GF017006 1000r0r010 010701D阳权901049低 D-0108

09101970106m00( 0:0t0 010ze010Ye-o53 G -00

07101

0 1 0 1 0 0 日 0 1 0 6 1 0GtchureQ0.0 顶 0Dare¹o

4c00100160100110C!010

**第6章** **流数据处理**



**6.1** **流数据处理应用**

有一类数据密集型应用，数据快速到达，转瞬即逝，需要及时进行处理。

这些应用来自不同的领域，包括网络监控 (Network Monitoring)、电信数据管理 (Telecommunication Data Management)、工业制造 (Manufacturing) 、 传感器网络 (Sen- sor Network)、电子商务 (Electronic commerce)、量化交易 (Algorithm Trading) 等。

比如，(1)在网络监控应用中，数据处理系统需要对网络上流过的网络数据包进行分 析，以便进行人侵检测、实施防火墙规则、对可疑数据包进行拦截等。(2)在传感器网络 监控领域，在物理世界大范围部署的传感器，产生大量的数据，需要及时进行合并 (Combine) 和分析 (Analyze) 。(3) 量化交易程序(自动执行股票/期货交易的程序)持 续监控价格数据流和新闻数据流，对数据进行及时分析，以指导买/卖决策。

**6.2** **流式处理和批处理的区别**

流式处理 (Stream Processing) 是针对批处理 (Batch Processing) 来讲的，即它们是 两种截然不同的数据处理模式，具有不同的特点，适用于不同的应用场合。不能简单地认 为其中一种数据处理模式优于另外一种数据处理模式。

对于批处理来讲，首先数据被不断地采集，保存到数据库中(不一定是关系数据库， 可以是HBase 或者Hive 数据库),然后进行分析处理(包括SQL 查询)。批处理适用于对 大量数据 (High Volume) 进行处理的场合。人们需要等到整个分析处理任务完成，才能 获得最终结果。由于需要处理的数据集大小以及计算机系统的计算能力的差异，整个过程 有时需要耗费相当长的时间，即获得最终分析处理结果的延迟较大。批处理是最通用的数 据处理模式。传统的关系数据库系统、Hadoop 以及Spark 大数据处理平台等，都采用了



这样的数据处理模式，或者以该处理模式为主。由于需要完整地保存整个数据集，并且在 上面进行分析处理，比起流式数据处理系统，人们需要投人更多的硬件资源。

批处理系统可以存取已经人库的所有数据(虽然某个批处理作业有可能仅仅处理数据 的一个子集),人们可以对数据进行复杂深入的分析，分析处理的延迟以分钟或者小时计。 批处理作业 (Job) 一旦提交，则一直运行，直到获得最终结果或者失败退出，无须用户 人工干预。这些作业在整个数据集或者某个子集上执行一系列的分析，产生分析结果。在 性能方面，批处理系统主要考虑系统的吞吐能力，而不是单个分析处理任务的处理延迟 (Latency)。

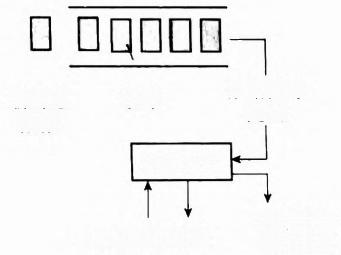
在流式数据处理模式里，数据持续到达，系统及时处理新到达的数据，并不断产生输 出。处理过的数据一般丢弃掉，当然也可以保存起来。流式数据处理模式强调数据处理的 速度 (Velocity) 。 部分原因是数据产生的速度很快，需要及时进行处理。由于流式数据处 理系统能够对新到达的数据进行及时的处理，所以它能够给决策者提供最新的事物发展变 化的趋势.以便对突发事件进行及时响应，调整应对措施。

在这里，需要强调实时 (Real-time) 数据处理系统和流式数据处理系统是两个不同 的概念，虽然在很多文献中人们把流式数据处理系统冠之以实时数据处理系统的帽子。 对于批量数据处理系统，当分析处理的响应速度快到一定的程度，这样的系统可以称为 实时系统(响应时间达到毫秒级)或者交互式系统(响应时间达到秒级或者亚秒级)。 提高批量数据处理系统数据分析和处理的速度，有多种技术手段，比如把整个数据集保 存到内存中(采用内存数据库)、采用高速硬盘保存数据以保证快速存取、利用索引技 术快速地定位相关数据避免无效的数据扫描、通过物化视图实现汇总数据的预先计算以 加快查询等。

流式数据处理模式对每个新到达的数据元素 (Data Element) 或者一个比较小的时间 窗口内的数据元素进行计算。数据上的计算或者分析处理相对来讲是比较简单的，不是复 杂的分析处理。完成分析处理的时间，需要达到实时或者接近实时 (Near Real time) 的 响应时间要求。不同数据元素上的分析处理，相互之间是独立的。流式数据处理系统一般 以异步方式实现，上游数据源不断向流式数据处理系统注入数据.流式数据处理系统要尽 快完成数据的处理。上游数据源无须等待下游流式数据处理系统处理完毕。

在一个电子商务网站，把每月(周、天)所有的交易记录保存起来，然后在固定的时 间点，比如月末(周末、日终)进行一些SQL 查询分析，这种处理模式就是批处理模式。 当然，在系统不断累积交易记录的过程中，比如在某天的中午，管理者想看一下某个商品 最近一段时间的销售量，他临时发起一个SQL 查询，这个查询称为即席查询 (Ad-Hoc Query), 即席查询也采用批处理的模式进行处理。Hadoop 系统突破传统关系数据库的扩 展性局限，能够管理很多年的交易数据，并且提供 MapReduce 等计算模型，对大数据进 行分析处理。

如果我们需要对每个交易记录进行监控。必要时执行某种商业逻辑 (Business Log- ic) 。 比如，购买额度超过一定限额，则给一个优惠的折扣等，需要对每个新的交易记录及 时进行处理，相关处理逻辑仅仅依赖于当前的交易记录，和其他交易记录相互独立、没有 关系，这种处理模式就是典型的流式处理。和上述把所有的交易记录保存起来，然后进行 处理的批处理模式是完全不同的。每个交易记录需要及时地进行处理，响应时间要求达到 毫秒级，流式数据处理就是这样戴上实时处理的帽子的。数据处理模式见图6-1。



缓冲

查询处理

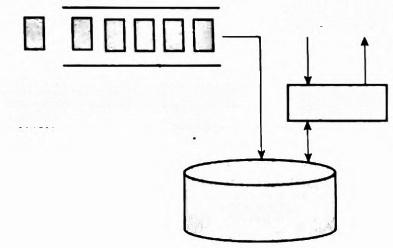
数据丢弃 或者归档

数据及时 处理

**新到达** **数据**

持续查询结果

(a) 流式数据处理模式



查询 结果

数据首先 入库

数据库

新到达 数据

**查询处理**

缓冲

(b) 批处理模式

**图6** **-** **1** **数据处理模式**

**6.3** **流数据模型**

在上述实际应用中，把持续到达的数据装载到数据库中，然后进行分析处理是不合适 的。传统的数据库(比如关系数据库),没有为单个数据元素(单条记录)的快速、持续 的装载进行设计和优化。传统数据库也不支持持续查询 (Continuous Query),这样的查 询是数据流应用的典型负载。此外，在快速到达的数据流上，由于处理时间的紧迫性，有 时候只能计算一些近似结果 (Approximation)。 数据到达的速率不断变化，流式数据处理 系统应该能够根据当前软硬件资源富足情况和数据的特点，调整查询执行计划 (Adaptivi- ty)。这也是传统数据库系统所欠缺的。传统数据库执行经过编译的执行计划，执行过程 中一般无须调整执行计划，最后获得一个精确的结果。

为了对快速持续到达的数据流进行有效的处理，支持上层应用程序的要求，需要新的 数据模型和查询处理办法。

在流数据模型 (Stream Data Model) 中将要进行处理的数据，从一个或者多个上游数 据源持续不断地到达，而不是从保存在磁盘或者内存中的数据源进行随机存取。流数据模 型和传统的关系模型 (Relational Model) 有几个重要的区别：

(1)数据流的数据元素持续到达。

(2)流数据处理系统不能控制数据元素到达的顺序。

(3)数据流有可能是无限的，或者说数据流的大小是无限大 (Infinite)。

(4)数据流的一个数据元素被处理后，可以丢弃或者归档 (Archived), 一般不容易 再次提取，除非目前该数据元素还在内存中。能够保存在内存中的数据元素，相对于整个 数据流来讲是极少量的数据。

需要注意的是，对数据流进行处理，并不排斥保存在传统关系数据库中的数据库表的 参与。有时候对数据流的查询，需要在数据流和已经保存起来的关系数据库表之间进行连 接 (Join) 操作。在这种情况下， 一般假定这些已经保存起来的数据库表的数据是静态 的，无须进行更新操作。

在关系数据库的数据模型里，数据以元组(记录)的形式组织在关系表中 (Relation- al Table)。我们可以对这些表格里的元组及其属性进行任意的增加、删除、修改。在流数

据模型中，数据流可以看作只允许进行元组添加操作 (Append Only) 的关系表。对应关 系数据库的SQL 查询语言，在数据流上，我们可以使用经过扩展的 SQL 语言，进行数据 流的查询。

**6.4** **流数据上的查询实例**

数据流上的查询和传统数据库上的查询(比如关系数据库上的SQL 查询)有很多共 同的特点，但是两者有两个重要的区别。

第一个区别是一次性查询 (One Time Query) 和持续查询 (Continuous Query) 的区 别。一次性查询(比如关系数据库的 SQL 查询)指的是在数据集的某个时刻的快照 (Point in Time Snapshot) 上执行的查询，对数据进行分析，获得结果后返回给用户。持 续查询则是在一系列持续到达的数据流的数据元素上执行的查询，它产生一系列结果。这 些结果是根据查询不断执行时不断看到的新数据而产生的。持续查询的结果，根据新数据 不断到达进行更新，这些结果可以保存起来，或者作为一个新的数据流。

第二个区别是预定义查询 (Predefined Query) 和即席查询 (Ad-Hoc Query) 对系统 的影响。预定义查询是数据准备好之前已经提交给流数据处理系统的查询。预定义查询一 般是持续查询，当然我们也可以预先定义一些一次性查询。即席查询则是在数据流的数据 开始流动起来，数据不断到达的时候，才提交给流数据处理系统的。即席查询可以是一次 性查询，也可以是持续查询。即席查询使得系统的设计和实现复杂化了。由于事先不知道 查询的具体形式，所以没有办法进行深入的查询优化，比如在查询间找出公共子表达式 (Common Subexpression,即不同查询的公用的查询条件)等。对即席查询来讲，要获得 正确的结果，有时候需要参考早先到达的数据元素，这些数据元素有可能已经丢弃，于是 问题就变得复杂了。传统数据库系统(比如关系数据库)对预定义查询和即席查询的处 理，没有上述复杂情况需要考虑。

**SQL 语言是关系数据库的标准查询语言。它简单易学，功能强大，为广大开发人员** **甚至普通用户所熟悉。流数据管理系统，比如** **Sandford Stream Data Manager, 一般通过** **提供面向流数据处理的一些原语** **(Primitive) 扩展** **SQL 语言，支持用户通过熟悉的SQL 查询语言操作数据流。**

扩展的原语主要提供时间窗口 (Time Windows) 的定义办法。Sandford Stream Data Manager 通过扩展SQL-99, 提供对时间窗口的支持，包括物理时间窗口 (Physical Win- dow) 和逻辑时间窗口 (Logical Window)。物理时间窗口通过ROWS 关键字指定，比如 ROWS 50 PRECEDING, 表示数据流里包括当前记录的前50个记录，逻辑时间窗口通过 RANGE 关键字指定，比如RANGE15 MINUTES PRECEDING, 表示当前时间戳以及之 前15分钟之内的所有记录。

**Sandford Stream Data Manager还通过PARTITION 关键字，对数据流的记录进行划** **分。在包含客户** **ID(customer\_id) 、** **呼叫类型** **(type) 、** **呼叫时长** **(minutes) 、** **时间戳** **(timestamp)** **等属性的电话呼叫记录数据流上** **(Telephone** **Call** **Records),通过如下查询** **语句，可以计算每个用户，最近10个长途呼叫的平均呼叫时长。**



|  |
| --- |
| **SelectAvg(**S.minutes)  **From** Calls S  [ **Partition By** S.customer\_id **Rows** 10 **Preceding Where** S.type='Long Distance'] # 按 照customer\_id 划分数据流，提取每个客户最近10个长途电话呼叫  #然后计算平均通话时长 |

下面我们通过几个实例来介绍流数据处理系统中的持续查询。在一个假想的互联网接 入供应商 (Internet Service Provider,ISP) 网络上，网络管理员需要监控网络使用状况。 于是他在若干网络链路上持续地采集数据包并且进行分析。现在，网络管理员需要深人了 解两个特别的链路.一个是链路C, 连接了ISP 网络和某个用户，另一个是链路B, 连接 ISP 的骨干网络的两台路由器。

通过对网络数据包的快速提取和初步解析，形成一系列数据元素(即记录或者元组) 构成的数据流。每个数据元素包含5个数据项(从网络数据包的首部 Header 分析出来), 分别是：(1)src. 为发送方的IP 地址；(2) dest, 为接收方的 IP 地址；(3) len, 为数据 包的大小(长度);(4) time, 为获取数据包的时间；(5)id, 为发送方赋予数据包的编 号，用于接收方识别每个数据包 (Packet)。

**1.** **查询实例Query1**

针对上述数据流的第一个查询是一个持续查询。该查询语句针对链路B, 在每分钟的 间隔内，计算网络流量负载，当负载超过一定的阈值时通知网络管理员。其中， notify\_ operator 是一个用户自定义函数，其功能是把某分钟里的流量负载通知管理员。 get\_mi- nute是一个系统函数，其功能是获得时间字段的分钟数。

|  |
| --- |
| #当某分钟之内的累计流量超过阈值，通知网络管理员  **Select** notify\_operator(sum(len))  **From** B# 分析数据流B,即链路B  **Group By** get\_minute(time)# 按照每分钟对记录分组，以便累计每分钟的流量 **Having Sum(**len)>t# 当某分钟之内的累计流量超过阈值，通知网络管理员 |

当然这个功能可以使用传统关系数据库里的触发器 (Trigger) 来实现。在数据流上 直接实现查询，具有更大的优势。比如，当链路B的吞吐量非常高时，我们可以通过在数 据流上进行随机采样，获得查询的近似结果，这是传统数据库的触发器机制所不支持的。

**2.** **查询实例** **Query2**

查询Q2 的 SQL 语句如下，这是一个嵌套查询。

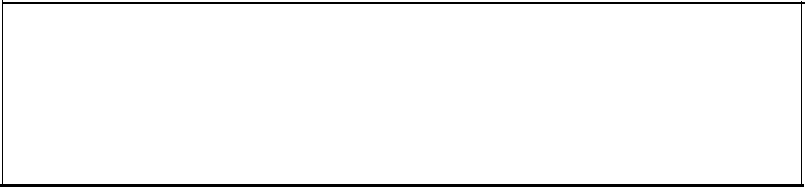
**Select** src,dest,flowid,sum(len)As flowlen

**From(Select** src,dest,len,time

**From B**

**Order By** time

)

**Group By** src,dest,getflowid(src,dest,time)As

flowid

这个查询，在链路 B 上把各个 flow 隔离开来，然后计算每个flow 的流量。flow 在这 里定义为从某个特定的源 IP 发送到特定目标 IP, 按照时间进行分组的一系列数据包。

getflowid是一个用户自定义函数，参数是源 IP 地址、目标 IP 地址以及数据包的时间

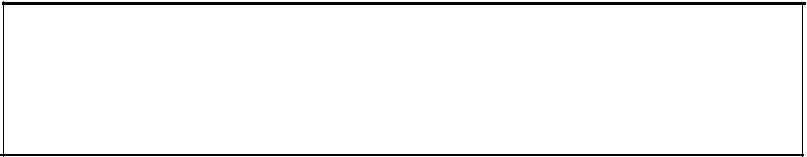


戳，它返回数据包隶属的flow ID。内重查询利用Order By语句，保证数据流的各个记录 按照时间戳排序。外围查询则通过Group By 语句，分组计算每个flow 的流量累计值。

Group By语句和Order By语句给流数据上的查询处理带来了新的挑战，因为它们要 求查询执行计划看到该看到的所有输入数据以后才能正确进行排序或者分组，而不是看到 一条记录就可以产生输出。

**3.** **查询实例** **Query3**

查询3的SQL 语句如下，它包含两个独立的子查询。它的功能是计算出骨干链路 B 上的流量有多大比例是由链路C (连接特定客户)造成的。



**(Select** count(\*) **From** C,B

**Where** C.src =B.src **And** C.dest=B.dest **And** C.id=B.id)

/

**(Select** count(\* **)From** B)

在网络开始发生堵塞时，网络管理员可以提交这个即席的持续查询 (Ad-Hoc Contin- uous Query), 以便确定骨干链路的堵塞是不是由某条客户链路的流量造成的。

第一个子查询，对C 和 B 两个数据流进行连接操作，计算在两个链路上的公共数据包 (数量)。即从链路C 过来，流过链路B 的数据包，具有相同的源IP 地址、目标IP 地址以 及应用程序 ID。第二个子查询，计算链路B 上的所有数据包(数量)。

对两个数据流进行Join 操作，如果中间结果比较大，有可能造成存储空间的需求过 大。如果同一个数据包到达链路C 和链路B 之间的延迟没有一个固定的上界，就会发生这 种情况。由于我们知道两个数据流上能够连接的元组(即同一个数据包),它们的时间间 隔是在一定范围之内的，我们可以限定查询处理所需的存储空间(内存)。

否则，由于计算一个准确的结果代价太大，可以考虑计算一个近似结果。一种办法是 建立两个数据流的梗概 (Synopsis), 基于这两个梗概进行连接操作，计算数据包数量的 一个近似值，这样会消耗较少的内存。

**4.** **查询实例Query4**

最后一个实例是一个持续查询。它可以监控每对源IP 到目标IP 在骨干链路B 上的流 量，并且把 TOP5% 的<Src IP,dest IP>对，查找出来。

**With Load As**

**(Select** src,dest,sum(len)As traffic) **From B**

**Group By** src,dest)# 建立一个视图

**Select** src.dest,traffic

**From** Load As L1

**Where(Select** count(\*)

**From** Load As L2

**Where** L2.traffic<L1.traffic)

>

**(Select** 0.95\*count(\* **)From** Load) **Order By** traffic

第一个 SQL 语句建立数据流上的一个视图，所有的<Src IP,dest IP>对，通过

Group By进行分组，即在视图中，每对<Src IP,dest IP>只出现一次。

接着的SQL 语句，包含三个子查询。Where 条件中的第一个Select 子查询，把Load 表里流量小于L1.traffic 的记录数量计算出来。Where 条件中的第二个Select 子查询，算 出所有的<Src IP,dest IP>对的数量的95%。外层查询基于这个条件，把流量达到TOP 5 % 的<Src IP,dest IP>对查找出来，并且按照流量多少进行降序排列。

**6.5** **流数据处理系统的查询处理**

**6.5.1 内存需求** **(Memory Requirement)**

大部分数据流是无法预知最终大小的，或者说数据流的大小(记录数量)可能是无限 的。在这种情况下，如果要在数据流上计算一个准确的结果(比如累计数),需要的存储 空间将无法预知，有可能超过可用的内存。

虽然人们为传统关系数据库设计了基于外存(磁盘的)算法，但是这些算法不适用于 流数据处理系统。这些算法不支持持续查询，由于需要存取磁盘，其响应时间太慢了。

在某些流数据处理应用中，新数据以极高的速率到达，以至于老数据都还没来得及处 理。我们需要尽量降低处理每个数据元素的时间，每个数据元素要处理得够快，否则流数 据的处理跟不上数据到达的速度。为了达到高速的数据处理，流数据处理系统一般优先采 用基于内存的数据处理算法，无须存取磁盘。

**6.5.2 近似查询结果** **(Approximate Query Answering)**

在内存容量有限的情况下，获得一个准确的结果是不太可能的。正好在很多应用场 合，我们无须一个准确的答案。近似的查询结果，只要足够好，可以作为准确结果的 替 代 。

在流数据处理领域，人们为数据流上的查询研究了一系列数据缩减 (Data Reduction) 或者摘要 (SummaryConstruction) 构建技术，具体包括数据轮廓 (Sketche) 、 随机采样 (Random Sampling)、直方图 (Histogram) 、 小波变换 (Wavelet) 等。基于这些摘要 (Summarization) 数据结构，实现了计算近似结果的方法，包括聚集查询 (Aggregate Query) 和连接查询 (Join Query)。

**6.5.3 滑动窗口** **(Sliding Window)**

从数据流上产生近似查询结果的一种技术是滑动窗口及其之上的查询处理技术。滑动 窗口上的查询处理指的是在数据流的最近数据元素(记录)上执行查询，而不是在数据流 的所有历史记录上执行查询。比如，进行查询处理时，仅仅存取数据流上最近一天的数据 元素，一天前的数据元素则丢弃掉。

滑动窗口及其之上的查询处理具有若干优势。首先，滑动窗口的语义清楚，在滑动窗 口上的计算结果易于理解。比如，上文提到的每个用户最近10个长途呼叫的平均呼叫时 长的查询就可以在一个滑动窗口上计算。其次，在滑动窗口上计算的结果是确定的 (De- terministic), 在数据流的采样上进行计算，则只能获得一个近似结果。最后，滑动窗口强 调最近的数据，即最近的数据权重更高。或者换句话说，最近的数据比老数据更加重要， 这一点在具体应用中具有实际的意义。比如，在上述网络链路流量监控应用中，对最近网 络流量的观察有助于我们了解当前网络运行状况。

为了实现数据流上基于滑动窗口的查询，一般需要在查询语言里，增加滑动窗口的定 义方法(一般是对SQL 语言进行扩充)。滑动窗口的大小是任意的，当滑动窗口过大时， 窗口里的数据元素(记录)过多，不能缓存在内存里，需要利用磁盘保存部分数据，这将 增加处理的延迟，研究人员在研究利用有限的内存实现近似计算的算法。

**6.5.4 查询数据流的历史数据** **(Referencing Past Data)**

在标准的流数据处理模式中，当某个数据元素处理结束后，将无法再访问到。这就意 味着某些数据被丢弃以后，用户发起即席查询 (Ad-Hoc Query), 将无法获得准确的 结 果 。

针对这个问题，最简单的办法是规定即席查询只能参考它提交以后到达的新数据，之 前的历史数据直接忽略掉。这种办法简单粗暴，但是在很多应用中，这样的规定是可以接 受 的 。

另外一个办法则稍微复杂一些，它允许新提交的即席查询参考历史数据。历史数据不 是原原本本地保存起来而是保存一个摘要 (Summary), 是数据的一个梗概 (Synopsis) 或者聚集汇总 (Aggregate)。 这些数据摘要有助于为未来的即席查询计算一个近似的结 果。这种办法需要考虑到系统支持什么类型的查询，然后利用内存资源，维护一个数据摘 要，最大限度地支持这些类型的查询。

**6.5.5** **多查询优化与查询计划的适应性**

在流数据处理系统中，大多数的查询是长时间运行的持续查询。系统同时运行大量的 查询，可以通过多查询优化 (Multi Query Optimization) 技术，提高查询处理的性能。由 于系统不断有新的即席查询提交上来，为一组查询寻找最佳的执行计划，需要在线 (On- line) 进行优化决策。

此外，即席查询带来一个问题，就是查询计划的适应性 (Adaptivity in Query Plan)。 比如 Eddy 流数据处理系统，提出了柔性查询计划的概念。它的查询执行器可以根据数据 到达的速率(以及数据流的其他特征)适时调整执行计划，即持续调整执行计划里不同操 作符的顺序 (Reorder Operators Continuously)。

**6.5.6** **堵塞操作**

堵塞操作 (Blocking(perator) 是这样的操作，它需要看到所有的输入数据以后才能开 始产生输出结果。排序就是堵塞操作的一个实例，此外，包括 Sum,Count,Min,Max,



Avg等聚集操作也是堵塞操作，因为只有看到所有的输入数据，才能开始产生输出。

数据流上的堵塞操作带来了严峻的挑战。因为持续的数据流有可能是无限的 (Infi- nite), 堵塞操作将永远不会看到整个数据集，于是无法产生任何输出结果。

像排序、聚集等操作是通用的一些数据操作，我们不能一弃了之。让流数据处理系统 有效处理排序、聚集等操作是一个严峻的挑战。

其中的一种技术称为标点技术 (Punctuation) 。 标点就是一个断言 (Assertion), 它 规定在剩下的数据流数据中，什么数据可以出现，什么数据不可以出现。标点和数据元素 交织在一块，被插入在数据流的不同位置上，帮助数据流上的操作做出决策。

比如，一个数据流有一个字段daynumber。 如果数据流里有一个标点，标点的断言是 “daynumber≥10”, 它表示在数据流里，剩下的数据元素的daynumber 属性都大于10。那 么遇到这样的标点以后，针对daynumber 进行分组聚集的操作就可以把关于daynumber 小于10的聚集结果输出了。连接操作符则可以把先前保存的涉及daynumber 小于10的元 组的连接结果进行输出，并且把内存缓冲区释放掉，减少内存开销。

**6.5.7 数据流里的时间戳** **(Timestamps in Stream)**

滑动窗口是基于数据流元素的时间戳 (Timestamp) 或者顺序号 (Sequence Number) 属性进行定义的。对于来自一个数据流的数据来讲，时间戳一般不存在歧义。但是在一些 场合，我们必须对时间戳给予关注，原因是：(1)如果滑动窗口是在从多个数据流上产生 的组合元组上定义的(比如一个Join 操作，连接了两个数据流 S1/S2), 如果来自两个数 据流的元素的时间戳数值不一样，那么两个元组连接产生的组合元组应该赋予什么时间戳 是一个问题。(2)当若干分布式的数据流构成一个逻辑数据流时，以及在分布式的传感器 网络上，比较不同数据流的数据元素的时间戳具有实际业务意义。

时间戳分两种，分别是隐式时间戳 (Implicit Timestamp), 系统为每个元组增加一个 特殊的表示顺序的字段，以及显式时间戳 (Explicit Timestamp), 数据流的每个元组的某 个属性被指定为时间戳。当数据流的一个元组描述了客观世界里特定时间发生的一个事 件，一般我们在数据流里使用显式时间戳，因为它不仅决定了各个事件发生的先后，而且 确定某个事件发生的时间，对于表示元组的实际意义是重要的一个属性。

当我们采用显式时间戳时，由于设备传输方式等原因，元组到达的顺序可能不是按照时 间戳的顺序，也就是有些具有较晚时间戳的元组先到达，有些具有较早时间戳的元组后到 达。元组乱序到达使得我们很难定义在显式时间戳上的滑动窗口计算，以及其他基于元组顺 序的数据流处理。这个问题不是太严重，因为一般来讲，每个数据流乱序到达的数据元素 (元组)基本上是有序的，只有少量的扰动(少量元组出现乱序情况),只要我们对元组进行 适当缓冲，就可以把乱序问题纠正过来，获得有序的数据流的元组序列。

对于上文提到的在两个数据流S1/S2 上进行Join 操作产生的元组，如何赋予其时间 戳的问题.有两种解决策略。第一个策略是 “Best Effort”,Join操作产生的每个元组被赋 予该元组产生时刻的时间戳。这种策略的优势是易于实现，它最大的缺点是在Join 操作产 生的元组上，已经不能精确定义确定性的滑动窗口 (Deterministic Sliding Window),因 为元组丢失了先前的时间戳及其语义，获得的新时间戳是Join操作的时间戳。第二个策略 允许用户通过查询定义如何把时间戳赋予Join 操作产生的元组。为了简单起见， 一般采用 这样的做法，如果数据流连接的顺序是S1/S2, 那么它表示 S1 的优先级高于S2, 于 是

Join操作产生的元组的时间戳，被赋予具有更高优先级的原始数据流的元组所具有的时间 戳。比如来自S1 、时间戳为“2011-01-0112:00:00”的元组和来自 S2 、时间戳为 “2011-01-0112:01:00”的元组进行连接所产生的元组，时间戳是“2011-01-0112: 00:00”。

**6.5.8** **批处理、采样、梗概**

处理近似查询的一种策略是使用某种可以增量式维护的数据结构，利用这个结构获得 数据流查询的结果。 一般来讲，这个数据结构包含两个接口，分别是update(Tuple) 和 computeAnswer() 。update(Tuple) 处理数据流新来的一个元组，computeAnswer()

利用系统当前维护的数据结构计算查询结果。如果这两个操作的执行速度相对于数据流的 数据元素到达的速率足够快，那是最好不过了，这样就可以及时处理数据流的各个数据 元素。

**1. 批量处理** **(Batch Processing)**

实际情况往往不是这样的。第一种情形是，update 操作足够快，但是computeAnswer 操作很慢，跟不上数据流数据到达的速率。最自然的办法是以批处理方式处理数据，也就 是新来的数据元素先缓存起来，在资源允许的情况下，定期计算查询的结果。查询结果不 是根据最新的数据进行计算的(最新数据缓存起来，有待下一次进行成批处理),而是有 一定的时间延迟。这种计算方法获得的查询结果是准确的，只不过它是最近的准确结果而 不是当前的准确结果。也就是说，因为使用批量处理，牺牲了结果的及时性，但是跟上了 数据流新数据到达的速率。

**2. 采样** **(Sampling)**

在第二种情况下，computeAnswer 操作足够快，但是 update 操作很慢，不足以及时 处理新到达的数据。由于数据到达实在太快，没有必要利用所有的数据来计算查询的结 果。我们可以忽略一部分元组，在数据流上进行采样，在采样上而不是整个数据流上计算 查询的结果。

**3. 梗概** **(Synopsis)**

我们希望有某种数据结构，既支持快速的 update 操作，又支持快速的computeAn- swer 操作，能够及时处理数据流新到达的数据。对于很多数据流上的查询，根本就不存 在两者兼得的数据结构。于是人们设计一种近似数据结构，它是数据流的一个梗概 (Syn- opsis or Sketch)。梗概是一个比较小的数据结构，它能够把每个元素的处理代价保持到最 低水平，从而使得流数据处理系统能够赶上数据到达的速度。梗概技术的细节，请参考下 一节内容。

**6.6** **查询处理的基础算法**

**6.6.1** **随机采样**

数据上的随机采样 (Random Sample) 可以看作一种摘要式的数据结构 (Summary

**Structure), 它包含了整个数据集的基本特征。在随机采样上，我们还可以建立各种梗概**

(Synopsis)。

人们研发了各种采样方法，其中分层采样 (Stratified Sampling) 方法首先按照对观察 指标影响较大的某种特征，将总体分为若干类别，再从每一类别内随机抽取一定数量的样 本，合起来组成一个样本。这种采样方法的样本代表性好，抽样误差小，可以替代均匀采 样 (Uniform Sampling)。

除此之外，蓄水池采样 (Reservoir Sampling) 方法.只需对数据进行一遍扫描，特 别适合于数据流的采样。蓄水池采样的基本原理是，首先建立一个数组，将数据流里前k 个数保存在数组中，即所谓的蓄水池。对于第 n 个数据元素(元组)An, 以 k/n 的概率取 A 并以1/k 的概率随机替换蓄水池中的某个元素，如果没有发生替换，则蓄水池数组元素 不变，依此类推处理新到达的其他各个元素。该算法可以保证取到数据的随机性。

**6.6.2 梗概技术** **(Sketch Technique)**

梗概 (Sketch) 技术是在数据流上使用少量的内存，建立一个摘要结构。这个摘要结 构可以用于特定查询的近似结果的估计。梗概技术能够解决数据流上的很多问题。比如估 计数据集的二阶矩的大小、估计数据集自连接 (Self Join) 的大小、获得数据集中热门元 素的列表等。

我们通过例子了解 K 阶矩的计算和应用。令m;表示一个数据集中元素i 的个数，数 据集合的K 阶矩Fk=∑m,K 。Fk 捕抓了数据集中数值分布的特点。 F。是序列中不同值 (Distinct Value) 的个数，F₁ 是序列的长度，F₂ 为自连接的大小 (Self Join Size)。F₂也可 以看作向量的L₂ 范数，即向量的第i维分量为m;。

假设Z 是一个初始值为0的计数器， H 是一个能够将数据流中各元素均匀映射到 {-1,1}的哈希函数。对于在时刻t 到达的数据元素a, 按照Z=Z+H(a,) 修改计数器 Z 值。可以证明 E(Z)=∑m,², 也就是Z 的平方的期望值，就是二阶矩。比如，在一个数 据流中，经过的元素都是[1,5]之间的整数，某一个哈希函数能够将这些数从{1,2. 3,4,5}分别映射为{1, - 1,1, - 1, - 1}。流上经过的10个数分别是{1,5,3, 2,4,2,5,1,2,4},则各元素的个数{m;} 分别是{2,3,1,2,2}。计数器Z 的值 = ( 1 ) × 2 + ( 一 1 ) × 3 + ( 1 ) × 1 + ( 一 1 ) × 2 + ( 一 1 ) × 2 = - 4 , 那 么Z²= ( 一 4)²=16,F₂=2²+3²+1²+2²+2²=22。Z 可以作为F₂ 的一个近似值。为了进一步降 低误差，可以设置多组计数器，以及多组相互独立的哈希函数，在所有计数器中，取中值 为估计值，从而在理论上保证误差不超过预定义值。

**6.6.3** **直** **方** **图**

直方图 (Histogram) 是一种摘要数据结构，人们使用直方图来捕抓数据集里的一个 字段或者一组字段的取值的分布情况。在数据库里，直方图一般用来进行查询结果集大小 估计(Query Size Estimation),给出近似的查询结果 (Approximate Query Answering) 以及用于数据挖掘 (Data Mining)。

直方图的类型多样，以下几种直方图在近似查询处理方面得到了广泛应用。

(1)V 最优直方图 (V-Optimal Histogram),它通过一个分段常值函数7(i)(Piece-

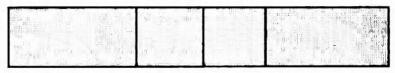
wise Constant Function), 对 一 组数值v₁,V₂,…,vn 的分布情况进行近似表示，使得

的取值最小化。 V 最优直方图可以用于计算数据序列的L₂ 范数。L₂ 范数 是指向量各元素的平方求和，然后求平方根。 L₁ 范数是指向量中各个元素绝对值之和， Lo 范数是指向量中非0的元素的个数。比如，坐标为 (x₁,y₁) 的 点P₁ 与坐标为 (x₂,y2)

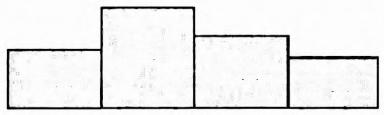
的 点P₂ 的 L₁ 范 数 |x₁—x₂|+Iy₁-y₂| 为曼哈顿距离，L₂ 范数sqrt((x₁—x₂)²+(y₁ 一 y₂)²) 为欧式距离。

(2)等深直方图 (Equi-Depth Histogram), 它把数值的值域划分成一系列的桶，每 个桶所包含数值的数量是均匀的 (Uniform)。 换言之，等深直方图维护了底层数据分布的 分位点 (Quantile), 即直方图的桶的边界就是分位点，于是等深直方图可以用于计算分 位点。

普通的直方图是等宽直方图。比如， 一个单位的1000名职工的工资收入为1000~ 3000元，我们把1000～3000元分成500元一个间隔，根据每个区段内的职工人数绘制 出一个柱状图，就是等宽直方图。等深直方图则把职工按照工资从低到高排序，然后寻找 25%,50%,75%和100%的分位点，以该分位点为桶的边界，分为四个桶，每个桶里的 职工数量是一样的，即柱状图的高度一样。等宽直方图和等深直方图见图6-2。



250人 250人 250人 250人



345人

205人 265人 **185人**

1000 1500 2000 2500 3000 1000 1700 2050 2320 3000

**图6-2** **等宽直方图和等深直方图**

(3)端点带偏的直方图 (End Biased Histogram),它维护少量的发生频率较高(高于 一定阈值)的数值的准确计数，同时对于其他数值则以均匀分布来近似表示其分布特征。 端点带偏的直方图可以用来对冰山查询 (Iceberg Query) 进行处理，给出查询结果。

在很多的应用场合中，人们关注某些超过一定阈值 (Threshold) 的简单的统计量 (比如计数统计),这样的查询称为冰山查询。比如，在搜索引擎日志分析处理中，人们对 一段时间里搜索次数处于前1%的搜索关键字非常感兴趣。

**6.6.4 小波** **(Wavelet) 分析**

小波系数(即对数据进行小波变换后得到的系数)经常作为时域数据的一种摘要式的 表示 (Summary Representation)。

类似于傅立叶变换，小波变换根据输入的一系列数值转换成一系列的小波系数，并且 少数几个小波系数就拥有大部分能量。根据这个特性，可以选择少数小波系数，近似还原 原始信号。实际上，小波系数是一系列数值投影到一组正交的基向量上所获得的系数 (Coefficient), 基向量的选择决定了小波的类型。

小波具有一些优异的特性，从小波系数重建的信号(一系列数值),可以很好地逼近 原始信号(原始数值),也就是和原始数值的差别不大。在数据库中，人们一般采用哈尔 (Haar) 变换，因为哈尔变换比较容易计算。

小波变换可以用于选择率估计 (Selectivity Estimation)、数据立方体的近似计算

(Data Cube Approximation)、多维聚集的计算 (Multi-Dimensional Aggregate) 等。利用 小波系数进行近似计算的结果，在使用相同内存的情况下，比从直方图获得的结果更为精 确。除此之外，小波变换还可以用于普通的聚集查询处理，包括对选择 (Selection)、连 接 (Join) 、 聚集 (Aggregation) 的处理，所有处理都只需在小波系数上进行而无须参考 原来的数据。

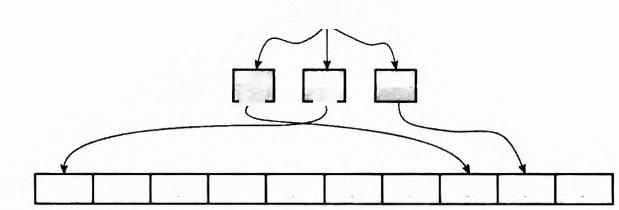
为了把小波变换扩展到数据流处理上，计算数据流的摘要，需要设计在数据流上计算 小波系数的算法。数据流的数据是不断到达的，通过适当的算法，可以在新数据不断到达 时不断更新最重要的小波系数，作为底层数据的摘要式表示。

**6.6.5 布隆过滤器** **(Bloom Filter)**

Bloom Filter是一种简单、高效的数据结构，用来判断一个元素是否属于一个集合。 对其操作包括初始化、元素插入和元素查询过程。 Bloom Filter由一个长度为m 的 bit 数 组和k 个 Hash 函数构成。m 和 k 两个参数可以根据我们接受的假阳性 (False Positive) 比率进行调整。

在 bit 数组里的所有bit, 刚开始设置为0。当一个元素“加入”(并未真正插入数组 中 )Bloom Filter,Bloom Filter对元素运行k 个 Hash 函数，获得h₁, … ,h, 然后对bit

数组里相应的位置置上1。如果一个位置多次被置为1,那么只有第1次会起作用，如图 6-3所示。



Element

h₁ h₂ h₃

0 0 0 0 1 工 0

m=10

1 0

k=3

0

**图6-3** **Bloom Filter示意图**

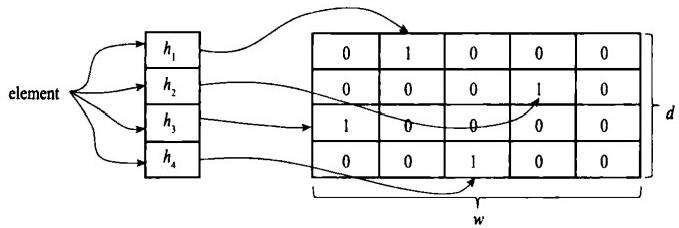
当需要查询某个元素是否属于该集合时，需要对该元素执行k 个 Hash 操作，然后检 查对应的位置上是否都为1。如果其中有任何一个bit 为0,那么该元素不属于该集合。由 于使用Hash 函数，理论上有可能有两个元素Hash 到同样的bit 数组的下标，于是导致没 有在集合里的元素也被判断为属于该集合，这就是假阳性。比如元素 A 和 A',A 属于集 合而A '不属于集合，但是两者经过Hash 操作以后，都对应 bit 数组的一系列下标，使得 Boom Filter 认为A'也属于该集合。虽然Bloom Filter有假阳性，但是绝对不会出现假阴 性 (False Negative),也就是本该属于集合的元素被报告为不是集合的成员。

通过在每一个位置上都用计数器代替比特位，不仅能够判断元素是否属于集合，而且 能够估算元素的频率，可以用于数据流不同数据元素的频率估计。

**6.6.6** **计数最小梗概**

计数最小梗概 (Count-Min Sketch) 使用一个次线性空间 (Sub-Linear Space) 来计算

频率。它包含 d 行 w 列的一个矩阵，w 和d 的选择体现了准确性和时间/空间开销的折中 (Trade Off)。每一行有一个 Hash 函数，当一个元素到达，它被针对每行进行Hash 操作， 即使用每行对应的 Hash 函数对元素数据进行映射得到每行的一个下标，于是对应这些下 标的列的元素保存的计数器 (Counter), 增加1 . 如图6-4所示。可以看出， Count-Min Sketch和 Bloom Filter有一些相似之处。



**图6-4** **Count-Min Sketch 示意图**

每个元素的频率通过下面的办法进行估计。该元素通过 Hash 操作得到每行的一个下 标，提取对应这些下标的列的元素保存的计数器 (Counter), 从中取最小值即可。使用最 小的计数器做某个元素频率的估计值，是一个比较接近的近似值。Count-Min Sketch是解 决计数问题的有力工具。

基于Count-Min Sketch,可以进行基本的查询，包括点查询 (Point Query)、范围查 询 (Range Query) 和内积查询 (Product Query) 等，也可以用于解决数据流中的分位数 查找等问题。

**6.7** **流数据处理系统**

**6.7.1 Storm简介**

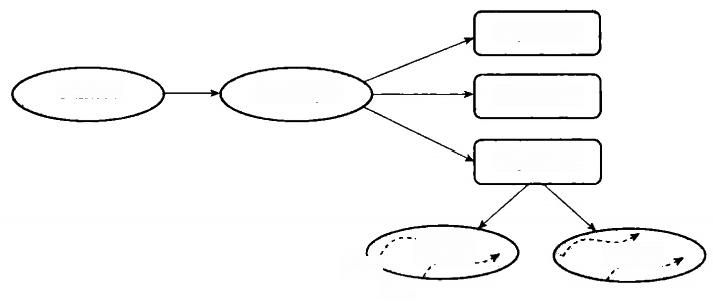
Storm 是一个分布式的、高度容错的实时数据(流数据)处理的开源系统。 Storm 是 为流数据处理设计的，具有很高的处理性能。一个小集群每秒钟可以处理数以百万计的消 息。Storm 保证每个消息至少能够得到一次完整的处理。任务失败时，它会负责从消息源 重试消息.从而支持可靠的消息处理。 Storm 由 Twitter开发并且开源，它使用Clojure 语言实现。

用户可以使用多种语言为Storm 编写应用程序，包括Clojure,Java,Ruby 和 Python 等，还可以通过实现Storm 通讯协议，提供其他语言的支持。

**1.Storm 系统架构**

Storm 集群由一个主节点和多个工作节点组成(见图6-5)。主节点运行一个 “Nim- bus” 守护进程，它的工作是分配代码、布置任务以及故障检测。每个工作节点运行一个 “Supervisor” 守护进程，用于监听、开始并终止工作 (Worker) 进程。





Supervisor

Supervisor

Supervisor

worker

task

task

Nimbus Zookeeper

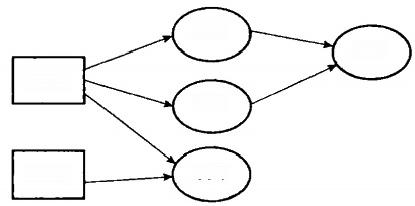
Worker

**图6-5** **Storm 系统架构**

Nimbus 和 Supervisor 都是无状态的(客户端的两次调用间不维持状态),使得两者变 得十分健壮。它们之间的协调工作由 Zookeeper 来完成， Zookeeper 用于管理集群中的不 同组件。Supervisor 监听分配给它那台机器的工作，根据需要启动/关闭工作进程，这个 工作进程称为Worker 。ZeroMQ 是 Storm 的内部消息传递系统。

**2.Storm 编程模型**

Storm 的编程模型如图6-6所示。下面对编程模型的各个组件进行详细的介绍。



bolt

bolt

**spout**

bolt

bolt

**spout**

**图6-** **6** **Storm 编程模型**

(1)数据流 (Stream) 。 数据流是 Storm 的一个关键的概念。 一个数据流是一个没有 边界的Tuple (元组)序列，这些Tuple 以分布式的方式，并行地创建和处理。 Tuple 的 字段类型，可以是短整型 (Short)、 整 型 (Integer)、 长整型 (Long)、 字节型 (Byte)、 字 符 串 型 (String) 、 单 精 度 浮 点 型 (Float) 、 双 精 度 浮 点 型 (Double) 、 布 尔 型 (Boolean) 以及字节数组 (Byte Array) 等不同类型。

(2)计算拓扑 (Topology)。 在 Storm 里，一个实时计算应用程序的处理逻辑封装成

一 个Topology 对象，称为计算拓扑。 Storm 的 Topology, 相当于 Hadoop 里 的 MapRe- duce Job,关键的区别是， MapReduce Job最终会结束，而一个Storm 的 Topology 会一直 运行，除非管理员杀死该应用(相关进程)。

一 个Topology 是 由Spout 和 Bolt 组成的拓扑结构，连接Spout 和 Bolt 的则是数据分 发策略 (Stream Grouping) 。Stream Grouping负责实现Spout 和 Bolt 之间的数据交换。

(3)消息源 (Spout)。 在 Storm 里，消息源称为 Spout, 是消息的生产者。消息源 **Spout** **可以是可靠的，也可以是不可靠的。** **一个可靠的消息源，当一个** **Tuple** **没有被** **Storm 成功地处理，可以重新发送该** **Tuple 。一个不可靠的消息源，** **一旦发出一个Tuple, 就彻底把它忘掉(丢弃),不可能再次发送了。**



**(4)消息处理者** **(Bolt)。所有的消息处理逻辑被封装在消息处理者** **(Bolt)** **里。Bolt** **可以做很多事情，包括过滤、聚集等操作。Bolt的一般处理流程是，处理一个输人** **Tup-** **le,** **发送0个或者多个** **Tuple,** **然后调用** **ack** **接口，通知** **Storm** **自己已经处理过该** Tuple 了。

**(5)Spout 和** **Bolt 之间的数据分发策略** **(Stream Grouping) 。Spout 和** **Bolt 之间的数**

据分发策略称为 Stream Grouping。

(6 **)工作进程** **(Worker) 。Supervisor 监听分配给它那台机器的工作，根据需要启动/** 关闭工作进程，这些工作进程称为 Worker。

一个运行的 Topology, 由运行在很多机器上的工作进程组成。 一个Topology 的处理 任务，就在一个或者多个工作进程里执行，每个工作进程执行整个Topology 的一部分 工 作 。

(7)任务 (Task) 和执行器 (Executor) 。Topology 的每个Spout 或者 Bolt, 当作若 干任务 (Task) 在整个集群里执行。 一个进程包含若干线程。默认情况下，每一个 Task 对应一个线程，称为 Executor, 这个线程用来执行这个Task 。 同一个Spout/Bolt 的 Task 可能会共享一个物理线程。

由于一个 Spout 对应一批 Task 而一个 Bolt也对应一批 Task. 于是 Spout 到 Bolt的 数据分发策略 (Stream Grouping), 定义了如何从Spout 对应的若干Task, 到 Bolt 对应 的若干Task 进行数据分发。

**3.Storm 数据分发策略** **(Stream Grouping)**

Stream Grouping定义上游的数据流以及 Spout 或者 Bolt 的输出如何发送到下游的 Bolt 的各个 Task 上(也就是上游数据如何在下游的 Bolt 的各个 Task 之间做划分)。 Storm 提供了8种数据分发策略，开发者可以根据应用程序的处理需要选择这些数据分发 策 略 。

(1)洗牌式分组 (Shuffle Grouping),上游数据流的Tuple 随机地分发到下游 Blot 的 Task 上，保证每个下游 Bolt 获得一样数目的 Tuple。

(2)按字段分组 (Fields Grouping), 上游数据流的Tuple 按照指定的字段进行分组。 比如，按userid 来分组，具有同样 userid 的 Tuple 会被分到相同的 Task, 不同的 userid 则会被分配到不同的 Task。 对单词进行计数时 (Word Count 程序),必须使用该分发 策 略 。

(3)广播分组 (All Grouping), 上游数据流的Tuple 复制到下游 Bolt 的所有Task。

(4)全局分组 (Global Grouping), 整个上游数据流的 Tuple 到达下游 Bolt 的其中一 个Task, 即具有最小id的 Task。

**(5)不分组** **(Non-Grouping), 与** **Shuffle Grouping的运行机制类似，Storm 把具有**

Non-Grouping 设置的Bolt 推到这个 Bolt 订阅的上游Spout 或者 Bolt 上，在一个线程里 执 行 。

(6)直接分组 (Direct Grouping),Tuple的生产者 (Producer) 决定下游的Bolt 的 哪个Task 接收到该Tuple。

**(7)本地分组** **(Local or Shuffle Grouping), 如果目标** **Bolt 有一个或者多个Task 在**

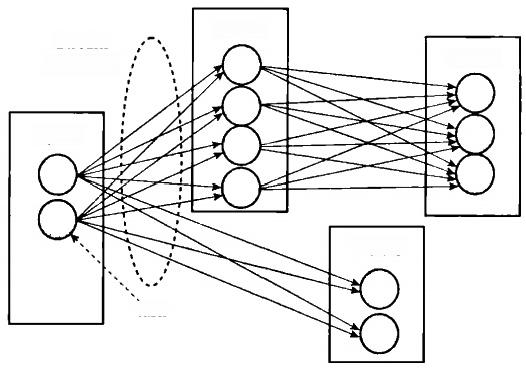
**同一个工作进程中，Tuple** **将会被随机地分给这些** **Task,** **否则，本地分组和洗牌式分组的**



原理是一样的。

(8)部分Key 分 组 (Partial Key Grouping) 类似于Fields Grouping, 上游数据流的 Tuple 按照指定的字段进行分组，但是在下游两个Bolt 之间做负载均衡，可以提高资源的 利用率，特别是在输入数据分布出现倾斜的情况下。

Storm 的 Toplogy 和 Steam Grouping 如图6 - 7所示。



Bolt A

Bolt B

Spout

Bolt C

task

Stream Grouping

**图6-** **7** **Storm** **的Topology和** **Stream** **Grouping**

**4.Storm** **如何保证每个消息都被处理**

Storm 保证每个 Tuple ( 元 组 ) 被 Topology 完整地执行。它是如何做到这点的呢? Storm 追踪由每个Spout Tuple 所产生的Tuple 树，并且跟踪这棵 Tuple 树什么时候成功 处理完。每个从Spout(Storm 中的数据源)发出的 Tuple(Storm 中最小的数据单元), 可能会生成成千上万个新的 Tuple, 它们根据血缘关系形成一颗Tuple 树。当整棵 Tuple 树的所有节点都被成功处理了，我们说从某个 Spout 发出的 Tuple 被完全处理了。

每个Topology 都有一个消息超时设置(默认是30秒),如果 Storm 在这个时间内检 测不到某个Tuple 树到底有没有成功执行，那么 Topology 会把这个 Tuple 标记为执行失 败，过一会儿重新发送该Tuple。

Storm 以其成熟、高性能的流数据处理技术，开源以后成为流数据处理的首选工具， 在各类流数据处理应用场合发挥了重要作用。除了 Twitter 之 外 ，Yahoo 、Yelp 、Flip-

board 、Groupon 、Klout 、 阿里巴巴 (Alibaba) 、 百 度 (Baidu) 、 爱奇艺、淘宝等，都在 实际业务中使用了Storm 。 比如淘宝，使用Storm 从日志数据中计算统计指标，然后从中 抽取有用的信息，帮助进行业务决策，包括实时统计、实时风控、实时推荐等。淘宝的 Storm 系统每天处理的日志数量达到10亿条。

在实际应用中， Storm 也逐渐显露出不足。 Twitter 开发了新一代流数据处理系统 Heron, 它是一个全新设计的实时流数据处理系统，更加稳定，更易于调试，性能比 Storm 高，而且易于管理。 Heron 的技术细节，读者可以参考相关资料。

**6.7.2** **其他流数据处理系统**

除了经典的Storm 系统，其他的流数据处理系统还有Apex,Flink,Onyx 等。下面

简要介绍这些系统。

**1.Apex**

Apache Apex是一个建立在 Hadoop 平台上的流数据处理系统，广泛用于数据导人 (Ingestion) 、ETL 、 实时分析 (Real-Time Analytics) 等应用场合。 Apex 使用 Hadoop HDFS 文件系统作为存储层，并且依赖于Hadoop 平台的YARN 资源管理器，实现资源 分配和应用运行。Apex 保证日志数据不会丢失，每个事件都得到处理。它利用基于内存 的数据处理，获得极高的性能。 Apex 的扩展性好，容错性高，成为Storm 及其后继者 Heron 的有力竞争者。

**2.Spark Streaming**

Spark 大数据平台本质上是一个批处理平台。在Spark 平台上，Spark Streaming通过 一系列小批量数据 (Mini Batch) 的及时处理，实现数据流处理。它把数据流缓存并且分 割成一系列的小批量数据，每个Mini Batch 一次进行处理。由此可见， Spark Streaming 并不是真正的流数据处理系统，它使用批处理系统仿真实现了流数据处理模式。

**3.Flink**

Apache Flink是一个开源的分布式流数据处理系统，它具有极高的性能、高度的容错 性和扩展能力。Flink 被 Alibaba 用于优化电子商务网站的搜索结果(用户对商品的搜 索),他们对商品的一些细节属性和库存信息进行实时更新，提高查询结果的相关性。此 外 .Flink 还应用到网络/传感器监控及错误检测、ETL 等应用场合 (<https://>

flink.apache.org/usecases.html)。

**4.Onyx 和** **Samza**

Onyx 是一个无中心的、容错的分布式计算系统，它支持批处理和流数据处理两种数 据处理模式。(nyx 应用于实时事件流处理、持续计算、 ETL 等应用场合。 Onyx 使用 Clojure 语言写成、开发人员可以使用Clojure 或Java 语言编写程序。

Apache Samza是一个开源的分布式流数据处理框架。它使用Apache Kafka 作为消息 队列，暂时存储不断到达的数据，保证数据不丢失，同时它利用Hadoop YARN 资源管理 和应用程序调度框架，获得高度的容错性和扩展能力。

流数据处理系统层出不穷，没有一个系统能够满足所有应用，或者在所有方面都胜 出。在实际应用中，用户应该根据自己业务的需要，选用合适的系统。选择的办法， 一个 是采用一个流数据处理的评测基准，对各个目标系统进行评测，选择最合适的系统；另一 个是直接针对目标应用系统，根据用户业务的典型负载编写测试程序，并且使用实际数 据，对目标系统进行评测，进而做出选择。

**6.8** **思考题**

(1)流数据处理应用。

(2)流数据模型及其与传统关系数据库的数据模型的区别。

(3)流数据上的典型查询。

(4)滑动窗口及其上的查询处理实例。

(5)流数据的近似查询及其实例。



(6)流数据的时间戳。

(7)流数据的随机采样、梗概技术、直方图、小波变换。

(8)Storm 流数据处理的系统架构。

(9)Storm 流数据处理的编程模型。

(10)简要介绍几个流数据处理系统及其特点。

(11)CEP(Complex Event Processing) 与流数据处理的区别与联系。