

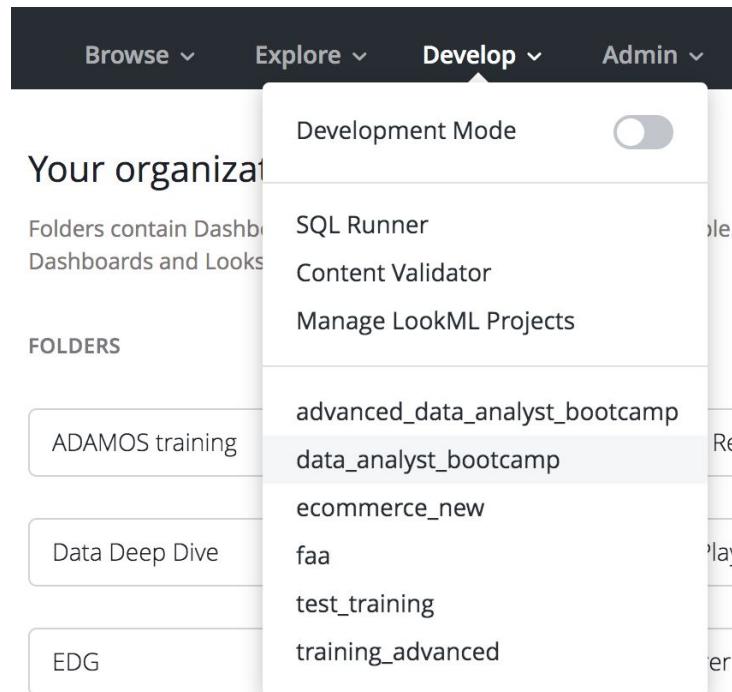
Before we get started, have you...

...reviewed prerequisites / today's agenda?

...ensured laptop is connected to live power strip?

...silenced your phone?

...logged into teach.corp.looker.com, and the “Develop” menu shows?



Looker Developer Workshop

**BUSINESS INTELLIGENCE
LEVEL 1 - TECHNOLOGY DELIVERY**

Good morning, I'm Bernard



Partner Enablement Lead, EMEA
Looker Professional Services
Dublin, Ireland

Please Introduce Yourselves!

Your name

Your company

Experience level with Looker

What you're hoping to take away from today's workshop

Please keep your introduction to one minute or less

BI Level 1 - Technology Delivery Agenda

Introduction to Looker

Dimensions & Measures

Working with Model Files

Caching & Datagroups

Derived Tables

Administration

Day 1

Day 2

BI Level 1 - Technology Delivery Agenda

Customization with Liquid

Parameters & Templated Filters

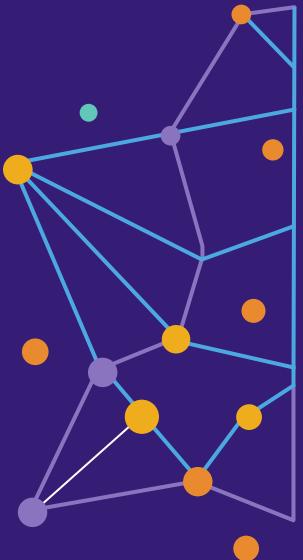
Extends

Debugging

Best Practices

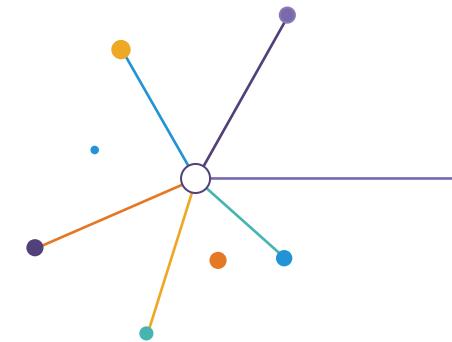


Day 3



Introduction to Looker

Let's Begin by **Defining Terms**



Term: LookML

Informs Looker using abstracted SQL

Creates modeling layer between database and user

```
dimension_group: created {
  type: time
  timeframes: [raw, time, date, week, month, quarter, year]
  sql: ${TABLE}.created_at ;;
}

dimension: status {
  type: string
  sql: ${TABLE}.status ;;
}

dimension: traffic_source {
  type: string
  sql: ${TABLE}.traffic_source ;;
}

dimension: user_id {
  type: number
  sql: ${TABLE}.user_id ;;
}

measure: count {
  type: count
  drill_fields: [id, users.id, users.first_name, users.last_name]
}

measure: count_html {
  type: count
  drill_fields: [id, users.id, users.first_name, users.last_name]
  html:
    {% if value > 400 %}
      <b><p style="color:white; background-color:darkgreen; margin: 0"> {{value}} </p>
    {% elseif value > 380 %}
      <b><p style="color:white; background-color:goldenrod; margin: 0"> {{value}} </p>
    {% else %}
      <b><p style="color:white; background-color:darkred; margin: 0"> {{value}} </p>
    {% endif %}
  ;
}
```

LookML Defines...

Join logic between tables (*Views*)

Custom tables (*Derived Tables*)
defined in Looker

Fields taken directly from the
database

Custom fields defined in Looker

```
dimension_group: created {
  type: time
  timeframes: [raw, time, date, week, month, quarter, year]
  sql: ${TABLE}.created_at ;;
}

dimension: status {
  type: string
  sql: ${TABLE}.status ;;
}

dimension: traffic_source {
  type: string
  sql: ${TABLE}.traffic_source ;;
}

dimension: user_id {
  type: number
  sql: ${TABLE}.user_id ;;
}

measure: count {
  type: count
  drill_fields: [id, users.id, users.first_name, users.last_name]
}

measure: count_html {
  type: count
  drill_fields: [id, users.id, users.first_name, users.last_name]
  html:
  {% if value > 400 %}
    <b><p style="color:white; background-color:darkgreen; margin: 0"> {{value}} </p>
  {% elseif value > 380 %}
    <b><p style="color:white; background-color:goldenrod; margin: 0"> {{value}} </p>
  {% else %}
    <b><p style="color:white; background-color:darkred; margin: 0"> {{value}} </p>
  {% endif %}
  ;
}
```

Term: LookML Project

Highest-level Looker object

Almost independent Looker instance

Usually used for completely different data sources

View files cannot be shared between different projects (without using a project import)

LookML Projects		Configure New Model	New LookML Project
Project	Models		
ecommerce	ecommerce_data	Configure	
faa	faa	Configure	
	looker_training	Configure	
trainembed_thelook	looker	Configure	
	trainembed_thelook	Configure	
video_thelook	video_thelook	Configure	

Term: Model

Contains data connection information and Explore definitions

Can be used to

- restrict user access to certain Explores
- separate and organize Explores by business area

```
connection: "events_ecommerce"
include: "*.*.view"

explore: users {
    label: "Users Backgrounds"
}

explore: inventory_items {
    group_label: "Inventory Analysis"
    join: products {
        type: left_outer
        sql_on: ${inventory_items.product_id} = ${products.id} ;;
        relationship: many_to_one
    }

    join: inventory_facts {
        view_label: "Inventory Items"
        type: left_outer
        sql_on: ${inventory_items.product_id} = ${inventory_facts.product_id} ;;
        relationship: many_to_one
    }

    join: distribution_centers {
        type: left_outer
        sql_on: ${products.distribution_center_id} = ${distribution_centers.id} ;;
        relationship: many_to_one
    }

    join: order_items {
        type: left_outer
        sql_on: ${inventory_items.id} = ${order_items.inventory_item_id} ;;
        relationship: one_to_many
    }
}
```

Term: Explore

Use Explores to begin analysis

Clearly organize Explores around business themes to minimize confusion for end users

- Users
- Orders
- Inventory

The screenshot shows the Looker interface with the 'Explore' tab selected in the top navigation bar. A search bar labeled 'Find an Explore' is present. Below it, the interface is organized into sections by business theme:

- Ecommerce Data**: Events, Inventory Items, Order Items, Users
- Faa**: Airports, Flights
- Looker Training**: Ecomm Looker Training Set, Fruit Basket, HR Looker Training Set
- Sales**: Inventory, Orders and Revenue

Term: View

View files can correspond to

- Tables in Database (*Standard View*)
- Looker-defined virtual tables (*Derived Tables*)
- Looker-defined tables physically written (*materialized View*) to database (*Persistent Derived Table or PDT*)

```
connection: "events_ecommerce"
include: "*.*.view"

explore: users {
  label: "Users Backgrounds"
}

explore: inventory_items {
  group_label: "Inventory Analysis"
  join: products {
    type: left_outer
    sql_on: ${inventory_items.product_id} = ${products.id} ;;
    relationship: many_to_one
  }

  join: inventory_facts {
    view_label: "Inventory Items"
    type: left_outer
    sql_on: ${inventory_items.product_id} = ${inventory_facts.product_id} ;;
    relationship: many_to_one
  }

  join: distribution_centers {
    type: left_outer
    sql_on: ${products.distribution_center_id} = ${distribution_centers.id} ;;
    relationship: many_to_one
  }

  join: order_items {
    type: left_outer
    sql_on: ${inventory_items.id} = ${order_items.inventory_item_id} ;;
    relationship: one_to_many
  }
}
```

Term: View

One or more view files joined together create an Explore

Views become headers in Explores

Dimensions and Measures are defined within View files

```
connection: "events_ecommerce"
include: "*.view"

explore: users {
  label: "Users Backgrounds"
}

explore: inventory_items {
  group_label: "Inventory Analysis"
  join: products {
    type: left_outer
    sql_on: ${inventory_items.product_id} = ${products.id} ;;
    relationship: many_to_one
  }

  join: inventory_facts {
    view_label: "Inventory Items"
    type: left_outer
    sql_on: ${inventory_items.product_id} = ${inventory_facts.product_id} ;;
    relationship: many_to_one
  }

  join: distribution_centers {
    type: left_outer
    sql_on: ${products.distribution_center_id} = ${distribution_centers.id} ;;
    relationship: many_to_one
  }

  join: order_items {
    type: left_outer
    sql_on: ${inventory_items.id} = ${order_items.inventory_item_id} ;;
    relationship: one_to_many
  }
}
```

Term: Dimension

Always in the GROUP BY part of any query

Automatically created for all fields within a table

▼ DATA

RESULTS

SQL

SELECT

```
users.city AS "users.city",
COUNT(*) AS "users.count"
FROM public.users AS users
```

GROUP BY 1

ORDER BY 2 DESC

LIMIT 500

Term: Measure

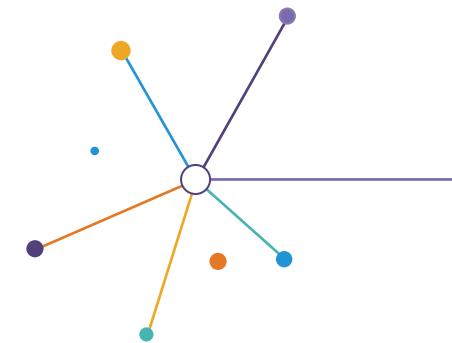
Always part of any aggregate function

Defined as a function of fields that have already been aggregated

▼ DATA RESULTS SQL

```
SELECT
    users.city AS "users.city",
    COUNT(*) AS "users.count"
FROM public.users AS users
GROUP BY 1
ORDER BY 2 DESC
LIMIT 500
```

Creating a **New Project**



LookML Projects Consist Of...

Looker models data using LookML projects

A project usually consists of

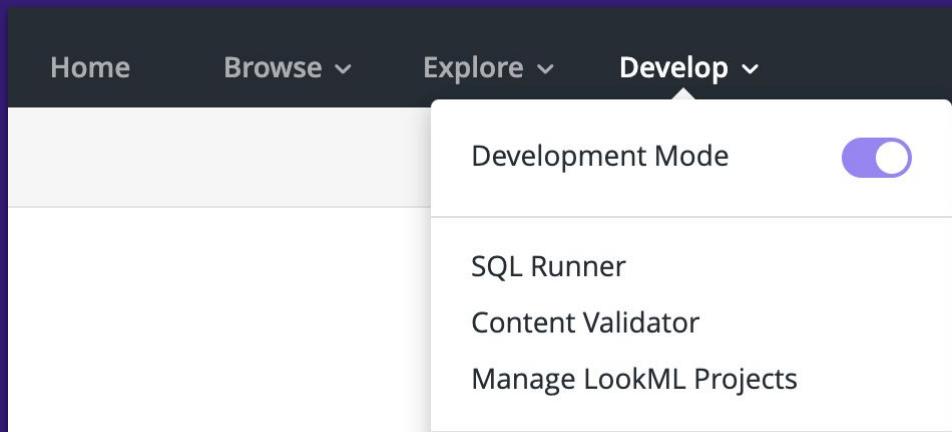
- One (or more) model files defining the project's Explore options and joins
- Multiple view files, with each corresponding to a database table or derived table
- One or more dashboard files which define their data and layouts, if you choose to use [LookML Dashboards](#) in addition to [User-Defined Dashboards](#)

EXERCISE TIME!

EXERCISE 1: Create Your LookML Project

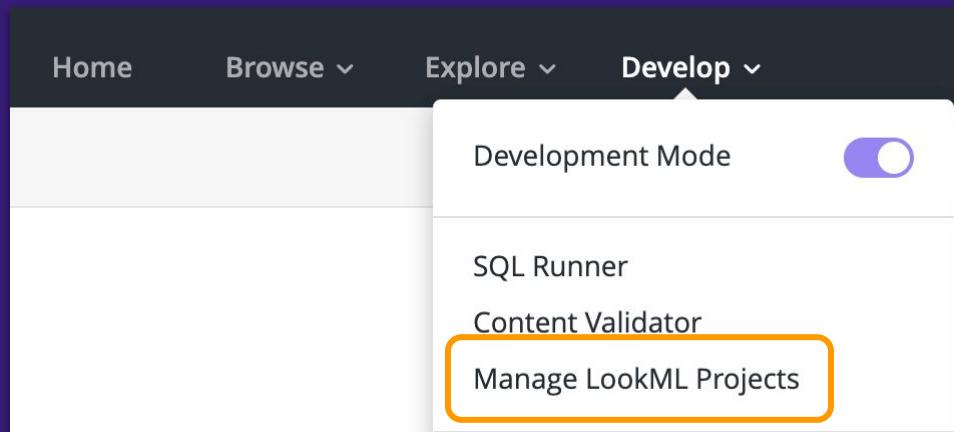
Step 1: Browse to your Looker instance via
<https://teach.corp.looker.com>

Step 2: Ensure you've placed your Looker instance into Development Mode



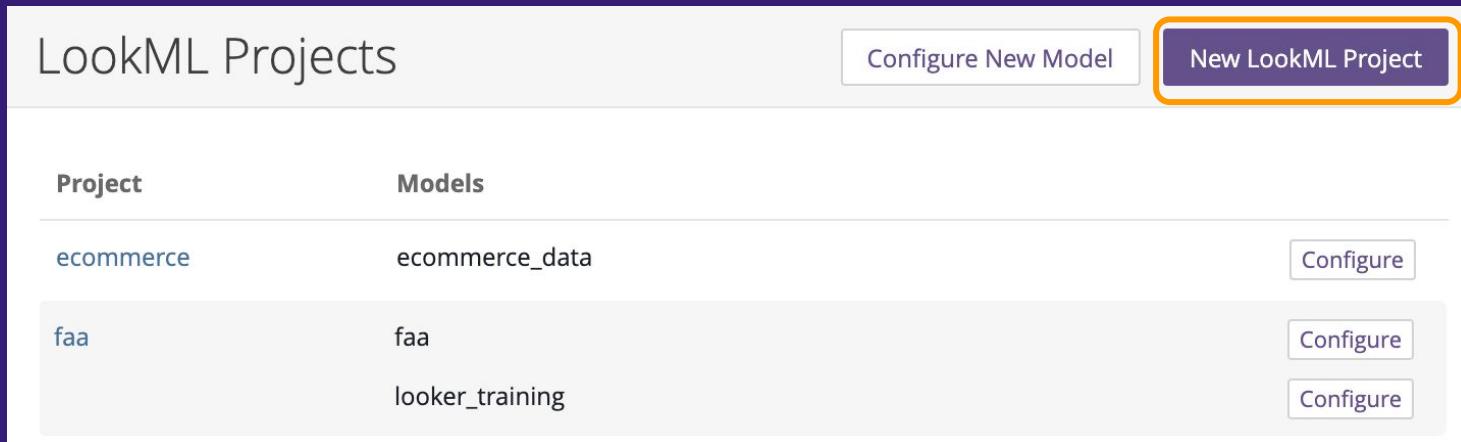
EXERCISE 1: Create Your LookML Project

Step 3: Navigate to Develop >
Manage LookML Projects



EXERCISE 1: Create Your LookML Project

Step 4: Select Manage LookML
Projects > New LookML Project



The screenshot shows the 'LookML Projects' page. At the top, there are two buttons: 'Configure New Model' and 'New LookML Project'. The 'New LookML Project' button is highlighted with a thick orange border. Below the buttons is a table with three rows, each representing a project. The columns are labeled 'Project' and 'Models'. The first row contains 'ecommerce' and 'ecommerce_data'. The second row contains 'faa' and 'faa'. The third row contains 'looker_training' and is partially visible. To the right of each project name is a 'Configure' button.

Project	Models	
ecommerce	ecommerce_data	Configure
faa	faa	Configure
looker_training		Configure

EXERCISE 1: Create Your LookML Project

Step 5: Create Project Name

New Project

Project Name

May contain lowercase letters, numbers, underscores, and dashes.
Other characters will be lowercased or replaced with "_".

Blank Project

Connection: events_ecommerce

Build Views From: All Tables Single Table
Table name

Schemas

Ignore Prefixes

Create Project

EXERCISE 1: Create Your LookML Project

Step 6: Choose your project starting point

New Project

Project Name

May contain lowercase letters, numbers, underscores, and dashes.
Other characters will be lowercased or replaced with "_".

Starting Point

- Generate Model from Database Schema
- Generate from SQL
- Clone Public Git Repository
- Blank Project

Table name

Schemas

Ignore Prefixes (?)

Create Project

EXERCISE 1: Create Your LookML Project

Step 7: Choose your database connection for your project

The screenshot shows the 'New Project' interface. At the top, there is a 'Project Name' input field with placeholder text: 'May contain lowercase letters, numbers, underscores, and dashes. Other characters will be lowercased or replaced with "_".' Below it is a 'Starting Point' section with two radio buttons: 'Generate Model from Database Schema' (selected) and 'Generate from SQL'. A large dropdown menu titled 'Connection' is open, listing several database connections: 'bigquery_publicdata' (selected and highlighted with an orange border), 'flightstats', 'looker_internal_analytics', 'thelook_events', and 'looker'. At the bottom of the interface, there are fields for 'Schemas' and 'Ignore Prefixes', and a 'Create Project' button.

New Project

Project Name

Starting Point

Connection

- bigquery_publicdata
- flightstats
- looker_internal_analytics
- thelook_events
- looker

Schemas

Ignore Prefixes

Create Project

EXERCISE 1: Create Your LookML Project

Step 8a: Choose to build your views from all available tables or a single table (*recommended*)

New Project

Project Name May contain lowercase letters, numbers, underscores, and dashes. Other characters will be lowercased or replaced with "_".

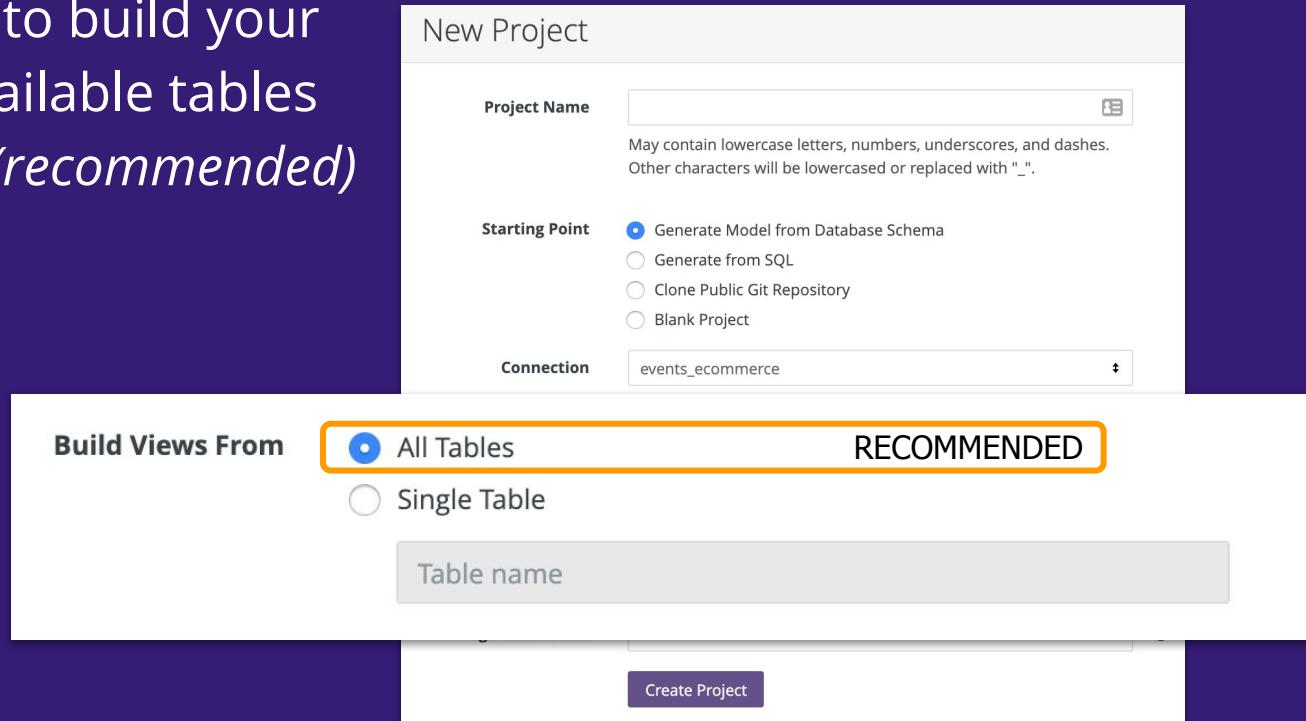
Starting Point Generate Model from Database Schema
 Generate from SQL
 Clone Public Git Repository
 Blank Project

Connection

Build Views From All Tables **RECOMMENDED** Single Table

Table name

Create Project



EXERCISE 1: Create Your LookML Project

Step 8b: Ignore the *Schemas* and *Ignore Prefixes* sections for now

Step 9: Click “*Create Project*” to create your new LookML project

New Project

Project Name May contain lowercase letters, numbers, underscores, and dashes. Other characters will be lowercased or replaced with "_".

Starting Point Generate Model from Database Schema
 Generate from SQL
 Clone Public Git Repository
 Blank Project

Connection

Build Views From All Tables
 Single Table

Schemas

Ignore Prefixes ?

Create Project

EXERCISE 1: Create Your LookML Project

This is what your project landing page should look like

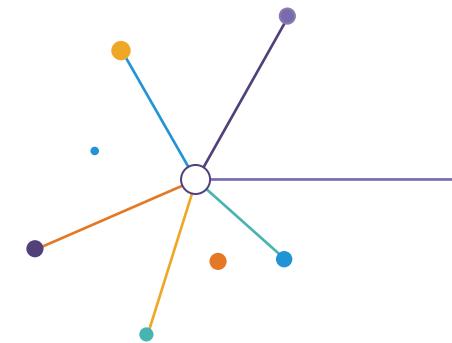
The screenshot shows the Looker interface in Development Mode. The top navigation bar includes 'Home', 'Browse', 'Explore', 'Develop', 'Admin', and 'Exit Development Mode'. A search bar at the top right contains 'Find & Replace in Project' and 'Go'. On the left, a sidebar shows 'sample_project_1' with sections for 'models' (containing 'sample_project_1.model') and 'views'. The main content area displays the 'sample_project_1.model' file's code. The code defines a datagroup named 'sample_project_1_default_datagroup' with a query to 'etl_log', a max cache age of '1 hour', and a persist with rule. It also lists various explores such as 'a', 'bsandell', 'company_list', 'connection_reg_r3', 'daily_active', 'daily_activity', 'distribution_centers', and 'events' which joins with 'users'. A 'Configure Git' button is visible on the left. A note on the left says 'Project has changed, errors are out of date.' with a 'Validate Again' button. A 'Quick Help' panel on the right provides a detailed explanation of the 'model' element.

```
sample_project_1.model
1 connection: "thelook_events"
2
3 # include all the views
4 include: "/views/**/.view"
5
6 datagroup: sample_project_1_default_datagroup {
7   # sql_trigger: SELECT MAX(id) FROM etl_log;
8   max_cache_age: "1 hour"
9 }
10
11 persist_with: sample_project_1_default_datagroup
12
13 explore: a {}
14
15 explore: bsandell {}
16
17 explore: company_list {}
18
19 explore: connection_reg_r3 {}
20
21 explore: daily_active {}
22
23 explore: daily_activity {}
24
25 explore: distribution_centers {}
26
27 explore: events {
28   join: users {
29     type: left_outer
30     sql_on: ${events.user_id} = ${users.id} ;
31     relationship: many_to_one
32   }
33 }
34 }
```

A **model** references a combination of related explores. Unlike other LookML elements, a model is not declared explicitly with the `model` keyword.

```
model: {
  access_grant: identifier
  case_sensitive: yes or no
  connection: "string"
  datagroup: identifier
  explore: identifier
  fiscal_month_offset: number
  include: "string"
  label: possibly-localized-string
  map_layer: identifier
  named_value_format: identifier
  persist_for: "string"
  persist_with: datagroup-ref
  test: identifier
  view: identifier
  week_start_day: monday or ...
}
```

Looker **Dev Environment**



Prod Mode vs. Dev Mode

In **PRODUCTION MODE**,
users typically explore data in
Looker

The data model is shared
across all users, and LookML
files are treated as read-only

In **DEVELOPMENT MODE**,
developers can make LookML
changes here **only**

This mode accesses a totally
separate version of the data
model that only developers
can see and edit

In Git terms, development is
handled by a separate branch

**Development mode
allows developers to
make and test LookML
changes without
affecting other users**

Toggling Development Mode

You can use *Develop > Development Mode* like we did earlier today, or **CTRL+SHIFT+D**

The LookML and Explore menus will display different options in prod mode vs. dev mode

You are in **Development Mode**.

sample_project_1

Version control hasn't been set up for this project.

Configure Git

Project Settings

Run All Tests

Project has changed, errors are out of date.

Validate Again

SAMPLE_PROJECT_1 + 🔎 🗂️

models

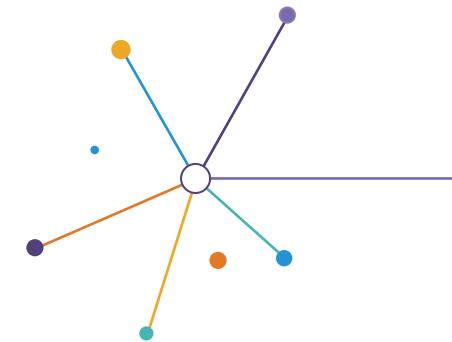
sample_project_1.model

views

sample_project_1.model

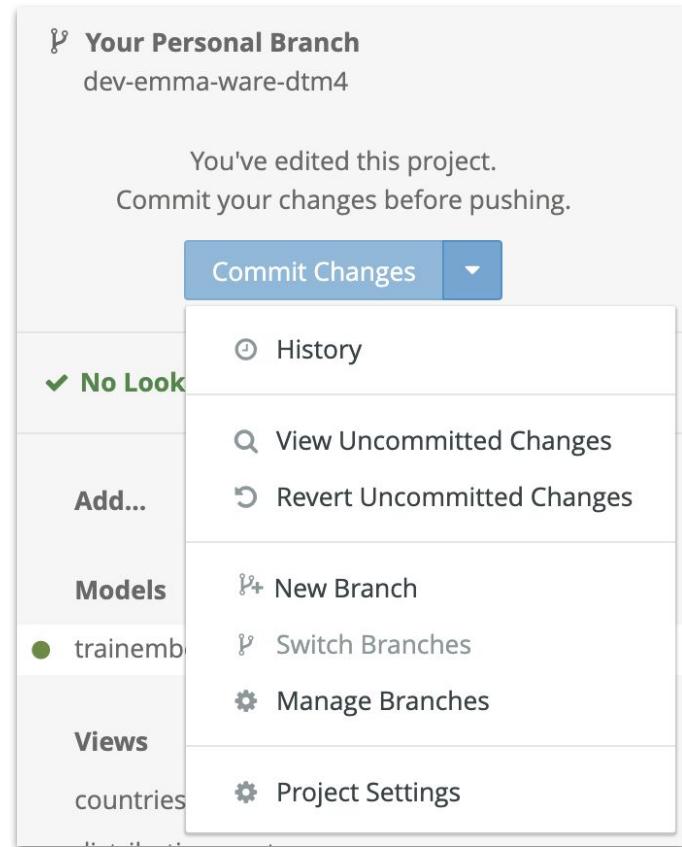
```
connection: "thelook_events"
# include all the views
include: "/views/**/*.view"
datagroup: sample_project_1
# sql_trigger: SELECT MAX(
max_cache_age: "1 hour"
}
persist_with: sample_project_
explore: a {}
explore: bsandell {}
explore: company_list {}
```

Project Version **Control**



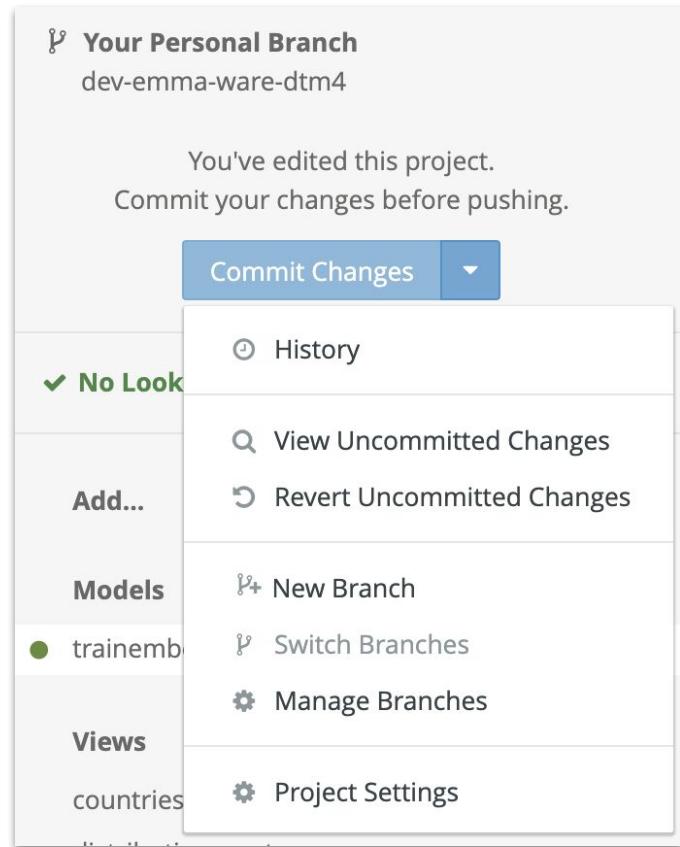
Looker & Version Control

Looker integrates with Git, empowering you to test changes, save work and collaborate with other developers via Looker's own IDE



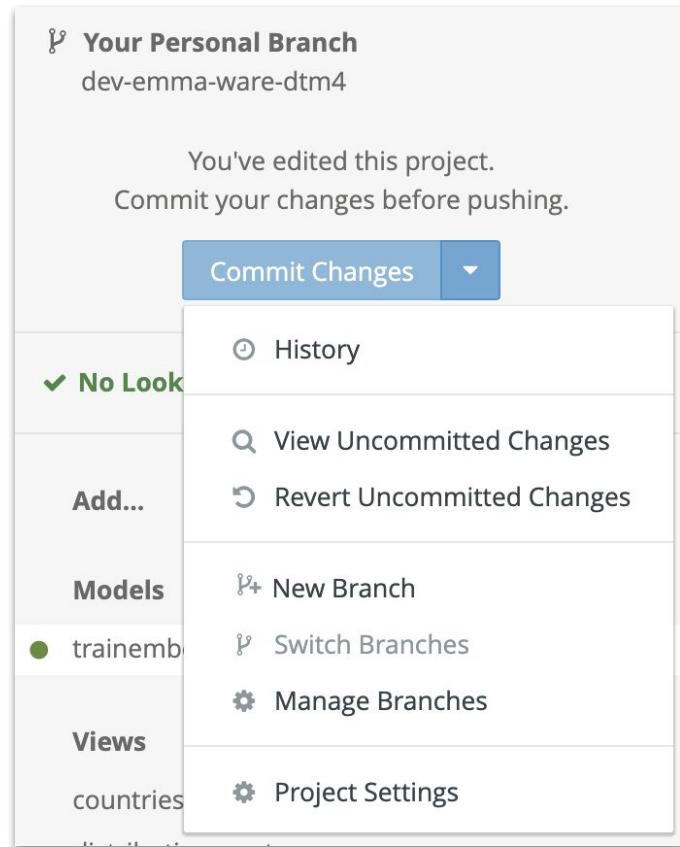
Version Control & Looker

Each LookML project corresponds with a Git repo, with each developer branch correlates to a Git branch



Version Control & Looker

Looker can connect to GitHub, Bitbucket or any Git server that uses HTTPS or SSH key for authentication



Looker:Git Workflow

Any LookML files with changes appear with a green dot next to their name

Depending on your projects code quality setting, you may be required to validate the changes and resolve surfaced errors

The screenshot shows a Looker interface with a validation dialog overlaid on a project page. The validation dialog has a blue header bar with the text "Validate LookML" and a dropdown arrow. Below the dialog, the project page displays the following information:

- Your Personal Branch:** dev-emma-ware-dtm4
- Please validate your LookML code.**
- Validate LookML** button (highlighted with an orange border)
- Project has changed, errors are out of date.**
- Validate Again** button
- Add...** button with a plus sign
- Models** section:
 - trainembed_thelook (green dot)
 - ... (three dots)
- Views** section:
 - countries

Looker:Git Workflow

Commit changes to your Git branch, and add a brief description of the changes

The screenshot shows a Git commit dialog box. At the top, it says "Your Personal Branch" and "dev-emma-ware-dtm4". Below that, a message states "You've edited this project. Commit your changes before pushing." A large blue button labeled "Commit Changes" is highlighted with an orange border. The main body of the dialog is titled "Commit". It has a "Message" field containing "Renamed event view, updated model file|", followed by a placeholder "Briefly describe the changes you've made to the project.". Under "Files", there is a list of three items, each with a checked checkbox:

- event.view.lkml (New file)
- crunchbase.model.lkml (Modified)
- events.view.lkml (Deleted)

At the bottom right are "Cancel" and "Commit" buttons.

Looker:Git Workflow

Push your changes to production, also pulling from production if other developers have made changes within the project branch

Verify no content has broken due to changes by using the content validator

This branch has **1 undeployed** commit.

Deploy to Production ▾

The production version of this project has been modified.

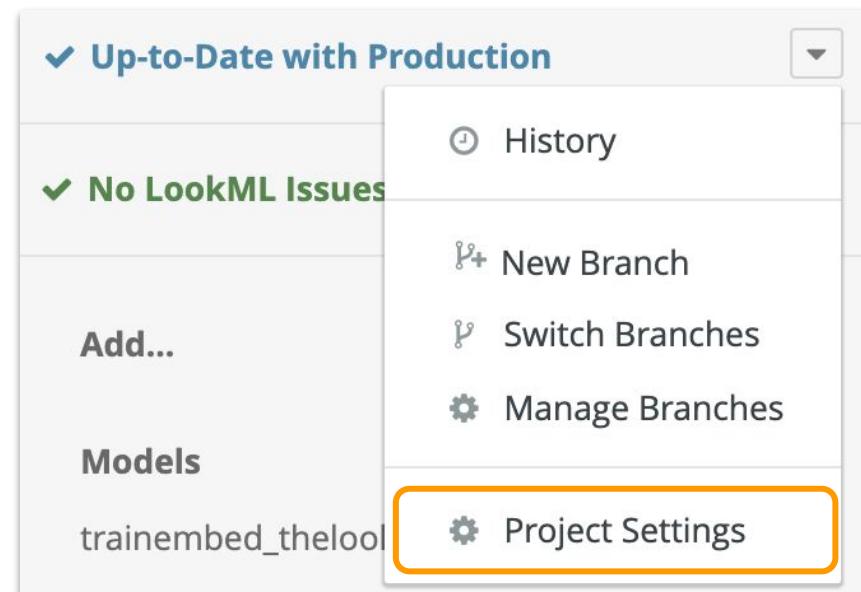
There is **1 commit** to be pulled.

Pull from Production ▾

Git Settings

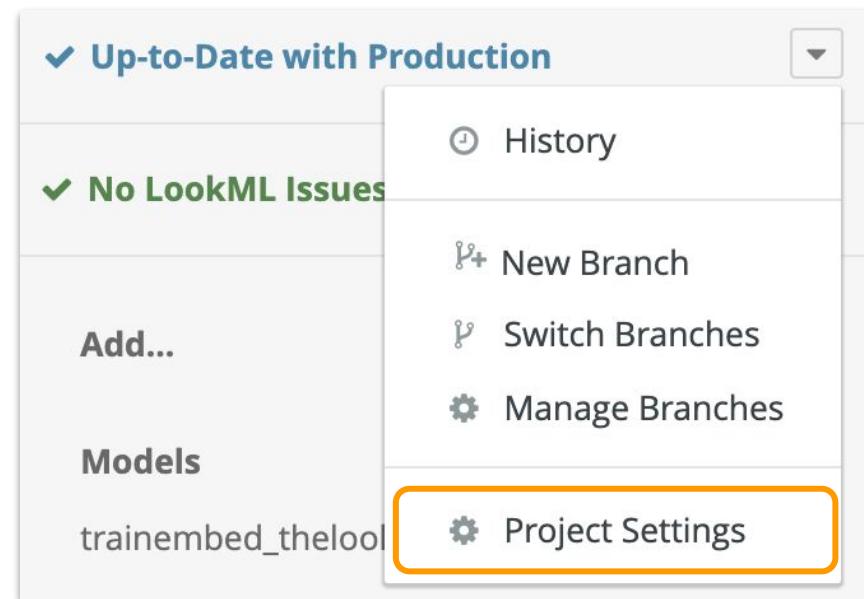
By default, developers can commit and deploy their changes to production

To restrict commit/deploy access, configure your project with either *Pull Requests Recommended* or *Pull Requests Required* options toggled



Git Settings

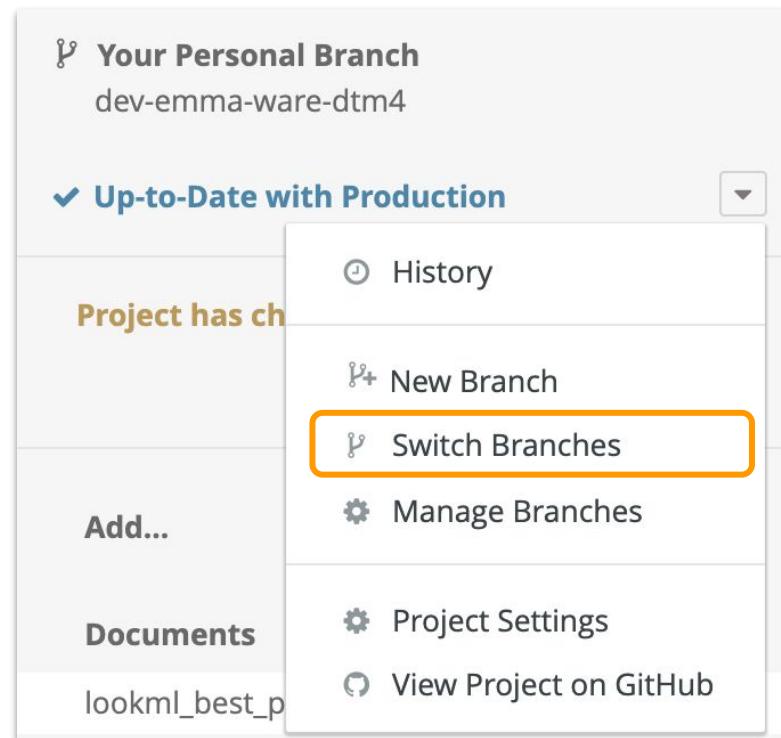
To enable Pull Requests, select Project Settings and follow the setup instructions



Git Branches

In Dev Mode, Looker automatically creates a personal development for you

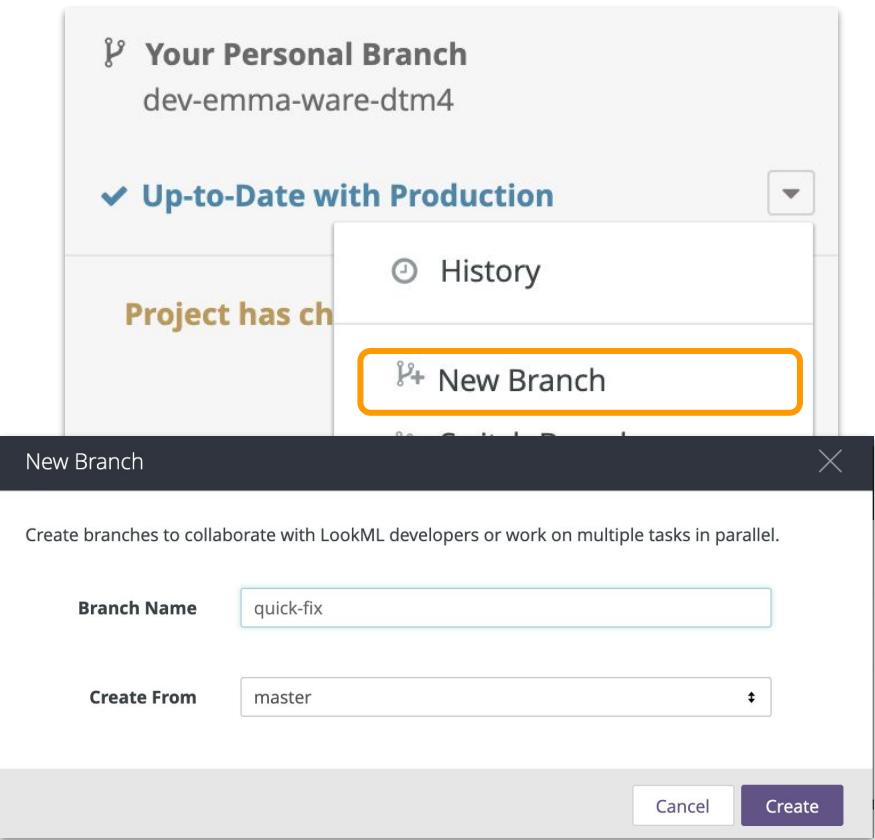
- This is specific to you and read-only to other developers
- You can view and test other developers' personal branches through the Switch Branch option of the Git drop-down menu



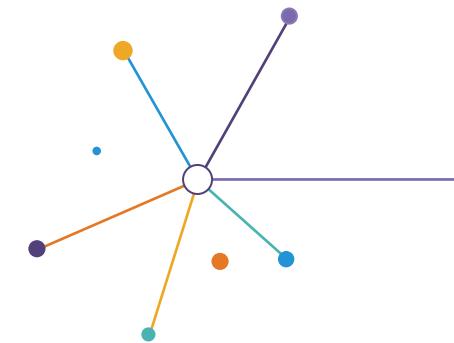
Git Branches

You can also create other branches if you

- want to resolve a minor issue or implement a quick fix during a major project
- collaborate with other developers on a project using shared branches
- work on multiple sets of features



How Looker **Writes SQL**



How Looker Writes SQL

Dimensions & Measures in **base view**

- Dimensions are in the **GROUP BY**
- Measures are in **aggregating functions**

Dimensions & Measures in the **joined view**

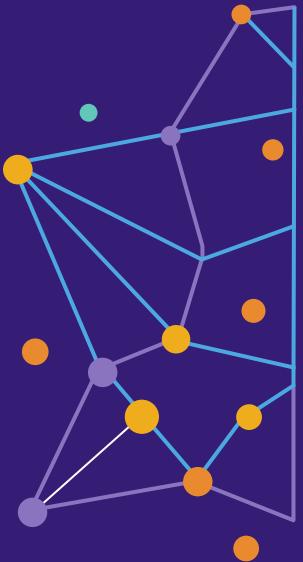
- Base view is still joined in
- Unnecessary views are **not** joined in

How Looker Writes SQL

Filter Dimensions & Measures

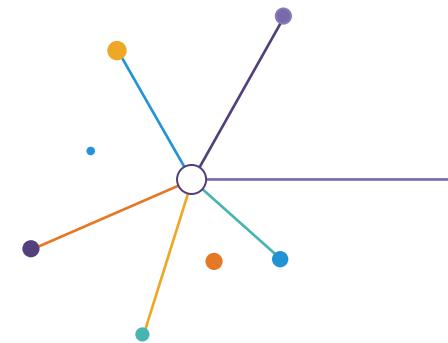
- Dimensions filter in the **WHERE** clause
- Measures filter in the **HAVING** clause

Row limit in the **LIMIT**



Creating Dimensions & Measures

Dimensions



Referencing Objects

How to reference a database object in Looker

- \${TABLE} references the table defined in the view
- Looker automatically creates dimensions for every field in the view

```
dimension: sale_price {  
    type: number  
    sql: ${TABLE}.sale_price ;;  
}  
  
measure: total_revenue {  
    type: sum  
    sql: ${sale_price} ;;  
}  
  
measure: average_sale_price {  
    type: average  
    sql: ${sale_price} ;;  
}
```

Referencing Objects

How to reference another Looker object:

`${field_name}` references the Looker object

```
dimension: sale_price {  
    type: number  
    sql: ${TABLE}.sale_price ;;  
}  
  
measure: total_revenue {  
    type: sum  
    sql: ${sale_price} ;;  
}  
  
measure: average_sale_price {  
    type: average  
    sql: ${sale_price} ;;  
}
```

Dimension Types

string

- Used for text dimensions
- Default dimension type

number

- Used for numeric dimensions
- Commonly used with date calculations or basic row-level math across fields

```
dimension: full_name {  
  type: string  
  sql: ${first_name} || ${last_name}  
}
```

```
dimension: days_since_signup {  
  type: number  
  sql: DATEDIFF(day, ${created_date},  
    current_date) ;;  
}
```

Dimension Types

yesno

- Defines logical condition in SQL parameter
- Field becomes either yes or no

tier

- Buckets a dimension using CASE statements
- Styles include classic, interval, integer and relational

```
dimension: is_new_customer {  
  type: yesno  
  sql: ${days_since_signup} <= 90 ;;  
}
```

```
dimension: days_since_signup_tier {  
  type: tier  
  tiers: [0, 30, 90, 180, 360, 720]  
  sql: ${days_since_signup} ;;  
  style: integer  
}
```

Dimension Groups - Time

Looker can cast a date or timestamp into different forms of time

The `timeframes` parameter is used to specify the specific date and time parts required

```
dimension_group: created {  
  type: time  
  timeframes: [raw, time, date, hour,  
    hour_of_day, day_of_week,  
    day_of_week_index, time_of_day,  
    week,  
    month_num, month, year, quarter,  
    quarter_of_year]  
  sql: ${TABLE}.created_at ;;  
}
```

Dimension Groups - Time

Number of dimension fields created is dependent upon the number of timeframes listed

Reference fields by appending the data or time part desired to the name of the dimension group with an underscore

```
dimension_group: created {  
    type: time  
    timeframes: [raw, time, date, hour,  
                hour_of_day, day_of_week,  
                day_of_week_index, time_of_day, week,  
                month_num, month, year, quarter,  
                quarter_of_year]  
    sql: ${TABLE}.created_at ;;  
}
```

```
 ${created_time}, ${created_date},  
 ${created_hour_of_day}, etc.
```

Dimension Groups - Duration

Duration calculates a set of interval-based duration dimensions

Use the `intervals` parameter to specify the specific date and time durations you desire

```
dimension_group: enrolled {  
  type: duration  
  intervals: [second, minute, hour,  
    day, week, month, quarter, year]  
  sql_start: ${TABLE}.enrollment_date ;  
  sql_end: ${TABLE} .graduation_date ;;  
}
```

Dimension Groups - Duration

Number of created dimension fields depends on the number of intervals listed

Reference fields by prepending the duration component desired to the name of the dimension group with an underscore

```
dimension_group: enrolled {  
    type: duration  
    intervals: [second, minute, hour,  
               day, week, month, quarter, year]  
    sql_start: ${TABLE}.enrollment_date ;;  
    sql_end: ${TABLE}.graduation_date ;;  
}
```

```
 ${hours_enrolled}, ${days_enrolled},  
 ${years_enrolled}, etc.
```

EXERCISE TIME!

EXERCISE 2: Working with Dimensions

Task 1: Create a new dimension in your users view that combines City and State into one single field

[Level 1](#)

[Level 2](#)

[Level 3](#)

Task 2: Create a dimension that groups individual ages into the following age group buckets: 18, 25, 35, 45, 55, 65, 75, 90

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 2: Working with Dimensions

Task 3: Create a new dimension that calculates whether the Traffic Source that brought in a given user was “Email” or not. Your solution should include the yesno dimension type

[Level 1](#)

[Level 2](#)

[Level 3](#)

Task 4: Create a Shipping Days dimension that calculates the number of days between the order ship date and the order delivered date within the order_items view

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 2: Task 1 Level 1

Task 1: Create a new dimension in your users view that combines City and State into one single field

```
dimension: _____ {  
    type: _____  
    sql: ${____} || ‘, ‘ || ${____};;  
}
```

EXERCISE 2: Task 1 Level 2

Task 1: Create a new dimension in your users view that combines City and State into one single field

```
-----: ----- {  
  ___: -----  
  __: ${__} || ‘, ‘ || ${__};;  
}  
  
-----: ----- {  
  ___: -----  
  __: -----(${__}, ‘, ‘, ${__});;  
}
```

EXERCISE 2: Task 1 Level 3

Task 1: Create a new dimension in your users view that combines City and State into one single field

EXERCISE 2: Task 2 Level 1

Task 2: Create a Shipping Days dimension that calculates the number of days between the order ship date and the order delivered date within the order_items view

```
dimension: _____ {  
    type: _____  
    sql: DATEDIFF(___, ${_____}, ${_____});;  
}
```

```
dimension_group: _____ {  
    type: _____  
    sql_start: ${_____};;  
    sql_end: ${_____};;  
    intervals: [___]  
}
```

EXERCISE 2: Task 2 Level 2

Task 2: Create a Shipping Days dimension that calculates the number of days between the order ship date and the order delivered date within the order_items view

```
-----: ----- {  
    ___: ____  
    ___: _____(____, ${_____} , ${_____});;  
}  
  
-----: ----- {  
    ___: ____  
    -----: ${_____};;  
    -----: ${_____};;  
    -----: [__]  
}
```

EXERCISE 2: Task 2 Level 3

Task 2: Create a Shipping Days dimension that calculates the number of days between the order ship date and the order delivered date within the order_items view

EXERCISE 2: Task 3 Level 1

Task 3: Create a new dimension that calculates whether the Traffic Source that brought in a given user was “Email” or not. Your solution should include the yesno dimension type

```
dimension: _____ {  
    type: _____  
    sql: ${_____} = '_____';;  
}
```

EXERCISE 2: Task 3 Level 2

Task 3: Create a new dimension that calculates whether the Traffic Source that brought in a given user was “Email” or not. Your solution should include the yesno dimension type

```
-----: ----- {
    ___: ____
    ___: ${_____} = '____' ;;
}
```

EXERCISE 2: Task 3 Level 3

Task 3: Create a new dimension that calculates whether the Traffic Source that brought in a given user was “Email” or not. Your solution should include the yesno dimension type

EXERCISE 2: Task 4 Level 1

Task 4: Create a dimension that groups individual ages into the following age group buckets: 18, 25, 35, 45, 55, 65, 75, 90

```
dimension: _____ {  
    type: ____  
    tiers: [18, 25, 35, 45, 55, 65, 75, 90]  
    sql: ${__} ;;  
    style: _____  
}
```

EXERCISE 2: Task 4 Level 2

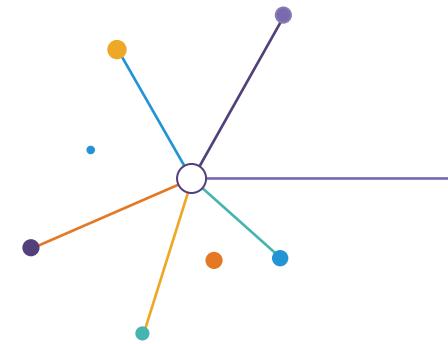
Task 4: Create a dimension that groups individual ages into the following age group buckets: 18, 25, 35, 45, 55, 65, 75, 90

```
-----: ----- {  
  ___: ___  
  ___: [___, ___, ___, ___, ___, ___, ___, ___]  
  ___: ${___} ;;  
  ___: -----  
}  
-----
```

EXERCISE 2: Task 4 Level 3

Task 4: Create a dimension that groups individual ages into the following age group buckets: 18, 25, 35, 45, 55, 65, 75, 90

Measures



Looker Measure Types

Three (3) most common
Measure types

- sum
- average
- count

```
dimension: cost {  
    type: number  
    sql: ${TABLE}.cost ;;  
}
```

```
measure: total_cost {  
    type: sum  
    sql: ${cost} ;;  
}
```

```
measure: average_cost {  
    type: average  
    sql: ${cost} ;;  
}
```

Looker Measure Types

Two types of counts

- type: count counts rows in that table and does NOT require a SQL parameter
- type: count_distinct computes a distinct count of the field in the SQL parameter

In order_items.view:

```
measure: count_items_ordered {  
  type: count  
}
```

```
measure: count_users {  
  type: count_distinct  
  sql: ${user_id} ;;  
}
```

EXERCISE TIME!

EXERCISE 3: Working with Measures

Task 1: Create a measure that calculates the distinct number of orders within the *order_items* view

[Level 1](#)

[Level 2](#)

[Level 3](#)

Task 2: Create a measure that calculates total sales (use the *sale_price* dimension)

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 3: Working with Measures

Task 3: Create a measure that calculates average sales (use the sale_price dimension)

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 3: Task 1 Level 1

Task 1: Create a measure that calculates the distinct number of orders within the `order_items` view

```
measure: _____ {  
    _____: "A count of unique orders"  
    type: _____  
    sql: ${_____} ;;  
}
```

EXERCISE 3: Task 1 Level 2

Task 1: Create a measure that calculates the distinct number of orders within the `order_items` view

```
-----: _____ {  
    -----: "A count of unique orders"  
    ___: _____  
    __: ${_____} ;;  
}  
_____
```

EXERCISE 3: Task 1 Level 3

Task 1: Create a measure that calculates the distinct number of orders within the order_items view

EXERCISE 3: Task 2 Level 1

Task 2: Create a measure that calculates total sales (sale_price)

```
measure: _____ {  
    _____: "A count of unique orders"  
    type: _____  
    sql: ${_____} ;;  
}
```

EXERCISE 3: Task 2 Level 2

Task 2: Create a measure that calculates total sales (sale_price)

```
-----: ----- {  
    -----: "A count of unique orders"  
    -----:  
    ___: ${_____} ;;  
}  
-----:
```

EXERCISE 3: Task 2 Level 3

Task 2: Create a measure that calculates total sales (sale_price)

EXERCISE 3: Task 3 Level 1

Task 3: Create a measure that calculates average sales (sale_price)

```
measure: _____ {  
    type: _____  
    sql: ${_____} ;;  
}
```

EXERCISE 3: Task 3 Level 2

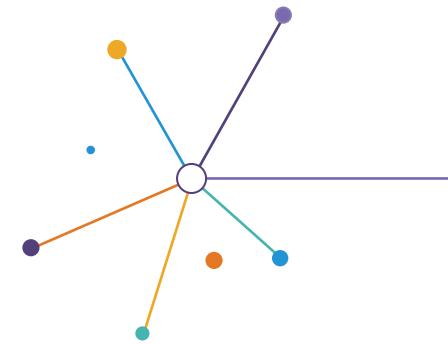
Task 3: Create a measure that calculates average sales (sale_price)

```
-----: ----- {  
  ____: _____  
  ___: ${_____} ;;  
 }  
 -----
```

EXERCISE 3: Task 3 Level 3

Task 3: Create a measure that calculates average sales (sale_price)

Advanced **Fields**



Filtered Measures

You can use `yesno` dimensions as filters

Looker transfers the logic of `yesno` into the case statement that produces the measure

You can use other dimension types as filters as well

```
measure: count_female_users {  
    type: count  
    filters: {  
        field: gender  
        value: "Female"  
    }  
}
```

```
measure: total_sales_new_users {  
    type: sum  
    sql: ${sale_price} ;;  
    filters: {  
        field: users.is_new_user  
        value: "Yes"  
    }  
}
```

Using Measures to Define Other Measures

Use measures within other measures for more complex calculations

```
measure: percentage_female_users {  
    type: number  
    value_format_name: percent_1  
    sql: 1.0*${count_female_users}  
        /NULLIF(${count}, 0) ;;  
}
```

Using Measures to Define Other Measures

Measures that use other measures in their SQL definitions should use
type: number

Use value_format_name parameter to format final output

```
measure: percentage_female_users {  
    type: number  
    value_format_name: percent_1  
    sql: 1.0*${count_female_users}  
        /NULLIF(${count}, 0) ;;  
}
```

Referencing Fields in Other View Files

You can reference fields in another view if the two views have been joined together in an Explore

```
dimension: profit {  
    type: number  
    value_format_name: usd  
    sql: ${sale_price} -  
        ${inventory_items.cost} ;;  
}
```

Referencing Fields in Other View Files

Fields that reference other views require an Explore with defined joins for both involved views

When selecting a field in the UI that references multiple views, Looker needs to know how to execute the joins necessary for completing the calculation

```
dimension: profit {  
  type: number  
  value_format_name: usd  
  sql: ${sale_price} -  
    ${inventory_items.cost} ;;  
}
```

EXERCISE TIME!

EXERCISE 4: Advanced Fields

Task 1: Create a filtered measure that calculates the total sales for only users that came to the website via the Email traffic source [order_items view]

[Level 1](#)

[Level 2](#)

[Level 3](#)

Task 2: Create a field that calculates the percentage of sales that are attributed to users coming from the email traffic source [order_items view]

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 4: Advanced Fields

Task 3: Calculate the average spend per user by dividing the total sales measure by the user count measure, and format the output so that it shows up in USD

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 4: Task 1 Level 1

Task 1: Create a filtered measure that calculates the total sales for only users that came to the website via the Email traffic source [order_items view]

```
measure: total_sales_email_users {  
    type: ___  
    sql: ${_____} ;;  
    filters: {  
        field: users.is_email_source  
        value: "___"  
    }  
}
```

EXERCISE 4: Task 1 Level 2

Task 1: Create a filtered measure that calculates the total sales for only users that came to the website via the Email traffic source [order_items view]

```
-----: ----- {  
  ___: ___  
  ___: ${_____} ;;  
  _____: {  
    ____: users.is_email_source  
    ____: "___"  
  }  
}
```

EXERCISE 4: Task 1 Level 3

Task 1: Create a filtered measure that calculates the total sales for only users that came to the website via the Email traffic source [order_items view]

EXERCISE 4: Task 2 Level 1

Task 2: Create a field that calculates the percentage of sales that are attributed to users coming from the email traffic source [order_items view]

```
measure: percentage_sales_email_source {  
    type: _____  
    value_format_name: _____2  
    sql: 1.0*${_____}  
        /NULLIF(${_____}, 0) ;;  
}
```

EXERCISE 4: Task 2 Level 2

Task 2: Create a field that calculates the percentage of sales that are attributed to users coming from the email traffic source [order_items view]

```
-----: ----- {  
  -----: -----  
  -----: -----  
  -----: 1.0*${-----}  
  /NULLIF(${-----}, 0) ;;  
}
```

EXERCISE 4: Task 2 Level 3

Task 2: Create a field that calculates the percentage of sales that are attributed to users coming from the email traffic source [order_items view]

EXERCISE 4: Task 3 Level 1

Task 3: Calculate the average spend per user by dividing the total sales measure by the user count measure, and format the output so that it shows up in usd

```
measure: percentage_sales_email_source {  
    type: _____  
    value_format_name: _____2  
    sql: 1.0*${_____}  
        /NULLIF(${_____}, 0) ;;  
}
```

EXERCISE 4: Task 3 Level 2

Task 3: Calculate the average spend per user by dividing the total sales measure by the user count measure, and format the output so that it shows up in usd

```
-----: ----- {  
  ___: _____  
  -----: _____2  
 ___: 1.0*${_____}  
 /NULLIF(${_____}, 0) ;;  
 }
```

EXERCISE 4: Task 3 Level 3

Task 3: Calculate the average spend per user by dividing the total sales measure by the user count measure, and format the output so that it shows up in usd

Helpful Field Parameters

`hidden`: hides a field from the user interface while still allowing it to be available for modeling (*great for fields like primary keys that are not meaningful to users*)

`label`: changes how a field name will appear in the Field Picker

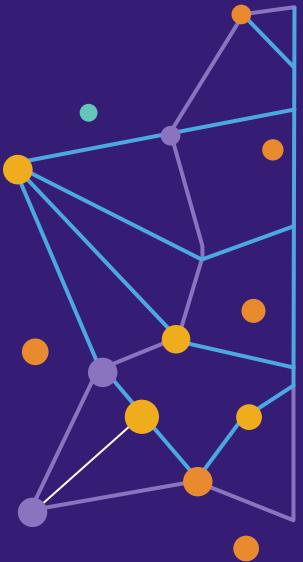
`description`: displays additional information about a field to users upon hovering

Helpful Field Parameters

`value_format_name`: formats Looker cells using built-in or custom format names

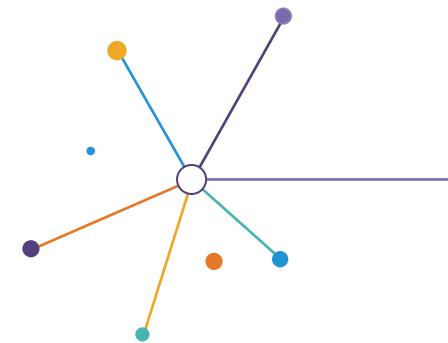
`drill_fields`: controls what fields are shown to a user when he or she clicks on the value of a table cell to “drill” into the data while Exploring

`group_label`: combines fields into custom groups within a view in the Field Picker



Working with Model Files

Explores & **Join Logic**



Building Explores - Key Join Parameters

join: the name of the join, which is typically the name of the view being joined

type: the type of join that should occur (left_outer join by default)

```
explore: inventory_items {  
  join: products {  
    type: left_outer  
    sql_on: ${inventory_items.product_id}  
          = ${products.id} ;;  
    relationship: many_to_one  
  }  
}
```

Building Explores - Key Join Parameters

`sql_on`: the fields that should be used within the ON clause of the SQL query in order to join the two tables together

`relationship`: how the two tables relate to each other

```
explore: inventory_items {  
  join: products {  
    type: left_outer  
    sql_on: ${inventory_items.product_id}  
          = ${products.id} ;;  
    relationship: many_to_one  
  }  
}
```

Anatomy of an Explore

```
explore: flights {  
    join: carriers {  
        type: left_outer  
        sql_on: ${flights.carrier} = ${carriers.code} ;;  
        relationship: many_to_one  
    }  
}
```

Explore Name
Base View

```
join: aircraft {  
    type: left_outer  
    sql_on: ${flights.tail_num} = ${aircraft.tail_num} ;;  
    relationship: many_to_one  
}
```

Standard Join

Types of Joins

Standard join

Joins renaming the view,
allowing the same view joined
twice

Indirect join

```
explore: flights {
  join: carriers {
    type: left_outer
    sql_on: ${flights.carrier} = ${carriers.code} ;;
    relationship: many_to_one
  }

  join: aircraft {
    type: left_outer
    sql_on: ${flights.tail_num} = ${aircraft.tail_num} ;;
    relationship: many_to_one
  }

  join: aircraft_origin {
    from: airports
    type: left_outer
    sql_on: ${flights.origin} = ${aircraft_origin.code} ;;
    relationship: many_to_one
    fields: [full_name, city, state, code, map_location]
  }

  join: aircraft_destination {
    from: airports
    type: left_outer
    sql_on: ${flights.destination} = ${aircraft_destination.code} ;;
    relationship: many_to_one
    fields: [full_name, city, state, code]
  }

  join: aircraft_flight_facts {
    view_label: "Aircraft"
    type: left_outer
    sql_on: ${aircraft.tail_num} = ${aircraft_flight_facts.tail_num} ;;
    relationship: one_to_one
  }
}
```

Building SQL from Explores

Typical SQL statement:

```
SELECT column_a, column_b, ..., column_n  
FROM table_a as table_a  
JOIN table_b as table_b on table_a.field_1 = table_b.field_1  
JOIN table_c as table_c on table_a.field_2 = table_c.field_2  
WHERE some condition equals some value
```

```
SELECT user-chosen dimensions and measures  
FROM clause plus any JOINS = Explore  
WHERE clause = user-added filters
```

Translating SQL into a LookML Explore

```
SELECT
    flights.destination AS "flights.destination",
    carriers.name AS "carriers.name",
    aircraft.name AS "aircraft.name",
    aircraft_origin.city AS "aircraft_origin.city",
    COUNT(*) AS "flights.1_count"

FROM flights AS flights

LEFT JOIN public.carriers AS carriers
    ON flights.carrier = carriers.code

LEFT JOIN public.aircraft AS aircraft
    ON flights.tail_num = aircraft.tail_num

LEFT JOIN public.airports AS aircraft_origin
    ON flights.origin = aircraft_origin.code

WHERE (flights.cancelled = 'N') AND (aircraft_origin.state = 'CA')
GROUP BY 1,2,3,4,5,6
```

```
explore: flights {

    join: carriers {
        sql_on: ${flights.carrier} = ${carriers.code} ;;
        relationship: many_to_one
    }

    join: aircraft {
        sql_on: ${flights.tail_num} = ${aircraft.tail_num} ;;
        relationship: many_to_one
    }

    join: aircraft_origin {
        from: airports
        sql_on: ${flights.origin} = ${aircraft_origin.code} ;;
        relationship: many_to_one
        fields: [full_name, city, state, code]
    }
}
```

Helpful Explore Parameters

`label`: changes how an Explore name will appear

`description`: displays additional information about an Explore upon hovering over the information icon within the Explore dropdown menu

`view_label`: changes the label of the view within the field picker in the Explore

Helpful Explore Parameters

`group_label`: combines Explores into custom groups within the Explore dropdown menu the Explore

`fields`: limits the scope of fields that are available within an Explore or view

EXERCISE TIME!

EXERCISE 5: Building an Explore

Task 1: Create a users Explore that starts with the users view and then joins in the order_items view

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 5: Task 1 Level 1

Task 1: Create a users Explore that starts with the users view and then joins in the order_items view

```
explore: _____ {  
    join: _____ {  
        type: _____  
        sql_on: ${_____.__} = ${_____.user_id} ;;  
        relationship: one_to_many  
    }  
}
```

EXERCISE 5: Task 1 Level 2

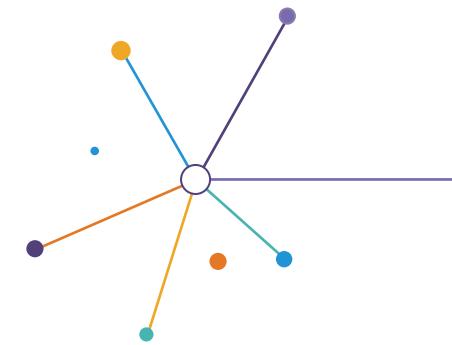
Task 1: Create a users Explore that starts with the users view and then joins in the order_items view

```
-----: ____ {  
  ____: _____ {  
    ____:  
      ____: ${_____.__} = ${_____.___.____} ;;  
      _____: _____  
  }  
}
```

EXERCISE 5: Task 1 Level 3

Task 1: Create a users Explore that starts with the users view and then joins in the order_items view

Symmetric **Aggregation**



What is the Fanout Problem?

Consider these two tables individually

- `count(*)` produces a count of customers and orders
- `sum(visits)` and `sum(amount)` give total visits and total revenue

customer_id	first_name	last_name	visits
1	Amelia	Earhart	2
2	Charles	Lindbergh	2
3	Wilbur	Wright	4

order_id	amount	customer_id
1	25.00	1
2	50.00	1
3	75.00	2
4	100.00	3

What is the Fanout Problem?

Consider joining these two tables on customer_id, like we would in the following Explore:

```
explore: customers {
  join: orders {
    sql_on: ${customers.customer_id} = ${orders.customer_id} ;;
  }
}
```

customer_id	first_name	last_name	visits
1	Amelia	Earhart	2
2	Charles	Lindbergh	2
3	Wilbur	Wright	4

order_id	amount	customer_id
1	25.00	1
2	50.00	1
3	75.00	2
4	100.00	3

What is the Fanout Problem?

This join “fans out” the customer table

- count(*) and sum(amount) still work on the order-table fields, but...
- count(*) and sum(visits) produce bad results on customer-table fields
- Looker can handle this situation with Symmetric Aggregates!

customer_id	first_name	last_name	visits	order_id	amount	customer_id
1	Amelia	Earhart	2	1	25.00	1
1	Amelia	Earhart	2	2	50.00	1
2	Charles	Lindbergh	2	3	75.00	2
3	Wilbur	Wright	4	4	100.00	3

Using Symmetric Aggregates

To use symmetric aggregation, two things must be done

Specify primary
keys in the view
files

Correctly specify
the “relationship”
parameter

Symmetric Aggregates - Primary Keys

This means a field that uniquely identifies each row

If none exists, we can make one by concatenating fields together

```
view: orders {  
    dimension: order_id {  
        type: number  
        primary_key: yes  
        sql: ${TABLE}.order_id ;;  
    }  
}
```

```
view: customers {  
    dimension: customer_id {  
        type: number  
        primary_key: yes  
        sql: ${TABLE}.customer_id ;;  
    }  
}
```

Symmetric Aggregates - Relationship parameter

Four possible relationship values include

- one_to_one
- one_to_many
- many_to_one
- many_to_many

```
explore: customers {
  join: orders {
    sql_on: ${customers.customer_id} = ${orders.customer_id} ;;
    relationship: one_to_many
  }
}
```

Symmetric Aggregates - Relationship parameter

Left side: the view joined from
(*other view used in “sql_on:”*)

```
explore: customers {
  join: orders {
    sql_on: ${customers.customer_id} = ${orders.customer_id} ;;
    relationship: one_to_many
  }
}
```

Right side: the view joined to
(*the name next to “join:”*)

The order of the `sql_on` clause does not matter in the relationship

Identifying the Correct Join Relationship

Test each possible relationship in a sentence:

✗ one	customer	can relate to	only one	order
✓ one	customer	can relate to	many	orders
✗ many	customers	can relate to	only one	order
✗ many	customers	can relate to	many (possibly same)	orders

Results of Incorrect Join Relationships

Using this example

```
explore: customers {  
  join: orders {  
    sql_on: ${customers.customer_id} = ${orders.customer_id} ;;  
    relationship: one_to_one  
  }  
}
```

The measures from the Orders table are correct, while the Measures from the fanned out Customers Table are not

▼ DATA	RESULTS	SQL	Calculations	Row Limit 500	<input type="checkbox"/> Totals
Customers Count ▼	Customers Total Visits	Orders Count	Orders Total Amount	250	
1	4	10	4		

Let's Fix the Previous Example

Identify Primary Keys and correctly specify the relationship parameter

```
explore: customers {
  join: orders {
    sql_on: ${customers.customer_id} = ${orders.customer_id} ;;
    relationship: one_to_many
  }
}
```

All measures in all Views calculate correctly

▼ DATA	RESULTS	SQL	Calculations	Row Limit 500	<input type="checkbox"/> Totals
Customers Count ▾	Customers Total Visits	Orders Count	Orders Total Amount	250	
1	3	8	4		

How Does All of This Work? - Counts

Counts are simple: Looker does a count distinct of the primary keys

```
SELECT
    COUNT(orders.order_id ) AS "orders.count",
    COUNT(DISTINCT customers.customer_id ) AS "customers.count"
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id
```

How Does All of This Work? - Sums & Averages

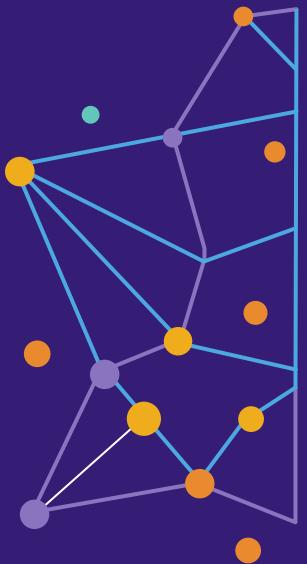
Sums & averages are more complex, but function the same way

```
SELECT
    COALESCE(COALESCE(CAST( ( SUM(DISTINCT (CAST(FLOOR(COALESCE(customers.visits ,0)*(1000000*1.0)) AS DECIMAL(38,0))) + CAST(STRTOLEFT(MD5(CAST(customers.customer_id AS VARCHAR)),15),16) AS DECIMAL(38,0))* 1.0e8 + CAST(STRTOLRIGHT(MD5(CAST(customers.customer_id AS VARCHAR)),15),16) AS DECIMAL(38,0)) ) - SUM(DISTINCT CAST(STRTOLEFT(MD5(CAST(customers.customer_id AS VARCHAR)),15),16) AS DECIMAL(38,0))* 1.0e8 + CAST(STRTOLRIGHT(MD5(CAST(customers.customer_id AS VARCHAR)),15),16) AS DECIMAL(38,0))) ) AS DOUBLE PRECISION) / CAST((1000000*1.0) AS DOUBLE PRECISION),
    0) AS "customers.total_visits",
    COALESCE(SUM(orders.amount ), 0) AS "orders.total_amount"
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id
LIMIT 500
```

How Does All of This Work?

$\text{SUM}(\text{DISTINCT } (\text{visits} + \text{MD5}(\text{customer_id}))) - \text{SUM}(\text{DISTINCT } (\text{MD5}(\text{customer_id})))$

customer_id	first_name	last_name	visits	order_id	amount	customer_id
1	Amelia	Earhart	2	1	25.00	1
1	Amelia	Earhart	2	2	50.00	1
2	Charles	Lindbergh	2	3	75.00	2
3	Wilbur	Wright	4	4	100.00	3



Filtering Explores

Filtering Explores: Learning Objectives

You will understand the most commonly utilized options for applying default filters to an Explore

- `sql_always_where` and `sql_always_having`
- `always_filter`
- `Conditionally_filter`

You will recognize common use cases for adding Explore filters

sql_always_where & sql_always_having

WHAT: A filter within an Explore that cannot be changed by users

- No indicator in the UI (*unless user looks at generated SQL*)
- Applies to
 - Users queries
 - Looks and dashboards
 - Scheduled content
 - Embeds
- Written in SQL and can use LookML substitutions like \${field_name}

WHY: You should filter out certain values of the Explore for ALL USERS (such as test_user, internal orders, etc.)

sql_always_where & sql_always_having

```
explore: order_items {  
    sql_always_where: ${order_items.created_date} >= '2012-01-01';;
```

```
    join: users {  
        type: left_outer  
        sql_on: ${order_items.user_id} = ${users.id}  
        relationship: many_to_one  
    }
```

SELECT
 users.city AS "users.city",
 COUNT(*) AS "order_items.count"
FROM public.order_items AS order_items
LEFT JOIN public.users AS users ON order_items.user_id = users.id

```
WHERE (DATE(order_items.created_at )) >= '2012-01-01'
```

```
explore: order_items {  
    sql_always_having: ${order_count} > 10 ;;
```

```
    join: users {  
        type: left_outer  
        sql_on: ${order_items.user_id} = ${users.id}  
        relationship: many_to_one  
    }
```

SELECT
 users.city AS "users.city",
 COUNT(*) AS "order_items.count"
FROM public.order_items AS order_items
LEFT JOIN public.users AS users ON order_items.user_id = users.id

GROUP BY 1
HAVING (COUNT(DISTINCT order_items.order_id)) > 10

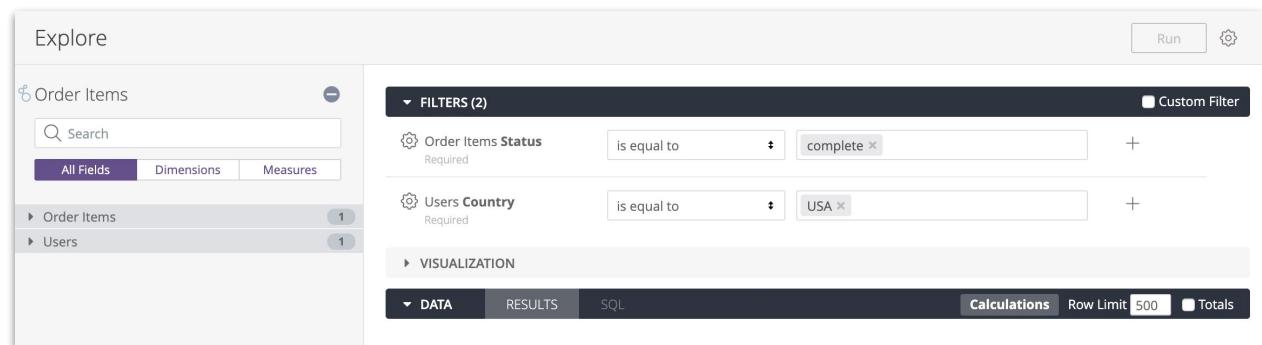
always_filter

WHAT: Required filter fields that are automatically added to the Explore

- Filter value can be changed but the filter itself cannot be removed
- Default values are written as Looker Filter Expressions

WHY: Prompts users to leverage appropriate filters when querying data

```
explore: order_items {  
  always_filter: {  
    filters: {  
      field: status  
      value: "complete"  
    }  
  
    filters: {  
      field: users.country  
      value: "USA"  
    }  
  }  
}
```



The screenshot shows the Looker Explore interface with the following details:

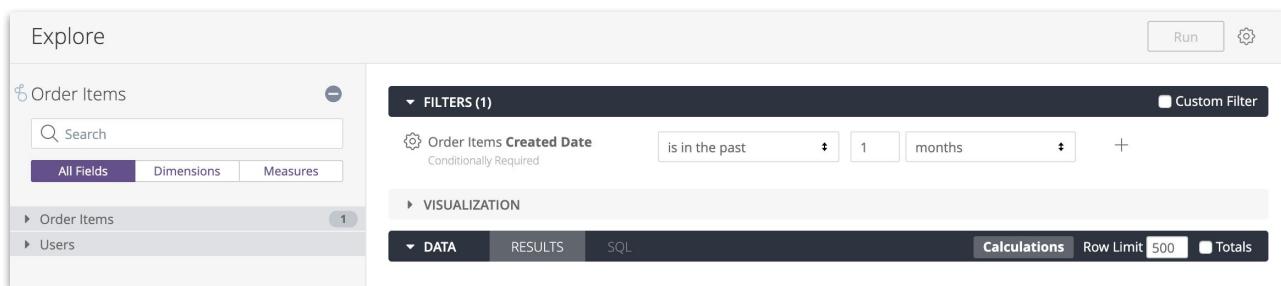
- Explore View:** The title is "Order Items". The sidebar shows "All Fields" selected under "Dimensions".
- FILTERS (2):**
 - Order Items Status:** Set to "is equal to" and "complete".
 - Users Country:** Set to "is equal to" and "USA".
- DATA:** Row Limit is set to 500.
- RESULTS:** Totals are included.

conditionally_filter

WHAT: A default filter that can be removed if at least one of the specified alternative filter fields is selected

WHY: Typically used to prevent users from accidentally creating very large queries that may be too expensive to run on your database

```
explore: order_items {  
  conditionally_filter: {  
    filters: {  
      field: created_date  
      value: "1 month"  
    }  
  
    unless: [users.id, users.state]  
  }  
}
```



The screenshot shows the Looker interface with the 'Explore' page for 'Order Items'. The sidebar on the left lists 'Order Items' and 'Users'. In the main area, there is a single filter named 'Order Items Created Date' under the 'FILTERS (1)' section. The filter is set to 'Conditionally Required' and has the condition 'is in the past 1 months'. At the bottom of the screen, there is a navigation bar with tabs for 'DATA', 'RESULTS', and 'SQL', along with other options like 'Calculations', 'Row Limit 500', and 'Totals'.

EXERCISE TIME!

GUIDED EXERCISE 6: Working with Filters

Let's add a filter to the Order Items Explore that always excludes returned items.

Then we'll add a second filter that always limits the results to only those in which the total sales amount is > \$200

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 6: Working with Filters

Task 2: Add a filter to the Order Items Explore that forces this Explore to always include only orders with a status of “Complete”.

Then add a second filter that always limits the results to only show values in which the order item count is greater than 5

[Level 1](#)

[Level 2](#)

[Level 3](#)

GUIDED EXERCISE 6: Working with Filters

Let's add a filter (in the UI) to the Users Explore to only show users that have created orders before today.

Then, we'll add a filter to the Order Items Explore to only include orders created within the past 2 years unless a filter is applied to the users.id field

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 6: Working with Filters

Task 4: Add a filter to the Order Items Explore that defaults to only show orders that have been created within the last 30 days but the user can change the filter.

Then, add a filter to the Users Explore to only include users created within the past 90 days unless a filter is applied to the users.id or users.state fields.

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 6: Task 1 Level 1

Task 1: Add a filter to the Order Items Explore that always excludes returned items. Then add a second filter that always limits the results to only those in which the total sales amount is > \$200

sql_always_where: \${_____}.\${____}_date} IS NULL ;;

sql_always_having: \${_____}.\${____}_sales} > 200 ;;

EXERCISE 6: Task 1 Level 2

Task 1: Add a filter to the Order Items Explore that always excludes returned items. Then add a second filter that always limits the results to only those in which the total sales amount is > \$200

-----: \${-----.-----} IS NULL ;;

-----: \${-----.-----} > 200 ;;

EXERCISE 6: Task 1 Level 3

Task 1: Add a filter to the Order Items Explore that always excludes returned items. Then add a second filter that always limits the results to only those in which the total sales amount is > \$200

EXERCISE 6: Task 2 Level 1

Task 2: Add a filter to the Order Items Explore that forces this Explore to always include only orders with a Status of “complete”. Then add a second filter that always limits the results to only show values in which the order item count is greater than 5

```
sql_always_where: ${-----.-----date} IS NULL ;;
```

```
sql_always_having: ${-----.-----count} > 5 ;;
```

EXERCISE 6: Task 2 Level 2

Task 2: Add a filter to the Order Items Explore that forces this Explore to always include only orders with a Status of “complete”. Then add a second filter that always limits the results to only show values in which the order item count is greater than 5

-----: \${-----.-----} = '-----' ;;

-----: \${-----.-----} > ----- ;;

EXERCISE 6: Task 2 Level 3

Task 2: Add a filter to the Order Items Explore that forces this Explore to always include only orders with a Status of “complete”. Then add a second filter that always limits the results to only show values in which the order item count is greater than 5

EXERCISE 6: Task 3 Level 1

Task 3: Add a filter to the Users Explore to only show users that have created orders before today. Then, add a filter to the Order Items Explore to only include orders created within the past 2 years unless a filter is applied to the users.id field

```
always_filter: {  
    filters: {  
        field: _____ . _____ date  
        value: "_____ - _____"  
    }  
}
```

```
conditionally_filter: {  
    filters: {  
        field: _____ . _____  
        value: "_____ - _____"  
    }  
    unless: [_____ . ____]  
}
```

EXERCISE 6: Task 3 Level 2

Task 3: Add a filter the Users Explore to only show users that have created orders before today. Then, add a filter to the Order Items Explore to only include orders created within the past 2 years unless a filter is applied to the users.id field

```
-----: {  
  -----: {  
    -----: -----.  
    -----: "-----"  
  }  
}
```

```
-----: {  
  -----: {  
    -----: -----.  
    -----: "-----"  
  }  
  -----: [-----]  
}
```

EXERCISE 6: Task 3 Level 3

Task 3: Add a filter to the Users Explore to only show users that have created orders before today. Then, add a filter to the Order Items Explore to only include orders created within the past 2 years unless a filter is applied to the users.id field

EXERCISE 6: Task 4 Level 1

Task 4: Add a filter to the Order Items Explore to only show orders that have been created within the last 30 days. Then, add a filter to the Users Explore to only include users created within the past 90 days unless a filter is applied to the users.id or users.state fields

```
always_filter: {  
    filters: {  
        field: _____._____.date  
        value: "____ - __ - ____"  
    }  
}
```

```
conditionally_filter: {  
    filters: {  
        field: _____._____  
        value: "____ - __ - ____"  
    }  
    unless: [_____.__, _____.___.  
    ]  
}
```

EXERCISE 6: Task 4 Level 2

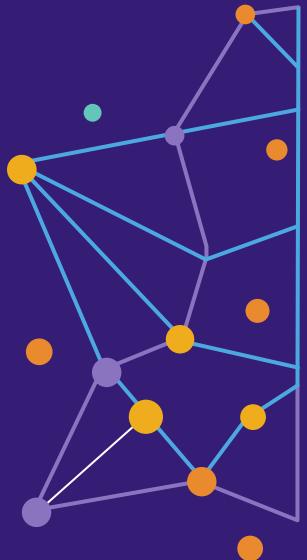
Task 4: Add a filter the Order Items Explore to only show orders that have been created within the last 30 days. Then, add a filter to the Users Explore to only include users created within the past 90 days unless a filter is applied to the users.id or users.state fields

```
-----: {  
  -----: {  
    -----: -----.  
    -----: "-----"  
  }  
}
```

```
-----: {  
  -----: {  
    -----: -----.  
    -----: "-----"  
  }  
  -----: [-----., -----.]  
}
```

EXERCISE 6: Task 4 Level 3

Task 4: Add a filter to the Order Items Explore to only show orders that have been created within the last 30 days. Then, add a filter to the Users Explore to only include users created within the past 90 days unless a filter is applied to the users.id or users.state fields



Caching & Datagroups

Caching in Looker

Using cached results of prior queries helps to reduce database load

Caching policies can be set up in Looker using datagroups

How Caching Works in Looker

A query is run by a user and cached (cache results are stored in an encrypted file on the Looker instance)

How Caching Works in Looker

For any new queries, the cache is checked to see if the same query was previously run before running the query against the database

- If the query is not found, Looker runs the query against the database and caches the new result
- If the query is found and the results are still valid then Looker uses the cached results
- If the query is found and the results are no longer valid, Looker runs the query against the database and caches the new result

Implementing Caching in Looker

These caching policies can then be applied to various Looker objects:

- At the **model** or **Explore** level: use persist_with parameter to specify which Explores use each policy for clearing the query cache
- In a **PDT** definition: use datagroup_trigger to specify which policy to use in rebuilding the PDT
- On **Looks** and **dashboards**: build schedules that trigger based on datagroups to cause content to run and send immediately after the cache has been invalidated, thus warming the cache with the latest results

Datagroups

WHAT: Named **caching policies** within Looker that can be applied to models, Explores, or Persistent Derived Tables

WHY: Integrate Looker more closely with ETL processes or guarantee a refreshed cache

- Define one or more datagroup parameters at the model level
- Different caching policies require separate datagroup definitions

Configuring Datagroups

Caching policy parameters:

- `sql_trigger` parameter
 - Should be SQL query that returns one row with one column
 - Typically will query a field that serves as a good indicator that the underlying data has been updated, such as a `max(date)` or will return a specific time of day
- `max_cache_age` to indicate the longest amount of time in which a query should be cached before being invalidated

Only one of these parameters is required, but both are recommended

```
datagroup: daily_etl {  
    max_cache_age: "24 hours"  
    sql_trigger: SELECT max(id) FROM my tablename ;;  
}
```

Applying Datagroups to Query Results

A datagroup's caching policy can be applied to one, some or all Explores in a model.

- As a default for all Explores in a model: use the `persist_with` parameter at the model level and specify the name of the datagroup
- For a specific Explore: use the `persist_with` parameter in that Explore's definition and specify the name of the datagroup
- For a group of Explores: use the `persist_with` parameter in each Explore's definition and specify the name of the same datagroup

Applying Datagroups to Query Results

Datagroups can also be used to add persistence to derived tables,
which will be covered in the next section

ecommerce_data.model ▾

```
1 connection: "thelook_events"
2 persist_with: order_items
3
```

```
explore: order_items {
  persist_with: order_items
  join: users {
    type: left_outer
    sql_on: ${order_items.user_id} = ${users.id} ;;
    relationship: many_to_one
  }
}
```

EXERCISE TIME!

EXERCISE 7: Working with Datagroups

Task 1: Set up a default datagroup that triggers at midnight each day with the date changes. Ensure that the cache age will never exceed 24 hours. Apply this datagroup to the Users Explore

[Level 1](#)

[Level 2](#)

[Level 3](#)

Task 2: Set up an order_items datagroup that triggers any time the maximum created_at timestamp in the order_items table changes. Ensure that the cache age will never exceed 4 hours. Apply this data group to the Order Items Explore

[Level 1](#)

[Level 2](#)

[Level 3](#)

EXERCISE 7: Task 1 Level 1

Task 1: Set up a default datagroup that triggers at midnight each day with the date changes. Ensure that the cache age will never exceed 24 hours. Apply this data group to the Users Explore

```
datagroup: _____ {  
    sql_trigger: select _____ ;;  
    max_cache_age: "__ ____"  
}
```

```
explore: _____ {  
    persist_with: _____  
}
```

EXERCISE 7: Task 1 Level 2

Task 1: Set up a default datagroup that triggers at midnight each day with the date changes. Ensure that the cache age will never exceed 24 hours. Apply this data group to the Users Explore

```
-----: ----- {  
    -----: ----- ----- ;;  
    -----: "-----"  
}  
  
-----: ----- {  
    -----: -----  
}
```

EXERCISE 7: Task 1 Level 3

Task 1: Set up a default datagroup that triggers at midnight each day with the date changes. Ensure that the cache age will never exceed 24 hours. Apply this data group to the Users Explore

EXERCISE 7: Task 2 Level 1

Task 2: Set up an order_items datagroup that triggers any time the maximum created_at timestamp in the order_items table changes. Ensure that the cache age will never exceed 4 hours. Apply this data group to the Order Items Explore

```
datagroup: _____ {  
    sql_trigger: select max(_____) from _____ ;;  
    max_cache_age: "_ ____"  
}
```

```
explore: _____ {  
    persist_with: _____  
}
```

EXERCISE 7: Task 2 Level 2

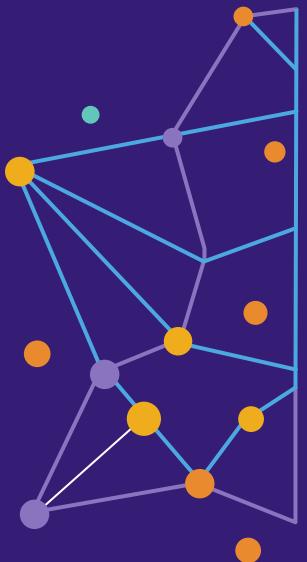
Task 2: Set up an order_items datagroup that triggers any time the maximum created_at timestamp in the order_items table changes. Ensure that the cache age will never exceed 4 hours. Apply this data group to the Order Items Explore

```
-----: ----- {
  -----: ----- (-----) ----- ;;
  -----: "-----"
}

-----: ----- {
  -----: -----
}
```

EXERCISE 7: Task 2 Level 3

Task 2: Set up an order_items datagroup that triggers any time the maximum created_at timestamp in the order_items table changes. Ensure that the cache age will never exceed 4 hours. Apply this data group to the Order Items Explore

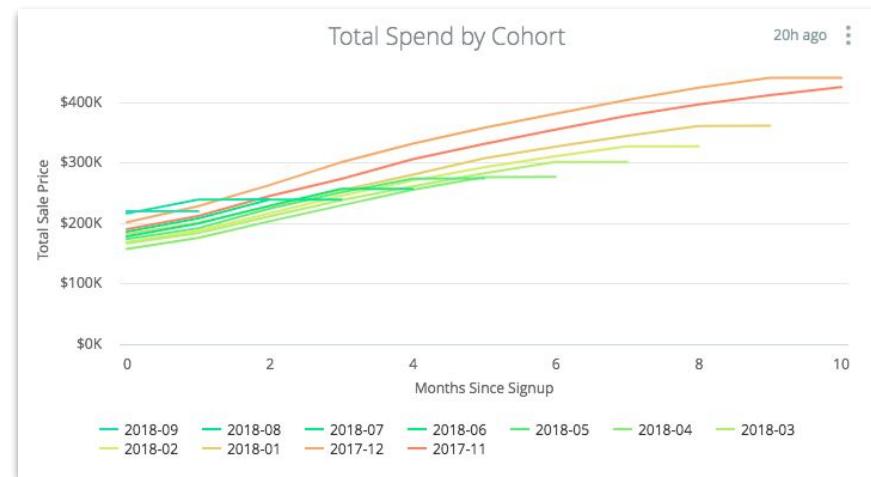


Derived Tables

Joining Views to Explores is great but...

You may want more flexibility to **restructure data** and **define complex query logic** to do cool things like...

- **Cohort** and **retention** analysis
- **Sessionalize** event data
- **Pre-aggregate** fields to aggregate aggregates



Derived Tables Can Help!

Derived Tables provide the flexibility to restructure data from right within Looker

- Utilize window functions to convert detailed event data into organized sessions based on activity timeframes
- Aggregate order information by customer to understand customer value and loyalty

Derived Tables Can Help!

Derived Tables provide the flexibility to restructure data from right within Looker

- Join together order and product information from different tables in the underlying database to do market basket analysis
- Create a restructured user table that contains a record for every user and month combination so that periods of activity and inactivity are represented for retention and engagement analysis

What Are Derived Tables?

Manually written query whose result set can be queried like a regular database table

Integrated into Looker as views

Can be joined into Explores just like standard views

They can be ephemeral or written into the database (PDT)

```
view: repeat_purchase_facts {
  derived_table: {
    sql: SELECT
      order_items.order_id
      , COUNT(DISTINCT repeat_order_items.id) AS number_subsequent_orders
      , MIN(repeat_order_items.created_at) AS next_order_date
      , MIN(repeat_order_items.order_id) AS next_order_id
    FROM ecomm.order_items
    LEFT JOIN ecomm.order_items repeat_order_items
      ON order_items.user_id = repeat_order_items.user_id
      AND order_items.created_at < repeat_order_items.created_at
    GROUP BY 1
  }
}
```

Two Types of Derived Tables

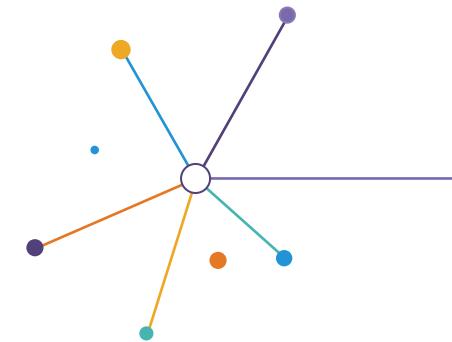
SQL Derived Table

- Easy to learn
- Easy to understand
- Uses complex joins, calculations and functions

Native Derived Table

- Maximum code reusability
- Easier to maintain
- Easier to read and understand

SQL Derived Tables



Understanding Orders

Find **average number** of items purchased per order, by age group?

Current Structure
Order Items
order_item_id (PK)
order_id
user_id

We can calculate the number of items (i.e. a count)

We cannot do an average of this count

(this would require SQL subquery)

Understanding Orders

Find **average number** of items purchased per order, by age group?

What We Need
Orders
order_id (PK)
user_id
item_count

We could calculate the average number of items (essentially `average(item_count)`)

We could look at this average by age group in the Explore to answer the question

Step 1 - Build / Test Query in SQL Runner

You are in **Development Mode**.

Exit Development Mode

looker

Browse ▾ Explore ▾ Develop ▾ Admin ▾

SQL Runner

Run

Database Model History

CONNECTION events_ecommerce

Search this connection

SCHEMA public

TABLES aggregation distribution_centers etl_jobs events

ORDER_ITEMS

- created_at
- delivered_at
- # id
- # inventory_item_id
- # order_id
- returned_at
- # sale_price
- shipped_at
- status

▼ SQL QUERY

Amazon Redshift

```
SELECT
    order_items.order_id
    ,order_items.user_id
    ,count(*) AS order_item_count
    ,sum(sale_price) AS order_total
FROM order_items
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 10
```

▼ RESULTS

order_id	user_id	order_item_count	order_total
1 16172	17132	4	97.24000072479248
2 21965	23311	4	212.48999977111816
3 21291	22587	4	318.8999996185303
4 2952	3127	4	155.06999969482422
5 29296	31144	4	56.43000030517578
6 13471	14285	4	258
7 13536	14352	4	329.9599952697754
8 142	152	4	250.6000022881836
9 22130	23494	4	138.0799987003174
10 334	355	4	226.2100009918213

Step 2 - Add Query into your LookML Project

You are in **Development Mode**.

looker

Browse Explore Develop Admin

SQL Runner

Run

Download... Add to Project... Get Derived Table LookML... Explore Clear

SQL QUERY

```
SELECT
    order_items.order_id
    ,order_items.user_id
    ,count(*) AS order_item_count
    ,sum(sale_price) AS order_total
FROM order_items
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 10
```

RESULTS

order_id	user_id	order_item_count	order_total
1 16172	17132	4	97.24000072479248
2 21965	23311	4	212.48999977111816
3 21291	22587	4	318.8999996185303
4 2952	3127	4	155.06999969482422
5 29296	31144	4	56.43000030517578
6 13471	14285	4	258
7 13536	14352	4	329.9599952697754
8 142	152	4	250.60000228881836
9 22130	23494	4	138.0799987003174
10 334	355	4	226.2100009918213

Step 3 - Name your New View

You are in Development Mode. [Exit Development Mode](#)

Looker [Browse](#) [Explore](#) [Develop](#) [Admin](#) [Run](#) [?](#) [@](#)

SQL Runner

Add to Project

Project advanced_data_analyst_bootcamp

View Name order_facts

Cancel Add

#	order_id	status	order_qty	order_total
1	13471	shipped_at	4	56.4300030517578
2	13536	shipped_at	4	258
3	142	shipped_at	4	329.9599952697754
4	22130	shipped_at	4	250.6000228881836
5	334	shipped_at	4	138.07999897003174
6	29296	status	4	226.2100009918213
7	13471	status	4	14285
8	13536	status	4	14352
9	22130	status	4	23494
10	334	status	4	355

Step 4 - Review & Clean Up Your View

Looker creates a new view with the SQL Runner query and will write dimensions for every field as well as a count measure:

The screenshot shows the Looker Development Mode interface. The top navigation bar includes 'Browse', 'Explore', 'Develop', 'Admin', and 'Exit Development Mode'. The main area displays a LookML code editor for a 'order_facts' view. The code defines a derived table named 'order_facts' with an SQL SELECT statement that joins 'order_items' and 'order_items.user_id', counts items, and sums sale price. It also includes measures for count and dimensions for order_id, user_id, and order_item_count. A sidebar on the right provides 'Quick Help' for the 'sql' block, explaining it specifies the SQL SELECT statement for generating a derived table.

```
i 1 ✓ view: order_facts {
2   derived_table: {
3     sql: SELECT
4       order_items.order_id
5       ,order_items.user_id
6       ,count(*) AS order_item_count
7       ,sum(sale_price) AS order_total
8     FROM order_items
9     GROUP BY 1, 2
10    ORDER BY 3 DESC
11    LIMIT 10
12  }
13
14
15  measure: count {
16    type: count
17    drill_fields: [detail*]
18  }
19
20  dimension: order_id {
21    type: number
22    sql: ${TABLE}.order_id ;;
23  }
24
25  dimension: user_id {
26    type: number
27    sql: ${TABLE}.user_id ;;
28  }
29
30  dimension: order_item_count {
31    type: number
32    sql: ${TABLE}.order_item_count ;;
33 }
```

EXERCISE TIME!

EXERCISE 1: Understanding Users

What is the Average Lifetime Value and Average Lifetime Order Count for customers in each State?

EXERCISE 1: Understanding Users

What is the Average Lifetime Value and Average Lifetime Order Count for customers in each State?

SELECT

```
order_items.user_id AS user_id  
,COUNT(distinct order_items.order_id) AS lifetime_order_count  
,SUM(order_items.sale_price) AS lifetime_revenue  
,MIN(order_items.created_at) AS first_order_date (optional)  
,MAX(order_items.created_at) AS latest_order_date (optional)  
FROM order_items  
GROUP BY user_id
```

EXERCISE 2: Product Inventory Analysis

What is the percentage of inventory sold by SKU? How can we then find the average percentage of inventory sold by Brand or Category?

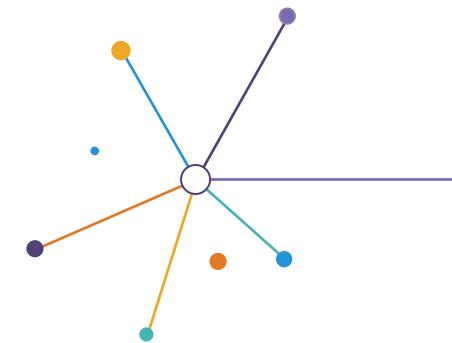
NOTE: percentage of inventory sold = cost of goods sold / total cost of inventory

EXERCISE 2: Product Inventory Analysis

What is the percentage of inventory sold by SKU? How can we then find the average percentage of inventory sold by Brand or Category?

```
SELECT
    product_sku AS product_sku
    ,SUM(cost) AS total_cost
    ,SUM(CASE WHEN sold_at is not null THEN cost ELSE NULL END) AS
cost_of_goods_sold
FROM public.inventory_items
GROUP BY 1
```

Native **Derived Tables**



Maximizing Code Reusability

The SQL for the User Facts table just built included the following definitions:

- COUNT(distinct
order_items.order_id) as
lifetime_order_count
- SUM(order_items.sale_price)
as lifetime_revenue

```
measure: order_count {  
  description: "A count of unique orders"  
  type: count_distinct  
  sql: ${order_id} ;;  
}
```

```
measure: total_revenue {  
  type: sum  
  value_format_name: usd  
  sql: ${sale_price} ;;  
  drill_fields: [detail*]  
}
```

BUT WAIT! These measures were already defined within the LookML in the order_items view

Native Derived Tables

How can we take advantage of dimensions and measures that have already been defined within the LookML?

Native Derived Tables are derived tables that perform the same function as a written SQL query, but are expressed natively in the LookML language

- Easier to read and understand when modeling data
- Enables code to be reused
- More maintainable since physical database references are minimized

Step 1 - Build Your Query

Explore

Order Items

FILTERS

VISUALIZATION

DATA RESULTS SQL Calculations Row Limit 500 Totals

⚠ Row limit reached. Results may be incomplete.

	Users ID	Order Items Total Revenue	Order Items Order Item Count
1	10994	\$2,552.02	23
2	81435	\$2,211.05	9
3	2153	\$2,126.75	17
4	3278	\$2,051.05	26
5	78788	\$1,966.27	23
6	40626	\$1,960.98	7
7	12296	\$1,934.62	24
8	18483	\$1,909.82	21
9	13919	\$1,890.74	22
10	12527	\$1,866.46	22
11	1469	\$1,857.47	17
12	16637	\$1,842.27	24
13	224	\$1,798.58	17
14	8035	\$1,790.74	21
15	5947	\$1,733.29	28
16	51	\$1,730.62	22

PIVOT FILTER \$

Step 2 - Obtain LookML for your Query

Explore

Order Items

All Fields Dimensions Measures

Order Items

Order Item Count FILTER

Total Margin

Total Profit

Total Revenue FILTER \$

Products

Top 5 Brands (Select Metric to Rank By)

Users

DIMENSIONS

Age

City

Country

Created Date

Email

First Name

Gender

ID PIVOT FILTER \$

FILTERS

VISUALIZATION

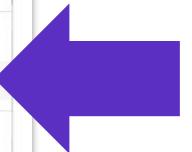
DATA RESULTS SQL

Row limit reached. Results may be incomplete.

Users ID	Order Items Total Revenue	Order Items Count
1	\$2,552.02	10994
2	\$2,211.05	81435
3	\$2,126.75	2153
4	\$2,051.05	3278
5	\$1,966.27	78788
6	\$1,960.98	40626
7	\$1,934.62	12296
8	\$1,909.82	18483
9	\$1,890.74	13919
10	\$1,866.46	12527
11	\$1,857.47	1469
12	\$1,842.27	16637
13	\$1,798.58	224
14	\$1,790.74	8035
15	\$1,733.29	5947
16	\$1,730.62	51

500 rows · 5.1s · 2m ago Run

Save as a Look... Save to Dashboard... Download... Send... Save & Schedule... Share... Get Dashboard LookML... Get Derived Table LookML... Remove Fields & Filters Clear Cache & Refresh



Step 3 - Copy the LookML

The screenshot shows the Looker interface with an 'Explore' view titled 'Order Items'. The left sidebar lists dimensions like 'Order Item Count', 'Total Margin', 'Total Profit', and 'Total Revenue'. A modal window titled 'Native Derived Table LookML' is open in the center, containing the following LookML code:

```
view: add_a_unique_name_1510179528 {  
  derived_table: {  
    explore_source: order_items {  
      column: id { field: users.id }  
      column: total_revenue {}  
      column: order_item_count {}  
    }  
  }  
  dimension: id {  
    type: number  
  }  
  dimension: total_revenue {  
    value_format: "$#,##0.00"  
    type: number  
  }  
  dimension: order_item_count {  
    type: number  
  }  
}
```

The modal has a close button ('X') in the top right corner. To the right of the modal is a context menu with the following options and counts:

- Save as a Look... (145)
- Save to Dashboard... (138)
- Download... (136)
- Send... (135)
- Save & Schedule... (135)
- Share... (96)
- Get Dashboard LookML... (28)
- Get Derived Table LookML... (28)
- Remove Fields & Filters (36K)
- Clear Cache & Refresh (22)

Step 4 - Create a New View & Paste in LookML

Paste the LookML into a fresh view file within the appropriate project

Change the auto-generated view name to something representative of the data set

Include model file with the Explore being used

```
user_facts_ndt ▾
1 include: "training_advanced.model.lkml"
2
3 view: user_facts {
4   derived_table: {
5     explore_source: order_items {
6       column: id { field: users.id }
7       column: total_revenue {}
8       column: order_item_count {}
9     }
10   }
11   dimension: id {
12     type: number
13   }
14   dimension: total_revenue {
15     value_format: "$#,##0.00"
16     type: number
17   }
18   dimension: order_item_count {
19     type: number
20   }
21 }
```

Native Derived Table Parameters

`explore_source`: the Explore defined within Looker that contains the field and join definitions required for the desired query

`column`: specifies an output column for the derived table

- often paired with a “field” parameter to link the new table column back to the appropriate underlying column
- can be named differently from the underlying field referenced

`filters` can be used for applying filters to the derived table using the same syntax as a filtered measure

Native Derived Table Parameters

`derived_column`: specify one or more columns that don't exist in the Explore specified by the `explore_source` parameter

`bind_filters`: used for applying templated filters to the native derived table

`expression_custom_filter`: specify one or more custom filter expressions on an `explore_source` query

EXERCISE TIME!

EXERCISE 3: Native Derived Tables

What's the average number of items purchased per order and the average order value by User Location?

If you are feeling **ambitious**:

Add a derived column into the derived table that ranks all orders based on total revenue (so that users can easily filter to the top 50 orders)

EXERCISE 3: Native Derived Tables

What's the average number of items purchased per order and the average order value by User Location?

```
include: "advanced_data_analyst_bootcamp.model.lkml"
view: order_facts {
    derived_table: {
        explore_source: order_items {
            column: id { field: order_items.order_id }
            column: total_revenue {}
            column: order_item_count {}
            derived_column: order_revenue_rank {
                sql: rank() over(order by total_revenue desc) ;;
            } (optional)
        }
    }
} ...
```

EXERCISE 4: Native Derived Tables

Create an NDT which shows the average profit per order line item per Created Month

Hint: you will need to make use of a derived column!

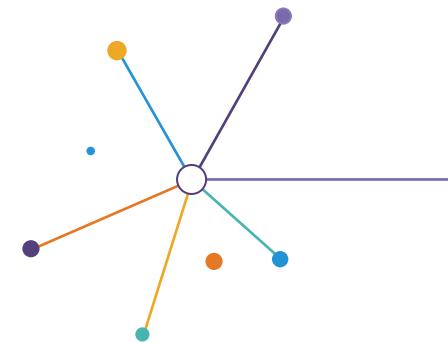
Challenge: Making use of Redshift's lag() window function, look back 12 months and pull in the average profit per order line. This will then allow for year-over-year analysis

EXERCISE 4: Native Derived Tables

Create an NDT which shows the average profit per order line item per Created Month

```
include: "advanced_data_analyst_bootcamp.model.lkml"
view: monthly_profitability_summary {
    derived_table: {
        explore_source: order_items {
            column: created_month{}
            column: total_profit {}
            column: order_item_count {}
            derived_column: total_profit_per_item {
                sql: ${total_profit}/${order_item_count};;
            }
            derived_column: total_profit_per_item_last_year {
                sql: lag(${total_profit}/${order_item_count},12)
OVER(order by created_month asc);;
            }
        }
    }
}
```

Persistent **Derived Tables**



Persistent Derived Tables

Add two (2) parameters to a derived table when persisting it:

1. Table refresh logic for table rebuilding
 - a. datagroup_trigger: triggered by some change that takes place in the underlying data as defined within a datagroup
 - b. sql_trigger_value: triggered by a change in the underlying data
 - c. persist_for: a set time period
2. Indexes
 - a. A single or multiple index for most databases
 - b. Sort key(s) and a distribution key for Redshift

Derived Tables Refresh Logic

Use a datagroup_trigger in the PDT's definition to rebuild the PDT based on change in the value returned by that query

```
GROUP BY user_id  
;;  
datagroup_trigger: order_items
```

Use a sql_trigger_value in the PDT's definition to rebuild the PDT based on a change in the value returned by that query

```
GROUP BY user_id  
;;  
sql_trigger_value: SELECT current_date ;;
```

Use persist_for to set the length of time the derived table should be stored before it is dropped from the database

```
GROUP BY user_id  
;;  
persist_for: "8 hours"
```

Indexing Derived Tables

- indexes: [field1, field2]
- Redshift
 - distribution: field1 (or distribution_style of "all" or "even")
 - Controls how the data is distributed across the nodes
 - Joining on distribution keys is most efficient
 - sortkeys: [field1, field2]
 - Controls the order the data is written to disk
 - Filtering on sort_keys is most efficient
 - indexes: [field1, field2]
 - This creates Interleaved Sort Keys
 - Works best for very large datasets

Ephemeral vs Persistent Derived Tables

Ephemeral derived tables will build at runtime as a temporary table (mysql) or via a SQL common table expression

```
WITH user_order_facts AS (SELECT
    order_items.user_id as user_id
    , COUNT(DISTINCT order_items.order_id) as lifetime_orders
    , SUM(order_items.sale_price) AS lifetime_revenue
    , MIN(NULLIF(order_items.created_at,0)) as first_order
    , MAX(NULLIF(order_items.created_at,0)) as latest_order
    , COUNT(DISTINCT DATE_TRUNC('month', NULLIF(order_items.created_at,0))) as number_of_distinct_months_with_orders
    , SUM(order_items.sale_price) AS order_value
  FROM order_items
  GROUP BY user_id
)
```

Derived Table SQL

```
SELECT
    user_order_facts.lifetime_orders AS "user_order_facts.lifetime_orders",
    COUNT(DISTINCT order_items.order_id ) AS "order_items.order_count"
  FROM public.order_items AS order_items
  LEFT JOIN user_order_facts ON user_order_facts.user_id = order_items.user_id
  GROUP BY 1
  ORDER BY 2 DESC
  LIMIT 500
```

Explore SQL

Ephemeral vs. Persistent Derived Tables

Persistent derived tables will be stored as physical tables within the database once built. Looker will then simply query those physical tables as needed. Looker will build separate PDTs in development and production modes

```
-- use existing user_order_facts in teach_scratch.LR$KDYI2NQM4DW046XHR9XQH_user_order_facts
SELECT
    user_order_facts.lifetime_orders AS "user_order_facts.lifetime_orders",
    COUNT(DISTINCT order_items.order_id) AS "order_items.order_count"
FROM public.order_items AS order_items
LEFT JOIN teach_scratch.LR$KDYI2NQM4DW046XHR9XQH_user_order_facts AS user_order_facts ON user_order_facts.user_id = order_items
    .user_id
GROUP BY 1
ORDER BY 2 DESC
LIMIT 500
```

EXERCISE TIME!

EXERCISE 5: Persistent Derived Tables

Persist the order_facts native derived table to refresh based on the order_items datagroup. This dataset is in Redshift, so don't forget to add distribution and sortkeys!

EXERCISE 5: Persistent Derived Tables

Persist the order_facts native derived table to refresh based on the order_items datagroup. This dataset is in Redshift, so don't forget to add distribution and sortkeys!

```
view: order_facts {  
    derived_table: {  
        distribution: "order_id"  
        sortkeys: ["order_id"]  
        datagroup_trigger: order_items  
  
        explore_source: order_items { ... }  
    }  
    ...  
}
```

Cascading Derived Tables

Reference one derived table in the definition of another using:

`${derived_table_name.SQL_TABLE_NAME}`

```
view: user_facts {  
  derived_table: {  
    sql: SELECT  
      order_items.user_id as user_id  
      , COUNT(DISTINCT order_items.order_id) as lifetime_orders  
      , SUM(order_items.sale_price) AS lifetime_revenue  
      , MIN(NULLIF(order_items.created_at,0)) as first_order  
      , MAX(NULLIF(order_items.created_at,0)) as latest_order  
      , SUM(order_items.sale_price) AS order_value  
    FROM order_items  
    GROUP BY user_id  
  };  
  datagroup_trigger: ecommerce_case_study_default_datagroup  
  sortkeys: ["user_id"]  
  distribution: "user_id"  
}
```

```
view: cascading_derived_table{  
  derived_table: {  
    sql:  
      SELECT  
        lifetime_orders  
        , AVG(lifetime_revenue) as average_revenue  
      FROM ${user_facts.SQL_TABLE_NAME}  
      GROUP BY lifetime_orders  
    ;;  
  }  
}
```

SUMMARY - Derived Tables

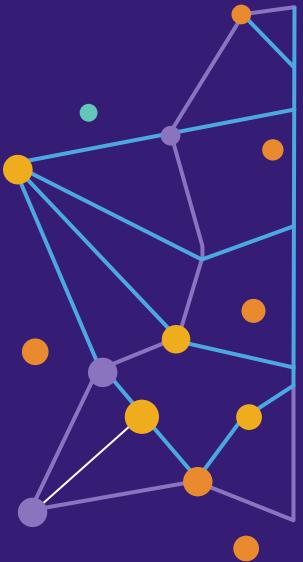
WHAT: Tables defined within Looker that do not exist in the database

- Two types of derived tables
 - Ephemeral: built at query time
 - Persisted: stored in the database
- Two ways to write derived tables
 - SQL
 - Native (using LookML)
- Defined within the LookML
- Referenced in the LookML just like any other table

SUMMARY - Derived Tables

WHY: Expand the sophistication of analyses

- Aggregate data to a different level of granularity (*ex: aggregate fact data*)
- Speed up performance (*ex: precompute joins*)
- Write custom SQL for advanced use cases (*ex: utilize window functions*)



Introduction to Looker Administration

Intro to Looker Admin - Learning Objectives

Understand how to add and manage users within Looker

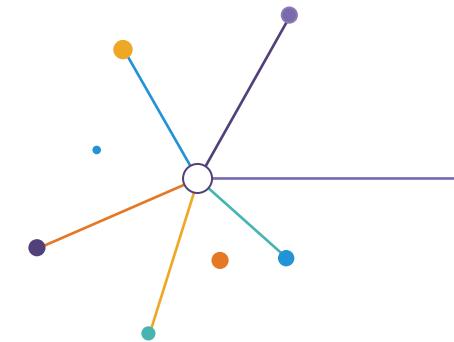
- Data Access
- Feature Access
- Content Access

Set up and apply user attributes

Identify ways of monitoring PDT and query performance

Be aware of settings and options available within the Admin panel

Managing **Users & Groups**



Adding New Users

Go to the Admin tab

Click on “Users”

Select “Add Users”

The screenshot shows the Looker Admin interface with the following details:

- Header:** Home, Browse, Explore, Develop, Admin, Search bar, and user count (11 active - 0 disabled).
- Left Sidebar:** General, Settings, Labs, Internal Help Resources, Localization, **Users** (selected), Users, Groups, Roles, Content Access, User Attributes, Custom Welcome Email.
- Table:** A list of users with columns: ID, Name, Credentials, Groups, Roles, and Actions (Sudo, Edit, Disable). The table contains 14 rows, each with a unique ID and specific role assignments.

ID	Name	Credentials	Groups	Roles	Actions
8			All Users	Admin	Sudo Edit Disable
2			All Users	Admin	Sudo Edit Disable
5			All Users	Admin	Sudo Edit Disable
7			All Users	Developer, User, Viewer	Sudo Edit Disable
4			All Users	Admin	Sudo Edit Disable
3			All Users	Admin	Edit
10			All Users, User	User	Sudo Edit Disable
11			All Users, User	User	Sudo Edit Disable
12			All Users	Developer	Sudo Edit Disable
13			All Users	Developer	Sudo Edit Disable
14			All Users	Developer	Sudo Edit Disable

Adding New Users

Enter email addresses of new users

Select Role(s) and Group(s) for the new users

The screenshot shows the 'Add Users' page in the Looker interface. On the left, there's a sidebar with navigation links: General, Settings, Labs, Internal Help Resources, Localization, Users, Groups, Roles, Content Access, and User Attributes. The main area has two tabs: 'Email addresses' and 'Send setup emails'. The 'Email addresses' tab contains a text input field with placeholder text 'Separate with commas, semicolons, or new lines.' The 'Send setup emails' tab has a checked checkbox. Below these are sections for 'Roles' and 'Groups'. The 'Roles' section lists Admin details, User details, Developer details, and Viewer details, each with an unchecked checkbox. A note says 'Use the Roles tab to create new roles with specific permissions and access'. The 'Groups' section lists User (roles: User) with an unchecked checkbox. A note says 'Use the Groups tab to create new groups'. At the bottom is a purple 'Add Users' button.

Controlling Access within Looker

Data Access: dictates which data sets a user has the ability to access via the implementation of model security

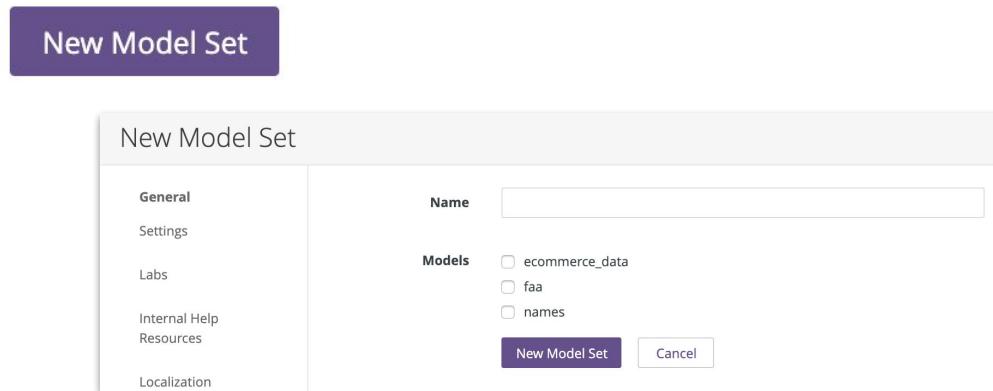
Feature Access: dictates what a user is able to do within Looker and what features are available

Content Access: dictates which spaces a user is able to access and, based on space permissions, what content a user is able to view and edit

Data Access within Looker

Data access security can be applied within Looker using Model Sets

- Defines what data and LookML fields a user or group can see
- Can include a single model or a group of models



Feature Access within Looker

Feature access can be defined within Looker using Permission Sets

- Defines what a user or group can do within Looker
- Some permissions are nested and require that other permissions be enabled
- Permissions are additive

New Permission Set

General	Name
Settings	<input type="text"/>
Labs	
Internal Help Resources	
Localization	
Users	Permissions
Users	<input checked="" type="checkbox"/> access_data <input type="checkbox"/> see_lookml_dashboards <input checked="" type="checkbox"/> see_looks <input type="checkbox"/> see_user_dashboards <input checked="" type="checkbox"/> explore <input checked="" type="checkbox"/> create_table_calculations <input type="checkbox"/> save_content <input type="checkbox"/> create_public_looks <input type="checkbox"/> download_with_limit <input type="checkbox"/> download_without_limit
Groups	
Roles	

Permission Set + Model Set = Role

Roles combine permission sets and model sets

- Users or groups are given roles within Looker
- Some feature access can be varied on a per-model set basis
- Other features are independent of model sets

Permission	Depends On	Type
explore	see_looks	Model Specific
create_table_calculations	explore	Instance Wide

Content Access within Looker

Content access controls which users can view or edit content within folders or change folder settings

- Access levels can be adjusted for each Folder
- Changes can be made within the Browse section of Looker or from the Content Access page in the Admin panel

Manage access to Finance Updates X

Who can access this folder?

The same users as the parent folder, Shared.

A custom list of users.

User Type	Action
Admins	Manage Access, Edit
All Users 14 users	Manage Access, Edit
Finance	View Add
Finance Team	0 users

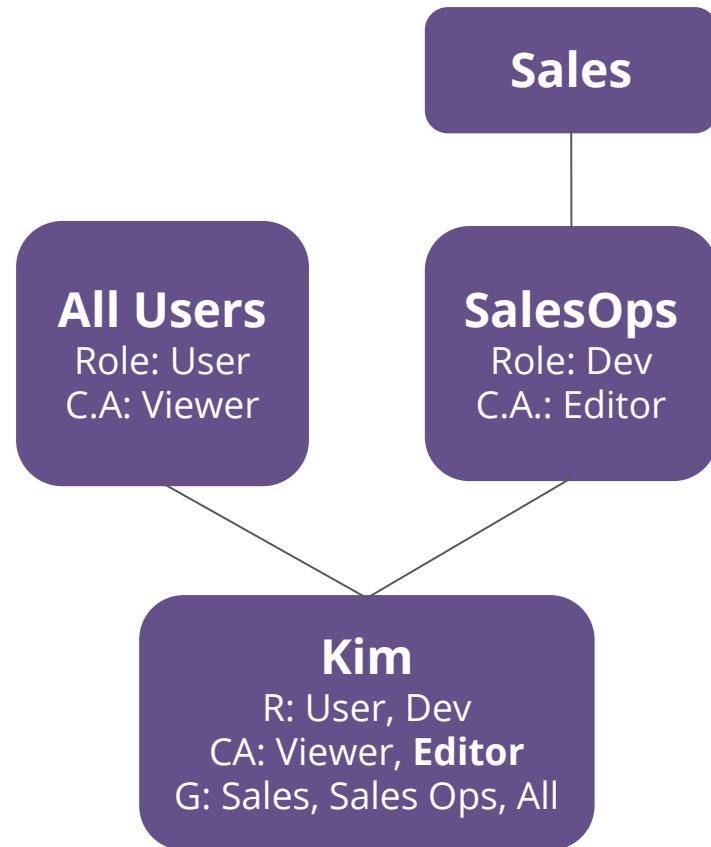
Cancel Save

Applying Groups

Users in multiple groups get the SUM of privileges of all the groups they're in

- Groups can be purely organizational
- Groups can be nested

Once Edit access is given, it can't be revoked in lower levels of the hierarchy



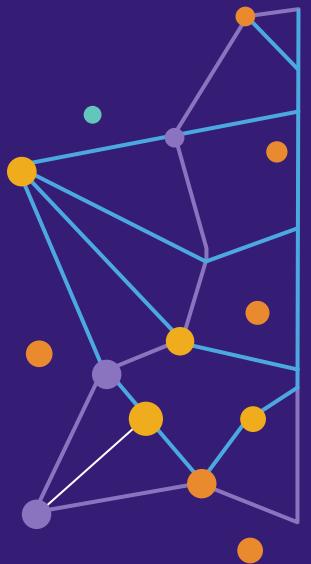
User & Group Management Best Practices

Have a plan for all three levels (data, features, and content), even if the current plan is to have no restrictions

Use Groups instead of Users to manage Roles and Content Access whenever possible

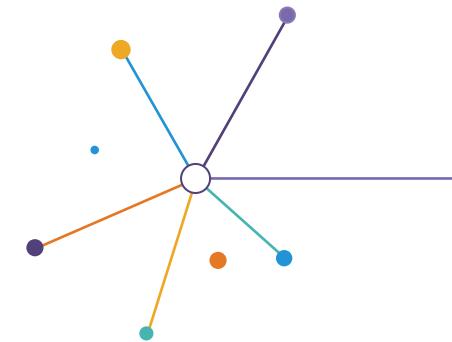
Most people should be Viewers of the Shared space if you want to lock any Spaces down

Make changes lower in the tree first, and then revoke access higher up



User Access

Access Filters **& User Attributes**



Access Filters

WHAT: Apply user-specific data filters to an Explore

- Needs to be used in conjunction with other settings in Looker
- Filter values applied are based on the individual user

WHY: Adding row-level security into an Explore

```
explore: users {  
  access_filter: {  
    field: email  
    user_attribute: email  
  }  
}
```



```
SELECT  
  users.state AS "users.state"  
FROM public.users AS users  
  
WHERE (users.email = '██████████')  
GROUP BY 1  
ORDER BY 1  
LIMIT 500
```

Applying Row-Level Data Security to Explores

Limit the data that a user can access based on who they are and which Explore they are using for their query

One or more access filters can be defined for any Explore

Best way of handling user-specific data restrictions

Applying Row-Level Data Security to Explores

ALL users, including Admins, MUST have a user attribute value set when an attribute is used for access filtering

- No value set = no data shown
- Assign wildcard characters to users who should have full access (%)

Inserts user attribute value as a condition in the SQL WHERE clause of a query

Setting Up an Access Filter on an Explore

Access filters are similar to other Explore filters that can be applied

```
explore: users {  
  access_filter: {  
    field: email  
    user_attribute: email  
  }  
}
```

Field affected by user attribute value

User attribute value inserted into query

How the Filter is Translated into the Explore

Insert filter into the SQL query behind the scenes in the WHERE clause

```
explore: users {  
  access_filter: {  
    field: email  
    user_attribute: email  
  }  
}
```



```
SELECT  
  users.email AS "users.email"  
FROM public.users AS users  
  
WHERE (users.email = 'maire@looker.com')  
GROUP BY 1  
ORDER BY 1  
LIMIT 500
```

User Attributes

User Attributes are arbitrary values that can be used in other parts of Looker to provide custom experiences for each user

Looker automatically includes some user attributes

- email
- name
- user_id

User Attributes

User Attributes are arbitrary values that can be used in other parts of Looker to provide custom experiences for each user

Other common user attributes include

- customer_id
- company_id
- region

EXERCISE 11: Looker Admin - User Attributes

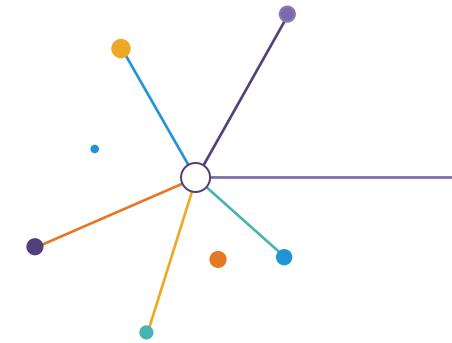
Task 1: A user attribute for State has been set up to help you limit data access to a specific state or set of states. Apply this user attribute to the Users Explore on the users.state field

All Levels

EXERCISE 11: Task 1 - All Levels

Task 1: A user attribute for State has been set up to help you limit data access to a specific state or set of states. Apply this user attribute to the Users Explore on the users.state field

Creating **User Attributes**



Creating User Attributes

Find and edit User Attributes within the “User Attributes” section of the Admin Panel

[Create new User Attributes](#) by clicking on the “Create User Attribute” button

The screenshot shows the Looker Admin Panel interface. At the top, there is a navigation bar with the Looker logo, 'Browse', 'Explore', and 'Develop' buttons. Below the navigation bar, the title 'User Attributes' is displayed. On the left side, there is a sidebar with several categories: 'General', 'Settings', 'Labs', 'Legacy Features', 'Users', 'Groups', 'Roles', 'Content Access', 'User Attributes' (which is currently selected and highlighted in grey), and 'Database'. To the right of the sidebar, there is a main content area. At the top of this area is a purple button labeled 'Create User Attribute'. Below this button is a dropdown menu labeled 'Name' with a small arrow icon. A list of user attribute names is shown, each preceded by a small blue square icon: 'allowed_brand', 'brand', 'brand2', 'can_see_ssn', 'country', 'database_connection', 'dev_schema', 'email', and 'email_view'. The 'email' item is also highlighted in grey, matching the 'User Attributes' category in the sidebar.

Creating User Attributes

Give user attribute a unique name

Select data type via drop-down

Set User Access to "None" (recommended) or "View" (Cannot use "edit" for an Access Filter)

Choose whether to hide values

Hit Save

New User Attribute: Brand

Definition

Name This is how you reference this attribute in Looker expressions and LookML. Attribute Names can only contain lower case letters, underscores, and numbers. They cannot start with a number.

Label This is the user-friendly name displayed in the app for lists and filters.

Data Type

User Access None View Edit If "None", non-admins will not be able to see the value of this attribute for themselves. "View" is required to use this attribute in query filters. If "Edit", the user will be able to set their own value of this attribute, so the user attribute will not be able to be used as an access filter.

Hide Values Yes No If "Yes", the value will be treated like a password, and once set, no one will be able to decrypt and view it. It will only be able to be used in passwords and secure webhook authentication. Once set to "Yes" for an attribute, cannot be unset.

Set a default value

Save

Creating User Attributes

Assign values to Groups or Users

For values assigned to multiple groups, drag the groups into order based on which value should take precedence should a user belong to multiple groups

Individual user values supercede group membership values

Assignment rules at the top of the list will override any rules below them for individuals in multiple groups. Drag rows to change ordering.

Group Name	Value	
BU Training	Levi's	Edit Value Remove Value
Developer Training	Allegra K, Carhartt, Nike	Edit Value Remove Value

+ Add Group

What Can I Do with User Attributes?

Apply access filters within Explores for data security

Control field access using access grants

Change many DB connection details like host, port, username & password, etc

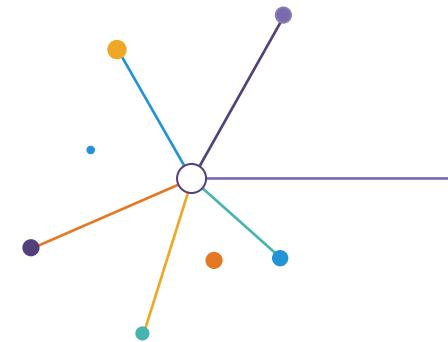
Configure data actions to send user-specific info along with data

Set filters on Explores, Looks and Dashboards customized by user

Run Look & Dashboard schedules to tailor results for email recipients

Leverage user attribute variables using Liquid in LookML

Field Permissions



Use LookML to Achieve Detailed Security

Looker has a robust set of security features including:

- Content Access
- Users and groups
- Roles, Permission Sets, Model Sets
- Row Level Security (Access Filters)

Use LookML to Achieve Detailed Security

Field Level Permissioning completes the picture by allowing for granular control over

- Individual Dimensions
- Individual Joins
- Individual Explores

Parameters Control Field-Level Permissioning

access_grant: An access policy that leverages user attributes and specific “allowed values” for the selected user attribute

required_access_grants: Apply an access_grant to specific LookML objects (dimensions, measures, joins, explores)

```
access_grant: privileged_departments {  
    user_attribute: department  
    allowed_values: ["hr", "finance"]  
}  
  
dimension: salary {  
    required_access_grants:  
    [privileged_departments]  
    type: number  
    sql: ${TABLE}.salary ;;  
}
```

When Business Users Do Not Have Access

Explore

- Does not show up on the Explore nav drop-down
- Dashboard tiles will show up blank
- “The page you requested could not be found. It either doesn't exist or you don't have permission to view it.”

When Business Users Do Not Have Access

Field

- Does not show up on the field picker
- “employees.salary” no longer exists on Employees, or you do not have access to it, and it will be ignored.”

When Business Users Do Not Have Access

View or Join

- Will behave as if you didn't have access to all the fields inside
- "employees.salary" no longer exists on Employees, or you do not have access to it, and it will be ignored."

Combining Access Grants

`required_access_grants` accepts an array of values. More than one access policy can be defined and then applied to a LookML object

In the case of multiple access grants, a user must have all access grants that are required. If they only have a subset of the required access grants, the LookML object will be restricted

```
dimension: client_specific_privileged_field {  
    required_access_grants: [client_a, admins]  
    type: string  
    sql: ${TABLE}.client_specific_privileged_field ;;  
}
```

Intersecting Grants on Related LookML Objects

Intersecting Grants occur when there are grants placed on multiple levels within the LookML (*e.g. there is a grant at the explore level as well as an individual join*)

When there are Intersecting Grants, the user must fulfill all of access grants in order to properly see the LookML Objects (*ex. A field requires grants A and B, while its view requires grant C. You will need A, B, and C in order to see the field*)

The same applies for intersections between any other permission-able objects

Important Notes

If you reference a restricted object in the model via \${}, the referencing object will not inherit its restrictions

Extending an object, however, will inherit the required_access_grants since it is a defined parameter of the base object

Important Notes

If the extending object specifies a new required_access_grants, it will overwrite the base (it will not combine/intersect)

The content validator will not confuse a restricted field for a broken field; it knows the difference

Access Grants - Example

Restrict the access to the distribution centers join in the Order Items explore to only users who have a user attribute value of Inventory for the accessible_departments user attribute

Access Grants - Example

Restrict the access to the distribution centers join in the Order Items explore to only users who have a user attribute value of Inventory for the accessible_departments user attribute

```
access_grant: inventory {  
    user_attribute: accessible_departments  
    allowed_values: ["Inventory"]  
}
```

```
join: distribution_centers {  
    required_access_grants: [inventory]  
    type: left_outer  
    sql_on:  
        ${products.distribution_center_id} =  
            ${distribution_centers.id} ;;  
    relationship: many_to_one  
}
```

EXERCISE TIME!

PRACTICE EXERCISE 13: Access Grants

User email, first name, last name and full name are required to be hidden to only those authorized to see PII information. Your trainer will now create a new user attribute called `is_pii_viewer`. Create an access grant and then apply that as a required access grant on the email, first name, last name and full name fields (in the users view). Also unhide any fields that have a hidden parameter

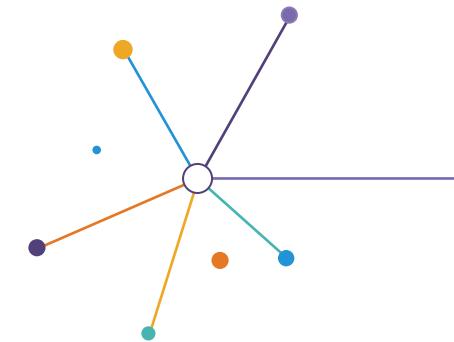
PRACTICE EXERCISE 13: Access Grants

User email, first name, last name and full name are required to be hidden to only those authorized to see PII information. Your trainer will now create a new user attribute called `is_pii_viewer`. Create an access grant and then apply that as a required access grant on the email, first name, last name and full name fields (in the users view). Also unhide any fields that have a hidden parameter

```
access_grant: is_pii_viewer {  
    user_attribute: is_pii_viewer  
    allowed_values: ["Yes"]  
}
```

```
dimension: email {  
    required_access_grants: [is_pii_viewer]  
    type: string  
    sql: ${TABLE}.email ;;  
}
```

Additional **Admin Panel Features**



Administration Panel Features

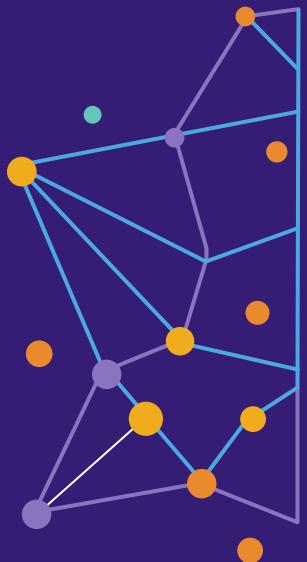
Looker features

- Labs: Beta and Experimental features that can be optionally enabled
- Legacy: Deprecated Looker features that will be removed

Performance monitoring

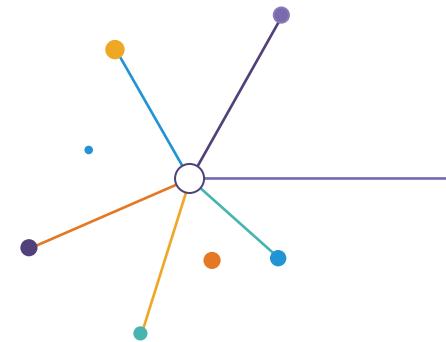
- Queries: List of queries recently run or currently running in the instance
- Persistent Derived Tables: Provides PDT trigger and load information

Looker instance usage data



Liquid Customization

What **is** Liquid?



What is Liquid?

Open-source, Ruby-based template language created by Shopify

Used in conjunction with LookML to build more flexible, dynamic code

What is Liquid?

Code denoted by braces {} and falls into three (3) different categories:

- **Objects:** tell Liquid where to show content on a page

```
dimension: product_image {  
    sql: ${product_id} ;;  
    html:  ;;  
}
```

- **Tags:** Create the logic and control flow for templates

```
label: "{% if _user_attributes['customer'] == 'A' %} Standard Margin  
      {% else %} Gross Margin {% endif %}"
```

- **Filters:** Change the output of a Liquid object

```
{% assign last_filter = part_split_at_sorts | first %}  
{% assign user_filters = user_filters | append:'&f' %}
```

Using Liquid in Looker

There are several places in LookML where Liquid can be used:

- the `action` parameter
- the `html` parameter
- the `label` parameter of a field
- the `link` parameter
- parameters that begin with `sql`
 - `sql`
 - `sql_on`
 - `sql_table_name`

A reference sheet for all available Liquid variables can be found [here](#)

Common Use Cases

Liquid can be used in a wide variety of ways in Looker. Some of the most popular use cases include

- Creating dynamic links or rendering dynamic images
- Setting up custom drills
- Changing the label of a field based on the model being used
- Aggregate awareness
- Adding custom conditional formatting
- Integrating templated filters and parameters

Liquid Parameters - Referencing LookML Objects

Variable	Definition	Example Output
value	Field value returned by the database query	8521935
rendered_value	Field value with Looker's default formatting	\$8,521,935.00
filterable_value	Field value formatted for use as a filter in a Looker URL	8521935
link	URL to Looker's default drill link	/explore/thelook/orders?fields=orders.order_amount&limit=500
linked_value	Field value with Looker's default formatting and linking	<u>\$8,521,935.00</u>

Liquid Parameters - Referencing LookML Objects

Variable	Definition	Example Output
_model._name	Model name for this field	thelook
_view._name	View name for this field	orders
_explore._name	Explore name for this field	order_items
_field._name	Field name itself	total_order_amount

Liquid Parameters - Liquid Tags

Variable	Definition	Example Output
{% date_start date_filter_name %}	The beginning date in a date filter you ask for with date_filter_name	2020-01-01
{% date_end date_filter_name %}	The ending date in a date filter you ask for with date_filter_name	2020-01-01
{% condition filter_name %} sql_or_lookml_reference {% endcondition %}	Filter value you ask for with filter_name applied to the sql_or_lookml_reference as SQL - Used with templated filters & conditional joins	Link 1 Link 2
{% parameter parameter_name %}	The value of the parameter filter you ask for with parameter_name	Link

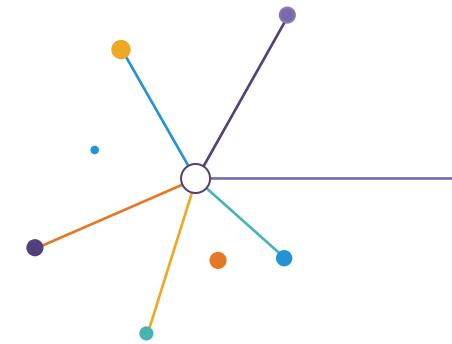
Liquid Parameters - Referencing LookML Objects 2

Variable	Definition	Example Output
_user_attributes['name_of_attribute']	The value of the user attribute you ask for with name_of_attribute, for the particular user running the query, when using user attributes	northeast
_query._query_timezone	The time zone in which the query was run	America/Los_Angeles
_filters['view_name.field_name']	The user filters applied to the field you ask for with view_name.field_name	NOT NULL
parameter_name._parameter_value	Injects the value of the parameter filter you ask for with parameter_name into a logical statement	Link

Liquid Parameters - Referencing Query Values

Variable	Definition	Example Output
view_name._in_query	Returns true if any field from the view is included in the query	true
view_name.field_name._in_query	Returns true if the field you ask for with view_name.field_name appears in the query data table, is included in a filter query, or is included in a query via the required_fields parameter	true
view_name.field_name._is_selected	Returns true if the field you ask for with view_name.field_name appears in the query data table	true
view_name.field_name._is_filtered	Returns true if the field you ask for with view_name.field_name is included in a query filter	true

Custom **Links & Drills**



Custom Links & Drills - Building Workflows

Set up custom workflows between Looker content or between Looker and other internal or external resources

- Link from an executive Dashboard to a detail Dashboard
- Link from a Look or Dashboard to an Explore
- Link from a value in a Look or Dashboard to a related page on the external web (*i.e. a SalesForce page*)

Custom Links & Drills - Building Workflows

Set up custom workflows between Looker content or between Looker and other internal or external resources

- Link from a value in a Look or Dashboard to a related page in a different internal system (i.e. an internal application or intranet page)
- Set up a custom drill path into a dimension or measure

Link Parameter

Most links are added to dimensions and measures using the [link](#) parameter

- **label** is the name this link will have in the drill menu
- **url** is the link URL and supports full liquid (but not full HTML)
- **icon_url** is the image URL to be used as an icon for this link

```
dimension: field_name {  
  link: {  
    label: "desired label name"  
    url: "desired_url"  
    icon_url: "url_of_an_image_file"  
  }  
}
```

Link Parameter - Example

Most Viewed Brands Online			
	Brand	Sessions ▾	Cart to Checkout Conversion
1	Levi's ...	6,088	86.41%
2	Allegra K ...	2,854	86.17%
3	Columbia ...	2,585	85.28%
4	Dockers DRILL INTO COLUMBIA	2,570	85.20%
5	Carhartt by Category	2,439	85.97%
6	Ray-Ban by Item Name	2,366	87.37%
7	Champs EXPLORE	2,036	84.53%
8	Hanes Filter on "Columbia"	1,388	85.32%
9	Lee	1,207	87.62%
10	Tren ...	1,082	86.05%
11	Russell ...	862	83.52%
12	Calvin Klein ...	793	84.35%
13	Patty ...	726	84.50%
14	Dick's ...	718	88.38%
15	Duofold ...	691	85.24%

Links to Google & a Dashboard - Example

The brand dimension will contain links to a Brand dashboard in Looker and to Google

```
dimension: brand {  
    sql: TRIM(${TABLE}.brand) ;;  
    link: {  
        label: "Website"  
        url: "http://www.google.com/search?q={{ value | encode_uri }}"  
        icon_url: "http://www.google.com/s2/favicons?domain=www.{{ value |  
            encode_uri }}.com"  
    }  
}
```

Links to Google & a Dashboard - Example

The brand dimension will contain links to a Brand dashboard in Looker and to Google

```
dimension: brand {  
  sql: TRIM(${TABLE}.brand) ;;  
  link: {  
    label: "Website"  
    url: "http://www.google.com/search?q={{ value | encode_uri }}"  
    icon_url: "http://www.google.com/s2/favicons?domain=www.{{ value | encode_uri }}.com"  
  }  
  
  link: {  
    label: "{{value}} Analytics Dashboard"  
    url: "/dashboards/694?Brand={{ value | encode_uri }}"  
    icon_url: "http://www.looker.com/favicon.ico"  
  }  
}
```

EXERCISE TIME!

EXERCISE 6: External Links

Add a link to the Brand dimension in the Products view that allows a user to click on a brand in the user interface and link to a google search for that brand

EXERCISE 6: External Links

Add a link to the Brand dimension in the Products view that allows a user to click on a brand in the user interface and link to a google search for that brand

```
dimension: brand {  
    type: string  
    sql: ${TABLE}.brand ;;  
    link: {  
        label: "Google"  
        url: "http://www.google.com/search?q={{ value }}"  
        icon_url: "http://google.com/favicon.ico"  
    }  
}
```

EXERCISE 7: Looker Link

Add a link to the Email dimension in the Users view that allows a user to click on an email in the user interface and link to the eCommerce Sample User Dashboard (Dashboard #1813) in the Training Advanced Space

EXERCISE 7: Looker Link

Add a link to the Email dimension in the Users view that allows a user to click on an email in the user interface and link to the eCommerce Sample User Dashboard (Dashboard #1813) in the Training Advanced Space

```
dimension: email {  
    type: string  
    sql: ${TABLE}.email ;;  
    link: {  
        label: "Category Detail Dashboard"  
        url: "/dashboards/1813?Email={{value}}"  
    }  
}
```

Linking to an Explore - Example #1

Links can also take users to a pre-configured Explore with dimensions, measures, and filters already present on the page. The selected value is often input as a filter into the Explore, providing a customized drill-through experience

```
dimension: city {  
    sql: ${TABLE}.metro ;;  
    link: {  
        label: "Link To An Explore"  
        url: "/explore/model/explore_name?fields=view.field_1,view.field_2,&f[view.filter_1]={  
            value }"  
        icon_url: "https://looker.com/favicon.ico"  
    }  
}
```

Linking to an Explore - Example #2

A customized drill to an Explore can show detailed customer information for any state selected

```
dimension: state {  
  sql: ${TABLE}.state ;;  
  link: {  
    label: "Drill Down to See Customers"  
    url: "/explore/events_ecommerce/users?fields=users.id,users.name&f[users.state]={{_filters['users.state']} | url_encode }}"  
    icon_url: "https://looker.com/favicon.ico"  
  }  
}
```

EXERCISE 8: Linking to an Explore

Using the Inventory Items Explore, create an explore with the Category, Name and Inventory Item Count. Filter by Category and then link to this Explore from the Category Dimension

EXERCISE 8: Linking to an Explore

Using the Inventory Items Explore, create an explore with the Category, Name and Inventory Item Count. Filter by Category and then link to this Explore from the Category Dimension

```
dimension: category {  
    type: string  
    sql: ${TABLE}.category ;;  
    link: {  
        label: "View Category Detail"  
        url:  
        "/explore/advanced_data_analyst_bootcamp/inventory_items?fields=inventory_items.product_category,inventory_items.product_name,inventory_items.count&f[products.category]={{value | url_encode }}"  
    }  
}
```

HTML Parameter

For even more customized drilling and linking, use the html parameter

- The dimension value will be shown in Looker and will also be a hyperlink
- Clicking the value will take a user to the specified link within the html
- Additional adjustments can be made to customize the user experience

```
dimension: field_name {  
  html: <a href="/dashboards/dashboard_number?NameFilter={{ value }}">{{  
    value }}</a> ;;  
}
```

Linked Buttons using HTML

	Users ID	Users Name	Users History ^	Users History Button
1		1 Robert Carroll	Order History	<button>Order History</button>
2		10 Caitlin Newsom	Order History	<button>Order History</button>
3		100 William Jones	Order History	<button>Order History</button>

```
dimension: order_history_button {  
    label: "History Button"  
    sql: ${TABLE}.id ;;  
    html: <a  
        href="/explore/events_ecommerce/order_items?fields=order_items.detail*&f[user  
        s.id]={{{ value }}}"><button>Order History</button></a> ;;  
}
```

Custom Drilling with HTML

```
dimension: history {  
    sql: ${TABLE}.user_name ;;  
    html: <a  
        href="/explore/thelook/orders?fields=orders.detail*&f[users.id]={  
            { id._value }}">Orders</a  
        | <a  
            href="/explore/thelook/order_items?fields=order_items.detail*&f[u  
            sers.id]={{ id._value }}">Items</a>;
```

Custom Drilling with HTML

Orders Created Date		Users Name	Users History	Orders Total Order Profit
1	2015-10-30	Cynthia Jones	Orders Items	\$36.8
2	2015-10-26	Robert Brooks	Orders Items	\$98.65
3	2015-10-27	Donald Parenteau	Orders Items	\$199.34
4	2015-11-04	Bill Wellman	Orders Items	\$11.69
5	2015-10-29	Rebecca Moll	Orders Items	\$16.65

Diagram illustrating custom drilling: An orange arrow points from the 'Items' link in the 'Users History' column of the first row of the top table to the 'Products Item Name' column of the corresponding row in the bottom table.

Orders Created Date		Order Items ID	Orders ID	Users Name	Users History	Products Item Name	Products Brand Name	Products Category Name	Products Department Name	Order Items Total Sale Price
1	2015-10-30	51392	30197	Cynthia Jones	Orders Items	BCBGMAXAZRIA Women's Hazel Rib Tank Dress	BCBGMAXAZRIA ↗	Dresses ↗	Women	\$48.00
2	2015-10-30	51393	30197	Cynthia Jones	Orders Items	Calvin Klein Men's 3 Pack No Show Liner Socks	Calvin Klein ↗	Plus ↗	Women	\$20.00
3	2015-07-25	44792	26281	Cynthia Jones	Orders Items	Devon & Jones DP310W Ladies' Short Sleeve Cotton/Lycra Stretch Jersey Tennis Polo Shirt	Devon & Jones ↗	Tops & Tees ↗	Women	\$15.99
4	2015-06-18	42352	24832	Cynthia Jones	Orders Items	Low Profile Dyed Cotton Twill Cap - Putty W39S55D	MG ↗	Accessories ↗	Women	\$5.95
5	2015-06-18	42353	24832	Cynthia Jones	Orders Items	TAUPE JUMPSUIT LACING POCKETS LAGENLOOK - FITS - 2X 3X 4X - T593S LOTUSTRADERS	LOTUSTRADERS ↗	Jumpsuits & Rompers ↗	Women	\$47.99
6	2015-06-18	42354	24832	Cynthia Jones	Orders Items	Pedipette by Peds 3 Pair No Show Footies NUDE Size 5-10 Size 5-10	Peds ↗	Socks & Hosiery ↗	Women	\$6.00
7	2015-01-16	32881	19270	Cynthia Jones	Orders Items	A. Byer Juniors Tropical Cambridge Pant	A. Byer ↗	Pants & Capris ↗	Women	\$40.00

Custom Link to SalesForce

```
dimension: id {  
    sql: ${TABLE}.opportunity_id ;;  
    html: <a href="https://na9.salesforce.com/{{ value }}" target="_new">  
        </a> ;  
}
```

Opportunity ID	Opportunity Closed Date	Account Name	Salesrep Name	Opp Acv
1	2015-11-06	Green Pages Australia	Kena Roker	1000000000000000000
2	2015-11-06	Zentu	Daphne Bolander	1000000000000000000
3	2015-11-06	Magis Networks	Ethel Zbierski	1000000000000000000
4	2015-11-06	Cartedge	Diane Conn	1000000000000000000

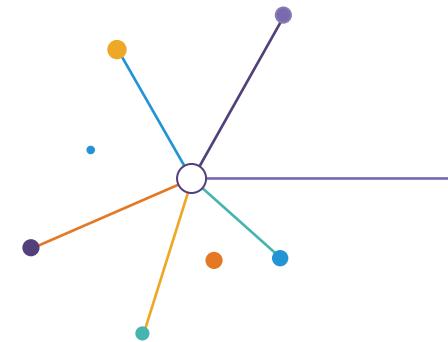
The screenshot shows a Looker dashboard interface. On the left, there is a table of opportunity data. An orange arrow points from the first row of the table to the corresponding account record in the Salesforce interface on the right. The Salesforce interface displays the account details for 'Internet Creations' (Account Name), which is a business account owned by Scott VonSchilling (Account Owner). The account has a status of 'B' and a test status of 'A'. The 'Custom Links' section on the left sidebar contains a link to the Salesforce account detail page.

Advanced Linking with Liquid

For even more advanced use cases that require different links in different Explores, use liquid tags to incorporate conditions into Links:

```
dimension: state {  
    sql: ${TABLE}.state ;;  
    html: {% if _explore._name == "order_items" %}  
        <a href=  
            "/explore/advanced_data_analyst_bootcamp/order_items?fields=order_items.detail*&f[users.state]  
            ]= {{ value }}">{{ value }}</a>  
        {% else %}  
        <a href=  
            "/explore/advanced_data_analyst_bootcamp/users?fields=users.detail*&f[users.state]=  
            {{ value }}">{{ value }}</a>  
        {% endif %} ;;  
    }  
}
```

Dynamic **SQL**



Dynamic SQL

Liquid variables can be used to dynamically change the SQL query that Looker executes based on the fields and filters that a user selects within the Explore interface. A few examples of use cases in which liquid variables can be used include:

- Adjust granularity level of derived tables based on fields in an Explore
- Adjust the join logic within an Explore based on views included in a user query

Dynamic SQL

Liquid variables can be used to dynamically change the SQL query that Looker executes based on the fields and filters that a user selects within the Explore interface. A few examples of use cases in which liquid variables can be used include:

- Direct Looker to an alternate database table depending on the level of granularity required for a user query (aggregate awareness)
- Apply a complex filter statement within the SQL only when a user has actually utilized the filter in his/her query

Adjusting the Granularity of Derived Tables

Liquid can be used to adjust the SQL in a derived table based on the query that a user runs in an Explore. Liquid prevents the need for multiple derived tables. Employ this when

- Using the highest level of granularity possible for a derived table to optimize query performance
- Handling complex calculations that are dependent on the granularity of the query, such as complex allocations or attribution
- Building snapshots for point-in-time analysis (valuable for calculating inventory metrics such as sell-through and turnover rates)

Adjusting the Granularity of Derived Tables

Dynamically identify a user's most recent web session by time period:

```
derived_table: {
  sql: SELECT
    {% if latest_session.dynamic_granularity_date._in_query %} DATE_TRUNC('day', created_at)
    {% elsif latest_session.dynamic_granularity_week._in_query %} DATE_TRUNC('week',created_at)
    {% else %} DATE_TRUNC('month',created_at)
    {% endif %} as dynamic_granularity
  ,user_id
  {% if latest_session.session_ending_uri._in_query %}
  ,SUBSTRING(MAX(created_at::varchar(10) || uri ),11) as session_ending_uri
  {% else %}
  ,NULL  as session_ending_uri
  {% endif %}
  FROM public.events
  WHERE
    {% condition latest_events_snapshot.dynamic_granularity_date %} created_at {% endcondition %}
  AND {% condition latest_events_snapshot.dynamic_granularity_week %} created_at {% endcondition %}
  AND {% condition latest_events_snapshot.dynamic_granularity_month %} created_at {% endcondition %}
  GROUP BY 1,2,3 ;;
}
```

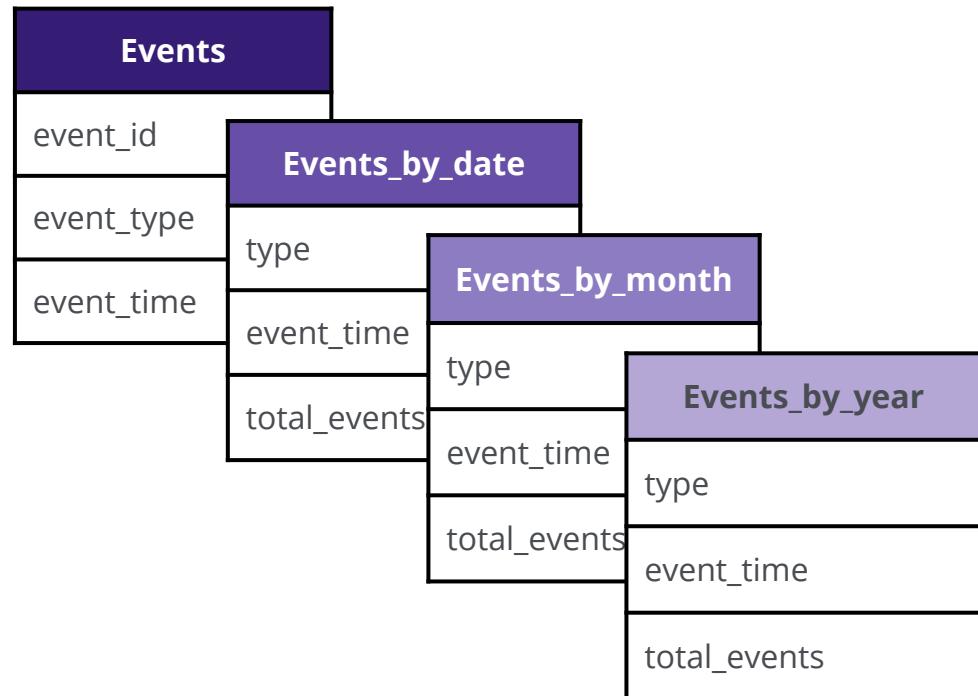
Adjusting the Granularity of Derived Tables

Dynamically identify a user's lifetime spend by time period:

```
derived_table: {  
    sql:  
        SELECT  
            users.id as customer_id,  
            SUM(sale_price) AS lifetime_spend  
        FROM  
            public.users as users JOIN public.order_items as order_items on users.id = order_items.user_id  
        WHERE  
            {% condition order_range %} order_items.created_at {% endcondition %}  
        GROUP BY 1;;}  
  
filter: order_range {  
    type: date  
}
```

Aggregate Awareness

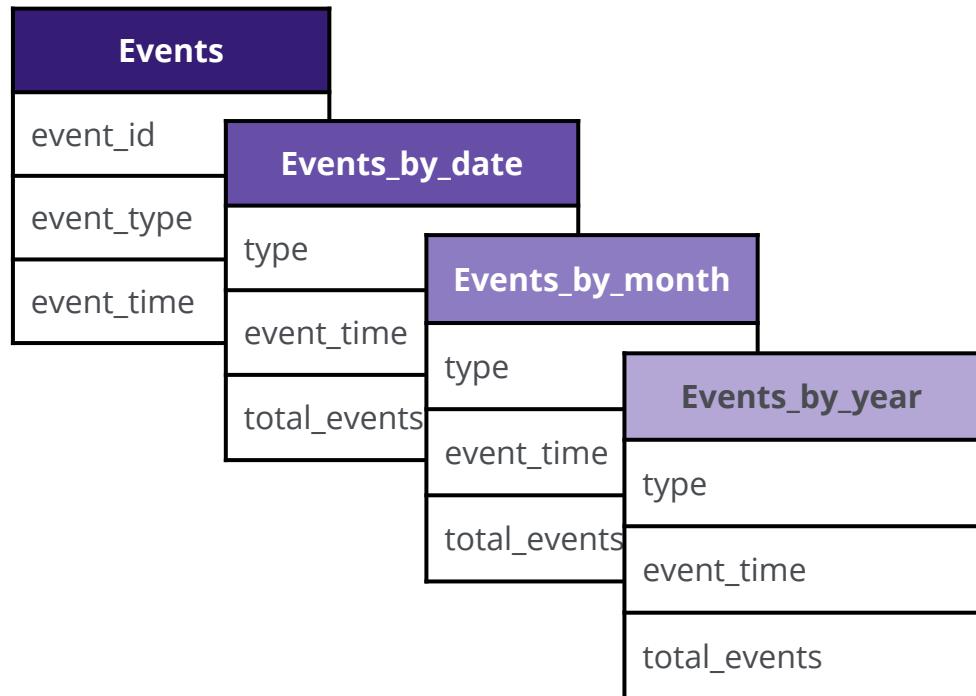
Using liquid variables in SQL to dynamically select a database table to query can be a great solution for handling aggregate tables with **similar column structures**



Aggregate Awareness

Without table name injection:

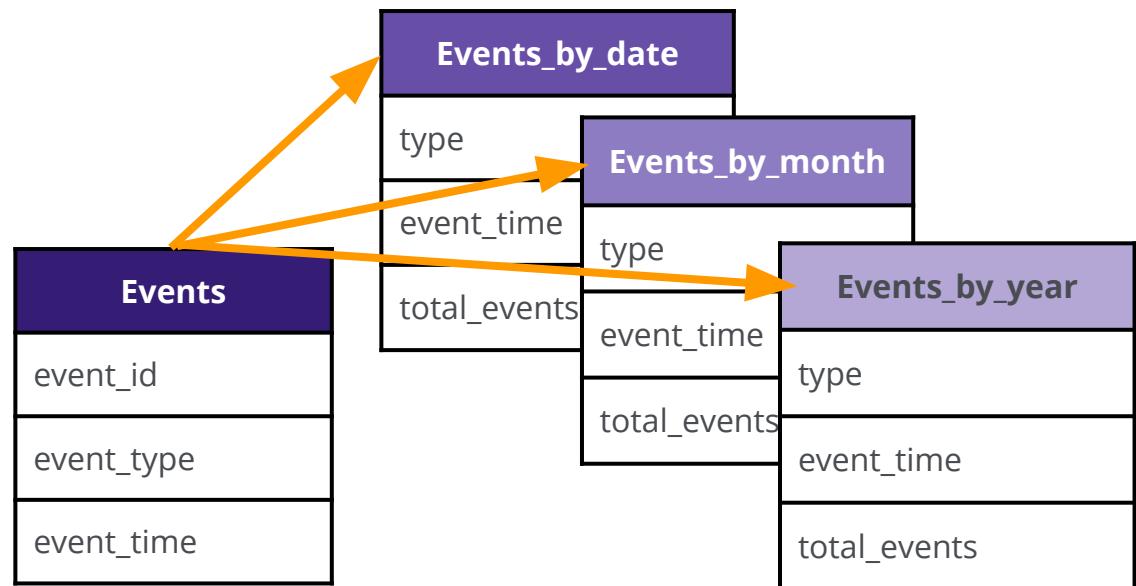
- 4 LookML Views
- LookML Joins for each View
- Users responsible for using Optimal View



Aggregate Awareness

With table name
injection:

- 1 LookML Views
- 1 LookML Join
- Optimal View always used



Aggregate Awareness

```
view: events {  
    sql_table_name:  
        {% if event_id._in_query %} events  
        {% elseif event_time._in_query %} events  
        {% elseif event_date._in_query %} events_by_date  
        {% elseif event_month._in_query %} events_by_month  
        {% else %} events_by_year {% endif %} ;;  
  
    dimension: event_id {  
    }  
  
    dimension_group: event {  
        type: time  
        sql: ${TABLE}.event_time ;;  
        timeframes: [time, date, month, year]  
    }  
  
    dimension: event_type {  
    }  
  
    measure: total_events {  
        type: sum  
        sql: {% if event_id._in_query %} 1  
              {% elseif event_time._in_query %} 1  
              {% else %} ${TABLE}.total_events {% endif %} ;;  
    }  
}
```



sql_table_name parameter

- Includes `._in_query` liquid variable to return true or false depending on which date fields are present in a query
- Contains liquid tagging with `if...else` conditions to direct Looker to the appropriate database table based on the query that a user runs

Aggregate Awareness

```
view: events {  
    sql_table_name:  
        {% if event_id._in_query %} events  
        {% elif event_time._in_query %} events  
        {% elif event_date._in_query %} events_by_date  
        {% elif event_month._in_query %} events_by_month  
        {% else %} events_by_year {% endif %} ;;  
  
    dimension: event_id {  
    }  
  
    dimension_group: event {  
        type: time  
        sql: ${TABLE}.event_time ;;  
        timeframes: [time, date, month, year]  
    }  
  
    dimension: event_type {  
    }  
  
    measure: total_events {  
        type: sum  
        sql: {% if event_id._in_query %} 1  
            {% elif event_time._in_query %} 1  
            {% else %} ${TABLE}.total_events {% endif %} ;;  
    }  
}
```

measure argument

- Lowest level events table wouldn't have a `total_events` column to sum
- Other aggregate tables would have an aggregate `total_events` column
- Appropriate calculation is done in all scenarios

Aggregate Awareness

One Explore

Optimal table
used for every
query

Queries adopt
additional
aggregation
levels

The screenshot shows the Looker interface with the following components:

- Left Panel (Fields):** Shows the hierarchy of fields. "Event Types" is expanded, showing "Events". "Events" is expanded, showing "DIMENSIONS": "Event Date", "Event ID", "Event Type"; and "MEASURES": "Total Events".
- Top Bar:** Buttons for "DATA", "RESULTS", "SQL", and "Calculations".
- SQL View:** The SQL code is:

```
SELECT
    COALESCE(SUM(events.total_events), 0) AS "events.to"
FROM
    events_by_year AS events
LIMIT 500
```

The line "events_by_year AS events" is highlighted with an orange box.

Aggregate Awareness

One Explore

Optimal table
used for every
query

Queries adopt
additional
aggregation
levels

The screenshot shows the Looker interface with an 'Explore' titled 'Event Types'. The sidebar on the left lists dimensions and measures: Dimensions include Event Date (Date, Month, Time, Year), and Measures include Total Events. The main area shows a SQL query:

```
SELECT
  DATE(events.event_time) AS "events.event_date",
  COALESCE(SUM(events.total_events), 0) AS "events.events"
FROM events_by_date
AS events
GROUP BY 1
ORDER BY 1 DESC
LIMIT 500
```

A yellow box highlights the 'events_by_date' table in the FROM clause of the SQL query.

Aggregate Awareness

One Explore

Optimal table
used for every
query

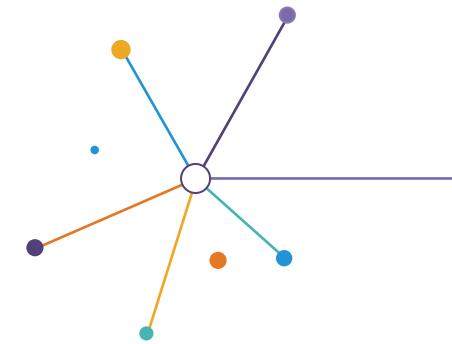
Queries adopt
additional
aggregation
levels

The screenshot shows the Looker interface. On the left, a sidebar titled 'All Fields' contains tabs for 'Dimensions' and 'Measures'. Under 'Dimensions', 'Event Types' is expanded, showing 'Events' as a child node. Under 'Events', 'Event Date' is expanded, showing 'Date', 'Month', 'Time', and 'Year'. Under 'Time', there are buttons for 'PIVOT', 'FILTER', and a gear icon. Under 'Measures', 'Total Events' is listed with a 'FILTER' button and a gear icon. On the right, the main area has tabs for 'DATA', 'RESULTS', 'SQL', 'Calculations', and 'Rows'. The 'SQL' tab is selected, displaying the following SQL code:

```
SELECT
    TO_CHAR(events.event_time, 'YYYY-MM-DD HH24:MI:SS') AS "event_time"
    , COALESCE(SUM(1
    ), 0) AS "events.total_events"
FROM events
AS events
GROUP BY 1
ORDER BY 1 DESC
LIMIT 500
```

A red box highlights the 'COALESCE(SUM(1), 0)' part of the query.

Conditional **Formatting**



Conditional Formatting

HTML can be used to apply custom formatting to any fields in Looker.

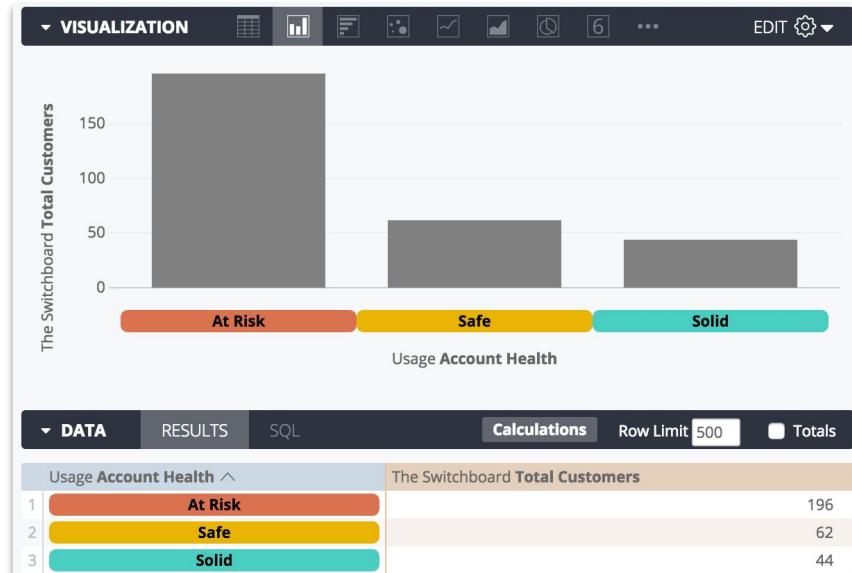
- Add custom colors to dimension labels or header backgrounds
- Include picture or icons as part of displayed values
- Change the size or font of displayed text
- Add custom details via a drop down into the cell of a table
- Build a progress bar into the cell of a table that compares the cell value against a goal

Conditional Formatting - Example

```

dimension: account_health {
    sql: ${TABLE}.account_health ;
    html: {% if value == 'At Risk' %}
        <b><p style="color: black; background-color: #dc7350;
margin: 0; border-radius: 5px; text-align:center">{{ value
}}</p></b>
    {% elif value == 'Safe' %}
        <b><p style="color: black; background-color: #e9b404;
margin: 0; border-radius: 5px; text-align:center">{{ value
}}</p></b>
    {% else %}
        <b><p style="color: black; background-color: #49cec1;
margin: 0; border-radius: 5px; text-align:center">{{ value
}}</p></b>
    {% endif %}
    ;
}

```



Conditional Formatting - Example

```
measure: total_sale_price {  
    type: sum  
    value_format_name: usd  
    sql: ${sale_price}  
    drill_fields: [detail*]  
    html: <font size="+5">{{  
linked_value }}</font>  
}
```

Orders Created Year	2014	2015
Orders Created Month Num	Order Items Total Sale Price	Order Items Total Sale Price
1	\$52,420.05	\$452,882.20
2	\$75,952.37	\$470,074.48
3	\$107,126.84	\$550,303.18
4	\$125,647.48	\$567,885.49
5	\$157,792.55	\$612,118.65
6	\$176,296.54	\$664,639.53
7	\$208,308.34	\$738,711.55
8	\$225,887.66	\$774,477.23

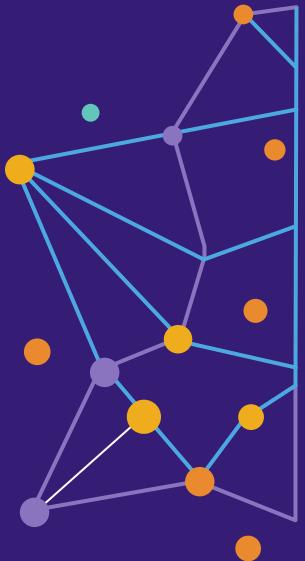
Conditional Formatting - Advanced Example

```
measure: total_gross_margin {
  type: sum
  value_format_name: usd
  sql: ${gross_margin} ;;
}

html:

<div style="width:100%"> <details>
<summary style="outline:none">{{ total_gross_margin._linked_value }}</summary> Sale Price: {{ total_sales_price._linked_value }}<br/>
Inventory Costs: {{ inventory_items.total_cost._linked_value }}</details>
</div>;;
```

Order Items Created Year <	2016	Order Items Total Gross Margin
Order Items Created Month Num ^		
1	1	▼ \$153,275.68 Sale Price: \$290,816.84 Inventory Costs: \$137,541.15
2	2	► \$149,192.79
3	3	► \$161,288.00
4	4	► \$162,927.63
5	5	► \$167,593.41
6	6	► \$167,510.27



Parameters & Templated Filters

Parameters & Templated Filters

Increase interactivity in Explores, Looks, and Dashboards for users

The screenshot shows a Looker interface with a 'FILTERS (1)' section at the top. A dropdown menu is open under the heading 'Species Animal Conservation Status'. The dropdown contains a list of conservation statuses: Breeder, Endangered, Extinct, In Recovery, Migratory, Proposed Endangered, Proposed Threatened, Resident, and several others listed below them. Each item in the list has a count to its right: 6,534, 6,239, 4,629, 4,409, 3,985, 3,840, 3,455, 3,390, 3,266, and 2,693. The 'Endangered' option is highlighted with a brown background. The 'Proposed Threatened' option is also highlighted with a brown background. The 'Custom Filter' checkbox is unchecked.

Conservation Status	Count
Breeder	6,534
Endangered	6,239
Extinct	4,629
In Recovery	4,409
Migratory	3,985
Proposed Endangered	3,840
Proposed Threatened	3,455
Resident	3,390
	3,266
	2,693

Parameters & Templated Filters

WHAT: User-input values that can be added into a query dynamically

- Parameters: **specific, fixed values** that can be entered by users and then passed directly into a SQL query using liquid
- Templated Filters: user-entered values that are passed into SQL queries using **intelligently written conditional logic**

Parameters & Templated Filters

WHY: Provide **greater flexibility** in how user inputs can influence the SQL queries written

- Dynamic dimensions and measures to consolidate code
- Dynamic derived tables
- Conditionally displayed values

How Do We Do This in Looker?

Step 1

Developer sets up back-end logic

Step 2

User inputs value into front-end filter

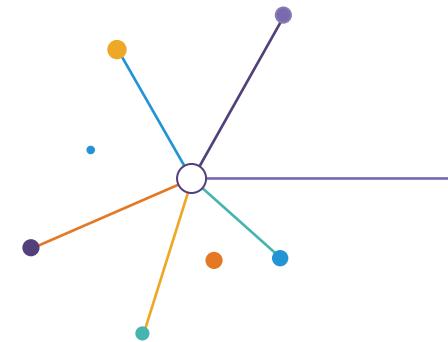
Step 3

Value is inserted into the SQL & new query is run

Step 4

Explore is returned

Parameters



Step 1 - Developer Sets Up Back-End Logic

Parameter field that takes a single user-input value

- type: string / number / unquoted / date, etc.
- allowed_value or suggestions

Used in the syntax

{% parameter parameter_name %}

```
parameter: field_to_select {  
    type: unquoted  
    allowed_value: {  
        value: "Category"  
        label: "Category"  
    }  
    allowed_value: {  
        value: "Conservation_status"  
        label: "Conservation Status"  
    }  
    allowed_value: {  
        value: "Common_names"  
        label: "Common Names"  
    }  
}  
  
dimension: dynamic_column_select {  
    type: string  
    sql: ${TABLE}.{% parameter field_to_select %} ;;  
    label_from_parameter: field_to_select  
}
```

Step 1

Step 2

Step 3

Step 4

Step 2 - User Inputs Value into Front-End Filter

Parameters create filter-only field on the front end in an Explore

allowed_values appear as drop-down options

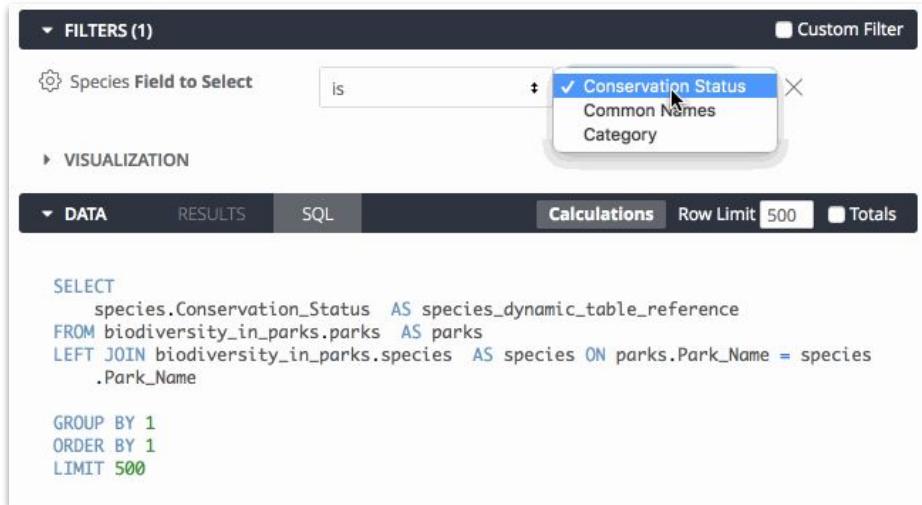
The screenshot shows the Looker interface with the following elements:

- FILTERS (1)**: A dropdown menu titled "Species Field to Select" with three options: "Conservation Status" (selected), "Common Names", and "Category".
- VISUALIZATION**: A visualization section.
- DATA**: A tabbed section showing the underlying SQL query:

```
SELECT
    species.Conservation_Status AS species_dynamic_table_reference
FROM biodiversity_in_parks.parks AS parks
LEFT JOIN biodiversity_in_parks.species AS species ON parks.Park_Name = species.Park_Name
GROUP BY 1
ORDER BY 1
LIMIT 500
```

Step 3 - Value Inserted into SQL & New Query Run

Parameter value is inserted
into the {
% parameter
parameter_name %} portion
of SQL



The screenshot shows a Looker interface with a "FILTERS (1)" section. A dropdown menu is open under "Species Field to Select", showing options: "Conservation Status" (selected), "Common Names", and "Category". Below the filters, the "DATA" tab is selected, showing an SQL query:

```
SELECT
    species.Conservation_Status AS species_dynamic_table_reference
FROM biodiversity_in_parks.parks AS parks
LEFT JOIN biodiversity_in_parks.species AS species ON parks.Park_Name = species
    .Park_Name

GROUP BY 1
ORDER BY 1
LIMIT 500
```

Step 4 - Explore is Returned

A screenshot of a Looker dashboard interface. At the top, it shows "5,000 rows · from cache · 28m ago" with "Run" and "Settings" buttons. Below that is a "FILTERS (1)" section with a "Species Field to Select" dropdown set to "is" and a "Common Name" dropdown. A "Custom Filter" checkbox is unchecked. The main area is titled "VISUALIZATION" with various chart icons. Below is a list titled "Species Common Names" with 10 items numbered 1 to 10:

Rank	Species Common Name
1	'A'ō, Newell's Shearwater
2	'Akole
3	'Anauau, Kunana, Naunau
4	'Golden Sedge
5	'I 'O Nui, Laukahī
6	'I'iwi
7	'Loki' Juniper Hairstreak
8	'Mojave' Dotted Blue
9	'O'ōpu Alamo'o
10	'O'ōpu Nakea

Step 1

Step 2

Step 3

Step 4

Parameters - Example

There's a hierarchy within the product view, and dashboard users need to view dashboard visualizations by any level of this hierarchy

The hierarchy includes the following fields:

- Department (highest level)
- Category
- Brand (lowest level)



Parameters - Set Up Input Logic via LookML

Label: what the user will see in the filter options

Value: the value that will be inserted into the SQL query

Default Value: the value that will be inserted automatically if a user has not yet made a selection

```
parameter: select_product_detail {  
    type: unquoted  
    default_value: "department"  
    allowed_value: {  
        value: "department"  
        label: "Department"  
    }  
    allowed_value: {  
        value: "category"  
        label: "Category"  
    }  
    allowed_value: {  
        value: "brand"  
        label: "Brand"  
    }  
}
```

Parameters - Dynamic Dimension Creation

Input the parameter value directly into the SQL as the field name:

```
dimension: product_hierarchy {  
    label_from_parameter:  
    select_product_detail  
    type: string  
    sql: ${TABLE}.{%  
        parameter  
        select_product_detail %}  
    ;;  
}
```

The screenshot shows the Looker interface with a parameter filter and its corresponding SQL query.

FILTERS (1):
Products Select Product Detail is Category

VISUALIZATION

DATA (selected), **RESULTS**, **SQL**

SQL:

```
SELECT  
    products.category AS "products.product_hierarchy"  
FROM public.order_items AS order_items  
LEFT JOIN public.inventory_items AS inventory_items ON order_items.inventory_item_id = inventory_items.id  
LEFT JOIN public.products AS products ON inventory_items.product_id = products.id  
GROUP BY 1  
ORDER BY 1  
LIMIT 500
```

Parameters - Dynamic Dimension Creation

What if there are transformations done on these dimensions in the LookML?

Pulling from the underlying table directly will bypass any SQL logic applied to these dimension in Looker

Use LookML dimensions instead

```
dimension: product_hierarchy {  
    label_from_parameter: select_product_detail  
    type: string  
    sql:  
        {% if select_product_detail._parameter_value  
== 'department' %}  
            ${department}  
        {% elseif  
select_product_detail._parameter_value ==  
            'category' %}  
            ${category}  
        {% else %}  
            ${brand}  
        {% endif %} ;;  
}
```

Parameters - Example Summary

There are TWO ways to build dynamic fields that allow users to change the way that they look at visualizations (without Exploring)

- Align the parameter value to the underlying column name and input those values directly into the \${TABLE}.column_name statement
 - **Pro:** Simple and requires little extra code
 - **Con:** Will not account for any LookML logic applied to these dimensions

Parameters - Example Summary

There are TWO ways to build dynamic fields that allow users to change the way that they look at visualizations (without Exploring)

- Use liquid if tags to reference parameter values and input the appropriate LookML dimensions
 - **Pro:** Logic added or changed to these LookML dimensions will be inherited by the dynamic dimension
 - **Con:** Additional lines of code and knowledge of liquid required

EXERCISE TIME!

EXERCISE 9: Parameters

Dashboard users would like to be able to view dashboard visualizations by different time periods. Create a dynamic dimension field within the Order Items view that enables users to choose between different order creation date fields

Time frames to include

- Date
- Week
- Month

EXERCISE 9: Parameters

```
parameter: select_timeframe {  
    type: unquoted  
    default_value: "created_month"  
    allowed_value: {  
        value: "created_date"  
        label: "Date"  
    }  
    allowed_value: {  
        value: "created_week"  
        label: "Week"  
    }  
    allowed_value: {  
        value: "created_month"  
        label: "Month"  
    }  
}
```

EXERCISE 9: Parameters

What if there are transformations done on these dimensions in the LookML? Pulling from the underlying table directly will bypass any SQL logic applied to these dimension in Looker. Use LookML dimensions instead:

```
dimension: dynamic_timeframe {  
    label_from_parameter: select_timeframe  
    type: string  
    sql:  
        {% if select_timeframe._parameter_value == 'created_date' %}  
        ${created_date}  
        {% elif select_timeframe._parameter_value == 'created_week' %}  
        ${created_week}  
        {% else %}  
        ${created_month}  
        {% endif %} ;
```

EXERCISE 10: Parameters

Depending on the types of web events that someone is analyzing, the event duration may be better measured in minutes vs. seconds. Add a parameter into the event_session_funnel view and update the time_in_funnel dimension to allows users to choose whether the time in the funnel is calculated in minutes or seconds.

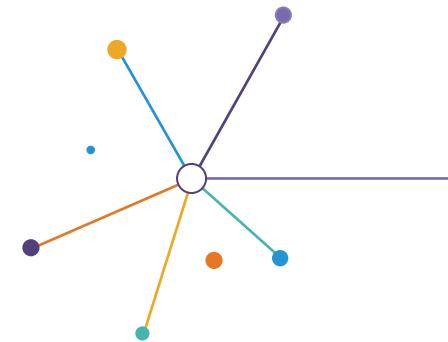
Be sure to include the following options:

- Seconds
- Minutes

EXERCISE 10: Parameters

```
parameter: time_type {  
    type: unquoted  
    default_value: "min"  
    allowed_value: {  
        label: "Minutes"  
        value: "min"  
    }  
    allowed_value: {  
        label: "Seconds"  
        value: "sec"  
    }  
}  
  
dimension: time_in_funnel {  
    type: number  
    sql:  
        datediff(  
            {% parameter time_type %}  
            ,${event1_raw}  
            ,COALESCE(${event3_raw}, ${event2_raw})  
        )  
        ;;  
}
```

Templated **Filters**



Step 1 - Developer Sets Up Back-End Logic

Filter field that utilizes Looker's generated SQL filter logic with string, number, date, etc types

Value generates logic in SQL

```
filter: animal_conservation_status {  
    type: string  
    suggest_dimension: species.conservation_status  
    suggest_explore: parks  
}
```

Used in the syntax

```
{% condition filter_name %} field_to_affect {% endcondition %}
```

Step 2 - User Inputs Value into Front-End Filter

A filter-only field is created on the front end in an Explore

suggest_dimension values appear as drop-down options

The screenshot shows the 'FILTERS (1)' section of a Looker Explore interface. A dropdown menu is open under the heading 'Species Animal Conservation Status'. The options listed are: Breeder, Endangered, Extinct, In Recovery, Migratory, Proposed Endangered, Proposed Threatened, and Resident. At the bottom of the dropdown, there is a truncated SQL query:

```
SELECT
    parks.State AS parks_s
    COUNT(DISTINCT CASE WHEN
        THEN (RIGHT(species.Specie
        species_changeable_coi
    FROM biodiversity_in_parks.parks AS parks
    LEFT JOIN biodiversity_in_parks.species AS species ON parks.Park_Name = species
        .Park_Name
    GROUP BY 1
    ORDER BY 2 DESC
    LIMIT 500
```

On the right side of the interface, there are buttons for 'Custom Filter', 'Row Limit 500', and 'Totals'.

Step 1

Step 2

Step 3

Step 4

Step 3 - Value Inserted into SQL & New Query Run

Templated filter value is written into the generated filter logic

FILTERS (1) Custom Filter

Species Animal Conservation Status

is equal to

VISUALIZATION

DATA RESULTS SQL

SELECT

```
    parks.State AS parks_s
    COUNT(DISTINCT CASE WHEN
    THEN (RIGHT(species.Specie
    species_changeable_co
FROM biodiversity_in_parks.parks AS parks
LEFT JOIN biodiversity_in_parks.species AS species ON parks.Park_Name = species
    .Park_Name
GROUP BY 1
ORDER BY 2 DESC
LIMIT 500
```

Row Limit 500 Totals

Step 1

Step 2

Step 3

Step 4

Step 4 - Explore is Returned

A screenshot of a Looker dashboard interface. At the top, it shows "5,000 rows · from cache · 28m ago" with "Run" and "Settings" buttons. Below that is a "FILTERS (1)" section with a "Species Field to Select" dropdown set to "is" and a "Common Name" dropdown. A "Custom Filter" checkbox is unchecked. The main area is titled "VISUALIZATION" with various chart icons. Below is a list titled "Species Common Names" with 10 items numbered 1 to 10:

Rank	Species Common Name
1	'A'ō, Newell's Shearwater
2	'Akole
3	'Anauau, Kunana, Naunau
4	'Golden Sedge
5	'I 'O Nui, Laukahī
6	'I'iwi
7	'Loki' Juniper Hairstreak
8	'Mojave' Dotted Blue
9	'O'ōpu Alamo'o
10	'O'ōpu Nakea

Step 1

Step 2

Step 3

Step 4

Templated Filters - Example 1

How does the profit margin of the Jeans category compare to the profit margin across all other categories?

▼ FILTERS (1) Custom Filter

 Products **Choose A Category to Compare** is equal to Jeans   

► VISUALIZATION

▼ DATA RESULTS SQL Calculations Row Limit 500 Totals

Products Category Comparator		Order Items Profit Margin
1	Jeans	48.17%
2	All Other Categories	52.96%

Templated Filters - Ex 1 - LookML Input Logic

Suggest Explore: the Explore that will be queried in order to pull a list of suggested filter values

Suggest Dimension: the dimension that should be used within the suggest Explore for providing a list of suggested filter value

```
filter:  
choose_a_category_to_compare {  
    type: string  
    suggest_explore:  
        inventory_items  
        suggest_dimension:  
            products.category  
    }  
}
```

Templated Filters - Ex 1 - Dynamic Dimension

```
dimension: category_comparator {  
    type: string  
    sql:  
        CASE  
        WHEN {% condition choose_a_category_to_compare %}  
            ${category}  
        {% endcondition %}  
        THEN ${category}  
            ELSE 'All Other Categories'  
        END  
        ;;  
}
```

Templated Filters - Ex 1 - Dynamic Dimension

FILTERS (1) Custom Filter

Products Choose A Category to Compare is equal to Jeans

VISUALIZATION

DATA RESULTS SQL Calculations Row Limit 500 Totals

```
SELECT
  CASE WHEN (products.category = 'Jeans') THEN products.category
    ELSE 'All Other Categories'
  END
    AS "products.category_comparator",
  (COALESCE(SUM((order_items.sale_price - inventory_items.cost) ), 0))/NULLIF((COALESCE(SUM(order_items.sale_price ), 0),
  0) AS "order_items.profit_margin"
FROM public.order_items AS order_items
LEFT JOIN public.inventory_items AS inventory_items ON order_items.inventory_item_id = inventory_items.id
LEFT JOIN public.products AS products ON inventory_items.product_id = products.id
GROUP BY 1
ORDER BY 1 DESC
LIMIT 500
```

EXERCISE TIME!

EXERCISE 11: Templatized Filters

The event_session_facts derived table needs to recalculate all values based on the time frame that a user has selected in the front-end Explore or dashboard. Pass a user filter on the session_start_date field that results in a filter being applied to the created_at column in the derived table. This will enhance performance and ensure that calculations are accurate.

EXERCISE 11: Templatized Filters

```
WITH session_facts AS (
    SELECT
        session_id
        ,COALESCE(user_id::varchar, ip_address) AS identifier
        ,FIRST_VALUE (created_at) OVER (PARTITION BY session_id ORDER BY created_at ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS session_start
        ,LAST_VALUE (created_at) OVER (PARTITION BY session_id ORDER BY created_at ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS session_end
        ,FIRST_VALUE (event_type) OVER (PARTITION BY session_id ORDER BY created_at ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS session_landing_page
        ,LAST_VALUE (event_type) OVER (PARTITION BY session_id ORDER BY created_at ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS session_exit_page
    FROM events
    WHERE {%- condition session_start_date %} created_at {%- endcondition %}
)
SELECT * FROM session_facts
GROUP BY 1, 2, 3, 4, 5, 6 ;
```

Templated Filters Summary

Input values for templated filters can come from

- Filter fields
- Regular dimension and measure fields

In derived tables, the templated filter will act upon a column(s) from the underlying database (*use the appropriate SQL reference*)

When regular fields are used, filters will be applied in two places

- The templated filter
- The WHERE clause of the outer query

EXERCISE 12: Templatized Filters (Advanced)

PART 1: Users on the marketing team need to analyze the share of users coming from each traffic source. Create a single measure within the Users view that allows a user to choose any available Traffic Source and see a count of users for that traffic source (to then be compared to the overall count of users to calculate a percentage)

NOTE: Be sure to include all Traffic Source options

EXERCISE 12: Templatized Filters (Advanced)

Users on the marketing team need to analyze the share of users coming from each traffic source. Create a single measure that allows a user to choose any available Traffic Source and see a count of users for that traffic source (to then be compared to the overall count of users to calculate a percentage)

HINT: Use a hidden dimension to adjust the filter criteria of a filtered measure!

EXERCISE 12: Tempered Filters (Advanced)

```
filter: incoming_traffic_source {  
    type: string  
    suggest_dimension:  
        users.traffic_source  
    suggest_explore: users }  
  
measure: changeable_count_measure  
{  
    type: count_distinct  
    sql: ${id} ;;  
    filters: {  
        field:  
            hidden_traffic_source_filter  
        value: "Yes"  
    }  
}
```

When to Use Parameters vs Templated Filters

Parameter Fields

Insert user input directly
(or using values you
define as allowed values)

Templated Filters

Insert values as
Looker-generated logical
statements

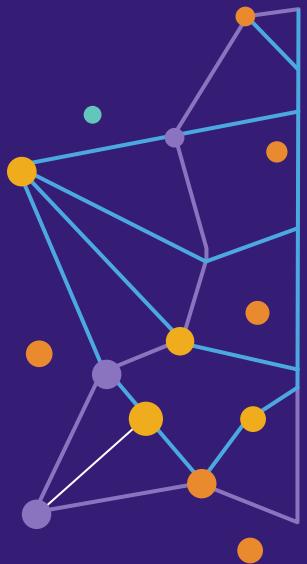
Additional Examples

Dynamically Selecting Fields using Parameters

Timeframe vs. Timeframe Analysis Using Templated Filters

Dynamically Query Tables Using Parameters

Dynamic Timeframes for Dimension Groups



Extends

What Are LookML Extensions?

Extending a LookML object = combining its contents with another LookML object

- Views
- Explores
- LookML Dashboards

Extends allow you to modularize code

They can serve as the building blocks for model development

Why Use Extends?

Writing DRY code

Easier and faster to make changes

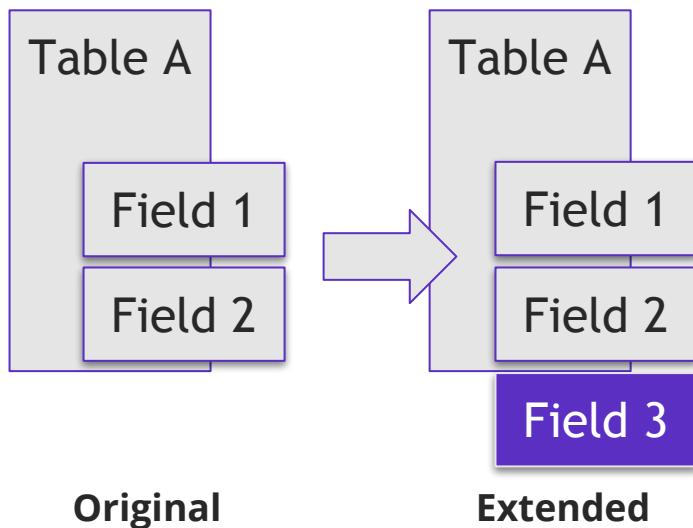
Consistency

Easier management of different field sets for different users

LookML View Extensions

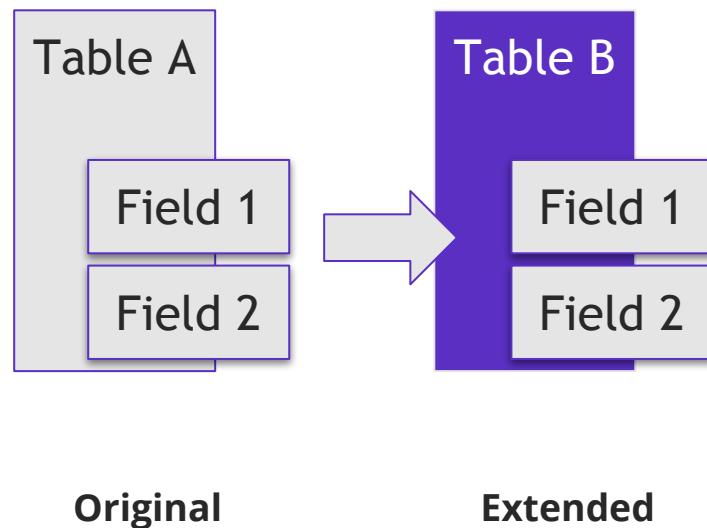
Adding Fields to a View

A view can be extended to include additional fields



Changing the Table of a View

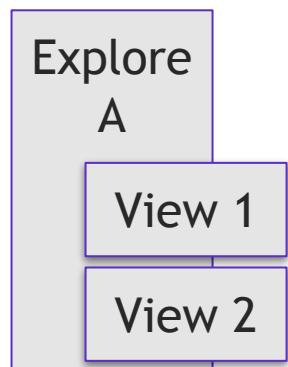
A View can be extended to change the table it's pointing to by overriding an object's parameters



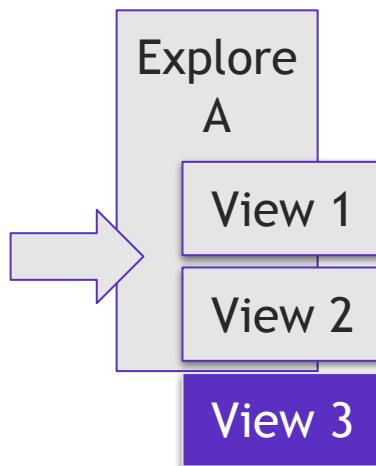
LookML Explore Extensions

Adding Views to an Explore

An Explore can be extended to include additional Views



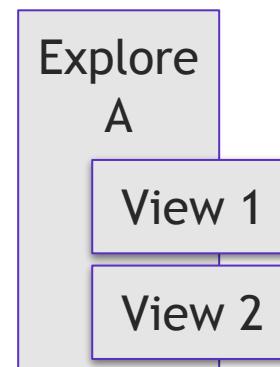
Original



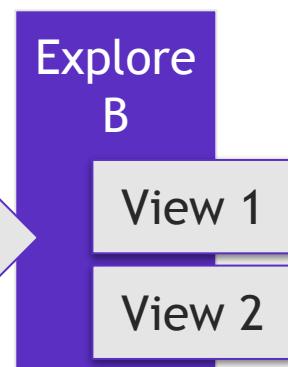
Extended

Changing Base View of an Explore

An Explore can be extended to change the Base View



Original



Extended

The Four Steps Involved in Extension Execution

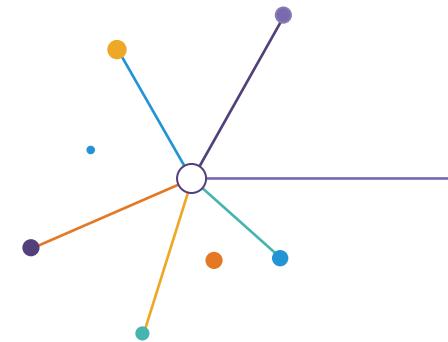
COPY: A copy is made of the defined LookML object being extended

MERGE: The copy is merged with the defined LookML object that is doing the extending

RESOLVE CONFLICTS: If a LookML object is defined in both places, the object doing the extending is used

FINISH: The new LookML object can be used within the model just like any other LookML object

View Extensions



View Extensions - A Use Case

Many database tables may have common columns. These may include system timestamps, ID fields added through ETL processes, or similar fields that all follow consistent naming conventions

View Extensions - A Use Case

Examples might include

- ID
- Created_at (timestamp)
- Updated_at (timestamp)
- Created_by
- Modified_by

These can be defined in a single file and then added to all relevant views using extends

View Extensions - Create File with Common Fields

```
view: system_fields {  
  extension: required  
  
  dimension: id {  
    hidden: yes  
    primary_key: yes  
    type: number  
    sql: ${TABLE}.id ;;  
  }  
  
  dimension_group: created {  
    type: time  
    timeframes: [raw, time, date, week, month, month_num, quarter, year]  
    sql: ${TABLE}.created_at ;;  
  }  
}
```

extension: required ensures this view cannot be used unless it is extended

No sql_table_name is defined here because this view will inherit the sql_table_name definition of any view it is extended into

View Extensions - Extend using Common Fields

```
include: "system_fields.view"
view: order_items {
  extends: [system_fields]
  sql_table_name: public.order_items ;;
}

include: "system_fields.view"
view: users {
  extends: [system_fields]
  sql_table_name: public.users ;;
}

include: "system_fields.view"
view: inventory_items {
  extends: [system_fields]
  sql_table_name: public.inventory_items ;;
}
```

Common fields are defined and maintained in a single place

The file with common field definitions is included at the top of the view files that will reference that shared logic

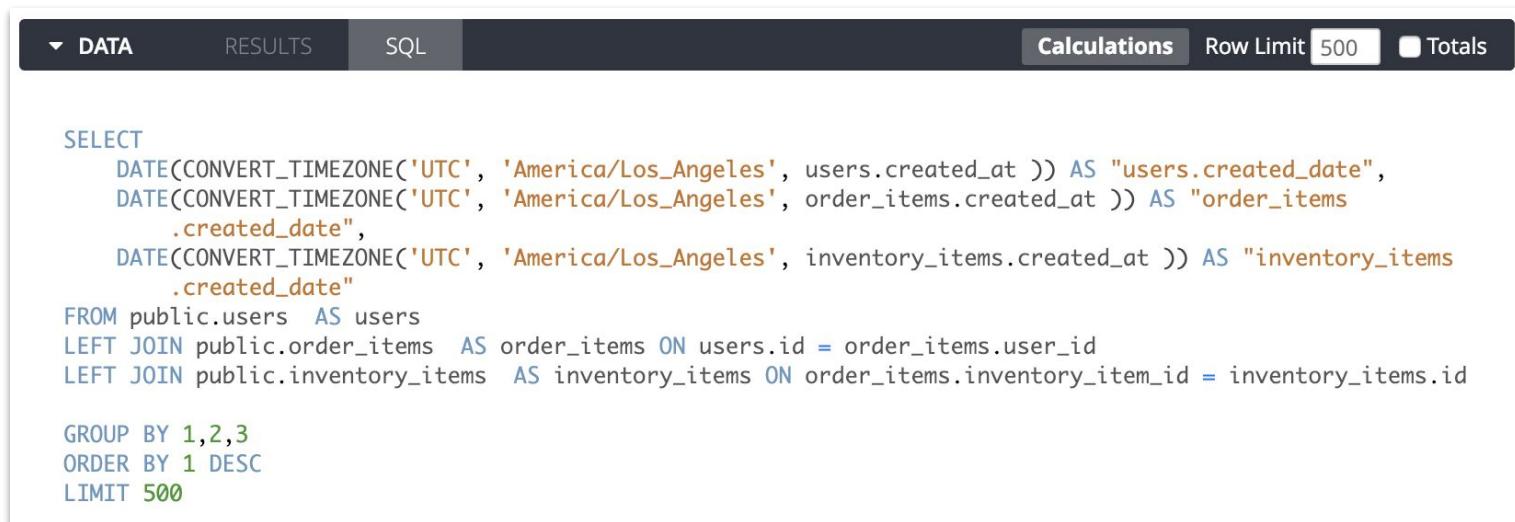
An `extends` parameter is defined to reference the view where the shared field logic is defined

The `sql_table_name` parameter is defined as normal

View Extensions - All Fields Available in Explore

Fields defined within the `system_fields` view are still visible in Explore

Fields are scoped with the correct table name



The screenshot shows the Looker interface with the following details:

- Top navigation bar: DATA, RESULTS, SQL, Calculations, Row Limit 500, Totals.
- SQL tab is selected.
- SQL code area:

```
SELECT
    DATE(CONVERT_TIMEZONE('UTC', 'America/Los_Angeles', users.created_at)) AS "users.created_date",
    DATE(CONVERT_TIMEZONE('UTC', 'America/Los_Angeles', order_items.created_at)) AS "order_items
        .created_date",
    DATE(CONVERT_TIMEZONE('UTC', 'America/Los_Angeles', inventory_items.created_at)) AS "inventory_items
        .created_date"
FROM public.users AS users
LEFT JOIN public.order_items AS order_items ON users.id = order_items.user_id
LEFT JOIN public.inventory_items AS inventory_items ON order_items.inventory_item_id = inventory_items.id
GROUP BY 1,2,3
ORDER BY 1 DESC
LIMIT 500
```

Let's Break This Down...

Step 1 - Copy

```
view: system_fields {  
  extension: required  
  
  dimension: id {...}  
  
  dimension_group:  
    created  
    {...}  
  }  
}
```

```
view: system_fields {  
  extension: required  
  
  dimension: id {...}  
  
  dimension_group:  
    created  
    {...}  
  }
```

COPY

Step 2 - Merge

```
view: system_fields {  
  extension: required  
  
  dimension: id {...}  
  
  dimension_group:  
    created  
    {...}  
}
```

```
view: system_fields {  
  extension: required  
  
  dimension: id {...}  
  
  dimension_group:  
    created  
    {...}  
}
```

```
view: order_items {  
  extends:  
    [system_fields]  
  sql_table_name:  
    public.order_items ;;  
  
  dimension: status {...}  
  
  dimension_group:  
    shipped  
    {...}  
}
```

MERGE

Step 3 - Resolve Conflicts

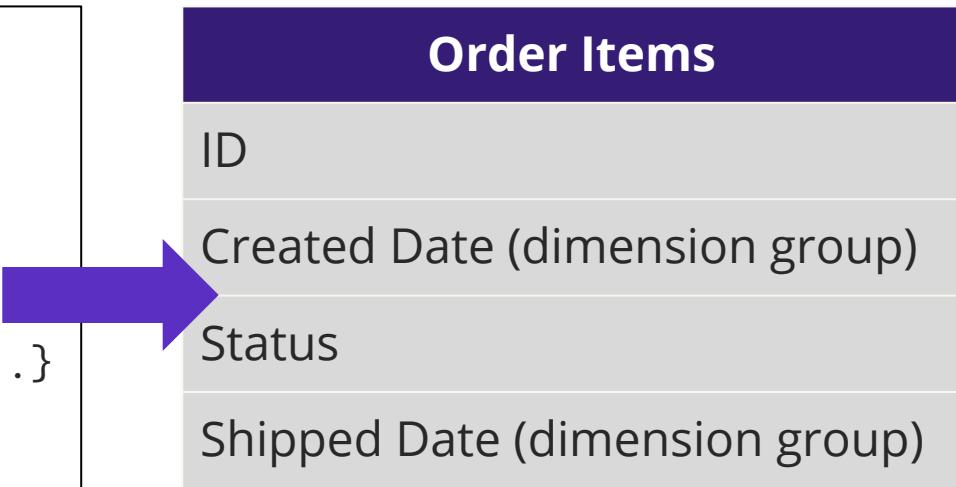
```
view: system_fields {  
  extension: required  
  
  dimension: id {...}  
  
  dimension_group:  
    created  
    {...}  
  }  
}
```

```
view: system_fields {  
view: order_items {  
  extension: required  
  sql_table_name:  
  public.order_items ;;  
  dimension: id {...}  
  
  dimension_group:  
    created  
    {...}  
    dimension: status {...}  
  
    dimension_group:  
    shipped  
    {...}  
  }
```

```
view: order_items {  
  extends:  
  [system_fields]  
  sql_table_name:  
  public.order_items ;;  
  
  dimension: status {...}  
  
  dimension_group:  
    shipped  
    {...}  
  }
```

Step 4 - Interpret Result

```
view: order_items {  
    sql_table_name:  
    public.order_items ;;  
  
    dimension: id {...}  
  
    dimension_group: created {...}  
  
    dimension: status {...}  
  
    dimension_group: shipped {...}  
}
```



EXERCISE TIME!

EXERCISE 14: View Extensions

The Event and User views each contain several geography dimensions. The underlying columns have the same names, and the LookML logic applied (such as map layers and custom geographic fields such as regions) should be the same across both views

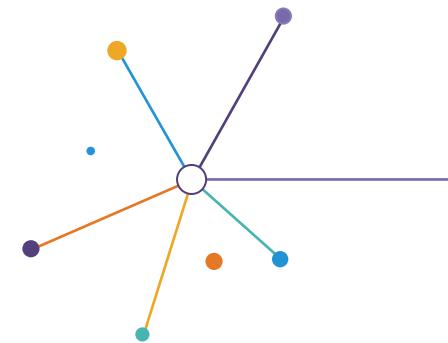
Create a centralized geography view and include those definitions in both the User and Event views using extends potentially leveraging fields like City, State, Country, Region, Location

EXERCISE 14: View Extensions

```
view: geography_dimensions {  
    extension: required  
  
    dimension: city { ... }  
    dimension: country { ... }  
    dimension: latitude { ... }  
    dimension: longitude { ... }  
    dimension: location { ... }  
    dimension: state { ... }  
    dimension: zip { ... }  
    dimension: region { ... }  
  
}
```

```
include: "geography_dimensions.view"  
view: events {  
    extends: [geography_dimensions]  
    sql_table_name: public.events ;;  
  
include: "geography_dimensions.view"  
include: "system_fields.view"  
view: users {  
    extends: [geography_dimensions  
,system_fields]  
    sql_table_name: public.users ;;
```

Explore **Extensions**



Explore Extensions - A Use Case

Most organizations have a variety of different types of users with different use cases

Imagine that we have a marketing team with one department focused on search engine optimization (SEO) and another focused on our conversion funnel

Explore Extensions - A Use Case

The SEO team needs

- Detailed event information
- User information

The conversion team needs

- Everything the SEO team needs
- Event funnel analysis
- Sales

Explore Extensions - Create Base Explore

Base Explore should have joins for all views that both teams need

```
explore: events {  
  description: "Start here for SEO Analysis"  
  join: event_session_facts {  
    type: left_outer  
    sql_on: ${events.session_id} = ${event_session_facts.session_id} ;;  
    relationship: many_to_one  
  }  
  join: users {  
    type: left_outer  
    sql_on: ${events.user_id} = ${users.id} ;;  
    relationship: many_to_one  
  }  
}
```

Explore Extensions - Extend Base Explore

Create a second Explore with additional joins

```
explore: conversions {
    description: "Start here for Conversion Analysis"
    fields: [ALL_FIELDS*, -order_items.profit]
    from: events
    view_name: events
    extends: [events]
    join: event_session_funnel {
        type: left_outer
        sql_on: ${events.session_id} = ${event_session_funnel.session_id} ;;
        relationship: many_to_one
    }
    join: order_items {
        type: left_outer
        sql_on: ${users.id} = ${order_items.user_id} ;;
        relationship: many_to_many
    }
}
```

Explore Extensions - Two Explores, One Join

Events Explore for SEO

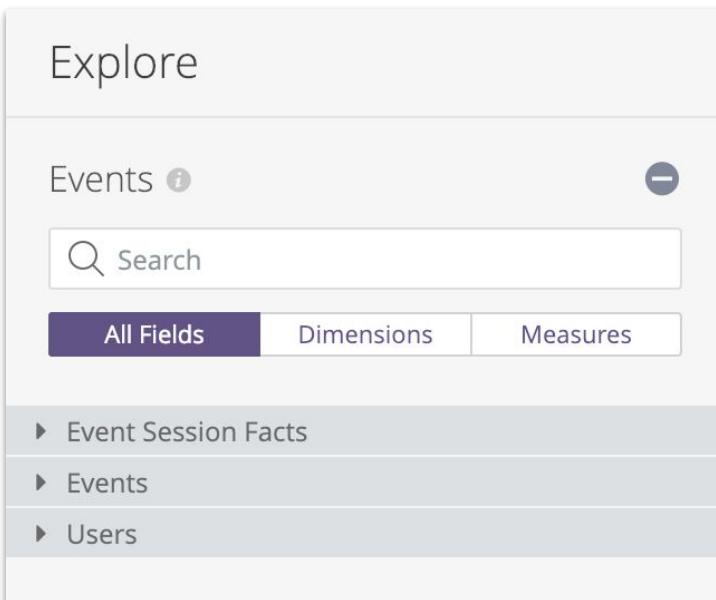
Explore

Events *i* -

Search

All Fields Dimensions Measures

▶ Event Session Facts
▶ Events
▶ Users



Conversions Explore

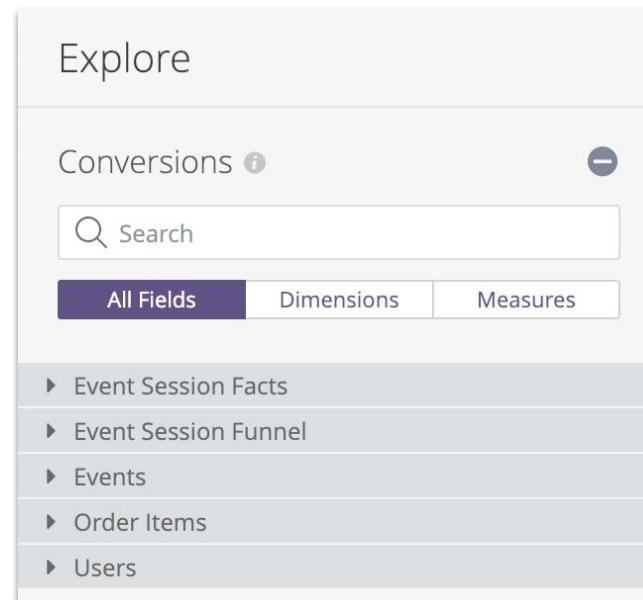
Explore

Conversions *i* -

Search

All Fields Dimensions Measures

▶ Event Session Facts
▶ Event Session Funnel
▶ Events
▶ Order Items
▶ Users



Let's Break This Down...

Step 1 - Copy

```
explore: events {  
  
  description: "Start  
  here for SEO Analysis"  
  
  join:  
    event_session_facts  
  { ... }  
  
  join: users { ... }  
  
}
```

```
explore: events {  
  
  description: "Start  
  here for SEO Analysis"  
  
  join:  
    event_session_facts  
  { ... }  
  
  join: users { ... }  
  
}
```



COPY

Step 2 - Merge

```
explore: events {  
  
  description: "Start  
  here for SEO Analysis"  
  
  join:  
    event_session_facts  
  { ... }  
  
  join: users { ... }  
  
}
```

```
explore: events {  
  
  description: "Start  
  here for SEO Analysis"  
  
  join:  
    event_session_facts  
  { ... }  
  
  join: users { ... }  
  
}
```

```
explore: conversions {  
  
  description: "Start here  
  for Conversion Analysis"  
  fields: [ALL_FIELDS*,  
    -order_items.profit]  
  from: events  
  view_name: events  
  extends: [events]  
  
  join:  
    event_session_funnel  
    { ... }  
  join: order_items { ... }
```

MERGE

Step 3 - Resolve Conflicts

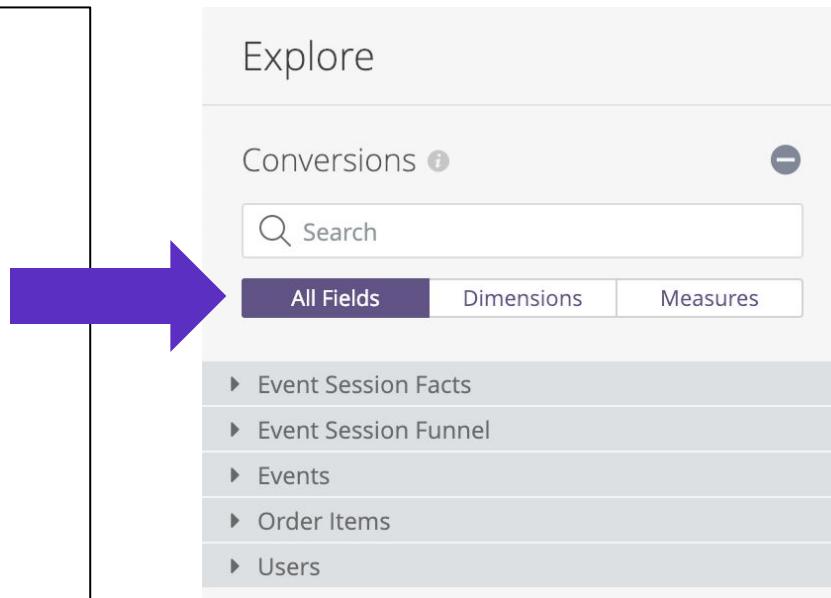
```
explore: events {  
  
  description: "Start  
  here for SEO Analysis"  
  
  join:  
    event_session_facts  
  { ... }  
  
  join: users { ... }  
  
}
```

```
explore: events {  
  
  explore: conversions {  
  
    description: "Start here  
    for SEO Analysis"  
    description: "Start here  
    for Conversion Analysis"  
  
    join:  
      event_session_facts {...}  
    join: users {...}  
    join:  
      event_session_funnel {...}  
      join: order_items {...}  
  }
```

```
explore: conversions {  
  
  description: "Start here  
  for Conversion Analysis"  
  fields: [ALL_FIELDS*,  
    -order_items.profit]  
  from: events  
  view_name: events  
  extends: [events]  
  
  join:  
    event_session_funnel  
    { ... }  
  join: order_items { ... }  
}
```

Step 4 - Interpret Result

```
explore: conversions {  
  description: "Start here for  
Conversion Analysis"  
  from: events  
  view_name: events  
  
  join: event_session_facts { ... }  
  
  join: users { ... }  
  
  join: event_session_funnel { ... }  
  
  join: order_items { ... }  
}
```



Advanced Use Cases

Creating a base model and extending it in different ways

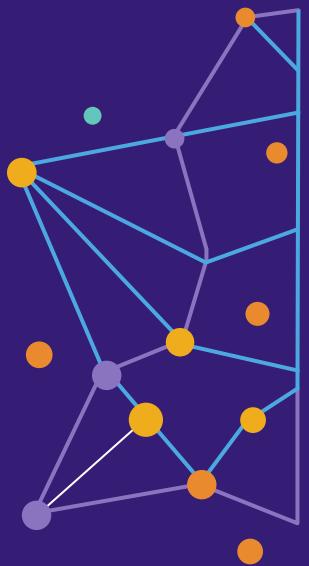
- Extension Required on Explores
- Expose different Explores for different teams
- Add model set security

Modularizing joins that are used frequently across multiple Explores
(in the same way that common fields can be defined in a single place and reused across views)

Advanced Use Cases

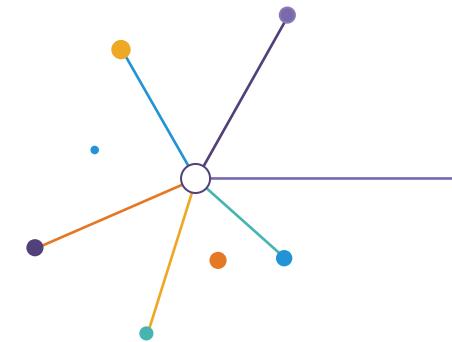
Leveraging core libraries

- Central BI team code that needs to be used differently by departments
- Looker blocks (*like Google Analytics*)



Debugging

Unknown or **Inaccessible Fields**



Unknown or Inaccessible Fields

▼ LookML Errors (2)



Unknown or inaccessible field

"user_order_facts.first_order_date" referenced in "users.days_since_first_order". Check for typos and missing joins.

users:17 (ecommerce:order_items) [? Documentation](#)

Unknown or inaccessible field

"user_order_facts.first_order_date" referenced in "users.days_since_last_order". Check for typos and missing joins.

users:22 (ecommerce:order_items) [? Documentation](#)

“Unknown or inaccessible field” is probably the most common error message

This happens when an Explore includes a View that references a field in a different View that's not joined into that Explore

Unknown or Inaccessible Fields

We create fields in users view that references fields in `user_order_facts` view

```
1 view: users {  
2   sql_table_name: public.users ;;  
3  
4   dimension: id {  
5     primary_key: yes  
6     hidden: yes  
7     type: number  
8     sql: ${TABLE}.id ;;  
9   }  
10  
11   dimension: days_as_customer {  
12     description: "Days between join and current date"  
13     type: number  
14     sql: DATEDIFF('day', ${created_date}, current_date) ;;  
15   }  
16  
17   dimension: days_since_first_order {  
18     description: "Days between first order and current date"  
19     type: number  
20     sql: DATEDIFF('day', ${user_order_facts.first_order}, current_date) ;;  
21   }  
22  
23   dimension: days_since_last_order {  
24     description: "Days between last order and current date"  
25     type: number  
26     sql: DATEDIFF('day', ${user_order_facts.latest_order}, current_date) ;;  
27   }  
28 }
```

```
, MIN(NULLIF(orders.created_at,0)) as first_order  
, MAX(NULLIF(orders.created_at,0)) as latest_order  
, COUNT(DISTINCT DATE_TRUNC('month', NULLIF(orders.created_at,0))) as month_count  
FROM orders AS orders  
INNER JOIN order_items oi  
ON orders.id = oi.order_id  
GROUP BY user_id  
;;  
sortkeys: ["user_id"]  
distribution_style: "EVEN"  
sql_trigger_value: SELECT current_date ;;  
}  
  
dimension_group: first_order {  
  type: time  
  timeframes: [date, week, month, year]  
  sql: ${TABLE}.first_order ;;  
}  
  
dimension_group: latest_order {  
  type: time  
  timeframes: [date, week, month, year]  
  sql: ${TABLE}.latest_order ;;  
}
```

Unknown or Inaccessible Fields

`user_order_facts` is joined to
users in the users Explore--no
problem!

The problem is coming from the
`order_items` Explore. The users
view is joined in but
`user_order_facts` is not

```
explore: users {
  label: "Users Explore"
  join: user_order_facts {
    type: left_outer
    sql_on: ${users.id} = ${user_order_facts.user_id} ;;
    relationship: one_to_one
  }
}

explore: order_items {
  join: orders {
    type: left_outer
    sql_on: ${order_items.order_id} = ${orders.order_id} ;;
    relationship: many_to_one
  }
}

join: subsequent_order_facts {
  view_label: "Orders"
  type: left_outer
  sql_on: ${subsequent_order_facts.order_id} = ${orders.order_id} ;;
  relationship: one_to_one
}

join: inventory_items {
  type: left_outer
  sql_on: ${order_items.inventory_item_id} = ${inventory_items.id} ;;
  relationship: many_to_one
}

join: users {
  type: left_outer
  sql_on: ${orders.user_id} = ${users.id} ;;
  relationship: many_to_one
}

join: products {
  type: left_outer
  sql_on: ${inventory_items.product_id} = ${products.id} ;;
  relationship: many_to_one
}
```

Unknown or Inaccessible Fields

This means those fields in users
that reference fields in
`user_order_facts` are broken!

*(But, only in the `order_items`
Explore)*

```
explore: users {
  label: "Users Explore"
  join: user_order_facts {
    type: left_outer
    sql_on: ${users.id} = ${user_order_facts.user_id} ;;
    relationship: one_to_one
  }
}

explore: order_items {
  join: orders {
    type: left_outer
    sql_on: ${order_items.order_id} = ${orders.order_id} ;;
    relationship: many_to_one
  }
}

join: subsequent_order_facts {
  view_label: "Orders"
  type: left_outer
  sql_on: ${subsequent_order_facts.order_id} = ${orders.order_id} ;;
  relationship: one_to_one
}

join: inventory_items {
  type: left_outer
  sql_on: ${order_items.inventory_item_id} = ${inventory_items.id} ;;
  relationship: many_to_one
}

join: users {
  type: left_outer
  sql_on: ${orders.user_id} = ${users.id} ;;
  relationship: many_to_one
}

join: products {
  type: left_outer
  sql_on: ${inventory_items.product_id} = ${products.id} ;;
  relationship: many_to_one
}
```

Reviewing Error Messages to Assist Debugging

We know **this field** is referenced
in **this field** and that causes a
problem

▼ **LookML Errors (2)** ?

Unknown or inaccessible field
`user_order_facts.first_order_date'` referenced in
`users.days_since_tirst_order`. Check for typos and
missing joins.

users:17 (ecommerce:order_items) ? Documentation

Unknown or inaccessible field
`"user_order_facts.first_order_date"` referenced in
`"users.days_since_last_order"`. Check for typos and
missing joins.

users:22 (ecommerce:order_items) ? Documentation

Reviewing Error Messages to Assist Debugging

The errors officially reside on line 17 and 22 of the users files

And are associated with the order_items Explore in the ecommerce model

▼ LookML Errors (2) ?

Unknown or inaccessible field
"user_order_facts.first_order_date" referenced in "users.days_since_first_order". Check for typos and missing joins.

users:17 (ecommerce:order_items) ? Documentation

Unknown or inaccessible field
"user_order_facts.first_order_date" referenced in "users.days_since_last_order". Check for typos and missing joins.

users:22 (ecommerce:order_items) ? Documentation

Reviewing Error Messages to Assist Debugging

This tells us that there are fields in user_order_facts referenced in users, and user_order_facts is not joined into the order_items Explore in the ecommerce model, thus breaking those fields

▼ LookML Errors (2) ?

Unknown or inaccessible field
"user_order_facts.first_order_date" referenced in "users.days_since_first_order". Check for typos and missing joins.

users:17 ecommerce:order_items ? Documentation

Unknown or inaccessible field
"user_order_facts.first_order_date" referenced in "users.days_since_last_order". Check for typos and missing joins.

users:22 ecommerce:order_items ? Documentation

How Do We Fix This?

OPTION 1

Join user_order_facts into the
order_items Explore

```
join: users {  
  type: left_outer  
  sql_on: ${orders.user_id} = ${users.id} ;;  
  relationship: many_to_one  
}  
  
join: user_order_facts {  
  type: left_outer  
  sql_on: ${users.id} = ${subsequent_order_facts.user_id} ;;  
  relationship: one_to_one  
}  
  
join: products {  
  type: left_outer  
  sql_on: ${inventory_items.product_id} = ${products.id} ;;  
  relationship: many_to_one  
}  
}
```

How Do We Fix This?

OPTION 2

Use the fields parameter in the Explore declaration to exclude those two fields from the order_items Explore

```
explore: order_items {
  fields: [ALL_FIELDS*, -users.days_since_first_order]
  join: orders {
    type: left_outer
    sql_on: ${order_items.order_id} = ${orders.order_id} ;;
    relationship: many_to_one
  }

  join: subsequent_order_facts {
    view_label: "Orders"
    type: left_outer
    sql_on: ${subsequent_order_facts.order_id} = ${orders.order_id} ;;
    relationship: one_to_one
  }

  join: inventory_items {
    type: left_outer
    sql_on: ${order_items.inventory_item_id} = ${inventory_items.id} ;;
    relationship: many_to_one
  }
}
```

How Do We Fix This?

OPTION 3

Use the fields parameter in the users join condition to include all the fields except the two fields causing the problem

```
explore: order_items {  
  join: orders {  
    type: left_outer  
    sql_on: ${order_items.order_id} = ${orders.order_id} ;;  
    relationship: many_to_one  
  }  
  
  join: subsequent_order_facts {  
    view_label: "Orders"  
    type: left_outer  
    sql_on: ${subsequent_order_facts.order_id} = ${orders.order_id} ..  
    relationship: one_to_one  
  }  
  
  join: inventory_items {  
    type: left_outer  
    sql_on: ${order_items.inventory_item_id} = ${inventory_item}  
    relationship: many_to_one  
  }  
  
  join: users [  
    fields: [user_fields_for_order_items*]  
    type: left_outer  
    sql_on: ${orders.user_id} = ${users.id} ;;  
    relationship: many_to_one  
  ]  
  
  set: user_fields_for_order_items {  
    fields: [  
      age,  
      age_tier,  
      city,  
      country,  
      created_date,  
      created_month,  
      days_as_customer,  
      email,  
      first_name,  
      last_name,  
      name,  
      gender,  
      state,  
      traffic_source,  
      zip,  
      count,  
      average_age  
    ]  
  }
```

The set is defined in the users View file

How Do We Fix This?

OPTION 4:

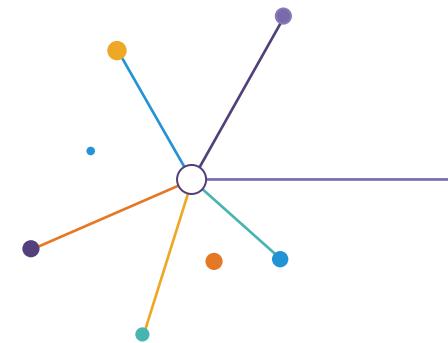
Extend a new view from the original, adding our problematic fields only to the extended version

Then we pick which version of the view to use in which Explore

```
15 - dimension_group: created {
16   type: time
17   timeframes: [raw, date, month, month_name, quarter, year]
18   sql: ${TABLE}.longitude ;;
19 }
20 }
21 }
22
i 23 - view: users_ext {
24   extends: [users]
25   view_label: "Users"
26   dimension: customer_age {
27     type: number
28     sql: datediff(day, ${created_raw}, ${orders.created_raw}) ;;
29   }
30 }

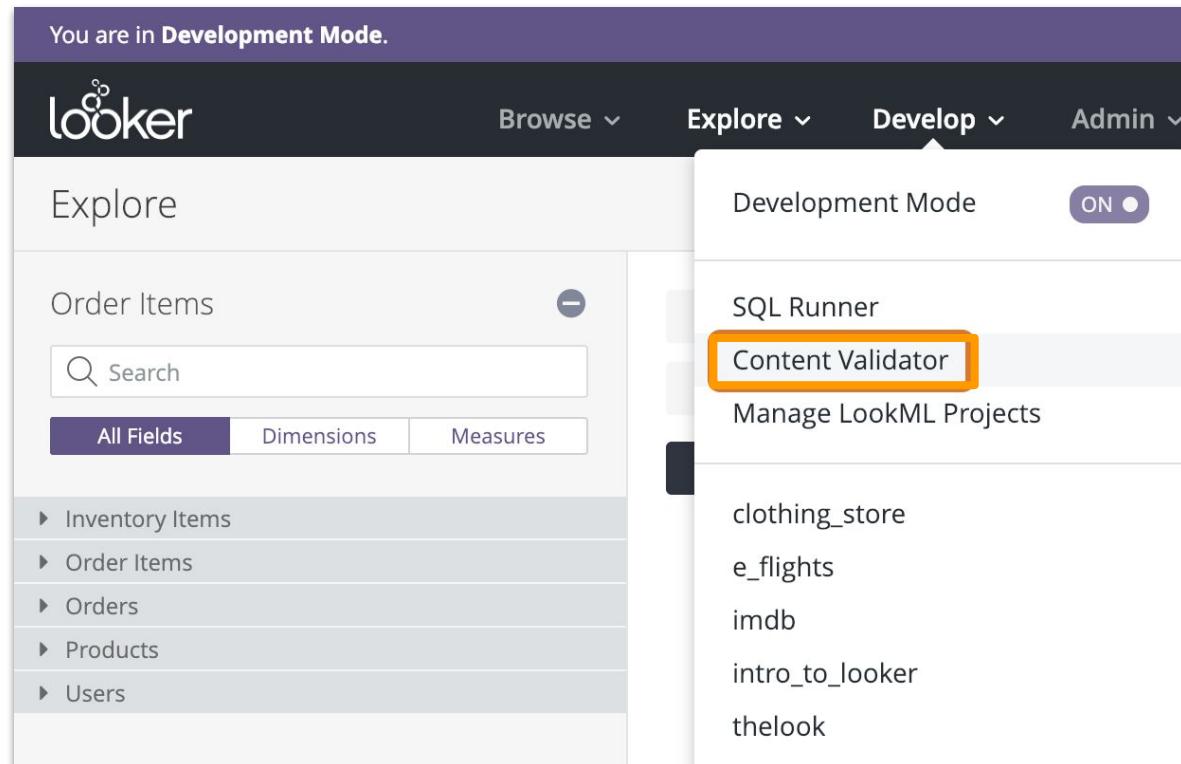
6 explore: users {}
7
8 - explore: order_items {
9   join: orders {
10     type: left_outer
11     sql_on: ${order_items.order_id} = ${orders.id} ;;
12     relationship: many_to_one
13   }
14 - join: users_ext {
15   type: left_outer
16   sql_on: ${order_items.user_id} = ${users_ext.id} ;;
17   relationship: many_to_one
18 }
19 }
20 }
```

When **Content Breaks**



Finding Broken Content

Identify Looks
and Dashboard
Tiles that are no
longer working by
navigating to the
Content Validator



You are in **Development Mode**.

looker

Browse ▾ Explore ▾ Develop ▾ Admin ▾

Development Mode **ON**

SQL Runner

Content Validator

Manage LookML Projects

clothing_store

e_flights

imdb

intro_to_looker

thelook

Explore

Order Items

Search

All Fields Dimensions Measures

- ▶ Inventory Items
- ▶ Order Items
- ▶ Orders
- ▶ Products
- ▶ Users

Using the Content Validator

Validation runs automatically when you start the Content Validator

Results show Looks and tiles that reference LookML objects that do not exist or cannot be found:

- Model names
- Explore names
- View names
- Field names

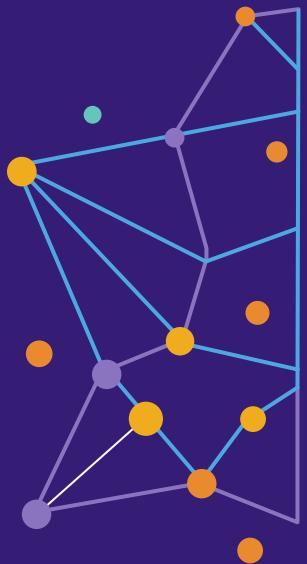
This is often due to a typo or a missing join

Fixing Broken Looks & Tiles

Group by tabs in the Content Validator impact how errors are grouped

- **Error:** List each error, grouping together Looks and tiles that have error, useful if you want to fix the same error in multiple Looks and tiles at once
- **Folder:** List each folder, grouping together its errored Looks and tiles, useful if you want to fix only the errors in a particular folder
- **Look/Tile:** List each Look and tile with errors, grouping them together, useful if you want to fix all of the errors in a single Look or tile

Use the Replace or Remove buttons to fix the errors described within the validator (*typically to replace an object that has been renamed or deleted from the model*)



Best Practices

Create a Positive Experience for Looker Users

Provide users with meaningful field names

Group similar fields together for easier navigation

Less is more

Add context to help users understand what fields and Explores to use

Build common workflows into Looker

Provide Users with Meaningful Field Names

Use the label parameter to apply friendly names to dimensions or measures for end users while maintaining database-friendly names within view and model files

Avoid exposing multiple fields with the same name - For example, count measures are automatically created within Looker with the name "Count". This results in most view files containing a count measure with the same name. Other culprits include created dates and updated dates

Provide Users with Meaningful Field Names

Provide clear names for yesno fields - For example, use “Is Returned” instead of “Returned” for naming a field that indicates whether an item has been returned.

Name ratios descriptively - For example, “Orders Per Purchasing Customers” is clearer than “Orders Percent”.

Name fields consistently and represent values consistently across the model - Use value formats such as currency symbols, percentages, and decimal precision to help make everything readable

Group Similar Fields Together for Easier Nav

Use the group_label and view_label parameters to consolidate dimensions and measures from individual or multiple joined views that are related - For example, grouping all geographic information into a “Geography” group will pull all address and location information together within the field picker rather than having it all listed alphabetically

Group Similar Fields Together for Easier Nav

Break up large, denormalized tables using the view_label parameter - Utilize the `view_label` parameter within fields in order to logically group fields together into separate headings within the field picker

Large, denormalized tables with a lot of fields can be difficult to navigate, so this gives the illusion of multiple views to users

Less is More

Avoid exposing too much to users upon an initial Looker roll-out -
Start small, and then expand the options

Hide dimensions that are not relevant to users from the user interface - Use the hidden parameter on dimensions that will never be used through the user interface

Utilize the fields parameter within Explores and joins to limit the number of fields available to users - Unlike the hidden parameter, this enables fields to be included or excluded on an Explore by Explore basis

Less is More

Hide any Explores that exist solely for populating specific Looks, dashboard tiles, or filters

Use the fewest number of Explores possible that allows users to easily get access to the answers they need - Consider splitting out Explores into different models for different audiences to limit the options available for each user group

Consider using the group_label parameter for Explores within a model in order to group them in a sensible way within the Explore dropdown menu

Add Context on What Fields & Explores to Use

Use the description parameter on dimensions and measures to provide additional information to end users on the logic or calculations utilized within the model. This is particularly important for dimensions and measures that leverage complex logic or calculations

However, descriptions should also be considered for ensuring that users understand the definitions behind simpler fields

Explore descriptions should be defined for users. Add a short description to each Explore to specify the purpose and audience using the description parameter

Build Common Workflows into Looker

Add `drill_fields` to all relevant measures - Drill fields enable users to click into aggregate values in order to access detailed data. Utilize sets to create reusable sets of fields that can then be applied to any number of measures within a view

Add drill-fields to all hierarchical dimensions - For example, adding a drill field for City into a State dimension will enable users to select a state and then drill deeper into the cities within that state

Note that this hierarchical drilling will automatically be applied within time dimension groups

Build Common Workflows into Looker

Set up links that enable users to easily navigate and pass filters to other Looker dashboards or to systems or platforms external to Looker

LookML Do's

Do define the relationship parameter for all joins - This will ensure that metrics aggregate properly within Looker. By default, Looker will utilize many_to_one join relationships for any joins in which a relationship is not defined

Do define a primary key within each and every views, including derived tables - All views, whether from the database directly or derived, should contain a primary key. This primary key should be a unique value that would enable Looker to uniquely identify any given record

LookML Do's

Do name LookML objects using all lowercase letters and underscores for spaces - The label parameter can be used for additional formatting of a name field

Do utilize datagroups for aligning PDT generation and Explore caching with underlying ETL processes - These can also be used in conjunction with schedules for pre-populating cache

LookML Don'ts

Don't use the from parameter for renaming views within an Explore - Use the view_label parameter instead

Use the from parameter in the following situations:

- Polymorphic joins (joining the same table multiple times such as an airport table that is joined on the destination airport code and the departing airport code)
- Self-joins (joining a table to itself such as an employee table that is joined to itself to create a relationship between employees and managers)
- Re-scoping an extended view back to its original view name

LookML Don'ts

Don't use the words "date" or "time" in a dimension group name -
Looker appends each timeframe to the end of the dimension group name

Don't use formatted timestamps within joins - Instead, use the raw timeframe option for joining on any date or time fields. This will avoid the inclusion of casting and timezone conversion in join predicates

Write Sustainable, Maintainable LookML

Substitution operators should be utilized throughout all LookML files

A LookML model should only have a single reference point to any object in the physical data model

Think of LookML objects as building blocks, and use Extends to combine objects together in different ways without repeating code

Utilize sets for maintaining reusable field lists within the model

Write Sustainable, Maintainable LookML

Define custom map layers centrally in a LookML file that can be included as needed across models

Set any custom value formats centrally within the model. Utilize the named_value_format parameter to set any custom formats within the model and then reference those using the value_format_name parameter in dimensions and measures

Define development guidelines to make it easier to develop and scale a LookML model

Optimize Looker Query Performance

Ensure that queries are built and executed optimally against your database

Utilize cache whenever possible to reduce database query traffic

- Build Explores using many_to_one joins whenever possible.
- Avoid joining views into Explores on concatenated primary keys declared in Looker
- Maximize cache usage by applying datagroups within Explores using the persist_with parameter

Optimize Looker Query Performance

Ensure that queries are built and executed optimally against your database

Utilize cache whenever possible to reduce database query traffic

- **Warm the cache for popular dashboards using schedules with datagroups** - When the dashboard runs via the schedule, all looks and tiles will be run, causing underlying queries to be loaded into the cache
- **Use the dashboard auto-refresh feature strategically** - If a dashboard uses auto-refresh, make sure it refreshes no faster than the ETL processes running behind the scenes

Optimize Looker Server Performance

Ensure that Looker server and application are performing optimally

- **Limit the number of elements within an individual dashboard** - There is no hard and fast rule to the number, since the design of the element impacts memory consumption based on a variety of factors, but dashboards with 25 or more tiles tend to be problematic when it comes to dashboard performance
- **Avoid over-using pivots within Tiles and Looks** - Queries with pivoted dimensions will consume more memory

Optimize Looker Server Performance

Ensure that Looker server and application are performing optimally

- **Limit the number of views included within a model when a large number of view files are present** - Including all views in a single model can slow performance
- **Avoid returning a large number of data points by default within dashboard Looks and Tiles** - Data volume will impact performance the most; the more data returning in an individual element, the more memory resources are consumed
- Ensure that appropriate filters and default filters are applied

Keep Looker Secure - Platform Security

Set up user authentication using either Looker's native username/password option or, preferably, using a more robust authentication mechanism like 2FA, LDAP, Google OAuth, or SAML

Never post API credentials publicly

Regularly audit any public access links your users create, and restrict the permission to create them, as necessary

Keep Looker Secure - Platform Security

Set up the most restrictive user permissions and content access that still allow people to carry out their work, paying special attention to who has admin privileges

Use the `access_filter` parameter in conjunction with user attributes for applying row-level data security by user or user group

Use model sets for applying data set security within the Looker model

Avoid assigning users to roles individually whenever possible - Assign roles based on group membership whenever possible

Keep Looker Secure - Content Security

Implement the appropriate type of Looker system based on the access controls that you need to support

- Completely Open
- Open With Content Restrictions
- Closed System

Keep Looker Secure - Content Security

Most users should be Viewers of the Shared space (*except in closed system setups*)

This ensures that users will be able to see content in the Shared spaces and any subspaces to which they have access

Providing users with Manage Access, Edit permissions on the Shared folders means they will retain those rights within any/all subfolders

It can be helpful to make changes lower in the tree first, and THEN revoke access higher up

Using (Or Not Using) Derived Tables

Distribution and sort keys (*Redshift only*) or indexes should be added to all persisted tables

Try to model data without using derived tables

Often a derived table is very useful for certain analysis, but they will also make the resulting Explore less flexible

Using (Or Not Using) Derived Tables

Common derived table use cases

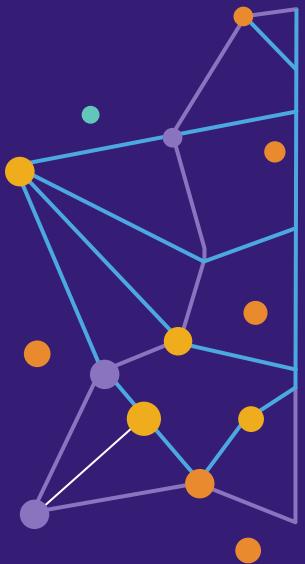
- Dimensionalizing a measure i.e. using an aggregation within a derived table so it can be used as a dimension within the LookML
- Aggregating the data to a different level of granularity
- Pre-joining data in order to increase query performance
- Adding primary keys into tables

Using (Or Not Using) Derived Tables

Scenarios in which persistent derived tables (PDTs) should typically not be used

- The cost and time involved in creating PDTs often enough is too high
- Filtering source tables

If derived views are largely being used for transforming data from the database, consider migrating these transformations into the ETL process



Questions?

THANK YOU!