ALIBABA CLOUD

Alibaba Cloud

Hologres

Best Practices

Document Version: 20220711

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Data warehouse construction 05
1.1. Use spatial functions to query data 05
2.Authorization 15
2.1. Authorize roles based on PostgreSQL privileges 15
3.Scenario Scheme 21
3.1. Recommended data warehouse layering solutions 21
3.2. Real-time report analysis 24
3.2.1. Build a real-time data warehouse and display data ana... 24
3.2.2. Analyze large amounts of MaxCompute data in real ti... 29
3.3. User behavior analysis 34
3.3.1. Overview 34
3.3.2. Batch UV calculation 36
3.3.3. Remove duplicate UVs in real time 43
3.4. User profile analysis 53
3.4.1. User profile analysis 53
3.4.2. Wide tables 56
3.4.3. Roaring bitmaps 57
3.4.4. Real-time tags 64

1.Data warehouse construction

1.1. Use spatial functions to query data

Hologres allows you to use PostGIS spatial functions to query spatial data in tables. This topic shows you how to import data from an on-premises machine to Hologres and then use PostGIS spatial functions to query data in Hologres.

Prerequisites

- A Hologres instance is purchased. For more information, see Purchase a Hologres instance.
- A database is created in the Hologres instance. For more information, see Create a database.
- The sample spatial data used in this topic is downloaded. To download the data, click the following links:
  - The accommodations table
  - The zipcodes table

Context

The demo tables used in this topic contain various types of spatial data, such as longitudes, latitudes, coordinates, and distances. You can perform Step 1 and Step 2 to use HoloWeb to import the downloaded sample spatial data from your on-premises machine to the **accommodations** and **zipcodes** Hologres tables that you create. Then, you can perform Step 3 to use PostGIS spatial functions to query the spatial data in the two Hologres tables.

PostGIS: PostGIS is a spatial database extension for PostgreSQL databases. Hologres is compatible with the protocols of PostgreSQL 11. You can use PostGIS spatial functions in Hologres.

Spatial functions: For more information, see Spatial functions.

The Hologres tables to be created by using HoloWeb are used to store the following data:

- The **accommodations** Hologres table stores the accommodation information of Berlin, such as the geographical location, including the longitude and latitude, and the name of each accommodation.
- The **zipcodes** Hologres table stores the ZIP codes in Berlin.

Procedure

Step	Description
Step 1: Create Hologres tables	Create the following two tables in the database in your Hologres instance: 1. The **accommodations** Hologres table that is used to store the accommodation information of Berlin, such as the geographical location, including the longitude and latitude, and the name of each accommodation. 2. The **zipcodes** Hologres table that is used to store the ZIP codes in Berlin.
Step 2: Import the sample spatial data	Use HoloWeb to import spatial data from your on-premises machine to the **accommodations** and **zipcodes** Hologres tables.

Step	Description
Step 3: Use spatial functions to query data	Use PostGIS spatial functions to query the spatial data in the two Hologres tables.

Step 1: Create Hologres tables

Perform the following operations to create the **accommodations** and **zipcodes** Hologres tables.

1. Log on to the HoloWeb console and go to the SQL Editor tab.
2. Click **Ad-hoc Query**. Select your Hologres instance from the Instance drop-down list and the database where you want to create Hologres tables from the Database drop-down list.
3. Install the PostGIS extension.

   Enter the following statement in the SQL editor and click **Run**:

   ```
   create extension if not exists postgis; -- Install the PostGIS extension.
   ```

4. Create the **accommodations** Hologres table.

   Execute the following SQL statements to create the **accommodations** Hologres table. The table is used to store the accommodation information of Berlin, such as the geographical location, including the longitude and latitude, and the name of each accommodation.

   > **Note** In the **Table Directory** section, you can click the **Refresh** icon and choose **public > Tables** to check whether the Hologres table is created. Alternatively, you can check the operational log in the **Run Log** section.

   ```
   CREATE TABLE public.accommodations (
     id INTEGER PRIMARY KEY,
     shape GEOMETRY,
     name VARCHAR(100),
     host_name VARCHAR(100),
     neighbourhood_group VARCHAR(100),
     neighbourhood VARCHAR(100),
     room_type VARCHAR(100),
     price SMALLINT,
     minimum_nights SMALLINT,
     number_of_reviews SMALLINT,
     last_review DATE,
     reviews_per_month NUMERIC(8,2),
     calculated_host_listings_count SMALLINT,
     availability_365 SMALLINT
   );
   ```

5. Create the **zipcodes** Hologres table.

   Execute the following SQL statements to create the **zipcodes** Hologres table. The table is used to store the ZIP codes in Berlin.

   > **Note** In the **Table Directory** section, you can click the **Refresh** icon and choose **public > Tables** to check whether the Hologres table is created. Alternatively, you can check the operational log in the **Run Log** section.

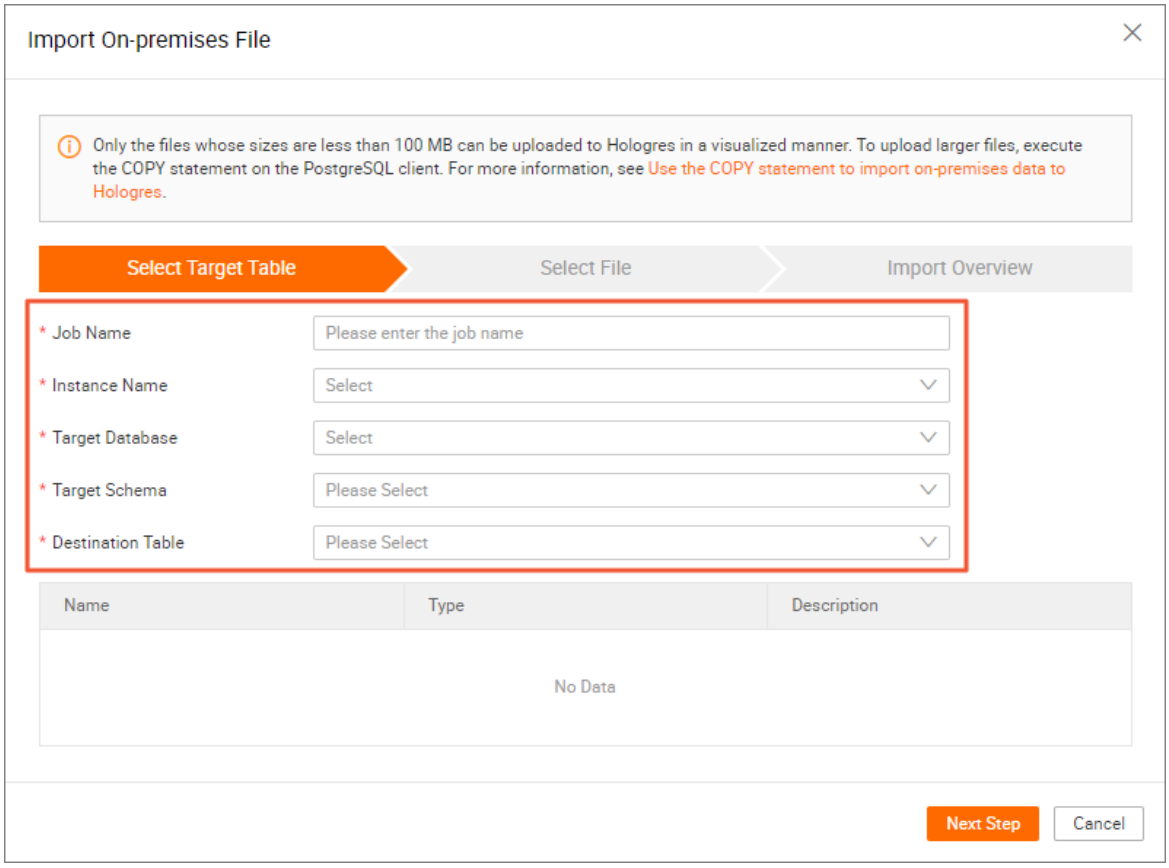
```
CREATE TABLE public.zipcode (
  ogc_field INTEGER PRIMARY KEY NOT NULL,
  wkb_geometry GEOMETRY,
  gml_id VARCHAR(256),
  spatial_name VARCHAR(256),
  spatial_alias VARCHAR(256),
  spatial_type VARCHAR(256)
 );
```

Step 2: Import the sample spatial data

After the **accommodations** and **zipcodes** Hologres tables are created, import the downloaded sample spatial data from your on-premises machine to the two tables on the **Import On-premises File** page.

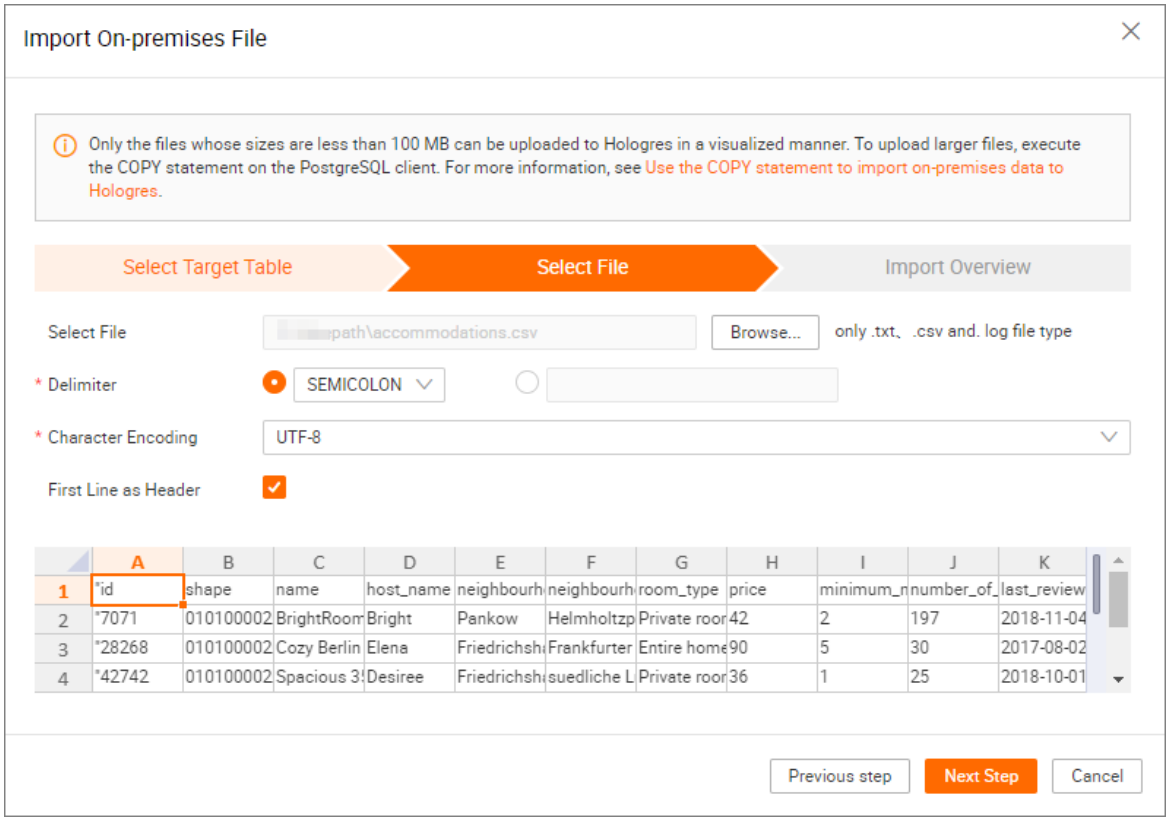
1. In the HoloWeb console, click **Data Solution** in the top navigation bar.
2. On the **Data Solution** tab, choose **Import On-premises File** in the left-side navigation pane. Then, click **New data import** on this page.
3. Specify the Hologres table to which you want to import data.

   In the **Import On-premises File** dialog box, enter a job name, select your Hologres instance, the created database, and a created Hologres table (**accommodations** or **zipcodes**), and then click **Next Step**.



4. Specify the data to be imported and the encoding format.

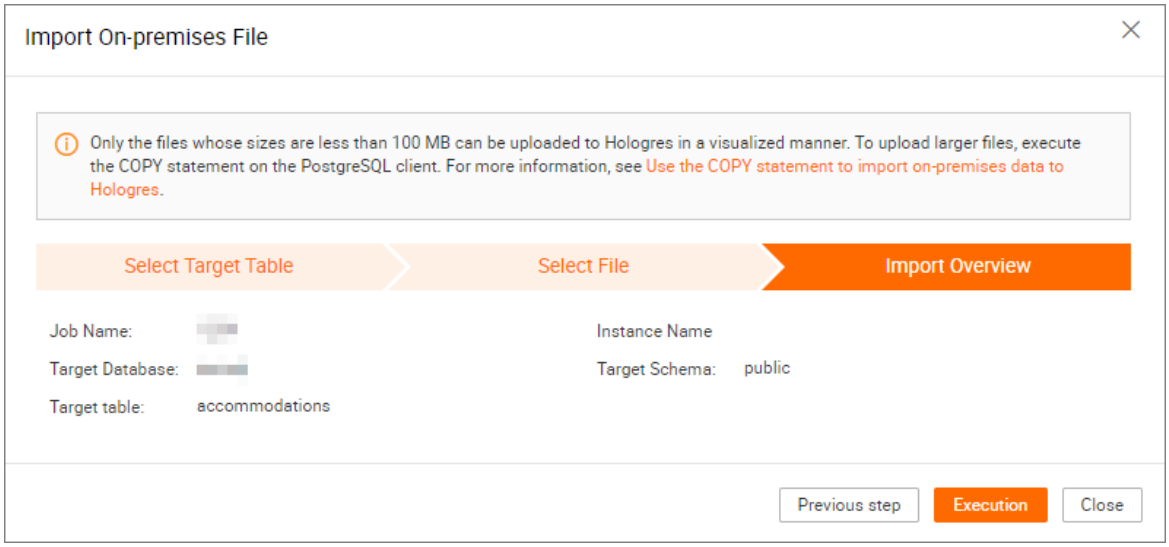
   In the **Select File** step, set the parameters as described in the following table and click Next Step.



Parameter	Description
Select File	The file that contains the data to be imported. Click **Browse...** and select a file from your on-premises machine. .txt, .csv, and .log files are supported. In this example, select the **accommodations** or **zipcodes** table that you downloaded.
Delimiter	The delimiter used to separate data entries. In this example, select SEMICOLON. **Note** You can also select the option to the right of the drop-down list and specify a custom delimiter based on your business requirements.
Character Encoding	The encoding format of the data. In this example, select **UTF-8**.
First Line as Header	Specifies whether to set the first line of the data as the header of the Hologres table. By default, this option is not selected.

5. Confirm the configurations.

   In the Import Overview step, check whether the configurations of the data import job are as expected and click **Execution**.



6. Verify the execution result.

   After the job is complete, the system shows whether the execution is successful in the Import Overview step. If the execution fails, you can view the error cause for troubleshooting and import the data again.

   You can also execute one of the following SQL statements in the SQL editor to query the number of data entries or detailed data in the destination Hologres table:

   - Query the number of data entries

     In this example, the **accommodations** Hologres table contains 22,248 data entries, and the **zipcodes** Hologres table contains 190 data entries.

     ```
     select count(*) from accommodations; -- Query the number of data entries in the accommodations Hologres table.
     select count(*) from zipcodes; -- Query the number of data entries in the zipcodes Hologres table.
     ```

   - Query the detailed data

     ```
     select * from accommodations; -- Query the detailed data in the accommodations Hologres table.
     select * from zipcodes; -- Query the detailed data in the zipcodes Hologres table.
     ```

Step 3: Use spatial functions to query data

After the required Hologres tables are created and the sample spatial data is imported to the tables, you can use spatial functions to query the spatial data in Hologres. The following examples are for your reference. For information about the syntax of spatial functions, see Spatial functions.

- Query the number of data entries in the **accommodations** Hologres table with the spatial reference system identifier (SRID) set to 4326.
  - Sample code:

    ```
    SELECT count(*) FROM public.accommodations WHERE ST_SRID(shape) = 4326;
    ```

- Return results:

```
 count
-------
 22248
(1 row)
```

- Use the well-known text (WKT) format to query geometry objects that meet the specified conditions. In this example, you can check whether the ZIP codes in the zipcodes Hologres table are stored in World Geodetic System 1984 (WGS84). The system uses an SRID of 4326.

  > **Note** Only the spatial data entries that are in the same spatial reference system can be referenced by each other.

  - Sample code:

```
SELECT  ogc_field
        ,spatial_name
        ,spatial_type
        ,ST_SRID(wkb_geometry)
        ,ST_AsText(wkb_geometry)
FROM    public.zipcode
ORDER BY spatial_name
;
```

  - Return results:

```
ogc_field  spatial_name  spatial_type  st_srid  st_astext
-----------------------------------------------------------------
0          10115         Polygon        4326     POLYGON((...))
4          10117         Polygon        4326     POLYGON((...))
8          10119         Polygon        4326     POLYGON((...))
...
(190 rows returned)
```

- Use the GeoJSON format to query the surface, the surface size, and the number of points on the surface for Mitte in Berlin with the SRID set to 10117.
  - Sample code:

```
SELECT  ogc_field
        ,spatial_name
        ,ST_AsGeoJSON(wkb_geometry)
        ,ST_Dimension(wkb_geometry)
        ,ST_NPoints(wkb_geometry)
FROM    public.zipcode
WHERE   spatial_name = '10117'
;
```

- Return results:

```
ogc_field  spatial_name  spatial_type                                    st_dimension  s
t_npoint
-------------------------------------------------------------------------------------
--------
4           10117         {"type":"Polygon", "coordinates":[[[...]]]}    2
331
```

- Query the number of accommodations within 500 meters of the Brandenburg Gate with the SRID set to 4326.
  - Sample code:

```
SELECT  COUNT(*)
FROM    public.accommodations
WHERE   ST_DistanceSphere(shape, ST_GeomFromText('POINT(13.377704 52.516431)', 4326)) <
500
;
```

  - Return results:

```
 count
-------
    29
(1 row)
```

- Perform a rough estimate of the location of the Brandenburg Gate based on the information about nearby accommodations.
  - Sample code:

```
WITH
    poi(loc) AS (
        SELECT st_astext(shape)
        FROM accommodations
        WHERE name LIKE '%brandenburg gate%' )
SELECT  COUNT(*)
FROM    accommodations a
        ,poi p
WHERE   ST_DistanceSphere(a.shape, ST_GeomFromText(p.loc, 4326)) < 500
;
```

  - Return results:

```
 count
-------
    60
(1 row)
```

- Query the detailed information about all the accommodations around the Brandenburg Gate and sort the accommodations in descending order by price.

- Sample code:

```
SELECT  name
        ,price
        ,ST_AsText(shape)
FROM    public.accommodations
WHERE   ST_DistanceSphere(shape, ST_GeomFromText('POINT(13.377704 52.516431)', 4326)) <
500
ORDER BY price DESC
;
```

- Return results:

```
                        name                         | price |                st_astext

-----------------------------------------------------+-------+--------------------------
----------------
 DUPLEX APARTMENT/PENTHOUSE in 5* LOCATION! 7583     |   300 | POINT(13.3826510209548 52
.5159819722552)
 DUPLEX-PENTHOUSE IN FIRST LOCATION! 7582            |   300 | POINT(13.3799997083855 52
.5135918444834)
 Luxury Apartment in Berlin Mitte with View          |   259 | POINT(13.3835653528534 52
.516360156825)
 BIG APT 4 BLNCTY-CNTR 43-H6                         |   240 | POINT(13.3800222998777 52
.5134224506894)
 BIG APARTMENT-PRIME LOCATION-BEST PRICE! B0303      |   240 | POINT(13.379745196599 52.
5162648947249)
 BIG APARTMENT IN BRILLIANT LOCATION-CTY CENTRE B53 |   240 | POINT(13.381383105167 52.
5157082721072)
 SONYCENTER: lux apartment - 3room/2bath. WIFI       |   235 | POINT(13.3743158954191 52
.5125308432819)
 CENTRE APARTMENT FOR 6 | 8853                       |   220 | POINT(13.3819039478615 52
.5134866767369)
 BIG APARTMENT FOR 6 - BEST LOCATION 8863            |   209 | POINT(13.3830430841658 52
.5147824286783)
 3 ROOMS ONE AMAZING EXPERIENCE! 8762                |   190 | POINT(13.3819898503053 52
.5144190764637)
 AAA LOCATION IN THE CENTRE H681                     |   170 | POINT(13.3821787206534 52
.5129769242004)
 H672 Nice Apartment in CENTRAL LOCATION!            |   170 | POINT(13.3803137710339 52
.5132386929089)
 "Best View -best location!"                         |   170 | POINT(13.3799551247135 52
.5147888483851)
 H652 Best Location for 4!                           |   170 | POINT(13.3805705422409 52
.5143845784482)
 H651 FIT´s for Four in a 5* Location!               |   150 | POINT(13.3822063502184 52
.5134994650996)
 NEXT TO ATTRACTIONS! H252                           |   110 | POINT(13.3823616629115 52
.5136258446666)
 CTY Centre Students Home| G4                        |   101 | POINT(13.3808081476226 52
.5130957830586)
 Room for two with private shower / WC               |    99 | POINT(13.3786877948382 52
.5208018292043)
 StudentsHome CityCentre Mitte 91-0703               |    95 | POINT(13.3810390515141 52
.5142363781923)
```

```
 FIRST LOCATION - FAIR PRICE K621                    |    80 | POINT(13.3823909855061 52
.5131554670458)
 LONG STAY FOR EXPATS/STUDENTS- CITY CENTRE | K921  |    75 | POINT(13.380320945399 52.
512364557598)
 Nice4Students! City Centre 8732                     |    68 | POINT(13.3810147526683 52
.5136623602892)
 Comfy Room in the heart of Berlin                   |    59 | POINT(13.3813167311819 52
.5127345388756)
 FO(U)R STUDENTS HOME-Best centre Location!          |    57 | POINT(13.380850032042 52.
5131726958513)
 Berlin Center Brandenburg Gate !!!                  |    55 | POINT(13.3849641540689 52
.5163902851474)
 !!! BERLIN CENTER BRANDENBURG GATE                  |    55 | POINT(13.379997730927 52.
5127577639174)
 Superb Double Bedroom in Central Berlin             |    52 | POINT(13.3792991992688 52
.5156572293422)
 OMG! That's so Berlin!                             |    49 | POINT(13.3754883007165 52.
5153487677272)
 Apartment in Berlin's old city center               |    49 | POINT(13.3821761577766 52
.514037240604)
(29 rows)
```

- Query the detailed information about the accommodation with the highest price and its ZIP code.
  - Sample code:

```
SELECT  a.price
        ,a.name
        ,ST_AsText(a.shape)
        ,z.spatial_name
        ,ST_AsText(z.wkb_geometry)
FROM    accommodations a
        ,zipcode z
WHERE   price = 9000
AND     ST_Within(a.shape, z.wkb_geometry)
;
```

  - Return results:

```
price   name                                st_astext
spatial_name      st_astext
-------------------------------------------------------------------------------------------
-----------------------------------------------------------
9000    Ueber den Dächern Berlins Zentrum    POINT(13.334436985013 52.4979779501538)
10777             POLYGON((13.3318284987227 52.4956021172799,...
```

- Query the popular accommodations in Berlin, group the accommodations by ZIP code, and then sort the groups by the order volume.

- Sample code:

```
SELECT  z.spatial_name AS zip
        ,COUNT(*) AS numAccommodations
FROM    public.accommodations a
        ,public.zipcode z
WHERE   ST_Within(a.shape, z.wkb_geometry)
GROUP BY zip
ORDER BY numAccommodations DESC
;
```

- Return results:

```
zip      numaccommodations
---------------------------
10245    872
10247    832
10437    733
10115    664
...
(187 rows returned)
```

2.Authorization

2.1. Authorize roles based on PostgreSQL privileges

This topic provides the best practices for Hologres when you authorize roles based on PostgreSQL privileges. This way, you can simplify authorization and manage privileges in a fine-grained manner.

Context

Hologres is compatible with PostgreSQL and supports authorization based on PostgreSQL privileges. Hologres also provides an authorization method called Simple Permission Model (SPM). For more information, see Overview.

However, SPM manages privileges in a coarse-grained manner. If you need to manage privileges in a fine-grained manner, see the "Best practice 1" and "Best practice 2" sections.

Overview of PostgreSQL privileges

For information about PostgreSQL privileges, see 5.7. Privileges.

PostgreSQL privileges have the following limits:

- PostgreSQL privileges apply only to *the existing objects and do not apply to new objects.* Example:
  i. User1 executes the `GRANT SELECT ON ALL TABLES IN SCHEMA public TO User2;` statement to authorize User2 to select all the tables in the **public** schema.
  ii. User1 creates a table named **table_new** in the *public* schema.
  iii. A `Permission denied` error occurs when User2 executes the `SELECT * FROM table_new` statement.

  The SELECT privilege that User1 grants to User2 apply only to the existing tables in the **public** schema and do not apply to new tables in the **public** schema. Therefore, the preceding error occurred.
- You can execute the `ALTER DEFAULT PRIVILEGES` statement to grant default privileges on the objects created in the future to all the roles. For more information, see ALTER DEFAULT PRIVILEGES. The default privileges apply only to the objects created in the future. The following statement is used as an example:

  ```
  ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO PUBLIC; // By default
  , all the roles can read the new tables in the public schema.
  ```

  You can also execute the `ALTER DEFAULTPRIVILEGES FOR ROLE xxx` statement to grant default privileges on new objects to xxx. The default privileges can be granted only when the current role and *xxx* meet one of the following requirements:
  - The current user is a member of the permission group to which *xxx* belongs.
  - The current role is a superuser. *xxx* can be either a role or a permission group.

  You can use the `\ddp` command to check whether the `ALTER DEFAULT PRIVILEGES` statement takes effect. Default privileges are stored in the `pg_catalog.pg_default_acl` catalog.

`ALTER DEFAULT PRIVILEGES` serves as a trigger. When you create a table, Hologres compares the table and the `pg_catalog.pg_default_acl` catalog based on the current role and schema information. If matches are found, the corresponding match rules are added.

> **Note**
>
> - Only the current user can be used for comparison. The permission group to which the current role belongs cannot be used for comparison.
> - The `ALTER DEFAULT PRIVILEGES` can be executed only when you create a table. If you execute the `ALTER TABLE <TABLE> OWNER TO XX` statement after you create a table, the `ALTER DEFAULT PRIVILEGES` statement is not executed.
>
> Assume that User1 belongs to Group1 and you want to grant privileges to a table created in the future and compare the table and Group1. The following results are obtained:
>
> - If the current role is User1, no matches are found during the comparison.
> - If you execute the `SET SESSION ROLE Group1` statement to change the current role to Group1 before you create a table, matches are found during the comparison. Then, the privileges are automatically granted to the table.

- Only the table owner can delete a table.

  You can decide whether the table can be deleted based on the current role. Only the following roles have the DELETE privilege:

  - The owner of the table.
  - The owner of the schema to which the table belongs.
  - Superuser

- By default, the user who creates a table is the owner of the table. The user has all the privileges on the table, including the DELETE privilege.

  The following sample statements are used to assign a table to a new owner:

  ```
  alter table <table> owner to ; // Assign the table to User2.
  alter table <table> owner to GROUP1; // Assign the table to GROUP1.
  ```

  The following limits apply when a table is assigned to a new owner:

  - User1 is the owner of the table.
  - User1 must directly or indirectly belong to Group1.

    For example, User1 is a member of Group1, or User1 is a member of a group in Group1.
  - Group1 must have the USAGE privilege on the schema.
  - A superuser can assign a table to a new owner.

Assign privileges

You must assign the following items before you manage privileges:

- The total number of permission groups.
- The privileges of the permission groups.
- The roles in each permission group.
- The roles that can delete tables and the time that tables can be deleted.

- The schemas to which the permission groups respectively belong.

We recommend that you perform the following operations before you manage privileges:

- Create permission groups and assign their privileges.

  Permission groups are divided into the following types:

  - XX_DEV_GROUP: the owner of a table. The owner has all the privileges on the table.
  - XX_WRITE_GROUP: the privileges to write data to a table.
  - XX_VIEW_GROUP: the privileges to view data in a table.

  XX indicates a project. For example, the permission groups of the PROJ1 project include PROJ1_DEV_GROUP, PROJ1_WRITE_GROUP, and PROJ1_VIEW_GROUP.

  > **Note** The preceding naming formats are only for reference.

- Assign schemas for the permission groups.

  We recommend that you assign a schema for the permission groups of a project.

  Each permission group can own multiple tables. However, each table can belong to only one permission group. For example, TABLE1 can belong only to PROJ1_DEV_GROUP.

  Each role can belong to multiple permission groups. For example, USER1 can belong to PROJ1_DEV_GROUP and PROJ2_DEV_GROUP.

Best practice 1

A table is used as an example.

All the members in the permission group where the owner belongs can manage or delete the table.

Perform the following operations to add a role to the permission group where the owner belongs:

1. Create permission groups.

   You can create permission groups based on your needs. Assume that the project is named PROJ1. The following statements are used as an example:

   ```
   create role PROJ1_DEV_GROUP; // The owner of the table. The owner has all the privilege
   s on the table.
   create role PROJ1_WRITE_GROUP; // The privileges to write data to the table.
   create role PROJ1_VIEW_GROUP; // The privileges to view data in the table.
   ```

2. Grant privileges to the schemas.

   You must grant privileges to the schemas to which the permission groups belong. Assume that PROJ1 belongs to schema1. The following statements are used as an example:

   ```
   Grant all the privileges of schema1 to PROJ1.
   grant create,usage on schema SCHEMA1 to PROJ1_DEV_GROUP;
   grant usage on schema SCHEMA1 to PROJ1_WRITE_GROUP;
   grant usage on schema SCHEMA1 to PROJ1_VIEW_GROUP;
   ```

> **Note**
>
> - Each project can belong to multiple schemas. Each schema can have multiple projects.
> - By default, all the roles in the **public** schema have the CREATE privilege and the USAGE privilege.

3. Create roles and manage the permission groups.

   After you grant privileges to the permission groups as a superuser, you must create and add a role to the permission groups. The following statements are used as an example:

```
create user "USER1";
grant PROJ1_DEV_GROUP to "USER1";
create user "USER2";
grant PROJ1_VIEW_GROUP to "USER2";
```

4. Create a table and grant the privileges on the table to the roles.

   When a table is created, the owner or a superuser must grant the privileges on the table to the roles. Take note that the owner must be a member of PROJ1_DEVE_GROUP. The following statements are used as an example:

```
grant all on table SCHEMA1.TABLE1 to PROJ1_WRITE_GROUP; // Grant PROJ1_WRITE_GROUP the
privileges to write data to table1.
grant select on table SCHEMA1.TABLE1 to PROJ1_VIEW_GROUP; // Grant PROJ1 _VIEW_GROUP th
e SELECT privilege.
alter table SCHEMA1.TABLE1 owner to PROJ1_DEV_GROUP; // Assign TABLE1 to PROJ1_DEV_GROU
P.
```

Best practice 2

The `ALTER DEFAULT PRIVILEGES` statement is used in this example.

Perform the following operations to assign a table to a new owner or change the project to which a role belongs:

1. Create permission groups.

   You can create permission groups based on your needs. Assume that the project is named PROJ1. The following statements are used as an example:

```
create role PROJ1_DEV_GROUP; // The owner of the table. The owner has all the privilege
s on the table.
create role PROJ1_WRITE_GROUP; // The privileges to write data to the table.
create role PROJ1_VIEW_GROUP; // The privileges to view data in the table.
```

2. Grant privileges to the schemas.

   You must grant privileges to the schemas to which the permission groups belong. Assume that PROJ1 belongs to schema1. The following statements are used as an example:

```
Grant all the privileges of schema1 to PROJ1.
grant create,usage on schema SCHEMA1 to PROJ1_DEV_GROUP;
grant usage on schema SCHEMA1 to PROJ1_WRITE_GROUP;
grant usage on schema SCHEMA1 to PROJ1_VIEW_GROUP;
```

> **Note**
>
> - Each project can belong to multiple schemas. Each schema can have multiple projects.
> - By default, all the roles in the **public** schema have the CREATE privilege and the USAGE privilege.

3. Create roles and grant default privileges to the roles.

   After privileges are granted to the schemas, a superuser needs to create roles and add the roles to the permission groups. The superuser also needs to grant default privileges to the roles.

   The following statements are used as an example:

   ```
   create user "USER1";
   alter default privileges for role "USER1" grant all on tables to PROJ1_DEV_GROUP; // Gr
   ant PROJ1_DEV_GROUP the default privileges on the table created by USER1.
   alter default privileges for role "USER1" grant all on tables to PROJ1_WRITE_GROUP; //
   Grant PROJ1_WRITE_GROUP the default privileges on the table created by USER1.
   alter default privileges for role "USER1" grant select on tables to PROJ1_VIEW_GROUP; /
   / Grant PROJ1_VIEW_GROUP the default privileges on the table created by USER1.
   grant PROJ1_DEV_GROUP to "USER1"; // Add USER1 to PROJ1_DEV_GROUP.
   ```

4. Assign the table to a new owner.

   If you want to authorize other members of PROJ1_DEV_GROUP to perform operations on the table, you can assign the table to PROJ1_DEV_GROUP.

   The ALTER TABLE <TABLE> OWNER TO PROJ1_DEV_GROUP statement must be executed by a superuser. Assume that the table is named TABLE1. The following statement is used as an example:

   ```
   alter table SCHEMA1.TABLE1 owner to PROJ1_DEV_GROUP; // Assign TABLE1 to PROJ1_DEV_GROU
   P.
   ```

   A table can be assigned to a new owner when the following requirements are met:

   - The table is newly created and a superuser modifies the owner on a regular basis.
   - The table is assigned to a new owner before operations are performed on it.
   - If the table is modified or deleted by the owner or a superuser, you do not need to execute the preceding statements.

5. Change the default project to which a role belongs.

   Only the owner or the role can execute the `ALTER DEFAULT PRIVILEGE` statement to revoke the default privileges granted to the current project. Then the owner or the role execute the `ALTER DEFAULT PRIVILEGE` statement again to grant default privileges to another project.

   If the project is changed, the table is not affected. The following statements are used as an example:

```
Disable the default privileges granted to the current project.
alter default privileges for role "USER1" revoke all on tables from PROJ1_DEV_GROUP;
alter default privileges for role "USER1" revoke all on tables from PROJ1_WRITE_GROUP;
alter default privileges for role "USER1" revoke select on tables from PROJ1_VIEW_GROUP
;
Grant default privileges to another project.
alter default privileges for role "USER1" grant all on tables to PROJ2_DEV_GROUP;
alter default privileges for role "USER1" grant all on tables to PROJ2_WRITE_GROUP;
alter default privileges for role "USER1" grant select on tables to PROJ2_VIEW_GROUP;
```

3.Scenario Scheme

3.1. Recommended data warehouse layering solutions

This topic describes the best practices for data warehouse layering in Hologres. You can use these best practices to develop your business by using real-time data warehouses that feature high performance and agility.

Context

Hologres is highly compatible with Realtime Compute for Apache Flink, MaxCompute, and DataWorks, and provides data warehousing solutions that integrate stream processing and batch processing. These solutions are applicable to a wide range of scenarios, such as real-time dashboards, real-time risk control, and fine-grained operations. Different scenarios pose varied requirements for the amount of data to be processed, data complexity, data sources, and real-time performance. To develop a traditional data warehouse based on the classic methodology, you need to develop the following layers in sequence: `Operational Data Store (ODS), Data Warehouse Detail (DWD), Data Warehouse Service (DWS), and Application Data Service (ADS)`. Data tasks are scheduled among the layers in an event-driven or micro-batch manner. Layering helps improve semantic abstraction and data reuse. However, layering also increases scheduling dependencies, reduces the real-time performance of data, and reduces the agility of data analysis.

Real-time data warehouses drive customers to make real-time business decisions. In most cases, rich contextual information is required to make business decisions. This poses challenges to the traditional development method that highly depends on business-oriented ADS customization. Thousands of ADS tables are difficult to maintain, and the utilization rate is low. An increasing number of customers expect to perform multi-dimensional data comparison and analysis at the DWS or even DWD layer. This poses higher requirements for computing efficiency, scheduling efficiency, and I/O efficiency of the query engine.

The computing capabilities of Hologres are improved in each new version due to the application of various query engine optimization technologies, such as computing operator vectorization and rewriting, fine-grained indexing, asynchronous execution, and multi-level caching. An increasing number of customers adopt an agile development method. In the pre-computing stage, the customers perform only data cleansing and basic large table association and widening. Data modeling stops at the DWD and DWS layers. This reduces the number of modeling layers. The customers use the interactive search engine of Hologres to perform flexible queries. Hologres performs second-level interactive analytics to support the trend of data democratization.

To meet the requirements of different business scenarios, we recommend that you design layers and process data by using the three solutions. This can help increase the agility of your development process.

- Solution 1: ad hoc queries. In this solution, data is preprocessed at the DWD layer in Realtime Compute for Apache Flink. The processed data is directly written to Hologres. Hologres provides online analytical processing (OLAP) queries and online services. This way, data can be used immediately after it is written.
- Solution 2: minute-level quasi-real-time data warehousing. In this solution, micro-batch processing is implemented. Data is preprocessed at the DWD layer in Realtime Compute for Apache Flink. After the processed data is written to Hologres, Hologres processes the data at the aggregation layer and

then provides data services to upper-layer applications.

- Solution 3: real-time statistics collection of incremental data. In this solution, event-driven processing is implemented. Data is processed at the DWD and DWS layers in Realtime Compute for Apache Flink. Then, the processed data is written to Hologres for upper-layer applications.

Rules for selecting solutions

After data is written to Hologres, you can use Hologres to implement one of the preceding data warehousing solutions.

- Select Solution 1 if the following conditions are met: Your business requires high real-time performance. You expect data to be available for queries immediately after the data is written to Hologres and expect data updates to be synchronized to Hologres in real time. Ad hoc queries need to be supported. You have sufficient resources. For more information, see Solution 1: Ad hoc queries.
- Select Solution 2 if your business requires real-time analytics and you prioritize development efficiency over real-time performance. The minute-level quasi-real-time solution is suitable for more than 80% of real-time data warehousing scenarios. For more information, see Solution 2: Minute-level quasi-real-time data warehousing.
- Select Solution 3 if the following conditions are met: Your business focuses on providing online services such as dashboards and risk control. The data volume for your business is small and only incremental data is required to generate statistical results. You prioritize real-time performance over development efficiency and cost-effectiveness of computing. For more information, see Solution 3: Real-time statistics collection of incremental data.

Solution 1: Ad hoc queries

In this solution, the query patterns of upper-layer applications are unknown. The data is stored to support flexible ad hoc queries.

We recommend that you apply the following policies:

- Perform simple cleansing and association on data from the ODS layer and store the processed data to the DWD layer. Then, write the detail data to Hologres without processing or aggregating the data.
- Use Realtime Compute for Apache Flink to process incremental data and update detail data in Hologres in real time. Write the batch tables that are processed by MaxCompute to Hologres.
- Encapsulate SQL logic into views at the common data model (CDM) or ADS layer because the analysis SQL statements of upper-layer applications are not fixed.
- Query the encapsulated views in upper-layer applications to implement ad hoc queries.

Advantages:

- The flexibility is high. The views can be adjusted based on your business logic in a flexible way.
- The metrics are easy to correct. The logic is encapsulated into views and no aggregate tables exist in the upper layers. To update data, you need to only update data in underlying tables. This process involves only one layer. You do not need to update tables for upper-layer applications.

Disadvantages: If the logic of views is complex and the data volume is large, the query performance is low.

Use scenarios: Data originates from databases and event tracking systems, high flexibility is required, high queries per second (QPS) is not required, and the computing resources are sufficient.

Solution 2: Minute-level quasi-real-time data warehousing

The computing efficiency provided by Solution 1 cannot meet high QPS requirements. Solution 2 is an upgraded version of Solution 1. In Solution 2, the views are materialized into tables. Solution 2 uses the same logic as Solution 1, but a smaller volume of data is stored in tables. This helps improve query performance.

We recommend that you apply the following policies:

- Perform simple cleansing and association on data from the ODS layer and store the processed data to the DWD layer. Then, write the detail data to Hologres without processing or aggregating the data.
- Use Realtime Compute for Apache Flink to process incremental data and update detail data in Hologres in real time.
- Store data in physical tables at the CDM or ADS layer. Schedule DataWorks to periodically write data to the tables.
- Query the physical tables in real time from upper-layer applications. The real-time performance of data depends on the scheduling cycle that is configured in DataWorks. For example, DataWorks supports 5-minute and 10-minute scheduling cycles. This way, you can implement minute-level quasi-real-time data warehousing.

Advantages:

- The query performance is high. Upper-layer applications query only aggregate data. Compared with view queries, table queries are performed on less data and provide higher query performance.
- Data can be updated in a short period of time. If a step error or a data error occurs, you need to only run scheduled nodes again in DataWorks. All the logic is fixed. You do not need to perform complicated link revision operations.
- The business logic can be adjusted in a short period of time. If you need to add or adjust the business code at each layer, you can develop business scenarios based on SQL statements in what you see is what you get (WYSIWYG) mode. This helps shorten the release cycle of your business application.

Disadvantages: The real-time performance provided by Solution 2 is lower than that provided by Solution 1 because more processing and scheduling steps are involved.

Use scenarios: Data originates from databases and event tracking systems, and high QPS and real-time performance are required. This solution is suitable for 80% of real-time data warehousing scenarios and can meet the requirements of most business scenarios.

Solution 3: Real-time statistics collection of incremental data

Incremental computing is required if your business is sensitive to data latency and your business requires data to be processed immediately after the data is generated. In this case, you can use Realtime Compute for Apache Flink to process and aggregate data at the DWD and DWS layers and store aggregated result sets for upper-layer applications.

We recommend that you apply the following policies:

- Use Realtime Compute for Apache Flink to cleanse, transform, and aggregate incremental data. Store the application data that is generated at the ADS layer in Hologres.
- Write the result sets that are generated in Realtime Compute for Apache Flink in dual-write mode. The result sets are delivered to the message topic at the next layer and exported to data sinks in Hologres at the same layer. This way, you can check and refresh the status of historical data in subsequent operations in a convenient manner.
- Collect statistics in Realtime Compute for Apache Flink by using incremental streams, incremental streams connected to static dimension tables, or incremental streams connected to incremental streams. Write the collected data to Hologres.

- Hologres provides tables for upper-layer applications to perform real-time queries.

Advantages:

- The real-time performance is high. This solution can meet the requirements of latency-sensitive business scenarios.
- The metrics are easy to correct. This solution is different from traditional incremental computing in that the intermediate status is persistently stored in Hologres. This helps improve the flexibility of subsequent analysis operations. If the quality of intermediate data cannot meet your requirements, you can modify tables to update data.

Disadvantages: Real-time incremental computing relies on Realtime Compute for Apache Flink. Users must be skilled and proficient in using Realtime Compute for Apache Flink. This solution cannot meet requirements in scenarios in which data is frequently updated and cannot be aggregated or complex overhead computing scenarios such as multi-stream join queries.

Use scenarios: The data volume is not large, the data is collected from event tracking systems, and only incremental data is required to generate statistical results. This solution provides the highest real-time performance among the three solutions.

3.2. Real-time report analysis

3.2.1. Build a real-time data warehouse and display data analytics results

This topic describes how to connect Hologres to Realtime Compute to build a real-time data warehouse and then connect Hologres to a Business Intelligence (BI) tool to display data analytics results.

Prerequisites

- A Hologres instance is purchased and a development tool is connected to the instance. For more information, see Quick start to HoloWeb.
- Realtime Compute is activated.

> **Note** Make sure that you activate the Realtime Compute and Hologres services in the same region.

- DataV is activated. For more information, see Activate DataV.

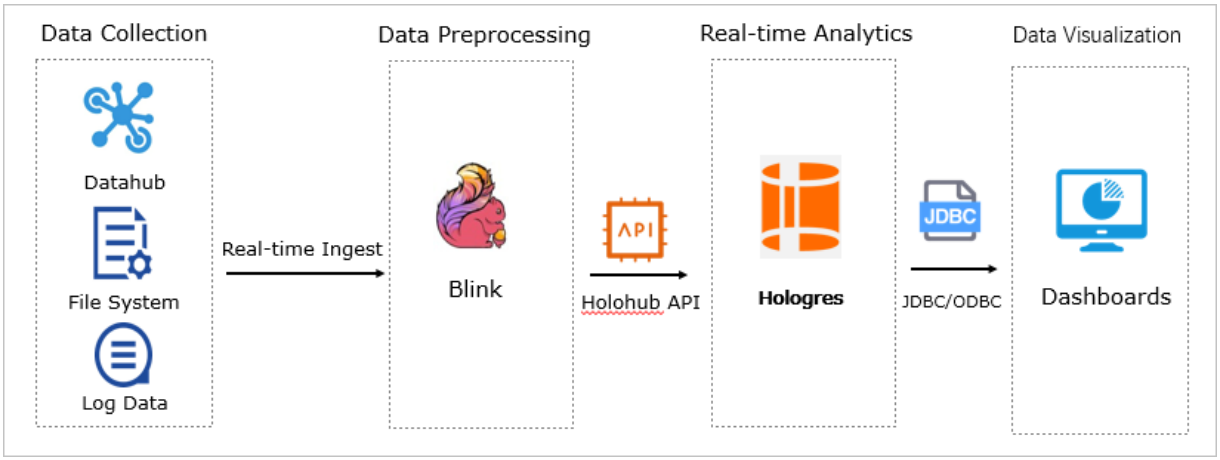
Context

Hologres is an interactive analytics service developed by Alibaba Cloud. Based on the built-in HoloHub API, Hologres connects to Realtime Compute to support real-time writes and queries in high concurrency. Hologres can respond to queries within seconds.

Hologres is compatible with PostgreSQL. You can connect Hologres to a BI tool to analyze queried data and display data analytics results in a visualized manner.

This topic uses an e-commerce store as an example to describe how to display operating metrics in real time. The metrics include the number of unique visitors (UVs) to the store, number of UVs to each product, sales amount in each city, and statistics of hot-selling products.

The following figure shows the process of using Hologres to display operating metrics on a dashboard in real time.



- Source data is collected and written to Realtime Compute in real time for cleansing and aggregation.
- Data processed by Realtime Compute is written to Hologres for interactive searches.
- Query results are displayed in DataV in real time, which is connected to Hologres.

Procedure

1. Collect source data.

   Use DataHub, a streaming data processing service, or service logs to collect source data.

   To simplify the process, this best practice uses Realtime Compute to generate source data. For more information, see step 3.

2. Create a table in Hologres for receiving data.

   Use HoloWeb to create a table for receiving data. Make sure that this table contains the same *fields* of the same *data types* as the source table. The following SQL statements are used as an example:

```
begin;
drop table if exists order_details;
create table order_details(user_id bigint, user_name text, item_id bigint, item_name te
xt, price numeric(38, 2), province text, city text, ip text, longitude text, latitude t
ext, sale_timestamp timestamptz not null);
call set_table_property('order_details', 'distribution_key', 'user_id');
call set_table_property('order_details', 'segment_key', 'sale_timestamp');
call set_table_property('order_details', 'clustering_key', 'sale_timestamp');
commit;
```

3. Use Realtime Compute to cleanse data.

   Log on to the Realtime Compute console. In the console, create a job to cleanse and aggregate collected data in the data source and call the HoloHub API to write processed data to Hologres in real time. The following SQL statements are used as an example:

```
// Create a data source.
create table order_details(
  user_id BIGINT,
  user_name VARCHAR,
  item_id BIGINT,
  item_name VARCHAR,
```

```
  price numeric(38, 2),
  province VARCHAR,
  city VARCHAR,
  ip VARCHAR,
  longitude VARCHAR,
  latitude VARCHAR,
  sale_timestamp TIMESTAMP
) with (
  type = 'custom',
  tableFactoryClass = 'com.alibaba.blink.connectors.hologres.table.factory.DemoDataGene
ratorFactory');
// Create a connection to Hologres.
create table hologres_sink(
    user_id BIGINT,
    user_name VARCHAR,
    item_id BIGINT,
    item_name VARCHAR,
    price numeric(38, 2),
    province VARCHAR,
    city VARCHAR,
    ip VARCHAR,
    longitude VARCHAR,
    latitude VARCHAR,
    sale_timestamp TIMESTAMP
  ) with (
  type = 'custom',
  tableFactoryClass = 'com.alibaba.blink.connectors.hologres.table.factory.HologresTabl
eFactory',
  endpoint = 'Virtual Private Cloud (VPC) endpoint and port number used to call the Hol
oHub API',
  dbName = 'Name of the Hologres database to be connected to',
  tableName = 'Name of the Hologres table for receiving data',
  username = 'AccessKey ID of the current Alibaba Cloud account',
  password = 'AccessKey secret of the current Alibaba Cloud account',
  batchSize = '500',
  bufferSize = '500'
);
// Write data to Hologres.
insert into hologres_sink
select
  user_id
, user_name
, item_id
, item_name
, price
, province
, city
, latitude
, longitude
, ip
, sale_timestamp
from order_details;
```

You can run the following command in the target Hologres instance to query the VPC endpoint

used to call the HoloHub API:

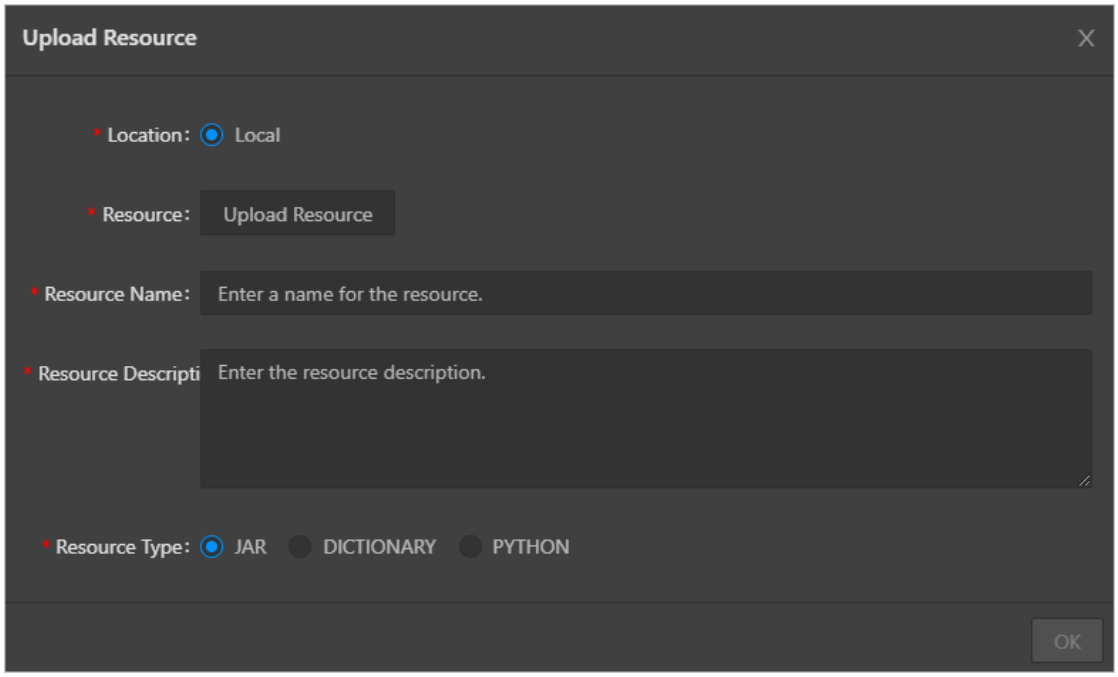
```
show hg_datahub_endpoints;
```

4. Publish a Realtime Compute job.

   To commit and publish a Realtime Compute job to the production environment, perform the following steps:

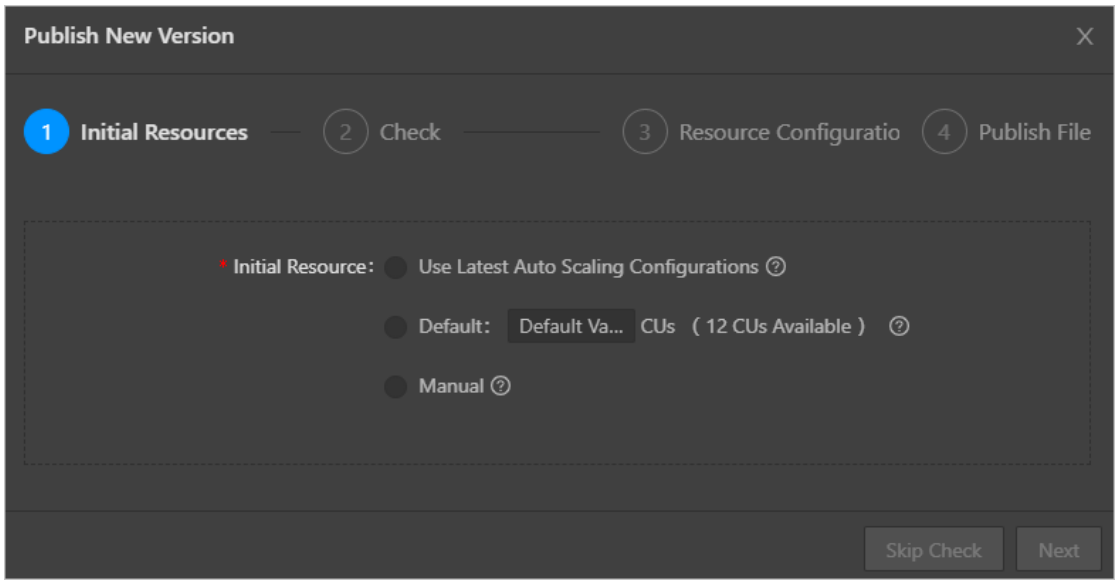
   i. Reference a resource package in a job.

   Log on to the Realtime Compute console. In the left-side navigation pane, click **Resources**. On the page that appears, click **Create Resource**. In the **Upload Resource** dialog box, set parameters as required to upload a Realtime Compute resource package. To obtain a sample Realtime Compute resource package, click Blink.



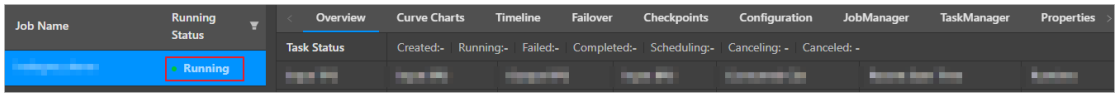
ii. Publish the job.

After a resource package is referenced, click Save and then **Publish**. Set resource parameters as required and publish the job to the production environment.



iii. Start the job.

After the job is published, go to Administration page to start the job. The job then enters the Running state, as shown in the following figure.



5. Use Hologres to query data in real time.

Use the SELECT statement to query data that is written to Hologres in real time from different dimensions. The following SQL statements are used as an example:

```
select sum(price) as "GMV" from order_details ;
select count(distinct user_id) as "UV" from order_details ;
select city as "City", count(distinct user_id) as "Number of customers who purchased pr
oducts" from order_details group by "City" order by "Number of customers who purchased
products" desc limit 100;
select item_name as "Product", sum(price) as "Sales amount" from order_details group by
"Product" order by "Sales amount" desc limit 100;
select to_char(sale_timestamp, 'MM-DD') as "Date", sum(price) as "GMV" from order_detai
ls group by "Date" order by "GMV" desc limit 100;
```

6. Create a dashboard in DataV to display query results in Hologres.

To create a dashboard in DataV to display query results in Hologres, perform the following steps:

i. Add a connection to the data source.

Log on to the DataV console. Click the **Data Sources** tab. On the Data Sources tab, click **Add Source**. In the **Add Data Source** dialog box, set parameters as required.

Select Hologres from the **Type** drop-down list.

ii. Create a dashboard to display real-time data.

Select widgets to be contained on the dashboard and configure a data source for each widget based on your business requirements. For more information, see Widget overview.

This best practice selects the basic column chart, carousel widget, basic flat map, and ticker board for the dashboard. Take a multiple pie chart as an example.

a. Set parameters for the **data source** used by the multiple pie chart.

b. Set the title, border, font, and color for the multiple pie chart.

iii. Decorate the dashboard.

After you configure the widgets and their data sources, you can decorate the dashboard as required.

- The total number of UVs to each product and sales amount in top cities are shown on the left in real time.
- The map in the middle highlights the location of each transaction order and refreshes the total sales amount in real time. The total number of UVs to the store is shown on the top of the map.
- The sales percentage and sales ranking of each product are shown on the right in real time.

3.2.2. Analyze large amounts of MaxCompute data in real time

This topic describes how to query large amounts of MaxCompute data and analyze and display the query results in a visualized manner.

Prerequisites

- MaxCompute is activated. For more information, see Activate MaxCompute and DataWorks.

  > **Note** Make sure that you activate the MaxCompute and Hologres services in the same region.

- A Hologres instance is purchased and connected to HoloWeb. For more information, see Quick start to HoloWeb.
- Quick BI is activated. For more information, see Prerequisites.

Context

Hologres is a real-time interactive analytics engine. It is compatible with PostgreSQL and integrates seamlessly with MaxCompute.

You can create a foreign table in Hologres to directly query data in MaxCompute.

This topic uses a Taobao store as an example to describe how to create a customer persona with the following information: the regional distribution and age composition of customers, the number of preferred customers, and the regional distribution of preferred customers who were born between 1980 and 1990.

The complete link for speeding up the query of maxcompute data using hologres is shown below.

1. Save data of customers who visited the store to MaxCompute tables.

2. Create a foreign table in Hologres to directly query data in MaxCompute.
3. Connect Quick BI to the target Hologres instance to display the customer persona in a visualized manner.

Procedure

1. Prepare a MaxCompute data source.

   Create a table in MaxCompute and import data to the table. For more information, see Create tables.

   This best practice uses the following existing tables in the MaxCompute project **public_data**.

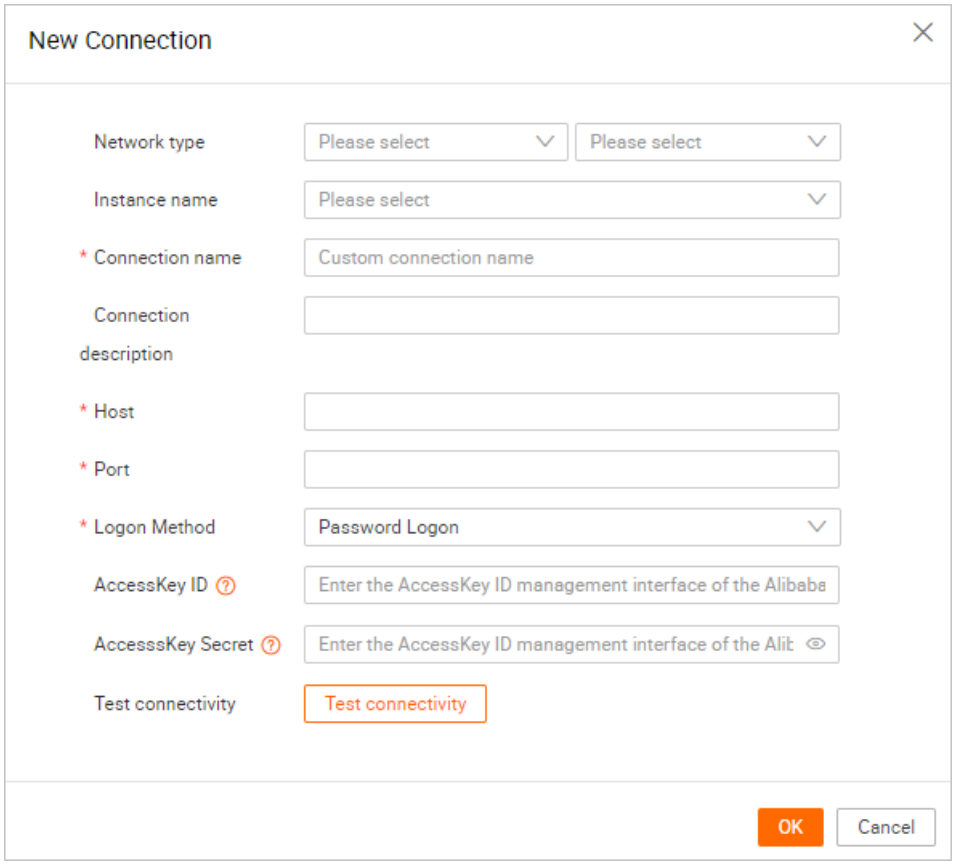
MaxCompute table	Data entries
customer	12 million
customer_address	6 million
customer_demographics	1.92 million

2. Create a foreign table in Hologres and query the table.

   To use HoloWeb to create a foreign table in Hologres for accelerating data queries from MaxCompute, perform the following steps:

   i. Connect a Hologres instance to HoloWeb.

   Log on to HoloWeb. On the **Connection Management** tab, click **Data Connection**. In the **New Connection** dialog box, set parameters as required and click **OK**.



Parameter	Description	Remarks
Connection name	The name of the connection. Enter a name as required.	None
Connection description	The description of the connection.	None
Host	The public endpoint of the Hologres instance.	You can view the public endpoint of the Hologres instance on the **Configuration** tab of the instance details page in the Hologres console.
Port	The public port number of the Hologres instance.	You can view the port number of the Hologres instance on the **Configuration** tab of the instance details page in the Hologres console.
Initialize database	The name of the Hologres database to be connected to HoloWeb.	None
User name	The AccessKey ID of the current Alibaba Cloud account.	You can obtain the AccessKey ID in the User Management console.
Password	The AccessKey secret of the current Alibaba Cloud account.	You can obtain the AccessKey secret in the User Management console.
Test connectivity	Check whether the data connection is successful. ■ Successful: The **"Test passed"** message appears. ■ Failed: The **"Test failed"** message appears.	None

ii. Create a foreign table.

Click **Connection Management** in the top navigation bar. On the Connection Management page, click **External Table**. On the New external table tab, set parameters as required to create a foreign table in a visualized manner.

Enter the name of the MaxCompute table to be queried. Then, fields in the table appear. Select fields to be synchronized and click **Submit**.

> **Note**
>
> - Hologres does not support querying MaxCompute tables that reside in a different region from the current Hologres instance.
> - A server is required for storing a foreign table. You can directly call the **odps_server** server created at the underlying layer of Hologres. For more information, see postgres_fdw.

You can use the following SQL statement to create multiple foreign tables at a time:

```
IMPORT FOREIGN SCHEMA public_data LIMIT to(
  customer,
  customer_address,
  customer_demographics,
  inventory,item,
  date_dim,
  warehouse)
  FROM server odps_server INTO PUBLIC options(if_table_exist 'update');
```

iii. Preview a foreign table.

After you create a foreign table, click **My Connections** on the **Connection Management** tab.

Right-click the target foreign table and click **Data Preview** to view data in the MaxCompute table mapped to the foreign table.

The **Data Preview** tab only shows partial data in the foreign table.

iv. Query data in the foreign table.

Click Query in the top navigation bar. On the page that appears, click SQL Window. In the New SQL Query dialog box, set parameters as required. Select the connection and database to which the target foreign table belongs. In the SQL editor, write an SQL statement to query data as required.

For example, you can use the following SQL statements:

```
# SQL 1: Query the number of non-preferred customers and the numbers of preferred c
ustomers with various flags, and sort the query results in descending order of the
number of customers.
SELECT c_preferred_cust_flag,
       count(*) AS cnt
FROM customer
WHERE c_preferred_cust_flag IS NOT NULL
GROUP BY c_preferred_cust_flag
ORDER BY cnt DESC LIMIT 10;
# SQL 2: Query the number of customers born in each year, and display the years in
which more than 1,000 customers were born in descending order of the number of cust
omers.
SELECT c_birth_year,
       count(*) AS cnt
FROM customer
WHERE c_birth_year IS NOT NULL
GROUP BY c_birth_year HAVING count(*) > 1000
ORDER BY cnt DESC LIMIT 10;
# SQL 3: Query the number of customers in each city, and display the cities where m
ore than 10 customers reside in descending order of the number of customers.
SELECT ca_city,
       count(*) AS cnt
FROM customer ,
     customer_address
WHERE c_current_addr_sk = ca_address_sk
  AND ca_city IS NOT NULL
GROUP BY ca_city HAVING count(*) > 10
ORDER BY cnt DESC LIMIT 10;
# SQL 4: Query the number of customers who were born between 1980 and 1990 in each
city, and display the cities in which more than 10 customers born between 1980 and
1990 reside in descending order of the number of customers.
SELECT ca_city,
       count(*) AS cnt
FROM customer ,
     customer_address
WHERE c_current_addr_sk = ca_address_sk
  AND c_birth_year >= 1980
  AND c_birth_year < 1990
  AND c_preferred_cust_flag = 'Y'
  AND ca_city IS NOT NULL
GROUP BY ca_city HAVING count(*) > 10
ORDER BY cnt DESC LIMIT 10;
```

3. Use Quick BI to analyze data.

   To connect Quick BI to the target Hologres instance to analyze and display data queried from MaxCompute in a visualized manner, perform the following steps:

   i. Add a connection.

   Log on to the Quick BI console and add a PostgreSQL connection to Hologres. For more information, see Quick BI.

ii. Create a dataset.

After you connect Quick BI to Hologres, create a dataset and import the required data to the dataset to produce reports.

This best practice uses **ad hoc query SQL statements** to create a dataset.

iii. Display the customer persona in a visualized manner.

3.3. User behavior analysis

3.3.1. Overview

In user behavior analysis and user identification scenarios, you often need to filter hundreds of millions of users or even billions of users to obtain metric data that has specific tags. This topic describes how to perform user behavior analysis in Hologres.

Background information

The unique visitor (UV) metric is most frequently used in behavior analysis and indicates the number of distinct individuals that have visited pages of a website within a specific time range. The UV metric can also be used to reflect the value of a specific metric within a time range after deduplication. For example, a seller needs to calculate the number of UVs in real time in a large e-commerce promotion. This way, the seller can adjust operation strategies at the earliest opportunity to achieve the sales goal.

When you calculate the number of UVs, the calculation dimension and data volume vary based on your business scenarios. In general, the following scenarios are involved:

- Your business involves hundreds of millions of data entries in more than 10 dimensions each day. You want to freely customize dimensions to query a huge amount of data.
- In addition to data query and update by day, week, month, or year, you want to query and update data in real time at a finer granularity.
- You want to accurately remove duplicate users.

In the preceding complex UV calculation scenarios, a pre-calculation system such as Apache Kylin or the Flink-MySQL solution with a fixed dimension is often used. However, such a solution has the following disadvantages:

- If UV calculation involves a large number of dimensions, a large amount of storage space is required, and the pre-calculation time is long.
- Accurate deduplication consumes a large number of resources. As a result, out-of-memory (OOM) errors easily occur.
- Real-time updates are difficult to implement, and data cannot be processed in a more flexible and open time window.

Solutions and benefits

Hologres is a real-time data warehouse for hybrid serving and analytical processing (HSAP). Hologres uses a distributed architecture, supports real-time data writing, and can analyze and process petabytes of data with high concurrency and low latency. Hologres is compatible with the PostgreSQL protocol and allows you to use existing tools for data analytics.

Hologres provides high performance and can accurately calculate hundreds of millions of UVs by using Roaring bitmaps and auto-increment columns of the SERIAL type.

- RoaringBitmap

  Roaring bitmaps are compressed bitmaps for indexing. The data compression and deduplication features of Roaring bitmaps are ideal for UV calculation in big data scenarios. Roaring bitmaps have the following characteristics:

  - In a Roaring bitmap 2^{16} chunks are constructed for 32-bit integers and correspond to the 16 most significant bits of the 32-bit integers. The 16 least significant bits of the 32-bit integer are mapped to a single bit in each chunk. The capacity of a single chunk is determined by the existing maximum value in the chunk.
  - A Roaring bitmap uses one bit to represent a 32-bit integer. This greatly compresses data.
  - Roaring bitmaps provide bitwise operations for deduplication.

  For more information about how to use Roaring bitmaps, see Roaring bitmap functions.

- Serial

  Auto-increment columns of the SERIAL type are often used for user ID (UID) mapping when you join a source table with a dimension table. In many cases, UIDs collected in business systems or tracking points are of the STRING or LONG type. In these cases, you need to create a UID mapping table. UIDs stored in Roaring bitmaps must be 32-bit integers and need to be consecutive if possible. The UID mapping table contains a column of the SERIAL type that consists of auto-increment 32-bit integers. This way, the UID mapping is automatically managed and remains stable.

Batch UV calculation

In batch UV calculation, all data of the previous day is aggregated into UIDs based on the largest query dimension and stored in Roaring bitmaps. The Roaring bitmaps and query dimensions are stored in an aggregation result table. The aggregation result table stores only millions of data entries per day. When you query data, Hologres uses its powerful column-oriented computing capability to query the aggregation result table based on a query dimension, performs OR operations on the field that stores the Roaring bitmaps to remove duplicates, and then calculates the cardinality of the Roaring bitmaps. This way, you can obtain the number of UVs and calculate the number of page views (PVs) based on the number of UVs. The entire query process takes only sub-seconds.

You need only to perform pre-aggregation once at the finest granularity and generate only one pre-aggregation result table with the finest granularity. This solution requires few pre-calculation operations and little space due to the powerful computing power of Hologres. For more information, see Batch UV calculation.

Real-time UV calculation

Hologres is highly compatible with Flink. Hologres supports high-throughput data writes from Flink in real time and real-time queries of the written data. Hologres allows you to join a source table with a dimension table by executing Flink SQL statements. Hologres also allows you to use the change data capture (CDC) feature for data analytics.

You can calculate UVs in real time by integrating Hologres with Flink. Specifically, you can use Flink to join a source table with a Hologres dimension table and use Roaring bitmaps to deduplicate user tags in real time. This way, you can obtain fine-grained UV and PV data in real time. In addition, you can adjust the minimum statistical window, such as UVs in the past 5 minutes, based on your business requirements. This brings effects similar to real-time monitoring and facilitates data display, such as data display on a big screen. Compared with deduplication by day, week, or month, this solution provides better performance in finer-grained deduplication of data on a specified business date. This solution also provides deduplicated data within a comparatively long period by aggregating deduplication results.

This solution is easy to use. You can freely customize dimensions for calculation. This solution stores data in bitmaps, which significantly reduces the storage space required. In addition, this solution returns deduplication results in real time. All these benefits help build a multi-dimensional data warehouse that provides abundant features and supports flexible data analytics in real time. For more information, see Remove duplicate UVs in real time.

3.3.2. Batch UV calculation

This topic describes how to perform batch unique visitor (UV) calculation in Hologres.

Procedure

1. Create required tables.
   i. Install the extension for Roaring bitmaps.

      Before you use Roaring bitmaps, make sure that you have installed the extension for Roaring bitmaps and the version of your Hologres instance is V0.10 and later. You can execute the following statement to install the extension:

      ```
      CREATE EXTENSION IF NOT EXISTS roaringbitmap;
      ```

ii. Create a user detail table.

Create a user detail table named ods_app by using the following DDL statements. The ods_app table is used to store the huge amount of detailed user data and is partitioned by day.

```
BEGIN;
CREATE TABLE IF NOT EXISTS public.ods_app (
     uid text,
     country text,
     prov text,
     city text,
     channel text,
     operator text,
     brand text,
     ip text,
     click_time text,
     year text,
     month text,
     day text,
     ymd text NOT NULL
);
CALL set_table_property('public.ods_app', 'bitmap_columns', 'country,prov,city,chan
nel,operator,brand,ip,click_time, year, month, day, ymd');
-- Specify a distribution key so that data can be properly distributed to shards to
adapt to real-time data query needs.
CALL set_table_property('public.ods_app', 'distribution_key', 'uid');
-- Prepare the fields that can be used in the WHERE clause. We recommend that you s
et a field that contains time information, such as year, month, and date, as a clus
tering key or an event time column.
CALL set_table_property('public.ods_app', 'clustering_key', 'ymd');
CALL set_table_property('public.ods_app', 'event_time_column', 'ymd');
CALL set_table_property('public.ods_app', 'orientation', 'column');
COMMIT;
```

iii. Create a UID mapping table.

Create a UID mapping table named uid_mapping by using the following DDL statements. The uid_mapping table is used to establish mappings between UIDs and 32-bit integers.

UIDs stored in Roaring bitmaps must be 32-bit integers and need to be consecutive if possible. However, UIDs collected in business systems or tracking points are usually of the STRING type. Therefore, you need to create a UID mapping table. The UID mapping table contains a column of the SERIAL type that consists of auto-increment 32-bit integers. This way, the UID mapping is automatically managed and remains stable.

> **Note** In this example, the UID mapping table can be row-oriented or column-oriented. If you need to calculate UVs in real time, such as by integrating Hologres with Flink, the UID mapping table must be row-oriented. This way, the queries per second (QPS) performance can be improved when you use Flink to join a source table with a dimension table.

```
BEGIN;
 CREATE TABLE public.uid_mapping (
     uid text NOT NULL,
     uid_int32 serial,
            PRIMARY KEY (uid)
 );
 -- Set the UID column as a clustering key and a distribution key to quickly find t
he 32-bit integers corresponding to the UIDs.
CALL set_table_property('public.uid_mapping', 'clustering_key', 'uid');
CALL set_table_property('public.uid_mapping', 'distribution_key', 'uid');
CALL set_table_property('public.uid_mapping', 'orientation', 'row');
COMMIT;
```

iv. Create an aggregation result table.

Create an aggregation result table named dws_app by using the following DDL statements. The aggregation result table is used to store the aggregation results of Roaring bitmaps.

A basic dimension is the finest dimension for PV and UV query and calculation. In this example, the aggregation result table uses the following columns as basic dimensions: country, proc, and city.

```
begin;
create table dws_app(
    country text,
  prov text,
  city text,
  ymd text NOT NULL, -- The date column.
  uid32_bitmap roaringbitmap, -- The column that stores Roaring bitmaps for UV calc
ulation.
  pv integer, -- The column that stores PVs.
  primary key(country, proc, city, ymd)-- Set columns about query dimensions and ti
me as the primary key to prevent data from being repeatedly inserted.
);
CALL set_table_property('public.dws_app', 'orientation', 'column');
-- Set the date column as a clustering key and an event time column to facilitate d
ata filtering.
CALL set_table_property('public.dws_app', 'clustering_key', 'ymd');
CALL set_table_property('public.dws_app', 'event_time_column', 'ymd');
-- Set columns about query dimensions as a distribution key.
CALL set_table_property('public.dws_app', 'distribution_key', 'country,prov,city');
end;
```

2. Update the uid_mapping table and the dws_app table.

i. Update the uid_mapping table.

Execute the following statement to insert new UIDs from the ods_app table into the uid_mapping table. New UIDs are UIDs that were generated on the previous day and do not exist in the uid_mapping table.

```
WITH
-- In the WHERE clause, the ymd parameter is set to 20210329, which indicates the d
ata of the previous day.
    user_ids AS ( SELECT uid FROM ods_app WHERE ymd = '20210329' GROUP BY uid )
    ,new_ids AS ( SELECT user_ids.uid FROM user_ids LEFT JOIN uid_mapping ON (user_
ids.uid = uid_mapping.uid) WHERE uid_mapping.uid IS NULL )
INSERT INTO uid_mapping SELECT  new_ids.uid
FROM    new_ids
;
```

ii. Update the dws_app table.

After you update the uid_mapping table, perform the following steps to aggregate data and insert the aggregated results to the dws_app table:

a. Perform the INNER JOIN operation on the ods_app table and the uid_mapping table to obtain the aggregation conditions and corresponding 32-bit UIDs for the previous day.

b. Aggregate data based on the aggregation conditions, and insert the aggregated data into the dws_app table as the aggregation results of the previous day.

c. You need only to aggregate data once a day and store the aggregation results in the aggregation result table. The number of data entries in the aggregation result table equals the number of UVs. Hundreds of millions of incremental data entries in the ods_app table are aggregated into millions of data entries and stored in the dws_app table each day.

Execute the following statement to insert data into the dws_app table:

```
WITH
    aggregation_src AS( SELECT country, prov, city, uid_int32 FROM ods_app INNER JO
IN uid_mapping ON ods_app.uid = uid_mapping.uid WHERE ods_app.ymd = '20210329' )
INSERT INTO dws_app SELECT  country
        ,prov
        ,city
        ,'20210329'
        ,RB_BUILD_AGG(uid_int32)
        ,COUNT(1)
FROM    aggregation_src
GROUP BY country
         ,prov
         ,city
;
```

3. Query UVs and PVs.

When you query data, Hologres performs aggregation on the dws_app table based on a query dimension and calculates the cardinality of Roaring bitmaps. This way, you can obtain the number of UVs based on the condition specified by the GROUP BY clause.

```
-- Perform the following RB_AGG operation to query data. We recommend that you disable
the three-stage aggregation feature first. By default, this feature is disabled.
set hg_experimental_enable_force_three_stage_agg=off
-- You can query UVs and PVs within a time range based on a custom combination of basic
dimensions.
SELECT  country
        ,prov
        ,city
        ,RB_CARDINALITY(RB_OR_AGG(uid32_bitmap)) AS uv
        ,sum(1) AS pv
FROM    dws_app
WHERE   ymd = '20210329'
GROUP BY country
         ,prov
         ,city;
-- You can execute the following statement to query UVs and PVs within a month:
SELECT  country
        ,prov
        ,RB_CARDINALITY(RB_OR_AGG(uid32_bitmap)) AS uv
        ,sum(1) AS pv
FROM    dws_app
WHERE   ymd >= '20210301' and ymd <= '20210331'
GROUP BY country
         ,prov;
Alternatively, you can use execute the following statement to query UVs and PVs within
a month:
SELECT  country
        ,prov
        ,city
        ,COUNT(DISTINCT uid) AS uv
        ,COUNT(1) AS pv
FROM    ods_app
WHERE   ymd = '20210329'
GROUP BY country
         ,prov
         ,city;
SELECT  country
        ,prov
        ,COUNT(DISTINCT uid) AS uv
        ,COUNT(1) AS pv
FROM    ods_app
WHERE   ymd >= '20210301' and ymd <= '20210331'
GROUP BY country
         ,prov;
```

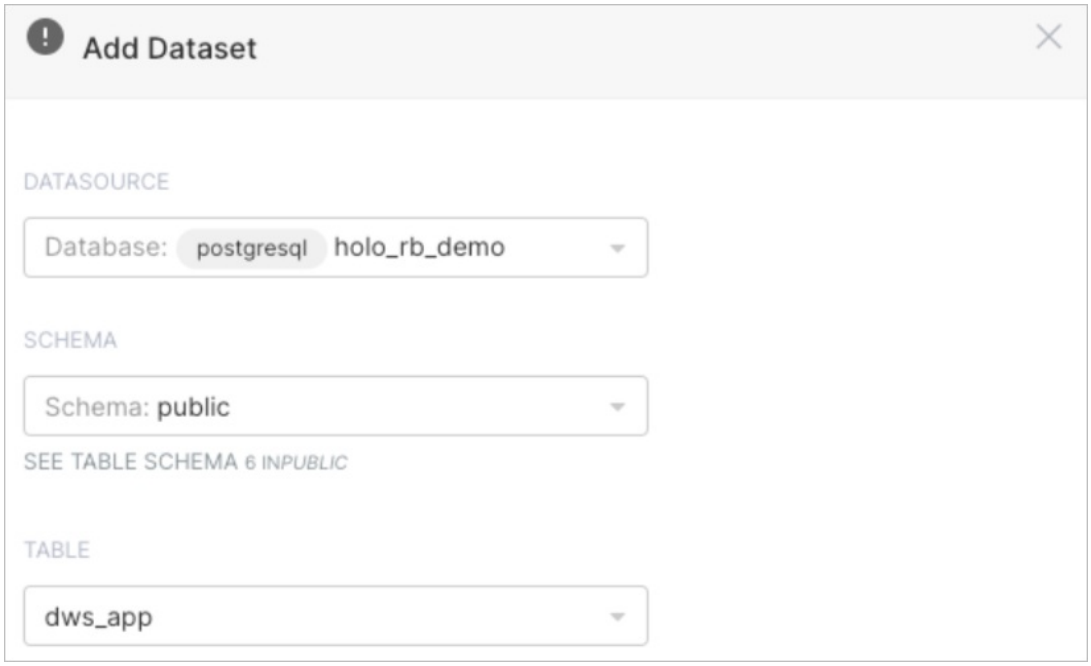
4. Visually display data.

   In most cases, you need to use Business Intelligence (BI) tools to visually display the calculated UVs and PVs. In the calculation process, RB_CARDINALITY and RB_OR_AGG functions are used to aggregate data. Therefore, BI tools must support custom aggregation functions. You can use common BI tools such as Apache Superset and Tableau.

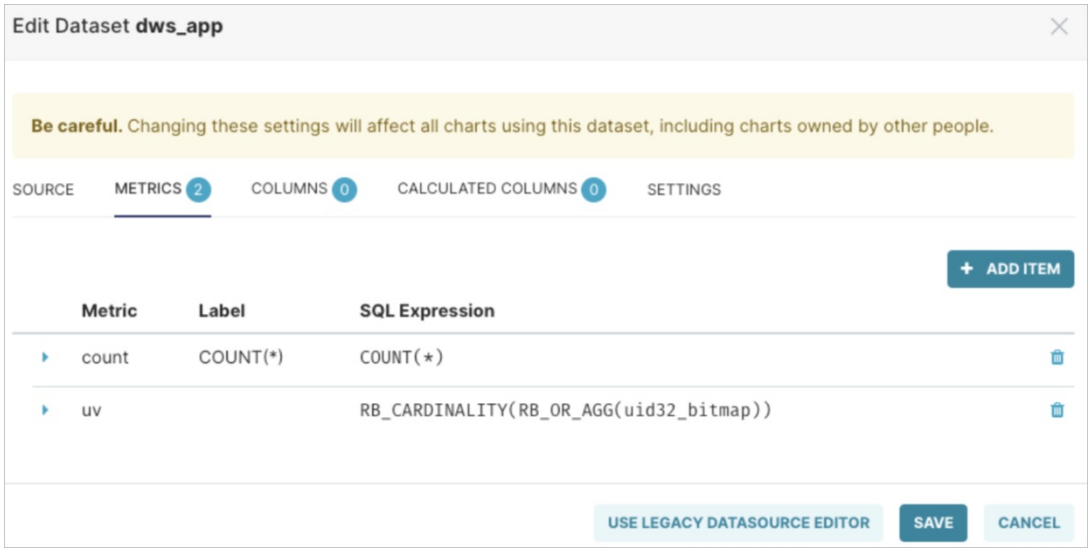
   - Apache Superset

     a. Connect Apache Superset to Hologres. For more information, see Apache Superset.

b. Set the dws_app table as a dataset.



c. Create a metric named UV in the dataset by using the expression that is shown in the following figure.




```
RB_CARDINALITY(RB_OR_AGG(uid32_bitmap))
```

Then, you can start to explore data.

d. Optional. Create a dashboard.

For more information about how to create a dashboard, see Creating Your First Dashboard.

- Tableau

a. Connect Tableau to Hologres. For more information, see Tableau.

You can use pass-through functions in Tableau to customize functions. For more information, see Pass-Through Functions (RAWSQL).

b. Create a calculation field by using the expression that is shown in the following figure.


```
RAWSQLAGG_INT("RB_CARDINALITY(RB_OR_AGG(%1))", [Uid32 Bitmap])
```

Then, you can start to explore data.

c. Optional. Create a dashboard.

For more information about how to create a dashboard, see Create a Dashboard.

3.3.3. Remove duplicate UVs in real time

You can integrate Hologres with Flink to count unique visitors (UVs) in real time. This topic describes how to remove duplicate UVs in real time when UVs are counted.

Prerequisites

- A Hologres instance is created, and a development tool is used to connect to the instance. In this example, HoloWeb is used. For more information about how to connect to a Hologres instance by using HoloWeb, see HoloWeb quick start.
- A Flink cluster environment is prepared and built. You can use fully managed Flink of Realtime Compute for Apache Flink or open source Apache Flink.

Context

Hologres is highly compatible with Flink. Hologres supports high-throughput data writes from Flink in real time and real-time queries of the written data. Hologres allows you to join a source table with a dimension table by executing Flink SQL statements. Hologres also allows you to use the change data capture (CDC) feature for data analytics. In addition, you can integrate Hologres with Flink to remove duplicate UVs in real time. The following figure shows the workflow.

1. Flink subscribes to newly collected data in real time. The data can be collected from logs, such as Kafka logs.
2. Flink converts the subscribed data streams into a source table. Then, Flink joins the source table

with a Hologres dimension table to write the data of the source table to Hologres in real time.

3. Hologres processes the written data in real time.
4. The processed data is used by upper-layer data services, such as DataService Studio and Tableau.

How it works

You can use Roaring bitmaps supported by Hologres to count UVs and remove duplicate UVs in real time. The following figure shows the flowchart.

1. In Flink, subscribe to user data from data sources such as Kafka or Redis, and use DataStream programs to convert data streams into a source table.
2. Create a unique ID (UID) mapping table in Hologres to store the UIDs of historical users and corresponding auto-increment 32-bit UIDs.

   > **Note** In many cases, UIDs collected in business or tracking point-related activities are of the STRING or LONG type. In these cases, you must create a UID mapping table. UIDs stored in Roaring bitmaps must be 32-bit integers. Consecutive integers are preferred. The UID mapping table contains a column of the SERIAL type that consists of auto-increment 32-digit integers. This way, the UID mapping is automatically managed and remains stable.

3. In Flink, use the UID mapping table as a Hologres dimension table, and use the insertIfNotExists feature of the Hologres dimension table to efficiently map UIDs based on auto-increment 32-bit integersauto-increment 32-bit integers. Join the source table with the Hologres dimension table and convert the joined results into data streams.
4. Create a table in Hologres to aggregate the processed results. Flink processes the joined results based on the lifecycle of Flink time windows and runs Roaring bitmap functions based on query dimensions.
5. Query the aggregation result table based on query dimensions. Calculate the number of data entries in the queried results and the number of ROARINGBITMAP data entries by using the `OR` operator. The calculated result is the number of deduplicated UVs.

This way, you can obtain fine-grained UV and page view (PV) data. You can adjust the minimum statistical window such as UVs in the past 5 minutes based on your business requirements. This has similar effects to real-time monitoring and displays better in business intelligence (BI) tools a big screen. This solution provides better performance in finer-grained deduplication of data on a specified business date than deduplication by day, week, or month. This solution can also provide deduplicated data for a relatively long period of time by aggregating deduplication results.

This solution is easy to use. You can set dimensions for calculation. This solution stores data in bitmaps, which significantly reduces the storage space required. In addition, this solution returns deduplication results in real time. All of these benefits together help build a multi-dimensional data warehouse that provides abundant features and supports flexible data analytics in real time.

Procedure

1. Create tables in Hologres.

i. Create a UID mapping table.

Execute the following statements to create a UID mapping table named uid_mapping in Hologres. The UID mapping table is used to establish mappings between UIDs and corresponding 32-bit integers. If the original UIDs are 32-bit integers, skip this step.

- In many cases, UIDs collected in business or tracking point-related activities are of the STRING or LONG type. In these cases, you must create a UID mapping table. UIDs stored in Roaring bitmaps must be 32-bit integers, and consecutive integers are preferred. The UID mapping table contains a column of the SERIAL type that consists of auto-increment 32-digit integers. This way, the UID mapping is automatically managed and remains stable.
- Data streams about UIDs are collected in real time and converted into a row-oriented source table. This ensures high QPS performance when you join the source table with the Hologres dimension table in Flink.
- GUC parameters must be specified to use optimized execution engines to write data to the table that contains columns of the SERIAL type. For more information, see Accelerate the execution of SQL statements by using fixed plans.

```
BEGIN;
CREATE TABLE public.uid_mapping (
uid text NOT NULL,
uid_int32 serial,
PRIMARY KEY (uid)
);
-- Set the UID column as the clustering key and distribution key to quickly find th
e 32-bit integers corresponding to the UIDs.
CALL set_table_property('public.uid_mapping', 'clustering_key', 'uid');
CALL set_table_property('public.uid_mapping', 'distribution_key', 'uid');
CALL set_table_property('public.uid_mapping', 'orientation', 'row');
COMMIT;
```

ii. Create an aggregation result table.

Create an aggregation result table named dws_app to store the aggregated results.

Before you use Roaring bitmap functions, make sure that you have installed an extension for roaring bitmaps and the version of your Hologres instance is V0.10 or later.

```
CREATE EXTENSION IF NOT EXISTS roaringbitmap;
```

To ensure good performance, we recommend that you set an appropriate number of shards based on the amount of data in the aggregation result table. we recommend that you keep the number of shards at no more than 60% of the total number of CPU cores. We recommend that you use a pivot table to set the number of shards for a table group. The following sample code provides an example:

```
-- Create a table group that has 16 shards.
-- In this example, millions of data entries are collected, the total number of CPU cores is 128, and the number of shards is 16.
BEGIN;
CREATE TABLE tg16 (a int);                              -- Create a pivot table for a table group.
CALL set_table_property('tg16', 'shard_count', '16');
COMMIT;
```

Compared with offline result tables, this aggregation result table adds a timestamp column to calculate data collected based on the lifecycle of Flink time windows. The following DDL statements provide an example:

```
BEGIN;
CREATE TABLE dws_app(
  country text,
  prov text,
  city text,
  ymd text NOT NULL, -- The date column.
  timetz TIMESTAMPTZ, -- The timestamp column used to calculate data collected based on the lifecycle of Flink time windows.
  uid32_bitmap roaringbitmap, -- The ROARINGBITMAP data used to calculate UVs.
 PRIMARY KEY (country, prov, city, ymd, timetz) -- Set columns about query dimensions, the date column, and the timestamp column as primary key columns to prevent data from being repeatedly inserted.
);
CALL set_table_property('public.dws_app', 'orientation', 'column');
-- Set the date column as the clustering key and event time column to filter data.
CALL set_table_property('public.dws_app', 'clustering_key', 'ymd');
CALL set_table_property('public.dws_app', 'event_time_column', 'ymd');
-- Create the table in a table group that has 16 shards.
call set_table_property('public.dws_app', 'colocate_with', 'tg16');
-- Set columns about query dimensions as distribution key columns.
CALL set_table_property('public.dws_app', 'distribution_key', 'country,prov,city');
COMMIT;
```

2. In Flink, read data streams in real time and update the aggregation result table.

For information about the complete sample code, see alibabacloud-hologres-connectors. The following steps are performed:

i. Read data streams and convert the data into a source table.

Flink reads data from a data source in streaming mode. You can select a CSV file or a Kafka or Redis data source based on your business requirements. The following sample code provides an example on how to convert the data into a table:

```
-- In this example, the data source is a CSV file. You can also select a Kafka or R
edis data source.
DataStreamSource odsStream = env.createInput(csvInput, typeInfo);
-- Before you join the source table with a dimension table, add a column that descr
ibes the proctime attribute to the source table. For more information, see JOIN sta
tements for dimension tables at https://www.alibabacloud.com/help/en/realtime-compu
te-for-apache-flink/latest/join-statements-for-dimension-tables.
Table odsTable =
    tableEnv.fromDataStream(
    odsStream,
    $("uid"),
    $("country"),
    $("prov"),
    $("city"),
    $("ymd"),
    $("proctime").proctime());
-- Create a catalog view.
tableEnv.createTemporaryView("odsTable", odsTable);
```

ii. Join the source table with a Hologres dimension table named uid_mapping.

When you create a Hologres dimension table in Flink, set the `insertIfNotExists` parameter to true. This ensures that you can manually insert data into the dimension table if the data is not automatically inserted. The uid_int32 field is the column of the SERIAL type that contains auto-increment 32-bit integers in the Hologres dimension table. The following sample code provides an example on how to join the tables:

```
-- Create a Hologres dimension table. The insertIfNotExists parameter specifies whe
ther to manually insert data into the dimension table if the data cannot be automat
ically inserted.
String createUidMappingTable =
    String.format(
    "create table uid_mapping_dim("
    + "  uid string,"
    + "  uid_int32 INT"
    + ") with ("
    + "  'connector'='hologres',"
    + "  'dbname' = '%s'," // The Hologres database in which the Hologres dimension
table resides.
    + "  'tablename' = '%s'," // The name of the Hologres dimension table.
    + "  'username' = '%s'," // The AccessKey ID of your Alibaba Cloud account.
    + "  'password' = '%s'," // The AccessKey secret of your Alibaba Cloud account.
    + "  'endpoint' = '%s'," //Hologres endpoint
    + "  'insertifnotexists'='true'"
    + ")",
    database, dimTableName, username, password, endpoint);
tableEnv.executeSql(createUidMappingTable);
-- Join the source table with the Hologres dimension table.
String odsJoinDim =
    "SELECT ods.country, ods.prov, ods.city, ods.ymd, dim.uid_int32"
    + "  FROM odsTable AS ods JOIN uid_mapping_dim FOR SYSTEM_TIME AS OF ods.procti
me AS dim"
    + "  ON ods.uid = dim.uid";
Table joinRes = tableEnv.sqlQuery(odsJoinDim);
```

iii. Convert the joined results into data streams.

Use Flink time windows to process data streams and run Roaring bitmap functions to remove duplicate data. The following sample code provides an example:

```
DataStream<Tuple6<String, String, String, String, Timestamp, byte[]>> processedSour
ce =
    source
    -- The dimensions by which data is queried. In this example, the dimensions are
the country, prov, city, and ymd columns.
    .keyBy(0, 1, 2, 3)
    -- The Flink tumbling window. In this example, the data source is a CSV file, s
o data streams are assigned to the windows based on processing time. In actual scen
arios, you can assign data streams based on either processing time or event time to
suit your business requirements.
    .window(TumblingProcessingTimeWindows.of(Time.minutes(5)))
    -- The trigger. You can obtain the aggregated results before the windows are re
moved.
    .trigger(ContinuousProcessingTimeTrigger.of(Time.minutes(1)))
    .aggregate(
    -- The aggregate function used to aggregate the results based on the specified
query dimensions.
    new AggregateFunction<
        Tuple5<String, String, String, String, Integer>,
        RoaringBitmap,
        RoaringBitmap>() {
            @Override
```

```
            public RoaringBitmap createAccumulator() {
                return new RoaringBitmap();
            }
            @Override
            public RoaringBitmap add(
                Tuple5<String, String, String, String, Integer> in,
                RoaringBitmap acc) {
                -- Run Roaring bitmap functions for the 32-digit UIDs to remove dup
licate UIDs.
                acc.add(in.f4);
                return acc;
            }
            @Override
            public RoaringBitmap getResult(RoaringBitmap acc) {
                return acc;
            }
            @Override
            public RoaringBitmap merge(
                RoaringBitmap acc1, RoaringBitmap acc2) {
                return RoaringBitmap.or(acc1, acc2);
            }
        },
    -- The Window function used to generate the aggregated results.
    new WindowFunction<
        RoaringBitmap,
        Tuple6<String, String, String, String, Timestamp, byte[]>,
        Tuple,
        TimeWindow>() {
            @Override
            public void apply(
                Tuple keys,
                TimeWindow timeWindow,
                Iterable<RoaringBitmap> iterable,
                Collector<
                Tuple6<String, String, String, String, Timestamp, byte[]>> out)
                throws Exception {
                RoaringBitmap result = iterable.iterator().next();
                // Optimize the results of Roaring bitmap functions.
                result.runOptimize();
                // Convert the results of Roaring bitmap functions into byte arrays
and store them in Hologres.
                byte[] byteArray = new byte[result.serializedSizeInBytes()];
                result.serialize(ByteBuffer.wrap(byteArray));
                // The Tuple6 parameter specifies that the data streams are process
ed based on the lifecycle of the windows. The value of the parameter is of the TIME
STAMP type, in seconds.
                out.collect(
                    new Tuple6<>(
                        keys.getField(0),
                        keys.getField(1),
                        keys.getField(2),
                        keys.getField(3),
                        new Timestamp(
                            timeWindow.getEnd() / 1000 * 1000),
```

```
                        byteArray));
            }
        });
```

iv. Write the deduplicated data to the Hologres aggregation result table.

Write the deduplicated data to the Hologres aggregation result table named dws_app. The results of Roaring bitmap functions are stored as byte arrays in Hologres. The following sample code provides an example:

```
 -- Convert the processed results into a table.
Table resTable =
    tableEnv.fromDataStream(
        processedSource,
        $("country"),
        $("prov"),
        $("city"),
        $("ymd"),
        $("timest"),
        $("uid32_bitmap"));
-- Create an aggregation result table in Hologres. Store the results of Roaring bit
map functions into the table as byte arrays.
String createHologresTable =
    String.format(
        "create table sink("
        + "  country string,"
        + "  prov string,"
        + "  city string,"
        + "  ymd string,"
        + "  timetz timestamp,"
        + "  uid32_bitmap BYTES"
        + ") with ("
        + "  'connector'='hologres',"
        + "  'dbname' = '%s',"
        + "  'tablename' = '%s',"
        + "  'username' = '%s',"
        + "  'password' = '%s',"
        + "  'endpoint' = '%s',"
        + "  'connectionSize' = '%s',"
        + "  'mutatetype' = 'insertOrReplace'"
        + ")",
    database, dwsTableName, username, password, endpoint, connectionSize);
tableEnv.executeSql(createHologresTable);
-- Write the results to a table named dws_app.
tableEnv.executeSql("insert into sink select * from " + resTable);
```

3. Query UVs.

Calculate UVs based on data in the dws_app table. Perform an aggregation operation based on query dimensions and query the number of bits in a bitmap. This way, you can calculate the UVs under the conditions specified by the GROUP BY clause.

- Example 1: Query the UVs of each city on a specific day

```
-- Perform the following RB_AGG operation to query data. You can disable the three-st
age aggregation feature for better performance. You can enable or disable this featur
e based on your requirements. By default, the feature is disabled.
set hg_experimental_enable_force_three_stage_agg=off;
SELECT  country
        ,prov
        ,city
        ,RB_CARDINALITY(RB_OR_AGG(uid32_bitmap)) AS uv
FROM    dws_app
WHERE   ymd = '20210329'
GROUP BY country
         ,prov
         ,city
;
```

- Example 2: Query the UVs and PVs of each province within a specific period of time

```
-- Perform the following RB_AGG operation to query data. You can disable the three-st
age aggregation feature for better performance. You can enable or disable this featur
e based on your requirements. By default, the feature is disabled.
set hg_experimental_enable_force_three_stage_agg=off;
SELECT  country
        ,prov
        ,RB_CARDINALITY(RB_OR_AGG(uid32_bitmap)) AS uv
        ,SUM(1) AS pv
FROM    dws_app
WHERE   time > '2021-04-19 18:00:00+08' and time < '2021-04-19 19:00:00+08'
GROUP BY country
         ,prov
;
```

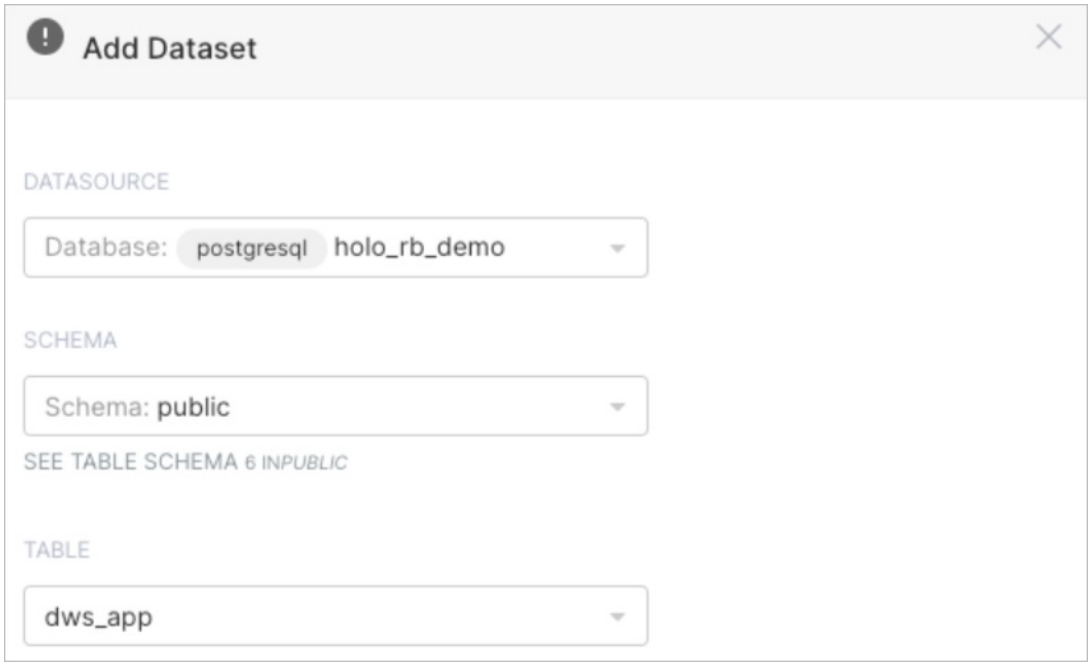
4. Visually display data.

   In most cases, you need to use Business Intelligence (BI) tools to visually display the calculated UVs and PVs. In the calculation process, RB_CARDINALITY and RB_OR_AGG functions are used to aggregate data. Therefore, BI tools must support custom aggregation functions. You can use common BI tools such as Apache Superset and Tableau.

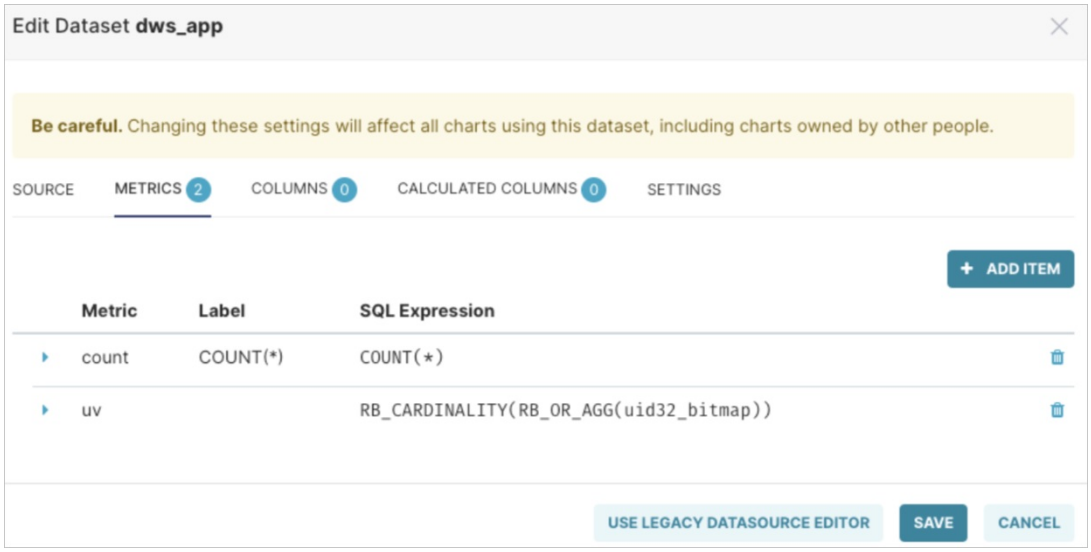
   - Apache Superset

     a. Connect Apache Superset to Hologres. For more information, see Apache Superset.

b. Set the dws_app table as a dataset.



c. Create a metric named UV in the dataset by using the expression that is shown in the following figure.




```
RB_CARDINALITY(RB_OR_AGG(uid32_bitmap))
```

Then, you can start to explore data.

d. Optional. Create a dashboard.

For more information about how to create a dashboard, see Creating Your First Dashboard.

- Tableau

a. Connect Tableau to Hologres. For more information, see Tableau.

You can use pass-through functions in Tableau to customize functions. For more information, see Pass-Through Functions (RAWSQL).

b. Create a calculation field by using the expression that is shown in the following figure.


```
RAWSQLAGG_INT("RB_CARDINALITY(RB_OR_AGG(%1))", [Uid32 Bitmap])
```

Then, you can start to explore data.

c. Optional. Create a dashboard.

For more information about how to create a dashboard, see Create a Dashboard.

3.4. User profile analysis

3.4.1. User profile analysis

This topic describes the best practices for tagging and profile analysis in Hologres.

Background

Profile analysis is the process of exploring user interests and analyzing group characteristics based on the natural, behavioral, and preference properties of intended users. User profiling is an important means to depict the comprehensive characteristics of an individual user or a user group. It provides information such as user preferences and behavior for operation analysis personnel to optimize operational strategies. It also provides accurate role information for targeted product designs. A profiling system typically integrates the user characteristics processing and profile analysis features to provide real-time group analysis and identification after offline processing of characteristics, mapping of tags, and loading of ad hoc analysis data.

Profile analysis has been widely applied in a variety of industries and has become an important means to optimize operational strategies and implement refined operations and precise marketing. The following examples are typical scenarios for which profile analysis is suited.

- Advertising: Profile analysis provides insights into users to implement targeted advertising.
- Gaming: Profile analysis provides analysis on churn rates so that operational strategies can be adjusted to increase user viscosity.
- Education: Profile analysis provides analysis on course quality to improve the retention rate.

However, profile analysis faces challenges in system stability, maintainability, and scalability that are caused by complex data, large amounts of data, and query modes.

- The O&M personnel must maintain multiple data links for real-time offline processing, which leads to heavy workloads. Traditional online analytical processing (OLAP) engines use an architecture in which storage is coupled with computing. As a result, in scenarios in which computing and storage resources are not proportionate to each other, resource waste occurs and system scaling and migration costs are high.
- The operations personnel require flexible identification capabilities. To describe a single user, thousands of dimensions may be required, including property and behavior data. Multidimensional OLAP (MOLAP) provides responses within milliseconds but lacks flexibility. Relational OLAP (ROLAP) provides flexibility but takes time to respond and compromises performance.

Hologres solutions

To address the preceding issues, Hologres allows you to determine a solution that offers high performance and scalability by configuring data links, selecting plug-in libraries, and considering the size of your business system.

- Data links

  Hologres supports real-time offline data processing without the need to maintain multiple data links. This prevents common issues such as data inconsistency and data silos. Hologres provides the following benefits in data integration:

  - Hologres is seamlessly integrated with DataWorks. Complex data dependency issues can be resolved by making access configurations, and stable offline data processing and loading processes are provided.
  - Hologres provides row-oriented storage based on the log-structured merge (LSM) structure for scenarios that involve real-time writes. Hologres is integrated with Flink to provide stable performance support for real-time tagging and real-time characteristic processing.
  - Hologres provides the federated query capability and allows access to external data storage services such as MaxCompute, Object Storage Service (OSS), and other Hologres instances by using foreign tables.

- Profile computing

  Hologres is compatible with the PostgreSQL ecosystem and provides an abundance of built-in functions. In addition, many efficient profile computing plug-ins have been developed on top of the best practices of Alibaba Cloud and its users.

  - Precise deduplication: Roaring bitmap functions

    Hologres supports Roaring bitmaps. It supports union and intersection operations on sets and bitwise aggregate operations by using efficient compressed bitmaps. Roaring bitmaps are suitable for computing tables that contain unique data with multiple dimensions and are typically used in deduplication (unique visitor (UV) computing), tag-based filtering, and quasi-real-time user profile analysis.

- Action data-based user identification: Target user identification functions

  In action data-based user identification scenarios, action data is recorded in a table by day or hour. Users who take specific actions within a specific period of time cannot be directly queried because the action data is scattered across multiple rows. The action data table must be joined with itself multiple times to query such users. Assume that you want to query users whose actions are `[click Shopping cart]` and [view Favorites] with the ds value ranging from 20200216 to 20200218.

  Hologres provides the `bit_construct` and `bit_or`, and `bit_match` functions to minimize performance burdens of JOIN statements and simplify SQL operations. These functions are used to filter users. Users whose uid meet specific filter conditions are stored as bit arrays. Then the `bit_match` function is used to perform AND operations on the bit arrays. The following statement shows an example.

  ```
  WITH tbl as (
  SELECT uid, bit_or(bit_construct(
    a := (action='click' and page='Shopping cart'),
    b := (action='view' and page='Favorites'))) as uid_mask
    FROM ods_app_dwd
  WHERE ds > '20210218' AND event_time < '20210216'
  GROUP BY uid )
  SELECT uid from tbl where bit_match('a&b', uid_mask);
  ```

  - `bit_construct`: returns values for expressions and stores the values in bit arrays. For example, this function returns `[1,0], [0,0], [0,1]...` for conditions a and b in the preceding SQL statement.
  - `bit_or`: performs OR operations on the two bit arrays to query users who meet the filter conditions.
  - `bit_match`: determines whether a bit array matches an expression. For example, for the `a&b` expression, this function returns True for `[1,1]` and False for `[1,0]`.

- Funnel analysis: Funnel analysis functions

  Funnel analysis is a popular conversion analytics method used to understand user behavior and calculate conversion rates. Funnel analysis is widely used for data operations and analysis scenarios such as the analysis of user behavior, application data traffic, and product goal conversion.

  You can use the windowFunnel function to query events from a sliding time window. This function calculates the maximum number of events that can match the query conditions. Retention analysis is the most common and typical scenario where user growth is analyzed. In most cases, you can use charts to analyze user retention. The funnel and retention functions can be used to calculate user retention and conversion rates, reduce overheads in complex JOIN statements, and improve performance.

- Vector processing: Vector processing

  Proxima is a high-performance software library developed by Alibaba DAMO Academy. It allows you to search for the nearest neighbors of vectors. Proxima provides higher stability and performance than similar open source software such as Facebook AI Similarity Search (Fassi). Proxima provides basic modules that have leading performance and effects in the industry and allows you to search for similar images, videos, or human faces. Hologres is deeply integrated with Proxima to provide a high-performance vector search service. K-nearest neighbors (KNN) searches, Radius nearest neighbors (RNN) searches, and DOT_PRODUCT are supported.

- Solutions

Different cost and performance requirements are imposed at different development stages of profiling systems. Hologres provides the following solutions based on practical experience and factors such as system data size, implementation cost, and query performance:

- Wide tables

  This solution is suited for scenarios in which less than 1,000 tags are used and data is infrequently updated. Stable property tables are aggregated into wide tables offline, and JOIN operations on multiple tables are converted into operations on a single wide table. If new tags are required, columns are added to the wide table for these tags. This enables flexible tag-based computing by using tables. For more information, see Wide tables.

- Roaring bitmaps

  This solution is suited for scenarios in which large amounts of data is involved, a large number of tags are used, and deduplication is required. The structured storage of Roaring bitmaps implements natural deduplications, prevents JOIN overheads, simplifies operations, and accelerates data retrieval. For more information, see Roaring bitmaps.

- Summary

  Hologres supports a wide range of profile analysis plug-ins and delivers excellent performance. It is widely used in tag computing and profile analysis scenarios by multiple core businesses within Alibaba Group, such as Alimama, search applications, and AMap, and many public cloud users. The service scalability and stability of Hologres have been tested in production. Hologres has proven itself as the best choice for building a profile analysis platform with high stability and scalability and low development and O&M costs.

3.4.2. Wide tables

This topic describes the best practices for using wide tables to perform tag computing in Hologres.

Context

In the offline data warehouse model, user tag data is stored in multiple theme- and dimension-oriented tables. This is helpful to build a tag system and maintain and manage data. However, if such a data model is used in online profile analysis to organize tag data, multiple tag tables must be joined to filter tags, which is too costly for database services.

Solution

In the wide table solution, stable property tables are aggregated into wide tables offline and JOIN operations on multiple tables are converted into operations on a single wide table. If new tags are required, columns are added to the wide table for these tags. This solution is suited for the following scenarios:

- Scenarios in which less than 1,000 tags are used.
- Scenarios in which data is infrequently updated.

When data is stored in wide tables, the AND, OR, and NOT operations on filter conditions of multiple columns are automatically processed by the optimization mechanism of column-oriented storage, which is more efficient than join operations. In addition, Hologres supports column-oriented storage, which prevents I/O operations from increasing. Traditional database modeling and development applications can be used in this solution.

Use example

In the following example, a wide table is used for profile analysis. SQL statements are executed to query men (`[gender = Male]`) in the Zhejiang province ([province = Zhejiang]) whose marriage status is married in the dws_userbase table. We recommend that you set proper indexes for the table based on the query mode to improve query performance. For more information, see CREATE TABLE.

```
-- The wide table.
BEGIN;
CREATE TABLE dws_userbase
(
  uid text not null primary key,
  province text,
  gender text,
  married text
  ...             -- Other property columns.
);
call set_table_property('dws_userbase', 'distribution_key', 'uid');
call set_table_property('dws_userbase', 'bitmap_columns', 'province,gender,married');
END;
-- Query based on basic properties.
SELECT count(distinct uid) as cnt,
       married
FROM dws_userbase ub
WHERE province = 'Zhejiang' and gender = 'Male'
GROUP BY married;
```

3.4.3. Roaring bitmaps

In scenarios in which more than 1,000 tags are used, the solution of using wide tables for tag computing is not suitable. This is because the update efficiency decreases when the number of columns increases. This topic describes how to perform tag computing and profile analysis in such scenarios.

Context

Hologres is compatible with the PostgreSQL ecosystem and supports Roaring bitmap functions. Indexes are created for tag tables. User IDs are encoded and stored as bitmaps. Relational operations are converted into intersection, union, and difference operations of bitmaps to improve the performance of real-time computing. In scenarios that require analysis of large amounts of user properties, Roaring bitmaps can be used to respond to queries within sub-seconds.

Scenarios

Roaring bitmaps are suitable for the following scenarios:

- Scenarios in which large amounts of tags are used: In such scenarios, JOIN operations are required for many large tables. The `BITMAP_AND` function can be used to replace JOIN operations to reduce memory consumption. The bitmap plug-in repository can improve the CPU utilization by 1% to 2% by means of Single Instruction Multiple Data (SIMD)-based optimization.
- Scenarios that involves large amounts of data and requires deduplication: Bitmaps provide intrinsic deduplication capabilities to prevent unique vector (UV) computing and memory overheads.

Tag types

Tags in profiling systems can be classified into the following types. Different types of tags use different computing modes. Specific types of tags must be converted into and stored as bitmaps.

- Property tags: Property tags describe user properties such as gender, province, and marriage status. Property tags are stable and can be filtered based on precise filter conditions. For these tags, the bitmap compression ratio is high, and bitmaps are suitable for related operations.
- Action tags: Action tags describe action characteristics of users and depict what users do at specific points in time. User actions include page views, purchases, and active logons. Action data is frequently updated and requires range scanning and aggregate filtering. For action tags, the bitmap compression ratio is low, and bitmaps are not suitable for related operations.

Property tags

Property tags describe user properties. Property tags are stable and can be filtered based on precise filter conditions. Bitmaps can be used for efficient compression and operations.

- Solution

  Assume that a data management platform (DMP) contains two property tag tables. The dws_userbase table describes basic user properties, and the dws_usercate_prefer table describes user preferences.

  If you want to obtain the number of users who meet the `[province = Beijing] & [cate_prefer = Fashion]` filter condition, you can perform association, filtering, and deduplication operations. However, in scenarios that involve large amounts of data, association and deduplication operations may bring heavy performance burdens.

  The bitmap-based optimization solution uses bitmap tables that contain pre-created tags to reduce the costs of ad hoc operations. In this example, data in the preceding tables are split by column to create two bitmap tables . Then, a bitwise AND operation is performed to obtain the users who meet the preceding filter condition. The rb_dws_userbase_province table describes the bitmap relationship between the province and uid columns, and the rb_dws_usercate_prefer_cprefer table describes the bitmap relationship between the cate_prefer and uid columns.

  However, the preceding solution has problems. When columns have hierarchical relationships, such splitting and operations may cause computing errors. Data in the dws_shop_cust table that describes the information about fresh, existing, and potential customers is split by column. The rb_dws_shop_cust_shop_id bitmap table that describes shop IDs and the `rb_dws_shop_cust_cust_type` bitmap table that describes customer types are created. If you filter the customers who meet the `[shop_id = A] & [cust_type = Fresh]` filter condition, you obtain a result of uid `[1]`. However, a uid column value of 1 that corresponds to a cust_type column value of Fresh does not exist. This is because the cust_type and shop_id columns are correlated. In a data warehouse model, cust_type is a metric for shop_id and cannot be used independently. You must use shop_id in combination with cust_type to create the rb_dws_shop_cust_sid_ctype bitmap table to prevent such error.

  You must compress the uid values into bitmaps and then perform bitwise AND, OR, and NOT operations to compute tags.

- Procedure

- Encode user information

  User IDs may be strings. However, bitmaps contain only integers. Therefore, you must create a table that contains an auto-increment field by using the SERIAL or BIGSERIAL data type before you can encode uid values of the string type into integers.

```
--Create a dictionary table.
CREATE TABLE dws_uid_dict (
    encode_uid bigserial,
      uid text primary key
);
--Insert uid values from the tag table.
INSERT INTO dws_uid_dict(uid)
SELECT uid
FROM dws_userbase ON conflict DO NOTHING;
```

  Encoded user IDs maintain continuity and can be easily stored as bitmaps. The following figure shows an example in which bitmap2 contains sparse data and delivers storage efficiency much lower than bitmap1. Therefore, if user IDs are encoded, storage costs can be reduced and computing efficiency can be improved.

	0	1	2	3	4	5	6	7	...	3461	...	65536
bitmap1	1	1	1	1	1	1	1	1	...	0	...	1
bitmap2	0	0	1	0	0	0	0	0	...	0	...	1

  Sparse numeric data can be encoded, but additional performance overheads may occur. For example, advertising DMPs not only require high-performance profiling, but also require real-time output of user details. If real-time output of user details is necessary, user ID tables must be joined to restore encoded user IDs, which causes additional performance overheads. You must determine whether to encode user IDs based on your specific scenario. We have different recommendations for different cases.

  - For user IDs of the string type, we recommend that you encode them.
  - For user IDs of the integer type that require frequent restoration of encoded user IDs, we do not recommend that you encode them.
  - For user IDs of the integer type that do not require frequent restoration of encoded user IDs, we recommend that you encode them.

- Process and query bitmaps

  Split the dws_userbase and dws_shop_cust tables into one bitmap table that contains the province and gender columns. The gender column values contain only Male and Female. Compressed bitmaps can be distributed only on two nodes in a cluster. As a result, computing and storage resources are not evenly distributed and the cluster resources are not fully used. In this case, the bitmaps must be split into multiple segments and distributed in the cluster for concurrent execution. For example, you can execute the following SQL statements to split the bitmaps into 65,536 segments:

```
-- Create a wide table named dws_userbase.
BEGIN;
CREATE TABLE dws_shop_cust
(
  uid text not null primary key,
  shop_id text,
    cust_type text
);
call set_table_property('dws_shop_cust', 'distribution_key', 'uid');
END;
-- Create a bitmap extension.
CREATE EXTENSION roaringbitmap;
BEGIN;
CREATE TABLE rb_dws_userbase_province (
    province text,
  bucket int,
    bitmap roaringbitmap
);
call set_table_property('rb_dws_userbase_province', 'distribution_key', 'bucket');
END;
BEGIN;
CREATE TABLE rb_dws_shop_cust_sid_ctype (
    shop_id text,
  cust_type text,
  bucket int,
    bitmap roaringbitmap
);
call set_table_property('rb_dws_shop_cust_sid_ctype', 'distribution_key', 'bucket');
END;
-- Write data into the bitmap table.
INSERT INTO rb_dws_userbase_province
SELECT province,
       encode_uid / 65536 as "bucket",
       rb_build_agg(b.encode_uid) AS bitmap
FROM dws_userbase a join dws_uid_dict b on a.uid = b.uid
GROUP BY province, "bucket";
INSERT INTO rb_dws_shop_cust_sid_ctype
SELECT shop_id,
       cust_type,
       encode_uid / 65536 AS "bucket",
       rb_build_agg(b.encode_uid) AS bitmap
FROM dws_shop_cust a
JOIN dws_uid_dict b ON a.uid = b.uid
GROUP BY shop_id, cust_type, "bucket";
```

If you want to obtain the users that meet the `[shop_id = A] & [cust_type = Fresh] & [provinc e = Beijing]` filter condition, you can perform related AND, OR, and NOT operations on the bitmaps. You can execute the following SQL statements:

```
SELECT SUM(RB_CARDINALITY(rb_and(ub.bitmap, uc.bitmap)))
FROM
  (SELECT rb_or_agg(bitmap) AS bitmap,
          bucket
   FROM rb_dws_userbase_province
   WHERE province = 'Beijing'
   GROUP BY bucket) ub
JOIN
  (SELECT rb_or_agg(bitmap) AS bitmap,
          bucket
   FROM rb_dws_shop_cust_sid_ctype
   WHERE shop_id = 'A'
     AND cust_type = 'Fresh'
   GROUP BY bucket) uc ON ub.bucket = uc.bucket;
```

Action tags

Typically, fact tables are organized by time. For example, user action tables are organized by day. User data for a specific day contains only limited entries. If such data is compressed into bitmaps, row-oriented storage overheads may cause storage space to be wasted. In addition, in typical computing modes of fact tables, data of multiple days must be aggregated for filtering. If bitmaps are used, they must be expanded before they can be aggregated for operations. Such data frequently changes and requires real-time update. In the `[option->bitmap]` storage structure, data that needs to be updated cannot be identified. Therefore, bitmaps are not suitable for action data, aggregation, or real-time update.

In scenarios that involve action tags, Hologres can use the original storage format. When fact tables and property tables need to be joined, bitmaps can be generated for the filter results of fact tables and then joined with the bitmap indexes of property tables. Because bitmap index tables use bucket as the distribution key, local join operations can improve the join performance.

You can execute the following SQL statements to obtain users that meet the `[province=Beijing] & [shop_id=A AND No purchase for 7 days]` filter condition.

```
-- Create an action table.
BEGIN;
CREATE TABLE dws_usershop_behavior
(
  uid int not null,
  shop_id text not null,
  pv_cnt int,
  trd_cnt int,
  ds integer not null
);
call set_table_property('dws_usershop_behavior', 'distribution_key', 'uid');
COMMIT;
-- Encode the action table.
BEGIN;
CREATE TABLE dws_usershop_behavior_bucket
(
  encode_uid int not null,
  shop_id text not null,
  pv_cnt int,
  trd_cnt int,
```

```
  ds int not null,
  bucket int
);
CALL set_table_property('dws_usershop_behavior_bucket', 'orientation', 'column');
call set_table_property('dws_usershop_behavior_bucket', 'distribution_key', 'bucket');
CALL set_table_property('dws_usershop_behavior_bucket', 'clustering_key', 'shop_id,encode_u
id');
COMMIT;
-- Write fact data.
INSERT INTO dws_usershop_behavior_bucket
SELECT *,
             encode_uid,
       shop_id,
       pv_cnt,
       trd_cnt,
       encode_uid / 65536
FROM dws_usershop_behavior a JOIN dws_uid_dictionary b
on a.uid = b.uid;
-- Join fact data and property data.
SELECT sum(rb_cardinality(bitmap)) AS cnt
FROM
  (SELECT rb_and(ub.bitmap, us.bitmap) AS bitmap,
          ub.bucket
   FROM
     (SELECT rb_or_agg(bitmap) AS bitmap,
             bucket
      FROM rb_dws_userbase_province
      WHERE province = 'Beijing'
      GROUP BY bucket) AS ub
   JOIN
     (SELECT rb_build_agg(uid) AS bitmap,
             bucket
      FROM
        (SELECT uid,
                bucket
         FROM dws_usershop_behavior_bucket
         WHERE shop_id = 'A' AND ds > to_char(current_date-7, 'YYYYMMdd')::int
         GROUP BY uid,
                  bucket HAVING sum(trd_cnt) = 0) tmp
      GROUP BY bucket) us ON ub.bucket = us.bucket) r
```

Offline processing of bitmaps

You can choose to process bitmap data offline to prevent bitmap data computing from affecting your business. You can load data from foreign tables in MaxCompute or Hive. Bitmaps are processed in similar manners both online and offline. Data can be generated through encoding and aggregation. The following code provides an example on how to create bitmap data offline in MaxCompute.

```
-- Select a project.
USE bitmap_demo;
-- Create a source table.
CREATE TABLE mc_dws_uid_dict (
encode_uid bigint,
bucket bigint,
```

```
uid string
);
CREATE TABLE mc_dws_userbase
(
  uid string,
  province string,
  gender string,
  marriaged string
);
-- Encode the uid values.
-- Calculate the new uid values.
WITH uids_to_encode AS
  (SELECT DISTINCT(ub.uid),
          CAST(ub.uid / 65336 AS BIGINT) AS bucket
   FROM mc_dws_userbase ub
   LEFT JOIN mc_dws_uid_dict d ON ub.uid = d.uid
   WHERE d.uid IS NULL),
-- Calculate the number of uids to be encoded in each bucket. Use the SUM function to obtai
n the bucket offset.
uids_bucket_encode_offset AS
  (SELECT bucket,
          sum(cnt) over (ORDER BY bucket ASC) - cnt AS bucket_offset
   FROM
     (SELECT count(1) AS cnt,
             bucket
      FROM uids_to_encode
      GROUP BY bucket) x),
-- Calculate the maximum number of encoded uids.
dict_used_id_offset AS
  (SELECT max(encode_uid) AS used_id_offset FROM mc_dws_uid_dict)
-- New uids = Maximum number of encoded uids + Bucket offset + Row number
INSERT INTO mc_dws_uid_dict
SELECT
       COALESCE((SELECT used_id_offset FROM dict_used_id_offset),0) + bucket_offset + rn,
       bucket,
       uid
FROM
  (SELECT row_number() OVER (partition BY ub.bucket ORDER BY ub.uid) AS rn,
          ub.bucket,
          bo.bucket_offset,
          uid
   FROM uids_to_encode ub
   JOIN uids_bucket_encode_offset bo ON ub.bucket = bo.bucket) j
-- Create bitmap-related functions.
add jar function_jar_dir/mc-bitmap-functions.jar as mc_bitmap_func.jar -f;
create function mc_rb_cardinality as com.alibaba.hologres.RbCardinalityUDF using mc_bitmap_
func.jar;
create function mc_rb_build_agg as com.alibaba.hologres.RbBuildAggUDAF using mc_bitmap_func
.jar;
-- Create a bitmap table and write data to the table.
CREATE TABLE mc_rb_dws_userbase_province
(
  province string,
  bucket int,
```

```
  bitmap string
);
INSERT INTO mc_rb_dws_userbase_province
SELECT province,
       b.bucket_num,
       mc_rb_build_agg(b.encode_uid) AS bitmap
FROM mc_dws_userbase a join mc_dws_uid_dict b on a.uid = b.uid
GROUP BY province, b.bucket_num;
```

Execute the following statements in Hologres:

```
-- Create a MaxCompute table.
CREATE TABLE mc_rb_dws_userbase_province (
    province text,
  bucket int,
    bitmap roaringbitmap
) server odps_server options(project_name 'bitmap_demo', table_name 'mc_rb_dws_userbase_pro
vince');
-- Write bitmap data from the MaxCompute table to Hologres.
INSERT INTO rb_dws_userbase_province
SELECT province,
       bucket::INT,
       roaringbitmap_text(bitmap, FALSE)
FROM mc_rb_dws_userbase_province;
```

After the preceding steps are performed, data is loaded to Hologres. You can then perform bitwise operations to speed up queries.

- You can use mc-bitmap to compute bitmap data in MaxCompute.
- For more information about the offline processing of bitmaps, see Batch UV calculation.

Real-time processing of bitmaps

In real-time computing scenarios, you can use Hologres in combination with Flink to perform real-time deduplication for user tags based on Roaring bitmaps. Perform the following steps:

1. Use a user ID dictionary table as the dimension table and use the INSERT ON CONFLICT statement of Hologres to add user IDs. Then, join the dimension and action tables in Flink.
2. Aggregate the resulting table with Roaring bitmaps by tag.
3. Write the resulting bitmaps into the bitmap table in Hologres.

For more information, see Real-time tags.

3.4.4. Real-time tags

This topic describes the best practices for real-time tag computing in Hologres.

Context

In tag computing and profile analysis, the real-time reverse transmission of processed data and real-time tag generation capabilities are of great importance.

- Real-time reverse transmission of processed data: Tags of users that are obtained from profile analysis are collected, such as the click and conversion rates of advertising systems. Such data serves

as a basis for real-time adjustment of decisions.

- Real-time tag generation: Real-time characteristics can be understood as the real-time performance of a user in an activity. Typical real-time characteristics include the number of views in the last N days and the products added to favorites within a day. Such characteristics can be analyzed to push operational strategies to targeted users, motivate purchase intention, and eventually make the deal.

Hologres is integrated with Flink to support high-performance data write and update operations. Data can be queried immediately after it is written. Real-time feedback and real-time tag generation can be implemented by a combination of Hologres and Flink in various business scenarios.

Real-time reverse transmission of processed data

Real-time reverse transmission of processed data is supported by the real-time capabilities provided by Hologres and Flink. Flink data is written to Hologres in real time. Then, tags are computed in real time by using the built-in profile analysis plug-ins of Hologres, such as the funnel and retention functions. This way, real-time decisions can be made for the data.

Real-time tag generation

The process for generating real-time tags is typically complex and requires a combination of Flink and Hologres. For example, JOIN operations of dimension tables are required. Procedure:

1. Create a real-time log table named dwd_user_visit_log in Flink. This table stores real-time action data from DataHub and Message Queue for Apache Kafka.
2. Create a historical action table named dws_user_visit in Hologres. This table stores action data within `T-N to T-1` days. Such action data is used as the initial data for calculating tags. The action table is used as a dimension table in Flink.
3. Join the real-time log table and the historical action table by using the Flink dimension table and calculate a new tag.
4. Update the tag in real time and store it in a Hologres table.
5. Synchronize the tag to an online store such as ApsaraDB for Redis for real-time callback.
6. If you use the initial data for tag computing, write data from Hologres to ApsaraDB for Redis.