分布式数据库 Final Report

小组成员

黄刚　　2009210911（组长）

汪汀　　2009210933

宋顺强　2009210981

Contents

第一部分　系统概述.....3
1　系统概述.....3
1.1　目标.....3
1.2　运行环境.....3
1.3　开发环境.....3
2　功能需求.....4
2.1　GUI 客户端.....4
2.2　SQL 解析.....5
2.3　通讯协议.....5
2.4　查询分解.....6
2.5　数据本地化.....7
2.6　全局查询优化.....7
第二部分　技术实现.....8
1.　系统架构.....8
2.　Parser.....8
3.　GDD.....8
4.　查询分解.....9
5.　数据本地化.....10
6.　全局查询优化.....10
7.　客户端.....11
第三部分　功能演示.....12
1　初始化.....12
2　查询与显示.....13
3　传输控制.....13
第四部分　分工总结.....15
1.　黄刚总结.....15
2.　汪汀总结.....15
3.　宋顺强总结.....16
4.　分工情况.....16

第一部分　系统概述

1　系统概述

1.1　目标

我们要实现一个分布式数据库系统，即通过网络将多个不同的局部数据库系统连接起来，使得用户可以通过分布式数据库管理系统达到透明性管理的目的。

在我们具体的实例中，我们通过网络将 3 个不同站点的局部数据库连接起来，并可以通过分布式数据库管理系统（Client）进行数据库创建，数据分片、分配、导入及 SQL 操作等管理。

1.2　运行环境

操作系统：　Windows XP or Linux (这个看着改吧)
软件需求：　JDK 1.5 以上 MYSQL5 以上

1.3　开发环境

开发语言：　JDK 1.6.0
底层局部数据库：　MYSQL 5.1
版本控制：　GIT
IDE：　Eclipse

2 功能需求

2.1 GUI 客户端

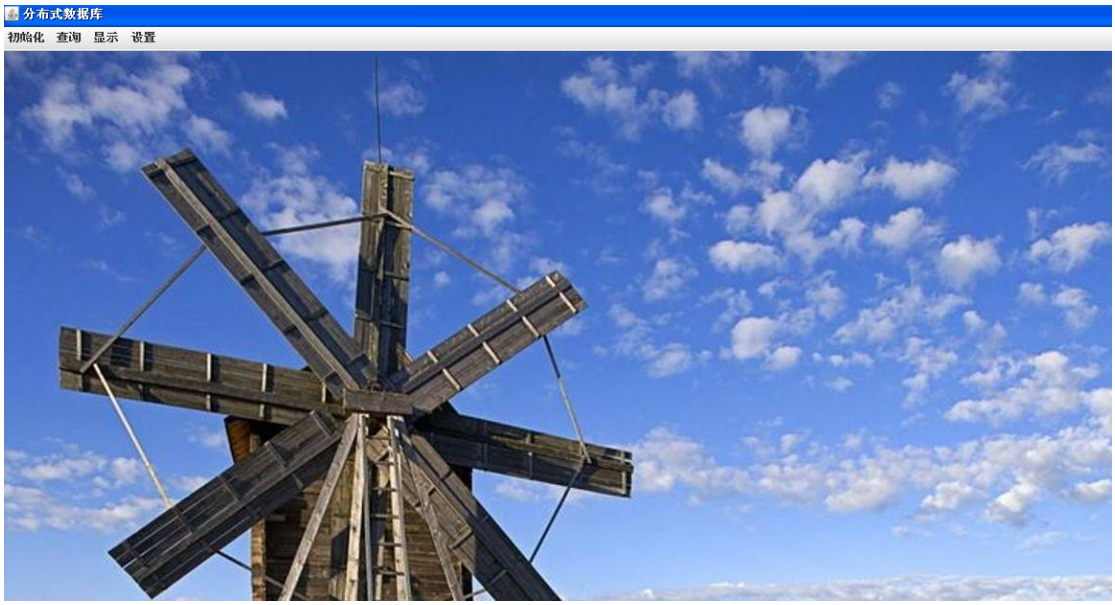


图 1：GUI 客户端启动外观

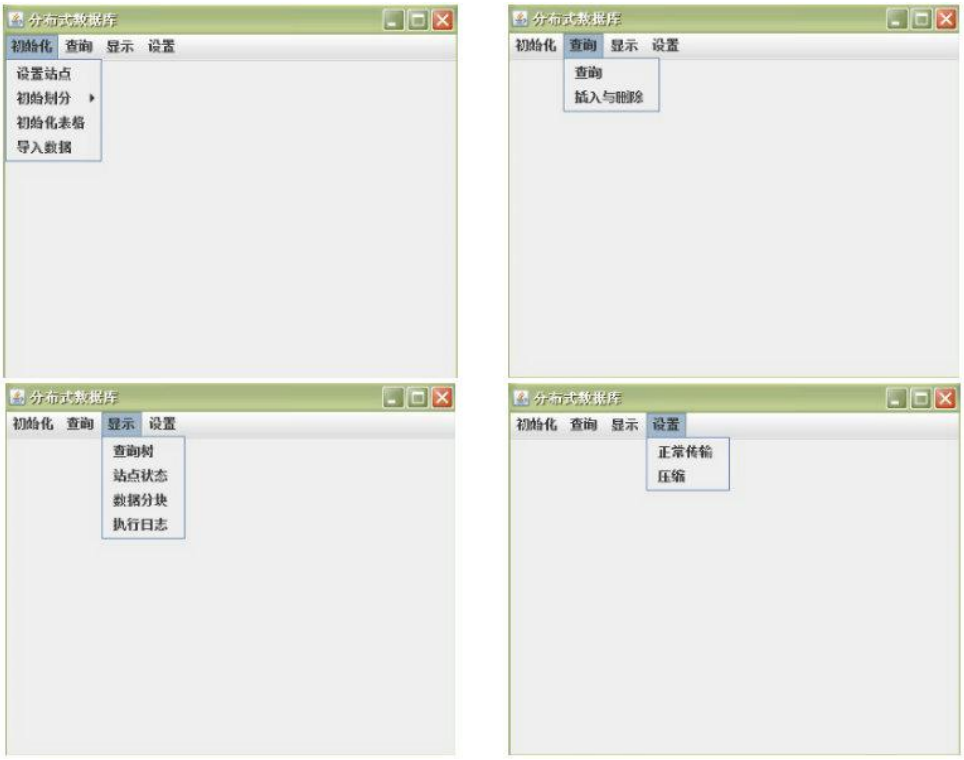


图 2：GUI 功能一览

GUI 要达到的基本功能就是提供分布式数据库的初始化控制，用户的查询输入及显示查询树；我们在此基础上还可对站点状况、数据分块情况及执行日志进行跟踪显示，并且提供数据压缩选项以便适应不同的网络情况。

具体地，当用户打开“查询”菜单并点击“查询”时，GUI 弹出一个输入窗口以供用户输入查询指令；当用户打开“显示”菜单并单击“查询树”时，GUI 会在窗体中显示当前查询对应的查询树。

2.2 SQL 解析

在 SQL 解析部分，我们要把从 UI 获得的 SQL 命令解析为系统可以理解的数据结构，以便后续查询分解及优化时使用。

在我们实验中，最基本的 SQL 命令，包括与集中式数据库相同与不同的，在下表中列出：

表格 1：SQL 解析用到的命令

	与集中式数据库相同	与集中式数据库不同
SQL 命令	CREATETABLE… INSERT INTO…VALUES… DELETE FROM…WHERE… SELECT…FROM…WHERE	DEFINESITE… … … HORIDIVIDETABLE… VERIDIVIDETABLE… ALLOCATE…TO… IMPORT…

表格左侧的命令不言自明，右侧是分布式数据库独有的，其中 DEFINESITE 用于初始化局部数据库站点，HORIDIVIDETABLE 和 VERIDIDVIDETABLE 分别用于将数据表进行水平划分和垂直划分（注：混合划分可以通过这两个命令来实现），ALLOCATE…TO…用于将划分后的子表分配到各个站点，IMPORT 则指定文本文件以便用通过插入文件内表项的操作完成数据表格的构建。

具体而言，

DEFINESITE 有三个参数，分别为站点名，站点 IP 和站点端口号；

HORIDIVIDETABLE 的第一个参数为要水平划分的表的名字，后面则有若干个划分条件，条件之间用逗号分隔，复杂划分条件中子条件用 AND 连接；

VERIDIVIDETABLE 的第一个参数为要垂直划分的表的名字，后面则有若干个划分条件，条件之间用空格分隔，条件的内容为划分表的列名（注：Primary Key 对应的列名在所有条件中出现）；

ALLOCATE…TO…有两个参数，分别为子表名和子表被分配的站点名；

IMPORT 只有一个参数，即表数据所在的文本文件名。

2.3 通讯协议

这部分主要包括客户端与服务器，服务器与服务器之间两部分。但对服务器端而言，他接受的命令无论来自哪里都是等价的。

我们的方案是将消息分类，分别处理。由于消息种类非常多，这里就不一一列举了，只

是简单的说明一下。

#Message1: 询问站点状态
Server Receive Message: Are you Ready?
Sever Response: Ready!
Server Receive: Message #1
Server Response: OK.
Server Receive : Bye!
Server Response: Bye!

#Message2: 查询指令
Server Receive Message: Are you Ready?
Sever Response: Ready!
Server Receive: Message #2
Server Response: OK.
Server Receive : Query
//Process Query…Done
Server Response: Done
Server Receive : Where & ID?
Server Response : SiteID & DataID
Server Receive: Bye!
Server Response: Bye!

#Message3: 获取数据
Server Receive Message: Are you Ready?
Sever Response: Ready!
Server Receive: Message #3
Server Response: OK.
Server Receive : DataID
Server Response: Data
Server Receive: Bye!
Server Response: Bye!

其他还有一些，大概总共有 15 种，格式与上面基本相同。

2.4 查询分解

在查询分解中，我们把从 UI 获得的命令映射成相应的操作树，并根据一些经验性规则，调整操作的先后顺序，获得初步优化后的操作树。

A. 规范化

在我们的实际系统中，我们假设获得的查询语句已经转换成规范形式。

B. 分析

在对查询语句的分析中，实现对类型错误和简单的语义错误的检查分析，除了确保查询语句是一个符合语法要求的正确的查询语句外，还包括对数据库中特定表格是否存在，表格是否具有特定的属性，以及谓词两端类型是否一致等进行检查，如果分析检查有错，则在这

一阶段直接拒绝该命令的执行。

C. 消除冗余

在我们的实际系统中，不实现消除谓词冗余的功能。

D. 重写

在重写阶段，首先根据查询命令生成查询树，在此过程中可以检测到孤立的表格，对这种情况，拒绝命令的执行。

根据一些经验性规则，对操作的先后顺序进行调整，以达到优化的目的，如要根据以下三条规则：

规则 1：优先执行 select 操作

规则 2：在 union 和 join 操作下增加 project 操作

规则 3：聚合对同一个表的 select 或 project 操作

基于以上三条，主要实现对查询树的以下调整操作：

1、 分离一元操作和二元操作

2、 有需要的话，对同一关系上的一元操作进行合并

3、 交换一元操作和二元操作，使部分一元操作优先执行

4、 调整二元操作的执行顺序

2.5 数据本地化

数据本地化是指对每一个站点进行数据管理，具体为根据分片信息，将数据定为到分片。另外我们可以进行一些局部查询优化，例如用分片组成的子树对分片进行剪枝。

2.6 全局查询优化

我们采用集中式的查询策略，即查询策略在 Control Site（运行 DDBMS 即 Client GUI 的机器）上进行，因此我们的全局查询优化也是集中式的。

在我们的系统中，我们用 Join 图检查合法性，即对所有类似 R.Column op Value 这样的条件，查询 GDD，筛选出所有相关的 Relation 和 Site，并根据所有出现过的 Column 名进一步筛选 Site 生成初等查询树，然后通过枚举找出最优的 Join 顺序并得到最终的查询树。

例如，对于最简单的两个表的 Join 操作，我们先检查两个表是否在一个站点，如果是在一个站点，我们不需要进行数据传输；否则比较两个表的大小并把规模较小的表通过网络传输到另一个站点。

第二部分　技术实现

1. 系统架构

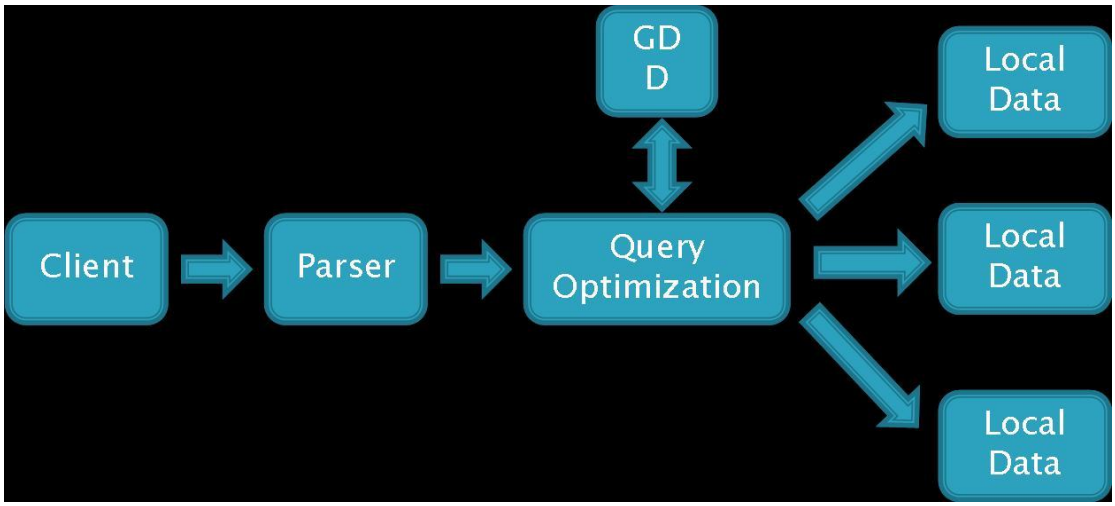


图 3：DBBMS 系统架构

如上图所示，整个分布式数据库管理系统分为 5 个功能模块，分别为 Client，Parser，GDD，Query Optimization 和 Local Data。

用户启动客户端 Client GUI 并传送控制命令，接着 Parser 解析用户命令，如果是查询命令则将查询信息送至 Query Optimization，Query Optimization 利用 GDD 中的站点和分片等信息计算出查询策略并送至 Local Data 模块，Local Data 模块对子查询命令做出响应并按策略进行数据传送；如果是站点初始化命令，则将信息送至 GDD 并对 Local Data 发出子初始化命令。

2. Parser

我们要解析的命令在表格 1 中已经列出，因此 Parser 这个模块实现起来方式很多，我们采用的是最简单的方式，即命令行的字符串分隔，具体如下：

对于类似 DEFINESITE 这样参数列表十分明显的命令，我们可以直接用空格作为字符串分隔符用 Java String 的 split 函数进行分隔从而直接解析出站点信息；

对于类似 CREATETABLE 这样参数列表比较复杂的命令，我们可以先用空格作为字符串分隔符用 split 进行分隔得到表的名字，接着对表的列描述信息进行进一步解析（我们用的是逗号分隔符），这样下去我们可以得到所有信息并存储在指定的数据结构中。

3. GDD

对查询做优化，首先必须知道查询涉及到的表是如何被划分的，每个片段的边界条件。同时为了最小化传输量，估计执行树中间结果的大小，这时候还需要提前知道每个表格的小片段的大小。

对于底层执行，需要知道每个片段所在的站点地址。

GDD 就是用来维护上面提到的这些信息的。

在我们 DDB 的实现中，GDD 可以算是系统的核心，优化模块是建立在 GDD 的基础之上的。

GDD 维护一系列 Table 的信息。分布式数据库需要对 Table 进行划分，划分分为 2 种：水平划分和垂直划分，混合划分可以表示成水平和垂直划分的组合。直观的看，水平划分就是将表格横向切割成若干子表格，垂直划分则是纵向。因此划分表格可以用一个树状结构来表示。树的每个节点是一个子表格，子表格的完整信息包括表格名称，列信息，列边界条件（ID>30000，nation=”USA”这类的列约束条件），表格条目数量，表格划分类型。如果划分类型为“无”，那么这个节点是划分树的叶子节点。如果是水平划分，那么这个节点有若干个子节点，每个子节点是对应 Table 的水平划分片段。垂直划分同理。

根据 GDD 维护的信息，给定若干个列的约束条件，可以找出所有和这些列约束条件相关的子表格（由于每个子表格结构都记录了其列约束，只需要对相同的列进行区间判断就行了）。

因此，给定一条 insert 语句，GDD 可以返回这条 insert 语句应该具体应该被插入到哪个（些）子表格。

进一步，给定对单个表格的 select query 可以找出这条 select 查询相关的子表格及其子站点，这是查询优化的基础。

4. 查询分解

一条典型的 SQL Query 的结构如下：

```
Select A.x, B.y, C.z
from A, B, C
where A.c1 = B.c2 and B.c3 = C.c4 and
      A.c2 > XXX and B.c1 <= YYY and C.c2 = “ZZZ”
```

由 parser 得到的这条语句分为 3 个片段：

1、 select 片段

2、 from 片段

3、 where 片段

为了优化，我们将 where 片段再进行分类

形如 A.c1 = B.c2 这样 列 op 列 的条件称为 join 条件

形如 A.c2 > XXX 这样 列 op 值的条件称为 select 条件

这样，一条 SQL Query 被分解成 4 个片段，其中：

select 片段对应查询树最顶端的 projection；

from 片段对应查询涉及的表格；

join 条件描述参与查询的表如何作 join；

select 条件约束参与 join 的表在 join 之前如何被筛选。

将查询分解后就可以调用优化算法进行优化了。

5. 数据本地化

我们是通过客户端将数据传向各个服务器的。分为如下步骤：

1. 客户端就当前数据库传输到 Control Site；
2. Control Site 判断每条数据应该属于哪个站点；
3. 通过 Control Site 把该数据传到指定站点，并存入指定数据库中。

6. 全局查询优化

全局查询优化算法是 DDB 的核心算法，在设计之初我们就像提出一种算法能够通用的应对所有查询，事实证明这种算法是存在的。

优化算法分描述如下：

1、 根据查询的 select 段获得查询树顶端 projection 的内容；

2、 如果 join 字段为空，那么这是一条对单个表的 select，通过 GDD 可以将这条 select 分解成子表格的 select 结果再做 union(水平划分)/join(垂直划分)。
将这个 select 结果返回就是查询树。

3、 join 字段不为空，说明查询涉及到多表 join，那么：
   a) 根据 select 字段的表格列约束条件和 join 字段的关系传递，统计出参与 join 的所有表格的所有列的约束条件。
   b) 根据这些约束条件，通过 GDD 将表格分解成和约束相关的子表格。
   c) 枚举所有子表格的 join 集合，例如：通过 b 我们发现 A 表的相关子表格为 A1,A3，B 表的相关子表格为 B1,B2,B3，C 表的相关子表格 C1,C2，那么所有的 join 集合为 (A1B1C1), (A1B1C2), (A1B2C1), (A1B2C2), (A1B3C1), (A1B3C2), (A3B1C1), (A3B1CC), (A3B2C1), (A3B2C2), (A3B3C1), (A3B3C2)。
   d) 通过判断 join 集合中子表格约束列的区间，筛去那些 join 结果肯定为空的 join 集合。
   e) 将剩下的 join 集合生成子执行树，生成方式是首先对子片段做 select，然后枚举 join 的顺序，估计各种顺序的 join 代价，选择最小代价的 join 顺序（这个就是 R*算法）。
   f) 将所有的 join 集合生成的子执行树 union 起来，放在顶端 projection 之下，返回查询树。

查询树不能直接交给执行模块，需要事先去重。因为查询树有大量的重复节点，例如子表格 A1 可能被 select 很多次，而这些结果都是一样的，在执行的过程只需要被执行一次就可以了。因此，我们需要对树节点进行判重编号，拥有相同编号的执行树节点的执行结果一定相同，这样可以极大地减少执行段的重复计算。

由于将 union 提到了顶层，因此表格与表格 join 被分解成了子表格与子表格 join 再 union，因此这种算法最大化了剪枝。但是，如果约束条件不多的情况下，这样会生成大量的子节点，从上面的例子看到仅仅是 2，3，2 个子片段的情况就有 2*3*2=12 个子节点，如果是 4，4，4，4，那么将会产生 256 个子节点！

在实际的实现中，我们采用了简化版的查询优化，步骤为：

1、 根据查询的 select 段获得查询树顶端 projection 的内容；

2、 如果 join 字段为空，那么这是一条对单个表的 select，通过 GDD 可以将这条 select 分解成子表格的 select 结果再做 union(水平划分)/join(垂直划分)。
将这个 select 结果返回就是查询树。

3、 如果 join 不为空，那么枚举大表 join 的序列。
通过 GDD，和 select 字段的约束，将大表分解成相关子表。
模拟 join 的过程，如果是 2 个表做 join，那么分解成子表 join 并剪枝；
如果是表和 join 后的表做 join，那么直接将表和中间表做 join。
模拟过程中计算每一步的代价，最后选择最小代价的 join 序列，得到执行树，返回。

这种算法可以一定程度上的剪掉多余的枝。如果参与 join 的表格多起来的话，算法不会将所有的 join 都分解到子表格，而是只分解一部分，剩下的做整体 join，通过估价函数求相对的最小值，这样在节点个数和剪枝力度之间做了一个权衡。

事实证明，这样的权衡的效果还是不错的。

7. 传输策略

这是本次实验比较重点考察的部分。首先我们会制定一个总体策略，总体策略制定的过程如下：

1. 首先是根据查询树计算出本次有多少数据需要处理
2. 统计每个站点上各自存储有多少数据
3. 找到存储数据最多的站点，其他站点主要往该站点上转移

通过上面的步骤就制定出了一个总体的策略，然而实际中这样简单处理得到的结果通常不是最优的。一般造成偏差的原因是有 JOIN 操作，JOIN 操作可能会使两个原本很大的数据变成一组很小的数据，因此程序对 JOIN 操作进行了特殊的处理。主要是对 JOIN 后的结果进行了简单的预估，之后对 JOIN 操作进行了各种转移的判断，比如将数量量较小的数据传到数据量较大的数据，JOIN 完成后再向主站点转移。这样通过比较各种情况找到最优解，进而调整该节点的策略。

当然还有一些优化，比如可以将数据分成两组往两个站点上转移，这样做也适用于一些情况，但是在实验中似乎没有出现。

8. 多线程并行处理

我们在得到查询树以后，会发现它的很多分支所需要的数据并不重叠的，而且位于不同的站点上，因此我们可以进行并行化处理。并行化处理主要思想是检查当前树是是否存在可以计算的节点（可以计算的充要条件是该节点所有的孩子节点的计算均以完成），如果可以，就新创建一个线程计算该节点，这样就可以达到并行计算的要求。

当一个线程运算结束后，它会给控制站点发送一个完成信息，这样控制站点就可以重新扫描是否存在一个新的可以计算的节点。

实现并行化后系统处理查询的速度大约提升了 20%左右。

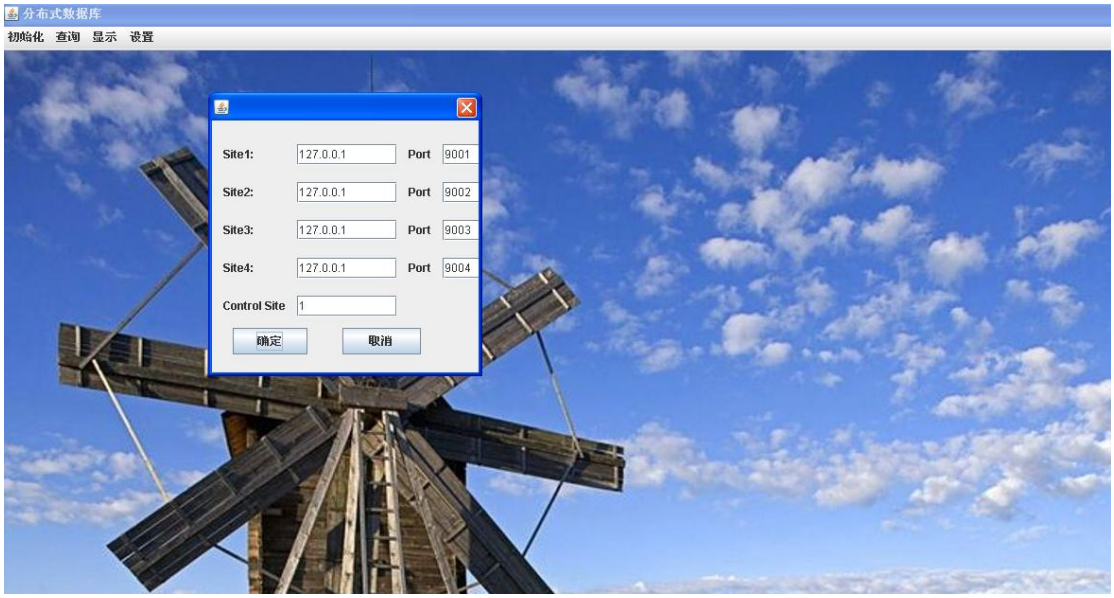
9. 客户端

我们用 JFrame 制作分布式数据库管理系统，即客户端 GUI。具体内容见功能演示。

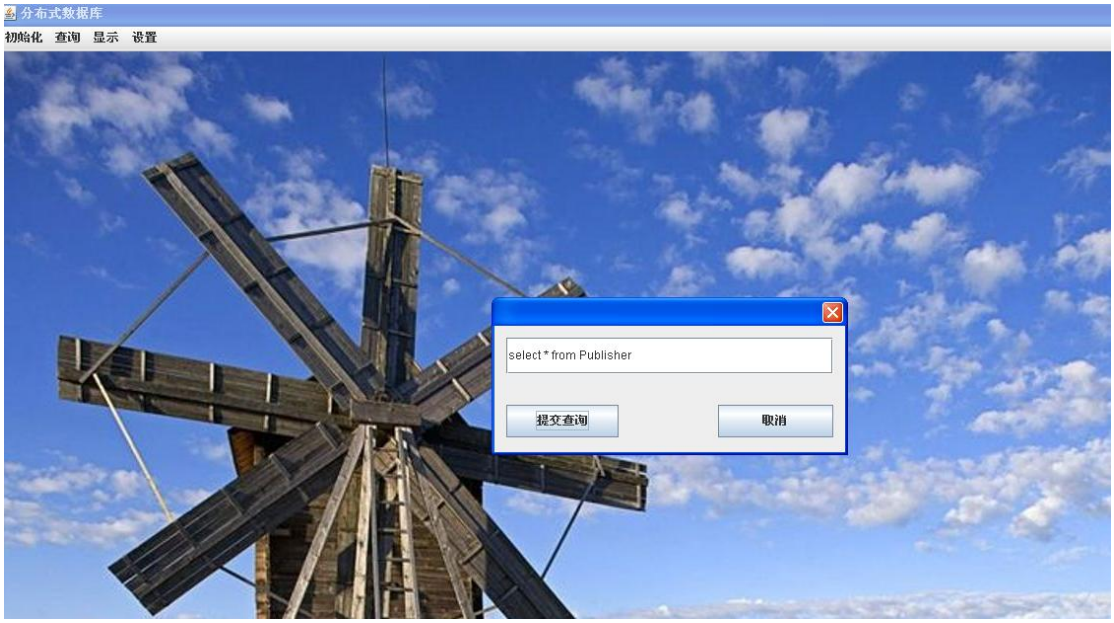
第三部分　功能演示

1 初始化

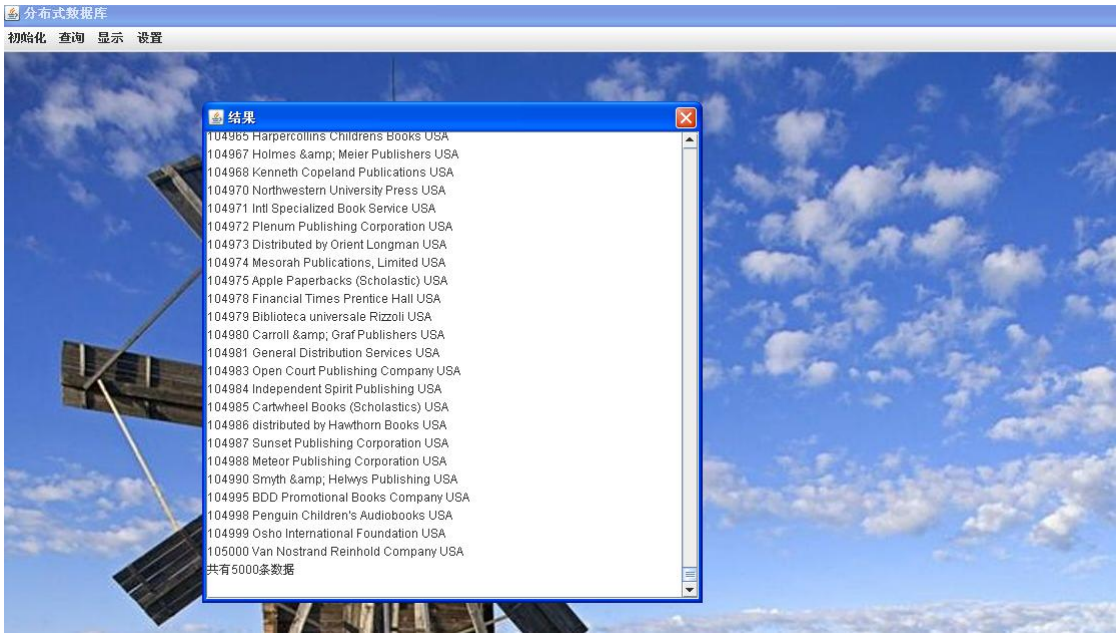
通过菜单栏设置好初始划分，初始站点即可。可以通过状态栏查询每个站点是否已经正常工作。



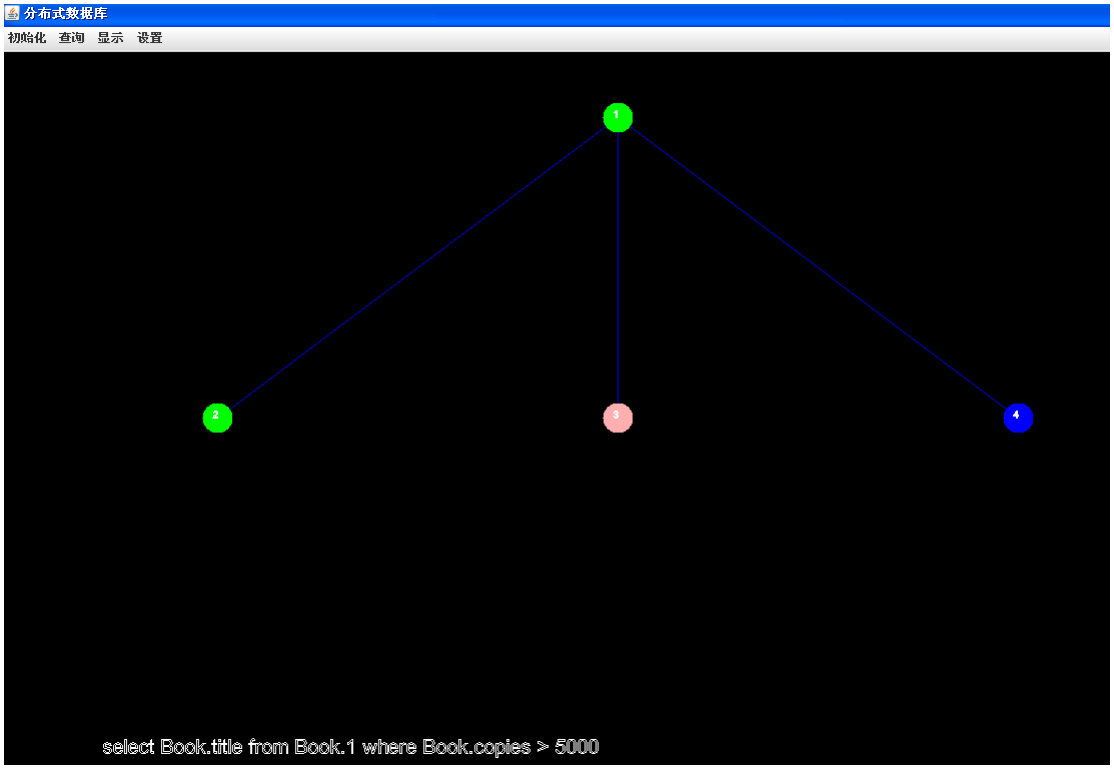
2 查询与显示


分布式数据库
初始化 查询 显示 设置
select * from Publisher
提交查询
取消


分布式数据库
初始化 查询 显示 设置
选择一个选项
查询完成！总共用时1秒,服务器用时1秒，数据传输总量86668Bytes,是否显示结果
是(Y)
否(N)
取消



3 显示优化树



4 传输控制

我们添加了传输控制选项，即是否对传输数据进行压缩，这是考虑在不同网络环境下提升系统性能的结果。在网络延时较小，带宽较大的网络环境（例如局域网）中，只要传输数据不是特别大，我们并不对数据进行压缩而是直接传输；而网络环境较差时，我们对传输数

据进行压缩，期望压缩带来的时间损耗小于数据差额的网络传输时间，以达到提升数据库性能的目的。

具体的压缩算法是对传输文本采用 huffman 编码的方式进行压缩，效果还是不错的。另外针对本次数据集基本只出现了数字和字母的特点进行了一下特殊的优化，当然这部分本身并不是实验要求，只是简单的实现了一下。

第四部分　分工总结

1. 黄刚总结

DDB 我负责的部分是 GDD 和查询优化，即整个 DDB 结构的中间层，上面是 parser 解析的查询，下面则是底层执行模块。我的工作便是输入 parse 好的查询，规划出详细的执行计划，即执行树。为了实现查询优化，全局数据信息是必不可少的，因此我还实现了 GDD，在我们的系统中查询优化和 GDD 属于同一个模块。

全局数据字典，GDD，主要负责维护表格划分情况和表格大小这些信息。在实现中我没有考虑到要持久化 GDD，只将 GDD 存在了内存中，导致了底层重启之后将会丢失 GDD 信息，影响整个系统的可维护性，这是我的失误。

对于查询优化，最开始我希望设计一种通用的算法，能够优美的处理各种类型的查询和所有的划分结构。但是在实际的编程中遇到了各种各样的问题，于是最后混合划分的支持并不完全。我对查询优化的理解就是提 union。通过将 union 操作提到执行树的最顶端，将多个表的 join 分解成他们子片段的 join 的并，从而通过子片段的划分条件，预测那些 join 结果为空的子片段 join 节点，剪枝。

具体的细节就是，先通过查询条件找出所有表格的相关片段，然后枚举所有可能的 join 顺序，将这些片段 join 后，union 成查询结果。这种算法的好处是查询树可以一步生成，由于将 union 提到了最顶，因此最大化了剪枝。但是这样的缺点就是产生的子节点非常多，例如测试集中，4 个表不做任何条件约束的 join，在这种算法下可能产生 100-200 个 union 子节点（union 子节点数为 4 个表的划分数的乘积），对执行速度的影响非常大。

因此最终我采用了折中的方法，首先枚举大表 join 的顺序，然后在固定大表 join 顺序的情况下，尽可能的剪枝优化。这样虽然不能将所有可能的剪枝都剪掉，但能有效减少执行树节点的个数，从执行效率来看并不比提 union 的方法差。

总的来说，这次 DDB 我的优化模块写的并不算特别优秀，由于只有一周的时间，最后没有进行多少测试就直接投入使用了，导致在最后检查的时候出现了一些 bug，不过并不算特别严重，很快就被我修正了。

这次实验的收获非常大，通过实现 GDD 和优化模块，对分布式数据库系统的执行原理有了深刻的认识，同时也看到了理论和实际的一些不同，这些收获对今后的学习有着很大的帮助。

最后，作为组长我没有尽到组长的责任，对整体进度的把握和人员分工存在着相当多的不成熟。如果能早一点规划这次实验，相信结果会更加完美。

2. 汪汀总结

DDB 大实验是一个比较重的任务，我主要负责的是底层平台，通讯协议及服务器和客户端的开发，简单来说就是在已经有 GDD 和 QueryTree 的情况下让系统正确运行，并且决

定传输策略。通讯协议前面已经介绍了，我设计的并不是很复杂，因为我觉得这个不是实验的重点。由于本次实验比较强调传输量，因此我主要在这里做了一些优化。从总的表现来看还不错。另外为了提高程序整体的运行速度我将程序进行了多线程并行化处理，收到了一定的成效。

大实验是三个人一组的，本次我们合作的很愉快。我觉得我们的分工是比较合理的，相互之间的交集较少，主需要约定简单的接口即可。这样三个人可以同时进行自己的工作，不会因为别人工作没有完成而卡住，从而达到最高效率。事实上，从实验开始到系统能正确运行大约只花了一周的时间，相对于这么浩大的一个工程，我认为我们完成的还是很迅速很出色的。

通过这次实验，我对分布式数据库系统也有了更深刻的认识。本次实验中我们做了不少工作，但还远远不够，实际的系统会比我们实验所设计的系统复杂的多，他要求系统具有更高的鲁棒性，能适应各种情况，今后，我在设计系统框架的时候会更多的考虑这方面要求。

当然遗憾也是存在的。比如没有实现混合划分。本来开始时我们决定实现这一点的，可惜后来因为时间比较紧张而放弃了这个尝试。不过大体上，我还是比较满意的。

3. 宋顺强总结

按照 Ivan Sutherland 在 *Technology and Courage* 一文中的说法，这门课程应该是一个很好的“启动工程”，对我们这样刚入学的研究生算是很好的 warm-up。

选课之际，作为一个本科非计算机系的研究生，我希望通过这门课程锻炼一下 coding 能力和做系统的感觉；最后由于另两名组员 coding 能力远超过我加上我没有积极承担任务，我在这次大作业中贡献十分有限，惭愧之余对另两名组员表示感谢。

不管怎样，我还是通过这次大作业感受到搭建一个稳定可靠的系统并不是一件容易的事情，总是会有莫名其妙的 bug 困扰着你并消磨你的热情；另一个体会是，永远不要把工作往后推，因为一到期末 deadline 很多，你并不会像自己预期那样高效，连续的熬夜也不能迅速打造完美的系统。

4. 分工情况

姓名	任务
黄刚	GDD，查询分解，全局查询优化
汪汀	通讯协议，底层平台，服务器，客户端
宋顺强	Parser