大数据管理系统与大规模数据分析 - June 10'rd, 2018

Computation Concepts

https://github.com/rh01

MapReduce/Hadoop

MapReduce 编程模型

MapReduce 的数据模型

- <key, value>
  - 数据由一条一条的记录组成
  - 记录之间是无序的
  - 每一条记录有一个 key，和一个 value
  - key: 可以不唯一
  - key 与 value 的具体类型和内部结构由程序员决定，系统基本上把它们看作黑匣
- MapReduce
  - Map(ik, iv) $\longrightarrow${<mk, mv>}
  - Reduce(mk, {mv})$\longrightarrow${<ok, ov>}
- Map 函数
  - 输入是一个 key-value 记录：<ik, iv>
    - 我们用‘i’代表 input
  - 输出是 0 ～多个 key-value 记录：<mk, mv>
    - 我们用’m’代表 intermediate
  - 注意：mk 与 ik 很可能完全不同
- Shuffle (由系统完成)
  - Shuffle = group by mk
  - 对于所有的 map 函数的输出，进行 group by
  - 将相同 mk 的所有 mv 都一起提供给 Reduce
- Reduce 函数
  - 输入是一个 mk 和与之对应的所有 mv
  - 输出是 0 多个 key-value 记录：<ok, ov>
    - 我们用’o’代表 output
  - 注意：ok 与 mk 可能不同

MapReduce vs. SQL

MapReduce	SQL Select
Map	Selection/projection
Shuffle	Group by
Reduce	Aggregation, Having
选择的功能更加丰富 程序实现的, 类似最简单的 SQL select, 但不支持 join	功能由数据类型和 SQL 语言标准定义 有 UDF, 但支持得不好

表 1: MapReduce vs. SQL Select

MapReduce/Hadoop 系统架构

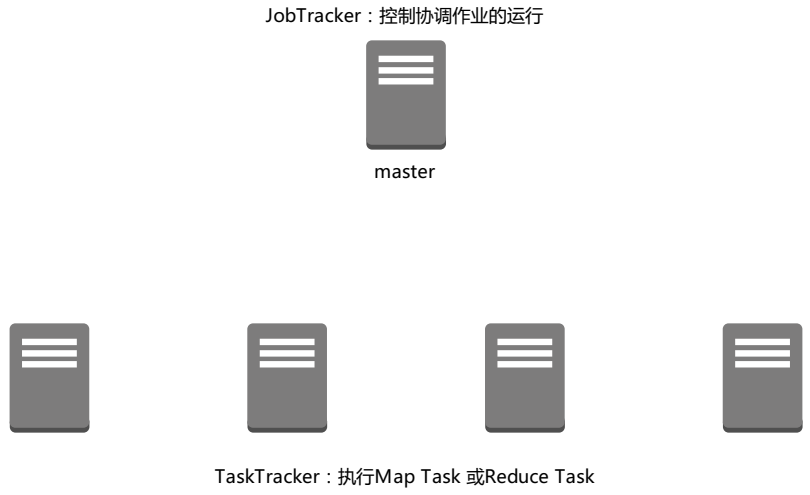

图 1：MapReduce/Hadoop 系统架构.

MR 运行流程图

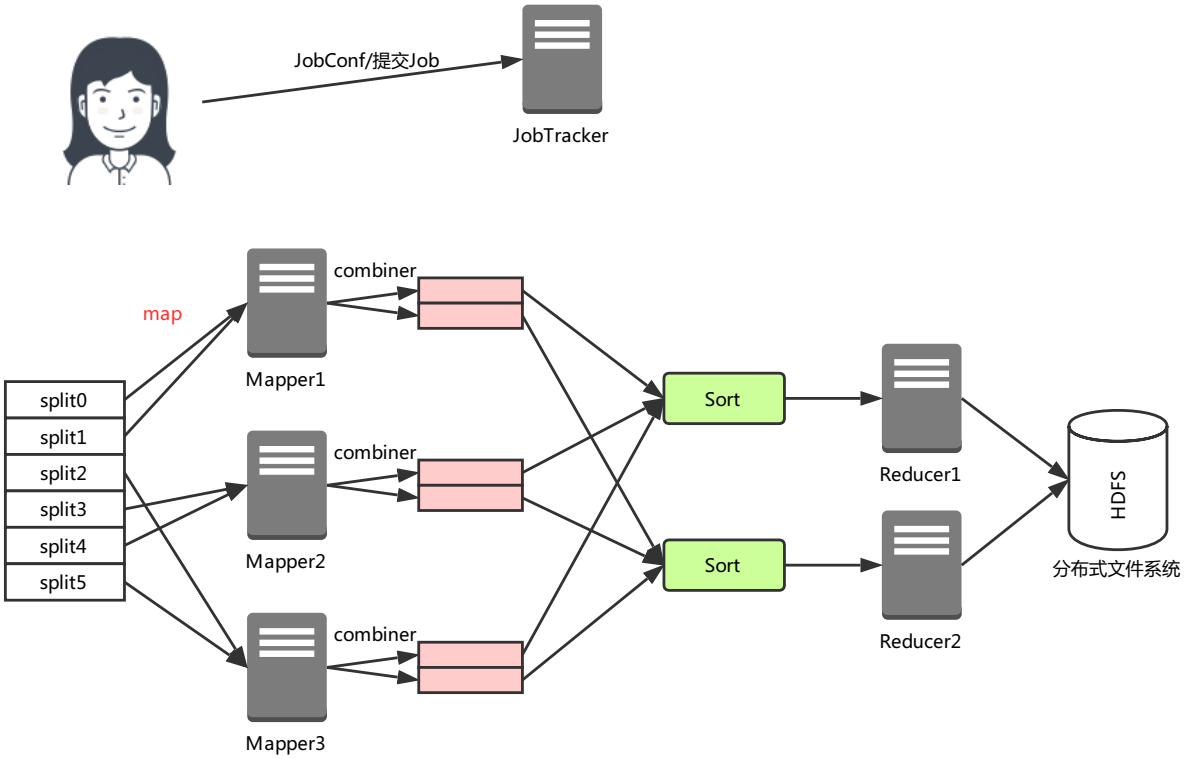

图 2：MR 运行.

MR: Fault Tolerance (容错)

- HeartBeat(心跳) 消息
  - 定期发送，向 JobTracker 汇报进度
- JobTracker 可以及时发现不响应的机器或速度非常慢的机器
  - 这些异常机器被称作 Stragglers
- 一旦发现 Straggler
  - JobTracker 就将它需要做的工作分配给另一个 worker
- Straggler 是 Mapper，将所对应的 splits 分配给其它的 Mapper
  - 输入数据是分布式文件，所以不需要特殊处理
  - 通知所有的 Reducer 这些 splits 的新对应 Mapper
  - Shuffle 时从新对应的 Mapper 传输数据
- Stragger 是 Reducer，在另一个 TaskTracker 执行这个 Reducer
  - 这个 Reducer 需要重新从 Mappers 传输数据
  - 注意：因为 Mapper 的输出是在本地文件中的，所以可以多次传输

Microsoft Dryad

- Dryad 是对 MapReduce 模型的一种扩展
  - 组成单元不仅是 Map 和 Reduce，可以是多种节点
  - 节点之间形成一个有向无环图 DAG(Directed Acyclic Graph)，以表达所需要的计算
  - 节点之间的数据传输模式更加多样
    * 可以是类似 Map/Reduce 中的 shuffle
    * 也可以是直接 1:1、1: 多、多:1 传输
  - 比 MapReduce 更加灵活，但也更复杂
    * 需要程序员规定计算的 DAG

同步图计算系统

MapReduce + SQL 系统

Hive

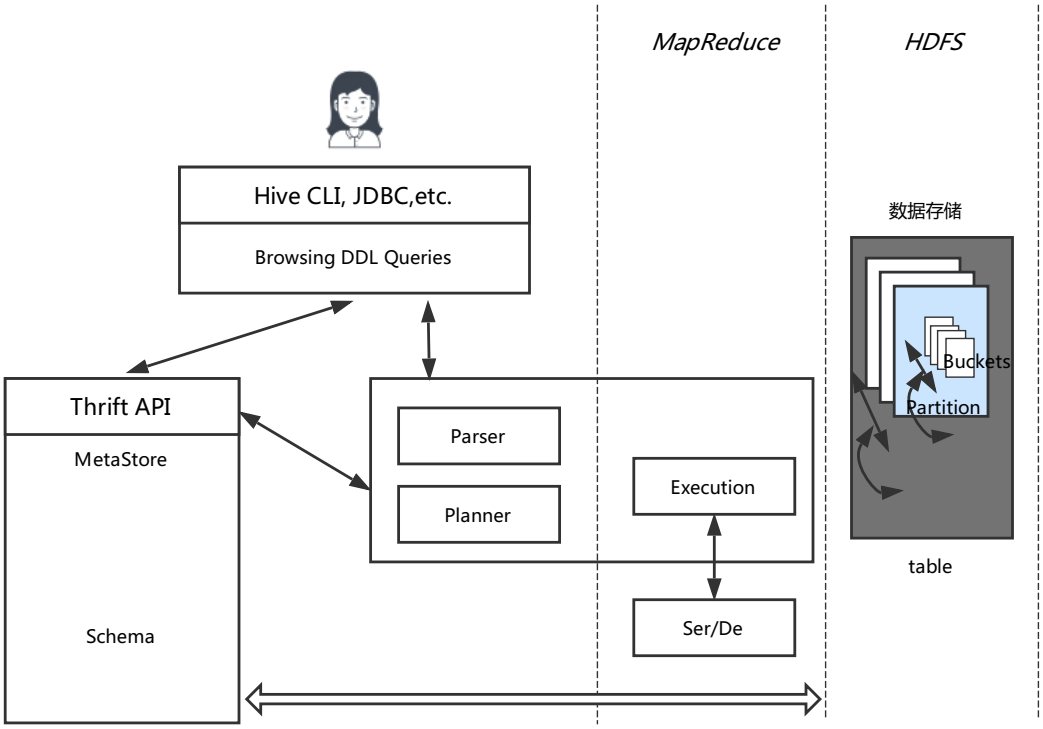

图 3：Hive.

- 数据存储在 HDFS 上
- Table：一个单独的 hdfs 目录
- Table 可以进一步划分为 Partition
- Partition 可以进一步划分为 Bucket
- Partition：每个 Partition 是 Table 目录下的子目录

- Bucket：每个 Bucket 是 Partition 目录下的子目录
- Hive QL
  - 类似 SQL
  - 部分 SQL 和扩展
  - 采用 MapReduce 执行
- SerDe
  - 序列化/反序列化
- 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 CLI
  - 命令行客户端，可以执行各种 HiveQL 命令

Hive 数据模型

- 关系型表 + 扩展
- 关系型表
  - 无序的记录
  - 每个记录可以包含多个列
  - 每个列可以是原子数据类型
    * 例如：integer types, float, string, date, boolean

Example

```
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

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
- 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

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

Partition 使用举例

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

表达 MapReduce

```
FROM (
MAP doctext USING 'python wc_mapper.py' AS (word, cnt)
FROM docs
CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

Example Query (Filter)

```
SELECT * FROM status_updates
WHERE status LIKE 'michael jackson'
```

Example Query (Aggregation)

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

Join 实现

- Multi-way join 优化
  - (a.key = b.key) and (b.key = c.key)
  - 同时传输多个表
- Map-side join
  - 没有 reducer
  - 其中一个表 R 很小
  - 那么每个 Map task 都读整个的 R，与 S 的一部分 join
  - Mapper 在这里只是运行并行程序的容器，Map 函数会完成一整个 simple hash join

内存计算

关系型内存数据库

- Main memory database system
- Memory-resident database system
- 上述两个名词有区别
  - Memory-resident：可能是在 buffer pool 中
  - MMDB: 可能彻底不用 buffer pool，改变了系统内部设计

Sorting

- 使用 quick sort 而不是 replacement selection
  - 顺序访问 vs. 随机访问
  - 当快排缩小到 cache size 以下时，就没有 cache miss

其它关键技术

- Vectorization
  - 每个 Operator 不是一次调用 next() 仅返回一条记录
  - 而是返回一组记录，提高代码利用率，形成紧凑循环
- 处理器加速
  - SIMD

  - GPU
  - Multi-core
- 压缩
  - 简单的压缩
    * 例如，根据一个列的取值范围为 [0,2 k -1]，采用 K bit 表示
  - 字典压缩
    * 把数据映射为整数

MonetDB

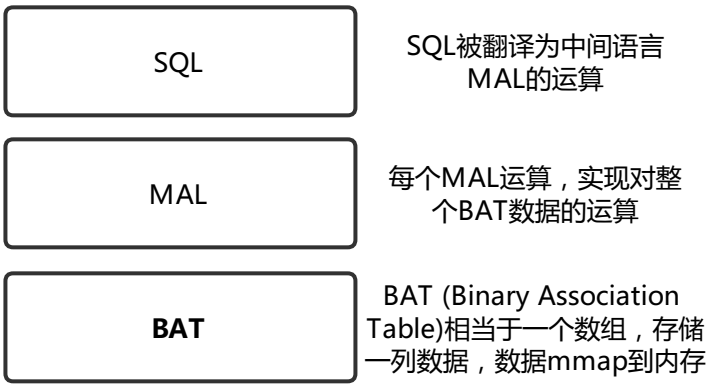

图 4: MonetDB.

内存 key-value 系统

- Memcached
  - 用户：Facebook, twitter, flickr, youtube, ⋯
  - 单机的内存的 key-value store
  - 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
  - 提供更加丰富的类型
  - key 可以包含 hashes, lists, sets 和 sorted sets
  - 支持副本和集群

Memcached

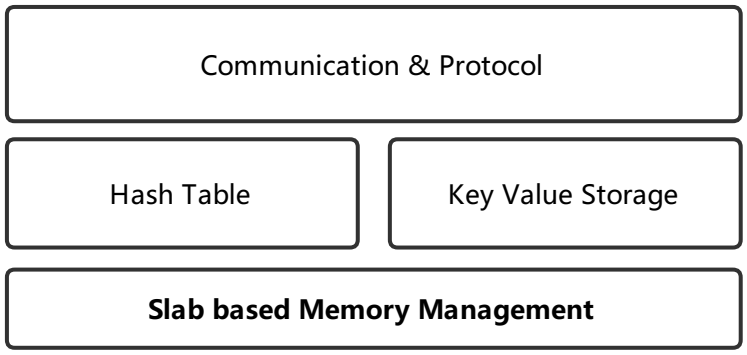

图 5: Memcached.

- 实现自定义的协议，支持 GET/PUT 等请求和响应-**Communication & Protocol**
  - 文本方式的协议
  - 二进制协议
- 采用多个链表管理内存-**Slab based Memory Management**
  - 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
- Hash Table
  - Key-Value 采用一个全局的 Hash Table 进行索引
  - 多线程并发互斥访问
- Key Value Storage
  - 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

内存 MapReduce

Spark: 面向大数据分析的内存系统

- 可以从 HDFS 读数据，但是运算中数据放在内存中，不使用 Hadoop，而是新实现了分布式的处理
- 目标是低延迟的分析操作
- MapReduce 的问题
  - 通过 HDFS 进行作业间数据共享，代价太高
- Spark 的思路

  - 内存容量越来越大
  - 把数据放入多台机器的内存
  - 避免 HDFS 开销
- Spark 的基础数据结构：RDD
  - Resilient Distributed Data sets
    * 一个数据集
    * 只读，整个数据集创建后不能修改
    * 通常进行整个数据集的运算
  - 优点
    * 并发控制被简化了
    * 可以记录 lineage（数据集上的运算序列），可以重新计算
      · 并不需要把 RDD 存储在 stable storage 上

RDD vs. Distributed Shared Memory

Aspect	dRDDs	Distr. Shared Mem.
Writes	dCoarse-grained	Fine-grained
Consistency	dTrivial (immutable)	Up to app / runtime
Fault recovery	dFine-grained and lowoverhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	dPossible using backup tasks	Difficult
Work placement	dAutomatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	dSimilar to existing data flow systems	Poor performance (swapping?)

表 2: Comparison of RDDs with distributed shared memor

两类 RDD 运算

- Transformation
  - 输入是 RDD(数据集)
  - 输出也是 RDD(数据集)
  - RDD $\Longrightarrow$ RDD
- Action
  - 输入是 RDD(数据集)
  - 输出是某种计算结果（例如，一个数值或者一列数值）
    * 注意：RDD 可能非常大，但是计算结果总是比较小的
  - RDD $\Longrightarrow$ 计算结果

Transformation

Trans.	$map(f : T \Rightarrow U)$: RDD[T] $\Rightarrow$ RDD[U] $filter(f : T \Rightarrow Bool)$: RDD[T] $\Rightarrow$ RDD[T] $flatMap(f : T \Rightarrow Seq[U])$: RDD[T] $\Rightarrow$ RDD[U] $sample(fraction : Float)$: RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling) $groupByKey()$: RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])] $reduceByKey(f : (V, V) \Rightarrow V)$: RDD[(K, V)] $\Rightarrow$ RDD[(K, V)] $union()$: (RDD[T],RDD[T]) $\Rightarrow$ RDD[T] $join()$: (RDD[(K, V)],RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))] $cogroup()$: (RDD[(K, V)],RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))] $crossProduct()$: (RDD[T],RDD[U]) $\Rightarrow$ RDD[(T, U)] $mapValues(f : V \Rightarrow W)$: RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning) $sort(c : Comparator[K])$: RDD[(K, V)] $\Rightarrow$ RDD[(K, V)] $partitionBy(p : Partitioner[K])$: RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Action	$count()$: RDD[T] $\Rightarrow$ Long $collect()$: RDD[T] $\Rightarrow$ Seq[T] $reduce(f : (T, T) \Rightarrow T)$: RDD[T] $\Rightarrow$ T $lookup(k : K)$: RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs) $save(path : String)$: Outputs RDD to a storage system, e.g., HDFS

表 3: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

系统实现

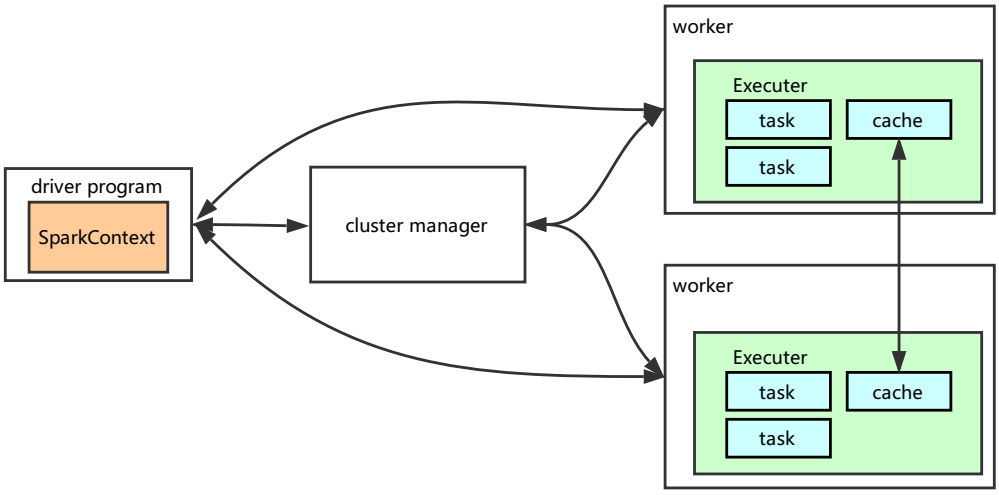

图 6: Spark 的系统结构

运算过程

- 每个应用程序
  - 一个自己的 SparkContext，多个 Executor
  - SparkContext 从外部的某种资源管理系统获取资源
    - 例如：standalone, hadoop YARN, apache Mesos
  - 每个 executor 运行在一个不同的 worker node 上
  - SparkContext 协调多个 worker 运行
- 应用程序
  - 有一个 driver 主程序，创建 SparkContext，发出各种 RDD 操作要求
- Executor：执行并行的运算，存储数据
- 多个应用程序
  - 各自有自己的 SparkContext
    - 互相隔离，但是也无法共享数据
  - 必须通过外部的文件系统进行数据共享

Spark 运算的运行

- Transformation
  - Lazy execution
  - 仅记录，不运算
- Action
  - 当遇到 Action 时，需要返回结果，才真正执行已经记录的前面的运算

- 容错/内存缓冲替换：当内存缓冲的 RDD 丢失时
  - 可以重新执行记录的运算，重新计算这个 RDD

Cloudera Impala

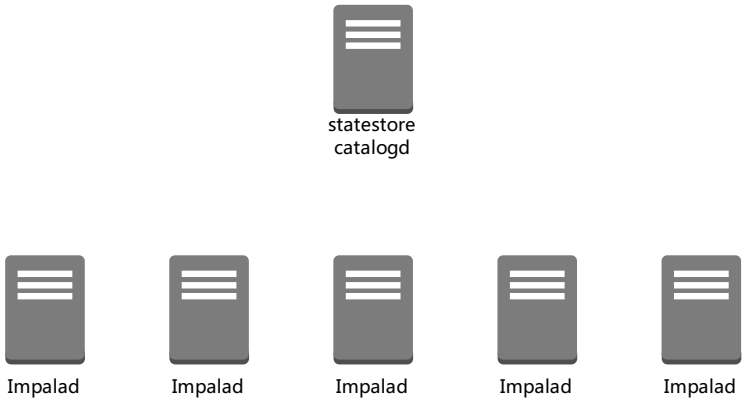

图 7: Impala 的系统结构

- Impalad：并行运算
- Statestore：监控 impalad 的状态
- Catalogd：SQL 的表格的 schema 的增删改操作