**大数据计算模型**

**概要设计说明书**

**V0.1**

**2014年5月24日**

**说明：黑色字体为较为确定可以成文的部分。红色字体是未确定部分，想法不太成熟，亟待完善的。黄色字体是暂时搁置，等下一个版本或者过一段时间再进行完善的部分。**

# 术语定义

**1.1 模型相关**

**Input Adaptor：**数据流的入口节点，输入适配，负责输入的接收和预处理、时间戳epoch的申请、eBag id的分配、eBag的封装和发送等工作。

**Operator/Processor：**流式作业中的**算子/计算单元**，实现了特定的数据处理逻辑（map，reduce和迭代计算）。

**Operator/Processor Task:** Operator/Processor的并行化单元。每个Operator/Processor可以被并行化为N个Task，每个Task处理数据流的一部分数据。

**Epoch：**表示时间周期. input adapter在封装每一个eBag时会给定一个epoch，代表eBag产生的时序。每个Epoch可以有多个EBag.

**EBag：**数据传输的基本单位。由input adapter负责封装和发送，processor可以接受、处理、生成并发送eBag。

**EBag id：**每一个eBag在生成的时候需要配置一个eBag id作为它唯一的认证记录，一个eBag id中包括task id、sequence number。EBag id保证唯一性、时间上的递增性。

**Tuple：**input adapter接受的每一条数据称为一个Tuple. Tuple也是Operator处理的单位。一个或多个tuple可以封装成eBag，进行小批量发送。

**Partition**：数据流或内存数据按某个key被**划分**(Partitioning)成多个**分区**(Partition)。

**1.2 系统相关**

**Cluster Manager：**集群管理者，负责整个集群的工作状态监控、作业分配、失效时的恢复和调度等工作，一般一个集群只有一个Cluster Manager（和一个备用Cluster Manager），运行在Head Node上，失效时并不会影响当前计算作业，但是无法分配新的计算作业。

**Job Manager：**作业管理者。每个作业都有一个对应的Job Manager。在每一个计算作业（Job）被提交的时候，Cluster Manager会根据当前集群各个Compute Node的工作状态来为这个作业分配一个Job Manager，由它负责作业的调度工作，例如：Timer的管理、Checkpoint的控制等等。Job Manager一般工作在Compute Node上，并尽量多和分配的worker在更近的机器上。

**Compute Node：**对应一个物理节点，即具体的计算机器。每个Compute Node上有一个Node Supervisor负责监控机器，可以有多个worker进行计算任务和若干个Job Manager进行作业的管理。

**Node Manager/Supervisor：**在每一个Compute Node上运行一个Node **Manager** /Supervisor，负责监控这台机器的工作状态并向Cluster Manager做定时报告，同时负责配合Cluster Manager进行Job Manager、worker的分配和释放。

**Worker：**运行在Compute Node上，每个Worker代表一个进程/线程。每个Compute Node上可以有多个Worker，负责运行Processor Task，执行具体的计算任务。

**Timer：**维护在Job manager中，负责系统的计时，并分配epoch。

# 引言

目前，主要的大数据处理技术有批量数据处理、流数据处理和图计算。其中，批量数据处理针对大规模静态数据，以高吞吐为特征；流数据处理针对持续产生的数据流， 以快速响应为特征；图计算针对了图数据，以迭代计算为主要特征。这三种数据处理技术都有大量的真实应用，在应用中通常同时部署。随着大数据处理的深入应用，这种多系统并存的状况不但推高了数据处理成本，导致应用开发和运行管理麻烦，还阻碍了数据在不同应用之间的快速流动和共享，降低了数据的时效性。因此，如何融合这三种数据处理技术并建立统一的数据处理平台成为了亟待研究和解决的问题。本项目针对该问题开展研究，首先研究支持批量处理、实时流处理和迭代计算的统一计算模型；其次研究统一计算模型下的计算任务综合调度问题；最后设计实现其原型系统，并通过微博数据分析应用加以验证。鉴于数据处理系统在数据查询、分析、挖掘中的基础性作用，以上研究具有重要的学术意义和实用价值。

本项目的大数据计算模型采用了DAG（有向无环图）的系统逻辑模型，同时对于DAG模型进行改造，增加了一种特殊节点可以支持自身的数据迭代运算从而满足迭代运算的需求。在节点中，我们主要实现了mapper和reducer两种编程接口，对于需要迭代的节点则增加了iterate接口进行迭代的计算。同时在输入层，我们用Input Adapter作为输入适配进行数据封装，时间戳配置等工作。

在这个文档中，我们首先做了概要的介绍，然后我们在第二章中进行了系统需求的详细说明，第三章中呈现了我们的大数据计算模型的工作原理，包括系统结构、执行流程等，模型中的主要关键技术在第四章中得到说明，最后我们在第五章中举了两个例子来具体说明如何将我们的大数据计算模型运用到实际运算中。

# 研究背景

**批量数据处理**：主要针对静态数据集，以吞吐量大为显著特征，其典型的技术是MapReduce。MapReduce 是一个分布式编程框架，被广泛应用于数据抽取、转换、查询、统计等数据处理中，其应用实例包括分布grep，分布排序、web 连接图反转、每台机器的词矢量、web 访问日志分析、反向索引构建等。

**流数据处理：**在金融应用、网络监视、Web 事件处理等领域，数据持续不断的产生，并且以大量、快速、时变（可能是不可预知）的方式持续到达，称为流数据。流数据处理的实时性要求很高，通常需要以毫秒或秒为时间单位对事件做出反应。近年来，由于对大数据的重视和实时数据处理的需求，工业界及学术界又掀起了流数据的研究热潮，出现了大数据环境下的流式数据处理系统，包括Yahoo! S4、Twitter Storm、MapReduce Online，以及2013 年发表的Spark、MS naiad等。

**图数据处理：**图广泛适应于许多领域的应用，如社交网络、Web 网络、生物数据分析和软件代码剽窃检测等。图模型正在互联网和社交网络领域得到越来越广泛的应用。随着真实世界中实体规模的扩张，导致对应的图数据规模迅速增长，因而构建高效的大规模图数据处理系统也成为了急需解决的问题。图计算的算法和机器学习算法类似，通常是复杂的、多阶段或迭代的计算。在MapReduce 技术取得广泛应用时，也有学者研究用MapReduce 实现图算法和机器学习算法，工程实践上亦有采用多个MapReduce 作业依次执行以实现PageRank 的计算，然而MapReduce 毕竟不是为此目标而设计的，其离线批量处理模式缺乏对消息交互、迭代计算的支持，导致其对图数据处理的性能较差。针对此，学术界针对分布式图数据处理进行研究，其典型成果有Pregel，GraphLab，PowerGraph和Kineograph等图处理系统。

以上三种数据处理技术中，批量数据处理技术实现了大规模静态数据的高吞吐处理，流数据处理技术针对了持续产生的数据流的快速处理，而图数据处理针对了图数据的迭代计算。三种数据处理技术分别实现不同数据结构、不同计算模式和不同响应时间的数据处理。以搜索引擎为例，在抓取网页后，要进行文本处理，建立倒排索引，计算Page Rank 等，然后才能被搜索；而用户的搜索行为又被记录下来，留待进一步的分析。在该例中，文本处理通常采用MapReduce 技术，Page Rank 在Pregel 或MapReduce 中实现，用户检索行为记录则交给流式系统做实时分析或MapReduce 系统做离线分析，以更新用户偏好，从而更好地投放广告、推荐产品。这三种数据处理技术都有大量的真实应用，所以在大规模数据处理中，它们经常同时部署。

然而，随着大数据的深入应用，这种多系统并存的状况也导致应用开发和运行管理的麻烦（开发人员在多个编程框架上编写应用，运维管理人员维护多个计算系统及其上的应用），抬高了大数据处理的资源成本、时间成本和人员成本。更为重要的是，在追求数据时效性的今天，用户希望尽快得到最新信息、最新的变化，企业希望更及时地投放广告推荐产品，监管部门希望更及时地掌握社会事件或虚拟用户的影响力，然而由于数据处理应用分别运行于不同的计算系统中，应用之间的数据通道长且数据传递慢（可能需要经过网络、内存、磁盘等，并且在外存中停留较长时间），障碍了信息在应用之间的快速流动和共享，降低了数据的时效性。因此，如何融合三种数据处理技术并建立统一的数据处理平台成为了亟待研究和解决的问题，受到了工业界和学术界的广泛关注。

与此相对应，近来工业界和学术界也有把三种平台进行综合的动力和趋势。UC Berkly 研发的数据处理系统Spark以小批量的方式处理数据，某种程度上综合了批量处理和流式处理两种技术的部分特点；微软研究院2013 年提出的Naiad实现了流处理系统和迭代计算的结合，以更好支持机器学习和图计算。在工业界方面，国内的阿里巴巴公司的分布式流数据实时与持续计算平台（IProcess）是这方面很好的工程实践。需要明确的是，虽然以上研究和实践取得了重要成果，但是离建立支持批量处理、实时流处理和迭代计算的统一计算模型与执行系统尚有不小的距离。

针对三种数据处理技术的融合问题（下文简称统一计算问题），本项目拟研究支持批量处理、实时流处理和迭代计算的统一计算模型和系统。其次，在统一的计算模型和共享的集群资源下，一个机群里可能同时运行着批量、流式和迭代计算等多种类型的计算任务（下文简称异构计算任务），这些任务有不同的计算特征和资源需求，因此，本项目还将研究统一计算模型和系统下异构计算任务的混合调度问题，建立混合调度策略和算法，以充分利用机群的资源。最后，本项目将设计实现统一计算原型系统，并通过微博数据分析应用加以验证。鉴于大数据的重要性以及数据处理系统在大数据查询、分析、挖掘中的基础性作用，以上研究具有重要的学术意义和实用价值。

# 目标需求

这一部分说明了大数据处理模型需要支持的数据类型（计算类型），以及各种数据的计算特点，可以结合目前的主要系统和具体用例进行展开。

我们的大数据处理模型需要支持的计算类型有：

* 批量计算
* 流式计算
* 迭代计算(含图计算)
* 增量式计算

其中，**批量计算**以MapReduce为代表，一般针对离线的大数据集，特征是大吞吐量；**流式计算**系统如Storm，针对连续输入的小集合流式数据，特征是实时性要求比较高，需要连续处理；**图计算**的典型代表是Google Pregel，例如page rank，shortest path等应用场景，可能需要迭代运算；**迭代运算**很多存在于图计算中，也有一些机器学习算法需要迭代计算，例如spark针对迭代运算做了较多优化，比mapreduce要快100倍；**增量计算**存在于流式计算、图计算等多种场景中，对发现变化的数据进行处理，对于提升计算速度有较为明显的作用，其中Kineograph等系统通过增量式计算进行图的持续更新。

我们希望我们的大数据处理模型能够支持批量计算、流式计算，同时适用于图处理和迭代计算，同时可以实现增量式计算。我们希望将以上不同的计算模式综合在一套计算系统中，一方面节约资源成本和开发成本，另一个方面也便于不同的计算应用协同作用。而且目前的主要大数据处理系统都没有形成直接的对数据处理平台的统一，因此这项研究是具有学术意义和实用意义的。

# 模型

## 计算模型

为了实现统一计算的问题，我们首先需要确定一个可以适应批量计算、流式计算、迭代计算的统计的计算模型，这个模型不仅仅需要适用多种计算结构，同时需要保证计算任务的快速流动，也需要满足可靠性的要求。我们的大数据处理模型在这里采用一个**有向无环图（Directed Acyclic Graph, DAG）**的模型，这种处理模型在很多流式系统如storm，timestream等得到很好的运用，也证明了在流式计算中具有良好的表现。

在作业图中，有两种节点：**输入适配(Input Adapter)和算子 (Processor)，**其中*输入适配*读取(或接收)数据并产生数据流(key-value)，算子代表了一个计算过程，处理Key-Value数据。计算的主要编程模型包括map和reduce（详见5.2 编程模型）。为适应迭代计算，我们将DAG模型进行了一点修改：允许图中的节点有一个指向自己的边，即**允许一些节点有自环**。由此，可以通过节点的自环路实现迭代计算。我们认为这种单个节点的迭代可以基本满足主要的计算需求，而多个节点的环路结构太过复杂而且并无必要。



图1 计算模型

数据流是时序的key-value流，input adapter按照一定的策略将key-value封装为eBag，每个eBag都标记了所属的时间信息，记为epoch. 同一个epoch可以有多个eBag。Epoch由Input Adapter控制，全局范围内的协调保证epoch是唯一且递增的。

用户须先实现作业的完整拓扑结构，即确定图中节点之间的关系，以及每个节点对应的计算函数，然后作业才可以被提交和执行。这一点类似于Storm的topology。

### 输入适配

**Input Adapter**主要负责的工作包括对输入的接收和预处理、eBag的封装和发送这两项内容。

输入适配input adapter主要实现两个方面的工作：

* 预处理，即接收输入并将其翻译为可供模型处理的数据；
* 封装，将经过预处理的数据按照一定的封装策略封装为事件（eBag）封装并发送出去。

**多输入流问题**：在大数据处理模型中，可能存在多个数据输入源的问题。针对多输入流的问题，我们按输入类型区分，对于每一种输入需要分别设计一种input adapter：对输入数据类型相同的源，我们逻辑上设计一个input adapter（计算时一个input adapter可能并发为多个任务分布到不同节点上）；如果输入源类型不同，则需要分别针对每种输入设计input adapter分别处理。

**并行化：**对于同一个input adapter（逻辑上同一类型输入源对应的一个input adapter）在执行中可能被并发为多个任务并分布到多个机器节点上。每个input adapter的并发数量由应用程序给定，如果此值缺省我们默认为1。

### 算子(Operator)/计算单元(Processor)

作业图中，每一个算子o可以接受一个输入eBag流，处理eBag中的每个key-value型数据，同时保持一个算子状态So，以及一个计算函数*f*。

**类型**：我们定义了mapper、reducer、iterater三种计算接口，其中mapper和reducer的接口定义类似MapReduce中的相应计算，只不过增加了时序信息。Processor可以继承iterater接口，以及mapper和reducer两个接口的其中一个。根据计算函数的实现类型不同，我们可以将算子分为四类：

* Map
* Reduce
* IMap（Iterative Map）
* IReduce（Iterative reduce）

Map和Reduce节点分别需要实现map和reduce的计算，而iMap和iReduce节点除了继承mapper和reducer接口外，还需要继承和实现iterater接口。注意，我们不允许同时继承mapper和reducer接口。

**状态**：算子状态由用户定义，若用户在内存中保存应用数据，则算子是有状态的。若用户不保存应用数据，则算子是无状态的。算子设定了一个参数，标记该算子是否需要保存状态。

对于有状态的算子，我们设计了集合、向量、树、图等数据结构，以便于应用开发和数据管理。

**并行化**：在作业的执行中，每个算子都可能被并发为多个计算任务(Operator Task)，并分布到多个机器节点(Computer Node)上执行。每个并行的计算任务都执行一样的算子，但是分别维护不同分区的数据，处理不同分区的数据流。

### 事件封装

为了实现统一的处理模型，需要对于输入数据进行封装，把所有数据封装为事件（EBag），以后以eBag为单位进行分发和处理。因此，作业图中的信息流实际上是eBag流，每个eBag是封装了一部分key-value数据。每个EBag标记了所属的epoch。EBag只封装同一个epoch内的数据，不允许不同epoch的数据被封装到一个eBag。EBag最初由Input adapter构造，封装的粒度可以根据时间和数据量进行控制，例如可以将1或100或1000条key-value封装成一个eBag；也可以将64M或256M数据块中的key-values封装成一个eBag（该eBag不直接“保存”数据内容，而是给出数据块在外存的地址）。封装的粒度和方式也决定了批处理和流处理的差别。

EBag是数据的分发和传输单元，也是模型中容错的最小单元。EBag的封装策略如下：

1. **按照数据大小，每个eBag携带的数据量相同。**这种策略不考虑时间，当数据到达一定大小（或条数）后则封装为一个eBag，只要数据没有到达预置大小，不论经过多长时间，都不封装发送。这种策略数据吞吐量大但延迟高，没有实时性要求时可以采用，适用于批处理任务。用户只需配置数据大小（或数据条数）。
2. **按照时间，一个小时间片内数据封装为一个eBag。**这种策略不考虑数据大小的问题，每隔一段时间就封装一个eBag发出，如果没有输入则不发送数据。这样是为了满足流处理的实时性要求，避免无限等待。这种策略一般适用在输入数据量低且平均的应用中。用户只需配置时间片大小。
3. **按照数据大小，有超时时间。**当数据到达一定大小（或者一定数目的条数）后则封装为一个eBag，同时维护一个计时器，若计时器超过超时时间，但数据仍未达到规定阈值，则将已接收并预处理的数据直接封装为一个eBag发出。这种策略可以既保证实时性，又一定地提高了数据吞吐量，可用于数据率高的流处理，高效地应对突发数据流。这里需要用户配置数据大小阈值和超时时间这两个参数。

不同的eBag封装策略实际上是对输入的数据流做分段。小的分段能够降低延迟，但同时增加了附加开销，使系统的吞吐量加大；反之，大的分段吞吐量高，但会增加延迟。最优化的分段大小取决于具体应用。这一版本的eBag封装策略由用户自定义，后续版本将根据输入动态地优化封装的粒度。

每一个eBag携带类型为EBag id的信息，eBag id:(epoch id , task id , seq num)。其中，epoch id声明该eBag的时间属性；task id声明该eBag的生成位置，加入该信息的目的是为了在消息重发指明eBag的生成位置；seq num在本epoch内唯一。由此可知，每个eBag携带的eBag id在全局范围内唯一。

## 编程模型

对于**算子**，我们提供了三种计算接口：Mapper、Reducer、Iterator。Map和Reduce节点分别需要实现map和reduce的计算，而iMap和iReduce节点除了继承mapper和reducer接口外，还需要继承和实现iterator接口，我们不允许同时继承mapper和reducer接口。三种接口的实现：

public interface Mapper {

public void map (epoch, in\_key, in\_value);

}

public interface Reducer {

public void reduce(epoch, im\_key, im\_value list);

}

public interface Iterater {

// e.g.

// key – vertex id of destination vertex

// value – msg sent to the vertex

public void process(epoch, key, in value);

}

例如：一个iMap节点，我们可以继承Mapper和Iterater接口，并对其中的map()和process()两个函数进行实现。

针对**输入适配**，实例化input adapter对象后调用其中的setEBagStrategy()函数来确定eBag的封装策略，这里提供三种重载类型：

Public abstract class InputAdapter{

InputAdapter ();

~InputAdapter ();

public setEBagStrategy (ItemUnit Size);

public setEBagStrategy (Time Interval);

public setEBagStrategy (ItemUnit Size, Time Interval);

public precompute ();

}

如果没有设定封装策略则默认使用第二种，并默认时间间隔为100ms。

另外，我们提供了一个precompute () 的可选方法，可以通过重写该方法来进行数据的预处理，包括翻译为算子接收的数据类型或简单的筛选等，如果没有重写则默认将输入的数据按封装策略打包封装并发送出去。

## 执行流程1: 基本流程

下面我们简要说明数据在系统中的处理流程：



图2 执行流程

**Input Adapter:**

1. 首先数据通过input adapter进入数据处理系统，Input adapter对数据进行预处理；
2. Input adapter根据预定的eBag封装策略进行eBag的封装，等待eBag到达封装条件，当条件满足时封装eBag，并根据current epoch和task id为eBag生成eBag id。
3. input adapter将带有eBag id的eBag发送给下游的算子；

注：eBag id由task id + sequence id 构成，其中task id唯一区分了一个task, sequence id进一步区分了该task产生的eBag.

**Processor：**

1. Processor维护本地current epoch（详见6.3），在收到eBag时，与本地current epoch比较
2. 相同，转到(2);
3. 大于current epoch，将eBag放入eBag缓存队列;
4. 小于current epoch，丢弃。
5. Processor Task根据用户定义的计算函数处理current epoch内的eBag。
   1. 对于Map Processor task，依次处理eBag中的每条key-value，产生新的key-values
   2. 对于Reduce Processor task，则缓存eBag，等待该epoch的所有eBag到达，然后merge-group，调用reducer依次处理每个key group.
6. processor Task在本epoch结束时，将处理结果批量封装和发送：processor task将处理结果根据预定的eBag封装策略打包，新的eBag id继承了current epoch，processor根据processor task写入task id并生成seq num，eBag封装完成后发送给后续的计算节点。

## 执行流程2：增量、迭代

如果计算作业包含了迭代计算过程，则执行流程稍复杂些。流程中，Input Adapter的计算和Processor对eBag的处理流程与“5.3基本计算执行流程”类似，这里不再赘述，这里仅强调迭代相关eBag的处理。



当processor Task处理完上游算子发送的上一个 epoch的所有消息之后，启动对current epoch内eBag的相应处理：

* 1. Processor Task在收到eBag时，与本地current epoch比较：
     1. 相同，转到(2);
     2. 大于current epoch，将eBag放入eBag缓存队列;
     3. 小于current epoch，丢弃。
  2. 调用map函数(imap节点中定义)或reduce函数(ireduce节点中定义，需等到该epoch内的所有eBag都到达之后才计算)对eBag进行运算，之后processor task判断是否满足迭代条件（用户定义），如果满足，转入第(3)步，否则，转入第(5)步。
  3. 当current epoch内本轮接收到的所有eBag**都处理完成后**，Processor Task根据计算结果封装key-value，根据分发规则，发给本轮计算的task，转入第(4)步。
  4. Processor Task对接收到的key-value根据迭代函数进行处理，计算完成后，processor task判断是否到达结束条件（用户定义），如果到了，转入(5)，否则，转入(3)。
  5. processor Task在本epoch结束时，将处理结果批量封装和发送。processor task将处理结果根据预定的eBag封装策略打包，新的eBag id继承了current epoch，processor根据processor task写入task id，coordinator收到processor的申请后为eBag分配eBag id。eBag封装完成后发送给后续的计算节点，迭代计算结束。

## 持久化

我们默认数据是在内存中进行计算的，对于较小的流式数据可以只利用内存而不需固化到硬盘中（用户可以根据需要自己定义固化的操作），这样可以提升数据的计算效率。而对于较大的数据如在批量处理中的大数据块，我们会根据数据大小，当超过一定大小后才通过spill操作固化到硬盘中。

# 关键技术

## Architecture



如图，我们在系统实现时对于集群管理设置4层的管理结构：

**Cluster Manager：Cluster Manager**负责整个集群的状态监控和资源分配，Cluster Manager运行在一个head节点上；

1. 当job DAG提交时，Cluster Manager为其生成和维护一个Job Manager，并协调Job Manager分配DAG中的节点task到负载较低的worker上准备进行计算；
2. 在集群工作中，Cluster Manager需要定期访问每一个**supervisor**的状态，当一个物理节点停止工作时，Cluster Manager需要将其工作任务分配到其他物理节点上，并从最近的checkpoint开始重新执行。

Job Manager对应一个计算任务（Job DAG），主要负责job的管理工作：worker中task的数目（并行程度），Timer的管理和epoch分配，算子和存储位置的记录，定时checkpoint的管理，以及任务日志（log）等。

**Node Supervisor：**在每一个物理节点上运行一个**supervisor，**负责本地物理节点的状态管理和资源分配。supervisor管理所在物理节点上的worker，在任务分配时和任务完成时等时间节点上向Cluster Manager报告任务的执行状态，同时配合Cluster Manager进行任务的分配和回收。

**Worker：**在每一个物理节点上有多个**worker**负责执行计算任务，每个节点上的worker数量是预先分配和固定的。

## 并行化

**Input adaptor的并行化：**默认情况下，由上层应用决定一个作业部署多少个并行的Input adaptor task，每个adaptor task接受一部分数据。

1. 如果计算作业接受和处理流式数据，则每个adaptor task根据分发规则接收一部分数据，根据用户定义进行处理，然后发送数据给后续算子。
2. 如果计算作业接收并处理批量数据（比如存储在HDFS中的文件），则可以根据用户定义将数据分成若干block，同时生成相应数量的adaptor task，每个adaptor task处理一个block。

**Processor的并行化：**默认情况下，由上层应用决定一个作业部署多少个并行的Processor task，每个processor task负责存储与处理一部分数据。Processor并行化与数据划分相关联，对应一个数据Partitioning函数，包括Range Partitioning, Hash Partitioning等，允许用户自定义Partitioning函数。

**Processor Task由worker调用执行。每个物理节点的**worker数量提前设定和分配，由Cluster Manager根据物理节点的资源进行配置。

## Epoch

我们在这个大数据处理模型中使用epoch的概念作为时间戳，用于eBag的标记和计算的控制。

**Epoch的生成：**Epoch的配置由Job Manager统一计时并分配，Job Manager中由Timer负责计时，对于一个时间段的所有eBag给定一个相同的epoch，每经过一个时间段则epoch数加一，表示进入下一个时间阶段。

是epoch和eBag是一对多的关系，由于input adapter的多输入源以及自身的并行性使得一个epoch时间段内产生多个eBag，eBag由eBag id唯一确定，而epoch只能确定一段时间内的eBag。

**Epoch控制Processor的计算：**对于每一个Processor Task，它需要维护三个epoch信息：current epoch、received epoch、emitted epoch，分别记录本task当前的epoch、接收到的epoch、发送的最大epoch；假定processor的current epoch = n，应满足如下条件方可以更新为n+1：

1. 上游Processor（所有task）的current epoch ≥ n+1；
2. 本Processor task已处理完所有epoch=n的eBag；
3. 本Processor task已完成epoch=n的迭代计算；
4. 本processor task已将新生成的所有epoch = n的eBag发送至下游节点；

我们可以通过epoch对任务的执行过程进行管理，如：计算过程的分批，时间阶段的划分和同步等。

## 容错与故障恢复

由于算子的执行有先有后、有快有慢，我们将采用异步Checkpoint机制，每个Processor各自完成checkpoint的。

Checkpoint的时机是processor执行完一个epoch之后，保存每个节点的计算状态（current epoch, current eBag, 发送队列中的消息，processor Task的内存状态等）至存储系统上（可靠存储，比如多副本）。

Job Manager会根据epoch以及checkpoint策略，通知processor task实现checkpoint，processor task收到通知之后，按要求在epoch结束时实现checkpoint.

当节点进行checkpoint操作时，processor task不再执行计算任务，checkpoint完成后再继续。

当出现节点失效时则由Cluster Manager将整个物理节点上的计算任务重新分配给其他节点，并找到最近的checkpoint重新执行（仅仅是需要重新执行的任务以及相关的任务）。

### 可靠传输

我们在两个相邻节点之间实现可靠的数据传输. 每个input adapter task和Processor task维护本地一个eBag queue,用于记录需要发送的eBag; 同时还维护一个eBag log, 记录已经发送的eBag. 当input adapter/Processor生成一个eBag时, 该eBag首先被缓存到这个eBag queue中以备发送, 发送成功之后转入eBag log中, 以备故障重发。当确认后续节点发生故障, 需要向新节点重发消息时, 从eBag log中取出对应的消息,重新发送. 每次故障后恢复到上一个checkpoint（通常是epoch结束时），因此，需向新计算任务发送未完成的epoch的所有eBag。

Epoch=x

Epoch=y

图4 可靠消息

消息以checkpoint为粒度进行确认(因为故障恢复是按照checkpoint实现的). 在节点做完checkpoint之后, 可以向上游节点发送确认消息. 例如图4中,当下游节点B做完checkpoint之后, 向A发送确认消息. 由于checkpoint也是以epoch为单位实现的, 因此, 确认消息只需要包含B的current epoch即可. 当A收到来自B的确认消息之后,从eBag log中删除所有发送给B且比current epoch古老的eBag.

### 任务失败

processor失败之后, 将导致processor task的内存数据丢失, 消息丢失, 以及进展状态丢失. 对此, 我们利用checkpoint记录这些信息, 并且在故障发生之后, 重新启动processor task, 恢复最新的checkpoint, 继续执行. 此时, 上游节点可以重发eBag log中的eBag, 以便于计算.

**Input adapter任务失败：**恢复input adapter, 恢复eBag log即可。

### 任务槽(slot) 故障

**Job ManagerCluster Manager失败：**当Job Manager发生失效时，它上面保存的所有状态都会丢失，因此需要从最近的checkpoint开始恢复，将所有的节点恢复到这个checkpoint开始重新执行。

**Worker失败:** 每个worker中包含数个Processor或者input adapter任务。supervisor负责监控这些任务，当worker失败后，supervisor会尝试在本机重启它，如果重启成功则恢复没个processor task继续执行。

**Node Supervisor失败:** 由Cluster Manager监控状态，并根据失效情况及时重启，将其中的worker重新启动并执行失效的eBag。

**Cluster ManagerCluster Manager失败：**Head节点重启Cluster Manager，并从checkpoint开始重新执行。需要外部监控来及时重启。

### 集群节点（机器）故障

**Compute Node节点失败：**如果集群中的机器出现故障，Cluster Manager会将此机器上所有正在运行的任务转移到其他可用的机器上运行，并执行失效的eBag。

**Head Node节点失败：**当出现Head节点失效时，我们需要启动备份Cluster Manager，并从最近的checkpoint开始重新计算所有的任务。

# 应用

## Wordcount

和MapReduce的计算模型大体一致，区别在于同时可以完成批量处理和实时处理。



计算流程描述：

1. 系统接收输入的原始语句，交给InputAdapter处理。
   1. 如果是批处理，则输入是文本文件；
   2. 如果是流式计算，则输入是连续的tuple（小文本等）。
2. InputAdapter根据eBag封装策略将原始语句封装成eBag。
   1. 如果是批处理，则将文本文件的一个数据块（64MB）作为一个eBag；
   2. 如果是流式处理，则根据时间将一个时间片内的tuple封装为一个eBag。
3. A为map节点，输入为原始语句（文本），输出为（word，1）的k-v对。
4. B为reduce节点，输入为已按word单词序排序的列表（key-values list），输出为各个单词的个数。
   1. 如果是批处理，则将每个key（word）对应的value（即1）做加和计算，并输出结果。
   2. 如果是流式计算，维护一个word的频度队列。对于每一个key（word），先查看是否存在队列中，如果是则给队列中的对应word的value（即数量）加1，否则加入一个新的key到队列中，value为1。输出这个队列。

## TunkRank微博用户影响力计算

* 样例描述：利用微博用户之间的@（提及）关系，计算微博用户的影响力，筛选出影响力最大的k个用户。
  + 系统输入：微博服务器实时接收到的原始微博。
  + 系统输出：近一段时间内影响力最大的k个用户。
  + 影响力计算公式：TunkRank



* + eBag封装策略：同时规定eBag的时间间隔及其携带的最大数据量，即为public setEBagStrategy (ItemUnit Size, Time Interval)这一函数传入实参，保证结果输出的实时性，同时保证对突发数据流有足够的应对能力。
* 计算逻辑
  + 计算逻辑如图所示



计算流程描述：

1. 系统实时接收微博数据，交给InputAdapter处理。这里每一条微博作为一个tuple。
2. InputAdapter提取原始微博中的@关系，并格式化为(user\_from,user\_to)的key-value格式，根据eBag封装策略对格式化后的数据（即多个key-value对）进行封装，以eBag为单位输出。
3. A节点是迭代iMap节点，完成影响力计算工作，执行过程如下
   1. 根据（user\_from，user\_to）的信息，修改user\_to对应的user的影响力信息，并发送给自己msg声明影响力变化需要启动迭代运算。
   2. 当当前epoch的所有数据接收并处理完成后，对图的更新做增量式的迭代运算，迭代中输入输出为本节点的状态值变化，即该用户的影响力指标；
   3. 迭代到达终止条件,即图中各节点的值不变（或者超越迭代轮数），输出图中各个节点的值，即各用户的影响力指标，封装为新的eBag，发送到下游节点B。
4. B节点是map节点，它接收图中各节点的状态值，key为用户，value为其对应的节点状态值。输出新的k-v对，key为节点状态值，value为其对应的用户。
5. C节点是reduce节点，它接收已按节点状态值（即用户的影响力指标）排序的用户列表，输出其中前k个用户名。

# 附录1 后续版本功能

### 迭代计算由同步处理改为异步处理

迭代穿插在流数据的计算过程中；

### 迭代计算进一步改为事件触发

事件可以是：“完成一个epoch”、“处理的消息数量达到了阈值”、“数据对象改变到达了阈值”、“收到了Job manager的迭代通知”等。

### 默认的任务并发程度

### 根据负载调整任务并发数量

### 弹性替换机制

为了使得模型具有更好的灵活性，我们将在后续版本中实现节点的动态替换，从而实现对于并行度、计算过程分离的动态调整配置。

### EBag封装策略优化

根据任务进行自动配置

默认策略的优化，调整参数

### 数据一致性

# 附录 删除记录

**Msg：**在迭代节点中使用，逻辑图中迭代节点向自身发送的信息，对应于图计算中节点间传递的信息，用来启动迭代运算。

**EBag table：**维护在Job manager中，注册所有eBag信息（eBag id、父eBag等），用于跟踪EBag的进展。定时随checkpoint固化到硬盘，过一段时间后被清理（垃圾回收）。