

阿里云

表格存储Tablestore
解决方案

文档版本：20220324

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 <code>cd /d C:/window</code> 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	<code>bae log list --instanceid</code> <code>Instance_ID</code>
[] 或者 [a b]	表示可选项，至多选择一个。	<code>ipconfig [-all -t]</code>
{ } 或者 {a b}	表示必选项，至多选择一个。	<code>switch {active stand}</code>

目录

1.如何打造千万级Feed流系统	06
1.1. Feed流简介	06
1.2. 存储方案分析	07
1.3. 推送方案分析	09
1.4. 表格存储Feed流方案原理	11
1.5. 表格存储Feed流方案实现	13
1.6. 方案扩展	15
2.如何优化高并发IM系统架构	17
3.搭建海量智能元数据管理系统	25
3.1. 方案背景	25
3.2. 准备工作	27
3.3. 方案实现	28
4.现代IM系统中的消息系统	32
4.1. 现代IM系统中的消息系统—架构	32
4.2. 现代IM系统中的消息系统—模型	36
4.3. 现代IM系统中的消息系统—实现	40
5.海量气象格点数据解决方案	54
5.1. 方案背景	54
5.2. 方案设计	55
5.3. 方案实现	58
6.亿量级订单管理方案	64
6.1. 方案背景	64
6.2. 方案选择	64
6.3. 准备工作	66
6.4. 搭建订单系统	67
7.基于多元索引搭建亿量级店铺搜索系统	70

7.1. 方案背景	70
7.2. 准备工作	71
7.3. 方案实现	72
8.表格存储结合Spark流批处理实现一体化存储和计算	74
8.1. 方案背景	74
8.2. 准备工作	75
8.3. 方案实现	76
9.表格存储结合实时计算Flink进行大数据分析	81
9.1. 方案背景	81
9.2. 准备工作	82
9.3. 方案实现	82

1.如何打造千万级Feed流系统

1.1. Feed流简介

了解Feed流的基本概念、特征以及基本分类。

什么是Feed流产品

在互联网领域，尤其现在的移动互联网时代，Feed流产品非常常见。例如我们每天都会用到的朋友圈、微博就是一种非常典型的Feed流产品，图片分享网站Pinterest、花瓣网等又是另一种形式的Feed流产品。除此之外，很多应用都会有一个模块，有些叫动态、有些叫消息广场，这些也是Feed流产品。Feed流产品已经被广泛应用在各种主流应用中。

基础概念

概念	描述
Feed	Feed流中的每一条状态或者消息都是Feed，例如朋友圈中的一个状态就是一个Feed、微博中的一条微博就是一个Feed。
Feed流	持续更新并呈现给用户内容的信息流。每个人的朋友圈、微博关注页等都是一个Feed流。
Timeline	Timeline是一种Feed流的类型，微博、朋友圈都是Timeline类型的Feed流。但是由于Timeline类型出现最早、使用最广泛、最为人熟知，有时也用Timeline来表示Feed流。
关注页Timeline	展示其他人Feed消息的页面，例如朋友圈、微博的首页。
个人页Timeline	展示自己发送过的Feed消息的页面，例如微信中的相册、微博的个人页。

基本特征

Feed流包含以下几种基本特征。

特征	描述
多账号内容流	Feed流系统中存在成千上万的账号，账号之间可以进行关注、取关、加好友和拉黑等操作。
非稳定的账号关系	由于存在关注、取关等操作，所以系统中的用户之间的关系就会一直在变化，这是一种非稳定的状态。
读写严重不平衡	读多写少，一般读写比例在10：1，甚至100：1以上。
消息必达性要求高	例如发送了一条朋友圈后，结果部分朋友收到信息，部分朋友没收到，出现信息不对称。

系统分类

Feed流的分类有很多种，但最经典、常见的分类有两种。

- Timeline：按发布的时间顺序排序。先发布的先看到，后发布的排列在最上方，类似于微信朋友圈、微博，是最常见的Feed流形式。Timeline类型的产品适合用户Feed流中的Feed不多的情况，但需保证用户

可以看到每一个Feed。

- Rank: 按某个非时间的因子排序, 一般是按照用户的喜好度排序。用户最喜欢的排在最前面, 次喜欢的排在后面。Rank一般假定用户可能看到的Feed非常多, 但用户花费在这里的时间有限, 所以需要为用户选择出用户最想看的Top N结果。应用场景包括图片分享、新闻推荐类、商品推荐等。

Feed流还有其他的分类标准, 其他的分类标准会多出两种类型。

- Aggregate: 聚合类型。例如好几个朋友都看了同一场电影, 这个就可以聚合为一条Feed: A、B、C看了电影《你的名字》。这种聚合功能比较适合应用在客户端。一般的Aggregate类型是Timeline类型 + 客户端聚合。
- Notice: 通知类型, 属于功能类型。通知类型一般用于应用中的各种通知, 例如私信。Notice也是Timeline类型或Aggregate类型。

方案设计核心

由于系统中的所有用户不可能全部在线, 且不可能同时刷新和发布Feed, 那么一个能支撑千万量级Feed流的系统, 在产品上需支撑上亿量级的用户。设计一个Feed流系统, 最关键的两个核心:

- 存储, 详情请参见[存储方案分析](#)。
- 推送, 详情请参见[推送方案分析](#)。

1.2. 存储方案分析

存储是Feed流系统的核心之一。Feed流系统中需要存储的内容分为两部分: 账号关系(例如关注列表)、Feed消息内容。

设计思路

Feed流存储系统有几个问题是设计的关键:

- 如何能支持100TB, 甚至PB级数据量?
- 数据量与成本成正比, 如何能降低成本?
- 如何保证账号关系和Feed不丢失?

账号关系

账号关系特点

- 变长链表, 长度可达亿级别。
- 数据量大, 但关系简单。
- 性能敏感。直接影响关注、取关的响应速度。
- 有序性。不要求具有排序功能, 只需要能按照主键排序即可。只要能按照主键排序, 那么关注列表和粉丝列表的顺序就固定、可预期。

存储方案

根据以上特点, 最适合存账号关系(关注列表)的方案应该是分布式NoSQL数据库, 其中包括:

- 开源HBase

很多企业选择开源HBase来存储账号关系。开源HBase在满足了上述四个特征的同时可以把系统搭建起来, 但是仍会有一些问题:

- 需要自己运维、调查问题、修复问题, 运维成本高。
- GC会导致比较大的毛刺, 影响用户体验。

- 表格存储

阿里云的表格存储也属于有序性的分布式NoSQL数据库，很多知名的系统选择了使用表格存储。表格存储具有以下优势：

- 单表支持10万亿行+、10PB+的数据量，再快的数据增长速度都无需担心。
- 数据按主键列排序，可保证有序性和可预期性。
- 单key读写延迟在毫秒级别，可保证关注、取关的响应时间。
- 是全托管的分布式NoSQL数据库服务，无需任何运维。
- 全部采用C++ 实现，彻底无GC问题，不会由于GC而导致较大的毛刺。

以上可见，使用表格存储来存储账号关系是一个比较好的选择。

Feed消息

Feed消息特点

- 数据量大，而且在Feed流系统的写扩散（推模式）模式下数据量会再膨胀几个数量级，所以数据量很容易达到100TB，甚至PB级别。
- 数据格式简单。
- 数据可靠性要求高，不能丢失数据。
- 需保证个人发的Feed的消息ID在个人发件箱中都是严格递增的，这样读取时只需要一个范围读取即可。由于个人发布的Feed并发度很低，用时间戳也能满足基本需求，但是当应用层队列堵塞，网络延迟变大或时间回退时，用时间戳还是无法保证严格递增。最好的方法是适用主键自增功能。

存储方案

根据Feed消息特点，Feed消息的最佳的存储系统应该是具有主键自增功能的分布式NoSQL数据库，但是在开源系统里面没有完全符合特点的数据库，所以有两种存储方法：

- 关系型数据库 + 分库分表

目前有很多用户选择了关系型数据库+ 分库分表。虽然这个架构可以运行，但是存在一些问题：

- 分库分表带来了运维复杂性。
- 分库分表带来了逻辑层和数据层的极大耦合性。
- 关系型数据库的主键自增功能性能差，例如开源MySQL数据库。不管是用MyISAM还是InnoDB引擎，要保证自增ID严格递增，必须使用表锁。表锁粒度大，会严重限制并发度，影响性能。
- 关系型数据库的可靠性低于非关系型。关系型数据库的可靠性通常最多6个9，远低于分布式数据库，低4到5个级别。

- 表格存储

基于上述原因，一些技术公司开始考虑使用表格存储。表格存储是一个具有自增主键功能的分布式NoSQL数据库，这样就只需要使用一种系统，除此之外表格存储还有以下优势：

- 单表可达10PB、10万亿行。
- 10个9的SLA保障Feed内容不丢失。
- 天然分布式数据库，无需分库分表
- 两种实例类型：高性能实例可提供极佳的读写性能；容量型实例可提供低存储成本。
- 主键自增功能性能佳。其他系统的自增功能都需要加锁，但表格存储完全不需要加锁，包括表锁和行锁。

通过对比分析，表格存储在功能、性能、扩展性以及成本方面都要更加适合存储Feed消息。

1.3. 推送方案分析

推送方案分两种模式拉模式（读扩散）和推模式（写扩散）。本章节主要对比拉模式和推模式，并介绍搭建推送系统的数据库选择。

设计思路

Feed流系统最大的特点是读写严重不平衡，一般读写比例都在10：1，甚至100：1之上。并且刷新Feed时的延时本质上由推送方案决定，其他的任何操作都只能是优化。设计推送系统具有以下关键点：

- 如何才能提供千万的TPS和QPS？
- 如何保证读写延迟在10ms，甚至2ms以下？
- 如何保证Feed的必达性？

推模式和拉模式对比

拉模式和推模式在很多方面完全相反，并且对Feed流产品的用户而言，刷新Feed流（读取）时的延迟敏感度要远远大于发布（写入）。推模式和拉模式的对比请参见下表。

对比项	拉模式（读扩散）	推模式（写扩散）
发布	个人页Timeline（发件箱）	粉丝的关注页面（收件箱）
阅读	所有关注者的个人页Timeline	自己的关注页Timeline
网络最大开销	用户刷新时	发布Feed时
读写放大	放大读，读写比例到10000：1	放大写，减小读，读写比例到50：50
个性化	不支持	支持
定向广告	不支持	支持

以上对比可以看出，推模式要远优于拉模式，但是推模式也有一个缺点：数据会极大膨胀。针对这个缺点，可以从两个方面考虑：

- 以后存储成本会随着硬件技术升级、软件性能优化等继续降低。数据量越大价格越低。
- 如仍需减少成本，可以继续优化，可以对大V采用拉模式，普通用户使用推模式。对活跃粉丝采用推模式，非活跃粉丝采用拉模式（这种方式可以较好的避免大流量对平台的冲击）。

适用场景

以下是不同模式的适用场景。

模式	适用场景
拉模式（读扩散）	很多Feed流产品的第一版会采用这种方案，但很快就会抛弃。如采用拉模式+图计算，重心就会转换为图计算。

模式	适用场景
推模式（写扩散）	- Feed流系统中最常用、有效的模式。 - 用户关系数比较均匀，或者有上限，例如朋友圈。 - 偏推荐类，同一个Feed对不同用户价值不同，需要为不同用户计算分数，例如Pinterest。
推拉结合	大部分用户的账号关系都是几百个，但是有个别用户是1000万以上，例如微博。

数据库选择

如果要实现一个千万量级的Feed流产品，那么推送系统需要具备：

- 千万TPS/QPS的能力。
- 读写链路延迟敏感。读写直接会影响用户发布，刷新Feed流时的延迟，尤其是极其敏感的刷新时的延迟。
- Feed消息的必达性高。
- 主键自增功能，保证用户收件箱中的Feed ID是严格递增的，保证可以通过Scan（上次读取的最大ID --- >MAX）读取到最新未读消息。
- 最好能为用户存储Timeline中所有的Feed。

从上述特点来看，搭建推送系统最优的选择是使用高性能、高可靠，且有自增功能的NoSQL系统：

● 开源Redis

如果选择开源系统，通常会在选择了关系型数据库作为存储系统的基础上，选择开源Redis，这样既能覆盖上述的几个特征，还能保证Feed流系统正常运行起来，但是也会带来一些其他问题：

- 纯内存系统，内存价格极高，导致整体成本高。
- 属于单机系统，为了支持千万TPS和保证消息必达性，需要使用cluster和replica模式，这样不仅带来了运维的复杂性，而且还需要增加机器，导致成本再次上升。
- 成本上升了以后，需要考虑如何降低成本。要节省成本只能是减少开源Redis里面存储的数据量，一般有两种做法：在开源Redis中存储Feed ID、不存储Feed内容。采用这两种方法，整体数据量会大量减少，但是在读取的时候需要先读Feed ID，然后在到存储系统里面去读取Feed内容，网络开销增长了一倍，而且是串行的，对用户的刷新延迟有较大影响。只对普通用户或者活跃用户使用推模式，对大V和非活跃用户直接使用拉模式。
- 上述两个方案虽然可以节省成本，但是会牺牲用户体验，最终需要在成本和用户体验之间权衡。

● 表格存储

除了使用开源系统外，还可以使用阿里云的表格存储，有不少用户选择表格存储作为推送系统的原因无非下面几点：

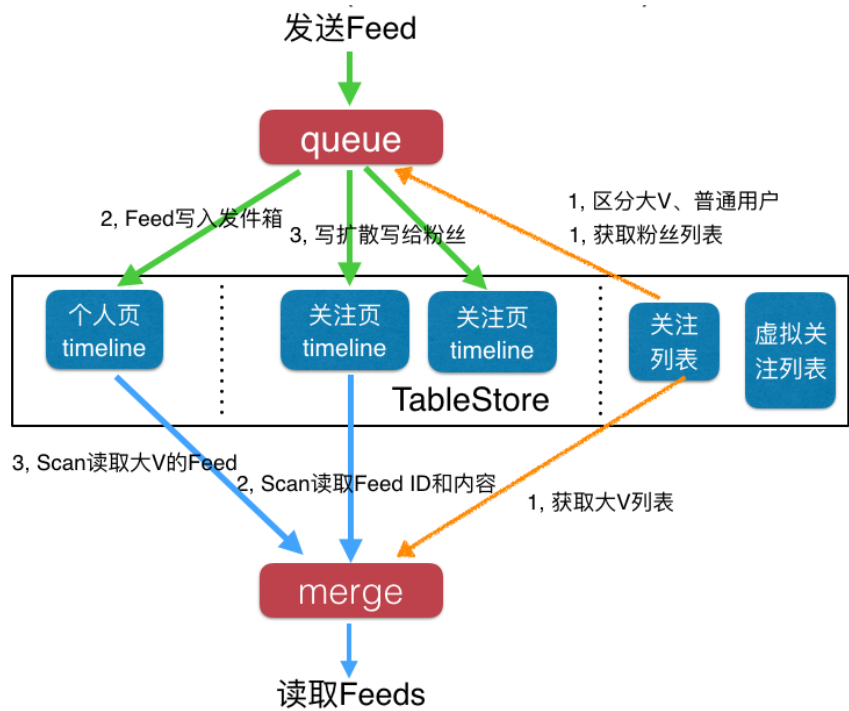
- 天然分布式，单表可支持千万级TPS/QPS。
- LSM存储引擎极大优化写，高性能实例极大优化读。
- 写入成功即保证落盘成功，数据可靠性提供10个9的SLA保障。
- 磁盘性数据库，费用比内存性的要低几个量级。
- 单表可存储十万亿行以上的数据，价格又低，轻松保存用户Feed流中的所有Feed数据。

上面分析了使用开源Redis和阿里云表格存储的异同，如果使用开源数据库可以用Redis，如果选择阿里云NoSQL数据库，可以选择使用表格存储。

1.4. 表格存储Feed流方案原理

本章节主要为您介绍使用表格存储搭建Feed流的方案原理。

方案架构图



存储模块

如架构图所示，Feed流中的以下内容使用表格存储进行存储。

- 个人页Timeline：每个用户的发件箱，也就是自己的个人页页面。
- 关注页Timeline：每个用户的收件箱，也就是自己的关注页页面，内容都是自己关注人发布的消息。
- 关注列表：保存账号关系，例如朋友圈中的好友关系、微博中的关注列表等。
- 虚拟关注列表：主要用来个性化和广告。

发布及读取流程

- 拉结合流程
 - 发布Feed
 - a. Feed消息先进入一个队列服务。
 - b. 先从关注列表中读取到自己的粉丝列表，以及判断自己是否是大V。
 - c. 将自己的Feed消息写入个人页Timeline（发件箱）。如果是大V，写入流程到此结束。
 - d. 如果是普通用户，还需要将自己的Feed消息写给自己的粉丝，如果有100个粉丝，那么就要写给100个用户，包括Feed内容和Feed ID。
 - e. 第三步和第四步可以合并在一起，使用BatchWriteRow接口一次性将多行数据写入表格存储。
 - f. 发布Feed的流程结束。

- 读取Feed流
 - a. 先读取自己关注的大V列表。
 - b. 通过GetRange读取自己的收件箱。范围起始位置是上次读取到的最新Feed的ID；结束位置可以是当前时间，也可以是MAX，建议是MAX值。
 - c. 如果有关注的大V，则再次并发读取每一个大V的发件箱。如果关注了10个大V，那么则需要10次访问。
 - d. 合并2和3步的结果，然后按时间排序，返回给用户。

至此，使用推拉结合方式的发布、读取Feed流的流程结束。

- 推模式流程

如果只是用推模式了，流程更简单。

- 发布Feed流：不用区分是否大V，所有用户的流程都一样。
- 读取Feed流：不需要第1步，也不需要第3步，只需要第2步即可，将之前的 $2 + N$ （N是关注的大V个数）次网络开销减少为1次网络开销。读取延时大幅降级。

个性化和定向广告

个性化和定向广告是需求度很高的两个功能。个性化可以服务好用户，增大产品竞争力和用户粘性。定向广告可以为产品增加盈利渠道，而且不引起用户反感。在Feed流里这两种功能的实现方式类似，我们以实现定向广告的流程为例来说明。

1. 通过用户特征分析对用户分类，例如今年刚上大学的新生（新生类）。

❓ 说明 具体的用户特征分析可以依靠表格存储 + MaxCompute。

2. 创建一个广告账号：新生广告。
3. 让这些具有新生特征的用户虚拟关注新生广告账号。

❓ 说明 用户看不到这一层关注关系。

4. 从七月份开始就通过新生广告账号发送广告。

每个用户可能会有多个特征，那么就可能虚拟关注多个广告账号。

方案优势

使用表格存储架构实现Feed流具有以下优势。

- 只使用1种系统，架构、实现简单。不再需要访问多个系统，架构、开发、测试、运维都能节省人力时间。
- 表格存储主键自增列功能性能极优。表格存储主键列自增不需要表锁和行锁，性能要远远高于关系型数据库。
- 表格存储采用分布式存储，可以存储所有Feed。
- 表格存储的存储成本低，您无须将Feed ID和内容分开存储。
- 全托管服务，无运维操作，更无需分库分表。
- 磁盘型数据库，成本低。
- 可靠性10个9，数据更可靠，更不易丢失。
- 大V和普通用户的切分阈值更高，读取大V的次数更少，整体延时更低。

缺陷及优化方法

如果使用大V/普通用户的切分方式，大V使用拉模式，普通用户使用推模式，那么这种架构就会存在一种很大的风险。例如某个大V突然发了一个很有话题性的Feed，那么就有可能导致整个Feed产品中的所有用户都没法读取新内容，以下是粉丝读取流程：

1. 大V发送Feed消息。
2. 大V使用拉模式。
3. 大V的活跃粉丝（用户群A）开始通过拉模式读取大V的新Feed。
4. Feed内容太有话题性，快速传播。
5. 未登录的大V粉丝（用户群B）开始登录产品，登录进去后自动刷新，再次通过读3步骤读取大V的Feed内容。
6. 非粉丝（用户群C）去大V的个人页Timeline里面去围观，再次需要读取大V个人的Timeline，同读3。

结果就是，平时正常流量只有用户群A，结果现在却是用户群A + 用户群B + 用户群C，流量增加了好几倍，甚至几十倍，导致读3路径的服务模块被打到server busy或者机器资源被打满，导致读取大V的读3路径无法返回请求，如果Feed产品中的用户都有关注大V，那么基本上所有用户都会卡死在读取大V的读3路径上，然后就没法刷新了。

所以设计时需重点关心下面两点：

- 单个模块的不可用，不应该阻止整个关键的读Feed流路径。如果大V的无法读取，但是普通用户的要能返回，等服务恢复后，再补齐大V的内容即可。
- 当模块无法承受这么大流量的时候，模块不应该完全不可服务，而应该能继续提供最大的服务能力，超过的拒绝掉。

优化方法

- 方法一：不使用大V/普通用户的优化方式，而使用活跃用户/非活跃用户的优化方式，这种优化方式能把用户群A和部分用户群B分流到其他更分散的多个路径上去。而且即使读3路径不可用，仍然对活跃用户无任何影响。
- 方法二：完全使用推模式可以彻底解决这个问题，但会增大存储量，并增长大V微博发送总时间，从发给第一个粉丝到发给最后一个粉丝可能要几分钟时间（一亿粉丝，100万行每秒，需要100秒），还需要为最大并发预留好资源（如果使用阿里云表格存储，则不需要考虑预留最大额度资源的问题）。

1.5. 表格存储Feed流方案实现

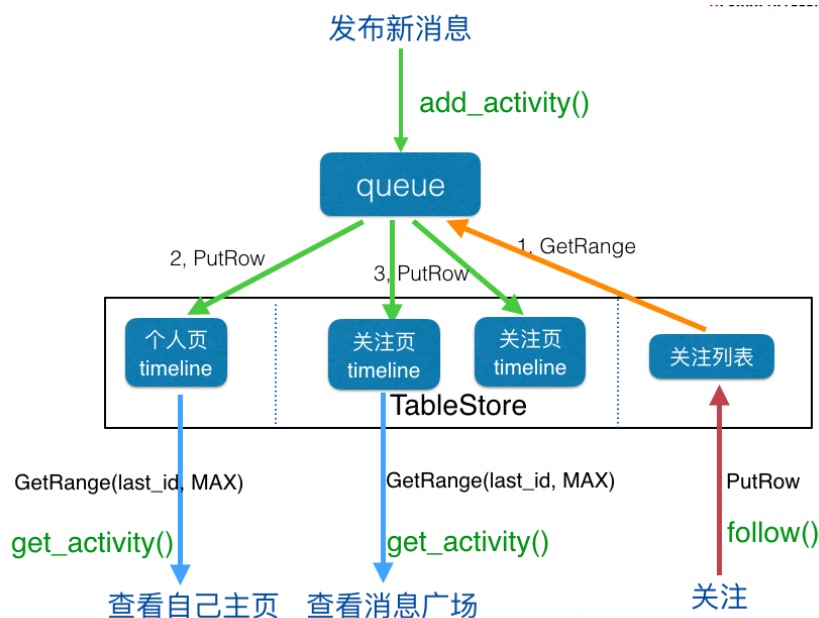
本文主要介绍如何使用表格存储实现一个消息广场功能。很多应用中都有动态或消息广场的功能，在消息广场中一般有两个Tab：一个是关注人，一个是广场。本章节重点分析关注人Tab。

背景

- 关注人Tab需要实现的功能
 - 关注：用户之间可以相互关注。
 - 发布：用户可以发布新消息。
 - 查看（获取Feed流消息）：用户可以查看自己发布的消息列表。用户可以查看自己关注的人的消息。
- 方案
 - 使用表格存储作为存储和推送系统。
 - 采用Timeline的显示方式，希望用户可以认真看每条Feed。
 - 采用推模式。

- 角色及角色功能
 - 发送者功能
 - 发送状态：add_activity()
 - 接收者功能
 - 关注：follow()
 - 读取Feed流：get_activity()
 - 消息
 - Feed消息中至少需要包括以下内容：
 - 发送人：actor
 - 类型：verb，例如图片、视频、文本
 - 文本文字：message

功能实现



您可以使用表格存储API实现以下功能：

- 发布新消息
 - 接口：add_activity()
 - 实现：使用get_range接口调用关注列表，返回粉丝列表。使用batch_write_row接口将feed内容和ID批量写入个人页表（发件箱）和所有粉丝的关注页表（收件箱），如果量太大，可以多次写入。或者调用异步batch_write_row接口，目前C++ SDK和JAVA SDK提供异步接口。
- 关注
 - 接口：follow()
 - 实现：使用put_row接口直接写入一行数据（关注人，粉丝）到关注列表和粉丝列表（粉丝，关注人）即可。
- 获取Feed流消息
 - 接口：get_activity()

- 实现：从客户端获取上次读取到的最新消息的ID: last_id。使用get_range接口读取最新的消息，起始位置是last_id，结束位置是MAX。如果是读取个人页的内容，访问个人页表即可。如果是读取关注页的内容，访问关注页表即可。

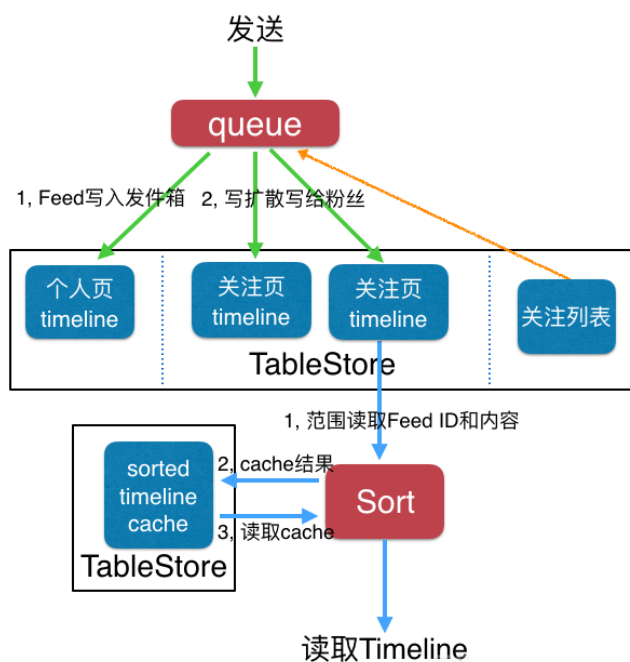
② 说明

- 各SDK接口参见[SDK概览](#)。
- 各API接口参见[操作汇总](#)。

1.6. 方案扩展

除了Timeline类型的Feed流类型，Rank类型的Feed流也比较常见。Rank类型的潜在Feed内容非常多，用户无法全部看完，也不需要全部看完，则需要为用户选出最想看到的内容。Rank类型典型的应用包括图片分享网站、新闻推荐网站等。本文主要为您介绍两种常见的实现Rank方式。

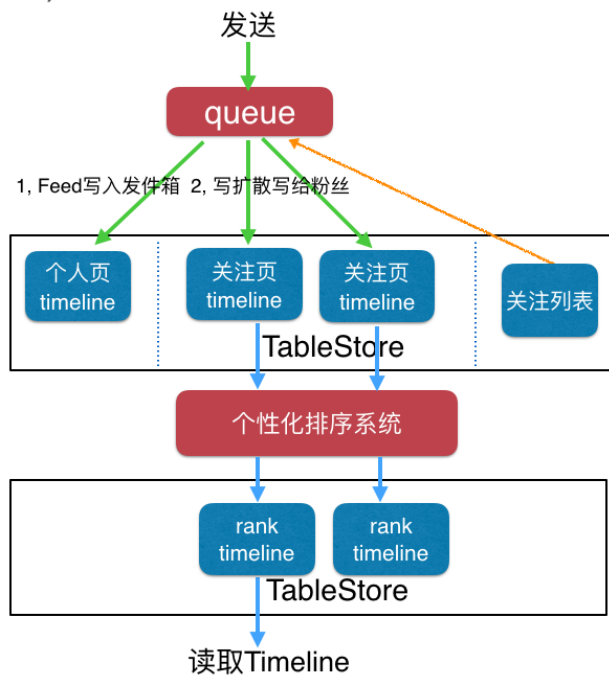
方式一（轻量级）



上面架构图展示的Rank方式比较轻量级，适用于推拉结合的场景。

- 写流程和Timeline基本一样。
- 读流程里面会先读取所有的Feed内容，这个和Timeline一样。Timeline中这部分会直接返回给用户，但是Rank类型需要在一个排序模块里面，按照某个属性值排序，然后将所有结果存入一个timeline cache中，并返回分数最高的N个结果，下次读取的时候再返回[N+1, 2N]的结果。

方式二（重量级）



上面架构图展示的Rank方式比较重量级，适用于纯推模式。

- 写流程和Timeline一样。
- 每个用户有两个收件箱。
 - 一个是关注页Timeline，保存原始的Feed内容，用户无法直接查看这个收件箱。
 - 一个是rank timeline，保存为用户精选的Feed内容，用户直接查看这个收件箱。
- 写流程结束后还有一个数据处理的流程。个性化排序系统从原始Feed收件箱中获取到新的Feed内容，按照用户的特征、Feed的特征计算出一个分数。每个Feed在不同用户的Timeline中可能分数不一样的，计算完成后再排序然后写入最终的rank timeline。

2.如何优化高并发IM系统架构

介绍如何使用表格存储的主键增列功能优化高并发IM系统架构。

背景

在构建社交IM和朋友圈应用时，最基本的需求是将用户发送的消息和朋友圈的更新及时、准确地更新给该用户的好友。这需要为用户发送的每一条消息或者朋友圈更新设置一个序号或者ID，并且保证递增，这个机制可以确保所有的消息能够按照正确的顺序被接收端处理。

高并发的IM系统通常选择NoSQL数据库存储产品来存储消息，但常见的NoSQL产品没有提供自增列的功能，通常要借助外部组件来实现消息序号和ID的递增，使得整体的架构更加复杂，且影响整条链路的延时。

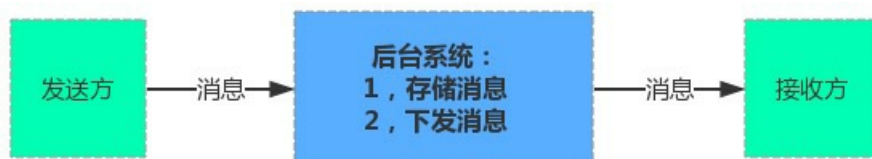
功能需求

IM系统通常需要实现下列功能：

- 支持用户一对一聊天
- 支持用户群组内聊天
- 支持同一个用户的多终端消息同步

现有架构

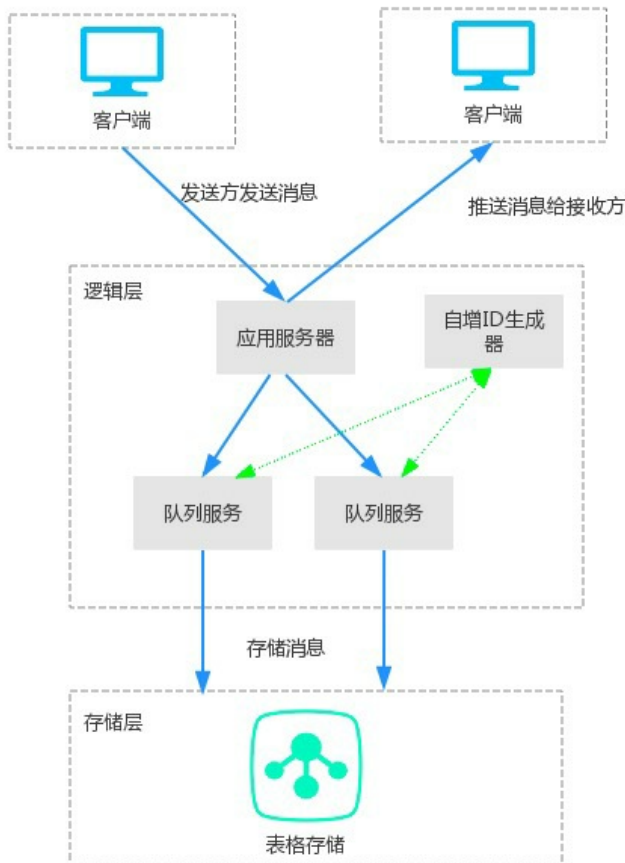
- 消息模型



消息模型的实现顺序如下。

- i. 发送方发送了一条消息后，消息会被客户端推送给后台系统。
- ii. 后台系统会先存储消息。
- iii. 存储成功后，会推送消息给接收方的客户端。

- 后台架构



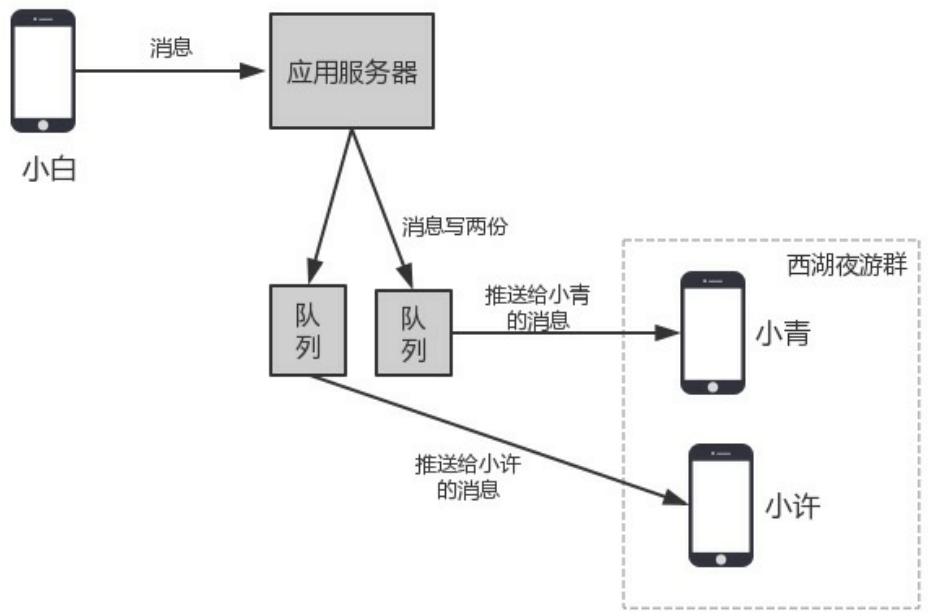
后台架构主要分为两部分：逻辑层和存储层。

- 逻辑层包括应用服务器、队列服务和自增ID生成器，是整个后台架构的核心，处理消息的接收、推送、通知、群消息写复制等核心业务逻辑。
- 存储层主要是用来存储持久化消息数据和其他一些需要持久化的数据。

对于一对一聊天，发送方发送消息给应用服务器后，应用服务器将消息存储到接收方为主键的表中，同时通知应用服务器中的消息推送服务，消息推送服务会将上次推送给接收方的最后一条消息的ID作为起始主键，从存储系统中读取之后的所有消息，然后将消息推送给接收方。

对于群组内的聊天，逻辑会更加复杂，需要通过异步队列来完成消息的扩散写，即发到群组内的一条消息会给群组内的每个人都保存一份。

下图展示了省略掉存储层后的群消息发送过程。



使用扩散写而非扩散读，主要由于以下两点原因：

- 群组内成员一般都不多，存储成本并不高，而且有压缩，成本更低。
- 消息扩散写到每个人的存储表（收件箱）后，为每个接收方推送消息时，只需要检查自己的收件箱即可，此时群聊和单聊的处理逻辑一样，实现简单。

● 存储系统

存储系统采用阿里云表格存储，表格存储具有以下优势：

- 表格存储写操作不仅支持单行写，也支持多行批量写，可满足大并发写数据需求。
- 表格存储支持按范围读，消息多时可翻页。
- 表格存储支持数据生命周期管理，可对过期数据进行自动清理，节省存储费用。
- 表格存储价格便宜，且稳定可靠。
- 表格存储读写性能极佳，对于聊天消息，延迟基本在毫秒，甚至微秒级别。

表格存储表结构的主键列部分请参见下表。

主键顺序	主键名称	主键值	说明
1	partition_key	md5 (receive_id) 前4位	分区键保证数据均匀分布
2	receive_id	receive_id	接收方的用户ID
3	message_id	message_id	消息ID

表格存储表结构包括主键列和属性列两部分。

◦ 主键列

- 最多支持4个主键列，第一个主键为分区键。
- 通过分区键可以让数据和请求均衡分布、避免热点。由于最终读取消息时要按照接收方读取，所以此处可以使用接收方ID作为分区键。为了更加均衡，可以使用接收方ID的md5值的部分区域，例如前4个字符，这样就可以将数据均衡分布了。
- 第一个主键只用了部分接收方ID，为了能定位到接收方的消息，需要保存完整的接收方ID，所以可以将接收方ID作为第二个主键。
- 第三个主键可以是消息ID，由于需要查询最新的消息，这个值需要是单调自增的。

 **说明** 主键列结构在使用过程中不能修改。

◦ 属性列

属性由多个属性列组成。每行的属性列个数没有限制，即每行的属性列可不同。一个属性列在某一行的值可为空。同一个属性列的值可以有多种数据类型。属性列可以保存消息内容和元数据等。

● 挑战

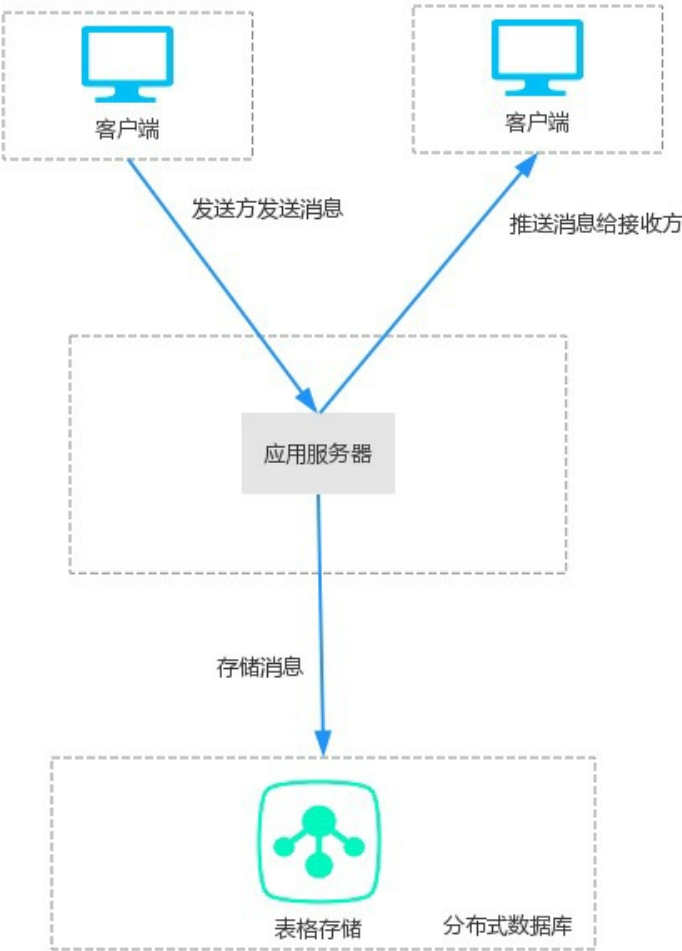
此架构虽可应用于高并发IM系统，但仍面临挑战。多个用户在一个队列中，队列串行执行，为了保证消息严格递增，执行过程中要持有锁，这个过程会有一个风险：如果发送给某个用户的消息量很大，这个用户所在的队列中消息会变多，就有可能堵塞其他用户的消息，导致同队列的其他用户消息出现延迟。

新架构

现有架构存在的挑战可通过使用表格存储的主键列自增功能轻松解决。

使用了表格存储主键列自增功能后的新架构特点如下：

- 与原架构相比，新架构最明显的区别是减少了队列服务和自增ID生成器两个组件，架构更加简单。
- 应用服务器接收到消息后，直接将消息写入表格存储，对于主键自增列message_id，在写数据时不需要填写具体的值，只需要填充一个特定的占位符即可，此值会在表格存储系统内部自动生成。
- 新架构中自增操作是在表格存储系统内部处理的，就算多个应用服务器同时给表格存储中的同一个接收方写数据，表格存储内部也能保证这些消息是串行处理，每个消息都有一个独立的消息ID，且严格递增。使用主键列自增功能，就不再需要队列服务，这样也就彻底解决了原架构的问题。
- 原架构只能有一个队列处理同一个用户的消息，新架构可以多个队列并行处理，当一些用户的消息量突然变大时，也不会立即堵塞其他用户，而是将压力均匀分布给了所有队列。
- 使用主键自增列功能后，应用服务器可以直接写数据到表格存储，不再需要经过队列和获取消息ID，性能表现会更加优秀。



方案实现

本文使用Java SDK实现优化方案。

说明 Java SDK 4.2.0版本已经支持主键列自增功能。

1. 创建数据表

数据表结构的主键列部分请参见下表。

主键顺序	主键名称	主键值	说明
1	partition_key	hash (receive_id) 前4位	分区键，保证数据均匀分布，可以使用md5作为hash函数
2	receive_id	receive_id	接收方的用户ID
3	message_id	message_id	消息ID

上表中，第三列主键（PK）是message_id，这一列是主键自增列，建表时指定message_id列的属性为AUTO_INCREMENT，且类型为INTEGER。

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta("message_table");
    //第一列为分区建。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("partition_key", PrimaryKeyType.STRING));
    //第二列为接收方ID。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("receive_id", PrimaryKeyType.STRING));
    //第三列为消息ID，自动自增列，类型为INTEGER，属性为PKO_AUTO_INCREMENT。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("message_id", PrimaryKeyType.INTEGER, PrimaryKeyOption.AUTO_INCREMENT));
    int timeToLive = -1; //永不过期，也可以设置数据有效期，过期了会自动删除。
    int maxVersions = 1; //只保存一个版本，目前支持多版本。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
    client.createTable(request);
}
```

完成以上操作后即创建了一个第三列PK为自动自增的表。

2. 写数据

您可以使用PutRow和BatchWriteRow接口写数据，这两个接口都支持主键列自增功能。写数据时，第三列message_id是主键自增列，这一列不需要填值，只需要填入占位符即可。

```
private static void putRow(SyncClient client, String receive_id) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
    ();
    //第一列的值为hash(receive_id)前4位。
    primaryKeyBuilder.addPrimaryKeyColumn("partition_key", PrimaryKeyValue.fromString(
    hash(receive_id).substring(4)));
    //第二列的值为接收方ID。
    primaryKeyBuilder.addPrimaryKeyColumn("receive_id", PrimaryKeyValue.fromString(
    receive_id));
    //第三列是消息ID，主键递增列，这个值是表格存储产生的，用户在这里不需要填入真实值，只需要一
    个占位符AUTO_INCREMENT即可。
    primaryKeyBuilder.addPrimaryKeyColumn("message_id", PrimaryKeyValue.AUTO_INCREM
    ENT);
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    RowPutChange rowPutChange = new RowPutChange("message_table", primaryKey);
    //此处设置返回类型为RT_PK，表示是在返回结果中包含PK列的值。如果不设置ReturnType，默认不
    返回。
    rowPutChange.setReturnType(ReturnType.RT_PK);
    //加入属性列，消息内容。
    rowPutChange.addColumn(new Column("content", ColumnValue.fromString(content)));
    //写数据到表格存储。
    PutRowResponse response = client.putRow(new PutRowRequest(rowPutChange));
    //打印出返回的PK列。
    Row returnRow = response.getRow();
    if (returnRow != null) {
        System.out.println("PrimaryKey:" + returnRow.getPrimaryKey().toString());
    }
    //打印出消耗的CU。
    CapacityUnit cu = response.getConsumedCapacity().getCapacityUnit();
    System.out.println("Read CapacityUnit:" + cu.getReadCapacityUnit());
    System.out.println("Write CapacityUnit:" + cu.getWriteCapacityUnit());
}
```

3. 读数据

您可以通过GetRange接口读取最近的消息。message_id这一列PK的起始位置是上一条消息的message_id+1，结束位置是INF_MAX，这样每次都可以读出最新的消息，然后发送给客户端。

```
private static void getRange(SyncClient client, String receive_id, String lastMessageId) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria("message_table");
    //设置起始主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    //第一列的值为hash(receive_id)前4位。
    primaryKeyBuilder.addPrimaryKeyColumn("partition_key", PrimaryKeyValue.fromString(hash(receive_id).substring(4)));
    //第二列的值为接收方ID。
    primaryKeyBuilder.addPrimaryKeyColumn("receive_id", PrimaryKeyValue.fromString(receive_id));
    //第三列的值为消息ID，起始于上一条消息。
    primaryKeyBuilder.addPrimaryKeyColumn("message_id", PrimaryKeyValue.fromLong(lastMessageId + 1));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    //设置结束主键。
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    //第一列的值为hash(receive_id)前4位。
    primaryKeyBuilder.addPrimaryKeyColumn("partition_key", PrimaryKeyValue.fromString(hash(receive_id).substring(4)));
    //第二列的值为接收方ID。
    primaryKeyBuilder.addPrimaryKeyColumn("receive_id", PrimaryKeyValue.fromString(receive_id));
    //第三列的值为消息ID。
    primaryKeyBuilder.addPrimaryKeyColumn("message_id", PrimaryKeyValue.INF_MAX);
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange的结果为:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

3.搭建海量智能元数据管理系统

3.1. 方案背景

本文主要为您介绍搭建海量智能元数据管理系统的技术点以及使用表格存储搭建智能元数据管理系统的方案。

用户存储海量的文档、媒体文件等数据的同时，对文件元数据（Meta）的管理不可或缺。元数据拥有多维度的字段信息，基本信息包含文件大小、创建时间、用户等。随着人工智能的发展，通过AI技术提取文件核心要素也成为文件元数据的重要信息。以图片为例：用户通过智能媒体服务，获取分析图片核心标签并为标签打分，用户还可提取人脸识别相关信息，以及地理位置等信息，提取的信息也需要存储到文件元数据信息中。因而文件元数据的信息量不断增加，格式、类型也不断呈现多元化。

需求场景

某智能媒体管理平台，为用户提供文件（图片、视频等）管理服务，用户通过自研（或售卖）的智能媒体分析工具，为目标文件进行分析。用分析后的信息丰富原有的元数据信息。因此，平台需要一套有效的元数据管理方案，为用户提供元数据信息的管理、分析、统计功能。例如：

用户A：（用户A的文件）*（近1年）*（标签含[开心]）*的所有图片，按标签分数排序

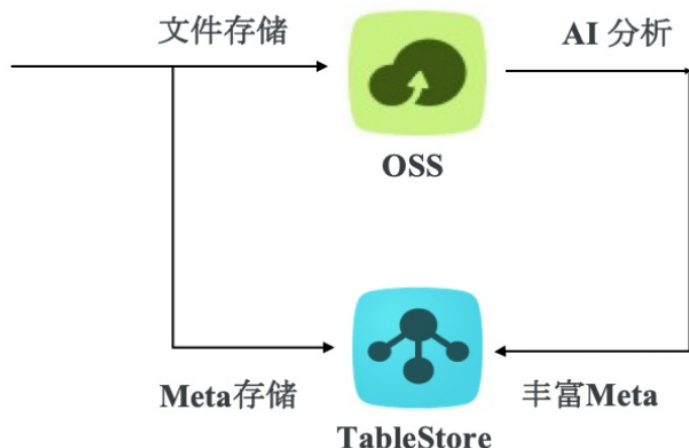
用户B：（用户B的文件）*（出现某某明星）*的所有视频，按明星相似度排序

.....

技术点

对于智能元数据管理系统，通常需要考虑的技术点，包含以下方面：

- 查询能力：具备强大的查询能力，如多类型索引、多维度组合查询等，同时具备排序、统计等功能。
- 横向扩展（多字段）：元数据的字段类型丰富，字段变动、增删频繁，数据库尽量schema free来保证横向扩展能力。
- 纵向扩展（数据量）：海量文件就会对应海量元数据，面对数据膨胀，数据库要满足易扩展、低成本等基本要求。
- 服务性能：应对高并发请求的同时，保证低延迟、强一致、高可用。



表格存储方案

表格存储作为阿里云提供的一款全托管、分布式NoSql型数据存储服务，具有【海量数据存储】、【热点数据自动分片】、【海量数据多维检索】等功能，天然地解决了数据大爆炸这一挑战；在应对数据横向、纵向扩展上，充分发乎其优势。多元索引随时创建，是Meta元数据管理的合适方案。同时，SearchIndex功能在保证用户数据高可用的基础上，提供了数据多维度搜索、统计等能力。针对多种场景创建多种索引，实现多种模式的检索。用户可以仅在需要的时候创建、开通索引。由表格存储来保证数据同步的一致性，这极大的降低了用户的方案设计、服务运维、代码开发等工作量。

样例内嵌在表格存储控制台中，用户可登录控制台体验系统（若为表格存储的新用户，需要点击开通服务后体验，开通免费，Meta数据存储于公共实例中，体验不消耗用户存储、流量、CU）。

管理控制台

消息 0/0 费用 工单 备案 企业 支持与服务 简体中文

项目样例

项目说明 图片搜索 文件多维搜索

用户编号: 文件编号: 创建时间 (起): 创建时间 (止): 标签: 文件类型: (全部) 排序方式: 逆序

标签: 文件标签 文件类型: (全部) 排序方式: 标签分数

搜索 清除

用户	文件 ID	类型	标签	图片大小	创建时间	图片链接
u01699	I0061832	图片	文件标签: 0.999999% 冰棍: 0.902224% 开心: 0.326385% 雪山: 0.686812%	9.02 MB	2018-07-17 22:26:12	点击查看
u05254	I052535742	视频	文件标签: 0.999997% 伤心: 0.062191% 雪山: 0.823651% 台风: 0.730322%	11.86 MB	2018-07-17 22:12:40	点击查看
u00216	I002152283	视频	文件标签: 0.999999% 沙漠: 0.709892% 享受: 0.495811% 沙尘暴: 0.930024%	3.55 MB	2018-07-17 23:43:41	点击查看
u00619	I006189991	图片	文件标签: 0.999999% 小鸟: 0.213583% 好奇: 0.546039% 蓝天: 0.232827%	6.54 MB	2018-07-17 21:26:18	点击查看
u05322	I053211157	图片	文件标签: 0.999999% 大草原: 0.7412447% 多云: 0.2400226% 蓝天: 0.53261115%	7.43 MB	2018-07-17 22:17:10	点击查看
u09248	I092471473	视频	文件标签: 0.999999% 戈壁滩: 0.500050% 沙尘暴: 0.666814% 伤心: 0.104247%	11.96 MB	2018-07-17 23:45:44	点击查看
u05179	I051789349	图片	文件标签: 0.999999% 晴天: 0.265276% 浪花: 0.291592% 戈壁滩: 0.700722%	809.61 KB	2018-07-17 23:25:22	点击查看
u09308	I093073801	视频	文件标签: 0.999999% 沙漠: 0.69303173% 虎头: 0.448165% 西南: 0.21378%	4.3 MB	2018-07-17 22:45:17	点击查看
u03835	I038444483	视频	文件标签: 0.999994% 雨天: 0.2318609% 戈壁滩: 0.187576%	2.29 MB	2018-07-17 23:31:06	点击查看
u08437	I084367841	图片	文件标签: 0.9999993% 开心: 0.3731887% 天空: 0.042732% 多云: 0.274208%	4.38 MB	2018-07-17 21:07:20	点击查看

共有100000000条, 每页显示: 10条

1 2 3 4

GO

[illegible]

3.2. 准备工作

本章节主要为您介绍使用表格存储搭建海量智能元数据管理系统前的准备工作。

开通表格存储

已开通表格存储服务。具体操作，请参见[开通表格存储服务](#)。

创建表格存储实例

实例是您使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。

创建表格存储实例。具体操作，请参见[创建实例](#)。

下载SDK

选择以下语言的SDK包进行下载安装：

- [Java SDK](#)
- [Python SDK](#)
- [Go SDK](#)
- [Node.js SDK](#)
- [.NET SDK](#)
- [PHP SDK](#)

设计数据表

表名：order_contract

列名	数据类型	索引类型	字段说明
_id (主键列)	String		MD5(fid)避免热点
fid	String	KEYWORD	文件编号
userId	String	KEYWORD	用户编号
tags	String	Nested: [{tag: String, score: LONG}]	多标签使用嵌套索引（数组字符串）' [{"tag": "表格存储", "score": 97.317251}, {"score": 50.770918, "tag": "沙漠"}]'
size	long	LONG	文件大小
createdAt	long	LONG	创建时间（时间戳）
url	String	KEYWORD	文件链接（存储于oss）
...	...	...	...

3.3. 方案实现

本章节主要为您介绍如何使用表格存储搭建海量智能元数据管理系统。

前提条件

您已经完成了准备工作。

步骤一： 创建数据表

通过以下两种方式创建一张店铺信息表：

- 通过控制台创建、管理数据表，具体参见[创建数据表](#)。
- 通过SDK直接创建、管理数据表。

步骤二： 创建数据表索引

Tablestore自动进行全量、增量的索引数据同步。您可以通过以下两种方式创建、管理多元索引：

- 通过控制台创建、管理多元索引，具体参见[使用控制台](#)。
- 通过SDK创建、管理多元索引。

步骤三： 导入数据

控制台[项目样例](#)中插入了1亿条数据，您可以通过控制台插入少量测试数据。

文件编号	文件ID (md5主键)	用户编号	标签 (数组字符串)	类型	链接	大小
f052535742	1bce....	u05253574	[{"score":99.999999,"tag":"表格存储"}, {"score":78.962224,"tag":"冰雪"}, {"score":18.328385,"tag":"开心"}, {"score":16.886812,"tag":"雪山"}]	image	https://prd-consol-e-demo.oss-cn-hangzhou.aliyuncs.com/image/imm1.jpg	9022066

步骤四： 读取数据

数据读取分为两类：

- 主键读取
基于原生表格存储的主键列获取：getRow, getRange, batchGetRow等。主键读取用于索引（自动）反查，用户也可以提供主键（文件编号md5）的单条查询的页面，亿量级下查询速度保持在十毫秒量级。单主键查询方式不支持多维度检索。
- 索引读取

基于新SearchIndex功能Query: search接口。用户可以自由设计索引字段的多维度条件组合查询。通过设置选择不同的查询参数, 构建不同的查询条件、不同排序方式; 目前支持: 精确查询、范围查询、前缀查询、匹配查询、通配符查询、短语匹配查询、分词字符串查询、嵌套查询、GEO查询, 并通过布尔与、或组合。

例如, 标签为: 表格存储, 创建时间为: [2018-01-01, 2018-12-01), 文件的信息: (SDK与控制查询)

```
List<Query> mustQueries = new ArrayList<Query>();
//嵌套字段Query
TermQuery termQuery = new TermQuery();
termQuery.setFieldName("tags.tag");
termQuery.setTerm(ColumnValue.fromString("表格存储"));
NestedQuery nestedQuery = new NestedQuery();
nestedQuery.setPath("tags");
nestedQuery.setScoreMode(ScoreMode.Avg);
nestedQuery.setQuery(termQuery);
mustQueries.add(nestedQuery);
//范围Query
RangeQuery rangeQuery = new RangeQuery();
rangeQuery.setFieldName("createdAt");
rangeQuery.setFrom(ColumnValue.fromLong(1514793600000, true);
rangeQuery.setTo(ColumnValue.fromLong(1543651200000, false);
mustQueries.add(rangeQuery);
//精确Query
TermQuery termQuery = new TermQuery();
termQuery.setFieldName("type");
termQuery.setTerm(ColumnValue.fromString("image"));
mustQueries.add(termQuery);
BoolQuery boolQuery = new BoolQuery();
boolQuery.setMustQueries(mustQueries);
```

数据查询

* 实例名:

console-demo-prd

* 表名:

meta_file

* 索引名:

meta_file_index

是否排序:

返回哪些列:

column1,column2

全部列:

✕ 移除所有条件

type

增加And条件

字段名	字段类型	条件类型	条件	操作
tags tag	嵌套文档字符串	精确查询	表格存储	
createdAt	长整型	范围查询	1514793600000 < To	
type	字符串	精确查询	image	

确定

取消

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

4.现代IM系统中的消息系统

4.1. 现代IM系统中的消息系统—架构

本章节主要介绍现代IM系统中的消息系统架构以及基于表格存储（Tablestore）自研的Timeline模型构建的消息系统。基于Timeline构建的现代消息系统能够同时支持消息系统的多种特性，包括多端同步、消息漫游和在线检索，在性能和规模上能够实现全量消息云端存储和索引、百万TPS写入以及毫秒级延迟的消息同步和检索能力。

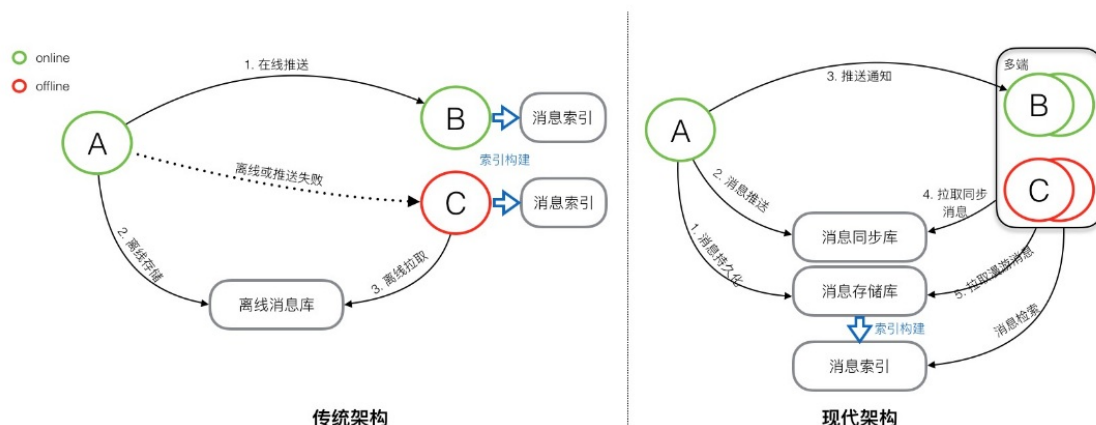
背景

在这个高度信息化的移动互联网时代，IM（Instant Messaging，即时通讯）类产品已经成为生活必备品，例如钉钉、微信、QQ。微信已经成长为一个生态型产品，但其核心功能仍是IM。此外，还有一些非以IM系统为核心的应用也经常出现在我们的娱乐社交生活，例如在线游戏、社交应用。IM系统已经是任何一个带有社交属性的应用需要具备的基础功能。

IM系统在互联网初期即存在，其基础技术架构在这十几年的发展中更新迭代多次，从早期的CS、P2P架构，到现在后台已经演变为一个复杂的分布式系统，涉及移动端、网络通信、协议、安全、存储和搜索等技术的方方面面。IM系统中最核心的部分是消息系统，消息系统中最核心的功能是消息同步、存储和检索。

- **消息同步**：指将消息完整的、快速的从发送方传递到接收方。消息同步系统最重要的衡量指标就是消息传递的实时性、完整性以及能支撑的消息规模。在功能上，至少要支持在线和离线推送，有些IM系统还支持多端同步。
- **消息存储**：指消息的持久化保存。传统消息系统通常只能支持消息在接收端的本地存储，数据基本不具备可靠性。现代消息系统能支持消息在服务端的在线存储，功能上对应的就是消息漫游，消息漫游的好处是可以实现账号在任意端登录查看所有历史消息。
- **消息检索**：消息一般是文本，所以支持全文检索也是必备的能力之一。传统消息系统通常来说也是只能支持消息的本地检索，基于本地存储的消息数据来构建。而现在消息系统在能支持消息的在线存储后，也具备了消息的在线检索能力。

传统架构 vs 现代架构



- 传统架构

传统架构下，消息是先同步后存储。对于在线的用户，消息会直接实时同步到在线的接收方，消息同步成功后，并不会在服务端持久化。而对于离线的用户或者消息无法实时同步成功时，消息会持久化到离线库，当接收方重新连接后，会从离线库拉取所有未读消息。当离线库中的消息成功同步到接收方后，消息会从离线库中删除。传统的消息系统，服务端的主要工作是维护发送方和接收方的连接状态，并提供在线消息同步和离线消息缓存的能力，保证消息一定能够从发送方传递到接收方。服务端不会对消息进行持久化，所以也无法支持消息漫游。消息的持久化存储及索引同样只能在接收端本地实现，数据可靠性极低。

- 现代架构

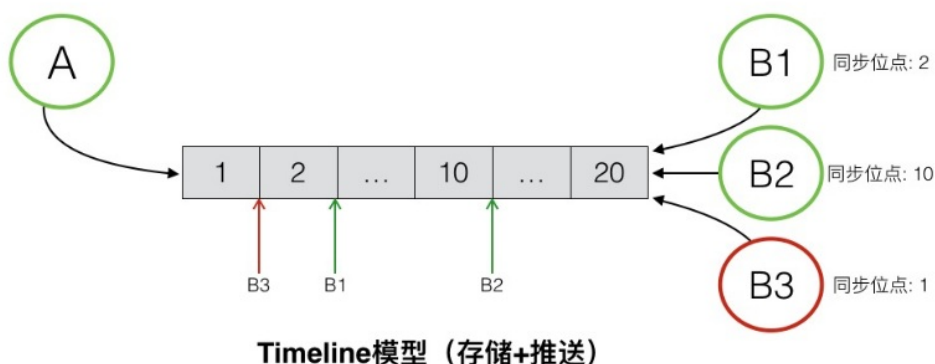
现代架构下，消息是先存储后同步。先存储后同步的好处是，如果接收方确认接收到了消息，那这条消息一定是已经在云端保存了。并且消息会有两个库来保存，一个是消息存储库，用于全量保存所有会话的消息，主要用于支持消息漫游。另一个是消息同步库，主要用于接收方的多端同步。消息从发送方发出后，经过服务端转发，服务端会先将消息保存到消息存储库，后保存到消息同步库。完成消息的持久化保存后，对于在线的接收方，会直接选择在线推送。但在线推送并不是一个必须路径，只是一个更优的消息传递路径。对于在线推送失败或者离线的接收方，会有另外一个统一的消息同步方式。接收方会主动的向服务端拉取所有未同步消息，但接收方何时来同步以及会在哪些端来同步消息对服务端来说是未知的，所以要求服务端必须保存所有需要同步到接收方的消息，这是消息同步库的主要作用。对于新的同步设备，会有消息漫游的需求，这是消息存储库的主要作用，在消息存储库中，可以拉取任意会话的全量历史消息。消息检索的实现依赖于对消息存储库内消息的索引，通常是一个近实时（NRT，near real time）的索引构建过程，这个索引同样是在线的。

以上是传统架构和现代架构的一个简单的对比。现代架构上整个消息的同步、存储和索引流程，并没有变复杂太多。现代架构的实现本质上是把传统架构内本地存储和索引都搬到云上，最大挑战是需要集中管理全量消息的存储和索引，优点是能实现多端同步、消息漫游以及在线检索。现代架构中最核心的就是两个消息库：消息同步库、消息存储库，以及对这两个消息库的实现。

Timeline模型

在深入讲解消息系统的设计和实现之前，需要先了解消息系统内的基本概念和基础模型。表格存储的 Timeline 是一个对消息系统内消息模型的一个抽象，能简化和更好的让开发者理解消息系统内的消息同步和存储模型，基于此模型我们会再深入探讨消息的同步和存储的选择和实现。

Timeline模型是一个对消息抽象的逻辑模型，该模型会帮助我们简化对消息同步和存储模型的理解，而消息同步库和存储库的设计和实现也是围绕Timeline的特性和需求来展开。



上图是Timeline模型的一个抽象表述，Timeline可以简单理解为一个消息队列，但这个消息队列有如下特性。

- 每条消息对应一个顺序ID：每个消息拥有一个唯一的顺序ID（SequenceId），队列消息按SequenceId排序。
- 新消息写入能自动分配递增的顺序ID，保证永远插入队尾：Timeline中是根据同步位点也就是顺序ID来同步消息，所以需要保证新写入的消息数据的顺序ID绝对不能比已同步的消息的顺序ID还小，否则会导致数据漏同步，所以需要支持对新写入的数据自动分配比当前已存储的所有消息的顺序ID更大的顺序ID。

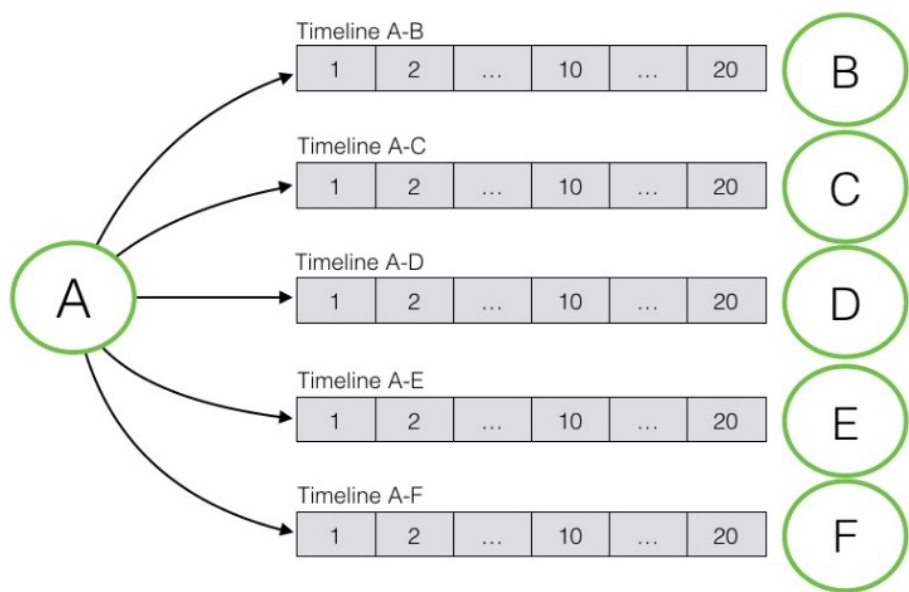
- 新消息写入也能自定义顺序ID，满足自定义排序需求：上面提到的自动分配顺序ID，主要是为了满足消息同步的需求，消息同步要求消息是根据『已同步』或是『已写入』的顺序来排序。而消息的存储，通常要求消息能根据会话顺序来排序，会话顺序通常由端的会话来决定，而不是服务端的同步顺序来定，这是两种顺序要求。
- 支持根据顺序ID的随机定位：可根据SequenceId随机定位到Timeline中的某个位置，从这个位置开始正序或逆序的读取消息，也可支持读取指定顺序ID的某条消息。
- 支持对消息的自定义索引：消息体内数据根据业务不同会包含不同的字段，Timeline需要支持对不同字段的自定义索引，来支持对消息内容的全文索引，或者是任意字段的灵活条件组合查询。

消息同步可以基于Timeline很简单的实现。图中的例子中，消息发送方是A，消息接收方是B，同时B存在多个接收端，分别是B1、B2和B3。A向B发送消息，消息需要同步到B的多个端，待同步的消息通过一个Timeline来进行交换。A向B发送的所有消息，都会保存在这个Timeline中，B的每个接收端都是独立的从这个Timeline中拉取消息。每个接收端同步完毕后，都会在本地记录下最新同步到的消息的SequenceId，即最新的一个位点，作为下次消息同步的起始位点。服务端不会保存各个端的同步状态，各个端均可以在任意时间从任意点开始拉取消息。

消息存储也是基于Timeline实现，和消息同步唯一的区别是，消息存储要求服务端能够对Timeline内的所有数据进行持久化，并且消息采用会话顺序来保存，需要自定义顺序ID。

消息检索基于Timeline提供的消息索引来实现，能支持比较灵活的多字段索引，根据业务的不同可有自由度较高的定制。

消息存储模型

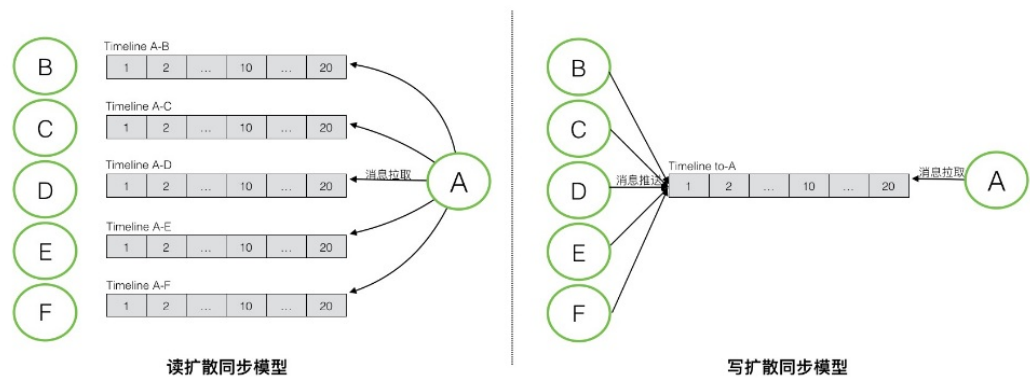


基于Timeline的消息存储模型

上图是基于Timeline的消息存储模型，消息存储要求每个会话都对应一个独立的Timeline。如图例子所示，A与B/C/D/E/F均发生了会话，每个会话对应一个独立的Timeline，每个Timeline内存有这个会话中的所有消息，消息根据会话顺序排序，服务端会对每个Timeline进行持久化存储，也就拥有了消息漫游的能力。

消息同步模型

消息同步模型会比消息存储模型稍复杂一些，消息的同步一般有读扩散（也叫拉模式）和写扩散（也叫推模式）两种不同的方式，分别对应不同的Timeline物理模型。



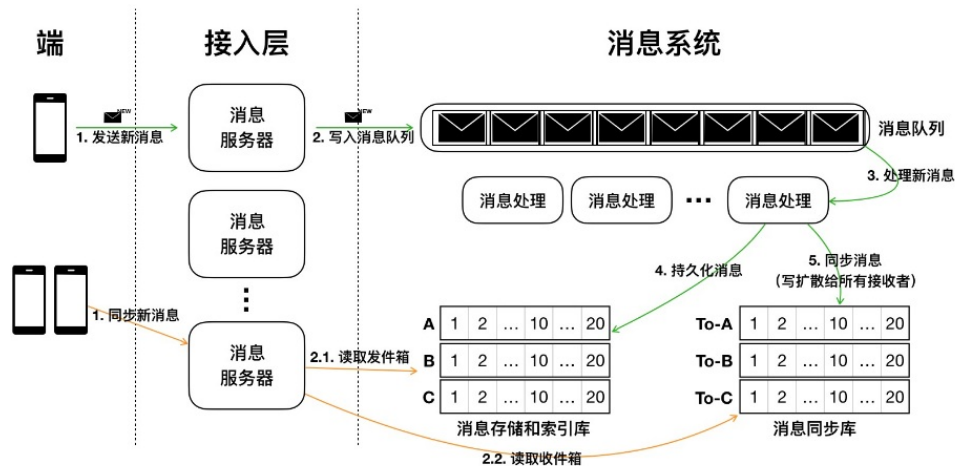
如图是读扩散和写扩散两种不同同步模式下对应的不同的Timeline模型。按图中的示例，A作为消息接收者，其与B/C/D/E/F发生了会话，每个会话中的新的消息都需要同步到A的某个端。

- 读扩散：消息存储模型中，每个会话的Timeline中保存了这个会话的全量消息。读扩散的消息同步模式下，每个会话中产生的新的消息，只需要写一次到其用于存储的Timeline中，接收端从这个Timeline中拉取新的消息。优点是消息只需要写一次，相比写扩散的模式，能够大大降低消息写入次数，特别是在群消息这种场景下。但其缺点也比较明显，接收端去同步消息的逻辑会相对复杂和低效。接收端需要对每个会话都拉取一次才能获取全部消息，读被大大的放大，并且会产生很多无效的读，因为并不是每个会话都会有新消息产生。
- 写扩散：写扩散的消息同步模式，需要有一个额外的Timeline来专门用于消息同步，通常是每个接收端都会拥有一个独立的同步Timeline（或者叫收件箱），用于存放需要向这个接收端同步的所有消息。每个会话中的消息，会产生多次写，除了写入用于消息存储的会话Timeline，还需要写入需要同步到的接收端的同步Timeline。在个人与个人的会话中，消息会被额外写两次，除了写入这个会话的存储Timeline，还需要写入参与这个会话的两个接收者的同步Timeline。而在群这个场景下，写入会被更加的放大，如果这个群拥有N个参与者，那每条消息都需要额外的写N次。写扩散同步模式的优点是，在接收端消息同步逻辑会非常简单，只需要从其同步Timeline中读取一次即可，大大降低了消息同步所需的读的压力。其缺点就是消息写入会被放大，特别是针对群这种场景。

Timeline模型不会对选择读扩散还是写扩散做约束，而是能同时支持两种模式，因为本质上两种模式的逻辑数据模型并无差别，只是消息数据是用一个Timeline来支持多端读还是复制到多个Timeline来支持多端读的问题。

针对IM这种应用场景，消息系统通常会选择写扩散这种消息同步模式。IM场景下，一条消息只会产生一次，但是会被读取多次，是典型的读多写少的场景，消息的读写比例大概是10:1。若使用读扩散同步模式，整个系统的读写比例会被放大到100:1。一个优化的好的系统，必须从设计上去平衡这种读写压力，避免读或写任意一维触碰到天花板。所以IM系统这类场景下，通常会应用写扩散这种同步模式，来平衡读和写，将100:1的读写比例平衡到30:30。当然写扩散这种同步模式，还需要处理一些极端场景，例如万人大群。针对这种极端写扩散的场景，会退化到使用读扩散。一个简单的IM系统，通常会在产品层面限制这种大群的存在，而对于一个高级的IM系统，会采用读写扩散混合的同步模式，来满足这类产品的需求。采用混合模式，会根据数据的不同类型和不同的读写负载，来决定用写扩散还是读扩散。

典型架构设计



上图是一个典型的消息系统架构，架构中包含几个重要组件。

- 端：作为消息的发送和接收端，通过连接消息服务器来发送和接收消息。
- 消息服务器：一组无状态的服务器，可水平扩展，处理消息的发送和接收请求，连接后端消息系统。
- 消息队列：新写入消息的缓冲队列，消息系统的前置消息存储，用于削峰填谷以及异步消费。
- 消息处理：一组无状态的消费处理服务器，用于异步消费消息队列中的消息数据，处理消息的持久化和写扩散同步。
- 消息存储和索引库：持久化存储消息，每个会话对应一个Timeline进行消息存储，存储的消息建立索引来实现消息检索。
- 消息同步库：写扩散形式同步消息，每个用户的收件箱对应一个Timeline，同步库内消息不需要永久保存，通常对消息设定一个生命周期。

新消息会由端发出，通常消息体中会携带消息ID（用于去重）、逻辑时间戳（用于排序）、消息类型（控制消息、图片消息或者文本消息等）、消息体等内容。消息会先写入消息队列，作为底层存储的一个临时缓冲区。消息队列中的消息会由消息处理服务器消费，可以允许乱序消费。消息处理服务器对消息先存储后同步，先写入发件箱Timeline（存储库），后写扩散至各个接收端的收件箱（同步库）。消息数据写入存储库后，会被近实时的构建索引，索引包括文本消息的全文索引以及多字段索引（发送方、消息类型等）。

对于在线的设备，可以由消息服务器主动推送至在线设备端。对于离线设备，登录后会主动向服务端同步消息。每个设备会在本地保留有最新一条消息的顺序ID，向服务端同步该顺序ID后的所有消息。

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

4.2. 现代IM系统中的消息系统—模型

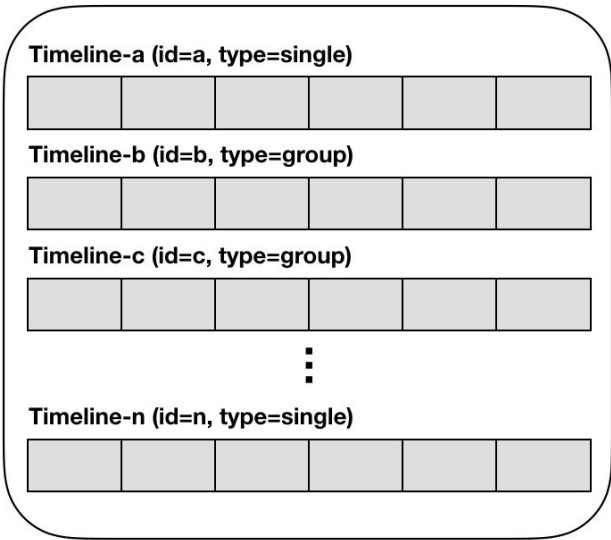
本章节主要介绍表格存储（Tablestore）的Timeline模型基本功能以及核心组件，并且会通过IM消息系统场景，介绍如何基于Timeline实现IM场景下消息同步、存储和索引等基本功能。

Timeline模型

Timeline模型以简单为设计目标，核心模块主要包括：

模块	说明
Store	Timeline存储库，类似数据库表的概念。
Identifier	用于区分Timeline的唯一标识。
Meta	用于描述Timeline的元数据，元数据描述采用free-schema结构，可自由包含任意列。
Queue	一个Timeline内所有Message存储在Queue内。
Message	Timeline内传递的消息体，也是一个free-schema的结构，可自由包含任意列。
Index	包含Meta Index和Message Index，可对Meta或Message内的任意列自定义索引，提供灵活的多条件组合查询和搜索。

Timeline Store



- Configuration**
- **Time to live:** forever or n-days
 - **Storage type:** HDD or SSD
- Capability**
- **Large scale:** up to 10-billion~ timelines
 - **High performance:** up to 10-million~ TPS, low latency read and scan.

Timeline Store

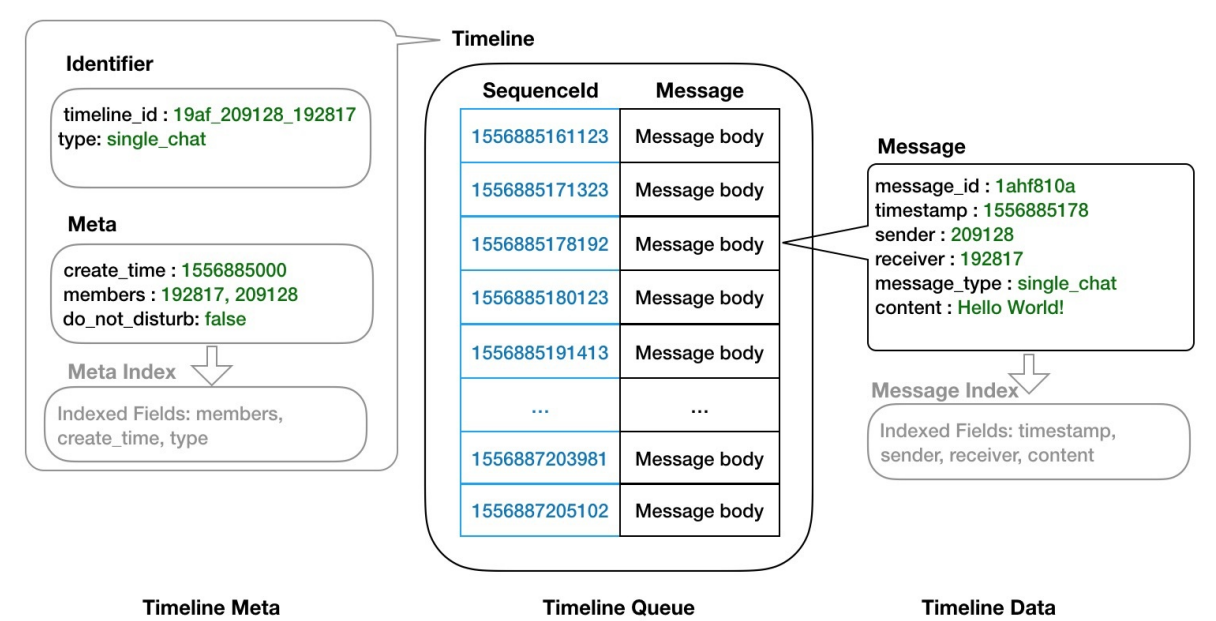
Timeline Store是Timeline的存储库，对应于数据库内表的概念。上图是Timeline Store的结构图，Store内会存储所有的Timeline数据。Timeline是一个面向海量消息的数据模型，同时用于消息存储库和同步库，需要满足多种要求。

- 支撑海量数据存储：对于消息存储库来说，如果需要消息永久存储，则随着时间的积累，数据规模会越来越大，需要存储库能应对长时间积累的海量消息数据存储，需要能达到PB级容量。
- 低存储成本：消息数据的冷热区分是很明显的，大部分查询都会集中在热数据，所以对于冷数据需要有一个比较低成本的存储方式，否则随着时间的积累数据量不断膨胀，存储成本会非常大。
- 数据生命周期管理：不管是对于消息数据的存储还是同步，数据都需要定义生命周期。存储库是用于在线存储消息数据本身，通常需要设定一个较长周期的保存时间。而同步库是用于写扩散模式的在线或离线推送，通常设定一个较短的保存时间。
- 极高的写入吞吐：各类场景下的消息系统，除了类似微博、头条这种类型的Feeds流系统，像绝大部分即时通讯或朋友圈这类消息场景，通常是采用写扩散的消息同步模式，写扩散要求底层存储具备极高的写入吞吐能力，以应对消息洪峰。
- 低延迟的读：消息系统通常是应用在在线场景，所以对于查询要求低延迟。

Tablestore Timeline的底层是基于LSM存储引擎的分布式数据库，LSM的最大优势就是对写入非常友好，天然适合消息写扩散的模式。同时对查询也做了极大优化，例如热数据进缓存、bloom filter等等。数据表采用Range Partition的分区模式，能提供水平扩展的服务能力，以及能自动探测并处理热点分区的负载均衡策略。为了满足同步库和存储库对存储的不同要求，也提供了一些灵活的自定义配置，主要包括：

配置	说明
Time to live（数据生命周期）	可自定义数据生命周期，例如永久保存，或者保存N天。
Storage type（存储类型）	自定义存储类型，对存储库来说，HDD是最好的选择，对同步库来说，SSD是最好的选择。

Timeline Module

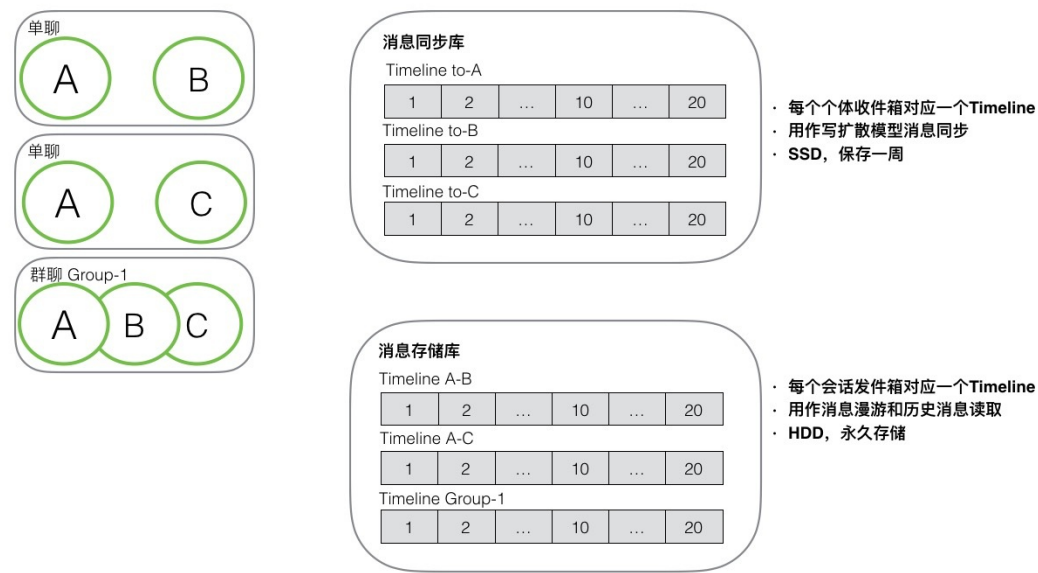


Timeline Store内能存储海量的Timeline，单个Timeline的详细结构图如上，可以看到Timeline主要包含了Timeline Meta、Timeline Queue、Timeline Data三大部分。

- Timeline Meta
 - 元数据部分，用于描述Timeline
 - Identifier：用于唯一标识Timeline，可包含多个字段。
 - Meta：用于描述Timeline的元数据，可包含任意个数任意类型的字段。
 - Meta Index：元数据索引，可对元数据内任意属性列建索引，支持多字段条件组合查询和检索。
- Timeline Queue
 - 用于存储和同步消息的队列，队列中元素由以下两部分组成。
 - Sequence Id：顺序ID，队列中用于定位Message的位点信息，在队列中顺序ID保持递增。
 - Message：队列中承载消息的实体，包含了消息的完整内容。
- Timeline Data
 - Timeline的数据部分就是Message，Message主要包含以下两部分。

- Message：消息实体，其内部也可以包含任意数量任意类型字段。
- Message Index：消息数据索引，可对消息实体内任意列做索引，支持多字段条件组合查询和检索。

IM消息系统建模



基于Timeline的消息库设计

以一个简易版IM系统为例，来看如何基于Tablestore Timeline模型建模。按照上图中的例子，存在A、B、C三个用户，A与B发生单聊，A与C发生单聊，以及A、B、C组成一个群聊，来看下在这个场景下消息同步、存储以及读写流程分别如何基于Tablestore Timeline建模。

消息同步模型

消息同步选择写扩散模型，能完全利用Tablestore Timeline的优势，以及针对IM消息场景读多写少的特性，通过写扩散来平衡读写，均衡整个系统的资源。写扩散模型下，每个接收消息的个体均拥有一个收件箱，所有需要同步至该个体的消息需要投递到其收件箱内。图上例子中，A、B、C三个用户分别拥有收件箱，每个用户不同的设备端，均从同一个收件箱内拉取新消息。

消息同步库

收件箱存储在同步库内，同步库中每个收件箱对应一个Timeline。根据图上的例子，总共存在3个Timeline作为收件箱。每个消息接收端保存有本地最新拉取的消息的SequenceID，每次拉取新消息均是从该SequenceID开始拉取消息。对同步库的查询会比较频繁，通常是对最新消息的查询，所以要求热数据尽量缓存在内存中，能提供高并发低延迟的查询。所以对同步库的配置，一般是需要SSD存储。消息如果已经同步到了所有的终端，则代表收件箱内的该消息已经被消费完毕，理论上可以清理。但设计上来说不做主动清理，而是给数据定义一个较短的生命周期来自动过期，一般定义为一周或者两周。数据过期之后，如果仍要同步拉取新消息，则需要退化到读扩散的模式，从存储库中拉取消息。

消息存储库

消息存储库中保存有每个会话的消息，每个会话的发件箱对应一个Timeline。发件箱内的消息支持按会话维度拉取消息，例如浏览某个会话内的历史消息则通过读取发件箱完成。一般来说，新消息通过在线推送或者查询同步库可投递到各个接收端，所以对存储库的查询会相对来说较少。而存储库用于长期存储消息，例如永久存储，相对同步库来说数据量会较大。所以存储库的选择一般是HDD，数据生命周期根据消息需要保存的时间来定，通常是一个较长的时间。

消息索引库

消息索引库依附于存储库，使用了Timeline的Message Index，可以对存储库内的消息进行索引，例如对文本内容的全文索引、收件人、发件人以及发送时间的索引等，能支持全文检索等高级查询和搜索。

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

4.3. 现代IM系统中的消息系统—实现

本文主要以钉钉（DingTalk）的功能为参照，详细说明如何基于表格存储的Timeline模型实现钉钉的IM功能。

以下内容按照聊天系统的消息存储、关系维护、即时感知、多端同步四个功能模块分块，分别介绍每一部分的功能、方案介绍、表设计以及实现代码等。

功能模块

功能：消息存储

消息系统中，消息存储是最基本的功能。对于消息存储（提供消息的读、写、持久化），一方面需要持久化写入，保证消息数据的不丢失；另一方面，适合用户的快速和高效查询。在IM场景中，写入方式通常是单行和批量写入，而读取需要按照消息队列范围读取。有时用户还有对历史消息的模糊查询需求，此时就需要使用多维检索和全文检索的能力。

消息的存储都是基于Timeline模型。关于模型的更多信息，请参见[Tablestore发布Timeline 2.0模型](#)。

表设计：表名为im_timeline_store_table，表结构如下图所示。

数据源: im_timeline_store_table表格数据最多显示50行。

	详细数据	timeline_id(主键)	sequence_id(主键)	conversation	send_time	sender	text	type
☐	详细数据	group_1	1563422738697000	group_1	1563422738694	user_1	今天谁使用表格存储了	GROUP
☐	详细数据	group_1	1563422738781000	group_1	1563422738779	user_2	表格+1	GROUP
☐	详细数据	group_1	1563422738871000	group_1	1563422738868	user_3	表格+1	GROUP
☐	详细数据	group_1	1563422738982000	group_1	1563422738979	user_4	表格+1	GROUP
☐	详细数据	single_2	1563422738479000	user_2	1563422738424	user_2	我们是好朋友了2	SINGLE
☐	详细数据	single_3	1563422738618000	user_3	1563422738614	user_3	我们是好朋友了3	SINGLE
☐	详细数据	single_4	1563422738646000	user_4	1563422738643	user_4	我们是好朋友了4	SINGLE
☐								

共有7条, 每页显示: 10条

存储库

功能：会话窗口消息展示

存储库是聊天会话消息所对应的存储表，消息以会话分类存储，每个会话是一个消息队列。单个消息队列（TimelineQueue）通过timelineId唯一标识，所有消息基于sequenceId有序排列。消息体中含有发送人、消息ID（消息去重）、消息发送时间、消息体内容、消息类型（消息类型包含图片、文件、普通文本等，本文仅适用普通文本）。



如上图所示，当用户单击某一个会话时，窗口会展示相应会话的最新一页消息。图中的消息均从存储库拉取，通过timelineId获取该会话的Queue实例，然后调用Queue的scan接口与ScanParam参数（sequenceId范围+倒序）拉取最新的一页消息。当用户向上滚动，展示完该页消息后，客户端会基于第一次请求的最小sequenceId发起第二次请求，获取第二页消息记录，单页消息数通常选择20~30条。会话的消息可以选择在客户端持久化，然后在感知到新消息之后更新本地消息，增加缓存减少网络IO。

存储库的消息需要永久保存，是整个应用的全量消息存储。存储库的数据过期时间（TTL）需要设置为-1。

核心代码

```
public List<AppMessage> fetchConversationMessage(String timelineId, long sequenceId) {
    TimelineStore store = timelineV2.getTimelineStoreTableInstance();
    TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", timelineId)
        .build();
    ScanParameter parameter = new ScanParameter()
        .scanBackward(sequenceId)
        .maxCount(30);
    Iterator<TimelineEntry> iterator = store.createTimelineQueue(identifier).scan(parameter);
    List<AppMessage> appMessages = new LinkedList<AppMessage>();
    while (iterator.hasNext() && counter++ <= 30) {
        TimelineEntry timelineEntry = iterator.next();
        AppMessage appMessage = new AppMessage(timelineId, timelineEntry);
        appMessages.add(appMessage);
    }
    return appMessages;
}
```

```
[Scan Conversation Message]: -> group_1
[user_4 say @2019-07-17 21:26:42]: 表格+1
[user_3 say @2019-07-17 21:26:42]: 表格+1
[user_2 say @2019-07-17 21:26:42]: 表格+1
[user_1 say @2019-07-17 21:26:42]: 今天谁使用表格存储了
[Total Count]: 4, [Update SyncCheckpoint]: 1563370002181000
```

功能：多维组合、全文检索

全文检索能力是对存储库的消息内容进行模糊查询，因而需要对存储库的数据建立多元索引。具体索引字段需要根据设计需求设计，例如钉钉公开群的检索需要对群ID、消息发送人、消息类型、消息内容以及时间建立索引，其中消息内容需要使用分词字符串（TEXT）类型，从而提供模糊查询的能力。



核心代码

```
public List<AppMessage> searchMessage(String timelineId, String content) {
    TimelineStore store = timelineV2.getTimelineStoreTableInstance();
    TermQuery termQuery = new TermQuery();
    termQuery.setFieldName("timeline_id");
    termQuery.setTerm(ColumnValue.fromString(timelineId));
    MatchPhraseQuery matchPhraseQuery = new MatchPhraseQuery();
    matchPhraseQuery.setFieldName("text");
    matchPhraseQuery.setText(content);
    BoolQuery boolQuery = new BoolQuery();
    boolQuery.setMustQueries(Arrays.asList(termQuery, matchPhraseQuery));
    SearchQuery searchQuery = new SearchQuery();
    searchQuery.setQuery(boolQuery);
    searchQuery.setLimit(30);
    List<SearchResult.Entry<TimelineEntry>> entryList = store.search(searchQuery).getEntries();
    List<AppMessage> appMessages = new LinkedList<AppMessage>();
    for (SearchResult.Entry<TimelineEntry> resultEntry : entryList) {
        String tId = resultEntry.getIdentifier().getField(0).getValue().toString();
        TimelineEntry timelineEntry = resultEntry.getData();
        AppMessage appMessage = new AppMessage(tId, timelineEntry);
        appMessages.add(appMessage);
    }
    return appMessages;
}
```

```
[Search Message in Conversation]: -> group_1 -> '表格'
[user_1 say @2019-07-17 21:26:42]: 今天谁使用表格存储了
[user_2 say @2019-07-17 21:26:42]: 表格+1
[user_3 say @2019-07-17 21:26:42]: 表格+1
[user_4 say @2019-07-17 21:26:42]: 表格+1
[Total Count]: 4
```

另外，为了做消息的权限管理，仅允许用户检索自己有权查看的消息。您可以通过在消息体字段中扩展接收人ID数组来实现消息内容的权限限制，当对所有群做检索时，增加接收人字段为自己的用户ID即可。样例中未实现此功能，用户可根据需求自行增加和修改。

同步库

功能：新消息即时统计

当客户端在线时，应用的系统服务会维护客户端的长连接，因而可以感知客户端在线。当用户的同步库有新消息写入时（即有新消息），应用会发出信号通知客户端有新消息，然后客户端会基于同步库checkpoint点，拉取同步库中该sequenceId之后的所有新消息，统计各会话的新消息数，并更新checkpoint点。



如上图所示，对于一个在线客户端，每个会话都会维护一个未读消息的计数（小红点），也会有一个总未读数的计数，这个数量一般会存储在客户端本地或者通过Redis持久化。未读消息是指通过同步库拉取并统计过，但是还未被用户点开过的消息数量。在拉取到新消息列表后，客户端（或应用层）会遍历所有新消息，然后将新消息所对应会话的未读计数累加1，这样实现了未读消息的即时感知与更新。只有当用户点开会话后，会话的未读计数才会清零。

在更新未读数的同时，会话列表中还会有最新消息的简短摘要信息以及最新消息的发送时间等。这些可以在遍历新消息列表时不断更新。统计、摘要都是依托同步库实现的，而非存储库。

核心代码

```
public List<AppMessage> fetchSyncMessage(String userId, long lastSequenceId) {
    TimelineStore sync = timelineV2.getTimelineSyncTableInstance();
    TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", userId)
        .build();
    ScanParameter parameter = new ScanParameter()
        .scanForward(lastSequenceId)
        .maxCount(30);
    Iterator<TimelineEntry> iterator = sync.createTimelineQueue(identifier).scan(parameter);
    ;
    List<AppMessage> appMessages = new LinkedList<AppMessage>();
    int counter = 0;
    while (iterator.hasNext() && counter++ <= 30) {
        AppMessage appMessage = new AppMessage(userId, iterator.next());
        appMessages.add(appMessage);
    }
    return appMessages;
}
```

在统计到会话列表中不存在的会话时，客户端会做一次额外请求。通过timelineID获取会话的基本描述信息，例如群头像或好友的头像、群名称等，并初始化未读数计时器0，然后累加新消息数、更新最新消息摘要等。

```
[Count Unread Message]
conversation: user_3, newUnread: 1, latestMessage: 我们是好朋友了3
conversation: user_2, newUnread: 1, latestMessage: 我们是好朋友了2
conversation: user_4, newUnread: 1, latestMessage: 我们是好朋友了4
conversation: group_1, newUnread: 4, latestMessage: 表格+1
[Total Count]: 7, [Update SyncCheckpoint]: 1563370002491000
```

同步库对于IM场景下的新消息即时感知统计这一核心功能，就是通过写入冗余的方式，提升新消息读取统计的效率与速度。对于IM场景没有收件箱的概念，因而同步库中冗余消息并没有永久保存的价值，提供7天过期时间已经足够保证功能正常。用户可以根据自身需求，调整同步库的数据过期时间（TTL）。

功能：异步写扩散

在本文的样例中，单聊会话的消息在写完存储库后同时写入了同步库，只有两行的写入开销很小。但是对于群会话，写完存储库后要获取群用户列表，然后依次写入相应用户的同步库。这种方式在群少和用户少时不会有问题，但随着用户体量和活跃度的增加，同步写的方式会面临性能问题，因此建议用户对群写扩散使用异步任务实现。

用户可以基于表格存储实现一个任务队列，将写扩散任务写入队列中后直接返回，然后由其他进程保证任务队列的执行。任务队列保存了群ID、消息的完整信息，消费进程不断轮询读取新任务，获取任务后，才会从群关系表中获取完整的群成员列表，并做相应的写扩散。

任务队列可以直接基于Tablestore实现，表设计为两列主键，第一列为topic，第二列为自增列，一个topic对应一个队列，任务会被有序写入单个队列中。当并发量持续膨胀后，可对任务做hash分桶，随机写入多个topic。这样可以增加消费者数量（消费并发量），提升写扩散效率。对应任务队列消费，用户只需要维护每个topic的checkpoint点。checkpoint点之前的为已完成任务，通过getRange的方式顺序获取checkpoint点之后未执行的新任务，保证任务的执行。失败的任务可以重新写入任务队列来提升容错，并增加重试计数。出现多次失败后放弃重写，然后将该任务写入特殊的问题队列，方便应用的开发者们查询、定位问题。

功能：元数据管理

元数据是指描述数据的数据，此处主要体现为用户元数据和会话元数据两类。此处群的元数据信息包括群ID（复用群的timelineId）、群名称、创建时间等信息，可以直接基于timelineMeta的管理表完成实现，所有Group类型的TimelineMeta可以映射为一个Group。但是用户的元数据却不能复用TimelineMeta，所以需要单独的表实现。

用户元数据

用户元数据即用户的属性信息，通过用户ID识别特定用户。在上文提到的用户关系中，通过用户的标识ID确认用户身份，但用户的属性信息（例如性别、签名、头像等）还是需要单独维护。

表设计：表名为im_user_table，表结构如下图所示。

数据源：im_user_table表格数据最多显示50行。

☐	详细数据	user_id(主键)	sexuality	user_name
☐	详细数据	user_1	MALE	名字1
☐	详细数据	user_10	FEMALE	名字10
☐	详细数据	user_2	FEMALE	名字2
☐	详细数据	user_3	MALE	名字3
☐	详细数据	user_4	FEMALE	名字4
☐	详细数据	user_5	MALE	名字5
☐	详细数据	user_6	FEMALE	名字6
☐	详细数据	user_7	MALE	名字7
☐	详细数据	user_8	FEMALE	名字8
☐	详细数据	user_9	MALE	名字9
☐				

共有10条，每页显示：10条

用户元数据以user_id为标识，与同步库中的timeline_id一一对应。用户同步新消息时，只会拉取同步库中自己对应的单个消息队列（TimelineQueue）。因此为了唯一ID的方便管理，您可以选择user_id与用户同步库的timeline_id使用同一个值。因此在消息写扩散时，只需知道群内用户的user_id列表和好友user_id，即可完成写扩散。

功能：用户检索

对于用户，添加好友的需求有很多种，本样例中您只需要维护用户表，并且创建多元索引，即可轻松实现。样例中未实现，您可以根据自己需求配置不同的索引字段设置。此处仅简单分析一下需求。

- 通过用户ID：主键查询
- 二维码（含用户ID信息）：主键查询
- 用户姓名：多元索引，用户名字段设置为分词字符串类型
- 用户标签：多元索引，数组字符串索引提供标签检索，嵌套索引提供多标签打分检索排序
- 附近的人：多元索引，GEO索引查询附近、特定地理围栏的人

关于多元索引的更多信息，请参见[多元索引简介](#)。

会话元数据

会话元数据即会话的属性信息，通过唯一会话ID识别特定会话，属性信息包括会话类别（群、单聊、公众号等）、群名称、公告、创建时间等。同时，通过群名称模糊查找群也是会话元数需要的重要能力。

在Timeline模型中，提供了Timeline Meta的管理能力，只需通过相应的接口即可实现会话元数据的管理。

数据源：im_timeline_meta_table表格数据最多显示50行。

☐	详细数据	timeline_id(主键)	create_time	group_name	type	users
☐	详细数据	group_1	1563422737593	表格存储群1	GROUP	
☐	详细数据	group_2	1563422737782	表格存储群2	GROUP	
☐	详细数据	group_3	1563422737819	表格存储群3	GROUP	
☐	详细数据	group_4	1563422737852	表格存储群4	GROUP	
☐	详细数据	group_5	1563422737886	表格存储群5	GROUP	
☐	详细数据	single_2	1563422737945		SINGLE	[user_1,user_2]
☐	详细数据	single_3	1563422738041		SINGLE	[user_1,user_3]
☐	详细数据	single_4	1563422738072		SINGLE	[user_1,user_4]
☐	详细数据	single_5	1563422738100		SINGLE	[user_1,user_5]

共有9条，每页显示：10条

存储库中管理的是会话的消息队列（TimelineQueue），此处与会话元数据中的行一一对应。客户端用户选中特定会话后，应用从相应的消息队列倒序批量拉取消息展示到客户端，群聊单聊的使用方式一样，因而不做会话类型的区分。

功能：群检索

用户如果有加入群的需求，首先需要查询到特定的群。查询群的方式与用户查询方式类似，功能也可以做相同的实现。用户可以根据自己需求定制不同的索引字段设置，需求实现方式如下：

- 群ID：主键查询
- 二维码（含用户ID信息）：主键查询
- 群名：多元索引，用户名字段设置为分词字符串类型
- 群标签：多元索引，数组字符串索引提供标签检索，嵌套索引提供多标签打分检索排序

说明 会话元数据可以直接维护单聊会话与人的映射关系。对于单聊的meta增加一列users字段用于存放两个用户ID，这样不用额外维护关系表（基于单聊关系表im_user_relation_table创建timeline_id为第一列主键的二级索引）。

关系维护

完成了元数据管理以及用户和群的检索，剩下的就是如何添加好友以及加入群聊了。此处就涉及到IM系统中另一个重要的功能点关系维护。关系维护包含人与人的关系、人与群的关系以及人与会话的关系。以下介绍如何基于Tablestore解决关系维护的需求。

单聊关系

功能：人与单聊会话的关系

单聊场景下，参与者仅有两个人，同时不考虑顺序。无论是我联系小明或是小明联系我，对应的会话必须有且仅有一个。如果使用表格存储维护该关系，建议使用如下设计方式。

第一列为主用户ID、第二列为次用户ID，在两个人成为好友后，关系表中需要插入两行数据，分别以自己的用户ID为main_user，以好友的用户ID为sub_user，然后将共同的会话timeline_id作为属性列，并且可以维护相互之间不同的昵称、显示。

表设计：表名为im_user_relation_table，表结构如下图所示。

数据源：im_user_relation_table表格数据最多显示50行。

	详细数据	main_user(主键)	sub_user(主键)	timeline_id
☐	详细数据	user_1	user_2	single_2
☐	详细数据	user_1	user_3	single_3
☐	详细数据	user_1	user_4	single_4
☐	详细数据	user_2	user_1	single_2
☐	详细数据	user_3	user_1	single_3
☐	详细数据	user_4	user_1	single_4

共有6条, 每页显示: 10条

基于该单聊关系表，还可以建立多元索引，方便用户好友列表的获取，同时支持根据添加好友时间排序、昵称排序等功能。如果考虑到延时和费用等因素，直接通过getRange接口也可以快速拉、高效的获取自己所有好友列表，实现好友关系的维护与查询。

功能：人与人的关系

基于上表im_user_relation_table，人与人的关系可以很简单实现，例如判断我与小明的好友关系，直接通过单行查询即可知道好友关系是否存在，如果存在就不会展示加好友按钮。而如果非好友，待完成好友添加后，写入两行不同主键顺序行，并生成一个唯一的timelineId即可。如此设计的好处在于用户可以直接通过自己的ID与好友的ID快速获取会话信息。只要用户在写入两行时做好一致性维护。

如果好友关系一旦解除，可以直接拼出关系表中两行主键对用户关系，通过做物理删除（删除行）或逻辑删除（属性列状态修改）结束两个人的好友关系即可。

核心代码

```
public void establishFriendship(String userA, String userB, String timelineId) {
    PrimaryKey primaryKeyA = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("main_user", PrimaryKeyValue.fromString(userA))
        .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.fromString(userB))
        .build();
    RowPutChange rowPutChangeA = new RowPutChange(userRelationTable, primaryKeyA);
    rowPutChangeA.addColumn("timeline_id", ColumnValue.fromString(timelineId));
    PrimaryKey primaryKeyB = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("main_user", PrimaryKeyValue.fromString(userB))
        .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.fromString(userA))
        .build();
    RowPutChange rowPutChangeB = new RowPutChange(userRelationTable, primaryKeyB);
    rowPutChangeB.addColumn("timeline_id", ColumnValue.fromString(timelineId));
    BatchWriteRowRequest request = new BatchWriteRowRequest();
    request.addRowChange(rowPutChangeA);
    request.addRowChange(rowPutChangeB);
    syncClient.batchWriteRow(request);
}

public void breakupFriendship(String userA, String userB) {
    PrimaryKey primaryKeyA = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("main_user", PrimaryKeyValue.fromString(userA))
        .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.fromString(userB))
        .build();
    RowDeleteChange rowPutChangeA = new RowDeleteChange(userRelationTable, primaryKeyA);
    PrimaryKey primaryKeyB = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("main_user", PrimaryKeyValue.fromString(userB))
        .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.fromString(userA))
        .build();
    RowDeleteChange rowPutChangeB = new RowDeleteChange(userRelationTable, primaryKeyB);
    BatchWriteRowRequest request = new BatchWriteRowRequest();
    request.addRowChange(rowPutChangeA);
    request.addRowChange(rowPutChangeB);
    syncClient.batchWriteRow(request);
}
```

```
[Create New SingleConversation]
[Establish Friendship] user_1 make friend with: user_2 to user_5
[Friends List] user_id: user_1, friends: [single_2:SINGLE:user_2:名字2, single_3:SINGLE:user_3:名字3, single_4:SINGLE:user_4:名字4, single_5:SINGLE:user_5:名字5]
[Friends List] user_id: user_2, friends: [single_2:SINGLE:user_1:名字1]
[Breakup Friendship] user_1 breakup with user_5
[Friends List] user_id: user_1, friends: [single_2:SINGLE:user_2:名字2, single_3:SINGLE:user_3:名字3, single_4:SINGLE:user_4:名字4]
[Friends List] user_id: user_5, friends: []
```

群聊关系

功能：群聊会话与人的关系

群聊时，主要的查询需求还是获取当前群内用户的列表。一方面方便群属性的展示，另一方面为应用做写扩散提供快速获取收件人列表的查询。因而在表设计上，我们会建议用户使用两列主键：第一列为群ID，第二列为用户ID。通过这样的设计，可以直接给予getRange接口拉取群所有用户的信息。

群聊关系表解决了群到用户的映射关系，但我们还需要用户到群的映射关系。如果为了查询用户所在群的列表而新建一张表，冗余成本、一致性维护成本就很高。这里可以使用两种索引来解决反向的映射关系。样例中，我们使用了二级索引，将用户ID字段作为索引主键，从而可以直接基于索引查询单用户的群列表。同步实时性更好，成本更低。当然用户也可以使用多元索引：对群、用户、入群时间做索引，可以查询到某用户的所有在群列表，并且基于入群时间排序。

表设计：im_group_relation_table

数据源：im_group_relation_table					表格数据最多显示50行。
	详细数据	group_id(主键)	user_id(主键)	join_time	
☐	详细数据	group_1	user_1	1563422737654	▼
☐	详细数据	group_1	user_2	1563422737760	▼
☐	详细数据	group_1	user_3	1563422737788	▼
☐	详细数据	group_1	user_4	1563422738272	▼
☐	详细数据	group_2	user_1	1563422737789	▼
☐	详细数据	group_2	user_2	1563422737798	▼
☐	详细数据	group_2	user_3	1563422737807	▼
☐	详细数据	group_3	user_1	1563422737826	▼
☐	详细数据	group_3	user_2	1563422737834	▼
☐	详细数据	group_3	user_3	1563422737843	▼
☐					共有16条，每页显示：10条

基于群关系表，可以直接基于关系主表通过getRange的方式获取单个群内所有的用户。在做写扩散时，可以直接获取群内用户ID列表，提升写扩散的效率。同时，也方便展示群内用户列表。

核心代码

```
public List<Conversation> listMySingleConversations(String userId) {
    PrimaryKey start = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("main_user", PrimaryKeyValue.fromString(userId))
        .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.INF_MIN)
        .build();
    PrimaryKey end = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("main_user", PrimaryKeyValue.fromString(userId))
        .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.INF_MAX)
        .build();

    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(userRelationTable);
    criteria.setInclusiveStartPrimaryKey(start);
    criteria.setExclusiveEndPrimaryKey(end);
    criteria.setMaxVersions(1);
    criteria.setLimit(100);
    criteria.setDirection(Direction.FORWARD);
    criteria.addColumnsToGet(new String[] {"timeline_id"});
    GetRangeRequest request = new GetRangeRequest(criteria);
    GetRangeResponse response = syncClient.getRange(request);
    List<Conversation> singleConversations = new ArrayList<Conversation>(response.getRows()
        .size());
    for (Row row : response.getRows()) {
        String timelineId = row.getColumn("timeline_id").get(0).getValue().asString();
        String subUserId = row.getPrimaryKey().getPrimaryKeyColumn("sub_user").getValue().asString();
        User friend = describeUser(subUserId);
        Conversation conversation = new Conversation(timelineId, friend);
        singleConversations.add(conversation);
    }
    return singleConversations;
}
```

```
[List User Single Conversation]: -> user_1
[timelineId: single_2]: type: SINGLE, userId: user_2, userName: 名字2, sexuality: FEMALE
[timelineId: single_3]: type: SINGLE, userId: user_3, userName: 名字3, sexuality: MALE
[timelineId: single_4]: type: SINGLE, userId: user_4, userName: 名字4, sexuality: FEMALE
[Total Count]: 3
```

功能：人与群聊会话的关系

获取单用户所有加入群列表，可以基于主表创建二级索引，将用户字段设为索引的第一列主键。索引的数据结构见下图。这样基于二级索引，可以直接通过getRange的方式获取单用户加入的群的TimelineId列表。

二级索引：im_group_relation_global_index

数据源: im_group_relation_global_index表格数据最多显示50行。

	详细数据	user_id(主键)	group_id(主键)
☐	详细数据	user_1	group_1
☐	详细数据	user_1	group_2
☐	详细数据	user_1	group_3
☐	详细数据	user_1	group_4
☐	详细数据	user_1	group_5
☐	详细数据	user_2	group_1
☐	详细数据	user_2	group_2
☐	详细数据	user_2	group_3
☐	详细数据	user_2	group_4
☐	详细数据	user_2	group_5

共有16条, 每页显示: 10条

核心代码

```
public List<Conversation> listMyGroupConversations(String userId) {
    PrimaryKey start = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("user_id", PrimaryKeyValue.fromString(userId))
        .addPrimaryKeyColumn("group_id", PrimaryKeyValue.INF_MIN)
        .build();
    PrimaryKey end = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("user_id", PrimaryKeyValue.fromString(userId))
        .addPrimaryKeyColumn("group_id", PrimaryKeyValue.INF_MAX)
        .build();
    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(groupRelationGlobalIndex);
    criteria.setInclusiveStartPrimaryKey(start);
    criteria.setExclusiveEndPrimaryKey(end);
    criteria.setMaxVersions(1);
    criteria.setLimit(100);
    criteria.setDirection(Direction.FORWARD);
    criteria.addColumnsToGet(new String[] {"group_id"});
    GetRangeRequest request = new GetRangeRequest(criteria);
    GetRangeResponse response = syncClient.getRange(request);
    List<Conversation> groupConversations = new ArrayList<Conversation>(response.getRows().size());
    for (Row row : response.getRows()) {
        String timelineId = row.getPrimaryKey().getPrimaryKeyColumn("group_id").getValue().asString();
        Group group = describeGroup(timelineId);
        Conversation conversation = new Conversation(timelineId, group);
        groupConversations.add(conversation);
    }
    return groupConversations;
}
```

```
[List User Group Conversation]: -> user_1
[timelineId: group_1]: type: GROUP, groupName: 表格存储群1, createAt: 2019-07-17 21:26:41
[timelineId: group_2]: type: GROUP, groupName: 表格存储群2, createAt: 2019-07-17 21:26:41
[timelineId: group_3]: type: GROUP, groupName: 表格存储群3, createAt: 2019-07-17 21:26:41
[timelineId: group_4]: type: GROUP, groupName: 表格存储群4, createAt: 2019-07-17 21:26:41
[timelineId: group_5]: type: GROUP, groupName: 表格存储群5, createAt: 2019-07-17 21:26:41
[Total Count]: 5
```

即时感知

让客户端即时感知消息的实现方案，请参见[Feed流设计总纲](#)中会话池的维护方式，此处仅做简要描述，不会在样例中实现。

会话池方案

即时感知新消息正是IM（Instant Message）场景下核心所在。让客户端及时感知到新信息的到来，然后客户端接收到通知后才会从同步库中拉取更新的消息，让用户更快速、更及时地提醒用户阅读新消息。可是，接受者如何才能快速感知到自己有了新消息呢？

让在线的客户端周期性的刷新拉取？这样的方式毫无疑问可以满足需求，但伴随而来的是大量无效的网络资源浪费。同时应用的压力也会随着用户量的不断增长变得更沉重。而当白天大量非活跃用户在线时，压力更为明显。面对这一问题，应用通常会维护一个推送会话池。会话池记录了在线客户端与用户信息，当在线用户有新的消息写入，通过推送池获取该用户的会话，然后通知客户端拉取同步库新消息。这样同步消息的压力只会随着真实消息量而增长，避免了大量不必要的同步库查询请求。

实现会话推送池的方案很多，可以使用内存型数据库，也可以直接使用表格存储，同时保证会话推送池的持久化。

在即时感知上，最直观的就是会话表中变动的未读消息数统计了。统计新消息的实现方式上，已在本文的【消息存储 > 第二类：同步库 > 新消息即时统计】部分做了详尽描述，不理解的可返回去重新看一下。持久化未读消息数是很必要的，否则在更换设备或重新登录后。未读消息数被清零，将会忽略很多新消息提醒，这是我们不能接受的。

多端同步

实现了以上功能，IM系统的基本需求已经完成。但实现多端数据同步上，还有两个注意事项。

其一，我们对于单客户端情况下，用户同步库做了一个checkpoint点的持久化，对应的概念是：“已读最新消息的sequenceld”。此时，checkpoint点无客户端的区分，如果使用本地做持久化，多端同步时就会出现不同客户端统计的未读消息数就会不一致。这是需要通过应用服务端维护checkpoint点，同时会话的未读消息数也需要在应用服务侧维护，这样才能保证多端统计数一致。同时，当有未读消息的会话被点击，会话未读数清0时，要让服务有感知，然后通知到其他在线端，维护实时一致性。

其二，多端情况下，自己在一个客户端发送了新消息，其他客户端在没有其他新消息时，是无法感知并刷新自己的发送消息，这在多端同步中也是要解决的小问题。这时，简单的解决方案就是将自己发送的消息，也写入自己的同步库。只要再统计未读信息时，对自己的信息不计数，但在最新消息摘要中需要做更新。这样，多端同步问题很容易实现。

添加好友、入群申请

添加好友或入群，不是主动发起请求就会直接完成的，这里需要主动方申请后，审核方完成统一才会真实完成。因而只有在审核方才会有权限发起关系的创建。

那如何让被添加用户或群主感知到申请？当然是借助同步库，作为一种新的消息类型或者特殊的会话，让用户即时感知到新申请，尽早完成审批。申请列表如果需要持久化，也可单独建表维护，只要保证用户新申请的即时感知即可。

样例实操

本文为了与用户一起梳理IM系统应用的功能点，基于Tablestore实现的样例简单功能，完整的样例代码已完成开源，代码地址[源码链接](#)。用户可以结合文章、代码一起阅读。代码在本地运行，使用前请确保您已经完成了以下操作：

- 已开通表格存储服务并创建实例
- 获取AccessKey
- 设置样例配置文件
- 实例支持二级索引

样例配置

在home目录下创建tablestoreCong.json文件，填写相应参数如下：

```
# mac或linux系统下：/home/userhome/tablestoreCong.json
# windows系统下：C:\Documents and Settings\%用户名%\tablestoreCong.json
{
  "endpoint": "http://instanceName.cn-hangzhou.ots.aliyuncs.com",
  "accessId": "*****",
  "accessKey": "*****",
  "instanceName": "instanceName"
}
```

endpoint：实例的接入地址，可在表格存储管理控制台实例详情页获取。

accessId：AccessKey ID。具体操作，请参见[如何获取AccessKey](#)。

accessKey：AccessKey的密码。具体操作，请参见[如何获取AccessKey](#)。

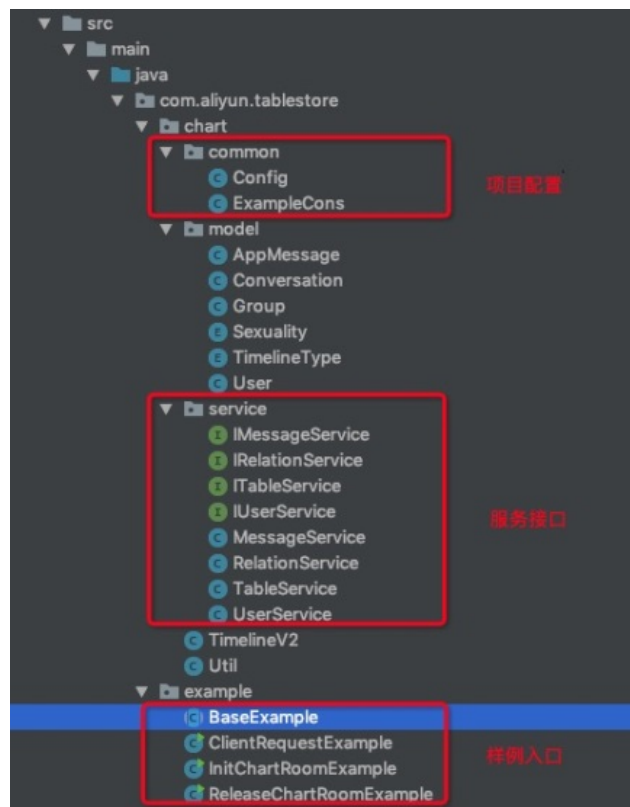
instanceName：使用的实例名称。

样例入口

样例中共有三个入口，用户需要根据先后顺序执行，使用后及时释放资源，避免不必要的费用开销。

入口	入口类名	功能
初始化	InitChartRoomExample	创建所有需要的表，同时根据配置创建相应的多元索引与二级索引。
模拟调用	ClientRequestExample	应用的接口使用，样例未做前后端联调调用，您可以通过接口返回数据的打印了解使用方式。
资源释放	ReleaseChartRoomExample	释放所有资源，先释放索引后删除表。

项目结构



专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

5. 海量气象格点数据解决方案

5.1. 方案背景

本文主要为您介绍基于表格存储的海量气象格点数据解决方案的背景及挑战。

背景

气象数据是一类典型的大数据，具有数据量大、时效性高、数据种类丰富等特点。气象数据中大量的数据是时空数据，记录了时间和空间范围内各个点的各个物理量的观测量或者模拟量，每天产生的数据量常在几十TB到上百TB的规模，且在爆发性增长。如何存储和高效的查询这些气象数据越来越成为一个难题。

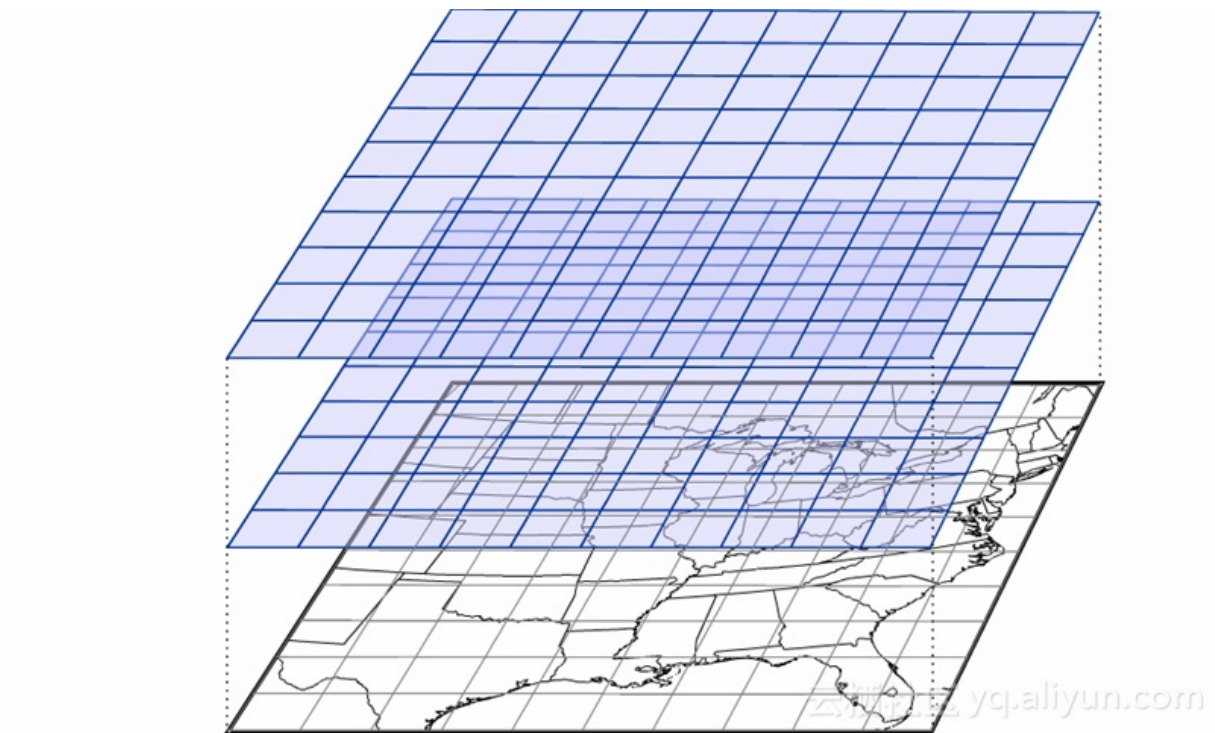
传统的方案采用关系型数据库加文件系统的方式实现这类气象数据的存储和实时查询。传统方案在可扩展性、可维护性和性能上都有缺陷，并且随着数据规模的增大，缺点越来越明显。表格存储是一款阿里云自研的分布式NoSQL服务，可以提供超大规模的存储容量，支撑超大规模的并发访问和低延迟的性能，可以很好的解决气象数据的规模和查询性能问题。本方案主要通过表格存储来实现气象格点数据的存储和查询。

挑战一：数据规模大

格点数据具有明显的多维特点，以模式系统每次产生的数据为例，一般包含以下五个维度。

- 物理量（或者称为要素，例如温度、湿度、风向、风速等）
- 时间（例如气象中的预报时效，未来3小时、6小时、9小时等）
- 高度
- 经度
- 纬度

当我们固定某一要素某一预报时效，那么高度、经度、纬度就构成一个三维网格数据，如下图所示。每个格点代表了一个三维空间上的点，上面的数值为该点在某一预报时效（例如未来三小时）下，某一物理量（例如温度）的预报值。



假设一个三维格点空间包含10个不同高度的平面，每个平面为一个2880 x 570的格点，每个格点保存一个4字节数据，那么这三维的数据量为2880 x 570 x 4 x 10，大约64MB。这仅仅是某个模式系统对某个物理量某一时效下的一次预报，可见模式数据的总量非常巨大，这对数据库的存储提出了一个不小的挑战。

挑战二：查询维度多

预报员会通过页面的形式浏览各种模式数据（格点数据），并进行数值模式预报。这个页面需要提供多种模式数据的查询方式，例如：

- 查询一个经纬度平面的格点数据
例如未来三小时全球地面温度的格点数据，或者未来三小时浙江省地面温度数据。
- 查询某个格点的时间序列数据
例如阿里云公司所在地未来3小时、未来6小时、一直到未来72小时的温度。
- 查询不同物理量的数据
例如查询某一预报时效、某一高度、某一点的全部物理量的预报数据。
- 查询不同模式系统产生的数据
例如同时查询欧洲中心的某一模式数据和中国气象机构产生的对应数据等。

一个格点数据集一般是一个五维结构，各种查询方式实际上就是对这个五维数据进行切分，例如查询某个平面、每个剖面，某个点序列、某个三维、四维子空间等等。方案设计要保证在各种查询条件的查询性能，这是数据查询方面的主要技术挑战。

5.2. 方案设计

本章节主要为您介绍基于表格存储的海量气象格点数据解决方案的模型及方案设计。

标准化格点数据模型

一个规整的五维网格数据为一个网格的数据集（GridDataSet），按照维度顺序五维分别为：

维度	说明
variable	变量，例如各种物理量
time	时间维度
z	z轴，一般表示空间高度
x	x轴，一般表示经度或纬度
y	y轴，一般表示经度或纬度

GridDataSet = F(variable, time, z, x, y)

一个GridDataSet除了包含五维数据，以及各个维度的长度等外，还包含一些其他信息：

名称	说明
GridDataSetId	唯一标记这个GridDataSet的ID。

名称	说明
Attributes	自定义属性信息，例如该数据的产生时间、数据来源、预报类型等等。 您可以自定义属性，也可以给某些属性建立索引，建立索引后就可以通过各种组合条件来查询符合条件的数据集。

例如，假设某种气象预报每次预报未来72小时的每个整点的各个高度、各个经纬度的各种物理量，则这次预报就是一个标准的五维数据，是一个单独的GridDataSet，下一次相同的预报则是另一个数据集。这两个数据集需要有不同的GridDataSetId。这两个数据集比较类似，只是起报时间不同，但是因为起报时间不在五维模型中（五维内的时间为一次预报中的未来不同时刻），所以属于不同的数据集，起报时间可以作为数据集的自定义属性。本方案中，也支持对自定义属性设置条件进行检索。

数据存储方案

表格存储设计了两张表分别存储数据集的meta和data：

- meta表示这个数据集的元数据，例如GridDataSetId、各维度长度、自定义属性等。
- data表示这个数据集里实际的网格数据。data相比meta在数据大小上要大很多。

将数据集的meta和data分开存储，主要是出于以下考虑：

- 用户会有根据多种条件查询数据集的要求，例如查询最近有哪些数据集已经完成入库，或者查询表中有哪些某种类型的数据集等。传统方案中主要是通过MySQL等关系型数据库来存储，在本方案中我们通过单独的meta表来存储，并通过表格存储的多元索引功能来实现多条件的组合查询和多种排序方式，相比传统方案更加易用。
- 在查询格点数据之前，一般要知道格点数据中各维度的长度等信息，这些信息就是存储在meta表中的，即需要先查询meta表，再查询data表。因为meta数据一般都很小，因此查询效率相比查询data要高，多一次查询并不会明显增加延迟。

meta表设计

由于GridDataSetId可以唯一标记一个GridDataSet，所以meta表的主键只有一列，用于记录GridDataSetId。各种系统属性和自定义属性保存在meta表的属性列中。

数据表名称： GRID_META_TABLE_EXAMPLE

调整生命周期与最大版本

数据生命周期： -1

最大数据版本： 1

数据有效版本偏差： 86400

最近一次调整时间： 2019-04-12 13:07:24

表格大小： 0 B

主键：

name	type	other
_id	STRING	(分片键)

查询meta表有两种方式：一种是通过GridDataSetId直接查询，另外一种是通过多元索引。可以根据多种属性条件组合进行查询，例如筛选某种类型的数据，按照入库时间从新到老返回等。

data表设计

data表的设计要解决五维数据在不同的切分模式下的查询效率问题，不能简单直接的对数据进行存储。

为了高效查询，需要尽量减少一次查询需要扫描的数据量。一个数据集的数据量可能在几GB的级别，但是一次查询往往只需要其中的几MB的数据，如果无法高效的定位要查询的数据，那么就要扫描全部的几GB的数据，从中筛选出符合某个范围的数据，显然效率是很低的。如何才能高效定位到需要的数据之中是提高查询效率的关键。

合理的表结构设计可以提高查询效率，在设计表data时，使用四个主键列：

主键	说明
GridDataSetId	数据集Id，唯一标记这个数据集。
Variable	变量名，即五维模型中的第一维。
Time	时间，即五维模型中的第二维。
Z	高度，即五维模型中的第三维。

数据表名称： GRID_DATA_TABLE_EXAMPLE

调整生命周期与最大版本

数据生命周期： -1

最大数据版本： 1

数据有效版本偏差： 86400

最近一次调整时间： 2019-04-12 13:07:24

表格大小： 0 B

主键：

name	type	other
_id	STRING	(分片键)
_variable	STRING	
_t	INTEGER	
_z	INTEGER	

这四列主键列标记一行表格存储中的数据，这行数据需要保存后两维的数据，即一个格点平面。

这种设计下，五维中的前三维都可以通过主键列的值来定位，即对于前三维的每一种情况，都对应表格存储中的一行。由于前三维分别代表变量、时间和高度，一般不会很多，每个维度在几个到几十个的级别，可以通过一些并行查询的方法来加速查询速度。

后两维代表了一个水平的平面，一般是一个经纬度网格，这两维的大小是比前三维要大很多的，每维在几百到几千的级别，随着数值预报越来越精细化，这个网格的大小还会成倍增加。这样的稠密的网格数据，不能把每个格点都用一行来保存，这样列的数量会非常多，存储效率也会非常的低。另一方面，如果我们把一个平面的格点数据存储到一行中，在整读整取时效率比较高，但是如果只读取某个点，就会读取很多无效数据，效率又会变得比较低。因此我们采取一种折中的方案，对平面的二维数据再次进行切分，切分成更小的平面数据块，这样就可以做到只读取部分数据块，而不总是读取整个平面，因此极大的提高了查询性能。

假设rows=481，cols=641，我们使切分后的方格的rows_=49，cols_=65

Data_0_0	Data_0_65	Data_0_130	...
Data_49_0	Data_49_65	Data_49_130	...
...	...	...	...
Data_441_0	Data_441_65	Data_441_130	...

5.3. 方案实现

本章节主要为您介绍如何基于表格存储实现海量气象格点数据解决方案。

基于[方案设计](#)，表格存储实现了一个Tablestore-Grid的library，基于这个library您可以非常方便的实现气象格点数据的存储、查询和管理。提供以下接口：

```
public interface GridStore {
    /**
     * 创建相关的meta、data表，数据录入前调用。
     * @throws Exception
     */
    void createStore() throws Exception;
    /**
     * 写入gridDataSet的meta信息。
     * @param meta
     * @throws Exception
     */
    void putDataSetMeta(GridDataSetMeta meta) throws Exception;
    /**
     * 更新meta信息。
     * @param meta
     * @throws Exception
     */
    void updateDataSetMeta(GridDataSetMeta meta) throws Exception;
    /**
     * 通过gridDataSetId获取meta。
     * @param dataSetId
     * @return
     */
}
```

```

    * @throws Exception
    */
    GridDataSetMeta getDataSetMeta(String dataSetId) throws Exception;
    /**
     * // 创建meta表的多元索引。
     * @param indexName
     * @param indexSchema
     * @throws Exception
     */
    void createMetaIndex(String indexName, IndexSchema indexSchema) throws Exception;
    /**
     * 通过多种查询条件来查询符合条件的数据集。
     * @param indexName 多元索引名。
     * @param query 查询条件，可以通过QueryBuilder构建。
     * @param queryParams 查询相关参数，包括offset、limit、sort等。
     * @return
     * @throws Exception
     */
    QueryGridDataSetResult queryDataSets(String indexName, Query query, QueryParams queryParams) throws Exception;
    /**
     * 获取GridDataWriter用于写入数据。
     * @param meta
     * @return
     */
    GridDataWriter getDataWriter(GridDataSetMeta meta);
    /**
     * 获取GridDataFetcher用于读取数据。
     * @param meta
     * @return
     */
    GridDataFetcher getDataFetcher(GridDataSetMeta meta);
    /**
     * 释放资源。
     */
    void close();
}

public interface GridDataWriter {
    /**
     * 写入一个二维平面。
     * @param variable 变量名。
     * @param t 时间维的值。
     * @param z 高度维的值。
     * @param grid2D 平面数据。
     * @throws Exception
     */
    void writeGrid2D(String variable, int t, int z, Grid2D grid2D) throws Exception;
}

public interface GridDataFetcher {
    /**
     * 设置要查询的变量。
     * @param variables
     * @return
     */
}

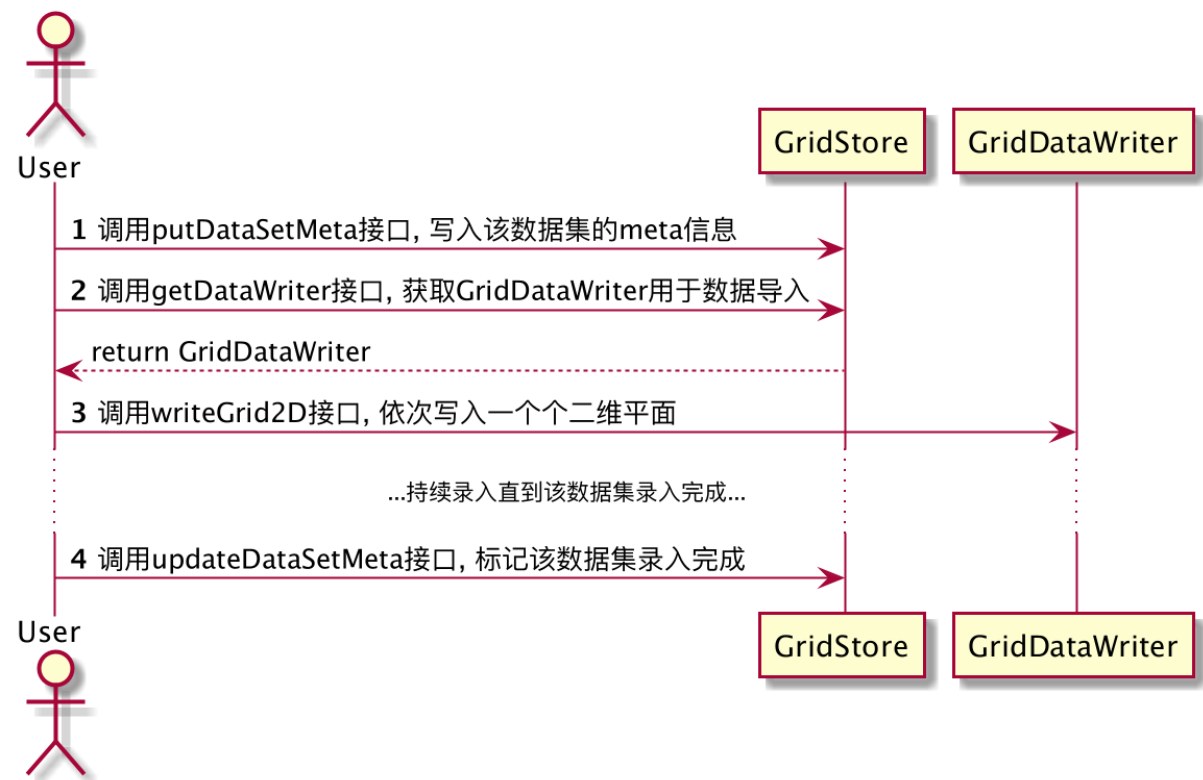
```

```

GridDataFetcher setVariablesToGet(Collection<String> variables);
/**
 * 设置要读取的各维度起始点和大小。
 * @param origin 各维度起始点。
 * @param shape 各维度大小。
 * @return
 */
GridDataFetcher setOriginShape(int[] origin, int[] shape);
/**
 * 获取数据。
 * @return
 * @throws Exception
 */
GridDataSet fetch() throws Exception;
}
    
```

数据录入

TablestoreGrid：数据集录入流程



数据录入流程可以分为三部分：

- 写入putDataSetMeta接口写入数据集的meta信息。
- 通过GridDataWriter录入整个数据集的数据。
- 通过updateDataSetMeta接口更新数据集的meta信息，标记数据已经录入完成。

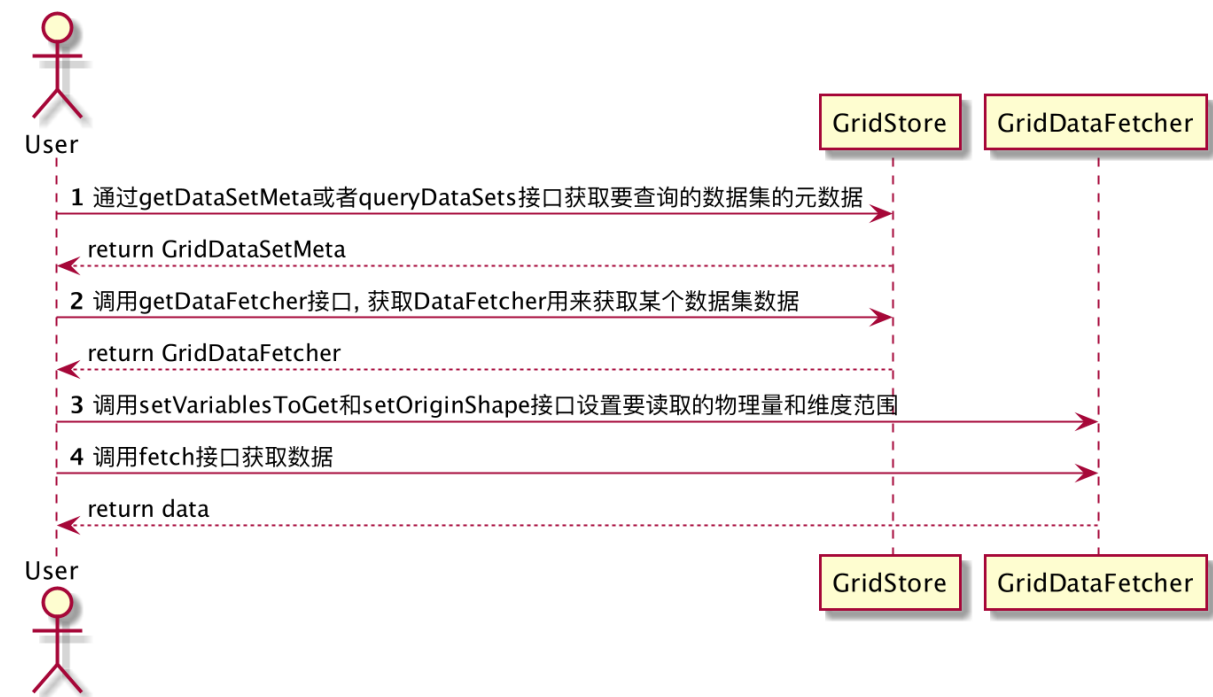
示例如下：

以下示例中读取一个NetCDF（气象格点数据常用的格式）文件，然后将其中的数据通过GridDataWriter录入到表格存储中。通过GridDataWriter每次写入时，只能写入一个二维平面，所以需要在外层进行3层循环，分别枚举变量维、时间维、高度维的值，然后读取对应的二维平面的数据进行录入。

```
public void importFromNcFile(GridDataSetMeta meta, String ncFileName) throws Exception {
    GridDataWriter writer = tableStoreGrid.getDataWriter(meta);
    NetcdfFile ncFile = NetcdfFile.open(ncFileName);
    List<Variable> variables = ncFile.getVariables();
    for (Variable variable : variables) {
        if (meta.getVariables().contains(variable.getShortName())) {
            for (int t = 0; t < meta.gettSize(); t++) {
                for (int z = 0; z < meta.getzSize(); z++) {
                    Array array = variable.read(new int[]{t, z, 0, 0}, new int[]{1, 1, meta
.getxSize(), meta.getySize()});
                    Grid2D grid2D = new Grid2D(array.getDataAsByteBuffer(), variable.getDat
aType(),
                        new int[] {0, 0}, new int[] {meta.getxSize(), meta.getySize()})
;
                    writer.writeGrid2D(variable.getShortName(), t, z, grid2D);
                }
            }
        }
    }
}
```

数据查询

TablestoreGrid：数据查询

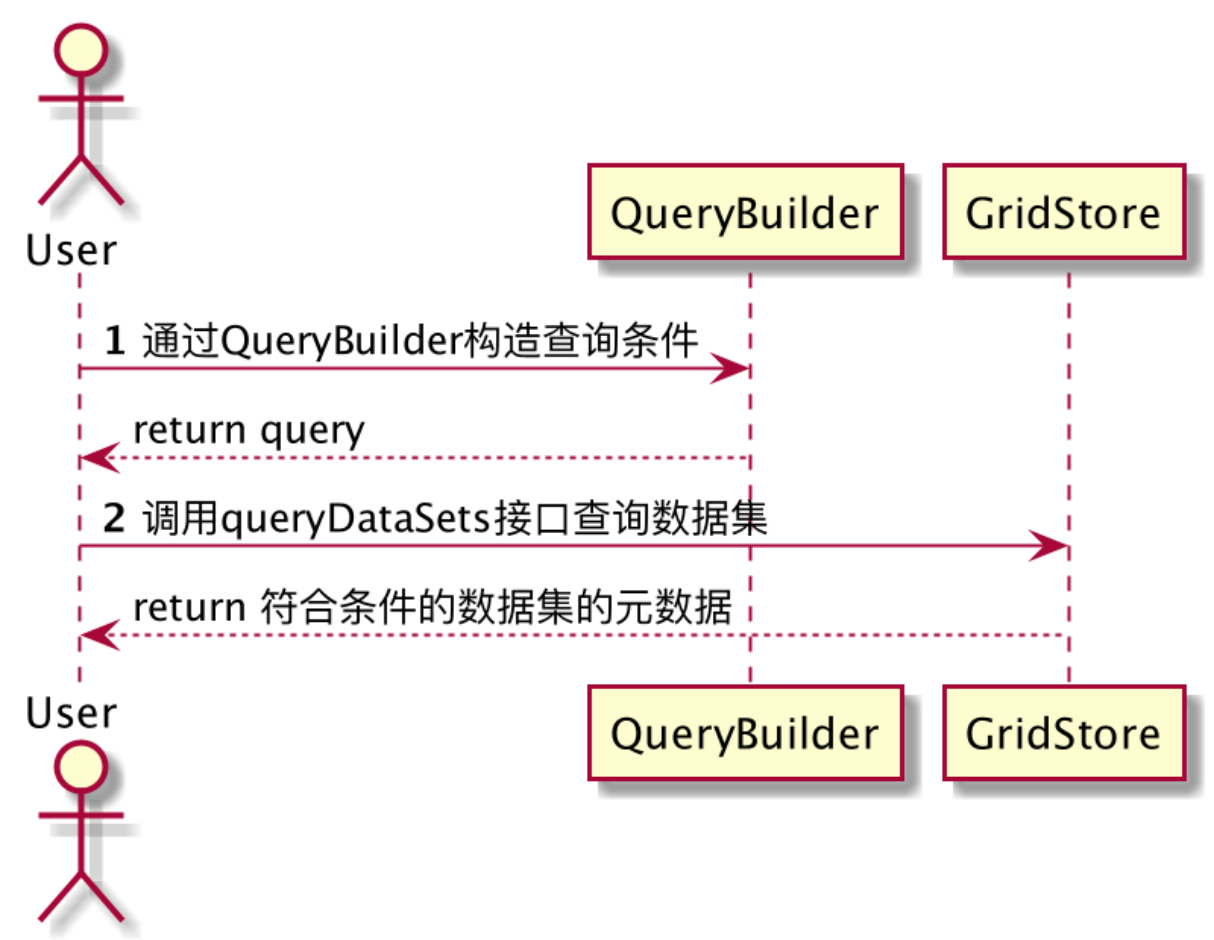


GridDataFetcher支持对多维数据进行任意维度的查询。第一维是变量维，通过setVariablesToGet接口设置要读取哪些变量，其余四维通过设置起始点（origin）和读取的大小（shape）就可以实现任意维度读取。

```
public Array queryByTableStore(String dataSetId, String variable, int[] origin, int[] shape) throws Exception {
    GridDataFetcher fetcher = this.tableStoreGrid.getDataFetcher(this.tableStoreGrid.getDataSetMeta(dataSetId));
    fetcher.setVariablesToGet(Arrays.asList(variable));
    fetcher.setOriginShape(origin, shape);
    Grid4D grid4D = fetcher.fetch().getVariable(variable);
    return grid4D.toArray();
}
```

多条件检索数据集

TablestoreGrid：多条件检索数据集



本方案中，对meta表建立多元索引后，可以支持通过各种组合条件来进行数据集检索，查询出符合条件的数据集，这个功能对于气象管理系统来说非常重要。

示例如下：

假设我们要查询已经完成入库的，创建时间为最近一天的，来源为ECMWF（欧洲中期天气预报中心）或者NMC（全国气象中心），精度为1km的气象预报，并按照创建时间从新到老排序，可以用以下代码实现：

```
查询条件： (status == DONE) and (create_time > System.currentTimeMillis - 86400000) and (source == "ECMWF" or source == "NMC") and (accuracy == "1km")
```

```
QueryGridDataSetResult result = tableStoreGrid.queryDataSets(  
    ExampleConfig.GRID_META_INDEX_NAME,  
    QueryBuilder.and()  
        .equal("status", "DONE")  
        .greaterThan("create_time", System.currentTimeMillis() - 86400000)  
        .equal("accuracy", "1km")  
        .query(QueryBuilder.or()  
            .equal("source", "ECMWF")  
            .equal("source", "NMC")  
            .build())  
        .build(),  
    new QueryParams(0, 10, new Sort(Arrays.<Sort.Sorter>asList(new FieldSort("create_time", SortOrder.DESC)))));
```

这一部分功能利用了表格存储的多元索引，多元索引可以实现多字段组合查询、模糊查询、全文检索、排序、范围查询、嵌套查询、空间查询等功能，给元数据管理场景提供了强大的底层能力。

代码的获取

可以在github上获取Tablestore-Grid的[实现代码和示例代码](#)。

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

6. 亿量级订单管理方案

6.1. 方案背景

订单系统存在于各行各业，例如电商订单、银行流水、运营商话费账单等，是一个非常广泛、通用的系统。随着互联网的发展，以及各企业对数据的重视，需要存储和持久化的订单量越来越大。这些为订单系统管理来了新挑战。

需求场景

某电商平台A，需要存储和维护订单数据。同时，基于所有的订单数据，系统还需要向三类人群提供多元化查询服务：消费者、店家、平台。通过订单系统，消费者可以查询自己的历史订单，商家可以统计热销产品，平台也可以分析用户行为、平台交易规模等。主要查询方式涵盖订单的多维度检索，以及订单数据的分析、统计等，示例如下：

面向消费者：【A消费者】*【近1年】*【卖出电脑】订单查询。

面向售货员：【B售货员】*【近1个月】销售订单。

.....

技术点

在订单场景中，需要考虑的技术点主要包含下几个方面：

- 查询能力：需要具备丰富的查询类型，例如多维度、范围、模糊查询等，同时具备排序、统计等功能。
- 数据量：存储海量数据的同时，满足强一致、高可用、低成本等要求。
- 服务性能：应对高并发请求高并发的同时，保证低延迟。

其中，多维、实时查询功能是订单管理解决方案的核心功能。

6.2. 方案选择

本章节主要为您介绍针对订单系统的一些传统解决方案，以及面对亿量级订单，表格存储提供的更全面的解决方案。



传统方案一：MySQL分库分表

MySQL自身拥有强大的数据查询、分析功能，基于MySQL创建订单系统，可以应对订单数据多维查询、统计场景。伴随着订单数据量的增加，采取分库、分表方案应对，通过这种伪分布式方案解决数据膨胀带来的问题。但数据一旦达到瓶颈，便出现明显的弊端：

- 数据纵向（数据规模）膨胀：采用分库分表方案，MySQL在部署时需要预估分库规模，数据量一旦达到上限后，重新部署并做数据全量迁移。
- 数据横向（字段维度）膨胀：schema需预定义，迭代新增新字段变更复杂。而维度到达一定量后影响数据库性能。

传统方案二：MySQL+HBase

引入双数据的方案应运而生，通过实时数据、历史数据分存的方案，可以一定程度解决数据量膨胀问题。该方案将数据归类成两部分存储：实时数据、历史数据。

- 实时订单数据（例如：近3个月的订单）：将实时订单存入MySQL数据库。实时订单的总量膨胀的速度得到了限制，同时保证了实时数据的多维查询、分析能力。
- 历史订单数据（例如：3个月以前的订单）：通过数据同步服务，将历史订单数据存入HBase，借助于HBase这一分布式NoSQL数据库，有效应对了订单数据膨胀困扰。也保证了历史订单数据的持久化。

但是，该方案牺牲了历史订单数据对用户、商家、平台的使用价值，假设了历史数据的需求频率极低。但是一旦有需求，便需要全表扫描，查询速度慢、I/O成本很高。而维护数据同步又带来了数据一致性、同步运维成本飙升等难题。

传统方案三：MySQL+Elasticsearch

此方案同样是将数据分两部分存储，可以一定程度解决订单索引维度增长问题。用户自己维护数据同步服务，保证两部分数据的一致性。

- 全量数据：将全量的订单数据存入MySQL数据库，订单ID之外的数据整体存为一个字段。该全量数据作为持久化存储，也用于非索引字段的反查。
- 查询数据：仅将需要检索的字段存入Elasticsearch（基于Lucene分布式索引数据库），借助于Elasticsearch的索引能力，提供可以应付维度膨胀的订单数据，然后必要时反查MySQL获取订单完整信息。

该方案应付了数据维度膨胀带来的困扰，但是随着订单量的不断膨胀，MySQL扩展性差的问题再次暴露出来。同时数据同步至Elasticsearch的方案，开发、运维成本很高，方案选择也存在弊端。

表格存储解决方案

由上可以看出，传统的方案并不能完全应对亿量级订单场景。使用表格存储研发的多元索引（SearchIndex）方案，则可以完美地解决亿量级订单系统问题。表格存储具有即开即用，按量收费等特点。多元索引随时创建，是海量电商订单元数据管理的优质方案。

表格存储作为阿里云提供的一款全托管、分布式NoSQL型数据存储服务，具有海量数据存储、热点数据自动分片、海量数据多维检索等功能，天然地解决了订单数据大爆炸这一挑战。同时，多元索引功能在保证用户数据高可用的基础上，提供了数据多维度搜索、统计等能力。针对多种场景创建多种索引，实现多种模式的检索。可以仅在需要的时候创建、开通索引。由表格存储来保证数据同步的一致性，这极大的降低了用户的方案设计、服务运维、代码开发等工作量。

下图为表格存储与传统订单方案使用的工具在各项性能上的对比：

能力分析	MySql	HBase	Elasticsearch	TableStore
存储方式	行存储	列存储	索引存储	列存储+索引存储
扩展性	单机、扩展性差	水平扩展	水平扩展	(自动)水平扩展
一致性	强一致性	强一致性、时序一致性		强一致性、时序一致性
检索	较弱的支持	不支持	支持	支持
数据量	~ 1T，~亿行	~10 PB，~万亿行	~1 PB，~千亿行	~10 PB，~万亿行

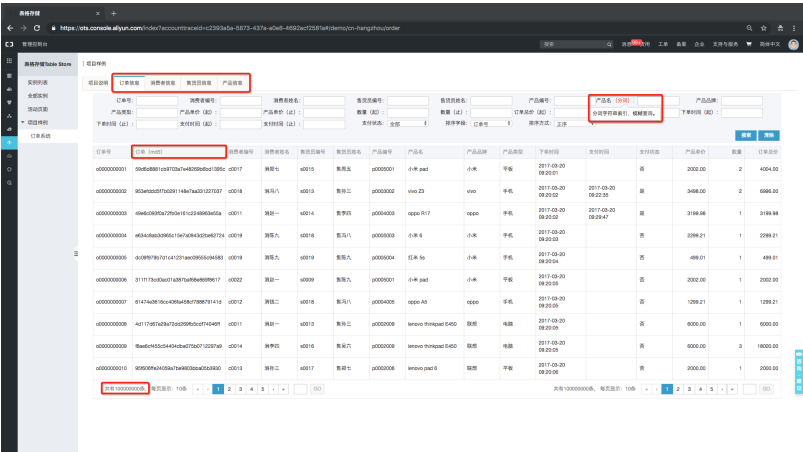
- 方案实现

请参见[准备工作](#)以及[搭建订单系统](#)。

● 方案样例

基于表格存储搭建的订单系统样例内嵌在表格存储控制台中，用户可登录控制台体验系统。该样例提供了亿量级订单数据，详细请参见[项目样例](#)。

 **说明** 若为表格存储的新用户，需要单击开通服务后体验，开通免费，订单数据存储于公共实例中，体验不消耗用户存储、流量、读写吞吐。



6.3. 准备工作

开始使用表格存储搭建订单系统前，您需要完成以下准备工作。

开通表格存储

已开通表格存储服务。具体操作，请参见[开通表格存储服务](#)。

创建表格存储实例

实例是您使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。

创建表格存储实例。具体操作，请参见[创建实例](#)。

下载SDK

选择以下语言的SDK包进行下载安装：

- [Java SDK](#)
- [Python SDK](#)
- [Go SDK](#)
- [Node.js SDK](#)
- [.NET SDK](#)
- [PHP SDK](#)

设计数据表

订单系统不仅仅是订单一张数据表，它应包含：消费者表、售货员表、产品表、供货商表、交易订单表、支付订单表等。在本次搭建示例中，主要使用最基本的四张表（消费者表、售货员表、产品表、交易订单表）。

表名：order_contract

列名	数据类型	索引类型	字段说明
_id(主键列)	String		MD5(oId)避免热点
oId(主键列)	String	KEYWORD	订单编号
pName	String	TEXT	产品名，TEXT类型索引可模糊查询，但不能排序
totalPrice	double	DOUBLE	订单总价
orderTime	long	LONG	下单时间（时间戳）
...	...	...	...

6.4. 搭建订单系统

本章节主要为您介绍如何使用表格存储搭建亿量级订单管理系统。

前提条件

您已经完成了[准备工作](#)。

步骤一：创建数据表

创建四张表：订单表、消费者表、售货员表、产品表。您仅需将四张表创建在同一个实例，您可以通过以下两种方式创建数据表：

- 通过控制台创建、管理数据表，具体参见[创建数据表](#)。
- 通过SDK直接创建、管理数据表。

步骤二：创建数据表多元索引

表格存储自动做全量、增量的索引数据同步。您可以通过以下两种方式创建、管理多元索引：

- 通过控制台创建、管理多元索引，具体参见[使用控制台](#)。
- 通过SDK创建、管理多元索引。

步骤三：导入数据

插入部分测试数据。

 **说明** 控制台[项目样例](#)中插入了1亿条数据，您可以通过控制台插入少量测试数据。

订单号	订单 (md5) (主键)	消费者编号	消费者姓名	售货员编号	售货员姓名	产品编号	产品名	产品品牌	产品类型	下单时间	支付时间	支付状态	产品单价	数量	总价钱
0000000000000000	c49f5fd5aba33159accacae0d3ecd749a7	c0001	消费陈九	s0001	售楚十	p0001001	vi vox21	vivox	手机	2018-07-07 14:44:39		否	2	2	4

消费者编号 (主键)	消费者姓名	消费者积分	注册时间
c0001	消费赵一	818	2018-07-07 14:33:51

售货员编号 (主键)	售货员姓名	售货员积分	入职日期
s0001	售赵一	613	2018-07-07 14:27:59

产品编号 (主键)	产品名	产品品牌	产品类型	产品单价	新增时间
p0001001	iphone 6	苹果	手机	6969.00	2018-07-07 14:44:39

步骤四： 读取数据

数据读取分为两类：

- 主键读取
基于原生表格存储的主键列获取：getRow、getRange、batchGetRow等。主键读取用于索引（自动）反查，用户也可以提供主键（订单md5）的单条查询的页面，亿量级下查询速度极快。单主键查询方式不支持多维度检索。
- 索引读取
基于新SearchIndex功能Query：search接口。用户可以自由设计索引字段的多维度条件组合查询。通过设置选择不同的查询参数，构建不同的查询条件、不同排序方式；目前支持：精确查询、范围查询、前缀查询、匹配查询、通配符查询、短语匹配查询、分词字符串查询，并通过布尔与、或组合。

如【c0001号消费者，且消费在99.99以上的订单】组合方式如下：

```
List<Query> mustQueries = new ArrayList<Query>();

TermQuery termQuery = new TermQuery();
termQuery.setFieldName("cId");
termQuery.setTerm(ColumnValue.fromString("c0001"));
mustQueries.add(termQuery);

RangeQuery rangeQuery = new RangeQuery();
rangeQuery.setFieldName("totalPrice");
rangeQuery.setFrom(ColumnValue.fromDouble(99.99));
mustQueries.add(rangeQuery);

BoolQuery boolQuery = new BoolQuery();
boolQuery.setMustQueries(mustQueries);
```

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

7. 基于多元索引搭建亿量级店铺搜索系统

7.1. 方案背景

本章节主要为您介绍基于多元索引搭建亿量级店铺搜索系统的需求场景及方案。

亿量级店铺搜索系统的核心点与瓶颈在于数据库是否有海量存储能力、低延迟读能力、高效查询能力以支持GEO+多维度数据检索。表格存储作为阿里云提供的一款全托管、零运维的分布式NoSQL型数据存储服务，具有海量数据存储、热点数据自动分、海量数据多维检索等功能，可以有效地解决GEO数据量大膨胀这一挑战。

需求场景

某店铺搜索平台，提供了亿量级的店铺信息。用户通过平台提供的PC端、移动端网页，按照自己的需求维度组合，搜索用户心仪的店铺。平台需要在地图上展示店铺的具体位置、店铺详细信息、店铺主页的跳转。

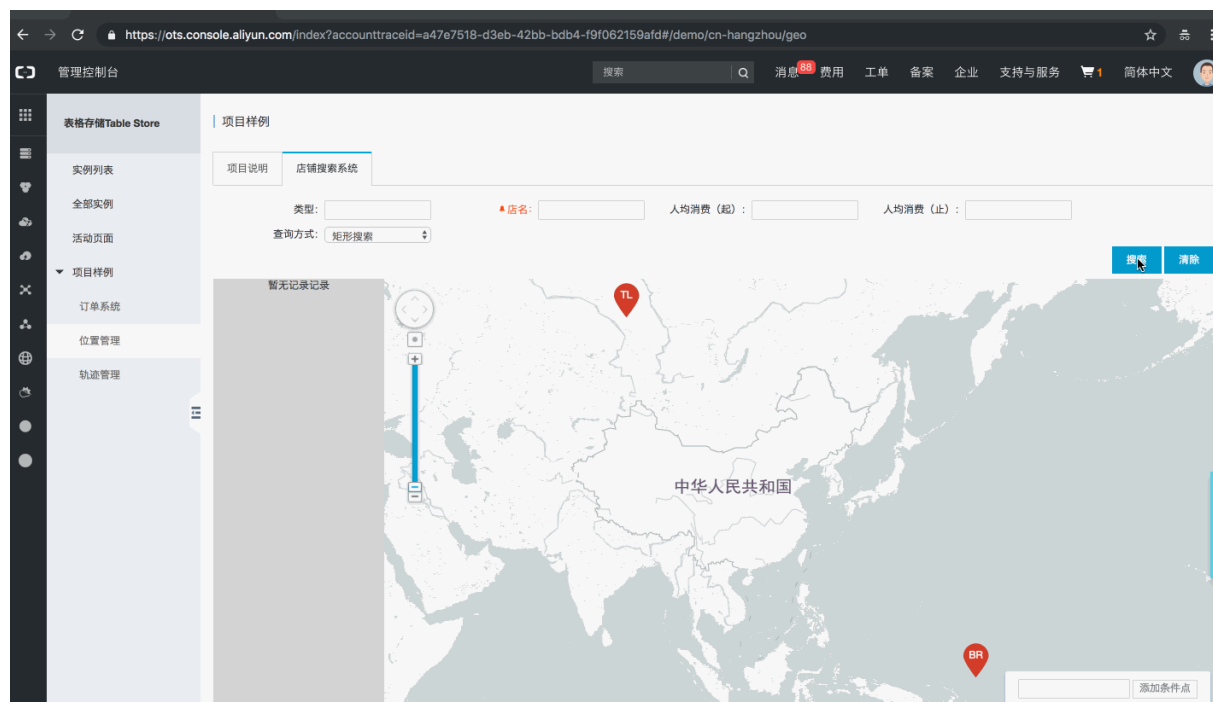
维度一：【距离1km内】【人均100以内】【评分最高】【奶茶店】

维度二：【杭州市内】【评分最高的】【沈家*】店铺

.....

实现快速、多维GEO查询功能，是GEO管理解决方案的核心功能，样例如下。

 说明 该样例提供了【亿量级】店铺数据。具体参见[项目样例](#)。



基于表格存储搭建的店铺搜索系统页面一览，样例内嵌在表格存储控制台中，用户可登录控制台体验系统（若为表格存储的新用户，需要单击开通服务后体验，开通免费，订单数据存储于公共实例中，体验不消耗用户存储、流量、CU）。

表格存储方案

使用表格存储研发的多元索引功能，可以轻松搭建一套亿量级店铺搜索系统。多元索引功能可以创建GEO索引、分词字符串索引等，为用户提供了GEO检索、多维组合检索等能力，用户可随时创建，存量、增量数据自动同步。

用户可以仅在需要的时候创建、开通索引。由表格存储来保证数据同步的一致性，这极大的降低了用户的方案设计、服务运维、代码开发等工作量。

7.2. 准备工作

开始使用表格存储搭建亿量级店铺搜索系统前，您需要完成以下准备工作。

开通表格存储

已开通表格存储服务。具体操作，请参见[开通表格存储服务](#)。

创建表格存储实例

实例是您使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。

创建表格存储实例。具体操作，请参见[创建实例](#)。

下载SDK

选择以下语言的SDK包进行下载安装：

- [Java SDK](#)
- [Python SDK](#)
- [Go SDK](#)
- [Node.js SDK](#)
- [.NET SDK](#)
- [PHP SDK](#)

设计数据表

店铺检索系统样例，仅简易使用一张店铺表，主要包含字段：店铺类型、店铺名称、店铺地理位置、店铺平均评分、人均消费消等。表设计如下：

表名：geo_position

列名	数据类型	索引类型	字段说明
_id(主键列)	String		MD5(pId)避免热点
pId	String		店铺编号
type	String	KEYWORD	类型
name	String	TEXT	店铺名，TEXT类型索引可模糊查询，但不能排序
pos	String	GEO_POINT	店铺位置："30.132,120.082"(纬度,精度)
point	double	DOUBLE	评分
...	...	...	...

7.3. 方案实现

本章节主要为您介绍如何使用表格存储搭建亿量级店铺搜索系统。

前提条件

您已经完成了[准备工作](#)。

步骤一：创建数据表

通过以下两种方式创建一张店铺信息表：

- 通过控制台创建、管理数据表，具体请参见[创建数据表](#)。
- 通过SDK直接创建、管理数据表。

步骤二：创建数据表索引

表格存储自动进行全量、增量的索引数据同步。您可以通过以下两种方式创建、管理多元索引：

- 通过控制台创建、管理多元索引，具体请参见[使用控制台](#)。
- 通过SDK创建、管理多元索引。

步骤三：导入数据

控制台[项目样例](#)中插入了1亿条数据，您可以通过控制台插入少量测试数据。

店铺编号	店铺 (md5) (主键)	类型	店铺名称	店铺位置	店铺评分	人均消费
o0057022192	0000000f470ef0ef0f0f548b925ceffela7e3	杭帮菜	韩村杭帮菜	36.76613, 111.41461	2.87	63.67

步骤四：读取数据

数据读取分为两类：

- 主键读取

基于原生表格存储的主键列获取。主键读取用于索引（自动）反查，用户也可以提供主键（订单md5）的单条查询的页面，亿量级下查询速度极快。单主键查询方式不支持多维度检索。

- 索引读取（店铺查询）

基于新SearchIndex功能Query：search接口。用户可以自由设计索引字段的多维度条件组合查询。通过设置选择不同的查询参数，构建不同的查询条件、不同排序方式；目前支持：精确查询、范围查询、前缀查询、匹配查询、通配符查询、短语匹配查询、分词字符串查询，并通过布尔与、或组合。

例如：“36.76613,111.41461”周边1km范围内的奶茶店，查询条件如下：

```
List<Query> mustQueries = new ArrayList<Query>();
TermQuery termQuery = new TermQuery();
termQuery.setFieldName("type");
termQuery.setTerm(ColumnValue.fromString(奶茶));
mustQueries.add(termQuery);
GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();
geoDistanceQuery.setFieldName("pos");
geoDistanceQuery.setCenterPoint("36.76613,111.41461");
geoDistanceQuery.setDistanceInMeter(1000);
mustQueries.add(geoDistanceQuery);
BoolQuery boolQuery = new BoolQuery();
boolQuery.setMustQueries(mustQueries);
```

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

8.表格存储结合Spark流批处理实现一体化存储和计算

8.1. 方案背景

介绍表格存储结合Spark流批处理实现一体化存储和计算的背景、适用场景、样例场景和架构设计。

背景

电子商务模式是指在网络环境和大数据环境下基于一定技术基础的商务运作方式和盈利模式。

在电商运营中数据的分析和可视化是最重要的部分之一，而通过电商大屏可以实现数据分析和可视化的完美结合。

电商大屏中包含全量订单和实时订单的聚合结果，全量订单的聚合结果展示全景的综合数据视图，实时订单的聚合结果展示实时的运营指标数据。

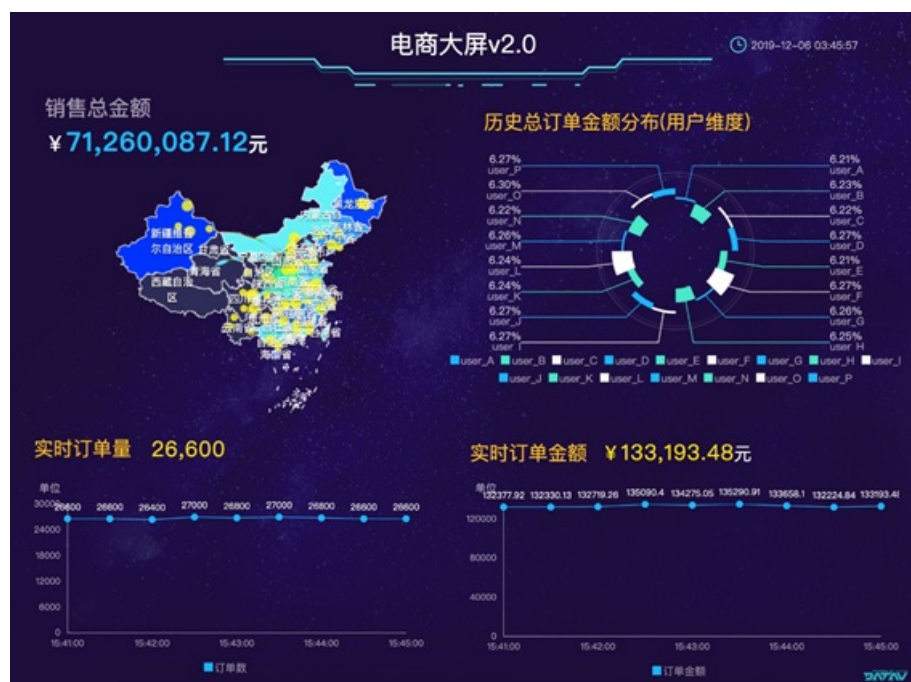
适用场景

方案适用于以下场景，请根据实际业务需要使用。

- 大数据计算场景：推荐系统，订单类业务等
- 实时数据计算场景：监控系统，电商数字大屏等

样例场景

某电商公司为了完成电商运营数据的分析和展示，使用表格存储统一存储电商数据，通过Spark的流批处理离线聚合或实时统计电商数据，最终在DataV大屏上实现电商数据的可视化展示，效果图如下。



架构设计

样例场景的实现过程说明如下，架构图如下图所示。

1. SQL客户端将原始订单数据实时插入到表格存储的数据表中。
2. 通过Spark实现实时数据和离线数据处理并将聚合结果保存到表格存储的数据表中。
 - 实时流计算通过Spark Structured Streaming实时统计一个窗口周期内订单数量和订单金额的聚合结果。
 - 离线批计算通过Spark SQL离线聚合原始订单数据的总金额和用户维度总金额。
3. 通过DataV将聚合结果展示在大屏上。



8.2. 准备工作

本文介绍环境准备和数据准备工作。

环境准备

- 已创建阿里云E-MapReduce的Hadoop集群。具体操作，请参见[创建集群](#)。
- 已下载E-MapReduce的最新SDK，SDK包的名称格式为emr-datascources_shaded_*.jar，emr-datascources_shaded_*.jar中包含Tablestore相关的Spark批流Source和Sink。
- 已开通表格存储服务并创建实例。具体操作，请参见[通过控制台使用](#)或者[通过命令行工具使用](#)。
- 当阿里云E-MapReduce的Hadoop集群在VPC中时，请确保已绑定表格存储的实例和E-MapReduce的Hadoop集群所在的VPC。具体操作，请参见[（可选）绑定VPC](#)。
- 已开通DataV服务并制作大屏。具体操作，请参见[开通DataV服务](#)。

数据准备

产品	数据收集	说明
E-MapReduce的Hadoop集群	登录EMR Header服务器的IP地址、用户名和密码。	用于通过远程登录工具（如putty等）登录EMR Header服务器进行Spark流批处理。

产品	数据收集	说明
表格存储Tablestore	- 访问表格存储服务的AccessKey ID和AccessKey Secret。 - 登录控制台的用户名和密码。 - 实例的名称和服务地址。	用于访问表格存储服务，将原始数据和聚合结果写入表格存储的数据表中。  说明 在表格存储中通过控制台或者SDK创建存储电商数据的原始订单表 OrderSource，OrderSource表有两个主键UserId（用户ID）和OrderId（订单ID）和两个属性列price（价格）和timestamp（订单时间）。
DataV数据可视化	登录DataV的用户名和密码。	用于添加表格存储数据源在大屏上显示数据。

8.3. 方案实现

介绍实现方案的详细配置操作，包括创建数据表、实时流计算、离线批计算和DataV展示。

步骤一：创建数据表

1. 使用远程登录工具登录EMR Header服务器。
2. 执行以下命令，启动SQL客户端。

SQL客户端用于批流的SQL计算，其中emr-datasources_shaded_*.jar为准备工作中下载的EMR最新版SDK包。

```
streaming-sql --driver-class-path emr-datasources_shaded_*.jar --jars
emr-datasources_shaded_*.jar --master yarn-client --num-executors 8
--executor-memory 2g --executor-cores 2
```

3. 创建原始订单数据表（OrderSource表）的外表order_source。
外表order_source用于流批处理的SQL执行。

```
DROP TABLE IF EXISTS order_source;
CREATE TABLE order_source
USING tablestore
OPTIONS (
  endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
  access.key.id="",
  access.key.secret="",
  instance.name="vehicle-test",
  table.name="OrderSource",
  tunnel.id="2b7bbf3d-d6c4-4cea-89fe-71998bccaf19",
  catalog='{"columns": {"UserId": {"col": "UserId", "type": "string"},
    "OrderId": {"col": "OrderId", "type": "string"},
    "price": {"cols": "price", "type": "double"},
    "timestamp": {"cols": "timestamp", "type": "long"}}}'
);
```

参数	说明
endpoint	表格存储实例的访问地址。
access.key.id	阿里云账号AccessKey ID。
access.key.secret	阿里云账号AccessKey Secret。
instance.name	表格存储的实例名。
table.name	表格存储的表名。
tunnel.id	表格存储的增量通道ID，用于实时的增量SQL，批量SQL时非必须。
catalog	表的字段Schema定义，示例代码中对应的四个列分别为UserId（主键）、OrderId（主键）、price、timestamp，数据类型分别为string、string、double、long。

步骤二：实时流计算

实时流计算将实时统计一个窗口周期时间内的订单数和订单金额统计，并将聚合结果写回表格存储的数据表中。

- 1. 创建流计算的Sink外表order_stream_sink（对应表格存储中的OrderStreamSink表）。
- 2. 执行实时流计算SQL命令进行实时聚合，实时得到聚合结果，并将聚合结果实时写回表格存储的OrderStreamSink表中。

Sink表的各参数含义和Source表一致，其中catalog字段的内容有所不同，Sink表的catalog中四个字段分别为begin（开始时间，主键列，格式为2019-11-27 14:54:00）、end（结束时间，主键列）、count（订单数）、totalPrice（订单总金额），数据类型分别为string、string、long、double。

```
//创建Sink表order_stream_sink对应表格存储的OrderStreamSink表（主键为begin和end两列）。
DROP TABLE IF EXISTS order_stream_sink;
CREATE TABLE order_stream_sink
USING tablestore
OPTIONS (
endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
access.key.id="",
access.key.secret="",
instance.name="vehicle-test",
table.name="OrderStreamSink",
catalog='{ "columns": { "begin": { "col": "begin", "type": "string" },
"end": { "col": "end", "type": "string" },
"count": { "col": "count", "type": "long" },
"totalPrice": { "col": "totalPrice",
"type": "double" } } }'
);
//在order_source表上创建视图order_source_stream_view。
CREATE SCAN order_source_stream_view ON order_source USING STREAM OPTIONS
("maxoffsetsperchannel"="10000");
//在视图order_source_stream_view上运行STREAM SQL作业，以下样例会按30s粒度进行订单数和订单金额的聚合。
//聚合结果将写回表格存储的OrderStreamSink表。
CREATE STREAM job1
options (
checkpointLocation='/tmp/spark/cp/job1',
outputMode='update'
)
INSERT INTO order_stream_sink
SELECT CAST(window.start AS String) AS begin, CAST(window.end AS String)
AS end, count(*) AS count, CAST(sum(price) AS Double) AS totalPrice FROM
order_source_stream_view GROUP BY window(to_timestamp(timestamp / 1000),
"30 seconds");
```

聚合结果样例如下图所示。

数据源: OrderStreamSink表格数据最多显示60行。

	详细数据	begin(主键)	end(主键)	count	totalPrice
☐	详细数据	2019-12-06 10:55:30	2019-12-06 10:56:00	18800	93988.24999999994
☐	详细数据	2019-12-06 10:55:00	2019-12-06 10:55:30	23800	119184.13
☐	详细数据	2019-12-06 10:54:30	2019-12-06 10:55:00	25400	127011.12999999998
☐	详细数据	2019-12-06 10:54:00	2019-12-06 10:54:30	23600	118741.59999999985
☐	详细数据	2019-12-06 10:53:30	2019-12-06 10:54:00	26200	130843.28000000006
☐	详细数据	2019-12-06 10:53:00	2019-12-06 10:53:30	22800	113727.050000000025
☐	详细数据	2019-12-06 10:52:30	2019-12-06 10:53:00	26200	131344.82999999993
☐	详细数据	2019-12-06 10:52:00	2019-12-06 10:52:30	24800	122971.66000000043

步骤三：离线批计算

离线批计算将进行原始订单数据的总金额和用户维度总金额的离线聚合。

- 1. 创建两张Sink表（OrderTotalSink表和OrderBatchSink表）分别存放历史总金额和用户维度总金额的聚合数据。

```
//批计算任务。
//用户维度结果表OrderBatchSink（主键UserId，属性列count、totalPrice）。
//总数据维度结果表OrderTotalSink（主键Count，属性列totalPrice）。
DROP TABLE IF EXISTS order_batch_sink;
CREATE TABLE order_batch_sink
USING tablestore
OPTIONS(
endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
access.key.id="",
access.key.secret="",
instance.name="vehicle-test",
table.name="OrderBatchSink",
tunnel.id="",
catalog='{ "columns": { "UserId": { "col": "UserId", "type": "string" },
"count": { "col": "count", "type": "long" },
"totalPrice": { "col": "totalPrice", "type": "double" } } }'
);
DROP TABLE IF EXISTS order_total_sink;
CREATE TABLE order_total_sink
USING tablestore
OPTIONS(
endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
access.key.id="",
access.key.secret="",
instance.name="vehicle-test",
table.name="OrderTotalSink",
tunnel.id="",
catalog='{ "columns": { "count": { "col": "count", "type": "long" },
"totalPrice": { "col": "totalPrice", "type": "double" } } }'
);
```

2. 在Source表（order_source表）上执行批计算SQL命令，得到聚合结果，并将聚合结果分别写回表格存储的Sink表（OrderTotalSink表和OrderBatchSink表）中。
 - o 执行批计算SQL语句更新用户维度的聚合结果。

```
//SQL命令。
INSERT INTO order_batch_sink SELECT UserId, count(*) AS count, sum(price)
AS totalPrice FROM order_source GROUP BY UserId;
//实际运行。
spark-sql> INSERT INTO order_batch_sink SELECT UserId, count(*) AS count,
sum(price) AS totalPrice FROM order_source GROUP BY UserId;
Time taken: 5.107 seconds
```

用户维度的聚合结果样例如下图所示。

数据源: OrderBatchSink		表格数据最多显示50行。		
	详细数据	User(主键)	count	totalPrice
☐	详细数据	user_A	75688	376216.4799999986
☐	详细数据	user_B	75475	377251.8799999907
☐	详细数据	user_C	75249	376723.30000000127
☐	详细数据	user_D	75634	379666.51000000123
☐	详细数据	user_E	75404	375893.8099999992
☐	详细数据	user_F	75950	379500.2100000018
☐	详细数据	user_G	75931	379052.8499999999
☐	详细数据	user_H	75766	378606.32000000036
☐	详细数据	user_I	75943	379577.4199999996
☐	详细数据	user_J	75754	379299.70000000094

- 执行批计算SQL命令更新总数据维度的结果。

```
//SQL命令。  
INSERT INTO order_total_sink SELECT count(*) AS count, sum(price)  
AS totalPrice FROM order_source;  
//实际运行。  
spark-sql> INSERT INTO order_total_sink SELECT count(*) AS count,  
sum(price) AS totalPrice FROM order_source;  
Time taken: 4.272 seconds
```

总数据维度的聚合结果样例如下图所示。

数据源: OrderTotalSink		表格数据最多显示50行。	
	详细数据	count(主键)	totalPrice
☐	详细数据	14251626	7.126008712000002E7
☐	详细数据	10468050	5.234753223000003E7
☐	详细数据	8080201	4.030721644000016E7
☐	详细数据	2967266	1.4834135849999985E7
☐	详细数据	2248788	1.1244914189999966E7

步骤四：DataV展示

通过在DataV中添加Tablestore数据源，实现在DataV的大屏上展示实时流处理或离线批处理聚合结果。添加Tablestore数据源的详细操作请参见[添加Tablestore数据源](#)。

9.表格存储结合实时计算Flink进行大数据分析

9.1. 方案背景

本文介绍了表格存储结合实时计算Flink实现大数据分析的样例场景、架构设计等。

背景信息

云数据库RDS MySQL基于阿里巴巴的MySQL源码分支，经过双十一高并发、大数据量的考验，拥有优良的性能。RDS MySQL支持实例管理、账号管理、数据库管理、备份恢复、白名单、透明数据加密以及数据迁移等基本功能。更多信息，请参见[云数据库RDS MySQL概述](#)。

阿里云实时计算Flink版是一套基于Apache Flink构建的一站式实时大数据分析平台，提供端到端亚秒级实时数据分析能力，并通过标准SQL降低业务开发门槛，助力企业向实时化、智能化大数据计算升级转型。更多信息，请参见[什么是阿里云实时计算Flink版](#)。

表格存储（Tablestore）是阿里云自研的多模型结构化数据存储，可提供海量结构化数据的存储和查询分析服务。表格存储的分布式存储和强大的索引引擎能够支持PB级存储、千万TPS以及毫秒级延迟的服务能力。更多信息，请参见[什么是表格存储](#)。多元索引基于倒排索引、列式存储、空间索引等，可解决大数据的复杂查询、分析聚合等需求。通过为数据表创建多元索引，可实现全文检索、前缀查询、模糊查询、组合查询、统计聚合等功能。

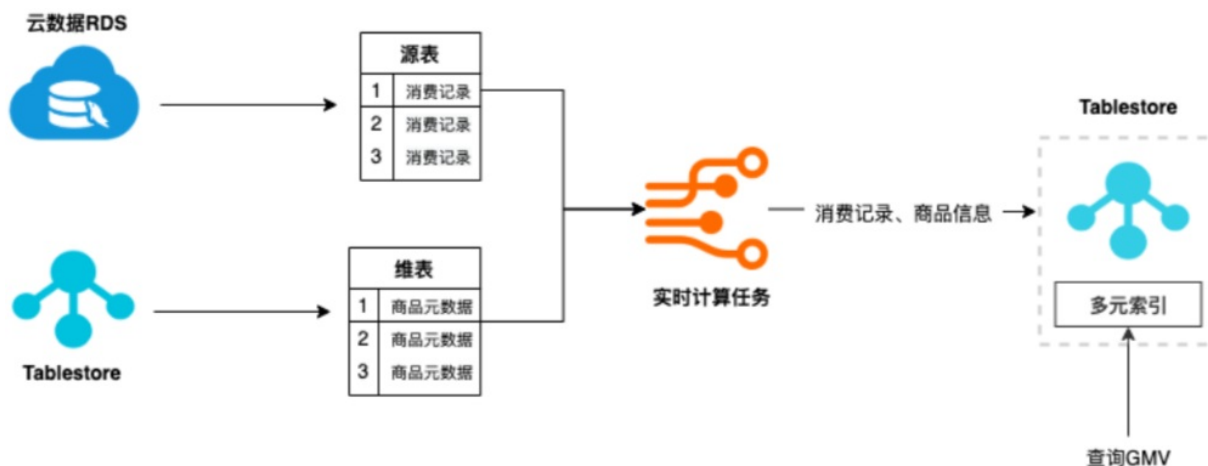
样例场景

某大型连锁超市会实时产生大量的消费数据，通过分析这些数据统计每分钟不同商品类别的GMV（商品交易总额），得到商品售卖热度、门店经营状态等信息，便于辅助经营者的商业决策。

架构设计

样例场景的实现过程说明如下，架构图如下图所示。

1. 使用云数据库RDS作为商品消费订单的存储库并通过mysql-cdc connector作为源表接入实时计算Flink。
2. 使用表格存储作为商品元信息的存储库并通过ots connector作为维表接入实时计算Flink。
3. 在Flink流计算作业中关联消费记录和商品信息数据并写入表格存储中。
4. 基于表格存储多元索引或者SQL查询功能进行数据分析。



9.2. 准备工作

本文介绍场景中RDS MySQL、实时计算Flink和表格存储产品的准备工作。

使用RDS MySQL服务

首次使用RDS MySQL时，请根据如下步骤创建RDS MySQL实例以及创建数据库和账号等。

1. 创建RDS MySQL实例。具体操作，请参见[创建RDS MySQL实例](#)。
2. 创建数据库和账号。具体操作，请参见[创建数据库和账号](#)。

使用实时计算Flink服务

首次使用实时计算Flink时，请根据如下步骤开通实时计算Flink服务以及创建Flink全托管实例。

1. 开通实时计算Flink服务。具体操作，请参见[开通实时计算Flink服务](#)。
2. 创建Flink全托管实例。具体操作，请参见[创建Flink全托管实例](#)。

使用表格存储服务

首次使用表格存储时，请根据如下步骤开通表格存储服务以及创建实例。

1. 开通表格存储服务。具体操作，请参见[开通表格存储服务](#)。
2. 创建实例。具体操作，请参见[创建实例](#)。

9.3. 方案实现

本文介绍实现方案的详细配置操作，包括在RDS MySQL中创建源表、在表格存储中创建结果表、在实时计算Flink中创建作业并启动和在表格存储中分析数据。

步骤一：在RDS MySQL中创建源表

1. 通过DMS登录RDS MySQL。具体操作，请参见[通过DMS登录RDS MySQL](#)。
2. 创建数据表consume_record。

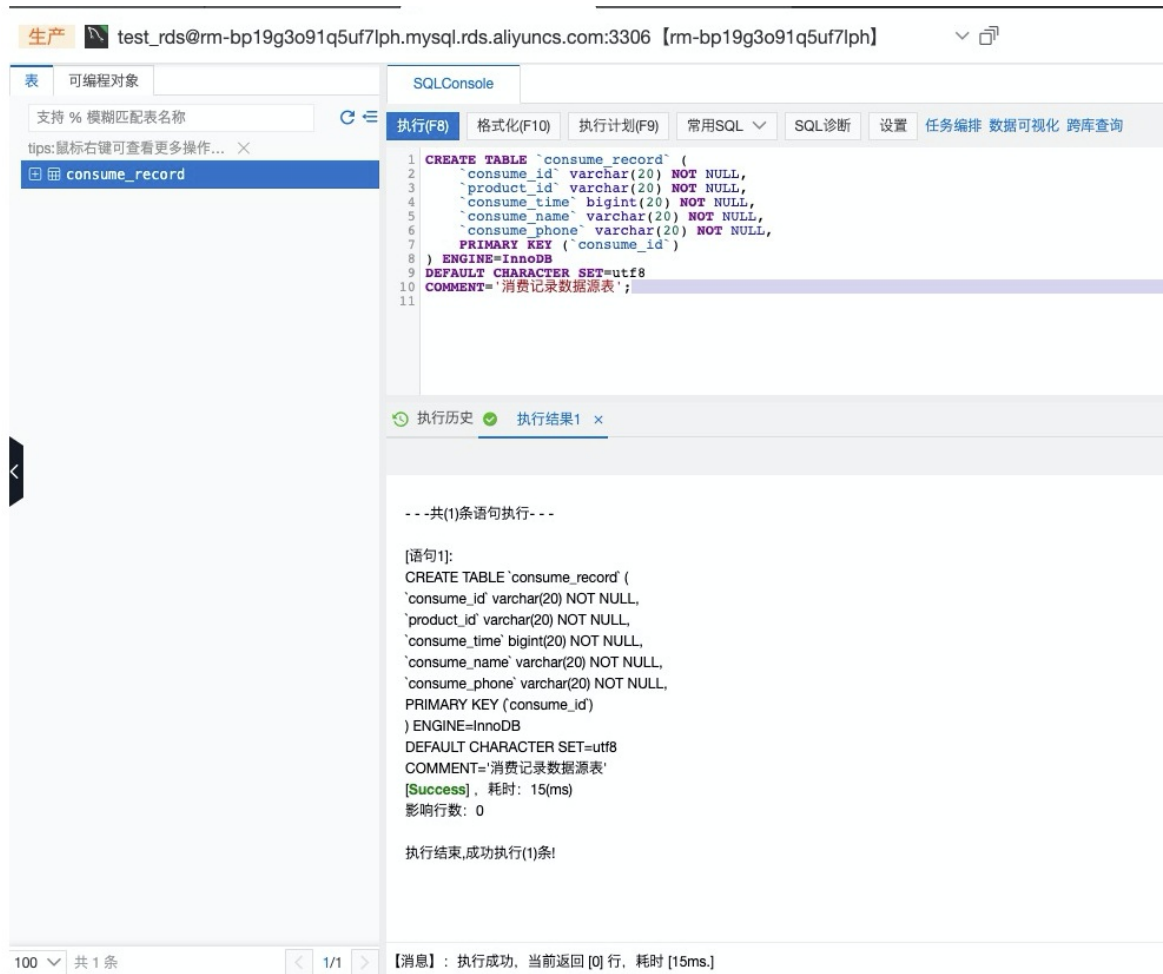
consume_record数据表作为流计算任务源表，用于接入实时计算Flink。

- i. 编写创建数据表的SQL语句。

本方案中SQL示例如下：

```
CREATE TABLE `consume_record` (  
  `consume_id` varchar(20) NOT NULL,  
  `product_id` varchar(20) NOT NULL,  
  `consume_time` bigint(20) NOT NULL,  
  `consume_name` varchar(20) NOT NULL,  
  `consume_phone` varchar(20) NOT NULL,  
  PRIMARY KEY (`consume_id`)  
) ENGINE=InnoDB  
DEFAULT CHARACTER SET=utf8  
COMMENT='消费记录数据源表';
```

- ii. 单击执行(F8)。



步骤二：在表格存储中创建维表和结果表

- 1. 登录表格存储控制台。
- 2. 创建数据表consume_product和product。具体操作，请参见创建数据表。

说明 由于表格存储的数据表是schema free的，只需要定义主键，无需定义属性列。

consume_product数据表作为流计算任务结果表，用于保存商品消费信息与商品元数据信息。表结构请参见下表。

字段名称	数据类型	是否主键	描述
consume_id	STRING	是	消费ID（主键）。
product_id	STRING	是	商品ID。
price	DOUBLE	否	商品单价。
consume_time	BIGINT(20)	否	消费时间。
consume_name	STRING	否	消费者名称。
consume_phone	STRING	否	消费者联系方式。

product数据表流计算任务维表，用于保存商品元数据信息。表结构请参见下表。

字段名称	数据类型	是否主键	描述
product_id	STRING	是	商品ID。
price	DOUBLE	否	商品单价。
product_type	STRING	否	商品类别。

步骤三：在实时计算Flink中创建作业并启动

- 1. 登录实时计算管理控制台。
- 2. 创建作业gmv_post_aggregation。
 - i. 在Flink全托管页签，单击目标工作空间操作列下的控制台。
 - ii. 在左侧导航栏，单击作业开发。
 - iii. 单击新建。
 - iv. 在新建文件对话框，填写作业配置信息。

作业参数	示例	说明
文件名称	flink-test	作业的名称。  说明 作业名称在当前项目中必须保持唯一。
文件类型	流作业/SQL	选择流作业中的SQL。
部署目标	vvp-workload	选择作业需要部署的集群名称。Flink全托管支持Per-Job集群和Session集群两种集群模式。两种集群模式的区别说明，请参见配置开发测试环境（Session集群）。
存储位置	作业开发	指定该作业的代码文件所属的文件夹。默认存放在作业开发目录。 您还可以在现有文件夹右侧，单击  图标，新建子文件夹。

- v. 单击确认。
- 3. 配置作业并上线。
 - i. 将以下作业代码示例拷贝到作业文本编辑区并根据实际修改配置信息。

```
-- mysql-cdc源表。
CREATE TEMPORARY TABLE consume_record (
  `consume_id` VARCHAR(20),
  `product_id` VARCHAR(20),
  `consume_time` BIGINT,
  `consume_name` VARCHAR(20),
  `consume_phone` VARCHAR(20),
  PRIMARY KEY(consume_id) NOT ENFORCED
) WITH (
```

```
'connector' = 'mysql-cdc',
'hostname' = '<hostname>',
'port' = '3306',
'username' = '',
'password' = '',
'database-name' = '',
'table-name' = ''
);
-- tablestore维表。
CREATE TEMPORARY TABLE product (
  product_id STRING,
  price DOUBLE,
  product_type STRING,
  PRIMARY KEY (product_id) NOT ENFORCED
) WITH (
  'connector' = 'ots',
  'endPoint' = '',
  'instanceName' = '',
  'tableName' = '',
  'accessId' = '',
  'accessKey' = ''
);
-- tablestore结果表。
CREATE TEMPORARY TABLE consume_product (
  consume_id STRING,
  product_id STRING,
  price DOUBLE,
  consume_time BIGINT,
  consume_name STRING,
  consume_phone STRING,
  PRIMARY KEY (consume_id,product_id) NOT ENFORCED
) WITH (
  'connector' = 'ots',
  'endPoint' = '',
  'instanceName' = '',
  'tableName' = '',
  'accessId' = '',
  'accessKey' = '',
  'valueColumns' = 'price,consume_time,consume_name,consume_phone'
);
insert into consume_product
select s.consume_id,d.product_ID as product_id,d.price,
      UNIX_TIMESTAMP(s.consume_time,'yy-MM-dd') as consume_time,
      s.consume_name,s.consume_phone
from `consume_record` as s
join `product` for system_time as of proctime() as d
on s.product_id = d.product_ID
```

配置项说明请参见下表。

分类	配置项	示例	描述
	connector	mysql-cdc	MySQL连接器，必须设置为mysql-cdc。

分类	配置项	示例	描述
mysql-cdc源表配置	hostname	myhost	主机名称。
	port	3306	主机端口。
	username	myflinktest	数据库账号。
	password	*****	数据库密码。
	database-name	flinktest	数据库名称。
	table-name	consume_record	数据表名称。
tablestore结果表	connector	ots	表格存储连接器，必须设置为ots。
	endPoint	https://myinstance.cn-hangzhou.ots.aliyuncs.com	实例的服务地址。更多信息，请参见 服务地址 。
	instanceName	myinstance	实例名称。
	tableName	consume_product	数据表名称。
	accessId	*****	阿里云账号或者RAM用户的AccessKey ID和AccessKey Secret。关于获取AccessKey ID和AccessKey Secret的具体操作，请参见 获取AccessKey 。
	accessKey	*****	
	valueColumns	price,consume_time	保存到结果表中的列以列表形式表示，多个列之间用半角逗号(,)分隔。 只有当表格存储作为结果表时才需要配置此项。

ii. 单击验证，进行语法检查。

iii. 验证通过后，单击上线。

4. 启动作业。

i. 在左侧导航栏，单击作业运维。

ii. 单击目标作业名称操作列中的启动。

iii. 在作业启动配置对话框，单击确认启动。

步骤四：在表格存储中分析数据

您可以通过多元索引或者SQL查询功能进行数据分析。

通过多元索引进行数据分析的具体步骤如下：

1. 为数据表consume_product创建多元索引。具体操作，请参见[创建多元索引](#)。
2. 通过多元索引的统计聚合功能实现数据分析。

关于统计聚合功能的更多信息，请参见[统计聚合](#)、

```
public void agg(SyncClient client) {  
    //构建查询语句。  
    SearchRequest searchRequest = SearchRequest.newBuilder()  
        .tableName("consume_product") //设置数据表名称。  
        .indexName("consume_product_index") //设置多元素索引名称。  
        .searchQuery(SearchQuery.newBuilder()  
            .query(QueryBuilders.matchAll())  
            .addGroupBy(GroupByBuilders.groupByField("groupByProductID", "produc  
t_id")).addSubAggregation(  
                AggregationBuilders.sum("sumagg", "price")  
            ))  
        .build()  
    .build();  
    SearchResponse searchResponse = syncClient.search(searchRequest);  
    for(GroupByFieldResultItem item : searchResponse.getGroupByResults().getAsGroupByFi  
eldResult("groupByProductID").getGroupByFieldResultItems()){  
        System.out.println("商品ID:"+item.getKey()+" ,交易总额:"+item.getSubAggregationRes  
ults().getAsSumAggregationResult("sumagg").getValue());  
    }  
}
```

结果示例如下：

```
商品ID:A001,交易总额:20.0  
商品ID:A002,交易总额:40.0  
商品ID:A004,交易总额:20.0  
商品ID:A003,交易总额:5.0  
商品ID:A005,交易总额:15.0  
商品ID:A006,交易总额:5.0  
商品ID:A008,交易总额:5.0
```

通过SQL查询进行数据分析的具体步骤如下：

1. 为数据表consume_product创建映射关系。具体操作，请参见[创建映射关系](#)。

创建映射关系的SQL语句示例如下：

```
CREATE TABLE consume_product (  
    consume_id VARCHAR(1024),  
    product_id VARCHAR(1024),  
    price DOUBLE,  
    consume_time BIGINT,  
    consume_name MEDIUMTEXT,  
    consume_phone MEDIUMTEXT,  
    PRIMARY KEY (consume_id,product_id)  
)
```

2. 执行以下SQL语句进行数据分析。

```
select product_id, sum(price) from consume_product group by product_id
```

结果示例如下图所示。

flink-test

consume_product

consume_id(varchar,PRI)

product_id(varchar,PRI)

consume_name(mediumte

consume_phone(mediumt

consume_time(bigint(20))

price(bigint(20))

执行SQL(F8)

1 select product_id, sum(price) from consume_product group by product_id

执行结果7 Rows (57ms)

product_id	sum(price)
A002	40
A004	20
A006	5
A008	5
A001	20
A003	5
A005	15