Part III. Derived Data

2019年7月9日 15:17

1. 记录和衍生数据系统Systems of Record and Derived Data

a. 记录系统Systems of record

   所有真实有效数据的记录，当其他系统与记录系统的数据不一致时，记录系统的数据是有效正确的

b. 衍生数据系统Derived data systems

   数据从已有的系统中传输转移或处理到另一个系统中，当丢失时可以从数据源再获取，典型的衍生数据系统是缓存、其他如索引、物化视图等都属于衍生数据

S10 Batch Processing

2019年7月9日 18:43

服务Services（在线系统online systems）

服务等待请求或指令，一旦有请求和指令就尽快完成并给出响应，响应时间response time是主要性能指标，服务的可用性非常重要

批处理系统Batch processing systems（离线系统offline systems）

一次系获取大量输入，允许一个任务来处理这批输入，并产生一个输出，一般来说批处理都是定期运行的，吞吐量throughput是主要性能指标

流处理系统Stream processing systems（准实时系统near-real-time-systems）

流处理器类似批处理消费输入产生输出，但在有输入时即运行并产生输出，类似于消息驱动，往往带来低延迟的效果，介于服务和批处理之间

1. 使用Unix工具进行批处理Batching Processing with Unix Tools

a. 简单日志分析Simple Log Analysis
   i. 命令链与自定义程序Chain of commands versus custom program
   ii. 排序对比内存聚合Sorting versus in-memory aggregation
b. UNIX哲学
   i. 统一接口A uniform interface
   ii. 逻辑与布线相分离Separation of logic and wiring
   iii. 透明度和实验Transparency and experimentation

2. MapReduce和分布式文件系统MapReduce and Distributed Filesystems

a. MapReduce任务执行
   i. Mapper函数

      每一条输入的记录都会调用一次mapper函数，每一个输入记录都会通过mapper产生不定数量的KV对，mapper不存在上下文，所有记录可以各自独立进行mapping
   ii. Reducer函数

      收集所有mapper产生的KV对，同一个key可能对有多个values，使用reducer迭代values并产生输出，不同key的recuing也可以独立进行
   iii. MapReduce的分布式执行Distributed execution of MapReduce

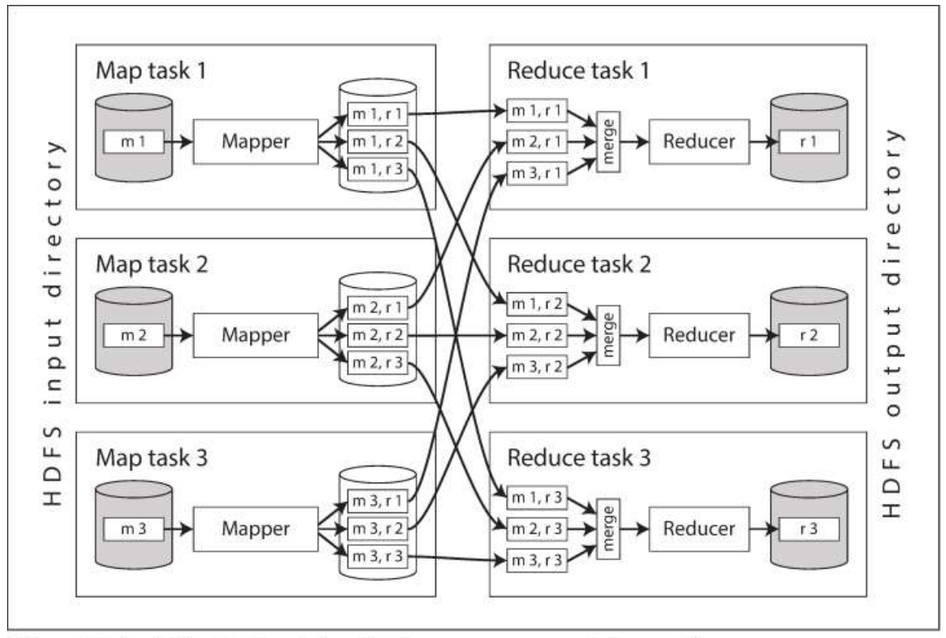

Figure 10-1. A MapReduce job with three mappers and three reducers.

b. Reduce侧的连接和分组Reduce-Side Joins and Grouping

i. 排序合并连接Sort-merge joins

根据mapper输出的key分区，随后再将获得的KV对进行排序，由此相同的key所需要连接的事件都连续相邻存放，随后进行二次排序secondary sort，将同一个key的不同事件也排序分类，再执行reduce时就可以对不同的事件相互之间笛卡儿积，只要从每一类事件的第一条扫描到最后一条，相同事件也已经有序了

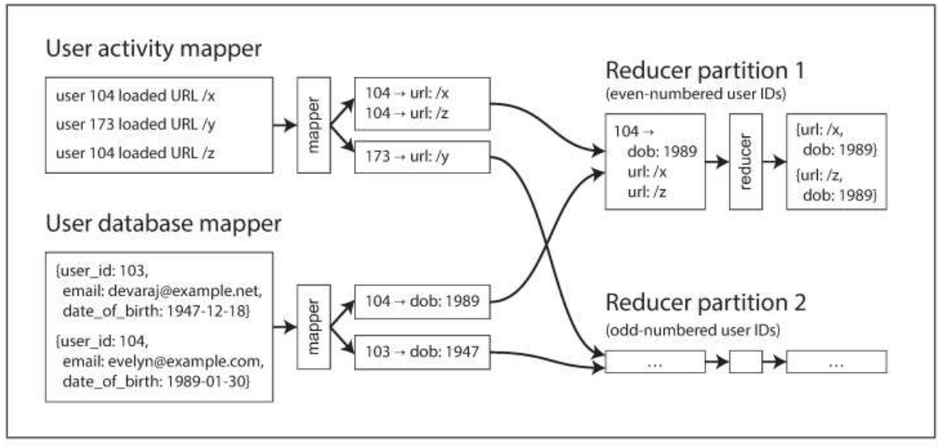

Figure 10-3. A reduce-side sort-merge join on user ID. If the input datasets are partitioned into multiple files, each could be processed with multiple mappers in parallel.

ii. 相关数据存放在一起Bringing related data together in the same place

类似于排序合并连接，mapper出来的KV直接存放在对应的位置，相关的KV存放在一起

iii. 分组GROUP BY

类似于排序合并连接，相关数据存放在一起，则只要对分组的列作为key来设置mapper，就可以在reducer中执行相关的分组后聚合

iv. 处理倾斜Handling skew

由于每个reducer处理单个key的所有KV对，因此对于热点值hot keys，就会导致出现单个reducer处理远超其他reducer的记录数，可以通过对reducer的key也进行分区，并行reducing，最后收集所有结果即可，这样可以分散热点数值的处理压力，可以通过较小的样本

reducing或是显示指定热点值来确定hot key并进行分区

c. Map侧的连接Map-Side Joins
   i. 广播散列连接Broadcast hash joins
      适用于最简单场景下的大数据集对小数据集的连接，小数据集需要足够小放入内存，随后mapper流水线处理大数据集，并根据需要连接的字段去内存中的小数据查询，执行连接后再输出给reducer
   ii. 分区散列连接Partitioned hash joins
      若输入的数据集是分区的，且连接的数据集也是相同模式的分区，则连接也可以逐分区进行，例如所有A数据集的01分区包含的数据只会与B数据集的对应01分区进行，则就可以按分区逐个连接而不用全表连接
   iii. Map侧归并连接Map-side merge joins
      若输入的数据与连接的数据按相同的模式分区，且都按相同的字段进行了排序，就可以执行类似归并的方式进行连接，此时数据集能无论够放在内存中都可以进行
d. 批处理工作流的输出The Output of Batch Workflows
   i. 建立搜索索引Building search indexes
      对于海量文档，将文档分区并根据关键字进行mapping并获得关键字对文档位置的KV对，从而高效建立搜索索引
   ii. 键值存储作为批处理的输出Key-value stores as batch process output
      当批处理输出的键值需要查询被用到时，先输出到新数据库，随后再与旧数据库做替换，而不是直接写入到唯一的数据库中，这样读写分离，且出错可以轻易回滚
   iii. 批处理输出的哲学Philosophy of batch process outputs
      将输入视为不可变且避免副作用，一旦输出出错就可以轻易回到原来的环境或继续使用旧的输出，由于不可变输入，因此重试也是安全的
e. 对比Hadoop和分布式数据库Comparing Hadoop to Distributed Databases
   i. 存储多样性Diversity of storage
      数据库要求按特定的模型来构造数据，而分布式文件系统则可以存储任意数据，但同样的，数据最终根据一定结构进行解释的负担就被转移到了操作者身上，因此常见的过程往往是某种原始数据存储到Hadoop中随后编写MapReduce来清理数据并转换其关系形式，构建好模型导入到数据库中
   ii. 处理模型多样性Diversity of processing models
      MPP数据库单体紧密集成，负责了磁盘布局、查询解释与优化、调度执行等所有操作，因此对于具体数据可以针对性的调整和优化查询达到极好的性能，但是分布式文件系统则支持更为通用的处理模型而不一定只能用SQL查询，由于Hadoop平台的开放性，采用更多自定义的或其他数据处理模式成为可能
   iii. 针对频繁故障的设计Designing for frequent faults
      MPP数据库若单个节点查询时崩溃，往往会终止整个查询随后由用户或内部重试整个查询，而MapReduce被设计成容忍单个Map或Reduce失败的情况，并无害的重试

3. MapReduce之外Beyond MapReduce
   a. 物化中间状态Materialization of Intermediate State
      i. 数据流引擎Dataflow engines
         参考Spark和Flink，相比于MapReduce，其优点在于：
         1) 排序等昂贵的操作只在必要时做，而不是mapper/reducer的默认行为
         2) 去除了不必要的map任务，mapper通常可以与上一级的reducer合并，因为mapper并不改变分区等情况
         3) 流处理中数据的上下依赖和连接都是显式声明的，因此调度器可以从全局执行一些优化
         4) 流数据引擎的算子operator可以在输入就绪后立即开始并产生输出，后续阶段不必等前驱过程结束即可开始

ii. 容错Fault tolerance

Spark使用弹性分布式数据集RDD resilient distributed dataset来跟踪数据，Flink使用检查点checkpoint持久化算子状态，来使得出错时能够重新进行计算，关键在于计算本身是确定的deterministic，即给定输入的输出是一样的

iii. 物化的讨论Discussion of materialization

b. 图与迭代处理Graphs and Iterative Processing

i. Pregel处理模型The Pregel processing model

在每次迭代中，每个顶点vertex调用处理函数，所有发送给这个定点的消息都发送给处理函数，并且将信息通过所有边edge发送给相邻节点，反复迭代这个过程直到没有更多的边需要跟进或系统达到一些指标收敛，处理过程类似于Actor模型，顶点状态句由容错性和耐久性

ii. 容错性Fault tolearance

所有顶点之间通过异步消息传递，提高了Pregel的性能，同时保证了容错性（参考TCP丢包重传），另一方面Pregel模型保证所有一轮迭代中发送的消息都在下轮迭代中送达，所以下一轮迭代开始前前一轮消息必须都送达，每一轮迭代结束时持久化保存所有顶点状态，由此确保任意时刻出错，都可以从上一轮成功的迭代中恢复并重试

iii. 并行执行Parallel execution

由于图计算依赖消息传递，因此跨机器网络开销成为制约图模型的重要因素

c. 高级API和语言

i. 向声明式查询语言的转变The move toward declarative query languages

原来MapReduce允许用户使用任何代码，但是就需要用户自己来设计连接、过滤等算法；通过向声明式查询语言的转变，在高级API中引入声明式的部分，由优化器来自动优化选择使用的算法，提高了人类和机器的效率，同时批处理框架也就变得更像MPP数据库

ii. 专业化的不同领域Specialization for different domains

批处理引擎被用于分布式执行日益广泛的各领域算法，内置更多功能和高级声明式算子；而MPP数据库更加灵活并且易于编程执行任何代码，两者越来越相似

S11 Stream Processing

2019年7月11日 12:55

1. 传递事件流Transmitting Event Streams

流数据中，记录record又是一个事件event，一个小的独立的self-contained、不可变immutable对象，包含着某个时间点发生的事件所有信息；由生产者producer, publisher, sender产生，并被多个消费者consumer, subscriber, recipient消费处理，所有相关的事件可以按照topic, stream来分为一组

a. 消息系统Messaging Systems

两个值得思考的问题：

如果生产者产生速度快过消费者消费速度：扔掉消息、缓冲队列、背压backpressure（也称为流量控制flow control）

如果节点崩溃或暂时不可用，数据会丢失吗：大部分数据库提供数据持久性，但是会有性能损失延迟增大

i. 直接从生产者传递给消费者Direct messaging from producers to consumers

直接使用UDP组播、TCP多播等实现生产者与消费者直接相连，容错程度极为有限，必须假设双方都处于正常运行状态

ii. 消息代理Message brokers

通过消息代理message broker（又称消息队列message queeu）来传递消息，本质上是一种处理消息流的数据库，生产者向消息代理写入数据，消费者从消息代理接收数据，同时由消息代理来负责数据的持久化和处理生产者和消费者的上下线（连接、断连、崩溃等）

iii. 消息代理与数据库对比Message brokers compared to databases

部分消息代理还支持2PC，使得与数据库更接近，但是他们依然有重要区别：

1) 数据库保存所有数据直到被显式删除，而消息代理会自动删除成功传输的消息
2) 由于成功传输的消息会被删除，消息代理假设总数居非常少，因此队列缓冲也可能很小
3) 数据库提供二级索引等各种方式便于数据搜索，而消息代理往往提供采用某种模式匹配订阅某一个topic的子集
4) 消息代理不支持任意的查询，但是数据一旦改变就会通知客户（消息驱动）

iv. 多消费者Multiple consumers

1) 负载均衡Load balancing

多个消费者下，每条数据可以传递给其中一个，进行某个topic消息的负载均衡

2) 扇出Fan-out

多个消费者下，每条数据可以传递给所有消费者，消费者相互独立

v. 确认与重新交付Acknowledgments and redelivery

由于消息代理发送给消费者后，消息是否真的被处理过是未知的，通过显式告知消费代理数据确认acknowledgments来使得消息代理确认消息已成功（注意：例如消息已处理，但是还未ACK消费者就崩溃，重启后导致重复消费消息，此时需要原子提交来解决问题，例如2PC）

当与负载均衡结合时，会出现无法保持消息消费顺序的问题，如下图消费者1顺序处理了M4, M3, M5消息

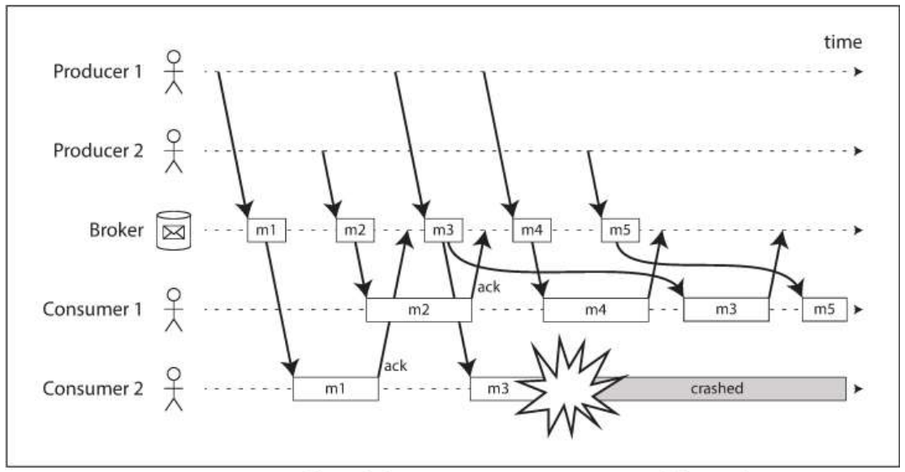

Figure 11-2. Consumer 2 crashes while processing m3, so it is redelivered to consumer 1 at a later time.

b. 分区日志Partitioned Logs

流数据系统如果不在成功传递消息后即删除，而是保存一定数量的消息（数据库保存所有历史数据），则在新的消费者加入时，就可以从最早未被删除的历史数据开始订阅直到最新，提供了更多灵活性，结合了数据库的持久化和流数据的低延迟，称为基于日志的消息代理log-based message brokers

i. 使用日志存储消息Using logs for message storage

消息可以顺序写入日志，对消息进行分区以获得更高的吞吐量和存储总量，并对每个分区的消息进行偏移量offset编号确定顺序，跨分区的顺序是未定义的

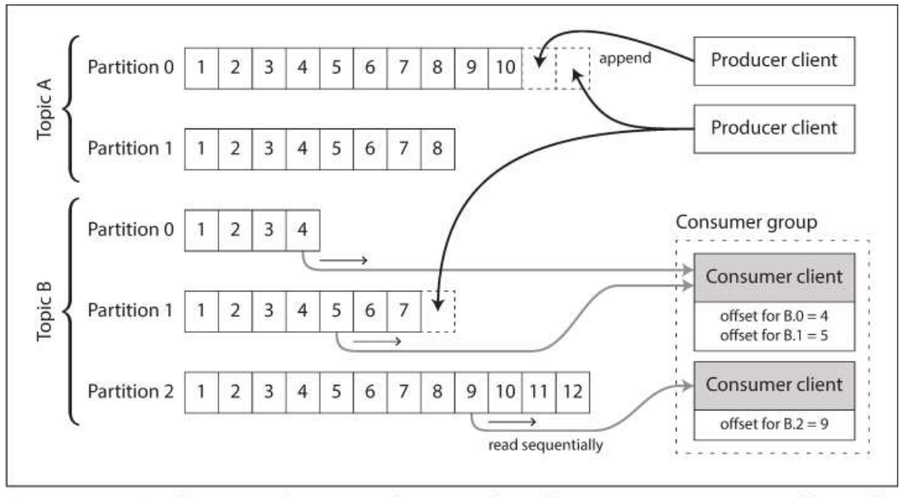

Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

ii. 日志与传统消息对比Logs compared to traditional messaging

基于log之后，负载均衡load balancing可以简单的通过每个分区日志分配消费者，扇出fan-out可以简单的通过每个消费者都读一个分区日志来实现

iii. 消费者偏移量Consumer offsets

通过偏移量来消费一个分区，使得消费到的消息判断很容易，偏移量小的都被消费，偏移量大的都还未被消费，因此消息代理不需要记录每条消息的ACK，只需要直到当前消费偏移量即可，并且提供了批量化和流水线化的可能，批量交付，偏移量批量增加等，类似于单主复制single-leader replication中的日志序列号log sequence number

iv. 磁盘空间的使用Disk space usage

日志被分为多个段segment，根据设定的最大保存量，定期新建segment并删除旧segment，因此特别缓慢或新加入的消费者无法获得已经被删除的segment中的数据

v. 当消费者跟不上生产者时When consumers cannot keep up with producers

vi. 重播旧消息Replaying old messages

在传统的消息系统中，消费并确认ACK消息会导致消息被删除，而对于基于日志的消息系统，则消费消息更像读取文件，重播未删除的旧消息是可能的，因此也使得基于日志的消息系统更像批处理，从故障和崩溃中恢复状态更加容易和可行

2. 数据库与流Databases and Streams

a. 保持系统同步Keeping Systems in Sync

往往一整个大系统中包含有数据库、流数据、应用程序等各个子系统，而使得整个系统同步就非常重要，例如数据库中的数据更新了，则对应的索引和缓存也必须更新，一种方式是双写dual write，即更新数据时同时写入到各个系统，但是双写存在数据竞争的问题会导致最终不同系统间数据不一致（同系统内依然一致，需要跨系统的异构分布式事务，原子提交和2PC）

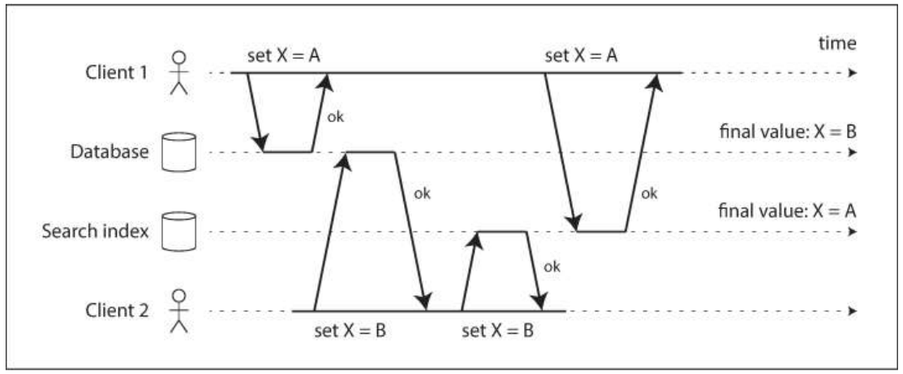

Figure 11-4. In the database, X is first set to A and then to B, while at the search index the writes arrive in the opposite order.

b. 变更数据捕获Change Data Capture

通过捕获数据库的内部log，获取数据库的所有变化，随后通过类似流的方式同步应用到其他系统上

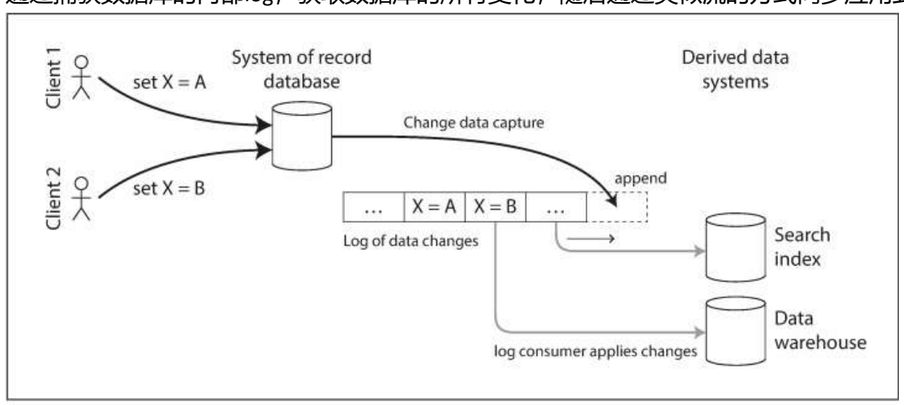

Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

i. 实现变更数据捕获Implementing change data capture

将某一个数据库作为leader，捕获其所有数据变更，并通过基于日志的消息代理发送给所有其他系统作为follower，同时这是异步消息系统，因为所有replication存在延迟的问题，这里都会存在

ii. 初始快照Initial snapshot

保存所有日志并重放可以重建系统，但是过慢且空间占用多，可以定期触发快照，并且快照包含消息偏移值（参考Raft算法的snapshot index、commit index、apply index）

iii. 日志压缩Log compaction

日志压缩类似于快照，保留所有key最新的value，丢弃过时值，例如对数据库的所有变更都保存一个主键，每次日志压缩保持主键的值最新丢弃过时值，则新加入消费者时从offset=0开始重放日志即可，日志由于定期压缩数量并不会很大，参考LSM-trees

iv. 变更流的API支持API support for change streams

c. 事件溯源Event Sourcing

暂时跳过

3. 流处理Processing Streams

批处理在于数据一批一批获得处理，每次数据进入都会积攒到下一个批次的节点才统一处理；流处理是消息驱动，一旦有数据就会处理

a. 流处理的使用Uses of Stream Processing

i. 复合事件处理CEP, Complex event processing

例如一个复合处理引擎，设定好一个处理模式，处理模式由引擎保存，并对持续输入符合条件的复合多个消息进行识别匹配，随后输出处理结果，例如设置上下文依赖的匹配过滤引擎，当连续多条消息满足某一个复杂规则就执行预设的处理

ii. 流分析Stream analytics

分析往往不重视每一条消息，而是关注一些在某个窗口window内的统计指标，则例如一个计算引擎，设定好一些统计函数并由引擎保存，对持续输入的流数据进行统计分析例如求和、取平均等，然后不断输出处理结果，类似聚合引擎

iii. 维护物化视图Maintaining materialized views

例如物化视图往往是一些数据的聚合指标，当所涉及的数据改变时，通过流数据通知物化视图也更新，从而维护物化视图

iv. 在流上搜索Search on streams

类似复合事件处理，设定好一个搜索表达式，由引擎保存，并对持续的输入的事件进行匹配输出，例如设置好select cola, colb from stream1 where colc = ?进行模式匹配过滤搜索

v. 消息传递和RPC

b. 时间推理Reasoning About Time

i. 事件时间与处理时间Event time versus processing time

事件时间event time是时间发生的时间，往往直接由消息的时间戳体现，而处理时间processing time是消息被流数据系统处理的时间，往往会由于延迟等原因使得其与事件时间有偏差（网络延迟、队列等待、重放消息、故障恢复等），因此基于处理时间的分析可能会导致错误的结论

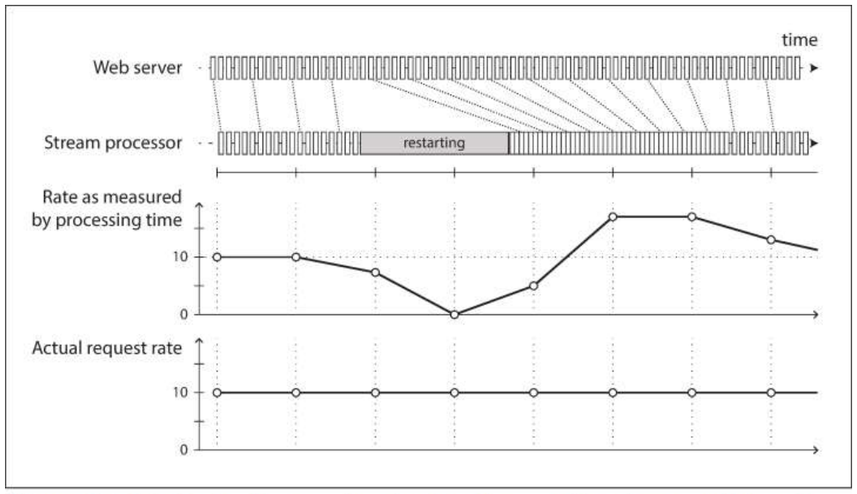

Figure 11-7. Windowing by processing time introduces artifacts due to variations in processing rate.

ii. 知道什么时候好了Knowing when you're ready

使用事件时间的一个问题就是你不知道什么时候是界限，可能由于网络延迟等，某些比如在大量收到39-41分钟数据时还出现了少量37-39分钟的数据，通常可以通过超时来判断某个时间窗口已经完成了，后续本应属于这个窗口的数据都是超时的，对于这些数据，通常可以：

1) 直接丢弃，通常是少量的，同时可以监控超时的数据量，若大量超时则触发告警
2) 发布更正信息，在收到超时数据时对已经计算好的窗口进行更正，可能还要收回原来的输出，棘手

iii. 你用的是谁的时钟Whose clock are you using, anyway?

例如数据由一些终端设备产生并带有终端设备的时间戳，则往往不能保证终端设备的时钟与流数据处理系统的时钟同步，那么就会处理事件时间反常的事件，通常可以通过一条事件携带事件时间（终端设备时间戳）、发送时间（终端设备时间戳）、接收时间（流系统时间戳），通过接收时间-发送时间来估算真正的事件时间（假定网络延迟相比可忽略）

iv. 窗口类型Types of windows

1) 滚动窗口Tumbling window

每个窗口长度固定，每条数据只属于一个窗口，例如03:00-03:59属于一个窗口，04:00-04:59属于另一个窗口

2) 跳动窗口Hopping window

每个窗口长度固定，但是允许窗口之间重叠，相邻窗口的起始间隔就是跳动间隔hop size，例如03:00-07:59属于一个窗口，04:00-08:59属于另一个窗口，跳动间隔是04:00-03:00=1分钟

3) 滑动窗口Sliding window

滑动窗口实际长度不定，一个滑动窗口包含了所有两两数据间隔小于窗口设定范围内的数据，例如01, 03, 05, 06, 07, 09, 12的数据，窗口间隔是4，则这一系列数据可以产生[01, 03], [03, 05, 06], [05, 06, 07], [06, 07, 09], [07, 09], [09, 12]窗口，因此滑动窗口就是前面旧数据逐个踢出，后面尽可能包含进间隔小于设定范围的数据，窗口变长并向前"滑动"

4) 会话窗口Session window

会话窗口聚合单个会话的所有数据，没有长度限制

c. 流式连接Stream Joins

i. 流-流连接（窗口连接）Stream-stream join (window join)

选择合适的连接窗口，如时间间隔，两个流数据在各自选定的连接窗口内进行连接

ii. 流-表连接（流扩展）Stream-table join (stream enrichment)

随着流数据持续流入，每一条数据与表进行连接，在表中查询可能响应慢，可以通过加载表的副本到流数据连接中，但是由于流数据

往往是长久的，表的数据也会存在更新，可以通过变更数据捕获CDC来解决，连接引擎可以订阅表的数据更新日志，在进行流表连接的同时也进行表的更新

iii. 表-表连接（维护物化视图）Table-table join (materialized view maintenance)

表与表连接完成后，要维护物化视图，则两表数据有更新时增量连接，即根据表的变化（流）在流处理引擎中更新连接的结果并输出

iv. 若将流数据连接看作是动态表与动态表的连接，本质性类似于乘法导数，(u*v)'=u*v'+v*u'，即各自的动态改变量与各自原有的数据

$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\ state(t)}{\mathrm{d}t}$$

Figure 11-6. The relationship between the current application state and an event stream.

v. 连接的时间依赖性Time-dependence of joins

流数据连接在一些场合下，事件的顺序非常重要，分区日志保留了单个分区内的顺序，但是跨分区的顺序是未定义的，通用跨流的顺序更加是未定义的，这就会导致流数据的连接发生在近似的时间范围内时，按照什么顺序进行处理，通常通过对特定版本的记录使用唯一的标识符来解决，从而使连接的顺序得以确定

d. 容错Fault Tolerance

对于批处理框架如MapReduce，如果任务失败，简单丢弃所有输出并重试即可，因为数据是不可变的，重试是透明的，而对于流数据框架则不能简单通过任务完成时输出失败重试，因为对于无限的流不存在任务完成的说法

i. 微批量与检查点Microbatching and checkpointing

微批量microbatch是将流分为小块，每个块则按批处理的方式进行处理，失败即重试，成功才输出，较小的批次会导致更大的协调和调度的开销，而较大的批次则意味着延迟更大；而检查点checkpoint则是流数据处理时定期生成的滚动存档点并持久化，如果算子operator崩溃，就从最近的检查点重启，并丢弃最近检查点后的所有输出

ii. 原子提交再现Atomic commit revisited

为了确保在出错时也能保证恰好一次exactly-once处理，要求所有输出都仅在处理一个事件成功时才有效，包括输出到终端消费者等外部系统、数据库写入、算子状态的变更、输入消息的确认等，这些都需要原子的全部发生或全部不发生，每一条消息都通过这种原子提交开销过大，例如Flink采用周期性异步checkpoint以及在checkpoint时进行2PC对全局checkpoint进行原子提交来实现容错

iii. 幂等性Idempotence

为了保证恰好一次exactly-once处理，即失败的输出会丢弃，安全的重试不会生效两次，一种方式是分布式事务，另一种方式就是依赖幂等性，例如每条消息带有一个单调递增的偏移量，则与外部系统交互时带上偏移量，外部系统就知道这条消息是否已经处理过，从而可以避免二次处理，达到恰好一次

iv. 失败后重建Rebuilding state after a failure

任何需要状态的流处理：聚合处理引擎、用于连接的表和索引等，都必须在崩溃后能够重建，可以在流处理系统本地节点保存状态，恢复时读取状态副本，例如Flink采用周期性异步checkpoint保存所有算子的快照

S12 The Future of Data Systems

2019年7月12日 0:31

1. 数据集成Data Integration

a. 组合使用衍生数据的工具Combining Specialized Tools by Deriving Data

i. 理解数据流Reasoning about dataflows

当需要在多个不同的系统中维护相同数据的副本以满足不同的访问模式时就需要对不同的系统以及数据流了如指掌，例如可以通过单个系统来提供所有用户输入，从而决定所有写入顺序，就可以在不同的系统上按相同的顺序处理写入，全局顺序达成共识

ii. 衍生数据与分布式事务Derived data versus distributed transactions

分布式事务通过原子提交来确保变更只生效一次，而基于日志的系统通过确定性重试deterministic retry和幂等性idempotence来确保只生效一次；分布式事务通过2PL等方式，可以提供线性一致性，而衍生数据系统通常异步更新，不提供相同的时序保证

iii. 全局有序的限制The limits of total ordering

1) 大多数情况下构建完全有序的日志需要汇报所有事件到一个领导节点，吞吐量就受此局限，而如果为了吞吐量进行分区，则不同分区内的事件顺序关系就不明确了
2) 如果数据中心地理位置相距很远，为了容忍整个数据中心掉线，每个数据中心有单独的主库（因为跨地域的同步复制效率低下，使用多主复制），此时会出现不同数据中心的事件顺序未定义
3) 进行微服务布置系统时，往往每个微服务独立进行持久化，微服务之间不共享状态，则来自两个不同服务的事件顺序未定义
4) 某些终端程序允许服务器不确认而直接生效，此时如果与服务器同步，服务器和终端可能以不同顺序看到事件

注意：大多数共识算法consensus是针对单个节点的吞吐量已满足使用来设计的，一旦涉及吞吐量超过单个节点、跨地域的服务就是新的研究问题

iv. 排序事件以捕捉因果关系Ordering events to capture causality

b. 批处理与流处理Batch and Stream Processing

i. 维护衍生状态Maintaining derived state

ii. 为了应用升级而重新处理数据Reprocessing data for application evolution

应用需要演化升级，通过旧架构和新架构并排维护，逐步将少量用户转移到新架构，以测试性能和发现问题，逐步进行转移直到最终删除旧架构完成演化升级

iii. Lambda架构The lambda architecture

Lambda架构就是并行运行批处理系统和流处理系统，流处理系统快速消耗事件并产生粗粒度输出，批处理系统批量消耗事件并产生精确输出，但也有需要维护两套系统的额外开销、两套系统的输出难以合并、增加了系统的复杂性等问题

iv. 统一批处理和流处理Unifying batch and stream processing

在一个系统中统一批处理和流处理需要以下功能：

1) 重放数据replay：通过重放历史数据是的流处理器也可以重复处理"流"数据
2) 故障容错：对流处理器来说exactly once语意，对批处理器来说丢弃任何失败任务的部分输出
3) 事件时间：按事件时间进行窗口化和处理，因为重放历史数据时按处理时间进行窗口化毫无意义

2. 分拆数据库Unbundling Databases

a. 组合使用数据存储技术Composing Data Storage Technologies

i. 创建索引Creating an index

创建索引时数据库需要重新扫描现有数据集，并从一致性快照中构建索引，并且随后新数据进入时需要保持索引更新，更新索引类似追随者

ii. 一切的元数据库The meta-database of everything

整个数据流就像一个巨大的数据库，每个模块就像数据库的子系统，例如批和流处理器就像触发器，不同的衍生数据系统就像不同的索引类型，根据主库的变化而同步或异步保持更新索引，某个子系统故障就中断"全局事务"

1) 联合数据库：统一读取Federated database: unifying reads

   为底层各系统提供一个统一的查询接口，遵循单一继承系统与关系型模型的传统，带有高级查询语言和优雅的语意，但极其复杂

2) 分拆数据库：统一写入Unbundled databases: unifying writes

   将子系统可靠的连接在一起，统一写入，确保整个系统的可靠和容错

iii. 开展分拆工作Making unbundling work

传统的同步写入方法需要跨异构存储系统的分布式事务，这是低效且复杂的，具有幂等性写入的异步日志是一种更加可靠和实用的方法

1) 系统级别，异步事件流使整个系统对各个组件的中断或性能下降更稳健，如果节点出错，事件日志可以缓冲消息，有问题的消费者也可以及时追赶上而不会错过数据，相比之下，分布式事务的同步交互往往会放大故障到整个系统不可用

2) 分拆后的数据系统可以允许不同团队独立开发，事件日志提供了一个足够强大的接口以满足相当强的一致性要求（持久化以及事件顺序）

iv. 分拆系统对比集成系统Unbundled versus integrated systems

b. 围绕数据流设计应用Designing Applications Around Dataflow

c. 观察衍生数据状态Observing Derived State

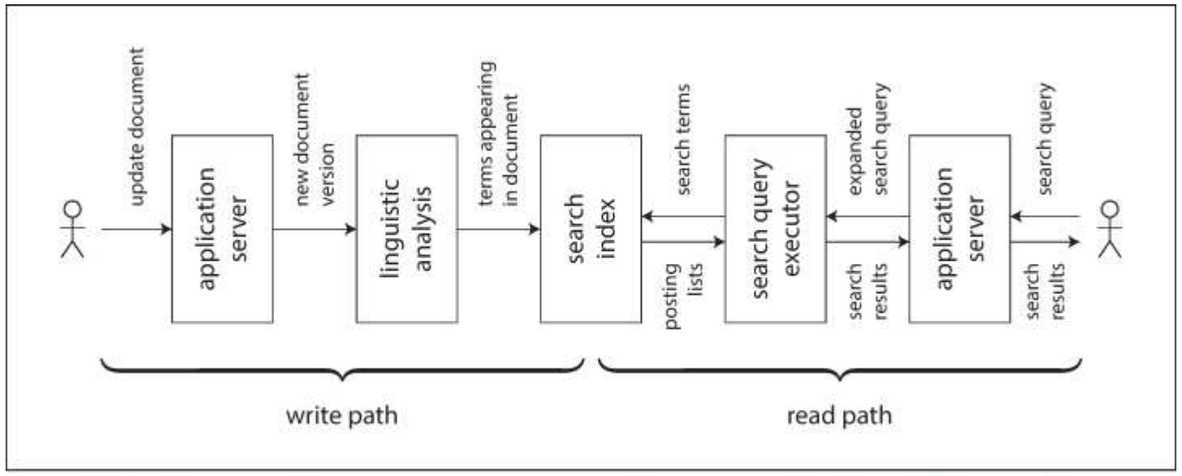

Figure 12-1. In a search index, writes (document updates) meet reads (queries).

i. 物化视图与缓存Materialized views and caching

通过物化视图或缓存将常见的查询预先计算结果

ii. 有状态，可离线的客户端Stateful, offline-capable clients

在设备上使用本地数据库保存状态，无需连接服务器即可使用，一旦联网就可以通过后台网络与远程服务器同步

iii. 推送状态变更给客户端Pushing state changes to clients

主动推送更新数据给客户端，并且通过偏移量来处理短暂离线的设备推送失败，每个终端都相当于是流数据的消费者

iv. 端到端的事件流End-to-end event streams

例如即时消息传递或在线游戏，端到端的事件流，两侧都是有状态的（但是无状态客户端的请求/响应模式已经根深蒂固）

v. 读也是事件Reads are events too

写入是通过事件日志进行的，而读取是临时的网络请求直接导向数据节点，也可以将读取也变为事件流，与写入一样送往流处理器，流处理器通过将读取结果发送到输出流来响应读取事件，读取也作为事件进行流处理可以更好的追踪因果关系，但是会产生额外的存储和IO开销

vi. 多分区数据处理Multi-partition data processing

对于单个分区没有必要使用流数据来查询和响应，但是对于多个分区的数据，本身数据就需要从多个分区进行合并、连接等操作，此时可以利用流数据提供的消息路由等功能

3. 目标是正确性Aiming for Correctness

a. 为数据库使用端到端的原则The End-to-End Argument for Databases

i. 正好执行一次操作Exactly-once execution of an operation

幂等性，对于非原生幂等的操作，需要额外维护一些元数据（例如每个操作ID有一个updated值，只要有updated值说明操作已经执行过了，参考dragonboat的session和updated）

ii. 抑制重复Duplicate suppression

iii. 操作标识符Operation identifiers

为每个操作生成一个唯一的标识符如UUID，将UUID存储在数据库中，则每次执行操作时如果数据库中已经有对应的UUID了，说明操作重复，可以由此抑制重复操作

iv. 端到端的原则The end-to-end argument

低级别的可靠性功能本身不足以确保端到端的正确性，例如用户亲自提交重复的请求、TCP/UDP校验和无法检测BUG导致的数据损坏、TLS/SSL可以阻挡网络攻击者但无法阻挡恶意服务器等，因此需要端到端的功能来确保安全、幂等等等特性

v. 在数据系统中应用端到端思考Applying end-to-end thinking in data systems

仅仅使用了相对更安全的数据系统（例如可序列化事务）并不意味着应用就安全了，应用需要更多端到端的措施

b. 强制约束Enforcing Constraints

i. 唯一性约束需要达成共识Uniqueness constraints require consensus

唯一性检查可以通过确保请求ID相同的请求都被路由到同一分区，然后每个分区有唯一的领导节点，由领导节点来确保只有一个请求成功（本质上就是达成共识问题）

ii. 基于日志消息传递中的唯一性Uniqueness in log-based messaging

日志确保所有消费者以相同顺序看见事件，即全序广播并且等价于共识，因此通过使用日志，所有流处理系统就可以看到所有事件的顺序，由此对于冲突ID，第一个请求事件成功，随后的都失败，并输出结果到输出流中，基本原理在于任何可能导致冲突的事件都会由相同的分区进行日志顺序处理解决冲突

iii. 多分区请求处理Multi-partition request processing

跨分区事务也可以通过分区日志来解决正确性而无需分布式事务的原子提交：

1) 某个请求同时操作了分区A和分区B的数据，则由客户端提供一个唯一的请求ID，并按ID写入相应分区
2) 流处理器读取请求日志，对于每个请求，继续流出两条消息，分区A的处理事件，分区B的处理事件，两条消息均带有原始请求ID
3) 后续的流处理器根据请求ID进行去重，并分别应用到分区A和B上

由于请求ID也写入了相应的分区，因此在1/2之间失败时可以根据分区日志重建请求，2/3也有相应的分区日志，失败时也可以根据分区日志重建事件，同时根据ID进行去重，因此A宕机B成功也可以最终A上重试到成功

注意：还可以通过额外的流处理器来校验数据，拒绝会破坏数据约束的操作如账户透支

c. 及时性与完整性Timeliness and Integrity

i. 数据流系统的正确性Correctness of dataflow systems

完整性Integrity是流处理系统的核心，而由于流处理本身是异步日志式的，不保证及时性Timeliness，恰好一次语义是保证完整性的机制

1) 将写入操作的内容表示为单条消息从而可以被轻松地原子写入
2) 使用与存储过程类似地确定性衍生函数，从一个消息中衍生出其他状态变更
3) 客户端生成地请求ID传递通过所有处理层次，实现端到端去重，提供幂等性保证
4) 消息不可变，允许衍生数据能够随时被重新处理，使错误恢复更容易

ii. 宽松地解释约束Loosely interpreted constraints

提供补偿性事务compensating transaction，即即使唯一性约束被破坏，例如超卖机票，也可以通过补偿和道歉来修复，这些也是可以接受的

iii. 无协调数据系统Coordination-avoiding data systems

数据流系统可以维持衍生数据地完整性保证而无需原子提交、线性一致性、同步跨分区协调，虽然严格的唯一性约束需要及时性和协调，但是适当宽松的约束只要保证完整性即可，而约束被临时违反后可以得到修复

d. 信任但验证Trust, but verify

i. 维护完整性，尽管软件有漏洞Maintaining integroity in the face of software bugs

ii. 不要轻信承诺Don't just blindly trust what they promise

iii. 验证的文化A culture of verification

iv. 为可审计性而设计Designing for auditability

v. 端到端原则的思考The end-to-end argument again

vi. 用于可审计数据系统的工具Tools for auditable data systems

4. 做正确的事Doing the Right Thing

a. 预测性分析Predictive Analytics

i. 偏见与歧视Biasa and discrimination

人类判断会有主观性，但是即使叫给算法，也不能排除算法存在系统性偏差，另一方面算法基于数据，给定的数据本身也可能存在偏差

ii. 责任与问责Responsibility and accountability

iii. 反馈循环Feedback loops

b. 隐私和追踪Privacy and Tracking

i. 监视Surveillance

ii. 同意与选择的自由Consent and freedom of choice

iii. 隐私与数据使用Privacy and use of data

iv. 数据资产与权力Data as assets and power

v. 记着工业革命Remembering the Industrial Revolution

vi. 立法和自律Legislation and self-regulation