ALIBABA CLOUD

阿里云

交互式分析Hologres

最佳实践

文档版本：20220711

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.数仓搭建

1.1. 使用空间函数查询数据方法

Hologres支持使用PostGIS的空间函数查询包含空间数据的表。本文以导入本地数据至Hologres并使用空间函数查询数据为例，为您介绍在Hologres中使用PostGIS空间函数的操作方法。

前提条件

- 已创建Hologres实例，创建实例方法请参见购买Hologres。
- Hologres实例中已创建数据库，创建数据库方法请参见创建数据库。
- 已下载好本示例所提供的示例数据表，数据表下载链接如下。
  - accommodations数据表
  - zipcodes数据表

背景信息

PostGIS是数据库PostgreSQL的空间扩展，PostGIS可以提供空间对象、空间索引、空间操作函数和空间操作符等空间信息服务功能。

本文为您提供了示例数据表，包含各类空间信息（精度、纬度、坐标、距离等）。您可通过如下操作使用Hologres的HoloWeb，创建表并一键导入本地数据至**accommodations**表和**zipcodes**表中，通过PostGIS的空间函数可查询两个表中所包含的空间信息数据。

操作步骤概述

操作流程	描述
步骤一：创建表	在Hologres实例的数据库中，创建**accommodations**表用于存储住宿地的地理位置（经度和纬度）、列表名称和其他数据。创建**zipcodes**表用于存储柏林邮政编码数据。
步骤二：导入测试数据	通过HoloWeb一键导入本地数据源至**accommodations**表和**zipcodes**表中。
步骤三：使用空间函数查询数据	使用PostGIS的空间函数查询表中的空间数据。

步骤一：创建表

根据如下步骤，在数据库中创建**accommodations**表和**zipcodes**表。

1. 登录HoloWeb控制台SQL编辑器页面。
2. 单击**新增SQL窗口**，选择已创建并登录的Hologres实例和数据库。
3. 加载PostGIS插件。

   在命令编辑区域，输入如下SQL命令，单击**运行**。

   ```
   create extension if not exists postgis; -- 加载PostGIS插件
   ```

4. 创建**accommodations**表。

   运行如下SQL语句，创建**accommodations**表，用于存储住宿地的地理位置（经度和纬度）、列表名称和其他数据。

> **说明** 创建成功后，您可右击左侧**表目录**下的**public > 表**，单击**刷新**查看表创建结果。或通过**运行日志**，查看表是否创建成功。

```
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. 创建**zipcodes**表。

   运行如下SQL语句，创建**zipcodes**表，用于存储柏林邮政编码数据。

   > **说明** 创建成功后，您可右击左侧**表目录**下的**public > 表**，单击**刷新**查看表创建结果。或通过**运行日志**，查看表是否创建成功。

```
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)
 );
```

步骤二：导入测试数据

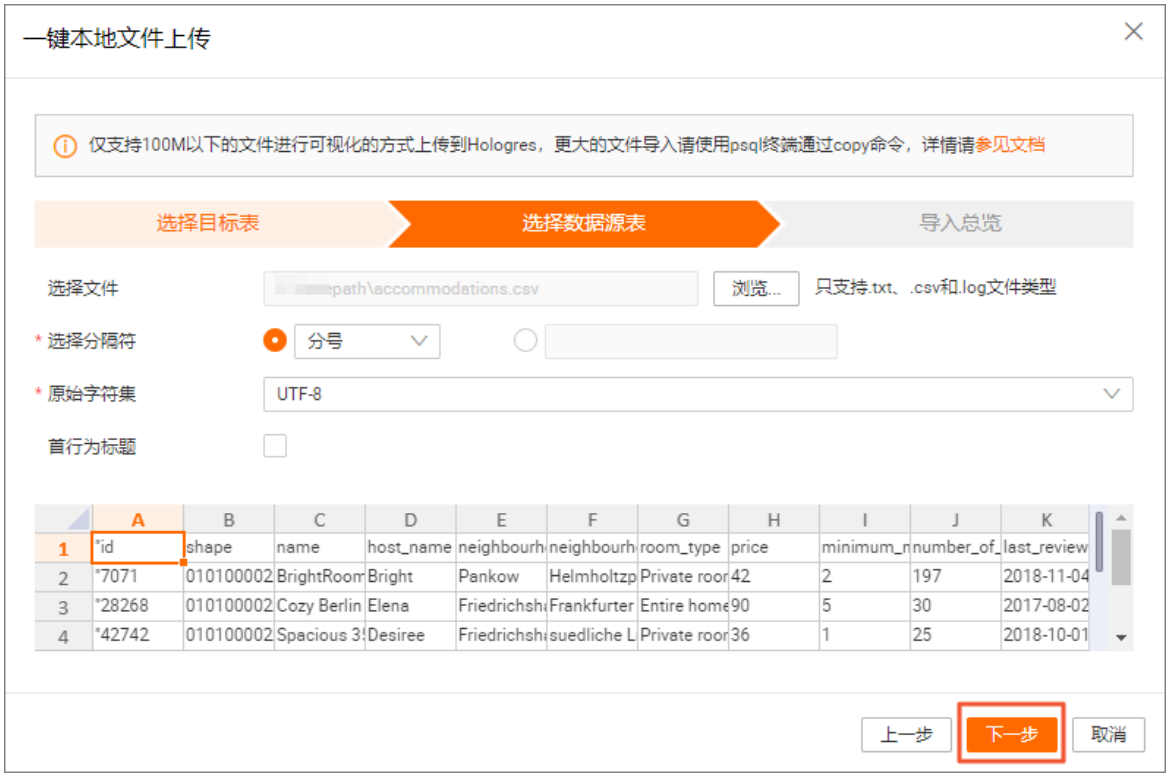
表创建成功后，您可通过HoloWeb的**一键本地文件导入**功能，向已创建成功的**accommodations**表和**zipcodes**表中导入数据。

1. 在HoloWeb控制台中，单击顶部**数据方案**。
2. 在**数据方案**页面，单击左侧**一键本地文件导入**，在右侧界面中单击**新建数据导入**。
3. 选择需要上传数据的表。

   在弹出的**一键本地文件上传**对话框，填写作业名称并选择已创建的实例、数据库和表（**accommodations**表或**zipcodes**表），单击**下一步**。

4. 选择需要上传的数据和格式。

   在**选择数据源表**页签下，您可根据如下内容进行配置，配置完成后单击下一步。



参数项	参数项选择描述
选择文件	单击**浏览**，上传需要导入的数据表。仅支持后缀为.txt、.csv、.log类型的文件。请选择前提条件中为您提供的**accommodations**数据表和**zipcodes**数据表。
选择分隔符	选择数据的分割符，本示例中选择分号。 **说明** 您也可以根据自身数据的内容，单击分隔符右侧的选项，使用自定义分隔符。
原始字符集	本示例中选择**UTF-8**。
首行为标题	默认为空。如您导入的数据表中的数据第一行为标题，请选中该选项。

5. 确认需要导入的数据信息。

   在导入总览页签下，您可查看需要导入的信息是否正确，完成后单击**执行**。

6. 查看数据导入执行结果。

   执行完成后，系统会提示您导入状态是否成功。如执行失败，系统将提示您失败原因，您可根据原因进行修改后重新导入数据。

   您也可以通过如下代码，在SQL编辑器中查看数据的记录数量以及表中的内容。

   - 查看表中记录数量

     accommodations表中有记录数22248条，**zipcodes**表中有记录数190条。

     ```
     select count(*) from accommodations; -- 查询accommodations表中的记录数量
     select count(*) from zipcode; -- 查询zipcode表中的记录数量
     ```

   - 查看表中内容

     ```
     select * from accommodations; -- 查询accommodations表中的记录
     select * from zipcode; -- 查询zipcode表中的记录
     ```

步骤三：使用空间函数查询数据

表数据创建并导入成功后，您可根据需求使用空间函数查询空间数据，部分查询操作示例如下。空间函数的语法详情，请参见空间函数。

- 查询accommodations表中的记录数，其中SRID取值为4326。
  - 代码示例：

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

  - 运行结果：

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

- 使用WKT格式查询符合某些附加属性的几何体对象。您可以验证此邮政编码数据是否也存储在WGS 84中，该系统使用SRID取值为4326。

> **说明** 空间数据必须存储在同一空间参照系中才能实现相互操作。

  - 代码示例：

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

  - 运行结果：

```
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)
```

- 使用GeoJSON格式查询柏林米特（SIRD取值为10117）的面、其尺寸和此面中的点数。
  - 代码示例：

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

- 运行结果：

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

- 查询勃兰登堡门（SRID取值为4326）500米范围内的住宿数量。
  - 代码示例：

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

  - 运行结果：

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

- 根据查询的附近住宿地数据中获取勃兰登堡门的粗略位置。
  - 代码示例：

```
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
;
```

  - 运行结果：

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

- 查询勃兰登堡门周围所有住宿的详细信息，并按照价格进行降序排列。

- 代码示例：

```
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
;
```

- 运行结果：

```
                        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)
```

- 查询价格最高的住宿信息以及其邮政编码。
  - 代码示例：

```
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)
;
```

  - 运行结果：

```
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,...
```

- 查询索柏林中列出的住宿数的热点，根据邮政编码将热点住宿进行分组，并按照供应量进行排序操作。

- 代码示例：

```
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
;
```

- 运行结果：

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

2.用户授权

2.1. 基于PostgreSQL标准权限模型授权

本文为您介绍在Hologres中，如何基于PostgreSQL标准模型（专家权限模型）进行授权的最佳实践。帮助您简化授权操作并使用更细粒度的权限管理。

背景信息

Hologres兼容PostgreSQL生态，支持PostgreSQL的标准权限模型（简称专家权限模型）。同时，Hologres提供了一套简单权限模型的授权模式，详情请参见简单权限模型概述。

简单权限模型的权限划分粒度比较粗，不太适用于细粒度的权限管理场景。PostgreSQL的标准授权对于权限的划分非常细致，如果您需要使用更细粒度的权限管理，请参照本文基于PostgreSQL授权的最佳实践进行操作。

PostgreSQL权限模型简介

标准PostgreSQL授权拥有详细的权限管理体系，详情请参见Postgresql授权。

PostgreSQL授权的限制如下：

- PostgreSQL授权*仅支持对现有对象授权，对未来的对象不生效*。示例如下。
  i. User1执行了 `GRANT SELECT ON ALL TABLES IN SCHEMA public TO User2;` 语句对User2授予了**public** Schema中所有表的SELECT权限。
  ii. User1在**public** Schema中创建了一张新表*table_new*。
  iii. User2执行 `SELECT * FROM table_new` 语句时显示报错 `Permission denied` 。

  由于User1对User2授予SELECT权限时，仅包含授权时刻**public** Schema中的所有表，而不包含未来在**public** Schema中创建的表，因此会产生上述报错。
- 您可以使用 `ALTER DEFAULT PRIVILEGES` 语句，对未来对象设置默认权限，详情请参见ALTER DEFAULT PRIVILEGES。该权限仅对未来对象生效。示例语句如下。

  ```
  ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO PUBLIC; --当前授权的人对在public Schema中新创建的表具有读权限。
  ```

  您也可以使用 `ALTER DEFAULTPRIVILEGES FOR ROLE xxx` 语句，为其他角色创建默认权限。当前用户和*xxx*满足以下关系时才能成功设置默认权限：
  - 当前用户是*xxx*组的成员。
  - 当前用户是超级用户（Superuser）时，*xxx*可以为用户或组。

  您可以使用psql命令 `\ddp` 查看（系统表为 `pg_catalog.pg_default_acl` ） `ALTER DEFAULT PRIVILEGES` 是否设置成功。

  `ALTER DEFAULT PRIVILEGES` 类似于一个触发器。创建新表时，Hologres会使用当前用户和Schema去检查 `pg_catalog.pg_default_acl` 系统表中是否有匹配项。如果存在匹配项，则系统自动添加匹配项规则。

> 说明
> - 您只能使用当前用户与匹配项规则进行匹配，不能使用当前用户的用户组去匹配。
> - `alter default privileges` 规则匹配只能在创建表时执行，在创建表之后修改表Owner（`alter table tablename owner to`）不会触发 `alter default privileges`。
>
> 例如，用户User1属于Group1，如果要给Group1匹配规则，授予未来表全权限。情况如下：
> - 如果当前用户是User1，则创建表时匹配不到规则。
> - 如果创建表之前先执行 `set session role group1;`，改变当前用户为Group1，则创建表时就可以匹配到规则，系统自动为新创建的表授权。

- 只有表Owner才可以删除表。

  在标准的PostgreSQL授权语句中，您需要根据Ownership来判断是否可以删除表。可以删除表的角色如下：
  - 表的Owner（即建表者）。
  - 表所在Schema的Owner。
  - Superuser
- 在PostgreSQL中，系统会默认创建表的用户为表的Owner，拥有该表的所有权限，包括删除表的权限。

  修改表Owner，示例如下语句。

```
alter table <tablename> owner to user2; //将表Owner由User1修改为User2。
alter table <tablename> owner to GROUP1;//将表Owner修改为Group1。
```

  修改表Owner的操作限制如下：
  - User1为表的Owner。
  - User1必须是Group1的直接或者间接成员。

    例如，User1是Group1的成员，或User1是Group1中某个组的成员。
  - Group1必须在表所在的Schema中有创建表的权限。
  - Superuser可以修改任意表的表Owner。

PostgreSQL权限模型规划

标准的PostgreSQL（专家权限模型）权限粒度划分比较细致，在使用之前需要对现有实例对象做如下权限规划：

- 总共有多少个权限组。
- 每个组的作用是什么。
- 每个组包含哪些用户。
- 哪些角色在什么时候可以删除表。
- 每个组在哪些Schema中工作。

建议您执行如下操作，规划实例对象：

- 确定权限组以及组的作用。

  权限组分为以下类型：
  - XX_DEV_GROUP：表的Owner，拥有表的所有操作权限。

- XX_WRITE_GROUP：表的写入权限，可以写入数据至相应表。
- XX_VIEW_GROUP：查看表数据权限，可以查看相应表的数据。

*XX*表示一个项目。例如PROJ1项目的权限组包括PROJ1_DEV_GROUP、 PROJ1_WRITE_GROUP及PROJ1_VIEW_GROUP。

> **说明** 组名的命名规范仅为建议参考，不做强制要求。

- 确定权限组所在的Schema。

  推荐每个项目的权限组使用一个Schema。

  一个DEV_GROUP可以拥有多张表，但每张表只能属于一个DEV_GROUP。例如，表TABLE1属于PROJ1_DEV_GROUP，则该表就不属于PROJ2_DEV_GROUP。

  一个用户可以属于多个DEV_GROUP。例如，User1可以既是PROJ1_DEV_GROUP的成员，也可以是PROJ2_DEV_GROUP的成员。

Hologres专家权限模型最佳实践一

本次实践以表对象为例，您也可以选用其他对象进行实验。

表的Owner是对应的XXX_DEV_GROUP权限组，因此，DEV_GROUP组中的任意成员都可以管理或删除该表。

例如，用户被添加到PROJ1_DEV_GROUP用户组之后，就拥有PROJ1项目中表的管理或删除权限。具体操作步骤如下：

1. 创建用户组。

   您可以根据业务需求划分权限模型，由Superuser创建相应的用户组，以PROJ1项目为例，示例语句如下。

   ```
   create role PROJ1_DEV_GROUP; //表的Owner，拥有表的所有操作权限。
   create role PROJ1_WRITE_GROUP; //表的写入权限，可以写入数据至相应表。
   create role PROJ1_VIEW_GROUP; //查看表数据权限，可以查看相应表的数据。
   ```

2. 授权用户组Schema的权限。

   您需要授予创建完成的用户组Schema的权限，示例项目PROJ1可以在Schema1中工作，语句如下。

   ```
   授权PROJ1拥有SCHEMA1中的相关权限。
   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;
   ```

   > **说明**
   > - 一个项目可以对应多个Schema，一个Schema也可以对应多个项目。
   > - 默认public Schema中所有用户都有CREATE和USAGE权限。

3. 创建用户并管理用户组。

   授权用户组Schema权限后，Superuser需要创建用户并添加用户至对应的用户组，示例语句如下。

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

4. 创建表并授权。

   创建表等对象时，由表的创建者（必须为PROJ1_DEVE_GROUP的成员）或Superuser执行相应的授权语句（假设新创建的表为TABLE1）。 示例如下。

```
grant all on table SCHEMA1.TABLE1 to PROJ1_WRITE_GROUP; //授予PROJ1_WRITE_GROUP写入数据至TABLE1的权限。
grant select on table SCHEMA1.TABLE1 to PROJ1_VIEW_GROUP; //授予PROJ1_VIEW_GROUP TABLE1的SELECT权限。
alter table SCHEMA1.TABLE1 owner to PROJ1_DEV_GROUP; //修改TABLE1的Owner为PROJ1_DEV_GROUP。
```

Hologres专家权限模型最佳实践二

本次实践，使用 `ALTER DEFAULT PRIVILEGES` 语句简化对每张表的授权操作。

您需要提前确定创建的表默认属于哪个项目。具体操作步骤如下：

1. 创建用户组。

   您可以根据业务需求划分权限模型，由Superuser创建相应的用户组，以PROJ1项目为例，示例语句如下。

```
create role PROJ1_DEV_GROUP; //表的Owner，拥有表的所有操作权限。
create role PROJ1_WRITE_GROUP; //表的写入权限，可以写入数据至相应表。
create role PROJ1_VIEW_GROUP; //查看表数据权限，可以查看相应表的数据。
```

2. 授权用户组Schema的权限。

   您需要授予创建完成的用户组Schema的权限，示例项目PROJ1可以在Schema1中工作，语句如下。

```
授权PROJ1拥有SCHEMA1中的相关权限。
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;
```

> **说明**
>
> - 一个项目可以对应多个Schema，一个Schema也可以对应多个项目。
> - 默认public Schema中所有用户都有CREATE和USAGE权限。

3. 创建用户并设置默认授权。

   完成Schema授权后，需要Superuser创建用户并添加用户至相应的组中，同时设置该用户创建表时拥有的默认权限。

   USER1创建的表默认属于PROJ1_DEV_GROUP，并且USER1为合法的阿里云账号。示例设置授权语句如下。

```
create user "USER1";
alter default privileges for role "USER1" grant all on tables to PROJ1_DEV_GROUP; //设置USER1创建的表，PROJ1_DEV_GROUP默认都有读写权限。
alter default privileges for role "USER1" grant all on tables to PROJ1_WRITE_GROUP; //设置USER1创建的表，PROJ1_WRITE_GROUP默认都有读写权限。
alter default privileges for role "USER1" grant select on tables to PROJ1_VIEW_GROUP; //设置USER1创建的表，PROJ1_VIEW_GROUP默认都有读写权限。
grant PROJ1_DEV_GROUP to "USER1"; //添加USER1至PROJ1_DEV_GROUP。
```

4. 修改表的Owner。

   如果您希望DEV_GROUP中的其他用户也可以管理或删除创建的表，则可以修改表的Owner为对应项目的DEV_GROUP，例如PROJ1_DEV_GROUP。

   修改表Owner的语句必须由表的创建者或Superuser执行。例如，示例中表的创建者必须是PROJ1_DEV_GROUP的成员。假设新创建的表为TABLE1。示例修改表Owner的语句如下。

   ```
   alter table SCHEMA1.TABLE1 owner to PROJ1_DEV_GROUP; //修改TABLE1的Owner为PROJ1_DEV_GROUP。
   ```

   如下情况，您可以修改表的Owner:

   - 新创建的表，由Superuser定期修改表Owner。
   - 在需要管理或删除表之前修改表Owner。

   > **说明** 如果您可以确认表的管理或删除操作是由表的创建者或Superuser执行的，您也可以不执行上述命令。

5. 修改用户的默认项目。

   调整用户的默认项目，需要Superuser或用户本人执行 `alter default privileges` 命令，撤销已设置的缺省权限后，使用新的 `alter default privileges` 创建默认授权。

   将USER1的默认项目由PROJ1改为PROJ2，该操作不影响现有的表。示例语句如下。

   ```
   取消原有默认授权。
   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
   ;
   创建新的默认授权。
   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.场景方案

3.1. Hologres推荐的数仓分层

本文为您介绍在Hologres中数仓分层的最佳实践，方便快速构建业务，建设集高性能、敏捷化于一体的实时数仓。

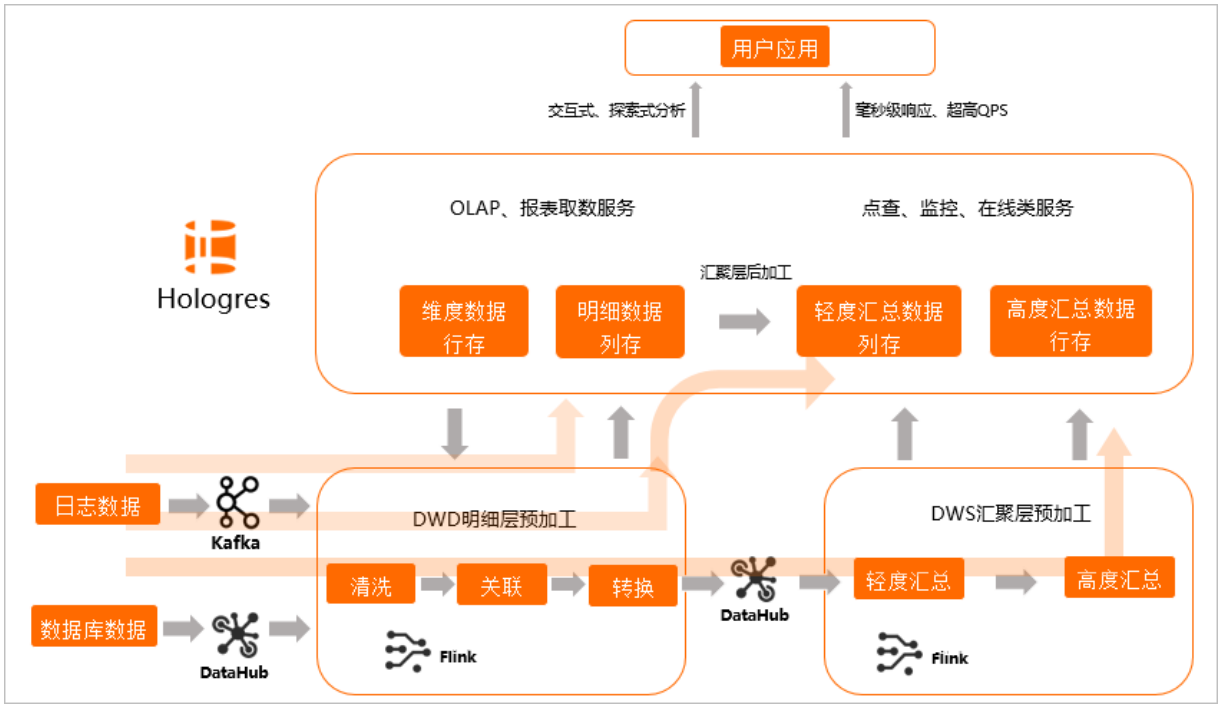
背景信息

Hologres与Flink、MaxCompute、DataWorks深度兼容，能够提供实时离线一体化联合解决方案。在该方案下有着非常丰富的应用场景，例如实时大屏、实时风控、精细化运营等。不同的应用场景对处理的数据量、数据复杂度、数据来源、数据实时性等会有不一样的要求。传统数仓的开发按照经典的方法论，采用 `ODS（Operational Data Store） > DWD（Data Warehouse Detail） > DWS（Data WareHouse Summary） > ADS（Application Data Service）` 逐层开发的方法，层与层之间采用事件驱动，或者微批次的方式调度。分层带来更好的语义层抽象和数据复用，但也增加了调度的依赖、降低了数据的时效性、减少了数据灵活分析的敏捷性。

实时数仓驱动了业务决策的实时化，在决策时通常需要丰富的上下文信息，因此传统高度依据业务定制ADS的开发方法受到了较大挑战，成千上万的ADS表维护困难，利用率低，更多的业务方希望通过DWS甚至DWD进行多角度数据对比分析，这对查询引擎的计算效率、调度效率、IO效率都提出了更高的要求。

随着计算算子向量化重写、精细化索引、异步化执行、多级缓存等多种查询引擎优化技术的应用，Hologres的计算能力在每个版本都有较大改善。因此越来越多的用户采用了敏捷化的开发方式，在计算前置的阶段，只做数据质量清理、基本的大表关联拉宽，建模到DWD、DWS即可，减少建模层次。同时将灵活查询在交互式查询引擎中执行，通过秒级的交互式分析体验，支撑了数据分析民主化的重要趋势。

为了满足业务场景的不同需求，建议您通过如下图所示三种方式进行数据分层和处理，以实现更加敏捷的开发需求。



- 场景一（即席查询，写入即服务）：在Flink中进行DWD数据明细层预加工，加工完的数据直接写入Hologres，由Hologres提供OLAP查询和在线服务。
- 场景二（分钟级准实时，微批次加工）：在Flink中进行DWD数据明细层预加工，写入Hologres后，在

Hologres中进行汇聚层加工，再对接上层应用。

- 场景三（增量数据实时统计，事件驱动加工）：DWD明细层预加工和DWS汇聚层预加工全部由Flink完成，写入Hologres提供上层应用。

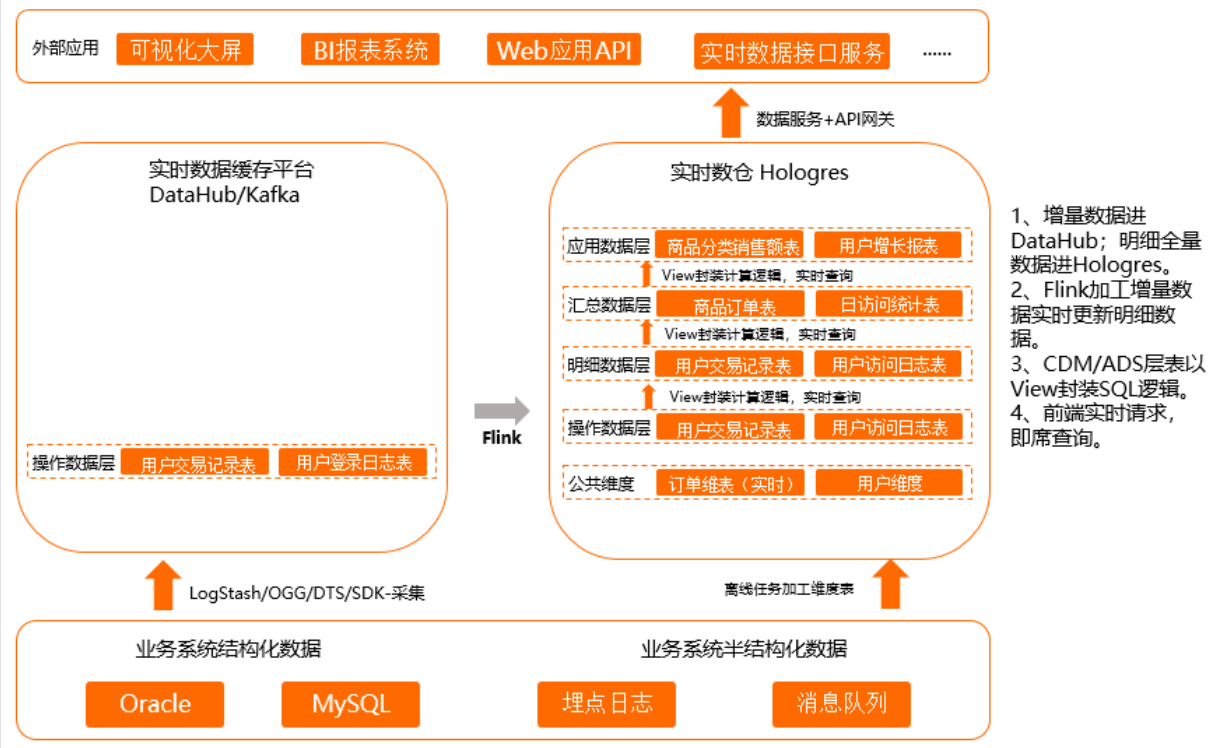
场景选择原则

当数据写入Hologres之后，Hologres里定义了三种实现实时数仓的方式：

- 实时要求非常高，要求写入即可查，更新即反馈，有即席查询需求，且资源较为充足，查询复杂度较低，适合实时数仓场景一：即席查询。
- 有实时需求，以分析为主，实时性满足分析时数据在业务场景具备实时含义，不追求数据产生到分析的秒级绝对值，但开发效率优先，推荐分钟级准实时方案，这个方案适合80%以上的实时数仓场景，平衡了时效性与开发效率，适合实时数仓场景二：分钟级准实时。
- 实时需求简单、数据更新少、只需要增量数据即可统计结果，以大屏和风控等在线服务场景为主，需要数据产生到分析尽量实时，可以接受一定开发效率的降低和计算成本的上升，适合实时数仓场景三：增量数据实时统计。

实时数仓场景一：即席查询

即席查询通俗来说就是不确定应用的具体查询模式，先把数据存下来，后续支撑尽量多灵活性的场景，如下图所示。



因此建议您应用如下策略：

- 将操作层（ODS层）的数据经过简单的清理、关联，然后存储到明细数据，暂不做过多的二次加工汇总，明细数据直接写入Hologres。
- Flink加工增量数据，实时更新明细数据至Hologres，MaxCompute加工后的离线表写入Hologres。
- 因为上层的分析SQL无法固化，在CDM/ADS层以视图（View）封装成SQL逻辑。
- 上层应用直接查询封装好的View，实现即席查询。

方案优势：

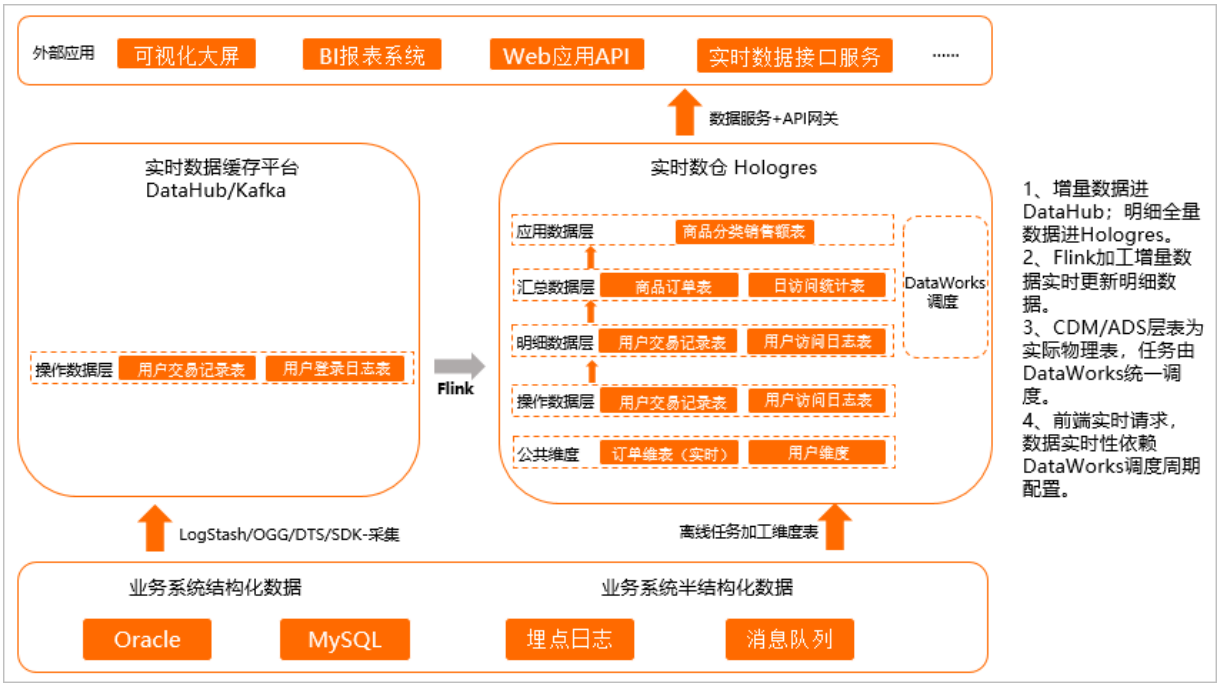
- 灵活性强，可随时根据业务逻辑调整View。
- 指标修正简单，上层都是View逻辑封装，只需要刷新一层数据，更新底表的数据即可，因为上层没有汇聚表，无需再次更新上层应用表。

方案缺点：当View的逻辑较为复杂，数据量较多时，查询性能较低。

适用场景：数据来源于数据库和埋点系统，适合对QPS要求不高，对灵活性要求比较高，且计算资源较为充足的场景。

实时数仓场景二：分钟级准实时

场景一的计算效率在某些场景上还存在不足，无法支撑更高的QPS，场景二是场景一的升级，把场景一中视图的部分物化成表，逻辑与场景一相同，但是最终落在表上的数据量变少，显著提升查询性能，可以获得更高的QPS，如下图所示。



建议您应用如下策略：

- 将操作层（ODS层）的数据经过简单的清理、关联，然后存储到明细数据，暂不做过多的二次加工汇总，明细数据直接写入Hologres。
- Flink加工增量数据实时更新明细数据至Hologres。
- CDM/ADS层为实际的物理表，通过DataWorks等调度工具调度周期性写入数据。
- 前端实时请求实际的物理表，数据的实时性依赖DataWorks调度周期配置，例如5分钟调度、10分钟调度等，实现分钟级准实时。

方案优势：

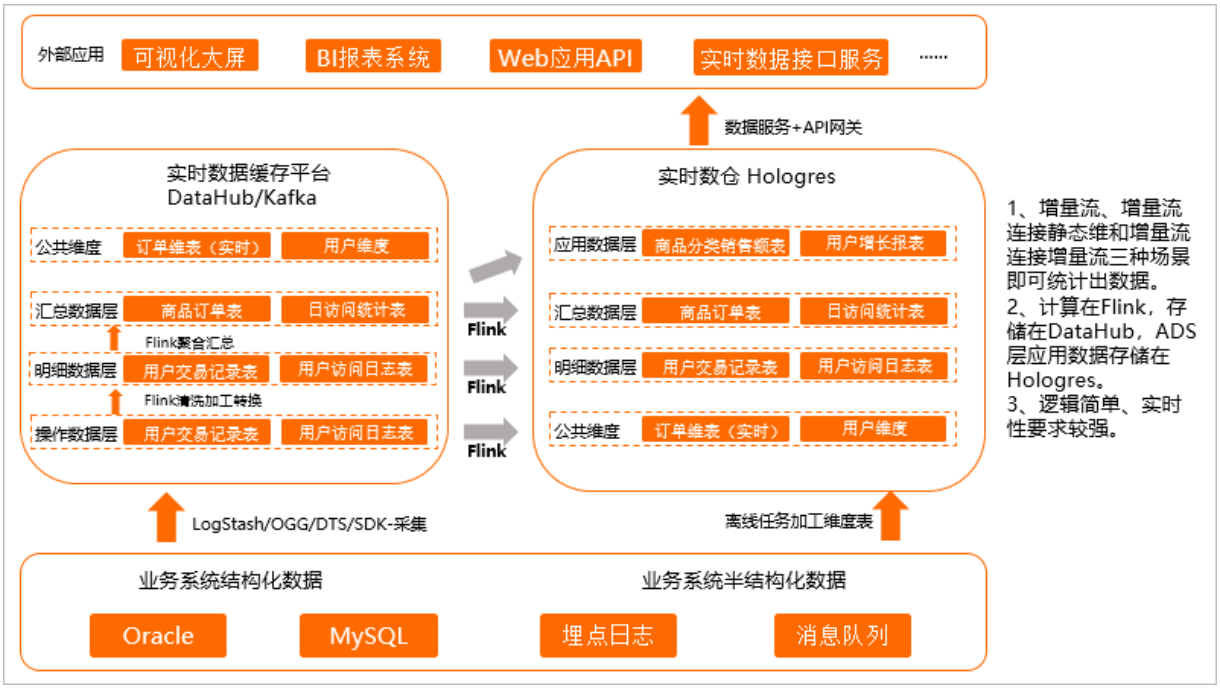
- 查询性能强，上层应用只查最后汇总的数据，相比View，查询的数据量更好，性能会更强。
- 数据重刷快，当某一个环节或者数据有错误时，重新运行DataWorks调度任务即可。因为所有的逻辑都是固化好的，无需复杂的订正链路操作。
- 业务逻辑调整快，当需要新增或者调整各层业务，可以基于SQL所见即所得开发对应的业务场景，业务上线周期缩短。

方案缺点：时效性低于方案一，因为引入了更多的加工和调度。

适用场景：数据来源于数据库和埋点系统，对QPS和实时性均有要求，适合80%实时数仓场景使用，能满足大部分业务场景需求。

实时数仓场景三：增量数据实时统计

增量计算的场景是因为一些场景对数据延迟非常敏感，数据产生的时候必须完成加工，此时通过增量计算的方式，提前用Flink将明细层、汇总层等层数据进行汇聚，汇聚之后把结果集存下来再对外提供服务，如下图所示。



在增量计算中，建议您应用如下策略：

- 增量计算的数据由Flink进行清洗加工转换和聚合汇总，ADS层应用数据存储在Hologres中。
- Flink加工的结果集采取双写的方式，一方面继续投递给下一层消息流Topic，一方面Sink到同层的Hologres中，方便后续历史数据的状态检查与刷新。
- 在Flink内通过增量流、增量流连接静态维表、增量流连接增量流这三种场景统计出数据，写入Hologres。
- Hologres通过表的形式直接对接上层应用，实现应用实时查询。

方案优势：

- 实时性强，能满足业务对实时性敏感的场景。
- 指标修正简单，与传统增量计算方式不一样的是，该方案将中间的状态也持久存储在Hologres中，提升了后续分析的灵活性，当中间数据质量有问题时，直接对表修正，重刷数据即可。

方案缺点：大部分实时增量计算都依赖Flink，对使用者Flink的技能和熟练度要求会更高一些；不适合数据频繁更新，无法累加计算的场景，不适合多流Join等计算复杂资源开销大场景。

适用场景：实时需求简单，数据量不大，以埋点数据统计为主的数据，只需要增量数据即可统计结果的场景，实时性最强。

3.2. 实时报表分析

3.2.1. 快速搭建实时数仓分析大屏

本文为您介绍如何使用交互式分析Hologres对接实时计算，快速搭建实时数仓分析大屏的最佳实践。

前提条件

- 开通Hologres并连接开发工具，详情请参见DataWorks快速入门或连接HoloWeb。
- 开通实时计算，详情请参见实时计算购买流程。

> **说明** 请确保实时计算和Hologres的地域相同。

- 开通DataV，详情请参见开通DataV服务。

背景信息

Hologres是阿里云的实时交互式分析产品，通过内置的实时数据API（详情请参见实时数据API）直接对接实时计算，实现高并发实时写入或查询实时数据，速度达到秒级。

Hologres兼容PostgreSQL，将查询到的数据直接对接BI分析工具，使用可视化方式快速分析和展现数据。

本文以搭建某电商店铺实时运营指标大屏为例，展示店铺的总流量、每个商店的访问量、区域销售量和热销商品数据。

使用Hologres搭建实时运营指标大屏的完整链路图如下所示。

- 采集源数据并实时写入实时计算，通过实时计算清洗并聚合数据。
- 实时写入处理后的数据至Hologres，使用Hologres进行交互式查询。
- Hologres对接数据大屏DataV，展示实时运营指标大屏。

操作步骤

1. 获取源数据。

   通过流式数据服务DataHub或者其他业务日志获取源数据。

   为了方便流程操作，本试验从实时计算直接产生数据，详情请参见第3步。

2. Hologres创建数据接收表。

   使用Holoweb创建*字段*和*数据类型*与源表相同的表，用于接收实时写入的数据。SQL语句示例如下。

```
BEGIN;
CREATE TABLE public.order_details (
 "user_id" int8,
 "user_name" text,
 "item_id" int8,
 "item_name" text,
 "price" numeric(38,2),
 "province" text,
 "city" text,
 "ip" text,
 "longitude" text,
 "latitude" text,
 "sale_timestamp" timestamptz NOT NULL
);
CALL SET_TABLE_PROPERTY('public.order_details','orientation', 'column');
CALL SET_TABLE_PROPERTY('public.order_details','clustering_key', 'sale_timestamp:asc');
CALL SET_TABLE_PROPERTY('public.order_details','segment_key', 'sale_timestamp');
CALL SET_TABLE_PROPERTY('public.order_details','bitmap_columns', 'user_name,item_name,p
rovince,city,ip,longitude,latitude');
CALL SET_TABLE_PROPERTY('public.order_details','dictionary_encoding_columns','user_name
:auto,item_name:auto,province:auto,city:auto,ip:auto,longitude:auto,latitude:auto');
CALL SET_TABLE_PROPERTY('public.order_details','time_to_live_in_seconds', '3153600000')
;
CALL SET_TABLE_PROPERTY('public.order_details','distribution_key', 'user_id');
CALL SET_TABLE_PROPERTY('public.order_details','storage_format', 'orc');
COMMIT;
```

3. 实时计算清洗数据。

   进入实时计算开发平台，使用实时计算清洗并聚合源数据，通过实时数据API将数据实时写入Hologres。SQL语句示例如下。

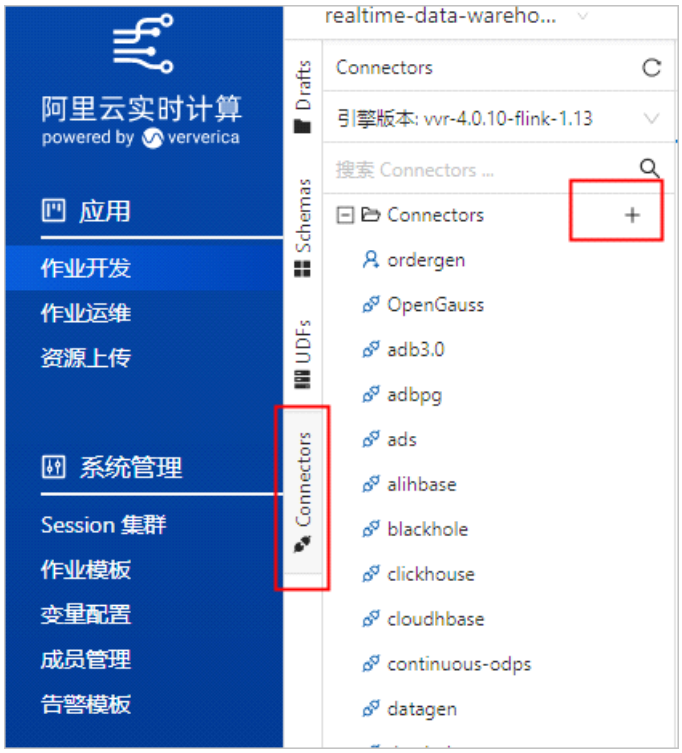
```
CREATE TEMPORARY table source_table (
   user_id BIGINT,
   user_name VARCHAR,
   item_id BIGINT,
      item_name VARCHAR,
    price numeric (38, 2),
   province VARCHAR,
   city VARCHAR,
      longitude VARCHAR,
   latitude VARCHAR,
    ip VARCHAR,
   sale_timestamp TIMESTAMP
  )
with ('connector' = 'ordergen');
CREATE TEMPORARY table hologres_sink (
   user_id BIGINT,
   user_name VARCHAR,
   item_id BIGINT,
      item_name VARCHAR,
    price numeric (38, 2),
   province VARCHAR,
   city VARCHAR,
      longitude VARCHAR,
   latitude VARCHAR,
    ip VARCHAR,
   sale_timestamp TIMESTAMP
  )
with (
   'connector' = 'hologres',
   'dbname' = 'Hologres的数据库名',
   'tablename' = 'Hologres接收数据的表名',
   'username' = '当前云账号的AccessKey ID',
   'password' = '当前云账号的AccessKey Secret',
   'endpoint' = 'Hologres的VPC网络地址：端口'
  );
insert into hologres_sink
select user_id,
  user_name,
  item_id,
  item_name,
  price,
  province,
  city,
  longitude,
  latitude,
  ip,
  sale_timestamp
from
 source_table;
```

4. 发布实时作业。

   提交并发布填写完成的作业至生产环境。操作步骤如下：

i. 引用资源包。

选择实时计算开发平台左侧菜单栏的**Connectors**，单击上方**新建资源**，配置**上传资源**的参数，上传JAR资源包。



ii. 发布作业。

成功引用资源包后，保存作业。单击**上线**，根据业务配置资源参数，提交作业至生产环境。

iii. 启动作业。

作业发布后，前往运维界面，手动启动作业。

5. 实时查询数据。

根据业务情况，在Hologres中从不同维度实时查询实时写入的数据，示例如下。

```
select sum(price) as "GMV" from order_details ;
select count(distinct user_id) as "UV" from order_details ;
select city as "城市", count(distinct user_id) as "购买用户数" from order_details group by "城市" order by "购买用户数" desc limit 100;
select item_name as "商品", sum(price) as "销售额" from order_details group by "商品" order by "销售额" desc limit 100;
select to_char(sale_timestamp, 'MM-DD') as "日期", sum(price) as "GMV" fromorder_details group by "日期" order by "GMV" desc limit 100;
```

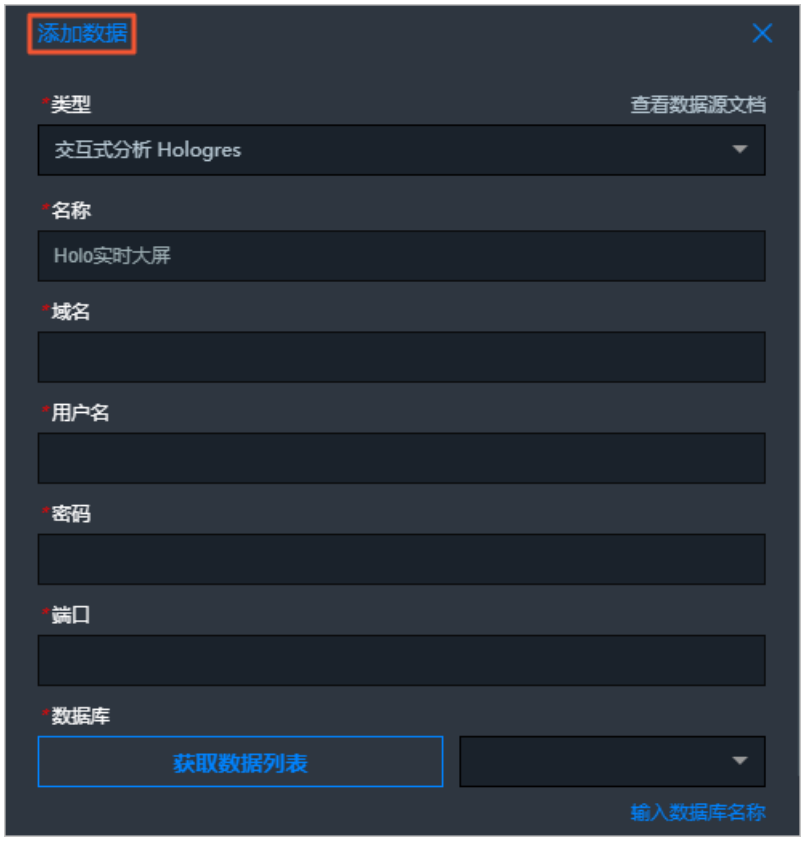
6. 展示DataV实时大屏。

   Hologres中查询出的数据直接对接DataV，用于制作实时大屏。操作步骤如下：

   i. 添加数据源。

   进入DataV控制台首页，单击**我的数据 > 添加数据**，配置**添加数据**的参数。

   类型选择交互式分析Hologres。



ii. 创建实时大屏。

根据大屏需要显示的内容，选择相应组件并配置数据源信息。详情请参见概述。

本试验中选择了基本柱状、轮播、基础平面地图和数字翻牌器。以配置数字翻牌器示例。

a. 配置数据源信息。如下图所示。

b. 配置数字翻牌器的边框、字体和颜色。

iii. 展示实时大屏。

配置大屏的插件参数及数据源信息后，可根据实际业务添加装饰元素，美化插件。如下图所示。

- 左侧实时显示商品的访问量以及城市的销售额。
- 中间地图实时显示每一笔交易订单的位置、总的销售额以及总访问次数。
- 右侧实时显示商品的销售额占比和商品的销售排行。

3.2.2. 实时分析海量MaxCompute数据

本文为您介绍交互式分析Hologres如何实时查询海量MaxCompute数据，并以可视化方式分析和展现查询结果的最佳实践。

前提条件

- 开通MaxCompute，详情请参见开通MaxCompute和DataWorks。

  > **说明** 请确保MaxCompute和Hologres的地域相同。

- 开通Hologres并连接至HoloWeb，详情请参见连接HoloWeb。
- 开通Quick BI，详情请参见连接数据源。

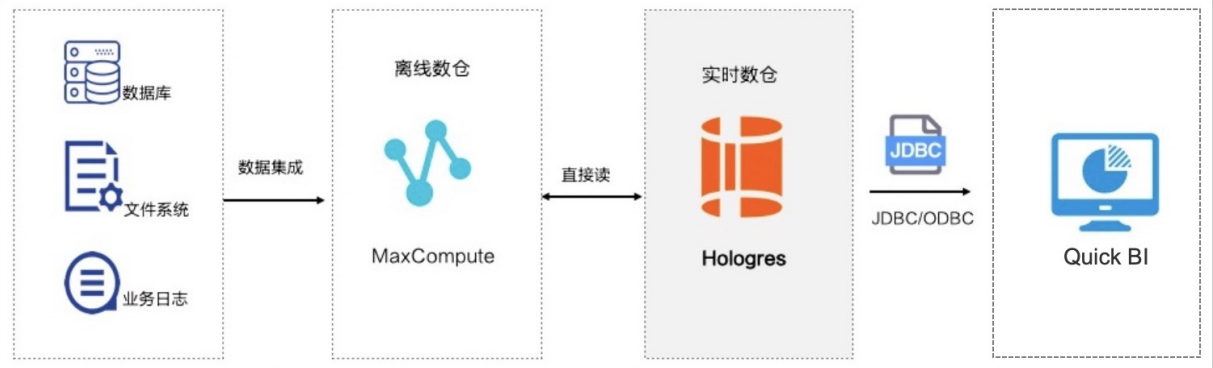
背景信息

Hologres是兼容PostgreSQL协议的实时交互式分析产品，在底层与MaxCompute无缝连接。

Hologres支持使用创建外部表的方式，实现查询，查看MaxCompute的数据。

本文以搭建访问某淘宝店铺的客户画像为例，展示客户所在城市、客户年龄、首选客户和出生于1980~1990年的首选客户所在城市人数的分布情况。

使用Hologres加速查询MaxCompute数据的完整链图如下所示。



1. 访问店铺的客户数据存储在MaxCompute表中。
2. 使用Hologres的创建外部表方式，实现查询。
3. Hologres对接数据大屏Quick BI，使用可视化方式展示客户画像。

操作步骤

1. 准备MaxCompute数据源。

   在MaxCompute中创建一张表并写入数据，详情请参见创建表。

   本次试验采用已有的阿里云MaxCompute中**public_data**项目的如下MaxCompute表。获取表的方法请参见公开数据集。

MaxCompute表名称	数据量
customer	1200万
customer_address	600万
customer_demographics	192万

2. Hologres创建外部表并查询表数据。

   通过使用HoloWeb创建外部表，实现查询。操作步骤如下：

i. 新建外部表。

登录HoloWeb，单击**元数据管理 > MaxCompute加速 > 创建外部表**，使用可视化的方式创建外部表。

输入MaxCompute表的名称，例如 `public_data.customer` 就可以索引出表的字段，您可以根据实际业务，选择需要同步的表字段，单击**提交**。

> **说明**
> - 目前暂不支持跨地域查询MaxCompute表的数据。
> - 创建外部查询MaxCompute数据是通过外部服务器来实现的，您可以直接调用Hologres底层已创建的名为**odps_server**的外部表服务器。其详细原理请参见Postgres FDW。

您也可以使用SQL语句批量创建外部表。

```
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');
```

ii. 预览外部表数据。

成功新建外部表后，在左侧**实例管理**目录，鼠标右击新建的外部表，单击**打开表**，在表详情页单击**数据预览**，预览MaxCompute表的数据。

> **说明** **数据预览**只展示部分数据。

iii. 查询外部表数据。

在表详情页单击**查询表**，在**临时Query查询**页面，输入如下示例SQL命令后单击**运行**，查询外部表的数据。

示例SQL语句如下。

```
# SQL1：查询首选客户分布情况，按人数降序排列。
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;
# SQL2：查询客户年龄人数大于1000的分布情况，按人数降序排列。
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;
# SQL3：查询客户所在城市的人数大于10的分布情况，按人数降序排序。
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;
# SQL4：查询首选客户出生于1980~1990年且所在城市的人数大于10的分布情况，按人数降序排列。
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. 使用Quick BI分析数据。

   Hologres查询出的MaxCompute数据直接对接Quick BI，使用可视化方式分析和展示数据。操作步骤如下：

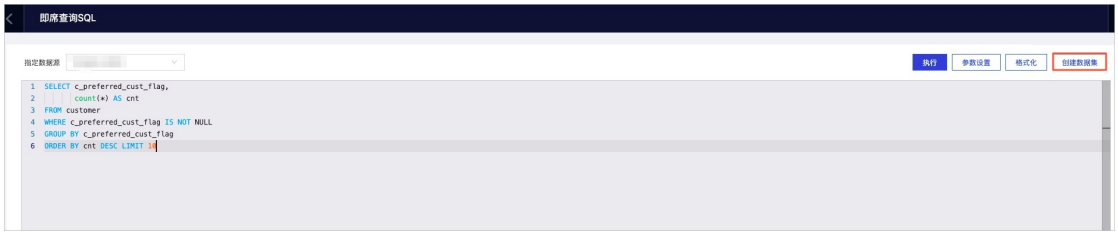
   i. 添加数据源。

   进入Quick BI控制台首页，选择PostgreSQL数据源，并填写配置信息，详情请参见Quick BI。

   ii. 创建数据集。

   成功连接Quick BI后，您可以将需要展示的数据创建为数据集并制作报表。

   本次试验使用**即席查询SQL**的方式来创建数据集，如下图所示。



iii. 使用可视化方式展示客户画像。

根据业务需求展示报表。示例所配置的报表展示如下。

3.3. 用户行为分析

3.3.1. 用户行为分析（UV）概述

在用户行为分析和圈人场景中，经常需要从亿级甚至几十亿级用户中快速筛选出符合特定标签的指标结果，本文为您介绍Hologres中如何进行用户行为分析。

行业背景与痛点

UV（Unique Visitor）是行为分析中最常见的指标，代表访问网页的自然人，可以引申为某段时间内某个指标精确去重后的量。例如大促时，电商商家需要实时计算店铺的实时UV，并根据UV情况及时调整运营策略，从而达成销售目标。

在计算用户UV时，由于业务需求不同，计算的维度和数据量也不同，通常来讲会有以下诉求。

- 用户数据量大，每天几亿条，维度多（10+以上），需要支持各维度间任意组合查询。
- 查询时间需要更灵活，不仅局限于天、周、月、年等，还需要支持更细粒度实时更新查询。
- 需要对用户数量精确去重。

面对上述高复杂度UV计算场景，业界常见的手段包括使用Apche Kylin等预计算系统或者Flink+MySQL的固定维度组合方案，但这些方案会遇见以下痛点。

- 需求维度过多时，会带来存储爆炸，预计算时间长。
- 精确去重需要消耗大量资源，容易出现OOM（Out Of Memory）。
- 实时更新难，无法支持更加灵活开放的时间周期处理。

解决方案与优势

Hologres是基于分析服务一体化理念（Hybrid Serving & Analytical Processing，HSAP）设计的实时数仓产品，采用分布式架构，支持数据实时写入，高并发、低延时的分析处理PB级数据，兼容PostgreSQL协议，使用最熟悉的工具就能进行开发。

在Hologres中，通过RoaringBitmap和Serial，结合Hologres自身的高性能，就能实现亿级UV精确计算。

- RoaringBitmap

RoaringBitmap是一种压缩位图索引，RoaringBitmap自身的数据压缩和去重特性十分适合对于大数据下UV计算。其主要原理如下：

- 对于32Bit数，RoaringBitmap会构造2^{16}个桶，对应32位数的高16位；32位数的低16位则映射到对应桶的一个Bit上。单个桶的容量由桶中已有的最大数值决定。
- Bitmap把32位数用1位表示，可以极大的压缩数据。
- Bitmap位运算为去重提供了手段。

RoaringBitmap使用详情请参见Roaring Bitmap函数。

- Serial

  自增序列Serial，常用于维表关联中的用户映射（user_mappping）。因为常见的业务系统或者埋点中的用户ID很多是字符串类型或Long类型，因此需要使用uid_mapping类型构建一张映射表。RoaringBitmap类型要求用户ID必须是32位INT类型且越稠密越好（即用户ID最好连续），映射表利用Hologres的SERIAL类型（自增的32位INT）来实现用户映射的自动管理和稳定映射。Serial使用详情请参见自增序列Serial（Beta）。

离线UV计算场景

离线UV计算应用T+1思想：将前一天的所有数据根据最大的查询维度聚合出uid结果放入RoaringBitmap中，RoaringBitmap和查询维度存放在聚合结果表（每天百万条）。之后查询时，利用Hologres强大的列存计算直接按照查询维度去查询聚合结果表，对其中关键的RoaringBitmap字段做或运算进行去重后并统计基数，即可得出对应用户数UV，计数条数即可计算得出PV，达到亚秒级查询。

只需进行一次最细粒度的预聚合计算，也只生成一份最细粒度的预聚合结果表。得益于Hologres的计算能力，该方案下预计算所需的次数和空间都达到较低的开销。详情使用请参见离线UV计算。

实时UV计算场景

Hologres与Flink有着强大的融合优化，支持Flink数据高通量实时写入，写入即可见，支持Flink SQL维表关联，以及作为CDC Source事件驱动开发。

实时UV计算主要是Hologres与Flink结合完成：Flink关联Hologres用户维表，并基于RoaringBitmap，实时对用户标签去重。这样的方式，可以较细粒度的实时得到用户UV、PV数据，同时便于根据需求调整最小统计窗口（如最近5分钟的UV），实现类似实时监控的效果，更好的在大屏等BI中展示。相较于以天、周、月等为单位的去重，更适合在活动日期进行更细粒度的统计，并且通过简单的聚合，也可以得到较大时间单位的统计结果。

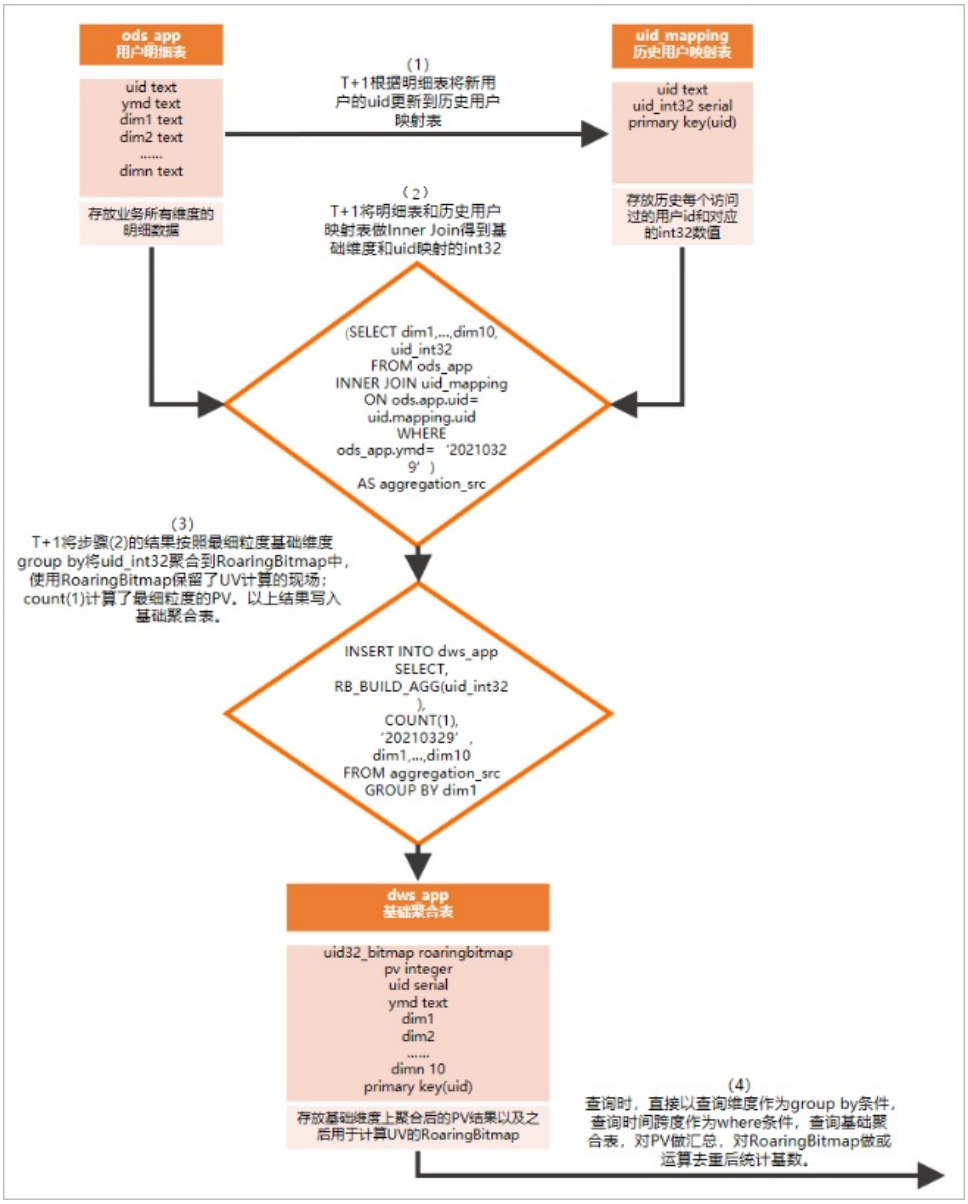
该方案数据链路简单，可以任意维度灵活计算，只需要一份Bitmap存储，也没有存储爆炸问题，还能保证实时更新，从而实现更实时、开发更灵活、功能更完善的多维分析数仓。使用详情请参见实时UV精确去重（Flink+Hologres）。

3.3.2. 离线UV计算

本文为您介绍在Hologres中如何进行离线UV计算。

背景信息

离线UV计算方案流程如下图所示。



1. 创建一张用户明细表，用于存放业务所有维度的明细数据。
2. 创建一张历史用户映射表，存放历史每个访问过的用户ID（uid）和对应的int32数值，其中int32主要是Serial类型，便于与明细表做用户uid映射。

   > **说明** RoaringBitmap类型要求用户ID必须是32位int类型且越稠密越好（用户ID最好连续），而常见的业务系统或者埋点中的用户ID很多是字符串类型，因此使用uid_mapping类型构建一张用户映射表。用户映射表利用Hologres的Serial类型（自增的32位int）来实现用户映射的自动管理和稳定映射。

3. 把T+1（上一天）的明细表和历史用户映射表做Inner Join得到基础维度表。
4. 根据业务逻辑，将基础维度表按照最细粒度基础维度group by，把上一天的所有数据根据最大的查询维度聚合出的uid结果放入RoaringBitmap中，并存放在聚合结果表（每天百万条）。
5. 按照查询维度查询聚合结果表，对其中关键的RoaringBitmap字段做 `or` 运算进行去重后并统计基数，即可得出对应用户数UV，计数条数即可计算得出PV，达到亚秒级查询。

操作步骤

1. 创建相关基础表

   i. 创建RoaringBitmap extention

   使用RoaringBitmap前需要创建RoaringBitmap extention，语法如下，同时Hologres实例需要V0.10及以上版本。

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

   ii. 创建用户明细表

   创建名称为ods_app的用户明细表，存放用户每天大量的明细数据 （按天分区），其DDL如下。

```
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');
--distribution_key根据需求设置，根据该表的实时查询需求，从什么维度做分片能够取得较好效果即可
CALL set_table_property('public.ods_app', 'distribution_key', 'uid');
--用于做where过滤条件，包含完整年月日时间字段推荐设为clustering_key和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. 创建用户映射表

创建名称为uid_mapping的用户映射表，用于映射uid到32位INT类型，其DDL如下所示。

RoaringBitmap类型要求用户ID必须是32位int类型且越稠密越好（用户ID最好连续），而常见的业务系统或者埋点中的用户ID很多是字符串类型，因此使用uid_mapping类型构建一张映射表。映射表利用Hologres的Serial类型（自增的32位int）来实现用户映射的自动管理和稳定映射。

> **说明** 该表在本例每天批量写入场景，可为行存表也可为列存表，没有太大区别。如需要做实时数据（例如和Flink联用），需要是行存表，以提高Flink维表实时JOIN的QPS。

```
BEGIN;
 CREATE TABLE public.uid_mapping (
     uid text NOT NULL,
     uid_int32 serial,
            PRIMARY KEY (uid)
 );
 --将uid设为clustering_key和distribution_key便于快速查找其对应的int32值
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. 创建聚合结果表

创建名称为dws_app的聚合结果表，用于存放RoaringBitmap聚合后的结果，其DDL如下所示。

基础维度为之后进行查询计算pv和uv的最细维度，这里以country、 prov、 city为例构建基础维表。

```
begin;
create table dws_app(
    country text,
  prov text,
  city text,
  ymd text NOT NULL,  --日期字段
  uid32_bitmap roaringbitmap, -- UV计算
  pv integer, -- PV计算
  primary key(country, prov, city, ymd)--查询维度和时间作为主键，防止重复插入数据
);
CALL set_table_property('public.dws_app', 'orientation', 'column');
--clustering_key和event_time_column设为日期字段，便于过滤
CALL set_table_property('public.dws_app', 'clustering_key', 'ymd');
CALL set_table_property('public.dws_app', 'event_time_column', 'ymd');
--distribution_key设为group by字段
CALL set_table_property('public.dws_app', 'distribution_key', 'country,prov,city');
end;
```

2. 更新用户映射表和聚合结果表

i. 更新用户映射表

每天从上一天的uid中找出新客户（用户映射表uid_mapping中没有的uid）插入到用户映射表中，命令如下。

```
WITH
-- 其中ymd = '20210329'表示上一天的数据
    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. 更新聚合结果表

更新完用户映射表后，将数据做聚合运算后插入聚合结果表，主要步骤如下。

a. 通过明细表Inner Join用户映射表，得到上一天的聚合条件和对应的uid_int32。

b. 按照聚合条件做聚合运算后插入RoaringBitmap聚合结果表，作为上一天的聚合结果。

c. 每天只需进行一次聚合，存放一份数据，数据条数等于UV的量。明细表每天几亿的增量，在聚合结果表每天只需存放百万级数据。

插入数据至聚合结果表命令如下。

```
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. UV、PV查询

查询时，从dws_app聚合结果表中按照查询维度做聚合计算，查询Bitmap基数，得出Group By条件下的用户数，命令如下。

```
--运行下面RB_AGG运算查询，可先关闭三阶段聚合开关性能更佳（默认关闭）
set hg_experimental_enable_force_three_stage_agg=off
--可以查询基础维度任意组合，任意时间段的uv pv
SELECT  country
        ,prov
        ,city
        ,RB_CARDINALITY(RB_OR_AGG(uid32_bitmap)) AS uv
        ,sum(pv) AS pv
FROM    dws_app
WHERE   ymd = '20210329'
GROUP BY country
         ,prov
         ,city;
--查一个月
SELECT  country
        ,prov
        ,RB_CARDINALITY(RB_OR_AGG(uid32_bitmap)) AS uv
        ,sum(pv) AS pv
FROM    dws_app
WHERE   ymd >= '20210301' and ymd <= '20210331'
GROUP BY country
         ,prov;
该查询等价于
SELECT  country
        ,prov
        ,city
        ,COUNT(DISTINCT uid) AS uv
        ,COUNT(pv) AS pv
FROM    ods_app
WHERE   ymd = '20210329'
GROUP BY country
         ,prov
         ,city;
SELECT  country
        ,prov
        ,COUNT(DISTINCT uid) AS uv
        ,COUNT(pv) AS pv
FROM    ods_app
WHERE   ymd >= '20210301' and ymd <= '20210331'
GROUP BY country
         ,prov;
```

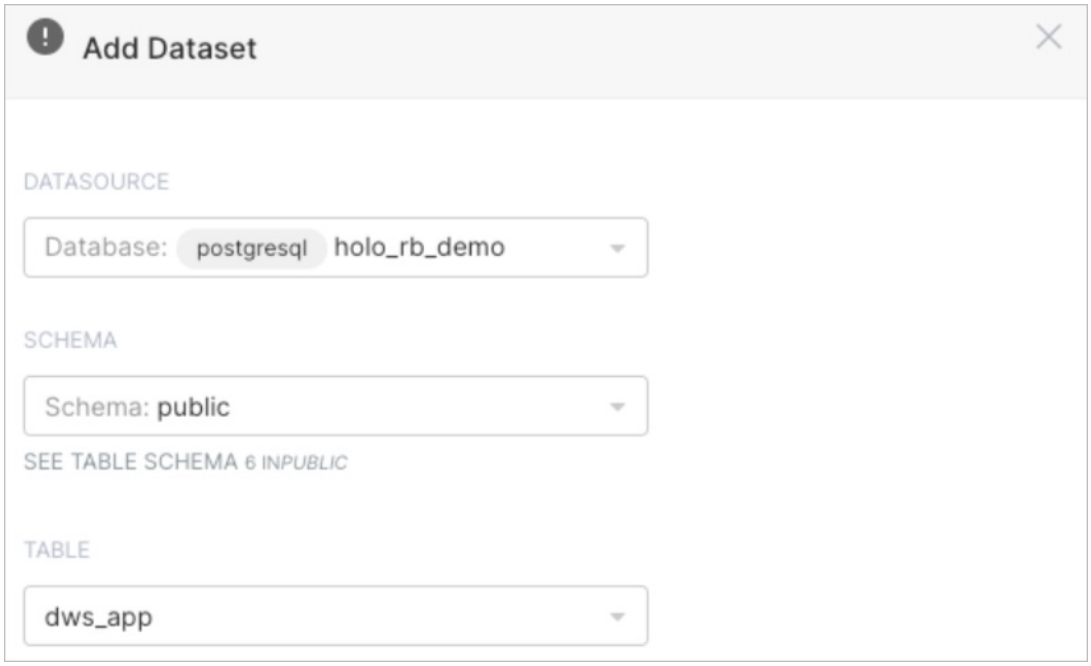
4. 可视化展示

   计算出UV、PV后，大多数情况需要使用BI工具以更直观的方式可视化展示，由于需要使用RB_CARDINALITY和RB_OR_AGG进行聚合计算，需要使用BI的自定义聚合函数的能力，常见的具备该能力的BI包括Apache Superset和Tableau。

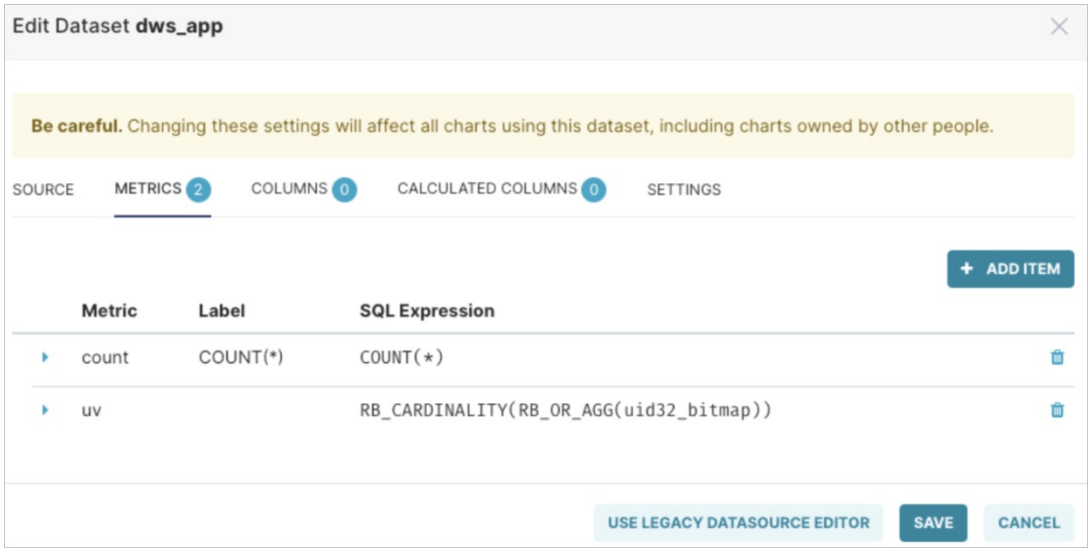
   - Apache Superset

     a. Apache Superset连接Hologres，详情请参见Apache Superset。

b. 设置dws_app表作为数据集。



c. 在数据集中创建一个名称为UV的单独Metrics，表达式如下。




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

完成后您就可以开始探索数据了。

d. （可选）创建Dashborad。

创建仪表板请参见Create Dashboard。

- Tableau

a. Tableau连接Hologres，详情请参见Tableau。

可以使用Tableau的直通函数直接实现自定义函数的能力，详细介绍请参见直通函数。

b. 创建一个计算字段，表达式如下。


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

完成后您就可以开始探索数据了。

c. （可选）创建Dashborad。

创建仪表板请参见Create a Dashboard。

3.3.3. 实时UV精确去重（Flink+Hologres）

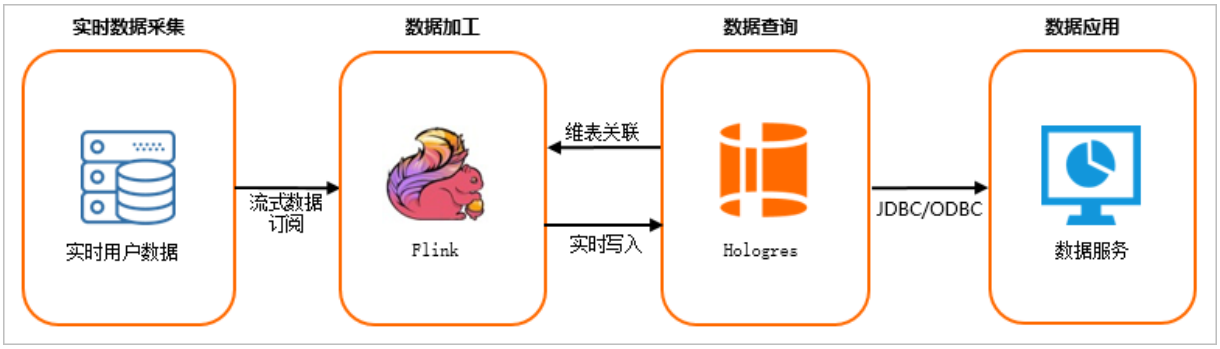
实时UV计算主要依赖Hologres与Flink结合完成，本文将为您介绍Hologres如何进行实时UV精确去重。

前提条件

- 开通Hologres并连接开发工具，示例使用HoloWeb，详情请参见连接HoloWeb。
- 准备并搭建好Flink集群环境，您可以使用阿里云Flink全托管或者开源Flink。

背景信息

Hologres与Flink有着强大的融合优化，支持Flink数据高通量实时写入，写入即可见，支持Flink SQL维表关联，以及作为CDC Source事件驱动开发。因此实时UV去重主要通过Flink和Hologres来实现，场景架构图如下图所示。

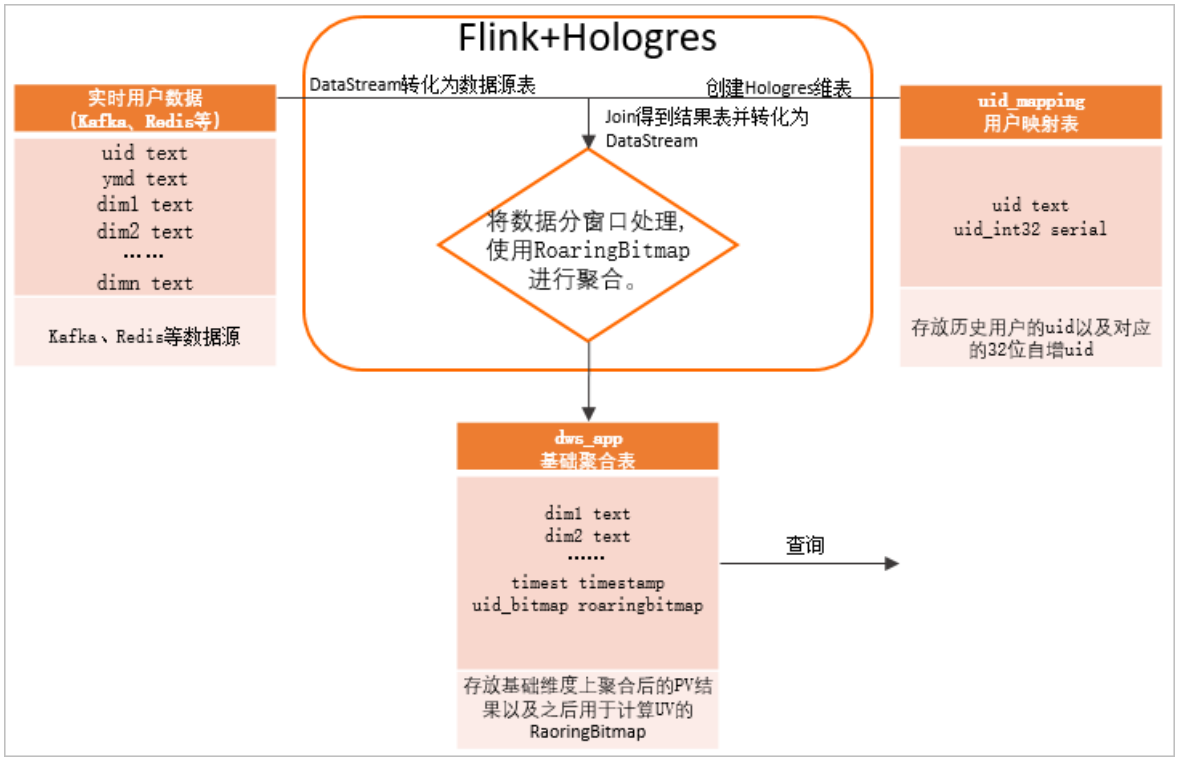


1. Flink实时订阅实时采集的数据，数据源可以来源于日志数据，如Kafka等。

2. Flink对数据做进一步加工处理，将流式数据转化为表与Hologres维表进行JOIN操作，实时写入Hologres。
3. Hologres对Flink实时写入的数据实时处理。
4. 最终查询的数据对接上层数据应用，如数据服务、Quick BI等。

实时UV计算方案流程

Flink与Hologres有着非常强的融合性，再结合Hologres天然支持的RoaringBitmap，完成实时UV计算，实时对用户标签去重，详细方案流程如下图所示。



1. Flink实时订阅用户数据，这些数据可以来源于Kafka、Redis等，并通过DataStream转化为数据源表。
2. 在Hologres中创建用户映射表，存放历史用户的uid以及对应的32位自增uid。

   > **说明** 常见的业务系统或者埋点中的用户ID很多是字符串类型或Long类型，因此需要使用uid_mapping类型构建一张映射表。RoaringBitmap类型要求用户ID必须是32位int类型且越稠密越好（即用户ID最好连续）。映射表利用Hologres的SERIAL类型（自增的32位int）来实现用户映射的自动管理和稳定映射。

3. 在Flink中，将Hologres中的用户映射表作为Flink维表，利用Hologres维表的insert If Not Exists特性结合自增字段实现高效的uid映射。维表与数据源表进行Join关联，并将Join得到的结果转化为流式数据DataStream。
4. Hologres中创建聚合结果表，Flink把维表关联的结果数据按照时间窗口进行处理，根据查询维度使用Roaring Bitmap函数。
5. 查询时，与离线方式相似，直接按照查询条件查询聚合结果表，并对其中关键的RoaringBitmap字段做 `or` 运算后并统计基数，即可得出对应用户数。

这样的方式，可以较细粒度的实时得到用户UV、PV数据，同时便于根据需求调整最小统计窗口（如最近5分钟的UV），实现类似实时监控的效果，更好的在大屏等BI展示。相较于以天、周、月等为单位的去重，更适合在活动日期进行更细粒度的统计，并且通过简单的聚合，也可以得到较大时间单位的统计结果。如果加工

聚合的粒度较细，但查询时缺少相应的过滤条件或聚合维度，则也会在查询时引起较多的二次聚合操作，对性能有不利影响。

该方案数据链路简单，可以任意维度灵活计算，只需要一份Bitmap存储，也没有存储爆炸问题，还能保证实时更新，从而实现更实时、开发更灵活、功能更完善的多维分析数仓。

操作步骤

1. 在Hologres中创建相关基础表

   i. 创建用户映射表

   在Hologres创建表uid_mapping为用户映射表，命令语句如下所示。用于映射uid到32位int类型。如果原始uid已经是int32类型，此步骤可忽略。

   - 常见的业务系统或者埋点中的用户ID很多是字符串类型或Long类型，因此需要使用uid_mapping类型构建一张映射表。RoaringBitmap类型要求用户ID必须是32位int类型且越稠密越好（即用户ID最好连续）。映射表利用Hologres的SERIAL类型（自增的32位int）来实现用户映射的自动管理和稳定映射。
   - 由于是实时数据，在Hologres中该表为行存表，以提高Flink维表实时JOIN的QPS。
   - 需要开启相应GUC使用优化的执行引擎对包含serial字段的表进行写入，详情请参见Fixed Plan加速SQL执行。

```
--开启GUC,支持含有Serial类型列的Fixed Plan写入
alter database <dbname> set hg_experimental_enable_fixed_dispatcher_autofill_series
=on;
alter database <dbname> set hg_experimental_enable_fixed_dispatcher_for_multi_value
s=on;
BEGIN;
CREATE TABLE public.uid_mapping (
uid text NOT NULL,
uid_int32 serial,
PRIMARY KEY (uid)
);
--将uid设为clustering_key和distribution_key便于快速查找其对应的int32值
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. 创建聚合结果表

创建表dws_app为聚合结果表，用于存放在基础维度上聚合后的结果。

使用Roaring Bitmap函数前需要创建RoaringBitmap extention，同时也需要Hologres实例为 V0.10及以上版本。

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

相比离线结果表，此结果表增加了时间戳字段，用于实现以Flink窗口周期为单位的统计，结果表DDL如下。

```
BEGIN;
CREATE TABLE dws_app(
  country text,
  prov text,
  city text,
  ymd text NOT NULL,  --日期字段
  timetz TIMESTAMPTZ,  --统计时间戳，可以实现以Flink窗口周期为单位的统计
  uid32_bitmap roaringbitmap, -- 使用roaringbitmap记录uv
 PRIMARY KEY (country, prov, city, ymd, timetz)--查询维度和时间作为主键，防止重复插入数据
);
CALL set_table_property('public.dws_app', 'orientation', 'column');
--日期字段设为clustering_key和event_time_column，便于过滤
CALL set_table_property('public.dws_app', 'clustering_key', 'ymd');
CALL set_table_property('public.dws_app', 'event_time_column', 'ymd');
--group by字段设为distribution_key
CALL set_table_property('public.dws_app', 'distribution_key', 'country,prov,city');
COMMIT;
```

2. Flink实时读取数据并更新聚合结果表

在Flink中的完整示例源码请参见alibabacloud-hologres-connectors examples，下面是在Flink中的具体操作步骤。

i. Flink流式读取数据源（DataStream）并转化为源表（Table）

在Flink中使用流式读取数据源，数据源可以是CSV文件，也可以来源于Kafka、Redis等，可以根据业务场景准备。在Flink中转化为源表的代码示例如下。

```
-- 此处使用csv文件作为数据源，也可以是kafka/redis等
DataStreamSource odsStream = env.createInput(csvInput, typeInfo);
-- 与维表join需要添加proctime字段
Table odsTable =
    tableEnv.fromDataStream(
    odsStream,
    $("uid"),
    $("country"),
    $("prov"),
    $("city"),
    $("ymd"),
    $("proctime").proctime());
-- 注册到catalog环境
tableEnv.createTemporaryView("odsTable", odsTable);
```

ii. 将源表与Hologres维表（uid_mapping）进行关联

在Flink中创建Hologres维表，需要使用 `insertIfNotExists` 参数，即查询不到数据时自行插入，uid_int32字段便可以利用Hologres的Serial类型自增创建。将Flink源表与Hologres维表进行关联（JOIN），代码示例如下。

```
-- 创建Hologres维表，其中insertIfNotExists表示查询不到则自行插入
String createUidMappingTable =
    String.format(
    "create table uid_mapping_dim("
    + "  uid string,"
    + "  uid_int32 INT"
    + ") with ("
    + "  'connector'='hologres',"
    + "  'dbname' = '%s'," //Hologres DB名
    + "  'tablename' = '%s',"//Hologres 表名
    + "  'username' = '%s'," //当前账号access id
    + "  'password' = '%s'," //当前账号access key
    + "  'endpoint' = '%s'," //Hologres endpoint
    + "  'insertifnotexists'='true'"
    + ")",
    database, dimTableName, username, password, endpoint);
tableEnv.executeSql(createUidMappingTable);
-- 源表与维表join
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. 将关联结果转化为DataStream

通过Flink时间窗口处理，结合RoaringBitmap进行对指标进行去重处理，代码示例如下所示。

```
DataStream<Tuple6<String, String, String, String, Timestamp, byte[]>> processedSour
ce =
    source
    -- 筛选需要统计的维度（country, prov, city, ymd）
    .keyBy(0, 1, 2, 3)
    -- 滚动时间窗口；此处由于使用读取csv模拟输入流，采用ProcessingTime，实际使用中可使用Eve
ntTime
    .window(TumblingProcessingTimeWindows.of(Time.minutes(5)))
    -- 触发器，可以在窗口未结束时获取聚合结果
    .trigger(ContinuousProcessingTimeTrigger.of(Time.minutes(1)))
    .aggregate(
    -- 聚合函数，根据key By筛选的维度，进行聚合
    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) {
                -- 将32位的uid添加到RoaringBitmap进行去重
                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);
            }
        },
    -- 窗口函数，输出聚合结果
    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();
                // 优化RoaringBitmap
                result.runOptimize();
                // 将RoaringBitmap转化为字节数组以存入Holo中
                byte[] byteArray = new byte[result.serializedSizeInBytes()];
                result.serialize(ByteBuffer.wrap(byteArray));
                // 其中 Tuple6.f4(Timestamp) 字段表示以窗口长度为周期进行统计，以秒为单位
                out.collect(
                    new Tuple6<>(
                        keys.getField(0),
                        keys.getField(1),
                        keys.getField(2),
                        keys.getField(3),
                        new Timestamp(
                            timeWindow.getEnd() / 1000 * 1000),
                        byteArray));
        }
    });
```

iv. 写入Hologres聚合结果表

经过Flink去重处理的数据写入至Hologres结果表dws_app，但需要注意的是Hologres中RoaringBitmap类型在Flink中对应Byte数组类型，Flink中代码如下。

```
 -- 计算结果转换为表
Table resTable =
    tableEnv.fromDataStream(
        processedSource,
        $("country"),
        $("prov"),
        $("city"),
        $("ymd"),
        $("timest"),
        $("uid32_bitmap"));
-- 创建Hologres结果表，其中Hologres的RoaringBitmap类型通过Byte数组存入
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);
-- 写入计算结果到dws_app表
tableEnv.executeSql("insert into sink select * from " + resTable);
```

3. 数据查询

在Hologres中对聚合结果表（dws_app）进行UV计算。按照查询维度做聚合计算，查询Bitmap基数，得出Group By条件下的用户数。

- 示例一：查询某天内各个城市的uv

```
-- 运行下面RB_AGG运算查询，可执行参数先关闭三阶段聚合开关（默认关闭），性能更好，此步骤可选
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
;
```

- 示例二：查询某段时间内各个省份的UV、PV

```
-- 运行下面RB_AGG运算查询，可执行参数先关闭三阶段聚合开关（默认关闭），性能更好，此步骤可选
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
;
```

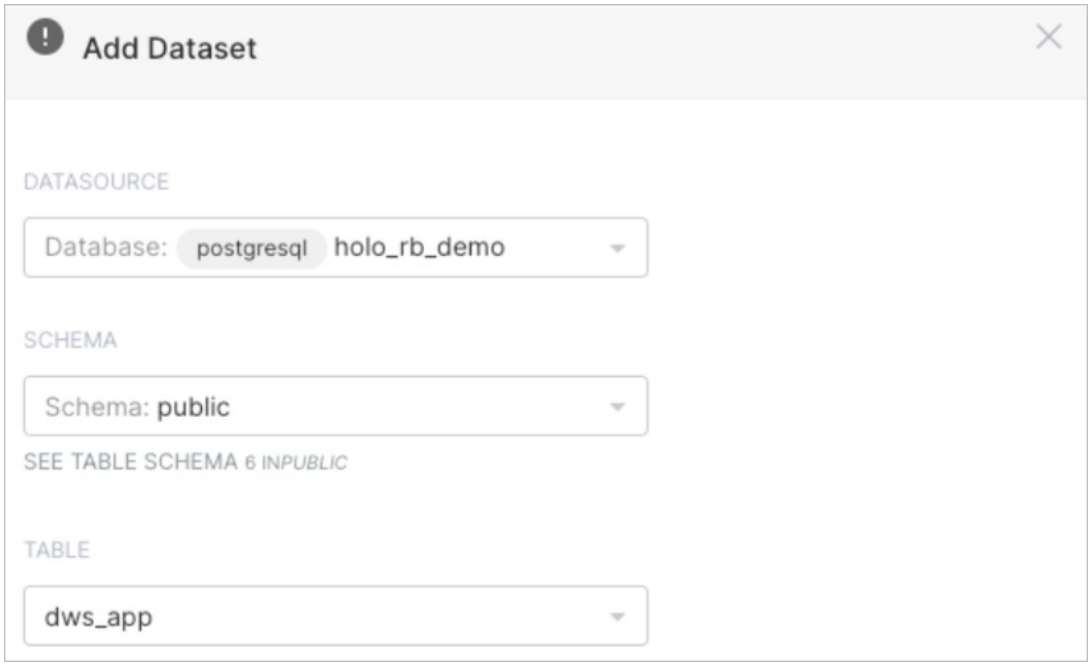
4. 可视化展示

   计算出UV、PV后，大多数情况需要使用BI工具以更直观的方式可视化展示，由于需要使用RB_CARDINALITY和RB_OR_AGG进行聚合计算，需要使用BI的自定义聚合函数的能力，常见的具备该能力的BI包括Apache Superset和Tableau。

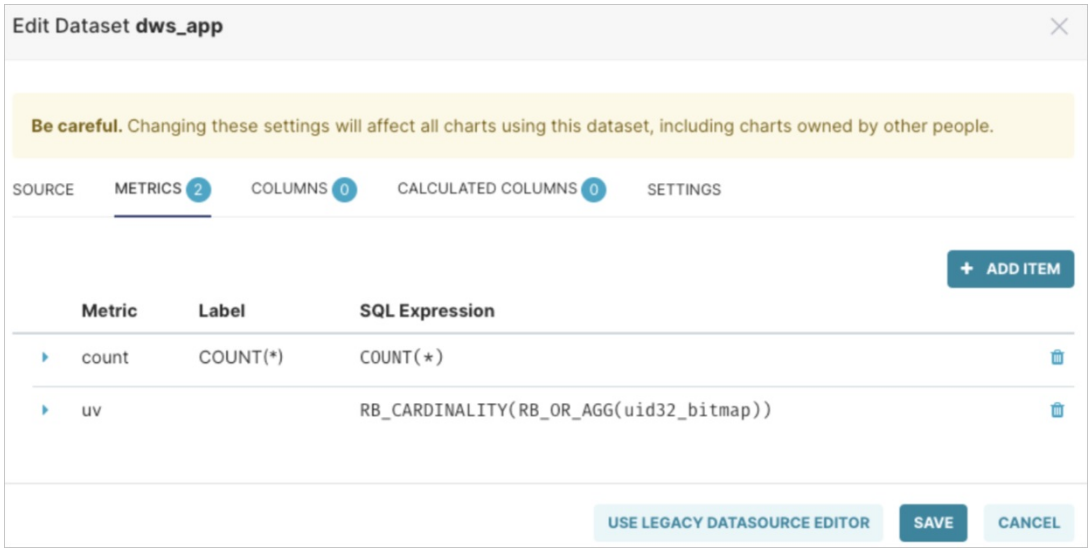
   - Apache Superset

     a. Apache Superset连接Hologres，详情请参见Apache Superset。

b. 设置dws_app表作为数据集。



c. 在数据集中创建一个名称为UV的单独Metrics，表达式如下。




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

完成后您就可以开始探索数据了。

d. （可选）创建Dashborad。

创建仪表板请参见Create Dashboard。

- Tableau

a. Tableau连接Hologres，详情请参见Tableau。

可以使用Tableau的直通函数直接实现自定义函数的能力，详细介绍请参见直通函数。

b. 创建一个计算字段，表达式如下。


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

完成后您就可以开始探索数据了。

c. （可选）创建Dashborad。

创建仪表板请参见Create a Dashboard。

3.4. 用户画像分析

3.4.1. 用户画像分析概述

本文为您介绍在Hologres中标签、画像分析场景的最佳实践。

行业背景与痛点

画像分析是指基于沉淀用户的自然属性、行为属性、偏好属性等属性挖掘用户兴趣点、分析群体特征的过程。用户画像是刻画出用户个体或者用户群体全方位特征的重要手段，能为运营分析人员提供用户的偏好、行为等信息进而优化运营策略，为产品提供准确的用户角色信息以便进行针对性的产品设计。画像系统通常集用户特征加工、画像分析功能于一身；经过离线特征加工、维度标签映射、载入即席分析数据等过程，提供实时人群分析、圈选能力。

画像分析方法论已经广泛应用于各个行业，是赋能经营策略优化、精细化运营、精准营销的重要手段。例如以下典型场景。

- 广告行业：通过人群画像洞察，实现精准广告定向投放。
- 游戏行业：分析高流失率客户群，调整策略增加用户粘性。
- 教育行业：分析课程质量，达到增加续保率的目标。

画像分析的工程场景往往由于数据复杂度、数据量级和查询模式等因素导致系统可稳定性、运维性、可扩展性面临重重困难。

- 运维人员需要维护多套数据链路用于实时离线处理，陷入繁重链路维护工作；传统OLAP（On-Line Analysis Processing）引擎存储计算耦合，计算存储不成比例场景浪费资源，系统扩容迁移成本高。

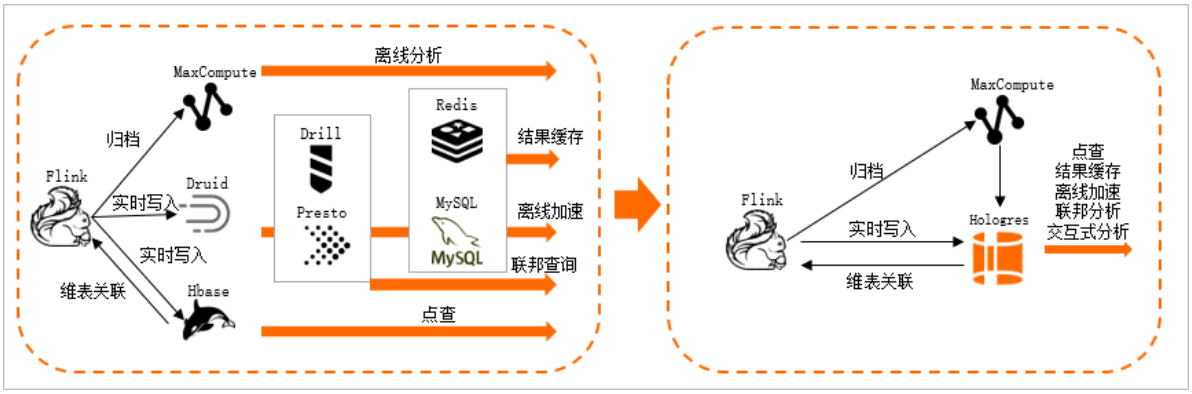
- 运营人员需要灵活的圈选能力，单用户描述维度多可能多达数千维度，涵盖属性、行为等数据模式，MOLAP（Multidimensional OLAP）产品可以毫秒响应但缺乏灵活性，ROLAP（Relational OLAP）产品灵活性好但响应时间较长，无法兼顾性能和灵活性。

Hologres解决方案

针对上述两方面问题，基于新一代实时数仓产品Hologres的系统能力，通过配置数据链路、选择插件库、根据系统规模选择方案步骤快速构建高性能、可扩展的系统方案。

- 数据链路

  依托Hologres通常只需要维护一套数据链路即可实现实时、离线的数据处理，避免常见的数据不同步、数据孤岛等问题，如下图所示。



  Hologres 数据集成能力方面的主要优势如下。

  - 无缝集成DataWorks产品，通过接入配置即可解决复杂数据依赖问题，构建稳定离线数据处理加载流程。
  - 为实时写入场景提供了基于LSM（Log-Structured Merge）结构的行存储，与Flink进行深度融合，能够为实时标签、实时特征处理等场景提供稳定的性能支撑。
  - 具有联邦查询能力，通过外部表方式直接访问MaxCompute、OSS、其他Hologres实例等外部数据存储。

- 画像计算

  Hologres兼容PostgreSQL生态，内置函数丰富；同时，经过阿里内部及云上客户实践，逐步沉淀了诸多高效的画像计算插件，如下所示。

  - 精确去重运算：Roaring Bitmap函数

    Hologres原生支持了Roaring Bitmap类型，通过高效率的Bitmap压缩算法，支持集合的交叉并等运算，支持Bitmap聚合，适合计算超高维度、基数的表，常用于去重（UV计算）、标签筛选、近实时用户画像等计算中。

- 行为数据圈人：明细圈人函数

  在行为类数据的圈人场景中，我们经常碰到这样的情况：行为数据按照天或者小时记录在行为表中，当需要找到一段时间内出现某些行为的用户时，因为数据记录成多行而没办法直接过滤，所以就需要使用行为表多次JOIN自己来实现过滤。例如如下场景，在记录用户行为明细表中找出 时间在[20200216~20200218之间 & [click购物车] & [view收藏页] 的用户。

  行为明细表
  ods_app_ded

uid	action	page	product_code	ds
A	click	购物车	MDS	20210216
B	click	首页	MDS	20210216
A	view	收藏页	MDS	20210217
B	click	购物车	MDS	20210218
A	click	收藏页	MDS	20210219

  Hologres提供了 bit_construct 、 bit_or bit_match 函数，能够规避JOIN的性能负担，简化SQL的复杂度。函数的主要思路是通过一遍数据过滤，将uid满足条件的集合以位数组形式存放，通过 bit_match 函数在位数组进行与运算实现数据过滤，示例如下。

```
WITH tbl as (
SELECT uid, bit_or(bit_construct(
  a := (action='click' and page='购物车'),
  b := (action='view' and page='收藏页'))) 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 函数：用于对表达求值并存储在响应位数组中，比如对SQL中的a、b两个条件，计算结果分别是 [1,0], [0,0], [0,1]... 。
  - bit_or 函数：用于将两个位数组按位进行或运算，用来聚合uid上满足的条件集合。
  - bit_match ：用于判断位数组是否符合某个表达式，比如计算 a&b 表达式 [1,1] 结果为True， [1,0] 为False。

- 漏斗留存分析：漏斗分析函数

  漏斗分析是常见的转化分析方法，它用于反映用户各个阶段行为的转化率，广泛应用于用户行为分析和App数据分析的流量分析、产品目标转化等数据运营与数据分析。

  窗口漏斗函数（WindowFunnel）可以搜索滑动时间窗口中的事件列表，并计算条件匹配的事件列表的最大长度。留存分析是最常见的典型用户增长分析场景，用户经常需要绘制数据可视化图形，分析用户的留存情况。通过漏斗函数、留存函数的使用，可以快速计算出用户留存效果以及对应的转化率，减少复杂Join开销，提高性能。

- 向量检索：Proxima向量计算

  Proxima是一款来自于阿里达摩院的实现向量近邻搜索的高性能软件库，相比于Fassi等开源的同类产品，Proxima在稳定性、性能等方面更为出色，能够提供业内高性能和效果显著的基础方法模块，支持图像、视频、人脸等各种应用场景。Hologres向量查询功能与Proxima深度整合，提供高性能的向量查询服务。支持快速的RNN（Radius Nearest Neighbor）搜索、KNN（K-Nearest Neighbor）搜索、dot_product向量化点积计算组件。

- 工程方案

  在画像系统发展的不同阶段，往往对工程方案有不同的成本和性能诉求。根据实践经验，综合系统数据规模、实现成本、查询性能等三因素，总结两种典型的工程方案如下。

  - 标签宽表方案

    宽表标签方案适合标签较少（通常小于1000个），数据更新不频繁的场景。主体思路是在离线阶段把相对稳定的属性表离线聚合成宽表，将多张表的关联操作转化一张宽表的运算，新的标签列的场景通过增加列的方式实现，以表的方式提供非常灵活的标签计算，详情请参见画像分析 - 标签宽表。

  - RoaringBitmap优化方案

    基于RoaringBitmap的超大规模画像分析场景，适合数据量大，标签规模多，需要去重处理的场景。通过结合RoaringBitmap结构化存储，实现天然去重，避免Join开销，降低运算复杂度，快速出结果。详情请参见画像分析 - RoaringBitmap优化方案。

- 小结

  Hologres通过丰富的画像分析插件支持，和自身优异的性能，被阿里集团内部多个核心业务广泛应用于标签计算、画像分析的场景，例如阿里妈妈、搜索、高德以及众多公共云用户使用。服务扩展能力和稳定性历经生产考验，这也证明Hologres是构建低开发运维成本，高稳定性扩展性画像分析平台的不二之选。

3.4.2. 画像分析 - 标签宽表

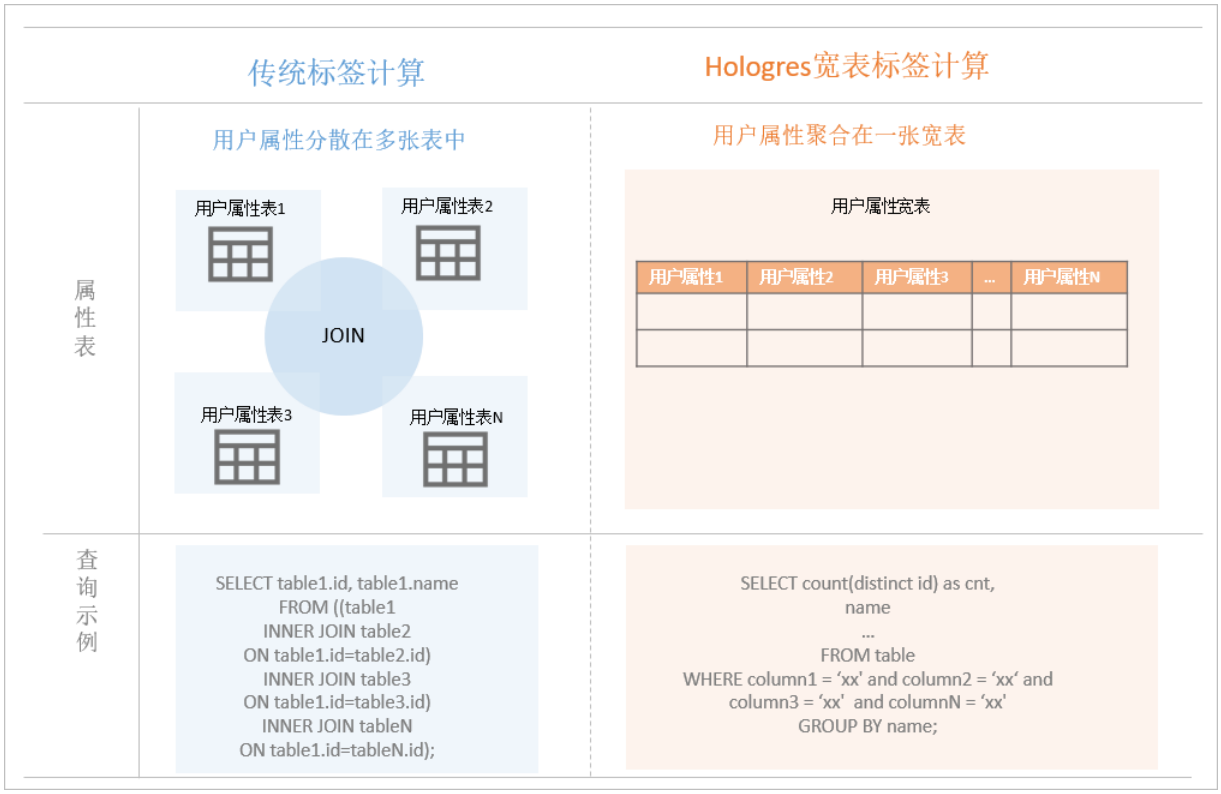
本文为您介绍在Hologres中如何通过宽表实现标签计算的最佳实践。

背景信息

离线数仓模型中，用户标签数据被分隔成面向主题、维度的多张表，这样的组织形式便于体系化的构建标签系统及数据维护管理。在线画像分析服务如果按照这样的数据模型组织标签数据，不可避免的需要Join多张标签表来完成多标签的过滤，这对于数据库产品开销太大。

方案介绍

Hologres标签宽表的方案是指将相对稳定的属性表离线聚合成宽表，将多张表的关联操作转化成对一张表的计算，新的标签列的场景可以通过增加列的方式实现，如下图所示。



这种方案适用于以下场景：

- 标签数量较少（小于1000）。
- 数据稳定（更新较少）。

将数据存在宽表中，多列之间过滤条件的与或非运算将被列存的优化机制自动处理，相比任何Join方式都要更加高效；另一方面，Hologres支持列式存储，不会产生IO放大的问题。使用宽表方案可沿用传统数据库模型建模和开发应用。

使用示例

使用标签宽表进行画像分析的示例如下，进行洞察dws_userbase表中`[省份='浙江'] & [性别=男性]`在已婚状态上的统计分析，示例SQL如下。建议根据查询方式对表设置合理的索引，以提升查询性能，详情请参见CREATE TABLE。

```
-- 属性数据宽表
BEGIN;
CREATE TABLE dws_userbase
(
  uid text not null primary key,
  province text,
  gender text,
  married text
  ...              -- 其他属性列
);
call set_table_property('dws_userbase', 'distribution_key', 'uid');
call set_table_property('dws_userbase', 'bitmap_columns', 'province,gender,married');
END;
-- 洞察基础属性
SELECT count(distinct uid) as cnt,
       married
FROM dws_userbase ub
WHERE province = '浙江' and gender = '男'
GROUP BY married;
```

3.4.3. 画像分析 - RoaringBitmap优化方案

当业务标签越来越多时（大于1000列），一般就认为是超大规模的场景，宽表标签计算的方案将不再适合，因为当列越多时，更新效率将会越慢。本文将会介绍如何通过Hologres进行超大规模标签计算、画像分析。

背景信息

Hologres兼容PostgreSQL生态，原生支持Roaring Bitmap函数。通过对标签表构建索引，将用户ID编码后以Bitmap格式保存，将关系运算转化Bitmap的交并差运算，进而加速实时计算性能。在超大规模用户属性洞察分析的场景中，使用RoaringBitmap组件能够实现亚秒级的查询响应。

适用场景

使用RoaringBitmap的方案的可以适用于如下场景。

- 标签数量多：需要对多张大表进行关联运算，可以使用 `BITMAP_AND` 替换JOIN运算，可以降低内存消耗，Bitmap插件库使用SIMD指令优化可以将CPU使用率提升1~2个数量级。
- 数据规模大且需去重运算：如数十亿数据需要去重，Bitmap结构天然去重，避免精确UV计算和内存的开销。

标签数据分类

在介绍Bitmap计算方案之前，我们需要区分画像系统中常用的两类标签数据，针对这两类数据的计算模式大相径庭，我们需要依据数据模型和运算模式将合理的部分转化为Bitmap格式存储。

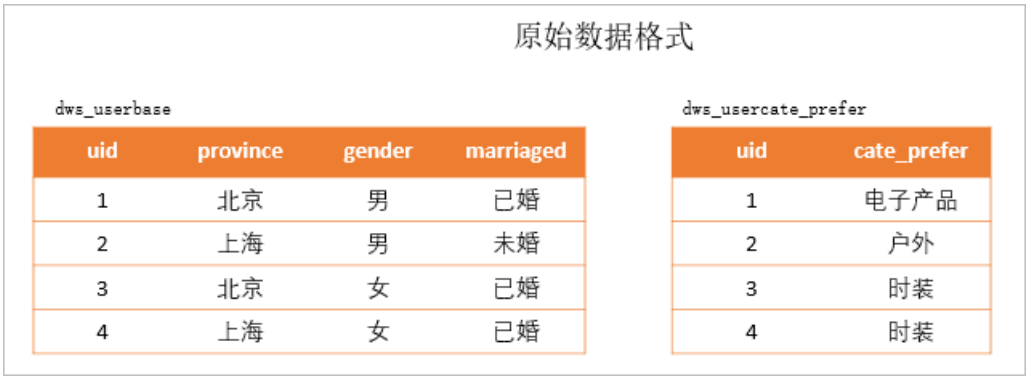
- 属性标签：主要描述用户的属性情况，例如用户性别、所在省份、已婚状态等，数据相对稳定，通常进行精准条件过滤。这类数据进行Bitmap压缩比会很高，且Bitmap适合对应的运算。
- 行为标签：主要描述用户行为特征，描述用户在某个时间做了一件什么事，比如用户店铺浏览购买行为、用户登录活跃行为等，数据变更频率高，通常需要进行范围扫描，聚合过滤。这类数据不适合进行Bitmap压缩，压缩比会很差，运算模式不适合Bitmap直接运算。

属性标签处理

属性标签处理场景通常是描述用户的属性情况，数据相对稳定，通常进行精准条件过滤，通过Bitmap能实现高效的压缩和运算。

- 方案介绍

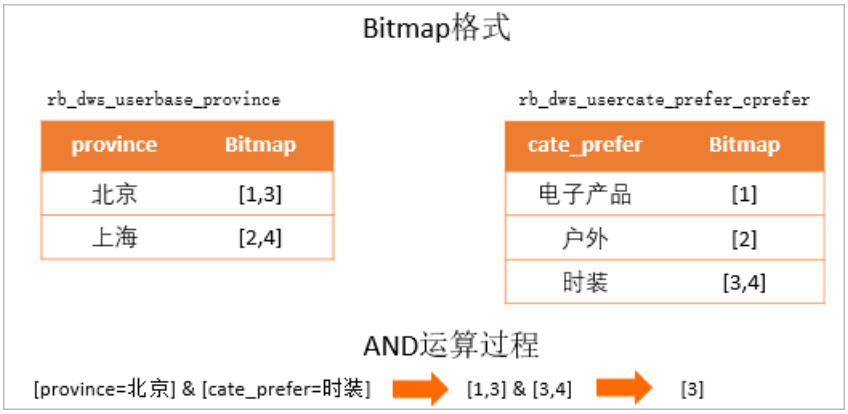
  假设现有的DMP（Data Management Platform）系统中存在两张属性标签表，如下图所示。



  dws_userbase表描述用户基础属性，dws_usercate_prefer表描述用户偏好。

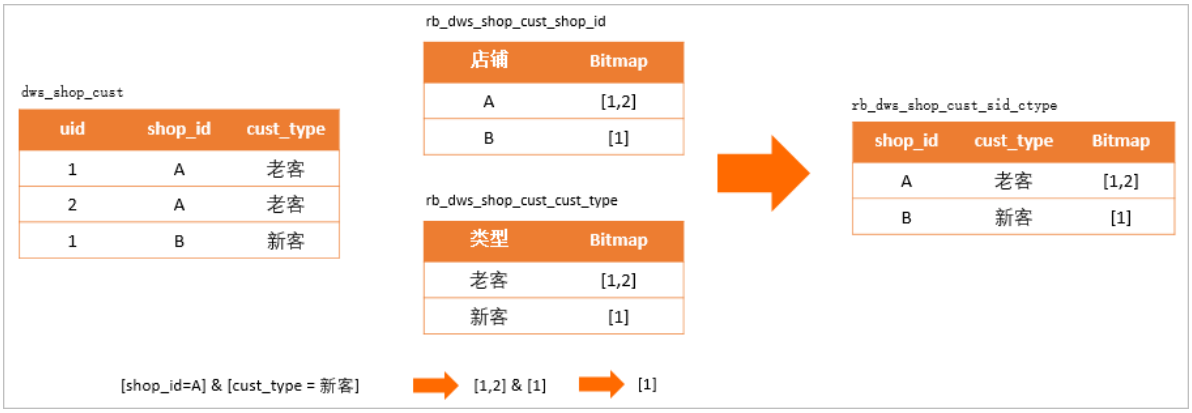
  典型的人群人数预估的场景比如计算 `[province = 北京] & [cate_prefer = 时装]` 的人数，通常可以关联、过滤、去重最终获得人数。但在数据规模较大的场景，关联、去重运算会带来严重的性能负担。

  Bitmap优化方案通过预构建标签的Bitmap表来减少即席运算的成本，比如将两张表中的列拆开分别构建Bitmap表如下图所示，通过两表中对应Bitmap的与运算计算出人数。



  rb_dws_userbase_province表描述省份和uid的Bitmap关系，rb_dws_usercate_prefer_cprefer描述类目和uid的Bitmap关系。

但是按列拆分的方案也存在一定的问题，当多列之间存在层级的组织关系时，上述的拆分和运算方式可能会导致计算错误的情况，如下图所示。



在描述店铺用户新客、老客、潜客信息的dws_shop_cust表中，按照列拆分成描述店铺名称的Bitmap表rb_dws_shop_cust_shop_id，描述客户类型的Bitmap表 `rb_dws_shop_cust_cust_type` 。当计算 `[shop_id = A] & [cust_type = 新客]` 的人群集合，会得到 `[1]` 的uid集合，然而在真实的表中这条数据不存在。产生这种错误的原因是cust_type、shop_id两个字段存在一定的关联性，在数仓模型中cust_type是shop_id维度的指标数据，脱离统计维度单独使用指标是错误的。因此可以将维度shop_id和指标cust_type组合值作为构建Bitmap的单元，生成rb_dws_shop_cust_sid_ctype表来避免这类错误。

根据以上方案介绍，需要将uid压缩存储进Bitmap，通过Bitmap与或非运算实现标签的对应运算。

- 方案实践

- 用户信息编码处理

  用户标识可能是字符类型的，由于Bitmap只能保存整数信息，因此需要先将uid进行整数编码进Hologres中，可以通过插入包含自增序列Serial（Beta）的表完成字符ID的编码。

```
-- 创建字典表
CREATE TABLE dws_uid_dict (
    encode_uid bigserial,
      uid text primary key
);
-- 录入标签表中的uid
INSERT INTO dws_uid_dict(uid)
SELECT uid
FROM dws_userbase ON conflict DO NOTHING;
```

  对用户标识进行编码不仅可以方便存储进Bitmap，还可以使ID信息保持连续性。如下图所示bitmap2由于存储的ID数据稀疏，存储效率相比bitmap1低很多，因此对ID编码处理，也会降低存储成本，提升运算效率。

	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

  对于数值类型，如果较为稀疏也可以考虑编码，但是进行编码有利有弊，例如在广告DMP系统中除了需要高性能的画像能力，也需要实时输出人群明细的能力；而一旦需要输出人群明细，就需要Join用户ID表进行还原，这一部分的性能开销，也应当纳入技术选型的考虑因素中。因此您可以依据使用场景权衡选择，是否进行编码建议如下。

  - 字符ID：建议编码。
  - 整数ID且需要频繁还原原始值：建议不编码。
  - 整数ID且不需要还原原始值：建议编码。

- Bitmap 加工和查询

  依据上述方案按列拆分的思路，将dws_userbase表与dws_shop_cust表进行拆分，按照分别为省份、性别的列Bitmap拆分为一个表，但是性别只有男、女两个选项，压缩出来的Bitmap只能分布于集群中的两个节点，计算存储都很不平均，集群的资源并不能充分利用。因此有必要将Bitmap拆分成多段，并将它们打散到集群中来提升并发执行的能力，假设将Bitmap打散成65536段，SQL命令如下。

```
-- dws_userbase 结构见宽表方案
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;
-- 创建bitmap插件
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;
-- 写入bitmap表
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";
```

当进行 `[shop_id = A] & [cust_type = 新客] & [province = 北京]` 的客群人数预估时，按照标签的与或非逻辑组织对应的Bitmap关系运算，即可完成对应的查询，SQL命令如下。

```
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 = '北京'
   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 = '新客'
   GROUP BY bucket) uc ON ub.bucket = uc.bucket;
```

行为标签处理

常见的实时表往往包含时间维度，比如按天汇总的用户行为实时表。就某一天的数据而言，用户的数据仅包含有限几条数据。由于Bitmap本身存在结构行存储开销，压缩成Bitmap不但不能节省存储空间，更可能造成存储浪费。另一方面，事实表典型的计算模式中需要汇总多天数据进行聚合过滤，如果使用Bitmap存储可能需要展开再汇总运算。同时由于这类数据变更频繁，有时更需要实时更新；在 `[option->bitmap]` 的存储结构中，无法直接找到需要更新的数据。因此Bitmap并不适合用于压缩行为数据，不适合进行汇总运算，不适合实时更新。

当涉及到行为标签数据处理的场景时，在Hologres中可以保持原有的存储格式。当发生事实表与属性表联合运算时，可以将实时表过滤结果实时生成Bitmap，再与属性表Bitmap索引进行运算。同时由于Bitmap索引表使用Bucket作为分布键（Distribution Key），通过Local Join提升性能。

当我们计算 `[province=北京] & [shop_id=A且7天未购买]` 的用户时，SQL命令如下所示。

```
-- 原始行为表
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;
-- 编码行为表
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;
-- 写入分桶的事实数据
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;
-- 事实数据和属性数据联合运算
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 = '北京'
      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
```

离线Bitmap处理方案

为了避免Bitmap数据计算对生产业务的影响，可以选择在离线完成Bitmap数据的加工，通过Hologres外部表能力从MaxCompute或者Hive中直接加载数据。离线处理Bitmap的过程与在线流程思路类似，也可以通过编码、聚合方式产生数据，在MaxCompute中离线构建Bitmap数据示例如下。

```
-- 选择project
USE bitmap_demo;
-- 创建原始表
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
);
-- 对新增的UID进行编码
-- 计算新增编码uid
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),
-- 计算每个分桶中待编码的uid数量，通过sum窗口累加出bucket编码的起始值
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),
-- 计算已使用编码值最大值
dict_used_id_offset AS
  (SELECT max(encode_uid) AS used_id_offset FROM mc_dws_uid_dict)
-- 新增用户的编码 = 已使用编码值最大值 + Bucket编码起始值 + RowNumber序号
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
-- 创建bitmap相关函数
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;
-- 创建bitmap表并写入
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;
```

在Hologres中执行以下命令。

```
-- 创建MaxCompute外部表
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');
-- 将外表中bitmap数据写到Hologres中
INSERT INTO rb_dws_userbase_province
SELECT province,
       bucket::INT,
       roaringbitmap_text(bitmap, FALSE)
FROM mc_rb_dws_userbase_province;
```

通过上述步骤，可以将数据加载至Hologres，然后按照标签圈选条件，组合Bitmap运算加速查询。

- MaxCompute中计算Bitmap数据可使用mc-bitmap。
- 离线加工使用Bitmap方案更多信息请参见离线UV计算

实时Bitmap处理方案

在实时计算场景中，可以使用Flink和Hologres组合的方式，基于RoaringBitmap实时对用户标签去重，主要思路如下。

1. 将用户ID字典表作为维表，利用Hologres的Insert on Conflict机制处理新增编码，在Flink中进行维表Join。
2. 将Join结果流按照标签维度进行RoaringBitmap聚合。
3. 输出的Bitmap结果写入Hologres对应的Bitmap表中。

详情请参见画像分析 - 实时标签。

3.4.4. 画像分析 - 实时标签

本文将为您介绍Hologres的实时标签计算最佳实践。

背景信息

标签计算、画像分析中，对实时效果回流和实时标签生产能力往往会更加关注。

- 实时效果回流能力：用来统计画像所圈选人群在业务上的标签，比如广告系统的点击率、转化率。使用回流数据能为做出实时决策调整提供基础条件。
- 实时标签生产能力：实时特征可以理解为用户在投放活动中的实时表现，典型的特征包括最近N天的浏览次数、当天收藏商品等，通过分析此类特征，实时向用户推送运营策略，可以刺激用户购买意向并最终形成转化。

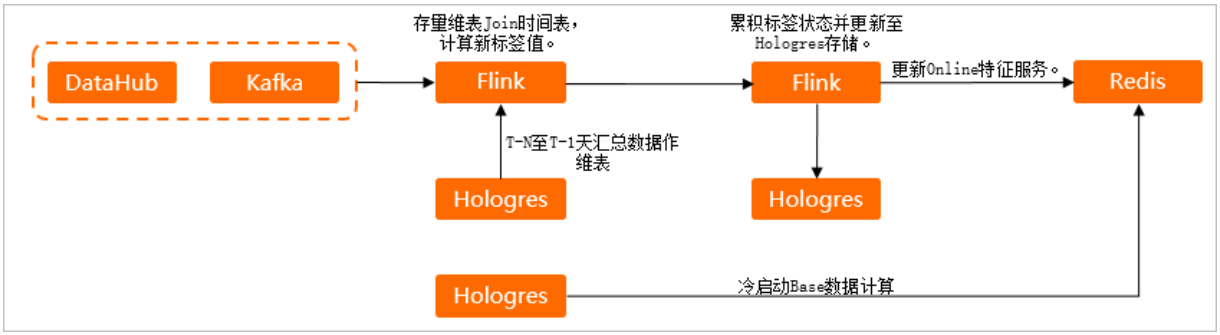
Hologres与Flink深度集成，支持高性能的数据写入与更新，数据写入即可查询。实时效果回流和实时标签生产都可以通过Hologres和Flink的组合方案来完成，应用于众多业务场景中。

实时效果回流场景

通过Hologres和Flink提供的实时能力直接支持，Flink数据实时写入Hologres，通过Hologres内置的画像分析插件，如漏斗留存函数等实时计算标签，达到实时数据实时决策的效果。

实时标签生产场景

实时标签生产链路通常来说会较为复杂，需要Flink和Hologres更多的结合（如维表关联）来完成。如下图所示近N天浏览次数标签生产和使用流程。



具体方案步骤如下。

1. Flink创建实时用户行为日志表dwd_user_visit_log，用于实时接收DataHub、Kafka中的实时行为数据。
2. Hologres创建历史行为表dws_user_visit，用于存放 `T-N ~ T-1` 天级汇总的行为数据，计算标签冷启动数据，并将此表作为Flink的维表。
3. 在Flink中，将实时用户行为日志表和历史用户行为表通过维表Join，计算新的标签值。
4. 新的标签累计状态后实时更新至Hologres表中存储。
5. 在Flink中将标签变更同步到在线特征存储（如Redis）中，用于实时的召回请求。
6. 若是需要冷启动Base数据计算，直接将Hologres中的数据写入Redis。