等值连接在MapReduce下的三种实现及比较[[1]](#footnote-1)🞼

汤凯 1101213993 陈晋升 1101213970

(北京大学 信息科学技术学院,北京 100871);

E-mail: {tangkai,chenjinsheng}@pku.edu.cn

摘要:

本文主要完成了在MapReduce框架下等值连接的多种实现，包括MapSide Join，ReduceSide Join和Broadcast Join，文章比较了他们的性能差别，并分析了可能的原因。文章还做了Mapper和Reducer数量的性能调优测试，并分析了原因。文章最后还试图测试大小差异很大的数据集下不同算法的性能差异。

本文特色在于实现了多种Join算法，并比较他们的差异。

完成此项目收获一方面锻炼了我们OOP的能力，另一方面让我们熟悉了当今非常热门的MapReduce框架，对于今后在计算机领域进一步学习实践都有很大的好处。

# 引言

我们生活在数据时代！很难估计全球存储的电子数据总量是多少，但是据IDC估计2006年"数字全球"项目(digital universe)的数据总量为0.18 ZB，并且预测到2011年这个数字将达到1.8 ZB，为2006年的10倍。一个ZB等于1021字节，或者说1ZB=一千（103）EB=一百万（106）PB=十亿（109）TB=千亿（1012）GB，往往PC机的硬盘空间都在0.5T到1T之间。

这一数据洪流有许多来源。考虑以下数据：

* 纽约证券交易所每天产生1 TB的交易数据。
* 著名社交网站Facebook的主机存储着约100亿张照片，占据PB级存储空间。
* 瑞士日内瓦附近的大型强子对撞机每年产生约15 PB的数据。
* 国内大型互联网公司，例如百度和淘宝，每天新增的数据都在PB

此外还有大量数据。但是你可能会想它对自己有何影响。大部分数据被锁定在最大的网页内容里面(如搜索引擎)或者是金融和科学机构，是不是所谓的"大数据"的出现会影响到较小的组织或个人？

个人数据的产生量正在快速地增长。微软研究院的MyLifeBits项目(http://research.microsoft.com/en-us/projects/mylifebits/default.aspx)显示，在不久的将来，个人信息档案将可能成为普遍现象。MyLifeBits是这样的一个实验：一个人与外界的联系(电话、邮件和文件)被抓取和存储供以后访问。收集的数据包括每分钟拍摄的照片等，导致整个数据量达到每月1 GB的大小。当存储成本下降到使其可以存储连续的音频和视频时，服务于未来MyLifeBits项目的数据量将是现在的许多倍。

个人数据的增长的确是大势所趋，但更重要的是，计算机所产生的数据可能比人所产生的数据更大。机器日志、RFID读取器、传感器网络、车载GPS和零售交易数据等，这些都会促使"数据之山越来越高"。

公开发布的数据量也在逐年增加。作为组织或企业，再也不能只管理自己的数据，未来的成功在很大程度上取决于它是否能从其他组织的数据中提取出价值。

这方面的先锋(如亚马逊网络服务器、Infochimps.org或者andtheinfo.org)的公共数据集，它们的存在就在于促进"信息共享"，任何人都可以共享并自由(或以AWS平台的形式，或以适度的价格)下载和分析这些数据。不同来源的信息混合处理后会带来意外的效果和至今难以想像的应用。

有句话是这么说的："算法再好，通常也难敌更多的数据。"意思是说对于某些问题(譬如基于既往偏好生成的电影和音乐推荐)，不论你的算法有多么猛，它们总是会在更多的数据面前无能为力(更不用说没有优化过的算法了)。

在海量数据面前，在高性能的计算机都显得无能为力，现在通常的做法是使用分布式平台经常处理。

对于数据，仅仅是存储是不够的，我们还需要查询（query）数据。但是查询如此海量的数据，我们需要一个完全不同的角度来审视查询算法。

在对于数据查询的各项操作中最常用的是Join操作，Join往往是根据某些条件，将一个表或者多个表合并在一起。根据条件不同，Join分为等值连接和非等值连接。在传统数据库中Join处理方法分为以下几种：Nested Loops Join、Sort-Merge Join和Hash Join。

大数据集的情况下，如何完成用分布式系统完成Join显得十分有意义。

课程设计实现了在目前非常流行的MapReduce框架下，等值连接的多种实现包括Mapside Join，Reduce Join的两种实现以及Broadcast Join。

本文的组织：在第二部分中，我们介绍了本课程设计需要的相关工作，包括Hadoop的基本框架和传统数据库中Join的几种方法；第三部分描述了课程设计的系统框架和核心算法；第四部分给出相关的实验数据；第五部分总结了全文，并且描述了完成本课程设计的收获。

# 相关工作

## Hadoop简介

Hadoop 是Apache Software Foundation 公司开发的一个基于java语言实现的MapRedu系统基础架构，由Apache Lucene 的设计者Doug Cutting开发的。2005年秋天Hadoop作为 Lucene 的子项目 Nutch 的一部分正式引入。它受到最先由 Google Lab 开发的 MapReduce 和 Google File System 的启发。2006 年 3 月份，MapReduce 和 Nutch Distributed File System (NDFS) 分别被纳入称为 Hadoop 的项目中。

Hadoop 是最受欢迎的在 Internet 上对搜索关键字进行内容分类的工具，但它也可以解决许多要求极大伸缩性的问题。例如，如果您要 grep 一个 10TB 的巨型文件，会出现什么情况？在传统的系统上，这将需要很长的时间。但是Hadoop在设计时就考虑到这些问题，因此能大大提高效率。

### Hadoop Job

为了能在Hadoop集群上运行一个Map/Reduce作业流，客户端程序不许建议一个JobConf 配置文件。配置文件内容如下：

定义实现Mapper和Reducer接口的类。

-JobConf.setMapperClass(),setReducerClass()

指定输入，输出

-FileInputFormat.addInputPath(conf)

-FileInputFormat.setoutputPath(conf)

其他选项：

-JobConf.setNumReduceTasks()

-JobConf.setOutputFormat()

JobTracker 将程序和JobConf文件放入到一个共享的位置。TaskTrackers运行一个简单的循环来查询JobTracker来索取作业特定的Jar和配置文件。

### Map和Reduce函数

在Hadoop中Map和Reduce类继承MapReduceBase类。定义如下：

Map函数：map(WritableComparable key，Writable value, OutputCollector output, Reporter reporter)

Reduce函数：reduce(WritableComparable key, Iterator values, OutputCollector output, Reporter reporter)

### Partitioning和Grouping

Hadoop框架从Mapper拿到输出数据后，会做接下来几项操作：

1. 把Mapper的输出Partition。
2. 在每个Partition内排序。
3. 把相关的Partition传给Reducer。
4. 把来自不同Mapper的partition merge起来。
5. 把partition的tuple分组，然后调用reduce函数

## Join 算法

在标准数据库环境中，有三种以实现且比较流行的Join算法

1. Nested Loops Join
2. Sort-Merge Join
3. Hash Join

### Nested Loops Join

这是一个比较旧和最简单的算法，它能基于任何连接条件在两个数据集上实现连接，他不像接下来要介绍的两个算法只能实现等值连接。但是在本文中所有使用Map/Reduce的Join算法都是基于等值连接。

### Sort-merge join

给定两个数据集P和Q，Sort-Merge Join按照连接属性给两个数据集进行排序，然后在两个数据集上找到需要Join的元组p。排序将在连接属性列上具有相同之的元组放在一个组，这样就是在连接属性列上具有相同值得元组进行分片或者分组。分片只需要比较一个分片上的P元组和在相同分片上的Q元组，因此避免逐条检查P和Q。这种基于分片的方法只适用于等值连接。

### Hash Join

Hash Join 算法主要有“build”阶段和“probe”阶段。在最检点的版本里，较小的数据集在bulid阶段被装载到内存的Hashtable中，在probe阶段，扫描较大的数据集并通过查看Hashtable连接相关的元组。这个算法像Sort-Merge Join 也只能进行等值连接。

# 课程设计的系统框架和核心算法

## Join综述

在Join的过程中，我们使用如下的Table作为sample Input，帮助理解Join在MapReduce框架下的应用。

Table1 sample:

|  |  |
| --- | --- |
| StuID | StuName |
| 1001 | Kevin |
| 1002 | Mike |

Table2 sample:

|  |  |  |
| --- | --- | --- |
| StuID | ClassName | ClassScore |
| 1001 | Algorithm | 98 |
| 1001 | Database | 99 |
| 1002 | Algorithm | 97 |
| 1002 | Database | 100 |
| 1002 | Soft Testing | 88 |

output sample:

|  |  |  |  |
| --- | --- | --- | --- |
| StuID | StuName | ClassName | ClassScore |
| 1001 | Kevin | Algorithm | 98 |
| 1001 | Kevin | Database | 99 |
| 1002 | Mike | Algorithm | 97 |
| 1002 | Mike | Database | 100 |
| 1002 | Mike | Soft Testing | 88 |

程序结构

─edu

└─pku

├─broadcast

│ BroadcastJoin.java

│ BroadcastMapper.java

│

├─mapside

│ MapSideJoin.java

│ SequenceFileIO.java

│ Sort.java

│

├─reduceside

│ JoinReducer.java

│ ReduceSideJoin.java

│ TableOneMapper.java

│ TableTwoMapper.java

│

├─reducesidenew

│ ReduceSideJoinNew.java

│

├─test

│ ConcatTest.java

│ DatasetGen.java

│ JobConfTest.java

│

└─util

DatasetFactory.java

TableOneParser.java

TableTwoParser.java

TextPair.java

所有源码放在**edu.pku.\***的Package下面。

**edu.pku.util**是整个Join框架所需的工具集，包括两个Table的Parser用于封装parse过程，让map阶段显得更干净。DatasetFactory用于生产测试数据。TextPair是一个自定义的数据类型，用于生产text对。

**edu.pku.mapside**是Mapside Join的实现，包括Mapper和Join过程。

**edu.pku.reduceside**是Reduceside Join的实现，包括Mapper，Reducer和Join过程。

**edu.pku.reducesidenew**是Reduceside Join NEW API的实现，包括Mapper，Reducer和Join过程。

**edu.pku.broadcast**是BroadcastJoin的实现，包括Mapper和Join过程。

**edu.pku.test**是少量中间test。

## Map-Side Join

Reduce-Side Join似乎是使用Map/Reduce实现数据集连接的一般方法，在key-value对到达Reducer之前它会用系统内置的功能对这些中间结果进行排序。但是排序这步是非常耗时间的。Hadoop提供了另一种数据集在到达Mapper之前实现连接的方法。这种方法的性能更加优异，已经被证实是使用Map/Reduce实现两个数据集连接最快的方法，但是要想使用Map-Side实现Join必须满足下面表1中条件：

|  |  |
| --- | --- |
| 限制条件 | 原因 |
| 所有的数据集必须使用相同的比较器排序 | 每个数据集中被连接的排好序的数据必须是相等的 |
| 所有的数据集必须用相同的分割器进行分片 | 在每个数据集中一个给定的key必须在相同的分片中，所以保留一个key的分片被连接到一起 |
| 在数据集中分片数必须相等的 | 在每个数据集中一个给定的key必须在相同的分片中，所以保留一个key的分片被连接到一起 |

这些限制条件虽然非常严格但是都合乎一个Hadoop job的任何一个输出数据集。因此作为预处理，我们简单地通过一个基本的Hadoop job传送两个数据集。这部分工作会用到IdentityMapper和IdentityReducer，它们不处理数据，只是传送到系统框架。系统会将这些数据进行分片、分组和排序。输出遵从上面提到的所有限制条件。

在一个连接中，尽管单个map任务体现不出数据局部性的优势，但是整个作业会因为可能淘汰reduce阶段和（或者）需要reduce的数据量大量减少而受益。该算法也支持数据集中有多个key副本这种情况。

## Reduce-Side Join

在这个算法中，正如这个名字所暗示的那样，正在的Join操作发生在框架的Reduce端。’map’ 阶段只对来自两个dataset中tuple的预处理，具体而言就是针对join key来对他们重排。

### Map阶段

**Map**函数每次通过HDFS的stream从两个数据集中读一个tuple。Tuple的的连接键被fetch作为**map**的key，tuple剩下部分作为与key关联的value。同时，我们在key里面还打tag用于标记这个tuple到底来自哪个数据集。我们自定义了一个类**TextPair**（详解附录）。这个类相当于Java中**Pair**类，只不过他的两个域都是存放**Text**类型的。**Map**函数用这个类来给key打tag标定他们的来源数据库，如果没有做来源数据集的区分，Join是无法实施的。**Map**函数如下列出。

**public** **class** TableOneMapper **extends** MapReduceBase **implements**

Mapper<LongWritable, Text, TextPair, Text> {

**private** TableOneParser parser = **new** TableOneParser();

**public** **void** map(LongWritable key, Text value,

OutputCollector<TextPair, Text> output, Reporter reporter)

**throws** IOException {

**if** (parser.parse(value)) {

output.collect(

**new** TextPair(parser.getPrimaryKey(), parser.getTag()),

**new** Text(parser.getContentColumn()));

}

}

}

**public** **class** TableTwoMapper **extends** MapReduceBase **implements**

Mapper<LongWritable, Text, TextPair, Text> {

**private** TableTwoParser parser = **new** TableTwoParser();

**public** **void** map(LongWritable key, Text value,

OutputCollector<TextPair, Text> output, Reporter reporter)

**throws** IOException {

**if** (parser.parse(value)) {

output.collect(

**new** TextPair(parser.getPrimaryKey(), parser.getTag()),

**new** Text(parser.getContentColumn()));

}

}

}

**TableOneParser**和**TableTwoParser**将在附录中详细说明。

### Partitioning和Grouping阶段。

Partitioner根据每个tuple的join key（连接键）完成对tuple的分区，确保同样的key进入到同样的reducer。考虑到我们的key是这样的形式的<Join-key, tag>的**TextPair**，我们需要overwrite默认的partitioner来确保仅仅是根据join key来进行分区。Tag则被partitioner忽略。Tag值只是在reducer阶段用于辨识tuple的母数据集。

KeyPartitioner方法如下所列。

**public** **static** **class** KeyPartitioner **implements** Partitioner<TextPair, Text> {

@Override

**public** **void** configure(JobConf job) {

}

@Override

**public** **int** getPartition(TextPair key, Text value, **int** numPartitions) {

**return** (key.getFirst().hashCode() & Integer.*MAX\_VALUE*)

% numPartitions;

}

}

但是仅是Paritioner还不够，即便Parition阶段可以确保有相同join key的tuple去同一个reducer，但是还是有一个问题没有解决。 Partition之后还要经过group阶段才会调用reducer，而grouping也是根据<join key, tag>的**TextPair**的key来分组，这意味着，对于<Key1, Tag1>和<Key1, Tag2>会调用两个不同的reducer，这显然与预期不是一致的。为了解决这个问题，我们自行指定了一个grouping的comparator函数。这个函数本质上是一个RawComparator，他继承了**WritableComparator**，overwrite内置的**compare**()方法。实现自定义类型**TextPair**的RawComparator的好处在于，在Hadoop框架中，当它被读成reduce端的一个key时需要先反序列化，才能比较，使用这个RawComparator可以使得TextPair仍然是字节流的时候就能比较大小。这个Comparator放在**TextPair**类中。

FirstComparator方法列出如下：

**public** **static** **class** FirstComparator **extends** WritableComparator {

**private** **static** **final** Text.Comparator *TEXT\_COMPARATOR* = **new** Text.Comparator();

**public** FirstComparator() {

**super**(TextPair.**class**);

}

@Override

**public** **int** compare(**byte**[] b1, **int** s1, **int** l1, **byte**[] b2, **int** s2, **int** l2) {

**try** {

**int** firstL1 = WritableUtils.*decodeVIntSize*(b1[s1])

+ *readVInt*(b1, s1);

**int** firstL2 = WritableUtils.*decodeVIntSize*(b2[s2])

+ *readVInt*(b2, s2);

**return** *TEXT\_COMPARATOR*

.compare(b1, s1, firstL1, b2, s2, firstL2);

} **catch** (IOException e) {

**throw** **new** IllegalArgumentException(e);

}

}

@Override

@SuppressWarnings("rawtypes")

**public** **int** compare(WritableComparable a, WritableComparable b) {

**if** (a **instanceof** TextPair && b **instanceof** TextPair) {

**return** ((TextPair) a).first.compareTo(((TextPair) b).first);

}

**return** **super**.compare(a, b);

}

}

### Reduce阶段

考虑我们的table是有主键的，在table1中join key每次只会出现一次。考虑到Hadoop框架会根据key值（即**TextPair**键值对）排序，然后把它们传到reduce端，由于是对key的整个键值对排序，且来自table1的数据tag值比来自table2的数据小，所以来自table1的tuple都排在table2的tuple前面。考虑到table1中join key是该table的主键，因此来自table1的tuple在同一个reducer里面只有一个，剩下的均为来自table2的tuple。

此时问题变得更加清晰，reducer的作用就是把来自第一个dataset的唯一一个tuple放在内存中buffer起来和后面来的每一个来自另外一个dataset的tuple来做join即可。

JoinReducer的代码列举如下：

**public** **class** JoinReducer **extends** MapReduceBase **implements**

Reducer<TextPair, Text, Text, Text> {

**public** **void** reduce(TextPair key, Iterator<Text> values,

OutputCollector<Text, Text> output, Reporter reporter)

**throws** IOException {

Text studentName = **new** Text(values.next());

**while** (values.hasNext()) {

Text record = values.next();

Text outValue = **new** Text(studentName.toString() + "\t"

+ record.toString());

output.collect(key.getFirst(), outValue);

}

}

}

## BroadCast join

如果实现join的数据集中有一个数据集特别小并且能放到内存中，那么就可以使用BroadCast join这种方法。它能避免从Mappers传送数值到Reducers而导致的传输开销。这种情况在现实应用中经常出现。例如：一个小量用户数据库可能和一个庞大的日志进行连接，这个小的数据集可以被复制到每个节点上。在参与Hadoop job时通过使用-file or -archive直接将这个文件传送到每个节点。Broadcast join是一个只需要Map的算法。

### Map阶段

Mapper将小的数据集调入内存中并对较大的数据集中的每个元组调用map函数。对于每一个（key，value），map函数在探测放在内存的数据集并且找到匹配项。这个过程可以通过将小数据集调入一个Hashtable中作进一步优化。然后写出已实现join的元组。下面显示的是**Mapper**类中**configure**函数，它会被每个Mapper调用。下面代码是用来读被传播的文件并装载到内存的Hashtable中。

**public** **void** configure(JobConf conf) {

T1 = **new** File(conf.get("TableOnePath"));

BufferedReader br = **null**;

String line = **null**;

**try** {

br = **new** BufferedReader(**new** FileReader(T1));

**while** ((line = br.readLine()) != **null**) {

String record[] = line.split("\t", 2);

**if** (record.length == 2) {

**if** (ht.containsKey(record[0])) {

ht.get(record[0]).add(record[1]);

} **else** {

ArrayList<String> value = **new** ArrayList<String>();

value.add(record[1]);

ht.put(record[0], value);

}

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

下面这段代码是具体的map函数，通过map函数从HDFS中来接受记录，检测Hashtable中包含的被传播文件的元组。值得注意的是它适用于多个相同的key这种情况。

**public** **void** map(LongWritable LineNumber, Text value,

OutputCollector<Text, Text> output, Reporter reporter)

**throws** IOException {

String[] rightRecord = value.toString().split("\t", 2);

**if** (rightRecord.length == 2) {

**for** (String leftRecord : ht.get(rightRecord[0])) {

output.collect(**new** Text(rightRecord[0]), **new** Text(leftRecord

+ "\t" + rightRecord[1]));

}

}

}

Broadcast join优势在于每个节点上用一个小的“本地”存储器来代替HDFS，将整个数据集装载到内存Hashtable中使得访问速度变得非常快。但是缺点在于当两个数据集都很大，没有一个能被存储到单个节点的本地中时就会出现运行问题。

# 实验

## 实验环境描述

### 硬件环境

实验机器1：

CPU：Intel Core 2 T5550M

Memory：2GB

Hard Disk：160GB 5400rpm

OS：Ubuntu 10.10 Maverick

Hadoop Ver：0.20.205.0

实验机器2：

CPU：Intel Core 2 T6670M

Memory：2GB

Hard Disk：500GB 5400rpm

OS：Ubuntu 10.10 Maverick

Hadoop Ver：0.20.205.0

这两台机器均为笔记本且并没有组成集群，分别在本地做了standalone实验。

### 数据集

数据集是我们使用随机函数生成的两个表，具体详见**edu.pku.util.DatasetFactory**类。我们主要分两种情况生成数据集。

第一种情况是普通情况，我们对table 1生成了1000000条数据，包括1 million学生的ID和他们的姓名，姓名可能存在重复，但是ID不会重复。为了增加Join的工作量，检验排序的效果，学生的ID不是递增的，而是经过我们乱序处理。学生的姓名则是4位随机的小写字母。

对table2我们对应生成1000000个学生的ID和科目以及他们的成绩。我们事先设定好一个30门课的课程库。在普通模式下，一位同学只能选1-3门课，选了几门课，是哪几门课，以及课程的分数都是随机生成。为了增加Join的工作量，检验排序的效果，学生的ID不是递增的，而是经过我们乱序处理。

在普通模式下，两个表的行数在同一数量级，考虑到第二个表字符稍多，最终生成table1的大小和table2的大小分布是13M和42M，约为3倍关系。

第二种情况是倾斜情况，即我们试图考虑当两个表大小相差较大时各个Join的表现。

在生成table1时我们采取相同的策略，在生成table2时，我们把让每个同学可以随机选1-30门课，这样结果是在内存的限制下，我们大约只能生成100000个学生（0.1million个学生）的数据。生成的两个table大小分别是1M和20多M，大约20倍关系。

## 实验数据

### 实验一：验证各个Join方法性能。

在实验机器2上，我们进行简单测试。在相同数据集上每个算法运行时间如下表所示：

|  |  |
| --- | --- |
| 算法 | 耗时 |
| ReduceSide Join | 40s |
| ReduceSide Join(NewAPI) | 40s |
| MapSide Join | 28s |
| BroadCast Join | 62s |

### 实验二：测试不同mapper和reducer数量对Join的性能影响

在实验机器1上，我们进行对ReduceSideJoin进行如下测试。

1. 默认情况

我们并不手动指定mapper和reducer数量，让系统自动去给定。

1. M1R1

我们指定mapper和reducer数量均为1。

1. M1R2

我们指定mapper和reducer数量分别为1和2。

1. M2R1

我们指定mapper和reducer数量分别为2和1。

1. M2R2

我们指定mapper和reducer数量均为2。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mapper Reducer数量设置 | 耗时  s | CPU时间  ms | 总提交堆的使用量bytes | 物理内存使用量bytes | 虚拟内存使用量bytes |
| Default | 95 | 80130 | 790298624 | 1385717760 | 2432024576 |
| M1R1 | 80 | 66110 | 436928512 | 730152960 | 1393410048 |
| M1R2 | 84 | 82720 | 382795776 | 966037504 | 1953124352 |
| M2R1 | 93 | 82280 | 742785024 | 1387130880 | 2444017664 |
| M2R2 | 102 | 91600 | 789577728 | 1545216000 | 2934829056 |

### 实验三：检验数据集大小相差很大的情况下Join性能

限于时间因素，这个实验并未实现，我们猜测这种情况下Broadcast Join的效果最好。因为当一个表小，另一个表大的时候，HashTable的优势就体现出来，把小表放HashTable里面，对于找大表而免去排序会体现出好处。

## 实验结论

### 实验一的结论

Mapside Join的速度最快，这是显而易见的，Mapside Join没有reduce过程，没有shuffle过程，没有partition和grouping过程，这个过程中减少了大量磁盘IO和CPU时间，效率最高是显而易见的。

Reduceside Join性能其次，其中NEW API（>0.20才支持）和OLD API性能接近，使用NEW API使用虚类而不是接口，拥有更好的扩展性。NEW API充分利用上下文对象Context，使用户代码能与MapReduc框架通信，不需要OLD API复杂的JobConf、OutputCollector和Reporter对象。

Broadcast Join性能最次，虽然也只有map阶段，但是Broadcast Join全程都使用了HashTable进行通信。但是不可否认的是，Broadcast Join对于输入数据格式最为灵活，对于Mapside Join和Reduceside Join，都要求table1是join key是primary key，即在table1中必须为一一对应的映射，但是broadcast join不需要这个限制，这样就更适合一些非正式的，不是数据库输入的场合。

例如我们有这样一个情景，很多人想去很多地方旅游，表一描述这些人在那几个时间上有空，表二描述这些景点在那些时间上开放，求他们最后的旅游计划，显然时间为join-key而且不是primary key，这种情况下使用Broadcast Join能够快速的解决这个问题，在web2.0时代，当数据变得越来越海量，非结构化时，使用Broadcast Join灵活快速的效果。

### 实验二的结论

### 性能接近在单机上reducer和mapper数量越多，时间越久。我们可以看到起多个mapper和reducer后，整个内存的使用量会大幅上升，在物理内存不够的情况下，虚拟内存的使用量也会上升，读写虚拟内存本质就是磁盘IO，因而对性能有大幅影响。

其次当起多个mapper和reducer后会有额外的merge开销，进程通信开销，这也会使Join的性能下降。

其次我们可以在log中观察map和reduce进度，发现当reduce设置成大于1的数量时，在map未完成时reduce也开始运行，这是并行化的体现。

# 总结

项目总结和个人收获

本项目先后经过三个周末的时间最终结束，在做项目的过程中我和陈晋升同学互相帮组克服了很多困难，我们最初通过文档wiki和书籍资料学习Hadoop的基本原理和操作流程，学习了新旧API的实现方式。然后开始编码和实现，经过各种调试，进一步熟悉了Java coding和OOP的感觉。

我们认为Hadoop的eclipse plugin是一个好工具，可以完成深度模拟，我们本来打算进一步放在集群上运行，限于时间因素未能成行，非常遗憾。

在调优部分，我们还有使用不同序列化框架，不同压缩框架的调优指标，以及诸如io.sort.mb等各种参数调优的打算，限于时间因素未能实现。

最后非常感谢高军老师和欧高炎助教的长期以来的指导和帮助。

References:

1. Apache hadoop. Website. <http://hadoop.apache.org>.
2. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparisonof join algorithms for log processing in mapreduce. In SIGMOD ’10 :Proceedings of the 2010 international conference on Management of data, pages 975–986, New York, NY, USA, 2010. ACM.
3. A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In SIGMOD ’09: Proceedings of the 35th SIGMOD international conference on Management of data, pages 165–178, New York, NY, USA, 2009. ACM.
4. T. White. Hadoop: The Definitive Guide. O’Reilly Media, 1 edition, June 2009.
5. Dawei Jiang Beng Chin Ooi Lei Shi and Sai Wu. The Performance of MapReduce: An Indepth Study In SIGMOD ’10

1. [↑](#footnote-ref-1)