文章编号: 2095-2163(2021)09-0006-06 中图分类号: TP311 文献标志码: A

基于代价模型的多平台分析任务流调度优化

徐超一[1], 刘晓清[1], 顾 淼[2], 王 巍[2]
(1 复旦大学 计算机科学技术学院, 上海 200441; 2 新浪, 北京 100193)

摘 要: 分析任务流的调度是人们关注的热点问题之一。在无法提前得知各子任务资源和时间开销以及算子间传输开销的情况下,现有研究缺少较好的调度方法,也无法很好地处理多平台环境下的任务调度。为此,本文提出了一种基于启发式规则优化的拓扑调度算法。该算法通过对同平台任务和后继任务优先选择的规则,对任务流调度顺序进行优化;其次,结合 Spark 平台下数据分析任务常见的 SQL 算子和机器学习算子的代价模型,有效对任务的开销做出估计。实验表明,调度优化算法能有效节约时间开销和内存开销,代价模型的估计也较为准确。

关键词: 任务流调度; 有向无环图; 拓扑排序; 代价模型; 启发式

Multi-platform analysis task flow scheduling optimization based on cost model

XU Chaoyi[1], LIU Xiaoqing[1], GU Miao[2], WANG Wei[2]
(1 Computer Science and Technology School, Fudan University, Shanghai 200441, China; 2 Sina, Beijing 100193, China)

【Abstract】 The task flow scheduling problem is one of the hot issues that people pay attention to. Existing research lacks a scheduling method when the resource and time cost of each subtask and the transmission cost between operators cannot be known in advance, and cannot handle task scheduling in a multi-platform environment well. For this reason, a topology scheduling algorithm based on heuristic rule optimization is proposed. The algorithm optimizes the task flow scheduling sequence through the rule of preferential selection of tasks on the same platform and subsequent tasks. Secondly, this article combines the cost model of common SQL operators and machine learning operators in data analysis tasks under the Spark platform, which can effectively estimate the cost of tasks. Experiments show that the scheduling optimization algorithm can effectively reduce time and memory costs, and the estimation results of cost model is also more accurate.

【Key words】 task flow scheduling; Directed Acyclic Graph(DAG); topological sorting; cost model; heuristic

0 引 言

数据量爆炸式增长的同时,数据分析的重要性也日益凸显。多年来,数据分析的各类需求日益旺盛,人们对数据分析的要求也不再局限于更强的数据处理能力,多个计算(分析)平台共同进行一个分析任务流的需求也随之而生。

现实中,人们经常根据数据分析需求的不同,来确定分析任务所使用的执行平台,甚至一个分析任务流中的分析算子,也可能运行在不同的计算平台上。分析任务流在不同平台之间的调度成为人们关注的一个重要问题。

RHEEM[1]是卡塔尔大学开源的跨平台数据处理系统,其支持在一个任务流中自动调用多种平台,来优化处理时间和处理性能。RHEEM 借由内部的优化器,通过代价模型自动为任务流中的算子选择平台,获取最优的执行计划。Apache Beam[2]为开源的统一编程模型,用于进行跨平台的大数据分析处理。Beam 是基于 Google 的 Dataflow Model[3]论文的一种实现。通过对数据分析的多维度规范和总结,构成了一套编程范式,实现了不同平台间的统一,但并没有关注任务流的调度问题。Kumar 等人基于 Actors 模型[4]研发了一个带图形界面的任务流系统 Amber[5]。但 Amber 的研究侧重于任务流执行过程中的实时调试,而不关注任务的调度顺序。而目前针对单个任务流调度的研究已经趋于成熟,HEFT[6]、CPOP[7]、PETS[8]等算法,以较高的调度效率被广泛接受。但其都需要知道每个算子的时间、资源开销以及算子间的传输开销,并且都没有考虑多平台的条件。

为此,本文研究了多平台环境下,数据分析任务流的调度和优化问题。具体研究内容包括:根据多

基金项目: 国家重点研发计划(2018YFC0830900)。

作者简介: 徐超一(1996-),男,硕士研究生,主要研究方向:大数据处理; 刘晓清(1964-),男,博士,高级工程师,主要研究方向:神经网络、深度学习、图像处理等。

通讯作者: 刘晓清 Email:xqliucs@fudan.edu.cn

收稿日期: 2021-06-19

平台的特性,提出了一种基于启发式规则优化的拓扑调度算法,来完成任务流调度;针对 SQL 算子加入了代价模型,能对任务流的 SQL 分析开销做出估计;训练了 GBDT 树,为用户动态选择机器学习任务的运行平台并确定开销;通过实验,验证了调度算法的优化性能和代价模型的准确率。

1　问题描述

通常来说,一个大任务可以拆分成一个子任务的集合(不同的子任务间存在一定的依赖关系)。此时,可以通过一个有向无环图 DAG(Directed Acyclic Graph)的形式来表示这些关系,形成了任务的 DAG 模型。因此,任务流的调度问题就转化为 DAG 的调度问题。显然,当考虑任务调度时,首要满足的是子任务之间的依赖关系。

1.1　拓扑排序

拓扑排序,是指对一个有向无环图 G 进行排序。对于图中边的集合 $E(G)$,若存在边 $<u,v> \in E(G)$,则在排序序列中,顶点 u 一定出现在顶点 v 之前。可见,拓扑排序的特性可以很好地契合子任务之间的依赖限制。通过对 DAG 做拓扑排序就可以得到一个满足任务流依赖关系的执行顺序。

Kahn 算法[9]是进行拓扑排序的常见算法,由 Kahn 在 1962 年提出。其中心思路是:每次取出一个没有先驱节点(即入度等于 0)的节点,将其放入排序序列中,然后将这个节点的所有后继节点的入度减一,重复这一过程直至排序完成。该算法的时间复杂度为 $O(E+V)$。但需要注意的是:有效的拓扑序列并不是唯一的,每次使用 Kahn 算法得到的拓扑序列也不一定是一致的。

1.2　拓扑排序的局限性

如图 1 所示,给出的一种存在两个不同平台算子的任务流。从 A 算子读取输入后,分别通过后续的其它算子计算,直至结束。对于这样一个任务流,其拓扑序列显然不唯一。可能的序列有(A,B,C,D,E,F,G,H,I,J,K,L),(A,B,D,F,C,E,G,I,J,K,L,H) 等等。在非常多的可能序列中,再次考虑两种比较极端的调度序列。第一种是序列L_1(A,B,C,D,E,J,F,G,K,I,L,H),第二种是序列L_2(A,B,D,F,C,E,G,H,I,J,K,L)。显然,序列 L_1 和序列 L_2 都是满足拓扑排序的序列,但序列 L_1 的调度效果是一个平台 P1 的算子和平台 P2 的算子交错运行。而序列 L_2 的调度效果是优先将满足依赖的同平台算子全部调度完后,再考虑调度其他平台的算子。

对于序列 L_1 来说,频繁的平台切换会带来额外的切换开销。更关键的是在进行平台切换的同时,需要将之前的计算结果保存,又会带来额外的内存开销。而过多的中间结果缓存会在一定程度上影响后续节点的计算速度。以 Spark 平台为例,过多的中间结果被缓存,会导致可用内存不足。此时,在读入当前节点数据时,需要进行 GC 和内存置换,从而会降低任务的处理速度。另一方面,如果一个平台内的算子操作可以连续进行,那么计算的中间结果可以用平台的内部结构表示,不仅使得整体的运算速度变快,也使整个流程的类型十分安全,可以应对复杂的计算流程,大大提高系统的鲁棒性。而例如 Spark 这样的平台,其原生数据结构还使得运算结果的展示具有极高的灵活性。

可以想象,当任务流的复杂度提升,或是任务流中平台数量增加,这两种调度序列所带来的开销差别是巨大的。

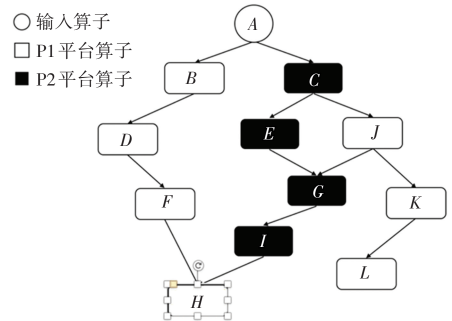

图 1　多平台任务流示例

Fig. 1　Example of multi-platform workflow

1.3　问题定义

朴素的拓扑排序可以解决任务流的调度问题,但在任务流的整体开销上,普通的拓扑排序是存在一定问题的。由于拓扑序列是不唯一的,对于一个给定的任务流 J,使用不同的拓扑序列对其进行调度,最终整体的内存和时间开销是不同的。

因此,该问题的描述是给定一个有向无环图 $G(V,E)$,图中节点 E 表示系统中不同平台的算子,图中有向边 V 表示算子之间的依赖关系。在满足依赖关系约束的情况下,本文希望找到一种节点序列,使得通过该顺序执行任务流时,较少的其它中间结果被缓存,且进行的平台切换最少,从而能优化整体的时间开销。

2　算法描述

针对这一问题,如果预先将所有的拓扑序列求

出,再一一进行比较,选出最优解,算法的复杂度会过高。所以本文选择通过启发式规则来求解。

2.1 基于启发式规则的拓扑调度

本算法思路:先将输入节点置入结果序列,然后将其指向的节点入度减1,每次选择下一个节点(入度为0的节点)时,遵循三条规则:

(1)总是优先选择与之前节点同平台的节点;

(2)如果有多个同平台的节点,优先选择当前节点的后继节点;

(3)如果不存在同一平台的节点,则选择任意满足依赖约束的节点。算法伪代码如算法1所示。

算法1 基于启发式规则优化的拓扑调度算法

输入 有向无环图 $G = (V,E)$,S 为入度为0的节点集合

输出 调度序列 L

```
L = { }
While S is not empty do
//启发式规则优化
if S has the same-platform-node with tail of L
if S has the successor of the tail of L
   remove the successor n from S
   add n to L
      else
remove the same-platform-node n' from S
add n' to L
else
remove another node k from S
add k to L
for each node m with edge e from n to m do
remove edge e from G
if m has no other incoming edges then
insert m to S
    return L
```

仍以图1为例,序列 $L_1(A,B,C,D,E,J,F,G,K,I,L,H)$ 和序列 $L_2(A,B,D,F,C,E,G,I,H,J,K,L)$。其中序列 L_2 为本算法得出的调度序列。输入算子 A 与后继算子的切换不计入考量,L_1 序列总共切换了8次平台,L_2 序列总共切换了2次。同时可以看出,L_1 序列中,在计算大部分节点时,都有无关的中间结果被缓存。如在计算 G 节点时,节点 F 和节点 G 的结果缓存,则增加了内存和时间开销。而在 L_2 序列中,基本保证了任务能够以一种近乎深度遍历的方式执行,减少了与当前计算无关的中间结果的缓存。

如算法1所示,本算法对图中的每个节点和每条边都会进行一次遍历。因此,本算法的复杂度为 $O(E + V)$,其中 E 为图中的节点数,V 为图中的边数。

2.2 代价模型

本文主要考虑的任务流算子为SQL类算子和机器学习算子,这些算子的主要运行平台是Spark和Python。因此,本文针对这两种情况分别加入相应的代价模型。

2.2.1 针对SparkSQL的代价模型

代价模型涵盖了3个基本的SQL操作:Projection、Selection和Join。

首先,系统收集一些SparkSQL相关的物理参数。参考Baldacci[10]等人的工作,系统收集的物理参数见表1。

表1 物理参数收集表

Tab. 1 Collection of physical parameter

参数	含义
N	Spark的服务器数
NC	每个服务器分配给Spark的核心数
$\delta_r(Para)$	Para并行度下读磁盘的吞吐量(MB/s)
$\delta_w(Para)$	Para并行度下写磁盘的吞吐量(MB/s)
$\rho_i(NC)$	不同服务器间网络的吞吐量(MB/s)
B	Shuffle用到的bucket数量

根据表1可以初步得到公式(1)和公式(2):

$$Read(Size) = \frac{Size}{\delta_r(NC)} \tag{1}$$

$$Write(Size) = \frac{Size}{\delta_w(NC)} \tag{2}$$

其中,$Read(RSize)$ 为根据读入大小得到的读入时间,$Write(WSize)$ 为根据写出大小得到的写出时间。

计算执行开销最简单的方式就是将其运行一遍,但这样就失去了代价模型的意义。事实上,代价模型不需要知道一个精确的代价,而只需要一个估计值。所以,代价模型可以对结果的数据量进行估计,再通过数据量的估算值来推断执行的代价。

在查询语句中,用户常常会使用一些过滤条件(即本文中提到的Filter操作),在SystemR[11]中对过滤条件的估算只考虑到了数据连续且分布均匀的情况。但现实中的数据往往不是连续均匀分布的。所以考虑数据的分布情况,对数据量的估计是至关

重要的。

其中，直方图是一种能够表示数据分布的统计方式。其通过分桶策略对数据做出划分，从而得到大致的分布情况。本系统选择了等深直方图来了解数据分布情况。与普通的等宽直方图不同，等深直方图尽可能的保证桶的深度相同。Piatetsky-Shapiro[12] 的研究指出，等深直方图的鲁棒性更强。

直方图的构建需要有序数据。考虑到数据量增大后导致的排序开销，所以本系统使用蓄水池采样，采样后再进行排序和直方图的构建。

系统将过滤条件分为 3 类：单列的范围查询、单列的等值查询、多列查询。下面将分别介绍 3 种情况的估计方法。

(1) 单列范围查询。对于单列的范围查询，可以通过等深直方图来进行估计。对于给定的一个范围查询，只需要知道其覆盖范围内的所有桶的深度即可。如果遇到桶的范围与查询范围有部分交集的情况，可以交集占桶大小的比例再做一次估算。此时，需要假设数据的分布是均匀且连续的。对于其它类型的数据，一般是将其映射成数字后再计算比例。

(2) 等值查询。对于等值查询，需要知道记录出现的频率。对于一般情况下频率的计算，人们倾向于使用 HashMap 来统计。而当数据量非常大时，一则要求的内存非常大，二则当 HashMap 的冲突很高时，时间复杂度的上升，导致无法满足实时性的需要。

本系统使用了 Count-Min Sketch[13] 算法，其是一种可以处理等值查询的方法，可以提供很强的准确性保证。该算法的基本思路是维护一个初始为 0 的 $D \times W$ 大小的数组。对于数据中出现的每一个值，分别用 D 个独立的哈希函数进行映射和计数。查询频率时，依旧对其进行 D 次哈希，找到每一行中对应的计数值，再取其中的最小值作为估计值。

Count-Min Sketch 可以看作布隆过滤器在统计方面的一个变形。其缺点是估计值总是大于等于真实值。

(3) 多列查询。本系统假设不同列之间是相互独立的，只需要把不同列的过滤结果相乘即可。

综上，可以得到一个函数 $Filter(cols, type)$。

其中，$cols$ 表示过滤的列，$type$ 表示过滤的种类。可得对 Project 操作的估计，如公式(3)所示：

$$Project(projCols, all) = \frac{\sum_{attr \in projCols} attr.Size}{\sum_{attr \in all} attr.Size} \tag{3}$$

其中，$projCols$ 表示被选中的列；all 表示表中所有列；$attr.Size$ 表示该列的平均大小。

接下来可以考虑就可以完成代价模型，本文考虑 3 种任务。

第一种任务类型是全表扫描任务，记为 SCAN。SCAN 任务可以包含 Filter、Project 和 Aggregate 操作。在 SparkSQL 中，Filter 操作和 Project 操作被 Spark 的优化器 Catalyst，通过谓词下推和列值剪裁来优化执行，减小无用的元组和列对整体开销的影响。此时，数据大小的估计如公式(4)所示。

$$WSize = RDDSize \cdot Project(projCols, all) \cdot Filter(projCols, type) \tag{4}$$

若不存在聚合操作，则整个任务可以在一个管道中边读入，边写出，整体开销如公式(5)所示。而存在聚合操作时，任务的写出必须在所有数据被读入后才能进行。此时，整体开销如公式(6)所示。

$$SCAN(Table) = \frac{RNum}{N \cdot NC} \cdot \mathrm{Max}(Read(RSize), Write(WSize)) \tag{5}$$

$$SCAN(Table) = \frac{RNum}{N \cdot NC} \cdot (Read(RSize) + Write(WSize)) \tag{6}$$

$$RNum = \frac{TableSize}{RDDSize} \tag{7}$$

其中，$TableSize$ 为表大小，$RDDSize$ 为 Spark 的 RDD 分区大小。若不存在写出操作，$Write(WSize)$ 取 0。

如果涉及 Join 操作，SparkSQL 将根据不同情况，使用以下几种不同的 Join 方式：

(1) Broadcast Hash Join 应用于小表（默认阈值为 10 MB）和大表之间的 Join。使用 Broadcast 的方式来完成 Join 操作，牺牲空间换取时间。此时，通过将小表广播到每个运行节点上，避免了 Shuffle 带来的大量时间开销。

(2) Shuffle Hash Join 适合较小表和大表之间的 Join。如果小表的大小大于 10 MB，此时将小表广播出去会造成较大的数据冗余和带宽内存消耗，使得运行节点的压力较大。所以，SparkSQL 转为使用 Shuffle Hash join，通过 Join 的 key 将两张表进行分区，即 Shuffle 操作。集群内的每个工作节点都会参与 Shuffle 操作，每个工作节点处理每个 Bucket 的一部分，然后对每个分区内的记录进行 Hash Join 的操作。

(3) Sort Merge Join 则适合两张大表之间的 Join。Hash Join 的方法是将其中一张表完全读入内存中，然后使用哈希的方法对另一张表进行探测和

连接。而当两张表都较大的情况下,使用哈希方法对内存的压力过大。此时 SparkSQL 通过 Join Key 将两张表进行 Shuffle 分区,以便后续的分布式处理,然后分别对每个分区进行排序和合并。

第二种任务类型是 Broadcast Join 任务。该任务的开销可以用函数 $BJ(\ )$ 表示。在进行 Broadcast Join 时,大表仍然通过一个 SCAN 任务读入,而小表要进行广播,所以不需要写操作。Broadcast 过程的开销,参考文献[10]的研究,得到公式(8)。

$$Broadcast(Size) = \frac{Size}{\rho_i(NC)} + \frac{Size \cdot N \cdot NC}{\rho_i(1)} \tag{8}$$

由于整个 Broadcast Join 是在内存中,通过 Hash 的方式来完成,速度非常快,瓶颈主要体现在最后写出的速度上。所以 Broadcast Join 的开销函数如公式(9)所示:

$$BJ(Table_1, Table_2) = \frac{PartitionNum}{N \cdot NC} \cdot Write(WSize) \tag{9}$$

其中, $PartitionNum$ 为 Spark 中设置的分区数。

第三种任务类型是 Shuffle Join。因为 Shuffle Hash Join 和 Sort Merge Join 的开销瓶颈都是 Shuffle 阶段,所以其开销都可以通过 Shuffle Join 任务来描述。该任务的开销可以用函数 $SJ(\ )$ 来表示。在进行 Shuffle Join 时,显然 Shuffle 的过程是瓶颈所在。另外,Shuffle Join 只有在整个分区的数据都被读入后才能进行,不能边读边写。但是在第一阶段的 Shuffle 过程中,数据是一边被分配到分区,一边被读取的。Shuffle 阶段的开销如公式(10)所示:

$$Shuffle(\ ) = Max\left(\frac{ExecSize}{\delta_r(N \cdot \mathrm{NC})}, \frac{ExecSize}{\rho_i(\mathrm{NC})}\right) \tag{10}$$

其中, $ExecSize = \frac{Size}{N}$, $ExecSize$ 表示的是每个节点处理的每个 Bucket 的大小。

对于 Shuffle Join 阶段的开销如公式(11)所示:

$$SJ(Table_1, Table_2) = \frac{B}{N \cdot NC} \cdot \left(Shuffle\left(\frac{Table_1.Size + Table_2.Size}{B}\right) + Write(WSize)\right) \tag{11}$$

综上所述,不同任务的开销可以通过上述 3 种函数的组合来计算。

2.2.2 针对机器学习任务的代价模型

由于本文考虑的机器学习任务的运行平台包括 Spark 和 Python(Scikit-Learn),因此会在确定运行平台后完成对机器学习任务的开销估计。

在确定了运行平台之后,就可以根据硬件信息、平台选择、任务种类(分类、聚类、回归)和训练数据的维度、数量等信息估计出这个机器学习分析算子的大致时间开销,可通过一个回归任务来完成这一开销估计。考虑到 GBDT(Gradient Boosting Decision Tree)的本质是回归树,且具有很强的泛化能力,因此本文使用 GBDT 回归树来完成对机器学习算子时间开销的估计。

3 实验结果与分析

本文在 Spark 集群上进行实验,其中包括 1 个 master 节点和 2 个 worker 节点。节点的硬件配置如下: Intel(R) Xeon(R) Silver 4208@2.10 GHz,24 核;64 GB 内存;8 TB 硬盘。软件配置为:Linux Ubuntu 18.04、Spark-2.3.1、Python 3.6.8 版本。

用于数据分析流程的训练数据为 Numpy 随机生成。而用于文本分析流程的数据,来源于新浪公司 CMS 内容管理系统真实发布的新闻文章(为 2019.01.01~2019.04.22 发布在新浪新闻娱乐频道的所有新闻文章),去重后共有2 172 925篇。实验时先根据关键词筛选出1 000篇文章进行训练。

第一组实验:测试调度算法的优化对系统性能的影响。

实验比较了朴素拓扑调度和基于启发式规则优化的拓扑调度在任务流调度中的开销差异。

本文首先通过 Numpy 生成了 12 GB 的训练数据,并限制系统内存为 10 GB,模拟复杂任务下的情况,并使用多种不同的算子组合构成任务流,进行实验。其中一种任务流如图 2 所示。实验结果见表 2。

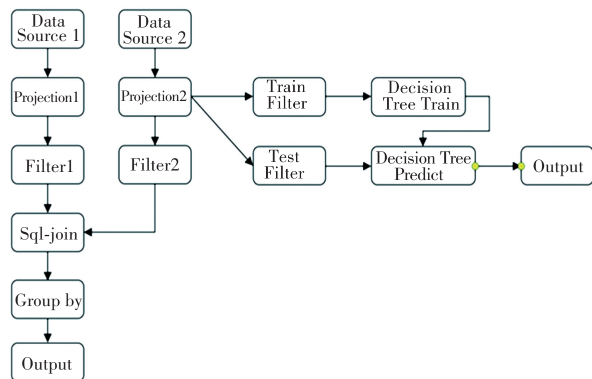

图 2 实验任务流示例

Fig. 2 Example of experiment workflow

表 2　调度性能测试

Tab. 2　Experiment of scheduling performance

调度算法	Spark 平均内存峰值开销/GB	平均时间开销/s
朴素拓扑调度	9.7	155.8
基于启发式规则优化的拓扑调度	7.9	143.3

第二组实验：测试代价模型的准确率。

本文在 TPC-H 的标准下，生成了 12 GB 的表数据。分别测试了不含 Join 操作的简单 SQL 和包含了 Broadcast Join 或 Shuffle Join 操作的稍复杂 SQL 的实际执行时间，并将之与代价模型给出的预估时间进行对比。实验结果见表 3。

表 3　代价模型准确性测试

Tab. 3　Experiment of the accuracy of cost model

查询类型	查询平均时长/s	平均误差率/%
不带 Join	24	25
Broadcast Join	22	28
Shuffle Join	58	33

误差率的定义如公式(12)所示：

$$误差率 = \frac{|实际时间 - 预估时间|}{实际时间} \times 100\% \quad (12)$$

由表 3 中可以看出，Join 操作带来的复杂性提升，使得不含 Join 的简单 SQL 的估计更加准确，对带 Join 的 SQL 的开销估计则稍微有所下降。而在两种 Join 类型之间，由于 Shuffle Join 涉及两张表的分桶和更多的数据传输，其过程更加复杂，使得这种类型下的平均误差率最高，达到了 33%。但 3 种情况下，代价模型的误差控制在 35%以内，达到了预期。

4　结束语

针对多平台条件的任务调度问题，本文提出了基于启发式规则优化的拓扑调度算法，并且结合代价模型完成了对 SparkSQL 任务和机器学习任务的开销估计。通过实验证明了调度算法的有效性。在后续工作中，可以结合历史运行数据，通过一个端到端的机器学习模型完善和改进系统的代价模型。

参考文献

[1] AGRAWAL D，OUZZANI M，PAPOTTI P，et al. RHEEM：enabling cross-platform data processing：may the big data be with you！[J]. Proceedings of the VLDB Endowment，2018，11(11)：1414-1427.

[2] Karau，Holden.［IEEE 2017 IEEE International Conference on Big Data (Big Data)-Boston，MA (2017.12.11-2017.12.14)］2017 IEEE International Conference on Big Data (Big Data)-Unifying the open big data world：The possibilities? of apache BEAM [C]// 2017 IEEE International Conference on Big Data (Big Data). IEEE，2017.

[3] AKIDAU T，SCHMIDT E，WHITTLE S，et al. The dataflow model：A practical approach to balancing correctness，latency，and cost in massive - scale，unbounded，out - of - order data processing[J]. Proceedings of the Vldb Endowment，2015，8(12)：1792-1803.

[4] AGHA G A. ACTORS - a model of concurrent computation in distributed systems[M]. University of Michigan，1985.

[5] KUMARA，WANG Z，NI S，et al. Amber：a debuggable dataflow system based on the actor model[J]. Proceedings of the VLDB Endowment，2020，13(5)：740-753.

[6] TOPCUOGLU H，HARIRI S，WU M Y. Performance-effective and low-complexity task scheduling for heterogeneous computing [J]. IEEE Transactions on Parallel and Distributed Systems，2002，13(3)：260-274.

[7] CANON L C，JEANNOT E，SAKELLARIOU R，et al. Comparative evaluation of the robustness of DAG scheduling heuristics[M]// SERGEI G. Grid computing：achievements and prospects. Berlin，Germany：Springer，2008：73-84.

[8] ILAVARASAN E，THAMBIDURAI P. Low complexity performance effective task scheduling algorithm for heterogeneous computing environments[J]. Journal of Computer Science，2007，3(2)：28-38.

[9] KAHN A B. Topological sorting of large networks[J]. Communications of the Acm，1962，5(11)：558-562.

[10] BaldacciL，Golfarelli M. A Cost Model for SPARK SQL[J]. IEEE Transactions on Knowledge and Data Engineering，2019，31(5)：819-832.

[11] ASTRAHAN M M，BLASGEN M W，CHAMBERLIN D D，et al. System R：Relational Approach to Database Management[J]. ACM Transactions on Database Systems，1976，1(2)：97-137.

[12] Piatetsky-Shapiro，Gregory，Connell，et al. Accurate estimation of the number of tuples satisfying a condition[J]. Acm Sigmod Record，1984.

[13] CORMODE G ，MUTHUKRISHNAN S. An Improved Data Stream Summary：The Count - Min Sketch and Its Applications [J]. Journal of Algorithms，2004，55(1)：58-75.