DTCC

数，造，未，来

第十二届中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2021

2021 年 10 月 18 日 - 20 日 | 北京国际会议中心

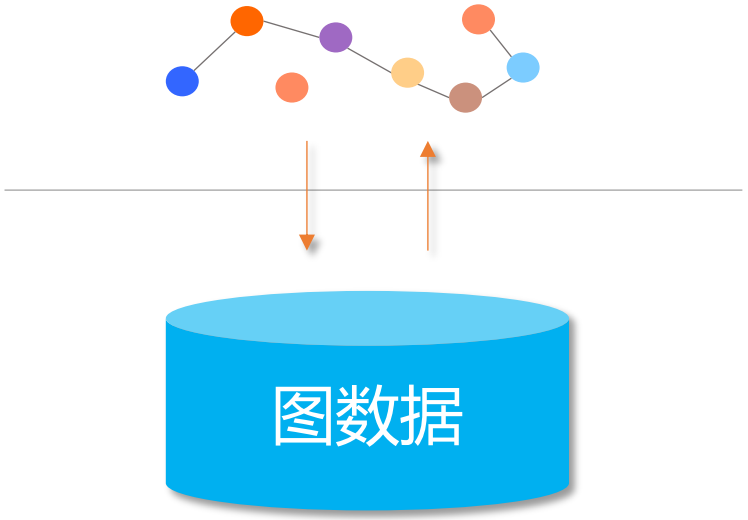
IT168.com ChinaUnix.net ITPUB

目录

1. 图计算是什么，能解决什么业务问题
2. 图计算与传统计算的区别，为什么需要图计算
3. 业界现状如何，大规模图计算所面临的挑战是什么
4. 如何应对挑战 -- 自研图计算
5. 图计算的架构与优势
6. 图算法开发与应用简介

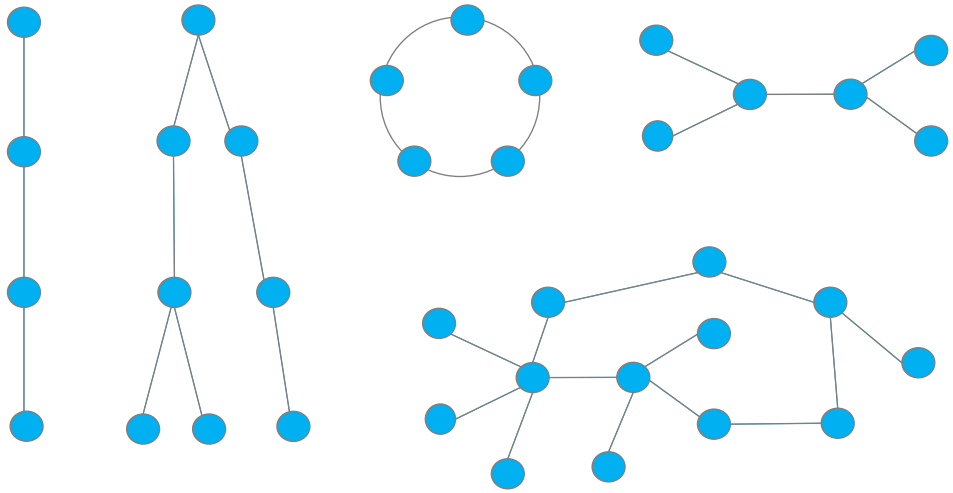
图数据库是什么？

- 存储、查询/分析图数据
- 灵活、高效处理图数据



图数据是什么样的数据？

- 由实体和关系组成的线、树、
- 环、叉、网等各种形状的数据



HugeGraph是什么？

百度自研图数据库

国内首家开源图数据库

HugeGraph

Apache 2 License

Apache Tinkerpop Enabled

大规模分布式图存储与图计算

HugeGraph应用分类

OLTP，即时查询，以图遍历为主

图查询

图计算

HugeGraph

HugeGraph应用分类

OLAP，离线分析，全图多轮迭代计算

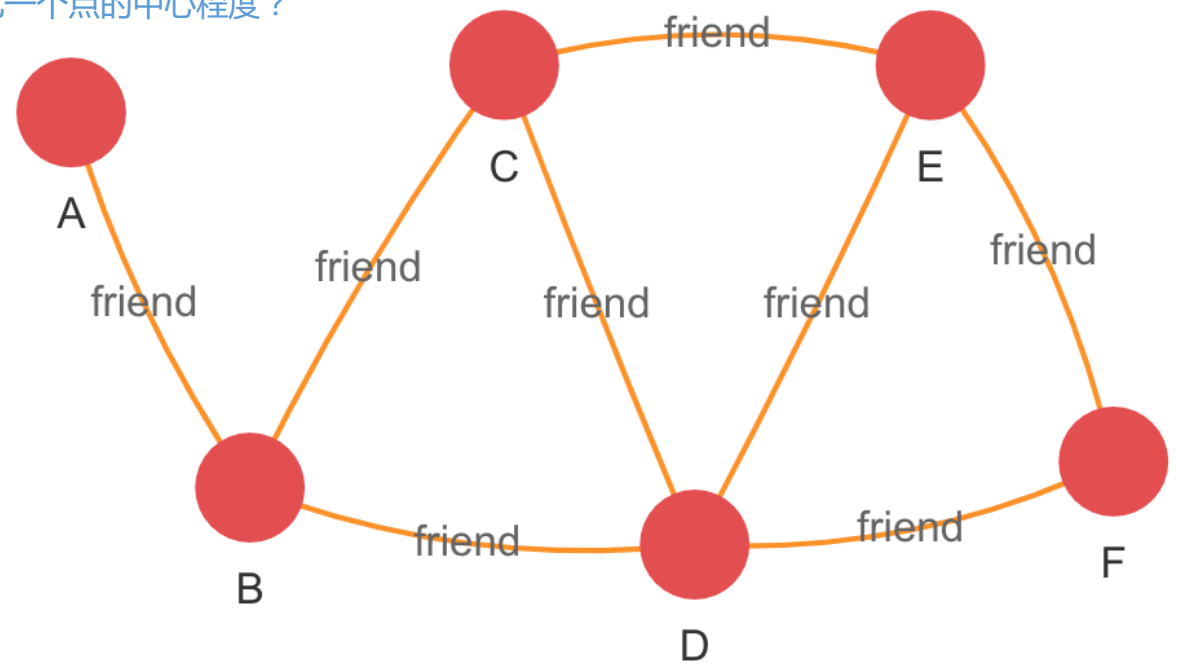
图查询　　图计算

HugeGraph

图计算是什么？

- 分析与抽象出复杂关联关系数据的特征，如实体/子图/路径等维度特征

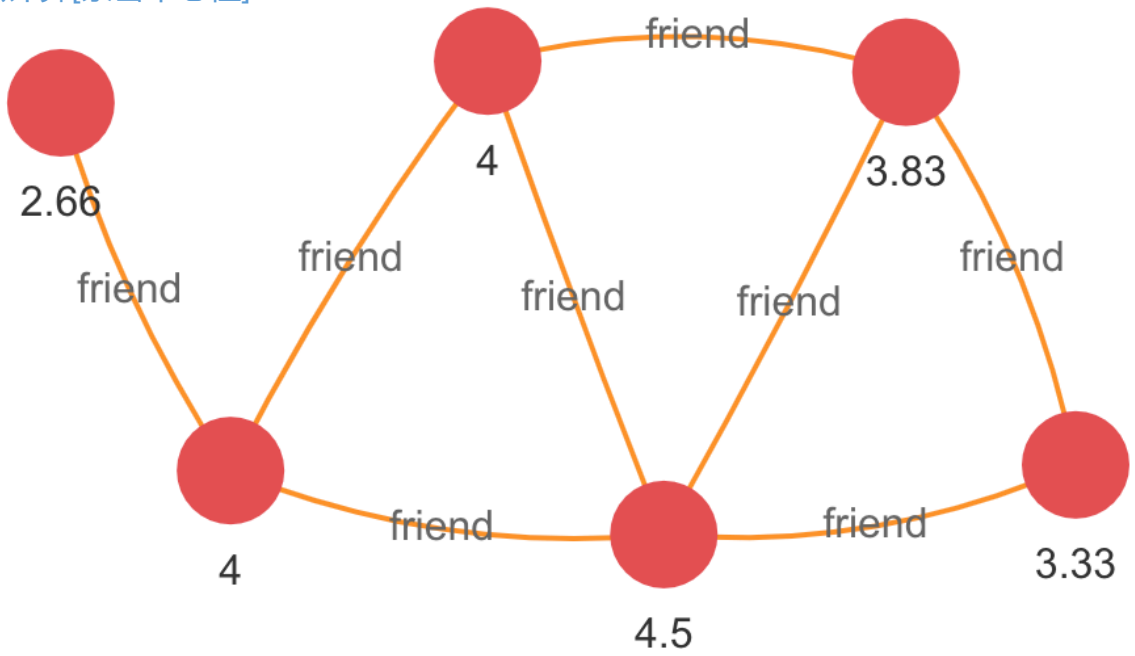
如何量化一个点的中心程度？



图计算是什么？

- 分析与抽象出复杂关联关系数据的特征，如实体/子图/路径等维度特征

通过图算法计算[紧密中心性]：



图计算背景

图计算是什么？

- 分析与抽象出复杂关联关系数据的特征，如实体/子图/路径等维度特征

图计算能解决什么业务问题？

- 金融风控、网络安全、情报关系、智能营销、智能推荐、智能运维等

图算法应用场景：

- 环路检测：循环担保
- 中介中心性：反洗钱
- 社区发现：团伙反欺诈

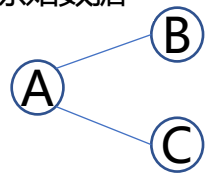
图计算与传统计算的区别

- 处理的数据维度多、关联关系错综复杂
- 非简单的单轮计算，而是具有依赖关系的多轮迭代
- 数据分布，逻辑上关联紧密的图无法简单按照单维度被划分到一起
- 分区不均问题，受现实世界中幂律分布而被放大，导致计算任务负载不均衡
- 多轮迭代决定上下游任务无法割裂，需要相互配合同步
- 每迭代中间结果量巨大，图算法复杂度O(n*(degree^depth))随深度指数增长

图计算业界现状

图的切分

原始数据



边切法（顶点和边存储在一起）

P1:	A -> B,C	C -> A
P2:	B -> A	

采用的框架

Pregel、Giraph

优点：计算方便　　缺点：数据不平衡

点切法（顶点的边存在多个分区）

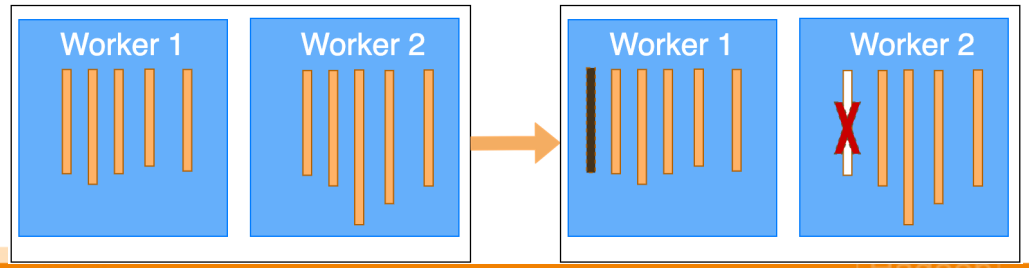
P1:	A -> B	B -> A
P2:	A -> C	C -> A

采用的框架

PowerGraph、
GraphX 、Gemini

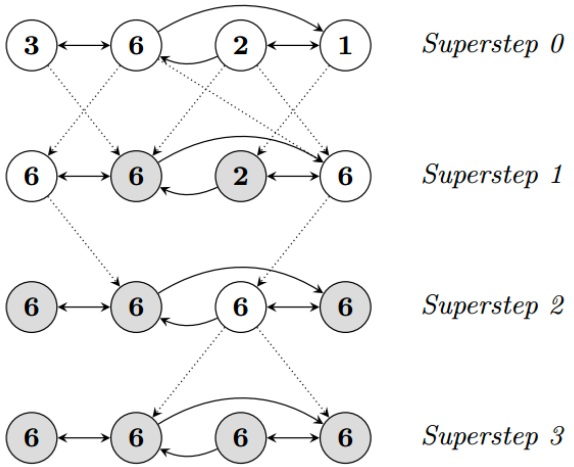
优点：边分布均衡　　缺点：镜像顶点值

HugeGraph：边切法 + Partition调整

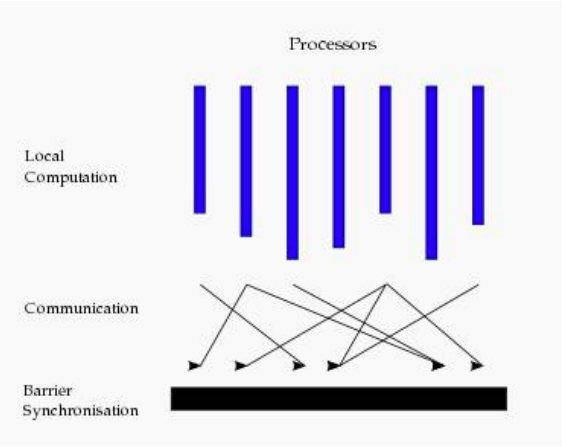


计算方式：迭代计算和发送消息

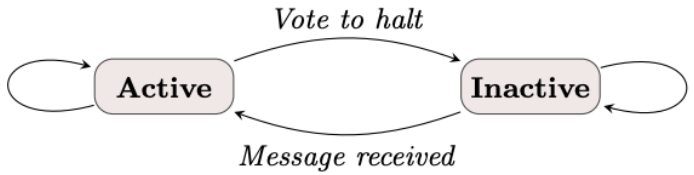
迭代计算执行流程



超步执行流程



顶点的状态转换



现有图计算框架
- 百亿以上图难以计算
- 面向开发者

HugeGraph图计算
- 大规模图稳定高效计算
- 面向普通用户

大规模图计算面临的挑战

图随机访问多，邻接边幂律分布导致任务划分难以均衡负载

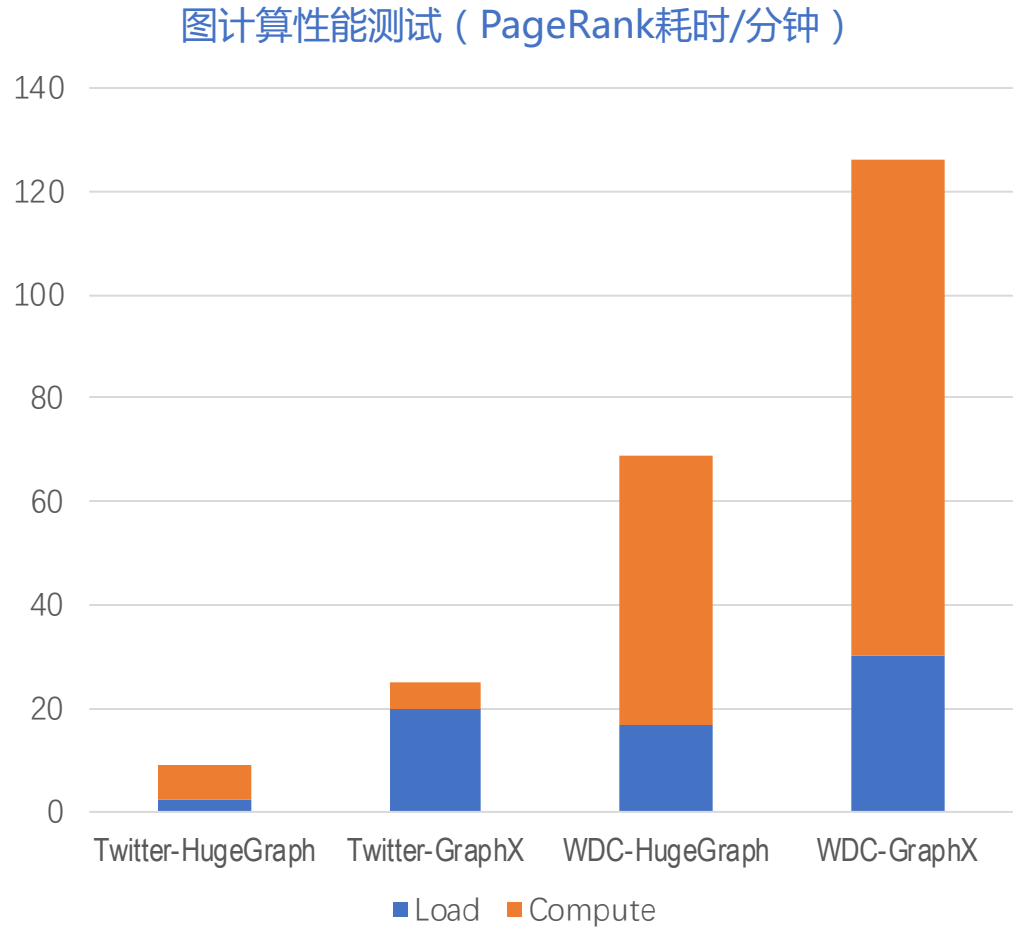
图算法复杂度高，消息与中间结果指数级增长

超过十亿规模图时，计算性能差、OOM严重

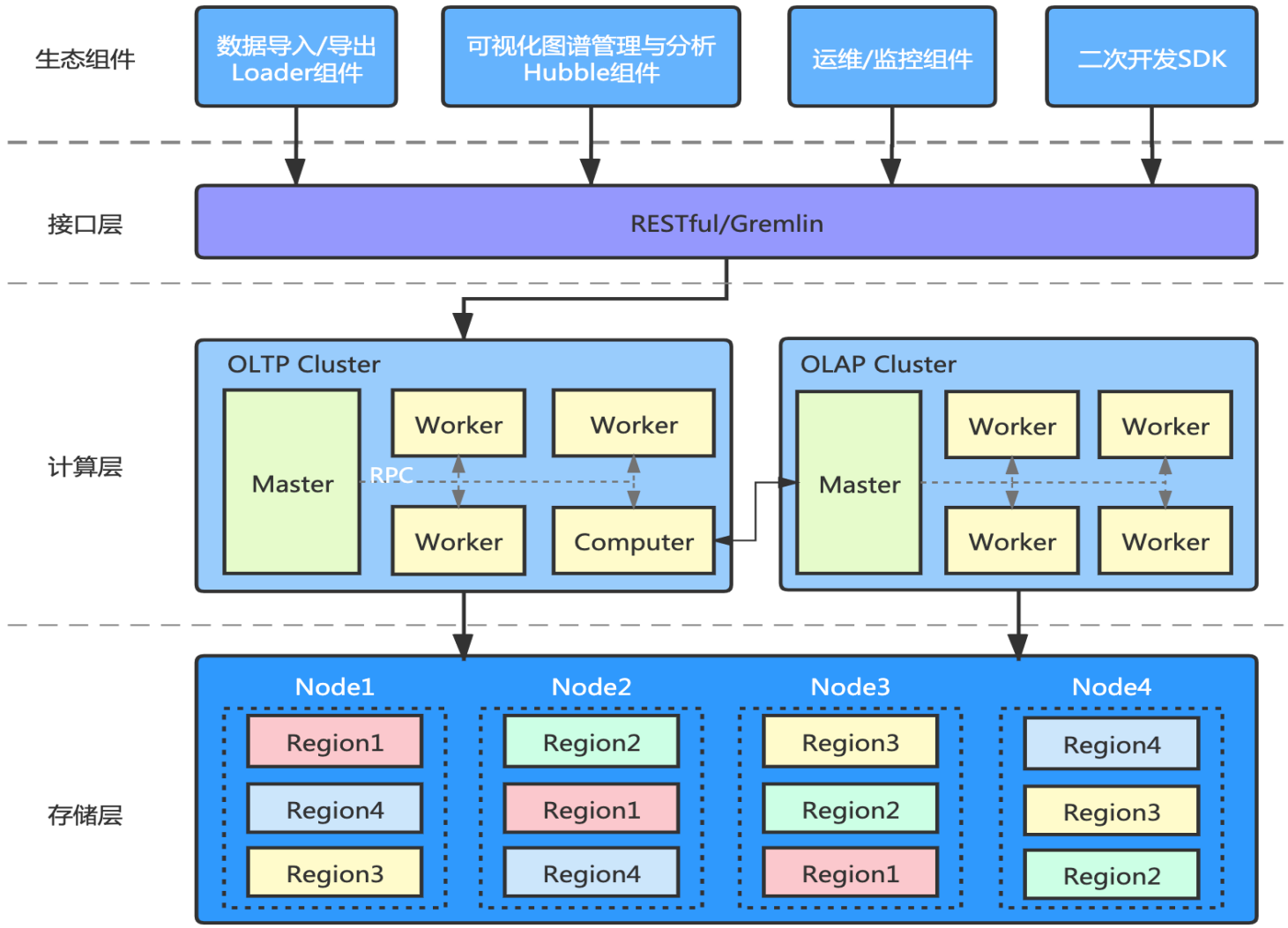
自研图计算平台实际效果

自研图计算效果：1)千亿大规模图并行计算、2)性能高效、3)无需依赖内存容量、4)高稳定性运行、5)资源消耗低。

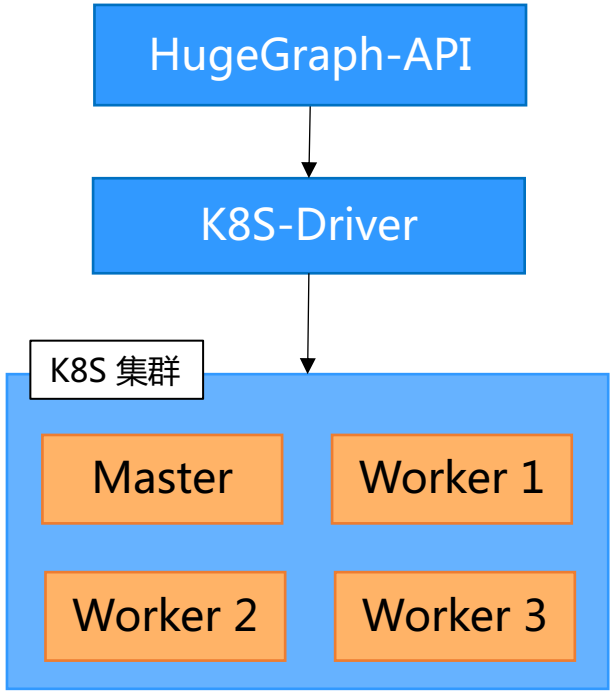
PageRank性能对比	Twitter - HugeGraph	Twitter - GraphX	WDC - HugeGraph	WDC - GraphX
数据集概览	14亿规模图	文本23GB	644亿规模图	文本1.2TB
Worker数	40	50	60	1500
CPU数	160	200	240	1500
内存	84 GB	1000 GB	1.8 TB	21 TB
磁盘	108 GB	-	2.4 TB	-
耗时	9 分钟	25 分钟	69 分钟	126 分钟
说明	每个Worker平均占内存2GB（0.1x）	使用200GB内存测试，耗时133分	消耗资源低一个数量级，稳定运行，无需反复尝试执行	使用10TB内存测试~50台机器，OOM 失败

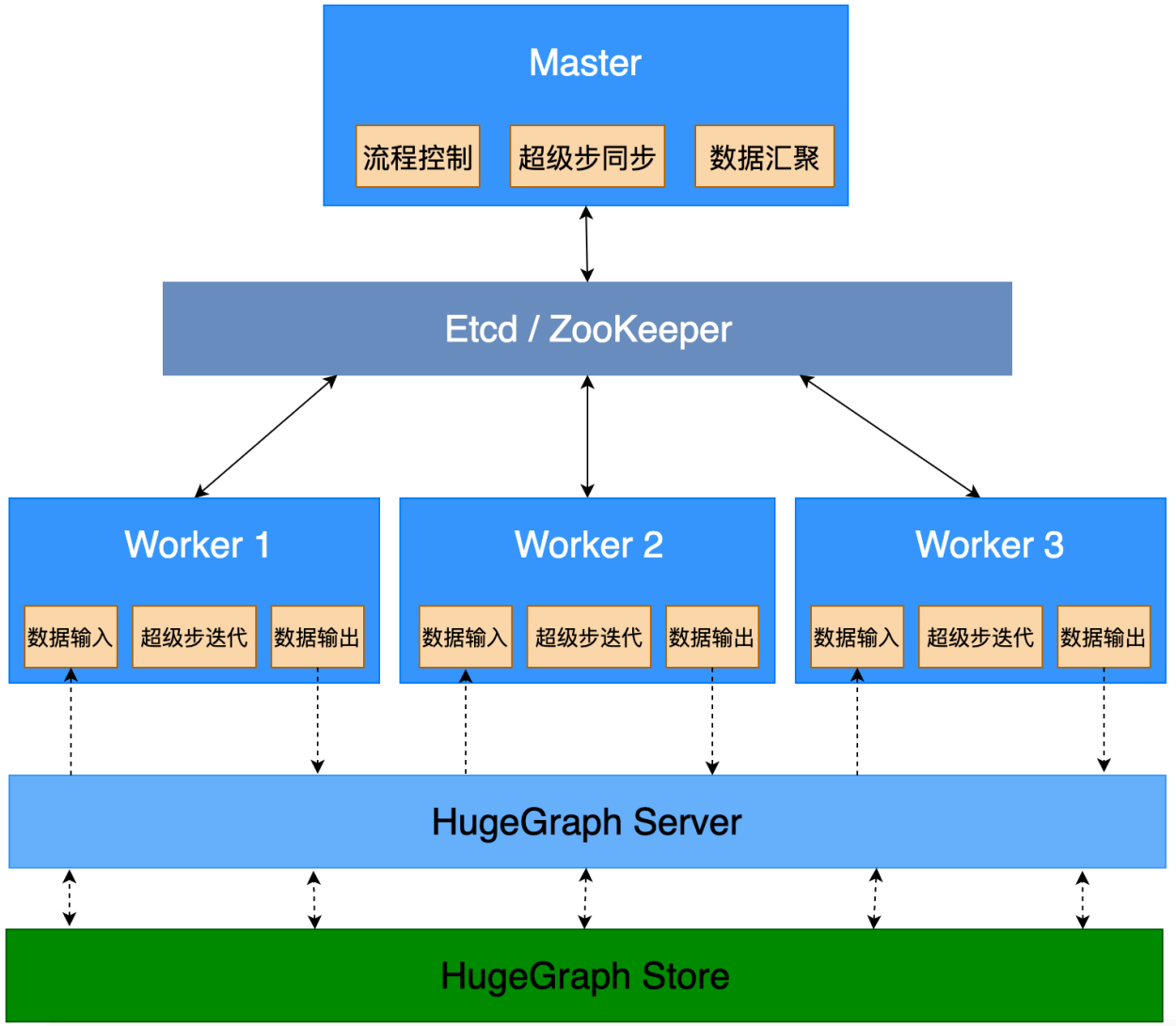


图查询&图计算 整体架构


生态组件
数据导入/导出
Loader组件
可视化图谱管理与分析
Hubble组件
运维/监控组件
二次开发SDK
接口层
RESTful/Gremlin
计算层
OLTP Cluster
Master
Worker
Worker
RPC
Worker
Computer
OLAP Cluster
Master
Worker
Worker
Worker
Worker
存储层
Node1
Region1
Region4
Region3
Node2
Region2
Region1
Region4
Node3
Region3
Region2
Region1
Node4
Region4
Region3
Region2

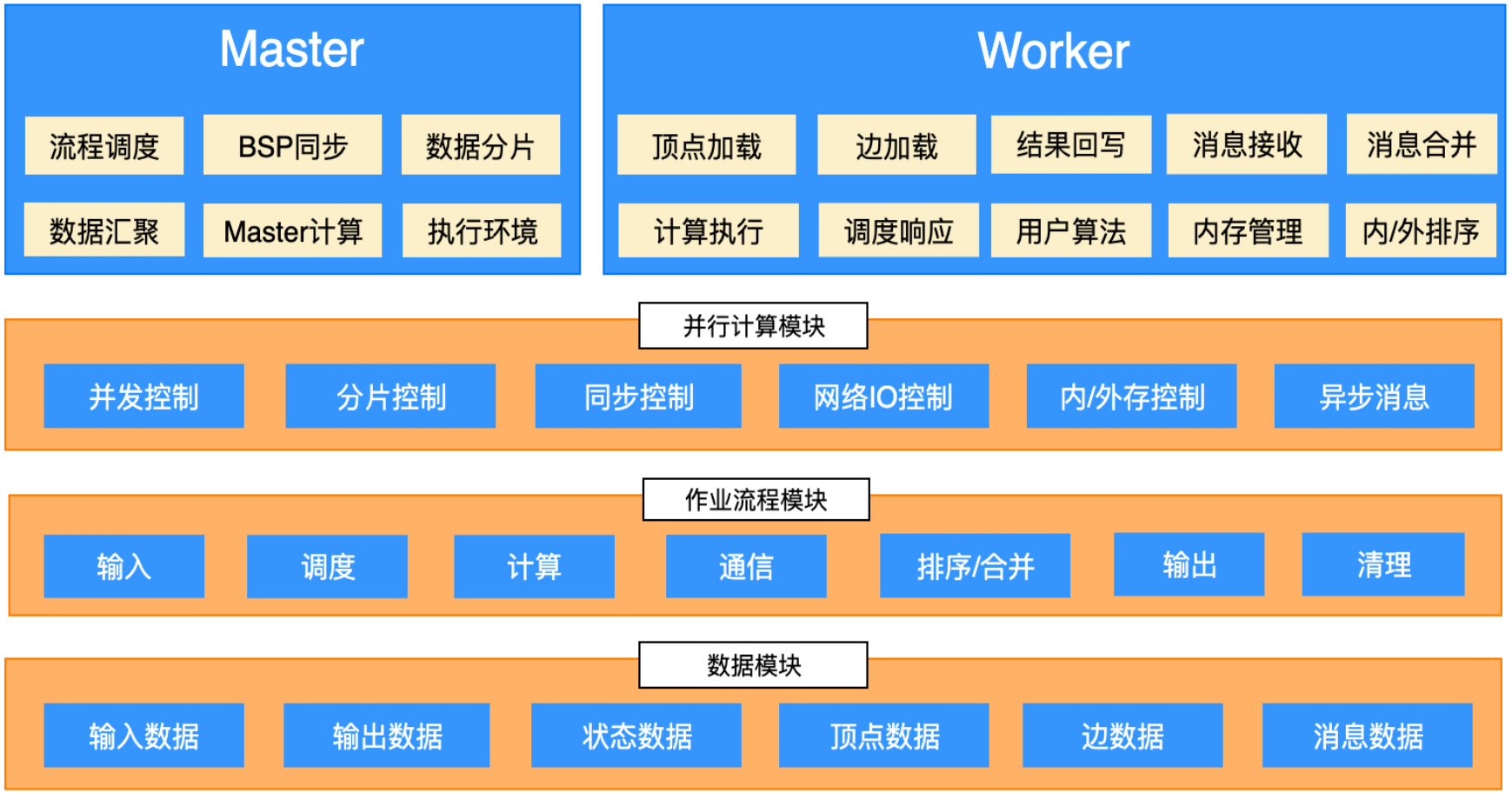
图计算架构 – 组件架构



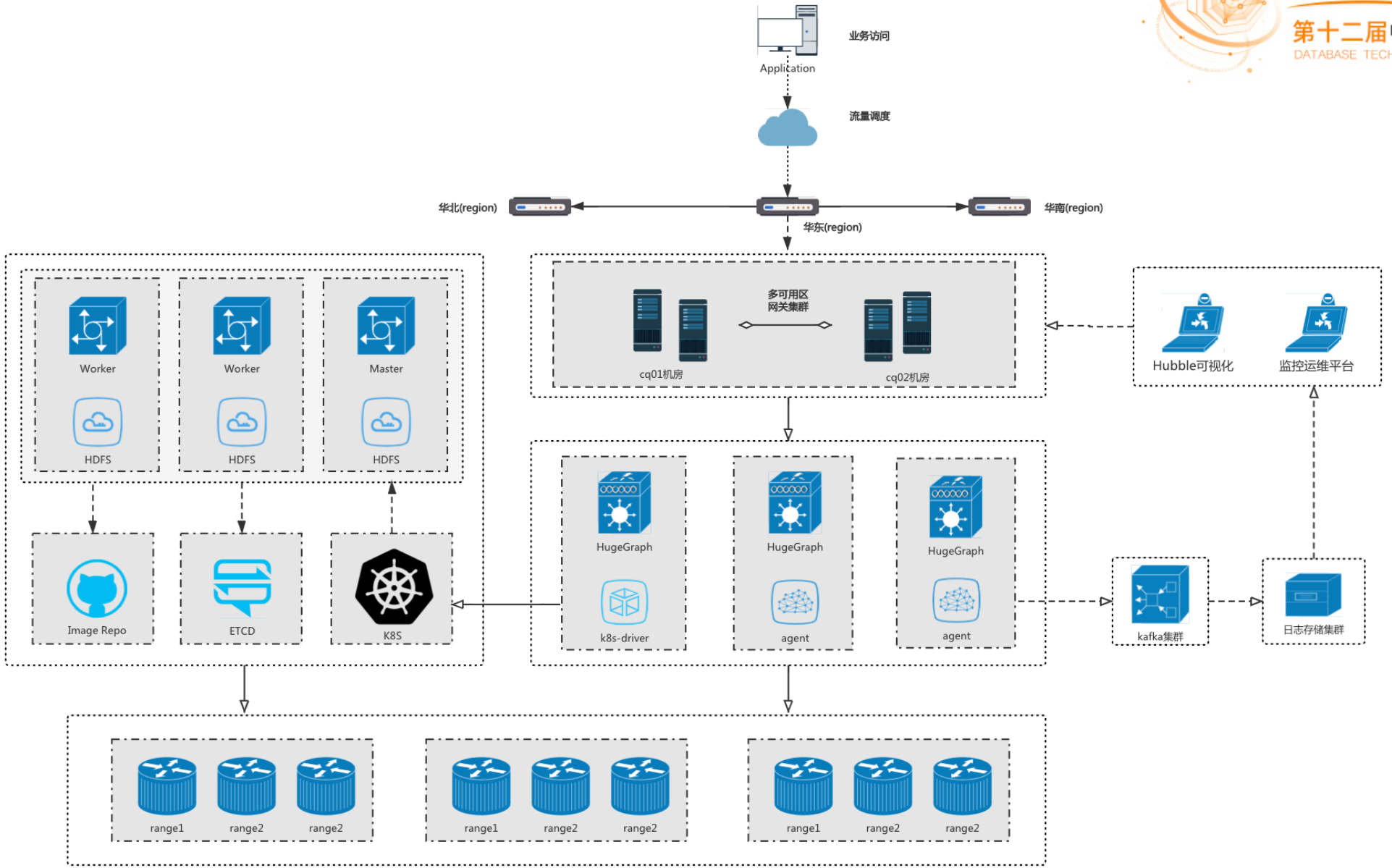


- 分布式并行计算
- 支撑千亿~万亿规模图
- 可横向扩展
- 打通图数据库

图计算架构 – 技术架构


Master
流程调度
BSP同步
数据分片
数据汇聚
Master计算
执行环境
Worker
顶点加载
边加载
结果回写
消息接收
消息合并
计算执行
调度响应
用户算法
内存管理
内/外排序
并行计算模块
并发控制
分片控制
同步控制
网络IO控制
内/外存控制
异步消息
作业流程模块
输入
调度
计算
通信
排序/合并
输出
清理
数据模块
输入数据
输出数据
状态数据
顶点数据
边数据
消息数据

图计算架构 – 部署架构


业务访问
Application
流量调度
华北(region)
华东(region)
华南(region)
多可用区
网关集群
cq01机房
cq02机房
Hubble可视化
监控运维平台
Worker
Worker
Master
HDFS
HDFS
HDFS
Image Repo
ETCD
K8S
HugeGraph
HugeGraph
HugeGraph
k8s-driver
agent
agent
kafka集群
日志存储集群
range1
range2
range2
range1
range2
range2
range1
range2
range2

图计算架构优势

- 规模大：千亿级规模图，自研图计算框架，横向线性扩展
- 性能好：并行计算，自研异步框架，多轮迭代BSP块同步
- 高稳定性：架构上批式计算，防止消息/超级点导致的OOM问题
- 简单易用：打通图数据库与图计算，以顶点为中心开发算法自然

图计算架构关键技术分解

并行

- **BSP并行迭代**
  轮内并行，跨轮同步
- **图分区并行**
  各分区并行计算
- **并行度高**
  顺序读取代随机读
- **计算-通信并行**
  收发消息不阻塞计算
- **并行排序**
  大规模消息并行排序
- **并行输入输出**
  数据并行输入输出

大规模

- **动态分配计算资源**
  算法按需申请计算资源
- **批式计算**
  矩阵向量模型计算
- **内存-磁盘自适应**
  计算不受限于内存容量
- **列存储结构**
  高效按需读取，顺序Join
- **高效消息汇聚**
  消息汇聚本地-远程相结合
- **高效消息处理**
  消息本地合并降低发送规模

异步

- **异步消息发送**
  多线程发送消息
- **异步数据排序**
  多线程排序数据及消息
- **异步消息接收处理**
  多线程接收消息及刷盘存储
- **异步网络协议**
  通过Pipeline提高网络吞吐

解耦

- **存算分离**
  存储与计算集群分离
- **OLTP-OLAP分离**
  TP数据与AP数据分离存储
- **算法-存储分离**
  算法无需关注数据格式
- **算法-框架分离**
  算法无需关注底层框架流程

图计算架构面临的关键技术挑战

并行计算
负载不均衡

消息传输量
巨大

内存OOM
严重

分布式节点
部分失败

图计算架构针对挑战的应对方法

并行计算
负载不均衡

消息传输量
巨大

内存OOM
严重

分布式节点
部分失败

- 并行计算：存算分离架构，使用 K8S 极易扩展，多 Worker 并行计算
- 若干 Partition，每个 Worker 负责部分 Partition 数据，按照点粒度分区
- 一个 Worker 按边总数量分配 Partition 个数，确保性能与负载均衡
- Worker 内部多 Partition 多线程并行计算，计算与消息异步并行

图计算架构针对挑战的应对方法

并行计算
负载不均衡

消息传输量
巨大

内存OOM
严重

分布式节点
部分失败

- 发往远程的消息本地 combine 合并，超级点消息延迟到目标 Worker 分发
- 消息传输与计算并行，消息传输通过 Pipeline 方式提升吞吐
- 自定义协议与格式，分离数据面与控制面，利用压缩编码减少数据传输量
- 大规模消息利用基于磁盘/内存文件的堆排序/败者树进行动态多路归并

图计算架构针对挑战的应对方法

并行计算
负载不均衡

消息传输量
巨大

内存OOM
严重

分布式节点
部分失败

- 不依赖内存容量来存储图计算的输入和中间结果，根本上解决 OOM
- 内存-磁盘自适应，不依赖内存 Map，通过顺序 Join 内存/磁盘 列存文件
- 重用内存对象，自分配内存方式管理内存，使用JVM堆外内存降低GC压力
- 批式计算，通过流水方式进行大数据计算，矩阵向量方式平衡性能

图计算架构针对挑战的应对方法

并行计算
负载不均衡

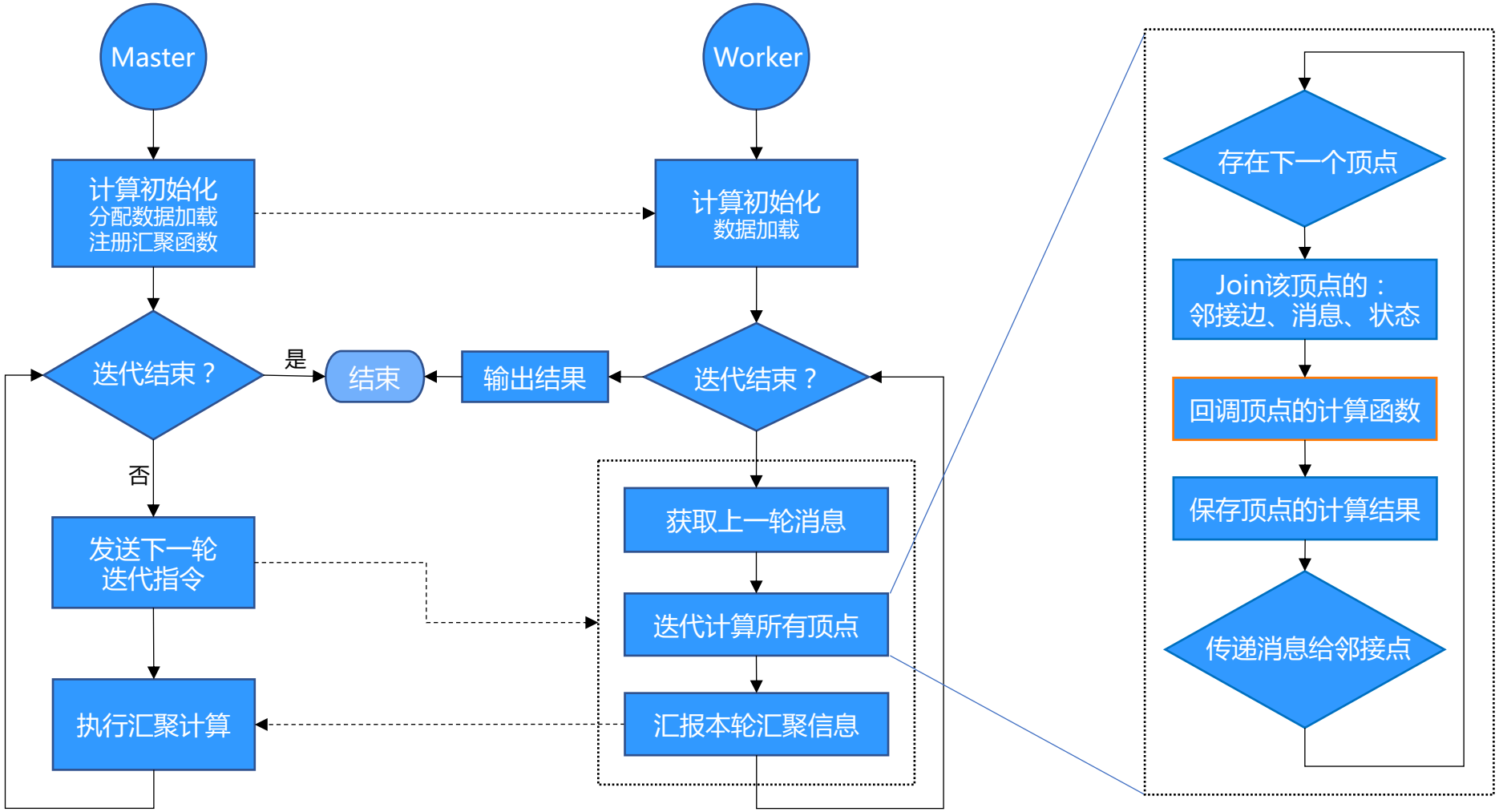
消息传输量
巨大

内存OOM
严重

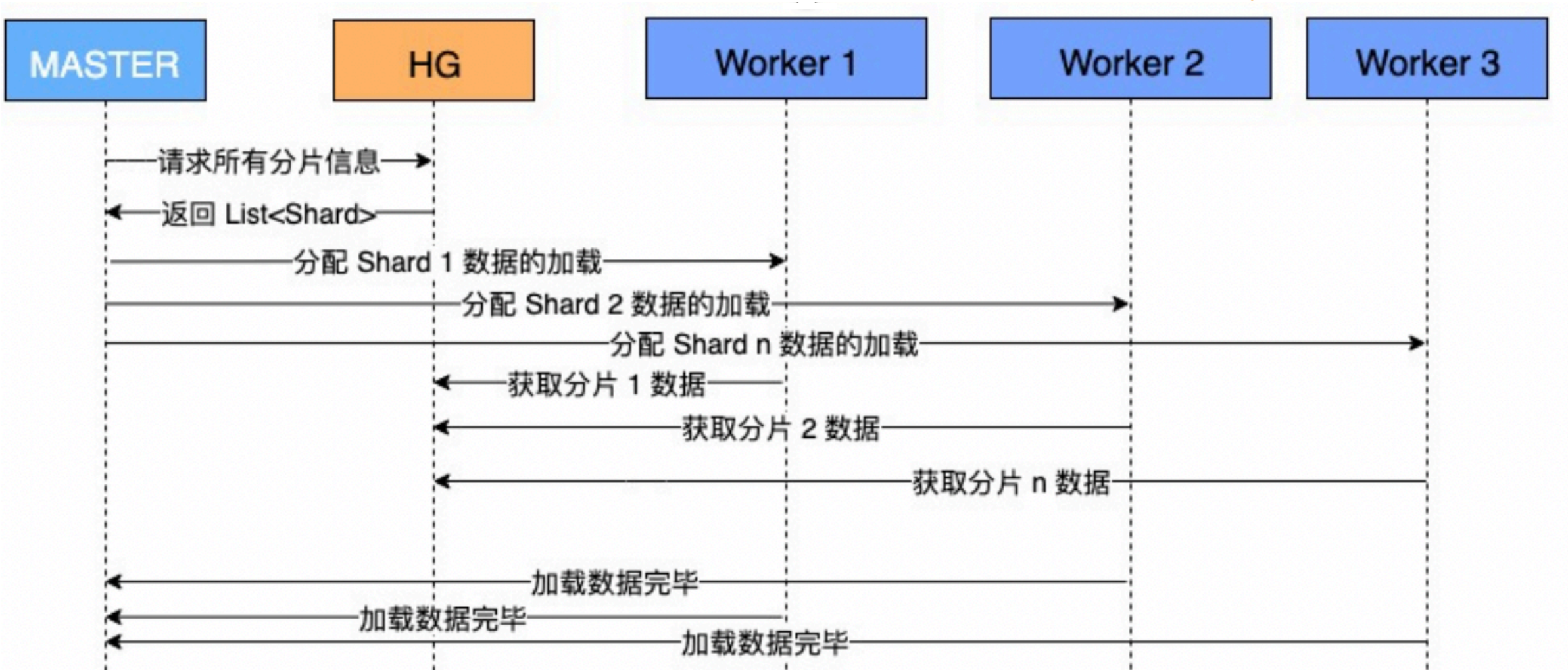
分布式节点
部分失败

- 部分节点失败，将导致整个集群计算中断，造成时间、计算资源的极大浪费
- 建立全链路负载反馈机制，尽可能避免部分节点因瞬时负载过高导致的失败
- 建立故障恢复机制，每一轮计算的中间结果快照到分布式文件系统上
- 部分节点失效时，无需从头计算，通过启动新的节点替代故障节点即可

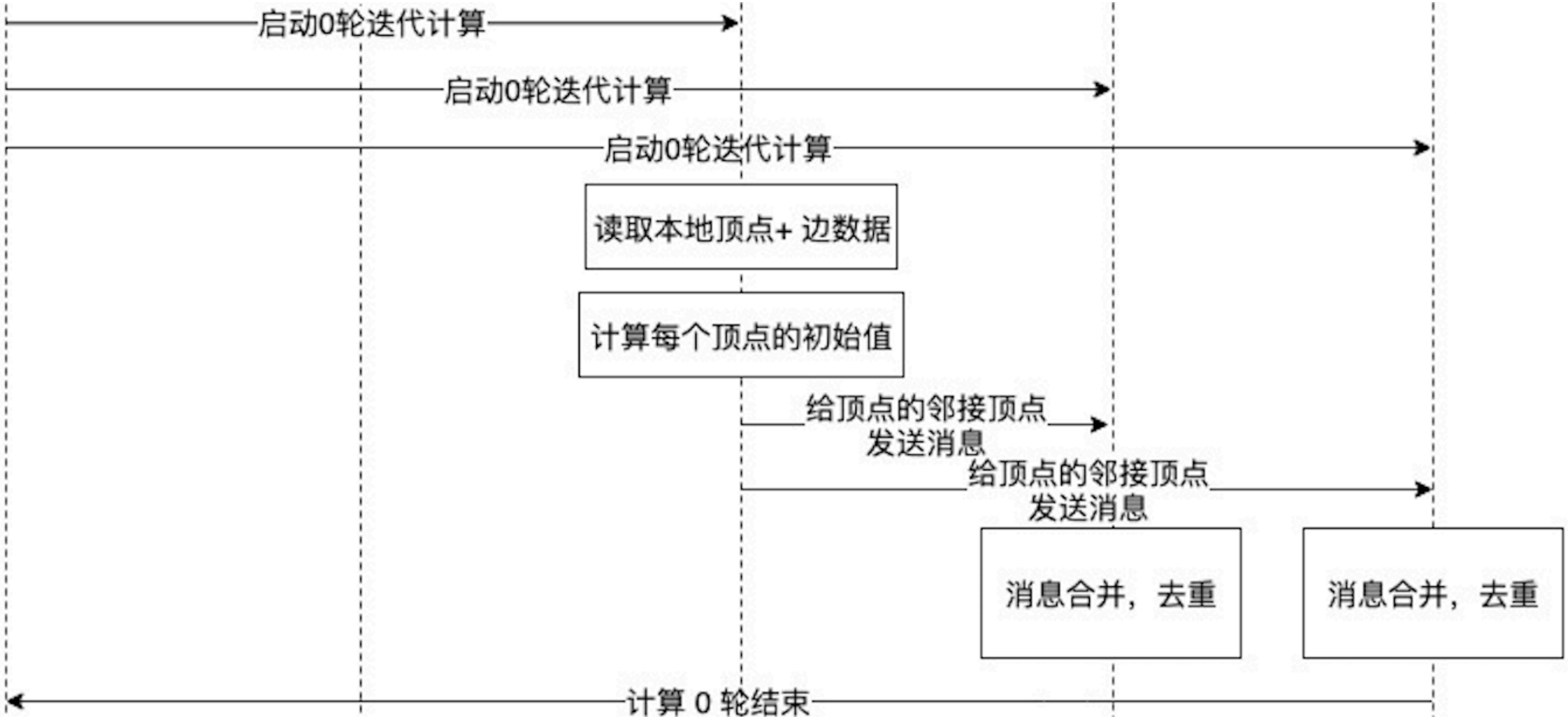
图计算流程


Master
计算初始化
分配数据加载
注册汇聚函数
迭代结束？
是
结束
否
发送下一轮
迭代指令
执行汇聚计算
Worker
计算初始化
数据加载
输出结果
迭代结束？
获取上一轮消息
迭代计算所有顶点
汇报本轮汇聚信息
存在下一个顶点
Join该顶点的：
邻接边、消息、状态
回调顶点的计算函数
保存顶点的计算结果
传递消息给邻接点

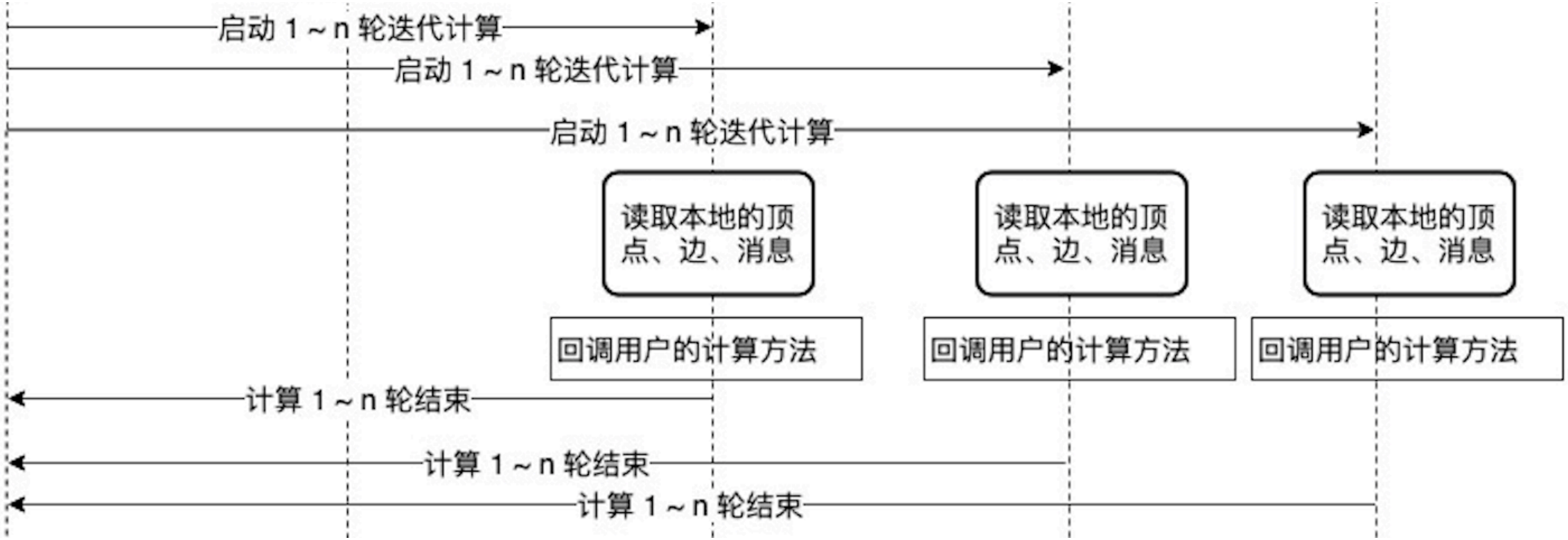
图计算流程 - 并行加载


MASTER
HG
Worker 1
Worker 2
Worker 3
请求所有分片信息
返回 List<Shard>
分配 Shard 1 数据的加载
分配 Shard 2 数据的加载
分配 Shard n 数据的加载
获取分片 1 数据
获取分片 2 数据
获取分片 n 数据
加载数据完毕
加载数据完毕
加载数据完毕

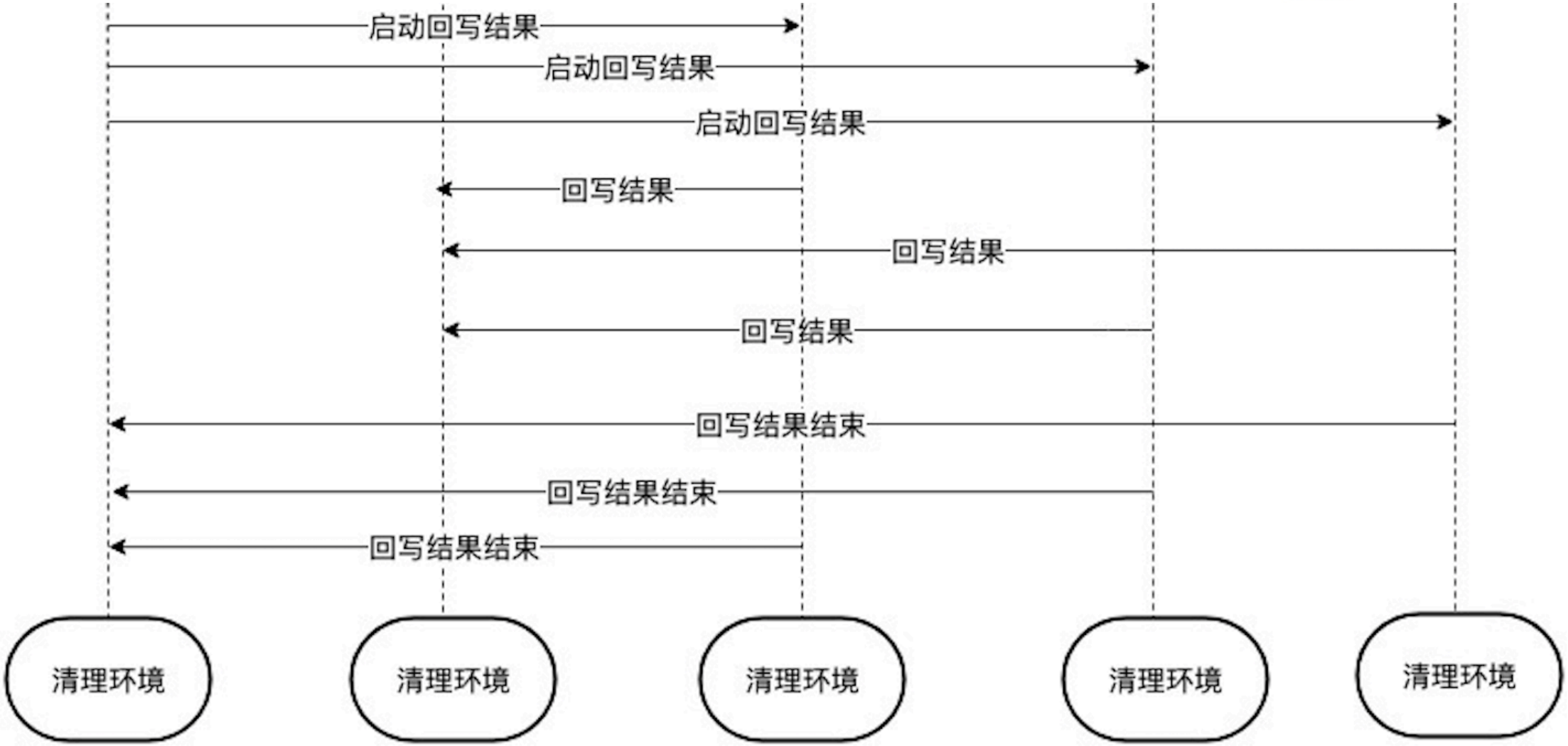
图计算流程 - 并行计算


启动0轮迭代计算
启动0轮迭代计算
启动0轮迭代计算
读取本地顶点+ 边数据
计算每个顶点的初始值
给顶点的邻接顶点
发送消息
给顶点的邻接顶点
发送消息
消息合并，去重
消息合并，去重
计算 0 轮结束

图计算流程 - 并行计算


启动 1～n 轮迭代计算
启动 1～n 轮迭代计算
启动 1～n 轮迭代计算
读取本地的顶点、边、消息
读取本地的顶点、边、消息
读取本地的顶点、边、消息
回调用户的计算方法
回调用户的计算方法
回调用户的计算方法
计算 1～n 轮结束
计算 1～n 轮结束
计算 1～n 轮结束

图计算流程 - 并行输出


启动回写结果
启动回写结果
启动回写结果
回写结果
回写结果
回写结果
回写结果结束
回写结果结束
回写结果结束
清理环境
清理环境
清理环境
清理环境
清理环境

图计算流程 - 算法流程

算法开发流程

一、创建算法
1. 引入 hugegraph-computer-api 依赖包；
2. 增加一个 Java/Python 文件；
3. 实现 Computation 接口，重点实现 compute() 方法；

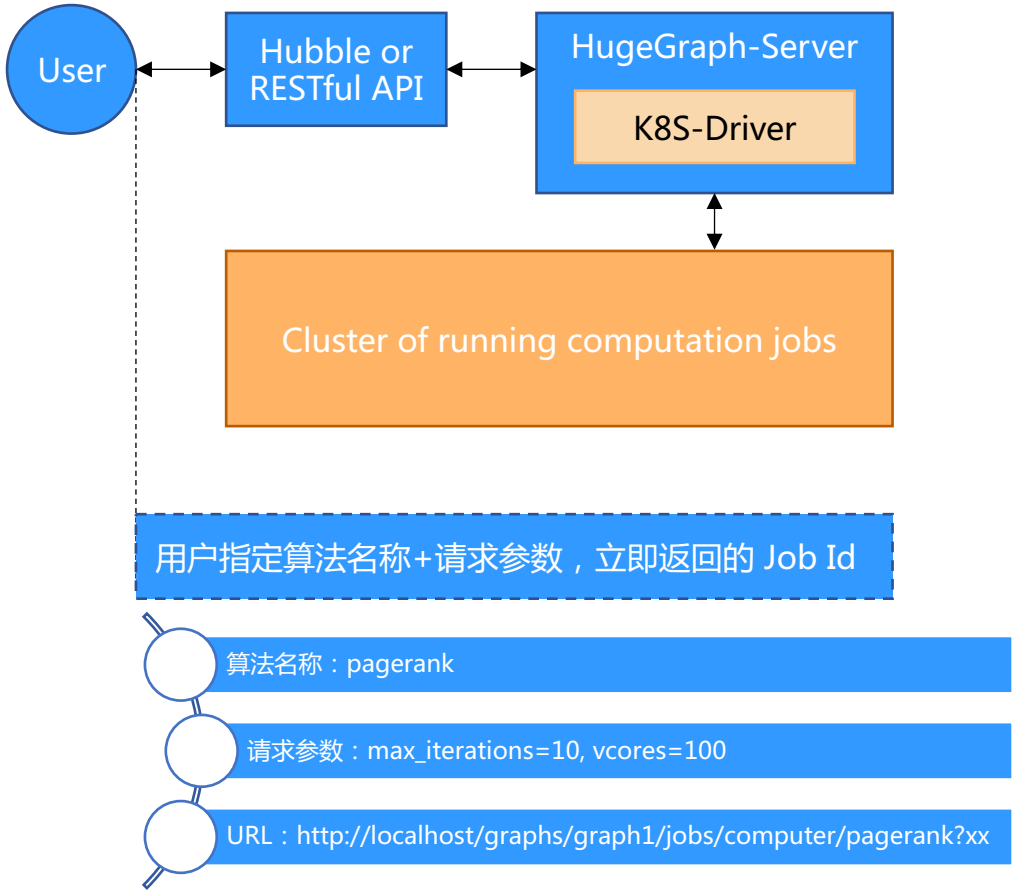
↓

二、编译算法
1. 编译算法文件得到 Jar 包；

↓

三、注册算法
1. 上传 Jar 包到图计算平台（后台会将 Jar 包转为 Docker 镜像）；
2. 声明算法名字、类路径；
3. 指定算法执行的默认参数，比如 CPU数量、内存大小、收敛阈值；

算法使用/执行流程



图计算流程 - 算法示例

HugeGraph图算法开发的特点：

- 算法开发简单，以顶点为中心编程直观；支持Java、Python等开发语言，轻松动态扩展算法。
- 无需关注框架流程与算法的执行细节，仅关注算法逻辑本身，专注每个顶点收到的消息及其处理。
- 计算的输入及输出格式均与算法本身解耦，采取HugeGraph标准的OLTP接口访问计算结果。

```
public void compute(ComputationContext context, Vertex vertex,
                    Iterator<DoubleValue> messages) {
    double rankRecved = Combiner.combineAll(context.combiner(), messages);
    double rank = rankRecved * (1.0 - this.alpha) +
                  this.alpha / context.totalVertexCount();
    vertex.value(rank);

    int edgeCount = vertex.numEdges();
    double contributeValue = rank / edgeCount;
    context.sendMessageToAllEdges(vertex, contributeValue);
}
```


THANKS
时序
开源
自研
分布式
文档
Spark
SQL
HANA
OLAP
Oracle
Hadoop
MPP
DB2