# Join Operation in Impala

Prepared by: Yuanhao Luo

# What to talk about？

- Impala 中连接的类型
- Impala 实现连接的三种方式
- Impala 实现连接的两种策略

# Impala 中连接的类型

- **Self-joins**
- **Cartesian joins**
- **Inner joins**
- **{Left/Right/Full} [Outer] Joins**
- **Equijoins and Non-Equijoins**
- **[Left/Right] Semi-joins**
- **[Left/Right] Anti-joins**
- **Natural joins (not supported)**

# Cartesian joins

- 在了解 SQL 连接之前我们首先应该知道Cartesian Product(笛卡尔积)，也称为cross join(交叉连接)
- 两个或多个表的笛卡尔积：
- 考虑如下两个表：

- product

| PID | Pname |
|---|---|
| 1 | Shirt |
| 2 | Pajabi |
| 3 | Lungi |

sale

| SID | ProductID | Price |
|---|---|---|
| 101 | 1 | 1000 |
| 102 | 2 | 800 |
| 103 | 5 | 400 |
| 104 | 2 | 600 |

# 两个或多个表的笛卡尔积：

QUERY: Select * from Product, Sale;

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 1 | Shirt | 102 | 2 | 800 |
| 1 | Shirt | 103 | 5 | 400 |
| 1 | Shirt | 104 | 2 | 600 |
| 2 | Pajabi | 101 | 1 | 1000 |
| 2 | Pajabi | 102 | 2 | 800 |
| 2 | Pajabi | 103 | 5 | 400 |
| 2 | Pajabi | 104 | 2 | 600 |
| 3 | Lungi | 101 | 1 | 1000 |
| 3 | Lungi | 102 | 2 | 800 |
| 3 | Lungi | 103 | 5 | 400 |
| 3 | Lungi | 104 | 2 | 600 |

# *笛卡尔积是表中记录的所有可能的组合*

假如有两个表$T_1$ 和 $T_2$

$T_1$ 有 $r_1$ 行和 $c_1$ 列

$T_2$ 有 $r_2$ 行和 $c_2$ 列

$T_1$ 和 $T_2$ 的笛卡尔积就有：

$r_1 * r_2$ 行和 $c_1 + c_2$ 列.

[对于多个表就是 $r_1 * r_2 * r_3$ … 行和 $c_1 + c_2 + c_3$ …列]

在后面的例子中都会使用以下两个表作为事例：

Product

| PID | Pname |
|---|---|
| 1 | Shirt |
| 2 | Pajabi |
| 3 | Lungi |

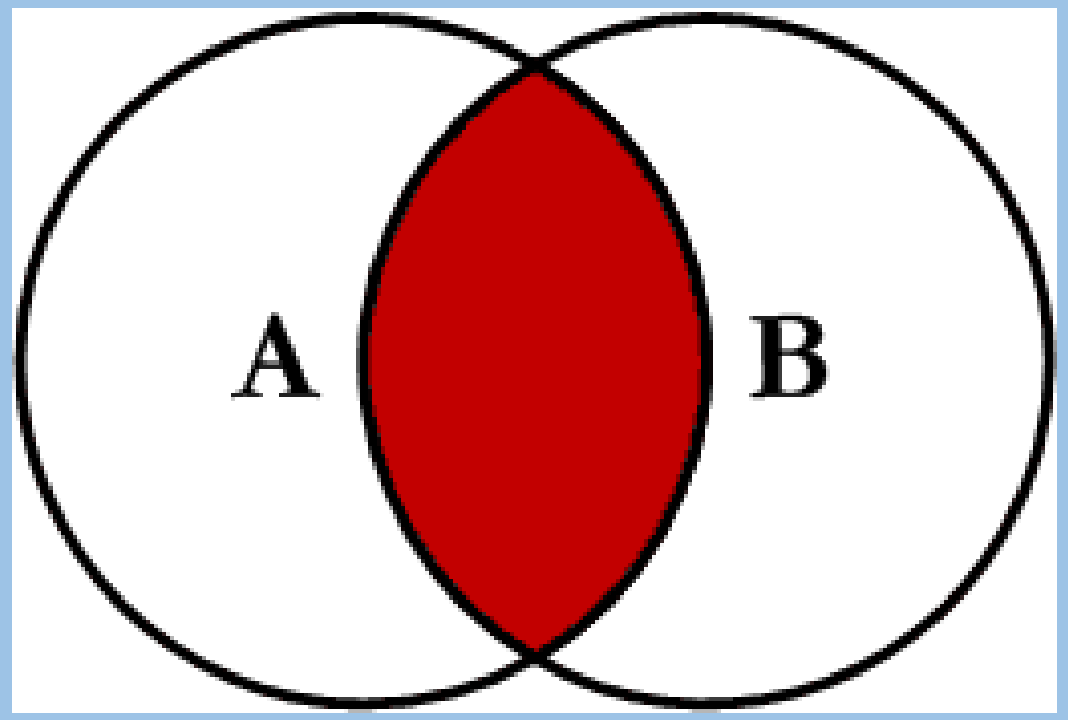
Sale

| SID | ProductID | Price |
|---|---|---|
| 101 | 1 | 1000 |
| 102 | 2 | 800 |
| 103 | 5 | 400 |
| 104 | 2 | 600 |

# Inner join

**Query:** Select * from Product p Inner join Sale s on p.PID = s.ProductID

内连接的结果只包括笛卡尔积中连接条件（ p.PID = s.ProductID ）完全匹配的记录




# Inner join

**Query:** Select * from Product p Inner join Sale s on p.PID = s.ProductID

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 1 | Shirt | 102 | 2 | 800 |
| 1 | Shirt | 103 | 5 | 400 |
| 1 | Shirt | 104 | 2 | 600 |
| 2 | Pajabi | 101 | 1 | 1000 |
| 2 | Pajabi | 102 | 2 | 800 |
| 2 | Pajabi | 103 | 5 | 400 |
| 2 | Pajabi | 104 | 2 | 600 |
| 3 | Lungi | 101 | 1 | 1000 |
| 3 | Lungi | 102 | 2 | 800 |
| 3 | Lungi | 103 | 5 | 400 |
| 3 | Lungi | 104 | 2 | 600 |

# Inner join

**Query:** `Select * from Product p Inner join Sale s on p.PID = s.ProductID`

最终连接结果如下：
要注意的是，如果有重复的行，所有的行都会放到结果中。

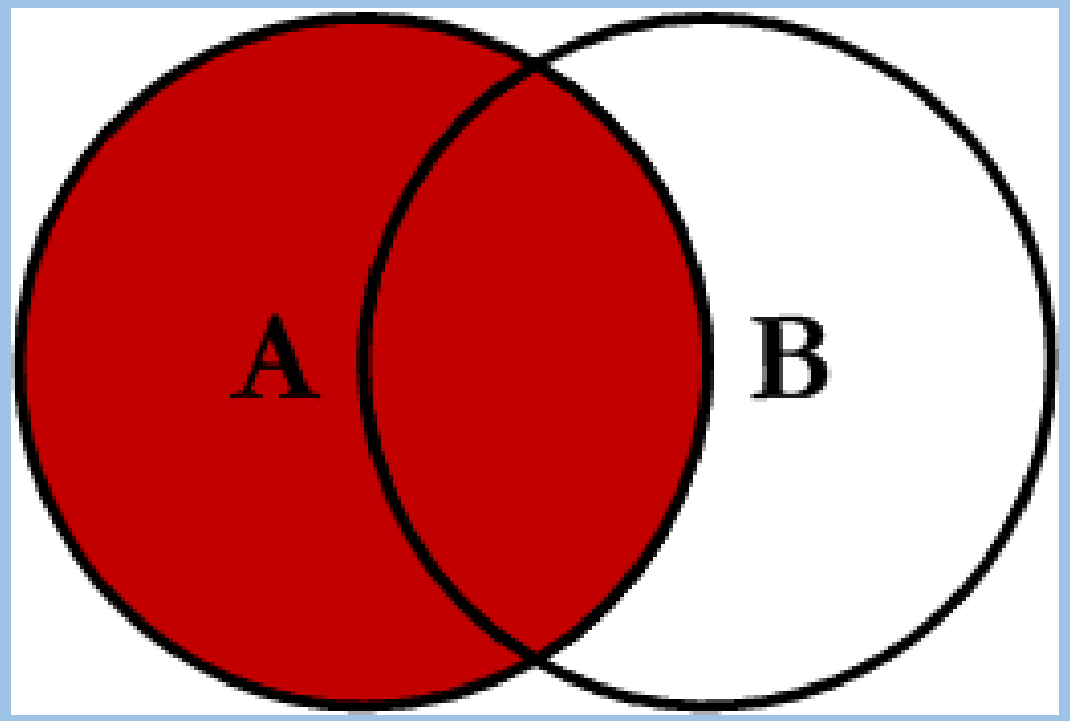
| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 2 | Panjabi | 102 | 2 | 800 |
| 2 | Panjabi | 104 | 2 | 600 |

# Left outer join

**Query:** Select * from Product p Left join Sale s on p.PID = s.ProductID

左外连接结果包括：

1. 内连接的结果
2. 左表中在右表没有对应行的记录，但右表中对应列的值都为NULL




# Left outer join

**Query:** Select * from Product p Left join Sale s on p.PID = s.ProductID

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 1 | Shirt | 102 | 2 | 800 |
| 1 | Shirt | 103 | 5 | 400 |
| 1 | Shirt | 104 | 2 | 600 |
| 2 | Pajabi | 101 | 1 | 1000 |
| 2 | Pajabi | 102 | 2 | 800 |
| 2 | Pajabi | 103 | 5 | 400 |
| 2 | Pajabi | 104 | 2 | 600 |
| 3 | Lungi | 101 | 1 | 1000 |
| 3 | Lungi | 102 | 2 | 800 |
| 3 | Lungi | 103 | 5 | 400 |
| 3 | Lungi | 104 | 2 | 600 |

# Left outerjoin

**Query:** Select * from Product p Left join Sale s on p.PID = s.ProductID

左外连接的结果如下：
在左表中 ***Product*** |**PID = 3 | Pname = Lungi**|
这一行在 Sale 中没有对应 PID= 3的列，因此右边列的值用 NULL填充。

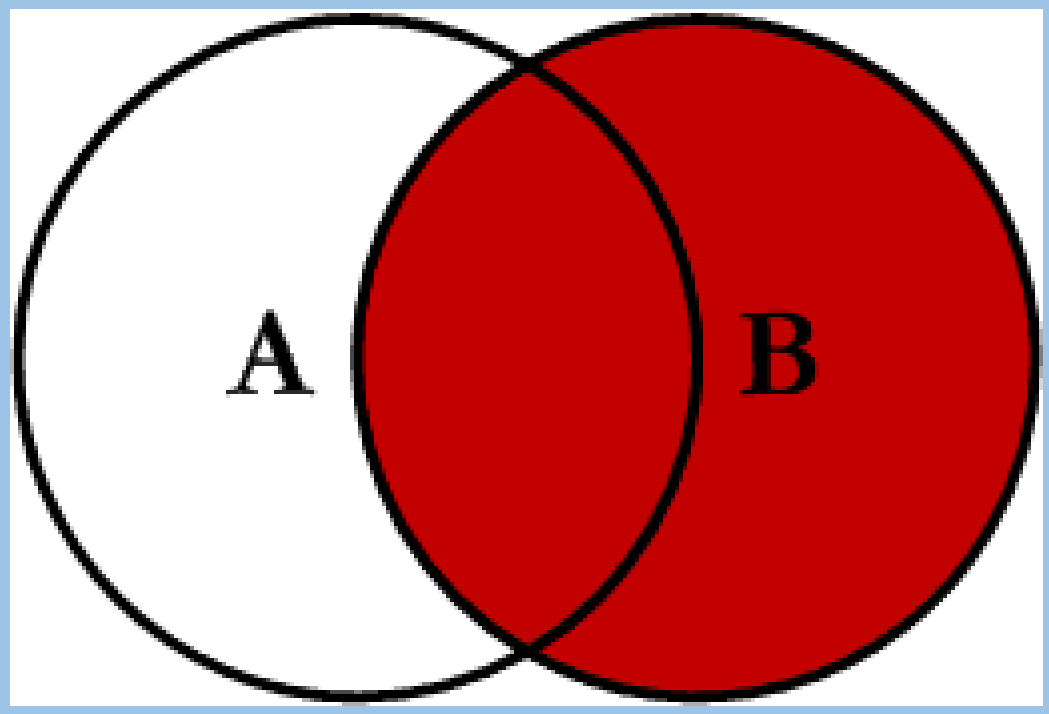
| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 2 | Panjabi | 102 | 2 | 800 |
| 2 | Panjabi | 104 | 2 | 600 |
| 3 | Lungi | null | null | null |

# Right outer join

**Query:** Select * from Product p Right join Sale s on p.PID = s.ProductID

右外连接的结果包括：

1. 内连接的结果
2. 右表中在左表中没有对应行的记录，其中左表中对应的值用NULL填充。




# Right outer join

**Query:** Select * from Product p Right join Sale s on p.PID = s.ProductID

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 1 | Shirt | 102 | 2 | 800 |
| 1 | Shirt | 103 | 5 | 400 |
| 1 | Shirt | 104 | 2 | 600 |
| 2 | Pajabi | 101 | 1 | 1000 |
| 2 | Pajabi | 102 | 2 | 800 |
| 2 | Pajabi | 103 | 5 | 400 |
| 2 | Pajabi | 104 | 2 | 600 |
| 3 | Lungi | 101 | 1 | 1000 |
| 3 | Lungi | 102 | 2 | 800 |
| 3 | Lungi | 103 | 5 | 400 |
| 3 | Lungi | 104 | 2 | 600 |

# Right outer join

**Query:** Select * from Product p Right join Sale s on p.PID = p.ProductID

右外连接的最终结果如下：
右表中 ***Sale*** **|SID = 103|ProductID = 5|Price = 400|行在左表中没有对应PID= 5 的行，因此左表对应的列的值用NULL填充。**

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 2 | Panjabi | 102 | 2 | 800 |
| 2 | Panjabi | 104 | 2 | 600 |
| null | null | 103 | 5 | 400 |

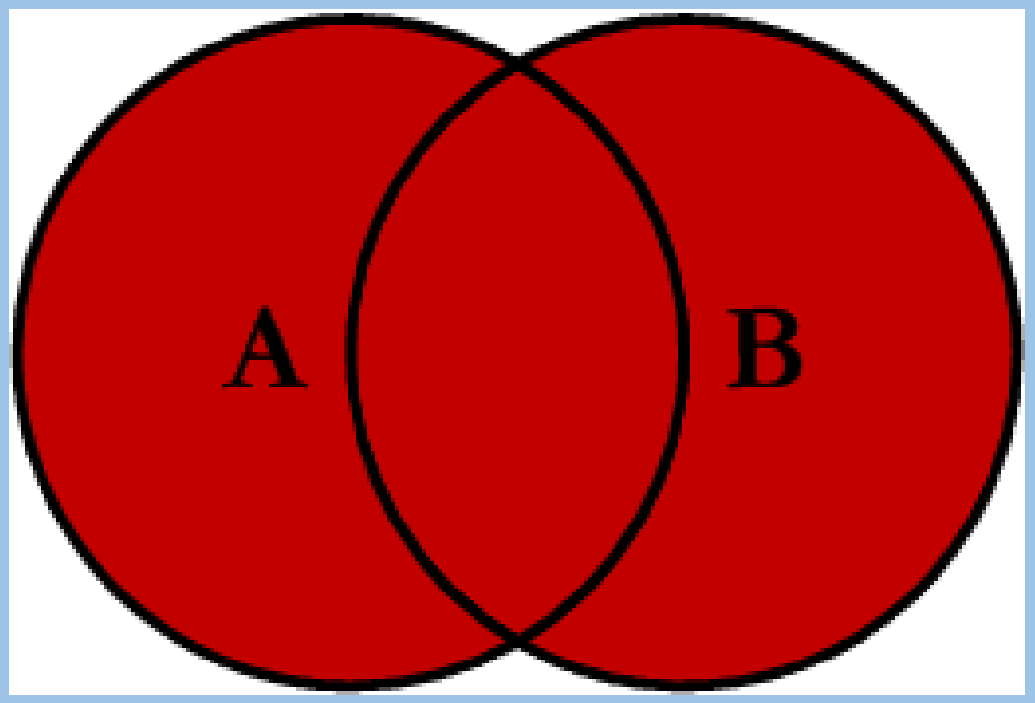
# Full outer join

**Query:** Select * from Product p full join Sale s on p.PID = s.ProductID

全连接的结果包括：

1. 内连接的结果
2. 左表中在右表没有对应行的记录
3. 右表中在左表没有对应行的记录

其中没有对应记录时列用NULL填充。




# Full outer join

**Query:** Select * from Product p full join Sale s on p.PID = s.ProductID

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 1 | Shirt | 102 | 2 | 800 |
| 1 | Shirt | 103 | 5 | 400 |
| 1 | Shirt | 104 | 2 | 600 |
| 2 | Pajabi | 101 | 1 | 1000 |
| 2 | Pajabi | 102 | 2 | 800 |
| 2 | Pajabi | 103 | 5 | 400 |
| 2 | Pajabi | 104 | 2 | 600 |
| 3 | Lungi | 101 | 1 | 1000 |
| 3 | Lungi | 102 | 2 | 800 |
| 3 | Lungi | 103 | 5 | 400 |
| 3 | Lungi | 104 | 2 | 600 |

# Full outer join

**Query:** Select * from Product p full join Sale s on p.PID = s.ProductID

第四行中左表在右表中没有对应的记录，因此右表对应的列用NULL填充；第五行相反。

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 1 | Shirt | 101 | 1 | 1000 |
| 2 | Panjabi | 102 | 2 | 800 |
| 2 | Panjabi | 104 | 2 | 600 |
| 3 | Lungi | null | null | null |
| null | null | 103 | 5 | 400 |

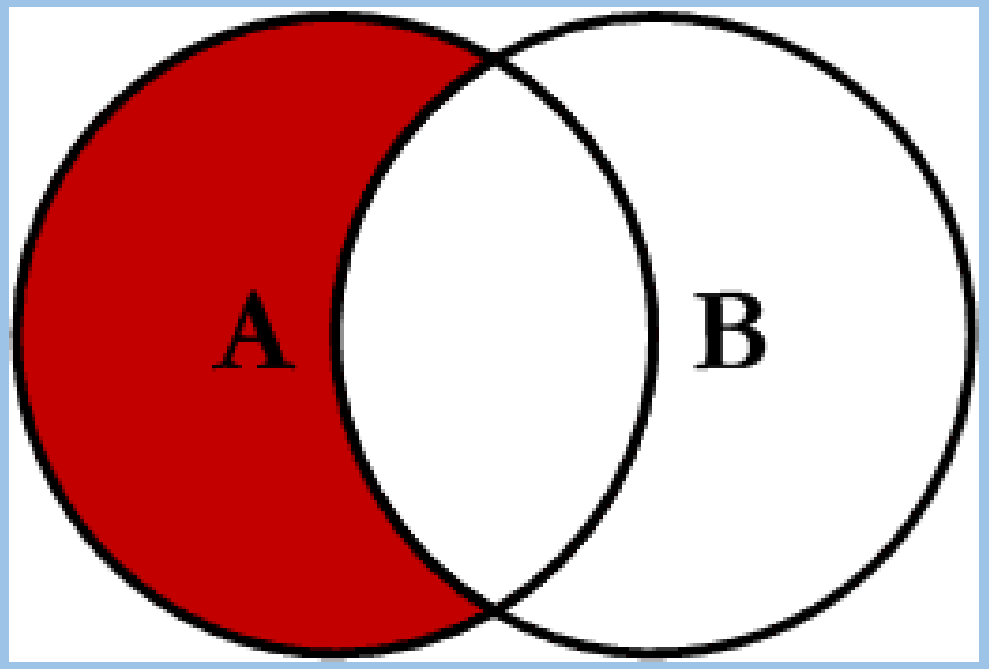
# Keep in mind

笛卡尔积不考虑匹配，而是获取所有可能的组合
而对于不同类型的连接：
或者匹配列的值相等，或者其中一个为NULL

- 内连接结果集中没有NULL.
- 左连接中右表对应列可能为NULL
- 右连接中左表对应列可能为NULL
- 全连接中左右表对应列都可能为NULL，但不能同时为NULL
- 左连接中左表的所有记录都会在结果集中
- 右连接中右表的所有记录都会在结果集中
- 连接结果的记录数目一定不大于笛卡尔积记录的数目

# Left anti join

反连接的结果只包括不能根据连接条件进行匹配的行，因此连接结果中一边一定为NULL




*左反连接其实就是左连接加一个限制条件：*

right key should be null.

因此所有右表的列都为null.

**Left anti join = Left join - Inner join**

# Left anti join

**Query:** `Select * from Product p Left anti join Sale s on p.PID = s.ProductID`

Product

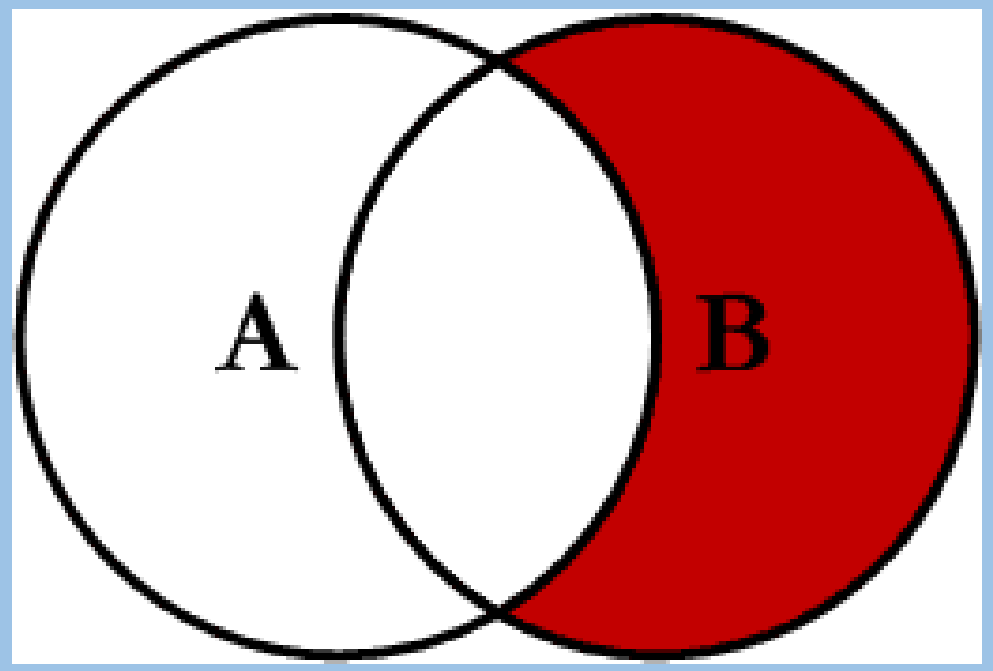
| PID | Pname |
|---|---|
| 1 | Shirt |
| 2 | Pajabi |
| 3 | Lungi |

Sale

| SID | ProductID | Price |
|---|---|---|
| 101 | 1 | 1000 |
| 102 | 2 | 800 |
| 103 | 5 | 400 |
| 104 | 2 | 600 |

| PID | Pname | SID | ProductID | Price |
|---|---|---|---|---|
| 3 | Lungi | null | null | null |

# Right anti join




*右反连接其实就是右连接加一个限制条件：*

left key should be null.

因此所有左表对应列的值都为 null.

**Right anti join = Right join - Inner join**

# Equijoins and Non-Equijoins

- 默认情况下，Impala 要求连接时 ON USING 和WHERE从句中使用等值比较，这种类型称为等值连接。内连接、外连接、全连接和半连接都可以是等值连接。
- Impala 1.2.2 及之后的版本中，也开始支持非等值连接，例如!= 或者<. 这种类型的连接尤其要注意结果集是否会过大。如果你有一个会产生可接受大小集的非等值连接，可以显示地使用 CROSS JOIN，并把额外的比较条件添加到 WHERE从句中

# Semi-joins

- 半连接在实际情况中很少使用。对于左半连接，只会返回左表中对于ON或者WHERE从句在右表中有匹配记录的数据。如果有多条匹配的记录，只会返回一条记录。右半连接恰恰相反。

# Impala 连接的三种方式

- Nested Loop Join
- Sort Merge Join
- Hash Join

# Nested Loop Join

- 从最简单的形式来说，一个nested loops join从其中的一个表取出的行（也叫outer 表）与另外一个表的行（也叫inner 表）进行比较，来寻找符合连接谓词的行。这边的outer 和inner不是指inner join 和 outer join这两个逻辑操作，而是我们重新定义的，其实叫外表（驱动表）和内表（被驱动表）。

- 该算法的伪代码描述如下：

```
for ( outer 表中的每一行R1)
    for ( inner 表中的每一行R2)
if( R1 joins with R2)
return (R1,R2)
```

# Nested Loop Join

- 嵌套循环连接的名字就是从该算法的第二个for(嵌套循环）中得来的。
- 所有的行比较数目，也就是，该算法的成本就相当于outer表与inner表的大小成比例关系。
- 对于被连接的数据子集（外表）较小的情况，嵌套循环连接是个较好的选择。在嵌套循环中，内表被外表驱动，外表返回的每一行都要在内表中检索找到与它匹配的行，因此整个查询返回的结果集不能太大（大于1 万不适合），要把返回子集较小表的作为外表（CBO 默认外表是驱动表），而且在内表的连接字段上一定要有索引。

- 循环嵌套连接支持所有的谓词，包括equijoin（相等谓词） 和不等谓词

- 嵌套循环连接支持的连接类型：
  - Inner join
  - Left outer join
  - Cross join
  - Cross apply and outer apply
  - Left semi-join and left anti-join

# Merge Join

- 与支持任何连接谓词的嵌套循环连接不同，合并连接要求至少一个连接谓词。此外，合并连接的输入必须在连接的键上（key）进行排序。例如，如果我们有一个连接谓词”T1.a = T2.b”，表 T1必须在 T1.a 上排序 和 T2 表必须在 T2.b 上进行排序。
- 合并连接的工作方式是一次同时读取并比较两个已经排好序的输入中的一行。在每一步，我们将从每个输入的下一行进行比较。如果行是相等的，我们就输出连接后的行并继续。如果行不相等，我们放弃两个输入中较小的一个并继续。由于输入已经排好序，我们知道我们正在放弃的行会小于任何输入的所有剩余行，因此，后面也就不可能再进行连接。

- 算法伪代码：

```
get first row R1 from input 1
get first row R2 from input 2
while not at the end of either input
    begin
        if R1 joins with R2  /* 符合谓词条件 */
            begin
                return (R1, R2)
                get next row R2 from input 2
            end
        else if R1 < R2
            get next row R1 from input 1
        else
            get next row R2 from input 2
    end
```

- 与嵌套循环连接不同的是，在嵌套循环连接里面总的成本可能会与输入表中的行数的乘积成正比，而合并连接中，每个表最多被read一次，所以总的成本与输入中的行数的总和成正比。因此，合并连接通常是在更大的输入时是更好的选择。但要求两边输入有序。

# Hash Join

- hash连接分两步执行：构建和探测（build and probe）。在构建阶段，它会从第一个输入里面读取所有的行（Impala中为右表），然后计算equijoin 键的hash值，然后创建一个在内存中的hash表。在探测阶段，它会从第二个输入（左表）读取所有的行，在相同的equijoin键上计算hash值，然后根据hash表进行查找。因为hash函数会导致冲突（两个不同的键值在经过hash计算后会得出相同的hash值），Impala 还需要检查每个潜在的匹配来确保确实符合连接条件。

# Hash Join

- 算法伪代码如下：

```
for each row R1 in the build table
    begin
        calculate hash value on R1 join key(s)
        insert R1 into the appropriate hash bucket
    end
for each row R2 in the probe table
    begin
        calculate hash value on R2 join key(s)
        for each row R1 in the corresponding hash bucket
            if R1 joins with R2
                return (R1, R2)
    end
```

- 注意到，不同于嵌套循环连接及合并连接会立刻开始返回输出行，hash连接会在它的构建输入时，阻止输出。也就是说，在返回任何输出之前它必须读取和处理它的整个构建输入。

- 更进一步讲，不同于其他连接方法，hash连接要求一块内存来存放hash表。也就是说，对某个指定的时间点，能同时执行hash连接的数目就需要有一个限制。

# 内存和溢出

- 在hash连接开始执行之前，Impala会尝试估算它需要多少内存来构建它的hash表。然后，我们会尝试保存这么多的内存，确保hash连接可以成功执行。

- 如果因为我们给了hash连接较少的内存，在这些情况下，hash连接的构建阶段就可能会出现运行内存不足。如果hash连接耗尽了内存，它会开始将总的hash表中的一小部分溢出（spill)到磁盘中。hash连接会跟踪hash表中的哪些部分仍然在内存中，哪些部分已经溢出到磁盘中。当我们从构建表（build table）中读取每一新行时，我们会检查一下是否hash到了内存中或者磁盘上。如果是hash到内存中，则进行正常的hash处理。如果是hash到磁盘上的，我们会将该行写入磁盘。这一耗尽内存和溢出到磁盘的过程可能重复多次，直到构建阶段已经完成为止。

- 我们在探测阶段会进行一个类似的过程。对探测表的每个新行，我们需要检查以查看是否hash到了内存中或者磁盘上。如果是hash到内存部分，我们会对hash表进行探测，生成任何合适的连接的行，并丢弃该行。如果hash到了磁盘部分，我们则将该行写入磁盘。一旦我们完成了对探测表的一次遍历，我们会逐个返回已经溢出的部分，将构建表中的行读回内存，为每一部分重建hash表，接着读取对应的探测部分来完成连接。

- The operator runs in these distinct phases:

1. Consume all build input and partition them. No hash tables are maintained.
2. Construct hash tables from as many partitions as possible.
3. Consume all the probe rows. Rows belonging to partitions that are spilled must be spilled as well.
4. Iterate over the spilled partitions, construct the hash table from the spilled build rows and process the spilled probe rows. If the partition is still too big, repeat steps 1-4, using this spilled partitions build and probe rows as input.

1. Consume all build input and partition them. No hash tables are maintained

```
ConstructBuildSide()
    ProcessBuildInput()
        Create and init PARTITION_FANOUT Partitions.
        ProcessBuildBatch()
    HashTableCtx::EvalAndHashBuild()
    // Choose the proper partition basing the hash generated above
    AppendRow()
        BufferTupleStream::AddRow() or
        AppendRowStreamFull()
            SpillPartitions()
                // Iterate over the partitions and pick the largest partitions to spill
                Partition::Spill()
```

2. Construct hash tables from as many partitions as possible

```
ProcessBuildInput()
    BuildHashTables()
        Partition::BuildHashTable()
    Partition::InsertBatch()
        HashTableCtx::EvalAndHashBuild()
        HashTableCtx::Insert()
Partition::Spill()
```

3. Consume all the probe rows. Rows belonging to partitions that are spilled must be spilled as well.
4. Iterate over the spilled partitions, construct the hash table from the spilled build rows and process the spilled probe rows. If the partition is still too big, repeat steps 1 -4 , using this spilled partitions build and probe rows as input.

```
GetNext()
  while() {
    OutputUnmatchedBuild()
    OutputNullAwareNullProbe()
    OutputNullAwareProbeRows()
    ProcessProbeBatch()
      EvalAndHashProbePrefetchGroup()
        HashTableCtx::EvalAndHashProbe()
        HashTable::PrefetchBucket()
      while(NextProbeRow()) {
        // NextProbeRow()
        //   Case 1: The build partition is in memory
        //     HashTable::FindProbeRow()
        //   Case 2: The build partition is either empty or spilled
        //     Case 1: The build partition is empty
        //     Case 2: The build partition is spilled
        //       Spill the probe row and move to the next row
        ProcessProbeRow()
          ProcessProbeRowXXX()
    }
  NextProbeRowBatch() or
  NextSpilledProbeRowBatch()
  CleanUpHashPartitions()
  PrepareNextPartition()
    Partition::BuildHashTable()
    ProcessBuildInput() // Repartition
}
```

| | Nested Loops Join | Merge Join | Hash Join |
|---|---|---|---|
| **最适合于 ...** | 相对较小的输入，且inner表中用于连接的健上有索引。 | 中等到大型的输入，且在equijoin的键上有索引来提供排序，或者我们在连接后请求了排序。 | 中型到大型的数据仓库的查询。线性扩展的并发执行。 |
| **并发** | 支持大量并发用户 | 多到一的连接，且有索引来提供排序的话，支持大量的并发用户。 | 最适合于少量的并发用户，但确有高吞吐量的场景。 |
| **stop and go (停止再执行)** | No | No | Yes (只在构建输入阶段) |
| **要求Equijoin（也就是需要有相等的谓词）** | No | Yes (除了 full outer join) | Yes |
| **外连接和半连接** | 只支持left 的连接（full outer 连接需要转换） | 所有的连接类型 | 所有的连接类型 |
| **使用内存** | No | No (在排序时，可能需要内存) | Yes |
| **使用tempdb** | No | Yes (只在many-to-many join的情况下) | Yes (如果连接耗光了内存，则可能还会溢出) |
| **需要排序** | No | Yes | No |
| **保留排序** | Yes (仅适用于外部(outer)输入) | Yes | No |

# Impala 实现连接的两种策略

- Broadcast Joins
- Partitioned Hash Joins

# Broadcast Joins

- 广播连接是Impala 最先使用、也是默认的连接策略
- 广播连接中 Impala 将较小的表通过网络分发到所有需要执行该连接的Impala后台进程中。分发完成后，参与连接的Impala进程会根据数据建立哈希表并保存在内存中。然后每个Impala后台进程读取大表在本节点中的数据并使用内存中的哈希表查找匹配的记录。
- 这种方式不需要讲整个大表的数据读入到内存，因此Impala使用1GB的缓存读取大表的数据，一部分一部分读入并进行连接。

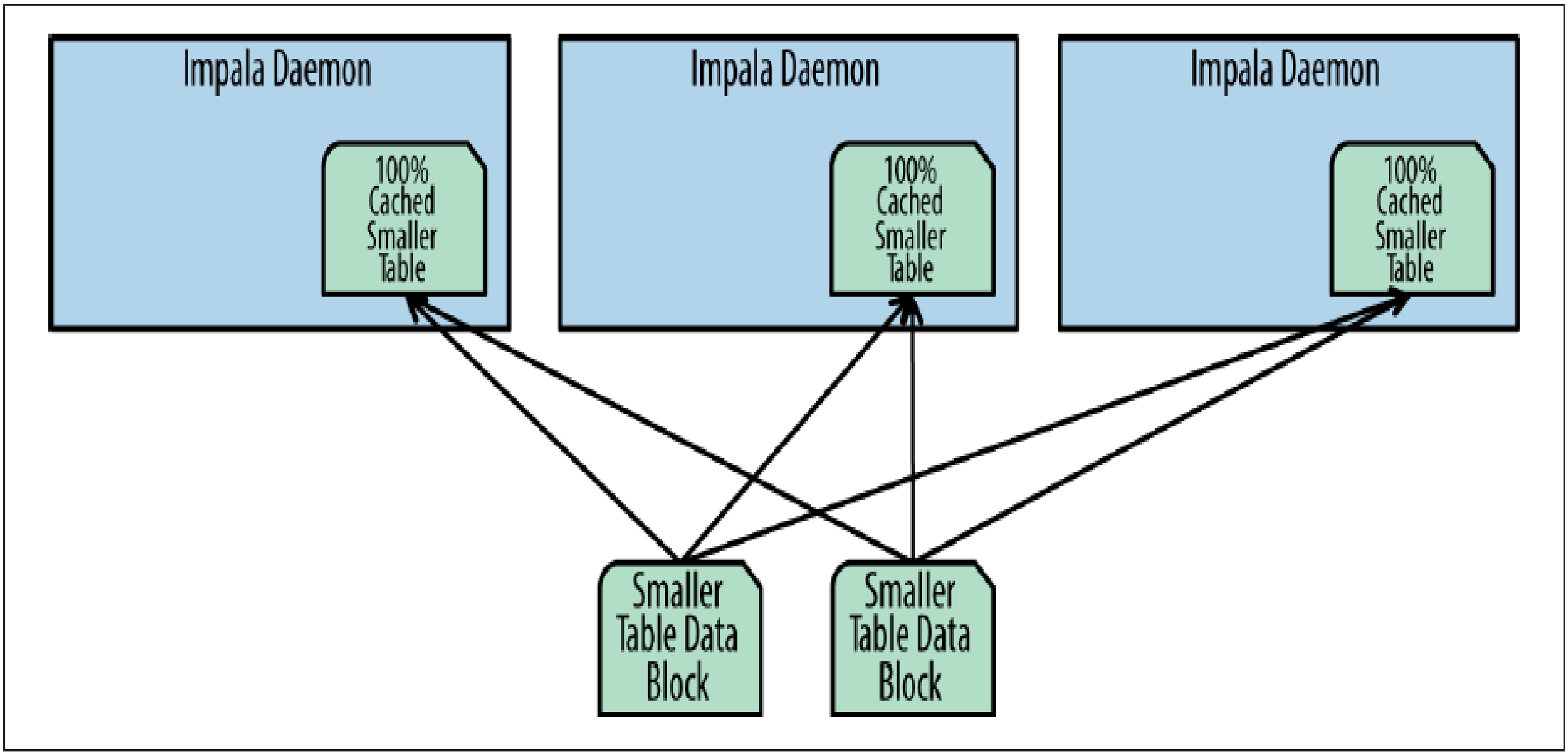



*Figure A-1. Smaller table caching in broadcast joins*

- 小数据集在每个节点中都占用内存。
- 缓存在内存中的数据并不是整个表的数据，而是连接列的哈希值以及查询需要用到的列。
- 小表会分发到所有Impala进程
- Impala 使用基于开销的优化估算表的大小并决定是否进行广播连接、哪个表比较小、哈希表需要多少内存等。

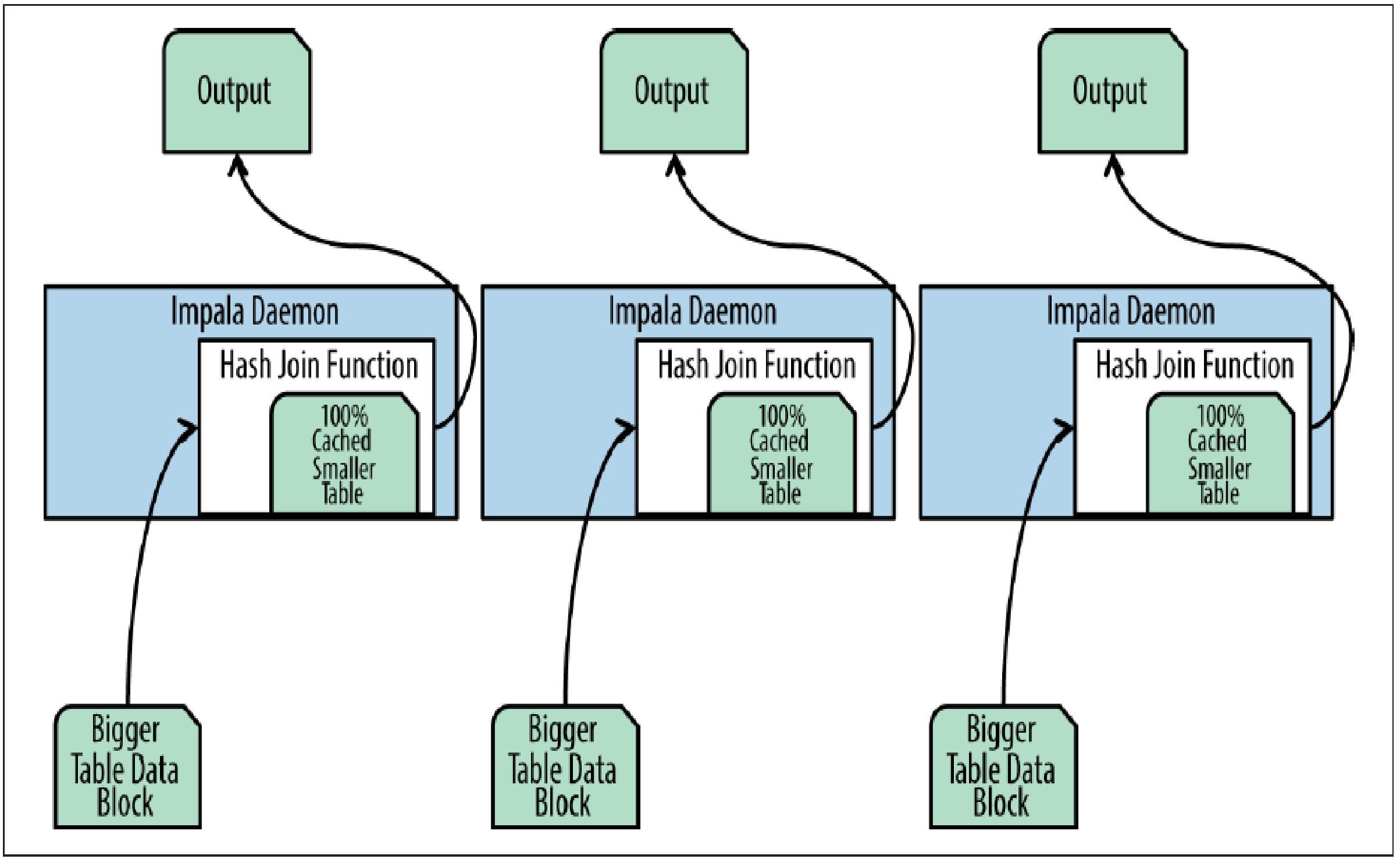



*Figure A-2. Broadcast joins in Impala*

- 小表数据分发并缓存完成后，大表的数据就流式地通过内存中小表的哈希表。每个Impala进程负责大表的一部分数据，扫面读入，并用哈希连接的函数计算值。

- 大表的数据一般由Impala进程从本地磁盘读入从而减少网络开销。由于小表的数据已经缓存在每个节点中，因此在此阶段唯一可能的网络传输就是将结果发送给查询计划中的另一个连接节点。

# Partitioned Hash Joins

- 分区哈希连接需要更多的网络开销，但可以允许大表的连接而不要求整个表的数据都能放到一个节点的内存中。当统计数据显示表太大而无法放到一个节点的内存中或者有查询提示时就会使用分区哈希连接。
- 进行分区哈希连接时（也称为shuffle join），每个Impala进程读取两个表的本地数据，使用一个哈希函数进行分区并把每个分区分发到不同的Impala进程。

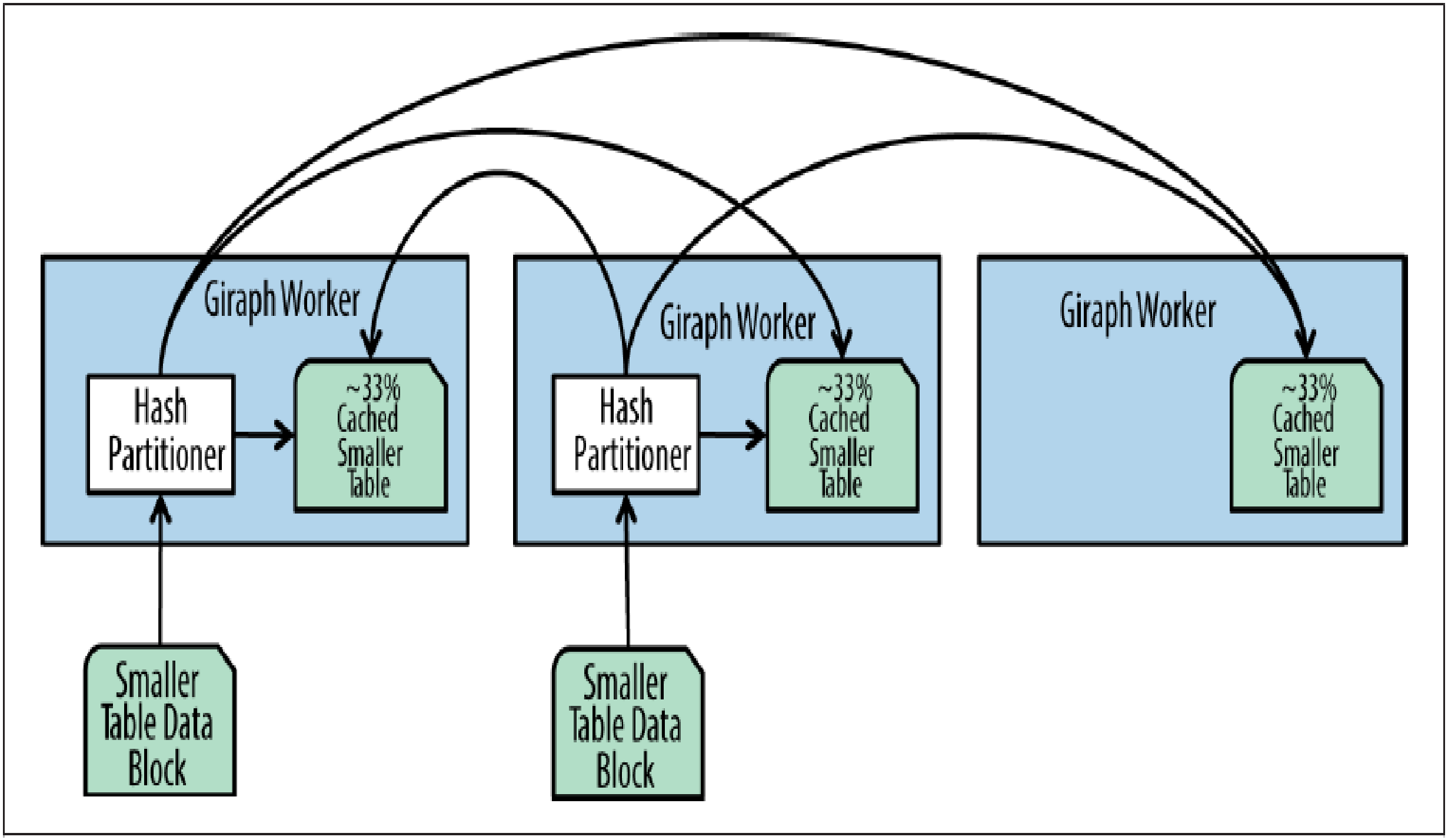



*Figure A-3. Smaller table caching in partitioned hash joins*

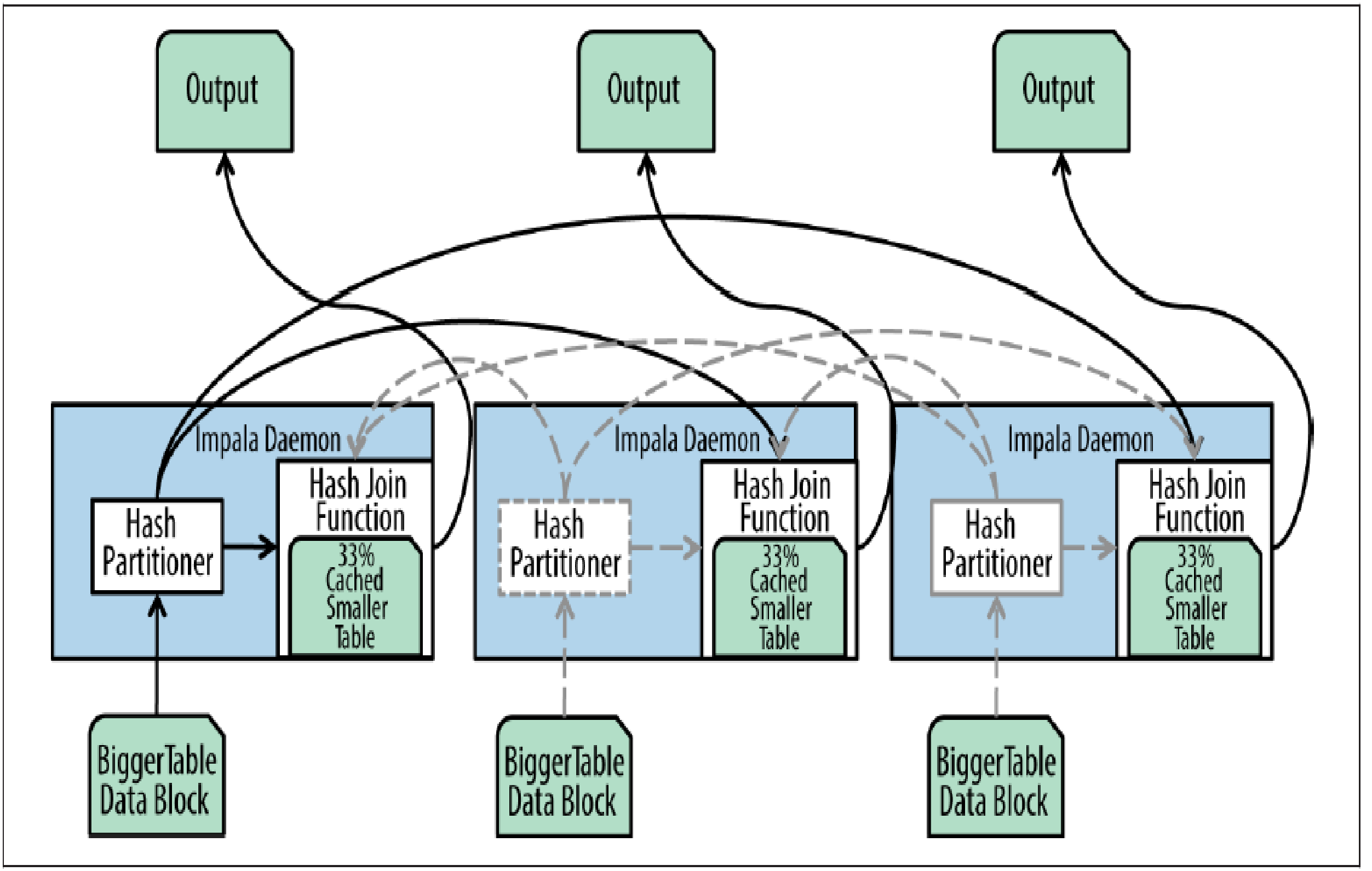



*Figure A-4. Partitioned hash joins in Impala*

- 正如上图所示，大表的数据也通过相同的哈希函数就行分区并把分区发送能和小表相应数据进行连接的结点。注意，和广播连接不同的是，广播连接只有小表的数据需要通过网络分发，而分区哈希连接需要通过网络分发大表和小表的数据，因此需要更高的网络开销。

- 总而言之，Impala有两种连接策略：广播连接，需要更多的内存并只适用于大小表连接。分区连接，需要更多的网络资源，性能比较低，但是能进行大表之间的连接。

- Questions & Thanks
- Tb1(id int, name string)
- Tb2(ID int, NAME string)
- Reference：
- Nested Loop Join
- Merge Join
- Hash Join
- Join Summary
- Hadoop Application Architectures
- Impala Docs