YonghongTech
Talk with Data

数据集市手册

Yonghong Z-Suite — V7.5

版权声明

免责声明

通讯方式

北京永洪商智科技有限公司

北京市朝阳区光华路 9 号光华路 SOHO 二期 C 座 9 层

电话：(86-10)-58430919

邮箱：public@yonghongtech.com

网站：http://www.yonghongtech.com

目录

工业革命之后，书籍等以文字为载体的知识大约每十年翻一番； 1970 年以后，知识大约每三年就翻一番；如今，全球信息总量每两年就翻一番； 2010 年互联网的数据量，比之前所有年份的总和还要多。

现在，人类每天产生数以 PB 的数据。在互联网、电子商务、生产制造、交通和物流、金融和保险、医疗卫生、地理信息、政府机构等行业，每天都在创造着大量的数据。大数据正在成为从工业经济向知识经济转变的重要特征，已经成为新时代最关键的生产要素和产品形态。

Google、 Yahoo、 Facebook 等公司正成为这场变革的推动力量，同时新企业也层出不穷。在商业智能（BI）领域， AsterData、 Greenplum、 Vertica 等公司刚刚卓然而生，便被传统 IT 巨头 EMC、 IBM、 HP 等公司各自收入囊中。经过对这些新生公司的大数据技术进行消化和整合之后，传统 IT 巨头们迅速推出了各自的大数据产品和服务。

数据库时代之后，随着可用数据的持续积累，各行业的领军企业逐步开始了数据价值的发现之旅，这一阶段的商业智能系统，一般是以数据仓库 +OLAP 为主。一般地，传统数据仓库能够存储大数据，但并不提供针对大数据的分析和统计功能，因此，在开发 OLAP 这种数据应用时，需要用户预先提出的分析及统计的需求，再预先计算好这些主观的分析及统计的结果，才能确保 OLAP 系统的实时交互能力。然而，数据仓库 +OLAP 这一组合有着其先天的缺陷，在终端用户眼中也许是一个微小的变化，却可能需要很长的响应周期。行业内企业整体经营管理水平的持续提高，竞争态势不断加剧，这对每个企业尤其是领军企业带来了巨大的挑战。

要很好地应对这种挑战，保持行业优势地位，企业对商业智能系统的提出了更高的要求。永洪认为直接导入细节数据的这一数据建模技术，将数据和应用之间的关系从紧耦合改造成松耦合，让大多数分析应用不引起数据层的任何改变；而基于 MPP 架构的商业智能系统，能够

直接对细节数据进行高性能分析。这样一来，用户可以快速开发出数据应用，并随即进行实时分析。建设随需应变的发现型、自服务商业智能系统。

永洪 Z-Data Mart 是基于自有技术研发的一款数据存储、数据处理的数据集市产品。针对客户需要处理需求数据的量级不同， IT 系统架构的不同和存储系统的不同，提供了两种解决方案供客户选择一种本地模式，一种是 MPP 模式。当需要处理的数据量级别处于 TB 级以下，或者采用普通存储结构，或者单机已经足够满足性能需求，我们建议用户选择我们的本地模式。当面对异构数据库存储系统，需要处理的数量级别在 TB 级和 PB 级及以上，或者 IT 系统和存储系统采用分布式，或者需要 MPP 模式才能满足性能需求，基于分布式架构的并行处理模式更适合客户的需求。

她完全摒弃了向上升级 (Scale-Up)，全面支持横向扩展 (Scale-Out)。

• 跨粒度计算 (In-Database Computing)

Z-Suite 支持各种常见的汇总，还支持几乎全部的专业统计函数。得益于跨粒度计算技术，Z-Suite 数据分析引擎将寻找出最优化的计算方案，继而把所有开销较大的、昂贵的计算都移动到数据存储的地方直接计算，我们称之为库内计算 (In-Database)。这一技术大大减少了数据移动，降低了通讯负担，保证了高性能数据分析。

• 并行计算 (MPP Computing)

Z-Suite 是基于 MPP 架构的商业智能平台，她能够把计算分布到多个计算节点，再在指定节点将计算结果汇总输出。 Z-Suite 能够充分利用各种计算和存储资源，不管是服务器还是普通的 PC，她对网络条件也没有严苛的要求。作为横向扩展的大数据平台， Z-Suite 能够充分发挥各个节点的计算能力，轻松实现针对 TB/PB 级数据分析的秒级响应。

• 列存储 (Column-Based)

Z-Suite 是列存储的。基于列存储的数据集市，不读取无关数据，能降低读写开销，同时提高 I/O 的效率，从而大大提高查询性能。另外，列存储能够更好地压缩数据，一般压缩比在 5 -10 倍之间，这样一来，数据占有空间降低到传统存储的 1/5 到 1/10 。良好的数据压缩技术，节省了存储设备和内存的开销，却大大了提升计算性能。

• 内存计算

得益于列存储技术和并行计算技术， Z-Suite 能够大大压缩数据，并同时利用多个节点的计算能力和内存容量。一般地，内存访问速度比磁盘访问速度要快几百倍甚至上千倍。通过内存

计算， CPU 直接从内存而非磁盘上读取数据并对数据进行计算。内存计算是对传统数据处理方式的一种加速，是实现大数据分析的关键应用技术。

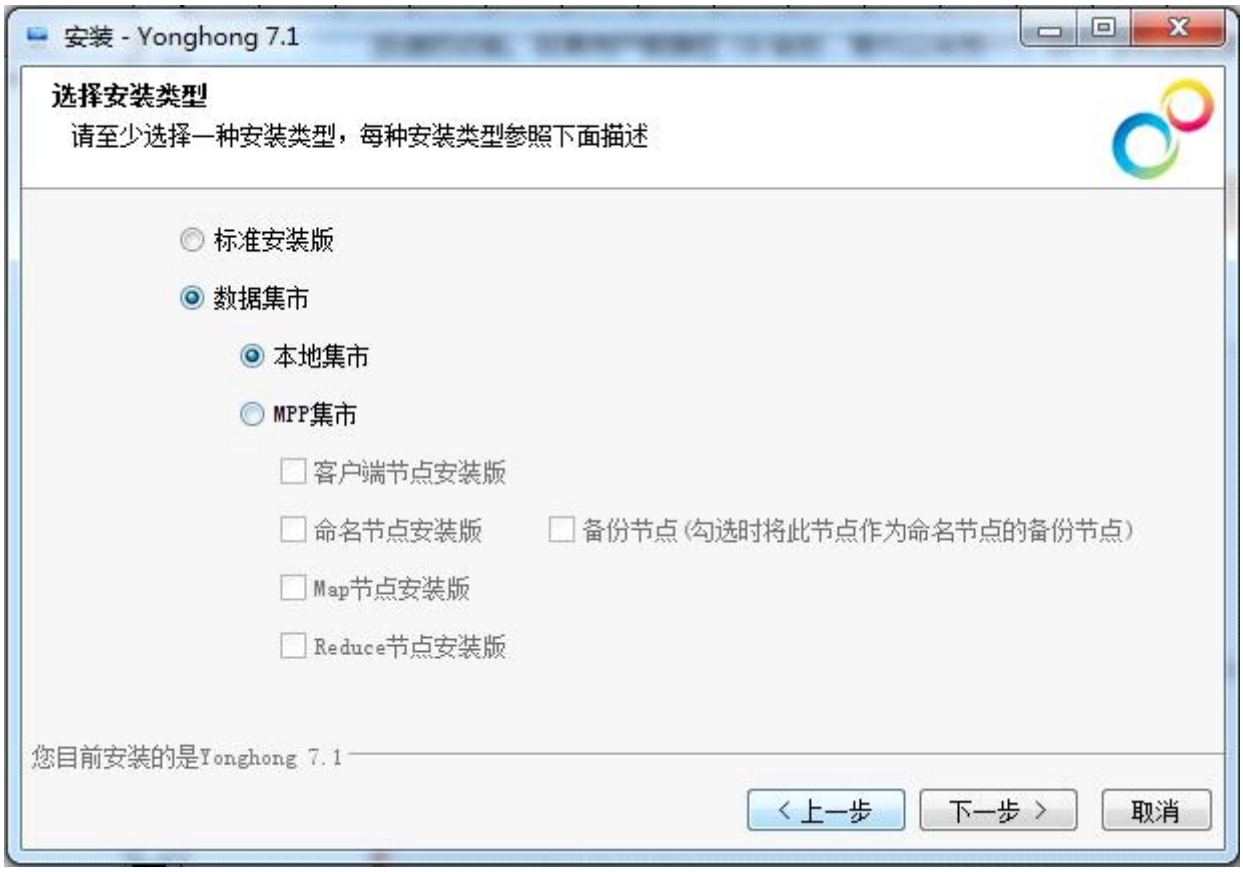
1. 快速搭建并使用

本章节主要介绍如何快速搭建并使用数据集市。

- 机器部署。
- 数据抽取。
- 数据建模。
- 数据应用。

1.1.机器部署

根据用户需求分析的数据量，合理选择机器硬件，以部署一个数据集市来支撑秒级的数据分析响应速度。如果用户数据量在 GB 级别，并且单机服务器配置还可以，采用单机版数据集市系统可以达到数据加速的功能。如果用户数据在 TB 级别，就可以采用一个 MPP 多机版数据集市系统来支撑秒级的数据分析响应速度。



❖ 如何部署单机版数据集市系统

做一个大体的估计：假设单机是一个 2 路 10 核 CPU，内存 32G 的 PC Server，可以支撑 500G 的数据，差不多 3 亿条数据。由于各个行业的数据特点不同，估算因子需要相应调整。

选择好机器硬件，按照安装说明，将永洪数据集市软件安装到操作系统中。安装时选择本地数据集市的安装步骤。如果安装时参数设置有误，可以打开 bihome 下的相关文件来修改参数。

这里列举几个必须参数设置加以说明。

dc.fs.naming.paths=c\:/bihome/cloud/qry_naming.m

dc.fs.sub.path=c\:/bihome/cloud/qry_sub.m

dc.fs.physical.path=c\:/bihome/cloud

dc.io.ip=192.168.1.99

配置命名节点的元数据文件地址， 子节点的元数据文件地址， 子节点存储物理文件的目录，以及本机 IP。

还有两个参数需要特殊考虑下。可以手动到本地配置文件中修改。

mem.serial.mem=10240

mem.proc.count=12

这是安装当前机器配置的。 12 个线程可调度。分配给内存用来装数据的内存为 10G。

另外，在安装时需要考虑给 JVM 所分配的内存大小。在当前环境下，如果没有其他 service 占用资源，可以分配 31G。安装时可以输入 31744。 JVM 的内存大小必须大于装载数据的内存块大小，起码要大于 700M。

安装完，请确认把访问权限给安装包，如果是 linux 下，最好以超级用户的身份来启动程序。

❖ **如何部署一个 MPP 多机版数据集市系统**

假设 4 个节点的集群，机器配置也是 2 路 10 核 CPU，内存 32G 的 PC Server，可以支持 1TB 左右数据的实时分析。由于各个行业的数据特点不同，估算因子需要相应调整。

一个分布式数据集市系统中只能有，一台机器做 Naming，可多台做 Map，可多台做 Reduce，也可以支持多台 Client。一台机器可以是多种类型的组合。用户可根据机器的数量，配置和数据访问量等指标来考虑定义各机器的节点类型。

➢**例如**：在当前情况下，采用 1 台做四种角色，另外 3 台只做 Map 节点。对于做四种角色的那台机器，可以选择配置比其他稍微好一点机器。安装永洪软件时，勾选 MPP 多机版数据集市，并复选四种角色。按照说明填好必选参数。在其他 3 台上也安装永洪软件，勾选 MPP 多机版数据集市，并勾选 Map 节点的角色。

另说明命名节点也可以做到冷备份。

dc.fs.naming.paths=c\:/bihome/cloud/qry_naming.m;\\192.168.2.99/bihome/cloud/qry_naming.m

这里用 ";" 来分隔两个不同路径下的元数据文件，如果当第一台宕机并磁盘损坏， 可以很快通过手动方式把第二台上的元数据信息获取，并启动一个新的命名节点。

安装完，请确认把访问权限给安装包，如果是 linux 下，最好以超级用户的身份来启动程序。

将所有机器节点启动后，即可按照使用文档的说明来提取数据并分析数据。

❖ **如何部署一个使用 Naming 节点双活的 MPP 多机版数据集市系统**

MPP 集市中，Naming 节点只有一个，会存在单点故障。Yonghong 通过 ZooKeeper 的领导者选举，选举新的 Naming 节点来实现 Naming 节点的双活。ZooKeeper 有 Server 和 Client, 在这里 Client 指的是 MPP 集市中的节点。通过在 MPP 集市系统中启用多个备份 Naming 节点，ZooKeeper 选举出一个 First 备份 Naming 节点，ZooKeeper Client 连接到 Server, 通过心跳保持连接，从而实时监控 Naming 节点的状态并实现 Naming 节点和 First 备份 Naming 节点的元数据文件同步。当 Naming 节点宕机后，备份 Naming 节点会成为 Naming 节点来保证集市系统的正常工作。

- 部署 zookeeper 并配置相关属性

ZooKeeper 的部署分为单机模式和集群模式。集群模式是指在多个节点上启动 ZooKeeper Server。例如一个 5 个节点的分布式集市环境，其中一个 CN 节点，一个 M 节点，一个 R 节点，两个备份 Naming 节点。如果使用 Naming 节点双活机制，则在安装每个节点时都需要勾选使用 Naming 节点双活机制，同时在奇数个节点（一般考虑在三台机器上）部署 ZookKeeper 并配置相应的属性，组成一个 ZooKeeper 集群。具体步骤见永洪安装手册。

- 启动和使用

首先启动 ZooKeeper 集群，待 zookeeper 集群稳定后，再启动数据集市中的 Naming 节点，然后再启动集市系统中的其他节点：备份 Naming 节点，Client 节点，Map 节点和 Reduce 节点。

当数据集市的节点与 zookeeper 集群连接后，不同的节点在日志中会显示不同的连接信息：

Client 节点 &Map 节点 &Reduce 节点：

Set watch on znode /naming_ip

Connecting to ZooKeeper

First 备份 Naming 节点：

Set watch on znode /naming_ip.

Chosen to be first backup node in zookeeper.

Connecting to ZooKeeper, node path is /election/node_n0000000015.

Chosen to be first backup node in zookeeper.

Election nodes are [node_n0000000015, node_n0000000014] ，其中 node_n0000000014 指的是 Naming 节点。

Set watch on znode /election/node_n0000000014.

Set watch children on znode /meta.

Starts to check meta info.

当 Naming 节点宕机后，所有的节点将被通知 Naming 节点已更换。First 备份 Naming 节点会成为新的 Naming 节点，其他备份 Naming 节点会产生出一个新的 First 备份 Naming 节点，可以通过查看相应的日志信息。

当 First 备份 Naming 节点宕机后，会从其他备份节点中产生出新的 First 备份 Naming 节点，可以在备份 Naming 节点中查看相应的日志信息获取到当前的 First 备份 Naming 节点。

Naming 节点和备份 Naming 节点之间的元数据同步采用主从备份机制，具体如下：

（1）在 GSFolders 增加版本信息，每次保存的时候，版本会加 1 ；

（2）元数据修改时，Naming 节点会将修改记录发给 ZooKeeper 保存，按 version 分目录保存，如 meta_v1，meta_v2。假设保存之前 version 为 v0，则日志记录存放在 meta_v1 中，保存后，version 为 v1，在 ZooKeeper 新建 znode （meta_v2），日志记录存放在 meta_v2 中；

（3）备份 Naming 节点在系统开始运行时，从 Naming 节点请求传输 meta file。根据 meta file 的 version(设为 v0)，监听 ZooKeeper 中 meta_v2 的目录是否存在，如果存

在，则合并 meta_v1 目录中的记录，保存成功后，删除 ZooKeeper 的 meta_v1 目录。再重复之前的步骤，实时保持和 Naming 节点落盘的 meta file 一致，并确保一般情况下 ZooKeeper 只有两个 version 的日志信息；

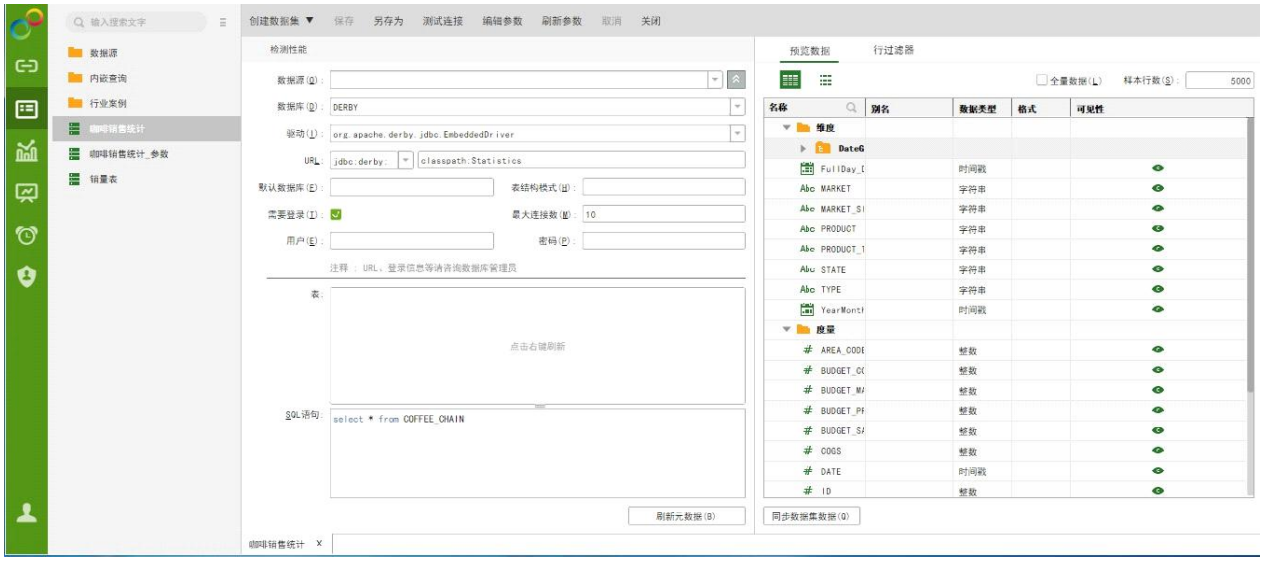
（4）定期（可以设置为 1 小时）将 Naming 节点的 metafile 和备份 Naming 节点的 metafile 进行比较，如果不一致，则备份节点请求传输 meta file。

1.2.数据抽取

数据抽取需要两个操作，首先要创建数据集，来与数据源创建连接，确定要抽取的数据；其次设定抽取任务，把数据源抽取进入永洪数据集市。

❖ 创建数据集

在数据集编辑器中，创建与数据库或文本文件等的数据连接，确定要抽取的数据。



❖ 数据抽取入库

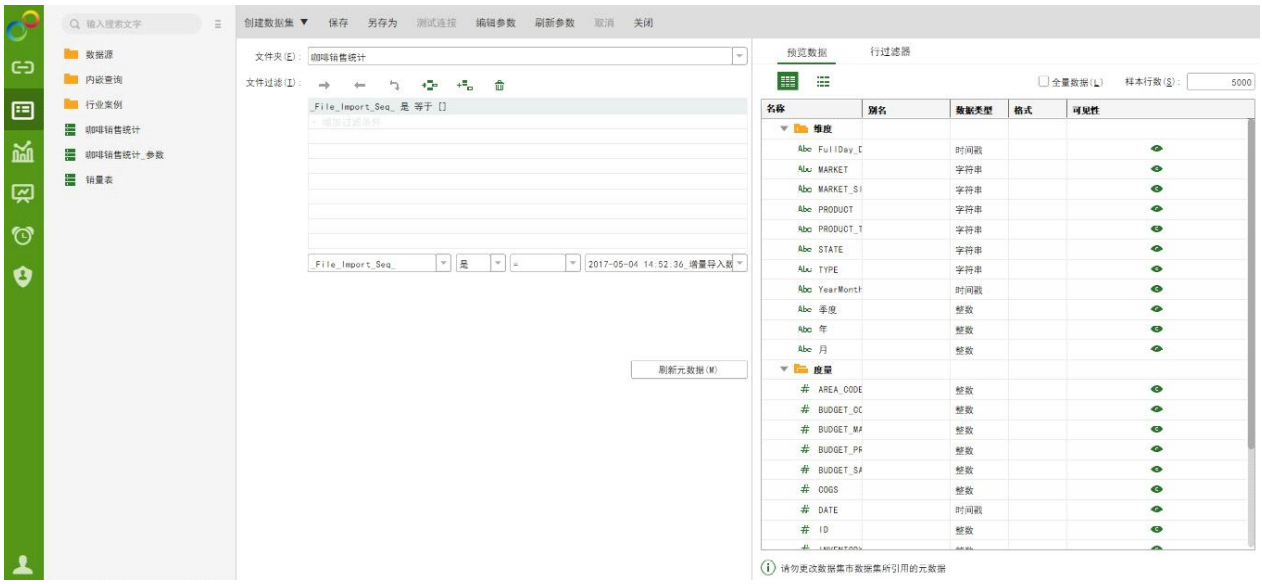
在调度任务中创建增量导入数据作业或任务，把数据库或文本文件等中的数据抽取出来，导入永洪数据集市中。

在抽取数据时用户可自定义触发器，在满足触发条件时，来执行此任务，完成数据抽取。

1.3.数据建模

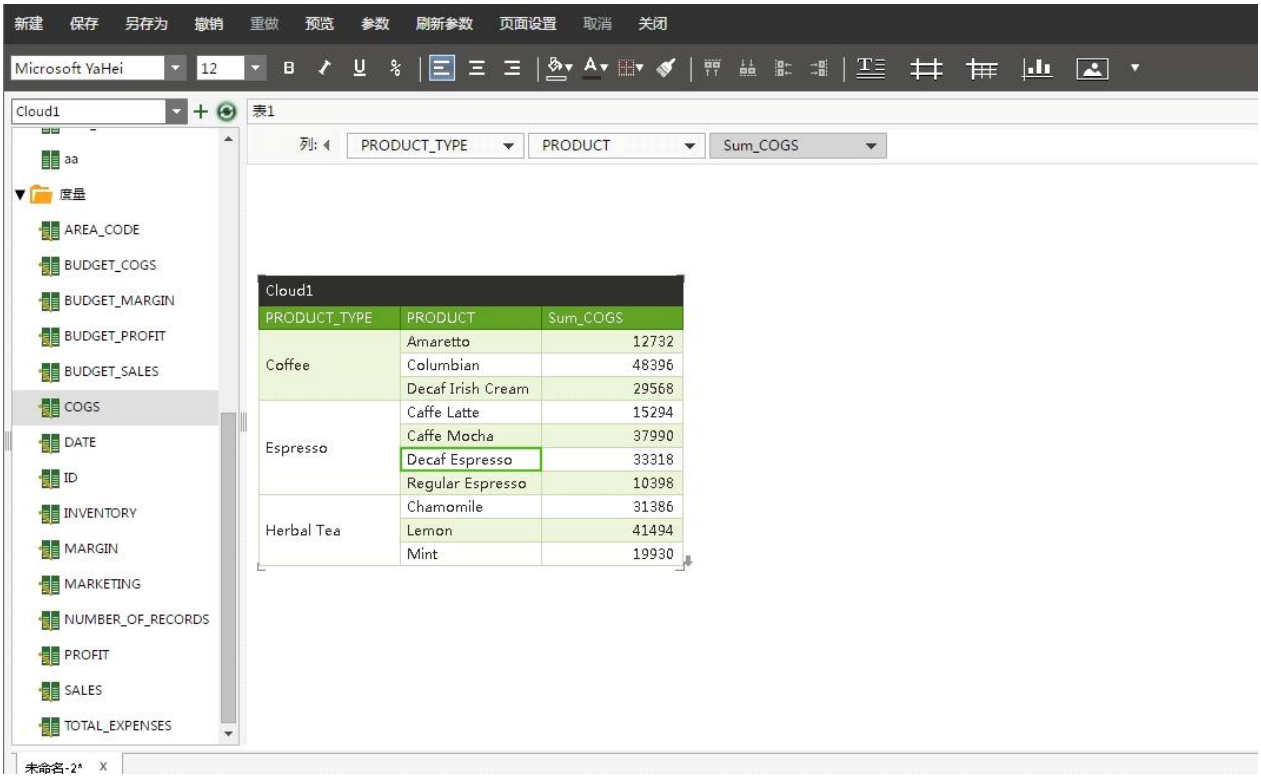
在数据集中通过创建数据集市数据集从数据集市中提取数据进行建模。用户需要在数据集中点击数据集市数据集，打开数据集市数据集编辑器。选择指定的云文件夹，从中提取数据进行建模。

【文件过滤】可实现从数据集市的云文件夹中提取相应的云文件进行建模。例如一云文件夹中的所有云文件均携带日期标签，用户即可通过文件标签来提取指定时间段的数据，如下图所示，提取 2017 年 5 月 4 号的数据进行建模。



1.4.数据应用

数据建模完成之后，需要在编辑器中进行数据分析展现。通过表格、图表等组件绑定数据集，进行数据展现分析。详细的数据展现分析请参考永洪制作报告手册。



2. ETL

本章节主要从以下几个方面介绍 ETL。

- 抽取方式。
- 数据源。
- 如何导入数据。
- 任务调度器。
- 增量更新。
- 加入标签。
- 多节点同时入库。

❖ 抽取方式

数据抽取有两种方式 Pull Data、 Push Data。

Pull Data 是从数据源（ 例如数据库或者文本文件等等 ）里拉取数据。

Push Data 是把数据推送到数据集市中，详细方法见附录。

❖ 数据源

ETL 支持的数据源包括文本， Hadoop （HDFS， HBase， Hive）， 以及各种数据库。

数据源	数据类型
文本	TXT
	CSV
	XML

数据源	数据类型
	AVL
	DSV
	.....
Hadoop	HDFS
	HBase
	Hive
数据库	ORACLE
	MYSQL
	INFORMIX
	DB2
	ACCESS
	DERBY
	SQLSERVER
	SYBASE

数据源	数据类型
	POSTGRESQL
	VERTICA
	HANA
	IMPALA
	KINGBASE
	MONGO
	SPARK
	PRESTO

❖ 如何导入数据

可通过两种方式将用户的数据导入到数据集市系统中（包含本地云和分布式云）。

• 自动方式

启动系统中这节点，在 Client 节点上会有日志显示 Naming 节点可用。在 Client 节点上进入调度任务界面，增加一个作业，定义该作业的任务是增量导入数据。该任务专门负责把一个 Query （数据集）里的数据，提取到集市系统中。选项指标：

Query：选择一个数据集，该数据集的数据会被提取出来。该数据集是通过创建数据集界面来定义的，访问一个数据库。

Folder：提取出来的数据块放到哪个文件夹下。

File：提取出来的数据块以什么文件名为前缀。如果有 2 个数据块，前缀为 aa，那就有 aa0.zb, aa1.zb 两个物理文件。

Append：是否追加文件，否则会删掉历史的文件。如果已有一个 aa 文件，可以再追加一个 bb 文件。如果要追加的文件名已存在，会停止追加。

Script：可以在运行 query 之前执行此脚本。此脚本可以修改 folder， file， append 的值，还可以通过 setMeta/getMeta 来修改元数据；还可以给参数赋值。

当一个作业被成功执行完后，就可以通过创建数据集市数据集来访问该文件夹里的数据。还可以采用过滤元数据的值，来跨粒度访问各数据块。

用户可以指定计划，每隔多久提取一次数据，并在提取的时候，通过加元数据属性，来给数据块打标签。如果加日期标签，可以控制只访问某时间段的数据。

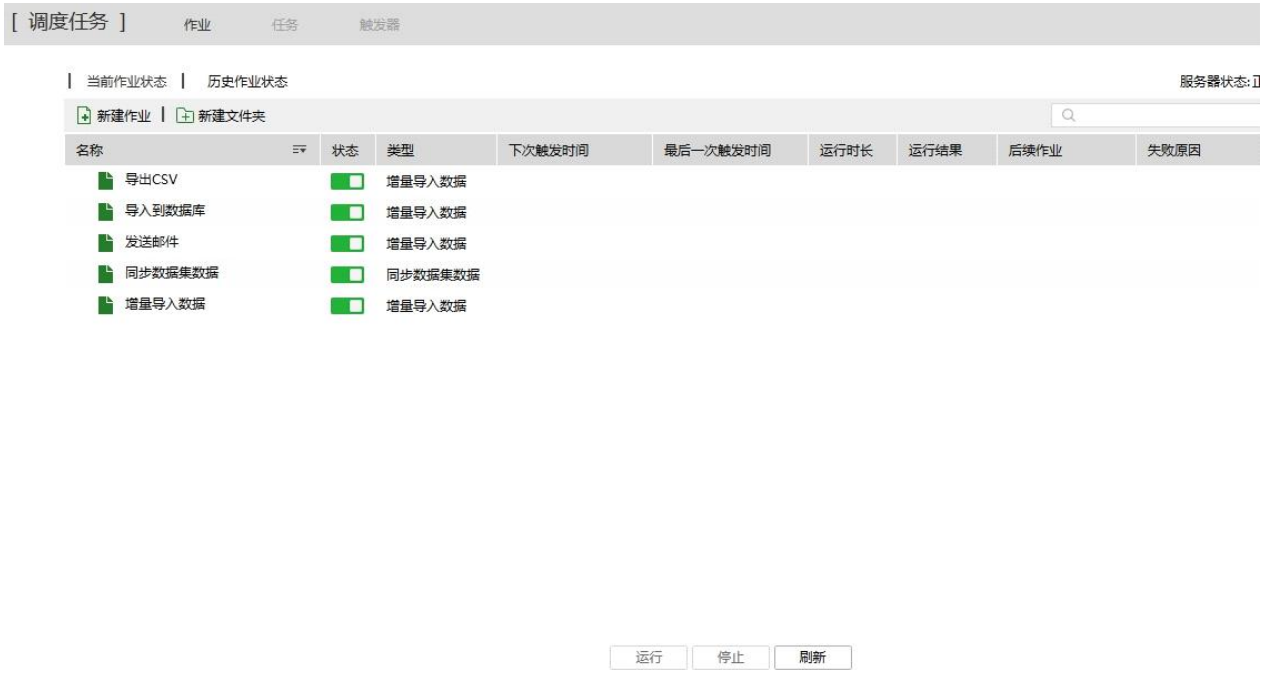
关于调度任务的具体说明，请参见相关文档。

- 手动方式

Z 产品提供了一些 API 接口来访问数据库，并读取数据，然后生成压缩后文件。数据集市系统提供了一些 API 管理命令。包括 AddFolderTask （将压缩后的物理文件，新加入一个 Folder 到云系统中），RemoveFolderTask（删除某个 Folder）， RemoveGSFileTask （删除某个文件）。

❖ 任务调度器

进入调度任务列表界面，用户可对作业列表中的作业进行监控，也可开始或暂停某一作业。

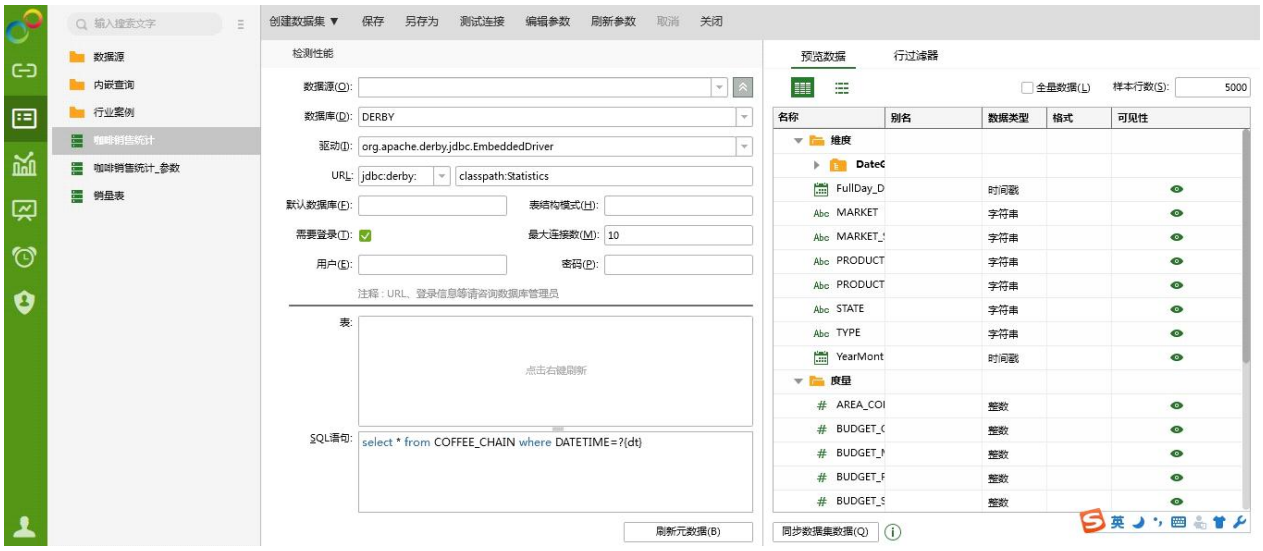


❖ 增量更新

用户通过设定更新条件，常见根据日期来更新，例如每天零点来进行数据更新，也可以按照其他合理规则来更新。

➢例如

在 SQL 语句中通过 where 设定过滤条件，接收日期参数。



在任务计划中传递参数值，把当天的日期传递给 SQL 中的参数，增加当天的数据到数据集市中。

❖ 加入标签

在调度任务中，新建作业，任务类型为增量导入数据，选择一个数据集，在脚本中通过 setMeta() 方法给入库的云文件加入标签，如下图脚本中所示。

关于按列分割，并行导入集市的功能，当用户勾选分割时，会自动弹出分割对话框。分割类型分为平均分割和分组分割，平均分割只能选择一个分割列，分割份数为整数，如：3。分组分割通过 Group By 进行分组，不能填写分割份数，但可以选择多个分割列。为了不影响导入数据的效率，建议分割列的列数不超过 10。选择分组分割，入集市的数据会自动按分组数据打 Meta，以方便对云文件夹中的云文件进行过滤。 Meta 中的 key 为分割列对应的列名 ,Meta 中的 value 为分割列对应的值。当用户不勾选分割时，云文件会按系统默认的设置进行生成。

➢**注意**：择分组分割，入集市的数据会自动按分组数据打 Meta，但是必须满足以下三个条件：

A. 数据的总行数 > dc.unit.rows (默认为 262144)

B. 分组的份数 <= dc.split.range (默认为 1000)

C. （数据的总行数 / 分组的份数） > dc.unit.rows (默认为 262144)

❖ 多点同时入库

多个 Client 节点可以同时执行 Scheduler 把数据抽取入数据集市。例如各个 Client 节点分别把数据提取进入数据集市的同一云文件夹中，彼此不受影响。

当任务类型为增量导入数据并且运行成功时，在结果＂成功＂后出现可以触发任务运行结果对话框的图标。在当前作业状态页面中，允许用户删除最后一次成功导入到集市中的数据，而保留之前导入的数据。如果用户通过追加的方式，连续多次将数据导入到集市，其中第二次导入到集市中的数据有误，那么用户可以在历史作业状态页面找到对应的作业执行记录，打开任务运行结果对话框，删除此次导入到集市中的数据，而保留其他数据。

3. 如何使用集市中的数据

本章节主要从以下几个方面介绍如何使用集市中的数据。

- 应用的模式。
- 文件系统。
- SQL 支持。
- ROLAP 支持。
- 如何使用标签。

❖ 应用模式

将大数据封装成 Yonghong BI 的 Query。

将大数据平台封装成集市。 SQL 语言，翻译成 ZMR。

将大数据平台封装成 OLAP Server。 OLAP 语言，翻译成 ZMR。

❖ 文件系统

关于 Folder：Folder 是一个文件集。只有 Naming Node 上才有 Folder。

关于 File：File 是某个文件集中的一个文件。 File 有三种形态：

1. Client Node 上的 File: 这时的 File 将被 Client Node 加入云系统中。
2. Naming Node 上的 File: 这时的 File 是元数据。这个元数据也在内存中保存了哪些 Map/Reduce Node 上存储有该 File 的信息。
3. Map/Reduce Node 上的 File: 这时的 File 是元数据，但也保存着指向物理文件的指针。

- 文件集

只有 Naming Node 上才存储有 Folder， Folder 完全是元数据，由一个或者多个 File 组成。

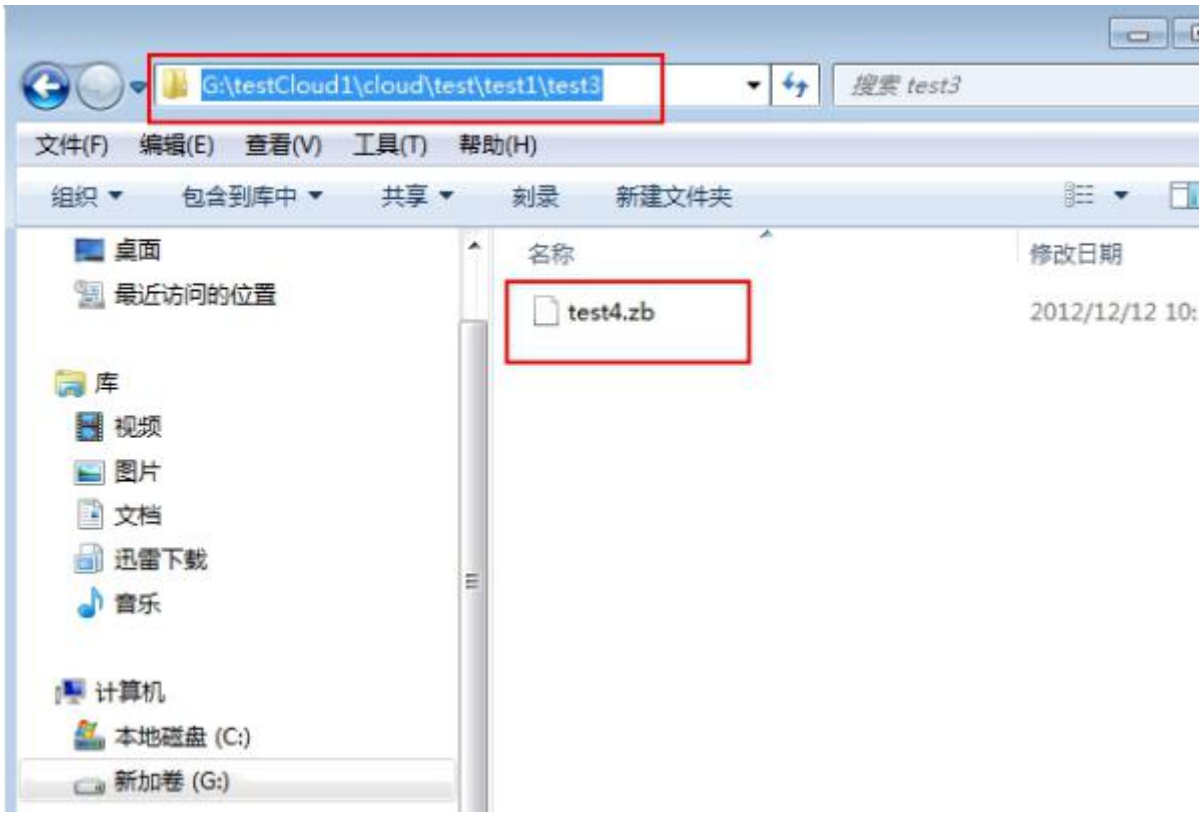
- Folder 的名称可以包含 '/'

为什么要这样呢？这是因为 Folder 的每一个 File 的全名就是 Folder_Name/File_Name，在存储物理文件的时候，如果 Folder_Name 可以包含 '/'，那么可以形成下例所示的全名：sales/product/file_1，其中 sales/product 是 Folder_Name。这样一来，存储物理文件就不限于两级目录而可以是多级目录。在操作系统中，如果同一目录下的文件太多，访问效率将大大降低。我们的文件系统支持多级目录机制，主要就是想避免这一问题。

➢例如

在任务计划中执行云任务时设定云文件夹的名称为 test/test1/test3，云文件的名称为 test4，如下图所示。

此任务执行完后，将在指定的路径中（在 root/abc@123 中指定的 dc.fs.physical.path=G\:/bihome/ cloud，详细介绍如何搭建本地数据集市系统）生成云文件夹以及云文件。



- 文件

Naming Node 上的 File。作为元数据， Naming Node 上的 File 存储在 Folder 中，由于 Map/Reduce Node 不断发送心跳报告， Naming Node 将能接收到某个 File 的物理文件在哪些 Map/Reduce Node 上存储。

Map/Reduce Node 上的 File。 Map/Reduce Node 上的 File，是一个元数据文件，并保存着指向本机上的物理文件的指针。 Map/Reduce Node 定期地向 Naming Node 发送心跳报告，以告诉 Naming Node 该 Node 可用，并给出存储于本 Node 上的 File 信息。

- GMeta 介绍

Naming Node 上的 File 有一个 GMeta，记载本文件的一些摘要信息。

1.File 上的 GMeta 存储的摘要信息可以由用户自定义。例如，用户可以定义这个文件存储数据的日期 2012-6-30。另外，系统会自动加上文件名作为其中的一项摘要信息：_FILE_NAME_。

2. Folder 存储于 Naming Node 上，也有一个 GMeta 记载文件夹的一些摘要信息。

同理， Folder 上的 GMeta 存储的摘要信息也可以由用户自定义。系统自动会加上两个方面的内容：

Folder 所对应的 Data Grid 的 Columns ；以及任意子 File 上的 GMeta 上存储了哪些可用的摘要信息，同样以 Columns 的形式保存。

- GMeta 的功用

GMeta 上存储的摘要信息，在数据集市系统中发挥着重要的作用。

➢例如：

当用户创建一个数据集市数据集的时候，需要知道有哪些 Columns 可以使用。这个信息可以从 Folder 的 GMeta 中得到，因为 Folder 的 GMeta 存储着这些信息。

当用户创建一个数据集市数据集时，他 / 她可能不想基于 Folder 下面所有的 File 来运行查询。因为很多时候这样去运行数据集没有必要，却非常消耗资源。这时，数据集市数据集可以定义 File Filter 来限制需要访问的 File。这个 File Filter 将基于 Naming Node 的 File 上的 GMeta 来运行，直接找出 Query 需要访问的那些 Files，这样能很大地提升数据集的运行性能，并减少资源消耗。

由于 Folder 上以 Columns 的形式保存了子 File 上的 GMeta 上存储了哪些可用的摘要信息，在定义 File Filter 的 GUI 上，数据集市系统返回这些可用的 Columns 给用户来定义 File Filter。

- 集市数据迁移

为了方便集市数据的备份和迁移，减少 meta 文件和物理文件不一致的情况，从 7.5 版本开始 Naming Node 以及 Map Node 上的 meta 存储方式发生了变化。原先所有集市文件夹的 meta 信息都会存储在 qry_naming.m 文件里，现在拆分存储，系统会为每个 GSFolder 创建一个同名文件夹来记录该 GSFolder 的 meta 信息，该同名文件夹和 qry_naming.m 文件处于同级目录。对于集市文件的 meta 信息，原先是单个 Map Node 上的所有 meta 信息都存储在 qry_sub.m 文件里，现在是直接将 meta 信息存储在集市文件本身。进行数据迁移或备份时，只需要迁移数据本身与该 GSFolder 对应的单个 meta 文件即可。

➢**说明**：对应的 meta 信息在分散存储的同时，仍然会存入 qry_naming.m、qry_sub.m 中。

➢**注意**：迁移集市数据到新的环境后，数据本身、单个 meta 文件所处的相对路径应该和迁移前保持一致。比如：迁移前 meta 文件所处的相对路径为 bihome/cloud/咖啡销售统计/东部市场，迁移后所处的相对路径应该还是 bihome/cloud/咖啡销售统计/东部市场。

- 状态

不管是 Folder， Naming Node 上的 File，还是 Map/Reduce Node 上的 File，都有一个状态标注其可用还是不可用。

Folder 的 Active: 如果 Folder 不是 Active 的，那么基于这个 Folder 的 Query 将无法运行，也不会检查这个 Folder 是否需要自动修正，或者进行自动修正。

Naming Node 上 File 的 Active: 如果 File 不是 Active 的，那么基于其 Folder 的 Query 运行时将看不见这个 File，系统也不会检查这个 File 是否需要自动修正，或者进行自动修正。

Map/Reduce Node 上 File 的 Active: 如果 File 不是 Active 的，那么心跳时这个 File 将不可见。

- 自动管理

在 Naming Node 上，有一个专职医生 Doctor 对整个云系统进行自动管理。

有这样一些错误将进行纠正：

不匹配的 File。例如，某个 Map Node 心跳报告中包含一个 File，这个 File 的版本号已经过期，那么 Doctor 将让这个 Map Node 删除掉这个不匹配的文件。或者这个 Map Node 所对应的 Folder 已经不复存在，那么 Doctor 也会让这个 Map Node 删除掉这个没有作用的文件。备份过量的 File。例如， Naming Node 上的某个 File 备份数为 3，但系统定义的合理备份数为 2，那么 Doctor 将选择多于的备份删除。其删除算法将首先选择那些负担比较重的 Map/Reduce Node。

备份不足的 File。例如， Naming Node 上的某个 File 备份数为 2，但系统定义的合理备份数为 3，那么 Doctor 将选择一个负担较轻的 Map/Reduce Node 来备份该 File。

- 手工管理

数据集市系统提供了一些 API 管理命令。

AddFolderTask：这个 Task 将新加入一个 Folder，或者追加新 File 到已经存在的 Folder 中。

RemoveFolderTask：这个 Task 能删除某个 Folder。

RemoveGSFileTask：这个 Task 能删除某个 File。

QueryTask：这个 Task 能 Query< 读写 > 某个 Host 上的某个 Node 一些信息。目前支持以下子功能：

Naming Node: active，查询数据集市系统是否可用。

Naming Node: activateFolder <folder>，让某个 Folder 可用。

Naming Node: deactivateFolder <folder>，让某个 Folder 不可用。

Naming Node: activateFile <file>，让某个 File 可用。

Naming Node: deactivateFile <file>，让某个 File 不可用。

Naming Node：existsFolder <folder>，查询某个 Folder 是否存在。

Naming Node：folders，查询存在的 Folders， Folder 之间用符号 "_sep_" 隔离。

❖ SQL 支持

基于 JDBC 的方式来访问，就需要对 SQL 语句的支持。这里采取都是 SQL92 中的大部分语句。

函数	说明
语句块	Select, Alias, From, Where, Parameter, Group By, Order By
常见函数	Count, Count(Distinct A), Sum, Max, Min, Avg
特殊函数	Range, Product, Median, Quartile, Mode, SumSQ, PthPercentile, Variance, PopulationVariance, StandardDeviation, StandardError, PopulationStandardDeviation, SumWT, WeightAvg, Covariance, Correlation

函数	说明
四则运算及函数	Sqrt, Sqr, ParseInt, ParseFloat, Abs, Substring, (a+b-c)*b/c
日期函数	Year, Quarter, Month, Week, Day, Hour, Minute, Second, Quarterpart, Monthpart, Weekpart, Daypart, Weekday, Hourpart, Minutepart, Secondpart
Join	不支持，由于基于主题的数据集市，Join 会影响性能
SubQuery	暂不支持

❖ ROLAP 的支持

维度的范围有大小的概念，例如国家的范围大，省的范围次之，城市的范围更小。可以把范围的大小的概念称之为层次。在维度节点下建立层次目录，把有范围大小的维度通过拖拽放入层次中。范围大的放在最上面，范围小的维度放在下面。维度的顺序，决定了钻取的顺序。当需要上钻时，会找到与当前维度最近的上一个维度（即范围大点的维度）。当需要下钻时，会找到与当前维度最近的下一个维度（即范围小点的维度）。

➢例如

某一 SQL 数据集中存在年、季度、月份字段，三个字段之间有范围大小的概念，年大于季度，季度大于月份，如下图所示。

名称	别名	数据类型	格式	可见性
维度				
Abc 季度		字符串		
Abc 年		字符串		
Abc 月份		字符串		
度量				
# 人均消费		双精度浮点		
# 月收入		单精度浮点		

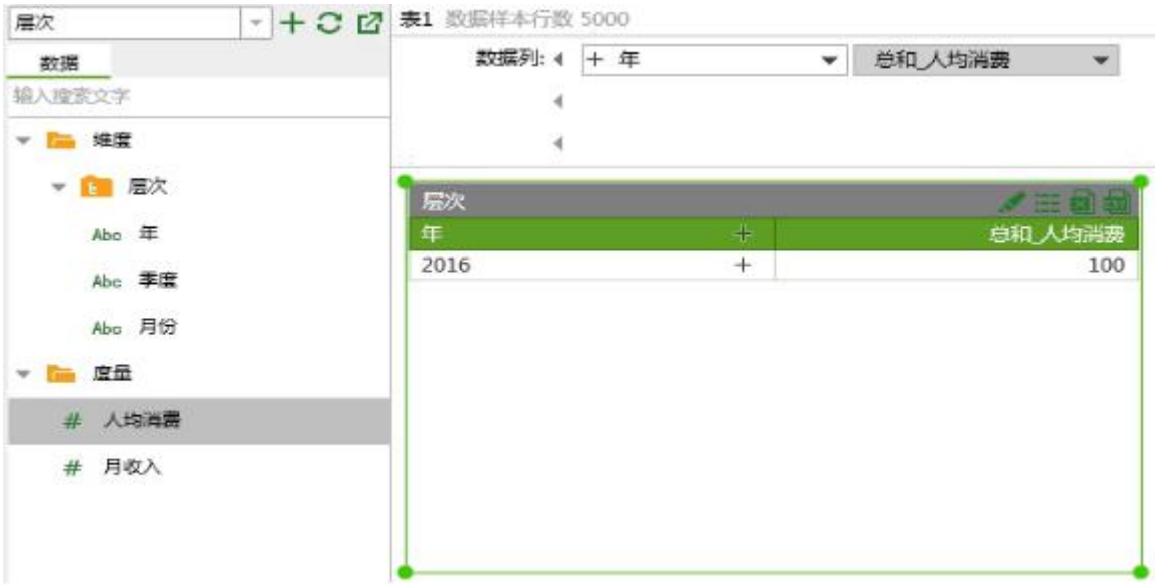
用户在元数据区域右键选择新建层次，如下图所示。

在弹出的对话框中输入层次名称，点击确定按钮则在元数据区域生成层次文件夹。

用户通过鼠标拖拽来把具有层次的字段放到层次文件夹中，如下图所示。

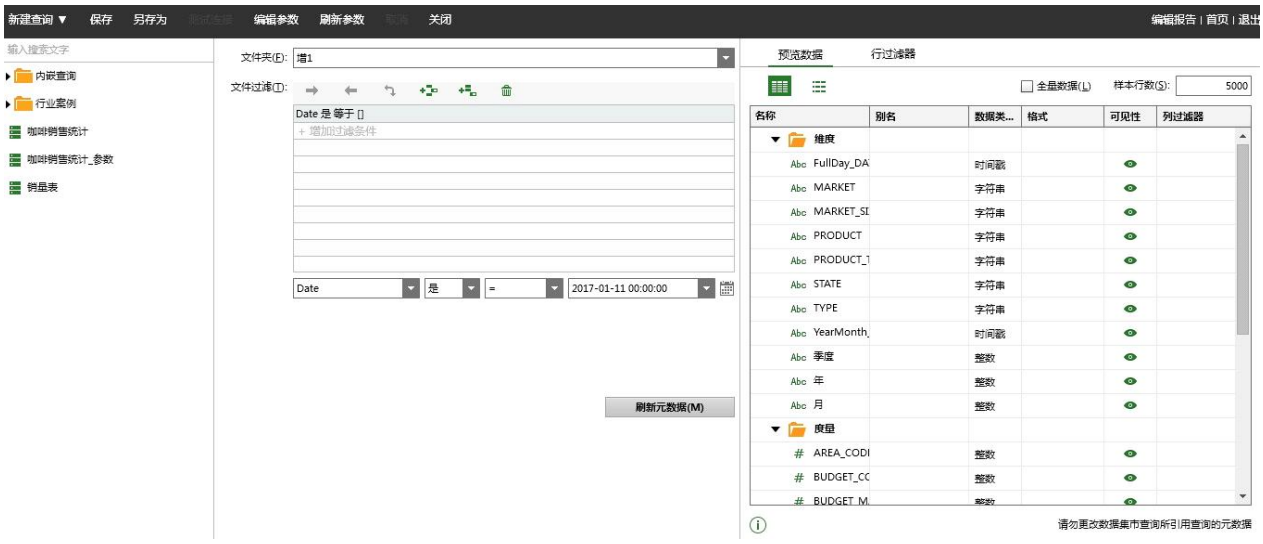
被拖拽到层次文件夹中的字段仍可通过鼠标的拖拽来调节位置。在上的字段范围较大，如下图所示。

当用户在报表编辑区中绑定该层次文件夹中的字段时，会在范围较大的数据段的右下角生成加号，当用户点击此加号时，比当前数据段的范围小一级的数据段将被自动绑定，同时加号变成减号。



❖ 如何使用标签

在创建数据集模块中新建数据集市数据集，选择云文件夹，在文件过滤区域鼠标右击选择创建过滤器，如下图所示在红圈内选择设定的标签，然后再选择标签值。



➢**例如**：在执行云任务时，给云文件打入标签 “Date”，标签值为当天的日期。在数据集中建模时即可根据标签值来过滤指定时间范围内的数据。

❖ Map Side Join

在分布式系统中，当有星形数据（一个大表，若干个小表）需要 join 的时候，可以将小表的数据复制到每个 Map 节点，执行 Map Side Join，而无须到 Reduce 节点进行连接操作，从而提升表连接的效率。

Hadoop 中使用 DistributedCache 来实现 Map side join。它可以将小文件分发到各个节点上，在连接的时候将小文件导入到内存中使用。Spark 在 Hadoop 的基础上，使用 Broadcast 来实现 Map side join。Broadcast 文件分发的效率要明显好于 DistributedCache，因为它采用更优化的文件广播算法，包括 P2P 算法等。

在 MPP 集市中，我们可以定义维度表，即 join 操作中的小表。在用户导入维度表的时候，将维度表分发到每个节点上，如下图所示：

当新加节点时，也需要将现有的维度表分发到该节点上。

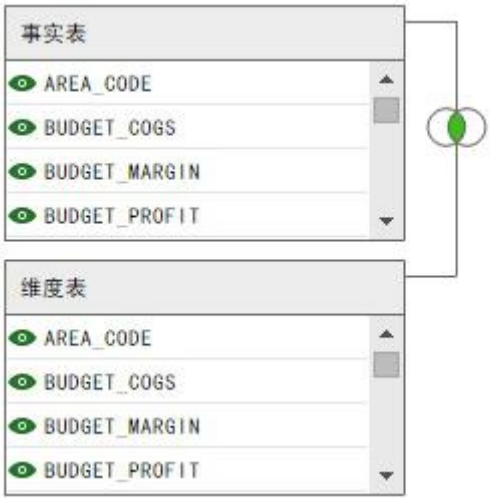
当执行数据集市数据集时，可以进行 Map Side Join 的条件是：

（1）组合数据集，自服务数据集或者是 Yonghong 的 SQL 数据集。

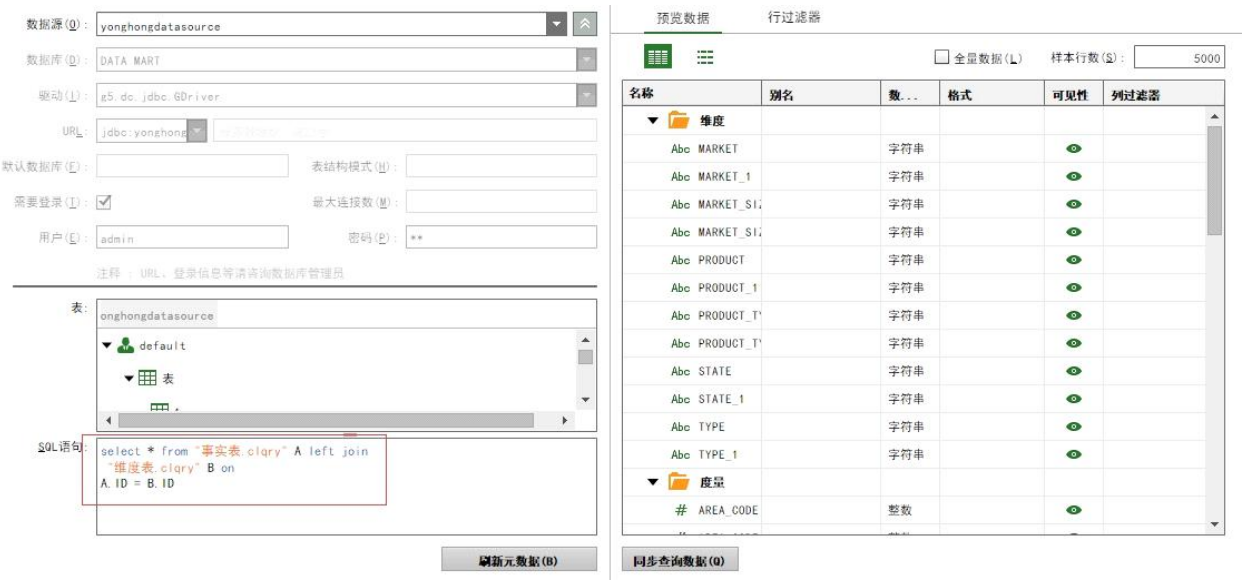
（2）Join 操作中，必须符合星型数据，且小表是维度表（要求 Join 操作中，所有表中有且只有一个表是非维度表）

Naming 节点部署 Map Task 的时候，优先将任务分配给本地已有维度表的节点，并将需要的维度表信息发送到相应的 Map 节点。如果该 Map 节点没有对应的维度表信息，则请求 Naming 节点获取维度表，存在本地。

其中组合数据集和自服务数据集使用联接操作符（内部联接，左侧联接，右侧联接，外部联接）实现 Join, 组合数据集如下图所示 ：



Yonghong 的 SQL 数据集实现 MapSideJoin 需要创建 Yonghong 的 SQL 数据集，在数据集中输入如下 SQL 语句可以实现事实表和维度表的 join，如下图所示：



4. 集市管理系统

集市管理系统（MPP Console）是 Yonghong Z-Suite 一个独立的功能模块。它被用来管理 Yonghong MPP 数据集市。可以监测集市中各台机器的 CPU、内存、磁盘的使用情况和网络速度；监测各个节点的 JVM （Java Virtual Machine）状态；可以修改全局参数和每个节点自己的参数；可以启动、停止、重启节点；可以更新集市里各个节点用到的 jar 文件。

集市管理系统模块可以通过任务计划模块制定周期性的信息汇报计划，通过生成 PDF 文档，以邮件的方式发送到用户手中。

❖ 启动集市管理系统

Yonghong MPP 集市中，每一个节点都需要通过集市管理程序 (Console) 来启动集市管理系统，它是独立于 MPP 集市的一套管理系统，这样即使集市的某个节点停止了，这套集市管理系统也依旧能正常运行并检测到集市中各个节点的异常情况。

启动 Console 并进入集市管理系统的步骤：

1. 配置数据库属性（必须）

在 Naming 节点安装目录 /manager/bin 下的 console.properties 文件中，配置数据库连接信息，这个数据库用来写入集市中各台机器的 CPU，内存，磁盘网络信息，和各个节点的 JVM 信息。

```
console.db.url=jdbc:mysql://192.168.1.104:3306/test
console.db.driver=com.mysql.jdbc.Driver
console.db.user=root
```

console.db.passwd=yonghong4

console.db.schema=test

console.db.vendor=mysql

2. 启动集市管理系统

2.1 启动 Console Manager:在 Naming 节点上，运行安装目录 /manager/bin 下的 startConsole.sh 命令来启动 Console Manager。

2.2 启动每个节点的 Console:在数据集市的每台节点上，运行安装目录 /console/bin 下的 startConsole.sh 命令来启动 Console。

对于 Naming 节点，需要在 /console/bin/console.properties 配置 console.port.offset=1, 保证 Console Manager 和 Console 的端口不会冲突，两者能同时启动。

3. 进入集市管理系统

从首页 -> 管理系统 -> 集市管理页面，可以看到页面左边的仪表盘分为三个部分：

- 服务器管理
- 集市管理
- 节点管理下面分别对这三个部分进行介绍。

❖ **服务器管理**

- 使用资源

此仪表盘用来查看集市中各台机器资源使用情况，包括 CPU 使用率，内存使用率，磁盘使用率，网络速度。

通过第一张表可以查看整个集市的资源情况。

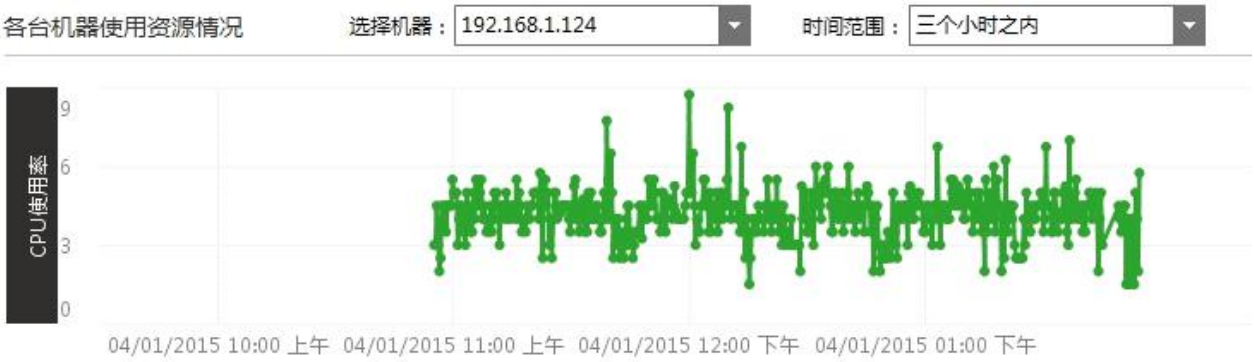
机器名称	CPU使用率	内存使用率	磁盘使用率	网络速度
192.168.1.121	0.68%	39.77%	3.52%	0.29
192.168.1.123	0.13%	26.98%	29.73%	1.57

CPU，内存，网络速度这三项的数据收集间隔为默认为 20 秒。

磁盘数据的收集间隔默认为 10 分钟。

可以在配置文件 /console/bin/console.properties 里通过属性 timer.interval.cpu，timer.interval.mem， timer.interval.net 和 timer.interval.disk 修改 CPU，内存，网络和磁盘数据收集的时间间隔。用户配置的
时间间隔需大于系统所设定的默认值，否则按照默认值的时间间隔来收集数据。

也可以通过表下方的图表查看单个机器的资源使用趋势，如 CPU 使用率：



❖ 集市管理

• 共享文件部署

每个节点在启动的时候，都要用到 product-swf.jar 和 product.jar 两个 jar 包，如果这两个包需要在所有节点上更新，可以在这个仪表盘里做到。

如下图，点击 “ 上传 ” 链接，

共享文件			
文件名	文件路径	删除	上传
product-swf.jar	PRODUCT_HOME/Yonghong/	删除	上传
product.jar	PRODUCT_HOME/Yonghong/	删除	上传

弹出 “ 导入资源 ” 页面，点击 “ 浏览 ” 选择文件，点 “ 确定 ” 即可上传成功。

从jar文件导入资源。

上传成功后，可以从下方的表 “ 上传文件结果 ” 中查看文件同步上传到各个节点的情况。

文件上传结果

文件名	节点名称	状态	时间
product-swf.jar	192.168.1.121	成功	2015-03-03 10:32:03
product-swf.jar	192.168.1.123	成功	2015-03-03 10:32:04

也可以往 bihome/ 和 PRODUCT_HOME/Yonghong/ 两个目录及对应的子目录下新增文件。

如下图，输入要新增的文件名，选择目录，也可以输入子目录名称，点击 “ 增加 ” 按钮，新增的文件名将出现在下方的表 “ 共享文件 ” 中，此时可以点击新文件名后的 “ 上传 ” 链接，即可以上传需要新增的文件。

- 集市参数

此仪表盘用来修改 MPP 集市的全局参数，即 global_bi.properties 中的参数。

- 点击单元格全局参数值修改参数内容
- 右键菜单删除行来删除某个参数
- 右键插入行来新加某个参数

修改完后，点击 “ 提交修改 ” 按钮来完成修改。

修改成功后，可以从下方的表 “ 参数修改结果 ” 中查看参数同步修改到各个节点的情况。

参数修改结果

节点名称	时间	状态
192.168.1.121	2015-03-04 13:08:43	成功
192.168.1.123	2015-03-04 13:08:43	成功

目前集群配置中，每个节点都有一个 global_bi.properties 配置文件，默认同一集群中的所有节点的 global_bi.properties 配置文件是一致的。但是在一些情况下，如用户手动修改了某一或者多个节点的 global_bi.properties 配置文件时，集群中节点的 global_bi.properties 配置文件会出现不一致，不正确的情况，有可能导致没有正确配置的节点出现异常，而用户无法感知集群中各节点的参数配置是否一致。

因此在集市管理部分加入了检测 lobal_bi.properties 配置的功能，能够让用户感知到各节点参数配置不一致的问题，进而可以统一修改 global_bi.properties 配置。

如果全局参数配置不同的话，会在全局参数值的单元格内显示 “ 不一致 ” 的字样，用户可根据详细参数值来确定该参数在不同节点所配置的值，进而可以通过修改参数功能将该全局参数在所有节点上更改为一致的参数值。

- 集市状态

此仪表盘用来查看集市里各个节点的状态，同时能对节点进行启动、停止和重启工作。

通过选择节点状态或节点类型，过滤出需要查看的节点，并对其进行启动、停止或重启的操作。

MPP 集市系统中一个节点对应一个 console，当节点上的 console 启动后，会监控每个节点的工作状态，当节点宕机后， console 可以每隔一定的时间间隔将节点自动启动起来，默认时间为 30s，可以通过属性：node.start.delay=30 设置不同的时间间隔。

❖ 节点管理

• 使用资源

此仪表盘用来监控各个节点（除了客户端节点）的 JVM 使用情况。

JVM 的收集间隔默认为 2 分钟，可以在安装包 /console/bin/console.properties 里通过属性 timer.interval.jvm 修改 JVM 数据的收集间隔。

• 运行参数

此仪表盘用来查看和修改单个节点的参数。

选择节点后，下面的表列出了这个节点对应的 bi.home/bi.properties 里的所有参数。

- 点击单元格修改参数内容
- 右键菜单删除行来删除某个参数
- 右键插入行来新加某个参数

参数名	参数值
.level	FINE
db.mode.config	mode.properties
dc.fs.naming.paths	/opt/YH/Yonghong/bihome/cloud/qry_naming.m
dc.fs.physical.path	/opt/YH/Yonghong/bihome/cloud
dc.fs.sub.path	/opt/YH/Yonghong/bihome/cloud/qry_sub.m
dc.global.path	/opt/YH/Yonghong/bihome/global_bi.properties
dc.io.ip	192.168.1.121
dc.node.types	nc
license	P+63p42o/NtPWdsG4CEUd1w-L85-V5.0-C4-U5-O5-D31edf3f
secure.authentication	g5.secure.fs.FSAuthentication
secure.authorization	g5.secure.fs.FSAuthorization
yonghongtech_guard1	1405431664291
yonghongtech_guard2	1425477162271

修改完后，点击 “ 提交修改 ” 按钮来完成修改。

❖ 属性配置说明

console.manager：必填属性，指定 Console Manager 的 IP， Console Manager 为 Naming 节点。

console.io.ip：必填属性，指定启动 Console 的节点 IP。

console.node.home ： 必 填 属 性 ， 指 定 节 点 的 bihome ， 如 ： /home/MR/Yonghong/bihome。

console.node.path：必填属性，指定节点的工作路径，如：/home/MR/tomcat/bin。

console.port.offset：端口偏移量，如果一台机器启动多个 Console，需要指定偏移量。

console.os：指定操作系统 linux。

collect.server.resource：如果是 true， Console Manager 会保存 JVM 数据到指定数据库， Console 会保存节点 CPU、内存、网络和磁盘信息到指定数据库。反之则不会保存。默认为 true。

timer.interval.cpu：收集 CPU 信息的时间间隔，默认为 20 秒。

timer.interval.mem：收集内存信息的时间间隔，默认为 20 秒。

timer.interval.net：收集网络信息的时间间隔，默认为 20 秒。

timer.interval.disk：收集磁盘信息的时间间隔，默认为 10 分钟。

timer.interval.jvm：收集 JVM 信息的时间间隔，默认为 2 分钟。

resources.save.days：数据库保存信息的天数，数据库中超过设置时间的记录会被删除，默认为 3 天。

node.start.delay=30：MPP 集市节点宕机自动启动的时间间隔，默认为 30s。

5. 配置和调优

本章节主要从以下几个方面介绍集市的配置和调优。

- 硬件要求。
- 常见配比关系。
- 常见配置属性。

❖ 硬件要求

入门级服务器数台到数十台。每台服务器内存 32G 以上， CPU 4 Processor 以上，独立硬盘 1TB 以上，千兆或万兆网卡。

集市搭建后，可以做到热插拔的方式横向扩展。

❖ 常见配比关系

每个 CPU Core 可以承担 2000 万行数据的实时计算。	
描述	按每条数据 1000 Byte 计算，2000 万条数据差不多是 20G 的原始数据，假设 20G 原始数据的压缩比为 1/3，而内存装载的数据是 1/3，内存里的数据差不多是 2G。
越多的 CPU，系统性能越好。	
描述	在 MPP 数据集市上，CPU 的扩展与性能的改善，基本上是线性相关的。由于基于内存计算，需要内存配合。
计算 CPU 时，还需要考虑到并发量。	
描述	在线用户数的多少、以及在线用户提交的并发查询的多少，对 CPU 和内存的要求也会不同。我们需要求出每秒执行的查询数，才能估算出并发量对 CPU 和内存的要求。一般来说，并发的查询越多，对 CPU 的要求就越高，热数据很可能会上升，而临时内存需求也会轻微上升。业界估算的公式一般是每个并发在原有基础上追加大约 5% 的内存和 CPU。
充足的内存可以减少数据交换，对性能的提升也有帮助。	
描述	每个系统的热数据量不一定，有的是 1/5，有的是 1/3，有的是 1/10。当内存足够多，所有的热数据都保持在内存，这之后再增加内存，对系统的性能就没有太大的作用了。
一般情况下，原始数据与内存量的比率可以定义为 10:1，而原始数据与 CPU Core 的比率可以定义为 20G：1。这样的配置情况下，查询速度可以达到秒级响应。	
描述	1T 的数据量，需要 100G 内存，50 个 CPU Core。可考虑如下配置：四台 2 通道 PC Server, 每台 2 个 CPU，且每个 CPU 有 6 Core，每台内存 24G。
	500G 的数据量，需要差不多 50G 内存，24 个 CPU Core。可考虑如下配置：三台 2 通道 PC Server。每台 2 个 CPU，且每个 CPU 有 4 Core，每台内存 16G。

❖ 常见配置属性

一个分布式数据集市系统中有两种配置文件，分别是本机属性配置 (Local Properties) 和全局属性配置(Global Properties)。

本机属性配置文件是每个机器节点必须配有的属性文件。默认存储在{bi.home}/bi.properties. 其中 bi.home 路径是指相对安装路径的 YH\Yonghong\bihome 目录。

mem.serial.mem=700 定义可分批给内存计算的内存块大小，单位为兆（M）。

mem.proc.count=2 定义可用来做内存计算的 CPU 个数。

全局属性配置文件是存储了所有机器群共享的属性文件。默认存储在{bi.home}/global_bi.properties. 其中 bi.home 路径是指节点的安装路径 YH\Yonghong\bihome 目录。

dc.rows.max=20000 定义结果集返回的最大行数 dc.io.timeout=15000 定义两个机器节点之间通讯的最大等待时间。

dc.task.timeout=180000 定义一个任务完成的最大时间。如果超出这个时间还未完成，系统将试着重新分配任务。

dc.doctor.repair=false 定义是否需要恢复丢失的文件

- 常见调优手段

-Xms 初始堆大小

-Xmx 最大堆大小

-Xmn 年轻代大小 (1.4or lator)

CMSInitiatingOccupancyFraction=70 使用 cms 作为垃圾回收，使用 70%后开始 CMS 收集

NewRatio 年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）

6. 详细配置参数

一个分布式数据集市系统中有两种配置文件，分别是本机属性配置 (Local Properties) 和全局属性配置(Global Properties)。

❖ **本机属性配置**

本机属性配置文件是每个机器节点必须配有的属性文件。默认存储在{bi.home}/bi.properties. 其中 bi.home 路径是指相对安装路径的YH\Yonghong\bihome 目录。

属性	可选/必选	说明
dc.io.handlers=1	可选	定义处理 IO 通讯的线程数。一般情况下，一个线程足够了。
dc.io.channels=2	可选	定义与其它机器节点通讯时，最大 Socket 连接数。
dc.io.ip=	必选	定义本机的 IP，尤其在多网卡的时候。如果未定义，将试着从操作系统得出 IP。
dc.node.types=mr	必选	定义本机的机器节点类型， 其中 m - Map Node, r - Reduce Node, n – Naming Node, c - Client Node。一般是这些值的组合。
dc.global.path=global_bi.	必选	定义各个机器节点共享的配置文件路径。

属性	可选/必选	说明
properties		
mem.serial.mem=700	必选	定义可分批给内存计算的内存块大小。单位为兆（M）
mem.proc.count=2	必选	定义可用来做内存计算的 CPU 个数。
dc.block.units=4	可选	定义一个数据块中数据单元的个数。这个数据块是内存装载或者从内存卸载的物理文件，分发到各个 Map 节点当中。
dc.unit.rows= 262144	可选	定义一个数据单元的行数。这个数据块将形成为一个物理文件，分发到各个 Map 节点当中。
dc.fs.naming.paths=	必选	定义命名节点存储元数据的文件路径，这里的文件路径可以是多个，以';' 隔开。这样元数据文件有更高的安全性。请输入绝对路径。默认值是{bihome}/cloud/cloud/qry_sub.m
dc.naming.waiting=30000	可选	定义启动 Naming Node 之后，至少等待多长时间才能切换到可用状态。
dc.naming.maps=1	可选	定义启动命名节点之后，至少有多少个活着的 Map 节点才能切换到可用状态。
dc.naming.reds=1	可选	定义启动命名节点之后，至少有多少个活着的

属性	可选/必选	说明
		Reduce 节点才能切换到可用状态。
dc.naming.check.file=true	可选	定义启动命名节点之后，要不要确保元数据正确之后才切换到可用状态。所谓正确的元数据，是指这些元数据包含的文件夹和文件都是可用的。
dc.fs.sub.path=	必选	定义 Map 节点或 Reduce 节点存储元数据的文件路径。请输入决定路径。默认值是默认值是{bihome}/cloud/cloud/qry_sub.m
dc.fs.physical.path=	必选	定义 Map 节点或 Reduce 节点存储物理数据的文件夹。请输入决定路径。默认值是默认值是{bihome}/cloud/cloud
dc.col.cache.count=20	可选	定义每种列存储类型的最大内存缓存个数。
dc.data.debug=false	可选	定义是否输出数据的调试信息。
dc.inverted.supported=false	可选	定义是否尝试生成列索引，以加快性能。
dc.inverted.ratio=3.1	可选	定义在尝试生成列索引的时候，平均每行的索引大小。
dc.buf.cache.count=10	可选	定义通信使用的数据缓冲区的缓存个数

属性	可选/必选	说明
dc.float.frags=4	可选	单精度浮点数入集市保留的小数位数。
dc.double.frags=4	可选	双精度浮点数入集市保留的小数位数。
dc.mr.debug=false	可选	执行 Map, Reduce 任务的时候，每隔 20 秒打印 Map， Reduce 的执行情况。
dc.orderby.limit=500000	可选	支持排序的最大分组数。
map.aggr.parallel=false	可选	是否在 Map 端对一个 zb 文件按分片、Hash 分区并行处理。
red.aggr.parallel=true	可选	是否在 Reduce 端按 Hash 分区并行处理。
map.part.size=4	可选	Map 端 hash 分区的个数。
red.part.size=32	可选	Reduce 端 Hash 分区的个数。
aggr.timeout=600000	可选	并行处理等待相关线程结束处理的超时时间。
parallel.min.groups=10000	可选	Reduce 端最小需要并行的分组数。

其中标有可选属性在系统中有默认值，默认值等于第一列描述中的符合后面的结果。

❖ 全局属性配置

全局属性配置文件是存储了所有机器群共享的属性文件。默认存储在{bi.home}/global_bi.properties. 其中 bi.home 路径是指节点的安装路径 YH\Yonghong\bihome 目录。

属性	可选/必选	说明
dc.io.local=true	可选	标注是单机还是多机版。默认是本地单机版。
dc.cache.max=5242880	可选	定义最大的内存缓存，超过这么多的数据被读入/ 写出，将发起至少一次物理读入/写出。
dc.io.timeout=15000	可选	定义两个机器节点之间通讯的最大等待时间。
dc.io.block=131072	可选	定义 Socket 读写的缓存大小。
dc.io.sport=5083	可选	定义各个机器节点之间通讯的端口。
dc.io.fport=5066	可选	定义各个机器节点之间传输文件的端口。
dc.node.naming=	必选	定义命名节点（Naming Node）的 IP，如果是本地单机版，则无需定义。
dc.fs.dup=2	可选	定义文件系统的复制份数。
dc.update.period=1500	可选	定义心跳的周期。每个心跳周期，Map/Reduce 节点将发出一份报告给命

属性	可选/必选	说明
0		名节点（Naming Node），申明自己的存活。
dc.task.timeout=60000	可选	定义一个任务完成的最大时间。如果超出这么长时间还未完成，系统将试着重新分配任务。
dc.nodes.pin=	可选	定义各个机器节点之间通讯时所用的 Pin 码。如果 Pin 为空，则不会检查 Pin。默认为空。
dc.doctor.repair=false	可选	定义是否需要恢复丢失的文件。
dc.mismatch.remove=false	可选	定义是否删除 Meta 中不存在的 zb 文件。
file.sync.interval=3600000	可选	定义全量更新元数据文件的时间间隔。
global.data.timeout=600000	可选	定义获取维度表的超时时间。
zk.conn.timeout=120000	可选	定义客户端到 ZooKeeper 集群节点间通讯的超时时间。

属性	可选/必选	说明
zk.conn.hosts	可选	定义客户端到 ZooKeeper 集群的地址，多个地址以逗号分隔，如 zk.conn.hosts=192.168.3.138:2181,192.168.3.138:2182,192.168.3.174:2181
dc.use.backup=false	可选	定义是否启用 Naming 的备份机制。
dc.backup.max.bytes=1048576	可选	定义启用 Naming 备份机制后，每次 Naming 节点到 ZooKeeper 最大可传输日志的大小。

7. 集市通讯和资源使用

本章节主要从以下几个方面介绍集市中的一些特性：

- 通讯层
- 内存管理
- 其它性能提升

7.1.通讯层

数据集市的通讯层采用可复用异步的方式。它有这样一些特点：

异步：当数据可用时，以消息通知机制与 Socket 底层交互，通讯层以异步模式工作。

可复用：由于通讯层是异步模式，多个任务可以共享通讯线程。系统的开销更小，效率更高。

锁定内存：由于通讯层需要收发大量的数据，不合理的内存管理，将严重影响整个系统的性能，或者导致不稳定。通讯层锁定了内存，一旦 Node 之间建立好连接，将不再有内存的申请和归还。

多路：Node 之间的连接按需管理，只要不超过设定的最大连接数。

❖ 通讯层协议

可复用多路异步通讯协议。该协议已经在引言中描述了。

文件传输协议。由于文件比较大，采用异步传输模式并不是最高效的。系统实现了一种通讯协议来专门传输文件。这种文件传输协议是同步的，而不是异步的。

❖ 相关的 Node

整个系统，参与通讯的 Node 有以下一些类型：

Naming Node：存储数据集市系统的元数据。

Map Node：存储一部分 Map 数据的元数据和物理数据，并执行 Map 任务。

Reduce Node：存储一部分 Reduce 数据的元数据和物理数据，并执行 Reduce 任务。

Client Node：发起对数据集市系统的访问。

其中，传输文件时采用文件传输协议，而其它信息交互采用可复用多路异步通讯协议。

❖ 相关的 **Properties**

• Global Properties

这些 Properties 是所有 Node 共享的。

dc.cache.max=5242880

<Optional> 定义最大的内存 Buffer，超过这么多的数据被读入 / 写出，将发起至少一次物理读入 / 写出。

dc.io.timeout=15000

<Optional> 定义两个 Node 之间通讯的最大等待时间。

dc.io.block=131072

<Optional> 定义 Socket 读写的 Buffer 大小。

dc.io.sport=5083

定义各个 Node 之间通讯的端口。

dc.io.fport=5066

定义各个 Node 之间传输文件的端口。文件传输主要从 Client Node 传往 Map/Reduce Node，或者发生在 Map/Reduce Node 之间。

dc.io.channel.expire.time=0

<Optional> 定义两个 Node 之间通信连接的过期时间，默认为 0，不过期。

• Local Properties

这些 Properties 每个 Node 可以自定义。

dc.io.handlers=1

<Optional> 定义处理 IO 通讯的线程数。一般情况下，一个线程足够了。

dc.io.channels=2

<Optional> 定义与其它 Node 通讯时，最大 Socket 连接数。

dc.io.ip=

<Optional> 定义本机的 IP，尤其在多网卡的时候。如果未定义，将试着从操作系统得出 IP。

❖ Pin 机制

为了避免未经验证的 Node 加入到云系统中窃取信息，系统采取 Pin 机制来加强通讯的安全性。

这里的 Pin 是一个 Global-wise 的 Property，如果它存在，两个 Node 之间的通讯首先需要验证 Pin，如果 Pin 验证失败，将拒绝通讯。

❖ 通信缓冲区动态扩张

由于系统限制了传输数据的上限，所以在传输较大数据时容易导致超出缓冲区（buffer）的大小，而这个过程中需要传输计算数据，因此会造成执行效率较慢的问题，因此产品对传输数据过程中 buffer 不够的情况进行了优化，使系统能够动态的调整通讯过程中缓冲区的大小，以此来减小传输过程中的数据量，提高通信效率。

❖ New RPC 梳理实现

数据集市中，通过 RPC 调用远程方法，如果遇到通信出错的问题，相关的任务可能会等待很久然后发生 timeout 超时的情况，如果这个查询是一个基础查询，那么所有依赖这个基础查询的任务都要等待。

为了提升稳定性，产品重新设计了 RPC 通讯机制，通过重复发送 CProc 来降低系统的响应延迟。假如单次通信故障，那么可以连续发送 3 次 CProc 来保持连接，只要有一次返回就算通讯成功。

通过这种机制可以更加稳定的检查作业的状态，减少作业等待时间并及早反馈结果。

- 相关属性配置：

以下属性配置在 global_bi.properties 中，配置的值都是系统默认值，可根据使用情况来决定是否进行调整，一般使用默认值即可。

属性	说明
rpc.repeat=true	定义控制重复发生 CProc 的开关
repeat.max.times=3	重复发送 CProc 的最大重发次数
repeat.period.quick=5000	Audit 查询重发间隔
repeat.period.moderate=10000	集市聚合查询重发间隔
repeat.period.slow=30000	集市明细查询重发间隔
repeat.error.ignore=false	是否忽略重发过程中返回的错误

❖ 网络短暂中断处理

在使用产品时，如果遇到网络不好的情况会出现网络短暂中断，导致请求出错，影响用户体验。

7.0 版本对这一问题进行了完美的解决，在前端出现网络短暂中断问题导致请求失败后，会自动重新发送请求获取期望的结果，在增加产品稳定性的同时也提升了用户体验和满意度。

7.2.内存管理

❖ 缓存交换策略

从冷热内存上加以区分，避免运行偶尔的错误使用，或者较少使用的查询占用太多内存资源。

一个或几个大的缓存结果占用了大部分的缓存空间，使得其他缓存结果要被交换出去，综合考虑缓存空间和内存情况调整。

在系统的缓存使用时，小的缓存结果因为大的缓存结果的存在，很容易被交换出去，导致缓存命中率下降，从而增加额外的资源开销。为了提升缓存的命中率，提高产品的性能，7.0 版本中为内存增加了冷区和热区的划分，将使用次数较少的缓存放入冷区中，将使用次数较多的缓存放入热区，其中冷区中的缓存将优先被交换出内存，从而可以使较少使用的查询结果或者大的缓存结果更容易被交换出去，来提升缓存命中率，也可以通过参数配置来调整冷区和热区的划分。

- 相关属性配置：

以下属性配置中的值都是系统默认值，可根据使用情况来决定是否进行调整，一般使用默认值即可。

属性	说明

属性	说明
link.class.name=g5.mem.PartitionedLink	使用 link 的类型，设置为 g5.mem.LRULink 则表示使用不分区的缓存空间
link.hot.times=2	缓存由冷区升入热区的最少使用次数
link.par.ratio=3	热区和冷区的比率

产品的缓存空间内部分为两个区域（固定大小）：冷区和热区，热区较大，冷区较小，热区与冷区默认比率为 3：1，冷区存放使用次数为 1 的数据，热区存放使用次数大于 1 的数据，冷区和热区的交换主要按照如下方式进行：

- 当执行完结果时，加入冷区；
- 当获取数据时，如果缓存为空，则装载数据，加入冷区；若缓存不为空且使用次数大于 1，则将此块数据加入热区；
- 当往热区加入数据时，如果热区已满，则被淘汰的数据，又加入冷区。

❖ DataGrid 生命周期管理

产品中定义了 DataGrid 接口用来表达运算结果，一般来说 DataGrid 对象被认为是产品中大量存在，而且占用内存空间较大的一类对象。因此，DataGrid 对象的生命周期直接关系到产品对内存的使用是否合理，影响计算效率。

因此在产品中对 DataGrid 对象的生命周期管理方式进行了优化，避免 DataGrid 对象被过早或者过晚回收，能够及时清除不被使用的 DataGrid 对象，减少内存的无端占用，有效控制 serial 目录的增长。

- 相关属性配置：

以下属性配置中的值都是系统默认值，可根据使用情况来决定是否进行调整，一般使用默认值即可。

属性	说明
sys.lifecycle.managed=true	定义是否开启 DataGrid 对象的生命周期管理
sys.lifecycle.debug=false	定义是否打开调试信息，使用时建议关闭
lifecycle.check.period=7200000	定义查看系统中存在的DataGrid信息的时间间隔

❖ 执行结果缓存设计

产品运行时会合理的将运算结果进行缓存，不但可以节省计算资源，还能够改善产品体验。目前，产品已经支持对运算结果的缓存。缓存以 Query 的基础运算结果为基本单元，一些在基础运算结果上的后处理则不进行缓存，基于同一 Query 的不同运算没有在缓存层面进行考虑。在 7.0 版本中，对上述缓存机制进行了提升，进一步提高缓存结果重用率，从而提高计算的效率。

是否缓存最终运算结果需要满足如下规则：

- 进行的运算是非详细查询的；
- 查询对应的报表模式为非编辑状态；

- 当最终运算结果和已经缓存的结果大小相同时（如无后续处理或仅包含表达式或者排序），不需要缓存；
- 当已缓存的结果记录行数较少时 ，或后续处理执行时间较短时，不需要缓存。

对于集市数据运算结果的缓存，目前将结果保存在内存当中，默认的有效时间为 15 分钟，为增强数据查看的体验，达到秒级打开报表的效果，我们将报表执行的结果进行永久缓存，保存在磁盘上，第一时间将数据展现出来。

缓存的结果会被保存在与 bihome 文件夹同级的 cache 文件夹（Yonghong/cache）里面，缓存的结果有效期默认为一周，一周之后缓存的结果将会被从磁盘上删除。

对于将运算结果保存在磁盘上的功能，目前只支持缓存使用集市数据（同步查询数据，增量导入数据）的报表结果，可以通过参数来控制是个开启结果缓存的功能，默认是开启的。同时也可以通过参数来控制缓存的结果是否随着报表的更改而更新，为了减轻系统的负担，默认不打开自动更新的功能，按照具体的需求可以通过参数来更改。

当报表结果被缓存到磁盘后，报表中配置的参数_REFRESH_将不再生效，每次查看报表时都会直接读取缓存到内存或者磁盘上的结果，并不会重新进行集市运算。如果想每次都重新进行集市计算的话，需要通过配置参数 result.serial=false 来关闭这个缓存的功能。

属性	说明
result.serial=true	默认打开缓存到磁盘的功能
result.serial.auto.refresh=false	是否自动更新已缓存到磁盘的文件，默认不缓存
result.serial.auto.refresh.interval=1	自动更新已缓存到磁盘的文件的时间间隔

属性	说明
800000	
result.serial.max.length=20	单个缓存到磁盘文件大小的限制，默认为 20M
result.serial.max.size=1000	限制可以缓存到磁盘的文件个数，默认为1000个，可以理解为一个 element 对应一个缓存文件
result.serial.timeout=604800000	缓存到磁盘的文件的过期时间，默认为一周没有被使用就会过期

❖ **线程执行的内存调度**

系统对线程执行的内存调度进行了优化，当通过多线程同时执行多个运算时，系统会根据系统负载情况来对内存进行控制，合理的分配系统资源，防止运算受限或者内存溢出，以提高 cpu 和内存的使用效率，从而提高运算速度。

在产品使用过程中，系统会监控计算内存（可通过参数 mem.calc.mem 配置）的使用，估算每个计算线程的内存占用，采用内存申请的机制来为每个线程分配合适的内存。

首先线程执行的时候该线程会向系统申请一个单位内存（可通过参数 calc.mem.units 配置），系统会设置一个检查点，当线程执行一段时间到达检查点后会估算线程总占用的内存大小，此时：

- 如果申请到的内存比预估的总占用内存大，则可以不申请内存，直至任务运行结束；
- 如果占用的内存比申请到的内存大，或快用完，则继续申请内存，如果申请不到，则该 Runnable 会暂停执行；

- 如果此时已无空闲内存，而所有线程都处于申请内存状态，则此时应该给予额外的内存，让其中的某个线程优先执行完，并释放出空闲内存，其中优先级高的线程可以更优先获得空闲内存。
- 相关属性配置：

以下属性配置中的值都是系统默认值，可根据使用情况来决定是否进行调整，一般使用默认值即可。

属性	说明
mem.apply.timeout=3600000	定义每个线程申请内存的超时时间
calc.mem.managed=true	定义是否使用计算内存机制
calc.mem.debug=false	定义是否打开计算内存的调试信息，使用时建议关闭
calc.mem.units=10	定义为每个线程分配的初始内存值，默认为 10M
mem.calc.mem=	定义计算内存的大小，默认为最大 JVM 的 3/10

7.3.其它性能提升

❖ **表达式运算执行效率提升**

目前，关于表达式的处理部分，一部分表达式使用产品自带的处理引擎，一部分表达式使用 js 引擎（jsEngine）来处理。一个表达式，既有系统支持的表达式，又有系统不支持的表达式，则还是会用 js 引擎（jsEngine）解析。

在处理大数据量的集市数据时，由 jsEngine 处理的表达式效率要比产品自身支持的表达式的效率低，因此为了提高表达式的处理效率，7.0 版本对产品自身支持的表达式范围进行了扩展，主要新增了对以下几个表达式的支持：

函数	说明
Right	截取字符串右边特定长度，如果字符串长度不够以字符 串长度为准
Left	截取字符串左边特定长度，如果子字符串长度不够以字 符串长度为准
Trim	去掉字符串两边的空格
Upper	把字符串字符都转化成大写
Lower	把字符串字符都转化成小写
Weekdayname	获取星期中的序号，比如默认星期天是 1
Year	返回日期的年份
Month	返回日期的月份
parseDate	把字符串解析成日期

函数	说明
FormatDate	把日期格式化成字符串

❖ Local Reduce 优化

MPP 数据集市在数据节点（Map 节点）可提前进行局部的 Reduce 计算，即 Local Reduce。

在一些计算场景中，有自助分析类的报表，这类报表依赖的数据集一般比较大，而且绑定维度不固定，结果集可能会比较大。当结果集较大时，reduce 节点的处理内存很容易成为瓶颈，无法满足计算处理需要，因此对 Local Reduce 进行了优化，当计算满足 Local Reduce 的时候，在 Map 节点等待一定时间来尽可能多的进行 Local Reduce 计算，充分利用集市计算资源，达到提升集市计算处理速度的目标。

目前 Local Reduce 主要从这几个方面判定 Map 结果是否需要进行 Local Reduce 运算：

- 汇总计算且 Map 计算的结果集小于 500000；
- 当前 local reduce 的压缩比率小于 0.33；
- 在 map 完成时已经有 local reduce 的执行，且满足上面两条。

8. 二次开发

本章节主要从以下几个方面介绍二次开发。

- 导数的 API。
- 删除文件 API。
- JDBC 访问的 API。

❖ 导数的 API

- 初始化环境

```
// prepare the environments.
String bihome ="E:/source/yonghongsoft/source/home";
System.setProperty("bi.home", bihome);
String license = Setting.get("license");

//need call init license to get the permission to execute data.
//the init must needed.
必需 init license，才可使用。
if(license == null || !g5.util.KeyUtil.init(license)) {
   System.out.println("_____key invalid____"); return;
}

//init the Data Mart server.
```

```
DCUtil.init(false);


//set the thread count. according to CPU count and license permission.
//Setting.setInt("mem.proc.count", 2);
```

如需要设置参数，譬如 “mem.proc.count” 可通过 Setting 设置。

```
//set false to ignore the naming checking.
//if the system status is not right while checking file is true, the system cannot be start.
Setting.setBoolean("dc.naming.check.file", false);
```

初始化数据文件夹及文件位置。

```
// init values for file/folder/append from
GSFolderC holder = new GSFolderC();
holder.setFolder("testQryFolder");
holder.setFile("testQryFile");
holder.setAppend(true);


// set meta.
//holder.setMeta("region", "beijing");
```

如需要设置 meta 属性，可在这里设置则生成的文件将被标明是 region=beijing。

```
// temp generate files path.
String genPath = Setting.getHome() + "/gen";
```

设置生成的临时文件保存的路径，此处设在 {bihome}/gen 下。

生成数据 DataGrid。

```
 // init the query
QueryRep rep = QueryRep.get();
Query qry = (Query) rep.getAsset(new AssetRef(Query.SQL, "testQry"), null);

// create the context for query, the context will contains some environment parameters.
QContext context = new QContextImpl(null, QContext.RT_MODE, null);

//set max rows for query, while execute query, only get max rows results.
//context.set(context.MAX_ROWS, 10000000);

//force to load data from real source.
context.set(QContext.REFRESH, Boolean.TRUE);

// set your parameter here for query.
//context.set("param", "value");
```

如需要设置参数传递给 query，则需要设置进 context。

```
// get query columns.
QCol[] cols = qry.cols(true);
```

```
// get DataGrid
GQuery gqry = GQuery.create(qry, context, cols, null, null, null, null);
DataGrid grid = gqry.getGrid();
```

从 DataGrid 生成数据文件。

```
//generate Data Mart files from DataGrid to a temp folder genPath.
File genFile = new File(holder.getFile());
Splitter split = new Splitter(cols, grid, null, genFile, null);
split.gen();
split.waitFor(1000 * 60 * 60 * 24);
```

➢**注意**：这里 split 需要等待，否则有可能生成失败，等待的时间由数据大小决定，设置足够大的值的话就好。

将数据文件通过 AddFolderTask 加载到系统内。

```
// Init GSFolder.
GSFolder folder = initFolder(holder, holder.getFolder(), genFile, cols);
AddFolderTask task = new AddFolderTask(folder, holder.isAppend());
GNodeResult result = task.exec();
```

具体可参考附录 -Push data 例子。

❖ 删除文件的 API

- 删除文件 RemoveFileTask

```
RemoveGSFileTask task = new RemoveGSFileTask(cloudName);
```

GNodeResult result = task.exec();

cloudName 为云文件名，譬如 "testCloud/dd.zb"，为相对 bihome 下 cloud 文件夹的相对目录。

- 删除文件夹 RemoveFolderTask

RemoveFolderTask task = new RemoveFolderTask(cloudName);

GNodeResult result = task.exec();

cloudName 为云文件夹名，譬如 "testCloud "，为相对 bihome 下 cloud 文件夹的相对目录。

❖ **JDBC 访问的 API**

Class.forName("g5.dc.jdbc.GDriver");

设置 Driver 为 "g5.dc.jdbc.GDriver"

Connection conn = DriverManager.getConnection("jdbc:yonghong:z", "admin", "g5");

Driver 的链接 url 为固定的 "jdbc:yonghong:z"，用户名为 "admin"，密码为 "g5"。

```
Statement st = conn.createStatement();

ResultSet rs = st.executeQuery("select CITY, sum(CONSUME) + 2 as tp from cloud.clqry group by CITY order by tp ");
```

创建 sql 数据集，并返回 results。

```
DataGrid grid = RSExecutor.create(rs);
```

根据 results 生成 DataGrid.

```
GridUtil.print(grid, 20);
```

打印 DataGrid。生成的 DataGrid 为可进行二次开发的数据集。

9. 应用举例

❖ Push data

从数据源的地方把数据推送到集群中，需要编写 usertask 程序来解析原始数据，然后将解析后的数据导入数据集市。请参考 Parse CSV 例子。

❖ Pull data

用调度任务里的作业去数据源里拉数据。如图所示，在调度任务中新建一个作业，任务类型选择增量导入数据，选择一个建好的数据集，执行后可将数据集结果导入数据集市。请参考 Parse CSV 例子。

❖ 增删 Cloud File 和 Cloud Folder

```
import g5.Setting;
import g5.dc.impl.DCUtil;
import g5.dc.node.GNodeResult;
import g5.dc.node.RemoveFolderTask;
import g5.sv.db.DBService;

public class TestRemoveFolderTask {
public static void main(String[] args) throws Exception {
String bihome ="E:/source/yonghongsoft/source/home";
String cloudName = "testCloud";
System.setProperty("bi.home", bihome);
String license = Setting.get("license");

if(license == null || !g5.util.KeyUtil.init(license)) {
System.out.println("_________key invalid____");
return;
}


Setting.setBoolean("dc.naming.check.file", false);
DCUtil.init(true);
DBService.get();
```

```
System.err.println("Press any key to generate...");
System.in.read();
System.in.read();
RemoveFolderTask task = new RemoveFolderTask(cloudName);
GNodeResult result = task.exec();

if(result != null) {
System.out.println("____result:_________"+result.state()+"|"+result.toString());
}
 }
}
```

❖ Parse CSV

ParserCsv.java

```
import g5.dc.fs.s.GSFolder;
import g5.dc.node.impl.GSFolderC;
import g5.grid.impl.GridUtil;
import g5.io.IOUtil;
import g5.meta.BCol;
import g5.meta.DType;
import g5.meta.QCol;
```

```
import g5.qry.impl.QColumnSeg;
import g5.qry.impl.QGrid;
import g5.qry.impl.QUtil;
import g5.sched.TaskContext;

import java.io.*;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.commons.lang.StringUtils;
import com.ibm.icu.text.DateFormat;
import com.ibm.icu.text.SimpleDateFormat;



/**
* Parse the disk file to generate datagrid.
*/
public final class ParserCsv {
public ParserCsv(ImportCsv csvJob, TaskContext context) {
    _context = context;
    _job = csvJob;
```

```
}

public void setGFilename(String foldername) {
    folder = foldername;
}

public void setFileDate(String m_ts) {
    this.log_ts = m_ts;
}


public void parseDirectory(String path) throws Exception {
File file = new File(path);

if(!file.exists()) {
throw new RuntimeException("The given path: " + path + " not found the folder");
}

parseDirectory(file);
}
```

```
/**
* Parse the text data under the given folder.
 */
private void parseDirectory(File file) throws Exception {
    if(!file.isDirectory()) {
throw new RuntimeException("The given path: " + file + " not found the folder");
}

File[] files = file.listFiles();

for(File fl : files) {
if(fl.isDirectory()) {
parseDirectory(fl);
}
else {
parse(fl);
}
}
}
```

```
public void parse(String fileName) throws Exception{
File file = new File(fileName);
parse(file);
}



public void parse(File file) throws Exception{
try {
log.log(Level.INFO, "start to parse file:___ " + file);
long ts = System.currentTimeMillis();
FileReader input = new FileReader(file);
BufferedReader readerInput = new BufferedReader(input);
String inputString = readerInput.readLine();
// map the text column index to grid index
long count = 0;
int r = 0;
int invalidRow = 0;
int parseRow = 0;
QGrid grid = (QGrid) QGrid.create(headers, types, cols);
```

```
while(inputString != null) {
if(inputString.trim().length() > 0) {

String[] str = StringUtils.splitPreserveAllTokens(inputString, ",");
 if(str.length == 6) {
grid.add(0, validNumber(str[0]) ?
        Long.parseLong(str[0]) : null);
grid.add(1, str[1]);
grid.add(2, str[2]);
Date date=format.parse(str[3]);
grid.add(3, date);
grid.add(4, str[4]);
grid.add(5, str[5]);
}
else {
// @temp humming, invalid data need to store the other file
log.log(Level.WARNING, "find invalid " + r + " record: " + inputString);
invalidRow++;
}
}

inputString = readerInput.readLine();
```

```
r++;
}

grid.complete();
log.log(Level.WARNING, "Add data for " + file + ", count=" + (count++) + " over ,
cost=" +
(System.currentTimeMillis() - ts));
log.log(Level.WARNING, "Invalid row record: " + invalidRow + " for total: " + r);
log.log(Level.WARNING, "Parse erro row record: " + parseRow + " for total: " + r);
GridUtil.print(grid, 10);
addGrid(grid, System.currentTimeMillis());
grid = (QGrid) QGrid.create(headers, types, cols);
}
catch(FileNotFoundException e) {
e.printStackTrace();
}
}

/**
* Invalid number.
*/
```

```
private boolean validNumber(String str) {
if ((str != null) && (str.length() > 0)) {
for (int i = str.length(); --i >= 0;) {
if (!Character.isDigit(str.charAt(i))) {
return false;
}
}
return true;
}
return false;
}

private void addGrid(QGrid grid, long ts) throws Exception {
// empty grid
if(!grid.exists(2, -1, true)) {
return;

}

GSFolderC holder = new GSFolderC();
holder.setFolder(folder);
holder.setFile(folder + ts);
```

```
holder.setAppend(true);
File dir = null;
GSFolder folder = null;

try {

dir = IOUtil.getTempFolder("CloudTask");
_job._split(_context, dir, holder.getFile(), grid, QUtil.metaCols(grid));
log.log(Level.INFO, ".....split datagrid.....");
// initialize GSFolder instance
folder = _job._initFolder(holder, holder.getFolder(), dir, QUtil.metaCols(grid));

//check the cloud name is activity or not
_job._checkActivity();

//remove the folder in GFS if exists and not append
if (!holder.isAppend()) {
_job._removeFolder("CloudTask", holder.getFolder());
}

// add into GFS
_job._addFolder("CloudTask", folder, holder.isAppend());
```

```
}
finally {
if(folder != null) {
folder.dispose();
}

if(grid != null) {
// close, release mem
grid.dispose();
grid = null;
}

if(dir != null) {
boolean removed = IOUtil.remove(dir);

if(!removed) {
log.log(Level.WARNING, "Failed to remove directory '" + dir.getAbsolutePath() + "'
for GSFolder '" + holder.getFolder() + "'!");
}
}
}
```

```
}
public static String[] headers = {"NUMBER","KEYWORD","HOST","TIME",
"PROVINCE","CITY"};
// the index 4 PTMSI is store the number with 16 rather than 10
public static byte[] types = {QColumnSeg.DYNAMIC_LONG,
QColumnSeg.DEF_STRING, QColumnSeg.DEF_STRING,
QColumnSeg.DYNAMIC_DATE_TIME, QColumnSeg.DEF_STRING,
QColumnSeg.DEF_STRING
};


public static QCol[] cols = {new BCol("NUMBER", DType.LONG),
new BCol("KEYWORD", DType.STRING,true), new BCol("HOST", DType.STRING,true),
new BCol("TIME", DType.DATE_TIME),new BCol("PROVINCE", DType.STRING,true),
new BCol("CITY", DType.STRING,true)
};

private static final Logger log =
Logger.getLogger(DianliParserDsv.class.getSimpleName());
DateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
private String folder = "CSV";
private TaskContext _context;
```

```
private ImportCsv _job;
}
```

ImportCsv.java

```
import g5.DataGrid;
import g5.dc.fs.s.GSFolder;
import g5.dc.impl.DCUtil;
import g5.dc.node.impl.GSFolderC;
import g5.meta.*;
import g5.sched.*;
import g5.sched.jobs.AbstractCloudTask;
import g5.util.KeyUtil;
import java.io.File;



/**
* import data to cloud.
 */
public class ImportCsv extends AbstractCloudTask{
@Override
public Parameter[] getParams() throws Exception {
return new Parameter[] {new Parameter("gs_Folder", DType.STRING, new String[]
{"theta"}),
```

```
new Parameter("csv_Folder", DType.STRING)};
}

@Override
public void run(TaskContext context) throws Exception {
ParserCsv p = new ParserCsv(this,context);
//get name of cloud folder from schedule
gs_Folder = context.getParam("gs_Folder").getValue().toString();
//get path of csv folder from schedule
csv_folder = context.getParam("csv_Folder").getValue().toString();
p.setGFilename(gs_Folder);

try {
p.setFileDate((String) context.getParam("log_ts").getValue());

}

catch(Exception ex) {
// ignore it
}

System.err.println("run:_____ " + gs_Folder + ", src=" + csv_folder);
```

```
try {
p.parseDirectory(csv_folder);
}
catch(Exception ex) {
ex.printStackTrace();
throw ex;
}
}
```

```
/**
* Split the grid.
 */
public void _split(TaskContext context, File dir, String file, DataGrid jGrid, QCol cols[])
throws Exception {
super.split(context, dir, file, jGrid, cols);
}
```

```
/**
```

```
* Remove folder.
*/
public void _removeFolder(String task, String folder) throws Exception {
super.removeFolder(task, folder);
}


/**
* Init folder.
*/
public GSFolder _initFolder(GSFolderC holder, String folder, File dir, QCol cols[]) {
return super.initFolder(holder, folder, dir, cols);
}


/**
* Check cloud activity.
*/
public void _checkActivity() {
super.checkActivity();
}
```

```
/**
* Add cloud folder.
*/
public void _addFolder(String task, GSFolder folder, boolean isAppend) throws
Exception {
super.addFolder(task, folder, isAppend);
}

/**
* Test main entrance.
*/
public static void main(String[] args) {
System.setProperty("bi.home", bihome);

// initialize license
if(!KeyUtil.installLicense()) {
System.out.println("invalid license!");
return;
}
```

```
DCUtil.init(true);
ImportCsv xJob = new ImportCsv();

try {
xJob.run(new SchedContext());
}
catch(Exception e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

}
private static String bihome = "D:/YH/Yonghong/bihome";
private String csv_folder = "E:/data_csv";
private String gs_Folder = "CSV";
}
```

❖ 运行 **Query** 获得结果

方法 1:执行 bihome 中已经建好的 query

```
import g5.DataGrid;
import g5.dc.impl.DCUtil;
```

```
import g5.grid.impl.GridUtil;
import g5.sched.*;
import g5.util.KeyUtil;

public class runQry {

public static void main(String[] args) {
System.setProperty("bi.home", bihome);

// initialize license
if(!KeyUtil.installLicense()) {
System.out.println("invalid license!");
return;
}



DCUtil.init(true);
try {
//run query
DataGrid grid = runQry("hive1.sqry",new SchedContext());
GridUtil.print(grid, 10);
```

```
}
catch (Exception e) {
e.printStackTrace();
}
}

private static DataGrid runQry(String path, TaskContext context) throws Exception {
return IRuiDataCache.getData(path, context);
}

private static String bihome = "D:/YH/Yonghong/bihome";
}
```

方法 2：通过 jdbc 连接已经建好的数据集市数据集。

```
import g5.dc.impl.DCUtil;
import g5.util.KeyUtil;

import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
```

```
import java.sql.DriverManager;

public class runCloudQry {

public static void main(String[] args) throws SQLException {
System.setProperty("bi.home", bihome);

// initialize license
if(!KeyUtil.installLicense()) {
System.out.println("invalid license!");
return;

}

DCUtil.init(true);
try {

Class.forName(driverName);

}
catch (ClassNotFoundException e) {
// TODO Auto-generated catch block
```

```
e.printStackTrace();
System.exit(1);
}

Connection con = null;

try {
con = DriverManager.getConnection("jdbc:yonghong:z", "admin", "g5");
Statement stmt = con.createStatement();

String sql = "select * from \"test.clqry\"";
System.out.println("Running1: " + sql);
ResultSet res = stmt.executeQuery(sql);

while (res.next()) {
System.out.println(String.valueOf(res.getString(1)));
}

}
finally {
try {
if(con != null) {
```

```
con.close();
}
}
catch(Exception ex) {
ex.printStackTrace();
}
}
}
}
private static String bihome = "D:/YH/Yonghong/bihome";
private static String driverName = "g5.dc.jdbc.GDriver";
}
```

❖ **Join Query**

```
import g5.DataGrid;
import g5.dc.impl.DCUtil;
import g5.grid.JoinGrid;
import g5.grid.impl.GridUtil;
import g5.sched.SchedContext;
import g5.sched.TaskContext;
import g5.util.KeyUtil;

public class joinQuery {
```

```
public static void main(String[] args) throws Exception {
System.setProperty("bi.home", bihome);


// initialize license
if(!KeyUtil.installLicense()) {
System.out.println("invalid license!");
return;
}

DataGrid grid1 = runQry("test1.sqry",new SchedContext());
DataGrid grid2 = runQry("test2.sqry",new SchedContext());
DCUtil.init(true);
JoinGrid g = null;
byte hint = JoinGrid.FINAL_JOIN;
byte op = JoinGrid.JOIN;
short lkeys[] = {0};
short rkeys[] = {0};
short lcols[] = {0,1,2,3,4,5};
short rcols[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};
g = JoinGrid.create(hint, grid1, grid2 , lcols, rcols, lkeys, rkeys, op, null);
GridUtil.print(g, 10);
```

```
}

private static DataGrid runQry(String path, TaskContext context) throws Exception {
return IRuiDataCache.getData(path, context);
}

private static String bihome = "D:/YH/Yonghong/bihome";
}
```