# 分布式流数据实时与持续计算

强琦

hic2011 2011.12.3

# 提纲

- 背景
- 目标
- 传统方案与业界进展
- 设计理念(重点)
- 技术架构
- 要点
- 例子
- 系统边界
- 计划

# 背景

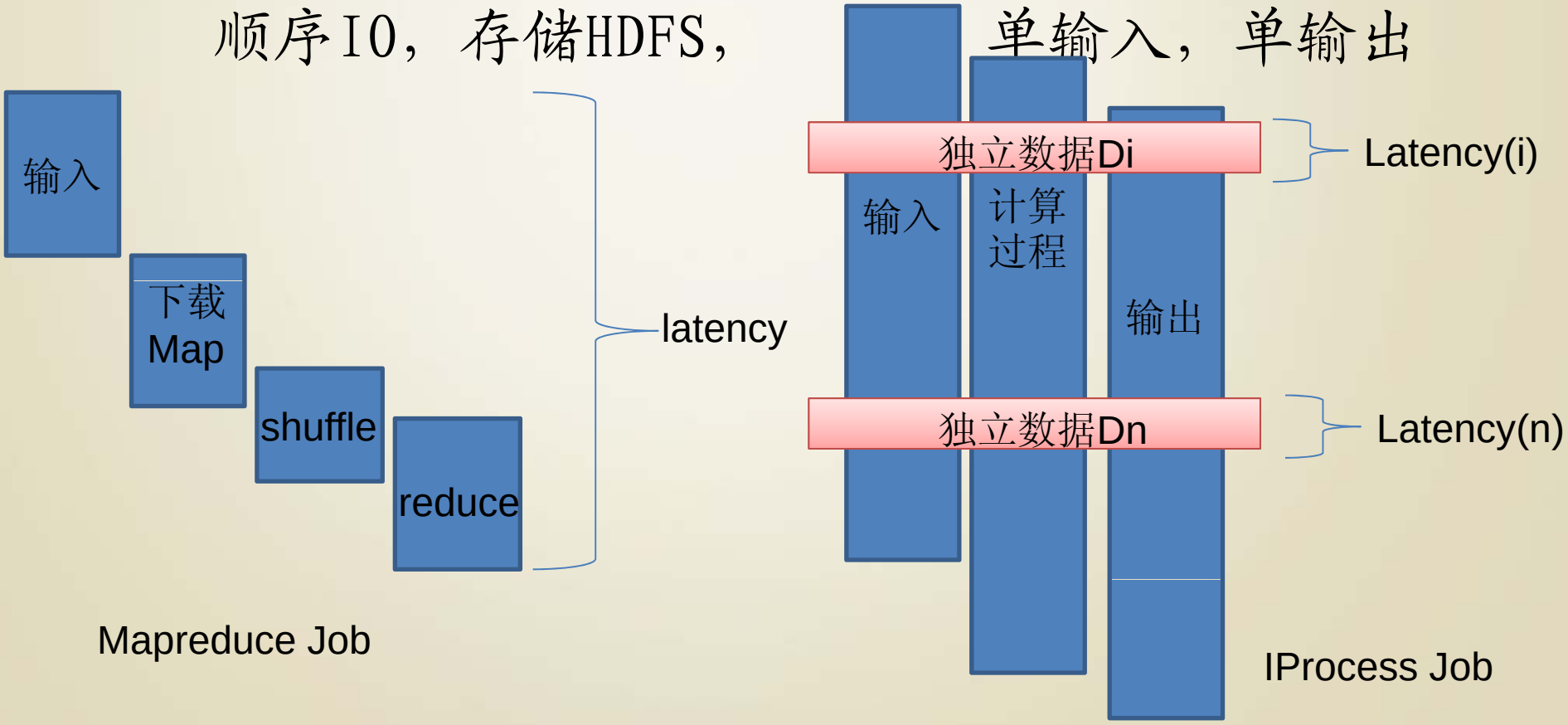
- 应用背景
  - 数据量急剧增加
  - Web 1.0→ web 2.0, public→ego net
  - 电子商务、移动互联网、移动支付
  - 欺诈、风控对海量交易实时性
  - 用户体验的个性化和实时性
  - 由点到面
    - 实时搜索、个人实时信息服务、SNS等

# 背景

- 技术背景
  - MapReduce、Dryad等全量/增量计算平台
  - S4、Storm等流计算框架
  - CEP以及EDA模型
  - Pregel等图计算模型

# 传统方案与业界进展

- 传统方案
  - MAPREDUCE：HDFS加载，存储LOCALITY(容错性)，顺序IO，存储HDFS，单输入，单输出



# Hadoop之于实时

- 问题(hadoop本质是为全量而生)
  - 任务内串行
  - 重吞吐量，响应时间完全没有保证
  - 中间结果不可见，不可共享
  - 单输入单输出，链式浪费严重
  - 链式MR不能并行
  - 粗粒度容错，可能会造成陷阱
  - 图计算不友好
  - 迭代计算不友好

# 图计算

- MapReduce为什么不适合图计算?
  - 迭代
  - 边的量级远大于节点
- 图计算特点
  - 适应于事件机制，规模大(边)，但单条数据不大
  - 很难分布式(locality、partition，一直都是难点)
  - 容错性
  - Google Pregel
    - 本质上还是全量
    - 中间结果不可见
    - 超步过多(IProcess)

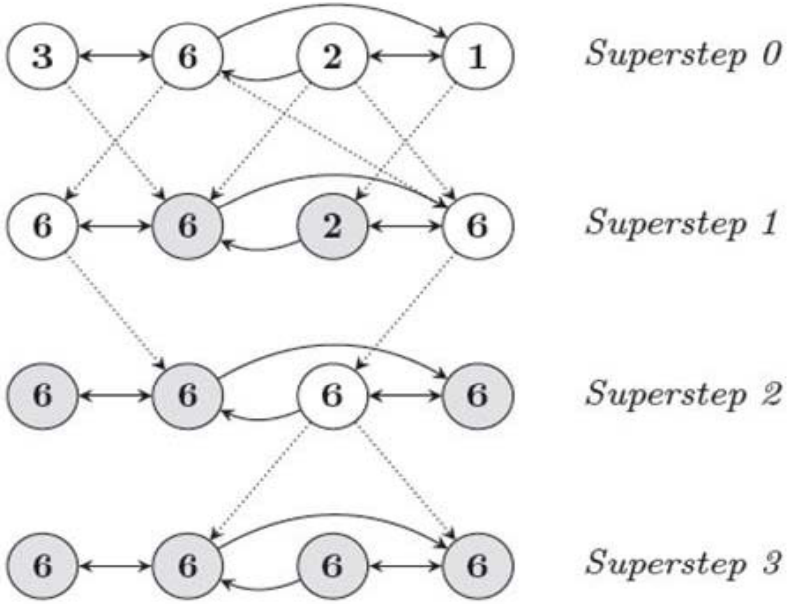
# Pregel vs. IProcess图计算

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```




- IProcess
  - ✓ 乱序执行，避免了不必要的超步
  - ✓ 实时图计算，图计算注定慢，但是效果的可以渐显。

# 迭代计算

- 特点
  - 结构固定
- 本质
  - Update
- 方案
  - 传统MR模型，hadoop效率太低
  - Haloop
  - Iprocess0.4

# 实时计算业界进展

- S4
  - 2010年底，Yahoo，0.3，window todo

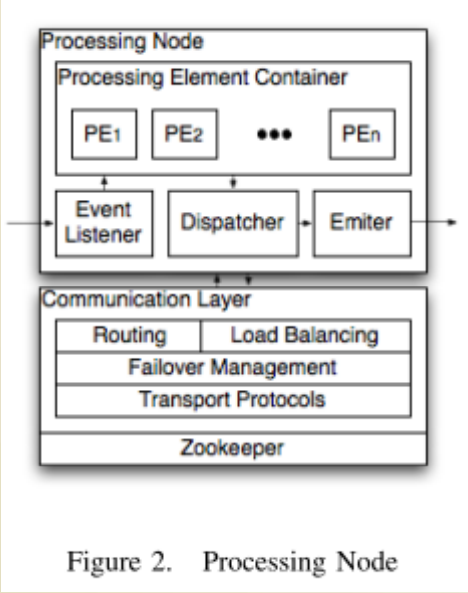
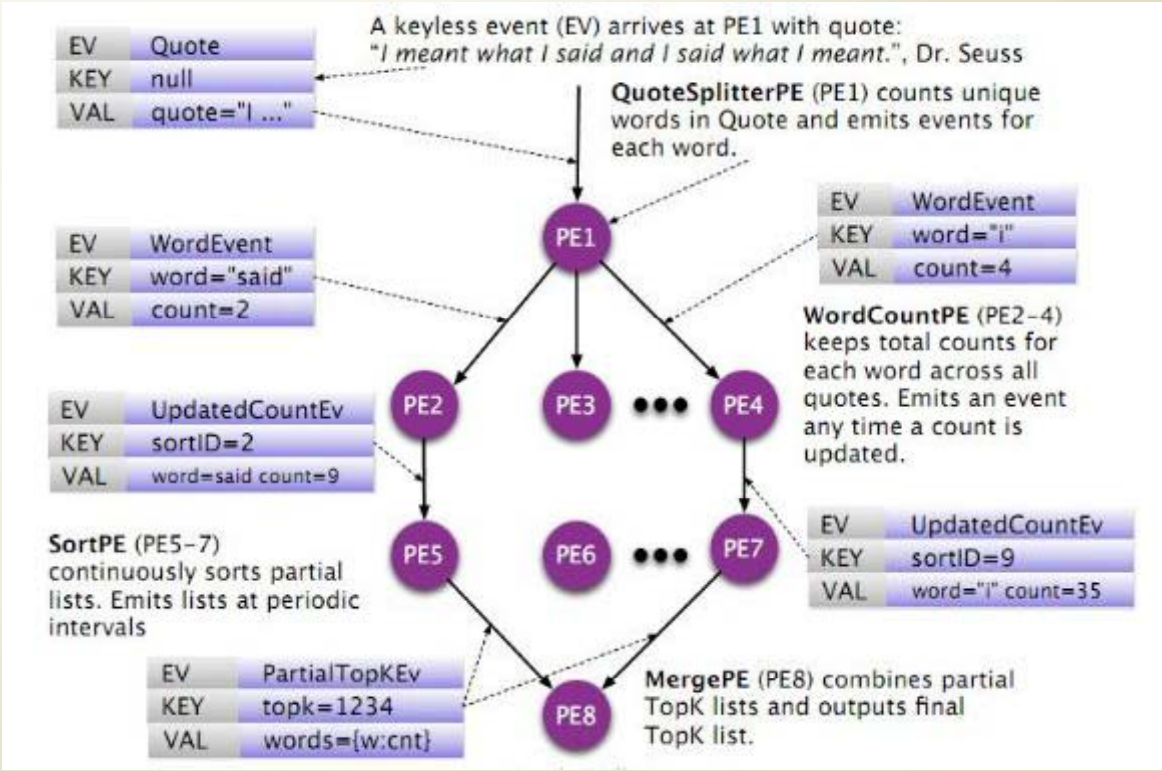
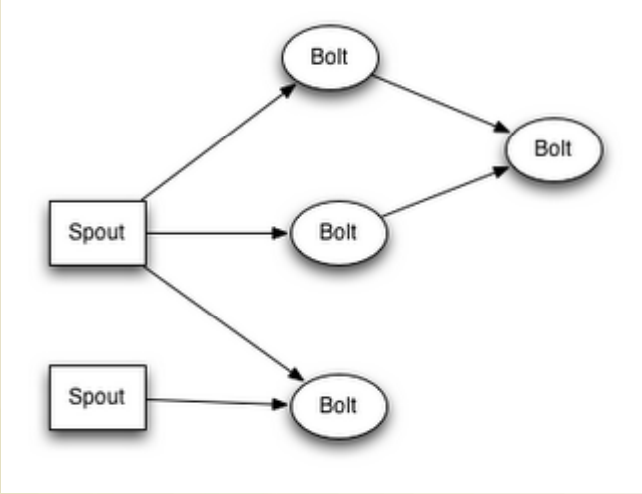



Figure 2. Processing Node



# 业界进展

- Storm:2011.9，twitter，0.5.2





Here's how you define this topology in Java:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(1, new KestrelSpout("kestrel.backtype.com",
                                     22133,
                                     "sentence_queue",
                                     new StringScheme()));
builder.setBolt(2, new SplitSentence(), 10)
        .shuffleGrouping(1);
builder.setBolt(3, new WordCount(), 20)
        .fieldsGrouping(2, new Fields("word"));
```

# 业界进展-Storm

```
public class SplitSentence implements IBasicBolt {
    public void prepare(Map conf, TopologyContext context) {
    }

    public void execute(Tuple tuple, BasicOutputCollector collec
        String sentence = tuple.getString(0);
        for(String word: sentence.split(" ")) {
            collector.emit(new Values(word));
        }
    }

    public void cleanup() {
    }

    public void declareOutputFields(OutputFieldsDeclarer declare
        declarer.declare(new Fields("word"));
    }
}
```

```
public class WordCount implements IBasicBolt {
    private Map<String, Integer> _counts = new HashMap<String, I

    public void prepare(Map conf, TopologyContext context) {
    }

    public void execute(Tuple tuple, BasicOutputCollector collec
        String word = tuple.getString(0);
        int count;
        if(_counts.containsKey(word)) {
            count = _counts.get(word);
        } else {
            count = 0;
        }
        count++;
        _counts.put(word, count);
        collector.emit(new Values(word, count));
    }
```

# 系统边界

- S4、Storm
  - 只能处理“独立”的流数据
  - 无法处理“复杂”事件(condition)，需要用户handle复杂的条件
  - 不能很好的适用于大部分需要相关数据集执行计算和流数据保序的实时场景
  - 容错性较差
  - 集群无法动态扩展

# 业界进展

- 其它
  - StreamBase
  - Borealis
  - StreamInsight
  - Percolator
  - Hbase coprocessor
  - Pregel
  - dremel
  - …

# 设计理念

- 负责任(Condition)
  - MapReduce本质上保证了Reduce触发的条件，即所有map都结束（但这点很容易被忽视）。
  - 实时计算Condition很容易被忽略。很多只是考虑了streaming，而没有考虑Condition。
- 实时(Streaming)
- 成本(Throughput)
- 有所为有所不为
  - 通用计算框架，用户组件只需关心业务逻辑。
  - 涉及到业务逻辑统统不做。

# 设计理念

- 举例
  - 实时JOIN(后面有具体代码)

    在storm（不考虑Condition）框架下，实现join，需要用户代码自己hold条件，判断条件，进而触发join后的逻辑处理。但在我们的设计理念下，这些condition完全可以抽象为复杂完备事件模型，所以作为通用系统应该提供condition的通用功能，用户只需进行配置而不是编码就可以完成condition，那么实时join在iprocess体系下，用户无需编码处理condition，而只需处理join后的逻辑。

# IProcess

- 通用的分布式流数据实时与持续计算平台
  - 有向图模型
    - 节点为用户编写的组件、边为事件
  - 触发器模式
  - 完备事件驱动的架构，定制复杂完备事件条件
  - 支持相关集计算和Reduce时数据集生成(k-mean)
  - 树存储模型，支持不同级别定制不同一致性模型和事务模型
  - 可扩展的编程模型
    - 提出并支持树型实时MR和增量/定时MR

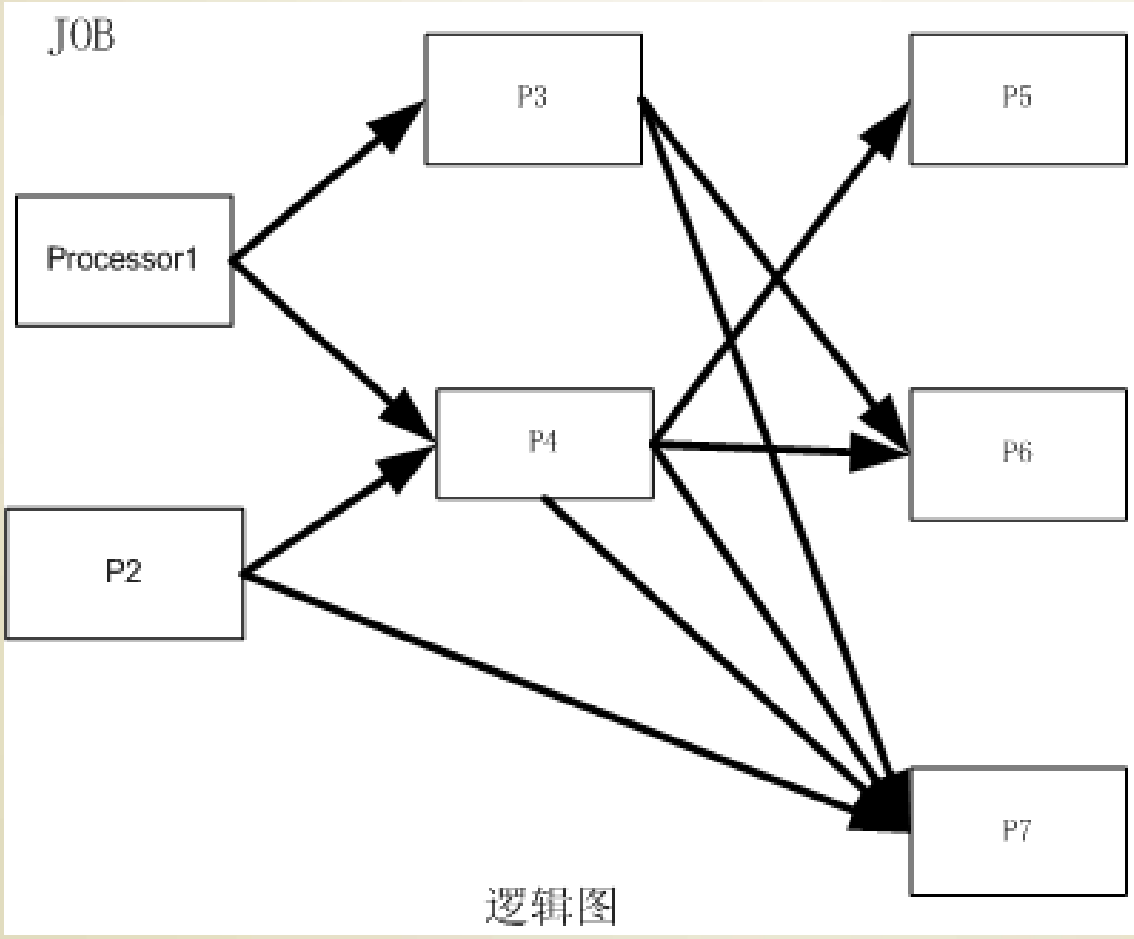
# IProcess

- 通用的分布式流数据实时与持续计算平台
  - 持续与AdHoc计算(endpoint)
  - 微内核+组件系统(系统级组件+用户组件)
  - 多任务服务化，任务沙箱，优先级，任务调度
  - 两级容错：应用级和系统级，运算时动态扩容
  - 系统级组件系统：实时join、二级索引、倒排表、物化视图、counter…
  - 分布式系统的容错，自动扩展，通讯，调度
  - 保序…

# IProcess

- 基础的运行系统
  - 引入CEP规则引擎模块（RPM），类似hive与MR
  - 引入数据集控制（用于机器学习），BI
  - 引入类SQL语言，DSL引擎
  - 引入图计算模型

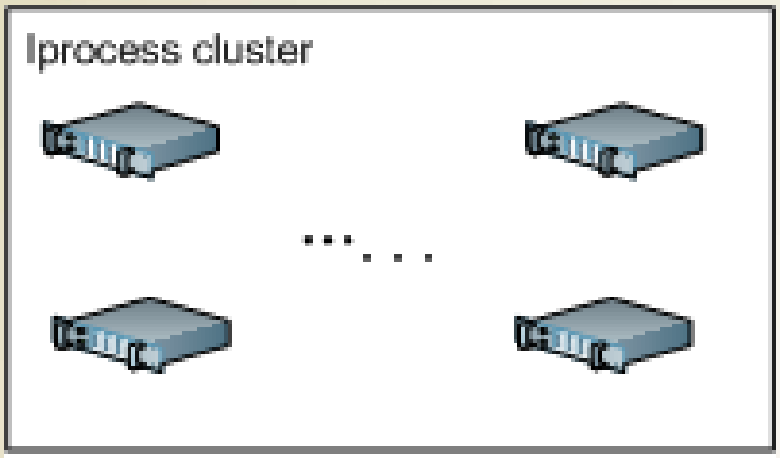
# 逻辑模型



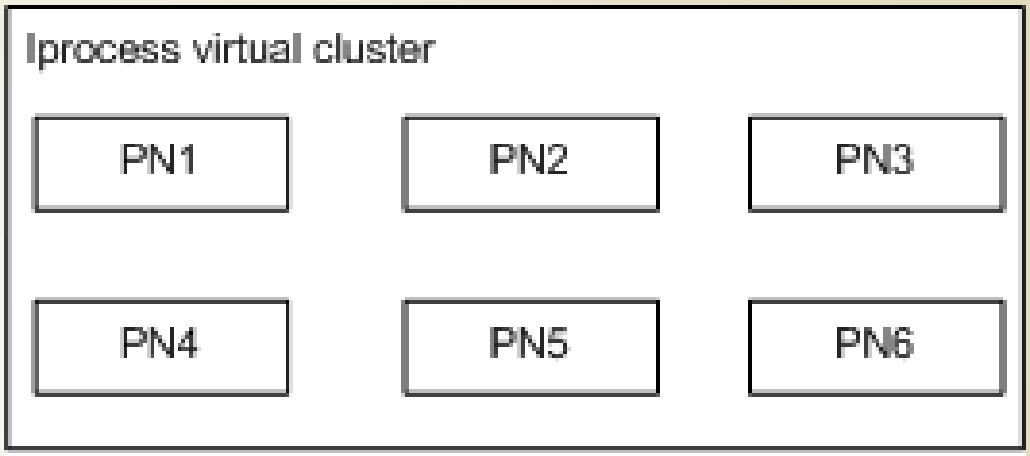

逻辑图

AND
OR
SPLIT

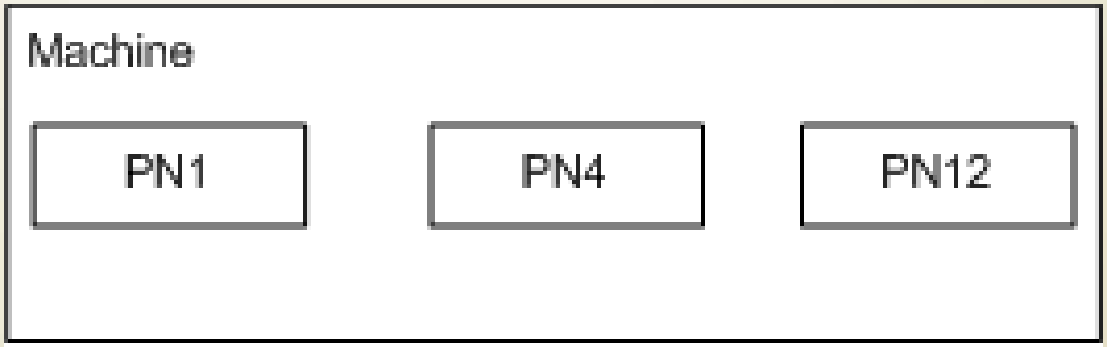
4类组件：
1. 容器类（MR，SQL引擎）；
2. 系统级组件（消息机制）；
3. 可复用组件（物化视图）；
4. 任务组件。



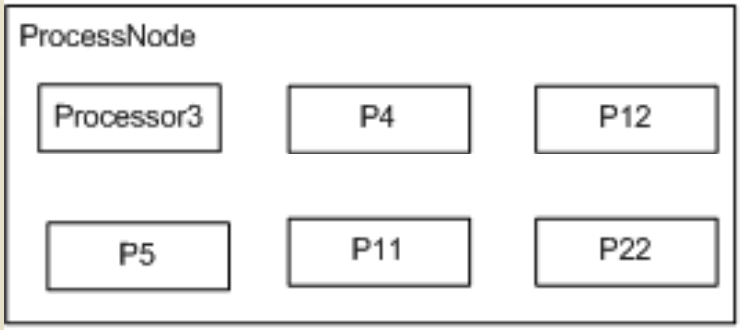

物理cluster




Virtual cluster

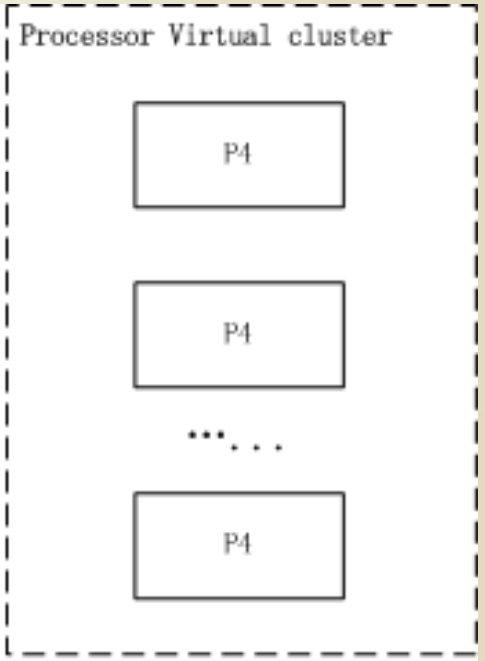



物理机与ProcessorNode




PN与processor

Processor之间可以选择是跨物理机，还是同进程，亦或是同线程（子图）。事件也可以选择保序。

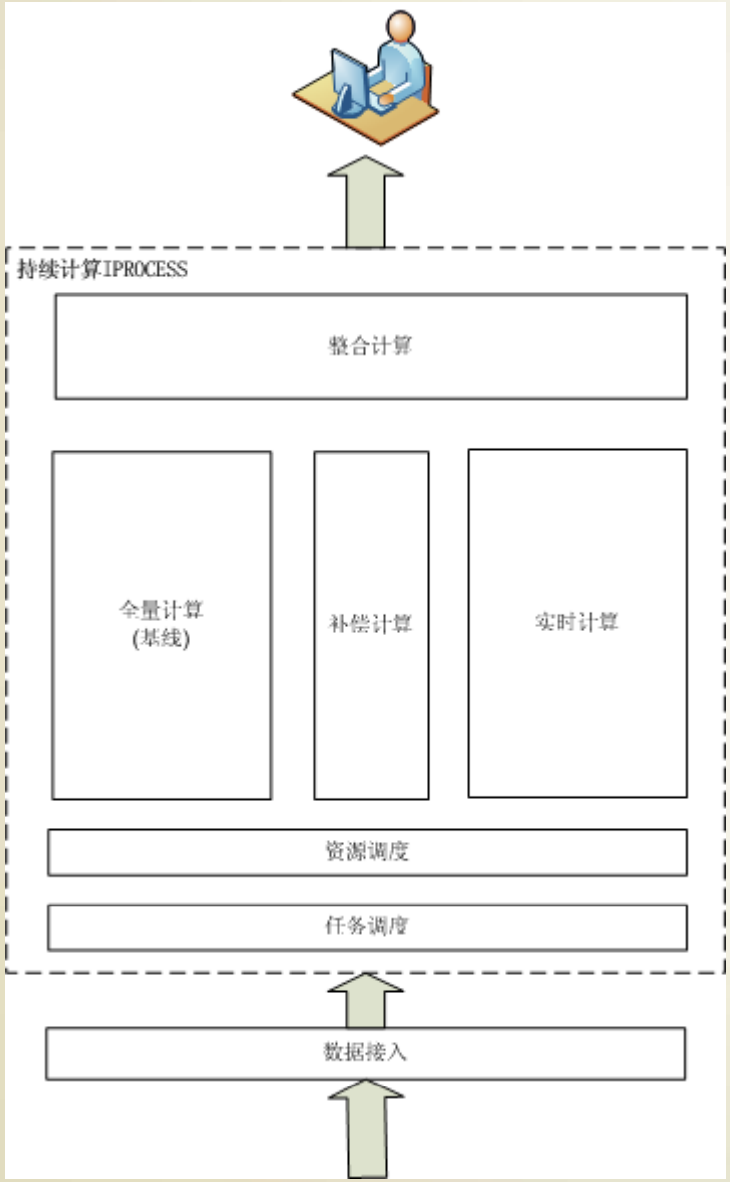



processor逻辑cluster

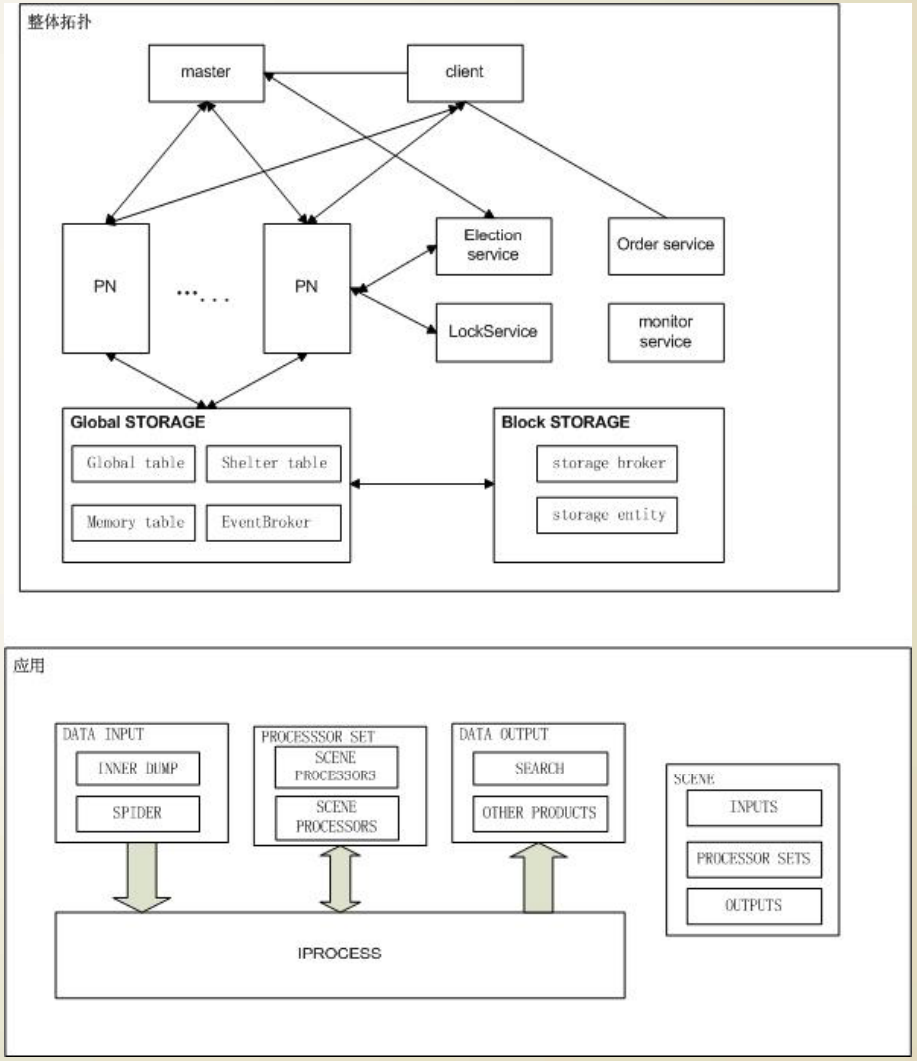
# 持续计算

- Ad-Hoc Query
  - 不可枚举
  - 用户搜索(online)，DB SQL
- 持续计算
  - 计算相对固定、可枚举
  - 数据流动
  - SQL、MR

# IProcess




整体架构




整体拓扑

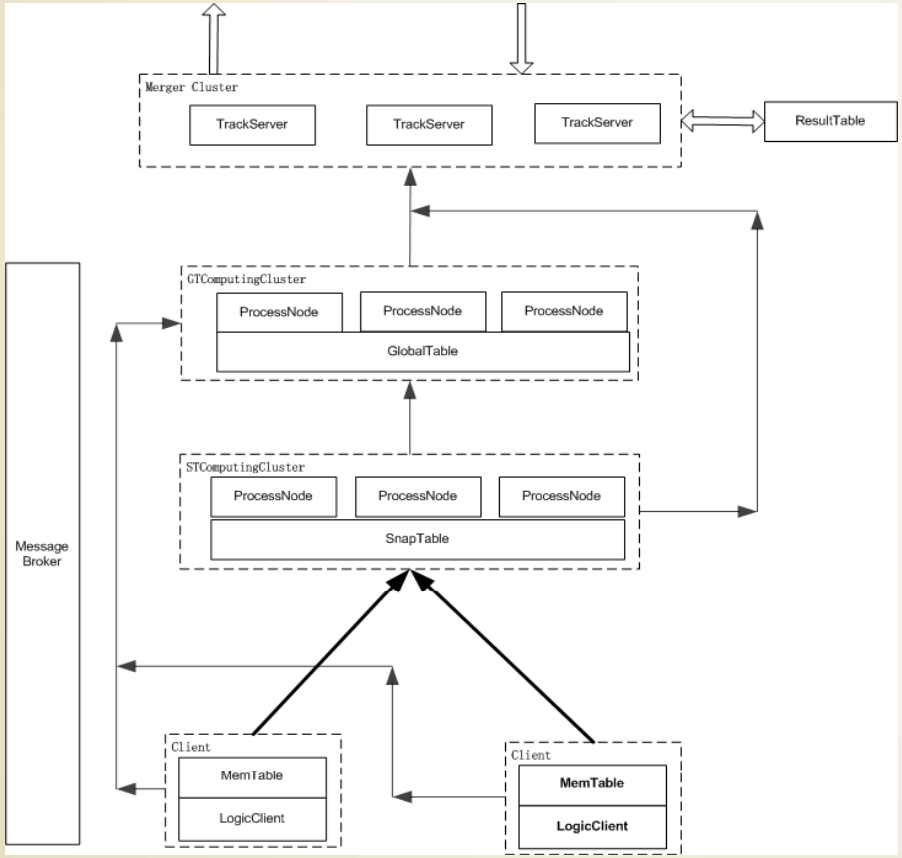
# 运行过程

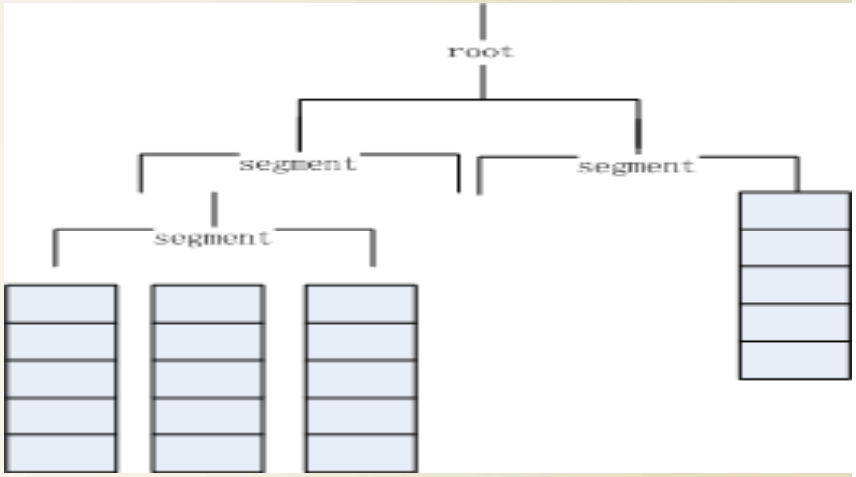
- 三个步骤
  - 简单事件发射（分布式）
  - 复杂事件完备性判断（集中式、分布式）
    - 分布式事务
      - 尽量避免（机制保证）
      - 强事务（MVCC）、逻辑事务、弱事务
  - 触发下一个环节

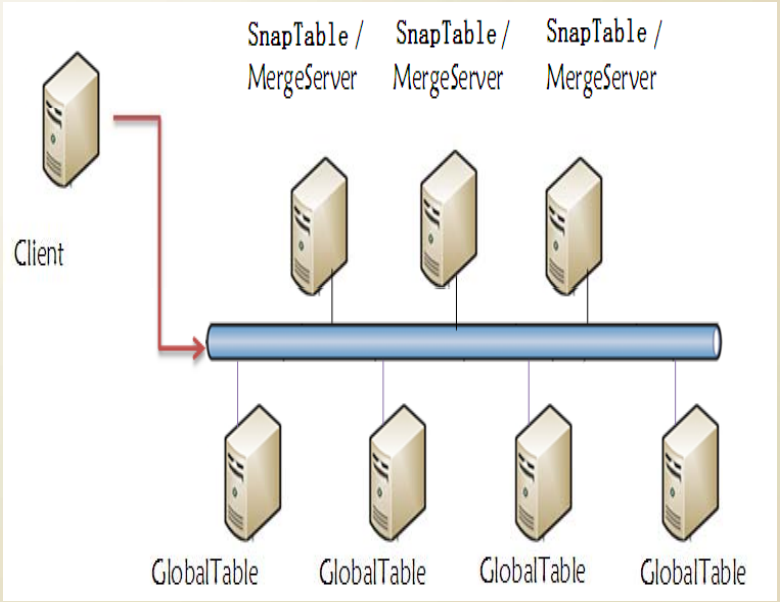
# IProcess的存储

- 树结构的存储
  - 不同的一致性和事务模型
- 区分实时数据与其它数据的存储
- 两级容错
  - 应用级和系统级
- 运算时动态扩容
- 保序
- Latency、throughput、可靠性
  - 动态tradeoff

# IProcess的存储



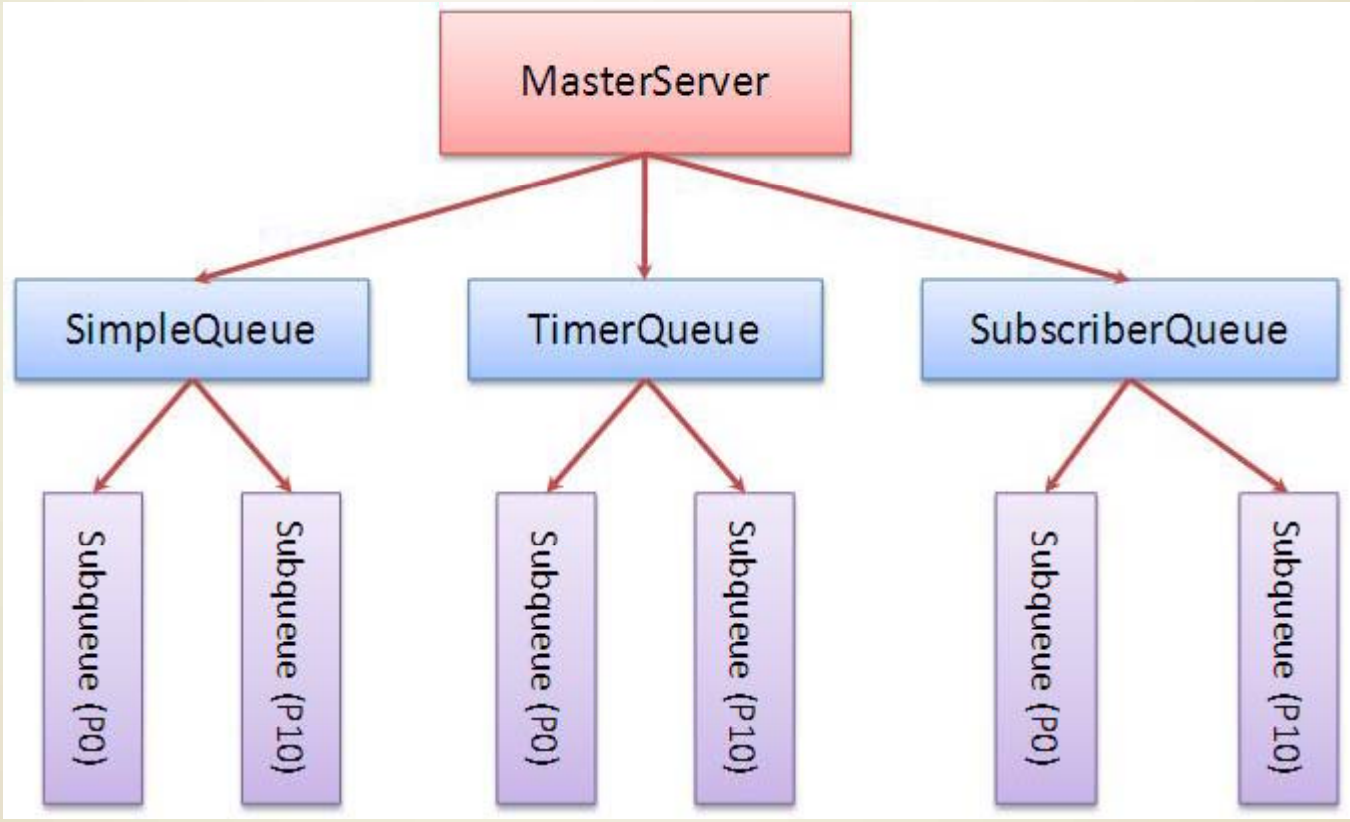




MR模型的本质Reduce(key,valueList,context)

实现STCacheStrategy接口

QStore: 持久化存储。

# IProcess的存储-amber



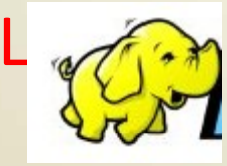

与MR容错性的区别：应用级体现在amber，系统级体现在st与gt

# IProcess的存储- GlobalTable

- Hbase维护分支
  - Segment分裂策略
  - Coprocessor沙箱
  - 类Redis接口
  - 容量规划
  - 剥离行事务
    - Yahoo Omid

# IProcess要点回顾

- 完备事件模型
  - 基础模型：触发器模式
- 可扩展的编程模型(类似于HIVE与hadoop的关系)
  - Spark（类似storm，完全的流处理，无condition）
  - Dumbo（实时MapReduce框架）
  - Graph computing（实时pregel）
  - SQL




# IProcess要点回顾

- 树状存储
- 事务模型
  - 逻辑事务
  - 弱事务
  - 强事务
- 运行时扩容
- 系统，应用量级容错
- 保序

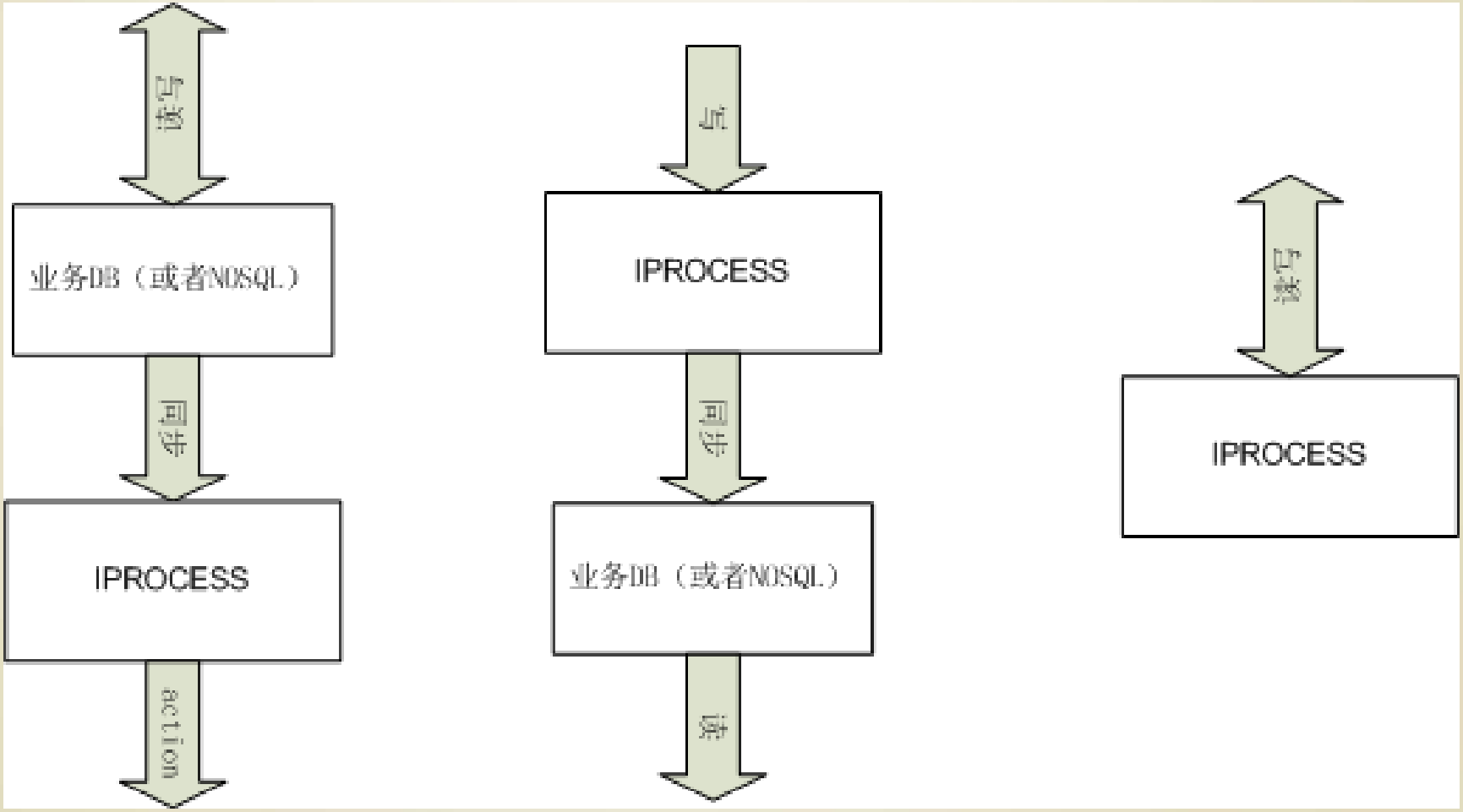
# 应用场景特点

- 响应时间:实时
  - 毫秒级别（子图）
  - 秒级别
  - 分钟级别
- 图复杂度
  - 节点简单且重、图复杂
  - 节点简单但轻、图复杂
  - 节点复杂但轻、图简单
  - 节点复杂且重、图简单

# 应用场景特点

- 语言
  - C++、Java、Shell
  - SQL
  - 规则
  - DSL
- 模型
  - 触发器、简单事件、实时MR、图计算
  - …

# 应用架构

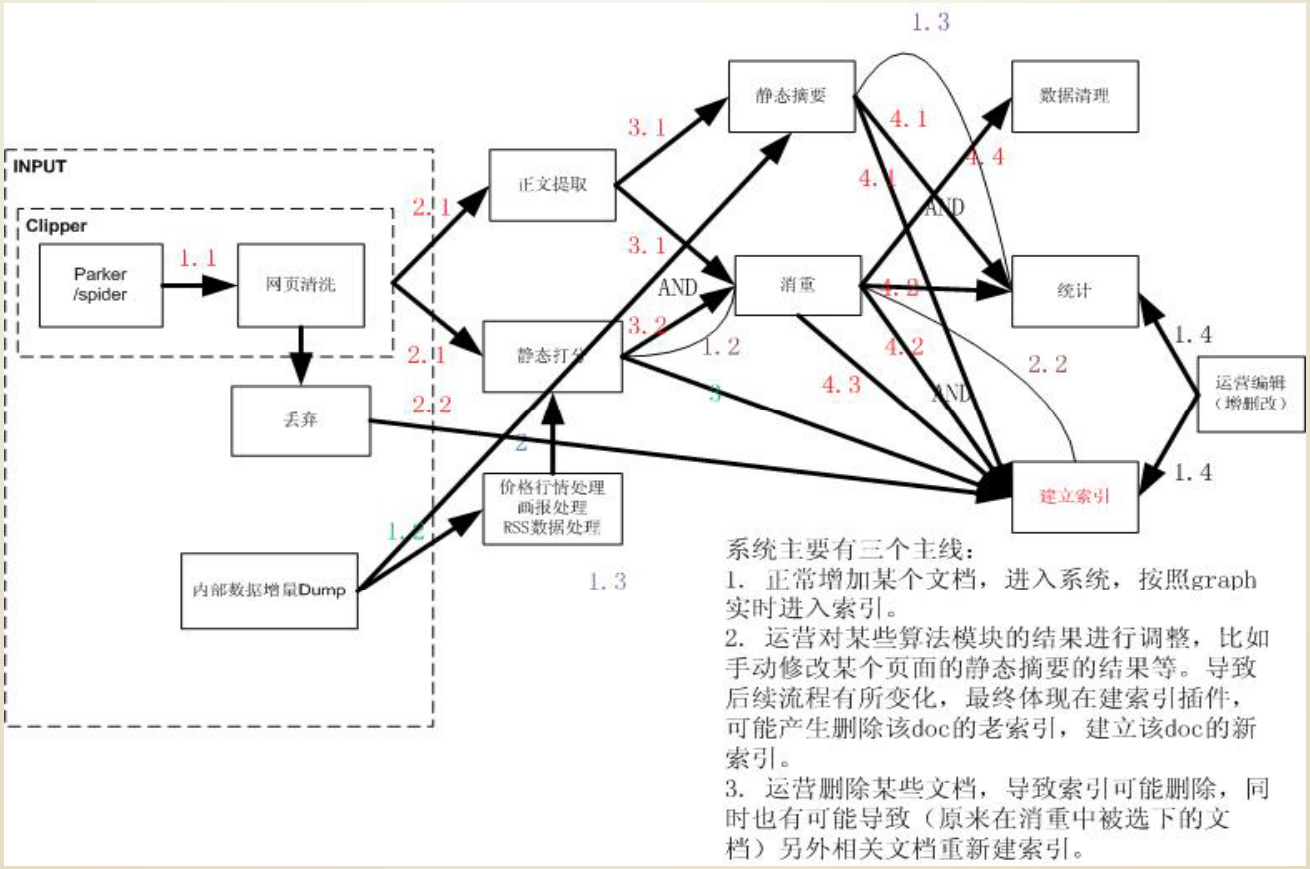

读写
业务DB（或者NOSQL）
同步
IPROCESS
action
写
IPROCESS
同步
业务DB（或者NOSQL）
读
读写
IPROCESS

# How to use?

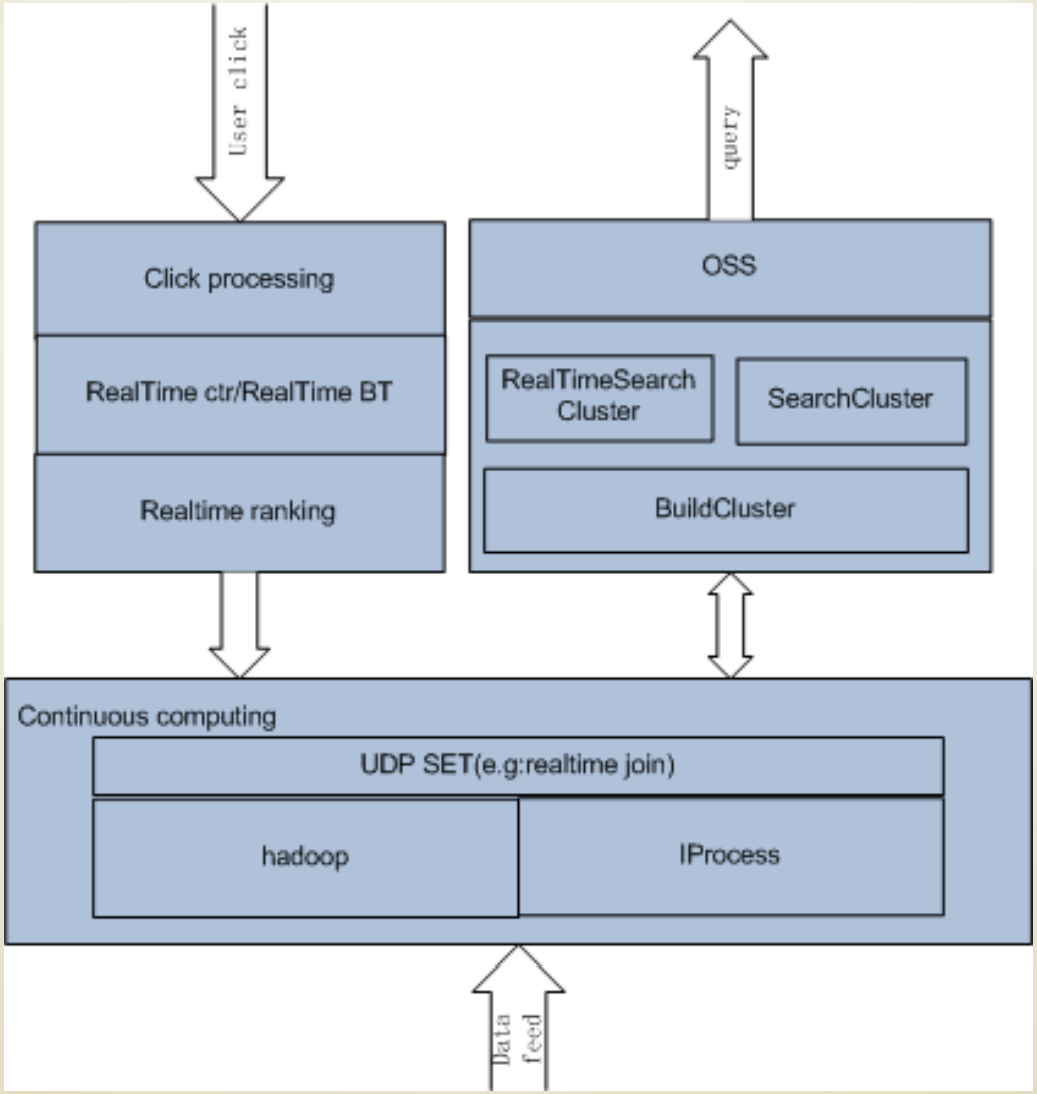
- 使用IProcess需要准备什么？
  - 组件集
  - 配置(有向图，事件)
  - 拓扑

# How to use? A demo

- 简化的资讯实时搜索



# 实时搜索


User click
query
Click processing
RealTime ctr/RealTime BT
Realtime ranking
OSS
RealTimeSearch Cluster
SearchCluster
BuildCluster
Continuous computing
UDP SET(e.g:realtime join)
hadoop
IProcess
Data feed

# 用户API简介

- 系统级（高级接口）
  - STCacheStrategy
  - LogicalConflictResolver
  - LazyConflictResovler
  - IUserDefinedCondition
  - ISeedGenerator
  - IPartitioner

# 用户API简介

- 应用级
  - IProcess原生
    - IProcessModule(JobContext)
  - Spark
    - EventProcessor(EventContext)
  - Dumbo(实时Mapreduce)
    - Mapper
    - Preparer
    - Reducer
    - Merger
      - 使用MapperContext、 ReducerContext等

# 系统API简介

- 重要接口和类
  - TableStrategy
  - StorageStrategy
  - Segment
  - NameMappingSegment。
  - SequencedSegment
  - TimedSegment
  - NameMappingRecord

# Dumbo例子

- 代码直接复用，效果大不一样
- 例子(实时，中间结果可见)
  - wordCount(与全量mapreduce区别在于：dumbo下的wordcount，实时reduce结果是可见的，即整个计算结果中间可被用户访问)
  - 访问记录
    - 一次map、多次reduce
  - SQL执行
  - K-mean聚类
  - 实时join
    - 代码见下页

# Dumbo例子

- 实时join代码(join好一条输出一条)

```
class MemberMapper : public Mapper
{
public:
   void map(const string& key, const RecordPtr value, MapperContextPtr context)
   {
      context->add(value->get_field("member_id").toString(),value,"member");
   }
}
class ProductMapper : public Mapper
{
public:
   void map(const string& key, const RecordPtr value, MapperContextPtr context)
   {
      context-> add (value->get_field("member_id").toString(), value, "product");
   }
}
```

# Dumbo例子

- 实时join代码(reduce触发的条件在配置文件中，即相同joinkey的a数据和b数据都ready(condition)，系统才会实时调用reduce-大家可以比较在storm下实现实时join的代码)

```
int32_t reduce(string key, map<string, RecordIterator> taged_value_iterator, ReducerContext context)
{
        string tag_a = “member";
        string tag_b = “product";
        RecordIterator iterator_a = taged_value_iterator.find("A")->second;
        RecordIterator iterator_b = taged_value_iterator.find("B")->second;
        RecordPtr record_a = iterator_a.begin();
        while(record_a)
        {
                RecordPtr record_b = iterator_b.begin();
                while(record_b)
                {
                        Record result = record_a->join(record_b);
                        context->add(result);//生成join的结果
                        record_b = iterator_b.next();
                }
                record_a = iterator_a.next();
        }
}
```

# 触发器模式例子

- SNS推荐系统
  - 用户将公司名修改，引发推荐的实时变化
  - 某用户增加一个好友会引发对自己和对别人的推荐变化
  - 实时人立方(删除关系)
- 风控CEP
  - 离线风险控制
  - 在线风险控制

# 系统边界

- 目前的问题
  - 跨语言
  - 吞吐量
  - 易用性
  - 服务化，云？
- 边界
  - 计算可枚举
  - 计算可加
  - 依赖相关集较小
- 建模
  - 介于BSP与DOT之间
  - Runtime的execute plan优化

# 目标

- 打造平台
  - 实时计算
  - 持续计算
  - Iprocess将专注于完备事件机制。
    - 只提供最基本的功能，提供高度可定制的接口，上层可定制出不同平台（计算模型）和业务系统。
- 构建技术生态体系
  - 合作开发容器类组件、通用组件
  - 协调可复用子图(物化VIEW)、可复用系统
  - Spark系统，Dumbo系统

# 目标

- 全面提升业务的实时处理能力
  - 落地各个业务线
    - 远离业务的通用平台的生命力不会强
    - 搜索，广告，交易，结算，风控，图算法，数据仓库...
  - 针对业务的响应时间
    - 毫秒、秒级、分钟级
- 业界有影响力的技术产品
  - 具有原创技术并发表高质量有影响力论文
    - 目前已经准备OSDI2012。
  - 开源技术产品

# Iproces RoadMap

- IProcess0.1:
  - 完备事件、图模型
- IProcess0.2:
  - 树存储、完善事件、保序等
- IProcess0.2.1、0.2.2:
  - 实时搜索、交易
- IProcess0.3:
  - 完备事件机制完善、吞吐量、可扩展模型
- IProcess0.3.x:
  - 易用性、落地、SQL规则引擎、完善分布式事务
- IProcess0.4:
  - 开放控制接口、调度、迭代计算、持续计算，组件计算迁移
- IProcess0.5:
  - 监控、服务化、IDE、DEBUG环境。
  - 基本功能完善

# 谢谢

www.weibo.com/iprocess