11 | 关联图谱分析：如何用 Lambda 架构实现实时的社交网络分析?

今天，我们来讨论实时流计算中第三类非常常见的算法，即关联图谱分析。

关联图谱是一种在许多业务场景下都需要使用的算法，比如社交关系、金融风控等。相比第 10 课时中时间维度聚合值的计算，实时计算关联图谱会复杂很多。这主要是因为“关联图谱”需要使用“图”这种数据结构来表示实体之间的关联关系。

下面是一个“图”数据结构的例子。

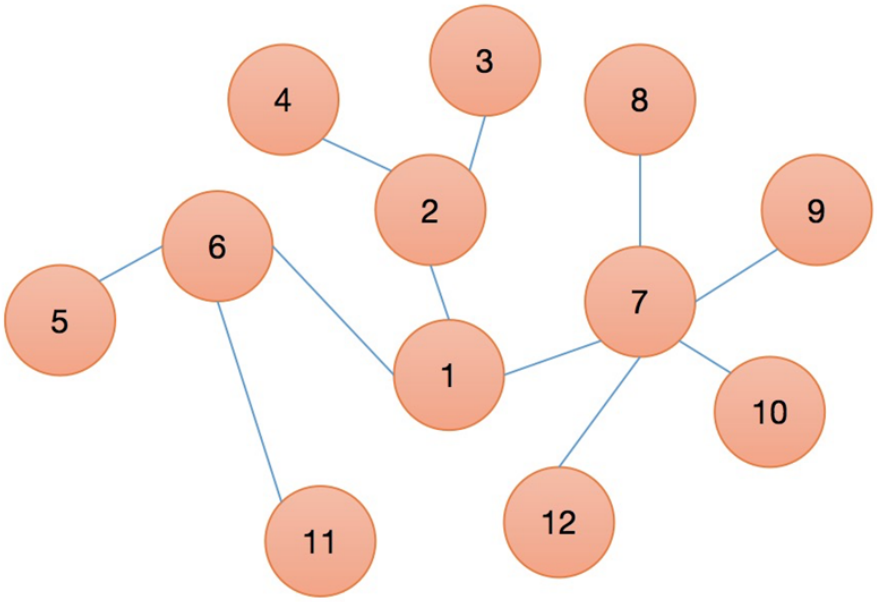

图 1 “图”数据结构示例

在上面这个“图”数据结构中，如果 1 号节点的一度邻居（也就从 1 号节点跳一次能够访问到的节点）有 100 个，而这 100 个一度邻居又各自再有 100 个一度邻居的话，那么从 1 号节点开始，遍历到它所有的二度邻居（也就是那些跳两次能够访问到的节点），不考虑节点重复的话，它将需要访问差不多 100 * 100 也就是 1 万个节点。

而如果更进一步，假设这些二度邻居也各自有 100 个邻居的话，那么从 1 号节点遍历完它的所有三度邻居（也就是那些跳三次能够访问到的节点）的话，就差不多要访问 100 * 100 * 100 也就是 100 万个节点了。

从上面的过程就可以看到，要遍历一个“图”数据结构的话，其节点数随着跳转的次数，是近乎成指数级增长的。特别是**当数据量很大时，“图”数据结构的遍历，尤其是二度邻居及更远邻居的遍历，都将是非常耗时的计算过程**。

正是由于“图”这种数据结构很复杂，而关联图谱又必须用到这种数据结构，这是不是就意味着，我们完全不能够做到实时的关联图谱分析了呢？不是的！我们还是有一些方法可循的。那具体怎么做？这就是今天接下来要重点讲解的内容了。

关联图谱分析

“关联图谱”可以认为是一种在“空间”维度上，对流数据进行的聚合分析。它是对第 10 课时中“时间”维度聚合分析的补充。

比如在风控场景中，我们经常需要计算一些诸如“用户账户使用 IP 的个数”“同一手机号码出现在不同城市的个数”“同一设备上关联用户的数目”“同一推荐人推荐的用户数”“在同一个设备上注册的用户登录过的设备数”“来自同一个 IP 的设备使用过的 IP 数”等，诸如此类的业务指标。

拿其中的“同一设备上关联用户的数目”来说，如果某个设备上注册的用户很多，是不是就意味着它的风险比较高？毕竟正常情况下，我们都只会用自己的手机注册自己的账号，而不会帮其他几十、上百人注册账号。

再比如在社交网络场景下，通过对社交网络分析，可以发现虚拟社区，评估个体影响力，探索信息传播规律等。

下面的图 2 就是一个社交网络关联图谱的例子，我们能够一目了然地发现该网络中有三个“团伙”，每个“团伙”各有 1 到 2 个“大哥”，并且三个“团伙”之间还通过“小弟”相互联系。

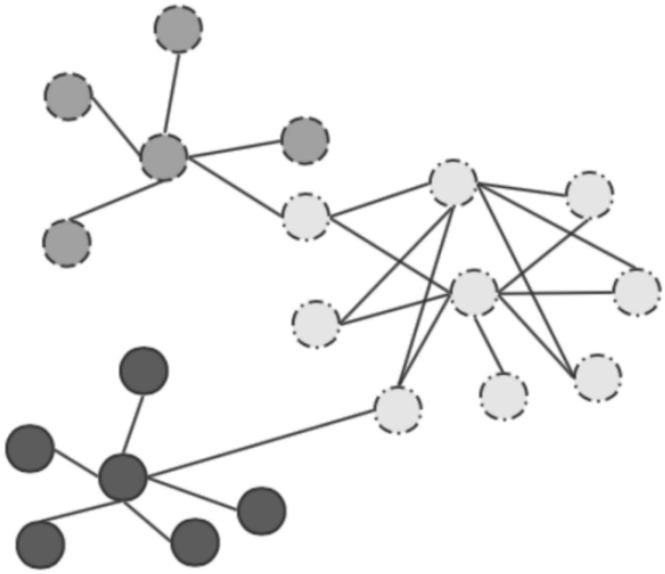

图 2 社交网络关联图谱

从上面的这两个例子可以看出，使用关联图谱计算的业务指标还是挺多样的，比如“用户账户使用 IP 的个数”“同一手机号码出现在不同城市的个数”“在同一个设备上注册的用户登录过的设备数”等，并且这些业务指标在文字描述上看着区别还很大。

那是不是每次我们都需要针对不同的业务指标，设计不同的算法呢？不是的。

经过仔细分析，我们可以发现这些业务指标虽然各不相同，但归纳起来，可以分为两类。我将其中一类称之为“一度关联”指标，而将另一类则称之为“二度关联”指标。

那具体这两类指标分别是什么？以及各自该怎么计算？这就是接下来要仔细讲解的内容了！

一度关联

首先来看“一度关联”指标。“一度关联”是指关联图谱中的一个节点有多少个与之直接相邻的节点。

我们在实时流上计算“一度关联”指标，**通常是为了统计一段时间内，某种属性上另一种属性不同取值的个数。**比如“过去一周内在同一个设备上注册的不同用户数”“过去一小时用户账户使用的不同 IP 数”“过去 3 个月同一手机号码关联的不同设备数”等。

如果用 SQL 来描述这类问题，就应该是类似于以下这些例子：

```
# 过去一周内在同一个设备上注册的不同用户数
SELECT COUNT(DISTINCT user_id) FROM stream
WHERE event_type = "create_account"
AND timestamp >= 1530547200000 and timestamp < 1531152000000
GROUP BY device_id;
# 过去一小时用户账户使用的不同IP数
SELECT COUNT(DISTINCT ip) FROM stream
WHERE event_type = "transaction"
AND timestamp >= 1531065600000 and timestamp < 1531069200000
GROUP BY user_id;
# 过去3个月同一手机号码关联的不同设备数
SELECT COUNT(DISTINCT device_id) FROM stream
WHERE event_type = "create_account"
AND timestamp >= 1530547200000 and timestamp < 1538496000000
GROUP BY phone_number;
```

从上面的 SQL 可以看出，一度关联的计算说白了就是 COUNT DISTINCT（去重计数）计算。所以，我们立刻就可以想到一种实现“一度关联”计算的方法。那就是，用一个集合（set）来记录变量所有不同的取值。利用集合自身的去重特性，每次只需要将变量的值添加到集合中，然后计算集合的大小，就可以得到所要求的“一度关联”指标了。

但是，这种方法在变量的不同取值非常多时，就不切实际了。比如全球 IP 地址有 40 多亿个，如果要保存所有这些 IP 地址，将会占用大量的存储空间。当数据量大到一定程度时，不仅性能会急剧下降，甚至内存都不够存储这些不同的值了。

所以，除非你能够确定变量的不同取值不会太多。否则，不要轻易地使用这种简单粗暴的方法。

既然如此，那该怎么办呢？在这里，我想给你介绍一种神奇的算法。不知道你是否听说过 HyperLogLog 算法？HyperLogLog 是一种与 Bloom Filter 类似的算法，都是用准确度来换取时间和空间的估计算法。HyperLogLog 能够帮助我们节省大量存储空间和计算时间。

并且非常幸运的是，我们在流计算中经常使用的内存数据库 Redis，已经颇有先见之明地为我们提供了 HyperLogLog 算法。以 Redis 中的 HyperLogLog 算法实现为例，只需要用 12K 字节的内存，就能够在 0.81% 的标准误差范围内，记录将近 2^{64} 个不同值的个数。

想象下，如果我们是将这些不同值都原原本本地记录下来，是不是几 T 的内存都不够用了。所以，之前那种用集合记录不同值的方式是万万不可取的。这正是我们需要使用 HyperLogLog 算法的重要原因。

另外 HyperLogLog 算法的插入和查询的时间复杂度都是 O(1)，所以在时间方面 HyperLogLog 算法也能够完全满足实时计算的要求。

下面我们就借助于 Redis 来讲解下 HyperLogLog 算法的使用方法。

在 Redis 中， HyperLogLog 算法提供了三个命令： PFADD 、PFCOUNT 和 PFMERGE 。

- PFADD 用于将元素添加到 HyperLogLog 寄存器；
- PFCOUNT 用于返回添加到 HyperLogLog 寄存器中不同元素的个数（是一个估计值）；
- PFMERGE 则用于合并多个 HyperLogLog 寄存器。

在有了 HyperLogLog 算法的加持后，我们就能够对一度关联的计算做出优化了。

首先，为变量创建一个 HyperLogLog 寄存器。然后通过 PFADD 命令将每次新到的数据，添加到 HyperLogLog 寄存器中。最后，通过 PFCOUNT 命令就可以返回变量不同取值的个数了，这就是“一度关联”值。

如果我们还需要对多个时间窗口内的不同值个数汇总，那么就使用 PFMERGE 命令先将多个窗口内的 HyperLogLog 寄存器合并起来，生成一个新的合并后的 HyperLogLog 寄存器，之后对这个寄存器使用 PFCOUNT 命令，就可以返回合并多个时间窗口后变量的不同取值个数了。

你看，上面使用 HyperLogLog 的方法是不是很方便地解决了“一度关联”值实时计算的问题？不过，我这里还需要稍微补充下。

HyperLogLog 算法计算出的值是一个估计值，并且这个值的精度与寄存器的长度有关。比如寄存器长度为 12K 字节时，估计误差为 0.81%；当寄存器长度是 256 字节时，估计误差为5.63%；当寄存器长度是 128 字节时，估计误差为 7.96%。

所以，你可以根据自己的精度需要，来调整寄存器的长度。毕竟寄存器越短，占用的空间也就越少了。

二度关联

讨论完了“一度关联”，接下来我们再来讨论“二度关联”。

“二度关联”是对“一度关联”的扩展，它是由节点的一度关联节点再做一次一度关联之后的节点数。比如“过去 30 天在同一个设备上注册的用户登录过的设备数”“过去一个周内来自同一个 IP 的设备使用过的 IP 数”。

下图 3 描述了一个节点的二度关联节点，其中所有标记为 1 的节点都是标记为 0 的节点的一度关联节点，而所有标记为 2 的节点都是标记为 0 的节点的二度关联节点。

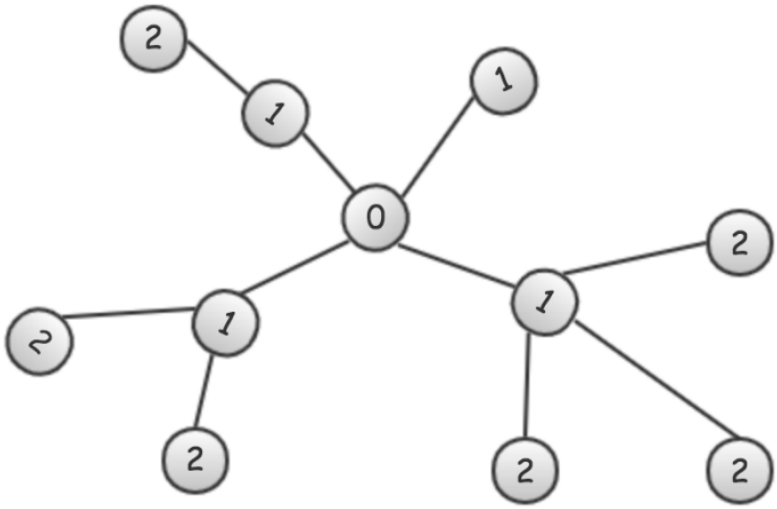

图 3 二度关联

从图 3 能够直观地看到，要计算一个节点的二度关联节点数，必须要先遍历一遍该节点的一度关联节点。

在之前，我们计算“一度关联”值的时候，是通过 HyperLogLog 算法，避免了记录所有一度关联节点的问题。但这次，我们再也不能避开这个问题了。

在实时流计算领域，目前尚且没有一种在大数据量情况下，方便、直接且行之有效的二度关联计算方案。虽然有很多图数据库如 JanusGraph 和 Dgraph 在分布式实时图计算方面已经有了非常大的突破，能够在一定程度上解决二度关联实时计算的问题，但相比实时流计算对响应时延以及吞吐量更严苛的要求，还是略显不足。

所以我们这次完全没辙了吗？这也未必。**如果我们能够接受一个稍有滞后的二度关联计算结果**，还是能够采取一定的手段，做到**二度关联的实时查询（注意只是查询部分是实时的，计算部分还是稍微有所滞后）**的。而这种手段，就是大名鼎鼎的 Lambda 架构！

Lambda 架构

Lambda 架构是一种实时大数据处理框架。它的核心思想是，对于计算量过大或者计算过于复杂的问题，将其分为批处理层（Batch Layer）和快速处理层（Speed Layer），其中批处理层是在主数据集上的全量计算，而快速处理层则是对增量数据的计算。当这两者各自计算出结果后，再将结果合并起来，就可以得到最终的查询结果了。

通过这种**批处理层**和**快速处理层**相结合的方式，Lambda 架构能够实时地在全量数据集上进行分析和查询。

下面，我们就以“过去 30 天在同一个设备上注册的用户登录过的设备数”这个计算目标，详细讲解具体实现方法。这里，我借用了 Hive SQL 来讲解具体如何实现。如果你以前没接触过 Hive 的话也没有关系，因为我们这里的讲解只会使用到 SQL，所以也很容易理解。

接下来，就来看看具体怎么做吧！

首先，我们定义下后面计算时会用到的数据库表结构。具体如下：

```
CREATE TABLE create_account_table(device_id string, user_id string) PARTITIONED BY (day string, h
CREATE TABLE login_table(user_id string, device_id string) PARTITIONED BY (day string, hour strin
```

在上面的 SQL 语句中，我们定义了两个表。其中，create_account_table 表用于保存注册（create_account）事件，而 login_table 表则用于保存登录（login）事件。

接下来，就是具体该怎么划分 Lambda 架构中的**批处理层**和**快速处理层**，以及怎样合并两者的计算结果了？这里，我借助于图 4 详细说明下。

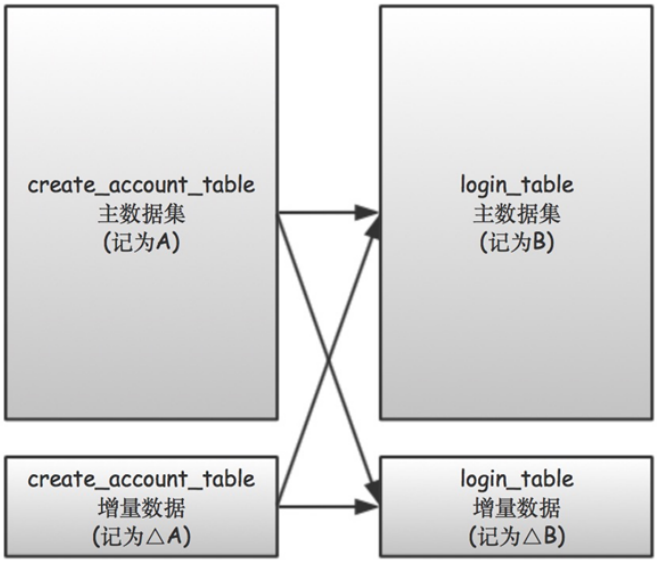

图 4 二度关联指标的增量计算方法

前面讲过，我们要计算的二度关联指标是“过去 30 天在同一个设备上注册的用户登录过的设备数”。这里，为了方便划分“主数据集”和“增量数据”，我以“批处理层”作业开始执行时的整点时刻为“时间分割点”，并将这个“时间分割点”记录为 T。比如，如果“批处理层”作业是在 2020/11/30 09:03:00 开始执行，那么时间分割点 T 就是 2020/11/30 09:00:00。

定义好时间分割点 T 之后，我们就可以划分“主数据集”和“增量数据”了。在上面的图 4 中，A代表了“注册事件的主数据集”，也就是“过去 29 天以及当天时间在 T 之前的注册事件集合”，B代表了“登录事件的主数据集”，也就是“过去 29 天以及当天时间在 T 之前的登录事件集合”，ΔA代表了“注册事件的增量数据”，也就是“当天时间在 T 之后的注册事件集合”，ΔB代表“登录事件的增量数据”，也就是“当天时间在 T 之后的登录事件集合”。

划分好“主数据集”和“增量数据”之后，接下来就是确定**批处理层**和**快速处理层**各自的计算内容了。在图 4 中，我使用有向线段→来表示数据之间的内联接（inner join）操作。由于“批处理层”应该计算的是“主数据集”，所以 A→ B代表的是批处理层需要计算的内容。另外，由于“快速处理层”应该计算的是“增量数据”，所以剩下的ΔA→ ΔB、ΔA→ B、A→ ΔB则代表了快速处理层需要计算的内容。

所以，最后**批处理层**和**快速处理层**合并的结果，就应该是将A→ B、ΔA→ ΔB、ΔA→ B、A→ ΔB这四部分的结果全部合并起来，也就是 SQL 中的 UNION 操作。

至此，我们就理清楚采用 Lambda 架构，计算“过去 30 天在同一个设备上注册的用户登录过的设备数”这个二度关联指标的完整计算思路了。

所以，接下来我们就按照上面的思路，来具体实现每一个步骤。

批处理层

我们先来看批处理层的计算，也就是图 4 中的A→ B这一部分。

这里，为了更加真实地反映出实际开发过程中，有关对时间和作业（job）调度的考量因素，我们假定A→ B的计算需要 120 分钟。换言之，也就是关联（inner join）计算“过去 29 天以及当天时间在 T 之前的注册事件”和“过去 29 天以及当天时间在 T 之前的登录事件”，需要 120 分钟。

> 请稍微记住下这里的 120 分钟，因为后面我们对每个步骤所需时间的估计，以及对作业调度的安排，都是基于这里的 120 分钟。

由于批处理层，也就是A→ B的计算需要 120 分钟，再考虑到实际安排作业调度时，为了避免前一个作业还没有结束，另一个作业就已经开始执行的情况，还需要添加一点时间间隔。

这里，我们添加的时间间隔是 60 分钟。于是，加上执行批处理层作业A→ B的 120 分钟，我们需要**设定每 180 分钟执行一次批处理层的计算**。比如在 2020/11/30 09:03:00 时刻，开始执行如下批处理层的 Hive SQL。

```
-- 每180分钟执行一次
CREATE TABLE temp_table_before_20201130_09 AS
SELECT DISTINCT
    create_account_table.device_id AS c_device_id,
    create_account_table.user_id AS user_id,
    login_table.device_id AS l_device_id
FROM
    create_account_table INNER JOIN login_table ON create_account_table.user_id = login_table.use
WHERE
    (
        create_account_table.day < "20201130" AND create_account_table.day >= "20201101"
        OR
        create_account_table.day = "20201130" AND create_account_table.hour < "09"
    )
    AND
    (
        login_table.day < "20201130" AND login_table.day >= "20201101"
        OR
        login_table.day = "20201130" AND login_table.hour < "09"
    );
```

在上面的 Hive SQL 中，我们将 create_account_table 表和 login_table 表通过共同的用户 user_id 关联起来，并通过 DISTINCT 关键字得到了去重后的用户注册和登录设备信息。这样就得到了批处理层的计算结果。

快速处理层

算完批处理层后，接下来就是“快速处理层”了。

前面我们已经说过，“快速处理层”计算的是“增量数据”，它包含了ΔA→ ΔB、ΔA→ B、A→ ΔB共三个部分。

由于我们在前面计算批处理层时A→ B，设定的是每隔 180 分钟就计算一次，并且A→ B的计算是需要 120 分钟。为了估计下ΔA→ ΔB、ΔA→ B、A→ ΔB这三个增量计算各自的时间，我们借助下面图 5 所示的批处理作业和快速处理作业调度时间来讲解下。

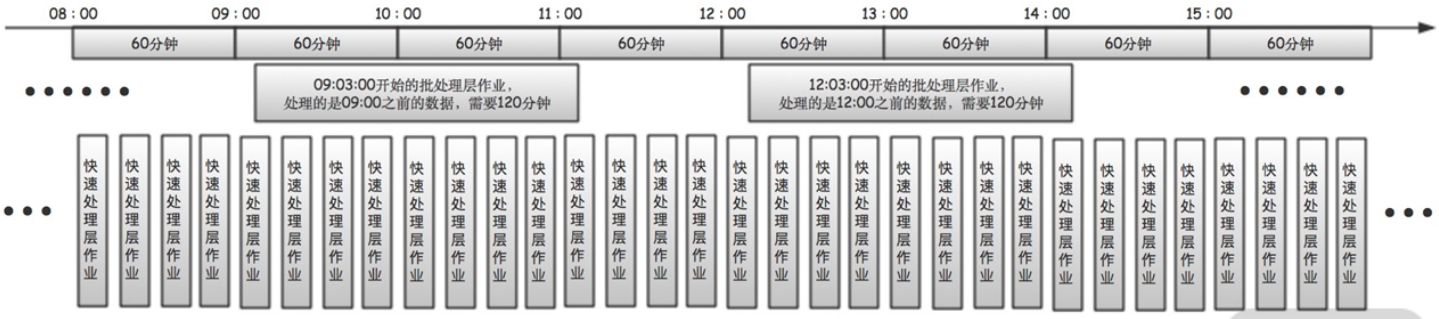

图 5 批处理作业和增量计算作业调度时间

在上面的图 5 中，可以看到两次“批处理层”作业之间的时间间隔为 180 分钟，而且每次“批处理层”作业处理的其实是作业开始前的数据，而不是两次“批处理层”作业之间 180 分钟的数据，所以每次“快速处理层”的作业，最多就需要计算最近 180 + 180 = 360 分钟的增量数据。我们就以这个最长的 360 分钟计算，这意味着**增量数据ΔA和ΔB各自包含了 360 分钟的数据**。而前面我们说过，A 代表的是“过去 29 天以及当天时间在 T 之前的注册事件集合”，B 代表的是“过去 29 天以及当天时间在 T 之前的登录事件集合”。

这意味着**主数据集 A 和 B 各自包含了大约**29 * 24 * 60 = 41760**分钟的数据**。现在A→ B的计算需要 120 分钟，所以**按照数据量的大小做个简单的比例计算**，ΔA→ B和A→ ΔB各自的计算时间就是 360 / 41760 * 120≈1 分钟左右，同理ΔA→ ΔB的计算时间就是 360 / 41760 * 1≈0.008 分钟左右。汇总下来，ΔA→ ΔB、ΔA→ B、A→ ΔB这三个增量计算的总时间就是 1 + 1 + 0.008≈2 分钟左右的时间了。换言之就是，“快速处理层”的计算时间在 2 分钟左右。

算到这里就有些尴尬了，“快速处理层”居然需要 2 分钟左右，那这还算是“实时计算”吗？

如果是那种从新数据到达系统后，必须在秒、毫秒甚至微秒级别的时间内，就要求新数据的效果反映在计算结果中的业务场景来说，这个 2 分钟显然是时间有些长了。

但是，请不要忘了我们做“实时计算”的初衷是什么？我们做“实时计算”，其实最初的出发点是为了**挖掘数据的实时价值**。而在上面使用 Lambda 架构思想改造后的计算过程，我们将全量计算的时间**从原来的 120 分钟，缩短为现在的 2 分钟左右**，这中间是 60 倍的性能提升！所以，这种尽可能向“实时计算”靠近的努力工作，还是非常有意义的！

所以，接下来就可以继续实现“快速处理层”了，具体代码如下：

```
-- 计算△A->△B部分
CREATE TABLE temp_table_after_20201130_09_p1 AS
SELECT DISTINCT
    create_account_table.device_id AS c_device_id,
    create_account_table.user_id AS user_id,
    login_table.device_id AS l_device_id
FROM
    create_account_table INNER JOIN login_table ON create_account_table.user_id = login_table.use
WHERE
    (
        create_account_table.day = "20201130" AND create_account_table.hour >= "09"
        AND
        login_table.day = "20201130" AND login_table.hour >= "09"
    );
-- 计算A->△B部分
CREATE TABLE temp_table_after_20201130_09_p2 AS
SELECT DISTINCT
    create_account_table.device_id AS c_device_id,
    create_account_table.user_id AS user_id,
    login_table.device_id AS l_device_id
FROM
    create_account_table INNER JOIN login_table ON create_account_table.user_id = login_table.use
WHERE
    (
        create_account_table.day < "20201130" AND create_account_table.day >= "20201101"
        AND
        login_table.day = "20201130" AND login_table.hour >= "09"
    );
-- 计算△A->B部分
CREATE TABLE temp_table_after_20201130_09_p3 AS
SELECT DISTINCT
    create_account_table.device_id AS c_device_id,
    create_account_table.user_id AS user_id,
    login_table.device_id AS l_device_id
FROM
    create_account_table INNER JOIN login_table ON create_account_table.user_id = login_table.use
WHERE
    (
        create_account_table.day = "20201130" AND create_account_table.hour >= "09"
        AND
        login_table.day < "20201130" AND login_table.day >= "20201101"
    );
```

在上面的 SQL 中，我们分别计算了ΔA→ ΔB、ΔA→ B、A→ ΔB的增量数据。根据前面的分析，这部分执行需要 2 分钟左右，但是可以连续不断执行。

现在，“批处理层”和“快速处理层”都已经算完了，接下来就是将两者的结果合并起来了。具体代码如下：

```
SELECT c_device_id, COUNT(DISTINCT l_device_id)
FROM
    temp_table_before_20201130_09
    UNION temp_table_after_20201130_09_p1
    UNION temp_table_after_20201130_09_p2
    UNION temp_table_after_20201130_09_p3
GROUP BY c_device_id;
```

在上面的 SQL 代码中，我们只需要将分别对应A→ B、ΔA→ ΔB、ΔA→ B、A→ ΔB四部分计算结果的临时表，用 UNION 操作合并起来即可。

至此，我们就完成了“过去 30 天在同一个设备上注册的用户登录过的设备数”这个二度关联指标的计算。

将结果保存到 Redis

不过，虽然完成了二度关联指标的计算，但是我们还需要做最后一个步骤。也就是将计算结果导入 Redis 缓存起来，以供**业务应用实时查询**使用。

这个步骤虽然很简单，但它是我们做实时计算非常必要的。因为，我们之前用 Hive SQL 计算的结果是放在 HDFS（Hadoop 分布式文件系统）上的，而 HDFS 并不能达到实时查询的性能要求。所以，我们必须将结果从 HDFS 中加载到 Redis 里。

当二度关联的计算结果存入 Redis 之后，后续的访问就可以非常快速了。比如对于“过去 30 天在同一个设备上注册的用户登录过的设备数”，只需给定设备id，用 GET 指令就可以从 Redis 中，快速查询到过去 30 天在该设备 id 上注册的那些用户，所登录过的不同设备数了。

至此，我们就用 Lambda 架构实现了计算“过去 30 天在同一个设备上注册的用户登录过的设备数”这个二度关联指标的功能。

总的来说，在这种方案下，当**业务应用查询**“过去 30 天在同一个设备上注册的用户登录过的设备数”时，**它只需要访问 Redis 即可，因此响应速度非常快，能够满足业务应用实时查询并返回结果的要求**。只是说，由于“快速处理层”需要将近 2 分钟的计算时间，所以从 Redis 查询出来的结果会有一定滞后。这个滞后最多是 2 * 2 = 4 分钟。也就是说，如果现在是 9 点过 5 分，那么你从 Redis 查询出来的数据，最早可能是 9 点过 1 分时由“快速处理层”和“批处理层”计算出的结果。

所以不管怎样，这是一个可以真实落地，并且行之有效的解决方案了。

最后，真心希望诸如 JanusGraph 和 Dgraph 等各种开源分布式图数据库能够变得更加强大和丰富。毕竟，关联图谱分析，本应该是图数据库分内之事啊！

另外，图数据库厂商 TigerGraph 专门针对目前几种主流图数据库做过性能对比测试，建议你查阅一下，可以重点关注其中“Table 8 - 两度路径查询时间”一表，应该会有所收获。

小结

今天，我们详细讨论了关联图谱中的一度关联和二度关联分析问题。

一度关联问题反映的是在实时计算场景下，由于数据量和计算时间的限制，有时候我们不得不采取用精确度来换时间和空间的方法。可以看到，通过牺牲一定的精确度，算法的时间复杂度和空间复杂度都简化到 O(1) 水平，这就保证了算法能够实时返回，并且极大地减少了需要使用的存储空间。当然，如果业务确实是需要严格保证精确度的话，那就不能使用 HyperLogLog 算法了，这个时候就需要像计算二度关联那样，采用 Lambda 架构了。

而在计算二度关联指标时，由于我们实在是无法避免对一度邻居节点的遍历计算，所以只能退而求其次选择了 Lambda 架构。可以看到，在 Lambda 架构中，我们为了尽可能满足业务的“实时性”要求，让系统架构变得复杂了许多。

但是，在实际开发中，我们还是会经常用到 Lambda 架构。这有两方面的原因。一是，问题本身很复杂，很难直接做成实时计算，比如今天讲到的二度关联计算，以及一些复杂的机器学习模型等。二是，因为有时候项目团队成员之间有各自擅长的专业领域和技术方向，这样“批处理层”和“快速处理层”可能会是不同的开发人员使用不同的技术来完成，比如“批处理层”使用了 Spark 机器学习库进行训练，而“快速处理层”使用 Flink jpmml 来进行预测，这种情况在实际工作中也会经常发生，可能就项目进度推进而言会更加高效。

所以，我们还是有必要理解 Lambda 架构的。在后面的课时中，我还会专门针对 Lambda 架构进行讨论。

最后留一个小作业，你在工作中有用过图数据库吗？是怎么用的呢？能否做到实时计算？特别是在大数据场景下，你会选择怎样的图数据库呢？可以将你的想法或问题写在留言区。

下面是本课时的知识脑图，可以帮助你理解本课时的内容。