深度长文探讨 JOIN 运算的简化和提速

连接运算（JOIN）一直是 SQL 中的老大难问题。在关联表稍多一点的时候，代码书写就变得很容易出错了。而且因为 JOIN 语句的复杂，导致关联查询也一向是 BI 软件的软肋，几乎没有 BI 软件能让业务用户顺畅地完成多表关联查询。对于性能优化也是，关联表较多或者数据量大时，JOIN 的性能也很难得到提升。

本文将对 JOIN 运算进行深入讨论，针对性地提出语法简化和性能优化的方法。

一. SQL 中的 JOIN

我们先来看 SQL 是如何理解 JOIN 运算的。

SQL 对 JOIN 的定义非常简单，就是两个集合（表）做笛卡尔积后再按某种条件过滤，写出来的语法就是 *A* JOIN *B* ON ...。理论上讲，笛卡尔积的结果集应该是以两个集合成员构成的二元组作为成员，不过由于 SQL 中的集合也就是表，其成员总是有字段的记录，而且也不支持泛型数据类型来描述成员为记录的二元组,所以就简单地把结果集处理成两表记录的字段合并后构成的新记录的集合。这也是 JOIN 一词在英语中的原意（即把两个记录的字段连接起来），并没有乘法（笛卡尔积）的意思。不过，把笛卡尔积成员理解成二元组还是合并字段的记录，并不影响我们后续的讨论。

JOIN 的定义中并没有约定过滤条件的形式，理论上，只要结果集是两个源集合笛卡尔积的子集，都是合理的 JOIN 运算。比如假设集合 A={1,2},B={1,2,3}，A JOIN B ON A<B 的结果就是{(1,2),(1,3),(2,3)}；A JOIN B ON A=B 的结果是{(1,1),(2,2)}。我们把过滤条件为等式的称为**等值 JOIN**，而不是等值连接的情况则称为**非等值 JOIN**。这两个例子中，前者是非等值 JOIN，后者是等值 JOIN。

对于数据表之间的等值 JOIN,条件可能由多个有 AND 关系的等式构成,语法形式 *A* JOIN *B* ON $A.a_i=B.b_i$ AND ...，其中 a_i 和 b_i 分别是 A 和 B 的字段。有经验的程序员都知道，现实中绝大多数 JOIN 都是等值 JOIN，非等值 JOIN 要少见得多，而且大多数情况都可以转换成等值 JOIN 来处理，所以我们在这里重点讨论等值 JOIN，并且后续讨论中也主要使用表和记录而不是集合和成员来举例。

根据对空值的处理规则，严格的等值 JOIN 又称为 **INNER JOIN**，还可以再衍生出 **LEFT JOIN** 和 **FULL JOIN**，共有三种情况（RIGHT JOIN 可以理解为 LEFT JOIN 的反向关联，不再单独作为一种类型）。谈论 JOIN 时一般还会根据两个表中关联记录（也就是满足过滤条件的二元组）的数量分为**一对一**、**一对多**、**多对一**以及**多对多**这几种情况，这些常规术语在 SQL 和数据库资料中都有介绍，这里就不再赘述了。

我们再来看 JOIN 的实现。

最容易想到的简单办法就是按照定义做硬遍历，不区分等值 JOIN 和非等值 JOIN。设表 *A* 有 *n* 条记录，*B* 有 *m* 条记录，要计算 *A* JOIN *B* ON $A.a=B.b$ 时，硬遍历的复杂度会是 $n*m$，即要进行 $n*m$ 次过滤条件的计算。

显然这种算法会比较慢。不过，支持多数据源的报表工具中有时就是用这种慢办法实现关联的，因为在报表中数据集的关联关系（也就是 JOIN 中的过滤条件）会拆散定义在单元

格的运算式中，已经看不出是多个数据集之间的 JOIN 运算，也就只能用遍历方法去计算这些关联表达式了。

成熟的数据库当然不会这么笨了，对于等值 JOIN，数据库一般会采用 HASH JOIN 算法。即将关联表的记录按其**关联键**（过滤条件中对应相等的字段，即 *A.a* 和 *B.b*）的 HASH 值分成若干组，将相同 HASH 值的记录分到一组。如 HASH 值范围是 1...*k*，则将 *A* 和 *B* 表都分成 *k* 个子集 $A_1,...,A_k$ 和 $B_1,...,B_k$。A_i 中记录的关联键 *a* 的 HASH 值是 *i*，B_i 中记录的关联键 *b* 的 HASH 值也是 *i*，然后，只要分别在 A_i 和 B_i 之间做遍历连接就可以了。因为 HASH 不同时字段值也必然不同，*i*!=*j* 时，A_i 中记录不可能和 B_j 中记录发生关联。如果 A_i 的记录数是 n_i，B_i 的记录数是 m_i，则过滤条件的计算次数为 SUM(n_i*m_i)，最平均的情况时，n_i=n/k，m_i=m/k，则总的复杂度只有原始硬遍历手段的 1/*k*，能有效地提高运算性能！

所以，多数据源关联报表要提速的话，也需要在数据准备阶段做好关联，否则数据量稍大时性能就会急剧下降。

不过，HASH 函数并不总能保证平均分拆，在运气不好的时候可能会发生某一组特别大的情况，那样性能提升效果就会差很多。而且还不能使用太复杂的 HASH 函数，否则计算 HASH 的时间又变多了。

当数据量大到超过内存时，数据库会使用 HASH 分堆的方法，算是 HASH JOIN 算法的推广。遍历 *A* 表和 *B* 表，将记录按关联键的 HASH 值拆分成若干小子集缓存到外存中，称为**分堆**。然后再在对应的堆之间做内存 JOIN 运算。同样的道理，HASH 值不同时键值也必然不同，关联一定发生在对应的堆之间。这样就把大数据的 JOIN 转换成若干小数据的 JOIN 了。

但是类似地，HASH 函数存在运气问题，有可能会发生某个分堆还特别大而无法装入内存，这时候就可能要进行二次 HASH 分堆，即换一个 HASH 函数对这组太大的分堆再做一次 HASH 分堆算法。所以，外存 JOIN 运算有可能出现多次缓存的现象，其运算性能有一定的不可控性。

分布式系统下做 JOIN 也是类似的，根据关联键的 HASH 值将记录分发到各个节点机上，称为 **Shuffle 动作**，然后再分别做单机的 JOIN。当节点比较多的时候，造成的网络传输量带来的延迟会抵消多机分摊任务得到的好处，所以分布式数据库系统通常有个节点数的极限，达到极限后，更多的节点并不能获得更好的性能。

二. 等值 JOIN 的剖析

我们来考察下面三种等值 JOIN：

1. 外键关联

表 *A* 的某个字段和表 *B* 的主键字段关联（所谓字段关联，就是前一节说过的在等值 JOIN 的过滤条件中要对应相等的字段）。*A* 表称为**事实表**，*B* 表称为**维表**。*A* 表中与 *B* 表主键关联的字段称为 *A* 指向 *B* 的**外键**，*B* 也称为 *A* 的外键表。

这里说的主键是指逻辑上的主键，也就是在表中取值唯一、可以用于唯一某条记录的字段（组），不一定在数据库表上建立过主键。

外键表是多对一的关系，且只有 JOIN 和 LEFT JOIN，而 FULL JOIN 非常罕见。

典型例子：商品交易表和商品信息表。

显然，外键关联是不对称的。事实表和维表的位置不能互换。

2. **同维表**

表 *A* 的主键与表 *B* 的主键关联，*A* 和 *B* 互称为**同维表**。同维表是一对一的关系，JOIN、LEFT JOIN 和 FULL JOIN 的情况都会有，不过在大多数数据结构设计方案中，FULL JOIN 也相对少见。

典型例子：员工表和经理表。

同维表之间是对称的，两个表的地位相同。同维表还构成是等价关系，*A* 和 *B* 是同维表，*B* 和 *C* 是同维表，则 *A* 和 *C* 也是同维表。

3. **主子表**

表 *A* 的主键与表 *B* 的部分主键关联，*A* 称为**主表**，*B* 称为**子表**。主子表是一对多的关系，只有 JOIN 和 LEFT JOIN，不会有 FULL JOIN。

典型例子：订单和订单明细。

主子表也是不对称的，有明确的方向。

在 SQL 的概念体系中并不区分外键表和主子表，多对一和一对多从 SQL 的观点看来只是关联方向不同，本质上是一回事。确实，订单也可以理解成订单明细的外键表。但是，我们在这里要把它们区分开，将来在简化语法和性能优化时将使用不同的手段。

我们说，这三种 JOIN 已经涵盖了绝大多数等值 JOIN 的情况，甚至可以说几乎全部有业务意义的等值 JOIN 都属于这三类，把等值 JOIN 限定在这三种情况之中，几乎不会减少其适应范围。

仔细考察这三种 JOIN，我们发现所有关联都涉及主键，没有多对多的情况，是不是可以不考虑这种情况？

是的！多对多的等值 JOIN 几乎没有业务意义。

如果两个表 JOIN 时的关联字段没有涉及到任何主键，那就会发生多对多的情况，而这种情况几乎一定还会有一个规模更大的表把这两个表作为维表关联起来。比如学生表和科目表在 JOIN 时，会有个成绩表把学生表和科目表作为维表，单纯只有学生表和科目表的 JOIN 没有业务意义。

当写 SQL 语句时发现多对多的情况，那大概率是这个语句写错了！或者数据有问题！这条法则用于排除 JOIN 错误很有效。

不过，我们一直在说“几乎”，并没有用完全肯定的说法，也就是说，多对多在非常罕见的情况下也会业务意义。可举一例，用 SQL 实现矩阵乘法时会发生多对多的等值 JOIN，具体写法读者可以自行补充。

笛卡尔积再过滤这种 JOIN 定义，确实非常简单，而简单的内涵将得到更大的外延，可以把多对多等值 JOIN 甚至非等值 JOIN 等都包括进来。但是，过于简单的内涵无法充分体现出最常见等值 JOIN 的运算特征。这会导致编写代码和实现运算时就不能利用这些特征，在运算较为复杂时（涉及关联表较多以及有嵌套的情况），无论是书写还是优化都非常困难。而充分利用这些特征后，我们就能创造出更简单的书写形式并获得更高效的运算性能，后面的内容中将会逐步加以说明。

与其为了把罕见情况也被包括进来而把运算定义为更通用的形式，还不如把这些情况定义成另一种运算更为合理。

三. JOIN 的语法简化

我们先看如何利用关联都涉及主键这个特征来简化 JOIN 的代码书写，分别讨论这三种情况。

1. 外键属性化

先看个例子，设有如下两个表：

employee 员工表
- id 员工编号
- name 姓名
- nationality 国籍
- department 所属部门

department 部门表
- id 部门编号
- name 部门名称
- manager 部门经理

employee 表和 department 表的主键都是其中的 id 字段，employee 表的 department 字段是指向 department 表的外键，department 表的 manager 字段又是指向 employee 表的外键（因为经理也是个员工）。这是很常规的表结构设计。

现在我们想问一下：哪些美国籍员工有一个中国籍经理？

用 SQL 写出来是个三表 JOIN 的语句：

```
SELECT A.*
FROM employee A
JOIN department B ON A.department=B.id
JOIN employee C ON B.manager=C.id
WHERE A.nationality='USA' AND C.nationality='CHN'
```

首先要 FROM employee 用于获取员工信息，然后这个 employee 表要和 department 做 JOIN 获取员工的部门信息,接着这个 department 表还要再和 employee 表 JOIN 要获取经理的信息，这样 employee 表需要两次参与 JOIN，在 SQL 语句中要为它起个别名加以区分，整个句子就显得比较复杂难懂。

如果我们把外键字段直接理解成它关联的维表记录，就可以换一种写法：

```
SELECT * FROM employee
WHERE nationality='USA' AND department.manager.nationality='CHN'
```

当然，这不是标准的 SQL 语句了。

第二个句子中粗体部分表示当前员工的“所属部门的经理的国籍”。我们把外键字段理解成维表的记录后，维表的字段被理解为外键的属性，department.manager 即是“所属部门的经理”，而这个字段在 department 中仍然是个外键，那么它对应的维表记录字段可以继续理解为它的属性，也就会有 department.manager.nationality，即“所属部门的经理的国籍”。

这种对象式的理解方式称为**外键属性化**,显然比笛卡尔积过滤的理解方式要自然直观得多。外键表 JOIN 时并不会涉及到两个表的乘法，外键字段只是用于找到维键表中对应的那条记录，完全不会涉及到笛卡尔积这种有乘法特性的运算。

我们前面约定，外键关联时时维表中关联键必须是主键，这样，事实表中每一条记录的外键字段关联的维表记录就是唯一的，也就是说 employee 表中每一条记录的 department 字段唯一关联一条 department 表中的记录，而 department 表中每一条记录的 manager 字段也唯一关联一条 employee 表中的记录。这就保证了对于 employee 表中的每一条记录，department.manager.nationality 都有唯一的取值，可以被明确定义。

但是，SQL 对 JOIN 的定义中并没有主键的约定，如果基于 SQL 的规则，就不能认定与事实表中外键关联的维表记录有唯一性，有可能发生与多条记录关联，对于 employee 表的记录来讲，department.manager.nationality 没有明确定义，就不能使用了。

事实上，这种对象式写法在高级语言（如 C，Java）中很常见，在这类语言中，数据就是按对象方式存储的。employee 表中的 department 字段取值根本就是一个对象，而不是编号。其实许多表的主键取值本身并没有业务意义，仅仅是为了区分记录，而外键字段也仅仅是为了找到维表中的相应记录，如果外键字段直接是对象，就不需要再通过编号来标识了。不过，SQL 不能支持这种存储机制，还要借助编号。

我们说过外键关联是不对称的，即事实表和维表是不对等的，只能基于事实表去找维表字段，而不会有倒过来的情况。

2. 同维表等同化

同维表的情况相对简单，还是从例子开始，设有两个表：

employee 员工表
- id 员工编号
- name 姓名
- salary 工资
- ...

manager 经理表
- id 员工编号
- allowance 岗位津贴
- ....

两个表的主键都是 id，经理也是员工，两表共用同样的员工编号，经理会比普通员工多一些属性，另用一个经理表来保存。

现在我们要统计所有员工（包括经理）的总收入（加上津贴）。

用 SQL 写出来还是会用到 JOIN：

```
SELECT employee.id, employee.name, employy.salary+manager.allowance
FROM employee
LEFT JOIN manager ON employee.id=manager.id
```

而对于两个一对一的表，我们其实可以简单地把它们看成一个表：

```
SELECT id,name,salary+allowance
FROM employee
```

类似地，根据我们的约定，同维表 JOIN 时两个表都是按主键关联的，相应记录是唯一对应的，salary+allowance 对 employee 表中每条记录都是唯一可计算的，不会出现歧义。这种简化方式称为**同维表等同化**。

同维表之间的关系是对等的，从任何一个表都可以引用到其它同维表的字段。

3. 子表集合化

订单及订单明细是典型的主子表：

Orders 订单表
- id 订单编号
- customer 客户
- date 日期

...

OrderDetail 订单明细

id 订单编号

no 序号

product 订购产品

price 价格

...

Orders 表的主键是 id，OrderDetail 表中的主键是(id,no)，前者的主键是后者的一部分。

现在我们想计算每张订单的总金额。

用 SQL 写出来会是这样：

```
SELECT Orders.id, Orders.customer, SUM(OrderDetail.price)
FROM Orders
JOIN OrderDetail ON Orders.id=OrderDetail.id
GROUP BY Orders.id, Orders.customer
```

要完成这个运算，不仅要用到 JOIN，还需要做一次 GROUP BY，否则选出来的记录数太多。

如果我们把子表中与主表相关的记录看成主表的一个字段，那么这个问题也可以不再使用 JOIN 以及 GROUP BY：

```
SELECT id, customer, OrderDetail.SUM(price)
FROM Orders
```

与普通字段不同，OrderDetail 被看成 Orders 表的字段时，其取值将是一个集合，因为两个表是一对多的关系。所以要在这里使用聚合运算把集合值计算成单值。这种简化方式称为**子表集合化**。

这样看待主子表关联，不仅理解书写更为简单，而且不容易出错。

假如 Orders 表还有一个子表用于记录回款情况：

OrderPayment 订单回款表

id 订单编号

date 回款日期

amount 回款金额

....

我们现在想知道那些订单还在欠钱，也就是累计回款金额小于订单总金额的订单。

简单地把这三个表 JOIN 起来是不对的，OrderDetail 和 OrderPayment 会发生多对多的关系，这就错了（回忆前面提过的多对多大概率错误的说法）。这两个子表要分别先做 GROUP，再一起与 Orders 表 JOIN 起来才能得到正确结果，会写成子查询的形式：

```
SELECT Orders.id, Orders.customer,A.x,B.y
FROM Orders
LEFT JOIN ( SELECT id,SUM(price) x FROM OrderDetail GROUP BY id ) A
    ON Orders.id=A.id
LEFT JOIN ( SELECT id,SUM(amount) y FROM OrderPayment GROUP BY id ) B
    ON Orders.id=B.id
WHERE A.x>B.y
```

如果我们继续把子表看成主表的集合字段，那就很简单了：

```
SELECT id,customer,OrderDetail.SUM(price) x,OrderPayment.SUM(amount) y
```

```
FROM Orders WHERE x>y
```

这种写法也不容易发生多对多的错误。

主子表关系是不对等的，不过两个方向的引用都有意义，上面谈了从主表引用子表的情况，从子表引用主表则和外键表类似。

我们改变对 JOIN 运算的看法，摒弃笛卡尔积的思路，把多表关联运算看成是稍复杂些的单表运算。这样，相当于把最常见的等值 JOIN 运算的关联消除了，甚至在语法中取消了 JOIN 关键字，书写和理解都要简单很多。

四. 维度对齐语法

我们再回顾前面的双子表例子的 SQL：

```
SELECT Orders.id, Orders.customer, A.x, B.y
FROM Orders
LEFT JOIN (SELECT id,SUM(price) x FROM OrderDetail GROUP BY id ) A
    ON Orders.id=A.id
LEFT JOIN (SELECT id,SUM(amount) y FROM OrderPayment GROUP BY id ) B
    ON Orders.id=B.id
WHERE A.x > B.y
```

那么问题来了，这显然是个有业务意义的 JOIN，它算是前面所说的哪一类呢？

这个 JOIN 涉及了表 Orders 和子查询 A 与 B，仔细观察会发现，子查询带有 GROUP BY id 的子句，显然，其结果集将以 id 为主键。这样，JOIN 涉及的三个表（子查询也算作是个临时表）的主键是相同的，它们是一对一的同维表，仍然在前述的范围内。

但是，这个同维表 JOIN 却不能用前面说的写法简化，子查询 A,B 都不能省略不写。

可以简化书写的原因在于：我们假定事先知道数据结构中这些表之间的关联关系。用技术术语的说法，就是知道数据库的元数据（metadata）。而对于临时产生的子查询，显然不可能事先定义在元数据中了，这时候就必须明确指定要 JOIN 的表（子查询）。

不过，虽然 JOIN 的表（子查询）不能省略，但关联字段总是主键。子查询的主键总是由 GROUP BY 产生，而 GROUP BY 的字段一定要被选出用于做外层 JOIN；并且这几个子查询涉及的子表是互相独立的，它们之间不会再有关联计算了，我们就可以把 GROUP 动作以及聚合式直接放到主句中，从而消除一层子查询：

```
SELECT Orders.id, Orders.customer, OrderDetail.SUM(price) x, OrderParyment.SUM(amount) y
FROM Orders
LEFT JOIN OrderDetail GROUP BY id
LEFT JOIN OrderPayment GROUP BY id
WHERE A.x > B.y
```

这里的 JOIN 和 SQL 定义的 JOIN 运算已经差别很大，完全没有笛卡尔积的意思了。而且，也不同于 SQL 的 JOIN 运算将定义在任何两个表之间，这里的 JOIN，OrderDetail 和 OrderPayment 以及 Orders 都是向一个共同的主键 id 对齐，即所有表都向某一套基准维度对齐。而由于各表的维度（主键）不同，对齐时可能会有 GROUP BY，在引用该表字段时就会相应地出现聚合运算。OrderDetail 和 OrderPayment 甚至 Orders 之间都不直接发生关联，在书写

运算时当然就不用关心它们之间的关系，甚至不必关心另一个表是否存在。而 SQL 那种笛卡尔积式的 JOIN 则总要找一个甚至多个表来定义关联，一旦减少或修改表时就要同时考虑关联表，增大理解难度。

我们称这种 JOIN 称为**维度对齐**，它并不超出我们前面说过的三种 JOIN 范围，但确实在语法描述上会有不同，这里的 JOIN 不象 SQL 中是个动词，却更象个连词。而且，和前面三种基本 JOIN 中不会或很少发生 FULL JOIN 的情况不同，维度对齐的场景下 FULL JOIN 并不是很罕见的情况。

虽然我们从主子表的例子抽象出维度对齐，但这种 JOIN 并不要求 JOIN 的表是主子表（事实上从前面的语法可知,主子表运算还不用写这么麻烦),任何多个表都可以这么关联，而且关联字段也完全不必要是主键或主键的部分。

设有合同表，回款表和发票表：

Contract 合同表
- id 合同编号
- date 签订日期
- customer 客户
- price 合同金额
- ...

Payment 回款表
- seq 回款序号
- date 回款日期
- source 回款来源
- amount 金额
- ...

Invoice 发票表
- code 发票编号
- date 开票日期
- customer 客户
- amount 开票金额
- ...

现在想统计每一天的合同额、回款额以及发票额，就可以写成：

```
SELECT Contract.SUM(price), Payment.SUM(amount), Invoice.SUM(amount) ON date
FROM Contract GROUP BY date
FULL JOIN Payment GROUP BY date
FULL JOIN Invoice GROUP BY date
```

这里需要把 date 在 SELECT 后单独列出来表示结果集按日期对齐。

这种写法，不必关心这三个表之间的关联关系，各自写各自有关的部分就行，似乎这几个表就没有关联关系，把它们连到一起的就是那个要共同对齐的维度（这里是 date）。

这几种 JOIN 情况还可能混合出现。

继续举例，延用上面的合同表，再有客户表和销售员表

Customer 客户表
- id 客户编号
- name 客户名称

area 所在地区
...
Sales 销售员表
id 员工编号
name 姓名
area 负责地区
...

其中 Contract 表中 customer 字段是指向 Customer 表的外键。

现在我们想统计每个地区的销售员数量及合同额：

```
SELECT Sales.COUNT(1), Contract.SUM(price) ON area
FROM Sales GROUP BY area
FULL JOIN Contract GROUP BY customer.area
```

维度对齐可以和外键属性化的写法配合合作。

这些例子中，最终的 JOIN 都是同维表。事实上，维度对齐还有主子表对齐的情况，不过相对罕见，我们这里就不深入讨论了。

另外，目前这些简化语法仍然是示意性 ，需要在严格定义维度概念之后才能相应地形式化，成为可以解释执行的句子。

我们把这种简化的语法称为 DQL（Dimensional Query Languange），DQL 是以维度为核心的查询语言。我们已经将 DQL 在工程上做了实现，并作为润乾报表的 DQL 服务器发布出来，它能将 DQL 语句翻译成 SQL 语句执行，也就是可以在任何关系数据库上运行。

对 DQL 理论和应用感兴趣的读者可以关注乾学院上发布的论文和相关文章。

五. 解决关联查询

我们重新审视和定义了等值 JOIN 运算，并简化了语法。一个直接的效果显然是让语句**书写和理解更容易**。外键属性化、同维表等同化和子表集合化方案直接消除了显式的关联运算，也更符合自然思维；维度对齐则可让程序员不再关心表间关系，降低语句的复杂度。

简化 JOIN 语法的好处不仅在于此，还能够**降低出错率**。

我们知道，SQL 允许用 WHERE 来写 JOIN 运算的过滤条件（回顾原始的笛卡尔积式的定义），很多程序员也习惯于这么写。当 JOIN 表只有两三个的时候，那问题还不大，但如果 JOIN 表有七八个甚至十几个的时候，漏写一个 JOIN 条件是很有可能的。而漏写了 JOIN 条件意味着将发生多对多的完全叉乘，而这个 SQL 却可以正常执行，一方面计算结果会出错（回忆一下以前说过的，发生多对多 JOIN 时，大概率是语句写错了），另一方面，如果漏写条件的表很大，笛卡尔积的规模将是平方级的，这极有可能把数据库直接“跑死”！

采用简化后的 JOIN 语法，就不可能发生漏写 JOIN 条件的情况了。因为对 JOIN 的理解不再是以笛卡尔积为基础，而且设计这些语法时已经假定了多对多关联没有业务意义，这个规则下写不出完全叉乘的运算。

对于多个子表分组后与主表对齐的运算，在 SQL 中要写成多个子查询的形式。但如果只有一个子表时，可以先 JOIN 再 GROUP，这时不需要子查询。有些程序员没有仔细分析，会把这种写法推广到多个子表的情况，也先 JOIN 再 GROUP，可以避免使用子查询，但计算结果是错误的。

使用维度对齐的写法就不容易发生这种错误了，无论多少个子表，都不需要子查询，一个子表和多个子表的写法完全相同。

重新看待JOIN运算，最关键的作用在于**实现关联查询**。

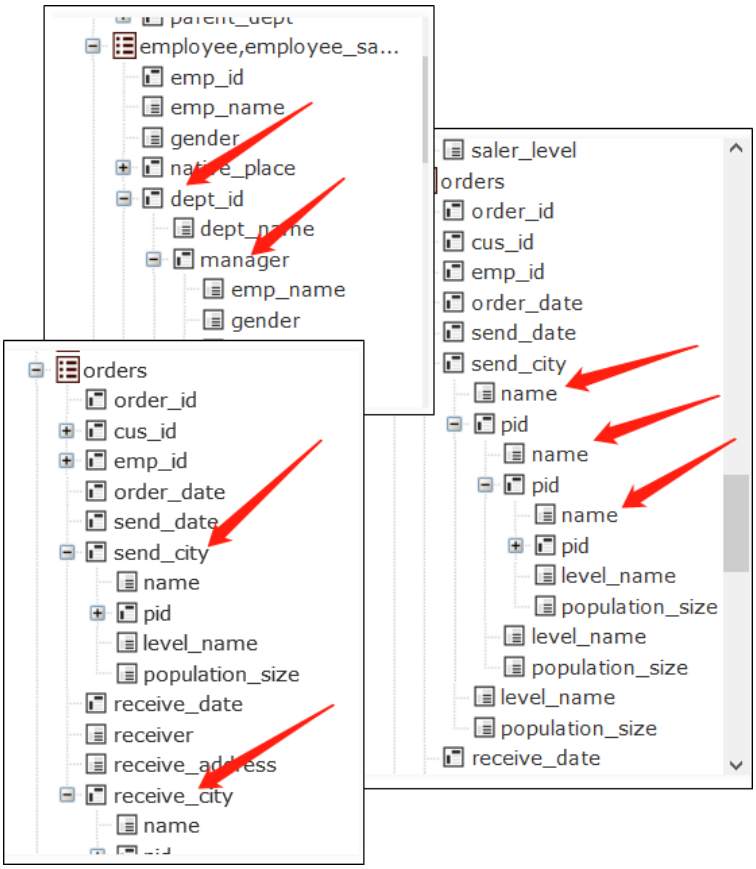
当前BI产品是个热门，各家产品都宣称能够让业务人员拖拖拽拽就完成想要的查询报表。但实际应用效果会远不如人意，业务人员仍然要经常求助于IT部门。造成这个现象的主要原因在于大多数业务查询都是有过程的计算，本来也不可能拖拽完成。但是，也有一部分业务查询并不涉及多步过程，而业务人员仍然难以完成。

这就是关联查询，也是大多数BI产品的软肋。在之前的文章中已经讲过为什么关联查询很难做，其根本原因就在于SQL对JOIN的定义过于简单。

结果，BI产品的工作模式就变成先由技术人员构建模型，再由业务人员基于模型进行查询。而所谓建模，就是生成一个逻辑上或物理上的宽表。也就是说，建模要针对不同的关联需求分别实现，我们称之为**按需建模**，这时候的BI也就失去敏捷性了。

但是，如果我们改变了对JOIN运算的看法，关联查询可以从根本上得到解决。回忆前面讲过的三种JOIN及其简化手段，我们事实上把这几种情况的多表关联都转化成了单表查询，而业务用户对于单表查询并没有理解障碍。无非就是表的属性（字段）稍复杂了一些：可能有子属性(外键字段指向的维表并引用其字段),子属性可能还有子属性(多层的维表)，有些字段取值是集合而非单值（子表看作为主表的字段）。发生互相关联甚至自我关联也不会影响理解（前面的中国经理的美国员工例子就是互关联），同表有相同维度当然更不碍事（各自有各自的子属性）。

在这种关联机制下，技术人员只要一次性把数据结构（元数据）定义好，在合适的界面下（把表的字段列成有层次的树状而不是常规的线状），就可以由业务人员自己实现JOIN运算，不再需要技术人员的参与。数据建模只发生于数据结构改变的时刻，而不需要为新的关联需求建模，这也就是**非按需建模**，在这种机制支持下的BI才能拥有足够的敏捷性。



六、外键预关联

我们再来研究如何利用 JOIN 的特征实现性能优化，这些内容的细节较多，我们挑一些易于理解的情况来举例，更完善的连接提速算法可以参考乾学院上的《性能优化》图书和 SPL 学习资料中的性能优化专题文章。

先看全内存下外键关联的情况。

设有两个表：

customer 客户信息表

key 编号

name 名称

city 城市

...

orders 订单表

seq 序号

date 日期

custkey 客户编号

amount 金额

...

其中 orders 表中的 custkey 是指向 customer 表中 key 字段的外键，key 是 customer 表的主键。

现在我们各个城市的订单总额（为简化讨论，就不再设定条件了），用 SQL 写出来：

```
SELECT customer.city, SUM(orders.amount)
FROM orders
JOIN customer ON orders.custkey=customer.key
GROUP BY customer.city
```

数据库一般会使用 HASH JOIN 算法，需要分别两个表中关联键的 HASH 值并比对。

我们用前述的简化的 JOIN 语法（DQL）写出这个运算：

```
SELECT custkey.city, SUM(amount)
FROM orders
GROUP BY custkey.city
```

这个写法其实也就预示了它还可以有更好的优化方案，下面来看看怎样实现。

如果所有数据都能够装入内存，我们可以实现**外键地址化**。

将事实表 orders 中的外键字段 custkey，转换成维表 customer 中关联记录的地址，即 orders 表的 custkey 的取值已经是某个 customer 表中的记录，那么就可以直接引用记录的字段进行计算了。

用 SQL 无法描述这个运算的细节过程，我们使用 SPL 来描述、并用文件作为数据源来说明计算过程：

	A
1	=file("customer.btx").import@b()
2	>A1.keys@i(key)
3	=file("orders.btx").import@b()
4	>A3.switch(custkey,A1)
5	=A3.groups(custkey.city;sum(amount))

A1 读出客户表，A2 为客户表设置主键并建立索引。

A3 读出订单表，A4 的动作是将 A3 的外键字段 custkey 转换成对应的 A1 的记录，执行完后，订单表字段 custkey 将变成客户表的某条记录。A2 建了索引能让 switch 更快，因为通常事实表远大于维表，这个索引能被复用很多次。

A5 就可以执行分组汇总了，遍历订单表时，由于 custkey 字段取值现在已经是一条记录，那么可以直接用.操作符引用其字段了，custkey.city 就可以正常执行。

完成 A4 中的 switch 动作之后，内存中事实表 A3 的 custkey 字段存储内容已经是维表 A1 的某条记录的地址，这个动作即称为外键地址化。这时候引用维表字段时，可以直接取出，而不需要再用外键值在 A1 中查找，相当于在常数时间内就能取到维表的字段，避免了 HASH 值计算和比对。

不过，A2 建主键索引一般也会用 HASH 办法，对 key 计算 HASH 值，A4 转换地址时也是计算 custkey 的 HASH 值与 A2 的 HASH 索引表对比。如果只做一次关联运算，地址化的方案和传统 HASH 分段方案的计算量基本上一样，没有根本优势。

但不同的是，如果数据能在内存中放下，这个地址一旦转换之后可以复用，也就是说 A1 到 A4 只要做一次，下次再做关于这两个字段的关联运算时就不必再计算 HASH 值和比对了，性能就能大幅提高。

能够这样做，正是利用了前面说过的外键关联在维表这一方具有的唯一性，一个外键字段值只会唯一对应一条维表记录，可以把每个 custkey 转换成它唯一对应的那条 A1 的记录。而延用 SQL 中对 JOIN 的定义，就不能假定外键指向记录的唯一性，无法使用这种表示法。而且 SQL 也没有记录地址这种数据类型，结果会导致每次关联时都要计算 HASH 值并比对。

而且，如果事实表中有多个外键分别指向多个维表，传统的 HASH 分段 JOIN 方案每次只能解析掉一个，有多个 JOIN 要执行多遍动作，每次关联后都需要保持中间结果供下一轮使用，计算过程复杂得多，数据也会被遍历多次。而外键地址化方案在面对多个外键时，只要对事实表遍历一次， 没有中间结果，计算过程要清晰很多。

还有一点，内存本来是很适合并行计算的，但 HASH 分段 JOIN 算法却不容易并行。即使把数据分段并行计算 HASH 值，但要把相同 HASH 值的记录归聚到一起供下一轮比对，还会发生共享资源抢占的事情，这将牺牲很多并行计算的优势。而外键式 JOIN 模型下，关联两表的地位不对等，明确区分出维表和事实表后，只要简单地将事实表分段就可以并行计算。

将 HASH 分段技术参照外键属性方案进行改造后，也能一定程度地改善多外键一次解析和并行能力，有些数据库能在工程层面上实施这种优化。不过，这种优化在只有两个表 JOIN 时问题不大，在有很多表及各种 JOIN 混在一起时，数据库并不容易识别出应当把哪个表当作事实表去并行遍历、而把其它表当作维表建立 HASH 索引，这时优化并不总是有效的。所以我们经常会发现当 JOIN 的表变多时性能会急剧下降的现象(常常到四五个表时就会发生，结果集并无显著增大）。而从 JOIN 模型上引入外键概念后，将这种 JOIN 专门处理时，就总能分清事实表和维表，更多的 JOIN 表只会导致性能的线性下降。

内存数据库是当前比较火热的技术，但上述分析表明，采用 SQL 模型的内存数据库在 JOIN 运算上是很难快起来的！

七、进一步的外键关联

我们继续讨论外键 JOIN，并延用上一节的例子。

当数据量大到无法全部放进内存时，前述的地址化方法就不再有效了，因为在外存无法保存事先算好的地址。

一般来讲，外键指向的维表容量较小，而不断增长的事实表要大得多。如果内存还能把维表放下的话，我们可以采用临时指向的方法来处理外键。

	A
1	=file("customer.btx").import@b()
2	>A1.keys@i(key)
3	=file("orders.btx").cursor@b()
4	>A3.switch(custkey,A1)
5	=A3.groups(custkey.city;sum(amount))

前两步与全内存时相同，第 4 步的地址转换是边读入边进行的，而且转换结果无法保留复用，下次再做关联时还要再计算 HASH 和比对，性能要比全内存的方案差。计算量方面，比 HASH JOIN 算法少了一次维表的 HASH 值计算，这个维表如果经常被复用时会占些便宜，但因为维表相对较小，总体优势并不算大。不过，这个算法同样具有全内存算法可以一次解析全部外键以及易于并行的特点，在实际场景下比 HASH JOIN 算法仍有较大的性能优势。

在这个算法基础上，我们还可以做个变种：**外键序号化**。

如果我们能把维表的主键都转换成从 1 开始的自然数，那么我们就可以用序号直接定位维表记录,就不需要计算和比对 HASH 值,这样就可以获得类似全内存下地址化的性能了。

	A
1	=file("customer.btx").import@b()
2	=file("orders.btx").cursor@b()
3	>A2.switch(custkey,A1:#)
4	=A2.groups(custkey.city;sum(amount))

维表主键是序号时就不需要再做原来建 HASH 索引的第 2 步了。

外键序号化本质上相当于在外存实现地址化。这种方案需要把事实表中的外键字段转换成序号,这类似在全内存运算时地址化的过程,这个预计算也可以得到复用。需要注意的是，维表发生重大变化时，需要同步整理事实表的外键字段，否则可能对应错位。不过一般维表变化频度低，而且大多数动作是追加和修改而非删除，需要重整事实表的情况并不多。工程上的细节处理也可以再参考乾学院中的资料。

SQL 使用了无序集合的概念，即使我们事先把外键序号化了，数据库也无法利用这个特点，不能在无序集合上使用序号快速定位的机制，只能使用索引查找，而且数据库并不知道外键被序号化了，仍然会去计算 HASH 值和比对。

如果维表也大到内存装不下呢？

我们仔细分析上面的算法会发现，过程中对于事实表的访问是连续的，但对于维表的访问则是随机的。我们以前讨论硬盘的性能特征时谈到过，外存不适合随机访问，所以外存中的维表不能再使用上述算法了。

外存中的维表可以事先按主键排序存储,这样我们就可以继续利用维表关联键是主键的特征来优化性能。

如果事实表很小,可以在内存装放下,那么用外键去关联维表记录实际上会变成一个(批量）外存查找动作。只要维表上针对主键建有索引，也可以很快地查找，这样可以避免遍历大维表，获得更好的性能。这种算法也可以同时解析多个外键。SQL 不区分维表和事实表，面对一大一小两个表时，优化过的 HASH JOIN 不会再做分堆缓存，通常会把小表读入内存而去遍历大表，这样仍然会有遍历大维表的动作，性能会比刚才说的外存查找算法差得多。

如果事实表也很大，则可以使用**单边分堆**的算法。因为维表已经按关联键（即主键）有序，可以方便地逻辑上分成若干段并取出每一段的边界值（每一段主键的最大最小值），然后将事实表按这些边界值做分堆，每一堆分别和维表的每一段再做关联就可以了。过程中只需要对事实表单边做物理分堆缓存,维表不需要再做物理分堆缓存,而且不使用 HASH 函数，直接用分段，不可能会出现 HASH 函数运气不好导致二次分堆，性能是可控的。而数据库的 HASH 分堆算法会将两个大表都做物理分堆缓存，也就是双边分堆，还可能出现 HASH 函数运气不好导致二次分堆的现象，性能要比单边分堆差得多，还不可控。

还可以借助集群的力量解决大维表问题。

一台机器的内存装不下，可以多搞几台机器来装下，把维表按主键值分段存放在多台机器上形成**集群维表**，然后就可以继续使用上面针对内存维表的算法了，也能获得一次解析多个外键和易于并行的好处。同样地，集群维表也可以使用序号化的技术。这种算法下，事实表不需要被传输，产生的网络传输量并不大，也不需要节点本地缓存数据。而 SQL 体系下不能区分出维表，HASH 拆分方法要将两个表都做 Shuffle 动作，网络传播量要大得多。

这些算法的细节仍有些复杂，这里限于篇幅无法详细解释，有兴趣的读者可以去乾学院的资料查阅。

八、有序归并

我们再来看同维表和主子表的 JOIN，这两种情况的优化提速手段是类似的。

我们前面讨论过，HASH JOIN 算法的计算复杂度（即关联键的比较次数）是 SUM(n_i*m_i)，比全遍历的复杂度 $n*m$ 要小很多，不过要看 HASH 函数的运气。

如果这两个表都对关联键有序，那么我们就可以使用归并算法来处理关联，这时的复杂度是 $n+m$；在 n 和 m 都较大的时候（一般都会远大于 HASH 函数的取值范围），这个数也会远小于 HASH JOIN 算法的复杂度。归并算法的细节有很多材料介绍，这里就不再赘述了。

但是，外键 JOIN 时不能使用这个办法，因为事实表上可能有多个要参与关联的外键字段，不可能让同一个事实表同时针对多个字段都有序。

同维表和主子表却可以！

因为同维表和主子表总是针对主键或主键的一部分关联，我们可以事先把这些关联表的数据按其主键排序。排序的成本虽然较高，但是一次性的。一旦完成了排序，以后就可以总是使用归并算法实现 JOIN，性能可以提高很多。

这还是利用了关联键是主键（及其部分）的特征。

有序归并对于大数据特别有效。像订单及其明细这种主子表是不断增长的事实表，时间长了常常会积累得非常大，很容易超出内存容量。

外存大数据的 HASH 分堆算法需要产生很多缓存，数据在外存中被读两次写一次，IO 开销很大。而归并算法时，两个表的数据都只要读一次就行了，没有写。不仅是 CPU 的计算量减少，外存的 IO 量也大幅下降。而且，执行归并算法需要的内存很少，只要在内存中为每个表保持数条缓存记录就可以了，几乎不会影响其它并发任务对内存的需求。而 HASH 分堆需要较大内存，每次读出更多数据，以减少分堆的次数。

SQL 采用笛卡尔积定义的 JOIN 运算不区分 JOIN 类型，不假定某些 JOIN 总是针对主键的，就没办法从算法层面上利用这一特点，只能在工程层面进行优化。有些数据库会检查数据表在物理存储上是否针对关联字段有序，如果有序则采用归并算法，但基于无序集合概念的关系数据库不会刻意保证数据的物理有序性，许多操作都会破坏归并算法的实施条件。使用索引可以实现数据的逻辑有序，但物理无序时的遍历效率还是会大打折扣。

有序归并的前提是将数据按主键排序，而这类数据常常会不断追加，原则上每次追加后就要再次排序，而我们知道大数据排序成本通常很高，这是否会导致追加数据难度很大呢？其实，追加数据再加入的过程也是个有序归并，把新增数据单独排序后和已有序的历史数据归并，复杂度是线性的，相当于把所有数据重写一次，而不像常规的大数据排序需要缓存式写出再读入。在工程上做些优化动作还可以做到不必每次都全部重写，进一步提高维护效率。这些在乾学院上都有介绍。

有序归并的好处还在于易于分段并行。

现代计算机的都有多核 CPU，SSD 硬盘也有较强的并发能力，使用多线程并行计算就能够显著提高性能。但传统的 HASH 分堆技术实现并行比较困难，多线程做 HASH 分堆时需要同

时向某个分堆写出数据，造成共享资源冲突；而第二步实现某组分堆关联时又会消费大量内存，无法让实施较大的并行数量。

使用有序归并实现并行计算时需要把数据分成多段，单个表分段比较简单，但两个关联表分段时必须同步对齐，否则归并时两个表数据错位了，就无法得出正确的计算结果，而数据有序就可以保证高性能的同步对齐分段。

先把主表（同维表则取较大的即可，其它讨论不影响）平均分成若干段，读出每段第一条记录的主键值，然后用这些键值到子表中用二分法寻找定位（因为也有序），从而获得子表的分段点。这样可以保证主子表的分段是同步对齐的。

因为键值有序，所以主表每段的记录键值都属于某个连续区间，键值在区间外的记录不会在这一段，键值在区间内的记录一定在这一段，子表对应分段的记录键值也有这个特性，所以不会发生错位情况；而同样因为键值有序，才可以在子表中执行高效的二分查找迅速定位出分段点。即数据有序保证了分段的合理性及高效性，这样就可以放心地执行并行算法了。

主子表这种主键关联的关系还有一个特征，就是子表只会和一个主表在主键上关联（其实同维表也有，但用主子表容易解释），它不会有多个相互无关的主表（可能有主表的主表）。这时候，还可以使用一体化存储的机制，把子表记录作为主表的字段值去存储。这样，一方面减少了存储量（关联键只要存储一次），又相当于预先做好了关联，不需要再做比对了。对于大数据，就能获得更好的性能。

我们已经将上述这些性能优化手段在集算器 SPL 中实现并在实际场景用得到了应用，也取得了非常好的效果。SPL 目前已经开源，读者可以去数速公司或润乾公司官网及论坛下载并获得更多资料。

结语

JOIN 运算确实是数据库中最复杂的运算，本文对 JOIN 运算进行了深入的剖析整理，篇幅已经不小，但仍然也没有完全穷尽所有方面。

感兴趣的同学可以进一步研读乾学院(c.raqsoft.com.cn)上的图书和文章。

本文内容还有个视频材料：http://c.raqsoft.com.cn/article/1653095843929