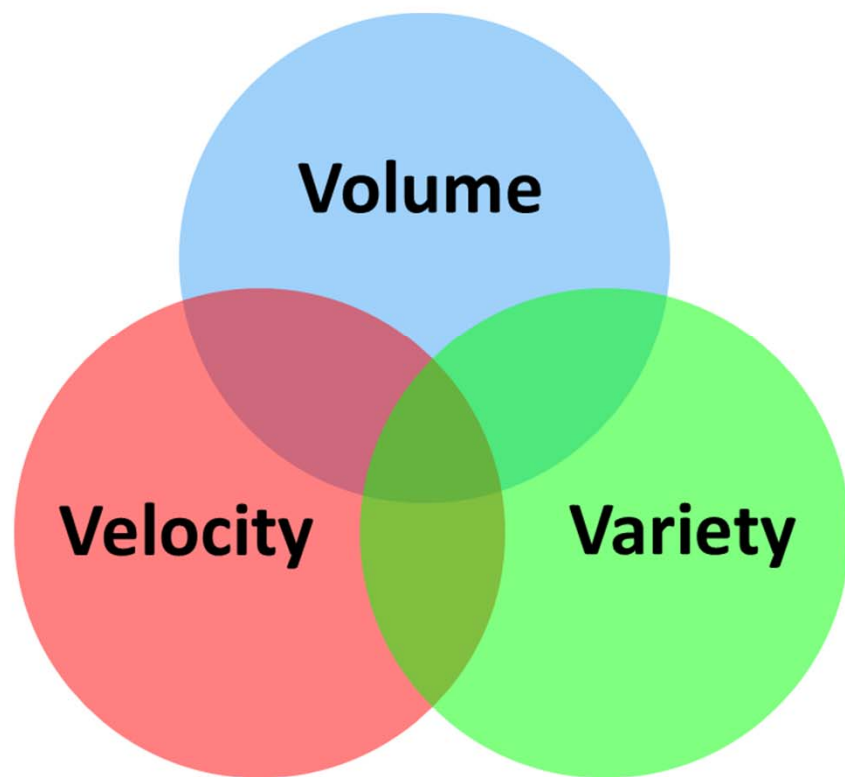


大数据系统与大规模数据分析

大数据运算系统 (2)



陈世敏

中科院计算所
计算机体系结构
国家重点实验室

©2015-2021 陈世敏

作业时间安排

周次	内容	作业
第4周, 3/31	大数据存储系统1: 基础, 文件系统, HDFS	作业1布置
第5周, 4/7	大数据存储系统2: 键值系统	
第6周, 4/14	大数据存储系统3: 图存储, document store	
第7周, 4/21	大数据运算系统1: MapReduce, 图计算系统	作业1提交 作业2布置
第8周, 4/28	大数据运算系统2: 图计算系统, MR+SQL	
第9周, 5/5=>5/8 (周六上周三的课)	大数据运算系统3: 内存计算系统	大作业布置 (系统,6人/组)
第10周, 5/12	分布式哈希表, 区块链技术中的加密算法	作业2提交
第11周, 5/19	最邻近搜索和位置敏感 (LSH) 算法	作业3
第12周, 5/26	奇异值分解与数据空间的维度约化	大作业布置 (分析,3人/组)
第13周, 6/2	推荐系统	大作业 仅选1个
第14周, 6/9	流数据采样与估计、流数据过滤与分析	
第15周, 6/16	期末考试	
第16周, 6/23	大作业验收报告 (上下午, 教1-208)	大作业验收

大数据运算系统比较

MPP
数据库系统

MapReduce

图计算
(Pregel, GAS)

- 不同的编程模型
 - 数据模型、运算方法
- 都可以支持大规模数据处理
 - 并行、可扩展、容错
- MPP：大规模并行

大数据运算系统比较

存储数据的运算处理

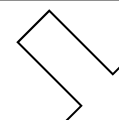
MPP
数据库系统

MapReduce

图计算
(**Pregel, GAS**)



数据流处理



内存计算

Outline

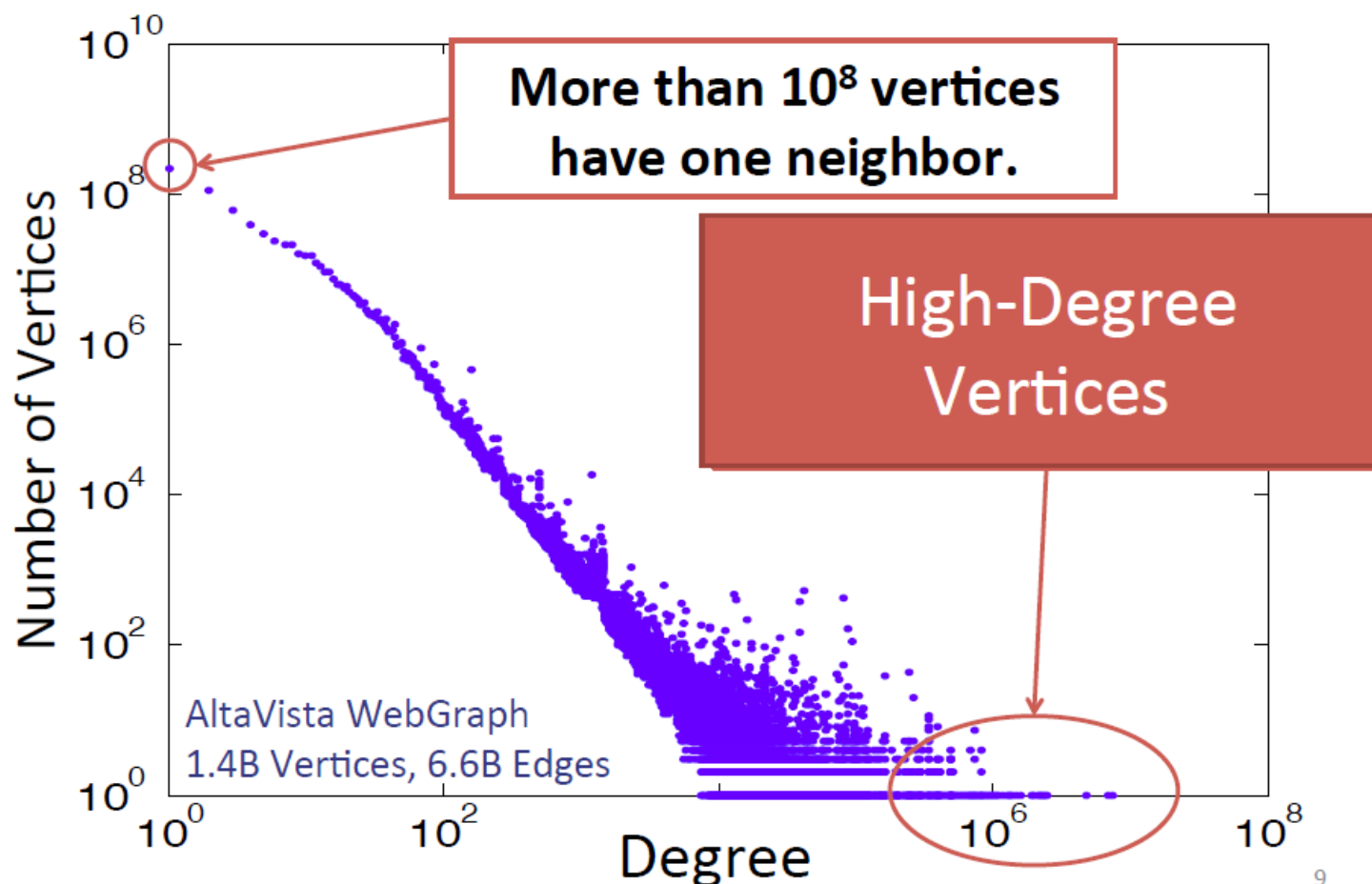
- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统

PowerGraph

- “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.” Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. **OSDI 2012**.
- 下面的slides基于OSDI'12

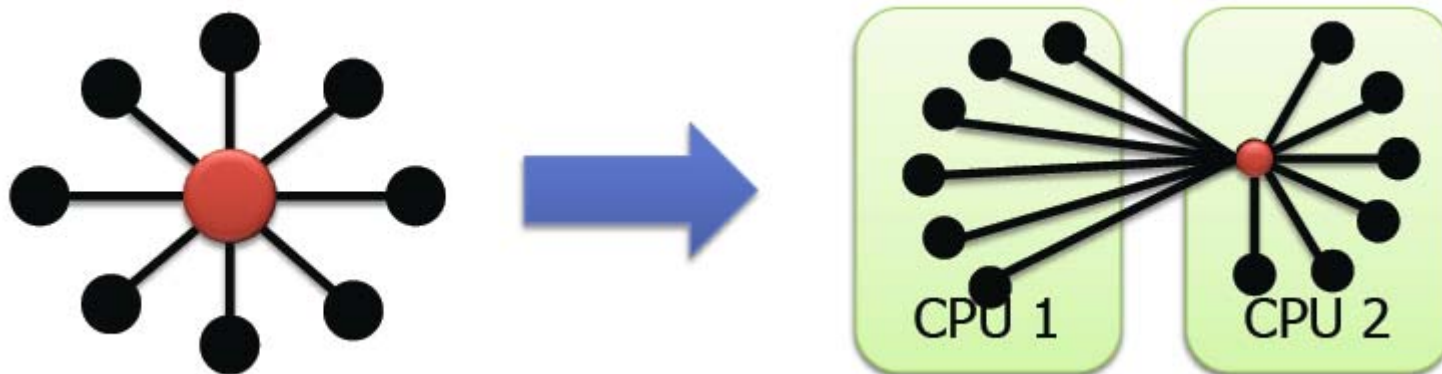
大量自然图符合Power-Law

Power-Law Degree Distribution



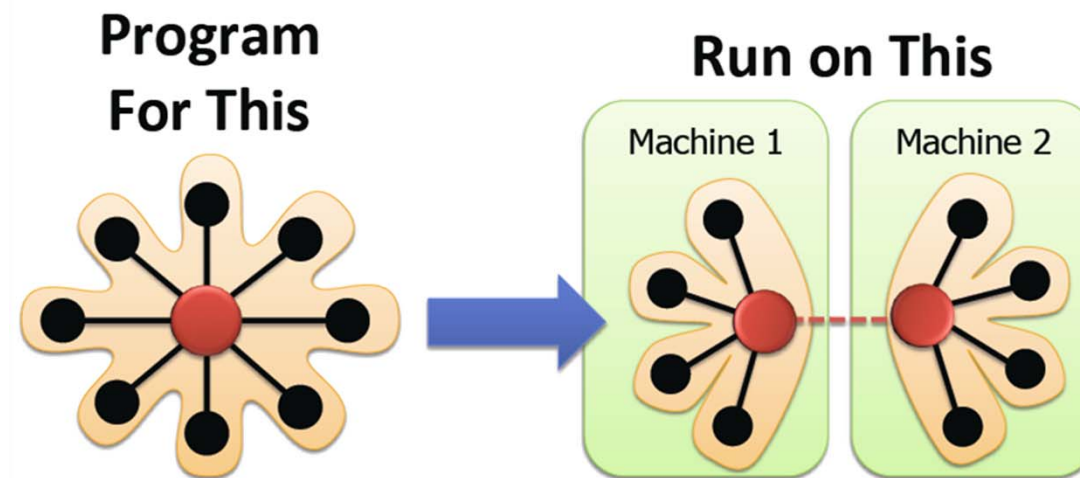
Power-Law引起的问题

- 图划分之间有大量的跨边
- Pregel
 - 需要传输大量的消息
- 如何优化？



PowerGraph

- 把单一的Compute()分成三个用户函数GAS来实现
 - ❑ Gather
 - ❑ Apply
 - ❑ Scatter
- 利用了一大类图计算算法的特点
- 实际的效果
好像把大度顶点
分裂成为了多个



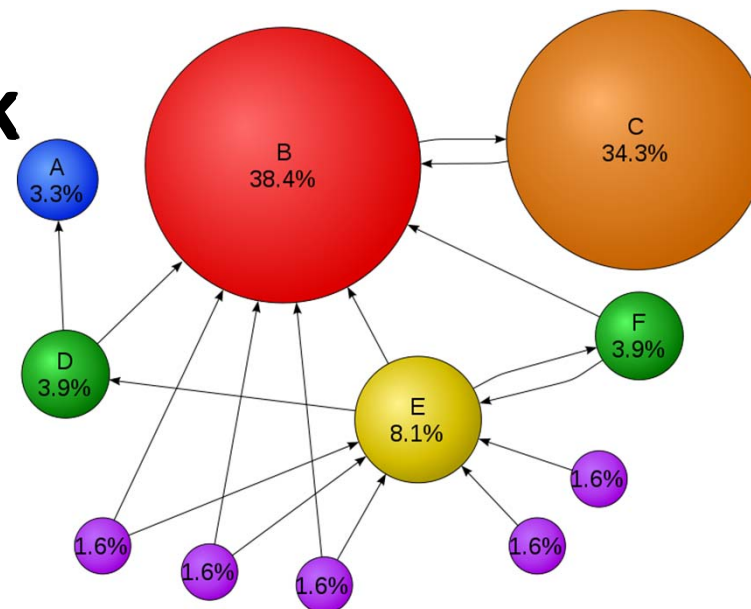
图算法举例：PageRank

- $R_u = 0.15 + 0.85 \sum_{v \in B(u)} \frac{R_v}{L_v}$

- R_v : 顶点v的PageRank*N
- L_v : 顶点v的出度（出边的条数）
- $B(u)$: 顶点u的入邻居集合
- damping factor为0.85
- N: 总顶点个数

- 计算方法

- 初始化：所有的顶点的PageRank为1
- 迭代：用上述公式迭代直至收敛



图来源：Wikipedia

GraphLite (Pregel模型) 的实现

```
double sum= 0.0;
for (; !msgs->done(); msgs->next()) {
    sum += msgs->getValue();
}
val = 0.15 + 0.85 * sum;
*mutableValue() = val;
int64_t n = getOutEdgeIterator().size();
sendMessageToAllNeighbors(val / n);
```

接收
消息

更新
状态

发送
消息

GAS抽象

```
double sum= 0.0;  
for (; !msgs->done(); msgs->next()) {  
    sum += msgs->getValue();  
}
```

改为
Gather

```
val = 0.15 + 0.85 * sum;  
*mutableValue() = val;
```

改为
Apply

```
int64_t n = getOutEdgeIterator().size();  
sendMessageToAllNeighbors(val / n);
```

改为
Scatter

GAS Decomposition

Gather (Reduce)

Accumulate information about neighborhood

User Defined:

► **Gather**() $\rightarrow \Sigma$

► $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$



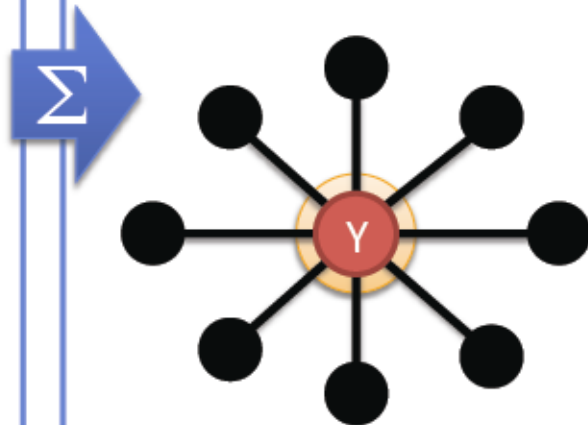
Parallel Sum  +  + ... +  $\rightarrow \Sigma$

Apply

Apply the accumulated value to center vertex

User Defined:

► **Apply**(, Σ) \rightarrow 

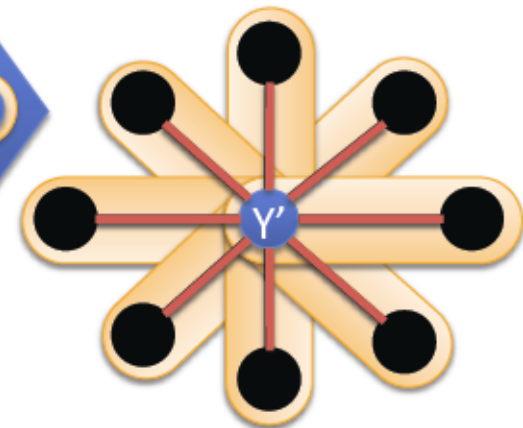


Scatter

Update adjacent edges and vertices.

User Defined:

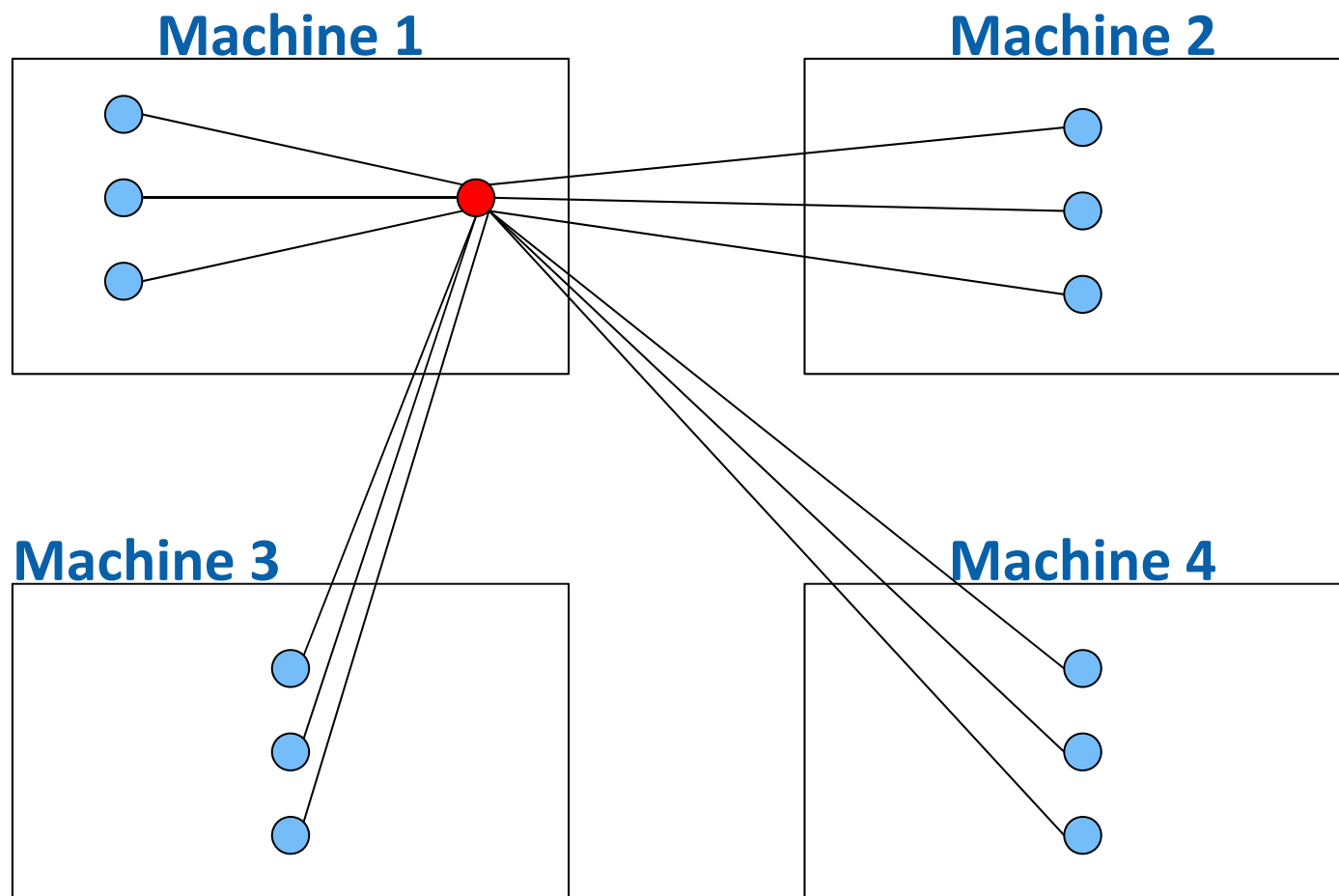
► **Scatter**() \rightarrow 



Update Edge Data & Activate Neighbors

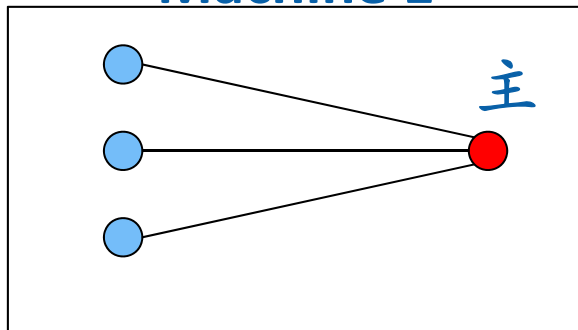
33

具体实现：通常的划分

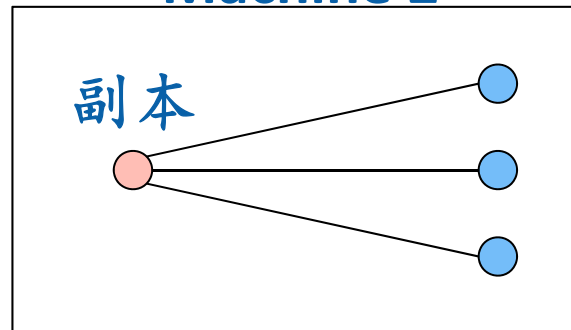


具体实现：PowerGraph

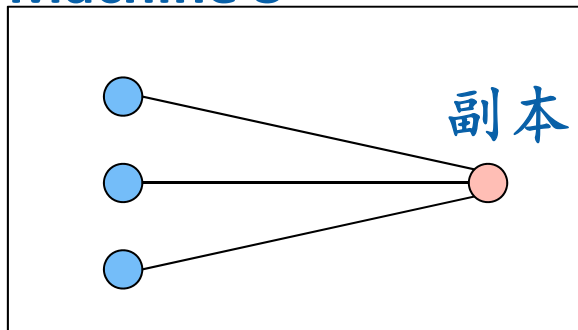
Machine 1



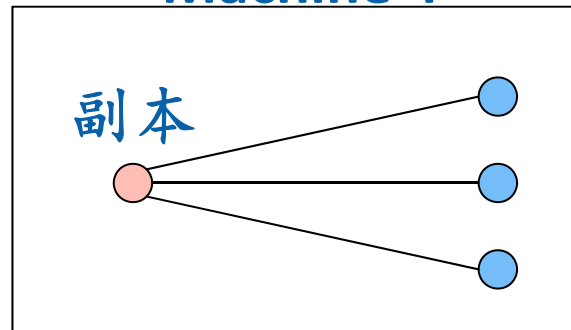
Machine 2



Machine 3



Machine 4



GAS Decomposition

Gather (Reduce)

Accumulate information
about neighborhood

User Defined:

► **Gather** () $\rightarrow \Sigma$

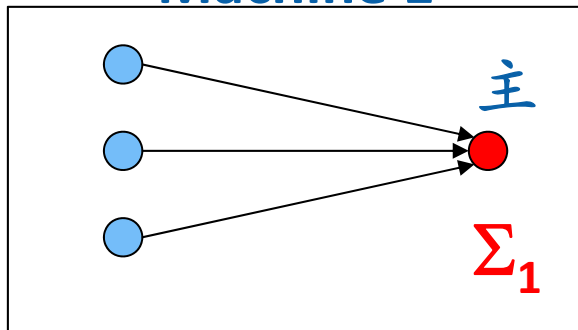
► $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$



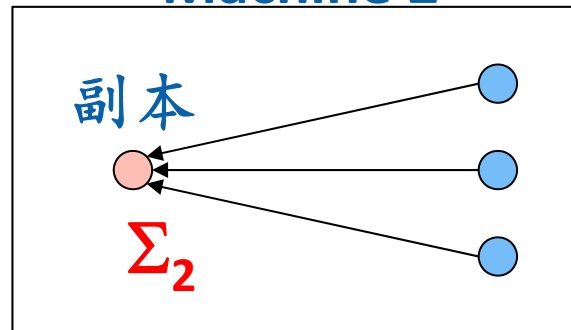
Parallel
Sum  +  + ... +  $\rightarrow \Sigma$

具体实现：Gather，收消息求局部和

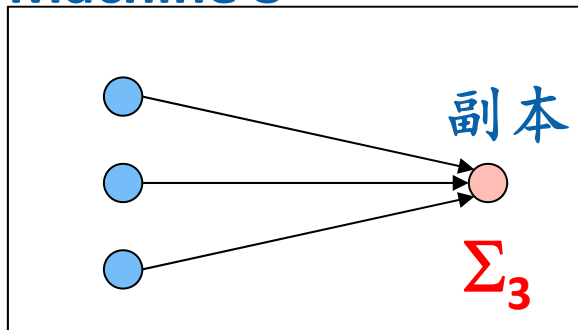
Machine 1



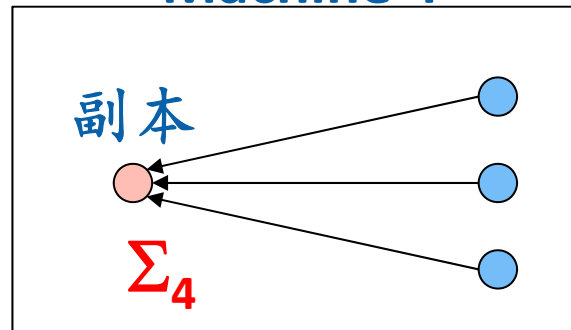
Machine 2



Machine 3



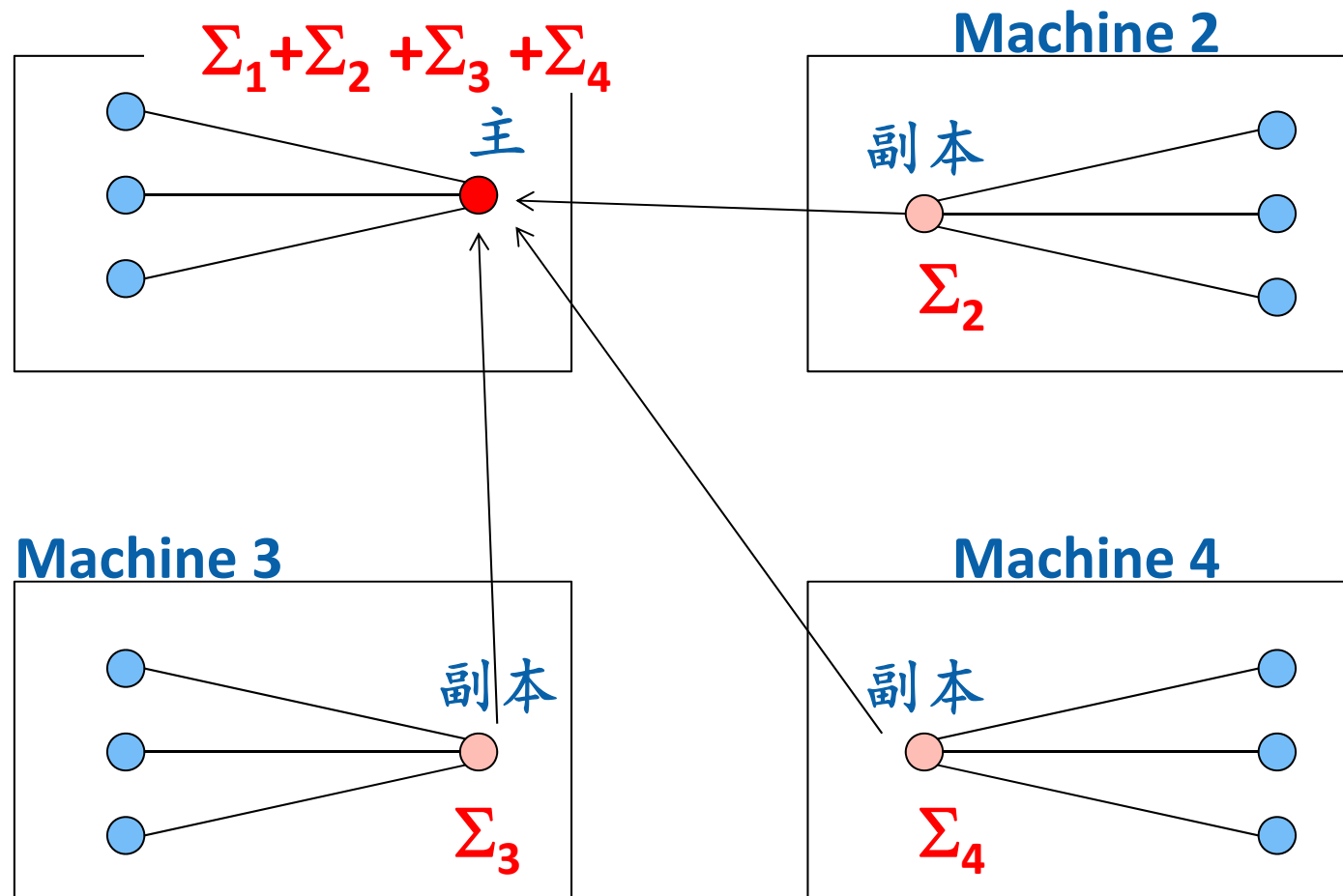
Machine 4



► **Gather**() $\rightarrow \Sigma$

► $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$

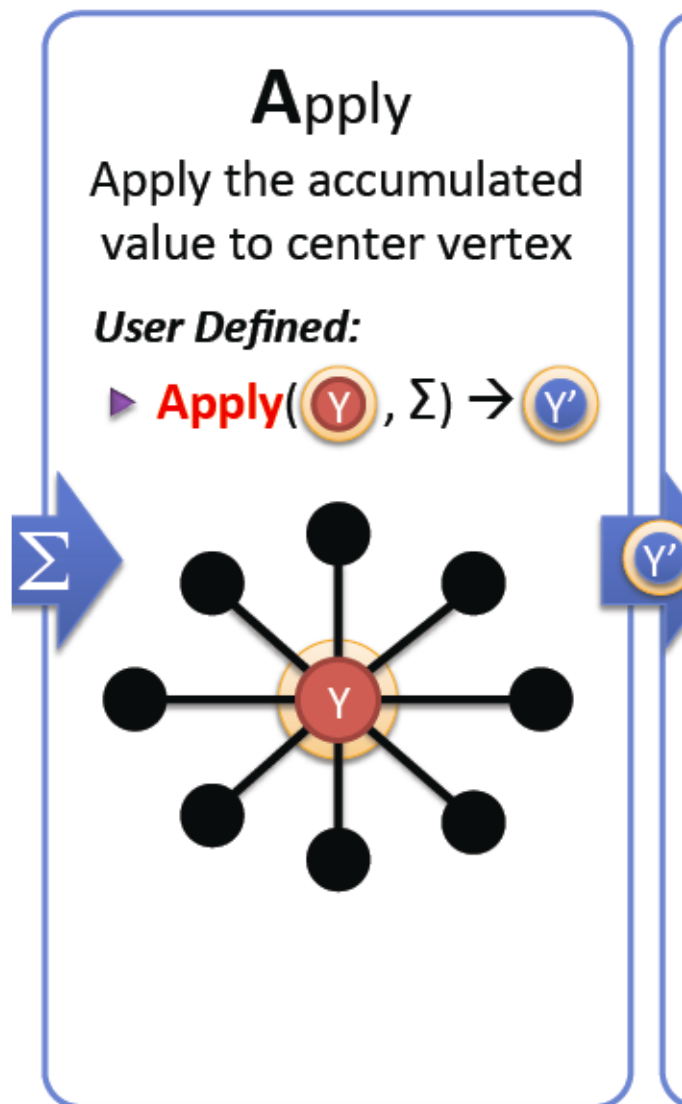
具体实现：Gather，局部和 \rightarrow 全局和



► **Gather**() $\rightarrow \Sigma$

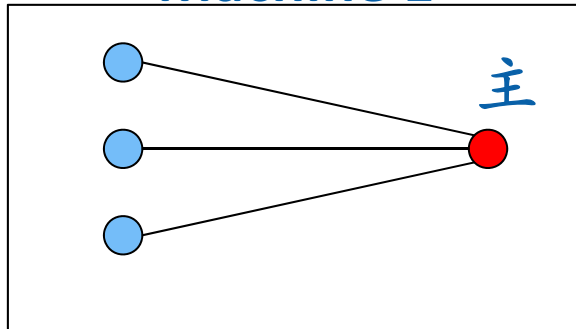
► $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$

GAS Decomposition

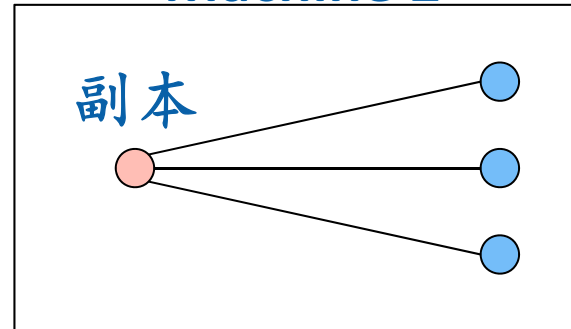


具体实现：Apply，在主顶点更新PageRank

Machine 1

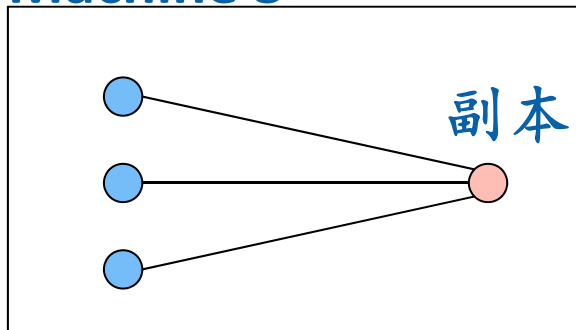


Machine 2

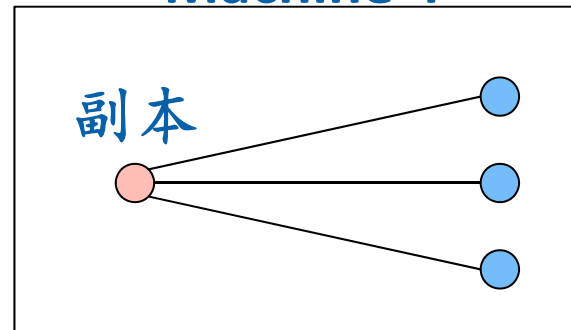


$$\text{Apply}(\text{Y}, \Sigma) \rightarrow \text{Y}'$$

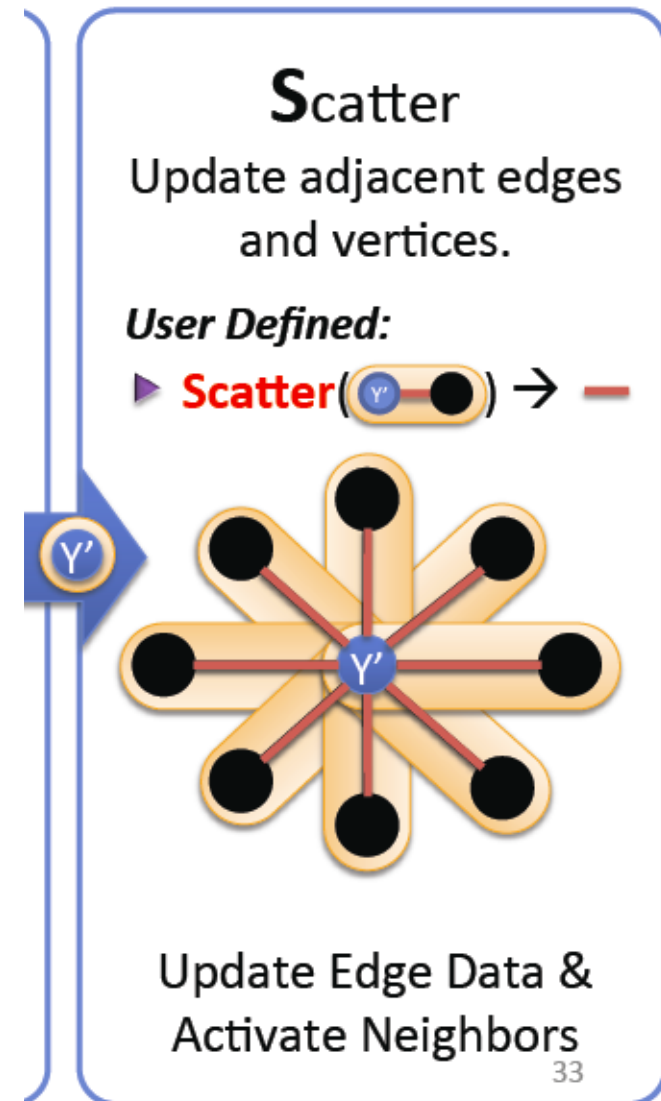
Machine 3



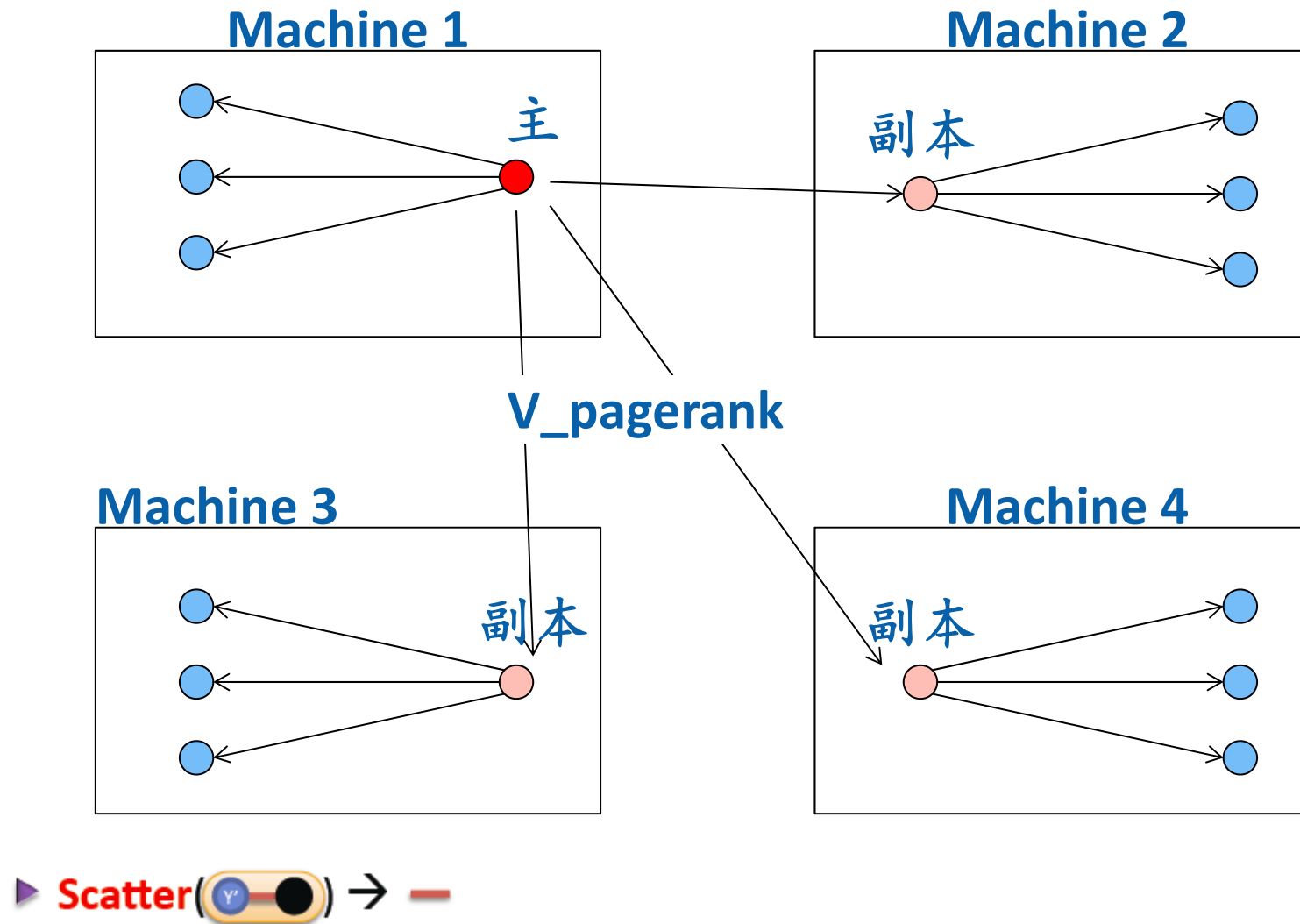
Machine 4



GAS Decomposition



具体实现：Scatter, 发送消息



GAS模型小结

- 针对Power-law图中跨边很多的问题
 - Pregel模型中可能引起大量的消息传递代价
- 提出GAS模型：把Pregel compute分为三个函数
 - Gather
 - 通过Reduce减少跨机器的消息
 - Apply
 - Scatter
 - Copy顶点状态减少跨机器的消息

Outline

- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统

MapReduce+SQL系统介绍

- MapReduce提供了一个分布式应用编写的平台
 - 程序员开发串行的Map和Reduce函数
 - 在串行的环境开发和调试
 - MapReduce系统可以在成百上千个机器节点上并发执行MapReduce程序，从而实现对大规模数据的处理
- MapReduce的问题
 - 这是一个编程的平台
 - 不太适合数据分析师的使用
 - 即使最基础的选择和投影操作，也必须写程序实现
- 对SQL的需求由此产生

MapReduce+SQL系统介绍

- 产业界研发了许多系统，希望在云平台上增加一层类似SQL的支持
- 这类系统包括
 - ❑ Facebook Hive
 - ❑ Yahoo Pig
 - ❑ Microsoft Scope
 - ❑ Google Sawzall
 - ❑ IBM Research JAQL
- 一些数据仓库产品也把云计算的能力集成进Execution engine
 - ❑ Greenplum, Aster Data, Oracle
 - ❑ 基于内部的某种MapReduce实现或者Hadoop
- Hive不是最早出现的，也不是最具创新性的，但是目前它被非常广泛地使用，我们主要介绍Hive

Hive



- Hive是蜂巢

- 简要发展

- 2008年，Facebook由于数据分析的需求研发了Hive
- Facebook公开了Hive的源码，Hive成为Apache开源项目
- 2008年3月，Facebook每天把200GB数据存入Hive系统
- 2012年，每日存入的数据量超过了15TB

- Hive

- 管理和处理结构化数据
- 在Hadoop基础上实现
- 提供类似SQL的HiveQL语言

Hive 系统



Hive CLI, JDBC, etc.

Browsing DDL Queries

目标

- 存储关系表 → HDFS
- 存储元信息
- 支持SQL → MapReduce

Adapted from Hive ApacheCon'08 talk

Map Reduce

HDFS

Hive 系统



Hive CLI, JDBC, etc.

Browsing DDL Queries

目标

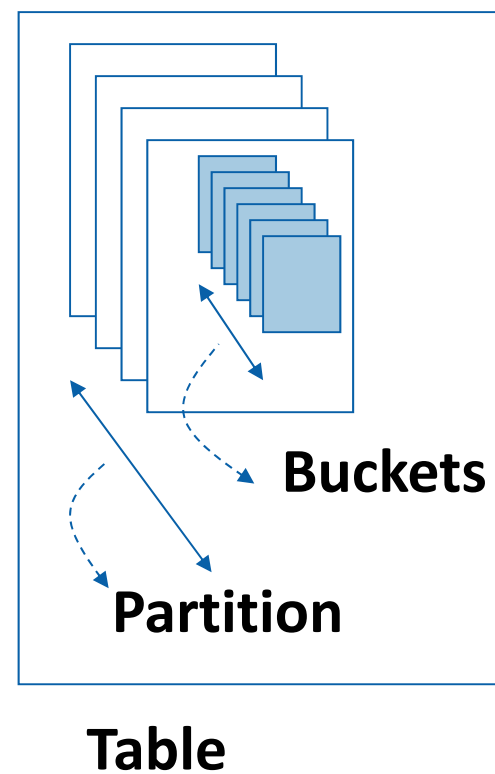
- 存储关系表 → HDFS
- 存储元信息
- 支持SQL → MapReduce

Adapted from Hive ApacheCon'08 talk

Map Reduce

HDFS

数据存储



Hive 系统

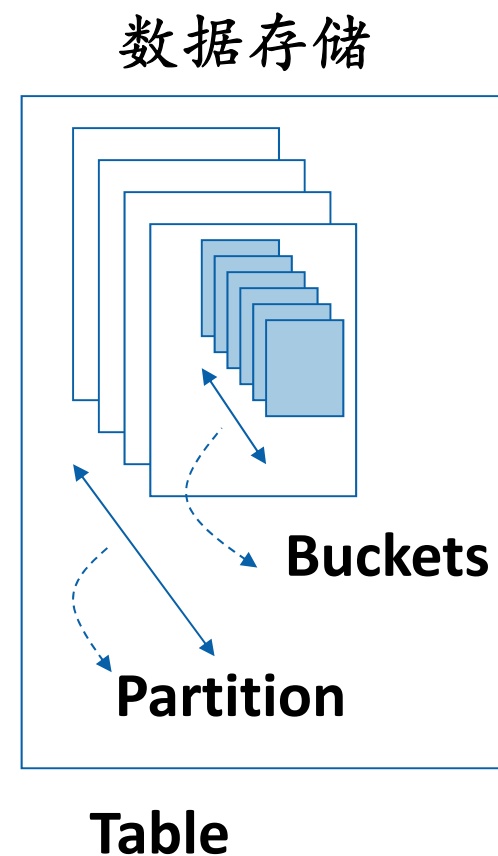
Adapted from Hive ApacheCon'08 talk



Map Reduce

HDFS

- 数据存储在HDFS上
 - hdfs目录: /usr/hive/warehouse/
- Table: 一个单独的hdfs目录
 - /user/hive/warehouse/表名
- Table可以进一步划分为Partition
- Partition可以进一步划分为Bucket



Hive 系统

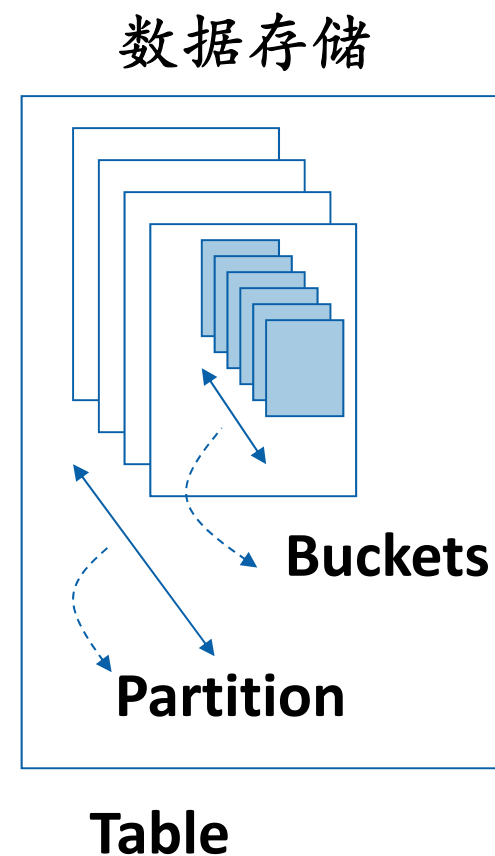
Adapted from Hive ApacheCon'08 talk



Map Reduce

HDFS

- Partition: 每个Partition是Table目录下的子目录
 - 假设pkey是partition key:
/user/hive/warehouse/表名/pkey=value
- Bucket: 每个Bucket是Partition目录下一个子目录
 - 假设pkey是partition key, bkey是bucket key:
/user/hive/warehouse/表名
/pkey=value/bkey=value



Hive 系统



Hive CLI, JDBC, etc.

Browsing DDL Queries

Thrift API

MetaStore

Schema,
存储方式等

Adapted from Hive ApacheCon'08 talk

Map Reduce

HDFS

目标

- 存储关系表 → HDFS
- 存储元信息
- 支持SQL → MapReduce

Hive 系统

Adapted from Hive ApacheCon'08 talk



Hive CLI, JDBC, etc.

Browsing DDL Queries

Thrift API

MetaStore

Schema,
存储方式等

Map Reduce

HDFS

- MetaStore

- 存储表的定义信息等
- 默认在本地\${HIVE_HOME}/metastore_db中
- 也可以配置存储在数据库RDBMS系统中

Hive 系统



Hive CLI, JDBC, etc.

Browsing DDL Queries

Thrift API

MetaStore

Schema,
存储方式等

Adapted from Hive ApacheCon'08 talk

Map Reduce

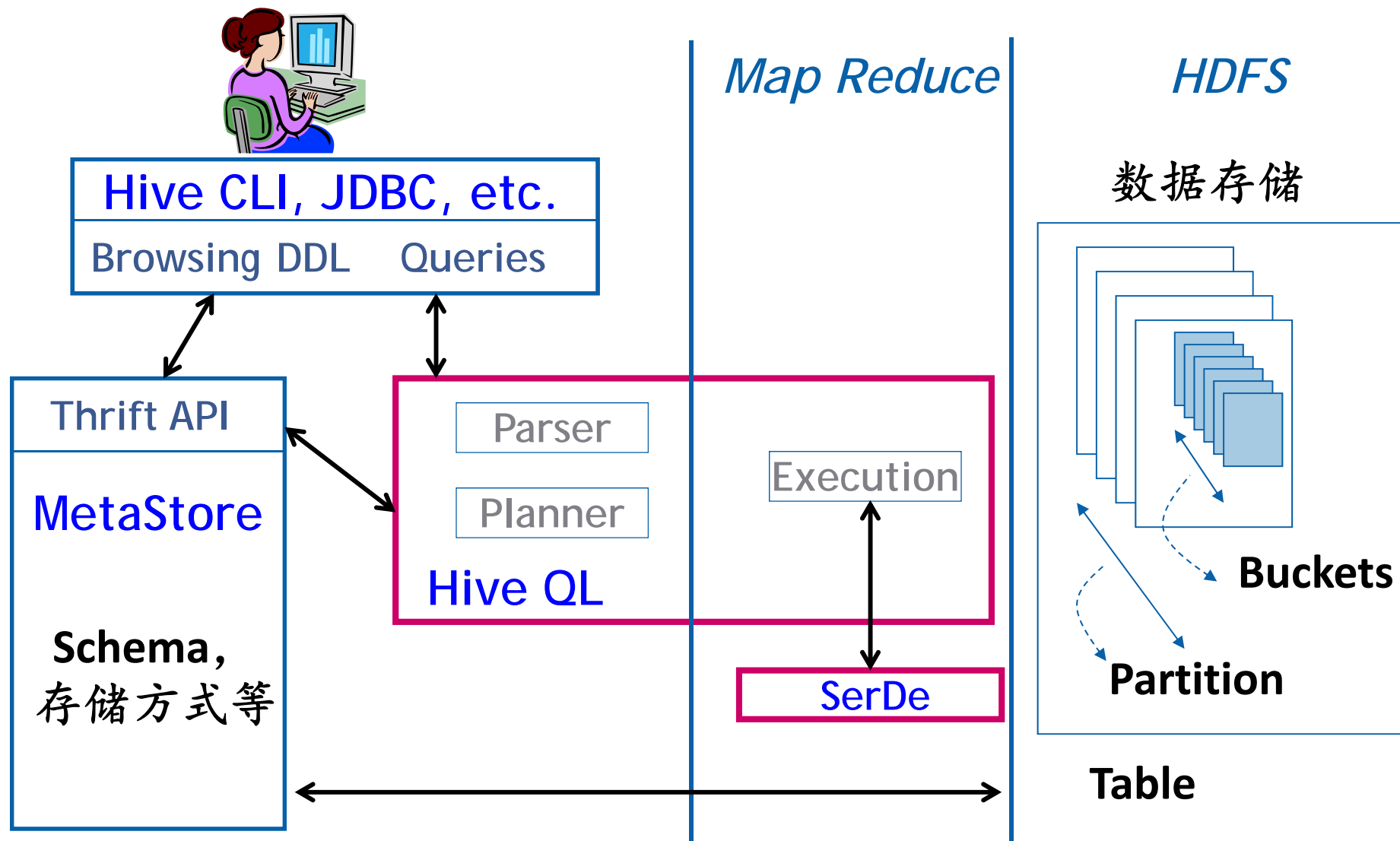
HDFS

目标

- 存储关系表 → HDFS
- 存储元信息
- 支持SQL → MapReduce

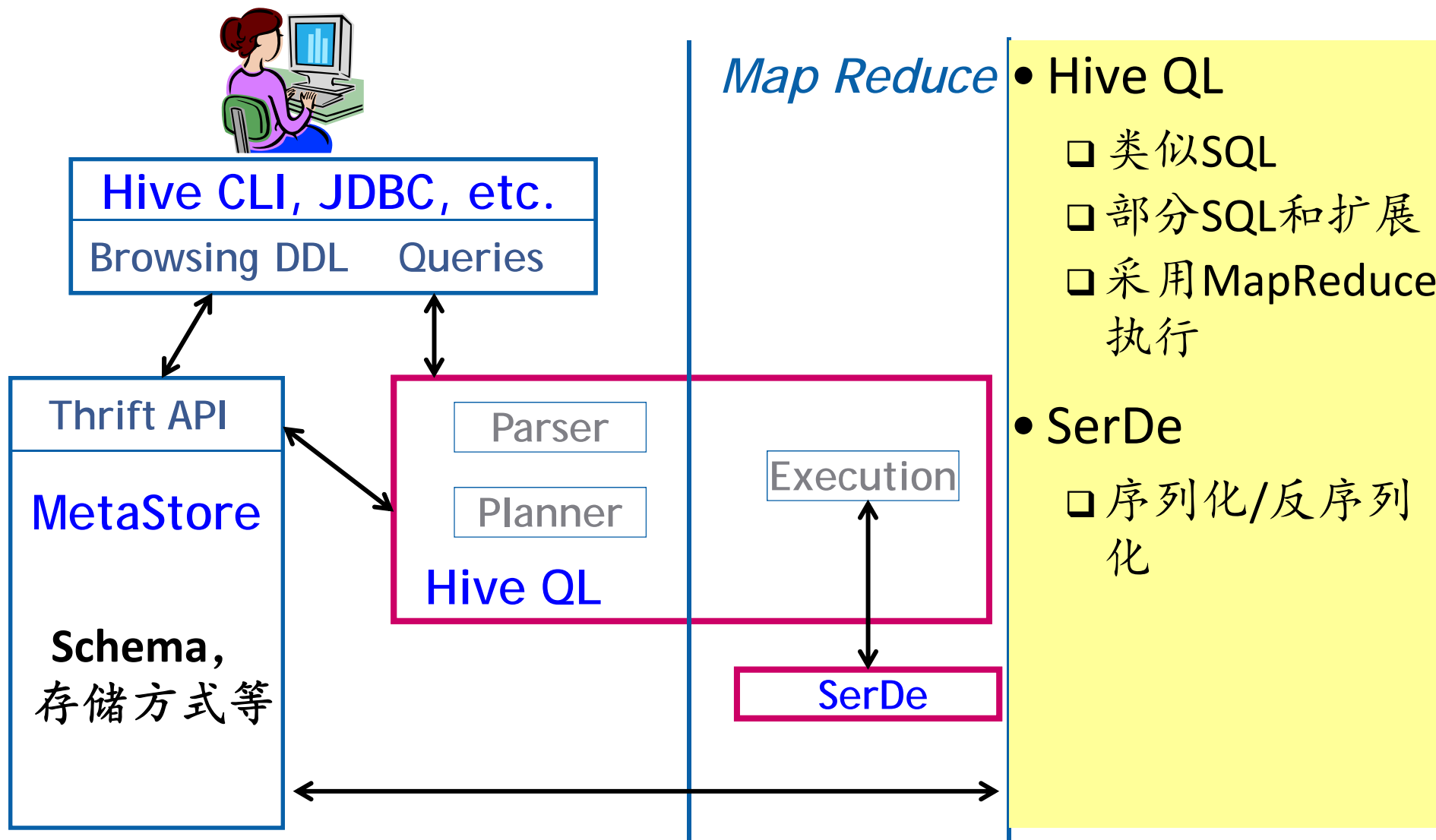
Hive 系统

- 支持 SQL → MapReduce



Hive 系统

- 支持SQL → MapReduce



HiveQL 举例

```
CREATE TABLE status_updates(  
    userid int,  
    status string  
)  
  
STORED AS SEQUENCEFILE;
```

存储在HDFS中
/user/hive/warehouse/status_update

Hive数据模型

- 关系型表+扩展

- 扩展1

- 列可以是更加复杂的数据类型

- ARRAY<data-type>

- 例如: a ARRAY<int>: a[0], a[1], ...

- MAP<primitive-type, any-type>

- 例如: m MAP<STRING, STRING>: m['key1'],...

- STRUCT<col_name: data_type, ...>

- 例如: s STRUCT {c: INT, d: INT}: s.c, s.d

Hive数据模型

- 关系型表+扩展
- 扩展1

```
CREATE TABLE t1(  
    st string,  
    fl float,  
    a array<int>,  
    m map<string, string>  
    n array<map<string, struct<p1:int, p2:int>>  
);
```

系统把复杂数据类型Serialize/deserialize (序列化/反序列化)
使用: `t1.n[0]['key'].p2`

Hive数据模型

- 关系型表+扩展
- 扩展2
 - 可以直接读取已有的外部数据
 - 程序员提供一个SerDe的实现
 - 只有在使用时，才转化读入

```
add jar /jars/myformat.jar;
```

```
CREATE TABLE t2
```

```
ROW FORMAT SERDE 'com.myformat.MySerDe';
```


Create/Alter/Drop Table

- 支持SQL的DDL (data definition language)
 - ❑ Create table
 - ❑ Alter table
 - ❑ Drop table

Insert

```
Insert into table status_updates values (123,  
    'active'), (456, 'inactive'), (789, 'active');
```

```
Insert into table status_updates  
    select 语句
```

```
Insert overwrite table status_updates  
    select 语句
```

注意：HDFS文件不支持修改

- Insert into是文件append
- Insert overwrite是删除然后新创建文件

举例

```
CREATE TABLE status_updates(  
    userid int,  
    status string  
)  
PARTITIONED BY (ds string, hr int)  
STORED AS SEQUENCEFILE;
```

注意： *ds*是partition key, *hr*是bucket key
它们都不包括在table schema中

存储在HDFS中，例如

/user/hive/warehouse/status_update/ds=v1/hr=v2

Partition使用举例

```
INSERT OVERWRITE TABLE  
status_updates PARTITION(ds='2009-01-01', hr=12)  
SELECT * FROM t;
```

在如下的子目录中，存储select的输出

/user/hive/warehouse/status_updates/ds=2009-01-01/hr=12

划分是手工进行的！

```
SELECT * FROM status_updates WHERE ds='2009-01-01';
```

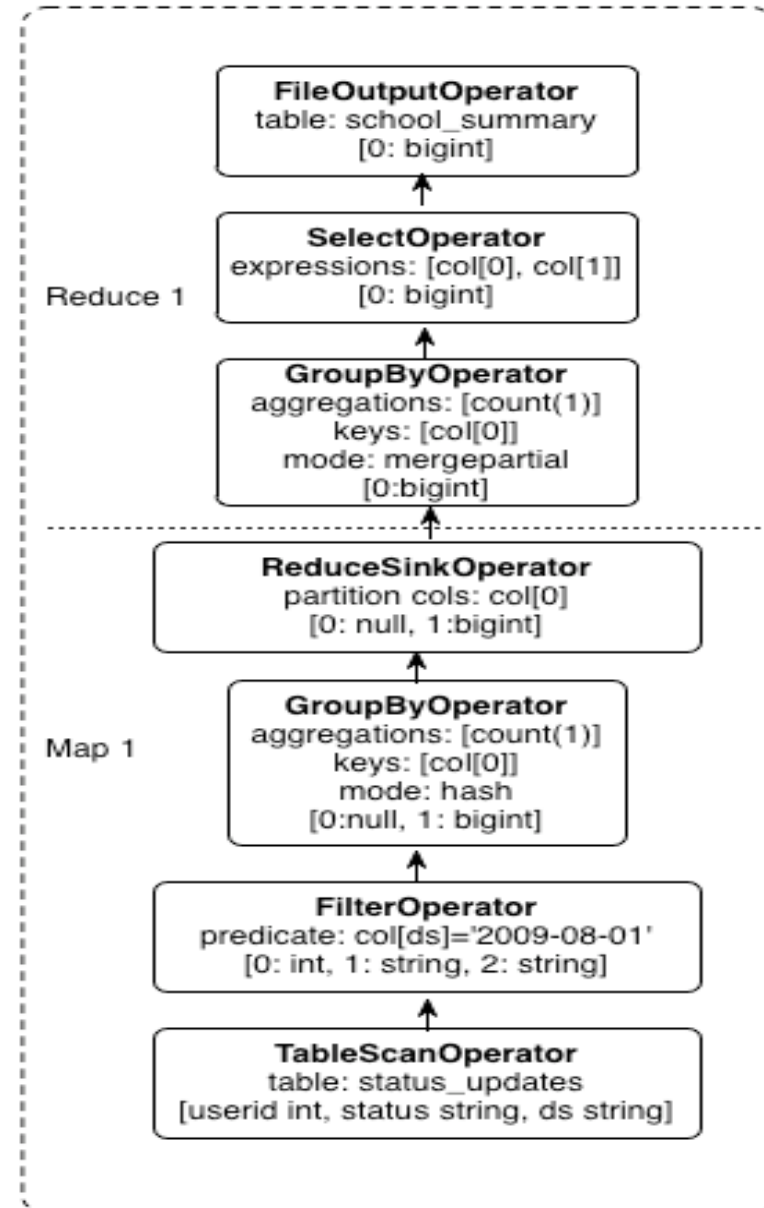
ds是partition key， 所以Hive只使用对应的子目录中的数据

Example Query (Aggregation)

```
SELECT COUNT(1)
FROM status_updates
WHERE ds = '2009-08-01'
```

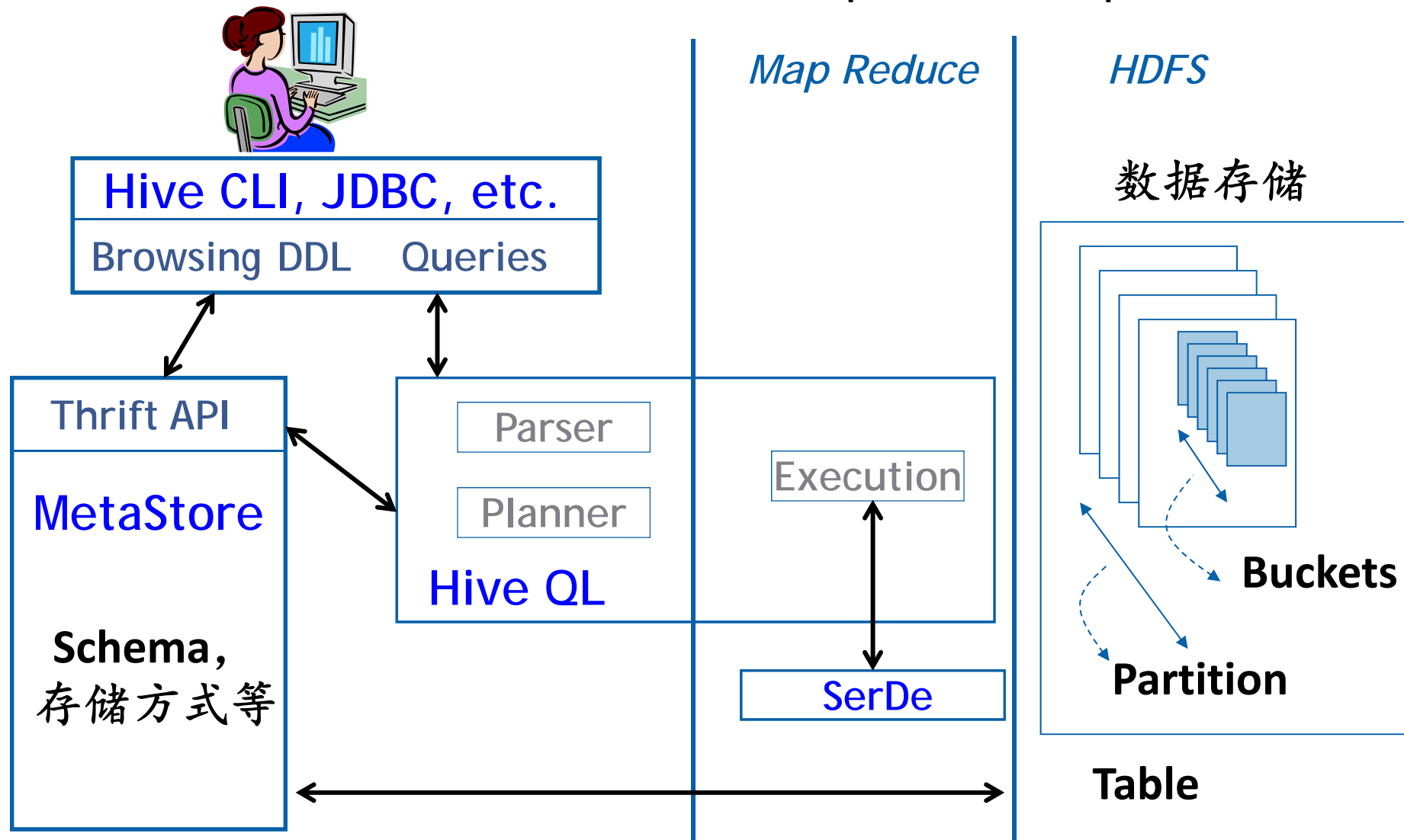
```
status_updates (
  userid int,
  status string
)
Partition key: ds
```

Hive ICDE'10 talk



Hive 系统

Adapted from Hive ApacheCon'08 talk



Hive & Hadoop Usage @ Facebook

Hive ICDE'10 talk

- Hive simplifies Hadoop

- ☐ New engineers go through a Hive training session
- ☐ ~200 people/month run jobs on Hadoop/Hive
- ☐ Analysts (non-engineers) use Hadoop through Hive
- ☐ 95% of hadoop jobs are Hive Jobs

- Types of Applications

- ☐ Reporting
 - Eg: Daily/Weekly aggregations of impression/click counts
- ☐ Ad hoc Analysis
 - Eg: how many group admins broken down by state/country
- ☐ Machine Learning (Assembling training data)
 - Ad Optimization, Eg: User Engagement as a function of user attributes
- ☐ Many others

Outline

- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统

数据流 vs. 存储数据

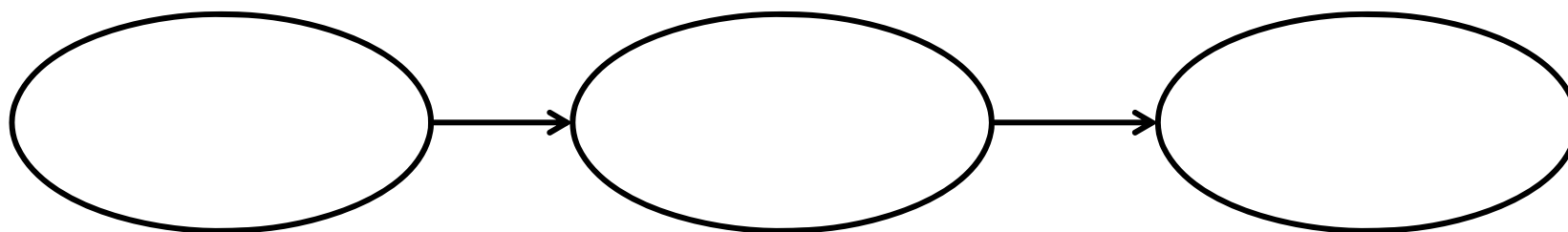
- 数据流概念

- 数据流过系统，在流动的数据上完成处理
- 只看到一次数据
- 看到数据的顺序确定

- 对比存储数据

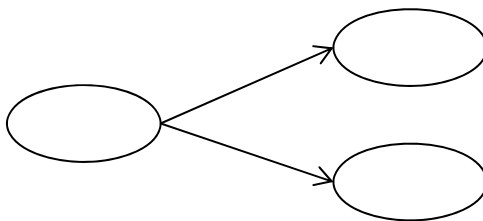
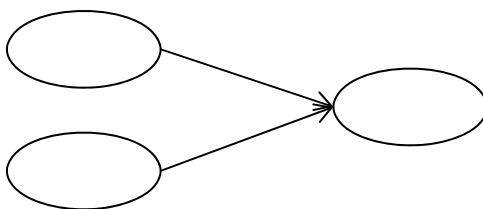
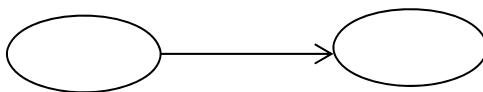
- 可以随机访问数据
- 可以多次访问数据

数据流运算可以表达为一张图



- 图的顶点：数据处理运算
- 图的边：运算之间数据流动的方向

数据流图可以有很多不同的结构



- 可以有很多不同结构

无状态和有状态的运算

- 无状态

- 有些运算是无状态的
- 可以对流过的单条记录进行计算
- 例如，选择（过滤），投影，Map

- 有状态

- 有些运算是有关状态的
- 需要对多条记录进行计算
- 例如，groupby + aggregation, join

两类有状态的运算

- 在所有记录上计算

- 比如：统计信息

- 数据流记录代表一次用户登陆，需要统计总登陆次数

- 数据流运算算子上记录状态信息

- 对每条记录都处理更新状态信息

- 基于Window的计算

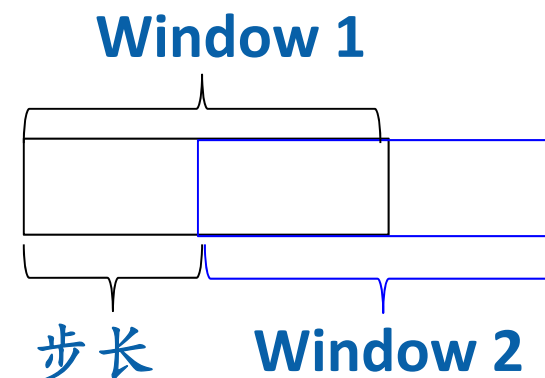
- 对一个窗口内的所有记录完成计算

- 例如，基于窗口的Aggregation，基于窗口的Join

- 统计每小时的登陆次数

- 设置窗口长度：时间、记录数

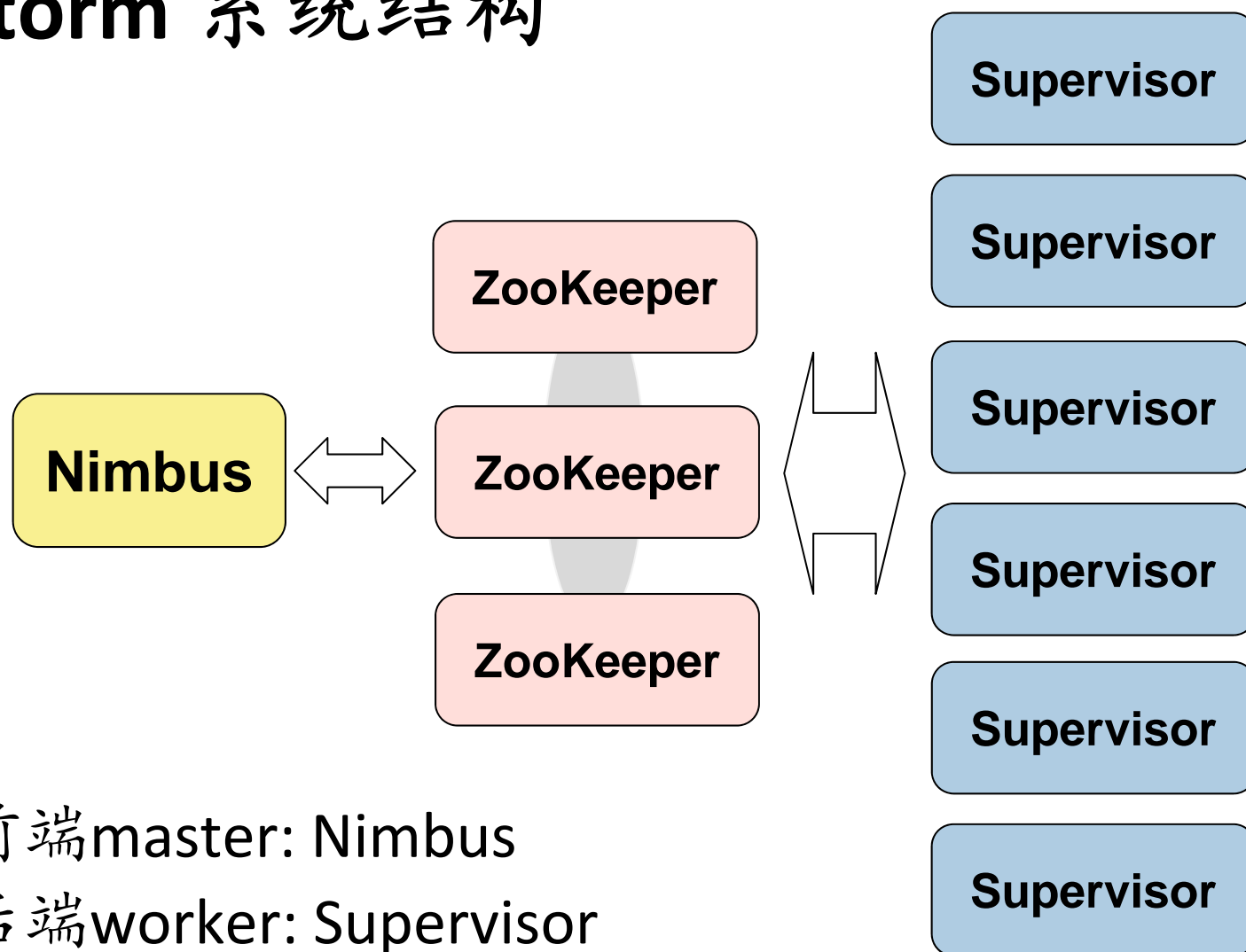
- 设置相邻两个窗口之间的距离：步长



Apache Storm

- Twitter于2011年9月发布
 - 目前是Apache的开源项目
- 数据流处理
- 内部实现：Java 与 Clojure混合实现
 - 大部分功能代码是Clojure写的
 - Clojure一种Lisp
 - 编译成为JVM bytecode
 - 提供的编程接口主要为Java
 - 函数说明等基本是Java
 - 通过thrift支持多种语言

Storm 系统结构

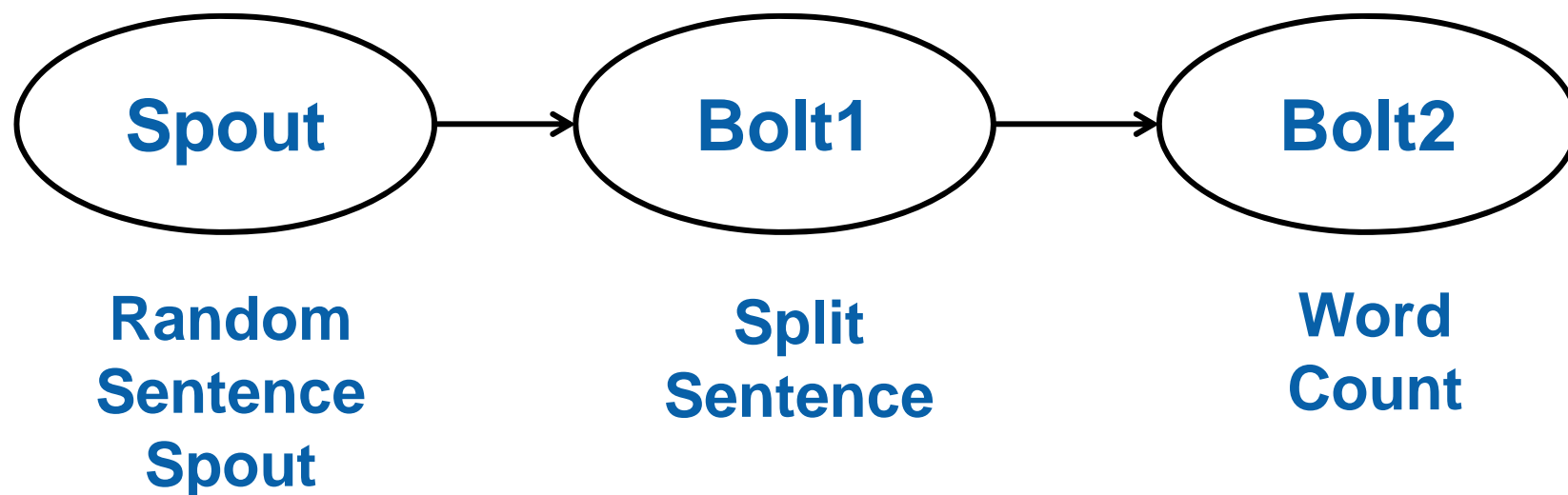


- 前端master: Nimbus
- 后端worker: Supervisor
- 通过ZooKeeper通信

Storm程序概念

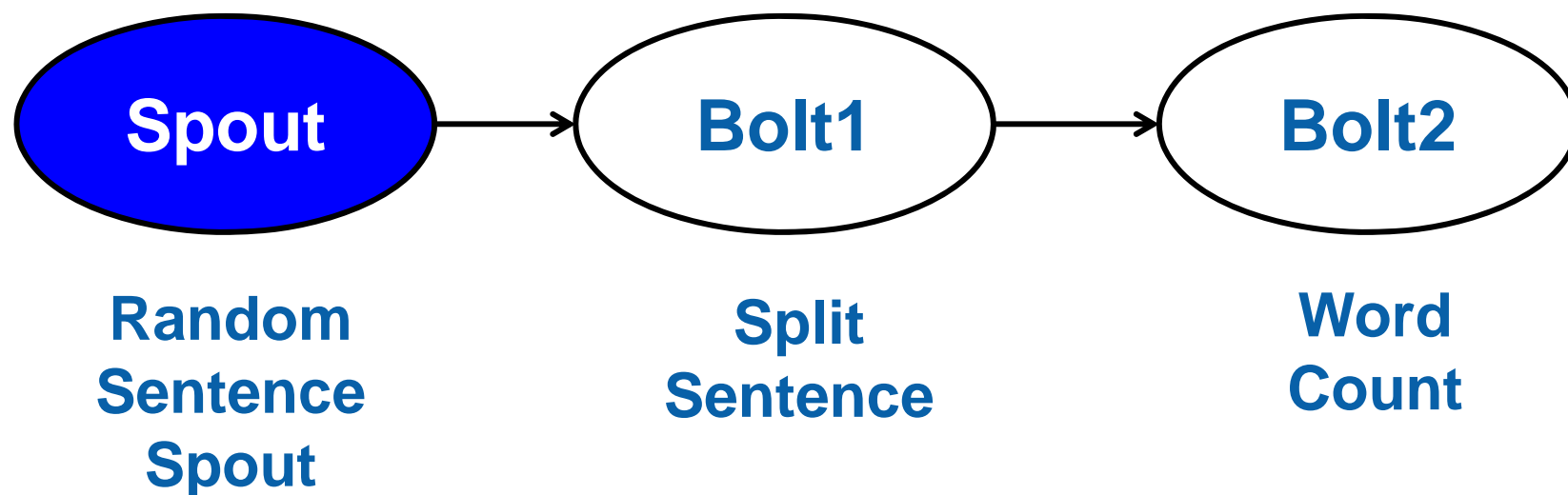
- 计算形成一个有向无环图DAG
 - DAG用一个Topology数据结构表示
 - 每个job有一个Topology
 - 对应于数据流处理运算关系的一张图
- Topology中的每个顶点代表一个运算
 - Spout: 产生数据流
 - 没有输入, 有输出
 - Bolt: 对数据流进行某种运算
 - 有输入, 有输出
- Topology中两个顶点之间的边代表数据流动的关系

举例:一个简单的Topology和程序



- Topology表示数据流上定义的一组运算

举例:一个简单的Topology和程序



- Random Sentence Spout:
每调用一次返回一个随机的句子

举例： RandomSentenceSpout

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }
    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{ "the cow jumped over the moon",
            "four score and seven years ago", "snow white and the seven dwarfs"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

实现一个
BaseRichSpout的子类

例子来源： Storm Example Code

举例： RandomSentenceSpout

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{ "the cow jumped over the moon",
            "four score and seven years ago", "snow white and the seven dwarfs"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

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

Open时初始化

举例： RandomSentenceSpout

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }
```

主要是实现nextTuple函数

```
@Override
public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[]{ "the cow jumped over the moon",
        "four score and seven years ago", "snow white and the seven dwarfs"};
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
}
```

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

举例：RandomSentenceSpout

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }
```

使用emit向Storm发出一个数据tuple

```
@Override
public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[]{ "the cow jumped over the moon",
        "four score and seven years ago", "snow white and the seven dwarfs" };
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
}
```

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

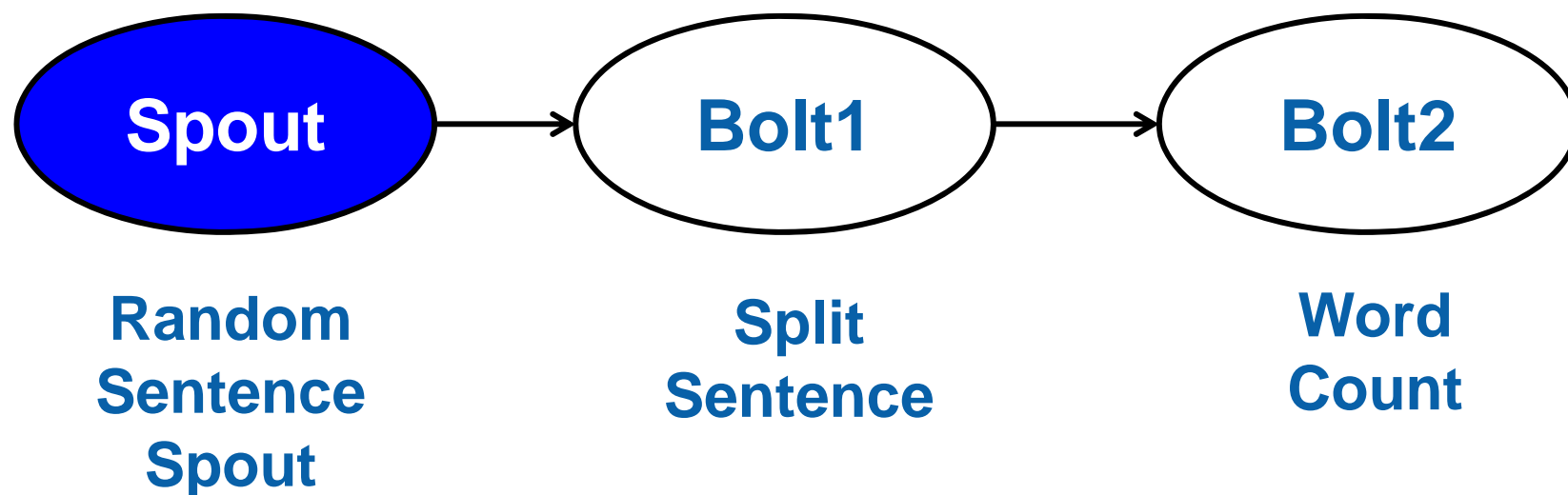
举例： RandomSentenceSpout

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context,
                    SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }
    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{ "the cow jumped over the moon",
            "four score and seven years ago", "snow white and the seven dwarfs"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

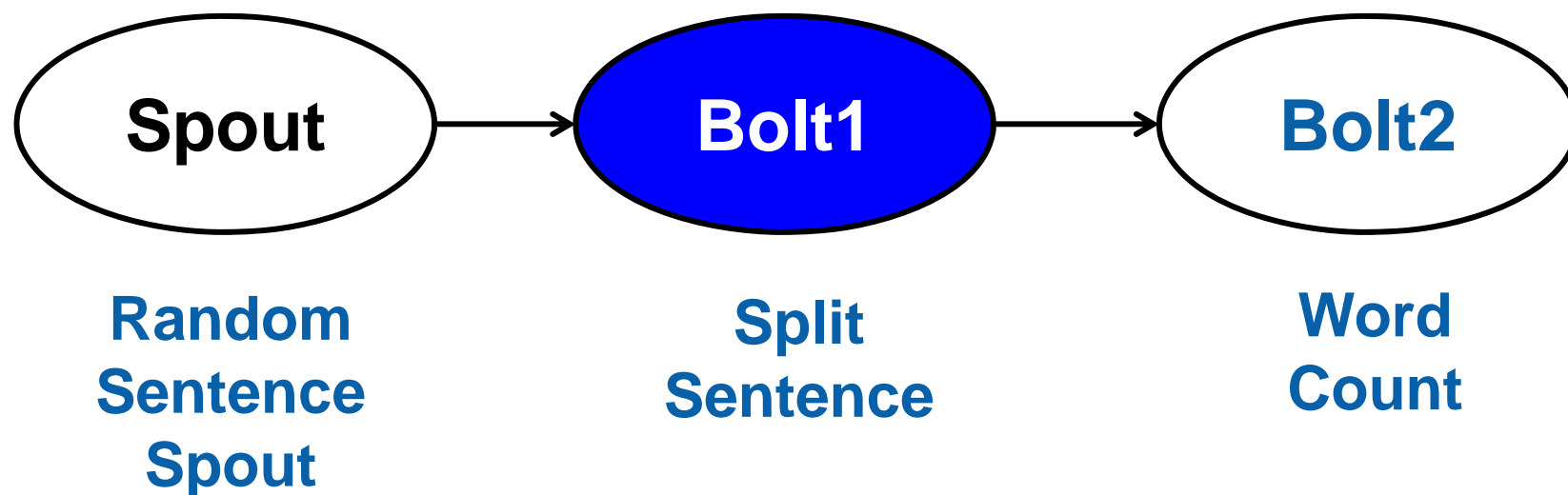
定义输出tuple各属性的名字

举例:一个简单的Topology



- Random Sentence Spout:
每调用一次返回一个随机的句子
- 实际使用时, Spout通常是获得外部数据, emit
 - 例如: 微博的推送

举例:一个简单的Topology



- Split Sentence: 把句子分成单词

举例：SplitSentence

```
public static class SplitSentence extends BaseBasicBolt {  
    @Override  
    public void execute(Tuple tuple, BasicOutputCollector collector) {  
        String sentence = tuple.getString(0);  
        StringTokenizer itr = new StringTokenizer(sentence);  
        while (itr.hasMoreTokens()) {  
            collector.emit(new Values(itr.nextToken()));  
        }  
    }  
    @Override  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("word"));  
    }  
}
```

实现一个
BaseBasicBolt的子类

举例：SplitSentence

主要实现execute函数，每一个输入tuple被调用一次，用emit发出输出的tuple

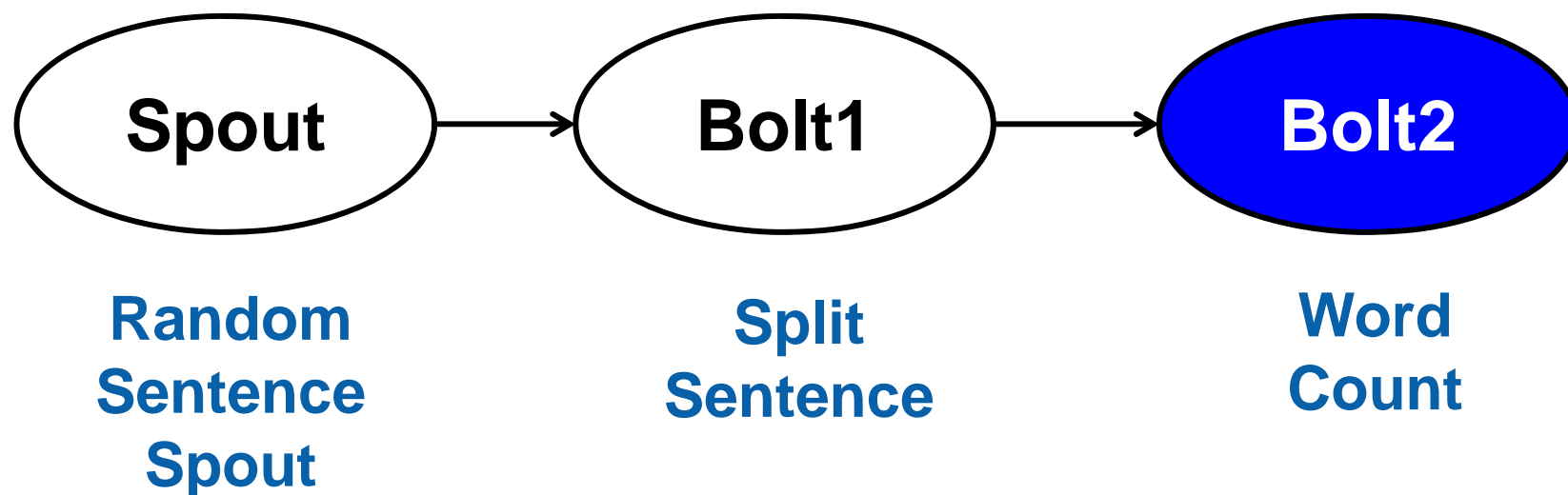
```
public static class SplitSentence extends
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String sentence= tuple.getString(0);
        StringTokenizer itr = new StringTokenizer(sentence);
        while (itr.hasMoreTokens()) {
            collector.emit(new Values(itr.nextToken()));
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

举例：SplitSentence

```
public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String sentence= tuple.getString(0);
        StringTokenizer itr = new StringTokenizer(sentence);
        while (itr.hasMoreTokens()) {
            collector.emit(new Values(itr.nextToken()));
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

定义输出tuple的属性的名字

举例:一个简单的Topology



- Word Count: 单词计数

举例： WordCount

BaseBasicBolt的子类

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

举例： WordCount

每个输入Tuple是一个
word，在HashMap中
计数，输出当前计数

```
public static class WordCount extends BaseB  
    Map<String, Integer> counts = new HashMap<
```

```
@Override  
public void execute(Tuple tuple, BasicOutputCollector collector) {  
    String word = tuple.getString(0);  
    Integer count = counts.get(word);  
    if (count == null) count = 0;  
    count++;  
    counts.put(word, count);  
    collector.emit(new Values(word, count));  
}
```

```
@Override  
public void declareOutputFields(OutputFieldsDeclarer declarer) {  
    declarer.declare(new Fields("word", "count"));  
}  
}
```

举例： WordCount

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

定义输出tuple的属性的名字

举例：主程序

```
public static void main(String[] args) throws Exception {  
  
    TopologyBuilder builder = new TopologyBuilder();  
  
    builder.setSpout("spout", new RandomSentenceSpout(), 5);  
    builder.setBolt("split", new SplitSentence(), 8)  
        .shuffleGrouping("spout");  
    builder.setBolt("count", new WordCount(), 12)  
        .fieldsGrouping("split", new Fields("word"));  
  
    Config conf = new Config();  
    conf.setMaxTaskParallelism(3);  
  
    LocalCluster cluster = new LocalCluster();  
    cluster.submitTopology("word-count", conf, builder.createTopology());  
    Thread.sleep(10000);  
    cluster.shutdown();  
}
```

建立
Topology

运行

举例：主程序

```
public static void main(String[] args) throws Exception {  
    TopologyBuilder builder = new TopologyBuilder;  
    builder.setSpout("spout", new RandomSentenceSpout(), 5);  
    builder.setBolt("split", new SplitSentence(), 8)  
        .shuffleGrouping("spout");  
    builder.setBolt("count", new WordCount(), 12)  
        .fieldsGrouping("split", new Fields("word"));  
  
    Config conf = new Config();  
    conf.setMaxTaskParallelism(3);  
  
    LocalCluster cluster = new LocalCluster();  
    cluster.submitTopology("word-count", conf, builder.createTopology());  
    Thread.sleep(10000);  
    cluster.shutdown();  
}
```

创建Spout, 并行度为5

举例：主程序

```
public static void main(String[] args) throws Exception {  
    TopologyBuilder builder = new TopologyBuilder;  
    builder.setSpout("spout", new RandomSentenceSpout(), 1);  
    builder.setBolt("split", new SplitSentence(), 8)  
        .shuffleGrouping("spout");  
    builder.setBolt("count", new WordCount(), 12)  
        .fieldsGrouping("split", new Fields("word"));  
  
    Config conf = new Config();  
    conf.setMaxTaskParallelism(3);  
  
    LocalCluster cluster = new LocalCluster();  
    cluster.submitTopology("word-count", conf, builder.createTopology());  
    Thread.sleep(10000);  
    cluster.shutdown();  
}
```

创建Bolt1, 并行度为8,
连接spout输出

举例：主程序

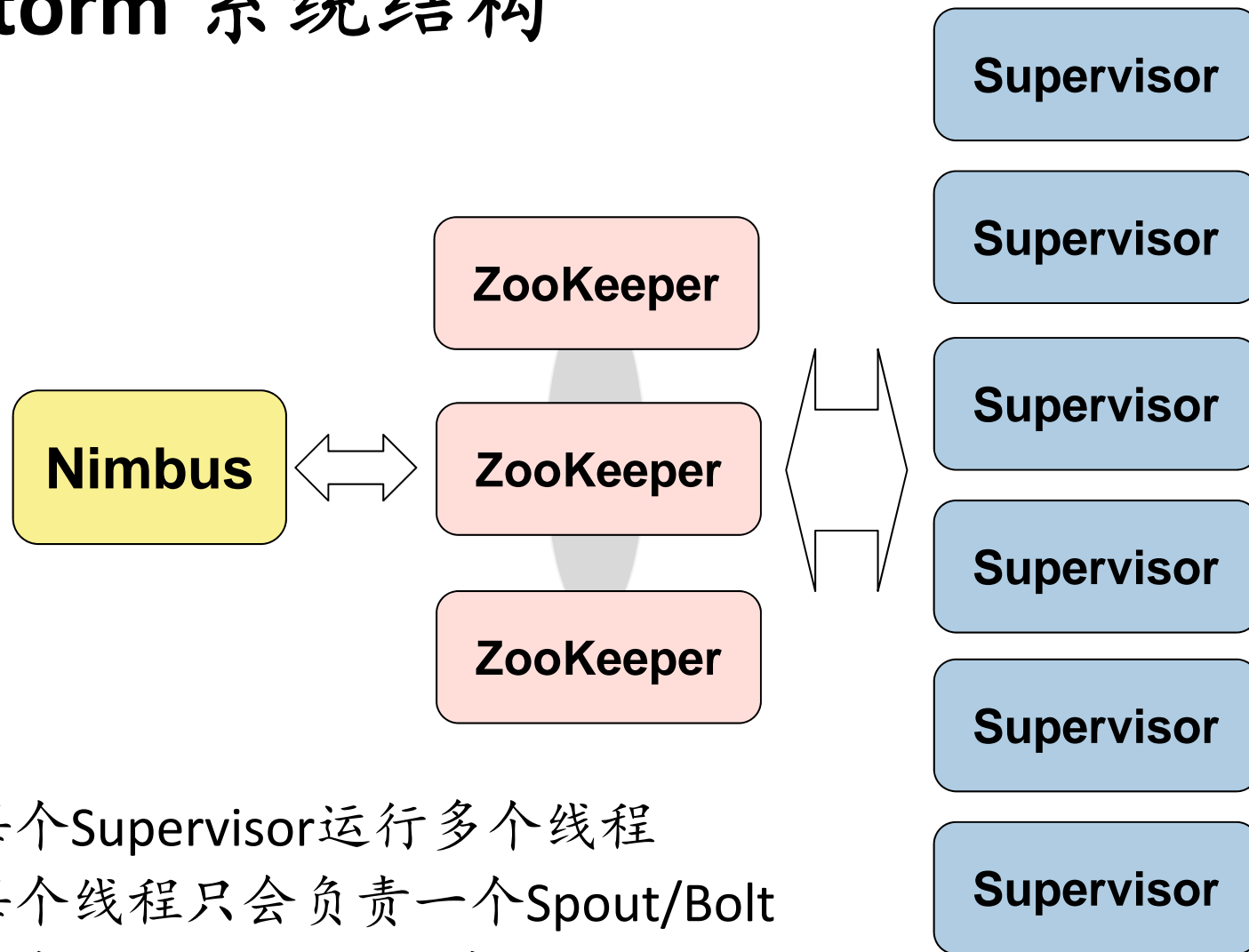
```
public static void main(String[] args) throws Exception {  
  
    TopologyBuilder builder = new TopologyBuilder();  
  
    builder.setSpout("spout", new RandomSentenceSpout(), 1);  
    builder.setBolt("split", new SplitSentence(), 1)  
        .shuffleGrouping("spout");  
    builder.setBolt("count", new WordCount(), 12)  
        .fieldsGrouping("split", new Fields("word"));  
  
    Config conf = new Config();  
    conf.setMaxTaskParallelism(3);  
  
    LocalCluster cluster = new LocalCluster();  
    cluster.submitTopology("word-count", conf, builder.createTopology());  
    Thread.sleep(10000);  
    cluster.shutdown();  
}
```

创建Bolt2, 并行度为12,
连接Bolt1输出

Stream Grouping

- 上游节点的输出如何分发到下游顶点?
 - 下游顶点有多个运行的实例
 - 上游产生的一条tuple, 应该发给哪个下游实例?
- Shuffle grouping: 随机
 - 随机发送给下游实例
- Fields grouping: group-by shuffle
 - 根据tuple中指定域的取值
 - 相同取值的tuple发给固定的下游实例
- 其它种类

Storm 系统结构

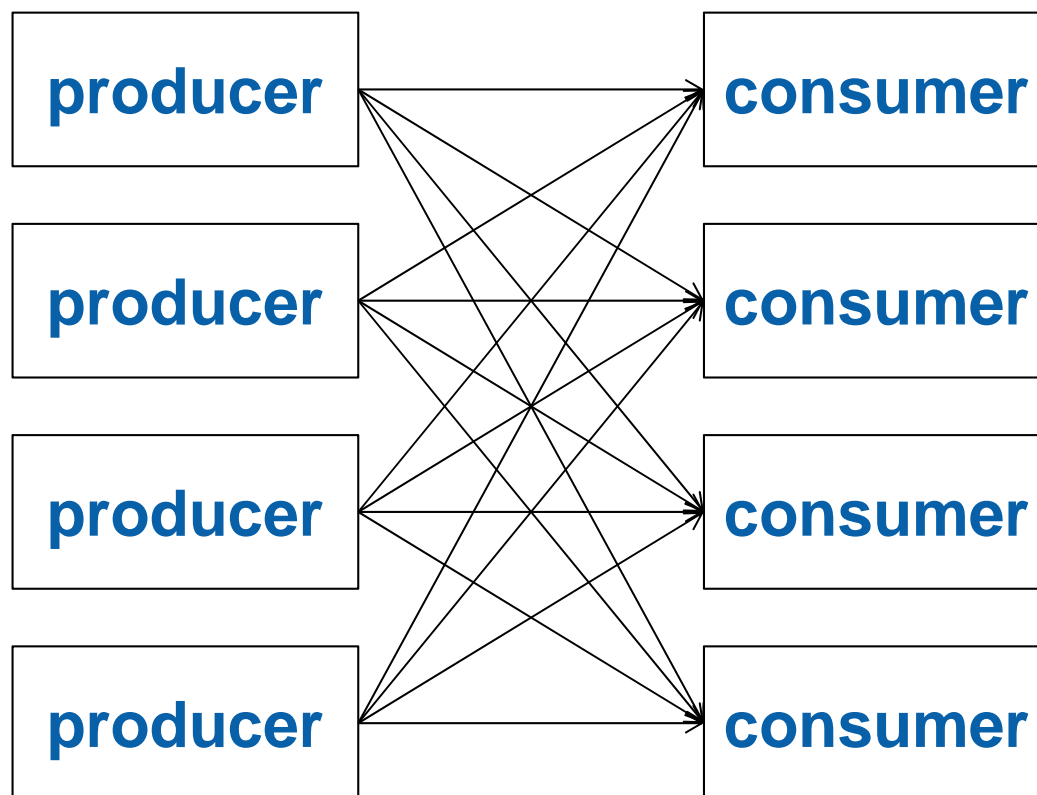


- 每个Supervisor运行多个线程
- 每个线程只会负责一个Spout/Bolt
- 一个Spout/Bolt可以对应多个Supervisor和多个线程

Outline

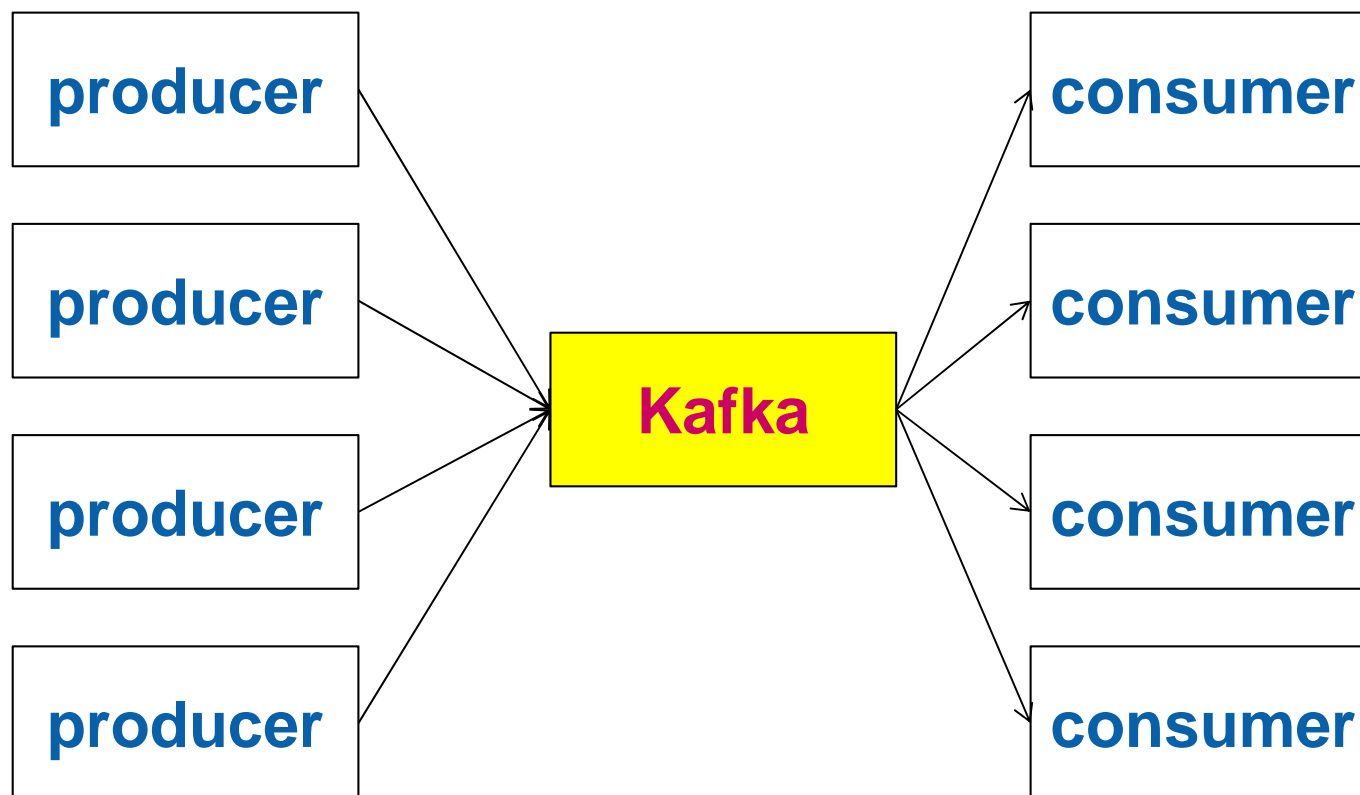
- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统

Message的发送和接收的问题



- N-N: 复杂，难以管理、维护，容易出错

Kafka的作用



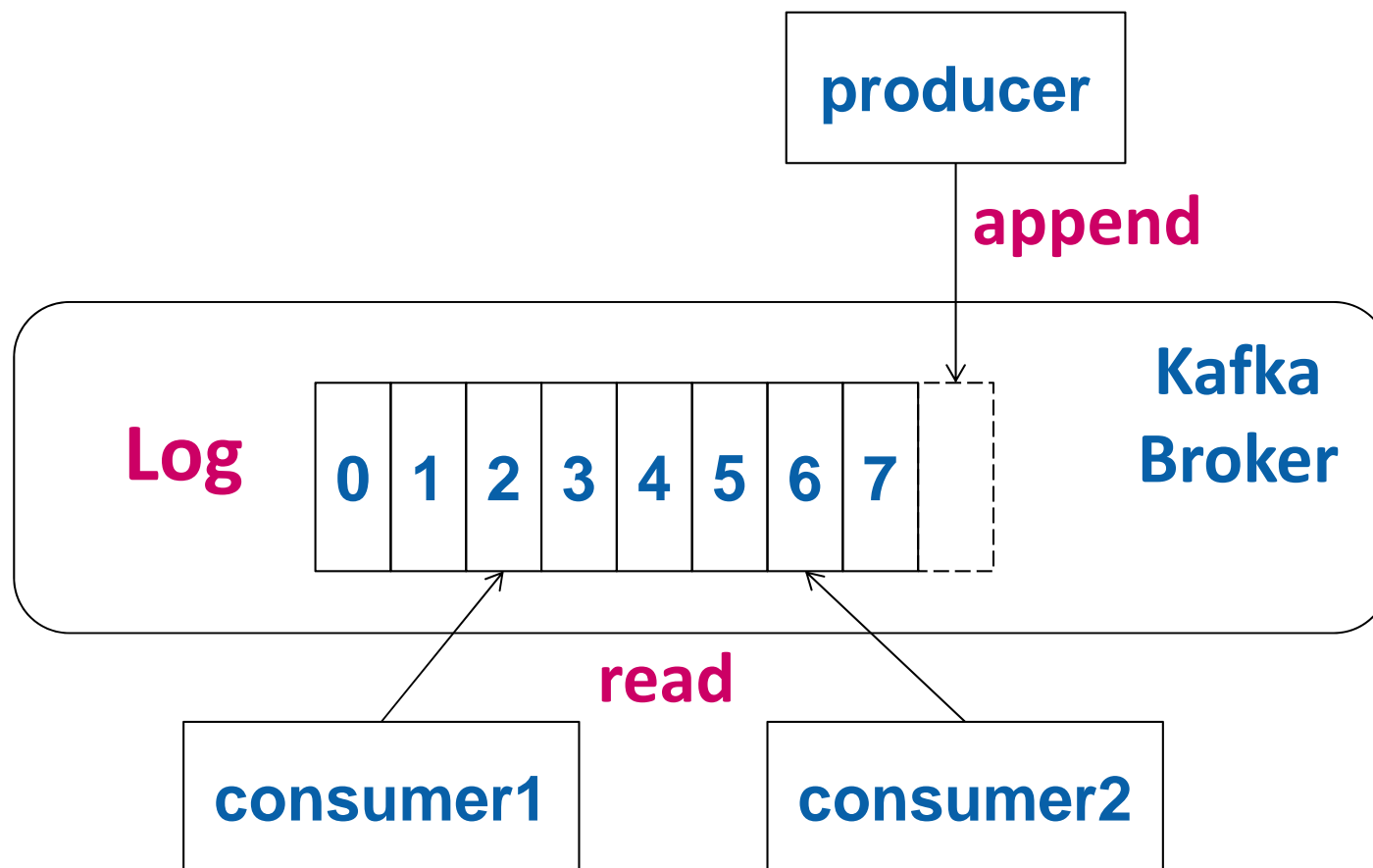
- **N-1-N**: 降低复杂度，隔离consumer和producer之间处理速度的差异

Kafka



- LinkedIn开发，2010年成为Apache开源项目
- Java/Scala实现
- 分布式的消息日志系统（Log）
 - 或者成为消息队列：message queue
 - 涵义类似：message是append到Log/queue中的
- 可以用于数据流系统的前端
 - 外部消息收集写入Kafka
 - 数据流系统从Kafka中读取数据进行处理

Kafka基本模型



消息保留的时间：给定时间/磁盘空间设置，超出就删除

Kafka基本概念

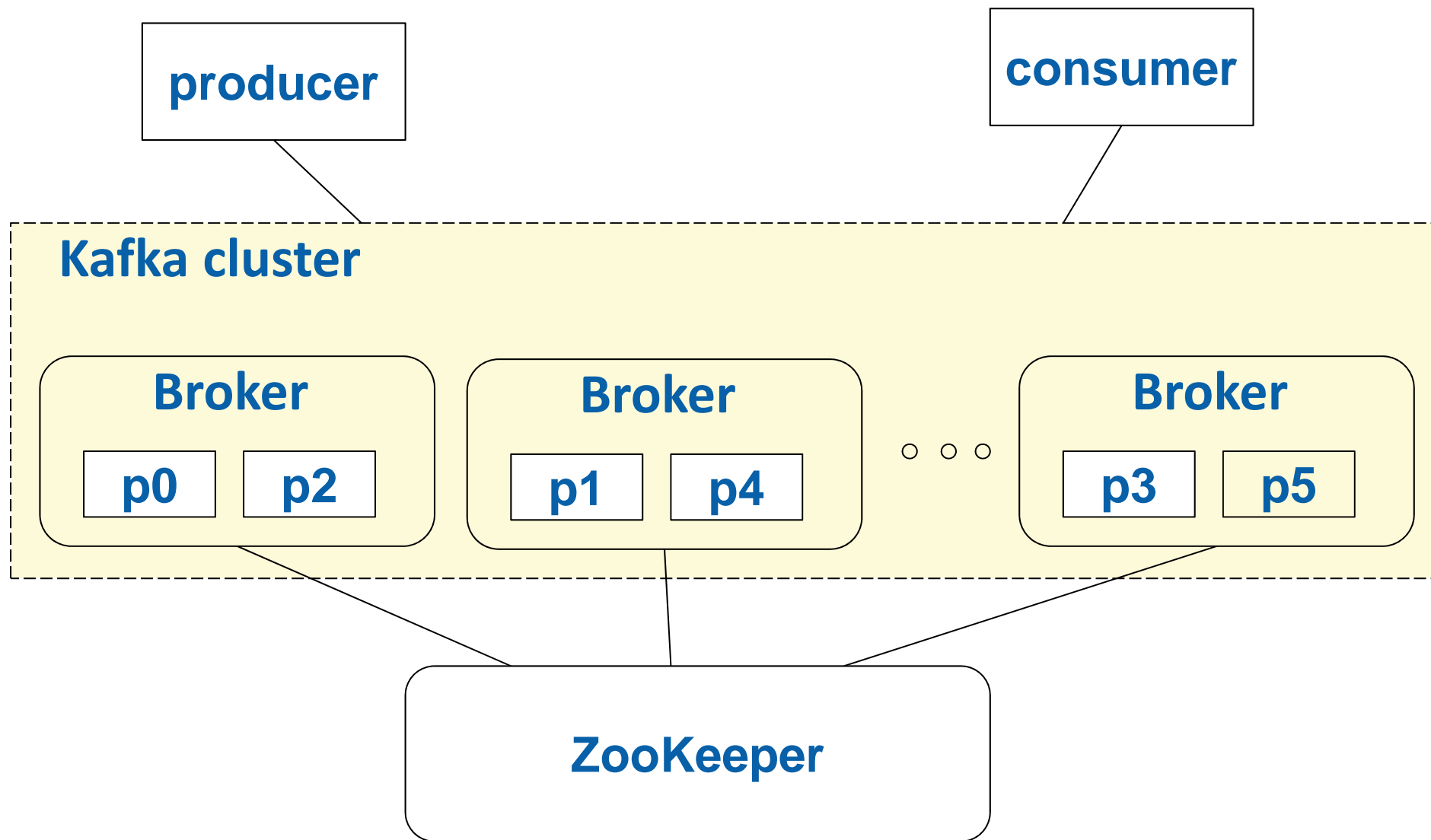
- 角色

- Producer
- Consumer: 主动去读 (Pull模型)
- Broker: Kafka服务器存储Producer发来的消息, 为Consumer提供消息读取服务

- 消息的组织

- Topic: 每个Topic对应一个Log/Queue
- Partition
 - 每个topic可以划分为多个partition
 - 每个partition内部消息有序
 - Partition之间消息无序
- Producer发布消息时需要指明topic和partition

Kafka 系统结构



容错支持

- Kafka保证

- ☐ At least once delivery
- ☐ Message order in a topic partition

- 备份

- ☐ Topic partition是备份的基本单位
- ☐ Primary-backup主从备份

Outline

- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统

内存处理

- 随着内存容量的指数级增加

- 越来越大的数据集可以完全存放在内存中
- 或者完全存放在一个机群的总和的内存中
- 例如，每台服务器64GB，一组刀片16台，就是1TB
- 1TB对于很多重要的热点的数据可能已经足够了

- 内存处理的优点

- 去除了硬盘读写的开销
- 于是提高了处理速度



关系型内存数据库

- 最早的提法出现在1980，1990初
- 第一代MMDB出现于1990初
 - 没有高速缓存的概念
 - 例如：TimesTen
- 第二代MMDB出现于1990末，2000初
 - 对于新的硬件进行优化
 - 主要是学术领域提出的
 - 例如：MonetDB
 - 这一时期，产业界也开始重视MMDB，但主要是用来作前端的关系型cache
- 近年来，主流数据库公司纷纷投入研发MMDB
 - IBM Blink, Microsoft Hekaton & Apollo, SAP HANA, IBM BLU, 等

主要挑战

- 内存墙问题

- 内存访问需要100~1000 cycles
- 思路1: 减少cache miss
 - 调整数据结构或算法
- 思路2: 降低cache miss对性能的影响
 - Software prefetch 预取指令

- 多核数据共享开销

- 私有数据结构
- 高效的并发控制方法

- 新的硬件特性

- NVM, GPU, FPGA, NUMA等

MonetDB

- MonetDB

- ❑ 荷兰CWI研究所研制
- ❑ 1999年数据库领域第一篇关于cache performance的文章
- ❑ 成熟的内存数据仓库,支持几乎所有的SQL
- ❑ 内存列式存储,数据在类似数组的结构中

SQL

SQL被翻译为中间语言
MAL的运算

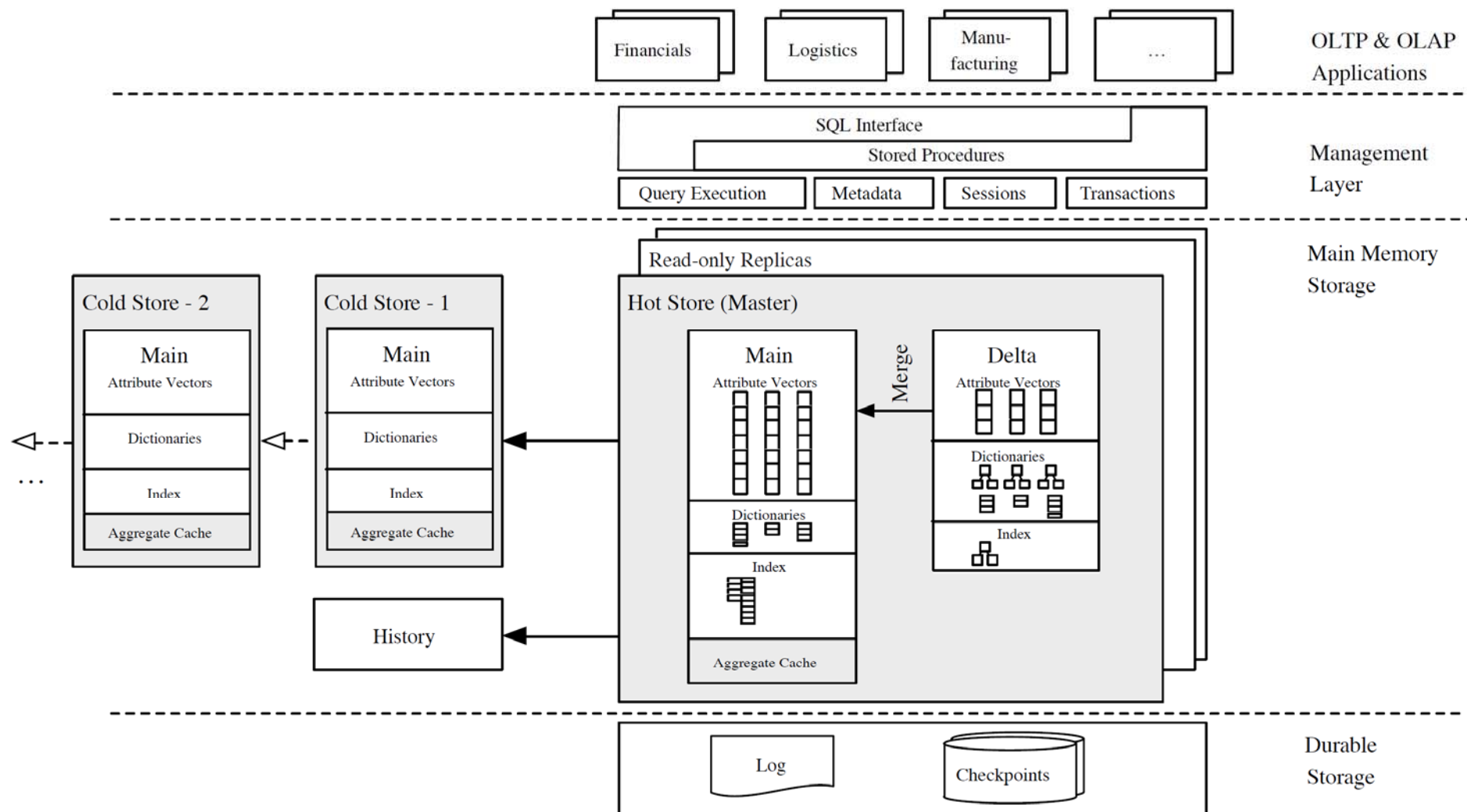
MAL

每个MAL运算,实现对整个BAT数据的运算

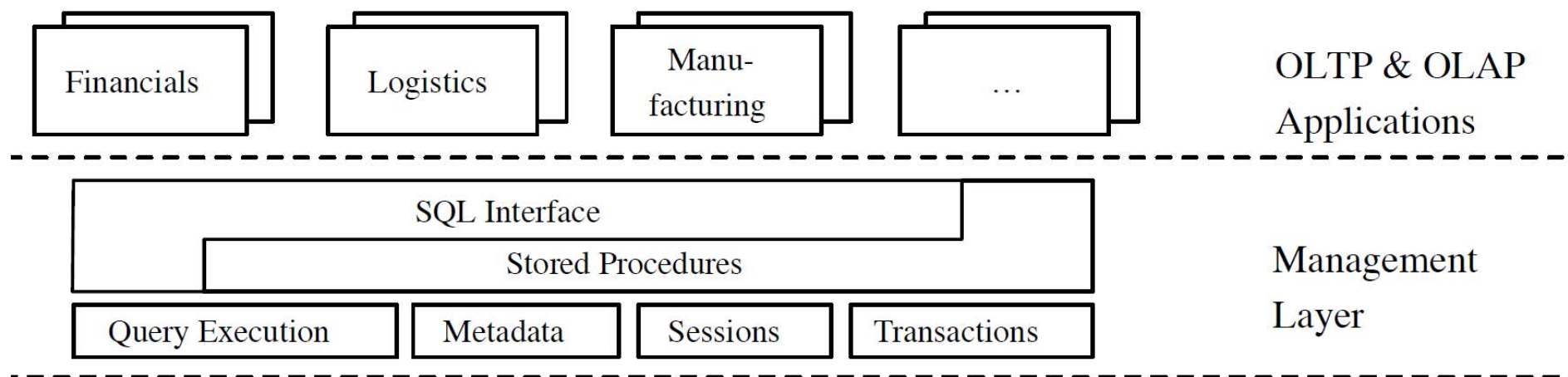
BAT

BAT (Binary Association Table)
相当于一个数组,存储一列数据,数据mmap到内存

SAP HANA

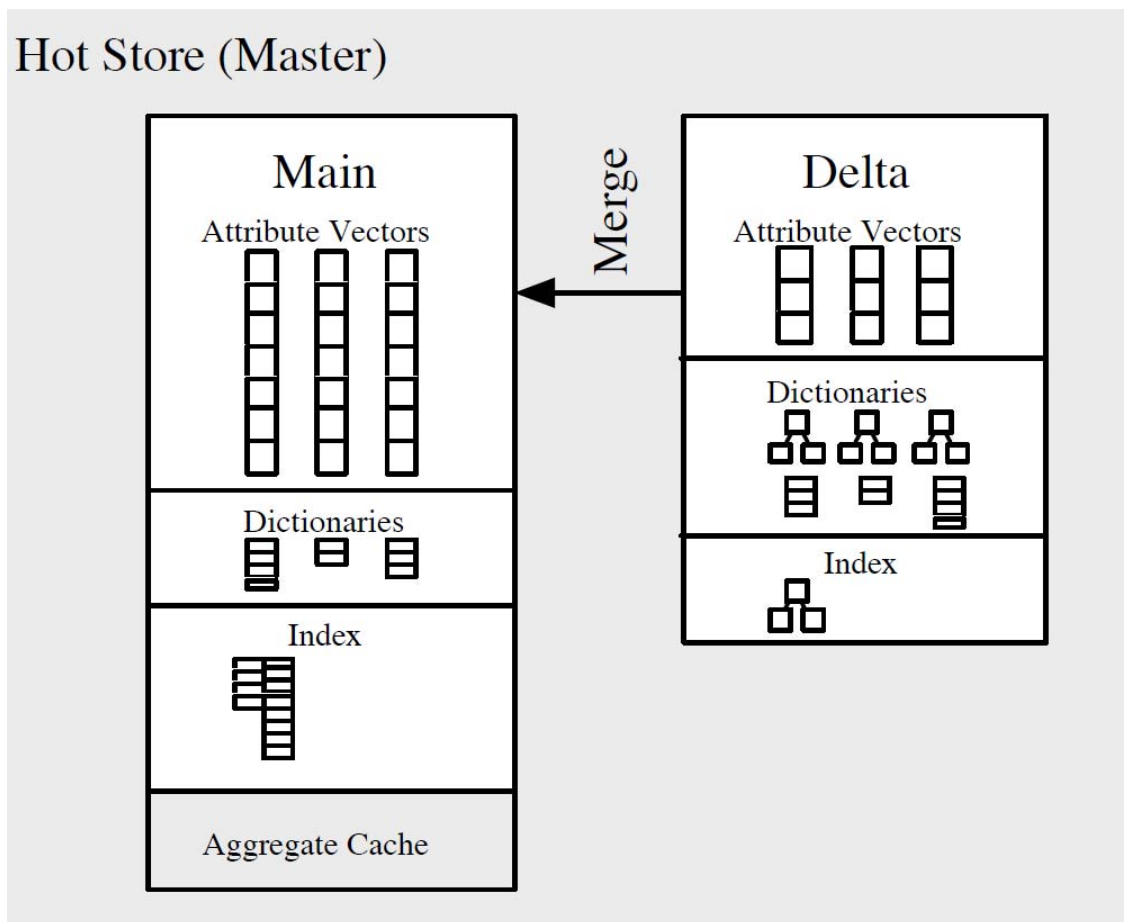


SAP HANA



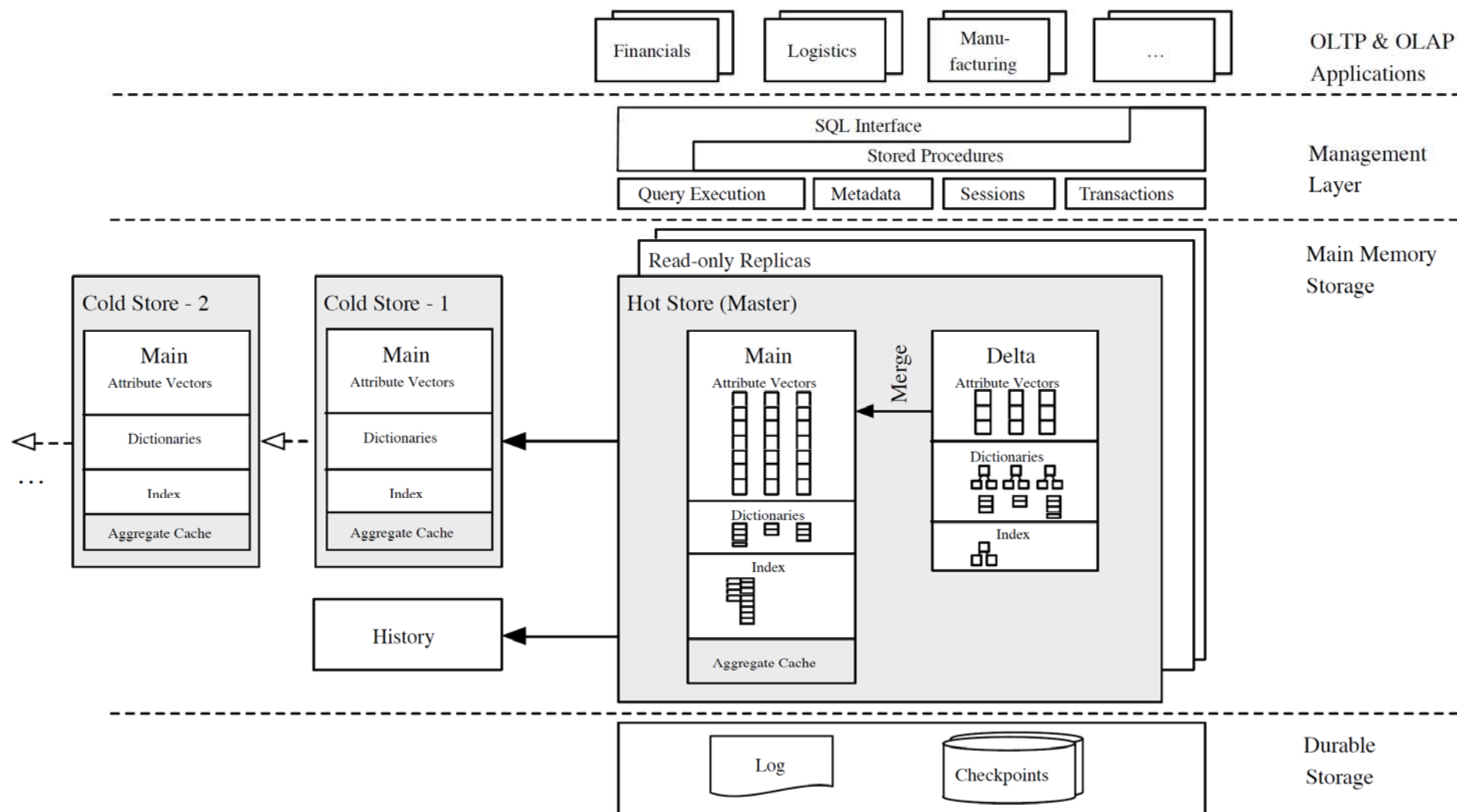
- 前端与普通数据库很相似

SAP HANA



- Main data
- Delta data
 - ❑ 新的数据
 - ❑ Updates
- Dictionary 压缩
- 轻量index

SAP HANA



其它主流数据库产品

- Microsoft

- Hekaton: OLTP
- Apollo: OLAP

- IBM

- Blink: IBM的数据仓库加速系统
- BLU: 列式数据库engine, 有硬盘存储, 使用了许多内存处理技术

- Oracle

- In-memory data analytics caching

Outline

- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统

内存key-value系统

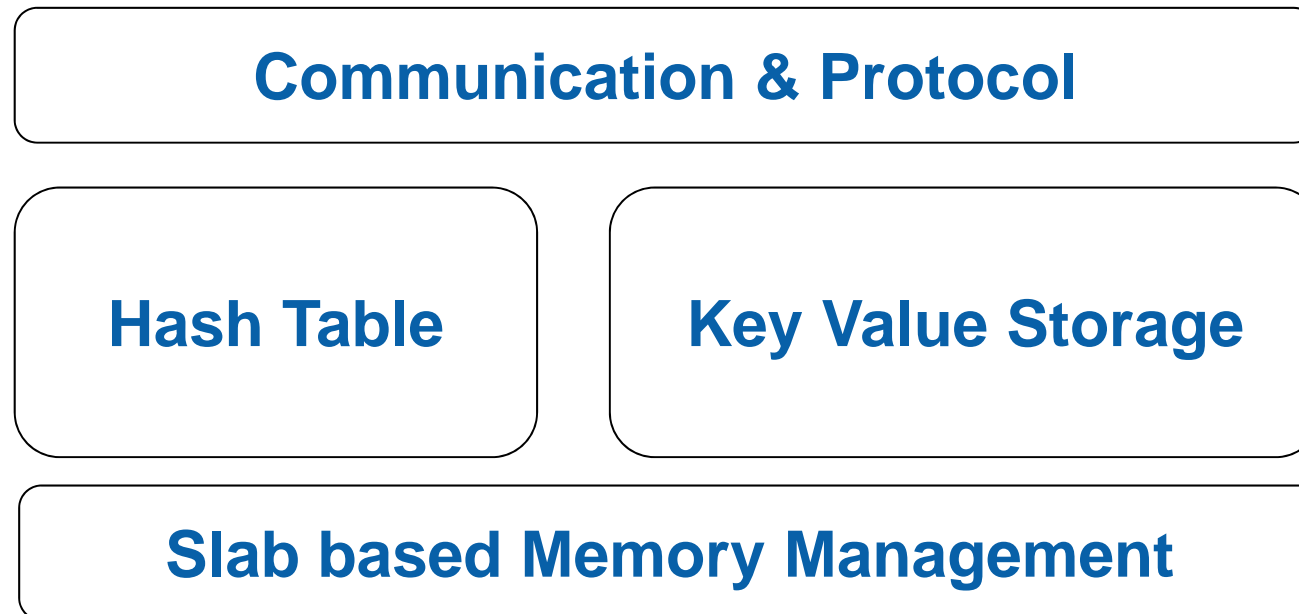
- Memcached

- ❑ 用户：Facebook, twitter, flickr, youtube, ...
- ❑ 单机内存键值对系统
- ❑ 数据在内存中以hash table的形式存储
- ❑ 支持最基础的<key, value>数据模型
- ❑ 通常被用于前端的cache
- ❑ 可以使用多个memcached+sharding建立一个分布式系统

- Redis：与memcached相比

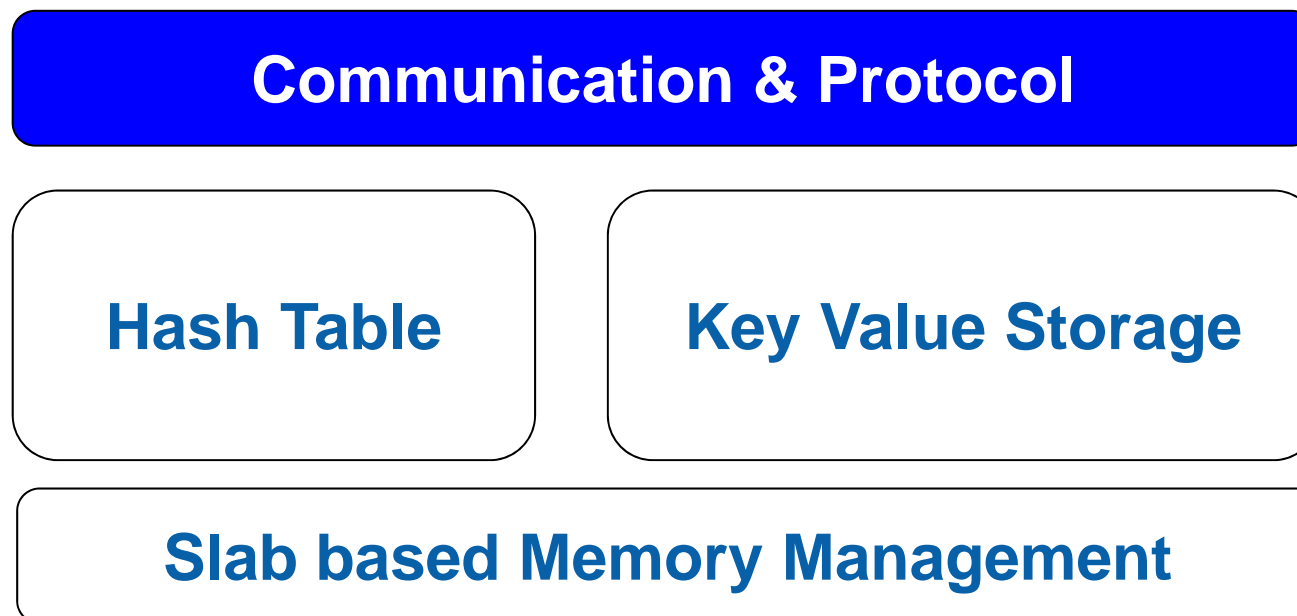
- ❑ 分布式内存键值对系统
- ❑ 提供更加丰富的类型，例如hashes, lists, sets 和sorted sets
- ❑ 支持副本和集群

Memcached内部结构



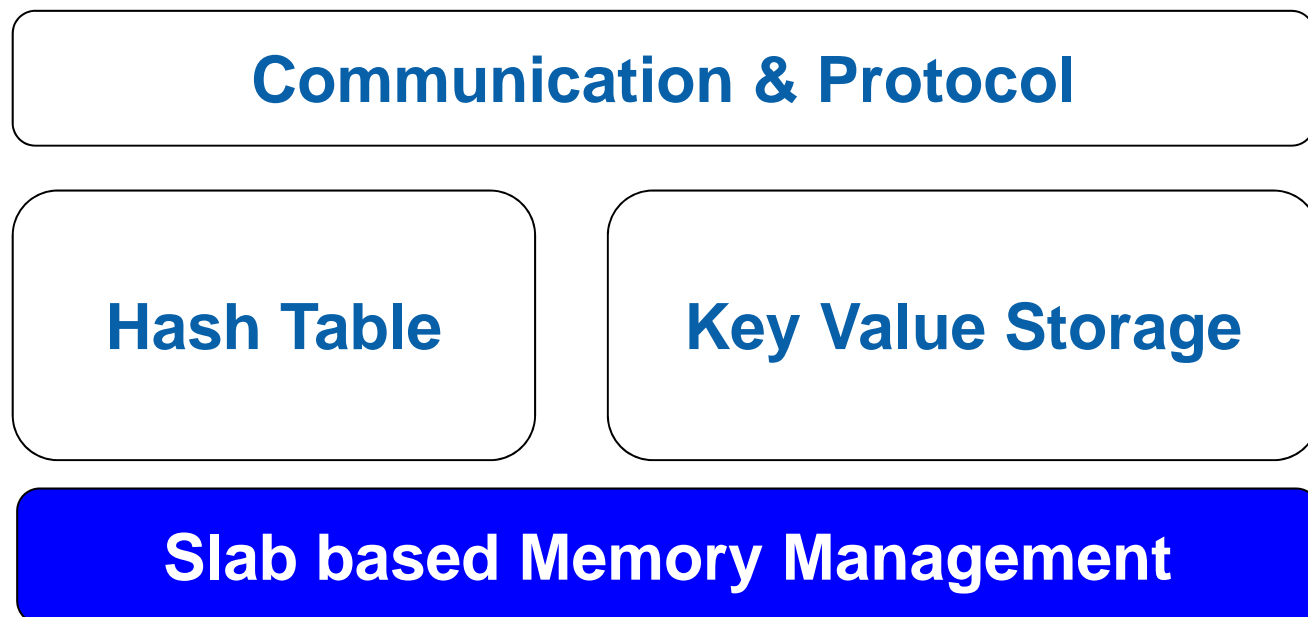
- 单机系统

Memcached内部结构



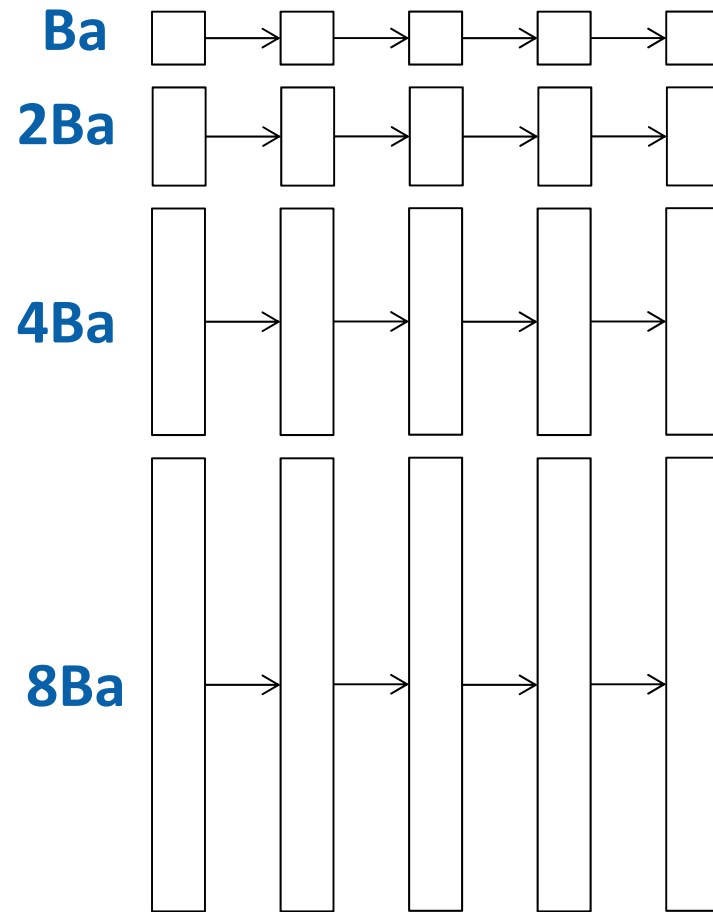
- 实现自定义的协议，支持GET/PUT等请求和响应
 - 文本方式的协议
 - 二进制协议

Memcached内部结构



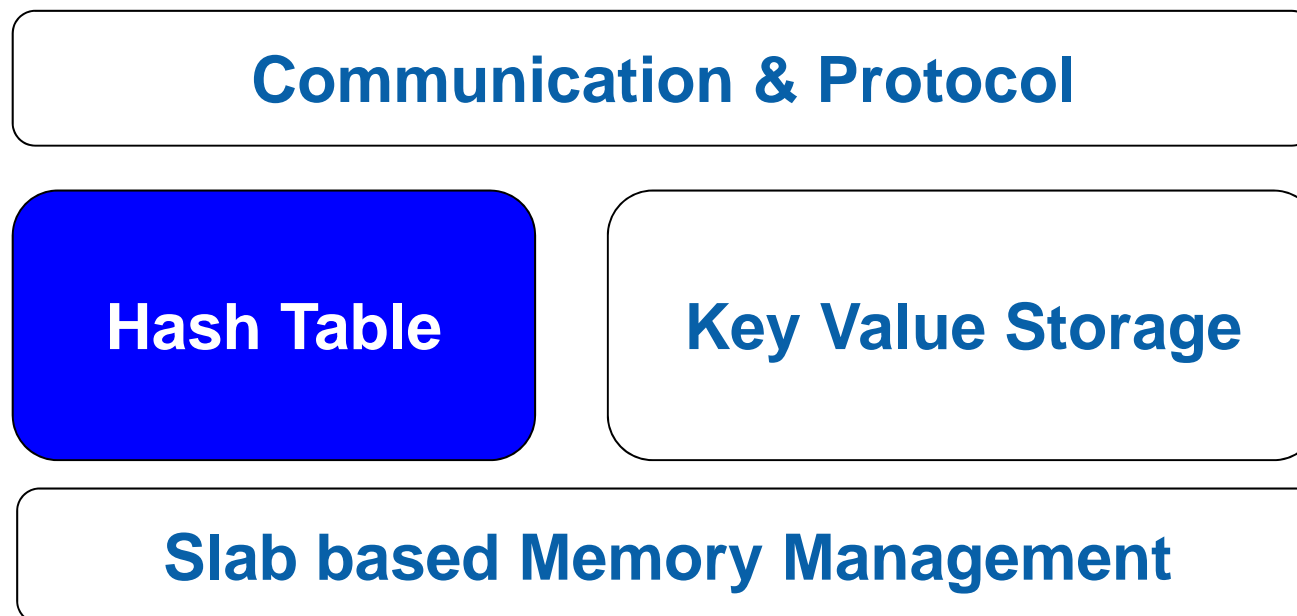
- 采用多个链表管理内存
 - 每个链表中的内存块大小相同
 - 第k个链表的内存块大小是 2^k Base字节
 - 只分配和释放整个内存块

Slab-Based Memory Management



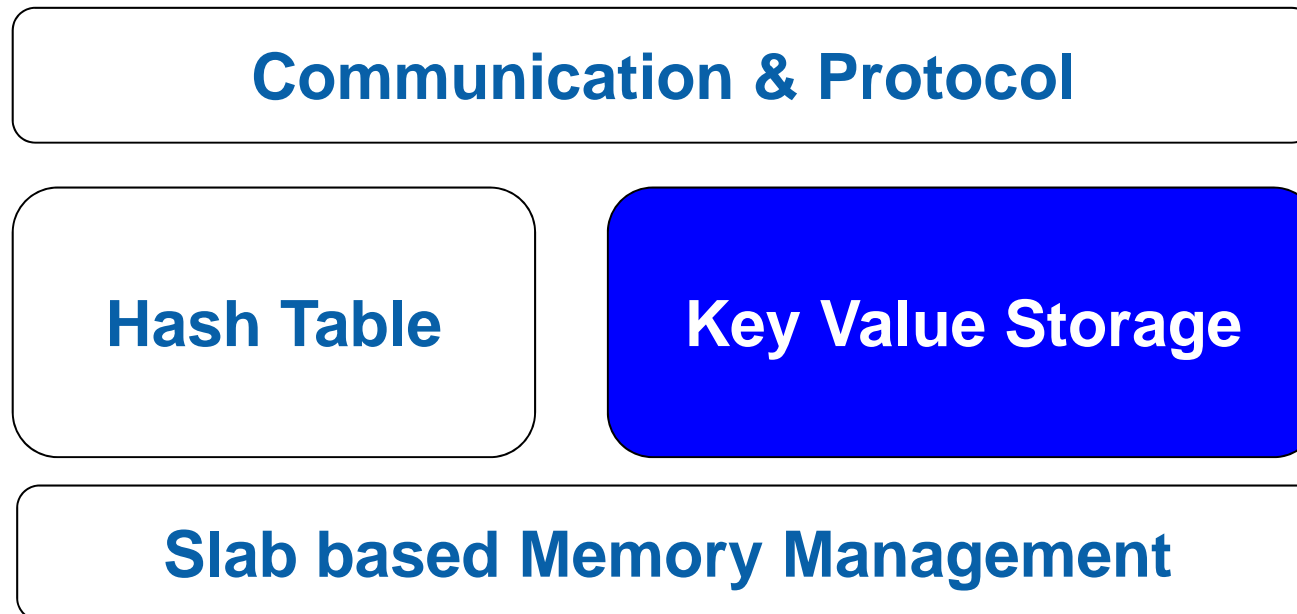
$Ba(se)$ 是最小的块的大小

Memcached内部结构



- Key-Value采用一个全局的Hash Table进行索引
- 多线程并发互斥访问

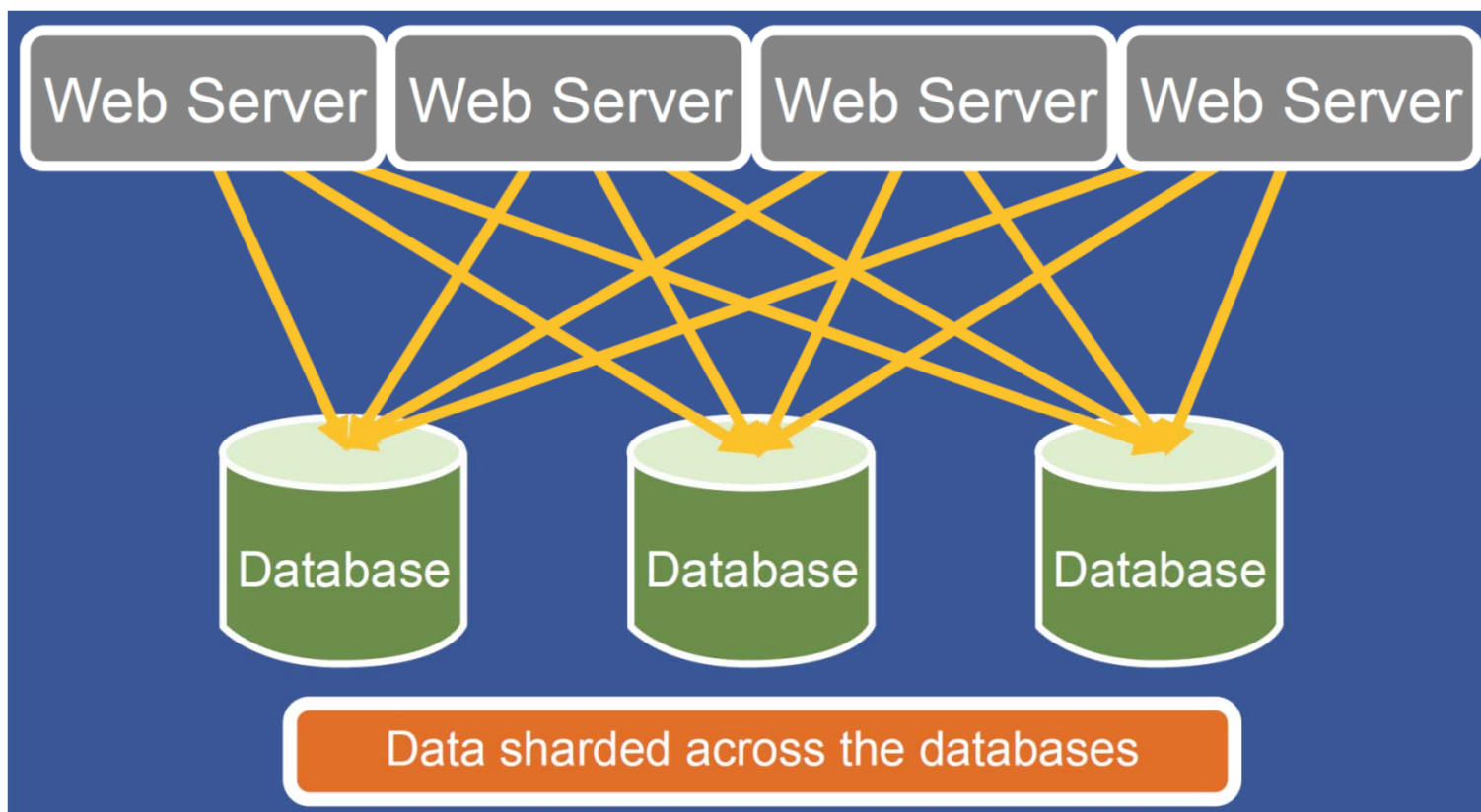
Memcached内部结构



- 每个Key-Value存储在一个内存块中
 - Hash link: chained hash table
 - LRU link: 相同大小的已分配内存块在一个LRU链表上
- 内存不够时，可以丢弃LRU项

Facebook

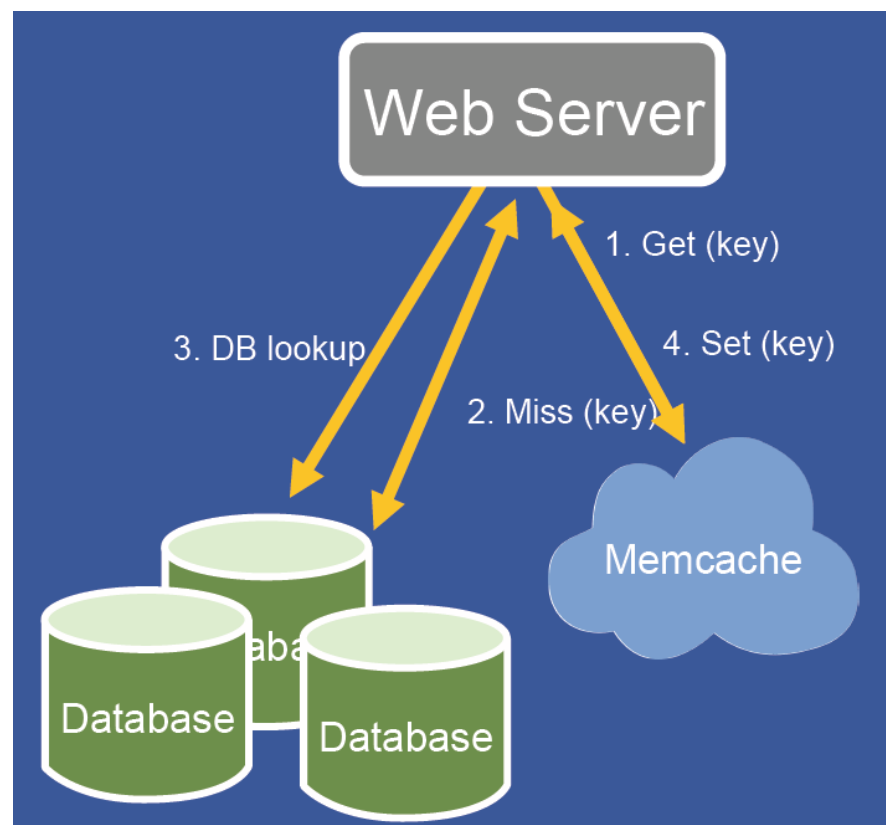
- “Scaling Memcache at Facebook”. NSDI’13
- 刚开始：直接用MySQL就足够了



图来源：NSDI’13 slides

数十台服务器，每秒上百万操作请求

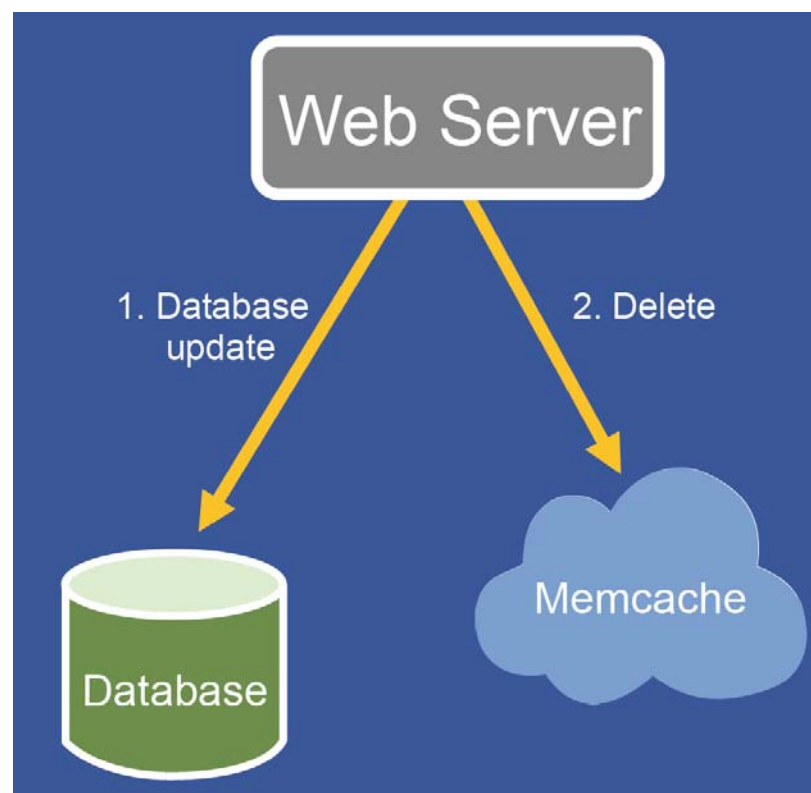
- 增加了几台 Memcached 服务器
- 读比写多2个数量级
- Look-aside cache
 - ❑ 多数hit在memcached
 - ❑ 如果miss，再读DB，然后放入memcached



图来源：NSDI'13 slides

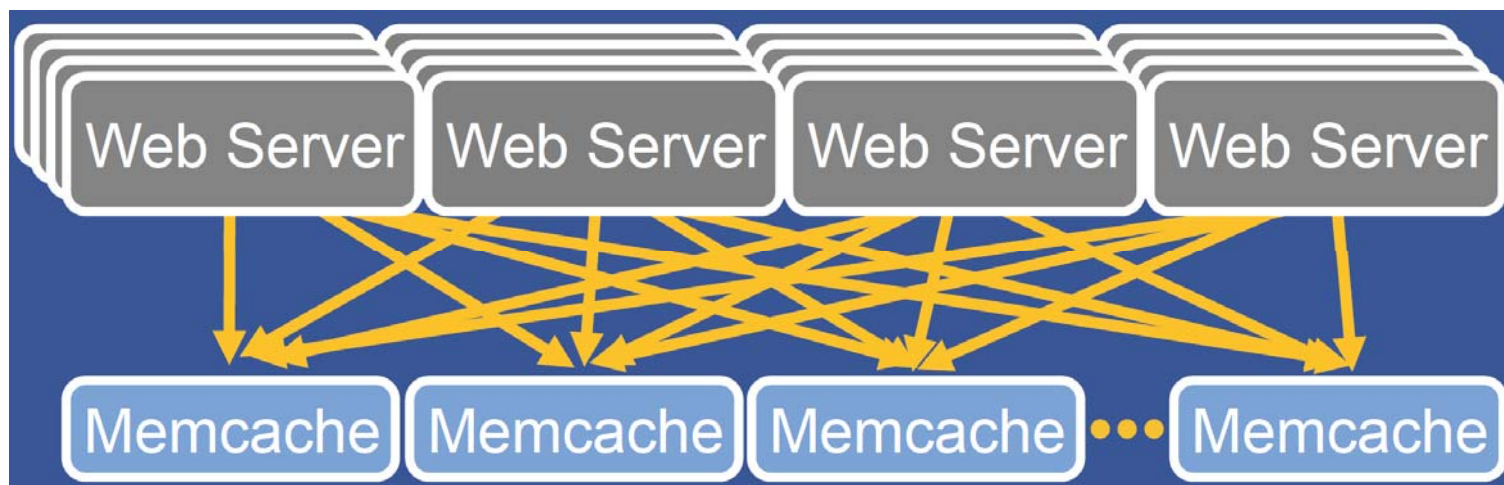
数十台服务器，每秒上百万操作请求

- 增加了几台 Memcached 服务器
 - 应用进行 sharding
- 读比写多2个数量级
- Look-aside cache
 - 在从DB中删除后
 - 也从memcached删除
- 问题
 - 分布式读写，DB与 memcached 不一致
 - 可能读到稍旧的数据



图来源：NSDI'13 slides

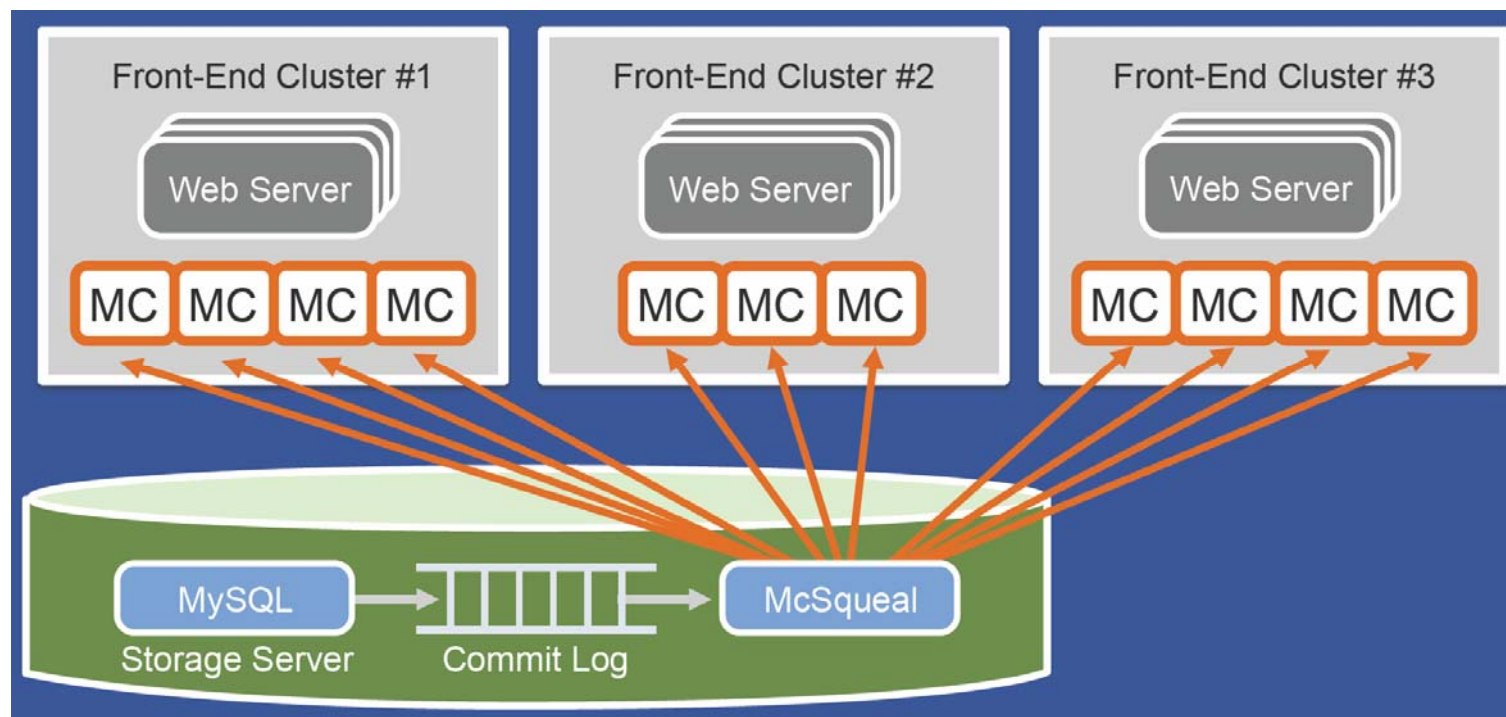
数百台服务器，每秒上千万操作请求



- 多个Memcached组成一个cluster
- 采用consistent hashing管理
- 每个web server为了满足一个网页，可能同时发出上百个请求

图来源：NSDI'13 slides

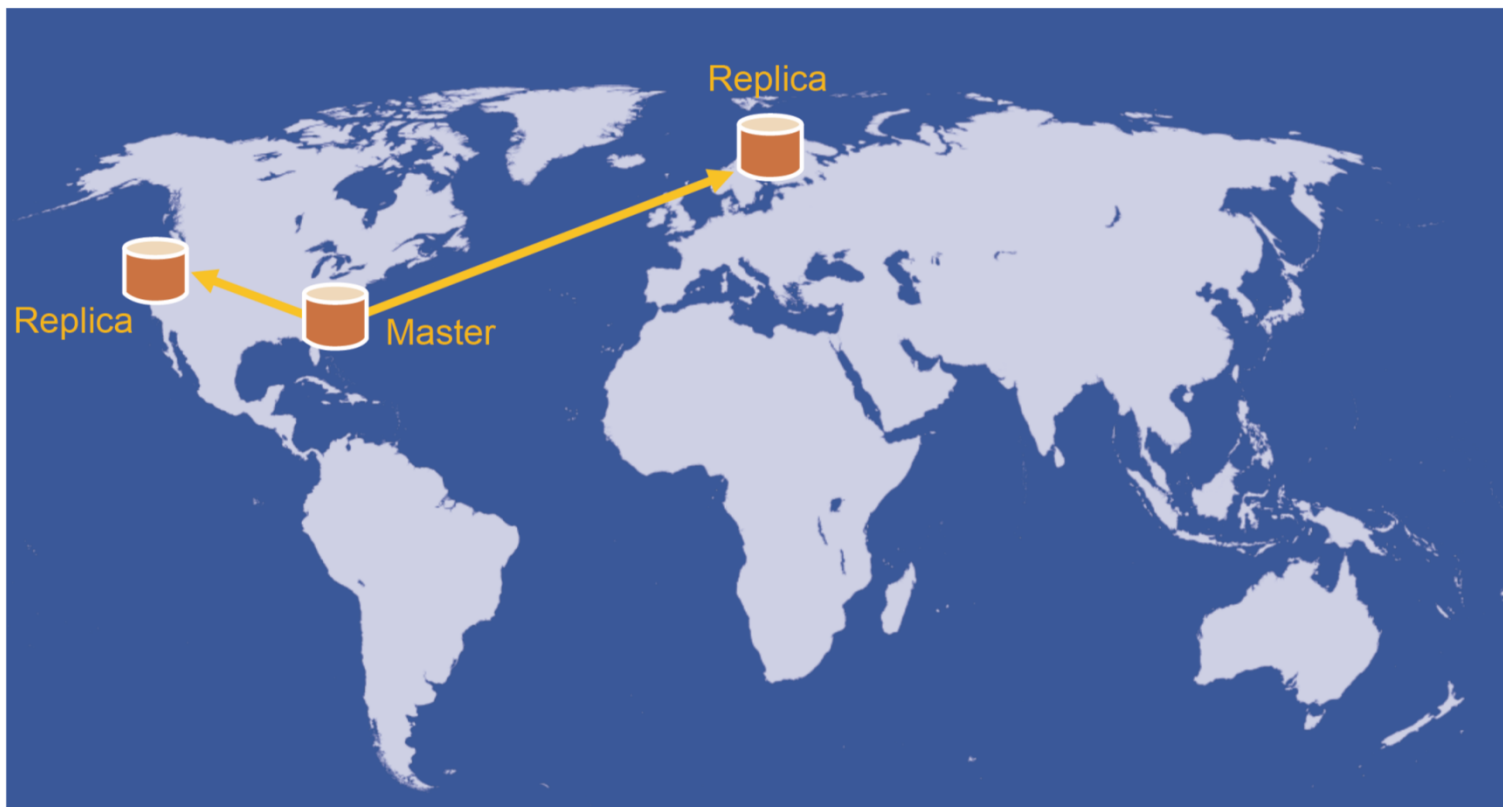
数千台服务器，每秒上亿次操作请求



- 多个Front-end cluster
 - 每个Front-end cluster由Web servers, memcached cluster组成
- 同一组DB服务器
 - 使用commit log来把数据更新发到memcached

图来源：NSDI'13 slides

数千台服务器，每秒数十亿次操作请求

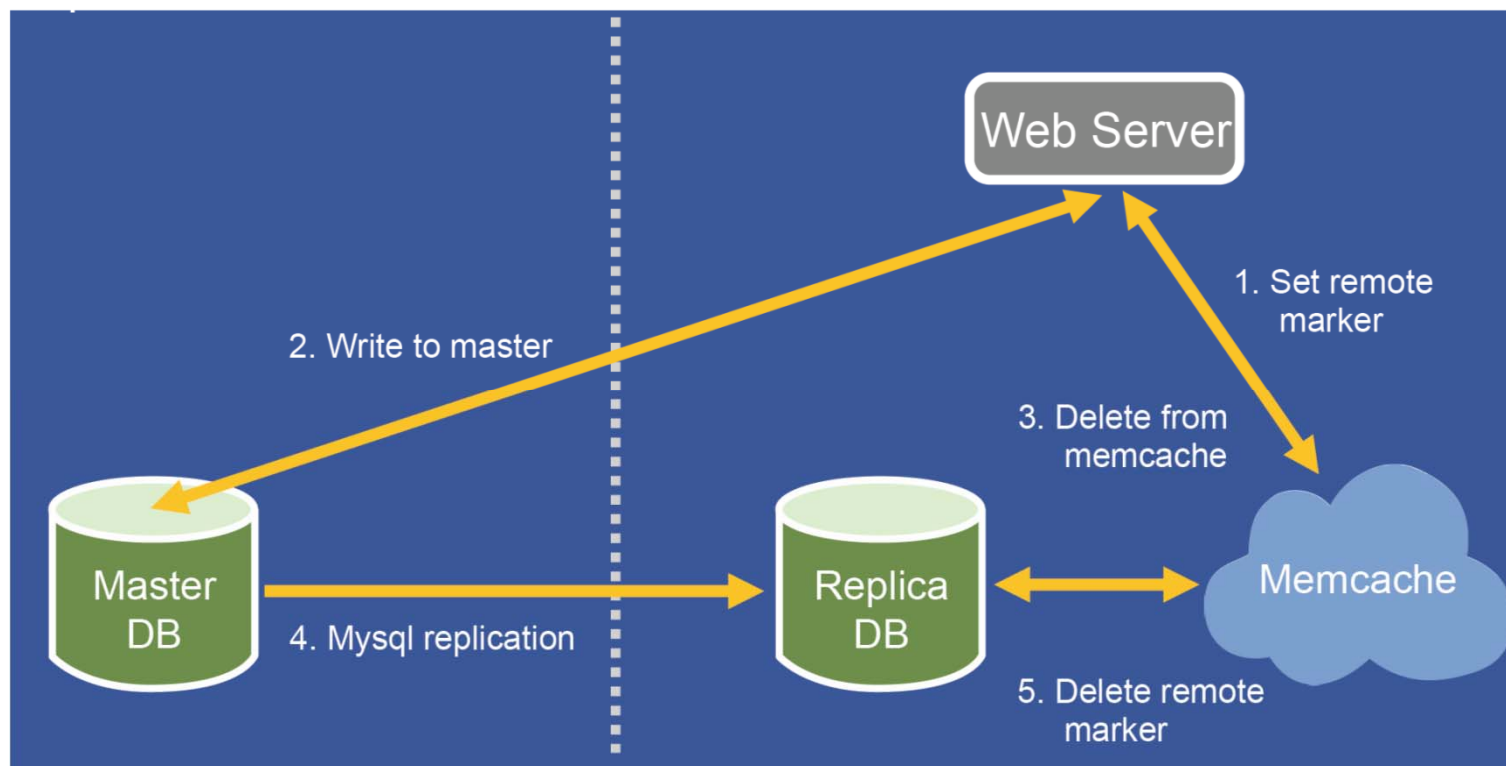


- 多个Replica

- 主DB与Replica DB不一致？

图来源：NSDI'13 slides

数千台服务器，每秒数十亿次操作请求



- 在DB write之前，先在memcached中写一个marker
- 这样读时发现marker，就一定要从主DB中取

图来源：NSDI'13 slides

Outline

- 图计算系统：GAS模型
- MapReduce + SQL系统
- 数据流处理
 - 数据流系统Storm
 - 消息日志系统Kafka
- 内存计算
 - 内存数据库
 - 内存键值系统