test Settings ---
DB_PATH = "test.db"
STATIC_TEST_CASES = [
    "show username and age from users",
    "list total of price from sales",
    "what is the name of all regions",
]
GOLDEN_QUERIES_PATH = "golden_queries.yaml" # For a future validator
```

create_pdf_archive.py

```python
import os
import argparse
from fpdf import FPDF, XPos, YPos

# A set of common directories and file extensions to ignore.
IGNORED_DIRS = {'.git', '__pycache__', '.vscode', '.idea', 'node_modules', 'venv'}
IGNORED_EXTENSIONS = {
    '.pyc', '.pyo', '.o', '.a', '.so', '.lib', '.dll', '.exe',
    '.img', '.iso', '.zip', '.tar', '.gz', '.rar', '.7z',
    '.png', '.jpg', '.jpeg', '.gif', '.bmp', '.tiff', '.ico',
    '.pdf', '.doc', '.docx', '.xls', '.xlsx', '.ppt', '.pptx',
    '.mp3', '.wav', '.mp4', '.mov', '.avi', '.mkv', '.db'
}

def add_file_to_pdf(pdf, file_path, base_dir):
    """
    Reads the content of a file and adds it to the PDF object.
    The file's relative path is used as a header.
    """
    relative_path = os.path.relpath(file_path, base_dir)

    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()
    except UnicodeDecodeError:
        print(f"  Skipping non-UTF-8 file (likely binary): {relative_path}")
        return
    except Exception as e:
        print(f" Error reading file {relative_path}: {e}")
        return

    # --- KEY CHANGE HERE ---
    # Sanitize content by encoding it to latin-1 and ignoring any characters
    # that can't be represented. This prevents errors with Unicode characters.
    content = content.encode('latin-1', 'ignore').decode('latin-1')
    # --- END OF CHANGE ---

    pdf.add_page()

    # --- Header (File Path) ---
    pdf.set_font("Helvetica", "B", 12)
    pdf.cell(0, 10, relative_path, 0, new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='L')
    pdf.ln(5)

    # --- Body (File Content) ---
    pdf.set_font("Courier", "", 10)
    pdf.multi_cell(0, 5, content)

    print(f" Added to PDF: {relative_path}")

def main():
```

```
    """
    Main function to parse arguments and generate the PDF.
    """
    parser = argparse.ArgumentParser(
         description="Recursively scan a directory and compile all text-based files into
a single PDF.",
        formatter_class=argparse.RawTextHelpFormatter
    )
    parser.add_argument("source_dir", help="The source directory to scan.")
      parser.add_argument("output_pdf", help="The name of the output PDF file (e.g.,
'archive.pdf').")

    args = parser.parse_args()

    if not os.path.isdir(args.source_dir):
        print(f"Error: Source directory '{args.source_dir}' not found.")
        return

    pdf = FPDF()
    pdf.set_auto_page_break(auto=True, margin=15)

    for root, dirs, files in os.walk(args.source_dir, topdown=True):
        dirs[:] = [d for d in dirs if d not in IGNORED_DIRS]
        files.sort()

        for filename in files:
            _, extension = os.path.splitext(filename)
            if extension.lower() in IGNORED_EXTENSIONS:
                print(f"  Skipping by extension: {filename}")
                continue

            file_path = os.path.join(root, filename)
            add_file_to_pdf(pdf, file_path, args.source_dir)

    try:
        pdf.output(args.output_pdf)
        print(f"\n Successfully created PDF: {args.output_pdf}")
    except Exception as e:
        print(f"\n Failed to create PDF. Error: {e}")

if __name__ == "__main__":
    main()
```

database_onboarding_and_qa_pipeline.txt

***

The Onboarding and QA Pipeline

This pipeline is the standard operating procedure for integrating a new database. It's a script (e.g., `run_validation.py`) that executes a series of **Generators** and **Validation Checks** in a specific order. The "flight recorder" debug logs are essential, providing a detailed audit trail if any check fails.

Phase 1: Graph Generation and Quality Assurance

Goal: To create a unified `relationship_graph.yaml` and validate its integrity.

G1: Graph Generator

* **Responsibility:** Your existing `graph_builder.py` script. It takes the new database's `schema.yaml` and the universal `keywords_and_functions.yaml` to produce a single, comprehensive `graph.yaml`.

V1: Graph QA Checks

* **Responsibility:** A new `test_graph.py` script that validates the output of G1.
* **V1.1: Content Completeness Check:**
    * **Purpose:** To guarantee that no data was lost or corrupted during graph creation.
    * **Process:** The validator loads the two source YAMLs and the output graph. It programmatically iterates through every table, column, function, and keyword from the sources and asserts that a corresponding, correctly formed node exists in the graph. The flight recorder logs which entities it's checking.
* **V1.2: Logical Cohesion Check:**
    * **Purpose:** To perform a sanity check on the relationships within the graph.
    * **Process:** The validator iterates through every column in the graph and checks it against every function. Using your `is_compatible` logic, it asserts that **every column has at least one compatible function**. This prevents "orphaned" columns and confirms the type/label system is coherent.

Phase 2: Knowledge Compilation and Full-System QA

Goal: To compile the verified graph into run-time artifacts and run a comprehensive suite of checks to ensure the entire system works for the new database.

G2: Knowledge Compiler

* **Responsibility:** Your `knowledge_compiler.py` script. It takes the validated `graph.yaml` and produces the two key run-time artifacts: `normalization_map.yaml` and `canonical_grammar.lark`.

V2: Artifact and Component QA Checks

These checks validate the outputs of G2 and the behavior of the individual run-time components.

* **V2.1: Map Coverage Check:** Verifies that every single alias defined anywhere in the `graph.yaml` exists as a key in the `normalization_map.yaml`.
* **V2.2: Grammar Vocabulary Check:** Verifies that the `canonical_grammar.lark` contains a terminal definition for every canonical term found in the graph.
* **V2.3: Canonical Phrase "Smoke Test":** A simple, deterministic check of the core grammar rules. It programmatically builds perfect, canonical phrases (e.g., `"select age from users"`) by picking valid canonical tokens from the graph and asserts the grammar can parse them.
* **V2.4: Grammar Stress Test:** This is the `SmartGenerator`. It uses the graph and the canonical grammar to generate hundreds of complex, valid canonical phrases to ensure the grammar has no structural flaws or ambiguities.
* **V2.5: Normalizer Spot-Check:** A unit-level check that takes a sample of important aliases from the `normalization_map` (e.g., `"user name"`, `"avg"`, `"the"`) and asserts that the `Normalizer` class correctly maps them to their canonical form (`"username"`, `"avg"`, `None`).
* **V2.6: Normalizer-Parser Integration Check:** This is a crucial test. It takes valid canonical phrases from the Grammar Stress Test (V2.4), "de-normalizes" them by replacing canonical terms with random aliases, and then asserts that the `Normalizer -> Parser` pipeline can successfully process them back into a valid parse tree.

V3: End-to-End System QA

This is the final certification stage, testing the entire system against realistic inputs.

* **V3.1: "Golden Set" Validation:** Tests the full `Normalizer -> Parser -> Transformer` pipeline against a static, curated list of human-written or LLM-generated natural language queries. This ensures the system can handle real-world phrasing.
* **V3.2: Live Database Execution Check:** The ultimate test. It takes the "golden set" queries, generates the final SQL, and executes it against the actual database (`test.db`). The check passes if the SQL executes without raising a database error. This confirms that the generated SQL is not just syntactically correct, but also semantically valid for the target database schema.

generate_graph_and_grammar.py

```
import yaml
import os
import re
from collections import defaultdict
from string import Template
from lark import Lark, Visitor, ParseError
from lark.exceptions import UnexpectedToken
import random
import lark
from lark.lexer import Token
# Correctly import all necessary classes
from lark.grammar import NonTerminal, Terminal

# --- File Paths ---
SCHEMA_PATH = "/app/schema.yaml"
KEYWORDS_PATH = "/app/keywords_and_functions.yaml"
GRAPH_PATH = "relationship_graph.yaml"
GRAMMAR_PATH = "generated_grammar.lark"
NORMALIZATION_MAP_PATH = "normalization_map.yaml" # The new output file

# --- Phase 1: Graph Generation Logic (No changes needed here) ---
def load_yaml(path):
    try:
        with open(path, 'r') as f:
            return yaml.safe_load(f)
    except FileNotFoundError:
        print(f"Error: YAML file not found at {path}")
        return None
    except yaml.YAMLError as e:
        print(f"Error: Failed to parse YAML from {path}")
        print(e)
        return None

def is_compatible(column_info, action_info):
    compatible_vars = {}
    applicable_types_dict = action_info.get('applicable_types')
    if not isinstance(applicable_types_dict, dict):
        return {}
    for var, allowed_types in applicable_types_dict.items():
        column_type = column_info['type']
        if 'any' in allowed_types or column_type in allowed_types:
            valid_labels = True
            for rule in action_info.get('label_rules', []):
                if rule.startswith('not '):
                    label_to_exclude = rule[4:]
                    if label_to_exclude in column_info['labels']:
                        valid_labels = False
                        break
                else:
                    if rule not in column_info['labels']:
                        valid_labels = False
```

```
                        break
            if valid_labels:
                compatible_vars[var] = True
    return compatible_vars

def generate_relationship_graph(schema_yaml, keywords_yaml):
    """
    Builds the graph with a CONSISTENT data structure for all nodes,
    correctly handling all structures from schema.yaml and keywords.yaml.
    """
    graph = defaultdict(lambda: {'entity_type': 'unknown', 'metadata': {}})

    # Process Schema (Handles tables and columns)
    if schema_yaml and 'tables' in schema_yaml:
        for table_name, table_data in schema_yaml['tables'].items():
            graph[table_name]['entity_type'] = 'table'
            graph[table_name]['metadata'] = table_data
            for column_name, column_data in table_data['columns'].items():
                graph[column_name]['entity_type'] = 'column'
                graph[column_name]['metadata'] = column_data

    # Process Keywords and Actions from keywords.yaml
    if keywords_yaml:
        # Handle the top-level 'keywords' section
        if 'keywords' in keywords_yaml and isinstance(keywords_yaml['keywords'], dict):
            for keyword_type, keyword_data in keywords_yaml['keywords'].items():
                # keyword_type is 'select_verbs', 'prepositions', etc.
                # keyword_data is the content under that key.

                # --- FIX IS HERE: Handle both dict and list formats ---
                if isinstance(keyword_data, dict):
                    # This handles structures like {select: [aliases]} or {equal:
{metadata}}
                    for canonical_name, metadata_val in keyword_data.items():
                        graph[canonical_name]['entity_type'] = keyword_type
                        if isinstance(metadata_val, list):
                            graph[canonical_name]['metadata'] = {'aliases':
metadata_val}
                        else: # It's a dict like for comparison_operators
                            graph[canonical_name]['metadata'] = metadata_val
                elif isinstance(keyword_data, list):
                    # This handles the old, simple list format like prepositions: [- of,
- from]
                    for keyword in keyword_data:
                        graph[keyword]['entity_type'] = keyword_type
                        graph[keyword]['metadata'] = {'aliases': [keyword]}

        # Handle 'sql_actions' and 'postgis_actions'
        for action_type in ['sql_actions', 'postgis_actions']:
            if action_type in keywords_yaml and isinstance(keywords_yaml[action_type],
dict):
                for canonical_name, metadata in keywords_yaml[action_type].items():
                    graph[canonical_name]['entity_type'] = action_type
                    graph[canonical_name]['metadata'] = metadata
```

```
    return dict(graph)

# --- Phase 2: Knowledge Compiler ---
# --- Helper Function ---
def _extract_entities(graph):
        """Traverses the graph and organizes all entities by type into a clean dictionary."""
    entities = defaultdict(dict)
    for key, node in graph.items():
        entity_type = node.get('entity_type')
        if entity_type:
            entities[entity_type][key] = node
    return entities

# --- Helper Function ---
def _build_normalization_map(entities):
    """
    Creates a map of all known aliases to their canonical names.
    This version correctly processes the categorized entities.
    """
    normalization_map = {'deterministic_aliases': {}}

    # Define all entity types that have aliases
    entity_types_to_map = [
        'table', 'column', 'sql_action', 'postgis_action', 'select_verbs',
        'prepositions', 'logical_operators', 'comparison_operators', 'filler_words'
    ]

    for entity_type in entity_types_to_map:
        for canonical_name, data_node in entities.get(entity_type, {}).items():
            metadata = data_node.get('metadata', {})
            # Ensure metadata is a dict before proceeding
            if isinstance(metadata, dict):
                # The canonical name maps to itself (lowercase for matching)
                     normalization_map['deterministic_aliases'][canonical_name.lower()] = canonical_name
                # All aliases also map to the canonical name
                for alias in metadata.get('aliases', []):
                             normalization_map['deterministic_aliases'][alias.lower()] = canonical_name

    return normalization_map

# --- Helper Function ---
def _build_canonical_grammar(graph):
       """Builds the lean grammar that only operates on a small set of canonical keywords."""
    grammar_parts = []

    # --- Identify the few essential canonical keywords ---
    # We assume simple canonical names like "select", "from", "of", "and", "or", "not"
    select_verb_canonical = "select" # Assuming 'select' is the canonical verb
```

```
    # Find all canonical preposition and logical operator names from the graph
    prepositions = {k for k, v in graph.items() if v['entity_type'] == 'prepositions'}
    logical_ops = {k for k, v in graph.items() if v['entity_type'] == 'logical_operators'}

    grammar_parts.extend([
        '// --- Lean Canonical Grammar ---',
        'start: query',
        'query: select_statement',
        '\n// --- CORE KEYWORD TERMINALS ---',
        f'SELECT: "{select_verb_canonical}"',
        'AND: "and"',
        'OR: "or"',
        'NOT: "not"',
        'COMMA: ","',
        'OF: "of"',
        'FROM: "from"',
        '\n// --- CANONICAL ENTITY TERMINALS ---',
        'CANONICAL_TABLE: BARE_WORD',
        'CANONICAL_COLUMN: BARE_WORD',
        'CANONICAL_FUNCTION: BARE_WORD',
        '\n// --- MAIN PARSER RULES ---',
        'selectable: column_name | function_call',
        'select_statement: SELECT column_list from_clause', # Uses SELECT token
        'column_name: CANONICAL_COLUMN',
        'table_name: CANONICAL_TABLE',
        'function_call: CANONICAL_FUNCTION (OF column_list)?',
        'column_list: selectable (COMMA selectable)*',
        'from_clause: FROM table_name', # Uses FROM token
        '\n// --- PRIMITIVES ---',
        'BARE_WORD: /[a-z0-9_]+/', # Now lowercase only
        '%import common.WS',
        '%ignore WS'
    ])

    return "\n".join(grammar_parts)

# --- Main Orchestrator ---
def compile_knowledge_artifacts(graph):
    print("Compiling knowledge artifacts...")

    # 1. Build the Normalization Map
    normalization_map = _build_normalization_map(graph)
    with open(NORMALIZATION_MAP_PATH, 'w') as f:
        yaml.dump(normalization_map, f, sort_keys=False)
    print(f"Normalization map successfully written to '{NORMALIZATION_MAP_PATH}'.")

    # 2. Build the Canonical Grammar
    grammar_string = _build_canonical_grammar(graph)
    with open(GRAMMAR_PATH, 'w') as f:
        f.write(grammar_string)
    print(f"Canonical grammar successfully written to '{GRAMMAR_PATH}'.")
```

```
# --- Phase 3: Validation Logic (No changes needed here) ---
def validate_grammar(grammar_path, start_rule, test_cases):
    try:
        with open(grammar_path, 'r') as f:
            grammar = f.read()
        parser = Lark(grammar, start=start_rule, parser='earley')

        for case in test_cases:
            try:
                print(f"--- Testing: '{case}' ---")
                tree = parser.parse(case)
                print(tree.pretty())
            except (ParseError, UnexpectedToken) as e:
                print(f"!!! PARSING FAILED for '{case}': {e}")
    except Exception as e:
        print(f"Error loading grammar: {e}")


# --- Phase 4: FINAL String Generation Logic ---
class GrammarAnalyzer:
    """
    Analyzes a Lark grammar to calculate the minimum RECURSION DEPTH
    required to fully expand each rule.
    """
    def __init__(self, parser):
        self.parser = parser
        self.rule_lookup = defaultdict(list)
        for rule in self.parser.rules:
            key = rule.origin.name.value if hasattr(rule.origin.name, 'value') else
rule.origin.name
            self.rule_lookup[key].append(rule)

        self.min_depths = {}
        self._calculate_min_depths()

    def _get_min_depth(self, term_name):
        # Terminals are the end of the line; they require 1 expansion step to resolve.
        if term_name.isupper() or term_name.startswith('"') or term_name in ["AND",
"COMMA", "OF", "FROM"]:
            return 1
        return self.min_depths.get(term_name, float('inf'))

    def _calculate_min_depths(self):
        # Iterate until the depths stabilize
        for _ in range(len(self.rule_lookup) + 2):
            for rule_name, expansions in self.rule_lookup.items():
                # The depth of a rule is 1 (for itself) + the minimum sum of its
children's depths.
                min_expansion_depth = min(
                    sum(self._get_min_depth(t.name) for t in r.expansion) if r.expansion
else 0
                    for r in expansions
                )
```

```
                total_min_depth = 1 + min_expansion_depth

                            if  rule_name  not  in  self.min_depths  or  total_min_depth  <
self.min_depths[rule_name]:
                    self.min_depths[rule_name] = total_min_depth

class SmartGenerator:
    """
    The final, robust generator. It uses two-pass analysis, a conservative
    budget-aware expansion strategy, and a recursion tracker to prevent
    infinite loops, with full conditional logging.
    """
    def __init__(self, parser, graph, analyzer):
        self.parser = parser
        self.graph = graph
        self.analyzer = analyzer
        self.rule_lookup = analyzer.rule_lookup

        # A hard limit on how many times a rule can be nested within itself.
        self.RECURSION_LIMIT = 4

        self.vocab = {
               "CANONICAL_COLUMN": [k  for  k,  v  in  graph.items()  if  v['entity_type']  ==
'column'],
                "CANONICAL_TABLE": [k  for  k,  v  in  graph.items()  if  v['entity_type']  ==
'table'],
               "CANONICAL_FUNCTION": [k  for  k,v  in  graph.items()  if  v['entity_type']  in
['sql_action', 'postgis_action']],
        }

    def generate(self, start_rule="query", max_depth=25):
        """
        Public method to start generation. Always returns a (result, log) tuple.
        """
        debug_log = []
        # --- FIX: Start the expansion with an empty recursion tracker ---
        result = self._expand(start_rule, max_depth, debug_log, "", {})

        if result is None:
            return None, debug_log
        return result, None

    def _expand(self, rule_name, depth, debug_log, indent, recursion_counts):
        log_msg = f"{indent}>> Expanding '{rule_name}' with depth_budget={depth}"

        if debug_log is not None:
            debug_log.append(log_msg)

        if depth <= 0:
            return None

        # --- RECURSION TRACKER LOGIC (START) ---
        # Copy the counts for this branch and increment the count for the current rule.
        current_counts = recursion_counts.copy()
```

```
        current_counts[rule_name] = current_counts.get(rule_name, 0) + 1
        # --- END RECURSION TRACKER ---

        # Base Case: The item is a terminal
        if rule_name not in self.rule_lookup:
            value = None
            if rule_name in self.vocab: value = random.choice(self.vocab[rule_name])
            else:
                for term_def in self.parser.terminals:
                    if term_def.name == rule_name:
                        choices = term_def.pattern.value.replace('(?:', '').rstrip(')').split('|')
                        value = random.choice([c.strip('"') for c in choices])
                        break
                if value is None: value = rule_name.strip('"')
            if debug_log is not None:
                debug_log.append(f"{indent}<< Returning terminal value: '{value}'")
            return value

        # --- It's a Rule, so we expand it ---
        possible_rules = self.rule_lookup[rule_name]
        valid_choices = possible_rules

        # --- RECURSION TRACKER LOGIC (CHOICE PRUNING) ---
        # If we've hit the recursion limit for this rule, we MUST choose a non-recursive path.
        if current_counts[rule_name] > self.RECURSION_LIMIT:
            if debug_log is not None:
                debug_log.append(f"{indent}   -- RECURSION LIMIT HIT for '{rule_name}'. Forcing a base case.")
            # Filter to only "base cases" - expansions that do not call the rule again.
            base_cases = [r for r in possible_rules if rule_name not in [t.name for t in r.expansion]]
            if not base_cases:
                if debug_log is not None:
                    debug_log.append(f"{indent}   !! FAILED: No base case available to terminate recursion.")
                return None
            valid_choices = base_cases
        # --- END RECURSION TRACKER ---

        # --- CONSERVATIVE CHOICE LOGIC (from your working version) ---
        chosen_rule = None
        if depth < (self.analyzer.min_depths.get(rule_name, 1) + 5):
            path_costs = {r: 1 + sum(self.analyzer.min_depths.get(t.name, 0) for t in r.expansion) for r in valid_choices}
            min_path_cost = min(path_costs.values())
            safest_choices = [r for r, cost in path_costs.items() if cost == min_path_cost]
            chosen_rule = random.choice(safest_choices)
            if debug_log is not None:
                debug_log.append(f"{indent}   -- CONSERVATIVE MODE: Chose safest path with cost {min_path_cost}")
        else:
```

```
            chosen_rule = random.choice(valid_choices)

        if debug_log is not None:
            debug_log.append(f"{indent}  -> Chose path: {[t.name for t in chosen_rule.expansion]}")

        if not chosen_rule.expansion:
            return ""

        parts = []
        for term in chosen_rule.expansion:
            # Pass the updated counts dictionary to the recursive call
            part = self._expand(term.name, depth - 1, debug_log, indent + "  ", current_counts)
            if part is None:
                if debug_log is not None:
                    debug_log.append(f"{indent}<< FAILED: Child '{term.name}' failed to expand.")
                return None
            parts.append(part)

        result = " ".join(filter(lambda x: x != "", parts)).strip()
        if debug_log is not None:
            debug_log.append(f"{indent}<< Success for '{rule_name}': returning '{result[:50]}...'")
        return result



# --- Main Workflow (No changes needed here) ---
if __name__ == '__main__':
    print("\n--- PHASE 1: GENERATING RELATIONSHIP GRAPH ---")
    schema_data = load_yaml(SCHEMA_PATH)
    keywords_data = load_yaml(KEYWORDS_PATH)
    if not schema_data or not keywords_data:
        exit()
    relationship_graph = generate_relationship_graph(schema_data, keywords_data)
    with open(GRAPH_PATH, 'w') as f:
        yaml.dump(relationship_graph, f, sort_keys=False)
    print(f"Relationship graph successfully written to '{GRAPH_PATH}'.")

    print("\n--- PHASE 2: GENERATING LARK GRAMMAR ---")
    # Call the new orchestrator function
    compile_knowledge_artifacts(relationship_graph)

    print("\n--- PHASE 3: VALIDATING GRAMMAR WITH STATIC TESTS ---")
    static_test_cases = [
        "show username and age of users",
        "list username of users",
        "what age and username of users",
    ]
    validate_grammar(GRAMMAR_PATH, 'query', static_test_cases)

    print("\n--- PHASE 4: VALIDATING GRAMMAR WITH DYNAMIC TESTS ---")
```

```
    try:
        with open(GRAMMAR_PATH, 'r') as f:
            grammar_content = f.read()
                parser  =  lark.Lark(grammar_content,  start='query',  parser='earley',
ambiguity='explicit')

        # --- FIX IS HERE: Create the Analyzer and the SmartGenerator ---
        print("Analyzing grammar for smart generation...")
        analyzer = GrammarAnalyzer(parser)
        generator = SmartGenerator(parser, relationship_graph, analyzer)
        # --- END OF FIX ---

        print("\n--- Testing dynamic phrases ---")
        success_count = 0
        for i in range(100):
            phrase, debug_log = generator.generate()

            if phrase is None:
                 print(f"--- Attempt {i+1}: Generator failed to produce a valid phrase.
---")
                if debug_log: # Print the log if it exists
                    print("\n--- DEBUG LOG (Generation Failure) ---")
                    for msg in debug_log:
                        print(msg)
                    print("------------------------------------\n")
                continue

            print(f"--- Testing generated phrase: '{phrase}' ---")
            try:
                tree = parser.parse(phrase)
                print("   +++ SUCCESS +++")
                success_count += 1
                # print(tree.pretty())
            except (ParseError, UnexpectedToken) as e:
                print(f"!!! PARSING FAILED for '{phrase}': {e}")
            except Exception as e:
                print(f"!!! UNEXPECTED ERROR for '{phrase}': {e}")

        print(f"\n--- Generation Complete: {success_count}/100 successes ---")

    except Exception as e:
        print(f"Error during dynamic testing phase: {e}")
```

keywords_and_functions.yaml


```yaml
keywords:
  select_verbs:
    # "select" is the canonical term for the grammar.
    # All words in the list are aliases that the Normalizer will map to "select".
    select:
      aliases:
        - show
        - list
        - what
        - get
        - display
        - find
        - retrieve
        - fetch
        - tell me
        - give me
  prepositions:
    # For keywords, the canonical name is the word itself.
    of:
      aliases: ["of"]
    from:
      aliases: ["from"]
    _on:
      aliases: ["on"]
    at:
      aliases: ["at"]
    belonging to:
      aliases: ["belonging to"]
  logical_operators:
    # The keys (and, or, not) are the canonical terms.
    and:
      aliases: ["and", "&&"]
      template: "{clause1} AND {clause2}"
    or:
      aliases: ["or", "||"]
      template: "{clause1} OR {clause2}"
    not:
      aliases: ["not", "!"]
      template: "NOT {clause}"
  comparison_operators:
    equal:
       aliases: ["is", "equals", equal to, '=', '==', is exactly, exactly, is the same
as]
      template: "{column} = {value}"
      applicable_types: {'column': [any], 'value': [any]}
      label_rules: []
      explanation: "Tests for equality between two values."
    not_equal:
      aliases: ["is not", '!=', '<>', is not equal to, does not equal, isn't]
      template: "{column} != {value}"
      applicable_types: {'column': [any], 'value': [any]}
```

```yaml
      label_rules: []
      explanation: "Tests for inequality between two values."
    greater_than:
      aliases: [is greater than, '>', more than, over, above, exceeds]
      template: "{column} > {value}"
      applicable_types: {'column': [numeric, date, timestamp], 'value': [numeric, date,
timestamp]}
      label_rules: []
      explanation: "Tests if the first value is greater than the second."
    less_than:
      aliases: [is less than, '<', smaller than, under, below]
      template: "{column} < {value}"
      applicable_types: {'column': [numeric, date, timestamp], 'value': [numeric, date,
timestamp]}
      label_rules: []
      explanation: "Tests if the first value is less than the second."
    greater_than_or_equal:
      aliases: [is greater than or equal to, '>=', at least, is at least, not less than]
      template: "{column} >= {value}"
      applicable_types: {'column': [numeric, date, timestamp], 'value': [numeric, date,
timestamp]}
      label_rules: []
      explanation: "Tests if the first value is greater than or equal to the second."
    less_than_or_equal:
      aliases: [is less than or equal to, '<=', at most, is at most, not more than, up
to]
      template: "{column} <= {value}"
      applicable_types: {'column': [numeric, date, timestamp], 'value': [numeric, date,
timestamp]}
      label_rules: []
      explanation: "Tests if the first value is less than or equal to the second."
    between:
      aliases: [between, is between, in the range of]
      template: "{column} BETWEEN {value1} AND {value2}"
      pattern: ["{value1}", "and", "{value2}"]
      applicable_types: {'column': [numeric, date, timestamp], 'value1': [numeric, date,
timestamp], 'value2': [numeric, date, timestamp]}
      label_rules: []
      explanation: "Tests if a value falls within a specified range."
    in:
      aliases: ["in", is in, is one of, one of]
      template: "{column} IN ({values})"
      applicable_types: {'column': [any], 'values': [any]}
      label_rules: []
      explanation: "Tests if a value is present in a given list of values."
    like:
      aliases: [like, matches, starts with, ends with]
      template: "{column} LIKE {value}"
      applicable_types: {'column': [text], 'value': [text]}
      label_rules: []
      explanation: "Performs a pattern matching search using wildcards."
    is_null:
      aliases: [is null, is empty, has no value, is missing]
      template: "{column} IS NULL"
```

```yaml
      applicable_types: {'column': [any]}
      label_rules: []
      explanation: "Tests if a column's value is NULL."
    is_not_null:
      aliases: [is not null, is not empty, has a value, exists, is present]
      template: "{column} IS NOT NULL"
      applicable_types: {'column': [any]}
      label_rules: []
      explanation: "Tests if a column's value is not NULL."
  filler_words:
    _skip:
      aliases:
        - me
        - the
        - all
        - for
        - are
        - a
        - an
        - some
        - any
        - their
        - with
        - that
        - which
        - who
  global_templates:
    select_template: "SELECT {columns} FROM {table} {constraints}"

# SQL functions and templates
sql_actions:
  count:
    aliases: [count, number, number of, how many, quantity of, total number of]
    template: "COUNT({column})"
    applicable_types: {'column': [any]}
    label_rules: ['not id']
    explanation: "Returns the number of rows that match a specified criterion."
  sum:
    aliases: [sum, total, sum of, total of, total amount of, aggregate]
    template: "SUM({column})"
    applicable_types: {'column': [numeric]}
    label_rules: ['not id']
    explanation: "Returns the sum of all values in a numeric column."
  avg:
    aliases: [average, average of, avg, mean]
    template: "AVG({column})"
    applicable_types: {'column': [numeric]}
    label_rules: ['not id']
    explanation: "Returns the average of all values in a numeric column."
  min:
    aliases: [minimum, least, min, smallest, lowest, bottom]
    template: "MIN({column})"
    applicable_types: {'column': [numeric, text, date, timestamp]}
    label_rules: []
```

```
    explanation: "Returns the minimum value of a column."
  max:
    aliases: [maximum, most, max, largest, highest, greatest]
    template: "MAX({column})"
    applicable_types: {'column': [numeric, text, date, timestamp]}
    label_rules: []
    explanation: "Returns the maximum value of a column."
  distinct:
    aliases: [distinct, unique, unique values of]
    template: "DISTINCT {column}"
    applicable_types: {'column': [any]}
    label_rules: ['not id']
    explanation: "Returns only the unique values in a column."
  order_by_asc:
    aliases: [order by ascending, sort by ascending, sort by, order by, in ascending order]
    template: "ORDER BY {column} ASC"
    applicable_types: {'column': [any]}
    label_rules: []
    explanation: "Sorts the results in ascending order based on a column."
  order_by_desc:
    aliases: [order by descending, sort by descending, in descending order]
    template: "ORDER BY {column} DESC"
    applicable_types: {'column': [any]}
    label_rules: []
    explanation: "Sorts the results in descending order based on a column."
  group_by:
    aliases: [group by, grouped by]
    template: "GROUP BY {column}"
    applicable_types: {'column': [any]}
    label_rules: []
    explanation: "Groups rows that have the same values into summary rows."
  having:
    aliases: [having, with]
    template: "HAVING {condition}"
    applicable_types: {'condition': [any]}
    label_rules: []
    explanation: "Filters groups based on a condition."
  limit:
    aliases: [limit, top, first, only]
    template: "LIMIT {value}"
    applicable_types: {'value': [numeric]}
    label_rules: []
    explanation: "Restricts the number of rows returned by the query."
  extract:
    aliases: [extract, get the]
    template: "EXTRACT({part} FROM {column})"
    pattern: ["{part}", "from", "{column}"]
    applicable_types: {'part': [any], 'column': [date, timestamp]}
    label_rules: []
    explanation: "Extracts a specific part (e.g., year, month) from a date or time value."
  length:
    aliases: [length in, length of, character count, string length]
```

```
    template: "LENGTH({column})"
    applicable_types: {'column': [text]}
    label_rules: []
    explanation: "Returns the length of a string."
  concat:
    aliases: [concat, concatenate, join, combine]
    template: "CONCAT({column1}, {column2})"
    pattern: ["{column1}", "and", "{column2}"]
    applicable_types: {'column1': [text], 'column2': [text]}
    label_rules: []
    explanation: "Joins two or more strings together."
  cast:
    aliases: [cast, convert, change type]
    template: "CAST({column} AS {to_type})"
    pattern: ["{column}", "to", "{to_type}"]
    applicable_types: {'column': [any], 'to_type': [any]}
    label_rules: []
    explanation: "Converts a value from one data type to another."
postgis_actions:
  st_perimeter:
    aliases: [perimeter, boundary length, outline length, length of boundary]
    template: "ST_Perimeter({geom})"
    applicable_types: {'geom': [geometry_polygon, geography_polygon]}
    label_rules: ['postgis']
    explanation: "Returns the perimeter (boundary length) of a polygonal geometry."
  st_distance:
    aliases: [distance, how far, distance between, separation of]
    template: "ST_Distance({geom1}, {geom2})"
    pattern: ["of", "{geom1}", "from", "{geom2}"]
    applicable_types: {'geom1': [geometry_point, geography_point, geometry_linestring, geography_linestring, geometry_polygon, geography_polygon], 'geom2': [geometry_point, geography_point, geometry_linestring, geography_linestring, geometry_polygon, geography_polygon]}
    label_rules: ['postgis']
    explanation: "Calculates the shortest distance between two geometries or geographies."
  st_intersects:
    aliases: [intersects, overlaps with]
    template: "ST_Intersects({geom1}, {geom2})"
    pattern: ["{geom1}", "with", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'geom2': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "Tests if two geometries or geographies intersect."
  st_area:
    aliases: [area, area of, size of, surface area]
    template: "ST_Area({geom})"
    applicable_types: {'geom': [geometry_polygon, geography_polygon]}
    label_rules: ['postgis']
    explanation: "Calculates the area of a polygonal geometry."
  st_length:
    aliases: [length along, length of, distance along, distance of, path length, line length]
```

```yaml
    template: "ST_Length({geom})"
    applicable_types: {'geom': [geometry_linestring, geography_linestring]}
    label_rules: ['postgis']
    explanation: "Calculates the length of a linestring geometry."
  st_x:
    aliases: [x coordinate, longitude, lon, x pos]
    template: "ST_X({point})"
    applicable_types: {'point': [geometry_point, geography_point]}
    label_rules: ['postgis']
    explanation: "Returns the X coordinate of a point geometry."
  st_y:
    aliases: [y coordinate, latitude, lat, y pos]
    template: "ST_Y({point})"
    applicable_types: {'point': [geometry_point, geography_point]}
    label_rules: ['postgis']
    explanation: "Returns the Y coordinate of a point geometry."
  st_within:
    aliases: [within, inside, is inside, is within, contained in]
    template: "ST_Within({geom1}, {geom2})"
    pattern: ["{geom1}", "in", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'geom2': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "Tests if a geometry is entirely contained within another."
  st_contains:
    aliases: [contains, encloses, surrounds]
    template: "ST_Contains({geom1}, {geom2})"
    pattern: ["{geom1}", "and", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_polygon], 'geom2': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "Tests if a geometry contains another geometry entirely."
  st_geometrytype:
    aliases: [geometry type, type of geometry, shape type, what kind of shape]
    template: "ST_GeometryType({geom})"
    applicable_types: {'geom': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "Returns the type of a geometry as a string."
  st_buffer:
    aliases: [buffer, buffer around, area around, expand by]
    template: "ST_Buffer({geom}, {radius})"
    pattern: ["{geom}", "by", "{radius}"]
    applicable_types: {'geom': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'radius': [numeric]}
    label_rules: ['postgis']
    explanation: "Creates a polygon that surrounds a geometry at a specified distance."
  st_union:
    aliases: [union, combine, merge, union of]
    template: "ST_Union({geom_collection})"
    applicable_types: {'geom_collection': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
```

```
    explanation: "Merges multiple geometries into a single geometry."
  st_centroid:
    aliases: [centroid, center, center point, geometric center]
    template: "ST_Centroid({geom})"
    applicable_types: {'geom': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "Returns the geometric center of a geometry."
  st_simplify:
    aliases: [simplify, simplify shape, generalize]
    template: "ST_Simplify({geom}, {tolerance})"
    pattern: ["{geom}", "by", "{tolerance}"]
    applicable_types: {'geom': [geometry_linestring, geometry_polygon, geography_linestring, geography_polygon], 'tolerance': [numeric]}
    label_rules: ['postgis']
    explanation: "Simplifies a geometry by reducing the number of vertices."
  st_touches:
    aliases: [touches, borders, is adjacent to]
    template: "ST_Touches({geom1}, {geom2})"
    pattern: ["{geom1}", "and", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'geom2': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "Tests if two geometries touch only at their boundaries."
  st_crosses:
    aliases: [crosses, goes across]
    template: "ST_Crosses({geom1}, {geom2})"
    pattern: ["{geom1}", "and", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_linestring, geometry_polygon], 'geom2': [geometry, geography, geometry_linestring]}
    label_rules: ['postgis']
    explanation: "Tests if two geometries cross each other."
  st_spatial_index:
    aliases: [spatial index]
    template: "{geom1} && {geom2}"
    pattern: ["{geom1}", "with", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'geom2': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "A spatial index operator that checks for intersection."
  st_distance_operator:
    aliases: [closest, nearest, closest to]
    template: "{geom1} <-> {geom2}"
    pattern: ["{geom1}", "to", "{geom2}"]
    applicable_types: {'geom1': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'geom2': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon]}
    label_rules: ['postgis']
    explanation: "A spatial operator that returns the distance between two geometries."
  st_transform:
    aliases: [transform, reproject, convert coordinate system]
    template: "ST_Transform({geom}, {srid})"
```

```
    pattern: ["{geom}", "to", "{srid}"]
    applicable_types: {'geom': [geometry, geography, geometry_point, geometry_linestring, geometry_polygon], 'srid': [numeric]}
    label_rules: ['postgis']
    explanation: "Transforms a geometry from one coordinate system to another."
```

layout.txt

```
/app
 nlq_to_sql_pipeline.py  # <-- The main orchestrator script
 config.py               # <-- All file paths and settings
|
 src/
    n2s_generators/
       graph_builder.py
       knowledge_compiler.py
    n2s_generators/
        graph_validator.py
        artifact_validator.py
        grammar_tester.py
|
 schema.yaml             # (Input)
 keywords_and_functions.yaml # (Input)
```

make_schema.py

```
import sqlite3
import yaml
import os
import inflect # NEW: Import the inflect library

DB_PATH = "/app/test.db"
OUTPUT_SCHEMA_PATH = "/app/schema.yaml"
# NEW: Path for our persistent, learning alias dictionary
ALIAS_DICT_PATH = "/app/alias_dictionary.yaml"

# Mapping from SpatiaLite geometry type codes to strings
GEOMETRY_TYPE_MAP = {
    1: 'POINT', 2: 'LINESTRING', 3: 'POLYGON', 4: 'MULTIPOINT',
    5: 'MULTILINESTRING', 6: 'MULTIPOLYGON', 7: 'GEOMETRYCOLLECTION'
}

def load_or_initialize_yaml(path):
    """Loads a YAML file if it exists, otherwise returns an empty dictionary."""
    if os.path.exists(path):
        with open(path, 'r') as f:
            return yaml.safe_load(f)
    return {}

# --- Intra-table alias collision prevention ---

def _propose_column_aliases(name: str, p_engine, alias_dict: dict) -> set:
    """
    Build a conservative alias set for a column name.
    - snake_case  'snake case'
    - last-word singular/plural variants
    - merge in learned aliases from alias_dictionary
    """
    aliases = set()
    aliases.add(name)
    aliases.add(name.replace('_', ' '))

    last_word = name.split('_')[-1]
    singular_last = p_engine.singular_noun(last_word) or last_word
    if singular_last != last_word:
        aliases.add(name.replace(last_word, singular_last).replace('_', ' '))

    plural_last = p_engine.plural(last_word)
    if plural_last != last_word:
        aliases.add(name.replace(last_word, plural_last).replace('_', ' '))

    # learned aliases (if any)
    for a in alias_dict.get(name, []):
        s = str(a).strip()
        if s:
            aliases.add(s)
```

```python
    return aliases


def _resolve_intra_table_alias_collisions(table_name: str, col_alias_map: dict) ->
tuple[dict, list]:
    """
    Given {col -> set(aliases)} within a table, remove any alias that appears
    on 2+ columns in the SAME table. Return (clean_map, warnings).
    """
    # alias -> [columns that want it]
    inv = {}
    for col, aliases in col_alias_map.items():
        for a in aliases:
            inv.setdefault(a.lower().strip(), []).append(col)

    dropped_events = []
    for alias, cols in inv.items():
        if len(cols) <= 1:
            continue  # no conflict
        # drop alias from ALL colliding columns
        for c in cols:
            if alias in {x.lower() for x in col_alias_map[c]}:
                # remove the exact-cased variant(s)
                to_remove = {x for x in col_alias_map[c] if x.lower() == alias}
                col_alias_map[c] -= to_remove
        dropped_events.append({
            "table": table_name,
            "alias": alias,
            "columns": sorted(cols),
            "action": "dropped_from_all"
        })
    return col_alias_map, dropped_events


def _emit_collision_warnings(events: list) -> None:
    for ev in events:
        print(
            f"WARNING: Intra-table alias collision in '{ev['table']}': "
            f"alias '{ev['alias']}' appeared on columns {ev['columns']}. "
            f"Action: {ev['action']}."
        )


def get_schema_from_db(db_path, alias_dict):
    """
    Queries an SQLite database and returns an enriched schema dictionary.
    Prevents intra-table column alias collisions by dropping the ambiguous alias
    from all conflicting columns within the same table (and logs a warning).
    """
    p = inflect.engine()

    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    conn.enable_load_extension(True)
```

```
    conn.execute("SELECT load_extension('mod_spatialite')")

    schema = {'tables': {}}
    geometry_info = {}

    # SpatiaLite geometry metadata (if present)
    try:
         cursor.execute("SELECT f_table_name, f_geometry_column, geometry_type, srid FROM
geometry_columns;")
        for row in cursor.fetchall():
            table_name, column_name, subtype_code, srid = row
            base_type = 'GEOGRAPHY' if srid == 4326 else 'GEOMETRY'
            subtype_str = GEOMETRY_TYPE_MAP.get(subtype_code, 'UNKNOWN')
                   geometry_info[(table_name,  column_name)]  =  {'base_type':  base_type,
'subtype': subtype_str}
    except sqlite3.Error:
        # Non-spatial DBs won't have this tabletotally fine.
        pass

    cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
    tables_to_exclude = {
        'sqlite_sequence', 'spatial_ref_sys', 'geometry_columns',
        'vector_layers', 'virts_geometry_columns', 'spatialite_history',
        'spatial_ref_sys_aux', 'views_geometry_columns', 'geometry_columns_statistics',
        'views_geometry_columns_statistics', 'virts_geometry_columns_statistics',
        'geometry_columns_field_infos', 'views_geometry_columns_field_infos',
        'virts_geometry_columns_field_infos', 'geometry_columns_time',
        'geometry_columns_auth', 'views_geometry_columns_auth',
        'virts_geometry_columns_auth', 'data_licenses', 'sql_statements_log',
        'SpatialIndex', 'ElementaryGeometries', 'KNN'
    }
    tables = [row[0] for row in cursor.fetchall() if row[0] not in tables_to_exclude]

    for table_name in tables:
        # --- Table aliases (dedup via set) ---
        t_aliases = set()
        singular = p.singular_noun(table_name) or table_name
        plural = p.plural(singular)
        t_aliases.update({singular, plural})
        t_aliases.update(alias_dict.get(table_name, []))

        # We'll populate columns after collision resolution
        schema['tables'][table_name] = {'aliases': sorted(t_aliases), 'columns': {}}
        alias_dict[table_name] = sorted(t_aliases)

        # --- Collect raw column info first (so we can resolve collisions) ---
        cursor.execute(f"PRAGMA table_info({table_name});")
        columns = cursor.fetchall()

        # Stage 1: propose aliases per column
        col_alias_proposals = {}
        col_labels = {}
        col_types = {}
```

```
        for column in columns:
            cid, name, ctype, notnull, dflt_value, pk = column

            # Type mapping / spatial detection
            mapped_type = (ctype or "").upper()
            labels = set()
            geom_data = geometry_info.get((table_name, name))
            if geom_data:
                                                                            mapped_type     =
f"{geom_data['base_type'].lower()}_{geom_data['subtype'].lower()}"
                labels.add('postgis')

            # Rule-based labels
            name_lower = name.lower()
            if 'id' in name_lower: labels.add('id')
            if 'latitude' in name_lower: labels.add('latitude')
            if 'longitude' in name_lower: labels.add('longitude')

            # Propose aliases (before collision resolution)
            proposed = _propose_column_aliases(name, p, alias_dict)

            col_alias_proposals[name] = proposed
            col_labels[name] = labels
            col_types[name] = mapped_type

        # Stage 2: drop ambiguous aliases within the table
        col_alias_proposals, dropped_events = _resolve_intra_table_alias_collisions(
            table_name, {k: set(v) for k, v in col_alias_proposals.items()}
        )
        if dropped_events:
            _emit_collision_warnings(dropped_events)

        # Stage 3: finalize column entries and update alias_dict
        for name in col_types.keys():
            aliases_final = sorted(list(col_alias_proposals[name]))
            schema['tables'][table_name]['columns'][name] = {
                'aliases': aliases_final,
                'type': col_types[name],
                'labels': sorted(list(col_labels[name])),
            }
            # persist only the finalized aliases (no dropped ones)
            alias_dict[name] = aliases_final

    conn.close()
    return schema, alias_dict


def main():
    # --- UPDATED: Load the persistent alias dictionary ---
    alias_dictionary = load_or_initialize_yaml(ALIAS_DICT_PATH)
    print(f"Loaded {len(alias_dictionary)} canonical names from alias dictionary.")

    print(f"Querying database '{DB_PATH}' for schema...")
    schema_data, updated_alias_dict = get_schema_from_db(DB_PATH, alias_dictionary)
```

```
    if schema_data:
        os.makedirs(os.path.dirname(OUTPUT_SCHEMA_PATH), exist_ok=True)
        with open(OUTPUT_SCHEMA_PATH, 'w') as f:
            yaml.dump(schema_data, f, sort_keys=False)
        print(f"Schema successfully written to '{OUTPUT_SCHEMA_PATH}'.")

        # --- UPDATED: Save the enriched alias dictionary back to disk ---
        with open(ALIAS_DICT_PATH, 'w') as f:
            yaml.dump(updated_alias_dict, f, sort_keys=False)
         print(f"Alias dictionary updated and saved to '{ALIAS_DICT_PATH}'. Now contains
{len(updated_alias_dict)} names.")

    else:
        print("Schema generation failed.")

if __name__ == '__main__':
    # You might want to run generate_db.py first to ensure test.db exists
    # import generate_db
    # generate_db.create_db()
    main()
```

nl_sql_maker_complete.md

nl_sql_maker

This is a pipeline for taking natural language and making an sql query from it. This is done via:
1. Create a generated database (placeholder for real database)
2. Gather the db schema with type and alias information - this uses the database and an alias_dictionary in order to generate a schema.yaml file
3. Run the nlq_to_sql_pipeline.py as a pipeline,
    a. Creates a relationship_graph.yaml - this uses the schema and keywords to create a relationship graph which is organizing the functions and keywords with the schema information.
    b. Creates a canonical_grammar.lark - the grammar is made from the relationship graph in order to examine the structure for all the query fucntions
    c. Creates a normalization_map.yaml - this is the vocabulary from the relationship graph, this is used to normalize natural language and bring it into a smaller set that can be processed by the grammar
    d. Various validators ensure that the pipeline generated correct and useful objects

Pipeline Details
```
### ## The Onboarding and QA Pipeline

This pipeline is the standard operating procedure for integrating a new database. It's a script (e.g., `run_validation.py`) that executes a series of **Generators** and **Validation Checks** in a specific order. The "flight recorder" debug logs are essential, providing a detailed audit trail if any check fails.

---

### ## Phase 1: Graph Generation and Quality Assurance

**Goal:** To create a unified `relationship_graph.yaml` and validate its integrity.

#### ### G1: Graph Generator

*   **Responsibility:** Your existing `graph_builder.py` script. It takes the new database's `schema.yaml` and the universal `keywords_and_functions.yaml` to produce a single, comprehensive `graph.yaml`.

#### ### V1: Graph QA Checks

* **Responsibility:** A new `test_graph.py` script that validates the output of G1.
* **V1.1: Content Completeness Check:**
        *   **Purpose:** To guarantee that no data was lost or corrupted during graph creation.
      * **Process:** The validator loads the two source YAMLs and the output graph. It programmatically iterates through every table, column, function, and keyword from the sources and asserts that a corresponding, correctly formed node exists in the graph. The flight recorder logs which entities it's checking.
* **V1.2: Logical Cohesion Check:**
    * **Purpose:** To perform a sanity check on the relationships within the graph.
```

    * **Process:** The validator iterates through every column in the graph and checks it against every function. Using your `is_compatible` logic, it asserts that **every column has at least one compatible function**. This prevents "orphaned" columns and confirms the type/label system is coherent.

Phase 2: Knowledge Compilation and Full-System QA

Goal: To compile the verified graph into run-time artifacts and run a comprehensive suite of checks to ensure the entire system works for the new database.

G2: Knowledge Compiler

* **Responsibility:** Your `knowledge_compiler.py` script. It takes the validated `graph.yaml` and produces the two key run-time artifacts: `normalization_map.yaml` and `canonical_grammar.lark`.

V2: Artifact and Component QA Checks

These checks validate the outputs of G2 and the behavior of the individual run-time components.

* **V2.1: Map Coverage Check:** Verifies that every single alias defined anywhere in the `graph.yaml` exists as a key in the `normalization_map.yaml`.
* **V2.2: Grammar Vocabulary Check:** Verifies that the `canonical_grammar.lark` contains a terminal definition for every canonical term found in the graph.
* **V2.3: Canonical Phrase "Smoke Test":** A simple, deterministic check of the core grammar rules. It programmatically builds perfect, canonical phrases (e.g., `"select age from users"`) by picking valid canonical tokens from the graph and asserts the grammar can parse them.
* **V2.4: Grammar Stress Test:** This is the `SmartGenerator`. It uses the graph and the canonical grammar to generate hundreds of complex, valid canonical phrases to ensure the grammar has no structural flaws or ambiguities.
* **V2.5: Normalizer Spot-Check:** A unit-level check that takes a sample of important aliases from the `normalization_map` (e.g., `"user name"`, `"avg"`, `"the"`) and asserts that the `Normalizer` class correctly maps them to their canonical form (`"username"`, `"avg"`, `None`).
* **V2.6: Normalizer-Parser Integration Check:** This is a crucial test. It takes valid canonical phrases from the Grammar Stress Test (V2.4), "de-normalizes" them by replacing canonical terms with random aliases, and then asserts that the `Normalizer -> Parser` pipeline can successfully process them back into a valid parse tree.

V3: End-to-End System QA

This is the final certification stage, testing the entire system against realistic inputs.

* **V3.1: "Golden Set" Validation:** Tests the full `Normalizer -> Parser -> Transformer` pipeline against a static, curated list of human-written or LLM-generated natural language queries. This ensures the system can handle real-world phrasing.
* **V3.2: Live Database Execution Check:** The ultimate test. It takes the "golden set" queries, generates the final SQL, and executes it against the actual database (`test.db`). The check passes if the SQL executes without raising a database error. This

confirms that the generated SQL is not just syntactically correct, but also semantically valid for the target database schema.
```

## Project Layout

The scripts for now are layed out like this:
 ./alias_dictionary.yaml
 ./config.py
 ./Containerfile
 ./keywords_and_functions.yaml
 ./make_schema.py
 ./nlq_to_sql_pipeline.py
 ./podman-compose.yaml
 ./pyproject.toml
 ./requirements.txt
 ./src
 ./src/n2s_generators
 ./src/n2s_generators/graph_builder.py
 ./src/n2s_generators/knowledge_compiler.py
 ./src/n2s_runtime
 ./src/n2s_runtime/normalizer.py
 ./src/n2s_validators
 ./src/n2s_validators/artifact_validator.py
 ./src/n2s_validators/grammar_validator.py
 ./src/n2s_validators/graph_validator.py
 ./src/n2s_validators/normalizer_validator.py
 ./src/n2s_validators/source_validator.py
 ./src/scripts
 ./src/scripts/generate_db.py

## Generated sqllite database

For testing purposes an sqllite database is created so that the schema may be used for grammar and volcabulary creation.  Additionally this database will be used for testing the pipeline once the Domain Specific Language is created.

```
#/src/scripts/generate_db.py
import sqlite3
import os
import random
from datetime import datetime, timedelta
from shapely.geometry import Point, LineString, Polygon
from shapely.wkb import dumps, loads

DB_NAME = 'test.db'
SCHEMA_FILE = os.path.join(os.path.dirname(__file__),
'../natural_language_sql/schema/schema.yaml')

# Helper function to create geometry objects and convert to WKB
def create_wkb(geom_type, coords):
    if geom_type == 'POINT':
        geom = Point(coords)

```
    elif geom_type == 'LINESTRING':
        geom = LineString(coords)
    elif geom_type == 'POLYGON':
        geom = Polygon(coords)
    else:
        return None
    return dumps(geom, hex=True)

def generate_dummy_data(cursor):
    # This SQL now uses SpatiaLite's `AddGeometryColumn` function
    # instead of trying to define the column type in the CREATE TABLE statement.
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS users (
            user_id INTEGER PRIMARY KEY,
            username VARCHAR(50) NOT NULL,
            age INT,
            balance DECIMAL(10, 2),
            is_active BOOLEAN,
            last_login TIMESTAMP
        );
    """)
    # Add the GEOMETRY column using SpatiaLite's function
    cursor.execute("SELECT AddGeometryColumn('users', 'location', 4326, 'POINT', 2);")

    users_data = []
    for i in range(1, 11):
        username = f"user_{i}"
        age = random.randint(20, 60)
        balance = random.uniform(100.0, 1000.0)
        is_active = random.choice([1, 0]) # SQLite stores BOOLEAN as 0 or 1
        last_login = datetime.now() - timedelta(days=random.randint(1, 365))

        # Insert data without the geometry column first
          cursor.execute("INSERT INTO users (user_id, username, age, balance, is_active, last_login) VALUES (?, ?, ?, ?, ?, ?)", (i, username, age, balance, is_active, last_login))

        # Then, update the geometry column with the SpatiaLite-formatted WKB
        location = Point(random.uniform(-180, 180), random.uniform(-90, 90))
            cursor.execute("UPDATE users SET location = ST_GeomFromText(?, 4326) WHERE user_id = ?", (location.wkt, i))

    # Table 2: Sales (same as before)
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS sales (
            sale_id INTEGER PRIMARY KEY,
            user_id INTEGER,
            product_name TEXT,
            sale_date DATE,
            quantity INT,
            price FLOAT,
            FOREIGN KEY(user_id) REFERENCES users(user_id)
        );
    """)
```

```
    sales_data = []
    for i in range(1, 21):
        user_id = random.randint(1, 10)
        product_name = random.choice(['Laptop', 'Mouse', 'Keyboard', 'Monitor'])
        sale_datetime_obj = datetime.now() - timedelta(days=random.randint(1, 180))
        sale_date = sale_datetime_obj.strftime('%Y-%m-%d')
        quantity = random.randint(1, 5)
        price = random.uniform(50.0, 1500.0)
        sales_data.append((i, user_id, product_name, sale_date, quantity, price))

    cursor.executemany("INSERT INTO sales VALUES (?, ?, ?, ?, ?, ?)", sales_data)

    # Table 3: Regions (with polygon geometry)
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS regions (
            region_id INTEGER PRIMARY KEY,
            name VARCHAR(50)
        );
    """)
    # Add the GEOMETRY column
      cursor.execute("SELECT AddGeometryColumn('regions', 'boundaries', 4326, 'POLYGON',
2);")

    regions_data = [
        (1, 'North', Polygon(((0, 0), (0, 45), (90, 45), (90, 0), (0, 0)))),
        (2, 'South', Polygon(((-90, -90), (-90, 0), (0, 0), (0, -90), (-90, -90)))),
    ]

    for region_id, name, polygon in regions_data:
            cursor.execute("INSERT INTO regions (region_id, name) VALUES (?, ?)",
(region_id, name))
          cursor.execute("UPDATE regions SET boundaries = ST_GeomFromText(?, 4326) WHERE
region_id = ?", (polygon.wkt, region_id))

    print("Dummy data generated and inserted successfully.")

def create_db():
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)

    conn = sqlite3.connect(DB_NAME)

    # Enable SpatiaLite extension loading
    conn.enable_load_extension(True)
    try:
        # Load the SpatiaLite extension
        conn.execute("SELECT load_extension('mod_spatialite')")
        print("SpatiaLite extension loaded successfully.")
    except sqlite3.OperationalError as e:
        print(f"Error loading SpatiaLite extension: {e}")
        print("Please ensure 'mod_spatialite' is correctly installed and in the system's
path.")
        conn.close()
```

```
        return

    cursor = conn.cursor()
    # SpatiaLite requires its metadata tables to be initialized
    cursor.execute("SELECT InitSpatialMetaData(1)")

    generate_dummy_data(cursor)
    conn.commit()
    conn.close()
    print(f"Database '{DB_NAME}' created.")

if __name__ == '__main__':
    create_db()
```

Creating a Schema File

By querying the database and applying a dictionary we should have a solid starting point for what will go into our grammar and vocabulary. The user may want to apply addtional labels to a column in order for it to be processed in a slightly different way then the generated process would automatically do.

```
# make_schema.py
import sqlite3
import yaml
import os
import inflect # NEW: Import the inflect library

DB_PATH = "/app/test.db"
OUTPUT_SCHEMA_PATH = "/app/schema.yaml"
# NEW: Path for our persistent, learning alias dictionary
ALIAS_DICT_PATH = "/app/alias_dictionary.yaml"

# Mapping from SpatiaLite geometry type codes to strings
GEOMETRY_TYPE_MAP = {
    1: 'POINT', 2: 'LINESTRING', 3: 'POLYGON', 4: 'MULTIPOINT',
    5: 'MULTILINESTRING', 6: 'MULTIPOLYGON', 7: 'GEOMETRYCOLLECTION'
}

def load_or_initialize_yaml(path):
    """Loads a YAML file if it exists, otherwise returns an empty dictionary."""
    if os.path.exists(path):
        with open(path, 'r') as f:
            return yaml.safe_load(f)
    return {}

def get_schema_from_db(db_path, alias_dict):
    """
    Queries an SQLite database and returns an enriched schema dictionary.
    """
    p = inflect.engine() # NEW: Initialize the inflect engine

    conn = sqlite3.connect(db_path)
```

```
    cursor = conn.cursor()
    conn.enable_load_extension(True)
    conn.execute("SELECT load_extension('mod_spatialite')")

    schema = {'tables': {}}
    geometry_info = {}
      cursor.execute("SELECT f_table_name, f_geometry_column, geometry_type, srid FROM
geometry_columns;")
    for row in cursor.fetchall():
        table_name, column_name, subtype_code, srid = row
        base_type = 'GEOGRAPHY' if srid == 4326 else 'GEOMETRY'
        subtype_str = GEOMETRY_TYPE_MAP.get(subtype_code, 'UNKNOWN')
          geometry_info[(table_name, column_name)] = {'base_type': base_type, 'subtype':
subtype_str}

    cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
    # UPDATED: A more comprehensive list of system tables to exclude
    tables_to_exclude = {
        'sqlite_sequence', 'spatial_ref_sys', 'geometry_columns',
        'vector_layers', 'virts_geometry_columns', 'spatialite_history',
        'spatial_ref_sys_aux', 'views_geometry_columns', 'geometry_columns_statistics',
        'views_geometry_columns_statistics', 'virts_geometry_columns_statistics',
        'geometry_columns_field_infos', 'views_geometry_columns_field_infos',
        'virts_geometry_columns_field_infos', 'geometry_columns_time',
        'geometry_columns_auth', 'views_geometry_columns_auth',
        'virts_geometry_columns_auth', 'data_licenses', 'sql_statements_log',
        'SpatialIndex', 'ElementaryGeometries', 'KNN'
    }
    tables = [row[0] for row in cursor.fetchall() if row[0] not in tables_to_exclude]

    for table_name in tables:
        # --- UPDATED: Alias Generation for Tables ---
        # Start with a set for automatic de-duplication
        aliases = set()
        # Add singular and plural forms
        singular = p.singular_noun(table_name) or table_name
        plural = p.plural(singular)
        aliases.add(singular)
        aliases.add(plural)
        # Add aliases from our persistent dictionary
        aliases.update(alias_dict.get(table_name, []))

        schema['tables'][table_name] = {
            'aliases': sorted(list(aliases)),
            'columns': {}
        }
        # Update the master dictionary with any new aliases found
        alias_dict[table_name] = sorted(list(aliases))

        cursor.execute(f"PRAGMA table_info({table_name});")
        columns = cursor.fetchall()

        for column in columns:
            cid, name, ctype, notnull, dflt_value, pk = column
```

```
            mapped_type = ctype.upper()
            # Use a set for labels to prevent duplicates
            labels = set()

            geom_data = geometry_info.get((table_name, name), None)
            if geom_data:
                                                                            mapped_type     =
f"{geom_data['base_type'].lower()}_{geom_data['subtype'].lower()}"
                # UPDATED: Add "postgis" to labels, not metadata
                labels.add('postgis')

            # --- UPDATED: Advanced, Rule-Based Labeling ---
            name_lower = name.lower()
            if 'id' in name_lower: labels.add('id')
            if 'latitude' in name_lower: labels.add('latitude')
            if 'longitude' in name_lower: labels.add('longitude')

            # --- UPDATED: Alias Generation for Columns ---
            col_aliases = set()
            col_aliases.add(name)
            col_aliases.add(name.replace('_', ' '))
            # Add singular/plural forms for the last word of the column name
            last_word = name.split('_')[-1]
            singular_last = p.singular_noun(last_word) or last_word
            if singular_last != last_word:
                   col_aliases.add(name.replace(last_word, singular_last).replace('_', '
'))
            plural_last = p.plural(last_word)
            if plural_last != last_word:
                 col_aliases.add(name.replace(last_word, plural_last).replace('_', ' '))

              # Add aliases from our persistent dictionary and the original schema.yaml
(if any)
            col_aliases.update(alias_dict.get(name, []))

            schema['tables'][table_name]['columns'][name] = {
                'aliases': sorted(list(col_aliases)),
                'type': mapped_type,
                'labels': sorted(list(labels)) # 'metadata' field is removed
            }
            # Update the master dictionary with any new aliases
            alias_dict[name] = sorted(list(col_aliases))

    conn.close()
    return schema, alias_dict

def main():
    # --- UPDATED: Load the persistent alias dictionary ---
    alias_dictionary = load_or_initialize_yaml(ALIAS_DICT_PATH)
    print(f"Loaded {len(alias_dictionary)} canonical names from alias dictionary.")

    print(f"Querying database '{DB_PATH}' for schema...")
    schema_data, updated_alias_dict = get_schema_from_db(DB_PATH, alias_dictionary)
```

```
    if schema_data:
        os.makedirs(os.path.dirname(OUTPUT_SCHEMA_PATH), exist_ok=True)
        with open(OUTPUT_SCHEMA_PATH, 'w') as f:
            yaml.dump(schema_data, f, sort_keys=False)
        print(f"Schema successfully written to '{OUTPUT_SCHEMA_PATH}'.")

        # --- UPDATED: Save the enriched alias dictionary back to disk ---
        with open(ALIAS_DICT_PATH, 'w') as f:
            yaml.dump(updated_alias_dict, f, sort_keys=False)
        print(f"Alias dictionary updated and saved to '{ALIAS_DICT_PATH}'. Now contains {len(updated_alias_dict)} names.")

    else:
        print("Schema generation failed.")

if __name__ == '__main__':
    # You might want to run generate_db.py first to ensure test.db exists
    # import generate_db
    # generate_db.create_db()
    main()
```

Normalizer

The normalizer is used in our runtime applications which will use all the artifacts created by the pipeline, but the normalizer can also be used in order to verify these artifacts as part of the validation within the pipeline.

```
# src/n2s_runtime/normalizer.py

from __future__ import annotations
import re
from dataclasses import dataclass, field
from typing import Callable, Dict, Iterable, List, Sequence, Tuple

Pair = Tuple[str, bool]
Phrase = List[Pair]
NDMap = Dict[str, List[str]]

# ---------- flight recorder ----------

@dataclass
class FlightRecorder:
    events: List[Tuple[str, Dict[str, object]]] = field(default_factory=list)
    def log(self, evt: str, **data: object) -> None: self.events.append((evt, data))
    def warn(self, evt: str, **data: object) -> None: self.events.append((f"WARNING:{evt}", data))
    def fail(self, evt: str, **data: object) -> None: self.events.append((f"FAIL:{evt}", data))
    def dump(self, print_fn: Callable[[str], None] = print) -> None:
        for e, d in self.events: print_fn(f"{e}: {d}")
```

```
# ---------- tokenization ----------

TOKENIZER_RE = re.compile(r"\|\||&&|<=|>=|!=|==|<>|[A-Za-z0-9_']+|[^\sA-Za-z0-9_]")

def tokenize(s: str) -> List[str]:
    return TOKENIZER_RE.findall(s)

def squash_spaces(s: str) -> str:
    return " ".join(s.split())

# ---------- map shaping ----------

def _coerce_listy(v: object) -> List[str]:
    if isinstance(v, list): return [("" if o is None else str(o)) for o in v]
    return ["" if v is None else str(v)]

SENTINEL_BLACKLIST = {"", "skip", "_skip"}

def _collect_canonicals(det: Dict[str, object], nd: Dict[str, object]) -> List[str]:
    vals: List[str] = []
    for v in det.values():
        if isinstance(v, str) and v not in SENTINEL_BLACKLIST:
            vals.append(v)
    for v in nd.values():
        if isinstance(v, list):
            for o in v:
                if isinstance(o, str) and o not in SENTINEL_BLACKLIST:
                    vals.append(o)
        elif isinstance(v, str) and v not in SENTINEL_BLACKLIST:
            vals.append(v)
    return vals

def leftmost_unmapped_index(phrase: Phrase) -> int | None:
    for i, (_, m) in enumerate(phrase):
        if not m: return i
    return None

def nd_matches_at(tokens: List[str], nd: NDMap, start: int, max_len: int, joiner: str =
" ") -> List[Tuple[str, int]]:
    out: List[Tuple[str, int]] = []
    run_len = len(tokens) - start
    for span_len in range(1, min(max_len, run_len) + 1):
        span = joiner.join(tokens[start:start+span_len])
        if span in nd:
            out.append((span, len(nd[span])))
    return out

def    inspect_leftmost(norm_map:    Dict[str,    Dict[str,    object]],    text:    str,
punctuation_as_mapped: Iterable[str] = (",",), joiner: str = " ") -> None:
    nd = build_nd_map(norm_map)
    toks = tokenize(text)
    seed = punctuation_passthrough(toks, punctuation_as_mapped)
    i = leftmost_unmapped_index(seed)
```

```python
    max_len = max(1, max(len(k.split(" ")) for k in nd.keys()))
    print(f"\n[inspect] '{text}'")
    print(f"  tokens: {toks}")
    print(f"  seed  : {seed}")
    if i is None:
        print("  fully mapped at seed")
        return
    spans = nd_matches_at(toks, nd, i, max_len, joiner=joiner)
    print(f"  leftmost unmapped idx={i} token='{toks[i]}'")
    if spans:
        print(f"  nd keys here: {spans}")
    else:
        print("  nd keys here: NONE  <-- coverage gap at leftmost")

def _probe_case(norm_map: Dict[str, Dict[str, object]], s: str, warn_every: int = 10) -> 
None:
    fr = FlightRecorder()
    outs = normalize_text(norm_map, s, fr=fr, warn_every=warn_every)
    print(f"\nINPUT: {s}\nOUTPUTS ({len(outs)}): {outs}")
    if not outs:
        print("---- flight recorder (only on fail) ----")
        fr.dump()
        inspect_leftmost(norm_map, s)

def build_nd_map(norm_map: Dict[str, Dict[str, object]]) -> NDMap:
    det = norm_map.get("deterministic_aliases", {}) or {}
    nd  = norm_map.get("non_deterministic_aliases", {}) or {}

    out: NDMap = {}
    for k, v in nd.items():
        out[str(k)] = _coerce_listy(v)

    for k, v in det.items():
        canon = "" if (v is None or v == "skip") else str(v)
        out.setdefault(str(k), [])
        if canon not in out[str(k)]:
            out[str(k)].append(canon)

    # identity for canonical outputs so canonical tokens are always mappable
    for c in _collect_canonicals(det, nd):
        out.setdefault(c, [])
        if c not in out[c]:
            out[c].append(c)

    return out

def  punctuation_passthrough(tokens:  Sequence[str],  passthrough:  Iterable[str])  -> 
Phrase:
    pt = set(passthrough)
    return [(t, t in pt) for t in tokens]

# ---------- safe permutations (optional) ----------

def expand_unmapped_permutations(
```

```
    seq: Phrase,
    joiner: str = " ",
    cap: int = 200,
    warn_every: int = 50,
    fr: FlightRecorder | None = None,
) -> List[Phrase]:
    res: List[Phrase] = []
    n = len(seq)
    i = 0
    while i < n and seq[i][1]: i += 1
    if i == n:
        return [seq[:]]
    j = i
    run: List[str] = []
    while j < n and not seq[j][1]:
        run.append(seq[j][0])
        j += 1

    def seg(tokens: List[str]) -> List[List[List[str]]]:
        out: List[List[List[str]]] = []
        def rec(k: int, cur: List[List[str]]) -> None:
            if k == len(tokens):
                out.append(cur[:]); return
            for r in range(k + 1, len(tokens) + 1):
                cur.append(tokens[k:r]); rec(r, cur); cur.pop()
        rec(0, [])
        return out

    for parts in seg(run):
        merged = [(joiner.join(p), False) for p in parts]
        cand = seq[:i] + merged + seq[j:]
        res.append(cand)
        if fr and len(res) % warn_every == 0:
            fr.warn("perm_count", count=len(res))
        if len(res) > cap:
            if fr: fr.fail("perm_cap_exceeded", cap=cap, count=len(res))
            break
    return res

# ---------- leftmost BFS (on-demand segmentation) ----------

def _max_key_len_words(nd: NDMap) -> int:
    m = 1
    for k in nd.keys():
        L = max(1, len(k.split(" ")))
        if L > m: m = L
    return m

def _serialize(ph: Phrase) -> Tuple[Tuple[str, bool], ...]:
    return tuple(ph)

def bfs_resolve_leftmost_spans(
    initial: Phrase,
    ndict: NDMap,
```

```
    *,
    joiner: str = " ",
    cap_nodes: int = 200,
    cap_results: int = 200,
    warn_every: int = 50,
    fr: FlightRecorder | None = None,
) -> List[str]:
    from collections import deque
    q = deque([initial])
    seen = {_serialize(initial)}
    finals: List[str] = []
    max_len = _max_key_len_words(ndict)
    node_expanded = 0

    while q:
        phrase = q.popleft()

        try:
            i = next(idx for idx, (_, m) in enumerate(phrase) if not m)
        except StopIteration:
            s = joiner.join(t for t, _ in phrase)
            finals.append(s)
            if fr and len(finals) % warn_every == 0:
                fr.warn("final_count", count=len(finals))
            if len(finals) > cap_results:
                if fr: fr.fail("final_cap_exceeded", cap=cap_results, count=len(finals))
                break
            continue

        r = i
        while r < len(phrase) and not phrase[r][1]: r += 1
        run_len = r - i
        tried = False

        for span_len in range(1, min(max_len, run_len) + 1):
            span_text = joiner.join(phrase[k][0] for k in range(i, i + span_len))
            options = ndict.get(span_text)
            if not options: continue
            tried = True
            for opt in options:
                new_phrase = phrase[:i] + [(opt, True)] + phrase[i + span_len:]
                key = _serialize(new_phrase)
                if key in seen: continue
                seen.add(key)
                q.append(new_phrase)
                node_expanded += 1
                if fr and node_expanded % warn_every == 0:
                    fr.warn("node_count", count=node_expanded)
                if node_expanded > cap_nodes:
                                 if  fr:  fr.fail("node_cap_exceeded",  cap=cap_nodes,
count=node_expanded)
                    return finals

        if not tried:
```

```
            if fr: fr.log("prune", leftmost=phrase[i][0], run_len=run_len)

    return finals

# ---------- normalizer ----------

def normalize_text(
    normalization_map: Dict[str, Dict[str, object]],
    text: str,
    *,
    tokenizer: Callable[[str], List[str]] = tokenize,
    joiner: str = " ",
    case_insensitive: bool = False,
    punctuation_as_mapped: Iterable[str] = (",",),
    cap_nodes: int = 200,
    cap_results: int = 200,
    warn_every: int = 50,
    fr: FlightRecorder | None = None,
) -> List[str]:
    nd = build_nd_map(normalization_map)
    s = text.casefold() if case_insensitive else text
    toks = tokenizer(s)
    if fr: fr.log("tokens", tokens=toks)
    init = punctuation_passthrough(toks, punctuation_as_mapped)
    if fr: fr.log("seed", phrase=init)
    finals_raw = bfs_resolve_leftmost_spans(
        initial=init,
        ndict=nd,
        joiner=joiner,
        cap_nodes=cap_nodes,
        cap_results=cap_results,
        warn_every=warn_every,
        fr=fr,
    )
    seen, outs = set(), []
    for f in finals_raw:
        clean = squash_spaces(f)
        if clean not in seen:
            seen.add(clean)
            outs.append(clean)
    if fr: fr.log("finals", count=len(outs))
    return outs
# ---------- demo main ----------

def _demo_norm_map() -> Dict[str, Dict[str, object]]:
    return {
        "deterministic_aliases": {
            ",": ",",
            "&&": "and",
            "||": "or",
            "and": "and",
            "or": "or",
            "of": "of",
            "from": "from",
```

```
            "select": "select",

            # tables
              "users": "users", "user": "users", "people": "users", "clients": "users",
"customers": "users",
            "sales": "sales", "sale": "sales", "orders": "sales", "order": "sales",
                   "purchases":  "sales",  "purchase":  "sales",  "transactions":  "sales",
"transaction": "sales",
                 "regions": "regions", "region": "regions", "zone": "regions", "zones":
"regions", "territory": "regions",

            # fields
            "user_id": "user_id", "username": "username",
            "age": "age", "price": "price", "quantity": "quantity",
            "sale_date": "sale_date", "product_name": "product_name",
            "region_id": "region_id", "boundaries": "boundaries",
            "is_active": "is_active", "last_login": "last_login",

            # ops
                     "order_by_asc":  "order_by_asc",  "order_by_desc":  "order_by_desc",
"group_by": "group_by",
             "st_buffer": "st_buffer", "st_length": "st_length", "st_x": "st_x", "st_y":
"st_y",
            "st_intersects": "st_intersects", "st_geometrytype": "st_geometrytype",
            "st_distance_operator": "st_distance_operator",
            "st_crosses": "st_crosses",
            "distinct": "distinct",
        },

        "non_deterministic_aliases": {
            # requests
            "tell me": ["select", ""],
            "give me": ["select", ""],

            # ids / names / dates
            "user id": ["user_id"],
            "user name": ["username"],
            "user names": ["username"],
            "order id": ["sale_id"],
            "order number": ["sale_id"],
            "transaction id": ["sale_id"],
            "region id": ["region_id"],
            "order date": ["sale_date"],
            "sale date": ["sale_date"],
            "sale dates": ["sale_date"],                 # <-- added
            "when it was sold": ["sale_date"],

            # quantities / amounts
            "number of items": ["quantity"],
            "number sold": ["quantity"],
            "quantity of": ["quantity"],
            "how much": ["price"],
            "product name": ["product_name"],
```

```
            # composed ops
            "length of": ["st_length"],
            "unique values of": ["distinct of"],
            "separation of": ["st_distance of"],

            # ordering / grouping
            "order by ascending": ["order_by_asc"],
            "order by descending": ["order_by_desc"],
            "sort by ascending": ["order_by_asc"],
            "sort by descending": ["order_by_desc"],
            "group by": ["group_by"],

            # geometry / spatial
            "y coordinate": ["st_y"],
            "x coordinate": ["st_x"],
            "x pos": ["st_x"],
            "overlaps with": ["st_intersects"],
            "closest to": ["st_distance_operator"],
            "buffer around": ["st_buffer"],
            "geometry type": ["st_geometrytype"],
            "goes across": ["st_crosses"],
            "inside": ["st_within"],                     # <-- added
            "inside of": ["st_within of"],               # <-- optional helper

            # domain ambiguity
            "area": ["regions", "st_area"],

            # status/time phrases
            "last signed in": ["last_login"],
            "last login": ["last_login"],
        },
    }


def _run_case(norm_map: Dict[str, Dict[str, object]], s: str) -> None:
    fr = FlightRecorder()
    outs = normalize_text(norm_map, s, fr=fr)
    ok = bool(outs)
       print(f"\nINPUT:  {s}\nOUTPUTS  ({len(outs)}):  {outs}\nSTATUS:  {'OK'  if  ok  else
'FAIL'}")
    if not ok:
        print("---- flight recorder ----")
        fr.dump()

def main() -> None:
    nm = _demo_norm_map()  # or your real map

    print("### happy path sanity ###")
    for t in [
        "||",
        "tell me user id",
        "region id , user id",
        "when it was sold",
        "user_id,username",
```

```
        "give me user id",
        "length of of last signed in",
        "unique values of of price",
        "separation of of regions",
        "inside of users",
        "inside , price from users",
        "order by descending",
        "order by descending of users",
        "group by of users",
        "length of last signed in",
        "length of of last signed in",
        "sale dates",
        "sale dates , group by",
        "overlaps with of transaction id",
        "closest to of sale date",
        "goes across of region",
        "x pos of users",
        "unique values of price",
        "unique values of of price",
        "separation of regions",
        "separation of of regions",
        "number of items of users",
        "quantity of from region",
    ]:
        _run_case(nm, t)

    print("\n### targeted micro-probes ###")
    # 1) inside/within + comma + from

    """
             print("\n###################### EXPECTED FAILURE CASES (for now) ##########################")
    for t in [

    ]:
        _run_case(nm, t)
    """

if __name__ == "__main__":
    main()

```

nl_to_sql_pipeline

The nl_to_sql_pipeline uses generators and validators in order to create and vet artifacts that will be used in the application.

```
# File: nlq_to_sql_pipeline.py

import yaml
import sys
from collections import defaultdict
```

```python
# Import functions from our new modules
import config
from lark import Lark
from src.n2s_validators.source_validator import validate_schema_yaml, validate_keywords_yaml
from src.n2s_generators.graph_builder import generate_relationship_graph
from src.n2s_validators.graph_validator import validate_content_completeness, validate_relationships
from src.n2s_generators.knowledge_compiler import build_normalization_map, build_canonical_grammar
from src.n2s_validators.artifact_validator import validate_map_coverage, validate_grammar_vocabulary
from src.n2s_validators.grammar_validator import validate_canonical_phrases, validate_with_smart_generator
from src.n2s_validators.normalizer_validator import validate_normalizer_spot_check, validate_normalizer_integration
#from src.n2s_runtime.normalizer import normalize_text

def load_yaml(path):
    try:
        with open(path, 'r') as f: return yaml.safe_load(f)
    except FileNotFoundError: return None

def write_yaml(data, path):
    with open(path, 'w') as f:
        yaml.dump(data, f, sort_keys=False)

def run_validation_check(validator_func, *args):
    """A helper to run a validator and report its status."""
    log = []
    is_success, final_log = validator_func(*args, log)
    if is_success:
        print(f"  {final_log[0]} -> PASS")
    else:
        print(f"  {final_log[0]} -> FAIL")
        print("\n--- Failure Log ---")
        for msg in final_log:
            print(msg)
        print("-------------------\n")
        # In a real pipeline, you might want to exit on failure
        # sys.exit(1)
    return is_success

def main():
    """The main orchestration pipeline."""
    print("--- Starting NLQ-to-SQL Onboarding & QA Pipeline ---")

    # --- Load Source Files ---
    print("\nLoading source YAML files...")
    schema_data = load_yaml(config.SCHEMA_PATH)
    keywords_data = load_yaml(config.KEYWORDS_PATH)
    if not (schema_data and keywords_data):
        print("Error: Source YAML files not found. Exiting.")
```

```
        return

    # --- NEW: Phase 0: Source File Validation ---
    print("\n[V0] Running Source File Validation...")
    if not run_validation_check(validate_schema_yaml, schema_data):
        return # Halt pipeline on failure
    if not run_validation_check(validate_keywords_yaml, keywords_data):
        return # Halt pipeline on failure

    # --- Phase 1: Graph Generation ---
    print("\n[G1] Generating Relationship Graph...")
    schema_data = load_yaml(config.SCHEMA_PATH)
    keywords_data = load_yaml(config.KEYWORDS_PATH)
    if not (schema_data and keywords_data):
        print("Error: Source YAML files not found. Exiting.")
        return

    relationship_graph = generate_relationship_graph(schema_data, keywords_data)
    write_yaml(relationship_graph, config.GRAPH_PATH)
    print(f"  -> Generated {config.GRAPH_PATH}")

    # --- Phase 2: Graph Validation ---
    print("\n[V1] Running Graph Quality Checks...")
        run_validation_check(validate_content_completeness,  schema_data,  keywords_data,
relationship_graph)
    run_validation_check(validate_relationships, relationship_graph)

    # --- Phase 3: Knowledge Compilation ---
    print("\n[G2] Compiling Knowledge Artifacts...")
    norm_map = build_normalization_map(relationship_graph)
    write_yaml(norm_map, config.NORMALIZATION_MAP_PATH)
    print(f"  -> Generated {config.NORMALIZATION_MAP_PATH}")

    grammar = build_canonical_grammar(relationship_graph)
    with open(config.GRAMMAR_PATH, 'w') as f: f.write(grammar)
    print(f"  -> Generated {config.GRAMMAR_PATH}")

    # --- WIP: Phase 4: Artifact Validation ---
    print("\n[V2] Running Artifact Quality Checks...")
    norm_map = load_yaml(config.NORMALIZATION_MAP_PATH)
    try:
        with open(config.GRAMMAR_PATH, 'r') as f:
            grammar_text = f.read()
        parser = Lark(grammar_text, start='query')
    except FileNotFoundError:
        print(f"  FAIL: Grammar file {config.GRAMMAR_PATH} not found.")
        sys.exit(1)

    run_validation_check(validate_map_coverage, relationship_graph, norm_map)
    run_validation_check(validate_grammar_vocabulary, relationship_graph, grammar_text)
    run_validation_check(validate_canonical_phrases, relationship_graph, grammar_text)
    run_validation_check(validate_with_smart_generator, relationship_graph, parser)

    # --- NEW: V2.5 and V2.6 checks ---
```

```
    run_validation_check(validate_normalizer_spot_check, norm_map)
    run_validation_check(validate_normalizer_integration, relationship_graph, parser, norm_map)

    # --- Phase 4 & 5: Artifact/System Validation & E2E Tests ---
    print("\n[V3] Running further validation checks...")
    print("  -> (Placeholder) All subsequent validation steps would be called here.")

    print("\n--- Pipeline Finished ---")

if __name__ == '__main__':
    main()
```

Generators

The generators need to do a few things -
1. Create our relationships which combines the functions, keywords and schema information. This is a middle product for creating a grammar and vocabulary for our project.
2. Create our grammar and normalization_map (vocabulary) - the vocabulary is for catching a wide range of natural language aliases and bringing them into our canonical token space so that the grammar may process it.

```
#src/n2s_generators/graph_builder.py
from collections import defaultdict

def generate_relationship_graph(schema_yaml, keywords_yaml):
    """
    Builds a clean graph with a consistent data structure, containing only
    true queryable entities from the source YAMLs.
    """
    graph = defaultdict(lambda: {'entity_type': 'unknown', 'metadata': {}})

    # Process Schema (tables and columns)
    if schema_yaml and 'tables' in schema_yaml:
        for table_name, table_data in schema_yaml['tables'].items():
            graph[table_name]['entity_type'] = 'table'
            graph[table_name]['metadata'] = table_data
            for column_name, column_data in table_data['columns'].items():
                graph[column_name]['entity_type'] = 'column'
                graph[column_name]['metadata'] = column_data

    # Process keywords.yaml by top-level section
    if keywords_yaml:
        # Process the nested 'keywords' section
        if 'keywords' in keywords_yaml and isinstance(keywords_yaml['keywords'], dict):
            for entity_type, entity_body in keywords_yaml['keywords'].items():
                if isinstance(entity_body, dict):
                    for canonical_name, metadata in entity_body.items():
                        graph[canonical_name]['entity_type'] = entity_type
                        graph[canonical_name]['metadata'] = metadata
```

```
        # Process 'sql_actions' and 'postgis_actions'
        for action_type in ['sql_actions', 'postgis_actions']:
              if action_type in keywords_yaml and isinstance(keywords_yaml[action_type],
dict):
                for canonical_name, metadata in keywords_yaml[action_type].items():
                    graph[canonical_name]['entity_type'] = action_type
                    graph[canonical_name]['metadata'] = metadata

    return dict(graph)

```
```
#src/n2s_generator/knowledge_compiler.py
from collections import defaultdict
import re
from typing import Dict, List, Any, Iterable, Tuple

def _extract_entities(graph):
        """Traverses the graph and organizes all entities by type into a clean
dictionary."""
    entities = defaultdict(dict)
    for key, node in graph.items():
        entity_type = node.get('entity_type')
        if entity_type:
            entities[entity_type][key] = node
    return entities

# -------------------------
# Config (same semantics)
# -------------------------
PREP_BARE = {"of", "from", "in", "on", "at"}
FILLER_ALIAS_DENY = {"is", "by", "with", "for", "are"}
GENERIC_DENY = {"order by", "sort by", "by", "ascending", "descending"}
OF_CANONICALS = {"distinct", "avg", "sum", "st_distance"}
DOMAIN_PREFER_SPATIAL = {
    "contains": ("like", "st_contains"),
    "intersects": ("st_spatial_index", "st_intersects"),
    "overlaps": ("st_spatial_index", "st_intersects"),
}
PLURAL_LASTWORD = {
    "date": "dates",
    "login": "logins",
    "id": "ids",
    "username": "usernames",
    "name": "names",
    "item": "items",
    "value": "values",
}
SAFE_PLURAL_LASTWORD = {"date", "login", "id", "username", "name", "item", "value"}
ALLOWED_TYPES_FOR_PLURAL = {"table", "column"}
EXTRA_ALIASES = {
    # columns / scalar fields
    ("location", "column"): ["place"],
    ("price",    "column"): ["prices"],   # plural safety
```

```python
    # SQL string length phrasing that shows up
    ("length", "sql_actions"): ["length in"],   # normalize to canonical 'length'

    # PostGIS path-length phrasing your generator emits
    ("st_length", "postgis_actions"): ["length along"],  # normalize to st_length
}

# -------------------------
# Utilities
# -------------------------
def _add(master: Dict[str, List[dict]], key: str, entry: dict) -> None:
    k = key.lower().strip()
    if not k:
        return
    if entry not in master[k]:
        master[k].append(entry)


def _dedupe_strict(seq: Iterable[str]) -> List[str]:
    seen, out = set(), []
    for s in seq:
        if s not in seen:
            seen.add(s)
            out.append(s)
    return out


# -------------------------
# Pass 1: collect aliases
# -------------------------
def _collect_aliases(graph: Dict[str, dict],
                     diagnostics: Dict[str, Any]) -> Dict[str, List[dict]]:
    master: Dict[str, List[dict]] = defaultdict(list)
    allowed_types = {
        "table", "column", "sql_actions", "postgis_actions",
        "select_verbs", "prepositions", "logical_operators",
        "comparison_operators", "filler_words"
    }
    def _normalize_alias_text(s: str) -> str:
        # normalize curly apostrophes to ASCII and trim
        return str(s).replace("", "'").strip()

    for canonical_name, node in graph.items():
        etype = node.get("entity_type")
        if etype not in allowed_types:
            continue

        entry = {"canonical": canonical_name, "type": etype}
        _add(master, canonical_name, entry)

        # Inject curated extras for this canonical/type, if any
        for extra in EXTRA_ALIASES.get((canonical_name, etype), []):
            _add(master, extra, entry)
```

```python
        md = node.get("metadata", {})
        aliases = md.get("aliases", []) if isinstance(md, dict) else []
        for alias in aliases:
            a = _normalize_alias_text(alias)
            if not a:
                continue

            al = a.lower()
            # keep these words available for operators/templates
            if etype == "filler_words" and al in FILLER_ALIAS_DENY:
                continue

            # drop generic ambiguous forms early
            if al in GENERIC_DENY:
                diagnostics["generic_denied"].append(al)
                continue

            _add(master, a, entry)

    return master


# -------------------------
# Synthesis: plural keys
# -------------------------
def _synthesize_plurals(master: Dict[str, List[dict]],
                        diagnostics: Dict[str, Any]) -> None:
    keys_snapshot = list(master.keys())
    for k in keys_snapshot:
        parts = k.split()
        if not parts:
            continue

        types_here = {m["type"] for m in master[k]}
        if not types_here or not types_here.issubset(ALLOWED_TYPES_FOR_PLURAL):
            continue  # only pluralize table/column aliases

        lw = parts[-1]
        if lw in PLURAL_LASTWORD and lw in SAFE_PLURAL_LASTWORD:
            plural_key = " ".join(parts[:-1] + [PLURAL_LASTWORD[lw]])
            if plural_key not in master:
                master[plural_key] = list(master[k])
                diagnostics["plural_added"].append({"from": k, "to": plural_key})


# -------------------------
# Prefix map for longest-key protection
# -------------------------
def _compute_prefix_to_longers(master: Dict[str, List[dict]]) -> Dict[str, List[str]]:
    multiword_keys = [k for k in master.keys() if " " in k]
    prefix_to_longers: Dict[str, List[str]] = defaultdict(list)
    for mw in multiword_keys:
        pfx = mw.split()[0]
```

```python
        prefix_to_longers[pfx].append(mw)
    return prefix_to_longers


# ------------------------
# Cleanups on master alias map
# ------------------------
def _apply_preposition_purity(alias: str,
                              meanings: List[dict],
                              diagnostics: Dict[str, Any]) -> List[dict]:
    if alias not in PREP_BARE:
        return meanings
    before = len(meanings)
    kept = [m for m in meanings if m["type"] == "prepositions"]
    if len(kept) < before:
        diagnostics["preposition_conflicts"].append(
            {"alias": alias, "dropped": before - len(kept)}
        )
    return kept


def _apply_domain_separation(alias: str,
                             meanings: List[dict],
                             diagnostics: Dict[str, Any]) -> List[dict]:
    if alias not in DOMAIN_PREFER_SPATIAL:
        return meanings
    lose, prefer = DOMAIN_PREFER_SPATIAL[alias]
    cset = {m["canonical"] for m in meanings}
    if lose in cset and prefer in cset:
        kept = [m for m in meanings if m["canonical"] != lose]
        diagnostics["domain_conflicts"].append(
            {"alias": alias, "dropped": lose, "kept": prefer}
        )
        return kept
    return meanings


def _apply_prefix_protection(alias: str,
                             meanings: List[dict],
                             prefix_to_longers: Dict[str, List[str]],
                             diagnostics: Dict[str, Any],
                             enable: bool = True) -> List[dict]:
    if not enable:
        return meanings
    if " " in alias:
        return meanings
    if alias not in prefix_to_longers:
        return meanings

    longers = prefix_to_longers[alias]
    before = len(meanings)
    kept = [m for m in meanings if m["type"] != "table"]
    if len(kept) < before:
        diagnostics["prefix_collisions"].append(
```

```
                            {"alias": alias, "longer_keys": list(longers), "action":
"dropped_table_meaning"}
        )
    return kept


def _clean_master(master: Dict[str, List[dict]],
                  prefix_to_longers: Dict[str, List[str]],
                  diagnostics: Dict[str, Any],
                  enable_prefix_protection: bool = True) -> Dict[str, List[dict]]:
    cleaned: Dict[str, List[dict]] = {}
    for alias, meanings in master.items():
        kept = list(meanings)
        kept = _apply_preposition_purity(alias, kept, diagnostics)
        kept = _apply_domain_separation(alias, kept, diagnostics)
        kept = _apply_prefix_protection(alias, kept, prefix_to_longers, diagnostics,
                                        enable=enable_prefix_protection)
        if kept:
            cleaned[alias] = kept
    return cleaned


# -------------------------
# Pass 2: partition
# -------------------------
def _partition(cleaned_master: Dict[str, List[dict]],
               diagnostics: Dict[str, Any]) -> Dict[str, dict]:
    out = {"deterministic_aliases": {}, "non_deterministic_aliases": {}}

    for alias, meanings in cleaned_master.items():
        is_multi_word = " " in alias
        is_ambiguous = len(meanings) > 1

        # filler_words  deterministic ""
        if meanings and all(m["type"] == "filler_words" for m in meanings):
            out["deterministic_aliases"][alias] = ""
            continue

        if is_multi_word or is_ambiguous:
            targets = [m["canonical"] for m in meanings]

            # of surface policy for selected operators when alias ends with " of"
            if alias.lower().endswith(" of"):
                adjusted = []
                for t in targets:
                    if t in OF_CANONICALS and not t.endswith(" of"):
                        adjusted.append(f"{t} of")
                        diagnostics["of_policy_adjusted"].append(
                            {"alias": alias, "canonical": t, "to": f"{t} of"}
                        )
                    else:
                        adjusted.append(t)
                targets = adjusted
```

```python
            out["non_deterministic_aliases"][alias] = _dedupe_strict(sorted(targets))
        else:
            out["deterministic_aliases"][alias] = meanings[0]["canonical"]

    return out


# -------------------------
# Public API
# -------------------------
def build_normalization_map(graph: Dict[str, dict],
                            *,
                            enable_prefix_protection: bool = True) -> Dict[str, dict]:
    diagnostics: Dict[str, Any] = {
        "prefix_collisions": [],
        "preposition_conflicts": [],
        "domain_conflicts": [],
        "echo_removed": [],
        "generic_denied": [],
        "plural_added": [],
        "of_policy_adjusted": [],
    }

    master = _collect_aliases(graph, diagnostics)
    _synthesize_plurals(master, diagnostics)
    prefix_to_longers = _compute_prefix_to_longers(master)
    cleaned = _clean_master(master, prefix_to_longers, diagnostics,
                            enable_prefix_protection=enable_prefix_protection)
    norm_map = _partition(cleaned, diagnostics)
    norm_map["_diagnostics"] = diagnostics
    return norm_map



def _build_keyword_terminals(entities):
    """
    Builds the grammar strings for all high-priority and general keyword Terminals,
    and returns a map of keywords to their new Terminal names.
    """
    keyword_terminals = ['// --- High-Priority Keyword Terminals ---']
    keyword_map = {}

                        all_keywords      =      {**entities.get('prepositions',      {}),
**entities.get('logical_operators', {})}

    for keyword, data in sorted(all_keywords.items()):
        terminal_name = keyword.upper().replace(' ', '_')
        keyword_map[keyword] = terminal_name
        keyword_terminals.append(f'{terminal_name}: "{keyword}"')

    select_verbs = entities.get('select_verbs', {}).keys()
    sorted_verbs = [f'"{w}"' for w in sorted(select_verbs)]
    keyword_terminals.append(f'SELECT: {" | ".join(sorted_verbs)}')
    keyword_map['select'] = 'SELECT'
```

```
    keyword_terminals.append('COMMA: ","')
    keyword_map[','] = 'COMMA'

    return keyword_terminals, keyword_map

# --- Main Grammar Builder ---
def build_canonical_grammar(graph):
    """Builds the final, truly canonical grammar."""
    entities = _extract_entities(graph)

    # 1. Generate all keyword terminals (SELECT, AND, COMMA, etc.)
    keyword_terminal_lines, _ = _build_keyword_terminals(entities)

    # 2. Generate all canonical entity terminals (populated with your vocabulary)
    canonical_tables = sorted(entities.get('table', {}).keys())
    canonical_columns = sorted(entities.get('column', {}).keys())
         canonical_functions  =  sorted(list(entities.get('sql_actions',  {}).keys())  +
list(entities.get('postgis_actions', {}).keys()))

    entity_terminal_lines = ['\n// --- CANONICAL ENTITY TERMINALS ---']
    if canonical_tables:
        table_literals = [f'"{name}"' for name in canonical_tables]
        entity_terminal_lines.append(f'CANONICAL_TABLE: {" | ".join(table_literals)}')
    if canonical_columns:
        column_literals = [f'"{name}"' for name in canonical_columns]
        entity_terminal_lines.append(f'CANONICAL_COLUMN: {" | ".join(column_literals)}')
    if canonical_functions:
        function_literals = [f'"{name}"' for name in canonical_functions]
                             entity_terminal_lines.append(f'CANONICAL_FUNCTION:    {"    |
".join(function_literals)}')

    # 3. Assemble the main parser rules using the canonical terminals
    main_rules = [
        '\n// --- MAIN PARSER RULES ---',
        'selectable: column_name | function_call',
        'select_statement: SELECT column_list from_clause',
        'column_name: CANONICAL_COLUMN',
        'table_name: CANONICAL_TABLE',
        # --- THE NEW, SIMPLE, AND CORRECT FUNCTION RULE ---
        'function_call: CANONICAL_FUNCTION (OF column_list)?',
        # --- The robust list rule ---
        'column_list: selectable (COMMA selectable)* (COMMA? AND selectable)?',
        'from_clause: (FROM | OF) table_name',
        '\n%import common.WS',
        '%ignore WS',
    ]

    # 4. Combine all parts into the final grammar
    #    (Note: The dynamic function rules/terminals are no longer needed)
    final_grammar_parts = [
        'start: query', 'query: select_statement',
        *keyword_terminal_lines,
        *entity_terminal_lines,
```

```
        *main_rules
    ]

    return "\n".join(final_grammar_parts)

```

Validators

The validators check for correctness along the pipeline so that issues are caught early on and can be fixed at the appropriate level.

0.1 and 0.2
This checks that the source fits the expected structure
```
# src/n2s_validators/source_validator.py
def validate_schema_yaml(schema_data, log):
    """V0.1: Performs a deep validation of the schema.yaml structure."""
    log.append("--- Running V0.1: Schema.yaml Structural Check ---")

    if not isinstance(schema_data, dict) or 'tables' not in schema_data:
        log.append("FAIL: schema.yaml must be a dictionary with a top-level 'tables' key.")
        return False, log

    if not isinstance(schema_data['tables'], dict):
        log.append("FAIL: The 'tables' key must contain a dictionary of tables.")
        return False, log

    for table_name, table_data in schema_data['tables'].items():
        if not isinstance(table_data, dict) or 'columns' not in table_data:
            log.append(f"FAIL: Entry for table '{table_name}' must be a dictionary with a 'columns' key.")
            return False, log

        if not isinstance(table_data['columns'], dict):
            log.append(f"FAIL: The 'columns' key in table '{table_name}' must contain a dictionary.")
            return False, log

        for col_name, col_data in table_data['columns'].items():
            if not isinstance(col_data, dict):
                log.append(f"FAIL: Entry for column '{col_name}' in table '{table_name}' must be a dictionary.")
                return False, log
            if 'type' not in col_data or not isinstance(col_data['type'], str):
                log.append(f"FAIL: Column '{col_name}' in table '{table_name}' is missing a 'type' string.")
                return False, log
            if 'aliases' not in col_data or not isinstance(col_data['aliases'], list):
                log.append(f"FAIL: Column '{col_name}' in table '{table_name}' is missing an 'aliases' list.")
                return False, log
```

```
    log.append("PASS: schema.yaml has the expected structure.")
    return True, log


def validate_keywords_yaml(keywords_data, log):
    """V0.2: Validates the basic structure of keywords_and_functions.yaml."""
    log.append("--- Running V0.2: Keywords.yaml Structural Check ---")

    if not isinstance(keywords_data, dict):
        log.append("FAIL: keywords.yaml should be a dictionary.")
        return False, log

    for required_section in ['keywords', 'sql_actions', 'postgis_actions']:
        if required_section not in keywords_data:
            log.append(f"FAIL: keywords.yaml is missing required top-level key
'{required_section}'.")
            return False, log

    # Check the structure within the 'keywords' section
    for keyword_type, keyword_data in keywords_data['keywords'].items():
        # --- FIX IS HERE: Add an exception for 'global_templates' ---
        # This section has a unique structure and should be skipped by this check.
        if keyword_type == 'global_templates':
            continue
        # --- END OF FIX ---

        if not isinstance(keyword_data, dict):
            log.append(f"FAIL: Section '{keyword_type}' in 'keywords' must be a
dictionary (e.g., canonical_name: {{'aliases': [...]}}).")
            return False, log

        for canonical_name, metadata in keyword_data.items():
            if not isinstance(metadata, dict) or 'aliases' not in metadata:
                log.append(f"FAIL: Entry '{canonical_name}' in '{keyword_type}' must be
a dictionary with an 'aliases' key.")
                return False, log
            if not isinstance(metadata['aliases'], list):
                log.append(f"FAIL: The 'aliases' key for '{canonical_name}' in
'{keyword_type}' must contain a list.")
                return False, log

    # (The validation for sql_actions and postgis_actions remains the same)
    for action_type in ['sql_actions', 'postgis_actions']:
        for func_name, func_data in keywords_data[action_type].items():
            if not isinstance(func_data, dict):
                log.append(f"FAIL: Entry for function '{func_name}' in '{action_type}'
must be a dictionary.")
                return False, log
            if 'aliases' not in func_data or not isinstance(func_data['aliases'], list):
                log.append(f"FAIL: Function '{func_name}' in '{action_type}' is missing
an 'aliases' list.")
                return False, log
            if 'template' not in func_data or not isinstance(func_data['template'],
str):
```

```
                log.append(f"FAIL: Function '{func_name}' in '{action_type}' is missing a 'template' string.")
                return False, log

    log.append("PASS: keywords_and_functions.yaml has the expected structure.")
    return True, log

```

1.1 and 1.2

These validators check that everything in the original sheme and functions are all present in the graph file.

```
# src/n2s_validators/graph_validator.py

from collections import defaultdict

def is_compatible(column_info, action_info):
    """
    Checks if a given action/function can be applied to a given column based
    on type and label rules.
    """
    compatible_vars = {}
    if not (isinstance(column_info, dict) and isinstance(action_info, dict)):
        return {}

    applicable_types_dict = action_info.get('applicable_types')
    if not isinstance(applicable_types_dict, dict):
        return {}

    for var, allowed_types in applicable_types_dict.items():
        column_type = column_info.get('type')
        column_labels = column_info.get('labels', [])

        if column_type and ('any' in allowed_types or column_type in allowed_types):
            valid_labels = True
            for rule in action_info.get('label_rules', []):
                if rule.startswith('not '):
                    label_to_exclude = rule[4:]
                    if label_to_exclude in column_labels:
                        valid_labels = False; break
                else:
                    if rule not in column_labels:
                        valid_labels = False; break
            if valid_labels:
                compatible_vars[var] = True
    return compatible_vars

def validate_content_completeness(schema_yaml, keywords_yaml, graph, log):
    """
    V1.1: Checks if all source data from schema and keywords YAMLs
    is present in the generated graph by validating the canonical names.
    """
```

```
    log.append("--- Running V1.1: Content Completeness Check ---")
    missing_items = []

    # 1. Check entities from schema.yaml
    for table_name, table_data in schema_yaml.get('tables', {}).items():
        if table_name not in graph:
            missing_items.append(f"Table '{table_name}'")
        for column_name in table_data.get('columns', {}).keys():
            if column_name not in graph:
                missing_items.append(f"Column '{column_name}'")

    # 2. Check entities from keywords_and_functions.yaml
        for  section_key,  section_data  in  keywords_yaml.items():  #  e.g.,  'keywords',
'sql_actions'
        if isinstance(section_data, dict):
            # This handles top-level sections like sql_actions, postgis_actions
            if section_key != 'keywords':
                for canonical_name in section_data.keys():
                    if canonical_name not in graph:
                          missing_items.append(f"Entity '{canonical_name}' from section
'{section_key}'")
            else:
                # This handles the nested 'keywords' section
                       for  keyword_type,  keyword_body  in  section_data.items():  #  e.g.,
'select_verbs'
                    if isinstance(keyword_body, dict):
                        for canonical_name in keyword_body.keys():
                            if canonical_name not in graph:
                                   missing_items.append(f"Entity '{canonical_name}' from
section '{keyword_type}'")

    if not missing_items:
        log.append("PASS: All source entities are present in the graph.")
        return True, log
    else:
         log.append("FAIL: The following entities from source YAMLs are missing from the
graph:")
        for item in sorted(missing_items):
            log.append(f"  - {item}")
        return False, log

def validate_relationships(graph, log):
    """
    V1.2: Checks for logical relationships, ensuring every column
    can be used by at least one function.
    """
    log.append("--- Running V1.2: Relationship Cohesion Check ---")

    columns = {k: v for k, v in graph.items() if v.get('entity_type') == 'column'}
        functions  =  {k:  v  for  k,  v  in  graph.items()  if  v.get('entity_type')  in
['sql_action', 'postgis_action']}

    if not columns or not functions:
        log.append("WARNING: No columns or functions found in the graph to validate.")
```

```
        return True, log

    all_columns_ok = True
    for col_name, col_data in columns.items():
        is_compatible_with_any_function = False
        for func_name, func_data in functions.items():
            if is_compatible(col_data.get('metadata', {}), func_data.get('metadata',
{})):
                is_compatible_with_any_function = True
                break

        if not is_compatible_with_any_function:
            log.append(f"FAIL: Column '{col_name}' is not compatible with ANY defined
function.")
            all_columns_ok = False

    if all_columns_ok:
        log.append("PASS: All columns are usable by at least one function.")

    return all_columns_ok, log

```
2.1 and 2.2

These validators look for expected vocabulary and expected canonical terms. Effectively just looking for a coverage of terms

```
# src/n2s_validators/artifact_validator.py

import re

def validate_map_coverage(graph, norm_map, log):
    """
    V2.1: Checks that every ALIAS defined in the graph exists as a key in
    either the deterministic or non-deterministic sections of the normalization map.
    """
    log.append("--- Running V2.1: Normalization Map Coverage Check ---")

    # Combine all known aliases from BOTH sections of the map for a single lookup
    d_keys = norm_map.get('deterministic_aliases', {}).keys()
    nd_keys = norm_map.get('non_deterministic_aliases', {}).keys()
    all_map_aliases = set(d_keys) | set(nd_keys)
    missing_aliases = []

    # Iterate through the graph and check ONLY the defined aliases for each entity
    for canonical_name, node in graph.items():
        metadata = node.get('metadata', {})
        if isinstance(metadata, dict):
            for alias in metadata.get('aliases', []):
                # Check if the alias from the graph is a key in the normalization map
                if str(alias).lower() not in all_map_aliases:
                    missing_aliases.append(f"Alias '{alias}' for '{canonical_name}'")
```

```
    if not missing_aliases:
        log.append("PASS: Normalization map covers all defined aliases.")
        return True, log
    else:
        log.append("FAIL: The following aliases from the graph are missing from the normalization map:")
        for item in sorted(missing_aliases):
            log.append(f"  - {item}")
        return False, log

def _check_terminal_vocabulary(terminal_name, canonical_names, grammar_text, log):
    """Helper function to verify all canonical names are in a grammar terminal."""
    pattern = re.compile(rf"^{terminal_name}:\s*(.*)", re.MULTILINE)
    match = pattern.search(grammar_text)

    if not match:
        log.append(f"FAIL: Terminal '{terminal_name}' is not defined in the grammar.")
        return False

    defined_literals = {item.strip() for item in match.group(1).split('|')}

    all_found = True
    for name in canonical_names:
        quoted_name = f'"{name}"'
        if quoted_name not in defined_literals:
            log.append(f"FAIL: Canonical name '{name}' is missing from the '{terminal_name}' terminal in the grammar.")
            all_found = False

    return all_found

def validate_grammar_vocabulary(graph, grammar_text, log):
    """
    V2.2: Verifies that the canonical_grammar.lark contains a terminal
    definition for every canonical term found in the graph.
    """
    log.append("--- Running V2.2: Grammar Vocabulary Check ---")

    # 1. Extract all canonical names from the graph by entity type
    tables = {k for k, v in graph.items() if v.get('entity_type') == 'table'}
    columns = {k for k, v in graph.items() if v.get('entity_type') == 'column'}
    functions = {k for k, v in graph.items() if v.get('entity_type') in ['sql_action', 'postgis_action']}

    # 2. Run checks for each canonical terminal type
    tables_ok = _check_terminal_vocabulary('CANONICAL_TABLE', tables, grammar_text, log)
    columns_ok = _check_terminal_vocabulary('CANONICAL_COLUMN', columns, grammar_text, log)
    functions_ok = _check_terminal_vocabulary('CANONICAL_FUNCTION', functions, grammar_text, log)

    all_checks_passed = tables_ok and columns_ok and functions_ok

    if all_checks_passed:
```

```
        log.append("PASS: Grammar vocabulary correctly matches all canonical entities from the graph.")

    return all_checks_passed, log
```

2.3 and 2.4

These validators do some basic testing against the generated grammar to ensure it can be used and it contains expected terms.

```
# src/n2s_validators/grammar_validator.py

import random
from collections import defaultdict
from lark import Lark, ParseError, UnexpectedToken
from lark.grammar import NonTerminal, Terminal

def validate_canonical_phrases(graph, grammar_text, log, num_tests=20):
    """
    V2.3: A simple, deterministic "smoke test" of the core grammar rules.
    """
    log.append("--- Running V2.3: Canonical Phrase Smoke Test ---")

    try:
        parser = Lark(grammar_text, start='query')

        # --- FIX: Use the correct PLURAL entity types from the graph ---
        verbs = [k for k, v in graph.items() if v.get('entity_type') == 'select_verbs']
        tables = [k for k, v in graph.items() if v.get('entity_type') == 'table']
        columns = [k for k, v in graph.items() if v.get('entity_type') == 'column']
        functions = [k for k, v in graph.items() if v.get('entity_type') in ['sql_actions', 'postgis_actions']]

        if not all([verbs, tables, columns, functions]):
            log.append("FAIL: Graph is missing essential entities (verbs, tables, columns, or functions).")
            # Log which specific lists are empty for better debugging
            if not verbs: log.append("  - No 'select_verbs' found.")
            if not tables: log.append("  - No 'table' entities found.")
            if not columns: log.append("  - No 'column' entities found.")
            if not functions: log.append("  - No 'sql_actions' or 'postgis_actions' found.")
            return False, log

        templates = [
            "{verb} {col} from {tbl}",
            "{verb} {col1}, {col2} from {tbl}",
            "{verb} {func} of {col} from {tbl}",
            "{verb} {col1} and {func} of {col2} from {tbl}",
        ]

        for i in range(num_tests):
```

```
            template = random.choice(templates)
            phrase = template.format(
                verb=random.choice(verbs),
                col=random.choice(columns),
                col1=random.choice(columns),
                col2=random.choice(columns),
                tbl=random.choice(tables),
                func=random.choice(functions)
            )

            try:
                parser.parse(phrase)
            except (ParseError, UnexpectedToken) as e:
                log.append(f"FAIL: The grammar could not parse a perfect canonical phrase.")
                log.append(f"     Generated Phrase: {phrase}")
                log.append(f"     Error: {e}")
                return False, log

        log.append("PASS: Core grammar rules successfully parsed all canonical phrases.")
        return True, log

    except Exception as e:
        log.append(f"FAIL: An unexpected error occurred during the smoke test: {e}")
        return False, log
# --- SmartGenerator and GrammarAnalyzer (Paste the complete, final classes here) ---
class GrammarAnalyzer:
    """
    Analyzes a Lark grammar to calculate the minimum RECURSION DEPTH
    required to fully expand each rule.
    """
    def __init__(self, parser):
        self.parser = parser
        self.rule_lookup = defaultdict(list)
        for rule in self.parser.rules:
            key = rule.origin.name.value if hasattr(rule.origin.name, 'value') else rule.origin.name
            self.rule_lookup[key].append(rule)

        self.min_depths = {}
        self._calculate_min_depths()

    def _get_min_depth(self, term_name):
        # Terminals are the end of the line; they require 1 expansion step to resolve.
        if term_name.isupper() or term_name.startswith('"') or term_name in ["AND", "COMMA", "OF", "FROM"]:
            return 1
        return self.min_depths.get(term_name, float('inf'))

    def _calculate_min_depths(self):
        # Iterate until the depths stabilize
        for _ in range(len(self.rule_lookup) + 2):
            for rule_name, expansions in self.rule_lookup.items():
```

```
                    # The depth of a rule is 1 (for itself) + the minimum sum of its children's depths.
                min_expansion_depth = min(
                    sum(self._get_min_depth(t.name) for t in r.expansion) if r.expansion else 0
                    for r in expansions
                )

                total_min_depth = 1 + min_expansion_depth

                if rule_name not in self.min_depths or total_min_depth < self.min_depths[rule_name]:
                    self.min_depths[rule_name] = total_min_depth

class SmartGenerator:
    """
    The final, robust generator. It uses two-pass analysis, a conservative
    budget-aware expansion strategy, and a recursion tracker to prevent
    infinite loops, with full conditional logging.
    """
    def __init__(self, parser, graph, analyzer):
        self.parser = parser
        self.graph = graph
        self.analyzer = analyzer
        self.rule_lookup = analyzer.rule_lookup

        # A hard limit on how many times a rule can be nested within itself.
        self.RECURSION_LIMIT = 4

        self.vocab = {
            "CANONICAL_COLUMN": [k for k, v in graph.items() if v['entity_type'] == 'column'],
            "CANONICAL_TABLE": [k for k, v in graph.items() if v['entity_type'] == 'table'],
            "CANONICAL_FUNCTION": [k for k,v in graph.items() if v['entity_type'] in ['sql_actions', 'postgis_actions']],
        }

    def generate(self, start_rule="query", max_depth=25):
        """
        Public method to start generation. Always returns a (result, log) tuple.
        """
        debug_log = []
        # --- FIX: Start the expansion with an empty recursion tracker ---
        result = self._expand(start_rule, max_depth, debug_log, "", {})

        if result is None:
            return None, debug_log
        return result, None

    def _expand(self, rule_name, depth, debug_log, indent, recursion_counts):
        log_msg = f"{indent}>> Expanding '{rule_name}' with depth_budget={depth}"

        if debug_log is not None:
```

```
            debug_log.append(log_msg)

        if depth <= 0:
            return None

        # --- RECURSION TRACKER LOGIC (START) ---
        # Copy the counts for this branch and increment the count for the current rule.
        current_counts = recursion_counts.copy()
        current_counts[rule_name] = current_counts.get(rule_name, 0) + 1
        # --- END RECURSION TRACKER ---

        # Base Case: The item is a terminal
        if rule_name not in self.rule_lookup:
            value = None
            if rule_name in self.vocab: value = random.choice(self.vocab[rule_name])
            else:
                for term_def in self.parser.terminals:
                    if term_def.name == rule_name:
                        choices = term_def.pattern.value.replace('(?:', '').rstrip(')').split('|')
                        value = random.choice([c.strip('"') for c in choices])
                        break
                if value is None: value = rule_name.strip('"')
            if debug_log is not None:
                debug_log.append(f"{indent}<< Returning terminal value: '{value}'")
            return value

        # --- It's a Rule, so we expand it ---
        possible_rules = self.rule_lookup[rule_name]
        valid_choices = possible_rules

        # --- RECURSION TRACKER LOGIC (CHOICE PRUNING) ---
        # If we've hit the recursion limit for this rule, we MUST choose a non-recursive path.
        if current_counts[rule_name] > self.RECURSION_LIMIT:
            if debug_log is not None:
                debug_log.append(f"{indent}  -- RECURSION LIMIT HIT for '{rule_name}'. Forcing a base case.")
            # Filter to only "base cases" - expansions that do not call the rule again.
            base_cases = [r for r in possible_rules if rule_name not in [t.name for t in r.expansion]]
            if not base_cases:
                if debug_log is not None:
                    debug_log.append(f"{indent}  !! FAILED: No base case available to terminate recursion.")
                return None
            valid_choices = base_cases
        # --- END RECURSION TRACKER ---

        # --- CONSERVATIVE CHOICE LOGIC (from your working version) ---
        chosen_rule = None
        if depth < (self.analyzer.min_depths.get(rule_name, 1) + 5):
            path_costs = {r: 1 + sum(self.analyzer.min_depths.get(t.name, 0) for t in r.expansion) for r in valid_choices}
```

```
            min_path_cost = min(path_costs.values())
            safest_choices = [r for r, cost in path_costs.items() if cost == min_path_cost]
            chosen_rule = random.choice(safest_choices)
            if debug_log is not None:
                debug_log.append(f"{indent}   -- CONSERVATIVE MODE: Chose safest path with cost {min_path_cost}")
        else:
            chosen_rule = random.choice(valid_choices)

        if debug_log is not None:
            debug_log.append(f"{indent}   -> Chose path: {[t.name for t in chosen_rule.expansion]}")

        if not chosen_rule.expansion:
            return ""

        parts = []
        for term in chosen_rule.expansion:
            # Pass the updated counts dictionary to the recursive call
            part = self._expand(term.name, depth - 1, debug_log, indent + "   ", current_counts)
            if part is None:
                if debug_log is not None:
                    debug_log.append(f"{indent}<< FAILED: Child '{term.name}' failed to expand.")
                return None
            parts.append(part)

        result = " ".join(filter(lambda x: x != "", parts)).strip()
        if debug_log is not None:
            debug_log.append(f"{indent}<< Success for '{rule_name}': returning '{result[:50]}...'")
        return result

def validate_with_smart_generator(graph, parser, log, num_phrases=100, success_threshold=0.85):
    """
    V2.4: Stress-tests the grammar using the SmartGenerator.
    """
    log.append("--- Running V2.4: Grammar Stress Test ---")

    try:
        analyzer = GrammarAnalyzer(parser)
        generator = SmartGenerator(parser, graph, analyzer)

        if not all(generator.vocab.values()):
            log.append("FAIL: SmartGenerator vocabulary is empty. Check entity types in the graph.")
            return False, log

        success_count = 0
        failures = 0
```

```
        for _ in range(num_phrases):
            phrase, debug_log = generator.generate()

            if phrase is None:
                failures += 1
                # Optionally, you could inspect the debug_log for the failure here
                continue

            try:
                parser.parse(phrase)
                success_count += 1
            except (ParseError, UnexpectedToken):
                failures += 1

        total_generated = success_count + failures
        if total_generated == 0:
            log.append("WARNING: Generator produced no phrases to test.")
            return True, log

        success_rate = success_count / total_generated
        log.append(f"  - Generation complete. Success Rate: {success_rate:.0%}")

        if success_rate >= success_threshold:
            log.append(f"PASS: Success rate ({success_rate:.0%}) meets threshold.")
            return True, log
        else:
            log.append(f"FAIL: Success rate ({success_rate:.0%}) is below threshold.")
            return False, log

    except Exception as e:
        log.append(f"FAIL: An unexpected error occurred during the stress test: {e}")
        return False, log
```

2.5 and 2.6

These validators use the normalizer and the grammar to run phrases from the level of aliases through the process to see if they can be applied to the grammar.

```
# File: src/n2s_validators/normalizer_validator.py
import random
from collections import defaultdict
from functools import partial
from lark import ParseError, UnexpectedToken

# These components are needed for the integration test
from src.n2s_runtime.normalizer import normalize_text
from src.n2s_validators.grammar_validator import SmartGenerator, GrammarAnalyzer

def validate_normalizer_spot_check(norm_map, log, num_spot_checks=50):
    """
    V2.5: A unit-level check that validates a sample of aliases from the
    normalization_map to ensure the Normalizer class is working correctly.
```

```
    """
    log.append("--- Running V2.5: Normalizer Spot-Check ---")

    try:
        normalizer = partial(normalize_text, norm_map)
        all_aliases = list(norm_map.get('deterministic_aliases', {}).items())

        if not all_aliases:
            log.append("WARNING: No deterministic aliases found to spot-check.")
            return True, log

        sample_size = min(num_spot_checks, len(all_aliases))
        alias_sample = random.sample(all_aliases, sample_size)

        for alias, canonical in alias_sample:
            expected_output = canonical if canonical is not None else ""

            # --- FIX IS HERE ---
            # normalize() returns a list of candidates. We expect one for a deterministic check.
            candidates = normalizer(alias)
            normalized_output = candidates[0] if candidates else ""
            # --- END OF FIX ---

            if normalized_output != expected_output:
                log.append("FAIL: Normalizer failed spot-check.")
                log.append(f"     - Input Alias: '{alias}'")
                log.append(f"     - Expected Canonical: '{expected_output}'")
                log.append(f"     - Actual Output: '{normalized_output}'")
                return False, log

        log.append(f"PASS: Normalizer correctly mapped all {sample_size} spot-checked aliases.")
        return True, log

    except Exception as e:
        log.append(f"FAIL: An unexpected error occurred during the spot-check: {e}")
        return False, log



def _setup_integration_test(parser, graph, norm_map):
    analyzer = GrammarAnalyzer(parser)
    generator = SmartGenerator(parser, graph, analyzer)
    normalizer = partial(normalize_text, norm_map)

    CONNECTORS = {"of", "by", "to", "with", "and", "from"}

    reverse_alias_map = defaultdict(list)
    full_alias_map = {**norm_map.get('deterministic_aliases', {}),
                      **norm_map.get('non_deterministic_aliases', {})}

    for alias, canonical_or_list in full_alias_map.items():
        if canonical_or_list is None:
```

```
            continue
            canonicals = canonical_or_list if isinstance(canonical_or_list, list) else
[canonical_or_list]
        for can in canonicals:
            # prune pathological alias forms for testing (optional)
            a = alias.strip()
            if not a:
                continue
            if any(a.lower().startswith(c + " ") for c in CONNECTORS):
                # leading connector tends to create nonsense when substituted per-token
                continue
            reverse_alias_map[can].append(a)

    return normalizer, generator, reverse_alias_map

def _denormalize_phrase(canonical_phrase, reverse_alias_map):
    """
    Converts a canonical phrase into a messy phrase by choosing aliases,
    but avoids connector duplication (e.g., 'average of of').
    If we must use an alias that already ends with a connector and the
    next canonical token is that same connector, we consume (skip) the
    next token to prevent duplication.
    """
    CONNECTORS = {"of", "by", "to", "with", "and", "from"}
    LOCK_CANONICAL = {",", "COMMA"} | CONNECTORS  # don't alias punctuation/connectors

    def bucket_aliases(aliases):
        """
        Partition aliases into: plain (no trailing connector) and endswith[connector].
        """
        endswith = {c: [] for c in CONNECTORS}
        plain = []
        for a in aliases:
            s = a.strip()
            s_low = s.lower()
            matched = False
            for c in CONNECTORS:
                if s_low.endswith(" " + c):
                    endswith[c].append(s)
                    matched = True
                    break
            if not matched:
                plain.append(s)
        return plain, endswith

    toks = canonical_phrase.split()
    out = []
    i = 0
    while i < len(toks):
        t = toks[i]

        # never alias commas/connectors; keep them literal
        if t in LOCK_CANONICAL or t == ",":
            out.append("," if t in {",", "COMMA"} else t)
```

```
            i += 1
            continue

        # alias choices for this canonical token
        choices = reverse_alias_map.get(t, None)
        if not choices:
            # fallback: keep canonical if we have no alias choices
            out.append(t)
            i += 1
            continue

        nxt = toks[i + 1].lower() if i + 1 < len(toks) else None
        plain, endswith = bucket_aliases(choices)

        if nxt in CONNECTORS:
            if plain:
                # Prefer aliases that do NOT already include the connector
                out.append(random.choice(plain))
                i += 1
            elif endswith[nxt]:
                 # If only connector-ending aliases exist, consume the next connector to
avoid duplication
                out.append(random.choice(endswith[nxt]))
                i += 2  # skip the next connector token
            else:
                # No connector-matched alias; fall back to any plain or any ending
                pool = plain or [a for lst in endswith.values() for a in lst]
                out.append(random.choice(pool))
                i += 1
        else:
            # No connector following; any alias is fine (prefer plain for stability)
              out.append(random.choice(plain or [a for lst in endswith.values() for a in
lst]))
            i += 1

    return " ".join(out)



def _run_pipeline_on_phrase(messy_phrase, normalizer, parser, log):
    """
    Runs a single phrase through the Normalizer -> Parser pipeline,
    with enhanced logging.
    """
    # --- ADDED LOGGING ---
    log.append(f"\n    --- V2.6 Sub-Test ---")
    log.append(f"    - De-Normalized Phrase: '{messy_phrase}'")

    normalized_candidates = normalizer(messy_phrase)
     log.append(f"     - Normalizer Produced {len(normalized_candidates)} Candidate(s):
{normalized_candidates}")

    _errors = []
    successful_candidate = None
    size_nc=len(normalized_candidates)
```

```
    cnt=0
    for candidate in normalized_candidates:
        cnt=cnt+1
        try:
            if candidate:
                parser.parse(candidate)
                successful_candidate = candidate
                # We found a valid interpretation, so we can stop.
                return True, [], normalized_candidates, log
        except Exception as e:
            _errors.append(f"error {cnt}/{size_nc}:\n{e}")
            continue

    # (ParseError, UnexpectedToken)
    # If we get here, no candidate succeeded
    log.append(f"    - Result: All candidates failed to parse.")
    return False, _errors, normalized_candidates, log



# --- The Main V2.6 Validator ---

def validate_normalizer_integration(graph, parser, norm_map, log, num_phrases=50,
success_threshold=0.90):
    """
    V2.6: Tests the full Generator -> Normalizer -> Parser pipeline using helper
functions.
    """
    log.append("--- Running V2.6: Normalizer-Parser Integration Check ---")

    try:
        normalizer, generator, reverse_map = _setup_integration_test(parser, graph,
norm_map)
        success_count, failures = 0, 0
        failed_examples = []

        for _ in range(num_phrases):
            # Stage A: Generate a canonical phrase
            canonical_phrase, _ = generator.generate()
            if not canonical_phrase: continue

            # Stage B: De-normalize it
            messy_phrase = _denormalize_phrase(canonical_phrase, reverse_map)

            # Stage C: Run it through the pipeline
            is_success, errors, candidates, log= _run_pipeline_on_phrase(messy_phrase,
normalizer, parser,log)

            if is_success:
                success_count += 1
            else:
                failures += 1
                if len(failed_examples) < 3:
                    failed_examples.append({
```

```
                    'original': canonical_phrase, 'messy': messy_phrase,
                    'normalized': candidates, 'errors': f'{errors}'
                })

        # Report Results
        total = success_count + failures
        if total == 0:
            log.append("WARNING: No phrases were generated to test.")
            return True, log

        rate = success_count / total
        log.append(f"  - Integration test complete. Success Rate: {rate:.0%}")
        if rate >= success_threshold:
            log.append(f"PASS: Success rate ({rate:.0%}) meets threshold.")
            return True, log
        else:
            log.append(f"FAIL: Success rate ({rate:.0%}) is below threshold.")
            log.append("  - Example Failures:")
            for fail in failed_examples:
                log.append(f"    - Original: '{fail['original']}'")
                log.append(f"    - Messy:    '{fail['messy']}'")
                log.append(f"    - Normalized To: {fail['normalized']}")
                log.append(f"    - Failure Reasons: {fail['errors']}")
            return False, log

    except Exception as e:
        log.append(f"FAIL: An unexpected error occurred during the integration test: {e}")
        return False, log

```

nlq_to_sql_pipeline.py

```
# File: nlq_to_sql_pipeline.py

import sys
import yaml

import config
from lark import Lark

# --- Generators (pure) ---
from src.n2s_generators.graph_builder import generate_relationship_graph
from src.n2s_generators.knowledge_compiler import (
    build_vocabulary,
    build_binder,
    build_canonical_grammar,
)

# --- Source validators (pure) ---
from src.n2s_validators.source_validator import (
    validate_schema_yaml,
    validate_keywords_yaml,
)

# --- Graph validators (pure) ---
from src.n2s_validators.graph_validator import (
    validate_graph_structure,                 # G1
    validate_graph_referential_integrity,     # G2
    validate_graph_type_label_coherence,      # G3
    validate_function_compatibility_matrix,   # G4
    validate_alias_hygiene,                   # G5
    validate_pluralization,                   # G6
    validate_reserved_token_safety,           # G7
    validate_graph_stability,                 # G8
)

# --- Vocabulary validators (pure) ---
from src.n2s_validators.vocabulary_validator import (
    validate_vocab_coverage,                  # V1
    validate_vocab_partition_sanity,          # V2
    validate_vocab_policy_enforcement,        # V3
    validate_vocab_identity_presence,         # V4
    validate_vocab_entropy,                   # V5
    validate_vocab_serialization_safety,      # V6
)

# --- Binder validators (pure) ---
from src.n2s_validators.binder_validator import (
    validate_binder_shape,                    # B1
    validate_binder_linkage,                  # B2
    validate_binder_unifiability,             # B3
    validate_binder_ambiguity_cost,           # B4
    validate_binder_connector_rules,          # B5
```

```
    validate_binder_dead_overlapping,         # B6
)

# --- Grammar validators (pure) ---
from src.n2s_validators.grammar_validator import (
    validate_grammar_vocab_alignment,         # Gm1
    validate_grammar_smoke_tests,             # Gm2
    validate_grammar_stress,                  # Gm3
    validate_grammar_ambiguity,               # Gm4
    validate_grammar_health,                  # Gm5
)

# --- Cross-artifact validators (pure canonical path; no normalizer) ---
from src.n2s_validators.cross_artifact_validator import (
    validate_canonical_roundtrip,             # C1
    validate_binder_sql_feasibility,          # C2 (optional)
    validate_negative_canonical,              # C3
)

# --- Full integration (uses normalizer) ---
from src.n2s_validators.full_integration_validator import (
    validate_full_integration_all,            # I1I3 aggregator
)


# ----------------------
# Filesystem helpers
# ----------------------
def load_yaml(path):
    try:
        with open(path, "r") as f:
            return yaml.safe_load(f)
    except FileNotFoundError:
        return None


def write_yaml(data, path):
    with open(path, "w") as f:
        yaml.dump(data, f, sort_keys=False)


def write_text(data_str, path):
    with open(path, "w") as f:
        f.write(data_str)


def run_validation_check(validator_func, *args):
    """
    Helper to run a validator and print its log.
    Validators must have signature (args, log) -> (bool, log_list)
    """
    log = []
    is_success, final_log = validator_func(*args, log)
    header = final_log[0] if final_log else f"{validator_func.__name__}"
```

```
    if is_success:
        print(f"  {header} -> PASS")
    else:
        print(f"  {header} -> FAIL")
        print("\n--- Failure Log ---")
        for msg in final_log:
            print(msg)
        print("-------------------\n")
    return is_success


# ----------------------
# Main pipeline
# ----------------------
def main():
    print("--- Starting NLQ-to-SQL Onboarding & QA Pipeline ---")

    # --- Phase 0: Load sources ---
    print("\nLoading source YAML files...")
    schema_data = load_yaml(config.SCHEMA_PATH)
    keywords_data = load_yaml(config.KEYWORDS_PATH)
    if not (schema_data and keywords_data):
        print("Error: Source YAML files not found. Exiting.")
        return

    # --- Phase 0: Source validation (structure only; pure) ---
    print("\n[V0] Running Source File Validation...")
    if not run_validation_check(validate_schema_yaml, schema_data):
        return
    if not run_validation_check(validate_keywords_yaml, keywords_data):
        return

    # --- Phase 1: Relationship Graph generation (pure) ---
    print("\n[G1] Generating Relationship Graph...")
    relationship_graph = generate_relationship_graph(schema_data, keywords_data)
    write_yaml(relationship_graph, config.GRAPH_PATH)
    print(f"  -> Generated {config.GRAPH_PATH}")

    # --- Phase 1.x: Graph validations (pure) ---
    print("\n[V1] Running Graph Quality Checks...")
    run_validation_check(validate_graph_structure, relationship_graph)
    run_validation_check(validate_graph_referential_integrity, relationship_graph)
    run_validation_check(validate_graph_type_label_coherence, relationship_graph)
    run_validation_check(validate_function_compatibility_matrix, relationship_graph)
    run_validation_check(validate_alias_hygiene, relationship_graph)
    run_validation_check(validate_pluralization, relationship_graph)
    run_validation_check(validate_reserved_token_safety, relationship_graph)
    # Optional stability pass: rebuild and compare (pure)
            run_validation_check(validate_graph_stability,   generate_relationship_graph,
schema_data, keywords_data)


    # --- Phase 2: Compile artifacts (pure) ---
    print("\n[G2] Compiling Knowledge Artifacts...")
```

```
    # 2.1 Vocabulary (replaces normalization_map)
    vocabulary = build_vocabulary(relationship_graph)
    vocab_path = getattr(config, "VOCAB_PATH", None) or getattr(config, "NORMALIZATION_MAP_PATH", None) or "vocabulary.yaml"
    write_yaml(vocabulary, vocab_path)
    print(f"  -> Generated {vocab_path}")

    # 2.2 Binder (new)
    binder = build_binder(relationship_graph)
    binder_path = getattr(config, "BINDER_PATH", None) or "binder.yaml"
    write_yaml(binder, binder_path)
    print(f"  -> Generated {binder_path}")

    # 2.3 Canonical Grammar
    grammar_text = build_canonical_grammar(relationship_graph)
    grammar_path = getattr(config, "GRAMMAR_PATH", None) or "canonical_grammar.lark"
    write_text(grammar_text, grammar_path)
    print(f"  -> Generated {grammar_path}")

    # Parser instance for grammar/cross-artifact tests
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        print(f"  FAIL: Could not build parser from grammar ({grammar_path}): {e}")
        sys.exit(1)

    # --- Phase 2.5: Artifact validators (pure) ---
    print("\n[V2] Running Artifact Quality Checks...")

    # Vocab checks
    run_validation_check(validate_vocab_coverage, relationship_graph, vocabulary)
    run_validation_check(validate_vocab_partition_sanity, relationship_graph, vocabulary)
    run_validation_check(validate_vocab_policy_enforcement, relationship_graph,vocabulary)
    run_validation_check(validate_vocab_identity_presence, relationship_graph, vocabulary)
    run_validation_check(validate_vocab_entropy, vocabulary)
    run_validation_check(validate_vocab_serialization_safety, vocabulary)

    # Binder checks
    run_validation_check(validate_binder_shape, binder)
    run_validation_check(validate_binder_linkage, binder, relationship_graph, vocabulary)
    run_validation_check(validate_binder_unifiability, binder)
    run_validation_check(validate_binder_ambiguity_cost, binder)
    run_validation_check(validate_binder_connector_rules, binder)
    run_validation_check(validate_binder_dead_overlapping, binder)

    # Grammar checks
    run_validation_check(validate_grammar_vocab_alignment, relationship_graph, grammar_text)
    run_validation_check(validate_grammar_smoke_tests, relationship_graph, grammar_text)
```

```python
    run_validation_check(validate_grammar_stress, relationship_graph, grammar_text)
    run_validation_check(validate_grammar_ambiguity, relationship_graph, grammar_text)
    run_validation_check(validate_grammar_health, grammar_text)

    # --- Phase 3: Cross-artifact validators (canonical-only; pure) ---
    print("\n[V3] Running Cross-Artifact Validations (canonical path)...")
    run_validation_check(validate_canonical_roundtrip, relationship_graph, grammar_text)
    # Optional (enable if you have SQL templates wired):
    run_validation_check(validate_binder_sql_feasibility, relationship_graph)
    run_validation_check(validate_negative_canonical, relationship_graph, grammar_text)

    # --- Phase 4: Full integration (requires normalizer) ---
    print("\n[V4] Running Full Integration Validations (with normalizer)...")
    # Thresholds and sizes can be configured via config if desired
    random_phrases = getattr(config, "INTEG_RANDOM_PHRASES", 100)
    random_threshold = getattr(config, "INTEG_RANDOM_THRESHOLD", 0.90)
    lossiness_phrases = getattr(config, "INTEG_LOSSINESS_PHRASES", 100)
    golden_threshold = getattr(config, "INTEG_GOLDEN_THRESHOLD", 1.0)
    golden_set = load_yaml(getattr(config, "GOLDEN_SET_PATH", "")) or []

    def run_full():
        log = []
        ok, log = validate_full_integration_all(
            graph=relationship_graph,
            vocabulary=vocabulary,
            grammar_text=grammar_text,
            log=log,
            random_phrases=random_phrases,
            random_threshold=random_threshold,
            lossiness_phrases=lossiness_phrases,
            golden_queries=golden_set,
            golden_threshold=golden_threshold,
            max_candidates=getattr(config, "NORMALIZER_MAX_CANDIDATES", 50),
            rng_seed=getattr(config, "RNG_SEED", None),
        )
        header = log[0] if log else "Full Integration"
        if ok:
            print(f"  {header} -> PASS")
        else:
            print(f"  {header} -> FAIL")
            print("\n--- Failure Log ---")
            for m in log:
                print(m)
            print("-------------------\n")
        return ok

    run_full()

    print("\n--- Pipeline Finished ---")


if __name__ == "__main__":
    main()
```

normalization_map.yaml


```yaml
deterministic_aliases:
  user_id: user_id
  username: username
  handle: username
  login: username
  usernames: username
  age: age
  ages: age
  balance: balance
  balances: balance
  credit: balance
  funds: balance
  is_active: is_active
  status: is_active
  last_login: last_login
  location: location
  coordinates: location
  locations: location
  place: location
  position: location
  users: users
  account: users
  accounts: users
  client: users
  clients: users
  customer: users
  customers: users
  people: users
  user: users
  sale_id: sale_id
  product_name: product_name
  item: product_name
  product: product_name
  sale_date: sale_date
  quantity: quantity
  quantities: quantity
  price: price
  charge: price
  cost: price
  prices: price
  value: price
  sales: sales
  order: sales
  orders: sales
  purchase: sales
  purchases: sales
  sale: sales
  transaction: sales
  transactions: sales
  region_id: region_id
  name: name
```

identifier: name
label: name
names: name
title: name
boundaries: boundaries
border: boundaries
boundary: boundaries
geometry: boundaries
outline: boundaries
shape: boundaries
regions: regions
areas: regions
district: regions
region: regions
territory: regions
zone: regions
zones: regions
select: select
display: select
fetch: select
find: select
get: select
list: select
retrieve: select
show: select
what: select
of: of
from: from
_on: _on
'on': _on
at: at
belonging to: belonging to
and: and
'&&': and
or: or
'||': or
not: not
'!': not
equal: equal
'=': equal
==: equal
equals: equal
exactly: equal
is: equal
not_equal: not_equal
'!=': not_equal
<>: not_equal
isn't: not_equal
greater_than: greater_than
'>': greater_than
above: greater_than
exceeds: greater_than
over: greater_than
less_than: less_than

```
<: less_than
below: less_than
under: less_than
greater_than_or_equal: greater_than_or_equal
'>=': greater_than_or_equal
less_than_or_equal: less_than_or_equal
<=: less_than_or_equal
between: between
in: in
like: like
matches: like
is_null: is_null
is_not_null: is_not_null
exists: is_not_null
count: count
number: count
sum: sum
aggregate: sum
total: sum
avg: avg
average: avg
mean: avg
min: min
bottom: min
least: min
lowest: min
minimum: min
smallest: min
max: max
greatest: max
highest: max
largest: max
maximum: max
most: max
distinct: distinct
unique: distinct
order_by_asc: order_by_asc
order_by_desc: order_by_desc
group_by: group_by
having: having
with: having
limit: limit
first: limit
only: limit
top: limit
extract: extract
length: length
concat: concat
concatenate: concat
join: concat
cast: cast
convert: cast
st_perimeter: st_perimeter
st_distance: st_distance
```

```
  distance: st_distance
  st_intersects: st_intersects
  intersects: st_intersects
  st_area: st_area
  st_length: st_length
  st_x: st_x
  lon: st_x
  longitude: st_x
  st_y: st_y
  lat: st_y
  latitude: st_y
  st_within: st_within
  inside: st_within
  within: st_within
  st_contains: st_contains
  contains: st_contains
  encloses: st_contains
  surrounds: st_contains
  st_geometrytype: st_geometrytype
  st_buffer: st_buffer
  buffer: st_buffer
  st_union: st_union
  merge: st_union
  union: st_union
  st_centroid: st_centroid
  center: st_centroid
  centroid: st_centroid
  st_simplify: st_simplify
  generalize: st_simplify
  simplify: st_simplify
  st_touches: st_touches
  borders: st_touches
  touches: st_touches
  st_crosses: st_crosses
  crosses: st_crosses
  st_spatial_index: st_spatial_index
  st_distance_operator: st_distance_operator
  closest: st_distance_operator
  nearest: st_distance_operator
  st_transform: st_transform
  reproject: st_transform
  transform: st_transform
  logins: username
  items: product_name
  values: price
non_deterministic_aliases:
  account number:
  - user_id
  client id:
  - user_id
  customer id:
  - user_id
  user id:
  - user_id
```

user identifier:
- user_id
user ids:
- user_id
user number:
- user_id
account name:
- username
user name:
- username
user names:
- username
years old:
- age
account balance:
- balance
amount:
- balance
- price
account status:
- is_active
active status:
- is_active
is active:
- is_active
is actives:
- is_active
last access:
- last_login
last active date:
- last_login
last login:
- last_login
last logins:
- last_login
last signed in:
- last_login
most recent login:
- last_login
geo location:
- location
geographic coordinates:
- location
order id:
- sale_id
order number:
- sale_id
purchase id:
- sale_id
sale id:
- sale_id
sale ids:
- sale_id
transaction id:

- sale_id
item name:
- product_name
item sold:
- product_name
product name:
- product_name
product names:
- product_name
order date:
- sale_date
purchase date:
- sale_date
sale date:
- sale_date
sale dates:
- sale_date
transaction date:
- sale_date
when it was sold:
- sale_date
amount sold:
- quantity
item count:
- quantity
number of items:
- quantity
number sold:
- quantity
how much:
- price
area id:
- region_id
region id:
- region_id
region ids:
- region_id
region number:
- region_id
zone id:
- region_id
area shape:
- boundaries
perimeter:
- boundaries
- st_perimeter
area:
- regions
- st_area
give me:
- select
tell me:
- select
equal to:

```
- equal
is exactly:
- equal
is the same as:
- equal
does not equal:
- not_equal
is not:
- not_equal
is not equal to:
- not_equal
is greater than:
- greater_than
more than:
- greater_than
is less than:
- less_than
smaller than:
- less_than
at least:
- greater_than_or_equal
is at least:
- greater_than_or_equal
is greater than or equal to:
- greater_than_or_equal
not less than:
- greater_than_or_equal
at most:
- less_than_or_equal
is at most:
- less_than_or_equal
is less than or equal to:
- less_than_or_equal
not more than:
- less_than_or_equal
up to:
- less_than_or_equal
in the range of:
- between
is between:
- between
is in:
- in
is one of:
- in
one of:
- in
ends with:
- like
starts with:
- like
has no value:
- is_null
is empty:
```

- is_null
is missing:
- is_null
is null:
- is_null
has a value:
- is_not_null
is not empty:
- is_not_null
is not null:
- is_not_null
is present:
- is_not_null
how many:
- count
number of:
- count
quantity of:
- count
total number of:
- count
sum of:
- sum of
total amount of:
- sum of
total of:
- sum of
average of:
- avg of
unique values of:
- distinct of
in ascending order:
- order_by_asc
order by ascending:
- order_by_asc
sort by ascending:
- order_by_asc
in descending order:
- order_by_desc
order by descending:
- order_by_desc
sort by descending:
- order_by_desc
group by:
- group_by
grouped by:
- group_by
get the:
- extract
character count:
- length
length in:
- length
length of:

- length
- st_length
string length:
- length
combine:
- concat
- st_union
change type:
- cast
boundary length:
- st_perimeter
length of boundary:
- st_perimeter
outline length:
- st_perimeter
distance between:
- st_distance
how far:
- st_distance
separation of:
- st_distance of
overlaps with:
- st_intersects
area of:
- st_area
size of:
- st_area
surface area:
- st_area
distance along:
- st_length
distance of:
- st_length
length along:
- st_length
line length:
- st_length
path length:
- st_length
x coordinate:
- st_x
x pos:
- st_x
y coordinate:
- st_y
y pos:
- st_y
contained in:
- st_within
is inside:
- st_within
is within:
- st_within
geometry type:

- st_geometrytype
shape type:
- st_geometrytype
type of geometry:
- st_geometrytype
what kind of shape:
- st_geometrytype
area around:
- st_buffer
buffer around:
- st_buffer
expand by:
- st_buffer
union of:
- st_union
center point:
- st_centroid
geometric center:
- st_centroid
simplify shape:
- st_simplify
is adjacent to:
- st_touches
goes across:
- st_crosses
spatial index:
- st_spatial_index
closest to:
- st_distance_operator
convert coordinate system:
- st_transform
client ids:
- user_id
customer ids:
- user_id
account names:
- username
last active dates:
- last_login
most recent logins:
- last_login
order ids:
- sale_id
purchase ids:
- sale_id
transaction ids:
- sale_id
item names:
- product_name
order dates:
- sale_date
purchase dates:
- sale_date
transaction dates:

```
  - sale_date
  area ids:
  - region_id
  zone ids:
  - region_id
connectors: []
_diagnostics:
  prefix_collisions:
  - alias: amount
    longer_keys:
    - amount sold
    action: kept_table_meaning
  - alias: account
    longer_keys:
    - account balance
    - account name
    - account names
    - account number
    - account status
    action: kept_table_meaning
  - alias: client
    longer_keys:
    - client id
    - client ids
    action: kept_table_meaning
  - alias: customer
    longer_keys:
    - customer id
    - customer ids
    action: kept_table_meaning
  - alias: user
    longer_keys:
    - user id
    - user identifier
    - user ids
    - user name
    - user names
    - user number
    action: kept_table_meaning
  - alias: item
    longer_keys:
    - item count
    - item name
    - item names
    - item sold
    action: kept_table_meaning
  - alias: product
    longer_keys:
    - product name
    - product names
    action: kept_table_meaning
  - alias: quantity
    longer_keys:
    - quantity of
```

```
  action: kept_table_meaning
- alias: order
  longer_keys:
  - order by ascending
  - order by descending
  - order date
  - order dates
  - order id
  - order ids
  - order number
  action: kept_table_meaning
- alias: purchase
  longer_keys:
  - purchase date
  - purchase dates
  - purchase id
  - purchase ids
  action: kept_table_meaning
- alias: sale
  longer_keys:
  - sale date
  - sale dates
  - sale id
  - sale ids
  action: kept_table_meaning
- alias: transaction
  longer_keys:
  - transaction date
  - transaction dates
  - transaction id
  - transaction ids
  action: kept_table_meaning
- alias: boundary
  longer_keys:
  - boundary length
  action: kept_table_meaning
- alias: geometry
  longer_keys:
  - geometry type
  action: kept_table_meaning
- alias: outline
  longer_keys:
  - outline length
  action: kept_table_meaning
- alias: shape
  longer_keys:
  - shape type
  action: kept_table_meaning
- alias: area
  longer_keys:
  - area around
  - area id
  - area ids
  - area of
```

```
  - area shape
  action: kept_table_meaning
- alias: region
  longer_keys:
  - region id
  - region ids
  - region number
  action: kept_table_meaning
- alias: zone
  longer_keys:
  - zone id
  - zone ids
  action: kept_table_meaning
- alias: get
  longer_keys:
  - get the
  action: kept_table_meaning
- alias: what
  longer_keys:
  - what kind of shape
  action: kept_table_meaning
- alias: at
  longer_keys:
  - at least
  - at most
  action: kept_table_meaning
- alias: not
  longer_keys:
  - not less than
  - not more than
  action: kept_table_meaning
- alias: equal
  longer_keys:
  - equal to
  action: kept_table_meaning
- alias: is
  longer_keys:
  - is active
  - is actives
  - is adjacent to
  - is at least
  - is at most
  - is between
  - is empty
  - is exactly
  - is greater than
  - is greater than or equal to
  - is in
  - is inside
  - is less than
  - is less than or equal to
  - is missing
  - is not
  - is not empty
```

```
    - is not equal to
    - is not null
    - is null
    - is one of
    - is present
    - is the same as
    - is within
    action: kept_table_meaning
  - alias: in
    longer_keys:
    - in ascending order
    - in descending order
    - in the range of
    action: kept_table_meaning
  - alias: number
    longer_keys:
    - number of
    - number of items
    - number sold
    action: kept_table_meaning
  - alias: sum
    longer_keys:
    - sum of
    action: kept_table_meaning
  - alias: total
    longer_keys:
    - total amount of
    - total number of
    - total of
    action: kept_table_meaning
  - alias: average
    longer_keys:
    - average of
    action: kept_table_meaning
  - alias: most
    longer_keys:
    - most recent login
    - most recent logins
    action: kept_table_meaning
  - alias: unique
    longer_keys:
    - unique values of
    action: kept_table_meaning
  - alias: length
    longer_keys:
    - length along
    - length in
    - length of
    - length of boundary
    action: kept_table_meaning
  - alias: convert
    longer_keys:
    - convert coordinate system
    action: kept_table_meaning
```

```yaml
- alias: distance
  longer_keys:
  - distance along
  - distance between
  - distance of
  action: kept_table_meaning
- alias: buffer
  longer_keys:
  - buffer around
  action: kept_table_meaning
- alias: union
  longer_keys:
  - union of
  action: kept_table_meaning
- alias: center
  longer_keys:
  - center point
  action: kept_table_meaning
- alias: simplify
  longer_keys:
  - simplify shape
  action: kept_table_meaning
- alias: closest
  longer_keys:
  - closest to
  action: kept_table_meaning
preposition_conflicts: []
domain_conflicts: []
generic_denied:
- order by
- sort by
plural_added:
- from: client id
  to: client ids
- from: customer id
  to: customer ids
- from: account name
  to: account names
- from: login
  to: logins
- from: last active date
  to: last active dates
- from: most recent login
  to: most recent logins
- from: order id
  to: order ids
- from: purchase id
  to: purchase ids
- from: transaction id
  to: transaction ids
- from: item
  to: items
- from: item name
  to: item names
```

```
- from: order date
  to: order dates
- from: purchase date
  to: purchase dates
- from: transaction date
  to: transaction dates
- from: value
  to: values
- from: area id
  to: area ids
- from: zone id
  to: zone ids
of_policy_adjusted:
- alias: sum of
  canonical: sum
  to: sum of
- alias: total amount of
  canonical: sum
  to: sum of
- alias: total of
  canonical: sum
  to: sum of
- alias: average of
  canonical: avg
  to: avg of
- alias: unique values of
  canonical: distinct
  to: distinct of
- alias: separation of
  canonical: st_distance
  to: st_distance of
identity_added: []
identity_overrides: []
```

podman-compose.yaml

```yaml
version: '3.8'

services:
  dev:
    image: natural-language-sql-dev:latest
    build:
      context: .
      podmanfile: Containerfile
    volumes:
      - .:/app:Z
    tty: true
    stdin_open: true
    command: /bin/bash
```

pyproject.toml

```toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "wcm-nlq2sql"
version = "0.1.0"
authors = [
  { name="Your Name", email="your.email@example.com" },
]
description = "A Natural Language to SQL converter using LLMs and Grammar."
readme = "README.md"
requires-python = ">=3.8"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.optional-dependencies]
dev = [
    "pytest",
    "pytest-cov",
    "flake8",
]

[tool.setuptools.packages.find]
where = ["src"]
```

relationship_graph.yaml


```
user_id:
  entity_type: column
  metadata: &id008
    aliases:
    - account number
    - client id
    - customer id
    - user id
    - user identifier
    - user ids
    - user number
    - user_id
    type: INTEGER
    type_category: numeric
    labels:
    - id
username:
  entity_type: column
  metadata: &id001
    aliases:
    - account name
    - handle
    - login
    - user name
    - user names
    - username
    - usernames
    type: VARCHAR(50)
    type_category: text
    labels: []
age:
  entity_type: column
  metadata: &id002
    aliases:
    - age
    - ages
    - years old
    type: INT
    type_category: numeric
    labels: []
balance:
  entity_type: column
  metadata: &id003
    aliases:
    - account balance
    - amount
    - balance
    - balances
    - credit
    - funds
    type: DECIMAL(10, 2)
```

```yaml
    type_category: numeric
    labels: []
is_active:
  entity_type: column
  metadata: &id004
    aliases:
    - account status
    - active status
    - is active
    - is actives
    - is_active
    - status
    type: BOOLEAN
    type_category: boolean
    labels: []
last_login:
  entity_type: column
  metadata: &id005
    aliases:
    - last access
    - last active date
    - last login
    - last logins
    - last signed in
    - last_login
    - most recent login
    type: TIMESTAMP
    type_category: timestamp
    labels: []
location:
  entity_type: column
  metadata: &id006
    aliases:
    - coordinates
    - geo location
    - geographic coordinates
    - location
    - locations
    - place
    - position
    type: geography_point
    type_category: geography_point
    labels:
    - postgis
users:
  entity_type: table
  metadata:
    aliases:
    - account
    - accounts
    - client
    - clients
    - customer
    - customers
```

```
    - people
    - user
    - users
    columns:
      user_id:
        aliases:
        - account number
        - client id
        - customer id
        - user id
        - user identifier
        - user ids
        - user number
        - user_id
        type: INTEGER
        type_category: numeric
        labels:
        - id
      username: *id001
      age: *id002
      balance: *id003
      is_active: *id004
      last_login: *id005
      location: *id006
sale_id:
  entity_type: column
  metadata: &id007
    aliases:
    - order id
    - order number
    - purchase id
    - sale id
    - sale ids
    - sale_id
    - transaction id
    type: INTEGER
    type_category: numeric
    labels:
    - id
product_name:
  entity_type: column
  metadata: &id009
    aliases:
    - item
    - item name
    - item sold
    - product
    - product name
    - product names
    - product_name
    type: TEXT
    type_category: text
    labels: []
sale_date:
```

```
  entity_type: column
  metadata: &id010
    aliases:
    - order date
    - purchase date
    - sale date
    - sale dates
    - sale_date
    - transaction date
    - when it was sold
    type: DATE
    type_category: date
    labels: []
quantity:
  entity_type: column
  metadata: &id011
    aliases:
    - amount sold
    - item count
    - number of items
    - number sold
    - quantities
    - quantity
    type: INT
    type_category: numeric
    labels: []
price:
  entity_type: column
  metadata: &id012
    aliases:
    - amount
    - charge
    - cost
    - how much
    - price
    - prices
    - value
    type: FLOAT
    type_category: numeric
    labels: []
sales:
  entity_type: table
  metadata:
    aliases:
    - order
    - orders
    - purchase
    - purchases
    - sale
    - sales
    - transaction
    - transactions
    columns:
      sale_id: *id007
```

```
      user_id: *id008
      product_name: *id009
      sale_date: *id010
      quantity: *id011
      price: *id012
region_id:
  entity_type: column
  metadata: &id013
    aliases:
    - area id
    - region id
    - region ids
    - region number
    - region_id
    - zone id
    type: INTEGER
    type_category: numeric
    labels:
    - id
name:
  entity_type: column
  metadata: &id014
    aliases:
    - identifier
    - label
    - name
    - names
    - title
    type: VARCHAR(50)
    type_category: text
    labels: []
boundaries:
  entity_type: column
  metadata: &id015
    aliases:
    - area shape
    - border
    - boundaries
    - boundary
    - geometry
    - outline
    - perimeter
    - shape
    type: geography_polygon
    type_category: geography_polygon
    labels:
    - postgis
regions:
  entity_type: table
  metadata:
    aliases:
    - area
    - areas
    - district
```

```
    - region
    - regions
    - territory
    - zone
    - zones
    columns:
      region_id: *id013
      name: *id014
      boundaries: *id015
select:
  entity_type: select_verbs
  metadata:
    aliases:
    - display
    - fetch
    - find
    - get
    - give me
    - list
    - retrieve
    - select
    - show
    - tell me
    - what
of:
  entity_type: prepositions
  metadata:
    aliases:
    - of
from:
  entity_type: prepositions
  metadata:
    aliases:
    - from
_on:
  entity_type: prepositions
  metadata:
    aliases:
    - _on
    - 'on'
at:
  entity_type: prepositions
  metadata:
    aliases:
    - at
belonging to:
  entity_type: prepositions
  metadata:
    aliases:
    - belonging to
and:
  entity_type: logical_operators
  metadata:
    aliases:
```

```
    - '&&'
    - and
    template: '{clause1} AND {clause2}'
or:
  entity_type: logical_operators
  metadata:
    aliases:
    - or
    - '||'
    template: '{clause1} OR {clause2}'
not:
  entity_type: logical_operators
  metadata:
    aliases:
    - '!'
    - not
    template: NOT {clause}
equal:
  entity_type: comparison_operators
  metadata:
    aliases:
    - '='
    - ==
    - equal
    - equal to
    - equals
    - exactly
    - is
    - is exactly
    - is the same as
    applicable_types:
      column:
      - any
      value:
      - any
    label_rules: []
    explanation: Tests for equality between two values.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    - name: value
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - '='
      - '{value}'
      commutative: false
```

```yaml
    - pattern:
      - '{column}'
      - is
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - equals
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - equal
      - to
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - '='
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - ==
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - exactly
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - exactly
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - the
      - same
      - as
      - '{value}'
      commutative: false
not_equal:
  entity_type: comparison_operators
  metadata:
    aliases:
    - '!='
    - <>
    - does not equal
    - is not
    - is not equal to
```

```yaml
    - isn't
    - not_equal
    applicable_types:
      column:
      - any
      value:
      - any
    label_rules: []
    explanation: Tests for inequality between two values.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    - name: value
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - '!='
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - not
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - '!='
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - <>
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - not
      - equal
      - to
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - does
      - not
```

```yaml
      - equal
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - isn't
      - '{value}'
      commutative: false
greater_than:
  entity_type: comparison_operators
  metadata:
    aliases:
    - '>'
    - above
    - exceeds
    - greater_than
    - is greater than
    - more than
    - over
    applicable_types:
      column:
      - numeric
      - date
      - timestamp
      value:
      - numeric
      - date
      - timestamp
    label_rules: []
    explanation: Tests if the first value is greater than the second.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - numeric
      - date
      - timestamp
    - name: value
      types:
      - numeric
      - date
      - timestamp
    surfaces:
    - pattern:
      - '{column}'
      - '>'
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
```

```
      - greater
      - than
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - '>'
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - more
      - than
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - over
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - above
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - exceeds
      - '{value}'
      commutative: false
less_than:
  entity_type: comparison_operators
  metadata:
    aliases:
    - <
    - below
    - is less than
    - less_than
    - smaller than
    - under
    applicable_types:
      column:
      - numeric
      - date
      - timestamp
      value:
      - numeric
      - date
      - timestamp
    label_rules: []
    explanation: Tests if the first value is less than the second.
  binder:
    returns_type: boolean
    class: predicate
```

```yaml
    clause: where
    args:
    - name: column
      types:
      - numeric
      - date
      - timestamp
    - name: value
      types:
      - numeric
      - date
      - timestamp
    surfaces:
    - pattern:
      - '{column}'
      - <
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - less
      - than
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - <
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - smaller
      - than
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - under
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - below
      - '{value}'
      commutative: false
greater_than_or_equal:
  entity_type: comparison_operators
  metadata:
    aliases:
    - '>='
    - at least
    - greater_than_or_equal
    - is at least
```

```yaml
  - is greater than or equal to
  - not less than
  applicable_types:
    column:
    - numeric
    - date
    - timestamp
    value:
    - numeric
    - date
    - timestamp
  label_rules: []
  explanation: Tests if the first value is greater than or equal to the second.
binder:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - numeric
    - date
    - timestamp
  - name: value
    types:
    - numeric
    - date
    - timestamp
  surfaces:
  - pattern:
    - '{column}'
    - '>='
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - is
    - greater
    - than
    - or
    - equal
    - to
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - '>='
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - at
    - least
    - '{value}'
```

```yaml
      commutative: false
    - pattern:
      - '{column}'
      - is
      - at
      - least
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - not
      - less
      - than
      - '{value}'
      commutative: false
less_than_or_equal:
  entity_type: comparison_operators
  metadata:
    aliases:
    - <=
    - at most
    - is at most
    - is less than or equal to
    - less_than_or_equal
    - not more than
    - up to
    applicable_types:
      column:
      - numeric
      - date
      - timestamp
      value:
      - numeric
      - date
      - timestamp
    label_rules: []
    explanation: Tests if the first value is less than or equal to the second.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - numeric
      - date
      - timestamp
    - name: value
      types:
      - numeric
      - date
      - timestamp
    surfaces:
    - pattern:
```

```
      - '{column}'
      - <=
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - less
      - than
      - or
      - equal
      - to
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - <=
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - at
      - most
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - at
      - most
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - not
      - more
      - than
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - up
      - to
      - '{value}'
      commutative: false
between:
  entity_type: comparison_operators
  metadata:
    aliases:
    - between
    - in the range of
    - is between
    applicable_types:
      column:
```

```yaml
    - numeric
    - date
    - timestamp
    value1:
    - numeric
    - date
    - timestamp
    value2:
    - numeric
    - date
    - timestamp
  label_rules: []
  explanation: Tests if a value falls within a specified range.
binder:
  returns_type: boolean
  class: predicate
  clause: where
  args:
  - name: column
    types:
    - numeric
    - date
    - timestamp
  - name: value1
    types:
    - numeric
    - date
    - timestamp
  - name: value2
    types:
    - numeric
    - date
    - timestamp
  surfaces:
  - pattern:
    - '{column}'
    - BETWEEN
    - '{value1}'
    - AND
    - '{value2}'
    commutative: false
  - pattern:
    - '{column}'
    - between
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
    - is
    - between
    - '{value}'
    commutative: false
  - pattern:
    - '{column}'
```

```yaml
      - in
      - the
      - range
      - of
      - '{value}'
      commutative: false
in:
  entity_type: comparison_operators
  metadata:
    aliases:
    - in
    - is in
    - is one of
    - one of
    applicable_types:
      column:
      - any
      values:
      - any
    label_rules: []
    explanation: Tests if a value is present in a given list of values.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    - name: values
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - IN
      - (
      - '{values}'
      - )
      commutative: false
    - pattern:
      - '{column}'
      - in
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - in
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
```

```yaml
      - one
      - of
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - one
      - of
      - '{value}'
      commutative: false
like:
  entity_type: comparison_operators
  metadata:
    aliases:
    - ends with
    - like
    - matches
    - starts with
    applicable_types:
      column:
      - text
      value:
      - text
    label_rules: []
    explanation: Performs a pattern matching search using wildcards.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - text
    - name: value
      types:
      - text
    surfaces:
    - pattern:
      - '{column}'
      - LIKE
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - like
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - matches
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
```

```yaml
      - starts
      - with
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - ends
      - with
      - '{value}'
      commutative: false
is_null:
  entity_type: comparison_operators
  metadata:
    aliases:
    - has no value
    - is empty
    - is missing
    - is null
    - is_null
    applicable_types:
      column:
      - any
    label_rules: []
    explanation: Tests if a column's value is NULL.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - IS
      - 'NULL'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - 'null'
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - empty
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - has
      - 'no'
```

```yaml
      - value
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - missing
      - '{value}'
      commutative: false
is_not_null:
  entity_type: comparison_operators
  metadata:
    aliases:
    - exists
    - has a value
    - is not empty
    - is not null
    - is present
    - is_not_null
    applicable_types:
      column:
      - any
    label_rules: []
    explanation: Tests if a column's value is not NULL.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - IS
      - NOT
      - 'NULL'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - not
      - 'null'
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - not
      - empty
      - '{value}'
      commutative: false
    - pattern:
```

```
      - '{column}'
      - has
      - a
      - value
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - exists
      - '{value}'
      commutative: false
    - pattern:
      - '{column}'
      - is
      - present
      - '{value}'
      commutative: false
count:
  entity_type: sql_actions
  metadata:
    aliases:
    - count
    - how many
    - number
    - number of
    - quantity of
    - total number of
    applicable_types:
      column:
      - any
    label_rules:
    - not id
    explanation: Returns the number of rows that match a specified criterion.
  binder:
    returns_type: numeric
    class: aggregate
    clause: select
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - COUNT
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - count
      - (
      - '{column}'
      - )
      commutative: false
```

```yaml
    - pattern:
      - how
      - many
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - number
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - number
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - quantity
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - total
      - number
      - of
      - (
      - '{column}'
      - )
      commutative: false
sum:
  entity_type: sql_actions
  metadata:
    aliases:
    - aggregate
    - sum
    - sum of
    - total
    - total amount of
    - total of
    applicable_types:
      column:
      - numeric
    label_rules:
    - not id
    explanation: Returns the sum of all values in a numeric column.
  binder:
    returns_type: numeric
    class: aggregate
```

```
    clause: select
    args:
    - name: column
      types:
      - numeric
    surfaces:
    - pattern:
      - SUM
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - aggregate
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - sum
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - sum
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - total
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - total
      - amount
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - total
      - of
      - (
      - '{column}'
      - )
      commutative: false
avg:
  entity_type: sql_actions
```

```yaml
  metadata:
    aliases:
    - average
    - average of
    - avg
    - mean
    applicable_types:
      column:
      - numeric
    label_rules:
    - not id
    explanation: Returns the average of all values in a numeric column.
  binder:
    returns_type: numeric
    class: aggregate
    clause: select
    args:
    - name: column
      types:
      - numeric
    surfaces:
    - pattern:
      - AVG
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - average
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - average
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - avg
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - mean
      - (
      - '{column}'
      - )
      commutative: false
min:
  entity_type: sql_actions
```

```yaml
metadata:
  aliases:
  - bottom
  - least
  - lowest
  - min
  - minimum
  - smallest
  applicable_types:
    column:
    - numeric
    - text
    - date
    - timestamp
  label_rules: []
  explanation: Returns the minimum value of a column.
binder:
  returns_type: any
  class: aggregate
  clause: select
  args:
  - name: column
    types:
    - numeric
    - text
    - date
    - timestamp
  surfaces:
  - pattern:
    - MIN
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - bottom
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - least
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - lowest
    - (
    - '{column}'
    - )
    commutative: false
  - pattern:
    - min
```

```
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - minimum
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - smallest
      - (
      - '{column}'
      - )
      commutative: false
max:
  entity_type: sql_actions
  metadata:
    aliases:
    - greatest
    - highest
    - largest
    - max
    - maximum
    - most
    applicable_types:
      column:
      - numeric
      - text
      - date
      - timestamp
    label_rules: []
    explanation: Returns the maximum value of a column.
  binder:
    returns_type: any
    class: aggregate
    clause: select
    args:
    - name: column
      types:
      - numeric
      - text
      - date
      - timestamp
    surfaces:
    - pattern:
      - MAX
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - greatest
```

```
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - highest
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - largest
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - max
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - maximum
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - most
      - (
      - '{column}'
      - )
      commutative: false
distinct:
  entity_type: sql_actions
  metadata:
    aliases:
    - distinct
    - unique
    - unique values of
    applicable_types:
      column:
      - any
    label_rules:
    - not id
    explanation: Returns only the unique values in a column.
  binder:
    returns_type: any
    class: scalar
    clause: select
    args:
    - name: column
      types:
```

```
      - any
    surfaces:
    - pattern:
      - DISTINCT
      - '{column}'
      commutative: false
    - pattern:
      - distinct
      - '{column}'
      commutative: false
    - pattern:
      - unique
      - '{column}'
      commutative: false
    - pattern:
      - unique
      - values
      - of
      - '{column}'
      commutative: false
order_by_asc:
  entity_type: sql_actions
  metadata:
    aliases:
    - in ascending order
    - order by
    - order by ascending
    - order_by_asc
    - sort by
    - sort by ascending
    applicable_types:
      column:
      - any
    label_rules: []
    explanation: Sorts the results in ascending order based on a column.
  binder:
    returns_type: none
    class: ordering
    clause: order_by
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - ORDER
      - BY
      - '{column}'
      - ASC
      commutative: false
    - pattern:
      - in
      - ascending
      - order
```

```yaml
      - BY
      - '{column}'
      - ASC
      commutative: false
    - pattern:
      - order
      - by
      - BY
      - '{column}'
      - ASC
      commutative: false
    - pattern:
      - order
      - by
      - ascending
      - BY
      - '{column}'
      - ASC
      commutative: false
    - pattern:
      - order_by_asc
      - BY
      - '{column}'
      - ASC
      commutative: false
    - pattern:
      - sort
      - by
      - BY
      - '{column}'
      - ASC
      commutative: false
    - pattern:
      - sort
      - by
      - ascending
      - BY
      - '{column}'
      - ASC
      commutative: false
order_by_desc:
  entity_type: sql_actions
  metadata:
    aliases:
    - in descending order
    - order by descending
    - order_by_desc
    - sort by descending
    applicable_types:
      column:
      - any
    label_rules: []
    explanation: Sorts the results in descending order based on a column.
  binder:
```

```yaml
    returns_type: none
    class: ordering
    clause: order_by
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - ORDER
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - in
      - descending
      - order
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - order
      - by
      - descending
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - order_by_desc
      - BY
      - '{column}'
      - DESC
      commutative: false
    - pattern:
      - sort
      - by
      - descending
      - BY
      - '{column}'
      - DESC
      commutative: false
group_by:
  entity_type: sql_actions
  metadata:
    aliases:
    - group by
    - group_by
    - grouped by
    applicable_types:
      column:
      - any
```

```
    label_rules: []
    explanation: Groups rows that have the same values into summary rows.
  binder:
    returns_type: none
    class: grouping
    clause: group_by
    args:
    - name: column
      types:
      - any
    surfaces:
    - pattern:
      - GROUP
      - BY
      - '{column}'
      commutative: false
    - pattern:
      - group
      - by
      - BY
      - '{column}'
      commutative: false
    - pattern:
      - group_by
      - BY
      - '{column}'
      commutative: false
    - pattern:
      - grouped
      - by
      - BY
      - '{column}'
      commutative: false
having:
  entity_type: sql_actions
  metadata:
    aliases:
    - having
    - with
    applicable_types:
      condition:
      - any
    label_rules: []
    explanation: Filters groups based on a condition.
  binder:
    returns_type: boolean
    class: predicate
    clause: having
    args:
    - name: condition
      types:
      - any
    surfaces:
    - pattern:
```

```yaml
      - HAVING
      - '{condition}'
      commutative: false
    - pattern:
      - having
      - '{condition}'
      commutative: false
    - pattern:
      - with
      - '{condition}'
      commutative: false
limit:
  entity_type: sql_actions
  metadata:
    aliases:
    - first
    - limit
    - only
    - top
    applicable_types:
      value:
      - numeric
    label_rules: []
    explanation: Restricts the number of rows returned by the query.
  binder:
    returns_type: none
    class: limit
    clause: limit
    args:
    - name: value
      types:
      - numeric
    surfaces:
    - pattern:
      - LIMIT
      - '{value}'
      commutative: false
    - pattern:
      - first
      - '{value}'
      commutative: false
    - pattern:
      - limit
      - '{value}'
      commutative: false
    - pattern:
      - only
      - '{value}'
      commutative: false
    - pattern:
      - top
      - '{value}'
      commutative: false
extract:
```

```
  entity_type: sql_actions
  metadata:
    aliases:
    - extract
    - get the
    applicable_types:
      part:
      - any
      column:
      - date
      - timestamp
    label_rules: []
    explanation: Extracts a specific part (e.g., year, month) from a date or time
      value.
  binder:
    returns_type: numeric
    class: scalar
    clause: select
    args:
    - name: part
      types:
      - any
    - name: column
      types:
      - date
      - timestamp
    surfaces:
    - pattern:
      - '{part}'
      - from
      - '{column}'
      commutative: false
length:
  entity_type: sql_actions
  metadata:
    aliases:
    - character count
    - length
    - length in
    - length of
    - string length
    applicable_types:
      column:
      - text
    label_rules: []
    explanation: Returns the length of a string.
  binder:
    returns_type: numeric
    class: scalar
    clause: select
    args:
    - name: column
      types:
      - text
```

```yaml
    surfaces:
    - pattern:
      - LENGTH
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - character
      - count
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - length
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - length
      - in
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - length
      - of
      - (
      - '{column}'
      - )
      commutative: false
    - pattern:
      - string
      - length
      - (
      - '{column}'
      - )
      commutative: false
concat:
  entity_type: sql_actions
  metadata:
    aliases:
    - combine
    - concat
    - concatenate
    - join
    applicable_types:
      column1:
      - text
      column2:
      - text
```

```yaml
    label_rules: []
    explanation: Joins two or more strings together.
  binder:
    returns_type: text
    class: scalar
    clause: select
    args:
    - name: column1
      types:
      - text
    - name: column2
      types:
      - text
    surfaces:
    - pattern:
      - '{column1}'
      - and
      - '{column2}'
      commutative: false
cast:
  entity_type: sql_actions
  metadata:
    aliases:
    - cast
    - change type
    - convert
    applicable_types:
      column:
      - any
      to_type:
      - any
    label_rules: []
    explanation: Converts a value from one data type to another.
  binder:
    returns_type: any
    class: scalar
    clause: select
    args:
    - name: column
      types:
      - any
    - name: to_type
      types:
      - any
    surfaces:
    - pattern:
      - '{column}'
      - to
      - '{to_type}'
      commutative: false
st_perimeter:
  entity_type: postgis_actions
  metadata:
    aliases:
```

```
  - boundary length
  - length of boundary
  - outline length
  - perimeter
  - st_perimeter
  applicable_types:
    geom:
    - geometry_polygon
    - geography_polygon
  label_rules:
  - postgis
  explanation: Returns the perimeter (boundary length) of a polygonal geometry.
binder:
  returns_type: any
  class: scalar
  clause: select
  args:
  - name: geom
    types:
    - geometry_polygon
    - geography_polygon
  surfaces:
  - pattern:
    - ST_Perimeter
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - boundary
    - length
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - length
    - of
    - boundary
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - outline
    - length
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - perimeter
    - (
    - '{geom}'
```

```
      - )
      commutative: false
    - pattern:
      - st_perimeter
      - (
      - '{geom}'
      - )
      commutative: false
st_distance:
  entity_type: postgis_actions
  metadata:
    aliases:
    - distance
    - distance between
    - how far
    - separation of
    - st_distance
    applicable_types:
      geom1:
      - geometry_point
      - geography_point
      - geometry_linestring
      - geography_linestring
      - geometry_polygon
      - geography_polygon
      geom2:
      - geometry_point
      - geography_point
      - geometry_linestring
      - geography_linestring
      - geometry_polygon
      - geography_polygon
    label_rules:
    - postgis
    explanation: Calculates the shortest distance between two geometries or geographies.
  binder:
    returns_type: numeric
    class: spatial
    clause: select
    args:
    - name: geom1
      types:
      - geometry_point
      - geography_point
      - geometry_linestring
      - geography_linestring
      - geometry_polygon
      - geography_polygon
    - name: geom2
      types:
      - geometry_point
      - geography_point
      - geometry_linestring
      - geography_linestring
```

```
      - geometry_polygon
      - geography_polygon
    surfaces:
    - pattern:
      - of
      - '{geom1}'
      - from
      - '{geom2}'
      commutative: false
    - pattern:
      - distance
      - '{geom1}'
      - from
      - '{geom2}'
      commutative: false
    - pattern:
      - distance
      - between
      - '{geom1}'
      - from
      - '{geom2}'
      commutative: false
    - pattern:
      - how
      - far
      - '{geom1}'
      - from
      - '{geom2}'
      commutative: false
    - pattern:
      - separation
      - of
      - '{geom1}'
      - from
      - '{geom2}'
      commutative: false
    - pattern:
      - st_distance
      - '{geom1}'
      - from
      - '{geom2}'
      commutative: false
st_intersects:
  entity_type: postgis_actions
  metadata:
    aliases:
    - intersects
    - overlaps with
    - st_intersects
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_point
```

```yaml
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: Tests if two geometries or geographies intersect.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - with
      - '{geom2}'
      commutative: false
st_area:
  entity_type: postgis_actions
  metadata:
    aliases:
    - area
    - area of
    - size of
    - st_area
    - surface area
    applicable_types:
      geom:
      - geometry_polygon
      - geography_polygon
    label_rules:
    - postgis
    explanation: Calculates the area of a polygonal geometry.
  binder:
    returns_type: numeric
```

```
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry_polygon
      - geography_polygon
    surfaces:
    - pattern:
      - ST_Area
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - area
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - area
      - of
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - size
      - of
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - st_area
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - surface
      - area
      - (
      - '{geom}'
      - )
      commutative: false
st_length:
  entity_type: postgis_actions
  metadata:
    aliases:
    - distance along
    - distance of
    - length along
```

```
  - length of
  - line length
  - path length
  - st_length
  applicable_types:
    geom:
    - geometry_linestring
    - geography_linestring
  label_rules:
  - postgis
  explanation: Calculates the length of a linestring geometry.
binder:
  returns_type: numeric
  class: spatial
  clause: select
  args:
  - name: geom
    types:
    - geometry_linestring
    - geography_linestring
  surfaces:
  - pattern:
    - ST_Length
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - distance
    - along
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - distance
    - of
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - length
    - along
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - length
    - of
    - (
    - '{geom}'
    - )
```

```
      commutative: false
    - pattern:
      - line
      - length
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - path
      - length
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - st_length
      - (
      - '{geom}'
      - )
      commutative: false
st_x:
  entity_type: postgis_actions
  metadata:
    aliases:
    - lon
    - longitude
    - st_x
    - x coordinate
    - x pos
    applicable_types:
      point:
      - geometry_point
      - geography_point
    label_rules:
    - postgis
    explanation: Returns the X coordinate of a point geometry.
  binder:
    returns_type: numeric
    class: spatial
    clause: select
    args:
    - name: point
      types:
      - geometry_point
      - geography_point
    surfaces:
    - pattern:
      - ST_X
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
```

```yaml
      - lon
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - longitude
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - st_x
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - x
      - coordinate
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - x
      - pos
      - (
      - '{point}'
      - )
      commutative: false
st_y:
  entity_type: postgis_actions
  metadata:
    aliases:
    - lat
    - latitude
    - st_y
    - y coordinate
    - y pos
    applicable_types:
      point:
      - geometry_point
      - geography_point
    label_rules:
    - postgis
    explanation: Returns the Y coordinate of a point geometry.
  binder:
    returns_type: numeric
    class: spatial
    clause: select
    args:
    - name: point
      types:
```

```
      - geometry_point
      - geography_point
    surfaces:
    - pattern:
      - ST_Y
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - lat
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - latitude
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - st_y
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - y
      - coordinate
      - (
      - '{point}'
      - )
      commutative: false
    - pattern:
      - y
      - pos
      - (
      - '{point}'
      - )
      commutative: false
st_within:
  entity_type: postgis_actions
  metadata:
    aliases:
    - contained in
    - inside
    - is inside
    - is within
    - st_within
    - within
    applicable_types:
      geom1:
      - geometry
```

```
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: Tests if a geometry is entirely contained within another.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - in
      - '{geom2}'
      commutative: false
st_contains:
  entity_type: postgis_actions
  metadata:
    aliases:
    - contains
    - encloses
    - st_contains
    - surrounds
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_polygon
      geom2:
      - geometry
      - geography
```

```
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: Tests if a geometry contains another geometry entirely.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - and
      - '{geom2}'
      commutative: false
st_geometrytype:
  entity_type: postgis_actions
  metadata:
    aliases:
    - geometry type
    - shape type
    - st_geometrytype
    - type of geometry
    - what kind of shape
    applicable_types:
      geom:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: Returns the type of a geometry as a string.
  binder:
    returns_type: text
    class: spatial
    clause: select
    args:
    - name: geom
```

```
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - ST_GeometryType
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - geometry
      - type
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - shape
      - type
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - st_geometrytype
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - type
      - of
      - geometry
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - what
      - kind
      - of
      - shape
      - (
      - '{geom}'
      - )
      commutative: false
st_buffer:
  entity_type: postgis_actions
  metadata:
    aliases:
```

```
    - area around
    - buffer
    - buffer around
    - expand by
    - st_buffer
    applicable_types:
      geom:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      radius:
      - numeric
    label_rules:
    - postgis
    explanation: Creates a polygon that surrounds a geometry at a specified distance.
  binder:
    returns_type: geometry_polygon
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: radius
      types:
      - numeric
    surfaces:
    - pattern:
      - '{geom}'
      - by
      - '{radius}'
      commutative: false
st_union:
  entity_type: postgis_actions
  metadata:
    aliases:
    - combine
    - merge
    - st_union
    - union
    - union of
    applicable_types:
      geom_collection:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
```

```
    label_rules:
    - postgis
    explanation: Merges multiple geometries into a single geometry.
  binder:
    returns_type: geometry
    class: spatial
    clause: select
    args:
    - name: geom_collection
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - ST_Union
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - combine
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - merge
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - st_union
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - union
      - (
      - '{geom_collection}'
      - )
      commutative: false
    - pattern:
      - union
      - of
      - (
      - '{geom_collection}'
      - )
      commutative: false
st_centroid:
```

```yaml
entity_type: postgis_actions
metadata:
  aliases:
  - center
  - center point
  - centroid
  - geometric center
  - st_centroid
  applicable_types:
    geom:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  label_rules:
  - postgis
  explanation: Returns the geometric center of a geometry.
binder:
  returns_type: geometry_point
  class: spatial
  clause: select
  args:
  - name: geom
    types:
    - geometry
    - geography
    - geometry_point
    - geometry_linestring
    - geometry_polygon
  surfaces:
  - pattern:
    - ST_Centroid
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - center
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - center
    - point
    - (
    - '{geom}'
    - )
    commutative: false
  - pattern:
    - centroid
    - (
    - '{geom}'
```

```yaml
      - )
      commutative: false
    - pattern:
      - geometric
      - center
      - (
      - '{geom}'
      - )
      commutative: false
    - pattern:
      - st_centroid
      - (
      - '{geom}'
      - )
      commutative: false
st_simplify:
  entity_type: postgis_actions
  metadata:
    aliases:
    - generalize
    - simplify
    - simplify shape
    - st_simplify
    applicable_types:
      geom:
      - geometry_linestring
      - geometry_polygon
      - geography_linestring
      - geography_polygon
      tolerance:
      - numeric
    label_rules:
    - postgis
    explanation: Simplifies a geometry by reducing the number of vertices.
  binder:
    returns_type: any
    class: scalar
    clause: select
    args:
    - name: geom
      types:
      - geometry_linestring
      - geometry_polygon
      - geography_linestring
      - geography_polygon
    - name: tolerance
      types:
      - numeric
    surfaces:
    - pattern:
      - '{geom}'
      - by
      - '{tolerance}'
      commutative: false
```

```yaml
st_touches:
  entity_type: postgis_actions
  metadata:
    aliases:
    - borders
    - is adjacent to
    - st_touches
    - touches
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: Tests if two geometries touch only at their boundaries.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - and
      - '{geom2}'
      commutative: false
st_crosses:
  entity_type: postgis_actions
  metadata:
    aliases:
    - crosses
```

```yaml
    - goes across
    - st_crosses
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_linestring
    label_rules:
    - postgis
    explanation: Tests if two geometries cross each other.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_linestring
    surfaces:
    - pattern:
      - '{geom1}'
      - and
      - '{geom2}'
      commutative: false
st_spatial_index:
  entity_type: postgis_actions
  metadata:
    aliases:
    - spatial index
    - st_spatial_index
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
```

```yaml
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: A spatial index operator that checks for intersection.
  binder:
    returns_type: boolean
    class: predicate
    clause: where
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - with
      - '{geom2}'
      commutative: false
st_distance_operator:
  entity_type: postgis_actions
  metadata:
    aliases:
    - closest
    - closest to
    - nearest
    - st_distance_operator
    applicable_types:
      geom1:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      geom2:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    label_rules:
    - postgis
    explanation: A spatial operator that returns the distance between two geometries.
```

```yaml
  binder:
    returns_type: numeric
    class: spatial
    clause: order_by
    args:
    - name: geom1
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: geom2
      types:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    surfaces:
    - pattern:
      - '{geom1}'
      - to
      - '{geom2}'
      commutative: false
st_transform:
  entity_type: postgis_actions
  metadata:
    aliases:
    - convert coordinate system
    - reproject
    - st_transform
    - transform
    applicable_types:
      geom:
      - geometry
      - geography
      - geometry_point
      - geometry_linestring
      - geometry_polygon
      srid:
      - numeric
    label_rules:
    - postgis
    explanation: Transforms a geometry from one coordinate system to another.
  binder:
    returns_type: geometry
    class: spatial
    clause: select
    args:
    - name: geom
      types:
      - geometry
      - geography
```

```yaml
      - geometry_point
      - geometry_linestring
      - geometry_polygon
    - name: srid
      types:
      - numeric
    surfaces:
    - pattern:
      - '{geom}'
      - to
      - '{srid}'
      commutative: false
_binder_meta:
  entity_type: _meta
  metadata:
    connectors:
    - _on
    - and
    - at
    - belonging to
    - between
    - from
    - in
    - of
    - 'on'
    - to
    - with
_diagnostics:
  entity_type: _meta
  metadata:
    alias_collisions:
    - alias: amount
      meanings:
      - canonical: balance
        type: column
      - canonical: price
        type: column
    - alias: area
      meanings:
      - canonical: regions
        type: table
      - canonical: st_area
        type: postgis_actions
    - alias: combine
      meanings:
      - canonical: concat
        type: sql_actions
      - canonical: st_union
        type: postgis_actions
    - alias: length of
      meanings:
      - canonical: length
        type: sql_actions
      - canonical: st_length
```

```
    type: postgis_actions
- alias: perimeter
  meanings:
  - canonical: boundaries
    type: column
  - canonical: st_perimeter
    type: postgis_actions
prefix_collisions:
- alias: amount
  longer_keys:
  - amount sold
- alias: account
  longer_keys:
  - account balance
  - account name
  - account number
  - account status
- alias: client
  longer_keys:
  - client id
- alias: customer
  longer_keys:
  - customer id
- alias: user
  longer_keys:
  - user id
  - user identifier
  - user ids
  - user name
  - user names
  - user number
- alias: item
  longer_keys:
  - item count
  - item name
  - item sold
- alias: product
  longer_keys:
  - product name
  - product names
- alias: quantity
  longer_keys:
  - quantity of
- alias: order
  longer_keys:
  - order by
  - order by ascending
  - order by descending
  - order date
  - order id
  - order number
- alias: purchase
  longer_keys:
  - purchase date
```

```
    - purchase id
  - alias: sale
    longer_keys:
    - sale date
    - sale dates
    - sale id
    - sale ids
  - alias: transaction
    longer_keys:
    - transaction date
    - transaction id
  - alias: boundary
    longer_keys:
    - boundary length
  - alias: geometry
    longer_keys:
    - geometry type
  - alias: outline
    longer_keys:
    - outline length
  - alias: shape
    longer_keys:
    - shape type
  - alias: area
    longer_keys:
    - area around
    - area id
    - area of
    - area shape
  - alias: region
    longer_keys:
    - region id
    - region ids
    - region number
  - alias: zone
    longer_keys:
    - zone id
  - alias: get
    longer_keys:
    - get the
  - alias: what
    longer_keys:
    - what kind of shape
  - alias: at
    longer_keys:
    - at least
    - at most
  - alias: not
    longer_keys:
    - not less than
    - not more than
  - alias: equal
    longer_keys:
    - equal to
```

```yaml
- alias: is
  longer_keys:
  - is active
  - is actives
  - is adjacent to
  - is at least
  - is at most
  - is between
  - is empty
  - is exactly
  - is greater than
  - is greater than or equal to
  - is in
  - is inside
  - is less than
  - is less than or equal to
  - is missing
  - is not
  - is not empty
  - is not equal to
  - is not null
  - is null
  - is one of
  - is present
  - is the same as
  - is within
- alias: in
  longer_keys:
  - in ascending order
  - in descending order
  - in the range of
- alias: number
  longer_keys:
  - number of
  - number of items
  - number sold
- alias: sum
  longer_keys:
  - sum of
- alias: total
  longer_keys:
  - total amount of
  - total number of
  - total of
- alias: average
  longer_keys:
  - average of
- alias: most
  longer_keys:
  - most recent login
- alias: unique
  longer_keys:
  - unique values of
- alias: length
```

```
  longer_keys:
  - length along
  - length in
  - length of
  - length of boundary
- alias: convert
  longer_keys:
  - convert coordinate system
- alias: distance
  longer_keys:
  - distance along
  - distance between
  - distance of
- alias: buffer
  longer_keys:
  - buffer around
- alias: union
  longer_keys:
  - union of
- alias: center
  longer_keys:
  - center point
- alias: simplify
  longer_keys:
  - simplify shape
- alias: closest
  longer_keys:
  - closest to
surface_warnings: []
inferred_args: []
surfaces_args_mismatch:
- canonical: between
  args:
  - column
  - value1
  - value2
  placeholders:
  - column
  - value
  - value1
  - value2
- canonical: in
  args:
  - column
  - values
  placeholders:
  - column
  - value
  - values
- canonical: is_null
  args:
  - column
  placeholders:
  - column
```

```yaml
  - value
- canonical: is_not_null
  args:
  - column
  placeholders:
  - column
  - value
missing_applicable_types: []
connectors:
- _on
- and
- at
- belonging to
- between
- from
- in
- of
- 'on'
- to
- with
```

requirements.txt

```
# Core dependencies
lark
PyYAML
Shapely

# Database dependencies
sqlalchemy
pysqlite3

inflect
```

schema.yaml


```yaml
tables:
  users:
    aliases:
    - account
    - accounts
    - client
    - clients
    - customer
    - customers
    - people
    - user
    - users
    columns:
      user_id:
        aliases:
        - account number
        - client id
        - customer id
        - user id
        - user identifier
        - user ids
        - user number
        - user_id
        type: INTEGER
        labels:
        - id
      username:
        aliases:
        - account name
        - handle
        - login
        - user name
        - user names
        - username
        - usernames
        type: VARCHAR(50)
        labels: []
      age:
        aliases:
        - age
        - ages
        - years old
        type: INT
        labels: []
      balance:
        aliases:
        - account balance
        - amount
        - balance
        - balances
        - credit
```

```
      - funds
      type: DECIMAL(10, 2)
      labels: []
    is_active:
      aliases:
      - account status
      - active status
      - is active
      - is actives
      - is_active
      - status
      type: BOOLEAN
      labels: []
    last_login:
      aliases:
      - last access
      - last active date
      - last login
      - last logins
      - last signed in
      - last_login
      - most recent login
      type: TIMESTAMP
      labels: []
    location:
      aliases:
      - coordinates
      - geo location
      - geographic coordinates
      - location
      - locations
      - place
      - position
      type: geography_point
      labels:
      - postgis
sales:
  aliases:
  - order
  - orders
  - purchase
  - purchases
  - sale
  - sales
  - transaction
  - transactions
  columns:
    sale_id:
      aliases:
      - order id
      - order number
      - purchase id
      - sale id
      - sale ids
```

```
  - sale_id
  - transaction id
  type: INTEGER
  labels:
  - id
user_id:
  aliases:
  - account number
  - client id
  - customer id
  - user id
  - user identifier
  - user ids
  - user number
  - user_id
  type: INTEGER
  labels:
  - id
product_name:
  aliases:
  - item
  - item name
  - item sold
  - product
  - product name
  - product names
  - product_name
  type: TEXT
  labels: []
sale_date:
  aliases:
  - order date
  - purchase date
  - sale date
  - sale dates
  - sale_date
  - transaction date
  - when it was sold
  type: DATE
  labels: []
quantity:
  aliases:
  - amount sold
  - item count
  - number of items
  - number sold
  - quantities
  - quantity
  type: INT
  labels: []
price:
  aliases:
  - amount
  - charge
```

```yaml
      - cost
      - how much
      - price
      - prices
      - value
      type: FLOAT
      labels: []
regions:
  aliases:
  - area
  - areas
  - district
  - region
  - regions
  - territory
  - zone
  - zones
  columns:
    region_id:
      aliases:
      - area id
      - region id
      - region ids
      - region number
      - region_id
      - zone id
      type: INTEGER
      labels:
      - id
    name:
      aliases:
      - identifier
      - label
      - name
      - names
      - title
      type: VARCHAR(50)
      labels: []
    boundaries:
      aliases:
      - area shape
      - border
      - boundaries
      - boundary
      - geometry
      - outline
      - perimeter
      - shape
      type: geography_polygon
      labels:
      - postgis
```

sql_func_list.txt

```
## sql_action_templates
'=',
'!=',
'>',
'<',
'>=',
'<=',
'IN',
'IS NULL',
'IS NOT NULL',
'LIKE',
'COUNT',
'SUM',
'AVG',
'MIN',
'MAX',
'BETWEEN',
'ORDER BY ASC',
'ORDER BY DESC',
'GROUP BY',
'HAVING',
'DISTINCT',
'LIMIT',
'EXTRACT',
'LENGTH',
'CONCAT',
'CAST',

## postgis_action_templates
'ST_Distance',
'ST_Intersects',
'ST_Area',
'ST_Length',
'ST_X',
'ST_Y',
'ST_Within',
'ST_Contains',
'ST_GeometryType',
'&&',
'ST_Buffer',
'ST_Union',
'ST_Centroid',
'ST_Simplify',
'ST_Touches',
'ST_Crosses',
'<->',
'ST_Transform',
```

test.out

```
--- PHASE 1: GENERATING RELATIONSHIP GRAPH ---
Relationship graph successfully written to 'relationship_graph.yaml'.

--- PHASE 2: GENERATING LARK GRAMMAR ---
Lark grammar successfully written to 'generated_grammar.lark'.

--- PHASE 3: VALIDATING GRAMMAR WITH STATIC TESTS ---
--- Testing: 'show username and age of users' ---
query
  select_statement
    show
    None
    column_list
      column_name
      column_name
    from_clause
      None
      table_name

--- Testing: 'list username of users' ---
query
  select_statement
    list
    None
    column_list
      column_name
    from_clause
      None
      table_name

--- Testing: 'what age and username of users' ---
query
  select_statement
    what
    None
    column_list
      column_name
      column_name
    from_clause
      None
      table_name


--- PHASE 4: VALIDATING GRAMMAR WITH DYNAMIC TESTS ---

--- Testing dynamic phrases ---
--- Testing generated phrase: 'what me id of all views_geometry_columns_statistics' ---
query
  select_statement
    what
```

```
    me
    column_list
      column_name
    from_clause
      all
      table_name

--- Testing generated phrase: 'show datum of views_geometry_columns_statistics' ---
query
  select_statement
    show
    None
    column_list
      column_name
    from_clause
      None
      table_name

--- Testing generated phrase: 'what (the sql_statement , of for data_licenses' ---
!!! UNEXPECTED ERROR for 'what (the sql_statement , of for data_licenses': No terminal
matches '(' in the current parser context, at line 1 col 6

what (the sql_statement , of for data_license
     ^
Expected one of:
 * __ANON_16
 * EXTENT_MAX_Y
 * __ANON_43
 * __ANON_24
 * __ANON_47
 * VIEW_GEOMETRY
 * __ANON_20
 * READ_ONLY
 * VIRT_GEOMETRY
 * __ANON_37
 * USERNAME
 * __ANON_42
 * __ANON_49
 * SALE_DATE
 * __ANON_3
 * AXIS_1_ORIENTATION
 * AXIS_1_NAME
 * __ANON_33
 * __ANON_30
 * PRICE
 * QUANTITY
 * LAST_LOGIN
 * __ANON_50
 * ROW_COUNT
 * REGION_ID
 * __ANON_6
 * TIME_START
 * NAME
 * __ANON_4
```

* __ANON_23
* ITEM_NO
* LAST_UPDATE
* LAST_DELETE
* __ANON_18
* VIEW_ROWID
* __ANON_29
* __ANON_22
* NULL_VALUES
* SEARCH_FRAME
* DOUBLE_VALUES
* IS_ACTIVE
* FID
* REF_GEOMETRY
* TIME_END
* ORDINAL
* COLUMN_NAME
* PROJECTION
* BLOB_VALUES
* F_GEOMETRY_COLUMN
* GEOMETRY
* __ANON_48
* __ANON_35
* __ANON_19
* F_TABLE_NAME
* VIRT_NAME
* __ANON_9
* ORIGIN_ROWID
* UNIT
* __ANON_5
* AXIS_2_NAME
* __ANON_45
* DATUM
* PRODUCT_NAME
* __ANON_1
* DB_PREFIX
* __ANON_41
* DOUBLE_MAX
* __ANON_8
* SUCCESS
* MAX_ITEMS
* __ANON_14
* SPHEROID
* BOUNDARIES
* __ANON_28
* USER_ID
* __ANON_0
* EXTENT_MAX_X
* HAS_FLIPPED_AXES
* SALE_ID
* INTEGER_VALUES
* ID
* USER_AGENT
* IS_GEOGRAPHIC

```
 * INTEGER_MIN
 * __ANON_11
 * AXIS_2_ORIENTATION
 * __ANON_36
 * DISTANCE
 * __ANON_2
 * VIEW_NAME
 * ERROR_CAUSE
 * __ANON_51
 * __ANON_52
 * __ANON_13
 * __ANON_32
 * HIDDEN
 * __ANON_40
 * SQL_STATEMENT
 * SRID
 * TEXT_VALUES
 * __ANON_15
 * __ANON_26
 * __ANON_17
 * LOCATION
 * DOUBLE_MIN
 * URL
 * LAST_VERIFIED
 * __ANON_46
 * EXTENT_MIN_X
 * __ANON_10
 * __ANON_44
 * EXTENT_MIN_Y
 * AGE
 * __ANON_38
 * __ANON_34
 * MAX_SIZE
 * FILLER
 * __ANON_25
 * __ANON_7
 * INTEGER_MAX
 * LAST_INSERT
 * BALANCE
 * __ANON_31
 * __ANON_12
 * __ANON_21
 * PRIME_MERIDIAN
 * __ANON_39
 * POS
 * __ANON_27

--- Testing generated phrase: 'show (the last login of sql_statements_log' ---
!!! UNEXPECTED ERROR for 'show (the last login of sql_statements_log': No terminal matches '(' in the current parser context, at line 1 col 6

show (the last login of sql_statements_log
     ^
Expected one of:
```

* __ANON_16
* EXTENT_MAX_Y
* __ANON_43
* __ANON_24
* __ANON_47
* VIEW_GEOMETRY
* __ANON_20
* READ_ONLY
* VIRT_GEOMETRY
* __ANON_37
* USERNAME
* __ANON_42
* __ANON_49
* SALE_DATE
* __ANON_3
* AXIS_1_ORIENTATION
* AXIS_1_NAME
* __ANON_33
* __ANON_30
* PRICE
* QUANTITY
* LAST_LOGIN
* __ANON_50
* ROW_COUNT
* REGION_ID
* __ANON_6
* TIME_START
* NAME
* __ANON_4
* __ANON_23
* ITEM_NO
* LAST_UPDATE
* LAST_DELETE
* __ANON_18
* VIEW_ROWID
* __ANON_29
* __ANON_22
* NULL_VALUES
* SEARCH_FRAME
* DOUBLE_VALUES
* IS_ACTIVE
* FID
* REF_GEOMETRY
* TIME_END
* ORDINAL
* COLUMN_NAME
* PROJECTION
* BLOB_VALUES
* F_GEOMETRY_COLUMN
* GEOMETRY
* __ANON_48
* __ANON_35
* __ANON_19
* F_TABLE_NAME

* VIRT_NAME
* __ANON_9
* ORIGIN_ROWID
* UNIT
* __ANON_5
* AXIS_2_NAME
* __ANON_45
* DATUM
* PRODUCT_NAME
* __ANON_1
* DB_PREFIX
* __ANON_41
* DOUBLE_MAX
* __ANON_8
* SUCCESS
* MAX_ITEMS
* __ANON_14
* SPHEROID
* BOUNDARIES
* __ANON_28
* USER_ID
* __ANON_0
* EXTENT_MAX_X
* HAS_FLIPPED_AXES
* SALE_ID
* INTEGER_VALUES
* ID
* USER_AGENT
* IS_GEOGRAPHIC
* INTEGER_MIN
* __ANON_11
* AXIS_2_ORIENTATION
* __ANON_36
* DISTANCE
* __ANON_2
* VIEW_NAME
* ERROR_CAUSE
* __ANON_51
* __ANON_52
* __ANON_13
* __ANON_32
* HIDDEN
* __ANON_40
* SQL_STATEMENT
* SRID
* TEXT_VALUES
* __ANON_15
* __ANON_26
* __ANON_17
* LOCATION
* DOUBLE_MIN
* URL
* LAST_VERIFIED
* __ANON_46

```
 * EXTENT_MIN_X
 * __ANON_10
 * __ANON_44
 * EXTENT_MIN_Y
 * AGE
 * __ANON_38
 * __ANON_34
 * MAX_SIZE
 * FILLER
 * __ANON_25
 * __ANON_7
 * INTEGER_MAX
 * LAST_INSERT
 * BALANCE
 * __ANON_31
 * __ANON_12
 * __ANON_21
 * PRIME_MERIDIAN
 * __ANON_39
 * POS
 * __ANON_27

--- Testing generated phrase: 'list null_values of data_licenses' ---
query
  select_statement
    list
    None
    column_list
      column_name
    from_clause
      None
      table_name

--- Testing generated phrase: 'list all unit of virts_geometry_columns_field_infos' ---
query
  select_statement
    list
    all
    column_list
      column_name
    from_clause
      None
      table_name

--- Testing generated phrase: 'get srid of (the ElementaryGeometries' ---
!!! UNEXPECTED ERROR for 'get srid of (the ElementaryGeometries': No terminal matches
'(' in the current parser context, at line 1 col 13

get srid of (the ElementaryGeometries
            ^
Expected one of:
 * GEOMETRY_COLUMNS_STATISTICS
 * SPATIALINDEX
 * VIEWS_GEOMETRY_COLUMNS
```

```
 * SQL_STATEMENTS_LOG
 * GEOMETRY_COLUMNS_AUTH
 * VIRTS_GEOMETRY_COLUMNS_AUTH
 * VIEWS_GEOMETRY_COLUMNS_AUTH
 * USERS
 * DATA_LICENSES
 * ELEMENTARYGEOMETRIES
 * SPATIAL_REF_SYS_AUX
 * GEOMETRY_COLUMNS_FIELD_INFOS
 * FILLER
 * SALES
 * VIRTS_GEOMETRY_COLUMNS_FIELD_INFOS
 * GEOMETRY_COLUMNS_TIME
 * VIRTS_GEOMETRY_COLUMNS_STATISTICS
 * REGIONS
 * VIEWS_GEOMETRY_COLUMNS_FIELD_INFOS
 * KNN
 * VIEWS_GEOMETRY_COLUMNS_STATISTICS

--- Testing generated phrase: 'show text values and of views_geometry_columns_field_infos' ---
!!! UNEXPECTED ERROR for 'show text values and of views_geometry_columns_field_infos': No terminal matches 'o' in the current parser context, at line 1 col 22

show text values and of views_geometry_columns_field_infos
                     ^
Expected one of:
 * __ANON_16
 * EXTENT_MAX_Y
 * __ANON_43
 * __ANON_24
 * __ANON_47
 * VIEW_GEOMETRY
 * __ANON_20
 * READ_ONLY
 * VIRT_GEOMETRY
 * __ANON_37
 * USERNAME
 * __ANON_42
 * __ANON_49
 * SALE_DATE
 * __ANON_3
 * AXIS_1_ORIENTATION
 * AXIS_1_NAME
 * __ANON_33
 * __ANON_30
 * PRICE
 * QUANTITY
 * LAST_LOGIN
 * __ANON_50
 * ROW_COUNT
 * REGION_ID
 * __ANON_6
 * TIME_START
```

* NAME
* __ANON_4
* __ANON_23
* ITEM_NO
* LAST_UPDATE
* LAST_DELETE
* __ANON_18
* VIEW_ROWID
* __ANON_29
* __ANON_22
* NULL_VALUES
* SEARCH_FRAME
* DOUBLE_VALUES
* IS_ACTIVE
* FID
* REF_GEOMETRY
* TIME_END
* ORDINAL
* COLUMN_NAME
* PROJECTION
* BLOB_VALUES
* F_GEOMETRY_COLUMN
* GEOMETRY
* __ANON_48
* __ANON_35
* __ANON_19
* F_TABLE_NAME
* VIRT_NAME
* __ANON_9
* ORIGIN_ROWID
* UNIT
* __ANON_5
* AXIS_2_NAME
* __ANON_45
* DATUM
* PRODUCT_NAME
* __ANON_1
* DB_PREFIX
* __ANON_41
* DOUBLE_MAX
* __ANON_8
* SUCCESS
* MAX_ITEMS
* __ANON_14
* SPHEROID
* BOUNDARIES
* __ANON_28
* USER_ID
* __ANON_0
* EXTENT_MAX_X
* HAS_FLIPPED_AXES
* SALE_ID
* INTEGER_VALUES
* ID

```
 * USER_AGENT
 * IS_GEOGRAPHIC
 * INTEGER_MIN
 * __ANON_11
 * AXIS_2_ORIENTATION
 * __ANON_36
 * DISTANCE
 * __ANON_2
 * VIEW_NAME
 * ERROR_CAUSE
 * __ANON_51
 * __ANON_52
 * __ANON_13
 * __ANON_32
 * HIDDEN
 * __ANON_40
 * SQL_STATEMENT
 * SRID
 * TEXT_VALUES
 * __ANON_15
 * __ANON_26
 * __ANON_17
 * LOCATION
 * DOUBLE_MIN
 * URL
 * LAST_VERIFIED
 * __ANON_46
 * EXTENT_MIN_X
 * __ANON_10
 * __ANON_44
 * EXTENT_MIN_Y
 * AGE
 * __ANON_38
 * __ANON_34
 * MAX_SIZE
 * __ANON_25
 * __ANON_7
 * INTEGER_MAX
 * LAST_INSERT
 * BALANCE
 * __ANON_31
 * __ANON_12
 * __ANON_21
 * PRIME_MERIDIAN
 * __ANON_39
 * POS
 * __ANON_27

--- Testing generated phrase: 'show (the integer_max of for data_licenses' ---
!!! UNEXPECTED ERROR for 'show (the integer_max of for data_licenses': No terminal matches '(' in the current parser context, at line 1 col 6

show (the integer_max of for data_licenses
     ^
```

```
Expected one of:
 * __ANON_16
 * EXTENT_MAX_Y
 * __ANON_43
 * __ANON_24
 * __ANON_47
 * VIEW_GEOMETRY
 * __ANON_20
 * READ_ONLY
 * VIRT_GEOMETRY
 * __ANON_37
 * USERNAME
 * __ANON_42
 * __ANON_49
 * SALE_DATE
 * __ANON_3
 * AXIS_1_ORIENTATION
 * AXIS_1_NAME
 * __ANON_33
 * __ANON_30
 * PRICE
 * QUANTITY
 * LAST_LOGIN
 * __ANON_50
 * ROW_COUNT
 * REGION_ID
 * __ANON_6
 * TIME_START
 * NAME
 * __ANON_4
 * __ANON_23
 * ITEM_NO
 * LAST_UPDATE
 * LAST_DELETE
 * __ANON_18
 * VIEW_ROWID
 * __ANON_29
 * __ANON_22
 * NULL_VALUES
 * SEARCH_FRAME
 * DOUBLE_VALUES
 * IS_ACTIVE
 * FID
 * REF_GEOMETRY
 * TIME_END
 * ORDINAL
 * COLUMN_NAME
 * PROJECTION
 * BLOB_VALUES
 * F_GEOMETRY_COLUMN
 * GEOMETRY
 * __ANON_48
 * __ANON_35
 * __ANON_19
```

* F_TABLE_NAME
* VIRT_NAME
* __ANON_9
* ORIGIN_ROWID
* UNIT
* __ANON_5
* AXIS_2_NAME
* __ANON_45
* DATUM
* PRODUCT_NAME
* __ANON_1
* DB_PREFIX
* __ANON_41
* DOUBLE_MAX
* __ANON_8
* SUCCESS
* MAX_ITEMS
* __ANON_14
* SPHEROID
* BOUNDARIES
* __ANON_28
* USER_ID
* __ANON_0
* EXTENT_MAX_X
* HAS_FLIPPED_AXES
* SALE_ID
* INTEGER_VALUES
* ID
* USER_AGENT
* IS_GEOGRAPHIC
* INTEGER_MIN
* __ANON_11
* AXIS_2_ORIENTATION
* __ANON_36
* DISTANCE
* __ANON_2
* VIEW_NAME
* ERROR_CAUSE
* __ANON_51
* __ANON_52
* __ANON_13
* __ANON_32
* HIDDEN
* __ANON_40
* SQL_STATEMENT
* SRID
* TEXT_VALUES
* __ANON_15
* __ANON_26
* __ANON_17
* LOCATION
* DOUBLE_MIN
* URL
* LAST_VERIFIED

```
 * __ANON_46
 * EXTENT_MIN_X
 * __ANON_10
 * __ANON_44
 * EXTENT_MIN_Y
 * AGE
 * __ANON_38
 * __ANON_34
 * MAX_SIZE
 * FILLER
 * __ANON_25
 * __ANON_7
 * INTEGER_MAX
 * LAST_INSERT
 * BALANCE
 * __ANON_31
 * __ANON_12
 * __ANON_21
 * PRIME_MERIDIAN
 * __ANON_39
 * POS
 * __ANON_27

--- Testing generated phrase: 'show (the max size of geometry_columns_time' ---
!!! UNEXPECTED ERROR for 'show (the max size of geometry_columns_time': No terminal matches '(' in the current parser context, at line 1 col 6

show (the max size of geometry_columns_time
     ^
Expected one of:
 * __ANON_16
 * EXTENT_MAX_Y
 * __ANON_43
 * __ANON_24
 * __ANON_47
 * VIEW_GEOMETRY
 * __ANON_20
 * READ_ONLY
 * VIRT_GEOMETRY
 * __ANON_37
 * USERNAME
 * __ANON_42
 * __ANON_49
 * SALE_DATE
 * __ANON_3
 * AXIS_1_ORIENTATION
 * AXIS_1_NAME
 * __ANON_33
 * __ANON_30
 * PRICE
 * QUANTITY
 * LAST_LOGIN
 * __ANON_50
 * ROW_COUNT
```

* REGION_ID
* __ANON_6
* TIME_START
* NAME
* __ANON_4
* __ANON_23
* ITEM_NO
* LAST_UPDATE
* LAST_DELETE
* __ANON_18
* VIEW_ROWID
* __ANON_29
* __ANON_22
* NULL_VALUES
* SEARCH_FRAME
* DOUBLE_VALUES
* IS_ACTIVE
* FID
* REF_GEOMETRY
* TIME_END
* ORDINAL
* COLUMN_NAME
* PROJECTION
* BLOB_VALUES
* F_GEOMETRY_COLUMN
* GEOMETRY
* __ANON_48
* __ANON_35
* __ANON_19
* F_TABLE_NAME
* VIRT_NAME
* __ANON_9
* ORIGIN_ROWID
* UNIT
* __ANON_5
* AXIS_2_NAME
* __ANON_45
* DATUM
* PRODUCT_NAME
* __ANON_1
* DB_PREFIX
* __ANON_41
* DOUBLE_MAX
* __ANON_8
* SUCCESS
* MAX_ITEMS
* __ANON_14
* SPHEROID
* BOUNDARIES
* __ANON_28
* USER_ID
* __ANON_0
* EXTENT_MAX_X
* HAS_FLIPPED_AXES

* SALE_ID
* INTEGER_VALUES
* ID
* USER_AGENT
* IS_GEOGRAPHIC
* INTEGER_MIN
* __ANON_11
* AXIS_2_ORIENTATION
* __ANON_36
* DISTANCE
* __ANON_2
* VIEW_NAME
* ERROR_CAUSE
* __ANON_51
* __ANON_52
* __ANON_13
* __ANON_32
* HIDDEN
* __ANON_40
* SQL_STATEMENT
* SRID
* TEXT_VALUES
* __ANON_15
* __ANON_26
* __ANON_17
* LOCATION
* DOUBLE_MIN
* URL
* LAST_VERIFIED
* __ANON_46
* EXTENT_MIN_X
* __ANON_10
* __ANON_44
* EXTENT_MIN_Y
* AGE
* __ANON_38
* __ANON_34
* MAX_SIZE
* FILLER
* __ANON_25
* __ANON_7
* INTEGER_MAX
* LAST_INSERT
* BALANCE
* __ANON_31
* __ANON_12
* __ANON_21
* PRIME_MERIDIAN
* __ANON_39
* POS
* __ANON_27

test_phrases.txt

```
test_phrases = [
    # --- Simple Queries (Columns Only) ---
    "show username from users",
    "list age, balance from users",
    "get product_name and quantity from sales",
    "what name of regions",
    "show sale_date from sales",
    "list username and is active from users",
    "get price, quantity, sale_date from sales",
    "what boundaries from regions",
    "list is geographic from spatial_ref_sys_aux",
    "show location from users",

    # --- Queries with Fillers ---
    "show me the username and age of users",
    "list all price from the sales",
    "get for me the name of regions",
    "what is the balance for all users",
    "show the location, username, age from users",
    "list the product_name and the price from sales",
    "get all boundaries of the regions",
    "show me sale_date of sales",
    "list for me all is active users",
    "what is the age of all users",

    # --- Simple Function Queries ---
    "get average of price from sales",
    "show sum of balance from users",
    "list total of quantity from sales",
    "what is the minimum of age from users",
    "show maximum of price from sales",
    "get count of username from users",
    "list distinct of region_id from sales",
    "show area of boundaries from regions",
    "what is the length of name from regions",
    "get centroid of location from users",

    # --- Mixed Columns and Functions ---
    "list username and average of age from users",
    "show name, area of boundaries from regions",
    "get product_name and total of price from sales",
    "what age, balance, and sum of balance from users",
    "list sale_date, count of product_name from sales",
    "show me the username and length of username from users",
    "get region_id, minimum of price from sales",
    "what name and centroid of boundaries from regions",
    "list all price, quantity, and average of price from sales",
    "show is active and count of user_id from users",

    # --- Complex Lists (multiple separators) ---
    "show username, age, balance and is active from users",
```

```
        "list product_name, quantity, and sale_date from sales",
        "get name, boundaries from regions",
        "what username, average of age, and maximum of balance of users",
        "show sale_date, price, and sum of price from sales",
        "list age, location and centroid of location from users",
        "get username, balance, is active, and last login from users",
        "show product_name, sale_date, and count of sale_id from sales",
        "what is the name, area of boundaries, length of name from regions",
        "list price, average of price, minimum of price, and maximum of price from
sales",

        # --- Using Multi-Word Aliases and Columns ---
        "list sort by descending of age from users",
        "show order by ascending of sale_date from sales",
        "get the x coordinate of location from users",
        "what is the y coordinate of location from users",
        "show how far of location from location from users", # Note: uses same column
twice
        "list group by of region_id from sales",
        "get the geometry type of boundaries from regions",
        "show me the last login from users",
        "what is the axis 1 name of spatial_ref_sys_aux",
        "list has flipped axes from spatial_ref_sys_aux",

        # --- More Variations and Edge Cases ---
        "get limit of price from sales",
        "show top of age from users",
        "list cast of balance from users",
        "what touches of boundaries from regions",

        "show average of age, sum of balance of users",
        "list minimum of price and maximum of price from sales",
        "get count of name, area of boundaries from regions",
        "what distinct of product_name from sales",
        "show length of username from users",
        "list total of quantity and average of price from sales",
        "get x coordinate of location, y coordinate of location from users",
        "what intersects of boundaries from regions",
        "show buffer of location from users",
        "get simplify of boundaries from regions",
        "list union of boundaries from regions",
        "what crosses of boundaries from regions",
        "show the spatial index of location from users",
        "get the closest of location from users",
        "what transform of boundaries from regions",
        "list the id, username, age, balance, is active, last login, location from
users",

        "show me count of sale_id from sales",
        "list for all users the average of their age",
        "get the total price from all sales",
        "what is the area of all regions",
        "show distinct of sale_date from sales",
        "get the length of product_name from sales",
```

```
    "list x coordinate of location from users",
    "what is the geometry type of location from users",
    "show the centroid of boundaries from regions",
    "get sum of quantity, average of price from sales",
    "list the username, and the count of user_id from users",
    "what is the minimum age from users",
    "show maximum balance of all users",
    "get average price from the sales table",
    "list me the names of all regions",
    "what is the total quantity sold from sales",
    "show me the distinct regions from sales"
]
```

src/wcm_nlq2sql/nlq_to_sql.lark

```
// --- The definitive, non-ambiguous grammar ---

start: select_statement

select_statement: SELECT_VERB [FILLER] columns_clause from_clause

columns_clause: "the"? column_list
column_list: column_name (("and" | ",") column_name)*

from_clause: "of" [FILLER] table_name

// Rules to give context to a WORD
column_name: WORD
table_name: WORD

// --- Directives and Terminals ---
%import common.WORD
%import common.WS

%ignore WS

// Define terminals for specific keywords
SELECT_VERB: "show" | "list" | "what"
FILLER: /(me|the|all|for)\s*/i
```

src/wcm_nlq2sql/nlq_to_sql.py

```python
import yaml
from lark import Lark, Transformer, v_args
import sqlite3
from .query_builder import QueryBuilder
from .sql_transformer import SQLTransformer

# Load the schema
def load_schema(schema_path):
    with open(schema_path, 'r') as f:
        return yaml.safe_load(f)

# The main function to process a query
def process_query(nl_query, schema, db_path):

    # 1. Initialize the QueryBuilder with the schema
    builder = QueryBuilder(schema)

    # 2. Load the grammar and parser
    with open("/app/src/wcm_nlq2sql/nlq_to_sql.lark", 'r') as f:
        grammar = f.read()

    # Use the simplified grammar
    parser = Lark(grammar, start='select_statement', parser='earley')

    print(f"\n--- DEBUGGING PARSE PROCESS ---")
    print(f"Input Query: '{nl_query}'")

    try:
        # Step 1: Tokenize the query (just for debugging)
        tokens = list(parser.lex(nl_query))
        print(f"Tokenized Input: {tokens}")

        # Step 2: Parse the tokens into a tree
        parse_tree = parser.parse(nl_query)
        print("\n--- PARSE TREE ---")
        print(parse_tree.pretty())

    except Exception as e:
        print(f"Error parsing query: {e}")
        return None

    # Step 3: Transform the tree into an SQL statement
    print("--- TRANSFORMER ACTIONS ---")
    transformer = SQLTransformer(builder)
    transformer.transform(parse_tree)

    sql_query = builder.build()

    print("\n--- FINAL OUTPUT ---")
    print(f"Generated SQL: {sql_query}")
```

```python
    try:
        conn = sqlite3.connect(db_path)
        cursor = conn.cursor()
        cursor.execute(sql_query)
        results = cursor.fetchall()
        conn.close()
        return results
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        return None


# --- Main demonstration loop ---
if __name__ == "__main__":
    SCHEMA_PATH = "/app/src/wcm_nlq2sql/schema/schema.yaml"
    DB_PATH = "test.db"

    schema = load_schema(SCHEMA_PATH)
    query = "show me the username and age of all users"

    print(f"\nProcessing NL Query: '{query}'")
    results = process_query(query, schema, DB_PATH)
    if results:
        print("Query Results:")
        for row in results:
            print(row)
    # --- Example Queries ---
    # queries = [
    #     "show me the username and age of all users",
    #     "show me all users where age is greater than 30",
    #     "what is the total sales amount?",
    #     "sort users by age descending",
    #     "show me the count of users",
    # ]

    # for query in queries:
    #     print(f"\nProcessing NL Query: '{query}'")
    #     results = process_query(query, schema, DB_PATH)
    #     if results:
    #         print("Query Results:")
    #         for row in results:
    #             print(row)
```

src/wcm_nlq2sql/old_nlq_to_sql.lark

```
// --- Grammar for a Natural Language to SQL Parser ---

start: query

// The main query structure
query: select_clause [where_clause] [group_clause] [having_clause] [order_clause] [limit_clause]

// The SELECT clause
select_clause: ( "show" | "list" | "what" ) [FILLER] (column_list | all_fields)
column_list: field_or_function_with_alias ( ("," | "and") [FILLER] field_or_function_with_alias)*
all_fields: ("all" | "*") [FILLER] table_name?

// This rule represents a single item in a SELECT clause
field_or_function_with_alias: (function_on_table_clause | function_on_field_clause | _field) [alias]
simple_field: _field [FILLER] table_name?

// A new rule for functions that operate on a field/column
function_on_field_clause: (aggregate_function | postgis_function) [FILLER] _field

// The WHERE clause
where_clause: ("where" | "for" | "with") [FILLER] condition_list
condition_list: condition ( ("and" | "or") [FILLER] condition)*

// This is the correct condition rule
condition: field_or_function_with_alias operator value
         | field_or_function_with_alias ("is" | "are") ("null" | "not null")
         | field_or_function_with_alias "between" value "and" value
         | postgis_condition
         | named_condition

// PostGIS specific conditions
postgis_condition: postgis_function "(" _field "," (coordinate_literal | _field) ")" (operator value)?
postgis_function: "distance" | "intersects" | "within" | "touches"

// Group by and Having clauses
group_clause: ("group" | "group by") [FILLER] (_field ("," _field)*)
having_clause: ("having") [FILLER] (condition_list)

// Order by clause
order_clause: ("sort" | "order") [FILLER] [table_name] "by" [FILLER] (order_field ("," [FILLER] order_field)*)
order_field: _field ("ascending" | "asc" | "descending" | "desc")?

// Limit clause
limit_clause: ("limit" | "top") [FILLER] INT

// --- Field, function, and value representations ---
```

```
// This is the key change for consistency
_field: WORD
table_name: WORD
amount_field: "amount" | "price" | "sales" | "total"
alias: ("as") WORD
value: NUMBER | QUOTED_STRING | coordinate_literal
coordinate_literal: "(" NUMBER "," NUMBER ")"

// Aliases for functions
function_on_table_clause: ("count" | "number") [FILLER] table_name
aggregate_function: "count" | "sum" | "average" | "maximum" | "minimum" | "total"

// --- Operators and keywords ---
operator: ("is" | "=") | ("is not" | "!=") | (">" | "greater than") | ("<" | "less
than")
        | (">=" | "greater than or equal to") | ("<=" | "less than or equal to")
        | ("like" | "contains") | "in"

// Named conditions for easier parsing of common phrases
named_condition: "last" INT "days"
               | "past" INT "weeks"
               | "active"

// --- Lark Terminals ---
%import common.WORD
%import common.INT
%import common.NUMBER
%import common.WS

%ignore WS

// Manual definition of a quoted string
QUOTED_STRING: /"([^"]|\\")*"/ | /'([^']|\\')*'/

// New filler rule to allow for more flexible phrasing
FILLER: /((me|the|that|from|for|by|is|of|with|a|an)\s*)+/i
```

src/wcm_nlq2sql/query_builder.py

```
import yaml
from collections import defaultdict

class QueryBuilder:
    def __init__(self, schema_yaml):
        """
        Initializes the QueryBuilder with the database schema.
        :param schema_yaml: A YAML string or dictionary representing the schema.
        """
        if isinstance(schema_yaml, str):
            self.schema = yaml.safe_load(schema_yaml)
        else:
            self.schema = schema_yaml

        self.tables = self.schema.get('tables', {})
        self.columns = {}
        for table_name, table_info in self.tables.items():
            for column_name, column_info in table_info.get('columns', {}).items():
                self.columns[column_name] = {'table': table_name, **column_info}

        self.sql_select = []
        self.sql_from = set()
        self.sql_where = []
        self.sql_group_by = []
        self.sql_having = []
        self.sql_order_by = []
        self.sql_limit = None

    def add_select(self, field, alias=None, function=None):
        if function:
            if alias:
                self.sql_select.append(f"{function}({field}) AS {alias}")
            else:
                self.sql_select.append(f"{function}({field})")
        else:
            if alias:
                self.sql_select.append(f"{field} AS {alias}")
            else:
                self.sql_select.append(field)

    def add_where(self, condition):
        """
        Adds a condition to the WHERE clause.
        :param condition: The SQL condition string.
        """
        self.sql_where.append(condition)

    def add_group_by(self, field):
        """
        Adds a field to the GROUP BY clause.
        :param field: The name of the column.
```

```python
        """
        self.sql_group_by.append(field)
        if field in self.columns:
            self.sql_from.add(self.columns[field]['table'])

    def add_having(self, condition):
        """
        Adds a condition to the HAVING clause.
        :param condition: The SQL condition string.
        """
        self.sql_having.append(condition)

    def add_order_by(self, field, direction='ASC'):
        """
        Adds a field to the ORDER BY clause.
        :param field: The name of the column.
        :param direction: 'ASC' or 'DESC'.
        """
        self.sql_order_by.append(f"{field} {direction}")
        if field in self.columns:
            self.sql_from.add(self.columns[field]['table'])

    def add_limit(self, value):
        """
        Adds a LIMIT clause.
        :param value: The integer value for the limit.
        """
        self.sql_limit = value

    def add_from(self, table_name):
        print(f"DEBUG: QueryBuilder.add_from('{table_name}') called.")
        self.sql_from.add(table_name)

    def build(self):
        query_parts = []

        if not self.sql_select:
            self.sql_select.append('*')

        query_parts.append(f"SELECT {', '.join(self.sql_select)}")

        if self.sql_from:  # Crucial change here: check if the set is not empty
            query_parts.append(f"FROM {', '.join(self.sql_from)}")

        if self.sql_where:
            query_parts.append(f"WHERE {' AND '.join(self.sql_where)}")

        if self.sql_group_by:
            query_parts.append(f"GROUP BY {', '.join(self.sql_group_by)}")

        if self.sql_having:
            query_parts.append(f"HAVING {' AND '.join(self.sql_having)}")

        if self.sql_order_by:
```

```
            query_parts.append(f"ORDER BY {', '.join(self.sql_order_by)}")

        if self.sql_limit is not None:
            query_parts.append(f"LIMIT {self.sql_limit}")

        return ' '.join(query_parts)

    def get_column_info(self, column_name):
        """
        Retrieves information about a column from the schema.
        :param column_name: The name of the column.
        :return: A dictionary of column info, or None if not found.
        """
        return self.columns.get(column_name)
```

src/wcm_nlq2sql/sql_build_helper.py

```
def get_relevant_functions(column_info, all_sql_funcs, all_postgis_funcs):
    """
    Determines which SQL and PostGIS functions are relevant for a given column.
    :param column_info: The dictionary of a column's schema info.
    :param all_sql_funcs: A list of all available SQL functions.
    :param all_postgis_funcs: A list of all available PostGIS functions.
    :return: A dictionary of relevant functions.
    """
    relevant_funcs = defaultdict(list)
    column_type = column_info.get('type', '').lower()
    metadata = column_info.get('metadata', [])

    # Example logic for numeric types
    if column_type in ('integer', 'numeric', 'float', 'decimal'):
        for func in ['=', '!=', '>', '<', '>=', '<=', 'BETWEEN', 'COUNT', 'SUM', 'AVG',
'MIN', 'MAX']:
            relevant_funcs['sql'].append(func)

    # Example logic for string types
    if column_type in ('text', 'varchar', 'char'):
        for func in ['=', '!=', 'LIKE', 'LENGTH', 'CONCAT']:
            relevant_funcs['sql'].append(func)

    # Example logic for PostGIS types (using metadata)
    if 'geometry' in metadata or 'latlon' in metadata:
        for func in ['ST_Distance', 'ST_Intersects', 'ST_Area', 'ST_Within', 'ST_X',
'ST_Y']:
            relevant_funcs['postgis'].append(func)

    # Extend this logic for all types and functions
    # ...

    return relevant_funcs
```

src/wcm_nlq2sql/sql_transformer.py

```python
from lark import Transformer, v_args, Token

class SQLTransformer(Transformer):
    def __init__(self, builder):
        self.builder = builder

    def select_statement(self, children):
        # Children is a list of the results from the child rules
        columns_results = children[2]  # The transformed result of columns_clause
        from_results = children[3]    # The transformed result of from_clause

        # Add the columns to the builder
        for col_name in columns_results:
            self.builder.add_select(col_name)

        # Add the table name to the builder
        self.builder.add_from(from_results)

    def columns_clause(self, children):
        # We need to find the column_list and return its children
        for child in children:
            if isinstance(child, list):
                return child

        # This will handle cases where the rule might return a single item
        if len(children) == 1 and isinstance(children[0], str):
            return children
        return [] # Return an empty list if no columns are found

    def column_list(self, children):
        # The children are now a list of the transformed column_name strings.
        # We just need to return this list.
        return children

    def from_clause(self, children):
        # The last child is the table name, which is now a string.
        return children[-1]

    def column_name(self, children):
        # The children of a column_name rule is a single WORD token.
        return str(children[0])

    def table_name(self, children):
        # The children of a table_name rule is a single WORD token.
        return str(children[0])

    def WORD(self, token):
        # The WORD token's value is what we need.
        return str(token)

    def __default__(self, data, children, meta):
```

```
return children
```

src/wcm_nlq2sql/schema/schema.yaml


```yaml
tables:
  users:
    columns:
      user_id:
        type: INT
        metadata: ['id']
      username:
        type: VARCHAR
      age:
        type: INT
      balance:
        type: DECIMAL
      is_active:
        type: BOOLEAN
      last_login:
        type: TIMESTAMP
      location:
        type: GEOMETRY
        metadata: ['postgis', 'geometry(point)']
  sales:
    columns:
      sale_id:
        type: INT
        metadata: ['id']
      user_id:
        type: INT
      product_name:
        type: TEXT
      sale_date:
        type: DATE
      quantity:
        type: INT
      price:
        type: FLOAT
  regions:
    columns:
      region_id:
        type: INT
        metadata: ['id']
      name:
        type: VARCHAR
      boundaries:
        type: GEOMETRY
        metadata: ['postgis', 'geometry(polygon)']
```

src/scripts/generate_db.py

```
import sqlite3
import os
import random
from datetime import datetime, timedelta
from shapely.geometry import Point, LineString, Polygon
from shapely.wkb import dumps, loads

DB_NAME = 'test.db'
SCHEMA_FILE                  =                   os.path.join(os.path.dirname(__file__),
'../natural_language_sql/schema/schema.yaml')

# Helper function to create geometry objects and convert to WKB
def create_wkb(geom_type, coords):
    if geom_type == 'POINT':
        geom = Point(coords)
    elif geom_type == 'LINESTRING':
        geom = LineString(coords)
    elif geom_type == 'POLYGON':
        geom = Polygon(coords)
    else:
        return None
    return dumps(geom, hex=True)

def generate_dummy_data(cursor):
    # This SQL now uses SpatiaLite's `AddGeometryColumn` function
    # instead of trying to define the column type in the CREATE TABLE statement.
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS users (
            user_id INTEGER PRIMARY KEY,
            username VARCHAR(50) NOT NULL,
            age INT,
            balance DECIMAL(10, 2),
            is_active BOOLEAN,
            last_login TIMESTAMP
        );
    """)
    # Add the GEOMETRY column using SpatiaLite's function
    cursor.execute("SELECT AddGeometryColumn('users', 'location', 4326, 'POINT', 2);")

    users_data = []
    for i in range(1, 11):
        username = f"user_{i}"
        age = random.randint(20, 60)
        balance = random.uniform(100.0, 1000.0)
        is_active = random.choice([1, 0]) # SQLite stores BOOLEAN as 0 or 1
        last_login = datetime.now() - timedelta(days=random.randint(1, 365))

        # Insert data without the geometry column first
          cursor.execute("INSERT INTO users (user_id, username, age, balance, is_active,
last_login)  VALUES  (?,  ?,  ?,  ?,  ?,  ?)",  (i,  username,  age,  balance,  is_active,
last_login))
```

```
        # Then, update the geometry column with the SpatiaLite-formatted WKB
        location = Point(random.uniform(-180, 180), random.uniform(-90, 90))
        cursor.execute("UPDATE users SET location = ST_GeomFromText(?, 4326) WHERE user_id = ?", (location.wkt, i))

    # Table 2: Sales (same as before)
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS sales (
            sale_id INTEGER PRIMARY KEY,
            user_id INTEGER,
            product_name TEXT,
            sale_date DATE,
            quantity INT,
            price FLOAT,
            FOREIGN KEY(user_id) REFERENCES users(user_id)
        );
    """)

    sales_data = []
    for i in range(1, 21):
        user_id = random.randint(1, 10)
        product_name = random.choice(['Laptop', 'Mouse', 'Keyboard', 'Monitor'])
        sale_datetime_obj = datetime.now() - timedelta(days=random.randint(1, 180))
        sale_date = sale_datetime_obj.strftime('%Y-%m-%d')
        quantity = random.randint(1, 5)
        price = random.uniform(50.0, 1500.0)
        sales_data.append((i, user_id, product_name, sale_date, quantity, price))

    cursor.executemany("INSERT INTO sales VALUES (?, ?, ?, ?, ?, ?)", sales_data)

    # Table 3: Regions (with polygon geometry)
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS regions (
            region_id INTEGER PRIMARY KEY,
            name VARCHAR(50)
        );
    """)
    # Add the GEOMETRY column
    cursor.execute("SELECT AddGeometryColumn('regions', 'boundaries', 4326, 'POLYGON', 2);")

    regions_data = [
        (1, 'North', Polygon(((0, 0), (0, 45), (90, 45), (90, 0), (0, 0)))),
        (2, 'South', Polygon(((-90, -90), (-90, 0), (0, 0), (0, -90), (-90, -90)))),
    ]

    for region_id, name, polygon in regions_data:
        cursor.execute("INSERT INTO regions (region_id, name) VALUES (?, ?)", (region_id, name))
        cursor.execute("UPDATE regions SET boundaries = ST_GeomFromText(?, 4326) WHERE region_id = ?", (polygon.wkt, region_id))

    print("Dummy data generated and inserted successfully.")
```

```
def create_db():
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)

    conn = sqlite3.connect(DB_NAME)

    # Enable SpatiaLite extension loading
    conn.enable_load_extension(True)
    try:
        # Load the SpatiaLite extension
        conn.execute("SELECT load_extension('mod_spatialite')")
        print("SpatiaLite extension loaded successfully.")
    except sqlite3.OperationalError as e:
        print(f"Error loading SpatiaLite extension: {e}")
        print("Please ensure 'mod_spatialite' is correctly installed and in the system's
path.")
        conn.close()
        return

    cursor = conn.cursor()
    # SpatiaLite requires its metadata tables to be initialized
    cursor.execute("SELECT InitSpatialMetaData(1)")

    generate_dummy_data(cursor)
    conn.commit()
    conn.close()
    print(f"Database '{DB_NAME}' created.")

if __name__ == '__main__':
    create_db()
```

src/wcm_nlq2sql.egg-info/PKG-INFO

```
Metadata-Version: 2.4
Name: wcm-nlq2sql
Version: 0.1.0
Summary: A Natural Language to SQL converter using LLMs and Grammar.
Author-email: Your Name <your.email@example.com>
Classifier: Programming Language :: Python :: 3
Classifier: License :: OSI Approved :: MIT License
Classifier: Operating System :: OS Independent
Requires-Python: >=3.8
Description-Content-Type: text/markdown
Provides-Extra: dev
Requires-Dist: pytest; extra == "dev"
Requires-Dist: pytest-cov; extra == "dev"
Requires-Dist: flake8; extra == "dev"
```

src/wcm_nlq2sql.egg-info/SOURCES.txt

```
pyproject.toml
src/scripts/generate_db.py
src/wcm_nlq2sql/nlq_to_sql.py
src/wcm_nlq2sql/query_builder.py
src/wcm_nlq2sql/sql_build_helper.py
src/wcm_nlq2sql/sql_transformer.py
src/wcm_nlq2sql.egg-info/PKG-INFO
src/wcm_nlq2sql.egg-info/SOURCES.txt
src/wcm_nlq2sql.egg-info/dependency_links.txt
src/wcm_nlq2sql.egg-info/requires.txt
src/wcm_nlq2sql.egg-info/top_level.txt
```

src/wcm_nlq2sql.egg-info/dependency_links.txt

src/wcm_nlq2sql.egg-info/requires.txt

```
[dev]
pytest
pytest-cov
flake8
```

src/wcm_nlq2sql.egg-info/top_level.txt

```
scripts
wcm_nlq2sql
```

src/n2s_generators/graph_builder.py

```
# src/n2s_generators/graph_builder.py

from __future__ import annotations
from collections import defaultdict
import re
from typing import Any, Dict, List, Iterable, Tuple

# ----------------------------
# Type & role normalization
# ----------------------------

def _type_category(db_type: str) -> str:
    """Map raw DB types to coarse categories the binder can reason about."""
    if not db_type:
        return "any"
    t = db_type.strip().lower()
    # Spatial first (already normalized in your pipeline)
    if t.startswith("geometry_") or t.startswith("geography_"):
        return t  # keep explicit subtype: geometry_point, geography_polygon, etc.
    if "int" in t:
        return "numeric"
    if any(k in t for k in ["decimal", "numeric", "real", "float", "double"]):
        return "numeric"
    if "char" in t or "text" in t or "string" in t:
        return "text"
    if "bool" in t:
        return "boolean"
    if "timestamp" in t:
        return "timestamp"
    if "date" in t:
        return "date"
    return "any"


# Defaults for function classification (extend as needed)
_RETURNS_TYPE: Dict[str, str] = {
    # sql aggregates/scalars
    "count": "numeric", "sum": "numeric", "avg": "numeric",
    "min": "any", "max": "any", "distinct": "any",
    "length": "numeric", "concat": "text", "cast": "any", "extract": "numeric",
    "order_by_asc": "none", "order_by_desc": "none",
    "group_by": "none", "having": "boolean", "limit": "none",
    # spatial
    "st_distance": "numeric", "st_length": "numeric", "st_area": "numeric",
    "st_x": "numeric", "st_y": "numeric",
    "st_geometrytype": "text",
    "st_within": "boolean", "st_contains": "boolean",
    "st_intersects": "boolean", "st_crosses": "boolean", "st_touches": "boolean",
    "st_spatial_index": "boolean",
           "st_buffer":  "geometry_polygon",  "st_union":  "geometry",  "st_centroid":
"geometry_point",
```

```
    "st_transform": "geometry",
    "st_distance_operator": "numeric",
}

_CLASS: Dict[str, str] = {
    "count": "aggregate", "sum": "aggregate", "avg": "aggregate",
    "min": "aggregate", "max": "aggregate", "distinct": "scalar",
    "length": "scalar", "concat": "scalar", "cast": "scalar", "extract": "scalar",
    "order_by_asc": "ordering", "order_by_desc": "ordering",
    "group_by": "grouping", "having": "predicate", "limit": "limit",
    # spatial
    "st_distance": "spatial", "st_length": "spatial", "st_area": "spatial",
    "st_x": "spatial", "st_y": "spatial", "st_geometrytype": "spatial",
    "st_within": "predicate", "st_contains": "predicate",
    "st_intersects": "predicate", "st_crosses": "predicate", "st_touches": "predicate",
    "st_spatial_index": "predicate",
    "st_buffer": "spatial", "st_union": "spatial", "st_centroid": "spatial",
    "st_transform": "spatial",
    "st_distance_operator": "spatial",
}

_CLAUSE: Dict[str, str] = {
    # projections
    "count": "select", "sum": "select", "avg": "select",
    "min": "select", "max": "select", "distinct": "select",
    "length": "select", "concat": "select", "cast": "select", "extract": "select",
    # modifiers/filters
    "order_by_asc": "order_by", "order_by_desc": "order_by",
    "group_by": "group_by", "having": "having", "limit": "limit",
    # spatial
    "st_distance": "select", "st_length": "select", "st_area": "select",
    "st_x": "select", "st_y": "select", "st_geometrytype": "select",
    "st_within": "where", "st_contains": "where",
    "st_intersects": "where", "st_crosses": "where", "st_touches": "where",
    "st_spatial_index": "where",
    "st_buffer": "select", "st_union": "select", "st_centroid": "select",
    "st_transform": "select",
    "st_distance_operator": "order_by",  # often used in ORDER BY <-> nearest
}

# -----------------------------
# Tokenization & template parsing
# -----------------------------

TOKEN_RE = re.compile(r"(>=|<=|!=|==|<>|\|\||&&|[A-Za-z0-9_']+|[^\sA-Za-z0-9_])")
PLACEHOLDER_RE = re.compile(r"\{([^}]+)\}")

def _tokenize_text(s: str) -> List[str]:
    return TOKEN_RE.findall(s or "")

def _parse_template_to_surface(tmpl: str) -> List[str]:
    """
    Convert a template like '{column} != {value}' or 'EXTRACT({part} FROM {column})'
    into a flat token list with placeholders preserved as '{name}' tokens.
```

```
    """
    if not tmpl:
        return []
    out: List[str] = []
    pos = 0
    for m in PLACEHOLDER_RE.finditer(tmpl):
        pre = tmpl[pos:m.start()]
        if pre.strip():
            out.extend(_tokenize_text(pre))
        name = m.group(1).strip()
        out.append(f"{{{name}}}")
        pos = m.end()
    tail = tmpl[pos:]
    if tail.strip():
        out.extend(_tokenize_text(tail))
    return [t for t in out if t]

def _normalize_pattern(pattern: Any) -> List[str] | None:
    """
    Accepts either a list of tokens/strings (possibly with placeholders) or None.
    Returns a token list or None if unusable.
    """
    if pattern is None:
        return None
    if isinstance(pattern, list):
        toks: List[str] = []
        for p in pattern:
            p = str(p)
            parts = []
            pos = 0
            for m in PLACEHOLDER_RE.finditer(p):
                pre = p[pos:m.start()]
                if pre.strip():
                    parts.extend(_tokenize_text(pre))
                parts.append(f"{{{m.group(1).strip()}}}")
                pos = m.end()
            tail = p[pos:]
            if tail.strip():
                parts.extend(_tokenize_text(tail))
            if not parts:
                parts = _tokenize_text(p)
            toks.extend(parts)
        return [t for t in toks if t]
    if isinstance(pattern, str):
        return _parse_template_to_surface(pattern)
    return None

def _extract_args_from_applicable_types(app: Any) -> List[Dict[str, Any]]:
    """
    Turn the 'applicable_types' mapping into an ordered arg list:
         {'column':[numeric], 'value':[any]} -> [{'name':'column','types':['numeric']},
...]
    """
    args: List[Dict[str, Any]] = []
```

```
    if not isinstance(app, dict):
        return args
    for name, types in app.items():
        if isinstance(types, (list, tuple)):
            tlist = [str(t) for t in types]
        else:
            tlist = [str(types)]
        args.append({"name": str(name), "types": tlist})
    return args

def _arg_names_from_tokens(tokens: Iterable[str]) -> List[str]:
    seen, names = set(), []
    for t in tokens:
        if t.startswith("{") and t.endswith("}"):
            name = t[1:-1].strip()
            if name and name not in seen:
                seen.add(name)
                names.append(name)
    return names

# -----------------------------
# Surfaces heuristics
# -----------------------------

def _default_surfaces_for_args(head: str, arg_names: List[str]) -> List[List[str]]:
    """
    Very small, safe default surfaces when no pattern/template is provided.
    - 1 arg: [head, "of", "{arg1}"]
    - 2 args: [head, "{arg1}", "and", "{arg2}"]
    """
    if not arg_names:
        return [[head]]
    if len(arg_names) == 1:
        return [[head, "of", f"{{{arg_names[0]}}}"]]
    if len(arg_names) == 2:
        a1, a2 = arg_names
        return [[head, f"{{{a1}}}", "and", f"{{{a2}}}"]]
    parts: List[str] = [head]
    for i, a in enumerate(arg_names):
        if i == 0:
            parts += [f"{{{a}}}"]
        else:
            parts += [",", f"{{{a}}}"]
    return [parts]

def  _alias_head_surfaces(aliases:  List[str],  base_surface:  List[str],  canonical_head:
str) -> List[List[str]]:
    """
    If the base surface starts with a single headword (e.g., 'avg'/'distance'),
    produce variants that swap just that head with alias phrases (e.g., 'average').
    Conservative: only when the first token is a single word and not a placeholder.
    """
    if not base_surface:
        return []
```

```
    first = base_surface[0]
    if first.startswith("{") or first in {",", "(", ")"}:
        return []
    out = []
    for a in aliases or []:
        s = str(a).strip()
        if not s:
            continue
        alias_tokens = _tokenize_text(s)
        out.append(alias_tokens + base_surface[1:])
    return out

# ----------------------------
# Diagnostics helpers
# ----------------------------

def _new_diagnostics() -> Dict[str, Any]:
    return {
        "alias_collisions": [],
        "prefix_collisions": [],
        "surface_warnings": [],
        "inferred_args": [],
        "surfaces_args_mismatch": [],
        "missing_applicable_types": [],
        "connectors": [],
    }

def _index_aliases(alias_index: Dict[str, List[Dict[str, str]]],
                   canonical: str,
                   etype: str,
                   aliases: Iterable[str]) -> None:
    for a in aliases:
        key = str(a).strip().lower()
        if not key:
            continue
        alias_index.setdefault(key, [])
        entry = {"canonical": canonical, "type": etype}
        if entry not in alias_index[key]:
            alias_index[key].append(entry)

def _finalize_diagnostics(graph: Dict[str, Dict[str, Any]],
                          alias_index: Dict[str, List[Dict[str, str]]],
                          connectors: List[str],
                          diags: Dict[str, Any]) -> None:
    # 1) alias collisions
    for alias, entries in sorted(alias_index.items()):
        if len(entries) > 1:
            diags["alias_collisions"].append({
                "alias": alias,
                  "meanings": [{"canonical": e["canonical"], "type": e["type"]} for e in
entries]
            })

    # 2) prefix collisions (single-word aliases that prefix multi-word aliases)
```

```
    keys = list(alias_index.keys())
    multi = [k for k in keys if " " in k]
    singles = [k for k in keys if " " not in k]
    multi_by_prefix = defaultdict(list)
    for mw in multi:
        pfx = mw.split()[0]
        multi_by_prefix[pfx].append(mw)
    for s in singles:
        if s in multi_by_prefix:
            diags["prefix_collisions"].append({
                "alias": s,
                "longer_keys": sorted(multi_by_prefix[s])
            })

    # 3) surfaces  args mismatch and missing types
    for canon, node in graph.items():
        binder = node.get("binder")
        if not binder:
            continue
        args = binder.get("args", [])
        arg_names = {a.get("name") for a in args if isinstance(a, dict)}
        # gather placeholders from all surfaces
        placeholders = set()
        for surf in binder.get("surfaces", []):
            toks = surf.get("pattern", [])
            for t in toks:
                if isinstance(t, str) and t.startswith("{") and t.endswith("}"):
                    placeholders.add(t[1:-1].strip())
        if arg_names != placeholders:
            diags["surfaces_args_mismatch"].append({
                "canonical": canon,
                "args": sorted(arg_names),
                "placeholders": sorted(placeholders)
            })
        if not node.get("metadata", {}).get("applicable_types") and not arg_names and
not placeholders:
            diags["missing_applicable_types"].append({"canonical": canon})

    # 4) connectors (final inventory)
    diags["connectors"] = list(connectors)

# -----------------------------
# Node builders
# -----------------------------

def _build_table_nodes(schema_yaml: Dict[str, Any],
                       graph: Dict[str, Dict[str, Any]],
                       alias_index: Dict[str, List[Dict[str, str]]]) -> None:
    if not schema_yaml or "tables" not in schema_yaml:
        return

    for tbl, tmeta in schema_yaml["tables"].items():
        # table aliases
        aliases = list(sorted(set(tmeta.get("aliases", []) + [tbl])))
```

```
        # build nested columns metadata for the table node
        cols = tmeta.get("columns", {}) or {}
        table_columns_meta: Dict[str, Dict[str, Any]] = {}

        for col, cmeta in cols.items():
            raw_type = cmeta.get("type") or ""
                      col_aliases  =  list(sorted(set(cmeta.get("aliases",  [])  +  [col,
col.replace("_", " ")])))
            col_meta = {
                "aliases": col_aliases,
                "type": raw_type,
                "type_category": _type_category(raw_type),
                "labels": list(sorted(set(cmeta.get("labels", [])))),
            }
            # table-nested copy
            table_columns_meta[col] = col_meta

            # top-level column node (unchanged behavior)
            graph[col] = {"entity_type": "column", "metadata": col_meta}
            _index_aliases(alias_index, col, "column", col_aliases)

        # finally, the table node with nested columns present
        graph[tbl] = {
            "entity_type": "table",
            "metadata": {
                "aliases": aliases,
                "columns": table_columns_meta,
            }
        }
        _index_aliases(alias_index, tbl, "table", aliases)


def _classify_function_name(name: str) -> Tuple[str, str, str]:
    n = name.strip()
    returns = _RETURNS_TYPE.get(n, "any")
    cls = _CLASS.get(n, "scalar")
    clause = _CLAUSE.get(n, "select")
    return returns, cls, clause

def _build_function_node(canon: str,
                         meta: Dict[str, Any],
                         diags: Dict[str, Any]) -> Dict[str, Any]:
    aliases = list(sorted(set(meta.get("aliases", []) + [canon])))
    template = meta.get("template")
    pattern = meta.get("pattern")
    app_types = meta.get("applicable_types")

    # build args
    args = _extract_args_from_applicable_types(app_types)

    # derive surfaces
    surfaces: List[List[str]] = []
    if pattern is not None:
```

```
        toks = _normalize_pattern(pattern)
        if toks:
            surfaces.append(toks)
        else:
                        diags["surface_warnings"].append({"canonical":  canon,  "reason":
"bad_pattern", "pattern": pattern})
    elif template:
        toks = _parse_template_to_surface(template)
        if toks:
            surfaces.append(toks)
        else:
                        diags["surface_warnings"].append({"canonical":  canon,  "reason":
"bad_template", "template": template})

    # If we still don't have args, try to infer from surfaces
    if not args and surfaces:
        names = _arg_names_from_tokens(surfaces[0])
        if names:
            args = [{"name": n, "types": ["any"]} for n in names]
            diags["inferred_args"].append({"canonical": canon, "arg_names": names})

    # Fallback default surface if none present
    if not surfaces:
        head = canon
        arg_names = [a["name"] for a in args]
        surfaces.extend(_default_surfaces_for_args(head, arg_names))

    # Alias-head variants (conservative)
    head = canon
    alias_head_variants = _alias_head_surfaces(aliases, surfaces[0], head)
    surfaces.extend(alias_head_variants)

    returns_type, fclass, clause = _classify_function_name(canon)

    return {
        "entity_type": meta.get("_entity_type_hint", "sql_actions"),
        "metadata": {
            "aliases": aliases,
            "applicable_types": app_types or {},
            "label_rules": meta.get("label_rules", []),
            "explanation": meta.get("explanation", ""),
        },
        "binder": {
            "returns_type": returns_type,
            "class": fclass,
            "clause": clause,
            "args": args,
            "surfaces": [{"pattern": s, "commutative": False} for s in surfaces],
        }
    }

def _build_comparison_node(canon: str,
                              meta: Dict[str, Any]) -> Dict[str, Any]:
    """
```

```python
    Build predicate operators like not_equal, greater_than, etc.
    Surfaces:
      - from template (e.g., "{column} != {value}")
      - per-alias expansions ("is not"  ["{column}","is","not","{value}"])
    """
    aliases = list(sorted(set(meta.get("aliases", []) + [canon])))
    template = meta.get("template") or "{column} = {value}"
    app_types = meta.get("applicable_types") or {"column": ["any"], "value": ["any"]}
    args = _extract_args_from_applicable_types(app_types)

    base_surface = _parse_template_to_surface(template)
    surfaces: List[List[str]] = [base_surface] if base_surface else []

    for a in meta.get("aliases", []):
        alias_tokens = _tokenize_text(str(a))
        surfaces.append(["{column}"] + alias_tokens + ["{value}"])

    return {
        "entity_type": "comparison_operators",
        "metadata": {
            "aliases": aliases,
            "applicable_types": app_types,
            "label_rules": meta.get("label_rules", []),
            "explanation": meta.get("explanation", ""),
        },
        "binder": {
            "returns_type": "boolean",
            "class": "predicate",
            "clause": "where",
            "args": args,
            "surfaces": [{"pattern": s, "commutative": False} for s in surfaces],
        }
    }

def _build_keyword_nodes(keywords_yaml: Dict[str, Any],
                         graph: Dict[str, Dict[str, Any]],
                         alias_index: Dict[str, List[Dict[str, str]]]) -> List[str]:
    """
    Create nodes for select_verbs, prepositions, logical_operators; collect connectors.
    Returns a list of connector tokens useful to the binder.
    """
    connectors: List[str] = []
    if not keywords_yaml or "keywords" not in keywords_yaml:
        return connectors

    kw = keywords_yaml["keywords"]

    # select_verbs
    for name, data in (kw.get("select_verbs") or {}).items():
        aliases = list(sorted(set((data or {}).get("aliases", []) + [name])))
        graph[name] = {"entity_type": "select_verbs", "metadata": {"aliases": aliases}}
        _index_aliases(alias_index, name, "select_verbs", aliases)

    # prepositions  these are pure connectors
```

```
    for name, data in (kw.get("prepositions") or {}).items():
        aliases = list(sorted(set((data or {}).get("aliases", []) + [name])))
        graph[name] = {"entity_type": "prepositions", "metadata": {"aliases": aliases}}
        _index_aliases(alias_index, name, "prepositions", aliases)
        connectors.extend(aliases)

    # logical_operators
    for name, data in (kw.get("logical_operators") or {}).items():
        aliases = list(sorted(set((data or {}).get("aliases", []) + [name])))
        graph[name] = {
            "entity_type": "logical_operators",
             "metadata": {"aliases": aliases, "template": (data or {}).get("template",
"")},
        }
        _index_aliases(alias_index, name, "logical_operators", aliases)

    # comparison_operators  full binder nodes
    for name, data in (kw.get("comparison_operators") or {}).items():
        node = _build_comparison_node(name, data or {})
        graph[name] = node
                          _index_aliases(alias_index,   name,   "comparison_operators",
node["metadata"]["aliases"])

    return list(sorted(set(connectors)))

def _build_action_nodes(section_name: str,
                        actions: Dict[str, Any],
                        graph: Dict[str, Dict[str, Any]],
                        alias_index: Dict[str, List[Dict[str, str]]],
                        diags: Dict[str, Any]) -> None:
    """
    Build nodes for sql_actions or postgis_actions with binder signatures and surfaces.
    """
    for name, data in (actions or {}).items():
        meta = dict(data or {})
        meta["_entity_type_hint"] = section_name
        node = _build_function_node(name, meta, diags)
        graph[name] = node
        _index_aliases(alias_index, name, section_name, node["metadata"]["aliases"])

# ----------------------------
# Connectors from surfaces
# ----------------------------

def _harvest_connectors_from_surfaces(graph: Dict[str, Dict[str, Any]],
                                      seed_connectors: Iterable[str]) -> List[str]:
    known = set(seed_connectors or [])
    CANDIDATES = {"of", "from", "between", "and", "to", "with", "in", "on", "at", ","}
    for node in graph.values():
        binder = node.get("binder")
        if not binder:
            continue
        for surf in binder.get("surfaces", []):
            for tok in surf.get("pattern", []):
```

```
                if tok in CANDIDATES:
                    known.add(tok)
    return list(sorted(known))

# ----------------------------
# Public API
# ----------------------------

def generate_relationship_graph(schema_yaml: Dict[str, Any],
                                keywords_yaml: Dict[str, Any]) -> Dict[str, Any]:
    graph: Dict[str, Dict[str, Any]] = {}
    diagnostics = _new_diagnostics()
    alias_index: Dict[str, List[Dict[str, str]]] = {}

    # 1) tables & columns
    _build_table_nodes(schema_yaml, graph, alias_index)

    # 2) keywords (verbs, preps, logical, comparisons)
    connectors = _build_keyword_nodes(keywords_yaml, graph, alias_index)

    # 3) actions (sql + postgis)
    if keywords_yaml:
            _build_action_nodes("sql_actions", keywords_yaml.get("sql_actions"), graph,
alias_index, diagnostics)
            _build_action_nodes("postgis_actions", keywords_yaml.get("postgis_actions"),
graph, alias_index, diagnostics)

    # 4) connectives inventory for binder
    connectors = _harvest_connectors_from_surfaces(graph, connectors)
    graph["_binder_meta"] = {
        "entity_type": "_meta",
        "metadata": {"connectors": connectors},
    }

    # 5) finalize diagnostics
    _finalize_diagnostics(graph, alias_index, connectors, diagnostics)
    graph["_diagnostics"] = {
        "entity_type": "_meta",
        "metadata": diagnostics,
    }

    return graph
```

src/n2s_generators/graph_generation.md

Step-by-step plan

0) Goals for the new graph

Each node in the graph should keep the original metadata **and** add what the binder needs:

* **Tables/columns**

  * `aliases` (from schema)
  * `type` (raw)
  * `type_category` (coarse: numeric|text|date|timestamp|boolean|geometry_point|)
  * `labels` (e.g., `id`, `postgis`)

* **Functions / operators (SQL + PostGIS + comparisons + logical)**

  * `aliases`
  * `signature` (binder-ready)

    * `returns_type`
    * `class` (aggregate|scalar|spatial|predicate|ordering|grouping|limit)
    * `clause` (select|where|having|order_by|group_by|limit)
    * `args`: ordered slots with `name`, `types` (type categories), optional label rules
  * `surfaces`: normalized token patterns that match how people say it (from `pattern` or `template` + alias expansions)

    * e.g., `["distance", "between", "{geom1}", "and", "{geom2}"]`
    * e.g., `["{column}", "is", "not", "{value}"]` and `["{column}", "!=", "{value}"]`

* **Binder meta**

  * `connectors`/prepositions (e.g., `of`, `from`, `and`, `between`, `to`, `with`)
  * (optional) `value_lexicon` (e.g., `"active"  true`) if you want it later

1) Read and normalize **schema.yaml**

* For each table:

  * create a node with `entity_type: table`
  * keep `aliases`
  * for each column:

    * create a node with `entity_type: column`
    * keep `aliases`, `type`, `labels`
    * compute `type_category` from `type` (e.g., `INT` `numeric`, `BOOLEAN` `boolean`, `geography_point` `geography_point`)

2) Read and normalize **keywords_and_functions.yaml**

* **keywords/**:

  * `select_verbs`, `prepositions`, `logical_operators`

    * create nodes with appropriate `entity_type`
    * keep `aliases`
    * for logical operators, keep `template` if present (binder can use later)
* **comparison_operators**:

  * create nodes with `entity_type: comparison_operators`
  * parse `template` to derive a canonical **surface** (`["{column}", "!=", "{value}"]`, etc.)
  * expand **alias surfaces** (e.g., `"is not"` `["{column}", "is", "not", "{value}"]`)
  * construct `signature`:

    * `returns_type: boolean`
    * `class: predicate`
    * `clause: where` (and having if you choose)
    * `args`: from `applicable_types` (e.g., `column:any`, `value:any`)
* **sql_actions** and **postgis_actions**:

  * create nodes with `entity_type` accordingly
  * derive or set:

    * `signature` with `returns_type`, `class`, `clause` (use a namedefaults map with sane heuristics)
    * `args` from `applicable_types` (or infer from `template`/`pattern`)
  * derive **surfaces**:

    * if `pattern` present (list), convert to normalized tokens
    * else parse `template` to a tokenized surface
    * add a fallback default surface (e.g., unary `["<head>", "of", "{arg1}"]`, binary `["<head>", "{arg1}", "and", "{arg2}"]`)
    * add **alias headword** surfaces where safe (e.g., `"average"` for `avg`)

3) Build **connectors** inventory for the binder

* Collect from `prepositions` keys/aliases + literals that appear in surfaces (`of`, `from`, `between`, `to`, `with`, `and`, `,`)

4) Emit **relationship_graph.yaml**

* Keep your current top-level shape (canonical name node)
* Add `_binder_meta` with `connectors`
* Keep it strictly data-only (no tuples/sets that break YAML)

Notes & extension points

* **Surfaces safety:** The code is intentionally conservative generating alias-head variants only when the surface clearly starts with a head literal. This prevents the alias explosion that used to wipe out your pass rate.
* **Types:** `type_category` will coarsen whatever SQL type you pull from the DB; your existing spatial subtype naming (`geography_point`, `geometry_polygon`) is preserved and flows straight into the binder.
* **Comparisons:** Each alias is turned into a surface slotting between `{column}` and

`{value}`, so `isn't` and `!=` both work.
* **Connectors:** We seed from `prepositions` and harvest from all function surfaces; that gives the normalizer/binder a canonical list to treat specially (avoid duplication, etc).
* **Heuristics:** The `returns_type`, `class`, `clause` maps are all easy to tune as your library grows.

src/n2s_generators/knowledge_compiler.py

```python
# src/n2s_generators/knowledge_compiler.py

from __future__ import annotations
from collections import defaultdict
from typing import Any, Dict, Iterable, List, Tuple

# ============================================================
# Helpers: entity extraction
# ============================================================

def _extract_entities(graph: Dict[str, Any]) -> Dict[str, Dict[str, Any]]:
    """
    Bucket nodes by entity_type for convenient access.
    """
    out = defaultdict(dict)
    for key, node in graph.items():
        et = node.get("entity_type")
        if not et:
            continue
        if key.startswith("_"):  # skip meta/diagnostics keys
            continue
        out[et][key] = node
    return out


# ============================================================
# Vocabulary builder (normalizer-facing)
# ============================================================

# Config knobs (kept small; structure lives in binder)
PREP_BARE = {"of", "from", "in", "on", "at"}
GENERIC_DENY = {"order by", "sort by", "by", "ascending", "descending"}
OF_CANONICALS = {"distinct", "avg", "sum", "st_distance"}
DOMAIN_PREFER_SPATIAL = {
    "contains": ("like", "st_contains"),
    "intersects": ("st_spatial_index", "st_intersects"),
    "overlaps": ("st_spatial_index", "st_intersects"),
}
PLURAL_LASTWORD = {
    "date": "dates",
    "login": "logins",
    "id": "ids",
    "username": "usernames",
    "name": "names",
    "item": "items",
    "value": "values",
}
SAFE_PLURAL_LASTWORD = set(PLURAL_LASTWORD.keys())
ALLOWED_TYPES_FOR_PLURAL = {"table", "column"}

def _add_alias(master: Dict[str, List[dict]], alias: str, entry: dict) -> None:
```

```python
    a = (alias or "").strip().lower()
    if not a:
        return
    L = master.setdefault(a, [])
    if entry not in L:
        L.append(entry)

def _collect_aliases_for_vocab(graph: Dict[str, Any],
                               diagnostics: Dict[str, Any]) -> Dict[str, List[dict]]:
    """
    Build alias -> [{canonical, type}] index from graph.
    """
    master: Dict[str, List[dict]] = defaultdict(list)
    allowed_types = {
        "table", "column", "sql_actions", "postgis_actions",
        "select_verbs", "prepositions", "logical_operators",
        "comparison_operators", "filler_words"
    }

    for canonical, node in graph.items():
        etype = node.get("entity_type")
        if etype not in allowed_types:
            continue

        # canonical is an alias to itself
        _add_alias(master, canonical, {"canonical": canonical, "type": etype})

        md = node.get("metadata", {}) or {}
        for alias in md.get("aliases", []) or []:
            alias_s = str(alias).strip()
            if not alias_s:
                continue
            if alias_s.lower() in GENERIC_DENY:
                diagnostics["generic_denied"].append(alias_s.lower())
                continue
            _add_alias(master, alias_s, {"canonical": canonical, "type": etype})

    return master

def _synthesize_plurals(master: Dict[str, List[dict]],
                        diagnostics: Dict[str, Any]) -> None:
    """
    Add safe plural forms (last word only) for table/column aliases.
    """
    keys = list(master.keys())
    for k in keys:
        types_here = {m["type"] for m in master[k]}
        if not types_here or not types_here.issubset(ALLOWED_TYPES_FOR_PLURAL):
            continue
        parts = k.split()
        if not parts:
            continue
        lw = parts[-1]
        if lw in SAFE_PLURAL_LASTWORD:
```

```
            plural_key = " ".join(parts[:-1] + [PLURAL_LASTWORD[lw]])
            if plural_key not in master:
                master[plural_key] = list(master[k])
                diagnostics["plural_added"].append({"from": k, "to": plural_key})

def _compute_prefix_to_longers(master: Dict[str, List[dict]]) -> Dict[str, List[str]]:
    multi = [k for k in master.keys() if " " in k]
    out = defaultdict(list)
    for m in multi:
        pfx = m.split()[0]
        out[pfx].append(m)
    return out
def _is_meta_entity_type(etype: str | None) -> bool:
    return etype not in {
        "table", "column",
        "sql_actions", "postgis_actions", "comparison_operators",
        "select_verbs", "prepositions", "logical_operators", "filler_words"
    }

def _should_identity_for_builder(canonical: str, etype: str | None) -> bool:
    """
    Identity (canonical -> canonical) should exist for all *real* canonicals:
    tables, columns, actions, comparison ops, keywords but not meta keys or filler.
    """
    if not isinstance(canonical, str) or not canonical:
        return False
    if canonical.startswith("_"):                 # e.g., _binder_meta, _diagnostics
        return False
    if _is_meta_entity_type(etype):
        return False
     if etype == "filler_words":                      # filler canonicals aren't used as
surface keys
        return False
    return True

def _enforce_identity_mappings(graph: Dict[str, Any],
                               vocab: Dict[str, Any],
                               diagnostics: Dict[str, Any]) -> None:
    """
    Ensure deterministic identity for every canonical that should have one,
    regardless of multi-word shape or policy filters that might have dropped it.
    If an ND entry exists for the same alias, remove it to avoid split semantics.
    """
    det = vocab.setdefault("deterministic_aliases", {})
    nd  = vocab.setdefault("non_deterministic_aliases", {})

    added: List[str] = []
    overridden: List[Dict[str, str]] = []

    for canon, node in (graph or {}).items():
        et = (node or {}).get("entity_type")
        if not _should_identity_for_builder(canon, et):
            continue
```

```
        prev = det.get(canon, None)
        if prev is None:
            det[canon] = canon
            if canon in nd:
                nd.pop(canon, None)
            added.append(canon)
        elif prev != canon:
            overridden.append({"alias": canon, "prev": str(prev), "now": canon})
            det[canon] = canon
            if canon in nd:
                nd.pop(canon, None)

    diagnostics["identity_added"] = added
    diagnostics["identity_overrides"] = overridden

def _apply_preposition_purity(alias: str,
                              meanings: List[dict],
                              diagnostics: Dict[str, Any]) -> List[dict]:
    """
    If alias is a bare preposition (e.g., 'in', 'of'), keep only preposition
    meanings  but only when at least one preposition meaning exists.
    Otherwise, leave meanings unchanged (avoid dropping identity for non-preps).
    """
    if alias not in PREP_BARE:
        return meanings

    preps = [m for m in meanings if m["type"] == "prepositions"]
    if preps:
        dropped = len(meanings) - len(preps)
        if dropped > 0:
                 diagnostics["preposition_conflicts"].append({"alias": alias, "dropped":
dropped})
        return preps

    # No preposition meaning present; don't drop everything.
    return meanings


def _apply_domain_separation(alias: str,
                             meanings: List[dict],
                             diagnostics: Dict[str, Any]) -> List[dict]:
    if alias not in DOMAIN_PREFER_SPATIAL:
        return meanings
    lose, prefer = DOMAIN_PREFER_SPATIAL[alias]
    cset = {m["canonical"] for m in meanings}
    if lose in cset and prefer in cset:
        kept = [m for m in meanings if m["canonical"] != lose]
         diagnostics["domain_conflicts"].append({"alias": alias, "dropped": lose, "kept":
prefer})
        return kept
    return meanings

def _apply_prefix_protection(alias: str,
                             meanings: List[dict],
```

```python
                              prefix_to_longers: Dict[str, List[str]],
                              diagnostics: Dict[str, Any]) -> List[dict]:
    """
    Warn-only: keep meanings but log that this single-token alias prefixes longer keys.
    We avoid dropping table meanings so coverage (V1) doesn't fail on table synonyms
    like 'user', 'region', etc.
    """
    if " " in alias:
        return meanings
    if alias not in prefix_to_longers:
        return meanings

    longers = sorted(prefix_to_longers[alias])
    diagnostics["prefix_collisions"].append(
        {"alias": alias, "longer_keys": longers, "action": "kept_table_meaning"}
    )
    return meanings


def _clean_alias_index(master: Dict[str, List[dict]],
                       diagnostics: Dict[str, Any]) -> Dict[str, List[dict]]:
    """
    Apply purity, domain preference, and prefix protection.
    """
    prefix_to_longers = _compute_prefix_to_longers(master)
    cleaned = {}
    for alias, meanings in master.items():
        kept = list(meanings)
        kept = _apply_preposition_purity(alias, kept, diagnostics)
        kept = _apply_domain_separation(alias, kept, diagnostics)
        kept = _apply_prefix_protection(alias, kept, prefix_to_longers, diagnostics)
        if kept:
            cleaned[alias] = kept
    return cleaned

def _dedupe(seq: Iterable[str]) -> List[str]:
    seen, out = set(), []
    for s in seq:
        if s not in seen:
            seen.add(s)
            out.append(s)
    return out

def build_vocabulary(graph: Dict[str, Any]) -> Dict[str, Any]:
    """
    Build the normalizer vocabulary from the graph.

    Guarantees:
      - All non-meta canonicals get a deterministic identity: c -> c
      - Filler words map deterministically to ""
      - Policy filters (preposition purity, domain preference, prefix protection)
        are applied to *aliases*, not to canonical identities
      - Multi-word canonicals still get deterministic identities
    """
```

```python
diagnostics = {
    "prefix_collisions": [],
    "preposition_conflicts": [],
    "domain_conflicts": [],
    "generic_denied": [],
    "plural_added": [],
    "of_policy_adjusted": [],
    "identity_added": [],
    "identity_overrides": [],
}

# 1) Collect all graph aliases (including canonical-as-alias)
master = _collect_aliases_for_vocab(graph, diagnostics)

# 2) Synthesize safe plural forms for tables/columns
_synthesize_plurals(master, diagnostics)

# 3) Apply policy-based cleaning to the alias index
cleaned = _clean_alias_index(master, diagnostics)

# 4) Emit vocabulary buckets
vocab = {"deterministic_aliases": {}, "non_deterministic_aliases": {}}

for alias, meanings in cleaned.items():
    is_multi = " " in alias
    is_amb   = len(meanings) > 1

    # Filler words -> deterministic empty string
    if meanings and all(m["type"] == "filler_words" for m in meanings):
        vocab["deterministic_aliases"][alias] = ""
        continue

    # Canonical identity: if the alias is *exactly* its single canonical,
    # emit deterministic regardless of multi-word shape.
    if not is_amb and len(meanings) == 1 and alias == meanings[0]["canonical"]:
        vocab["deterministic_aliases"][alias] = meanings[0]["canonical"]
        continue

    # Otherwise follow the standard partition rules:
    if is_multi or is_amb:
        targets = [m["canonical"] for m in meanings]

        # '... of' alias policy rewrite for specific canonicals
        if alias.endswith(" of"):
            adjusted = []
            for t in targets:
                if t in OF_CANONICALS and not t.endswith(" of"):
                    adjusted.append(f"{t} of")
                    diagnostics["of_policy_adjusted"].append(
                        {"alias": alias, "canonical": t, "to": f"{t} of"}
                    )
                else:
                    adjusted.append(t)
            targets = adjusted
```

```python
            vocab["non_deterministic_aliases"][alias] = _dedupe(sorted(targets))
        else:
            vocab["deterministic_aliases"][alias] = meanings[0]["canonical"]

    # 5) Hard-enforce deterministic identity for *every* real canonical
    #    (protects against policy filters that may have dropped the alias).
    _enforce_identity_mappings(graph, vocab, diagnostics)

    # 6) Pass through connectors for tests/denormalizer
    connectors = []
    meta = graph.get("_binder_meta") or {}
    if isinstance(meta, dict):
        connectors = list(meta.get("connectors") or [])
    vocab["connectors"] = connectors

    vocab["_diagnostics"] = diagnostics
    return vocab



# ==============================================================
# Binder builder (planner/transformer-facing)
# ==============================================================

def _locate_column_owner_table(entities: Dict[str, Dict[str, Any]]) -> Dict[str, str]:
    """
    Build column -> owning table map using table blocks in the graph.
    """
    owner: Dict[str, str] = {}
    for tname, tnode in entities.get("table", {}).items():
        cols = ((tnode.get("metadata") or {}).get("columns") or {})
        for cname in cols.keys():
            owner[cname] = tname
    return owner


def _pick(md: Dict[str, Any], *keys: str, default=None):
    if not isinstance(md, dict):
        return default
    for k in keys:
        if k in md:
            return md[k]
    return default


def _build_connectors_catalog(graph: Dict[str, Any]) -> Tuple[Dict[str, str], Dict[str,
str], Dict[str, Any]]:
    """
                Produce    binder.catalogs.connectors    (UPPER    ->    surface)    and
binder.catalogs.punctuation.
    Falls back to safe defaults if the graph's connector inventory is missing.
    Also returns a small debug block.
    """
```

```python
    meta = graph.get("_binder_meta") or {}
    conn_list = []
    if isinstance(meta, dict):
        conn_list = list(meta.get("connectors") or [])

    # Normalize to lowercase surfaces
    surfaces = {str(x).strip().lower() for x in conn_list if str(x).strip()}

    # Seed with required surfaces (so B5 won't fail on empty graphs)
    required = {"of", "from", "and"}
    maybe = {"in", "on", "at", "to", "with", "between"}
    punct = {","}

    # Union the sets so we keep what's in the graph
    all_surfaces = (surfaces | required | maybe | punct)

    connectors: Dict[str, str] = {}
    punctuation: Dict[str, str] = {}

    # Map surfaces to canonical UPPER tokens where it makes sense
    for s in sorted(all_surfaces):
        if s == ",":
            punctuation[","] = ","
            connectors["COMMA"] = ","  # convenience
        elif s == "of":
            connectors["OF"] = "of"
        elif s == "from":
            connectors["FROM"] = "from"
        elif s == "and":
            connectors["AND"] = "and"
        elif s == "in":
            connectors["IN"] = "in"
        elif s == "on":
            connectors["ON"] = "on"
        elif s == "at":
            connectors["AT"] = "at"
        elif s == "to":
            connectors["TO"] = "to"
        elif s == "with":
            connectors["WITH"] = "with"
        elif s == "between":
            connectors["BETWEEN"] = "between"
        # ignore anything else silently (can be added later as needed)

    debug = {
        "_connectors_from_graph": sorted(list(surfaces)),
        "_connectors_final_map": dict(connectors),
        "_punctuation_final_map": dict(punctuation),
    }
    return connectors, punctuation, debug


def build_binder(graph: Dict[str, Any]) -> Dict[str, Any]:
    """
```

```
    Consolidate the binder view from the graph into a **full binder** shape:
      {
        "templates": [],  # empty for now
        "catalogs": {
           "functions": {...},
           "columns": {...},
           "tables": [...],                 # LIST of table names (not dict)
           "connectors": {"OF":"of", ...}, # minimal required map present
           "punctuation": {",": ","}
        },
        "_diagnostics": { ... }
      }
    """
    entities = _extract_entities(graph)
    col_owner = _locate_column_owner_table(entities)

    # ---- columns (type/labels + owner) ----
    columns: Dict[str, Any] = {}
    for cname, cnode in entities.get("column", {}).items():
        md = cnode.get("metadata") or {}
        columns[cname] = {
            "type": md.get("type"),
            "type_category": md.get("type_category"),
            "labels": list(md.get("labels") or []),
            "table": col_owner.get(cname),
        }

    # ---- tables: keep list for binder.catalogs.tables (validator expects list) ----
    table_names: List[str] = sorted(list(entities.get("table", {}).keys()))

    # (Optional) keep richer per-table info under a side map; not required by validator
    table_meta: Dict[str, Any] = {}
    for tname, tnode in entities.get("table", {}).items():
        tmd = tnode.get("metadata") or {}
        table_meta[tname] = {
            "aliases": list(tmd.get("aliases") or []),
            "columns": list((tmd.get("columns") or {}).keys()),
        }

    # ---- functions & comparison operators ----
    def _harvest_funcs(kind: str) -> Dict[str, Any]:
        out: Dict[str, Any] = {}
        for fname, fnode in entities.get(kind, {}).items():
            b = fnode.get("binder") or {}
            md = fnode.get("metadata") or {}
            out[fname] = {
                 # binder-facing execution/binding info (not strictly required by current
validators)
                "returns_type": b.get("returns_type"),
                "class": b.get("class"),
                "clause": b.get("clause"),
                "args": list(b.get("args") or []),
                "surfaces": list(b.get("surfaces") or []),
                # useful compatibility info for future checks
```

```
                "applicable_types": md.get("applicable_types"),
                "label_rules": md.get("label_rules"),
                "aliases": list(md.get("aliases") or []),
            }
        return out

    functions: Dict[str, Any] = {}
    functions.update(_harvest_funcs("sql_actions"))
    functions.update(_harvest_funcs("postgis_actions"))

    # Keep comparison_operators separate (not required in catalogs.functions for current
checks)
    comparison_ops = _harvest_funcs("comparison_operators")

    # ---- connectors & punctuation (ensure required ones present) ----
    connectors_map, punctuation_map, conn_debug = _build_connectors_catalog(graph)

    diagnostics = graph.get("_diagnostics") or {}
    diagnostics = dict(diagnostics) if isinstance(diagnostics, dict) else {}
    diagnostics.setdefault("_binder_builder", {})["connectors_debug"] = conn_debug
    diagnostics["_table_meta"] = table_meta  # optional visibility

    # ---- Final binder in "full" shape ----
    binder: Dict[str, Any] = {
        "templates": [],  # empty is OK; validator treats as 'full' shape
        "catalogs": {
            "functions": functions,
            "columns": columns,
            "tables": table_names,         # LIST (fixes validator FAIL)
            "connectors": connectors_map,  # ensure OF/FROM/AND exist
            "punctuation": punctuation_map,
        },
          "comparison_operators": comparison_ops,  # bonus info; validator ignores this
key
        "_diagnostics": diagnostics,
    }
    return binder


# ==============================================================
# Grammar builder (parser-facing)
# ==============================================================

def _build_keyword_terminals(entities: Dict[str, Dict[str, Any]]) -> Tuple[List[str],
Dict[str, str]]:
    """
    Build terminals for prepositions/logicals and SELECT.
    We only include *canonical* tokens; aliases are normalized before parse.
    """
    lines = ['// --- High-Priority Keyword Terminals ---']
    kw_map = {}

    # Prepositions + logicals
    all_keywords = {}
```

```python
    all_keywords.update(entities.get("prepositions", {}))
    all_keywords.update(entities.get("logical_operators", {}))

    for keyword in sorted(all_keywords.keys()):
        term = keyword.upper().replace(" ", "_")
        kw_map[keyword] = term
        lines.append(f'{term}: "{keyword}"')

    # SELECT is just canonical "select"
    if "select_verbs" in entities and "select" in entities["select_verbs"]:
        lines.append('SELECT: "select"')
        kw_map["select"] = "SELECT"

    # Comma
    lines.append('COMMA: ","')
    kw_map[","] = "COMMA"

    return lines, kw_map

def build_canonical_grammar(graph: Dict[str, Any]) -> str:
    """
    Build a tiny canonical grammar from the graph.
    """
    entities = _extract_entities(graph)
    keyword_lines, _ = _build_keyword_terminals(entities)

    # Canonical terminals
    tables = sorted(entities.get("table", {}).keys())
    columns = sorted(entities.get("column", {}).keys())
    functions = sorted(list(entities.get("sql_actions", {}).keys()) +
                       list(entities.get("postgis_actions", {}).keys()))

    entity_lines = ['\n// --- CANONICAL ENTITY TERMINALS ---']
    if tables:
        tlits = [f'"{t}"' for t in tables]
        entity_lines.append(f'CANONICAL_TABLE: {" | ".join(tlits)}')
    if columns:
        clits = [f'"{c}"' for c in columns]
        entity_lines.append(f'CANONICAL_COLUMN: {" | ".join(clits)}')
    if functions:
        flits = [f'"{f}"' for f in functions]
        entity_lines.append(f'CANONICAL_FUNCTION: {" | ".join(flits)}')

    # Main rules
    main_rules = [
        '\n// --- MAIN PARSER RULES ---',
        'selectable: column_name | function_call',
        'select_statement: SELECT column_list from_clause',
        'column_name: CANONICAL_COLUMN',
        'table_name: CANONICAL_TABLE',
        'function_call: CANONICAL_FUNCTION (OF column_list)?',
        'column_list: selectable (COMMA selectable)* (COMMA? AND selectable)?',
        'from_clause: (FROM | OF) table_name',
        '\n%import common.WS',
```

```
    '%ignore WS',
]

parts = [
    'start: query',
    'query: select_statement',
    *keyword_lines,
    *entity_lines,
    *main_rules,
]
return "\n".join(parts)
```

src/n2s_generators/knowledge_generation.md

How each artifact is built

1) Vocabulary (normalizer-facing)

Goal: map every alias (single or multiword) to one or more canonical tokens the grammar understands, while staying YAML-safe and predictable.

Inputs used from the graph

* `node.metadata.aliases` for all entity types (tables, columns, select verbs, prepositions, logical/comparison ops, SQL/PostGIS actions).
* (Optionally) binder surfaces to add a few composed alias forms when theyre already curated (we still keep this conservative).

Process

1. **Collect** all aliases candidate meanings (canonical + type).
2. **Synthesize safe plurals** (last word only: idids, datedates,) for table/column aliases.
3. **Cleanups:**

    * *Preposition purity*: if an alias is exactly `of|from|in|on|at`, keep only preposition meaning.
    * *Domain preference*: collapse ambiguous spatial terms (`intersects` prefer `st_intersects` over `st_spatial_index`).
    * *Prefix protection*: if a single word (e.g., `order`) prefixes longer keys (`order id`, `order date`), drop its table meaning from the alias to reduce explosions in the normalizer.
4. **Partition**:

    * deterministic_aliases: single, unambiguous, single-word (or filler) aliases single canonical (or empty string for fillers).
    * non_deterministic_aliases: multi-word or ambiguous aliases list of canonicals.
    * **of-policy**: for a small whitelist of functions (sum/avg/distinct/st_distance) when the alias ends with of, map to `"canonical of"` to preserve structure for the grammar (prevents distinct price without of).
5. **Attach `_diagnostics`**: plural additions, collisions, domain drops, of-policy adjustments, etc.
6. **Attach `connectors`** copied from `graph["_binder_meta"]["connectors"]` (handy for denormalizer/tests).

> Why keep this simple? Because *structure* lives in the binder; the vocabulary should be predictable and reversible enough for the normalizers left-to-right BFS.

---

2) Binder (structure-facing)

Goal: everything the planner/transformer needs beyond raw tokens:

* Function/operator **signatures**: args with types, return type, class, clause

(select/where/order_by/).
* **Surfaces**: curated token patterns with placeholders (e.g., `["st_distance", "of", "{geom1}", "from", "{geom2}"]`).
* **Columns**: canonical`{type, type_category, labels, table}`.
* **Connectors**: prepositions/commas/logical connectors (for the phrase matcher).

Inputs used from the graph

* For functions/operators: `node.binder` (already created in your new graph builder).
* For columns/tables: table metadata (to recover columntable), and per-column `type`, `type_category`, `labels`.
* For connectors: `graph["_binder_meta"]["connectors"]`.

Process

* Copy/normalize per-function binder blocks from the graph.
* Build a **columns index** with its owning table (single source of truth for later SQL assembly).
* Pass through connectors and stash `_diagnostics` from the graph for easy visibility.

---

3) Canonical Grammar (parser-facing)

Goal: tiny, stable grammar over **canonical** tokens only.

* SELECT terminal includes only `"select"` (aliases map to this in the vocabulary).
* Prepositions/logicals emit upper-case terminals (`OF`, `FROM`, `AND`,) from canonical keywords.
* Canonical terminals for tables/columns/functions are the **exact canonical names** present in the graph.
* Same wide list rule you used (`column_list` with commas and optional trailing and).
* Function call syntax: `CANONICAL_FUNCTION (OF column_list)?` (clean and works with the of-policy).

What you get out of this

* **vocabulary.yaml**
  Deterministic/ND sections + connectors + fine-grained diagnostics. This keeps the normalizer robust without leaking structure into it.

* **binder.yaml**
  Pure, structure-aware signatures/surfaces/typing pulled straight from the graphs binder blocks (with columns & owners resolved). This is what your planner/renderer will love.

* **canonical_grammar.lark**
  Small, stable, and entirely canonical. All alias messiness happens before parse; all structure is resolved after parse by the binder.

src/n2s_validators/artifact_validator.py

```
import re

def validate_map_coverage(graph, norm_map, log):
    """
    V2.1: Checks that every ALIAS defined in the graph exists as a key in
    either the deterministic or non-deterministic sections of the normalization map.
    """
    log.append("--- Running V2.1: Normalization Map Coverage Check ---")

    # Combine all known aliases from BOTH sections of the map for a single lookup
    d_keys = norm_map.get('deterministic_aliases', {}).keys()
    nd_keys = norm_map.get('non_deterministic_aliases', {}).keys()
    all_map_aliases = set(d_keys) | set(nd_keys)
    missing_aliases = []

    # Iterate through the graph and check ONLY the defined aliases for each entity
    for canonical_name, node in graph.items():
        metadata = node.get('metadata', {})
        if isinstance(metadata, dict):
            for alias in metadata.get('aliases', []):
                # Check if the alias from the graph is a key in the normalization map
                if str(alias).lower() not in all_map_aliases:
                    missing_aliases.append(f"Alias '{alias}' for '{canonical_name}'")

    if not missing_aliases:
        log.append("PASS: Normalization map covers all defined aliases.")
        return True, log
    else:
        log.append("FAIL: The following aliases from the graph are missing from the
normalization map:")
        for item in sorted(missing_aliases):
            log.append(f"  - {item}")
        return False, log

def _check_terminal_vocabulary(terminal_name, canonical_names, grammar_text, log):
    """Helper function to verify all canonical names are in a grammar terminal."""
    pattern = re.compile(rf"^{terminal_name}:\s*(.*)", re.MULTILINE)
    match = pattern.search(grammar_text)

    if not match:
        log.append(f"FAIL: Terminal '{terminal_name}' is not defined in the grammar.")
        return False

    defined_literals = {item.strip() for item in match.group(1).split('|')}

    all_found = True
    for name in canonical_names:
        quoted_name = f'"{name}"'
        if quoted_name not in defined_literals:
            log.append(f"FAIL: Canonical name '{name}' is missing from the
'{terminal_name}' terminal in the grammar.")
```

```
            all_found = False

    return all_found

def validate_grammar_vocabulary(graph, grammar_text, log):
    """
    V2.2: Verifies that the canonical_grammar.lark contains a terminal
    definition for every canonical term found in the graph.
    """
    log.append("--- Running V2.2: Grammar Vocabulary Check ---")

    # 1. Extract all canonical names from the graph by entity type
    tables = {k for k, v in graph.items() if v.get('entity_type') == 'table'}
    columns = {k for k, v in graph.items() if v.get('entity_type') == 'column'}
    functions = {k for k, v in graph.items() if v.get('entity_type') in ['sql_action',
'postgis_action']}

    # 2. Run checks for each canonical terminal type
    tables_ok = _check_terminal_vocabulary('CANONICAL_TABLE', tables, grammar_text, log)
    columns_ok = _check_terminal_vocabulary('CANONICAL_COLUMN', columns, grammar_text,
log)
    functions_ok = _check_terminal_vocabulary('CANONICAL_FUNCTION', functions,
grammar_text, log)

    all_checks_passed = tables_ok and columns_ok and functions_ok

    if all_checks_passed:
        log.append("PASS: Grammar vocabulary correctly matches all canonical entities
from the graph.")

    return all_checks_passed, log
```

src/n2s_validators/binder_validator.py

```
# src/n2s_validators/binder_validator.py

from __future__ import annotations
from typing import Dict, Any, List, Tuple, Set, Iterable, Optional
from collections import defaultdict, Counter

# --------------------------------------------------------------
# Assumptions about binder structure (kept loose but explicit)
# --------------------------------------------------------------
# binder := {
#   "templates": [
#       {
#         "id": "function_call",
#         "pattern": ["CANONICAL_FUNCTION", ["OF", "column_list", "?"]],
#         "slots": [
#            {"name":"func","type":"function"},
#            {"name":"args","type":"column_list"}
#         ],
#         "constraints": { ... optional ... },
#         "cardinality": { "column_list": {"sep": ",", "allow_and": true} },
#         "connectors":  { "OF": {"required_for": "function_with_args"} },
#         "costs": {"function_with_args": 1.0, "function_without_args": 1.3}
#       },
#       ...
#   ],
#   "catalogs": {
#       "functions": {
#          "<func_name>": {
#             "entity_type": "sql_actions" | "postgis_actions",
#              "applicable_types": {"column": ["int","float","any"], "value": ["any"]}  #
optional
#             "label_rules": ["id", "not postgis"]                                     #
optional
#          },
#          ...
#       },
#       "columns": {
#          "<col_name>": {"type": "INT", "labels": ["id","postgis", ...]}
#       },
#       "tables": ["users","sales","regions"],
#       "connectors": {"OF":"of","FROM":"from","AND":"and","COMMA":","},
#       "punctuation": {",": ","}
#   }
# }
#
# graph := { canonical_name: {"entity_type": "...", "metadata": {...}} }
# vocabulary := {"deterministic_aliases": {...}, "non_deterministic_aliases": {...}}

# --------------------------------------------------------------
# Utilities
# --------------------------------------------------------------
```

```python
def _yaml_safe(obj: Any) -> bool:
    """Minimal YAML-safety heuristic: forbid tuples and non-primitive keys recursively."""
    if isinstance(obj, tuple):
        return False
    if isinstance(obj, dict):
        for k, v in obj.items():
            if isinstance(k, (list, dict, set, tuple)):
                return False
            if not _yaml_safe(v):
                return False
    elif isinstance(obj, list):
        return all(_yaml_safe(x) for x in obj)
    return True

# ---------- New helpers for binder shape/connector checks ----------

def _normalize_binder_views(binder: Dict[str, Any]) -> Tuple[List[Dict[str, Any]], Dict[str, Any], str, List[str]]:
    """
    Return (templates, catalogs, mode, notes)
    mode  {"full", "catalogs_only", "unknown"}
    - "full": binder has both 'templates' (list) and 'catalogs' (dict)
    - "catalogs_only": binder *is* a catalogs dict (or has only 'catalogs' without templates)
    - "unknown": structure doesn't match either; caller should fail
    """
    notes: List[str] = []
    if not isinstance(binder, dict):
        return [], {}, "unknown", ["binder is not a dict"]

    templates = binder.get("templates", None)
    catalogs  = binder.get("catalogs", None)

    # Full shape present
    if isinstance(templates, list) and isinstance(catalogs, dict):
        return templates, catalogs, "full", notes

    # Catalogs present without templates
    if isinstance(catalogs, dict) and templates is None:
        notes.append("binder has 'catalogs' but no 'templates' (catalogs-only mode)")
        return [], catalogs, "catalogs_only", notes

    # Flat catalogs (legacy) at top-level
    flat_keys = {"functions", "columns", "tables", "connectors", "punctuation"}
    if any(k in binder for k in flat_keys):
        notes.append("binder looks like flat catalogs (legacy); treating as catalogs-only")
        # Treat top-level as catalogs
        cands = {k: binder.get(k) for k in flat_keys if k in binder}
        # Normalize to dict shapes
        catalogs_norm: Dict[str, Any] = {
            "functions": cands.get("functions", {}) or {},
```

```
            "columns": cands.get("columns", {}) or {},
            "tables": cands.get("tables", []) or [],
            "connectors": cands.get("connectors", {}) or {},
            "punctuation": cands.get("punctuation", {}) or {},
        }
        return [], catalogs_norm, "catalogs_only", notes

    return [], {}, "unknown", ["binder missing both 'templates' and usable catalogs"]


def _collect_connector_inventory(
    binder: Dict[str, Any],
    vocabulary: Optional[Dict[str, Any]] = None,
    graph: Optional[Dict[str, Any]] = None,
) -> Dict[str, Any]:
    """
    Collect connector info from multiple sources so B5 can validate gracefully:
      - binder.catalogs.connectors (map of UPPER -> surface)
      - vocabulary.connectors (list of lowercase strings)
      - graph._binder_meta.connectors (list of lowercase strings)
    Returns a dict with sets/maps and a union for convenience.
    """
    catalogs = (binder.get("catalogs") or {}) if isinstance(binder, dict) else {}
     binder_map = (catalogs.get("connectors") or {}) if isinstance(catalogs, dict) else
{}
    if not isinstance(binder_map, dict):
        binder_map = {}

    vocab_conns = set()
    if vocabulary and isinstance(vocabulary, dict):
        vc = vocabulary.get("connectors") or []
        if isinstance(vc, list):
            vocab_conns = {str(x).strip().lower() for x in vc if str(x).strip()}

    graph_conns = set()
    if graph and isinstance(graph, dict):
        bm = graph.get("_binder_meta") or {}
        if isinstance(bm, dict):
            gc = bm.get("connectors") or []
            if isinstance(gc, list):
                graph_conns = {str(x).strip().lower() for x in gc if str(x).strip()}

    # Build a lowercase view of binder_map surfaces for easy membership checks
        binder_surfaces = {str(v).strip().lower() for v in binder_map.values() if
isinstance(v, str)}

    return {
        "binder_map": binder_map,               # expected style: {"OF":"of", ...}
        "binder_surfaces": binder_surfaces,     # {"of","from","and",...}
        "vocab_connectors": vocab_conns,        # {"of","from","and",...}
        "graph_connectors": graph_conns,        # {"of","from","and",...}
        "union": binder_surfaces | vocab_conns | graph_conns,
    }
```

```python
def _flatten_pattern(pat: Any) -> List[str]:
    """
    Turn a pattern like ["CANONICAL_FUNCTION", ["OF","column_list","?"]] into a flat,
    order-preserving signature, keeping non-terminals (like 'column_list') as-is and
    terminals as uppercase strings. Optional markers are ignored in the signature.
    """
    out: List[str] = []
    def rec(x: Any) -> None:
        if isinstance(x, str):
            out.append(x)
        elif isinstance(x, list):
            # convention: optional mark "?" appears as last element of a list form
            xs = [t for t in x if t != "?"]
            for t in xs:
                rec(t)
        else:
            out.append(str(x))
    rec(pat)
    return out

def _graph_canonicals(graph: Dict[str, Any]) -> Set[str]:
    return set(graph.keys())

def _graph_entities_by_type(graph: Dict[str, Any], etype: str) -> Set[str]:
    return {k for k, v in graph.items() if v.get("entity_type") == etype}

def _vocab_identity_ok(vocab: Dict[str, Any], token: str) -> bool:
    det = vocab.get("deterministic_aliases", {}) or {}
    return det.get(token) in (token, "", None) or det.get(token) == token

def _is_compatible(column_meta: Dict[str, Any], func_meta: Dict[str, Any]) -> bool:
    """
    Rough compatibility check (same spirit as your earlier validators):
    - applicable_types: dict(var -> [allowed_types])
      If any var includes 'column', a column is compatible if column.type in allowed OR 'any'
    - label_rules: ["id", "not postgis", ...] must be satisfied if present
    """
    if not (isinstance(column_meta, dict) and isinstance(func_meta, dict)):
        return False

    app = func_meta.get("applicable_types")
    if app and isinstance(app, dict):
        # If function expresses any constraints for 'column' var(s), enforce on this column
        # We treat any key containing 'column' as a column-var
        column_vars = [k for k in app.keys() if "column" in str(k).lower()]
        if column_vars:
            allowed_any = False
            allowed_types: Set[str] = set()
            for var in column_vars:
                allowed = app.get(var) or []
                if "any" in allowed:
```

```
                    allowed_any = True
                allowed_types.update(str(t).lower() for t in allowed if t != "any")
            col_type = (column_meta.get("type") or "").lower()
            if not (allowed_any or (col_type in allowed_types)):
                return False

    # label_rules
    # rules like "id", "not postgis" must be satisfied by column labels
    col_labels = set((column_meta.get("labels") or []))
    for rule in func_meta.get("label_rules", []) or []:
        r = str(rule)
        if r.startswith("not "):
            if r[4:] in col_labels:
                return False
        else:
            if r not in col_labels:
                return False

    return True

def _any_instantiation_for_template(
    tpl: Dict[str, Any], binder: Dict[str, Any]
) -> bool:
    """
     Conservative existence proof: for the common template shapes, try to witness at
least one
    (function, column_list, table) combo that satisfies constraints.
    """
    catalogs = binder.get("catalogs", {}) or {}
    funcs = (catalogs.get("functions") or {})
    cols  = (catalogs.get("columns") or {})
    tables = catalogs.get("tables") or []

    # Identify this as a function_call / select_stmt by looking at pattern
    sig = _flatten_pattern(tpl.get("pattern"))
    sig_str = " ".join(sig)

    # Simple heuristics:
    # - function_call: contains "CANONICAL_FUNCTION"
    # - column_list: contains "column_list"
     # - select_stmt: contains both "CANONICAL_TABLE" and "column_list" and SELECT/FROM
tokens
    has_func = any(tok == "CANONICAL_FUNCTION" for tok in sig)
    needs_cols = "column_list" in sig
    has_table = any(tok == "CANONICAL_TABLE" for tok in sig)

    if has_func:
           # pick a function, see if it either needs no columns or has at least one
compatible column
        for f_name, f_meta in funcs.items():
            f_meta = f_meta or {}
            if needs_cols:
                # require at least 1 compatible column
                for c_name, c_meta in cols.items():
```

```
                if _is_compatible(c_meta or {}, f_meta):
                    return True
            # try without strict types if no applicable_types specified
            if not f_meta.get("applicable_types"):
                if cols:
                    return True
        else:
            # no columns required by the pattern
            return True
    return False

     # If no function but needs columns (e.g., column_list template), just require >= 1
column
    if needs_cols and not cols:
        return False

    # If select_stmt-like: require both columns and tables exist
    if has_table and needs_cols:
        return bool(cols) and bool(tables)

    # Default: assume OK if template is not obviously unsatisfiable by catalogs
    return True

def _estimate_bindings(
    binder: Dict[str, Any],
    tpl: Dict[str, Any],
    cap: int = 1_000_000
) -> int:
    """
    Very rough combinatorial estimate to catch potential explosion.
    - For function_call: |functions| * max(1, |compatible_cols|)
    - For select_stmt:  |tables| * |columns|^k   (k is 12 based on pattern guess)
    """
    catalogs = binder.get("catalogs", {}) or {}
    funcs = (catalogs.get("functions") or {})
    cols  = (catalogs.get("columns") or {})
    tables = catalogs.get("tables") or []

    sig = _flatten_pattern(tpl.get("pattern"))
    has_func = "CANONICAL_FUNCTION" in sig
    needs_cols = "column_list" in sig
    has_table = "CANONICAL_TABLE" in sig

    if has_func:
        # count compatible columns per function (cap it for sanity)
        total = 0
        for f_name, f_meta in funcs.items():
            f_meta = f_meta or {}
            if not needs_cols:
                total += 1
            else:
                if not cols:
                    continue
                comp = 0
```

```
            for _, c_meta in cols.items():
                if _is_compatible(c_meta or {}, f_meta):
                    comp += 1
                    if comp > cap:
                        return cap
            total += max(1, comp)  # at least 1 if column_list optional
        if total > cap:
            return cap
    return total

if has_table and needs_cols:
    # If we cant read list-length/cardinality, assume k=1.5 avg
    k = 2 if "AND" in sig or "COMMA" in sig else 1
    base = len(tables) * (max(1, len(cols)) ** k)
    return min(base, cap)

if needs_cols:
    return max(1, len(cols))

# default low
return 1

def _collect_vocab_keys(vocab: Dict[str, Any]) -> Tuple[Set[str], Set[str]]:
    det = vocab.get("deterministic_aliases", {}) or {}
    nd  = vocab.get("non_deterministic_aliases", {}) or {}
    return set(det.keys()), set(nd.keys())

# ---------------------------------------------------------------
# B1. Shape & schema of binder templates
# ---------------------------------------------------------------

def  validate_binder_shape(binder:  Dict[str,  Any],  log:  List[str])  ->  Tuple[bool,
List[str]]:
    log.append("--- Running B1: Binder Shape & Schema ---")
    ok = True

    templates, catalogs, mode, notes = _normalize_binder_views(binder)

    for n in notes:
        log.append(f"DEBUG:B1 {n}")

    if mode == "unknown":
            log.append("FAIL:  Binder  must  contain  either  {templates  +  catalogs}  or  a
catalogs-only shape.")
        return False, log

    # Validate catalogs shape in all modes
    if not isinstance(catalogs.get("functions", {}), dict):
        ok = False; log.append("FAIL: binder.catalogs.functions must be a dict.")
    if not isinstance(catalogs.get("columns", {}), dict):
        ok = False; log.append("FAIL: binder.catalogs.columns must be a dict.")
    if not isinstance(catalogs.get("tables", []), list):
        ok = False; log.append("FAIL: binder.catalogs.tables must be a list.")
                  if     catalogs.get("connectors")     is     not     None     and     not
```

```
isinstance(catalogs.get("connectors"), dict):
            ok = False; log.append("FAIL: binder.catalogs.connectors must be a dict if provided.")
        if catalogs.get("punctuation") is not None and not isinstance(catalogs.get("punctuation"), dict):
            ok = False; log.append("FAIL: binder.catalogs.punctuation must be a dict if provided.")

    # If catalogs-only, treat lack of templates as a WARNING instead of a FAIL
    if mode == "catalogs_only":
        log.append("WARNING: Binder has no 'templates'; running in catalogs-only mode (B3/B4 ambiguity checks may be limited).")
        if ok:
            log.append("PASS: Catalogs-only binder shape is valid.")
        return ok, log

    # Full mode  validate templates too
    required_tpl_keys = {"id", "pattern", "slots"}
    allowed_slot_types = {"function", "column", "column_list", "table"}

    for i, tpl in enumerate(templates or []):
        if not isinstance(tpl, dict):
            ok = False; log.append(f"FAIL: Template #{i} must be a dict."); continue
        missing = required_tpl_keys - set(tpl.keys())
        if missing:
            ok = False; log.append(f"FAIL: Template '{tpl.get('id','<no-id>')}' missing keys: {sorted(missing)}")

        # pattern
        if "pattern" in tpl and not isinstance(tpl["pattern"], (list, str)):
            ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' pattern must be list/str.")

        # slots
        slots = tpl.get("slots", [])
        if not isinstance(slots, list) or not all(isinstance(s, dict) for s in slots):
            ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' slots must be a list of dicts.")
        else:
            for s in slots:
                if "name" not in s or "type" not in s:
                    ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' slot missing 'name' or 'type'.")
                elif s["type"] not in allowed_slot_types:
                    ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' slot '{s.get('name')}' has unsupported type '{s.get('type')}'.")

        # cardinality/connectors/costs (optional but typed)
        if "cardinality" in tpl and not isinstance(tpl["cardinality"], dict):
            ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' cardinality must be a dict.")
        if "connectors" in tpl and not isinstance(tpl["connectors"], dict):
            ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' connectors must be a dict.")
```

```
        if "costs" in tpl:
            if not isinstance(tpl["costs"], dict) or not all(isinstance(v, (int, float)) for v in tpl["costs"].values()):
                ok = False; log.append(f"FAIL: Template '{tpl.get('id')}' costs must be dict[str, number].")

    if ok:
        log.append("PASS: Binder templates and catalogs have valid shapes.")
    return ok, log


# --------------------------------------------------------------
# B2. Linkage to graph/vocabulary
# --------------------------------------------------------------

def validate_binder_linkage(
    binder: Dict[str, Any],
    graph: Dict[str, Any],
    vocabulary: Dict[str, Any],
    log: List[str]
) -> Tuple[bool, List[str]]:
    log.append("--- Running B2: Binder Linkage to Graph/Vocabulary ---")
    ok = True

    catalogs = binder.get("catalogs", {}) or {}
    g_canon = _graph_canonicals(graph)
    g_funcs = _graph_entities_by_type(graph, "sql_actions") | _graph_entities_by_type(graph, "postgis_actions")
    g_cols   = _graph_entities_by_type(graph, "column")
    g_tabs   = _graph_entities_by_type(graph, "table")

    # Functions in binder must exist in graph and have identity in vocabulary
    missing_funcs: List[str] = []
    bad_vocab_funcs: List[str] = []
    for fname in (catalogs.get("functions") or {}).keys():
        if fname not in g_funcs:
            missing_funcs.append(fname)
        elif not _vocab_identity_ok(vocabulary, fname):
            bad_vocab_funcs.append(fname)

    # Columns in binder must exist in graph and have identity in vocabulary
    missing_cols: List[str] = []
    bad_vocab_cols: List[str] = []
    for cname in (catalogs.get("columns") or {}).keys():
        if cname not in g_cols:
            missing_cols.append(cname)
        elif not _vocab_identity_ok(vocabulary, cname):
            bad_vocab_cols.append(cname)

    # Tables in binder must exist in graph and have identity in vocabulary
    missing_tabs: List[str] = []
    bad_vocab_tabs: List[str] = []
    for tname in (catalogs.get("tables") or []):
        if tname not in g_tabs:
```

```
            missing_tabs.append(tname)
        elif not _vocab_identity_ok(vocabulary, tname):
            bad_vocab_tabs.append(tname)

    # Connectors referenced in patterns should exist in vocabulary (deterministic)
    det_keys, nd_keys = _collect_vocab_keys(vocabulary)
    vocab_keys = det_keys | nd_keys
    missing_connectors: Set[str] = set()
    for tpl in binder.get("templates", []) or []:
        sig = _flatten_pattern(tpl.get("pattern"))
        for tok in sig:
                # Only check obvious connector tokens (uppercase words that are *not*
catalog terminals)
            if tok in {"OF","FROM","AND","COMMA"}:
                # Expect the surface form (lowercase) to exist as an alias
                lower = tok.lower() if tok != "COMMA" else ","
                if lower not in vocab_keys:
                    missing_connectors.add(lower)

    def _report(label: str, items: List[str]) -> None:
        if items:
            nonlocal ok
            ok = False
            log.append(f"FAIL: {label}:")
            for s in sorted(items)[:100]:
                log.append(f"  - {s}")
            if len(items) > 100:
                log.append(f"  ... +{len(items)-100} more")

    _report("Functions missing in graph", missing_funcs)
    _report("Functions lacking identity mapping in vocabulary", bad_vocab_funcs)
    _report("Columns missing in graph", missing_cols)
    _report("Columns lacking identity mapping in vocabulary", bad_vocab_cols)
    _report("Tables missing in graph", missing_tabs)
    _report("Tables lacking identity mapping in vocabulary", bad_vocab_tabs)

    if missing_connectors:
        ok = False
            log.append("FAIL: Connectors referenced in binder patterns are missing in
vocabulary:")
        for c in sorted(missing_connectors):
            log.append(f"  - '{c}'")

    if ok:
            log.append("PASS: Binder links cleanly to graph entities and vocabulary
identities.")
    return ok, log

# ---------------------------------------------------------------
# B3. Unifiability & type satisfaction
# ---------------------------------------------------------------

def validate_binder_unifiability(
    binder: Dict[str, Any], log: List[str]
```

```
) -> Tuple[bool, List[str]]:
    log.append("--- Running B3: Binder Unifiability & Type Satisfaction ---")
    ok = True

    dead: List[str] = []
    for tpl in binder.get("templates", []) or []:
        if not _any_instantiation_for_template(tpl, binder):
            dead.append(str(tpl.get("id", "<no-id>")))

    if dead:
        ok = False
           log.append("FAIL: Templates that cannot be instantiated with catalogs (dead
templates):")
        for t in sorted(dead):
            log.append(f"  - {t}")
    else:
        log.append("PASS: Every template has at least one legal instantiation.")
    return ok, log

# --------------------------------------------------------------
# B4. Ambiguity & cost model
# --------------------------------------------------------------

def validate_binder_ambiguity_cost(
    binder: Dict[str, Any],
    log: List[str],
    *,
    warn_threshold: int = 50_000,
    fail_threshold: int = 1_000_000
) -> Tuple[bool, List[str]]:
    log.append("--- Running B4: Ambiguity & Cost Model ---")
    ok = True

    too_many_warn: List[Tuple[str, int]] = []
    too_many_fail: List[Tuple[str, int]] = []
    missing_costs: List[str] = []

    for tpl in binder.get("templates", []) or []:
        est = _estimate_bindings(binder, tpl, cap=fail_threshold)
        tid = str(tpl.get("id", "<no-id>"))
        if est >= fail_threshold:
            too_many_fail.append((tid, est))
        elif est >= warn_threshold:
            too_many_warn.append((tid, est))
        # Suggest costs if pattern looks combinatorial and costs missing
        if est >= warn_threshold and not isinstance(tpl.get("costs"), dict):
            missing_costs.append(tid)

    if too_many_fail:
        ok = False
                    log.append(f"FAIL: Templates with extreme binding explosion (
{fail_threshold}):")
        for tid, est in sorted(too_many_fail):
            log.append(f"  - {tid}: ~{est} bindings")
```

```
    if too_many_warn:
        log.append(f"WARNING: Templates with high binding counts ( {warn_threshold}):")
        for tid, est in sorted(too_many_warn):
            log.append(f"  - {tid}: ~{est} bindings")

    if missing_costs:
        log.append("WARNING: Consider adding 'costs' to heavily ambiguous templates:")
        for tid in sorted(missing_costs):
            log.append(f"  - {tid}")

    if ok:
              log.append("PASS: Ambiguity levels acceptable (or actionable warnings
emitted).")
    return ok, log

# ------------------------------------------------------------
# B5. Connector/'OF' rules sanity
# ------------------------------------------------------------

def validate_binder_connector_rules(
    binder: Dict[str, Any],
    log: List[str],
    *,
    vocabulary: Optional[Dict[str, Any]] = None,
    graph: Optional[Dict[str, Any]] = None,
) -> Tuple[bool, List[str]]:
    log.append("--- Running B5: Connector / 'OF' Rules Sanity ---")
    ok = True

    # Collect connector sources
    inv = _collect_connector_inventory(binder, vocabulary=vocabulary, graph=graph)
    binder_map       = inv["binder_map"]
    binder_surfaces  = inv["binder_surfaces"]
    vocab_connectors = inv["vocab_connectors"]
    graph_connectors = inv["graph_connectors"]
    union_connectors = inv["union"]

    # Expected connector *surfaces* we want available somewhere
    expected_surfaces = {"of", "from", "and"}
         missing_anywhere = sorted([c for c in expected_surfaces if c not in
union_connectors])

    # Prefer binder.catalogs.connectors to carry an explicit map (UPPER -> surface)
    needed_upper = {"OF": "of", "FROM": "from", "AND": "and"}
    missing_in_binder_map = sorted([k for k, surf in needed_upper.items()
                                        if binder_map.get(k) not in {surf, surf.upper(),
surf.title()}])

    # Report logic:
    # - If an expected surface is missing *everywhere*  FAIL.
     # - If binder_map is missing entries but the surface exists in vocab/graph  WARN
(recoverable).
    if missing_anywhere:
```

```
        ok = False
            log.append("FAIL: Missing expected connectors in any artifact (binder/vocab/graph):")
        for s in missing_anywhere:
            log.append(f"  - {s.upper()} (surface '{s}')")

    # Warn if binder map lacks explicit entries, but we *do* have those surfaces in other artifacts.
    if missing_in_binder_map and not missing_anywhere:
        log.append("WARNING: binder.catalogs.connectors is missing explicit entries (present in vocab/graph though):")
        for k in missing_in_binder_map:
            expect = needed_upper[k]
            log.append(f"  - {k}: expected surface '{expect}'")
        log.append("  suggestion: add an explicit map in binder.catalogs.connectors, e.g., {'OF':'of','FROM':'from','AND':'and'}")

    # Pattern sanity (only if templates exist)
    templates = binder.get("templates") if isinstance(binder, dict) else None
    if isinstance(templates, list) and templates:
        dup_viol: List[str] = []
        for tpl in templates:
            sig = _flatten_pattern(tpl.get("pattern"))
            for i in range(1, len(sig)):
                if sig[i] in {"OF","FROM","AND","COMMA"} and sig[i-1] == sig[i]:
                    dup_viol.append(str(tpl.get("id", "<no-id>")))
                    break

            if "column_list" in sig:
                card = tpl.get("cardinality", {})
                cl = card.get("column_list", {})
                if not isinstance(cl, dict) or "sep" not in cl:
                    log.append(f"WARNING: Template '{tpl.get('id')}' uses 'column_list' but has no explicit cardinality (sep/allow_and).")

        if dup_viol:
            ok = False
            log.append("FAIL: Templates contain consecutive duplicate connectors:")
            for t in sorted(dup_viol):
                log.append(f"  - {t}")

    # Debug summary
    log.append(
        "DEBUG:B5 connector_sources: "
        f"binder_map_keys={sorted(list(binder_map.keys()))}, "
        f"binder_surfaces={sorted(list(binder_surfaces))}, "
        f"vocab_connectors={sorted(list(vocab_connectors))}, "
        f"graph_connectors={sorted(list(graph_connectors))}"
    )

    if ok:
        log.append("PASS: Connector rules look sane (with possible warnings).")
    return ok, log
```

```
# ------------------------------------------------------------
# B6. Dead / overlapping templates
# ------------------------------------------------------------

def validate_binder_dead_overlapping(
    binder: Dict[str, Any], log: List[str]
) -> Tuple[bool, List[str]]:
    log.append("--- Running B6: Dead / Overlapping Templates ---")
    ok = True

      # Dead templates already reported in B3; here we focus on overlapping shape
signatures.
    sig_map: Dict[Tuple[str, ...], List[str]] = defaultdict(list)
    for tpl in binder.get("templates", []) or []:
        sig = tuple(_flatten_pattern(tpl.get("pattern")))
        tid = str(tpl.get("id", "<no-id>"))
        sig_map[sig].append(tid)

    overlaps = {sig: ids for sig, ids in sig_map.items() if len(ids) > 1}
    if overlaps:
        log.append("WARNING: Overlapping templates (identical pattern signatures):")
        for sig, ids in overlaps.items():
            log.append(f"  - signature={list(sig)}  templates={sorted(ids)}")

      # If a binder provides an 'enabled' flag or 'priority', you could also detect
shadowing;
    # we just signal identical shapes for now.
    log.append("PASS: Overlap analysis complete (warnings reported as needed).")
    return True, log

# ------------------------------------------------------------
# Aggregator
# ------------------------------------------------------------

def validate_binder_all(
    binder: Dict[str, Any],
    graph: Dict[str, Any],
    vocabulary: Dict[str, Any],
    log: List[str],
    *,
    warn_threshold: int = 50_000,
    fail_threshold: int = 1_000_000
) -> Tuple[bool, List[str]]:
    """
    Run B1B6 for binder QA.
    """
    results: List[bool] = []

    ok1, log = validate_binder_shape(binder, log);
results.append(ok1)
    ok2, log = validate_binder_linkage(binder, graph, vocabulary, log);
results.append(ok2)
    ok3, log = validate_binder_unifiability(binder, log);
results.append(ok3)
```

```
                    ok4,     log     =     validate_binder_ambiguity_cost(binder,     log,
warn_threshold=warn_threshold, fail_threshold=fail_threshold); results.append(ok4)
     ok5, log = validate_binder_connector_rules(binder, log);
results.append(ok5)
     ok6, log = validate_binder_dead_overlapping(binder, log);
results.append(ok6)

    overall_ok = all(results)
    if overall_ok:
        log.append("PASS: Binder validations (B1B6) passed.")
    else:
        log.append("FAIL: One or more binder validations failed.")
    return overall_ok, log
```

src/n2s_validators/cross_artifact_validator.py

```
# src/n2s_validators/cross_artifact_validator.py
from __future__ import annotations
import re
import random
from typing import Dict, Any, List, Tuple, Optional, Set, Union

from lark import Lark, ParseError, UnexpectedToken

# Reuse the canonical SmartGenerator/Analyzer for phrase generation
from src.n2s_validators.grammar_validator import GrammarAnalyzer, SmartGenerator

# ------------------------------------------------------------------------------
# Helpers: canonicals & tokenization
# ------------------------------------------------------------------------------

def _canonicals(graph: Dict[str, Any]) -> Dict[str, Set[str]]:
    tables = {k for k, v in graph.items() if v.get("entity_type") == "table"}
    cols   = {k for k, v in graph.items() if v.get("entity_type") == "column"}
    funcs  = {k for k, v in graph.items() if v.get("entity_type") in ("sql_actions",
"postgis_actions")}
    return {"tables": tables, "columns": cols, "functions": funcs}

_TOKENIZER = re.compile(r',|[A-Za-z0-9_]+')

def _tok(s: str) -> List[str]:
    # Canonical phrases are already clean; keep comma as a separate token
    return _TOKENIZER.findall(s)

def _join_tokens(tokens: List[str]) -> str:
    out: List[str] = []
    for t in tokens:
        if t == ",":
            out.append(",")
        else:
            if out and out[-1] != ",":
                out.append(" ")
            out.append(t)
    return "".join(out).strip()

# ------------------------------------------------------------------------------
# Compatibility checks (rough, but useful for feasibility)
# ------------------------------------------------------------------------------

def _is_compatible(column_md: Dict[str, Any], func_md: Dict[str, Any]) -> bool:
    """
    Minimal compatibility check:
      - If func has applicable_types, at least one arg column type must match any
allowed type bucket.
      - label_rules: all positive labels must be present; no negated labels present.
    """
    if not (isinstance(column_md, dict) and isinstance(func_md, dict)):
```

```
        return True  # be permissive in validator

    app = func_md.get("applicable_types")
    if not isinstance(app, dict) or not app:
        # No constraints stated; assume OK
        type_ok = True
    else:
        col_type = str(column_md.get("type", "")).lower()
        type_ok = False
        for _var, allowed in app.items():
            if not isinstance(allowed, list):
                continue
            allowed_lc = {str(a).lower() for a in allowed}
            if "any" in allowed_lc or col_type in allowed_lc:
                type_ok = True
                break

    labels = {str(x).lower() for x in column_md.get("labels", []) if isinstance(x, str)}
    rules  = [str(r) for r in func_md.get("label_rules", []) if isinstance(r, str)]
    labels_ok = True
    for r in rules:
        if r.startswith("not "):
            if r[4:].lower() in labels:
                labels_ok = False; break
        else:
            if r.lower() not in labels:
                labels_ok = False; break

    return bool(type_ok and labels_ok)

def _get_func_meta(graph: Dict[str, Any], name: str) -> Dict[str, Any]:
    node = graph.get(name, {})
    md = node.get("metadata", {}) if isinstance(node, dict) else {}
    return md if isinstance(md, dict) else {}

def _get_col_meta(graph: Dict[str, Any], name: str) -> Dict[str, Any]:
    node = graph.get(name, {})
    md = node.get("metadata", {}) if isinstance(node, dict) else {}
    return md if isinstance(md, dict) else {}

# ------------------------------------------------------------------------------
# Minimal binder for canonical phrases (used for validation only)
# ------------------------------------------------------------------------------

class BindError(Exception): ...

Selectable = Union[
    Dict[str, str],  # {"type":"column","name":...}
    Dict[str, Any],  # {"type":"func","name":...,"args":[Selectable,...]}
]

class CanonicalBinder:
    """
    Binds canonical token streams shaped like:
```

```
      SELECT column_list [FROM|OF table]

    where:
       column_list   := selectable ( (',' selectable)* (','? 'and' selectable)? | 'and'
selectable )?
      selectable    := column | function_call
      function_call := function ['of' column_list]

    Parameters
    ----------
    strict_types : bool
        If True, type/label incompatibilities raise BindError.
        If False, incompatibilities are tolerated (and optionally coerced).
    coerce_types : bool
        If True (and not strict), replace incompatible arg columns with a
        compatible column from the current table (or globally as fallback).
    allow_ordering_funcs_in_args : bool
        If True, allow order_by_asc / order_by_desc as *argument* functions.
    """

    ARG_FUNC_DENY: Set[str] = {"order_by_asc", "order_by_desc"}

    def __init__(
        self,
        graph: Dict[str, Any],
        *,
        strict_types: bool = True,
        coerce_types: bool = False,
        allow_ordering_funcs_in_args: bool = False,
    ):
        c = _canonicals(graph)
        self.tables: Set[str] = c["tables"]
        self.columns: Set[str] = c["columns"]
        self.functions: Set[str] = c["functions"]
        self.graph = graph

        self.strict_types = bool(strict_types)
        self.coerce_types = bool(coerce_types)
        self.allow_ordering_funcs_in_args = bool(allow_ordering_funcs_in_args)

        # table -> [columns], and inverse
        self.table_columns: Dict[str, List[str]] = self._build_table_columns(graph)
        self.column_owner: Dict[str, str] = {
            col: t for t, cols in self.table_columns.items() for col in cols
        }

    # ---------- graph helpers ----------

    def _build_table_columns(self, graph: Dict[str, Any]) -> Dict[str, List[str]]:
        out: Dict[str, List[str]] = {}
        for tname, tnode in graph.items():
            if tnode.get("entity_type") != "table":
                continue
```

```
            cols = ((tnode.get("metadata") or {}).get("columns") or {})
            out[tname] = list(cols.keys())
        return out

     def  _first_compatible_col_for_func_in_table(self,  fn:  str,  tname:  str)  ->
Optional[str]:
        fn_md = _get_func_meta(self.graph, fn)
        for c in self.table_columns.get(tname, []):
            if _is_compatible(_get_col_meta(self.graph, c), fn_md):
                return c
        cols = self.table_columns.get(tname, [])
        return cols[0] if cols else None

    def _first_compatible_col_global(self, fn: str) -> Optional[str]:
        fn_md = _get_func_meta(self.graph, fn)
        for col in self.columns:
            if _is_compatible(_get_col_meta(self.graph, col), fn_md):
                return col
        # fallback: any column at all
        return next(iter(self.columns), None)

    def _any_table(self) -> Optional[str]:
        return next(iter(self.tables)) if self.tables else None

    # ---------- recursive descent over tokens ----------

    def bind(self, tokens: List[str], recorder: Optional[List[str]] = None) -> Dict[str,
Any]:
        """
        Parse a canonical token stream into a minimal binding using the original
        list/connector behavior, with optional flight recording (recorder).
        """
        pos = 0  # cursor over tokens

        def rec(msg: str) -> None:
            if recorder is not None:
                recorder.append(f"[pos={pos}] {msg}")

        def expect(token: str) -> None:
            nonlocal pos
            if pos >= len(tokens) or tokens[pos] != token:
                found = tokens[pos] if pos < len(tokens) else 'EOF'
                rec(f"EXPECT FAIL: wanted '{token}', found '{found}'")
                raise BindError(f"expected '{token}' at {pos}, found '{found}'")
            rec(f"EXPECT OK: '{token}'")
            pos += 1

        def peek() -> Optional[str]:
            return tokens[pos] if pos < len(tokens) else None

        def at_end() -> bool:
            return pos >= len(tokens)

        def parse_selectable() -> Selectable:
```

```
            """column_name | function_call (function may be zero-arg if no 'of')"""
            nonlocal pos
            t = peek()
            rec(f"parse_selectable: lookahead='{t}'")
            if t is None:
                raise BindError("unexpected end while parsing selectable")

            if t in self.columns:
                pos += 1
                rec(f"SELECTABLE = column '{t}'")
                return {"type": "column", "name": t}

            if t in self.functions:
                pos += 1
                rec(f"SELECTABLE = function '{t}'")
                fn = {"type": "func", "name": t, "args": []}
                if peek() == "of":
                    expect("of")
                    rec("function has 'of'  parse argument list")
                    fn["args"] = parse_column_list()
                else:
                    rec("function without 'of' (zero-arg call)")
                return fn

            rec(f"SELECTABLE FAIL: '{t}' not column/function")
            raise BindError(f"token '{t}' is neither a column nor a function")

        def parse_column_list() -> List[Selectable]:
            """selectable ( , selectable )* ( ,? AND selectable )?"""
            nonlocal pos
            rec("parse_column_list: ENTER")
            items: List[Selectable] = [parse_selectable()]
            rec(f"parse_column_list: first item parsed; next='{peek()}'")

            # Comma chain
            while peek() == ",":
                expect(",")
                rec(f"comma-branch: parse another selectable; next='{peek()}'")
                items.append(parse_selectable())

              # Optional Oxford comma: try consuming a comma; if not followed by 'and',
rewind.
            if peek() == ",":
                save = pos
                expect(",")
                if peek() == "and":
                    expect("and")
                    rec("Oxford ', and' tail: parse final selectable")
                    items.append(parse_selectable())
                else:
                    rec("not an Oxford ', and'  rewind")
                    pos = save

            # Or bare 'and' tail
```

```
            if peek() == "and":
                expect("and")
                rec("'and' tail: parse final selectable")
                items.append(parse_selectable())

            rec(f"parse_column_list: EXIT with {len(items)} item(s)")
            return items

        # ---- SELECT ----
        rec(f"BEGIN bind; tokens={tokens}")
        expect("select")

        selectables = parse_column_list()

        # ---- FROM | OF table ----
        nxt = peek()
        rec(f"post-selectables next token='{nxt}'")
        if nxt not in {"from", "of"}:
            raise BindError(f"expected 'from' or 'of' before table, found '{nxt}'")
        pos += 1
        rec(f"connector consumed: '{nxt}'")

        tbl = peek()
        if tbl not in self.tables:
            raise BindError(f"expected table after '{nxt}', found '{tbl}'")
        pos += 1
        rec(f"table = '{tbl}'")

        if not at_end():
            trailing = tokens[pos:]
            rec(f"TRAILING TOKENS: {trailing}")
            raise BindError(f"unexpected trailing tokens starting at {pos}: {trailing}")

        # Type compatibility checks (no semantic changes, just record)
        for sel in selectables:
            self._check_selectable_types(sel, recorder=recorder)

        binding = {
            "template_id": "select_cols_from_table",
            "table": tbl,
            "selectables": selectables,
        }
        rec("BIND SUCCESS")
        return binding



    # ---------- type/label checks (with optional coercion) ----------

     def _check_selectable_types(self, sel: Selectable, recorder: Optional[List[str]] =
None) -> None:
        def rec(msg: str) -> None:
            if recorder is not None:
                recorder.append(f"[typecheck] {msg}")
```

```
        if sel.get("type") == "column":
            rec(f"column '{sel['name']}' OK (no checks)")
            return

        if sel.get("type") == "func":
            fn = sel["name"]
            fn_md = _get_func_meta(self.graph, fn)
            args = sel.get("args", [])
            if not args:
                rec(f"function '{fn}' has no args; skip typecheck (ok if zero-arity)")
                return

            for arg in args:
                if arg.get("type") == "column":
                    col_name = arg["name"]
                    col_md = _get_col_meta(self.graph, col_name)
                    type_ok = _is_compatible(col_md, fn_md)
                    rec(f"func '{fn}'  arg column '{col_name}': compatible={type_ok}")
                    if not type_ok:
                            raise BindError(f"incompatible arg '{col_name}' for function
'{fn}'")
                elif arg.get("type") == "func":
                    rec(f"func '{fn}'  nested func '{arg.get('name')}'  recurse")
                    self._check_selectable_types(arg, recorder=recorder)
                else:
                    rec(f"func '{fn}'  unknown arg kind; ignored")




# ------------------------------------------------------------------------------
# Serialization (canonical; used to ensure binder+grammar agreement)
# ------------------------------------------------------------------------------

def _serialize_selectable(sel: Selectable) -> str:
    if sel["type"] == "column":
        return sel["name"]
    # function
    name = sel["name"]
    args = sel.get("args", [])
    if not args:
        return name
    return f"{name} of {_serialize_column_list(args)}"

def _serialize_column_list(items: List[Selectable]) -> str:
    if not items:
        return ""
    if len(items) == 1:
        return _serialize_selectable(items[0])
    if len(items) == 2:
                                return    f"{_serialize_selectable(items[0])}    and
{_serialize_selectable(items[1])}"
    # 3+ => commas + and
```

```
    head = ", ".join(_serialize_selectable(x) for x in items[:-1])
    return f"{head}, and {_serialize_selectable(items[-1])}"

def serialize_binding(binding: Dict[str, Any]) -> str:
    cols = _serialize_column_list(binding["selectables"])
    tbl  = binding["table"]
    return f"select {cols} from {tbl}"

# ------------------------------------------------------------------------------
# C1. Canonical  Binder  Grammar roundtrip
# ------------------------------------------------------------------------------

def validate_canonical_roundtrip(
    graph: Dict[str, Any],
    grammar_text: str,
    log: List[str],
    *,
    phrases: int = 100,
    success_threshold: float = 0.95,
) -> Tuple[bool, List[str]]:
    log.append("--- Running C1: Canonical  Binder  Grammar Roundtrip ---")

    # Grammar & generator for canonical phrases
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        log.append(f"FAIL: Could not build parser: {e}")
        return False, log

    analyzer = GrammarAnalyzer(parser)
    gen = SmartGenerator(parser, graph, analyzer)

    # IMPORTANT: relaxed binder for round-trip (structure > semantics)
    binder = CanonicalBinder(
        graph,
        strict_types=False,                # do not fail on type/label mismatches
        coerce_types=True,                 # try to swap in compatible columns
        allow_ordering_funcs_in_args=True  # permit order_by_* inside arg lists
    )

    ok = 0
    fail = 0
    examples: List[str] = []

    for _ in range(phrases):
        canonical, _ = gen.generate()
        if not canonical:
            fail += 1
            if len(examples) < 3:
                examples.append("GENERATOR_FAILED")
            continue

        tokens = _tok(canonical)
        try:
```

```
            bound = binder.bind(tokens)
            rebuilt = serialize_binding(bound)
            parser.parse(rebuilt)  # should parse as canonical
            ok += 1
        except Exception as e:
            fail += 1
            if len(examples) < 3:
                examples.append(f"Input='{canonical}' :: Error={e}")

    total = ok + fail
    if total == 0:
        log.append("WARNING: No phrases generated.")
        return True, log

    rate = ok / total
    log.append(f"  - Success Rate: {rate:.0%} ({ok}/{total})")
    if examples:
        log.append("  - Sample failures:")
        for ex in examples:
            log.append(f"    * {ex}")

    if rate >= success_threshold:
        log.append("PASS: Binder and grammar agree on canonical shapes.")
        return True, log
    else:
        log.append("FAIL: Roundtrip rate below threshold.")
        return False, log

# ------------------------------------------------------------------------------
# C2. Binder SQL feasibility (lightweight)
# ------------------------------------------------------------------------------

_PLACEHOLDER_RE = re.compile(r"\{([A-Za-z0-9_]+)\}")

def _first_compatible_col(graph: Dict[str, Any], func_name: str) -> Optional[str]:
    """Pick the first column compatible with the function (best-effort)."""
    fn_md = _get_func_meta(graph, func_name)
    for col in (k for k, v in graph.items() if v.get("entity_type") == "column"):
        if _is_compatible(_get_col_meta(graph, col), fn_md):
            return col
    return None

def _any_table(graph: Dict[str, Any]) -> Optional[str]:
    for t, v in graph.items():
        if v.get("entity_type") == "table":
            return t
    return None

def validate_binder_sql_feasibility(
    graph: Dict[str, Any],
    log: List[str],
    *,
    sample_functions: int = 50
) -> Tuple[bool, List[str]]:
```

```
    log.append("--- Running C2: Binder SQL Feasibility (Light) ---")

    # Iterate over a sample of functions and ensure each template is fillable
    funcs = [k for k, v in graph.items() if v.get("entity_type") in ("sql_actions",
"postgis_actions")]
    random.shuffle(funcs)
    funcs = funcs[:sample_functions] if sample_functions and len(funcs) >
sample_functions else funcs

    problems: List[str] = []
    table = _any_table(graph)

    for fn in funcs:
        md = _get_func_meta(graph, fn)
        tmpl = md.get("template")
        if not isinstance(tmpl, str):
            # Not all functions need a SQL template; warn but don't fail
            continue

        placeholders = set(_PLACEHOLDER_RE.findall(tmpl))
        # Build a minimal binding universe we could feed into a future SQL generator
        # Common placeholders seen in examples: {column}, {value}, {columns}, etc.
        env: Dict[str, Any] = {}
        if "column" in placeholders or "columns" in placeholders:
            col = _first_compatible_col(graph, fn)
            if not col:
                problems.append(f"{fn}: no compatible column found for template
placeholders {placeholders}")
                continue
            env["column"] = col
            env["columns"] = [col]
        if "table" in placeholders:
            if not table:
                problems.append(f"{fn}: no table available for template with {{table}}")
                continue
            env["table"] = table
        if "value" in placeholders:
            # We cannot guess a runtime value here; but template presence is okay.
            env["value"] = "__VALUE__"

        # Minimal feasibility = all placeholders have something non-empty to bind
        missing = [p for p in placeholders if p not in env or env[p] in (None, [], "")]
        if missing:
            problems.append(f"{fn}: unbound placeholders {missing}")

    if problems:
        log.append("WARNING: Some function templates are not trivially fillable:")
        for p in problems[:10]:
            log.append(f"  - {p}")
        if len(problems) > 10:
            log.append(f"  ... +{len(problems)-10} more")
        # Keep as non-fatal (optional check)
    else:
        log.append("PASS: Sampled function templates appear fillable with
```

```
binder-produced slots.")

    return True, log

# ------------------------------------------------------------------------------
# C3. Negative canonical tests
# ------------------------------------------------------------------------------

def _make_negative_examples(graph: Dict[str, Any], k: int = 8) -> List[str]:
    c = _canonicals(graph)
    if not (c["tables"] and c["columns"]):
        return []
    tbl = random.choice(list(c["tables"]))
    col = random.choice(list(c["columns"]))
    fn  = random.choice(list(c["functions"])) if c["functions"] else None

    cases = [
        f"select {col} {tbl}",                 # missing FROM/OF
        f"select , {col} from {tbl}",          # leading comma in list
        f"select and {col} from {tbl}",        # leading 'and'
        f"select {col} and , {col} from {tbl}",# bad ', and' order
        f"select from {tbl}",                  # empty select list
    ]
    if fn:
        cases += [
            f"select {fn} {col} from {tbl}",       # function missing 'of'
            f"select {fn} of from {tbl}",          # missing arg after 'of'
              f"select {fn} of {fn} of {col} from {tbl}",  # nested fn without args for
inner 'fn'
        ]
    # Deduplicate & sample
    uniq = list(dict.fromkeys(cases))
    random.shuffle(uniq)
    return uniq[:k]

def validate_negative_canonical(
    graph: Dict[str, Any],
    grammar_text: str,
    log: List[str],
    *,
    examples: int = 8
) -> Tuple[bool, List[str]]:
    log.append("--- Running C3: Negative Canonical Tests ---")
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        log.append(f"FAIL: Could not build parser: {e}")
        return False, log

    binder = CanonicalBinder(graph)
    negs = _make_negative_examples(graph, k=examples)
    if not negs:
        log.append("WARNING: Not enough canonicals to craft negative cases.")
        return True, log
```

```
    ok = True
    samples: List[str] = []
    for s in negs:
        tokens = _tok(s)
        bound_ok = True
        parse_ok = True
        try:
            binder.bind(tokens)
        except Exception:
            bound_ok = False
        try:
            parser.parse(s)
        except Exception:
            parse_ok = False

        # Expect: at least one should fail (binder or parser)
        if bound_ok and parse_ok:
            ok = False
            if len(samples) < 5:
                samples.append(s)

    if ok:
         log.append("PASS: Negative canonical examples failed as expected (binder and/or
parser).")
    else:
        log.append("FAIL: Some malformed canonicals passed both binder and parser:")
        for s in samples:
            log.append(f"  - {s}")
    return ok, log

# ---------------------------------------------------------------------------
# Aggregator
# ---------------------------------------------------------------------------

def validate_cross_artifacts_all(
    graph: Dict[str, Any],
    grammar_text: str,
    log: List[str],
    *,
    roundtrip_phrases: int = 100,
    roundtrip_threshold: float = 0.95,
    negative_examples: int = 8,
    feasibility_sample_functions: int = 50,
) -> Tuple[bool, List[str]]:
    """
    Run C1C3:
      - C1: canonical roundtrip via binder  serialized  grammar
      - C2: (optional) binder  SQL template feasibility
      - C3: negative canonical inputs
    """
    ok1, log = validate_canonical_roundtrip(
        graph, grammar_text, log,
        phrases=roundtrip_phrases,
```

```
        success_threshold=roundtrip_threshold,
    )

    ok2, log = validate_binder_sql_feasibility(
        graph, log, sample_functions=feasibility_sample_functions
    )

    ok3, log = validate_negative_canonical(
        graph, grammar_text, log, examples=negative_examples
    )

    overall = ok1 and ok2 and ok3
    log.append("PASS: Cross-artifact validations passed." if overall else "FAIL: Cross-artifact validations failed.")
    return overall, log
```

src/n2s_validators/full_integration_validator.py

```python
# src/n2s_validators/full_integration_validator.py
from __future__ import annotations
import random
import re
from collections import defaultdict, Counter
from typing import Dict, Any, List, Tuple, Optional, Iterable, Set

from lark import Lark, ParseError, UnexpectedToken

# Normalizer (alias->canonical)
from src.n2s_runtime.normalizer import normalize_text

# Canonical binder & generator (canonical-only)
from src.n2s_validators.cross_artifact_validator import CanonicalBinder
from src.n2s_validators.grammar_validator import GrammarAnalyzer, SmartGenerator


# -------------------------------------------------------------------------------
# Reverse map: canonical -> [aliases]
# -------------------------------------------------------------------------------

_CONNECTORS: Set[str] = {"of", "from", "and", "or"}  # grammar-facing connectors
_LOCK_TOKENS: Set[str] = {",", *(_CONNECTORS)}       # tokens we never replace

def _coerce_listy(v: Any) -> List[str]:
    if isinstance(v, list): return [("" if o is None else str(o)) for o in v]
    return ["" if v is None else str(v)]

def build_reverse_alias_map(vocabulary: Dict[str, Dict[str, Any]]) -> Dict[str,
List[str]]:
    """
    Invert normalization map (vocabulary) so we can pick aliases for a given canonical.
    Ensures identity forms (canonical -> canonical) are present.
    """
    d = vocabulary.get("deterministic_aliases", {}) or {}
    nd = vocabulary.get("non_deterministic_aliases", {}) or {}

    rev: Dict[str, List[str]] = defaultdict(list)

    # deterministic: alias -> canonical
    for alias, canonical in d.items():
        can = "" if canonical in (None, "skip", "_skip") else str(canonical)
        if can:
            if alias not in rev[can]:
                rev[can].append(str(alias))

    # non-deterministic: alias -> [canonicals]
    for alias, options in nd.items():
        for can in _coerce_listy(options):
            if can and alias not in rev[can]:
                rev[can].append(str(alias))
```

```python
    # identity pass: every canonical should be able to map to itself
    # harvest canonicals seen in values
    seen_canonicals = set()
    for v in d.values():
        if isinstance(v, str) and v not in {"", "skip", "_skip"}:
            seen_canonicals.add(v)
    for lst in nd.values():
        for o in _coerce_listy(lst):
            if o not in {"", "skip", "_skip"}:
                seen_canonicals.add(o)
    for c in seen_canonicals:
        if c not in rev or c not in rev[c]:
            rev[c].append(c)

    # Keep lists stable & unique
    for k in list(rev.keys()):
        seen = set()
        uniq = []
        for a in rev[k]:
            if a not in seen:
                seen.add(a); uniq.append(a)
        rev[k] = uniq

    return rev


# ------------------------------------------------------------------------------
# Canonical de-normalizer (connector-aware)
# ------------------------------------------------------------------------------

_WORD_OR_COMMA = re.compile(r',|[A-Za-z0-9_]+')

def _tok_canonical(s: str) -> List[str]:
    return _WORD_OR_COMMA.findall(s)

def _choose_alias_for_token(
    canonical: str,
    next_token: Optional[str],
    alias_pool: List[str],
) -> Tuple[str, bool]:
    """
    Choose an alias for a single canonical token with 'of' awareness.
    Returns (alias, consume_next_connector).
    Policy:
        - If next token is a connector (e.g., 'of'), prefer plain alias (no trailing
connector).
       - If only aliases that *end with the same connector* exist, pick one and consume
the next token.
      - Otherwise, prefer plain.
    """
    nxt = (next_token or "").lower()
    endswith: Dict[str, List[str]] = {c: [] for c in _CONNECTORS}
    plain: List[str] = []
```

```
    for a in alias_pool:
        aa = a.strip()
        low = aa.lower()
        matched = False
        for c in _CONNECTORS:
            if low.endswith(" " + c):
                endswith[c].append(aa); matched = True; break
        if not matched:
            plain.append(aa)

    if nxt in _CONNECTORS:
        if plain:
            return random.choice(plain), False
        if endswith[nxt]:
            return random.choice(endswith[nxt]), True
        # fallback: any
        pool = plain or [x for L in endswith.values() for x in L]
        if pool:
            return random.choice(pool), False
        return canonical, False
    else:
        # No connector follows: prefer plain; else any
        if plain:
            return random.choice(plain), False
        pool = [x for L in endswith.values() for x in L]
        if pool:
            return random.choice(pool), False
        return canonical, False

def denormalize_canonical(
    canonical_phrase: str,
    reverse_map: Dict[str, List[str]],
) -> str:
    """
    Replace canonical tokens with plausible aliases while avoiding 'of of' etc.
    We *do not* replace punctuation/connector tokens (locked).
    """
    toks = _tok_canonical(canonical_phrase)
    out: List[str] = []
    i = 0
    while i < len(toks):
        t = toks[i]
        if t in _LOCK_TOKENS:
            out.append(t); i += 1; continue

        choices = reverse_map.get(t, [])
        nxt = toks[i + 1] if (i + 1) < len(toks) else None
        alias, consume_next = _choose_alias_for_token(t, nxt, choices)
        out.append(alias)
        i += 2 if (consume_next and nxt in _CONNECTORS) else 1

    # compact spaces around commas
    s = " ".join(out)
    s = re.sub(r"\s*,\s*", ", ", s)
```

```python
    s = re.sub(r"\s{2,}", " ", s).strip()
    return s


# ------------------------------------------------------------------------------
# Full pipeline step: messy  normalize  bind  parse
# ------------------------------------------------------------------------------

def run_full_pipeline_on_text(
    text: str,
    vocabulary: Dict[str, Any],
    graph: Dict[str, Any],
    parser: Lark,
    *,
    max_candidates: int = 50
) -> Tuple[bool, Dict[str, Any]]:
    """
      Returns (success, stats) where stats includes detailed stage counts and failure
reasons.
    """
    stats: Dict[str, Any] = {
        "input": text,
        "normalizer_candidates": 0,
        "bound_candidates": 0,
        "parsed_candidates": 0,
        "fail_category": None,
        "binder_errors": [],
        "parse_errors": [],
        "picked": None,
        "binder_debug": [],
    }

    # Stage 1: normalize (may return many canonical candidates)
    try:
        candidates = normalize_text(vocabulary, text)
    except Exception as e:
        stats["fail_category"] = f"normalizer_exception:{e}"
        return False, stats

    stats["normalizer_candidates"] = len(candidates)

    if not candidates:
        stats["fail_category"] = "normalizer_zero"
        return False, stats

    if len(candidates) > max_candidates:
        candidates = candidates[:max_candidates]
        stats["fail_category"] = "normalizer_many"

    # Stage 2: bind & Stage 3: parse
    binder = CanonicalBinder(graph)
    any_success = False

    for cand in candidates:
```

```
        flight: List[str] = []
        try:
            try:
                    bound = binder.bind(_tok_canonical(cand), recorder=flight)   # new
signature
            except TypeError:
                # older binder without 'recorder' parameter
                bound = binder.bind(_tok_canonical(cand))
            stats["bound_candidates"] += 1
        except Exception as be:
            stats["binder_errors"].append(str(be))
            if flight:
                stats["binder_debug"] = flight  # keep last attempts trace
            continue

        # Serialize binding back to canonical (idempotent check)
        canonical_text = _serialize_binding(bound)
        try:
            parser.parse(canonical_text)
            stats["parsed_candidates"] += 1
            stats["picked"] = canonical_text
            if flight:
                stats["binder_debug"] = flight
            any_success = True
            break
        except Exception as pe:
            stats["parse_errors"].append(str(pe))
            if flight:
                stats["binder_debug"] = flight
            continue

    if not any_success:
        if stats["bound_candidates"] == 0:
            stats["fail_category"] = "binder_fail"
        elif stats["parsed_candidates"] == 0:
            stats["fail_category"] = "parser_fail"
        else:
            stats["fail_category"] = "unknown_fail"
        return False, stats

    return True, stats


def _serialize_binding(binding: Dict[str, Any]) -> str:
    # mirror the serializer from cross_artifact_validator for consistency
    def ser_sel(sel: Dict[str, Any]) -> str:
        if sel["type"] == "column":
            return sel["name"]
        name = sel["name"]
        args = sel.get("args", [])
        if not args: return name
        return f"{name} of {ser_list(args)}"

    def ser_list(items: List[Dict[str, Any]]) -> str:
```

```
        if len(items) == 1:
            return ser_sel(items[0])
        if len(items) == 2:
            return f"{ser_sel(items[0])} and {ser_sel(items[1])}"
        head = ", ".join(ser_sel(x) for x in items[:-1])
        return f"{head}, and {ser_sel(items[-1])}"

    cols = ser_list(binding["selectables"])
    tbl  = binding["table"]
    return f"select {cols} from {tbl}"


# ------------------------------------------------------------------------------
# I1. De-normalize  Normalize  Bind  Parse (random canonical inputs)
# ------------------------------------------------------------------------------

def validate_full_integration_random(
    graph: Dict[str, Any],
    vocabulary: Dict[str, Any],
    grammar_text: str,
    log: List[str],
    *,
    num_phrases: int = 100,
    success_threshold: float = 0.90,
    max_candidates: int = 50,
    rng_seed: Optional[int] = None,
) -> Tuple[bool, List[str]]:
    log.append("--- Running I1: Random Canonical  De-Norm  Norm  Bind  Parse ---")

    if rng_seed is not None:
        random.seed(rng_seed)

    # Build parser & generator
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        log.append(f"FAIL: Could not build parser: {e}")
        return False, log

    analyzer = GrammarAnalyzer(parser)
    generator = SmartGenerator(parser, graph, analyzer)

    # Build reverse alias map for de-normalization
    reverse_map = build_reverse_alias_map(vocabulary)

    # Stats
    success = 0
    fail = 0
    categories = Counter()
    examples: List[Dict[str, Any]] = []

    for _ in range(num_phrases):
        canonical_phrase, _ = generator.generate()
        if not canonical_phrase:
```

```
            fail += 1
            categories["generator_fail"] += 1
            if len(examples) < 3:
                               examples.append({"canonical":  None,  "messy":  None,  "stats":
{"fail_category": "generator_fail"}})
            continue

        messy = denormalize_canonical(canonical_phrase, reverse_map)

        ok, stats = run_full_pipeline_on_text(
            messy, vocabulary, graph, parser, max_candidates=max_candidates
        )
        if ok:
            success += 1
        else:
            fail += 1
            categories[stats.get("fail_category", "unknown")] += 1
            if len(examples) < 3:
                examples.append({"canonical": canonical_phrase, "messy": messy, "stats":
stats})

    total = success + fail
    if total == 0:
        log.append("WARNING: No phrases generated.")
        return True, log

    rate = success / total
    log.append(f"  - Success Rate: {rate:.0%} ({success}/{total})")
    if fail:
        log.append("  - Failure categories:")
        for k, v in categories.most_common():
            log.append(f"    * {k}: {v}")
        log.append("  - Sample failures:")
        for ex in examples:
            log.append(f"    - Canonical: {ex['canonical']}")
            log.append(f"      Messy:     {ex['messy']}")
            log.append(f"      Stats:     {ex['stats']}")

    if rate >= success_threshold:
        log.append("PASS: Full integration pipeline is healthy on randomized inputs.")
        return True, log
    else:
        log.append("FAIL: Full integration success rate below threshold.")
        return False, log


# ---------------------------------------------------------------------------------
# I2. Lossiness & coverage audit (aggregated over random run)
# ---------------------------------------------------------------------------------

def audit_full_integration_lossiness(
    graph: Dict[str, Any],
    vocabulary: Dict[str, Any],
    grammar_text: str,
```

```
    log: List[str],
    *,
    num_phrases: int = 100,
    max_candidates: int = 50,
    rng_seed: Optional[int] = None,
) -> Tuple[bool, List[str]]:
    log.append("--- Running I2: Lossiness & Coverage Audit ---")

    if rng_seed is not None:
        random.seed(rng_seed)

    # Parser & generator
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        log.append(f"FAIL: Could not build parser: {e}")
        return False, log

    analyzer = GrammarAnalyzer(parser)
    generator = SmartGenerator(parser, graph, analyzer)
    reverse_map = build_reverse_alias_map(vocabulary)

    hist_candidates = Counter()
    categories = Counter()

    for _ in range(num_phrases):
        canonical, _ = generator.generate()
        if not canonical:
            categories["generator_fail"] += 1
            continue

        messy = denormalize_canonical(canonical, reverse_map)
        ok, stats = run_full_pipeline_on_text(
            messy, vocabulary, graph, parser, max_candidates=max_candidates
        )
        hist_candidates[stats["normalizer_candidates"]] += 1
        if not ok:
            categories[stats.get("fail_category", "unknown")] += 1

    # Report
    log.append("  - Normalizer candidate count histogram:")
    for k in sorted(hist_candidates.keys()):
        log.append(f"    {k:>3}: {hist_candidates[k]}")

    if categories:
        log.append("  - Failure categories:")
        for k, v in categories.most_common():
            log.append(f"    * {k}: {v}")
    else:
        log.append("  - No failures observed in audit run.")

    return True, log
```

```python
# ------------------------------------------------------------------------------
# I3. Golden-set NL queries
# ------------------------------------------------------------------------------

def validate_golden_set(
    graph: Dict[str, Any],
    vocabulary: Dict[str, Any],
    grammar_text: str,
    golden_queries: List[str],
    log: List[str],
    *,
    success_threshold: float = 1.0,   # require all pass by default
    max_candidates: int = 50,
) -> Tuple[bool, List[str]]:
    log.append("--- Running I3: Golden-set NL Queries ---")

    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        log.append(f"FAIL: Could not build parser: {e}")
        return False, log

    binder = CanonicalBinder(graph)

    successes = 0
    failures: List[Dict[str, Any]] = []

    for q in golden_queries:
        ok, stats = run_full_pipeline_on_text(
            q, vocabulary, graph, parser, max_candidates=max_candidates
        )
        if ok:
            successes += 1
        else:
            failures.append({"query": q, "stats": stats})

    total = len(golden_queries)
    rate = (successes / total) if total else 1.0
    log.append(f"  - Success Rate: {rate:.0%} ({successes}/{total})")

    if failures:
        log.append("  - Failures:")
        for f in failures[:10]:
            log.append(f"    * Query: {f['query']}")
            log.append(f"      Stats: {f['stats']}")
        if len(failures) > 10:
            log.append(f"      ... +{len(failures)-10} more")

    if rate >= success_threshold:
        log.append("PASS: Golden-set passed threshold.")
        return True, log
    else:
        log.append("FAIL: Golden-set below threshold.")
        return False, log
```

```
# ----------------------------------------------------------------------------
# Aggregator
# ----------------------------------------------------------------------------

def validate_full_integration_all(
    graph: Dict[str, Any],
    vocabulary: Dict[str, Any],
    grammar_text: str,
    log: List[str],
    *,
    random_phrases: int = 100,
    random_threshold: float = 0.90,
    lossiness_phrases: int = 100,
    golden_queries: Optional[List[str]] = None,
    golden_threshold: float = 1.0,
    max_candidates: int = 50,
    rng_seed: Optional[int] = None,
) -> Tuple[bool, List[str]]:
    """
    Run I1I3:
      - I1: randomized canonical  de-norm  normalize  bind  parse
      - I2: lossiness/candidate hist + failure categories
      - I3: golden-set NL queries
    """
    ok1, log = validate_full_integration_random(
        graph, vocabulary, grammar_text, log,
        num_phrases=random_phrases,
        success_threshold=random_threshold,
        max_candidates=max_candidates,
        rng_seed=rng_seed,
    )

    ok2, log = audit_full_integration_lossiness(
        graph, vocabulary, grammar_text, log,
        num_phrases=lossiness_phrases,
        max_candidates=max_candidates,
        rng_seed=rng_seed,
    )

    ok3 = True
    if golden_queries:
        ok3, log = validate_golden_set(
            graph, vocabulary, grammar_text, golden_queries, log,
            success_threshold=golden_threshold,
            max_candidates=max_candidates,
        )

    overall = ok1 and ok2 and ok3
    log.append("PASS: Full integration validations passed." if overall else "FAIL: Full
integration validations failed.")
    return overall, log
```

src/n2s_validators/grammar_validator.py

```
# src/n2s_validators/grammar_validator.py

from __future__ import annotations
import random
import re
from typing import Dict, Any, List, Tuple, Iterable, Set
from collections import defaultdict

from lark import Lark, ParseError, UnexpectedToken, Tree

# --------------------------------------------------------------
# Utilities
# --------------------------------------------------------------

def _entities_by_type(graph: Dict[str, Any], etype: str) -> List[str]:
    return [k for k, v in graph.items() if v.get("entity_type") == etype]

def _canonicals(graph: Dict[str, Any]) -> Dict[str, List[str]]:
    return {
        "tables": _entities_by_type(graph, "table"),
        "columns": _entities_by_type(graph, "column"),
         "functions": _entities_by_type(graph, "sql_actions") + _entities_by_type(graph,
"postgis_actions"),
        "verbs": _entities_by_type(graph, "select_verbs"),
    }

def _extract_grammar_literals(grammar_text: str, terminal_name: str) -> Set[str]:
    """
    Parse a line like:
      CANONICAL_TABLE: "regions" | "sales" | "users"
    into {"regions","sales","users"}.
    """
    m = re.search(rf"^{terminal_name}:\s*(.+)$", grammar_text, flags=re.MULTILINE)
    if not m:
        return set()
    body = m.group(1)
    lits = [s.strip() for s in body.split("|")]
    out: Set[str] = set()
    for lit in lits:
        if lit.startswith('"') and lit.endswith('"'):
            out.add(lit[1:-1])
    return out

def _build_phrase_smoke(
    tables: List[str], columns: List[str], functions: List[str], verbs: List[str]
) -> List[str]:
    # Use canonical 'select' if present, else the first verb
    verb = "select" if "select" in verbs else (verbs[0] if verbs else "select")
    t = random.choice(tables)
    c1 = random.choice(columns)
    c2 = random.choice(columns)
```

```python
    f = random.choice(functions) if functions else None

    patterns = [
        f"{verb} {c1} from {t}",
        f"{verb} {c1} , {c2} from {t}",
    ]
    if f:
        patterns += [
            f"{verb} {f} of {c1} from {t}",
            f"{verb} {c1} and {f} of {c2} from {t}",
        ]
    return patterns

# -------------------------------------------------------------
# Grammar Analyzer & Smart Generator (for stress tests)
# -------------------------------------------------------------

class GrammarAnalyzer:
    """
    Computes minimal expansion depth per rule and provides rule lookup.
    """
    def __init__(self, parser: Lark):
        self.parser = parser
        self.rule_lookup = defaultdict(list)
        for r in self.parser.rules:
                  origin = r.origin.name.value if hasattr(r.origin.name, "value") else
r.origin.name
            self.rule_lookup[origin].append(r)

        self.min_depths: Dict[str, int] = {}
        self._calculate_min_depths()

    def _get_min_depth(self, term_name: str) -> int:
            # Terminals are upper-case or quoted; also treat a few special names as
terminals
           if term_name.isupper() or term_name.startswith('"') or term_name in {"AND",
"COMMA", "OF", "FROM", "SELECT"}:
            return 1
        return self.min_depths.get(term_name, 10 ** 9)

    def _calculate_min_depths(self) -> None:
        for _ in range(len(self.rule_lookup) + 3):
            for rule_name, expansions in self.rule_lookup.items():
                best = 10 ** 9
                for r in expansions:
                    if not r.expansion:
                        best = min(best, 1)  # empty production
                    else:
                        s = 1 + sum(self._get_min_depth(t.name) for t in r.expansion)
                        best = min(best, s)
                if best < self.min_depths.get(rule_name, 10 ** 9):
                    self.min_depths[rule_name] = best

class SmartGenerator:
```

```
    """
    Canonical phrase generator with recursion guard and depth-aware choice.
    """
    def __init__(self, parser: Lark, graph: Dict[str, Any], analyzer: GrammarAnalyzer):
        self.parser = parser
        self.rule_lookup = analyzer.rule_lookup
        self.analyzer = analyzer
        self.RECURSION_LIMIT = 4

        # Build canonical vocab from graph
        c = _canonicals(graph)
        self.vocab = {
            "CANONICAL_COLUMN": c["columns"],
            "CANONICAL_TABLE": c["tables"],
            "CANONICAL_FUNCTION": c["functions"],
            "SELECT": ["select"],  # canonical token used by our grammar builder
            "OF": ["of"],
            "FROM": ["from"],
            "AND": ["and"],
            "COMMA": [","],
        }

    def generate(self, start_rule: str = "query", max_depth: int = 25) -> Tuple[str | None, List[str] | None]:
        log: List[str] = []
        out = self._expand(start_rule, max_depth, log, "", {})
        return (out, None) if out is not None else (None, log)

    def _expand(self, sym: str, depth: int, log: List[str], indent: str, counts: Dict[str, int]) -> str | None:
        log.append(f"{indent}>> {sym} depth={depth}")
        if depth <= 0:
            return None

        # Terminal?
        if sym not in self.rule_lookup:
            # choose from known vocab if present, else echo the literal name (strip quotes)
            if sym in self.vocab and self.vocab[sym]:
                v = random.choice(self.vocab[sym])
            else:
                v = sym.strip('"')
            log.append(f"{indent}<< term '{v}'")
            return v

        # recursion guard
        counts = dict(counts)
        counts[sym] = counts.get(sym, 0) + 1
        if counts[sym] > self.RECURSION_LIMIT:
            # choose a non-recursive expansion if possible
            cands = [r for r in self.rule_lookup[sym] if sym not in [t.name for t in r.expansion]]
            if not cands:
                return None
```

```
        else:
            cands = list(self.rule_lookup[sym])

        # depth-aware choice
        if depth < (self.analyzer.min_depths.get(sym, 1) + 5):
            # prefer the cheapest expansions
            costs = {
                r: 1 + sum(self.analyzer.min_depths.get(t.name, 0) for t in r.expansion)
                for r in cands
            }
            min_cost = min(costs.values())
            opts = [r for r, c in costs.items() if c == min_cost]
        else:
            opts = cands

        rule = random.choice(opts)
        parts: List[str] = []
        for t in rule.expansion:
            sub = self._expand(t.name, depth - 1, log, indent + "  ", counts)
            if sub is None:
                return None
            parts.append(sub)
        s = " ".join(x for x in parts if x != "")
        log.append(f"{indent}<< '{s[:60]}'")
        return s

# ------------------------------------------------------------
# Gm1. Vocabulary alignment
# ------------------------------------------------------------

def validate_grammar_vocab_alignment(graph: Dict[str, Any], grammar_text: str, log:
List[str]) -> Tuple[bool, List[str]]:
    log.append("--- Running Gm1: Grammar Vocabulary Alignment ---")
    c = _canonicals(graph)

    g_tables = _extract_grammar_literals(grammar_text, "CANONICAL_TABLE")
    g_columns = _extract_grammar_literals(grammar_text, "CANONICAL_COLUMN")
    g_functions = _extract_grammar_literals(grammar_text, "CANONICAL_FUNCTION")

    ok = True
    missing_tables = sorted(set(c["tables"]) - g_tables)
    missing_columns = sorted(set(c["columns"]) - g_columns)
    missing_functions = sorted(set(c["functions"]) - g_functions)

    if missing_tables:
        ok = False
        log.append("FAIL: Missing table literals in grammar:")
        for t in missing_tables: log.append(f"  - {t}")
    if missing_columns:
        ok = False
        log.append("FAIL: Missing column literals in grammar:")
        for t in missing_columns: log.append(f"  - {t}")
    if missing_functions:
        ok = False
```

```
        log.append("FAIL: Missing function literals in grammar:")
        for t in missing_functions: log.append(f"  - {t}")

    if ok:
        log.append("PASS: All canonical tables/columns/functions are present in grammar terminals.")
    return ok, log

# --------------------------------------------------------------
# Gm2. Canonical smoke tests
# --------------------------------------------------------------

def validate_grammar_smoke_tests(graph: Dict[str, Any], grammar_text: str, log: List[str], num_tests: int = 24) -> Tuple[bool, List[str]]:
    log.append("--- Running Gm2: Canonical Smoke Tests ---")
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
        log.append(f"FAIL: Could not instantiate grammar: {e}")
        return False, log

    c = _canonicals(graph)
    if not (c["tables"] and c["columns"]):
        log.append("FAIL: Graph lacks tables or columns for smoke tests.")
        return False, log

    failures: List[str] = []
    for _ in range(num_tests):
        for phrase in _build_phrase_smoke(c["tables"], c["columns"], c["functions"], c["verbs"]):
            try:
                parser.parse(phrase)
            except (ParseError, UnexpectedToken) as e:
                if len(failures) < 5:
                    failures.append(f"{phrase}  ->  {e}")
    if failures:
        log.append("FAIL: Some canonical smoke phrases failed to parse:")
        for f in failures:
            log.append(f"  - {f}")
        return False, log

    log.append("PASS: Canonical smoke phrases parsed successfully.")
    return True, log

# --------------------------------------------------------------
# Gm3. Stress tests (canonical)
# --------------------------------------------------------------

def validate_grammar_stress(graph: Dict[str, Any], grammar_text: str, log: List[str], num_phrases: int = 100, success_threshold: float = 0.90) -> Tuple[bool, List[str]]:
    log.append("--- Running Gm3: Grammar Stress Test (Canonical) ---")
    try:
        parser = Lark(grammar_text, start="query")
    except Exception as e:
```

```
        log.append(f"FAIL: Could not instantiate grammar: {e}")
        return False, log

    analyzer = GrammarAnalyzer(parser)
    gen = SmartGenerator(parser, graph, analyzer)

    ok_count = 0
    fail_count = 0
    for _ in range(num_phrases):
        phrase, _ = gen.generate()
        if not phrase:
            fail_count += 1
            continue
        try:
            parser.parse(phrase)
            ok_count += 1
        except (ParseError, UnexpectedToken):
            fail_count += 1

    total = ok_count + fail_count
    if total == 0:
        log.append("WARNING: Generator produced no phrases.")
        return True, log

    rate = ok_count / total
    log.append(f"  - Success Rate: {rate:.0%} ({ok_count}/{total})")
    if rate < success_threshold:
        log.append(f"FAIL: Success rate below threshold ({success_threshold:.0%}).")
        return False, log

    log.append("PASS: Stress test meets success threshold.")
    return True, log

# --------------------------------------------------------------
# Gm4. Ambiguity checks (canonical inputs)
# --------------------------------------------------------------

def _is_ambiguous(tree: Tree) -> bool:
        # With ambiguity='explicit', ambiguous parses are wrapped in a Tree with
data='_ambig'
    return isinstance(tree, Tree) and tree.data == '_ambig'

def validate_grammar_ambiguity(graph: Dict[str, Any], grammar_text: str, log: List[str],
sample_phrases: int = 24) -> Tuple[bool, List[str]]:
    log.append("--- Running Gm4: Ambiguity Checks (Canonical) ---")
    try:
                    parser_amb = Lark(grammar_text, start="query", parser="earley",
ambiguity="explicit")
    except Exception as e:
        log.append(f"FAIL: Could not instantiate ambiguity parser: {e}")
        return False, log

    c = _canonicals(graph)
    if not (c["tables"] and c["columns"]):
```

```
        log.append("FAIL: Graph lacks tables or columns for ambiguity probes.")
        return False, log

    ambiguous: List[str] = []
    tried = 0
    while tried < sample_phrases:
        phrases = _build_phrase_smoke(c["tables"], c["columns"], c["functions"], c["verbs"])
        for p in phrases:
            tried += 1
            try:
                tree = parser_amb.parse(p)
                if _is_ambiguous(tree):
                    if len(ambiguous) < 5:
                        ambiguous.append(p)
            except Exception:
                # Skip parse failures here; Gm2/Gm3 cover acceptance
                pass
            if tried >= sample_phrases:
                break

    if ambiguous:
        log.append("WARNING: Ambiguity detected on some canonical inputs (parse forest > 1):")
        for p in ambiguous:
            log.append(f"  - {p}")
    else:
        log.append("PASS: No ambiguity observed for sampled canonical inputs.")
    return True, log

# ------------------------------------------------------------
# Gm5. Grammar health (reachability, recursion sanity, reserved tokens)
# ------------------------------------------------------------

def validate_grammar_health(grammar_text: str, log: List[str], start_rule: str = "query") -> Tuple[bool, List[str]]:
    log.append("--- Running Gm5: Grammar Health ---")
    try:
        parser = Lark(grammar_text, start=start_rule)
    except Exception as e:
        log.append(f"FAIL: Could not instantiate grammar for health checks: {e}")
        return False, log

    # Reachability
    rule_lookup = defaultdict(list)
    for r in parser.rules:
        origin = r.origin.name.value if hasattr(r.origin.name, "value") else r.origin.name
        rule_lookup[origin].append(r)

    reachable: Set[str] = set()
    def dfs(sym: str) -> None:
        if sym in reachable:
            return
```

```
        reachable.add(sym)
        for r in rule_lookup.get(sym, []):
            for t in r.expansion:
                name = t.name
                      if not (name.isupper() or name.startswith('"')):  # only follow
non-terminals
                    dfs(name)

    dfs(start_rule)
        all_rules = { (r.origin.name.value if hasattr(r.origin.name, "value") else
r.origin.name) for r in parser.rules }
    unreachable = sorted(all_rules - reachable)

    ok = True
    if unreachable:
        ok = False
        log.append("FAIL: Unreachable non-terminals:")
        for u in unreachable[:50]:
            log.append(f"  - {u}")
        if len(unreachable) > 50:
            log.append(f"  ... +{len(unreachable)-50} more")

    # Recursion sanity via minimal depth
    analyzer = GrammarAnalyzer(parser)
    infinite_like = [nt for nt, d in analyzer.min_depths.items() if d >= 10 ** 8]
    if infinite_like:
        ok = False
        log.append("FAIL: Potentially non-terminating rules (min-depth not finite):")
        for nt in infinite_like:
            log.append(f"  - {nt}")

    # Reserved tokens present
    needed_terms = ["OF", "FROM", "AND", "COMMA", "SELECT"]
    for term in needed_terms:
        if not re.search(rf"^{term}:\s*", grammar_text, flags=re.MULTILINE):
            ok = False
            log.append(f"FAIL: Missing terminal definition for '{term}'.")

    if ok:
         log.append("PASS: Grammar health looks good (reachability, recursion, reserved
tokens).")
    return ok, log

# ---------------------------------------------------------------
# Aggregator
# ---------------------------------------------------------------

def validate_grammar_all(
    graph: Dict[str, Any],
    grammar_text: str,
    log: List[str],
    *,
    smoke_tests: int = 24,
    stress_tests: int = 120,
```

```python
    stress_threshold: float = 0.90,
    ambiguity_samples: int = 32,
) -> Tuple[bool, List[str]]:
    """
    Run Gm1Gm5 end-to-end on the canonical grammar.
    """
    results: List[bool] = []

    ok1, log = validate_grammar_vocab_alignment(graph, grammar_text, log); results.append(ok1)
    ok2, log = validate_grammar_smoke_tests(graph, grammar_text, log, num_tests=smoke_tests); results.append(ok2)
    ok3, log = validate_grammar_stress(graph, grammar_text, log, num_phrases=stress_tests, success_threshold=stress_threshold); results.append(ok3)
    ok4, log = validate_grammar_ambiguity(graph, grammar_text, log, sample_phrases=ambiguity_samples); results.append(ok4)
    ok5, log = validate_grammar_health(grammar_text, log); results.append(ok5)

    overall = all(results)
    if overall:
        log.append("PASS: Grammar validations (Gm1Gm5) passed.")
    else:
        log.append("FAIL: One or more grammar validations failed.")
    return overall, log
```

src/n2s_validators/graph_validator.py

```python
# src/n2s_validators/graph_validator.py

from __future__ import annotations
from typing import Dict, Any, List, Tuple, Set, Callable, Optional
from collections import defaultdict, Counter
import copy

# -------------------------------------------------------------
# Helpers (pure, no I/O)
# -------------------------------------------------------------

_ALLOWED_ENTITY_TYPES: Set[str] = {
    "table", "column",
    "sql_actions", "postgis_actions",
    "select_verbs", "prepositions",
    "logical_operators", "comparison_operators",
    "filler_words",
}

_RESERVED_TOKENS: Set[str] = {
    ",", "&&", "||", "==", "!=", "<>", "<=", ">=", "=", "!", "<", ">",  # ops/punct
    "and", "or", "not",                                                  # logicals
    "of", "from", "in", "on", "at", "belonging to",                      # preps
    "select"                                                             # select verb
}

# for G6 (plural sanity)
_PLURAL_LASTWORD = {
    "date": "dates",
    "login": "logins",
    "id": "ids",
    "username": "usernames",
    "name": "names",
    "item": "items",
    "value": "values",
}

# -------- deep scanners --------

def _walk(obj: Any, fn) -> None:
    if isinstance(obj, dict):
        for k, v in obj.items():
            fn(k); _walk(v, fn)
    elif isinstance(obj, list):
        for v in obj:
            _walk(v, fn)
    else:
        fn(obj)

def _has_tuple(obj: Any) -> bool:
    hit = False
```

```
    def _probe(x):
        nonlocal hit
        if isinstance(x, tuple):
            hit = True
    _walk(obj, _probe)
    return hit

def _as_set_str(x: Any) -> Set[str]:
    if isinstance(x, list): return {str(i) for i in x}
    if isinstance(x, set): return {str(i) for i in x}
    if isinstance(x, tuple): return {str(i) for i in x}
    return {str(x)} if x is not None else set()

def _sorted_list_unique(seq: List[str]) -> List[str]:
    seen, out = set(), []
    for s in seq:
        if s not in seen:
            seen.add(s); out.append(s)
    out.sort()
    return out

# -------------------------------------------------------------
# Compatibility (reuses your semantics)
# -------------------------------------------------------------
def _column_type_aliases(column_info: Dict[str, Any]) -> Set[str]:
    """
    Build a set of type aliases for matching against function allowed types.
    Includes:
      - raw type (e.g., 'VARCHAR(50)')
      - normalized lowercase token without params (e.g., 'varchar')
      - existing 'type_category' (from graph builder)
      - spatial families ('geometry', 'geography') if subtype present
      - coarse families (numeric, text, date, timestamp, boolean) inferred from raw type
    """
    aliases: Set[str] = set()

    raw = (column_info.get('type') or '').strip().lower()
    cat = (column_info.get('type_category') or '').strip().lower()

    if raw:
        aliases.add(raw)
        # strip params like varchar(50) -> varchar
        if '(' in raw:
            aliases.add(raw.split('(')[0].strip())

        # coarse families
        if 'int' in raw or any(k in raw for k in ['decimal', 'numeric', 'real', 'float',
'double']):
            aliases.add('numeric')
        if 'char' in raw or 'text' in raw or 'string' in raw:
            aliases.add('text')
        if 'bool' in raw:
            aliases.add('boolean')
        if 'timestamp' in raw:
```

```python
            aliases.add('timestamp')
        if 'date' in raw:
            aliases.add('date')

        # spatial families from raw
        if 'geometry' in raw:
            aliases.add('geometry')
            # capture subtypes like geometry_point
            if '_' in raw:
                aliases.add(raw.split('_', 1)[0])  # geometry
        if 'geography' in raw:
            aliases.add('geography')
            if '_' in raw:
                aliases.add(raw.split('_', 1)[0])  # geography

    if cat:
        aliases.add(cat)
        # include spatial family from category, e.g., geography_point -> geography
        if cat.startswith('geometry_'):
            aliases.add('geometry')
        if cat.startswith('geography_'):
            aliases.add('geography')

        # also propagate coarse families if category already coarse
        # (kept simple since cat already computed in graph_builder)
        if cat in {'numeric', 'text', 'boolean', 'date', 'timestamp'}:
            aliases.add(cat)

    return aliases

def is_compatible(column_info: Dict[str, Any], action_info: Dict[str, Any]) -> Dict[str,
bool]:
    """
    Checks if a given action/function can be applied to a given column based
    on type and label rules.

    Now considers a family-expanded set of column type aliases, so e.g.
    'geography_point' matches allowed ['geography'] and 'varchar(50)' matches 'text'.
    """
    compatible_vars = {}
    if not (isinstance(column_info, dict) and isinstance(action_info, dict)):
        return compatible_vars

    applicable_types_dict = action_info.get('applicable_types')
    if not isinstance(applicable_types_dict, dict):
        return compatible_vars

    col_labels = column_info.get('labels', []) or []
    col_aliases = _column_type_aliases(column_info)  # <-- NEW family-expanded set

    for var, allowed_types in applicable_types_dict.items():
        # Coerce to lowercase set
        if isinstance(allowed_types, list):
            allowed = {str(t).strip().lower() for t in allowed_types}
```

```
        else:
            allowed = {str(allowed_types).strip().lower()}

        # TYPE CHECK: any intersection between allowed and our alias set
        type_ok = ('any' in allowed) or bool(col_aliases & allowed)
        if not type_ok:
            continue

        # LABEL RULES
        valid_labels = True
        for rule in action_info.get('label_rules', []) or []:
            if isinstance(rule, str) and rule.startswith('not '):
                lab = rule[4:]
                if lab in col_labels:
                    valid_labels = False; break
            else:
                if rule not in col_labels:
                    valid_labels = False; break

        if valid_labels:
            compatible_vars[var] = True

    return compatible_vars


# ------------------------------------------------------------
# G1. Structural shape & schema hygiene
# ------------------------------------------------------------

def  validate_graph_structure(graph:  Dict[str,  Any],  log:  List[str])  ->  Tuple[bool,
List[str]]:
    log.append("--- Running G1: Graph Structural & Schema Hygiene ---")
    ok = True

    counts = Counter()
    meta_nodes_seen: List[str] = []

    for canon, node in graph.items():
        if not isinstance(node, dict):
            log.append(f"FAIL: Node '{canon}' must be a dict, got {type(node)}.")
            ok = False; continue

        et = node.get("entity_type")
        md = node.get("metadata")

        # --- DEBUG: count and special-case meta nodes
        counts[et or "None"] += 1
        if et == "_meta":
            if not isinstance(md, dict):
                ok = False
                  log.append(f"FAIL: Node '{canon}' has entity_type '_meta' but metadata
is not a dict.")
            else:
                meta_nodes_seen.append(canon)
```

```
                log.append(f"DEBUG:G1: Skipping structural checks for meta node '{canon}'.")
            # Skip further checks for meta nodes
            continue

        if et not in _ALLOWED_ENTITY_TYPES:
            log.append(f"FAIL: Node '{canon}' has invalid entity_type: {et}.")
            ok = False
        if not isinstance(md, dict):
            log.append(f"FAIL: Node '{canon}' metadata must be a dict.")
            ok = False
            continue

        # basic per-type expectations
        if et == "table":
            if "columns" not in md or not isinstance(md["columns"], dict):
                log.append(f"FAIL: Table '{canon}' metadata missing 'columns' dict.")
                ok = False
            if "aliases" not in md or not isinstance(md["aliases"], list):
                log.append(f"FAIL: Table '{canon}' metadata missing 'aliases' list.")
                ok = False

        elif et == "column":
            if "type" not in md or not isinstance(md["type"], str) or not md["type"]:
                log.append(f"FAIL: Column '{canon}' metadata missing non-empty 'type' string.")
                ok = False
            if "labels" not in md or not isinstance(md["labels"], list):
                log.append(f"FAIL: Column '{canon}' metadata missing 'labels' list.")
                ok = False
            if "aliases" not in md or not isinstance(md["aliases"], list):
                log.append(f"FAIL: Column '{canon}' metadata missing 'aliases' list.")
                ok = False

        else:
            # keywords/functions/operators: must have aliases list
            if "aliases" not in md or not isinstance(md["aliases"], list):
                log.append(f"FAIL: Node '{canon}' ({et}) metadata missing 'aliases' list.")
                ok = False
            # template (optional) if present must be a string
            if "template" in md and not isinstance(md["template"], str):
                log.append(f"FAIL: Node '{canon}' ({et}) 'template' must be a string when present.")
                ok = False

    # YAML-serializability proxy: forbid tuples
    if _has_tuple(graph):
        log.append("FAIL: Graph contains Python tuples which serialize to YAML tags (unsafe).")
        ok = False

    # --- DEBUG: summary
    log.append(f"DEBUG:G1: Node type counts: {dict(counts)}")
```

```
    if meta_nodes_seen:
                         log.append(f"DEBUG:G1: Meta nodes encountered (skipped): {sorted(meta_nodes_seen)}")

    if ok:
        log.append("PASS: Graph structural checks passed.")
    return ok, log


# ---------------------------------------------------------------
# G2. Referential integrity
# ---------------------------------------------------------------

def validate_graph_referential_integrity(graph: Dict[str, Any], log: List[str]) -> Tuple[bool, List[str]]:
    log.append("--- Running G2: Referential Integrity ---")
    ok = True

    # Build quick lookup for top-level columns
    top_cols = {k: v for k, v in graph.items() if v.get("entity_type") == "column"}

    for tbl, node in graph.items():
        if node.get("entity_type") != "table":
            continue
        cols = (node.get("metadata", {}) or {}).get("columns", {}) or {}
        for col_name, col_md in cols.items():
            top = top_cols.get(col_name)
            if not top:
                   log.append(f"FAIL: Table '{tbl}' references column '{col_name}' which has no top-level column node.")
                ok = False
                continue

            top_md = top.get("metadata", {}) or {}
            # Compare key fields (type, labels, aliases presence)
            if top_md.get("type") != col_md.get("type"):
                                 log.append(f"FAIL: Column '{col_name}' type mismatch table:{col_md.get('type')} vs top:{top_md.get('type')}.")
                ok = False
            if sorted(top_md.get("labels", []) or []) != sorted(col_md.get("labels", []) or []):
                               log.append(f"FAIL: Column '{col_name}' labels mismatch table:{col_md.get('labels')} vs top:{top_md.get('labels')}.")
                ok = False
            # aliases may differ in order; compare as sets (non-empty)
                             if not isinstance(col_md.get("aliases"), list) or not isinstance(top_md.get("aliases"), list):
                   log.append(f"FAIL: Column '{col_name}' aliases must be lists in both places.")
                ok = False
            else:
                s_tbl = set(map(str, col_md.get("aliases") or []))
                s_top = set(map(str, top_md.get("aliases") or []))
                if not s_tbl.issubset(s_top):
```

```
                        log.append(f"WARNING: Column '{col_name}' table aliases not all present at top-level. Missing={sorted(list(s_tbl - s_top))}")

    if ok:
        log.append("PASS: Referential integrity checks passed (with possible warnings).")
    return ok, log

# ---------------------------------------------------------------
# G3. Type/label coherence
# ---------------------------------------------------------------

def validate_graph_type_label_coherence(
    graph: Dict[str, Any],
    log: List[str],
    *,
    allowed_labels: Optional[Set[str]] = None,
) -> Tuple[bool, List[str]]:
    log.append("--- Running G3: Type/Label Coherence ---")
    ok = True

    # If allowed_labels not given, use a permissive default
    allowed_labels = allowed_labels or {"id", "postgis", "latitude", "longitude"}

    for canon, node in graph.items():
        if node.get("entity_type") != "column":
            continue
        md = node.get("metadata", {}) or {}
        ctype = md.get("type")
        labels = md.get("labels", []) or []

        if not isinstance(ctype, str) or not ctype:
            log.append(f"FAIL: Column '{canon}' missing non-empty string type.")
            ok = False

        if not isinstance(labels, list):
            log.append(f"FAIL: Column '{canon}' labels must be a list.")
            ok = False
            continue

        for lab in labels:
            if not isinstance(lab, str):
                log.append(f"FAIL: Column '{canon}' label '{lab}' must be string.")
                ok = False
            elif lab not in allowed_labels:
                log.append(f"WARNING: Column '{canon}' label '{lab}' not in allowed label set.")

    if ok:
        log.append("PASS: Type/Label coherence checks passed (with possible warnings).")
    return ok, log

# ---------------------------------------------------------------
# G4. Function compatibility matrix
```

```python
# ------------------------------------------------------------

# --- REPLACE / ADD THESE HELPERS (near the other helpers) ---

def _column_type_aliases(md: Dict[str, Any]) -> Set[str]:
    """
    Produce a normalized set of type tokens for a column:
    - raw DB type (lower)
    - type_category (lower)
    - family tokens for geometry/geography subtypes
    Never injects 'numeric' unless the column truly is numeric.
    """
    out: Set[str] = set()
    raw = str(md.get("type", "") or "").strip().lower()
    cat = str(md.get("type_category", "") or "").strip().lower()

    if raw:
        out.add(raw)
    if cat:
        out.add(cat)

    # family expansions only if raw or category indicates spatial
    for tok in (raw, cat):
        if tok.startswith("geometry_"):
            out.add("geometry")
        if tok.startswith("geography_"):
            out.add("geography")

    # 'any' is a universal matcher for convenience
    out.add("any")
    return out


def _collect_present_type_aliases(columns: Dict[str, Any]) -> Set[str]:
    present: Set[str] = set()
    for cnode in columns.values():
        md = cnode.get("metadata", {}) or {}
        present |= _column_type_aliases(md)
    return present


def _flatten_applicable_types(app: Dict[str, Any]) -> Set[str]:
    req: Set[str] = set()
    for _, types in (app or {}).items():
        if isinstance(types, (list, tuple, set)):
            req |= {str(t).lower() for t in types}
        elif types is not None:
            req.add(str(types).lower())
    # 'any' is not a family; remove to make family-absence check meaningful
    req.discard("any")
    return req


def _family_absent(required_types: Set[str], present_types: Set[str]) -> bool:
```

```
    # If none of the required type families exist in the schema, the function is "absent
family"
    return len(required_types & present_types) == 0


def _function_domain(fnode: Dict[str, Any]) -> str:
    et = fnode.get("entity_type")
    return "spatial" if et == "postgis_actions" else "general"


def _column_relevant_type_overlap(col_aliases: Set[str], functions: Dict[str, Any]) ->
bool:
    """
     For a column, do *any* functions declare an applicable type that overlaps with the
column's aliases?
      Used to demote orphan columns to WARN when the entire family is absent from the
function catalog.
    """
    for fnode in functions.values():
        fmd = fnode.get("metadata", {}) or {}
        app = fmd.get("applicable_types")
        if not isinstance(app, dict) or not app:
            continue
        req = _flatten_applicable_types(app)
        if req & col_aliases:
            return True
    return False

# --- REPLACE THE WHOLE validate_function_compatibility_matrix FUNCTION ---

def validate_function_compatibility_matrix(graph: Dict[str, Any], log: List[str]) ->
Tuple[bool, List[str]]:
    log.append("--- Running G4: Function Compatibility Matrix ---")
    ok = True

    columns = {k: v for k, v in graph.items() if v.get("entity_type") == "column"}
    functions = {
        k: v for k, v in graph.items()
                     if v.get("entity_type") in {"sql_actions", "postgis_actions",
"comparison_operators"}
    }

    # --- DEBUG: counts
             log.append(f"DEBUG:G4: Column count={len(columns)}; Function/operator
count={len(functions)}")

    present_types = _collect_present_type_aliases(columns)

    # --- DEBUG: spatial coverage summary
    spatial_cols = []
    for cn, c in columns.items():
        md = c.get("metadata", {}) or {}
        alias_set = _column_type_aliases(md)
        if {'geometry', 'geography'} & alias_set or any(
```

```
        t.startswith(('geometry_', 'geography_')) for t in alias_set
    ):
        spatial_cols.append({
            "column": cn,
            "type": md.get("type"),
            "type_category": md.get("type_category"),
            "labels": md.get("labels"),
            "aliases": sorted(list(alias_set))[:8],  # preview
        })
log.append(f"DEBUG:G4: Spatial-like columns detected: {len(spatial_cols)}")
for sample in spatial_cols[:5]:
    log.append(f"DEBUG:G4: spatial_sample: {sample}")

if not columns:
    log.append("WARNING: No columns found.")
if not functions:
    log.append("WARNING: No functions/operators found.")

dead_funcs: List[str] = []
dead_func_details: List[str] = []

# ---- pass 1: functions with no compatible columns
for fname, fnode in functions.items():
    fmd = fnode.get("metadata", {}) or {}
    app = fmd.get("applicable_types")
    if not isinstance(app, dict) or not app:
        # unconstrained function  skip compatibility check
        continue

    # If the required family is totally absent in the schema, WARN (dont FAIL)
    required_types = _flatten_applicable_types(app)
    absent_family = _family_absent(required_types, present_types)

    any_ok = False
    for cname, cnode in columns.items():
        if is_compatible(cnode.get("metadata", {}), fmd):
            any_ok = True
            break

    if not any_ok:
        if absent_family:
            log.append(
                f"WARNING:G4: '{fname}' has no compatible columns "
                f"(absent_type_family); required_types={sorted(required_types)}; "
                f"present_types_sample={sorted(list(present_types))[:12]}"
            )
            # Keep a debug detail for traceability
            dead_func_details.append(
                f"DEBUG:G4: '{fname}' (domain={_function_domain(fnode)}) "
                f"required_types={sorted(required_types)}  present_types="
            )
        else:
            dead_funcs.append(fname)
            # --- DEBUG: capture why with first few columns
```

```
                sample_cols = []
                for i, (cn, c) in enumerate(columns.items()):
                    if i >= 5:
                        break
                    md = c.get("metadata", {}) or {}
                    sample_cols.append({
                        "column": cn,
                        "type": md.get("type"),
                        "type_category": md.get("type_category"),
                        "type_aliases": sorted(list(_column_type_aliases(md)))[:10],
                        "labels": md.get("labels"),
                    })
                dead_func_details.append(
                    f"DEBUG:G4: '{fname}' has applicable_types="
                      f"{ {var: [str(t).lower() for t in (ts or [])] for var, ts in (app
or {}).items()} }; "
                    f"first_cols_sample={sample_cols}"
                )

    orphan_cols: List[str] = []
    orphan_details: List[str] = []

    # ---- pass 2: columns with no compatible functions
    for cname, cnode in columns.items():
        c_md = cnode.get("metadata", {}) or {}
        c_ok = False
        for fname, fnode in functions.items():
            fmd = fnode.get("metadata", {}) or {}
            if not isinstance(fmd.get("applicable_types"), dict):
                continue
            if is_compatible(c_md, fmd):
                c_ok = True
                break

        if not c_ok:
               # If *no* function even declares overlap with this column's family, WARN
(dont FAIL)
            col_aliases = _column_type_aliases(c_md)
            has_overlap = _column_relevant_type_overlap(col_aliases, functions)
            if not has_overlap:
                log.append(
                    f"WARNING:G4: Column '{cname}' has no compatible functions "
                    f"(absent_type_family); type='{c_md.get('type')}', "
                    f"type_category='{c_md.get('type_category')}', "
                    f"type_aliases={sorted(list(col_aliases))[:12]}"
                )
            else:
                orphan_cols.append(cname)
                orphan_details.append(
                    f"DEBUG:G4: Column '{cname}' type='{c_md.get('type')}' "
                    f"type_category='{c_md.get('type_category')}', "
                    f"type_aliases={sorted(list(col_aliases))[:10]}, "
                    f"labels={c_md.get('labels')}"
                )
```

```
    if dead_funcs:
        ok = False
                  log.append(f"FAIL:  Functions/operators  with  no  compatible  columns:
{sorted(dead_funcs)}")
        log.extend(dead_func_details)

    if orphan_cols:
        ok = False
                  log.append(f"FAIL:  Columns  with  no  compatible  functions/operators:
{sorted(orphan_cols)}")
        log.extend(orphan_details)

    if ok:
        log.append("PASS: Compatibility matrix looks healthy.")
    return ok, log


# ---------------------------------------------------------------
# G5. Alias hygiene & collision analysis
# ---------------------------------------------------------------

# ----- G5 helpers: collision classification & domains -----
# NEW: map each column canonical -> set of owning tables (from table metadata)
def _build_column_owners(graph: Dict[str, Any]) -> Dict[str, Set[str]]:
    owners: Dict[str, Set[str]] = defaultdict(set)
    for tbl, node in graph.items():
        if node.get("entity_type") != "table":
            continue
        cols = (node.get("metadata", {}) or {}).get("columns", {}) or {}
        for col_name in cols.keys():
            owners[col_name].add(tbl)
    return owners


# NEW: decide if a pure columncolumn collision is resolvable by table context
def _columns_cross_table_ok(
    targets: List[Tuple[str, str]],
    column_owners: Dict[str, Set[str]],
    debug_log: List[str] | None = None,
) -> bool:
    """
    Returns True when:
      - all targets are columns
      - each column has at least one owning table
         -  owner  sets  are  pairwise  disjoint  (no  shared  table),  so  FROM  context  can
disambiguate.
    """
    cols = [c for (c, et) in targets if et == "column"]
    if len(cols) < 2:
        return False  # not a pure columncolumn collision (or trivial)

    owner_sets = []
    for c in cols:
```

```
        owners = set(column_owners.get(c, set()))
        if debug_log is not None:
            debug_log.append(f"DEBUG:G5: owners[{c}]={sorted(list(owners))}")
        if not owners:
            return False  # unknown ownership  keep as hard conflict
        owner_sets.append(owners)

    # Check pairwise disjointness
    for i in range(len(owner_sets)):
        for j in range(i + 1, len(owner_sets)):
            if owner_sets[i] & owner_sets[j]:
                if debug_log is not None:
                    debug_log.append(
                        f"DEBUG:G5: shared_table between columns: "
                                f"{cols[i]} & {cols[j]} -> {sorted(list(owner_sets[i] &
owner_sets[j]))}"
                    )
                return False

    if debug_log is not None:
        debug_log.append(f"DEBUG:G5: columns_cross_table_ok=True for targets={targets}")
    return True

def _domain_from_entity_type(etype: str) -> str:
    """
    Coarse domain buckets used for G5 policy:
      - 'column', 'table'
      - 'func_sql', 'func_spatial', 'func_pred' (comparison_operators)
      - 'keyword' (select_verbs, prepositions, logical_operators)
    """
    if etype == "column":
        return "column"
    if etype == "table":
        return "table"
    if etype == "sql_actions":
        return "func_sql"
    if etype == "postgis_actions":
        return "func_spatial"
    if etype == "comparison_operators":
        return "func_pred"
    if etype in {"select_verbs", "prepositions", "logical_operators", "filler_words"}:
        return "keyword"
    return "other"


def _summarize_collision(alias: str, targets: List[Tuple[str, str]]) -> Dict[str, Any]:
    """
    Build a concise summary for diagnostics.
    """
    domains = {_domain_from_entity_type(t) for _, t in targets}
    types   = {t for _, t in targets}
    canonicals = [c for c, _ in targets]
    return {
        "alias": alias,
```

```
        "targets": sorted(list({(c, t) for c, t in targets})),
        "domains": sorted(list(domains)),
        "entity_types": sorted(list(types)),
        "count": len({(c, t) for c, t in targets}),
        "canonicals": sorted(canonicals),
    }


def _classify_alias_collision(
    coll_summary: Dict[str, Any],
    column_owners: Dict[str, Set[str]],
    debug_log: List[str] | None = None,
) -> Tuple[str, str, str]:
    """
    Decide severity ('FAIL' | 'WARN' | 'INFO'), a short reason code, and a suggestion.

    Policy (updated):
      - All columns
             * If columns belong to different tables with disjoint ownership -> WARN
(binder can use FROM)
          * Else -> FAIL
      - All SQL (func_sql/func_pred) ............. FAIL (e.g., 'top' -> limit vs max)
      - Cross-domain with spatial vs non-spatial .. WARN
      - Table vs function ......................... WARN
      - Keyword involved .......................... WARN
      - Default ................................... FAIL
    """
    alias = coll_summary["alias"]
    domains = set(coll_summary["domains"])
    etypes  = set(coll_summary["entity_types"])
    targets = coll_summary["targets"]

    # 1) All columns: downgrade to WARN if cross-table resolvable
    if domains == {"column"}:
        if _columns_cross_table_ok(targets, column_owners, debug_log):
            return (
                "WARN",
                "columns_cross_table",
                 "Alias maps to columns in different tables. Binder can disambiguate via
FROM context; consider table-scoped wording if desired."
            )
        return (
            "FAIL",
            "columns_conflict",
             "Alias maps to multiple columns. Remove or qualify the alias on one column
(or add table-scoped wording)."
        )

    # 2) All SQL-ish functions/predicates -> hard conflict (e.g., 'top' -> limit vs max)
    if domains.issubset({"func_sql", "func_pred"}):
        return (
            "FAIL",
            "functions_conflict",
                "Alias maps to multiple SQL functions. Prefer one (edit YAML) or add
```

```
domain-scoped phrasing."
        )

    # 3) Spatial vs non-spatial function/column/table overlap -> warn
    if "func_spatial" in domains and ("func_sql" in domains or "func_pred" in domains or
"column" in domains or "table" in domains):
        return (
            "WARN",
            "cross_domain_spatial_overlap",
            "Ambiguous across spatial/non-spatial. Add domain preference in vocabulary
or require 'of' surfaces."
        )

    # 4) Table vs function -> warn
    if "table" in domains and ("func_sql" in domains or "func_spatial" in domains or
"func_pred" in domains):
        return (
            "WARN",
            "table_vs_function",
            "Alias is both a table and a function. Consider prefix-protection or
removing alias from the table."
        )

    # 5) Keyword overlaps -> warn
    if "keyword" in domains:
        return (
            "WARN",
            "keyword_overlap",
            "Alias overlaps with keyword/reserved use. Prefer keyword role; prune the
other alias."
        )

    # 6) Mixed, but not obviously resolvable -> default to FAIL
    return (
        "FAIL",
        "mixed_conflict",
        "Ambiguous alias across domains. Tighten alias lists or add policy in
vocabulary."
    )


def validate_alias_hygiene(graph: Dict[str, Any], log: List[str]) -> Tuple[bool,
List[str]]:
    log.append("--- Running G5: Alias Hygiene & Collisions ---")
    ok = True

    alias_index: Dict[str, List[Tuple[str, str]]] = defaultdict(list)   # alias ->
[(canonical, etype)]
    multiword_aliases: List[str] = []

    # Build column->tables index once
    column_owners = _build_column_owners(graph)
    log.append(f"DEBUG:G5: column_owners.keys()
sample={sorted(list(column_owners.keys()))[:8]}")
```

```
    # Build alias index
    for canon, node in graph.items():
        et = node.get("entity_type")
        aliases = (node.get("metadata", {}) or {}).get("aliases", []) or []
        for a in aliases:
            a_str = str(a).strip()
            if not a_str:
                continue
            key = a_str.lower()
            alias_index[key].append((canon, et))
            if " " in key:
                multiword_aliases.append(key)

    # Collisions & domain conflicts
    collisions: List[Tuple[str, List[Tuple[str, str]]]] = []
    classified: List[Tuple[str, Dict[str, Any], Tuple[str, str, str]]] = []  # (alias, summary, (severity,reason,suggestion))
    for alias, targets in alias_index.items():
        uniques = sorted(list({(c, t) for (c, t) in targets}))  # unique (canonical, etype)
        if len(uniques) > 1:
            collisions.append((alias, uniques))
            summary = _summarize_collision(alias, uniques)
            severity, reason, suggestion = _classify_alias_collision(summary, column_owners, log)
            classified.append((alias, summary, (severity, reason, suggestion)))

    # Domain conflicts (purely informational)
    domain_conflicts: List[Tuple[str, Set[str]]] = []
    for alias, targets in alias_index.items():
        doms = {_domain_from_entity_type(t) for _, t in set(targets)}
        if len(doms) > 1:
            domain_conflicts.append((alias, doms))

    # Prefix collisions (single-word alias that prefixes multi-word)
    prefix_to_longers: Dict[str, List[str]] = defaultdict(list)
    for mw in multiword_aliases:
        pfx = mw.split()[0]
        prefix_to_longers[pfx].append(mw)

    prefix_issues: List[Tuple[str, List[str]]] = []
    for pfx, longers in prefix_to_longers.items():
        if pfx in alias_index:
            prefix_issues.append((pfx, sorted(longers)))

    # Reserved overlaps
    reserved_violations: List[str] = []
    for alias, targets in alias_index.items():
        if alias in _RESERVED_TOKENS:
            domains = {t for (_, t) in set(targets)}
            expected_ok = any(d in {"prepositions", "logical_operators", "comparison_operators", "select_verbs"} for d in domains)
            if not expected_ok:
```

```
                reserved_violations.append(alias)

    # Echo anomalies
    echo_anomalies: List[Tuple[str, List[Tuple[str, str]]]] = []
    for alias, targets in alias_index.items():
        canonicals = {c for (c, _) in targets}
        if alias in canonicals and len(canonicals) > 1:
            echo_anomalies.append((alias, sorted(list({(c, t) for (c, t) in targets}))))

    # ---- Reporting with policy-aware severities ----
    # 1) Reserved token misuse -> FAIL
    if reserved_violations:
        ok = False
        log.append("FAIL: Reserved tokens used as aliases for non-reserved domains:")
        for a in sorted(reserved_violations):
            log.append(f"  - '{a}'")

    # 2) Collisions classified
    fails = [x for x in classified if x[2][0] == "FAIL"]
    warns = [x for x in classified if x[2][0] == "WARN"]
    infos = [x for x in classified if x[2][0] == "INFO"]

    if fails:
        ok = False
        log.append("FAIL: Alias collisions that require fixes:")
          for alias, summary, (sev, reason, suggestion) in sorted(fails, key=lambda z:
z[0]):
            log.append(f"  - '{alias}' [{reason}] targets={summary['targets']}")
            log.append(f"    suggestion: {suggestion}")

    if warns:
             log.append("WARNING: Alias collisions that are policy-manageable (review
suggested):")
          for alias, summary, (sev, reason, suggestion) in sorted(warns, key=lambda z:
z[0]):
            log.append(f"  - '{alias}' [{reason}] targets={summary['targets']}")
            log.append(f"    suggestion: {suggestion}")

    if infos:
        log.append("INFO: Benign alias overlaps:")
          for alias, summary, (sev, reason, suggestion) in sorted(infos, key=lambda z:
z[0]):
            log.append(f"  - '{alias}' [{reason}] targets={summary['targets']}")

    # 3) Cross-domain overlaps (pure FYI)
    if domain_conflicts:
        log.append("WARNING: Cross-domain alias overlaps:")
        for alias, doms in sorted(domain_conflicts):
            log.append(f"  - '{alias}' spans domains {sorted(list(doms))}")

    # 4) Prefix issues (FYI)
    if prefix_issues:
            log.append("WARNING: Prefix collisions (single-word alias prefixes longer
keys):")
```

```
        for pfx, longers in sorted(prefix_issues):
            log.append(f"  - '{pfx}' prefixes: {longers}")

    # 5) Echo anomalies (FYI)
    if echo_anomalies:
        log.append("WARNING: Echo/identity anomalies (alias equals a canonical but maps to others too):")
        for alias, tgs in sorted(echo_anomalies):
            log.append(f"  - '{alias}'  {tgs}")

    if ok and not (fails or reserved_violations):
        log.append("PASS: Alias hygiene looks good (with possible warnings).")

    # Final debug rollup
    log.append(
        f"DEBUG:G5: collisions_total={len(classified)} "
        f"(fails={len(fails)}, warns={len(warns)}, infos={len(infos)}); "
        f"reserved_violations={len(reserved_violations)}; "
        f"prefix_issues={len(prefix_issues)}; echo_anomalies={len(echo_anomalies)}"
    )
    return ok, log

# --------------------------------------------------------------
# G6. Pluralization & inflection sanity
# --------------------------------------------------------------

def validate_pluralization(graph: Dict[str, Any], log: List[str]) -> Tuple[bool, List[str]]:
    log.append("--- Running G6: Pluralization & Inflection Sanity ---")
    ok = True

    # Build aliascanonicals for tables/columns only
    alias_to_targets: Dict[str, Set[str]] = defaultdict(set)
    for canon, node in graph.items():
        et = node.get("entity_type")
        if et not in {"table", "column"}:
            continue
        for a in (node.get("metadata", {}) or {}).get("aliases", []) or []:
            alias_to_targets[str(a).lower()].add(canon)

    missing_pairs: List[Tuple[str, str]] = []  # (expected_missing_alias, reason)
    for alias in alias_to_targets.keys():
        parts = alias.split()
        if not parts:
            continue
        last = parts[-1]
        if last in _PLURAL_LASTWORD:
            plural = _PLURAL_LASTWORD[last]
            alias_plural = " ".join(parts[:-1] + [plural])
            if alias_plural not in alias_to_targets:
                missing_pairs.append((alias_plural, f"Plural form of '{alias}' expected."))

    if missing_pairs:
```

```
        # not a hard FAIL by default, but useful signal
        log.append("WARNING: Missing expected plural aliases:")
        for a, why in sorted(missing_pairs):
            log.append(f"  - '{a}': {why}")
    else:
        log.append("PASS: Pluralization looks sane.")
    return ok, log

# --------------------------------------------------------------
# G7. Reserved-token budget & tokenization safety
# --------------------------------------------------------------

def validate_reserved_token_safety(graph: Dict[str, Any], log: List[str]) -> Tuple[bool,
List[str]]:
    log.append("--- Running G7: Reserved Token Safety ---")
    ok = True

    offenders: List[Tuple[str, str]] = []  # (canonical, alias)
    for canon, node in graph.items():
        aliases = (node.get("metadata", {}) or {}).get("aliases", []) or []
        for a in aliases:
            s = str(a).strip().lower()
            if s in {",", "&&", "||"}:
                # Only acceptable if canonical belongs to expected domains
                et = node.get("entity_type")
                if et not in {"logical_operators"} and s in {"&&", "||"}:
                    offenders.append((canon, s))
                if s == ",":
                    # comma should be provided by grammar, not aliases
                    offenders.append((canon, s))

    if offenders:
        ok = False
                     log.append("FAIL:  Tokenization  hazards  found  (aliases  using
punctuation/operators):")
        for c, a in sorted(offenders):
            log.append(f"  - canonical='{c}' alias='{a}'")
    else:
        log.append("PASS: No tokenization hazards detected.")
    return ok, log

# --------------------------------------------------------------
# G8. Stability & reproducibility
# --------------------------------------------------------------

def _normalize_graph_for_compare(graph: Dict[str, Any]) -> Dict[str, Any]:
    """
    Produce a deterministic deep-sorted copy for stable comparison.
    - Sort alias lists
    - Sort columns under tables
    - Remove non-deterministic keys if any (none by default)
    """
    g = copy.deepcopy(graph)
    for canon, node in g.items():
```

```
        md = node.get("metadata", {}) or {}
        # sort aliases
        if "aliases" in md and isinstance(md["aliases"], list):
            md["aliases"] = sorted([str(a) for a in md["aliases"]])
        # sort labels
        if "labels" in md and isinstance(md["labels"], list):
            md["labels"] = sorted([str(a) for a in md["labels"]])
        # sort columns map deterministically
        if node.get("entity_type") == "table":
            cols = md.get("columns", {}) or {}
            ordered = {}
            for k in sorted(cols.keys()):
                col = cols[k]
                if isinstance(col, dict):
                    if "aliases" in col and isinstance(col["aliases"], list):
                        col["aliases"] = sorted([str(a) for a in col["aliases"]])
                    if "labels" in col and isinstance(col["labels"], list):
                        col["labels"] = sorted([str(a) for a in col["labels"]])
                ordered[k] = col
            md["columns"] = ordered
        node["metadata"] = md
        g[canon] = node
    # return with deterministic top-level order (Python 3.7+ dicts keep insertion order)
    return dict(sorted(g.items(), key=lambda x: x[0]))

def validate_graph_stability(
    build_fn: Callable[[Dict[str, Any], Dict[str, Any]], Dict[str, Any]],
    schema_yaml: Dict[str, Any],
    keywords_yaml: Dict[str, Any],
    log: List[str],
) -> Tuple[bool, List[str]]:
    log.append("--- Running G8: Stability & Reproducibility ---")
    ok = True
    g1 = _normalize_graph_for_compare(build_fn(schema_yaml, keywords_yaml))
    g2 = _normalize_graph_for_compare(build_fn(schema_yaml, keywords_yaml))
    if g1 != g2:
        ok = False
            log.append("FAIL:  Two  consecutive  graph  builds  differ  (non-deterministic
output).")
        # minimal diff signal
        k1 = set(g1.keys()); k2 = set(g2.keys())
        if k1 != k2:
            log.append(f"  - Canonical key set differs: missing={sorted(list(k1 - k2))}
extra={sorted(list(k2 - k1))}")
        else:
            # same keys; probe a few differing nodes
            diffs = []
            for k in sorted(k1):
                if g1[k] != g2[k]:
                    diffs.append(k)
                    if len(diffs) >= 5: break
            log.append(f"  - Example differing nodes: {diffs}")
    else:
        log.append("PASS: Graph builds are stable and reproducible.")
```

```
    return ok, log

# ------------------------------------------------------------
# Convenience: run the whole suite
# ------------------------------------------------------------

def validate_graph_all(
    graph: Dict[str, Any],
    log: List[str],
    *,
      build_fn: Optional[Callable[[Dict[str, Any]], Dict[str, Any]]] = 
None,
    schema_yaml: Optional[Dict[str, Any]] = None,
    keywords_yaml: Optional[Dict[str, Any]] = None,
) -> Tuple[bool, List[str]]:
    """
    Runs G1G7 against `graph`. If build_fn & source yamls provided, runs G8 too.
    Returns (ok, log). Individual warnings/errors are appended to `log`.
    """
    results = []

       ok1, log = validate_graph_structure(graph, log); 
results.append(ok1)
         ok2, log = validate_graph_referential_integrity(graph, log); 
results.append(ok2)
         ok3, log = validate_graph_type_label_coherence(graph, log); 
results.append(ok3)
          ok4, log = validate_function_compatibility_matrix(graph, log); 
results.append(ok4)
       ok5, log = validate_alias_hygiene(graph, log); 
results.append(ok5)
       ok6, log = validate_pluralization(graph, log); 
results.append(ok6)
        ok7, log = validate_reserved_token_safety(graph, log); 
results.append(ok7)

    if build_fn and schema_yaml is not None and keywords_yaml is not None:
        ok8, log = validate_graph_stability(build_fn, schema_yaml, keywords_yaml, log)
        results.append(ok8)

    overall_ok = all(results)
    if overall_ok:
        log.append("PASS: Graph validations (G1G8) passed.")
    else:
        log.append("FAIL: One or more graph validations failed.")
    return overall_ok, log
```

src/n2s_validators/normalizer_validator.py

```python
# File: src/validation/normalizer_validator.py
import random
from collections import defaultdict
from functools import partial
from lark import ParseError, UnexpectedToken

# These components are needed for the integration test
from src.n2s_runtime.normalizer import normalize_text
from src.n2s_validators.grammar_validator import SmartGenerator, GrammarAnalyzer

def validate_normalizer_spot_check(norm_map, log, num_spot_checks=50):
    """
    V2.5: A unit-level check that validates a sample of aliases from the
    normalization_map to ensure the Normalizer class is working correctly.
    """
    log.append("--- Running V2.5: Normalizer Spot-Check ---")

    try:
        normalizer = partial(normalize_text, norm_map)
        all_aliases = list(norm_map.get('deterministic_aliases', {}).items())

        if not all_aliases:
            log.append("WARNING: No deterministic aliases found to spot-check.")
            return True, log

        sample_size = min(num_spot_checks, len(all_aliases))
        alias_sample = random.sample(all_aliases, sample_size)

        for alias, canonical in alias_sample:
            expected_output = canonical if canonical is not None else ""

            # --- FIX IS HERE ---
            # normalize() returns a list of candidates. We expect one for a deterministic check.
            candidates = normalizer(alias)
            normalized_output = candidates[0] if candidates else ""
            # --- END OF FIX ---

            if normalized_output != expected_output:
                log.append("FAIL: Normalizer failed spot-check.")
                log.append(f"     - Input Alias: '{alias}'")
                log.append(f"     - Expected Canonical: '{expected_output}'")
                log.append(f"     - Actual Output: '{normalized_output}'")
                return False, log

        log.append(f"PASS: Normalizer correctly mapped all {sample_size} spot-checked aliases.")
        return True, log

    except Exception as e:
        log.append(f"FAIL: An unexpected error occurred during the spot-check: {e}")
```

```python
        return False, log



def _setup_integration_test(parser, graph, norm_map):
    analyzer = GrammarAnalyzer(parser)
    generator = SmartGenerator(parser, graph, analyzer)
    normalizer = partial(normalize_text, norm_map)

    CONNECTORS = {"of", "by", "to", "with", "and", "from"}

    reverse_alias_map = defaultdict(list)
    full_alias_map = {**norm_map.get('deterministic_aliases', {}),
                      **norm_map.get('non_deterministic_aliases', {})}

    for alias, canonical_or_list in full_alias_map.items():
        if canonical_or_list is None:
            continue
            canonicals = canonical_or_list if isinstance(canonical_or_list, list) else
[canonical_or_list]
        for can in canonicals:
            # prune pathological alias forms for testing (optional)
            a = alias.strip()
            if not a:
                continue
            if any(a.lower().startswith(c + " ") for c in CONNECTORS):
                # leading connector tends to create nonsense when substituted per-token
                continue
            reverse_alias_map[can].append(a)

    return normalizer, generator, reverse_alias_map

def _denormalize_phrase(canonical_phrase, reverse_alias_map):
    """
    Converts a canonical phrase into a messy phrase by choosing aliases,
    but avoids connector duplication (e.g., 'average of of').
    If we must use an alias that already ends with a connector and the
    next canonical token is that same connector, we consume (skip) the
    next token to prevent duplication.
    """
    CONNECTORS = {"of", "by", "to", "with", "and", "from"}
    LOCK_CANONICAL = {",", "COMMA"} | CONNECTORS  # don't alias punctuation/connectors

    def bucket_aliases(aliases):
        """
        Partition aliases into: plain (no trailing connector) and endswith[connector].
        """
        endswith = {c: [] for c in CONNECTORS}
        plain = []
        for a in aliases:
            s = a.strip()
            s_low = s.lower()
            matched = False
            for c in CONNECTORS:
```

```
                if s_low.endswith(" " + c):
                    endswith[c].append(s)
                    matched = True
                    break
            if not matched:
                plain.append(s)
        return plain, endswith

    toks = canonical_phrase.split()
    out = []
    i = 0
    while i < len(toks):
        t = toks[i]

        # never alias commas/connectors; keep them literal
        if t in LOCK_CANONICAL or t == ",":
            out.append("," if t in {",", "COMMA"} else t)
            i += 1
            continue

        # alias choices for this canonical token
        choices = reverse_alias_map.get(t, None)
        if not choices:
            # fallback: keep canonical if we have no alias choices
            out.append(t)
            i += 1
            continue

        nxt = toks[i + 1].lower() if i + 1 < len(toks) else None
        plain, endswith = bucket_aliases(choices)

        if nxt in CONNECTORS:
            if plain:
                # Prefer aliases that do NOT already include the connector
                out.append(random.choice(plain))
                i += 1
            elif endswith[nxt]:
                 # If only connector-ending aliases exist, consume the next connector to
avoid duplication
                out.append(random.choice(endswith[nxt]))
                i += 2  # skip the next connector token
            else:
                # No connector-matched alias; fall back to any plain or any ending
                pool = plain or [a for lst in endswith.values() for a in lst]
                out.append(random.choice(pool))
                i += 1
        else:
            # No connector following; any alias is fine (prefer plain for stability)
              out.append(random.choice(plain or [a for lst in endswith.values() for a in
lst]))
            i += 1

    return " ".join(out)
```

```
def _run_pipeline_on_phrase(messy_phrase, normalizer, parser, log):
    """
    Runs a single phrase through the Normalizer -> Parser pipeline,
    with enhanced logging.
    """
    # --- ADDED LOGGING ---
    log.append(f"\n    --- V2.6 Sub-Test ---")
    log.append(f"    - De-Normalized Phrase: '{messy_phrase}'")

    normalized_candidates = normalizer(messy_phrase)
     log.append(f"     - Normalizer Produced {len(normalized_candidates)} Candidate(s):
{normalized_candidates}")

    _errors = []
    successful_candidate = None
    size_nc=len(normalized_candidates)
    cnt=0
    for candidate in normalized_candidates:
        cnt=cnt+1
        try:
            if candidate:
                parser.parse(candidate)
                successful_candidate = candidate
                # We found a valid interpretation, so we can stop.
                return True, [], normalized_candidates, log
        except Exception as e:
            _errors.append(f"error {cnt}/{size_nc}:\n{e}")
            continue

    # (ParseError, UnexpectedToken)
    # If we get here, no candidate succeeded
    log.append(f"    - Result: All candidates failed to parse.")
    return False, _errors, normalized_candidates, log



# --- The Main V2.6 Validator ---

def validate_normalizer_integration(graph, parser, norm_map, log, num_phrases=50,
success_threshold=0.90):
    """
      V2.6: Tests the full Generator -> Normalizer -> Parser pipeline using helper
functions.
    """
    log.append("--- Running V2.6: Normalizer-Parser Integration Check ---")

    try:
           normalizer, generator, reverse_map = _setup_integration_test(parser, graph,
norm_map)
        success_count, failures = 0, 0
        failed_examples = []

        for _ in range(num_phrases):
```

```
            # Stage A: Generate a canonical phrase
            canonical_phrase, _ = generator.generate()
            if not canonical_phrase: continue

            # Stage B: De-normalize it
            messy_phrase = _denormalize_phrase(canonical_phrase, reverse_map)

            # Stage C: Run it through the pipeline
              is_success, errors, candidates, log= _run_pipeline_on_phrase(messy_phrase,
normalizer, parser,log)

            if is_success:
                success_count += 1
            else:
                failures += 1
                if len(failed_examples) < 3:
                    failed_examples.append({
                        'original': canonical_phrase, 'messy': messy_phrase,
                        'normalized': candidates, 'errors': f'{errors}'
                    })

        # Report Results
        total = success_count + failures
        if total == 0:
            log.append("WARNING: No phrases were generated to test.")
            return True, log

        rate = success_count / total
        log.append(f"  - Integration test complete. Success Rate: {rate:.0%}")
        if rate >= success_threshold:
            log.append(f"PASS: Success rate ({rate:.0%}) meets threshold.")
            return True, log
        else:
            log.append(f"FAIL: Success rate ({rate:.0%}) is below threshold.")
            log.append("  - Example Failures:")
            for fail in failed_examples:
                log.append(f"    - Original: '{fail['original']}'")
                log.append(f"    - Messy:    '{fail['messy']}'")
                log.append(f"    - Normalized To: {fail['normalized']}")
                log.append(f"    - Failure Reasons: {fail['errors']}")
            return False, log

    except Exception as e:
           log.append(f"FAIL: An unexpected error occurred during the integration test:
{e}")
        return False, log
```

src/n2s_validators/source_validator.py

```
def validate_schema_yaml(schema_data, log):
    """V0.1: Performs a deep validation of the schema.yaml structure."""
    log.append("--- Running V0.1: Schema.yaml Structural Check ---")

    if not isinstance(schema_data, dict) or 'tables' not in schema_data:
        log.append("FAIL: schema.yaml must be a dictionary with a top-level 'tables' key.")
        return False, log

    if not isinstance(schema_data['tables'], dict):
        log.append("FAIL: The 'tables' key must contain a dictionary of tables.")
        return False, log

    for table_name, table_data in schema_data['tables'].items():
        if not isinstance(table_data, dict) or 'columns' not in table_data:
            log.append(f"FAIL: Entry for table '{table_name}' must be a dictionary with a 'columns' key.")
            return False, log

        if not isinstance(table_data['columns'], dict):
            log.append(f"FAIL: The 'columns' key in table '{table_name}' must contain a dictionary.")
            return False, log

        for col_name, col_data in table_data['columns'].items():
            if not isinstance(col_data, dict):
                log.append(f"FAIL: Entry for column '{col_name}' in table '{table_name}' must be a dictionary.")
                return False, log
            if 'type' not in col_data or not isinstance(col_data['type'], str):
                log.append(f"FAIL: Column '{col_name}' in table '{table_name}' is missing a 'type' string.")
                return False, log
            if 'aliases' not in col_data or not isinstance(col_data['aliases'], list):
                log.append(f"FAIL: Column '{col_name}' in table '{table_name}' is missing an 'aliases' list.")
                return False, log

    log.append("PASS: schema.yaml has the expected structure.")
    return True, log


def validate_keywords_yaml(keywords_data, log):
    """V0.2: Validates the basic structure of keywords_and_functions.yaml."""
    log.append("--- Running V0.2: Keywords.yaml Structural Check ---")

    if not isinstance(keywords_data, dict):
        log.append("FAIL: keywords.yaml should be a dictionary.")
        return False, log

    for required_section in ['keywords', 'sql_actions', 'postgis_actions']:
```

```
        if required_section not in keywords_data:
            log.append(f"FAIL: keywords.yaml is missing required top-level key '{required_section}'.")
            return False, log

    # Check the structure within the 'keywords' section
    for keyword_type, keyword_data in keywords_data['keywords'].items():
        # --- FIX IS HERE: Add an exception for 'global_templates' ---
        # This section has a unique structure and should be skipped by this check.
        if keyword_type == 'global_templates':
            continue
        # --- END OF FIX ---

        if not isinstance(keyword_data, dict):
            log.append(f"FAIL: Section '{keyword_type}' in 'keywords' must be a dictionary (e.g., canonical_name: {{'aliases': [...]}}).")
            return False, log

        for canonical_name, metadata in keyword_data.items():
            if not isinstance(metadata, dict) or 'aliases' not in metadata:
                log.append(f"FAIL: Entry '{canonical_name}' in '{keyword_type}' must be a dictionary with an 'aliases' key.")
                return False, log
            if not isinstance(metadata['aliases'], list):
                log.append(f"FAIL: The 'aliases' key for '{canonical_name}' in '{keyword_type}' must contain a list.")
                return False, log

    # (The validation for sql_actions and postgis_actions remains the same)
    for action_type in ['sql_actions', 'postgis_actions']:
        for func_name, func_data in keywords_data[action_type].items():
            if not isinstance(func_data, dict):
                log.append(f"FAIL: Entry for function '{func_name}' in '{action_type}' must be a dictionary.")
                return False, log
            if 'aliases' not in func_data or not isinstance(func_data['aliases'], list):
                log.append(f"FAIL: Function '{func_name}' in '{action_type}' is missing an 'aliases' list.")
                return False, log
            if 'template' not in func_data or not isinstance(func_data['template'], str):
                log.append(f"FAIL: Function '{func_name}' in '{action_type}' is missing a 'template' string.")
                return False, log

    log.append("PASS: keywords_and_functions.yaml has the expected structure.")
    return True, log
```

src/n2s_validators/validator_plan.md

1) Graph validations (prevocabulary/binder/grammar)

Goal: ensure the raw knowledge (schema + keywords/functions) is complete, coherent, and predictable. If the graph is shaky, every downstream artifact inherits the wobble.

G1. Structural shape & schema hygiene

* **What:** Every node has the required fields (`entity_type`, `metadata`), expected subkeys (e.g., for tables `columns`; for columns `type`, `aliases`, `labels`), and correct types (lists for aliases, strings for types).
* **Why:** Prevents NoneType crashes and serialization issues later.
* **How:** A strict schema validator that asserts presence and type of each field. Also assert YAML-serializability (no tuples or unserializable objects).

G2. Referential integrity

* **What:** Columns appear under a table and as top-level column nodes, with consistent `type`, `labels`, and `aliases`. Optional: FK/relationship edges if you capture them.
* **Why:** Prevents split-brain views of the same entity.
* **How:** Cross-compare table->column entries with top-level column nodes; assert consistency.

G3. Type/label coherence

* **What:** Column `type` belongs to the known type system (e.g., `INT`, `FLOAT`, `geography_point`), labels are from a whitelisted vocabulary (e.g., `id`, `postgis`).
* **Why:** Later compatibility checks rely on these enums.
* **How:** Check against a canonical type set and label set.

G4. Function compatibility matrix

* **What:** For each functions `applicable_types` (and label rules), at least one real column is compatible; for each column, at least one function is compatible.
* **Why:** Detect orphan columns or dead functions early.
* **How:** Evaluate `is_compatible(column, function)` across the matrix. Report zeros.

G5. Alias hygiene & collision analysis

* **What:** Build an aliastarget index and flag:

  * **Prefix collisions** (e.g., `user` vs `user id`).
  * **Multi-domain collisions** (e.g., `intersects`   `st_spatial_index` vs `st_intersects`).
  * **Reserved/preposition overlaps** (`in`, `of`, `from`, etc.).
  * **Echo/identity anomalies** (alias identical to canonical across domains).
* **Why:** These are the root of many normalization 0 candidate failures.
* **How:** Deterministic checks over the alias index; produce a diagnostics section.

G6. Pluralization & inflection sanity

* **What:** Expected pairs exist or are intentionally absent (e.g., `order date` `order

dates`, `item` `items`).
* **Why:** Eliminates brittle NL failures.
* **How:** Audit last-word plural rules and diffs; emit diagnostics for missing/plural forms.

G7. Reserved-token budget & tokenization safety

* **What:** Ensure punctuation/operators (`,` `&&` `||` etc.) and reserved keywords arent assigned as ambiguous aliases.
* **Why:** Tokenizer and grammar stability.
* **How:** Simple deny-list pass over aliases; flag violations.

G8. Stability & reproducibility

* **What:** Rebuild graph twice and diff. Ordering and content should be stable.
* **Why:** Flaky graphs create flaky downstream artifacts.
* **How:** Deterministic sort, deterministic alias generation, and snapshot diff.

---

2) Vocabulary validations

Goal: verify the alias space maps cleanly to canonical tokens with clear determinism/ambiguity boundaries, and that policies (e.g., of behavior, domain preferences) are respected.

V1. Coverage

* **What:** Every alias in the graph appears as a key in vocabulary (deterministic or non-deterministic).
* **Why:** Coverage gaps = leftmost-unmapped failures.
* **How:** Set difference of graph aliases vs vocabulary keys.

V2. Determinism partition sanity

* **What:** Single-word, single-meaning aliases in deterministic; multi-word or multi-meaning in non-deterministic; filler words map to `""`.
* **Why:** Predictable normalization behavior.
* **How:** Recompute the aliastargets index and assert partitioning rules.

V3. Policy enforcement

* **What:** Check that:

  * **Preposition purity** (`of`, `from`, `in`,) are not overloaded.
  * **Domain preferences** applied (e.g., `intersects` prefer `st_intersects` over `st_spatial_index`).
  * **of-surface policy** applied (e.g., `unique values of` `distinct of`).
  * **Prefix protection** applied where configured.
* **Why:** Prevents structural nonsense (`of` mapped to columns) and semantic ambiguity.
* **How:** Deterministic rules over the vocabulary contents; produce diagnostics.

V4. Identity & canonical presence

* **What:** Every canonical token is present as an identity alias (so canonical text normalizes to itself).
* **Why:** Allows canonical phrases to pass through the normalizer untouched.
* **How:** Assert `vocab[canonical]` includes canonical.

V5. Fan-out & entropy checks

* **What:** Measure alias fan-out (aliasesN canonicals) and canonical fan-in (canonicalsN aliases). Flag extremes.
* **Why:** Very high ambiguity often signals modeling errors or overly broad aliases.
* **How:** Compute distributions; compare vs thresholds.

V6. Serialization safety

* **What:** YAML round-trip produces the same structure; no Python tuples, etc.
* **Why:** Avoid runtime loader errors.
* **How:** Loaddumpload comparison + allowed-type assertions.

---

3) Binder validations

Goal: ensure the binder (the structure-aware layer) can actually assemble well-typed queries from token streams and that its templates are sound.

B1. Shape & schema of binder templates

* **What:** Each template has an id, slots (with types/labels), connector requirements (e.g., needs `OF` between function and arg), cardinality rules (lists with `,` and optional `AND`), and fallbacks/defaults.
* **Why:** Prevents runtime pattern-matching errors.
* **How:** Static schema validation on the binder spec.

B2. Linkage to graph/vocabulary

* **What:** Every referenced canonical (function, column, table, connector) exists in graph and vocabulary.
* **Why:** No dangling references.
* **How:** Cross-check binder refs against graph entities and vocabulary keys/canonicals.

B3. Unifiability & type satisfaction

* **What:** For each binder template, there exists at least one legal (function, column-list, table) triple that satisfies all slot types and label rules.
* **Why:** Detect dead templates early.
* **How:** Programmatic search over the compatibility matrix; prove at least one instantiation per template.

B4. Ambiguity & cost model

* **What:** For common input patterns, binder should not explode into thousands of equivalent bindings; if it does, warn and require weights.
* **Why:** Keeps normalization+binding tractable.

* **How:** Feed canonical token sequences through binder matcher; measure number of valid bindings; enforce thresholds and cost-pruning hints.

B5. Connector/OF rules sanity

* **What:** Check that `OF` is required where expected (e.g., `function OF column_list`), commas+AND are handled, and duplication (e.g., `of of`) is forbidden.
* **Why:** Avoid classic aliasconnector collisions.
* **How:** Table-driven checks against binder connector rules.

B6. Dead/overlapping templates

* **What:** Identify templates never chosen or shadowed by broader ones.
* **Why:** Simplifies binder and reduces ambiguity.
* **How:** Coverage analysis using generated canonical inputs.

---

4) Grammar validations

Goal: guarantee the parser accepts all intended canonical shapes and isnt ambiguous or brittle.

Gm1. Vocabulary alignment

* **What:** Every canonical table/column/function in the graph has a matching terminal literal in the grammar.
* **Why:** Complete parsability of canonical text.
* **How:** Parse the grammar text and verify terminal lists match graph entities (by name).

Gm2. Canonical smoke tests

* **What:** Generate simple canonical phrases (`select col from table`, `select func of col from table`, lists) and parse successfully.
* **Why:** Catch missing tokens/rules early.
* **How:** Deterministic phrase factory parse assert success.

Gm3. Stress tests (canonical)

* **What:** Use a smart canonical generator to create complex, long phrases; all must parse.
* **Why:** Surface hidden recursion/ambiguity.
* **How:** SmartGeneratorparse; compute success rate and record failures.

Gm4. Ambiguity checks

* **What:** For canonical inputs, parse forest size should be 1 (or within strict bounds).
* **Why:** Ambiguity complicates binder/ranker later.
* **How:** Ask parser for parse forest size (or instrument with GLR/LALR diagnostics) and warn on >1.

Gm5. Grammar health

* **What:** No unreachable rules, no infinite recursion, bounded minimal depth per rule, reserved tokens only where intended.
* **Why:** Maintainability and performance.
* **How:** Static analysis: reachability from `start`, recursion depth computation, FIRST/FOLLOW diagnostics.

---

5) Cross-artifact validations (without the normalizer)

These tests exercise **graph + vocabulary + binder + grammar** with **canonical tokens only** (no alias noise).

C1. Canonical-to-binder-to-grammar roundtrip

* **What:** Given canonical token sequences (e.g., from SmartGenerator), binder must produce a valid binding; the corresponding canonical string must parse.
* **Why:** Ensures binder and grammar agree on structure.
* **How:** Generate canonical sequences bind serialize to canonical text (with connectors) parse.

C2. Binder SQL feasibility check (optional)

* **What:** For successful bindings, it should be possible to map them to a SQL template (even if SQL generation is out of scope here).
* **Why:** Prevents structurally-valid-but-unexecutable combos.
* **How:** Check each bound functions SQL template can be fully populated from bound slots.

C3. Negative canonical tests

* **What:** Intentionally malformed canonical sequences should fail binding or parsing cleanly with meaningful reasons.
* **Why:** Confirms good error surfaces.
* **How:** Generate illegal sequences (missing FROM, function without OF, etc.) assert clear failure categories.

---

6) Full integration validations (requires the normalizer)

These prove the pipeline works from **messy language** normalize (vocabulary) **bind (binder)** **parse (grammar)**.

I1. De-normalization + re-normalization loop

* **What:** Start from canonical phrases, replace canonicals with aliases (respecting connectors via the binders connector rules), run through normalizer binder grammar. Expect a high success rate.
* **Why:** Detect lossiness + alias-connector collisions in practice.
* **How:** Controlled de-normalizer (connector-aware), then feed to normalizer; count candidates; ask binder to bind each; parse best candidate.

I2. Lossiness & coverage audit

* **What:** Track which canonical bits are lost during normalization (e.g., alias mapped to multiple canonicals), and whether binder constraints recover structure robustly.
* **Why:** Prioritize vocabulary/binder fixes where recovery fails.
* **How:** Log candidate counts at each stage; classify failures (leftmost-unmapped, over-ambiguous, connector duplication, incompatible types).

I3. Golden-set NL queries

* **What:** A curated set of realistic NL queries passes through the full pipeline to a parse/bind (and optionally SQL).
* **Why:** Realistic acceptance criteria beyond synthetic canonical tests.
* **How:** Fixed test set; report per-query diagnostics and categories of failure.

What can be tested **without** the normalizer vs. **requires** it

Testable *without* the normalizer

* Graph structure, referential integrity, type/label coherence, function compatibility.
* Alias collision analysis as a static property of the graph (coverage, partitioning rules as *intent*).
* Vocabulary **structure** (coverage, partitioning, policies), identity-canonical presence, fan-out bounds, serialization.
* Binder **soundness** on canonical tokens: shape, linkage to graph, unifiability, ambiguity, dead/overlapping templates, connector rules.
* Grammar vocabulary alignment, canonical smoke/stress tests, ambiguity checks, grammar health.
* Cross-artifact canonical roundtrips (canonical binder grammar) and feasibility checks.

Tests that **require** the normalizer

* End-to-end messy language handling (aliases, multi-word fragments, punctuation, and connector collisions).
* Lossiness/ambiguity recovery in practice (how well binder constraints rebuild structure from normalized candidates).
* Success-rate metrics for de-normalized phrases; identifying failure buckets like leftmost-unmapped spans or of of duplication.
* Golden-set NL queries that include varied human phrasing.

Why this matters

* **Early isolation:** Graph and vocabulary errors manifest as no candidates later; catching them at G/V/B/Gm levels prevents chasing ghosts in I1/I3.
* **Binder as guardrails:** Its the structural antidote to normalization lossiness. Validating binder unifiability and connector rules is the single biggest boost to integration stability.
* **Grammar predictability:** A tight canonical grammar + binder lets you keep normalization simpler (and intentionally lossy) while still recovering the intended

structure.

src/n2s_validators/vocabulary_validator.py

```python
# src/n2s_validators/vocabulary_validator.py

from __future__ import annotations
from typing import Dict, Any, List, Tuple, Set, Optional, Iterable
from collections import defaultdict, Counter
import copy

# Optional YAML round-trip check for V6
try:
    import yaml
    _HAVE_YAML = True
except Exception:
    _HAVE_YAML = False

# ------------------------------------------------------------
# Policy knobs (same semantics as compiler)
# ------------------------------------------------------------

PREP_BARE: Set[str] = {"of", "from", "in", "on", "at"}   # preposition purity
OF_CANONICALS: Set[str] = {"distinct", "avg", "sum", "st_distance"}

GENERIC_DENY: Set[str] = {"order by", "sort by", "by", "ascending", "descending",
"order"}

def _build_graph_alias_index(graph: Dict[str, Any]) -> Dict[str, List[Tuple[str, str]]]:
    """
    alias (lowercased) -> [(canonical, entity_type), ...] from the graph's metadata.
    """
    idx: Dict[str, List[Tuple[str, str]]] = defaultdict(list)
    for canon, node in graph.items():
        et = node.get("entity_type")
        md = node.get("metadata", {}) or {}
        for a in (md.get("aliases", []) or []):
            s = str(a).strip().lower()
            if s:
                idx[s].append((canon, et))
    return idx

def _should_require_identity(canonical: str, entity_type: Optional[str]) -> bool:
    """
    Identity (c -> c) is required for true canonicals, not meta/diagnostic keys.
    """
    if canonical.startswith("_"):
        return False
    if entity_type is None:
        return False
    # Meta buckets we never require identities for:
    if entity_type not in {
        "table", "column",
        "sql_actions", "postgis_actions", "comparison_operators",
        "select_verbs", "prepositions", "logical_operators", "filler_words"
```

```
    }:
        return False
    return True


# Prefer spatial vs textual when both exist for the same alias
DOMAIN_PREFER_SPATIAL = {
    "intersects": ("st_spatial_index", "st_intersects"),
    "overlaps":   ("st_spatial_index", "st_intersects"),
    "contains":   ("like", "st_contains"),
}

# Reserved tokens (shouldnt be aliases except for their own domains)
RESERVED_TOKENS: Set[str] = {",", "&&", "||", "==", "!=", "<>", "<=", ">=", "=", "!",
"<", ">"}

# Reasonable entropy thresholds (tunable)
ALIAS_FANOUT_WARN = 10      # alias -> how many canonicals (warn)
ALIAS_FANOUT_FAIL = 20      # alias -> how many canonicals (fail)
CANON_FANIN_WARN  = 60      # canonical <- how many aliases (warn)
CANON_FANIN_FAIL  = 100     # canonical <- how many aliases (fail)

# ---------------------------------------------------------------
# Helpers (pure)
# ---------------------------------------------------------------

def _get_vocab_sections(vocab: Dict[str, Any]) -> Tuple[Dict[str, Any], Dict[str, Any]]:
    det = vocab.get("deterministic_aliases", {}) or {}
    nd  = vocab.get("non_deterministic_aliases", {}) or {}
    if not isinstance(det, dict) or not isinstance(nd, dict):
        # Normalize bad shapes to empty dicts (caller will log fail)
        det = det if isinstance(det, dict) else {}
        nd  = nd  if isinstance(nd, dict)  else {}
    return det, nd

def _alias_targets_map(vocab: Dict[str, Any]) -> Dict[str, List[str]]:
    """
    Combine deterministic & non-deterministic sections into
    alias -> [canonical, ...] lists. Keeps empty-string targets.
    """
    det, nd = _get_vocab_sections(vocab)
    out: Dict[str, List[str]] = {}
    for a, tgt in nd.items():
        out[str(a)] = [str(x) for x in (tgt or [])]
    for a, tgt in det.items():
        a = str(a)
        val = "" if tgt is None else str(tgt)
        out.setdefault(a, [])
        if val not in out[a]:
            out[a].append(val)
    return out

def _graph_canonical_types(graph: Dict[str, Any]) -> Dict[str, str]:
    """
```

```
    canonical -> entity_type
    """
    out = {}
    for c, n in graph.items():
        et = n.get("entity_type")
        if isinstance(et, str):
            out[c] = et
    return out

def _graph_aliases(graph: Dict[str, Any]) -> Set[str]:
    """
    All aliases provided in the graph metadata (lowercased).
    """
    out: Set[str] = set()
    for _, node in graph.items():
        md = node.get("metadata", {}) or {}
        for a in (md.get("aliases", []) or []):
            s = str(a).strip().lower()
            if s:
                out.add(s)
    return out

def _graph_canonicals(graph: Dict[str, Any]) -> Set[str]:
    """
    Canonical names of all nodes in the graph.
    """
    return set(graph.keys())

def _has_tuple(obj: Any) -> bool:
    if isinstance(obj, tuple):
        return True
    if isinstance(obj, dict):
        return any(_has_tuple(v) for v in obj.values())
    if isinstance(obj, list):
        return any(_has_tuple(v) for v in obj)
    return False

def _multiword(s: str) -> bool:
    return " " in s.strip()

def _prefix_to_longers(keys: Iterable[str]) -> Dict[str, List[str]]:
    keys = list(keys)
    multi = [k for k in keys if " " in k]
    out: Dict[str, List[str]] = defaultdict(list)
    for k in multi:
        p = k.split()[0]
        out[p].append(k)
    return out

# ------------------------------------------------------------
# V1. Coverage
# ------------------------------------------------------------

def   validate_vocab_coverage(graph:   Dict[str,   Any],   vocab:   Dict[str,   Any],   log:
```

```
List[str]) -> Tuple[bool, List[str]]:
    log.append("--- Running V1: Vocabulary Coverage ---")
    ok = True

    graph_aliases = _graph_aliases(graph)
    det, nd = _get_vocab_sections(vocab)
    vocab_keys = set(map(str, det.keys())) | set(map(str, nd.keys()))

    # Compare, but split by policy-dropped generics (warn) vs true misses (fail)
    missing_all = sorted(a for a in graph_aliases if a not in vocab_keys)
    policy_dropped = [a for a in missing_all if a in GENERIC_DENY]
    true_missing   = [a for a in missing_all if a not in GENERIC_DENY]

    # Helpful debug: show what each missing alias points to in the graph
    alias_idx = _build_graph_alias_index(graph)
    def _summarize_targets(alias: str) -> str:
        tgts = alias_idx.get(alias, [])
        # show up to 4 references to avoid log spam
        preview = ", ".join([f"({c}, {t})" for (c, t) in tgts[:4]])
        more = "" if len(tgts) <= 4 else f" (+{len(tgts)-4} more)"
        return f"[{preview}]{more}" if tgts else "[]"

    if true_missing:
        ok = False
             log.append(f"FAIL: {len(true_missing)} aliases from graph are missing in
vocabulary:")
        for m in true_missing[:50]:
            log.append(f"  - {m}  -> targets={_summarize_targets(m)}")
        if len(true_missing) > 50:
            log.append(f"  ... +{len(true_missing)-50} more")

    if policy_dropped:
              log.append("WARNING: Aliases missing due to policy (generic/ambiguous;
acceptable to omit):")
        for m in sorted(policy_dropped):
            log.append(f"  - {m}  -> targets={_summarize_targets(m)}")

    if not true_missing and not policy_dropped:
        log.append("PASS: All graph aliases are covered by the vocabulary.")

    return ok, log

# -------------------------------------------------------------
# V2. Determinism partition sanity
# -------------------------------------------------------------

def validate_vocab_partition_sanity(
    graph: Dict[str, Any], vocab: Dict[str, Any], log: List[str]
) -> Tuple[bool, List[str]]:
    log.append("--- Running V2: Determinism Partition Sanity ---")
    ok = True

    det, nd = _get_vocab_sections(vocab)
    alias_targets = _alias_targets_map(vocab)
```

```
    graph_canonicals = _graph_canonicals(graph)

    # 1) single-word, single-meaning aliases should live in deterministic
    offenders: List[str] = []
    for alias, targets in alias_targets.items():
        if not _multiword(alias) and len([t for t in targets if t is not None]) == 1:
            if alias not in det:
                offenders.append(alias)

    # 2) multi-word or multi-meaning aliases should be in non-deterministic,
     #     EXCEPT when it's a canonical identity (alias == canonical and det[alias] ==
alias).
    misbucket: List[str] = []
    exempted_identities: List[str] = []
    for alias, targets in alias_targets.items():
        if _multiword(alias) or len([t for t in targets if t is not None]) > 1:
            # identity exemption
            if alias in graph_canonicals and det.get(alias) == alias:
                exempted_identities.append(alias)
                continue
            if alias not in nd:
                misbucket.append(alias)

    if offenders:
        ok = False
          log.append("FAIL: Single-word, single-meaning aliases should be deterministic
but are not:")
        for a in sorted(offenders[:50]):
            log.append(f"  - '{a}'")
        if len(offenders) > 50:
            log.append(f"  ... +{len(offenders)-50} more")

    if misbucket:
        ok = False
                log.append("FAIL: Multi-word or multi-meaning aliases should be
non-deterministic but are not:")
        for a in sorted(misbucket[:50]):
            log.append(f"  - '{a}'")
        if len(misbucket) > 50:
            log.append(f"  ... +{len(misbucket)-50} more")

    # Debug: show what we exempted to confirm behavior
    if exempted_identities:
                                                                    log.append(f"DEBUG:V2
exempted_identity_canonicals={sorted(exempted_identities)[:20]}")

    if ok:
        log.append("PASS: Determinism partition looks sane.")
    return ok, log


# ---------------------------------------------------------------
# V3. Policy enforcement
# ---------------------------------------------------------------
```

```python
def _preposition_lexicon_from_graph(graph: Dict[str, Any]) -> Set[str]:
    """
    Returns a set of strings that are preposition tokens in the graph,
    including their canonical names and aliases (lowercased).
    """
    preps: Set[str] = set()
    for canon, node in graph.items():
        if node.get("entity_type") != "prepositions":
            continue
        preps.add(str(canon).strip().lower())
        md = node.get("metadata", {}) or {}
        for a in (md.get("aliases", []) or []):
            s = str(a).strip().lower()
            if s:
                preps.add(s)
    return preps


# ----------- V3 helpers (scoped to policy enforcement) -----------

def _preposition_lexicon_from_graph(graph: Dict[str, Any]) -> Set[str]:
    """
    Build the set of tokens that the graph declares as *prepositions*
    (both canonical names and their aliases), lowercased.
    """
    preps: Set[str] = set()
    for canon, node in graph.items():
        if node.get("entity_type") != "prepositions":
            continue
        preps.add(str(canon).strip().lower())
        md = node.get("metadata", {}) or {}
        for a in (md.get("aliases", []) or []):
            s = str(a).strip().lower()
            if s:
                preps.add(s)
    return preps


def _find_preposition_purity_violations(
    enforced_preps: Set[str],
    alias_targets: Dict[str, List[str]],
    canon_types: Dict[str, str],
) -> List[Tuple[str, List[str]]]:
    """
    For each enforced preposition token, all targets must be canonical prepositions.
    Returns list of (prep_alias, bad_targets).
    """
    prep_viol: List[Tuple[str, List[str]]] = []
    for prep in sorted(enforced_preps):
        tgts = alias_targets.get(prep, [])
        if not tgts:
            continue
        bad = [t for t in tgts if canon_types.get(t) != "prepositions"]
        if bad:
```

```python
            prep_viol.append((prep, bad))
    return prep_viol


def _find_domain_preference_violations(
    alias_targets: Dict[str, List[str]]
) -> List[str]:
    """
    If an alias has both the 'lose' and 'prefer' canonical present (per policy),
    return that alias so caller can fail with guidance.
    """
    viol: List[str] = []
    for alias, (lose, prefer) in DOMAIN_PREFER_SPATIAL.items():
        tgts = set(alias_targets.get(alias, []))
        if lose in tgts and prefer in tgts:
            viol.append(alias)
    return viol


def _find_of_policy_violations(
    alias_targets: Dict[str, List[str]]
) -> List[Tuple[str, str]]:
    """
    If an alias ends with ' of' and includes a canonical in OF_CANONICALS, ensure
    a '<canonical> of' form is present among targets. Otherwise it's a violation.
    Returns list of (alias, canonical).
    """
    of_viol: List[Tuple[str, str]] = []
    for alias, tgts in alias_targets.items():
        if not str(alias).lower().endswith(" of"):
            continue
        for t in tgts:
            if t in OF_CANONICALS and f"{t} of" not in tgts:
                of_viol.append((alias, t))
    return of_viol


def _find_prefix_protection_suggestions(
    alias_targets: Dict[str, List[str]],
    canon_types: Dict[str, str],
) -> List[Tuple[str, List[str], List[str]]]:
    """
    If a single-token alias maps to one or more table canonicals while
    longer multi-word aliases exist that start with this token, suggest
    prefix protection (warning).
    Returns list of (alias, longer_keys, table_targets).
    """
    suggestions: List[Tuple[str, List[str], List[str]]] = []
    prefix_map = _prefix_to_longers(alias_targets.keys())
    for a, longers in prefix_map.items():
        if " " in a:
            continue
        tgts = alias_targets.get(a, [])
        table_tgts = sorted([t for t in tgts if canon_types.get(t) == "table"])
```

```
        if table_tgts:
            suggestions.append((a, sorted(longers), table_tgts))
    return suggestions


def _find_reserved_token_misuse(
    alias_targets: Dict[str, List[str]],
    canon_types: Dict[str, str],
) -> List[str]:
    """
                    Reserved     tokens     should     only     map     to     canonical
prepositions/logicals/comparators/select_verbs.
    Returns list of offending aliases.
    """
    bad: List[str] = []
    for a in alias_targets.keys():
        if a not in RESERVED_TOKENS:
            continue
        tgts = alias_targets[a]
        if not all(canon_types.get(t) in {
            "prepositions", "logical_operators", "comparison_operators", "select_verbs"
        } for t in tgts):
            bad.append(a)
    return bad


def validate_vocab_policy_enforcement(
    graph: Dict[str, Any], vocab: Dict[str, Any], log: List[str]
) -> Tuple[bool, List[str]]:
    log.append("--- Running V3: Vocabulary Policy Enforcement ---")
    ok = True

    alias_targets = _alias_targets_map(vocab)
    canon_types   = _graph_canonical_types(graph)

    # Build actual preposition lexicon from the graph and intersect with policy set.
    graph_preps   = _preposition_lexicon_from_graph(graph)
    enforced_preps = set(PREP_BARE) & graph_preps
    skipped_preps  = set(PREP_BARE) - enforced_preps  # informative

    # 1) Preposition purity (only for preps that *exist as prepositions* in the graph)
        prep_viol  =  _find_preposition_purity_violations(enforced_preps,  alias_targets,
canon_types)

    # 2) Domain preference (spatial vs non-spatial)
    domain_viol = _find_domain_preference_violations(alias_targets)

    # 3) "of" surface policy
    of_viol = _find_of_policy_violations(alias_targets)

    # 4) Prefix protection (warning; binder/grammar stability)
    prefix_suggestions = _find_prefix_protection_suggestions(alias_targets, canon_types)

    # 5) Reserved token misuse (fail)
```

```
    reserved_viol = _find_reserved_token_misuse(alias_targets, canon_types)

    # ----- Reporting -----
    if prep_viol:
        ok = False
        log.append("FAIL: Preposition purity violations (alias  bad_targets):")
        for a, bad in prep_viol:
            log.append(f"  - '{a}'  {sorted(bad)}")

    if domain_viol:
        ok = False
        log.append("FAIL: Domain preference not enforced (preferred & losing targets both present):")
        for a in sorted(domain_viol):
            lose, prefer = DOMAIN_PREFER_SPATIAL[a]
            log.append(f"  - alias='{a}' kept both '{lose}' and '{prefer}'")

    if of_viol:
        ok = False
        log.append("FAIL: 'of' surface policy not enforced (need '<canon> of'):")
        for a, t in of_viol:
            log.append(f"  - alias='{a}'  contains '{t}' but missing '{t} of'")

    if prefix_suggestions:
        log.append("WARNING: Prefix protection suggested (single token maps to tables while longer keys exist):")
        for a, longers, tbls in sorted(prefix_suggestions):
            log.append(f"  - alias='{a}' prefixes {longers} but maps to tables {tbls}")

    if reserved_viol:
        ok = False
        log.append("FAIL: Reserved tokens used for non-reserved domains:")
        for a in sorted(reserved_viol):
            log.append(f"  - '{a}'")

    # Debug visibility for which PREP_BARE items were skipped (not present as prepositions in graph)
    if skipped_preps:
        log.append(f"DEBUG:V3 skipped_preps_no_graph_support={sorted(skipped_preps)}")

    if ok:
        log.append("PASS: Vocabulary respects policy constraints (with possible warnings).")
    return ok, log


# ------------------------------------------------------------------
# V4. Identity & canonical presence
# ------------------------------------------------------------------

def validate_vocab_identity_presence(graph: Dict[str, Any], vocab: Dict[str, Any], log: List[str]) -> Tuple[bool, List[str]]:
    log.append("--- Running V4: Identity & Canonical Presence ---")
    ok = True
```

```
    det, nd = _get_vocab_sections(vocab)
    # Build canonical -> entity_type map so we can skip meta/diagnostic keys
    canon_types = _graph_canonical_types(graph)

    missing_id: List[str] = []
    for c, et in sorted(canon_types.items()):
        if not _should_require_identity(c, et):
            continue
        # Must exist as a deterministic identity: c -> c ('' or None is also acceptable
for filler)
        val = det.get(c, None)
        if et == "filler_words":
            # For filler words, identity can be "" (skip) or None
            if val not in ("", None):
                missing_id.append(c)
        else:
            if val != c:
                missing_id.append(c)

    if missing_id:
        ok = False
        log.append("FAIL: Canonical tokens missing deterministic identity mapping
(canonical  canonical):")
        for m in missing_id[:100]:
            log.append(f"  - '{m}'")
        if len(missing_id) > 100:
            log.append(f"  ... +{len(missing_id)-100} more")
    else:
        log.append("PASS: All canonicals have identity mappings (meta nodes excluded).")
    return ok, log


# ---------------------------------------------------------------
# V5. Fan-out & entropy checks
# ---------------------------------------------------------------

def validate_vocab_entropy(
    vocab: Dict[str, Any],
    log: List[str],
    *,
    alias_fanout_warn: int = ALIAS_FANOUT_WARN,
    alias_fanout_fail: int = ALIAS_FANOUT_FAIL,
    canon_fanin_warn: int = CANON_FANIN_WARN,
    canon_fanin_fail: int = CANON_FANIN_FAIL,
) -> Tuple[bool, List[str]]:
    log.append("--- Running V5: Fan-out & Entropy ---")
    ok = True

    alias_targets = _alias_targets_map(vocab)
    fanout = {a: len([t for t in tgts]) for a, tgts in alias_targets.items()}  # include
"" as a target
    canon_counter: Counter = Counter()
    for tgts in alias_targets.values():
```

```
        for t in tgts:
            canon_counter[t] += 1

    # Alias fan-out extremes
    bad_aliases_fail = [a for a, n in fanout.items() if n >= alias_fanout_fail]
      bad_aliases_warn = [a for a, n in fanout.items() if alias_fanout_warn <= n <
alias_fanout_fail]

    # Canonical fan-in extremes
    bad_canon_fail = [c for c, n in canon_counter.items() if n >= canon_fanin_fail]
     bad_canon_warn = [c for c, n in canon_counter.items() if canon_fanin_warn <= n <
canon_fanin_fail]

    if bad_aliases_fail or bad_canon_fail:
        ok = False

    if bad_aliases_fail:
           log.append(f"FAIL: Aliases with extreme fan-out ( {alias_fanout_fail}
targets):")
        for a in sorted(bad_aliases_fail[:50]):
            log.append(f"  - '{a}'  {fanout[a]} targets")
        if len(bad_aliases_fail) > 50:
            log.append(f"  ... +{len(bad_aliases_fail)-50} more")

    if bad_aliases_warn:
        log.append(f"WARNING: Aliases with high fan-out ( {alias_fanout_warn}):")
        for a in sorted(bad_aliases_warn[:50]):
            log.append(f"  - '{a}'  {fanout[a]} targets")
        if len(bad_aliases_warn) > 50:
            log.append(f"  ... +{len(bad_aliases_warn)-50} more")

    if bad_canon_fail:
           log.append(f"FAIL: Canonicals with extreme fan-in ( {canon_fanin_fail}
aliases):")
        for c in sorted(bad_canon_fail[:50]):
            log.append(f"  - '{c}'  {canon_counter[c]} aliases")
        if len(bad_canon_fail) > 50:
            log.append(f"  ... +{len(bad_canon_fail)-50} more")

    if bad_canon_warn:
        log.append(f"WARNING: Canonicals with high fan-in ( {canon_fanin_warn}):")
        for c in sorted(bad_canon_warn[:50]):
            log.append(f"  - '{c}'  {canon_counter[c]} aliases")
        if len(bad_canon_warn) > 50:
            log.append(f"  ... +{len(bad_canon_warn)-50} more")

    if ok:
          log.append("PASS: Entropy levels are within expected bounds (with possible
warnings).")
    return ok, log

# --------------------------------------------------------------
# V6. Serialization safety
# --------------------------------------------------------------
```

```
def validate_vocab_serialization_safety(vocab: Dict[str, Any], log: List[str]) -> Tuple[bool, List[str]]:
    log.append("--- Running V6: Serialization Safety ---")
    ok = True

    # 1) No tuples anywhere (these crash safe_load with python/tuple tag)
    if _has_tuple(vocab):
        log.append("FAIL: Vocabulary contains Python tuples (YAML-unsafe).")
        ok = False

    # 2) Section shapes: det values must be str/None, nd values must be list[str]
    det, nd = _get_vocab_sections(vocab)

    bad_det: List[Tuple[str, Any]] = []
    for a, v in det.items():
        if v is None or isinstance(v, str):
            continue
        bad_det.append((str(a), v))

    bad_nd: List[Tuple[str, Any]] = []
    for a, v in nd.items():
        if not isinstance(v, list) or not all(isinstance(x, (str, type(None))) for x in v):
            bad_nd.append((str(a), v))

    if bad_det:
        ok = False
        log.append("FAIL: Deterministic section must map alias -> string (or None/''):")
        for a, v in bad_det[:50]:
            log.append(f"  - alias='{a}' value_type={type(v).__name__}")
        if len(bad_det) > 50:
            log.append(f"  ... +{len(bad_det)-50} more")

    if bad_nd:
        ok = False
        log.append("FAIL: Non-deterministic section must map alias -> list[str | None]:")
        for a, v in bad_nd[:50]:
            log.append(f"  - alias='{a}' value_type={type(v).__name__}")
        if len(bad_nd) > 50:
            log.append(f"  ... +{len(bad_nd)-50} more")

    # 3) YAML round-trip
    if _HAVE_YAML:
        try:
            txt = yaml.safe_dump(vocab, sort_keys=True)
            _ = yaml.safe_load(txt)
        except Exception as e:
            ok = False
            log.append(f"FAIL: YAML safe round-trip failed: {e}")

    if ok:
        log.append("PASS: Vocabulary is YAML-serializable and well-typed.")
```

```
    return ok, log

# ---------------------------------------------------------------
# Convenience: run the whole suite
# ---------------------------------------------------------------

def validate_vocabulary_all(
    graph: Dict[str, Any],
    vocab: Dict[str, Any],
    log: List[str],
    *,
    alias_fanout_warn: int = ALIAS_FANOUT_WARN,
    alias_fanout_fail: int = ALIAS_FANOUT_FAIL,
    canon_fanin_warn: int = CANON_FANIN_WARN,
    canon_fanin_fail: int = CANON_FANIN_FAIL,
) -> Tuple[bool, List[str]]:
    """
    Runs V1V6 against (graph, vocab).
    Returns (ok, log). Individual warnings/errors are appended to `log`.
    """
    results: List[bool] = []

     ok1, log = validate_vocab_coverage(graph, vocab, log);
results.append(ok1)
     ok2, log = validate_vocab_partition_sanity(graph, vocab, log);
results.append(ok2)
     ok3, log = validate_vocab_policy_enforcement(graph, vocab, log);
results.append(ok3)
     ok4, log = validate_vocab_identity_presence(graph, vocab, log);
results.append(ok4)
    ok5, log = validate_vocab_entropy(
        vocab, log,
        alias_fanout_warn=alias_fanout_warn, alias_fanout_fail=alias_fanout_fail,
        canon_fanin_warn=canon_fanin_warn, canon_fanin_fail=canon_fanin_fail
    ); results.append(ok5)
     ok6, log = validate_vocab_serialization_safety(vocab, log);
results.append(ok6)

    overall_ok = all(results)
    if overall_ok:
        log.append("PASS: Vocabulary validations (V1V6) passed.")
    else:
        log.append("FAIL: One or more vocabulary validations failed.")
    return overall_ok, log
```

src/n2s_runtime/binder_aware_normalizer.md

How the binder-aware normalizer works (step-by-step)

0) Inputs and roles

* **Vocabulary (a.k.a. normalization map):** alias canonical(s), split into deterministic and non-deterministic. Also contains the canonical identities, so canonical text can pass through unchanged.
* **Binder index (artifact):** a lightweight, structure-aware index distilled from the relationship graph, used to:

  * Recognize **canonical** tables / columns / functions.
  * Map **column table(s)** (ownership/inference).
  * Optionally carry function argument hints (e.g., whether OF is typically used).
* **Grammar:** expects canonical text in one shape:
  `select <column_list> (from|of) <table>` with `column_list := selectable (, selectable)* (,? and selectable)?` and `selectable := column | function_call`, `function_call := function (of column_list)?`.

1) Tokenize and alias segmentation

* Tokenize the user text (`&&`, `||`, operators, punctuation, words).
* Use a **leftmost BFS segmentation** to cover the token stream with vocabulary keys.
* Replace matched spans with canonical outputs, carrying forward multiple options for ambiguous aliases.
* Output: **canonical token sequences** (ordered, still structureless).

Why BFS? It finds a cover of the leftmost unmapped span first, keeps branching only when an alias span has multiple canonical options, and caps nodes/results to keep it tractable.

2) Binder-guided structuring (make it grammar-shaped)

For each canonical sequence:

* **Collect** which tokens are tables / columns / functions (using the binder).
* **Infer or select a table**:

  * If a table is explicitly present in the tokens, use it.
  * Else, infer by majority vote of owner tables for the found columns. (If multiple ties, produce candidates for each.)
* **Build a canonical `column_list`:**

  * Use the found columns as raw `selectable`s.
  * For each function found:

    * If there are columns, emit `function of <column_list>` (your grammars preferred shape).
    * If no columns were seen, emit bare `function` (allowed by grammar).
  * You can include both columns and function_call(s) in the list; the grammar allows it.
  * Join items with commas (`,`), optionally end with `and ` if you want variety; commas

alone are sufficient for parsing.
* **Insert missing connectors** and **normalize order**:

  * Make sure theres a leading `select`.
  * Choose `(from|of)` based on your policy (default to `from` for tables).
  * Render a single canonical phrase:
    `select <item1 , item2 , ...> from <table>`
* **Score/rank** candidates lightly:

  * Prefer phrases where **all columns belong to the chosen table**.
  * Prefer phrases that **insert fewer defaults** (e.g., didnt have to guess the table).
* Output: one or more **grammar-compatible canonical phrases**.

3) Reversibility / denormalization

* Build a **reverse alias index**: canonical aliases (including canonical identity).
* When denormalizing, pick aliases but avoid **connector duplication** (e.g., if you choose `distinct of` for a function and the next token is `of`, consume the next connector).
* This keeps **canonical messy canonical** reversible enough for testing.

4) Error handling, logging, and caps

* Return **multiple** candidates (ranked) when ambiguous.
* If the binder cant form structure, **fall back** to the raw canonical token sequence (so you dont regress vs. the old normalizer).
* Use a **FlightRecorder** to log segmentation steps, binder inferences, pruning, and reasons for empty outputs.

What this **does not** do (by design)

* It doesnt re-interpret unknown tokens; unknowns are ignored or logged.
* It doesnt try to outsmart types/labels at this stage (thats validated upstream).
* It doesnt change the **meaning** of function arity (just inserts `of` when there are column arguments).

Notes and extensibility

* If you later add **binder templates** with richer slots (e.g., per-function arity, typed column constraints), you can:

   * Expand `_build_select_list_items` to construct argument lists per template (and enforce types).
   * Add a ranking function that penalizes type/label violations instead of dropping candidates.
* If you adopt **of vs from** policies per function/table, thread that into `function_needs_of` and `(prefer_from)` or even per-candidate signals.
* The **fallback** (return the flat canonical if binding fails) preserves current success rates while you iterate on binder constraints.

This gives you a normalizer that:

* Uses the vocabulary to get into the canonical token space,
* Uses the binder to recover structure and produce grammar-shaped phrases,
* And stays reversible enough to support your de-normalize normalize integration tests.

src/n2s_runtime/canonical_core.py

src/n2s_runtime/canonical_utils.py

src/n2s_runtime/normalizer.py

```
# src/n2s_runtime/normalizer.py

from __future__ import annotations
import re
from dataclasses import dataclass, field
from typing import Callable, Dict, Iterable, List, Sequence, Tuple, Optional, Set

# ----------------------------
# Types
# ----------------------------
Pair = Tuple[str, bool]
Phrase = List[Pair]
NDMap = Dict[str, List[str]]

@dataclass(frozen=True)
class BinderIndex:
    """Minimal, structure-aware index derived from the relationship graph/binder artifact."""
    tables: Set[str]
    columns: Set[str]
    functions: Set[str]
    column_to_tables: Dict[str, List[str]]  # e.g., {'user_id': ['users'], 'sale_date': ['sales']}
    function_needs_of: Dict[str, bool] = field(default_factory=dict)  # optional hints

# ----------------------------
# Flight recorder
# ----------------------------
@dataclass
class FlightRecorder:
    events: List[Tuple[str, Dict[str, object]]] = field(default_factory=list)
    def log(self, evt: str, **data: object) -> None: self.events.append((evt, data))
    def warn(self, evt: str, **data: object) -> None: self.events.append((f"WARNING:{evt}", data))
    def fail(self, evt: str, **data: object) -> None: self.events.append((f"FAIL:{evt}", data))
    def dump(self, print_fn: Callable[[str], None] = print) -> None:
        for e, d in self.events: print_fn(f"{e}: {d}")

# ----------------------------
# Tokenization
# ----------------------------
TOKENIZER_RE = re.compile(r"\|\||&&|<=|>=|!=|==|<>|[A-Za-z0-9_']+|[^\sA-Za-z0-9_]")

def tokenize(s: str) -> List[str]:
    return TOKENIZER_RE.findall(s)

def squash_spaces(s: str) -> str:
    return " ".join(s.split())

# ----------------------------
```

```
# Vocabulary shaping (alias  canonicals) + identities
# ----------------------------
SENTINEL_BLACKLIST = {"", "skip", "_skip"}

def _coerce_listy(v: object) -> List[str]:
    if isinstance(v, list): return [("" if o is None else str(o)) for o in v]
    return ["" if v is None else str(v)]

def _collect_canonicals(det: Dict[str, object], nd: Dict[str, object]) -> List[str]:
    vals: List[str] = []
    for v in det.values():
        if isinstance(v, str) and v not in SENTINEL_BLACKLIST:
            vals.append(v)
    for v in nd.values():
        if isinstance(v, list):
            for o in v:
                if isinstance(o, str) and o not in SENTINEL_BLACKLIST:
                    vals.append(o)
        elif isinstance(v, str) and v not in SENTINEL_BLACKLIST:
            vals.append(v)
    return vals

def build_nd_map(vocabulary: Dict[str, Dict[str, object]]) -> NDMap:
    """Build a non-deterministic aliascanonicals map with canonical identities."""
    det = vocabulary.get("deterministic_aliases", {}) or {}
    nd  = vocabulary.get("non_deterministic_aliases", {}) or {}

    out: NDMap = {}
    for k, v in nd.items():
        out[str(k)] = _coerce_listy(v)

    for k, v in det.items():
        canon = "" if (v is None or v == "skip") else str(v)
        out.setdefault(str(k), [])
        if canon not in out[str(k)]:
            out[str(k)].append(canon)

    # identity for canonical outputs so canonical tokens are always mappable
    for c in _collect_canonicals(det, nd):
        out.setdefault(c, [])
        if c not in out[c]:
            out[c].append(c)

    return out

def build_reverse_alias_map(vocabulary: Dict[str, Dict[str, object]]) -> Dict[str,
List[str]]:
    """canonical  list of aliases (including identity)"""
    det = vocabulary.get("deterministic_aliases", {}) or {}
    nd  = vocabulary.get("non_deterministic_aliases", {}) or {}

    rev: Dict[str, List[str]] = {}
    # deterministic
    for alias, canonical in det.items():
```

```
        c = "" if (canonical is None or canonical == "skip") else str(canonical)
        if c == "":  # fillers dont get reverse
            continue
        rev.setdefault(c, [])
        if alias not in rev[c]:
            rev[c].append(alias)
        # identity
        if c not in rev[c]:
            rev[c].append(c)

    # non-deterministic
    for alias, clist in nd.items():
        for c in _coerce_listy(clist):
            if c == "":  # filler-like
                continue
            rev.setdefault(c, [])
            if alias not in rev[c]:
                rev[c].append(alias)
            # identity
            if c not in rev[c]:
                rev[c].append(c)

    # de-dup in place
    for c, arr in rev.items():
        seen, out = set(), []
        for a in arr:
            if a not in seen:
                seen.add(a); out.append(a)
        rev[c] = out
    return rev

# ----------------------------
# Punctuation passthrough
# ----------------------------
def   punctuation_passthrough(tokens:   Sequence[str],   passthrough:   Iterable[str])   ->
Phrase:
    pt = set(passthrough)
    return [(t, t in pt) for t in tokens]

# ----------------------------
# BFS segmentation over aliases
# ----------------------------
def _max_key_len_words(nd: NDMap) -> int:
    m = 1
    for k in nd.keys():
        L = max(1, len(k.split(" ")))
        if L > m: m = L
    return m

def _serialize(ph: Phrase) -> Tuple[Tuple[str, bool], ...]:
    return tuple(ph)

def bfs_resolve_leftmost_spans(
    initial: Phrase,
```

```
    ndict: NDMap,
    *,
    joiner: str = " ",
    cap_nodes: int = 200,
    cap_results: int = 200,
    warn_every: int = 50,
    fr: Optional[FlightRecorder] = None,
) -> List[str]:
    from collections import deque
    q = deque([initial])
    seen = {_serialize(initial)}
    finals: List[str] = []
    max_len = _max_key_len_words(ndict)
    node_expanded = 0

    while q:
        phrase = q.popleft()

        # find leftmost unmapped
        try:
            i = next(idx for idx, (_, m) in enumerate(phrase) if not m)
        except StopIteration:
            s = joiner.join(t for t, _ in phrase)
            finals.append(s)
            if fr and len(finals) % warn_every == 0:
                fr.warn("final_count", count=len(finals))
            if len(finals) > cap_results:
                if fr: fr.fail("final_cap_exceeded", cap=cap_results, count=len(finals))
                break
            continue

        # contiguous unmapped run
        r = i
        while r < len(phrase) and not phrase[r][1]: r += 1
        run_len = r - i
        tried = False

        for span_len in range(1, min(max_len, run_len) + 1):
            span_text = joiner.join(phrase[k][0] for k in range(i, i + span_len))
            options = ndict.get(span_text)
            if not options: continue
            tried = True
            for opt in options:
                new_phrase = phrase[:i] + [(opt, True)] + phrase[i + span_len:]
                key = _serialize(new_phrase)
                if key in seen: continue
                seen.add(key)
                q.append(new_phrase)
                node_expanded += 1
                if fr and node_expanded % warn_every == 0:
                    fr.warn("node_count", count=node_expanded)
                if node_expanded > cap_nodes:
                                 if  fr:  fr.fail("node_cap_exceeded",  cap=cap_nodes,
count=node_expanded)
```

```
                    return finals

        if not tried:
            if fr: fr.log("prune", leftmost=phrase[i][0], run_len=run_len)

    return finals

# -----------------------------
# Binder-aware structuring
# -----------------------------
CONNECTORS = {"of", "from", "and"}
PUNCT = {","}

def _is_table(tok: str, binder: BinderIndex) -> bool:
    return tok in binder.tables

def _is_column(tok: str, binder: BinderIndex) -> bool:
    return tok in binder.columns

def _is_function(tok: str, binder: BinderIndex) -> bool:
    return tok in binder.functions

def _infer_tables_for_columns(cols: List[str], binder: BinderIndex) -> List[str]:
        """Vote by column ownership. Returns candidate tables (ranked by votes,
deterministic)."""
    from collections import Counter
    tally = Counter()
    for c in cols:
        for t in binder.column_to_tables.get(c, []):
            tally[t] += 1
    if not tally:
        return []
    max_votes = max(tally.values())
    # stable order by (-votes, name)
     cands = sorted([t for t, v in tally.items() if v == max_votes], key=lambda x: (
-tally[x], x))
    return cands

def _build_select_list_items(columns: List[str], functions: List[str], binder:
BinderIndex) -> List[str]:
    """
    Returns a list of selectables (each selectable is a canonical snippet).
    Strategy:
      - include raw columns as selectables
      - for each function:
          * if there are columns -> 'func of <joined_columns>'
          * else -> 'func'
    """
    items: List[str] = []
    # raw columns as selectables
    for c in columns:
        items.append(c)
    # function_call selectables
    if functions:
```

```
        if columns:
            joined = " , ".join(columns)
            for f in functions:
                needs_of = binder.function_needs_of.get(f, True)  # default: use 'of' when args exist
                if needs_of:
                    items.append(f"{f} of {joined}")
                else:
                    # even if hint says no 'of', grammar allows optional; but we can skip
                    items.append(f"{f} {joined}")
        else:
            # bare functions allowed by grammar
            for f in functions:
                items.append(f)
    return items

def _canonicalize_to_grammar_shape(
    canonical_seq: str,
    binder: BinderIndex,
    *,
    prefer_from: bool = True,
    fr: Optional[FlightRecorder] = None,
) -> List[str]:
    """
    Take a flat canonical sequence and build one or more 'select ... from ...' phrases that the grammar accepts.
    If impossible, return [].
    """
    toks = canonical_seq.split()
    tables = [t for t in toks if _is_table(t, binder)]
    columns = [t for t in toks if _is_column(t, binder)]
    functions = [t for t in toks if _is_function(t, binder)]

    if fr:
        fr.log("binder_scan", tables=tables, columns=columns, functions=functions)

    # choose table(s)
    table_candidates: List[str] = []
    if tables:
        # explicit table(s) mentioned; keep unique + stable
        seen = set()
        for t in tables:
            if t not in seen:
                seen.add(t); table_candidates.append(t)
    elif columns:
        table_candidates = _infer_tables_for_columns(columns, binder)
    else:
        # no table and no columns  cannot form a valid select  from  phrase
        return []

    # build selectables
    items = _build_select_list_items(columns, functions, binder)
    if not items and not functions:
```

```python
        # absolutely nothing to select
        return []

    # join with commas (grammar accepts comma-only lists)
    select_list = " , ".join(items)

    # assemble phrases for each candidate table
    out: List[str] = []
    for tbl in table_candidates:
        connector = "from" if prefer_from else "of"
        phrase = f"select {select_list} {connector} {tbl}"
        out.append(phrase)

    # simple ranking: prefer candidates where all columns belong to the chosen table
    def _score(p: str) -> Tuple[int, int, str]:
        # higher is better
        parts = p.split()
        # table is last token
        tab = parts[-1] if parts else ""
        ok_cols = sum(1 for c in columns if tab in binder.column_to_tables.get(c, []))
        # fewer guesses (we guessed when there was no explicit table)
        guessed = 1 if not tables else 0
        return (ok_cols, -guessed, tab)

    out.sort(key=_score, reverse=True)
    return out

# -------------------------------
# Public API: normalization (with optional binder)
# -------------------------------
def normalize_text(
    vocabulary: Dict[str, Dict[str, object]],
    text: str,
    *,
    tokenizer: Callable[[str], List[str]] = tokenize,
    joiner: str = " ",
    case_insensitive: bool = False,
    punctuation_as_mapped: Iterable[str] = (",",),
    cap_nodes: int = 200,
    cap_results: int = 200,
    warn_every: int = 50,
    fr: Optional[FlightRecorder] = None,
    binder: Optional[BinderIndex] = None,
    prefer_from: bool = True,
) -> List[str]:
    """
    Returns canonical, grammar-shaped candidates when a binder is provided.
    Falls back to legacy "flat canonical strings" when binder is None or cannot bind.
    """
    nd = build_nd_map(vocabulary)
    s = text.casefold() if case_insensitive else text
    toks = tokenizer(s)
    if fr: fr.log("tokens", tokens=toks)
    init = punctuation_passthrough(toks, punctuation_as_mapped)
```

```
    if fr: fr.log("seed", phrase=init)

    finals_raw = bfs_resolve_leftmost_spans(
        initial=init,
        ndict=nd,
        joiner=joiner,
        cap_nodes=cap_nodes,
        cap_results=cap_results,
        warn_every=warn_every,
        fr=fr,
    )

    # de-dup clean strings
    seen, flat_canonicals = set(), []
    for f in finals_raw:
        clean = squash_spaces(f)
        if clean not in seen:
            seen.add(clean)
            flat_canonicals.append(clean)

    if binder is None:
        if fr: fr.log("finals_flat_only", count=len(flat_canonicals))
        return flat_canonicals

    # Try to bind each flat canonical into a grammar-shaped phrase
    bound: List[str] = []
    for c in flat_canonicals:
        candidates = _canonicalize_to_grammar_shape(c, binder, prefer_from=prefer_from, fr=fr)
        if candidates:
            bound.extend(candidates)
        else:
            # fallback: keep flat canonical (maintains old behavior and avoids regressions)
            bound.append(c)

    # Unique, stable
    uniq: List[str] = []
    seen2 = set()
    for b in bound:
        if b not in seen2:
            seen2.add(b); uniq.append(b)

    if fr: fr.log("finals_bound", count=len(uniq))
    return uniq

# ----------------------------
# Denormalization helpers (for validators/tests)
# ----------------------------
def _bucket_aliases_by_trailing_connector(aliases: List[str]) -> Tuple[List[str], Dict[str, List[str]]]:
    endswith: Dict[str, List[str]] = {c: [] for c in CONNECTORS}
    plain: List[str] = []
    for a in aliases:
```

```
        s = a.strip()
        low = s.lower()
        matched = False
        for c in CONNECTORS:
            if low.endswith(" " + c):
                endswith[c].append(s); matched = True; break
        if not matched:
            plain.append(s)
    return plain, endswith

def denormalize_phrase(
    canonical_phrase: str,
    reverse_alias_map: Dict[str, List[str]],
    *,
    connectors: Set[str] = CONNECTORS,
) -> str:
    """
     Replace canonical tokens with aliases while avoiding connector duplication (e.g.,
"... of" + "of").
    """
    LOCK = {",", "COMMA"} | connectors
    toks = canonical_phrase.split()
    out: List[str] = []
    i = 0
    while i < len(toks):
        t = toks[i]
        if t in LOCK or t == ",":
            out.append("," if t in {",", "COMMA"} else t)
            i += 1
            continue

        choices = reverse_alias_map.get(t, None)
        if not choices:
            out.append(t); i += 1; continue

        nxt = toks[i + 1].lower() if i + 1 < len(toks) else None
        plain, ends = _bucket_aliases_by_trailing_connector(choices)

        if nxt in connectors:
            if plain:
                out.append(plain[0]); i += 1
            elif nxt in ends and ends[nxt]:
                out.append(ends[nxt][0]); i += 2  # consume the next connector
            else:
                pool = plain or [a for arr in ends.values() for a in arr]
                out.append(pool[0]); i += 1
        else:
                out.append(plain[0] if plain else [a for arr in ends.values() for a in
arr][0])
            i += 1

    return " ".join(out)

# ----------------------------
```

```
# Debug probes (optional)
# ------------------------------
def inspect_leftmost(vocabulary: Dict[str, Dict[str, object]], text: str,
                       punctuation_as_mapped: Iterable[str] = (",",), joiner: str = " ")
-> None:
    nd = build_nd_map(vocabulary)
    toks = tokenize(text)
    seed = punctuation_passthrough(toks, punctuation_as_mapped)
    i = None
    for idx, (_, m) in enumerate(seed):
        if not m: i = idx; break
    max_len = max(1, max(len(k.split(" ")) for k in nd.keys()))
    print(f"\n[inspect] '{text}'")
    print(f"  tokens: {toks}")
    print(f"  seed  : {seed}")
    if i is None:
        print("  fully mapped at seed"); return
    # local probe of available keys at leftmost gap
    spans = []
    run_len = len(toks) - i
    for span_len in range(1, min(max_len, run_len) + 1):
        span = joiner.join(toks[i:i+span_len])
        if span in nd:
            spans.append((span, len(nd[span])))
    print(f"  leftmost unmapped idx={i} token='{toks[i]}'")
    print(f"  nd keys here: {spans if spans else 'NONE  <-- coverage gap at leftmost'}")
```

tests/conftest.py

```python
import pytest
import sqlite3
import os
from src.scripts.generate_db import create_db, DB_NAME

@pytest.fixture(scope="session")
def db_connection():
    """
    Creates the test database once per test session and yields a connection.
    """
    if not os.path.exists(DB_NAME):
        create_db()

    conn = sqlite3.connect(DB_NAME)
    yield conn
    conn.close()

@pytest.fixture
def clean_db_cursor(db_connection):
    """
    Provides a clean cursor for each test, ensuring no state leakage.
    """
    cursor = db_connection.cursor()
    # You might want to wrap this in a transaction if your tests modify data
    yield cursor
    cursor.close()
```