TBase 查询处理子系统的设计与实现

学　　院　计算机科学与技术

专　　业　计算机科学与技术

年　　级　2007 级

姓　　名　杨亚军

指导教师　张坤龙

2011 年 6 月 10 日

摘　　要

如今，数据库管理系统的地位越来越重要，数据库的教学也受到了人们的广泛重视，但是能够用于教学研究的 DBMS 却很少，尤其是在. NET 平台上更是一个空白。因此，我们设计和实现了 TBase 来满足这一需求。TBase 结构与一个商业数据库相同，包含：查询处理子系统，存储管理子系统和事务管理子系统。

论文介绍了查询处理子系统的设计和实现。查询处理子系统包括元数据管理器，语法分析器，优化器和执行器。元数据管理器保存数据库的表、字段、视图定义、索引和统计信息，为类型检查和执行计划开销评估等提供信息。语法分析通过词法分析和语法分析，将 SQL 语句转换为语法树和数据表示的内部结构，然后利用元数据管理器（目录）进行类型检查。优化器将 SQL 的内部表示的结构利用一种启发式的规则转换为一个比较优的执行计划。执行器利用迭代器执行这个计划。

论文对查询处理子系统进行了测试，通过输出查询执行的中间结果：包括语法树，类型检查的结果和执行计划树测试了系统是正确的。

关键字：查询处理子系统；语法分析；优化器；执行器

ABSTRACT

Now database management systems are more and more important, and the teaching of database management system attracts lots of people's attention. However, the DBMS which can be used for teaching and research is rarely, especially in the .NET platform, it is a blank. Therefore we design and implement the TBase to meet the demand. TBase has the same structure as a commercial database, including: query processing subsystem, memory management subsystem and the transaction management subsystem.

This paper describes the design and implementation of the query processing subsystem. Query processing subsystem includes Metadata manager , Parsing, Optimizer and Executor. Metadata manager record the tables, fields, view definitions, indexes and statistical information for type checking , Estimating costs and so on. Parsing including lexical analysis and syntax analysis convert the SQL statement into the syntax tree and data representation of the internal structure, and use the Metadata Management (catalog) for type checking. Optimizer converts the internal structure to a more optimal execution plan using a heuristic rule. Then executor performs the execution plan using iterator.

The paper tested the query processing subsystem by outputting the intermediate results of query execution: including syntax tree, the result of type checking and execution plan tree.

Key words: query processing subsystem; Metadata manager; Parsing; Optimizer; Executor

目　　录

外文资料

中文译文

致谢

第一章　　绪论

1.1　DBMS 简介

数据库管理系统（DBMS）是一种操纵和管理数据库的大型软件，用于建立、使用和维护数据库，是现代计算机环境中的一个核心部分。随着计算机的飞速发展，以及计算机系统在各个行业中的广泛应用，数据库管理系统也得到了越来越广泛的应用[6]。

如今，最成熟的数据库管理系统是关系数据库管理系统，它作为基础设施应用的骨干，包括银行业务、航空公司机票预订、医疗信息记录、人力资源管理、工资管理、客户关系管理以及供应链管理等等[1]。

一个典型的数据库系统包含三个主要的子系统：存储管理子系统，事务管理子系统和查询处理子系统[1][6]。其中存储管理子系统包括磁盘管理，缓冲区管理，存取方法。事务管理包括事务管理器，锁管理器和恢复管理器。查询处理包括元数据管理，语法分析，优化器，执行器。如图 1-1 为一个数据库管理系统的结构图。

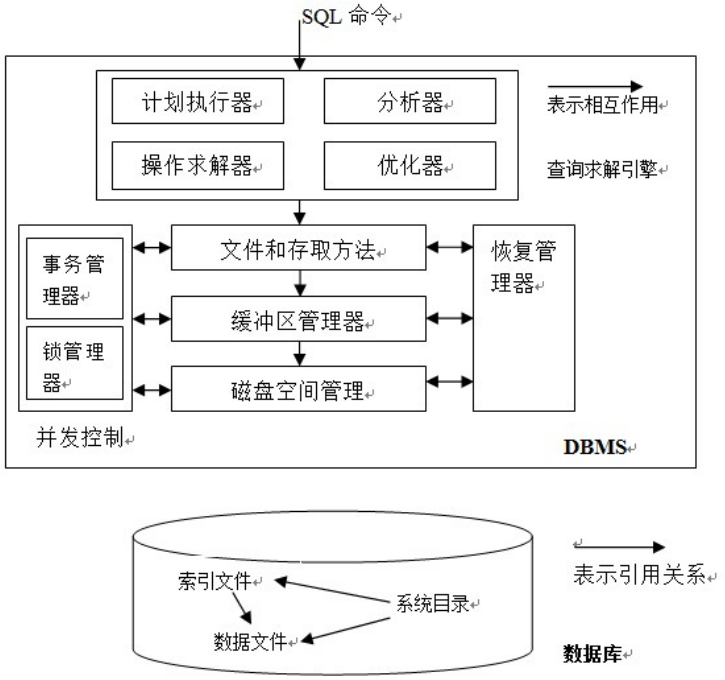

图 1-1　　DBMS 的体系结构

当用户提出一个查询，经过语法分析之后的查询被送到查询优化器。查询优化器借助元数据管理器提供的信息为计算查询生成高效的执行计划。执行计划是计算查询的蓝图，经常表示为关系操作树。关系操作的实现代码位于文件和存取

方法层之上，这一层支持文件存储结构。在 DBMS 中，文件是页或者记录的集合。这一层通常支持堆文件或者索引。文件和存取方法层代码位于缓冲区管理器之上，缓冲区管理器的责任是把页从磁盘取入主存，以响应读请求。DBMS 的最底层处理存储着数据的磁盘空间管理，其上层通过这层分配、回收、读和写页面。该层称为磁盘空间管理器。DBMS 通过仔细调度用户请求和维护记录数据库所有变化的日志，来支持并发控制和故障恢复。与并发控制和故障恢复有关的 DBMS 构件包括事务管理程序、锁管理器和恢复管理程序。事务管理程序确保事务按照一个合适的加锁协议来请求和释放锁，并调度事务的执行。锁管理器跟踪对锁的请求，当数据库对象可用时，在该对象上授权加锁。恢复管理程序负责维护日志，在系统崩溃后把系统恢复到一致状态[6]。

1.2 TBase 简介

TBase 是一个用于教学的，基于.NET 平台的关系数据库管理系统。它所支持的功能相对少一点，但是具备了一个商业数据库的基本模块。

1.2.1 TBase 的功能

TBase 支持以下功能：

1. 在一个表的单个字段上创建索引。索引的类型包括 B+树索引和哈希索引。索引的相关信息记录在索引元数据（索引目录）中。
2. 创建视图，视图的信息保存在视图元数据中。
3. 创建一张表。可以定义表的名字和表的字段。表的字段的类型只支持 int 和 varchar。
4. 单个表或者多个表上的查询。
5. 可以对表的单个字段进行更新。
6. 可以向表中插入记录，必须指定表的概念模式。
7. 可以从表中删除记录。

1.2.2 TBase 模块及其作用

TBase 由查询处理，存储管理，事务管理三个子系统组成。基本具备一个商业数据库系统的核心组成部分。三个子系统的作用如下：

1. 查询处理子系统[1][16]。查询处理子系统接收一个用户输入的 SQL 语句。经过词法分析和语法分析之后产生语法树，并产生相应的内部表示结构。然后对该内部表示的结构进行类型检查。类型检查无误后，传送到优化器[2][3][5]，生成一个执行计划，然后让执行器去执行这个计划，产生相应的结果。

2. 事务管理子系统。包括事务的并发控制，锁管理器和恢复管理。主要作用是实现多个线程或者进程并发访问数据库，并且如果数据库发生崩溃，则通过日志来恢复数据库。

3. 存储管理子系统。包括磁盘管理，缓冲池管理和存取方法管理等。提供了数据库底层支持。为上层的事务和查询提供数据和存取方法。

1.3 TBase 开发环境

TBase 的开发环境如表 1-1 所示。

表 1-1 TBase 的开发环境

名称描述	具体内容
开发语言	C#
开发平台	.Net Framework 3.5
IDE 工具	Visual Studio 2008

1.4 TBase 查询处理概述

TBase 的查询处理子系统的主要功能是将一个用户输入的 SQL 语句转换为可以执行的执行计划，然后交给执行器，利用下层提供的接口，产生出执行结果。我们将在接下来的章节对此做详细的介绍。

1.5 论文组织结构

本论文按如下结构组织，第一章为绪论，主要介绍 DBMS 和 TBase 的一些基本知识。第二章为 TBase 查询处理的模块及整体设计，比较详细的介绍了 TBase 查询处理的组成模块，执行流程，对外提供的接口和类图。第三章为元数据管理器，介绍 TBase 中的目录管理。包括目录的概念模式，详细设计和测试。第四章为语法分析器，详细阐述了语法分析器的概要设计和具体实现，以及测试。第五章为优化器，介绍了优化器的概要设计，具体实现和测试。第六章为执行器，内容包括执行器的概要设计，具体实现和测试。第七章为结论，概括了一下本论文的内容，并提出了未来的工作。

第二章　　TBase 查询处理的模块及整体设计

2.1　TBase 查询处理的模块

TBase 的查询处理的组成模块如图 2-1 所示。

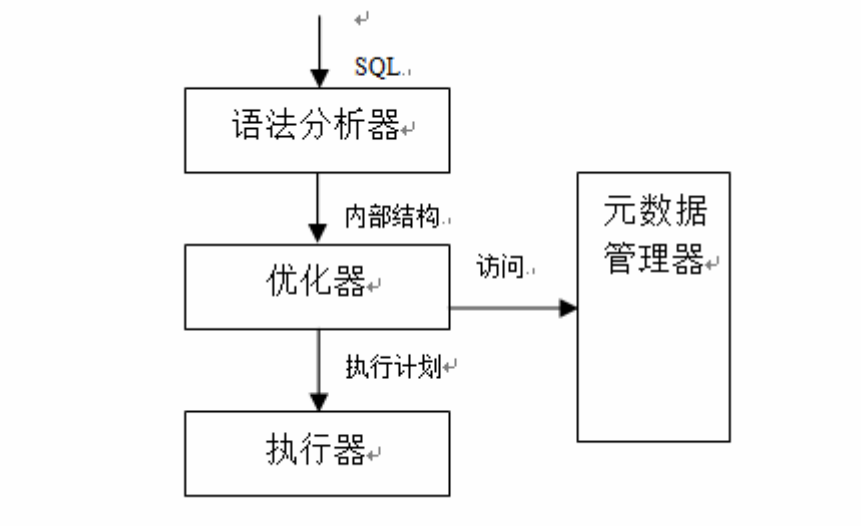

图 2-1　　查询处理的模块

查询处理包含的模块以及各模块的功能如下：

1. 元数据管理器，即目录。元数据管理器的作用是保存数据库中所有表的信息，字段信息，索引信息，视图信息和统计信息。目录本身也是一张表。元数据管理器包含以下几张表：①表目录。表目录中记录数据库中表的名字和该表的记录的长度。②字段目录。字段目录中保存了数据库中所有的表的所有字段的信息，每一个字段一条记录，保存了字段所属的表的名字，字段的名字，字段的类型，字段的长度和字段在该表中的偏移量。③索引目录。每个索引一条记录。保存了索引的名字，索引所在的表的名字和字段的名字。④视图目录。保存了视图的名字和视图的定义。⑤统计信息管理。统计信息不是表，也不会实物化到磁盘中，但是却是元数据管理中必不可少的一部分。因为统计信息记录了每个表的页数，每个字段上不同值的个数和记录的数目。这个为优化器评估执行计划的代价开销提供了必要的信息[15]。

2. 语法分析器。包含词法分析，语法分析，类型检查[1]。词法分析将 SQL 语句解析为 token 数组。语法分析利用 LL(1)对 token 数组进行分析，产生相应的语法树，并转换为数据库表示的内部结构。类型检查调用元数据管理器的信息，判断该查询的表和字段是否存在，连接条件是否兼容等等。

3. 优化器。利用一种启发式的规则[13]，只考虑左深计划产生一个比较优的执行计划。

4. 执行器[6]。执行器利用迭代器执行该计划，输出结果。

2.2 TBase 查询处理的执行流程

TBase 的查询执行流程如图 2-2 所示。

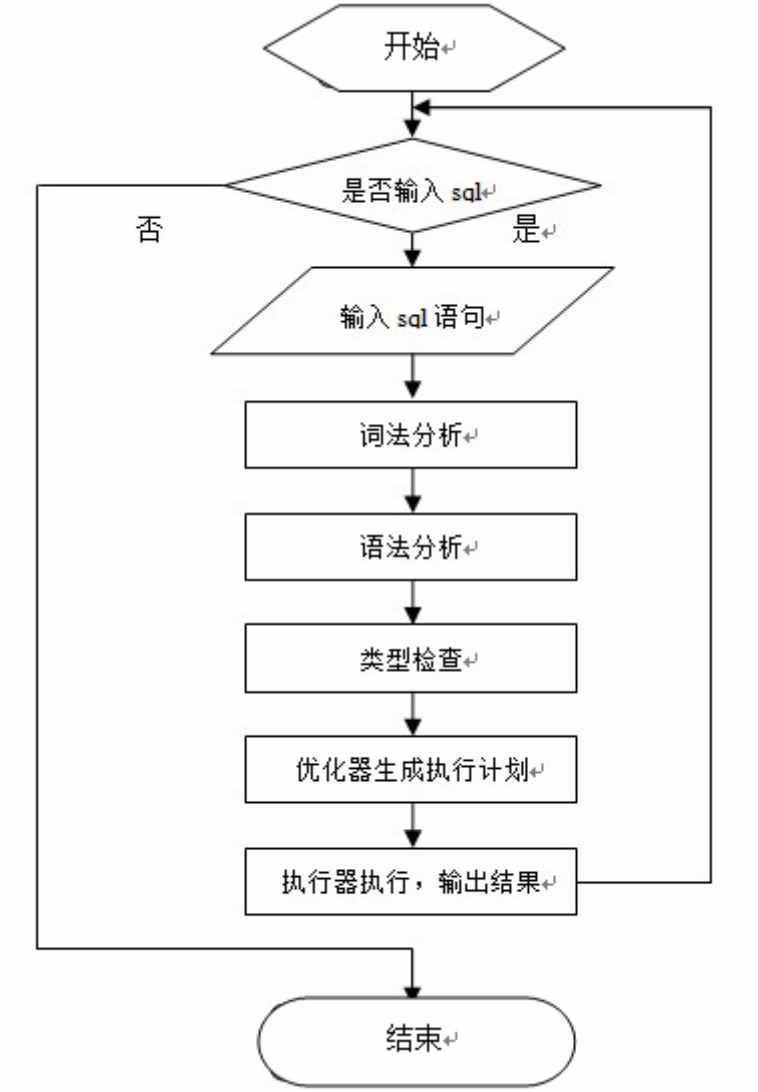

图 2-2　　查询处理的程序流程图

查询处理子系统接收一个用户输入的 SQL 语句。该 SQL 语句为 string 类型。首先经过词法分析将该 SQL 语句解析为一些 token 的集合。产生 token 的同时伴随着词法的检查，如果有词法错误的话会在这个阶段检查出来。将词法检查无误的 token 数组传递给语法分析。语法分析采用 LL(1)来进行语法分析（即自顶向下，从左到右处理输入的 token，每次只向前看一个 token），语法分析检查出语法错误，并向用户反馈错误信息。如果语法无误的话就生成语法分析树，并且将该 SQL 语句转换为数据库内部表示的结构。然后将该结构传递给类型检查，类

型检查通过调用元数据管理器，来判断该查询结构涉及到的表是否存在数据库中，投影的字段是否存在 FROM 子句的表中，以及 WHERE 条件中的字段是否包含在 FROM 子句的表中，并且检查条件表达式中两边的类型是否兼容，连接条件是否兼容。类型检查无误后的内部结构传递给优化器用来生成执行计划。优化器只考虑左深计划，通过一种贪心策略（首先选择记录最少的表作为最左端的基表，然后挨个加入是连接结果记录最少的表）构造执行计划。构建完的执行计划是一个节点为 Plan 的树。然后将该树传递给执行器，执行器打开每个 Plan 的 Scan，Scan 为一个迭代器，实现了三个接口，open()，next()，close()，通过调用根节点的 next()递归调用下层的 next()来返回执行结果。Scan 的实现通过下层的访问路径来提供，至于访问路径的讨论就不包括在本论文的范围内了。

2.3 TBase 查询处理整体设计

一. 查询处理对外接口设计

查询处理对外提供的接口如下：

第一，优化器中 Planner 类对外提供了两个接口调用。第一个是 public Plan createQueryPlan(string qry, Transaction tx) ，它的主要作用是接收一个查询的 SQL 语句，然后返回一个执行计划。传入参数 qry 为表示查询的 SQL 语句，tx 为执行该操作的事务。第二个是 public int executeUpdate(string cmd, Transaction tx)，它的主要作为是接收一个表示更新数据库的操作的 SQL 语句，然后执行，返回改变的记录的数目。

第二，优化器中 Plan 类对外提供了两个接口。第一个是 Schema schema()，它返回该计划的结果记录的概念模式。第二个是 Scan open()，它打开一个迭代器，用来取该计划产生的记录。

第三，执行器中 Scan 类对外提供了 7 个接口调用。第一个是 void beforeFirst()，它将迭代器的游标至于初始位置。第二个是 bool next()，它用来判断是否还有记录，有的话，将迭代器游标移动到下一个位置。第三个是 void close()，它关闭当前 Scan 对象。第四个是 Constant getVal(string fldname)，它获得当前记录的 fldname 字段的常量值。第五个是 int getInt(string fldname)，它获得当前记录的 fldname 的整型值。第六个是 string getString(string fldname)，它获得当前记录的 fldname 的字符串值。第七个是 bool hasField(string fldname)，它判断当前记录是否有字段 fldname。

第四，语法分析器中的 ParseTree 类对外提供了一个接口：public String toTstring()，它的主要作用是返回语法树的内容，用于语法树的输出。

二. TBase 查询处理类图

TBase 查询处理设计很多类，其中一些关键类的类图如图 2-3 所示。

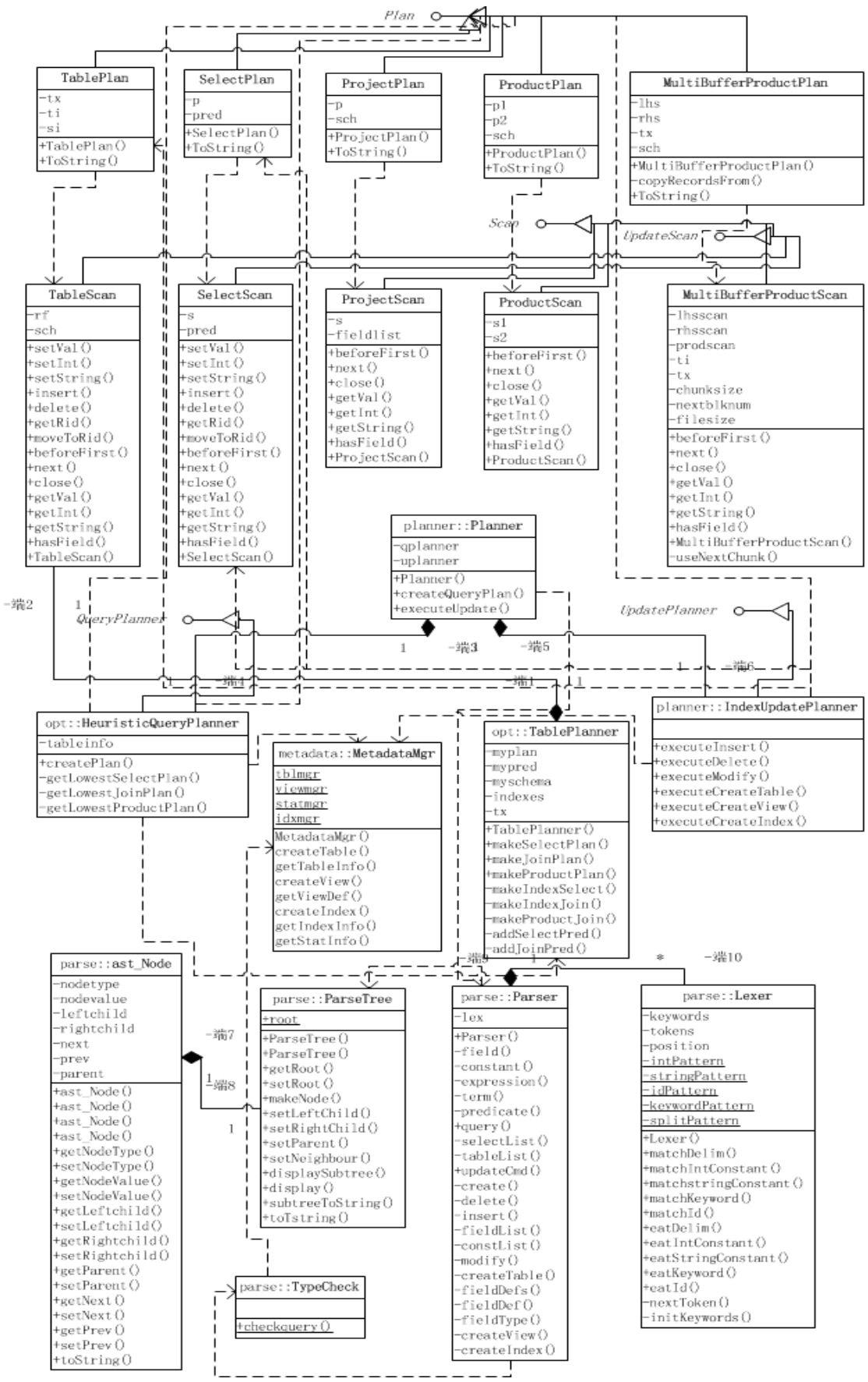

图 2-3　　TBase 查询处理部分类图

第三章　元数据管理器

3.1　元数据管理器的设计

元数据管理器需要存储数据库中所有表、字段、索引和视图的信息，而且要存储表的一些统计信息，比如记录的数目，某字段不同值的个数，页数等。统计信息需要定期的更新[1]。

针对其需求。需要建立 4 个目录和一个统计信息管理器。其中 4 个目录包括表目录，字段目录，索引目录，视图目录。每个目录本身作为一张表来存储，这样做的好处是：目录可以和其他表一样，使用 DBMS 的查询语言来查询[6]。所有用于实现和管理表的技术都可以直接用于目录。统计信息管理器通过定期的扫描数据库中的表来更新统计信息。其具体设计如下。

一．　表目录(tblcat)。表目录记录数据库所有存在的表（包括 4 个目录表），其概念模式如表 3-1 所示。其中字段 tblname 为表名，其类型为 varchar，字段 reclength 为该表中字段的长度，TBase 是采用定长记录存储的。数据库中的每一张表在 tblcat 中有一条记录。

表 3-1　　表目录概念模式

字段名	类型
tblname	varchar
reclength	int

二．　字段目录(fldcat)。字段目录记录数据库中所有表的所有字段的信息（包括 4 个目录表中的字段）。数据库中每个字段对应 fldcat 中的一条记录。其概念模式如表 3-2 所示。其中 tblname 为该字段所在的表的名字，其类型为 varchar。字段 fldname 表示该字段的名字，类型为 varchar。type 为该字段的类型（TBase 支持两种类型，int 和 varchar），其类型为 int。字段 length 为该字段的长度（占用的字节数），其类型为 int。字段 offset 为偏移量，即该字段在该记录的开始位置（按字节计算）。

表 3-2　　字段目录概念模式

字段名	类型
tblname	varchar
fldname	varchar
type	int
length	int
offset	int

三． 索引目录(idxcat)。索引目录记录数据库中所有的索引的定义信息。每个索引对应该表中的一条记录。其概念模式如表 3-3 所示。其中 indexname 为索引的名字，其类型为 varchar，tablename 为该索引建立的表的名字，其类型为 varchar，fieldname 为该索引建立的字段的名字，其类型为 varchar。

表 3-3 索引目录概念模式

字段名	类型
indexname	varchar
tablename	varchar
fieldname	varchar

四． 视图目录（viewcat)。视图目录存放数据库中所有的视图的定义。每一个视图对应该表中的一条记录。其概念模式如表 3-4 所示。其中 viewname 为视图的名字，其类型为 varchar, viewdef 为视图的定义信息（一个查询语句），其类型为 varchar。

表 3-4 视图目录概念模式

字段名	类型
viewname	varchar
viewdef	varchar

五． 统计信息管理。统计信息需要记录数据库中每个表的块数，记录的数目以及每个字段上不同值的数目。通过扫描表来统计。

3.2 元数据管理的实现

3.2.1 元数据管理器的类

元数数据管理器的命名空间为 TBase.metadata，包含 7 个类，分别是 MetadataMgr，TableMgr，IndexMgr，ViewMgr，StatMgr，IndexInfo，StatInfo。其中 MetadataMgr 封装了其余的几个 Mgr 类，对外提供访问元数据的接口。TableMgr 为表管理器，提供了对表目录和字段目录的访问。IndexMgr 为索引管理器，提供了对索引信息的访问。ViewMgr 为视图信息管理器，提供了对视图目录的访问，StatMgr 为统计信息管理器，计算数据库的统计信息。IndexInfo 为一个索引的相关信息，这些信息可以被优化器用于使用索引的开销估算，并且获得索引记录的概念模式。StatInfo 用来保存每个表的统计信息，该统计信息由

StatMgr 产生。

3.2.2 元数据管理器的实现

1. 元数据管理类（MetadataMgr）。该类如图 3-1 所示。元数据管理类对外提供调用。其内部包含了索引管理（idxmgr），统计信息管理(statmgr)，表管理（tblmgr, 表管理中包含表目录和字段目录）, 视图管理(viewmgr)。这四个都是私有静态变量。

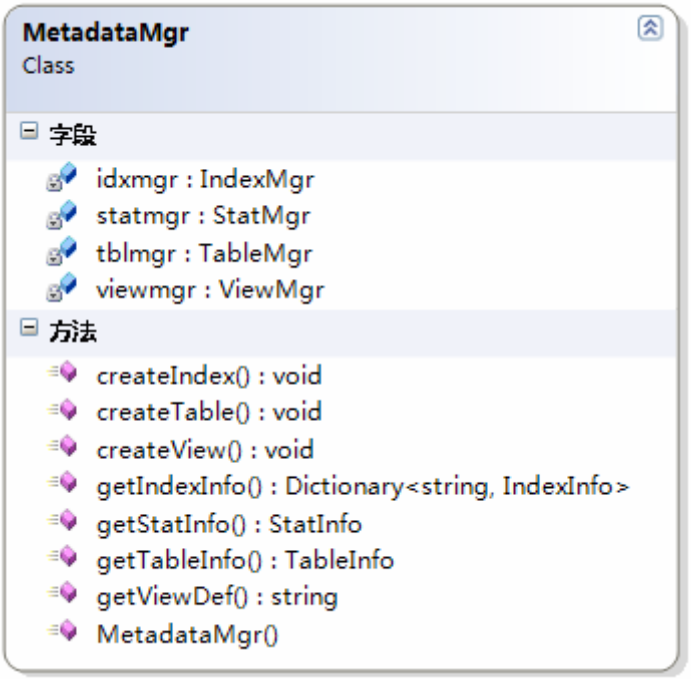

图 3-1　　元数据管理类

public void createIndex(string idxname, string tblname, string fldname, Transaction tx) ： 该方法通过调用 idxmgr 对象来创建一个索引。

public void createTable(string tblname, Schema sch, Transaction tx) ：通过调用 tblmgr 对象来创建表。

public TableInfo getTableInfo(string tblname, Transaction tx) ： 返回表的信息。该信息包括该表的概念模式，每个字段的偏移量，记录的长度和表的名字。

public void createView(string viewname, string viewdef, Transaction tx) ： 创建视图。通过调用 viewmgr 对象来完成。

public string getViewDef(string viewname, Transaction tx) ：通过调用 viewmgr 对象来返回视图的定义。

public Dictionary<string, IndexInfo> getIndexInfo(string tblname, Transaction tx) ： 通过 idxmgr 对象获得某个表上的所有的索引的信息。

public StatInfo getStatInfo(string tblname, TableInfo ti, Transaction tx) ： 通过 statmgr 对象获得某个表的统计信息。

2. 表管理器类（TableMgr）。表管理器管理表目录和字段目录的访问。该类如图 3-2 所示。

图 3-2　　表管理器类

fcatInfo 为字段目录的信息，tcatInfo 为表目录的信息，MAX_NAME 为表名和字段名的最大长度（按字节算），值为 16.

public void createTable(string tblname, Schema sch, Transaction tx) ： 创建一张表，传入参数 tblname 为表名，sch 为该表的概念模式，tx 为执行创建表操作的事务。该方法首先向表目录中插入一条记录，用来保存创建的这张表，然后向字段目录中，对该表 tblname 中的每个字段往字段目录中插入一条记录。用来保存该字段的信息。

public TableInfo getTableInfo(string tblname, Transaction tx) ：返回指定表的信息。传入参数 tblname 为表名，tx 为执行该操作的事务的引用。该方法首先访问表目录然后访问字段目录查找匹配的记录。

public TableMgr(bool isNew, Transaction tx) ： 构造函数。传入参数 isNew 表示数据库是否为新建的。该方法首先为 fcatInfo 和 tcatInfo 赋值，然后判断 isNew 的值，如果为 true 的话，表明数据库是新建的。那么就创建表目录和字段目录。

3. 索引管理器类（IndexMgr）。该类管理对索引目录的访问和与索引相关的一些操作。该类如图 3-3 所示。

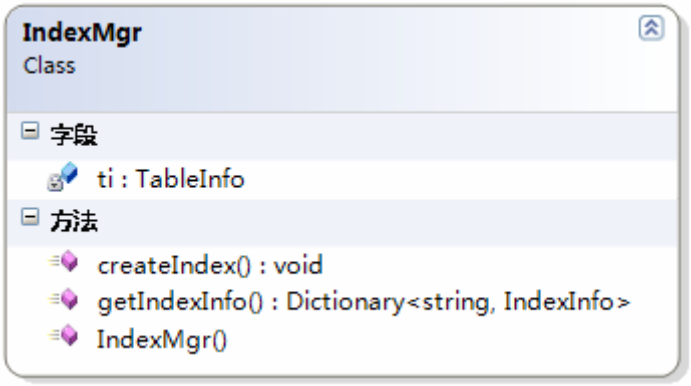

图 3-3　　索引管理器类

字段 ti 为索引目录表的信息。

public IndexMgr(bool isnew, TableMgr tblMgr, Transaction tx) 构造函数，传入参数 isnew 表示该数据库是否是新建的，tblMgr 为表管理器，tx 为执行该操作的事务。方法首先判断 isnew 的值，如果为真，就为 ti 赋值，然后创建该索引目录表。否则就从表目录中获得该索引目录表的信息赋值给 ti。

public void createIndex(string idxname, string tblname, string fldname, Transaction tx)： 创建索引，传入参数 idxname 为要创建的索引的名字，tblname 和 fldname 分别为索引所在的表和字段。该方法向索引目录中插入一条记录，保存该索引的信息。

public Dictionary<string, IndexInfo> getIndexInfo(string tblname, Transaction tx)： 获得某个表上的所有索引信息。返回的是一个数据字典，键是索引的名字，值是改索引的信息。传入参数 tblname 为表名，tx 为执行该操作的事务。该方法通过访问索引目录，查找并返回 tablename 字段为 tblname 的记录。

4. 统计信息管理器（StatMgr）。统计信息不是在磁盘中存放的，而是每次在系统启动的时候开始计算表的统计信息，并且当统计信息被访问了 100 次后，重新计算统计信息。该类如图 3-4 所示。

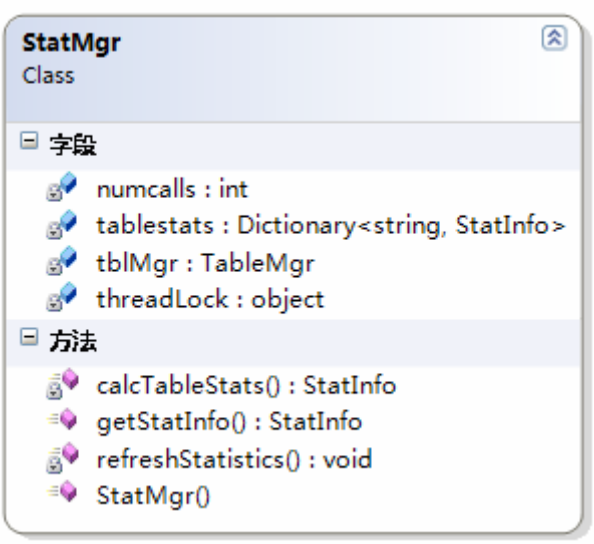

图 3-4　　统计信息类

字段 numcalls 记录该类的对象被访问了多少次。

tablestats 是个字典，键是表的名字，值是表的统计信息，表的统计信息包括该表有多少页，有多少个记录，每个字段上不同值的数目。

tblMgr 为表管理器对象

treadLock 为一个线程锁，用于并发。

public StatMgr(TableMgr tblMgr, Transaction tx) ：构造函数，传入参数 tblMgr 为表管理器类。tx 为事务。该方法首先为字段 tblMgr 赋值，然后调用 refreshStatistics 函数来产生统计信息。

public StatInfo getStatInfo(string tblname, TableInfo ti, Transaction tx) ：该方法获得指定表的统计信息。传入参数 tblname 为表名，ti 为表信息，tx 为执行该操作的事务。返回该表的统计信息 StatInfo。该方法首先进行加锁，然后判断 numcalls 的值是否大于 100，如果是的话，调用 refreshStatistics 重新生成统计信息。否则判断该表的统计信息是否存在 tablestats 中，如果存在的话，直接返回该表的值，否则的话调用 calcTableStats 计算该表的统计信息，然后将该记录插入到 tablestats 中。

private void refreshStatistics(Transaction tx) ： 更新统计信息。传入参数 tx 为执行该更新操作的事务。该方法首先从表管理器中获得表目录的信息，然后根据该信息打开表目录的 RecordFile。然后获得该文件中存放的表的表名，然后从表管理器中获得这些表的 TableInfo，然后调用 calcTableStats 计算该表的统计信息，并存入 tablestats 字典里。

private StatInfo calcTableStats(TableInfo ti, Transaction tx) 计算指定表的统计信息。传入参数 ti 为要计算的表的信息，tx 为执行该操作的事务，返回该表的统

计信息。该方法打开该表的 RecordFile，然后挨个扫描记录，来统计记录的数目和块数。

3.3 元数据管理器的测试

分为以下几个步骤来测试元数据管理器：

第一步，创建一个名为 student 的数据库，查看元数据表是否创建。结果如图 3-5 所示。可以看到四个表 fldcat.tbl,idx.tbl,tblcat.tbl 和 viewcat.tbl 都已创建。

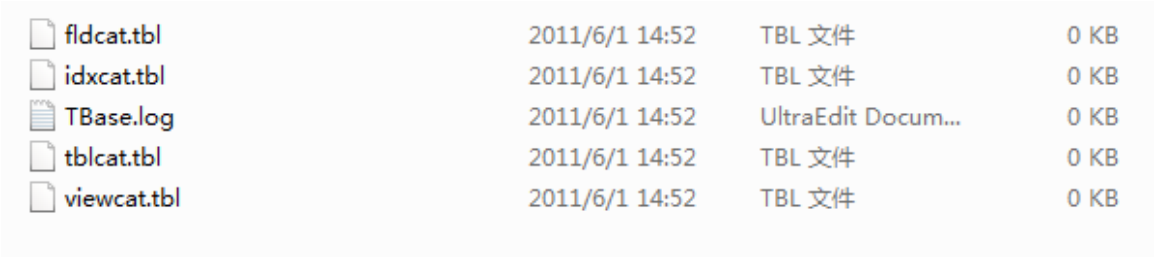

图 3-5　　student 数据库下的文件

第二步，创建一个表 S (SSNO VARCHAR(3), SNAME VARCHAR(10), STATUS INT, SCITY VARCHAR(10)) ，并且查询目录表和字段表中的内容，结果如图 3-6 和图 3-7 所示。从查询结果来看，表目录和字段目录的实现均正确无误。

显示查询结果

	tblname	reclength
1	tblcat	40
2	fldcat	84
3	viewcat	240
4	idxcat	108
5	s	62

图 3-6　　表目录查询结果

显示查询结果

	tblname	fldname	type	length	offset
1	s	ssno	12	3	0
2	s	sname	12	10	10
3	s	status	4	0	34
4	s	scity	12	10	38

图 3-7　　字段目录中 s 表字段查询结果

第三步，创建一个 s 表的 ssno 字段上的索引，并查询索引目录的内容。其结果如图 3-8 所示。标明索引目录的实现正确。

显示查询结果

	indexname	tablename	fieldname
1	indexofs	s	ssno

图 3-8　　索引目录的查询结果

第四步，创建一个视图 vs，创建语句为 CREATE VIEW vs AS SELECT ssno FROM s，并查看视图目录的内容。其结果如图 3-9 所示。证明视图目录正确。

显示查询结果

	viewname	viewdef
1	vs	select ssno from s

图 3-9　　视图目录的查询结果

经测试，元数据管理器正确无误。

第四章　　语法分析器

4.1　语法分析器的设计

语法分析器能够将一个 string 类型的 SQL 语句，解析为数据库创建执行计划需要的内部结构，要检查出存在的词法和语法错误，并且需要进行类型检查，判断 FROM 子句中的表是否存在于数据库中，SELECT 子句和 WHERE 子句中的字段是否在 FROM 子句中的表里。WHERE 子句表达式左右两端的类型是否兼容，并且应该提示错误信息。

一．　TBase 支持的语法

```
Command     = Query
              | Modify
              | Create
Query       = SELECT SelectList FROM TableList [ WHERE Predicate ]
SelectList = Field [ , SelectList ]
TableList   = TableName [ , TableList ]
Predicate   = Term [ AND Predicate ]
Term        = Expression = Expression
Expression = Field
             | Constant
Field       = FieldName
Constant    = String
             | Integer
Modify      = Insert
              | Delete
              | Update
Insert      = INSERT INTO TableName ( FieldList ) VALUES ( ConstList )
FieldList   = Field [ , FieldList ]
ConstList   = Constant [ , Constant ]
Delete      = DELETE FROM TableName [ WHERE Predicate ]
Update      = UPDATE TableName SET Field = Expression [ WHERE Predicate ]
Create      = CreateTable
              | CreateView
              | CreateIndex
CreateTable= CREATE TABLE TableName ( FieldDefs )
```

FieldDefs = FieldDef [, FieldDefs]

FieldDef = FieldName TypeDef

TypeDef = INT

| VARCHAR (Integer)

CreateView = CREATE VIEW ViewName AS Query

CreateIndex= CREATE INDEX IndexName ON TableName (Field)

二． 词法分析的设计

词法分析要通过正则表达式，将 SQL 解析为 token 数组，其中 token 可以是整数，字符串，ID 和关键字。他们的正则表达式分别如下：

intPattern ^[-+]?(0|[1-9]\d*)$

stringPattern ^'[^']*'$

idPattern ^[a-zA-Z_]\w*$

keywordPattern ^[a-zA-Z]*$

splitPattern \s{1}|(,){1}|(=){1}|(\(){1}|(\)){1}|('[^']*'){1}

其中 intPattern 表示一个整数，stringPattern 表示一个字符串值，idPattern 表示 TBase 的标识符，keywordPattern 表示 TBase 的关键字，这里为了简单，并没有严格的去匹配 TBase 支持的所有关键字。splitPattern 表示分隔符。对输入的 SQL 语句应用这些正则表达式去匹配，将匹配的结果保存在 token 数组里。

三． 语法分析的设计

采用 LL(1)语法分析，从左到右挨个处理 token 数组里的元素，自顶向下进行解析，每次只向前看一个 token[7]。根据输入的 token，来判断可能的产生式函数，然后将接下来的 token 传递给函数，这样递归处理，直到所有的 token 都处理完成或者是再没有匹配的产生式为止。

语法分析的时候要生成抽象语法树。语法树的结构如图 4-1 所示。每个大方块代表一个节点，大方块内的每个小方块代表一个指针。一个节点有五个指针，分别指向其父节点，左邻居，右邻居，左孩子和右孩子节点。如果有多个孩子节点，则左孩子指向最左端的孩子节点右孩子指针指向最右端的孩子节点。每次应用产生式，都将产生式左边的非终结符作为父节点，产生式右边非终结符作为子节点，子节点的顺序按非终结符在该产生式中的顺序排列。

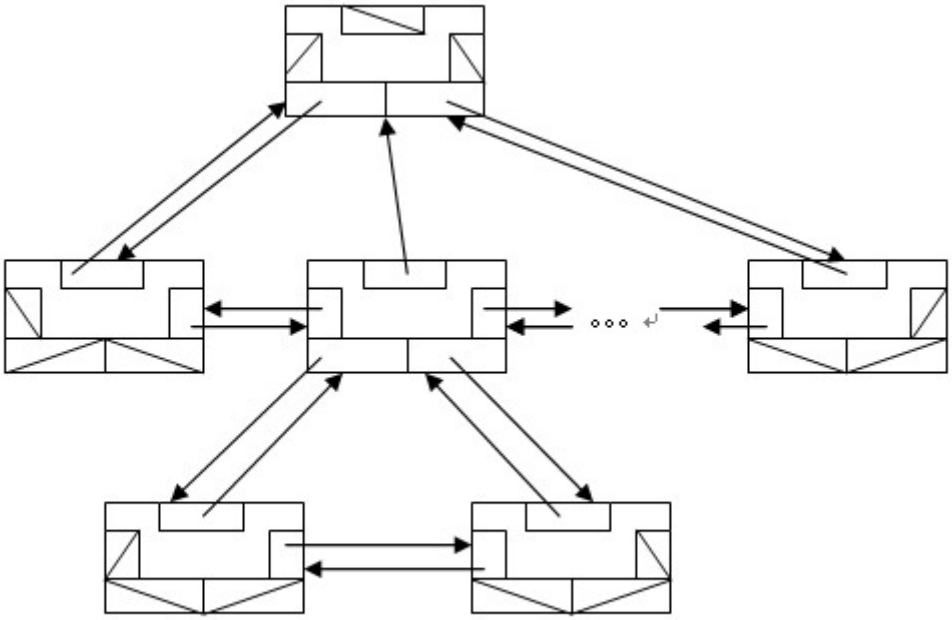

图 4-1　　抽象语法树的结构示意图

生成抽象语法树后，要将该语法树转换为数据库内部表示的结构。针对不同的操作有不同的数据结构（包括 CreateTableData，CreateIndexData，CreateViewData，DeleteData，InsertData，ModifyData 和 QueryData）。每种数据结构都要包含该数据库语句的主要信息，用于生成执行计划。

四． 类型检查设计

语法分析后产生的 QueryData 要进行类型检查。类型检查首先检查 FROM 子句，判断 FROM 子句中的表是否存在数据库中，如果都存在，则将 FROM 子句中的表信息都保存下来，然后检查 SELECT 子句，判断 SELECT 子句中的字段是否包含在 FROM 子句中的表中。然后再检查 WHERE 子句，判断 WHERE 子句中的字段是否包含在 FROM 子句的表中，并且判断 WHERE 子句中条件表达式两边的类型是否兼容。如果是两个字段的话，判断两个字段的类型是否一致，如果是一个字段一个常量的话，判断该字段的类型与常量的类型是否一致。类型检查要指出错误信息。类型检查不改变 QueryData 的结构。

4.2　语法分析器的实现

4.2.1　语法分析器的类

语法分析器的命名空间为 TBase.parse，包含 13 个类，分别是 BadSyntaxException，CreateIndexData，CreateTableData，CreateViewData，

DeleteData，InsertData，QueryData ，ModifyData，Lexer，Parser， ast_Node，ParseTree，TypeCheck。其中 BadSyntaxException 是一个异常的定义，用于表示语法错误的一个异常。接下来的七个 Data 类，是数据库内部对 SQL 的表示结构，根据名字的含义我们可以知道每个 Data 类表示哪种 SQL 类型。Lexer 是词法分析的类，Parser 是语法分析器的类，调用其他类来完成语法分析。ast_Node 表示一个抽象语法树的节点，ParseTree 是抽象语法树的类，TypeCheck 是类型检查的类。语法分析器的对外接口由 Parser 类和 TypeCheck 提供。

各个类的调用关系如下：一个 SQL 语句传入语法分析器 Parser 之后，Parser 首先调用 Lexer 进行词法解析，构建 token 表，如果检查出词法错误的话，就抛出 BadSyntaxException 异常，解析无误后，Parser 类根据 token 表，进行语法分析，语法分析的同时调用 ParseTree 类构建语法分析树，ParseTree 类构建语法树的时候调用 ast_Node 来创建抽象语法树的节点。构建语法树的同时，Parser 要将 sql 语句转换为数据库内部表示的各种 Data 类对象。完成之后产生各种 Data，接下来调用 TypeCheck 对 QueryData 进行类型检查。

4.2.2 词法分析

词法分析器(Lexer)需要一个 token 表来存储解析出来的 token，这里的 token 表用一个 string 数组来实现。除此之外需要定义正则表达式语句，包括表示整数常量的正则表达式 intPattern，表示字符串常量的正则表达式 stringPattern，表示标识符的正则表达式 idPattern，表示关键字的正则表达式 keywordPattern 和表示分隔符的正则表达式 splitPattern。他们的定义如图 4-2 所示。其中 position 变量用来保存词法检查的位置，即 token 数组中的索引位置。

```
        private List<string> keywords = new List<string>();
        private string[] tokens;
        private int position;
        private const string intPattern = @"^[-+]?(0|[1-9]\d*)$";
        private const string stringPattern = @"^'[^']*'$";
        private const string idPattern = @"^[a-zA-Z_]\w*$";
        private const string keywordPattern = @"^[a-zA-Z]*$";
        private const string splitPattern =
@"\s{1}|(,){1}|(=){1}|(\(){1}|(\)){1}|('[^']*'){1}";
```

图 4-2 Lexer 属性

有了这些定义之后，调用 System.Text.RegularExpressions. Regex. Split()函数将传入的 SQL 语句，按照分隔符 splitPattern 分割为一些 token，保存到 token 数组里。该类提供了一些用于外部访问的接口。

public bool matchDelim(char d):判断一个 token 是不是一个分隔符 d，返回 token 数组 position 位置的值与 d 相等操作的结果。

public bool matchIntConstant():判断 token 数组的 position 位置的 string 对象是否符合整型常量表达式。返回 Regex.IsMatch(tokens[position], intPattern)的结果。IsMatch 函数判断一个 string 对象是否符合一个给定的正则表达式。以下所有的 match 函数都是这样做的。

public bool matchstringConstant(): 判断 token 数组的 position 位置的 string 对象是否符合字符串常量表达式。

public bool matchKeyword(string w): 判断 token 数组的 position 位置的 string 对象是否是关键字 w。

public bool matchId(): 判断 token 是否是一个 Id。首先该 token 必须符合 Id 的正则表达式，其次它还不能在关键字列表中，即不是关键字。

public void eatDelim(char d): 吃掉一个分隔符 d，该函数的功能是首先判断 token 表中 position 位置的 token 是否是一个分隔符，如果不是，则抛出 BadSyntaxException 异常，否则就将 position 位置移动到下一个 token。接下的 eat 函数与此相似。

public int eatIntConstant(): 吃掉一个整型常量。

public string eatStringConstant(): 吃掉一个字符串常量。

public void eatKeyword(string w): 吃掉一个关键字，不需要返回吃掉的关键字，因为不会对他进行处理。

public string eatId(): 吃掉一个标识符。

private void nextToken(): 让 position 指向下一个 token。

private void initKeywords(): 初始化关键字列表。

4.2.3 语法分析

语法分析器对外提供的两个接口是 public QueryData query(ast_Node node)和 public object updateCmd()。这两个函数调用其余的访问类型为 private 的函数，来完成语法解析功能。

query 函数解析一个查询语句（select from where 查询块）。它的代码如图 4-3 所示。

```
public QueryData query(ast_Node node)
{
    ast_Node query = new ast_Node(NODE_TYPE.Query);
    if (node == null)
    {
        ParseTree parseTree = new ParseTree();
        ast_Node command = new ast_Node(NODE_TYPE.Command);
        parseTree.setRoot(command);
        command.setLeftchild(query);
        command.setRightchild(query);
        query.setParent(command);
    }
    else
    {
        node.setLeftchild(query);
        node.setRightchild(query);
        query.setParent(node);
    }
    lex.eatKeyword("select");
    ast_Node SelectList = new ast_Node(NODE_TYPE.SelectList);
    query.setLeftchild(SelectList);
    List<string> fields = selectList(SelectList);
    lex.eatKeyword("from");
    ast_Node TableList = new ast_Node(NODE_TYPE.TableList);
    SelectList.setNext(TableList);
    TableList.setPrev(SelectList);
    TableList.setParent(query);
    List<string> tables = tableList(TableList);
    Predicate pred = new Predicate();
    if (lex.matchKeyword("where"))
    {
        ast_Node Predicate = new ast_Node(NODE_TYPE.Predicate);
        TableList.setNext(Predicate);
        Predicate.setPrev(TableList);
        query.setRightchild(Predicate);
        Predicate.setParent(query);
        lex.eatKeyword("where");
        pred = predicate(Predicate);
    }
    else
    {
        query.setRightchild(TableList);
    }
    return new QueryData(fields, tables, pred);
}
```

图 4-3　　query 代码

其中 ast_Node 是创建语法树的节点。lex.eatKeyword(“select”)是匹配 select 关键字，如果成功的话，那么调用 selectList 函数来匹配投影列表。然后用 lex.eatKeyword(“from”)来匹配关键字 from，如果成功的话，调用 selectList 来分析 from 子句。接着调用 lex.eatKeyword(“where”)，如果存在 where 关键字的话，那么调用 predicate 来解析 where 子句。虽然对于 select，from，where 关键字的解析放在同一个函数中，但是这还是 LL(1)语法分析。因为还是从左到右挨个处理 token，并没有跳跃。无论在哪一步解析发生语法不匹配现象，都会抛出异常，来指示语法错误。

语法分析的同时建立语法树。对每一个产生式，都将左边的非终结符作为父

节点，右边的非终结符按顺序作为该父节点的子节点。调用 setNext，setPrev 来设置兄弟节点，调用 setParent 来设置父节点，调用 setLeftchild 和 setRightchild 设置孩子节点。

4.2.4 类型检查

类型检查的伪代码如图 4-4 所示：

```
public static bool checkquery(QueryData query, ref String errormsg,Transaction tx)
{
     List<TableInfo> tableinfos=new List<TableInfo>(); //保存检查后from子句中表的信息
     //首先检查From子句
     foreach (String table in tables)
     {
          if(table不在数据库中)
          {
               将该错误信息加入errormsg中;
          }
          else
          {
               将该表的信息加入tableinfos;
          }
     }
     //检查select子句中的字段是否在From子句中的表里
     foreach (String field in fields)
     {
          if (field不在tableinfos)
          {
               将该错误信息加入errormsg中;
          }
          if (! success)
          {
               返回false;
          }
     }
     //检查where子句
     foreach (Term term in terms)
     {
          首先获得term的左右两个表达式;
          if（左表达式是字段）
          {
               判断该字段是否在tableinfos的表中;
               如果在的话求出该字段的字段类型;
          }
          else
          {
               求出该常量的类型;
          }
          if（右表达式是字段）
          {
               判断该字段是否在tableinfos的表中;
               如果在的话求出该字段的字段类型;
          }
          else
          {
               求出该常量的类型;
          }
          if(左右表达式的类型不一样)
          {
               将该错误加入errormsg;
               return false;
          }
     }
}
```

图4-4　　类型检查伪代码

4.3　语法分析器的测试

语法分析器的测试工作主要着眼于以下几点：

一，词法分析器能够产生正确的 token 表。通过在词法分析产生 token 表之后设置断点，查看 token 数组内容来完成该测试，如图 4-5 所示。多次测试结果均正确。

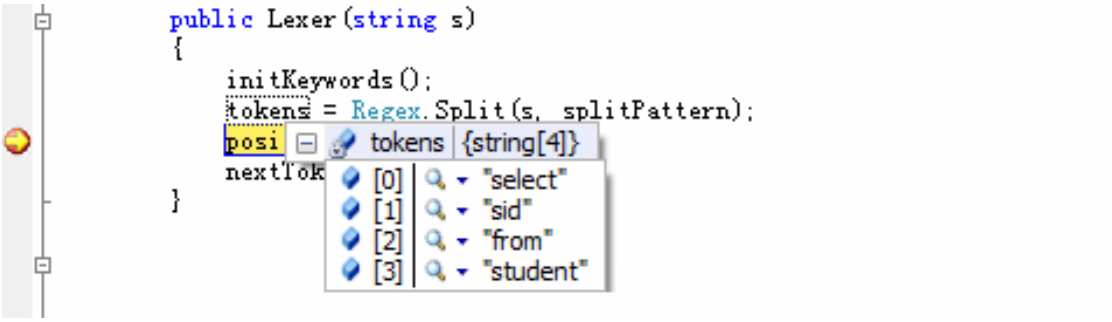

图 4-5　　“select sid from student”语句 token 数组的内容

二，语法分析能够正确的完成，并生成相应的语法树。通过输出语法树（如图 4-6）和设置断点查看语法分析后的内部结构来完成测试。经过多次测试和修改，该实现已经能达到预期的目标。

图 4-6　　“select sid from student”语句的抽象语法树

三，类型检查要能够检查出所有可能的错误。通过输入各种 SQL 语句，有的正确，有的错误，来统计类型检查是否能够检查出这些存在的错误来完成测试

工作。经过多次的测试和修改，类型检查已经能够检查出几乎是所有的错误，基本符合设计要求。

第五章　　优化器

5.1　优化器的设计

优化器的主要任务是将语法分析产生的 QueryData 转换成一个比较优的执行计划。这就有两方面的内容，首先要将 QueryData 转换为一个执行计划，其次这个执行计划是比较优的。

优化器的设计考虑以下几个方面：

第一，执行计划如何表示。一般的执行计划是一棵树。该树的节点（Plan）是操作符类[4]。包含叉积操作（ProductPlan）,投影操作（ProjectPlan），选择操作（SelectPlan），索引连接（IndexJoinPlan），索引选择（IndexSelectPlan），多缓冲叉积（MultiBufferProductPlan）和表扫描操作（TablePlan）。每个 Plan（除了表扫描 TablePlan）都以别的 Plan 作为输入，也可以作为另一个 Plan 的输入。一个查询计划如图 5-1 所示。树的最底端为 TablePlan，即数据库的存取方法，为上层的 Plan 提供数据。

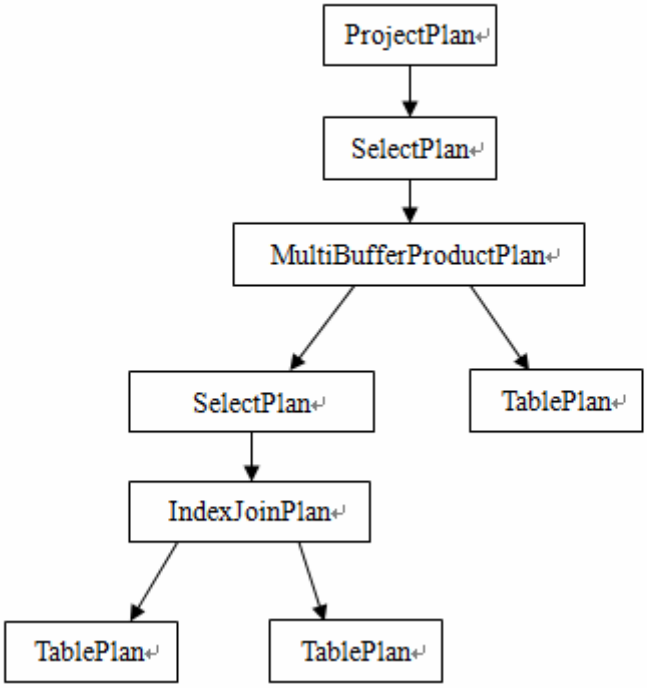

图 5-1　　一棵执行计划树

第二，怎么样生成一个比较优的执行计划。这是一个比较复杂的问题，也是体现一个数据库性能好坏的问题。几乎每个商业数据库系统都有自己的一些启发式规则来优化一个执行计划。本论文中，采用一种比较简单的优化策略：贪心策

略[5][14]。首先考虑的解空间为左深树。左深树是执行计划树的右子树都为基本表（TablePlan），或者是一个基本表应用选择和投影操作[6]。这样做的原因是如果考虑所有组合情况的话，那么解空间会相当大。而且优化器没有必要找出最优的执行计划，而只是找到相对优的，所有没有必要浪费那么多时间搜索所有的可能，来找到最优的执行计划。别忘了优化器的作用是要加快查询执行的速度，如果考虑所有的可能的话，那么优化过程本身就已经相当费时了。这样就得不偿失了。其次，考虑如何构建一个比较优的执行计划。采用以下的贪心策略：先将所有的 From 中的表生成 TablePlan，然后对于所有的 TablePlan 选择输出记录最少的 TablePlan 作为连接的第一个表（如果该 TablePlan 可以进行选择条件下推的话，那么先应用选择条件下推规则，然后再计算它的输出记录的数目[15][6]）。然后对于剩下的所有的 TablePlan，与当前已创建出的子执行计划生成连接计划，然后选择生成的连接计划输出记录最少的 TablePlan 作为下一个要进行连接的表（如果连接后的 Plan 可以应用选择条件下推的话，那么先应用选择条件下推，生成 SelectPlan，然后再记录输出的记录数目），然后加入到已创建的执行计划中，生成新的子执行计划。按同样的方法直到将所有的表都加入到执行计划中。最后进行投影，创建 ProjectPlan。它的程序执行流程图如图 5-2 所示

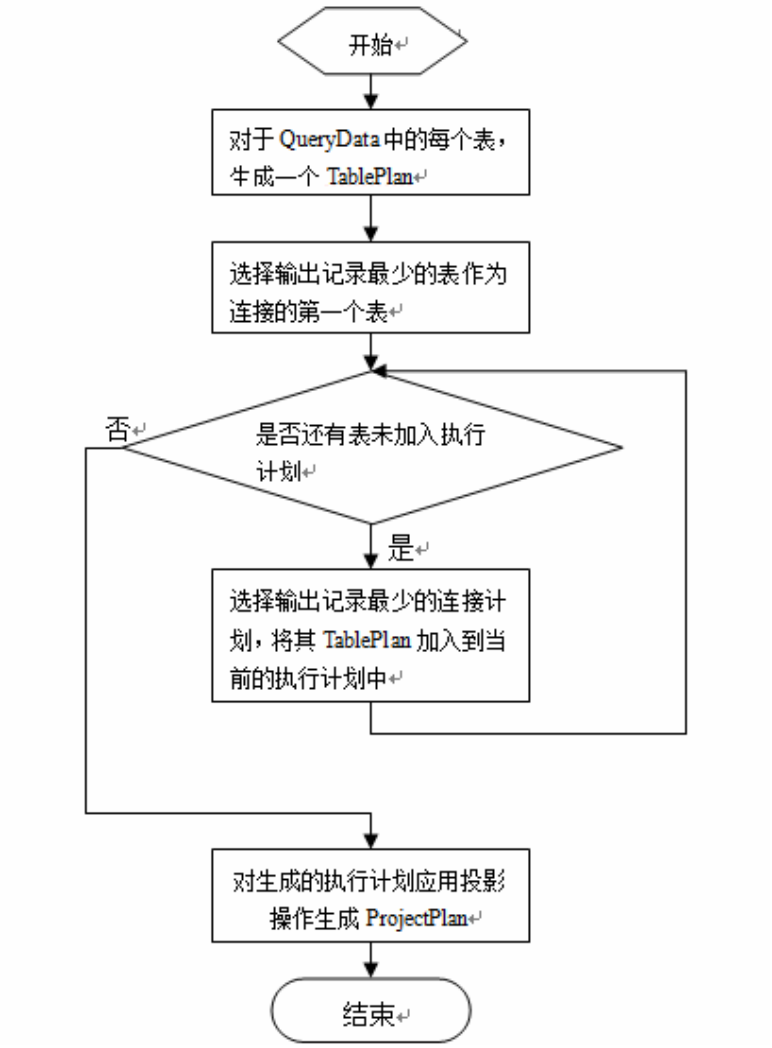

图 5-2　　优化器执行流程图

5.2　优化器的实现

5.2.1　优化器的类

优化器的实现包含在以下几个命名空间中：TBase.multibuffer，TBase.opt，TBase.query 和 TBase.index.query。各个命名空间包含的优化器类及功能如表 5-1 所示。

表 5-1　　优化器的类

类名	命名空间	作用
Plan	TBase.query	执行计划节点的接口，其余的执行计划节点类都要实现这个接口
SelectPlan	TBase.query	选择操作的执行计划

TablePlan	TBase.query	单个表上的执行计划，提供表中记录的访问，是执行计划的叶子页
ProductPlan	TBase.query	叉积操作的执行计划，简单的嵌套循环连接
MultiBufferProductPlan	TBase.multibuffer	多缓冲叉积执行计划，进行块嵌套循环连接
IndexSelectPlan	TBase.index.query	索引选择操作的执行计划，执行利用索引的扫描
IndexJoinPlan	TBase.index.query	索引连接操作的执行计划，内关系利用索引来取记录
HeuristicQueryPlanner	TBase.opt	优化器类，对外提供优化器调用接口，将 QueryData 作为输入，利用一个启发式规则产生执行计划树，返回执行计划树的根节点。
TablePlanner	TBase.opt	表执行计划器。能够根据需要，将一个 TablePlan 节点应用选择条件生成 SelectPlan，也可以和另一个 Plan 作叉积生成 MultiBufferProductPlan 等等
Constant	TBase.query	常量的接口。定义了一个 asCsharpVal 函数，用来返回常量的值，常量分为整型常量和字符串常量
IntConstant	TBase.query	表示整型常量的类
StringConstant	TBase.query	表示字符串常量的类
Expression	TBase.query	表达式接口，表达式是谓词中“=”两边的类型，比如一个 name=‘John’，其中 name 和 John 都为表达式
ConstantExpression	TBase.query	常量表达式，例如上例中的 John
FieldNameExpression	TBase.query	字段表达式，例如上例中的 name 字段
Term	TBase.query	谓词表中的一项，例如 name=‘John’就是一个 Term
Predicate	TBase.query	谓词类。一个谓词是多个 Term 用 AND 连接起来

5.2.2 优化器的实现

讨论一些关键类的实现：

一．Plan 接口

每个关系代数操作都有一个对应的 Plan 类，这个 Plan 接口是所有的 Plan 类都必须实现的。它提供了以下接口：

Plan getLeftPlan()： 函数返回该 Plan 的左边的输入 Plan，如果没有的话，返回 null。

Plan getRightPlan()： 函数返回该 Plan 的右边的输入 Plan，如果没有的话，返回 null。这两个函数用于显示执行计划。

Scan open()： 打开一个迭代器，用来取该 Plan 产生的记录，这个是执行器中的内容。会在下一章详细讨论。

int blocksAccessed()： 函数返回该 Plan 要访问多少个块。

int recordsOutput()： 函数返回该 Plan 输出记录的数目。

int distinctValues(string fldname)： 函数返回该计划的结果中指定字段的不同值的数目。

Schema schema()： 返回该查询的概念模式。

二．ProductPlan 类

看一个具体的 Plan 类。该类是做叉积运算的。其类如图 5-3 所示。

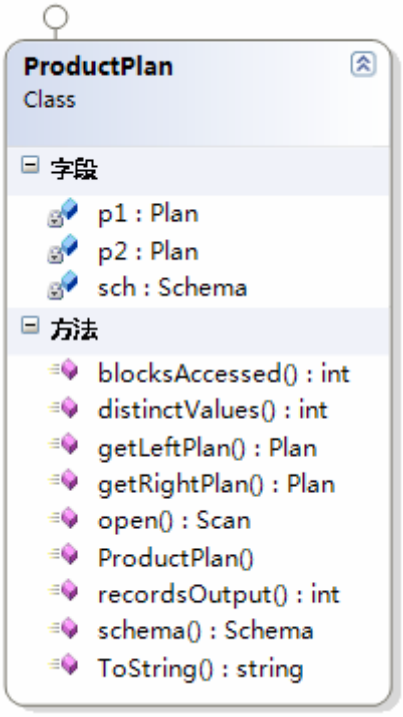

图 5-3 ProductPlan 的类

有两个输入 Plan，分别为 p1，p2。sch 是该 Plan 的输出记录的概念模式。

getLeftPlan 和 getRightPlan 函数分别返回 p1 和 p2.

public ProductPlan(Plan p1, Plan p2) ：构造函数，分别为 p1 和 p2 赋值，sch 的值为 p1 的 schema 和 p2 的 schema 的并。

public Scan open() ：分别打开 p1 和 p2 的 Scan，然后将他们作为参数传递给该类的 Scan。

public int blocksAccessed() ：叉积操作访问的块数，其计算公式为

$$B(product(p1,p2)) = B(p1) + R(p1)*B(p2)$$

B()代表该记录访问的块数，R()代表该计划输出的记录的数目。

public int recordsOutput()：叉积操作输出的记录的数目，其计算公式为：

$$R(product(p1,p2)) = R(p1)*R(p2)$$

R()代表计划输出记录的数目。

public int distinctValues(string fldname)：返回字段 fldname 上的不同值的数目。返回该字段所在输入 plan 中的不同值的数目。

三. TablePlanner 类

该类的作用是处理单个表计划。其类如图 5-4 所示。

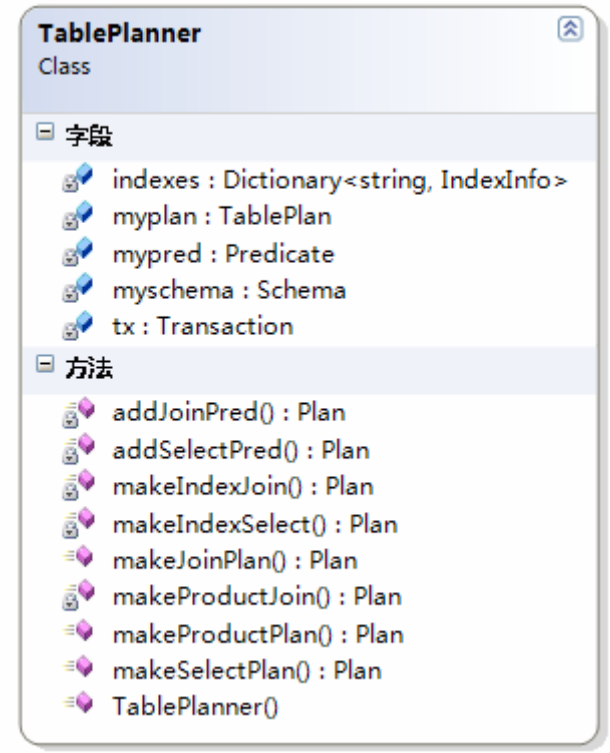

图 5-4　　TablePlanner 类

字段的含义如下：

indexex 保存该表的索引的信息，是一个数据字典，键是索引名，值是索引

信息。

myplan 是一个表计划 TablePlan。用它来作为输入，产生别的计划。

mypred 是该查询的查询谓词。

myschema 是该计划的概念模式，也就是 myplan 的概念模式。

tx 是执行该查询操作的事务的引用。

方法的作用如下：

public TablePlanner(string tblname, Predicate mypred, Transaction tx)：构造函数，产生 TablePlan，利用元数据管理器获得表 tblname 的索引信息，为字段赋值。

private Plan addSelectPred(Plan p) 为计划 p 应用选择条件。如果有存在只涉及计划 p 的概念模式上字段的谓词子句的话，那么就对计划 p 应用该谓词，构成一个 SelectPlan，并返回它，否则就返回计划 p。

private Plan addJoinPred(Plan p, Schema currsch)：为计划 p 应用连接条件。如果 p 计划所涉及的两个子 plan 有连接条件的话，那么就对 p 应用连接条件进行选择，形成一个选择计划。

private Plan makeIndexSelect()：创建索引选择计划。对于该执行计划的选择谓词，如果存在一个 Term 的表达式是形如 Field=constant 的形式，且 Field 上存在索引的话，那么就创建一个索引选择计划来取表中的记录。

private Plan makeIndexJoin(Plan current, Schema currsch) ：为 current 计划和当前表计划创建索引嵌套连接。其代码如图 5-5 所示。对于该表的所有索引字段 fldname，如果谓词中有形如 fldname= outerfield 或者 outerfield =fldname 的话，并且 outerfield 在计划 current 的概念模式 currsch 中的话，那么就创建一个以该表为内关系的索引嵌套连接。然后对该计划应用选择条件和连接条件。

```
private Plan makeIndexJoin(Plan current, Schema currsch)
{
    foreach (string fldname in indexes.Keys)
    {
        string outerfield = mypred.equatsWithField(fldname);
        if (outerfield != null && currsch.hasField(outerfield))
        {
            IndexInfo ii = indexes[fldname];
            Plan p = new IndexJoinPlan(current, myplan, ii, outerfield, tx);
            p = addSelectPred(p);
            return addJoinPred(p, currsch);
        }
    }
    return null;
}
```

图 5-5　　makeIndexJoin 代码

private Plan makeProductJoin(Plan current, Schema currsch)：为 current 计划和

当前表计划创建叉积连接。首先调用 makeProductPlan 创建 current 和当前表的叉积计划，然后对结果计划应用连接条件进行记录选择，形成一个选择计划。

public Plan makeSelectPlan()：为当前表创建选择计划。如果该表可以先生成索引选择计划的话，那么首先生成索引选择计划，然后下推该表上的选择操作，形成一个选择计划。否则，直接生成一个选择计划。

public Plan makeJoinPlan(Plan current)：为 current 计划和当前的表计划生成一个连接计划，其代码如图 5-6 所示。首先获得连接谓词，如果不存在的话，那么代表它们两个不能连接，直接返回 null。否则的话，创建一个索引连接，如果不成功的话，那么就创建一个叉积连接，然后返回计划 p。

```
public Plan makeJoinPlan(Plan current)
{
    Schema currsch = current.schema();
    Predicate joinpred = mypred.joinPred(myschema, currsch);
    if (joinpred == null)
        return null;
    Plan p = makeIndexJoin(current, currsch);
    if (p == null)
        p = makeProductJoin(current, currsch);
    return p;
}
```

图 5-6　　makeJoinPlan 代码

public Plan makeProductPlan(Plan current)：为 current 计划和当前表计划创建一个叉积计划。首先对当前的表计划应用选择条件下推操作，生成一个选择计划，然后为该选择计划再创建一个多缓冲叉积计划，并返回它。

四. HeuristicQueryPlanner 类

这个是查询优化器的主要类，对外提供调用接口，该类完成查询优化工作。该类如图 5-7 所示。

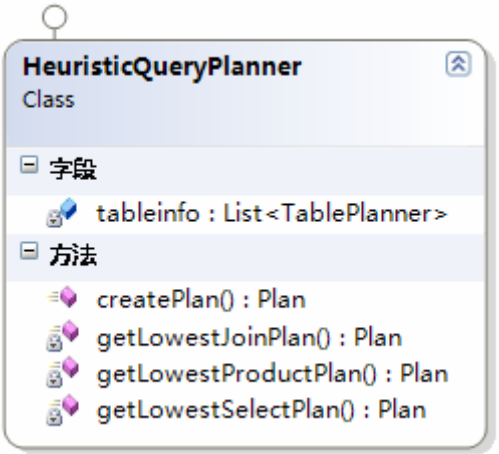

图 5-7　　HeuristicQueryPlanner 类

字段 tableinfo 的类型为 List<TablePlanner>，它是一个单个表执行计划生成器的链表。对于查询语句的 From 子句中的每一个表都对应一个 TablePlanner，在 tableinfo 都要增加一个元素。

首先介绍该类的三个私有的辅助函数，它们仅供类内部调用。

private Plan getLowestSelectPlan()：该函数的作用是从当前的 tableinfo 中，首先应用选择条件下推，然后找出输出记录最少的计划，并从 tableinfo 删除对应的 TablePlanner。其代码如图 5-8 所示。

```
private Plan getLowestSelectPlan()
{
    TablePlanner besttp = null;
    Plan bestplan = null;
    foreach (TablePlanner tp in tableinfo)
    {
        Plan plan = tp.makeSelectPlan();
        if (bestplan == null || plan.recordsOutput() < bestplan.recordsOutput())
        {
            besttp = tp;
            bestplan = plan;
        }
    }
    tableinfo.Remove(besttp);
    return bestplan;
}
```

图 5-8　　getLowestSelectPlan 代码

private Plan getLowestJoinPlan(Plan current)：返回包含 current 计划与一个单个表上的计划的开销最小的连接计划。对于 tableinfo 中的所有单个表都与 current 做连接，返回输出记录最少的连接计划，并把对应的 TablePlanner 从 tableinfo 中删除。

private Plan getLowestProductPlan(Plan current)：返回包含 current 计划与一个单个表上的计划的开销最小的叉积计划。对于 tableinfo 中的所有单个表都与 current 做叉积，返回输出记录最少的连接计划，并把对应的 TablePlanner 从 tableinfo 中删除。

接下来介绍一下 HeuristicQueryPlanner 对外提供调用的函数 createPlan，改函数的代码如图 5-9 所示。

```
public Plan createPlan(QueryData data, Transaction tx)
{
    //第一步：为每一个提到的表创建一个tablePlanner对象。
    foreach (string tblname in data.tables())
    {
        TablePlanner tp = new TablePlanner(tblname, data.pred(), tx);
        tableinfo.Add(tp);
    }

    //第二步：选择最小的计划来开始join顺序。
    Plan currentplan = getLowestSelectPlan();

    //第三步：重复添加一个计划到join顺序
    while (tableinfo.Count > 0)
    {
        Plan p = getLowestJoinPlan(currentplan);
        if (p != null)
            currentplan = p;
        else
            currentplan = getLowestProductPlan(currentplan);
    }

    //第四步：在域名上投影并返回。
    return new ProjectPlan(currentplan, data.fields());
}
```

图 5-9　　createPlan 代码

public Plan createPlan(QueryData data, Transaction tx)：该函数接收一个查询结构 QueryData（该结构是数据库对于一个 SQL 语句的内部表示）产生一个优化的查询计划。它利用一个简单的启发式规则：首先选择一个输出记录最少的表，然后从剩余的表中依次选择与当前计划做连接输出记录最少的表作为下一个要连接的表，最后直到所有的表都加入了查询计划中，并且生成了一个优化的查询计划。在评估输出记录的数目的时候，要将选择操作下推，即尽可能先应用选择谓词来减少输出记录的数目。

5.3 优化器的测试

对于优化器的测试分为以下几步：

第一，查看优化器生成的执行计划。因为优化器是一个树结构，所以可以通过遍历树节点来输出生成的执行计划。测试代码如图 5-10 所示。

```
private void plantree_Click(object sender, EventArgs e)
{
    Plan p=gl.getPlan();
    string result = subPlanTreeToString(p, 1);
    MessageBox.Show(result, "执行计划树", MessageBoxButtons.OK);

}

private string subPlanTreeToString(Plan p,int level)
{
    string result = "";
    int i = 1;
    for (i = 1; i <= level; i++)
    {
        result += "   ";
    }
    result = result + p.ToString();
    result = result + "\n";
    if(p.getLeftPlan()!=null)
    {
        result = result + subPlanTreeToString(p.getLeftPlan(), level + 1);
    }
    if (p.getRightPlan() != null)
    {
        result = result + subPlanTreeToString(p.getRightPlan(), level + 1);
    }
    return result;
}
```

图 5-10　　输出执行计划树代码

plantree_Click 是对显示执行计划树按钮的响应函数，调用 subPlanTreeToString 函数来将执行计划树保存到一个 string 对象中，然后用 MessageBox.Show(result, "执行计划树", MessageBoxButtons.OK)将其输出。subPlanTreeToString 是将一个子树的内容保存在一个 string 对象中，并且返回，这是一个递归函数。传入参数 p 代表子树的根节点，level 代表该节点所在的层数，根节点为第一层。通过反复的输出执行计划，并且修改错误，最终该功能可以正确的运行。

第二，通过黑盒测试的方法，在整合后的程序上执行查询语句来测试优化器是否能正常工作，经过多次测试和修改，更正了一些错误，现在已经能够正常的工作。

第六章　　执行器

6.1　执行器的设计

执行器接收一个优化器产生的执行计划树，然后从树的顶端开始递归的调用每一个操作符完成查询，产生执行结果。

执行器采用迭代器方式来实现。一个典型的迭代器如图 6-1 所示，要实现三个接口 open()，get_next(),close()[1][4][6]。查询中的每个操作符（执行计划树中的每个节点）都实现为上述超类 Iterator 的对象。迭代器的一个特点是任何迭代器子类可以成为别的迭代器的输入。因此每个迭代器的逻辑是独立于它的孩子和父亲的，没有必要为指定的连接方式编写特殊的代码。

```
class Iterator
{
    Iterator[] inputs;
    void open();
    tuple  get_next();
    void close();
}
```

图 6-1　　迭代器接口

通过调用执行计划树顶端的 open 函数，打开顶端节点的迭代器，然后递归调用子节点的 open 函数。直到打开所有的迭代器。通过调用顶端节点的 get_next()，然后递归调用子节点的 get_next()，直到到达叶子节点，通过表的访问路径来返回记录，然后向上层调用者传递，产生一条结果。反复调用执行计划顶端的 get_next()函数，直到所有输入的迭代器为空来产生执行结果。执行完后调用顶端的 close()函数，递归调用子节点的 close()函数来关闭所有的迭代器。

TBase 中，应该对每个支持的操作符创建对应的迭代器（迭代器的接口为 Scan），包括：TableScan，SelectScan，ProjectScan，ProductScan，MultiBufferProductScan，IndexJoinScan，IndexSelectScan。每个 Scan 都由对应的 Plan 执行 open()函数打开 Scan。然后调用 Scan 的 next 函数来定位记录，然后返回值。支持 close()函数来关闭迭代器。

6.2　执行器的实现

6.2.1　执行器的类

执行器的实现包含在以下几个命名空间中：TBase.multibuffer，TBase.opt，

TBase.query 和 TBase.index.query。各个命名空间包含的执行器类及功能如表 6-1 所示。

表 6-1　　执行器的类

类名	命名空间	作用
Scan	TBase.query	执行器的迭代器接口定义
UpdateScan	TBase.query	可更新的迭代器接口，用于可以更新的操作
SelectScan	TBase.query	选择操作的迭代器
TableScan	TBase.query	表扫描操作迭代器，是最底层的迭代器，为上层提供数据
ProductScan	TBase.query	叉积操作的迭代器，简单的循环嵌套连接
MultiBufferProductScan	TBase.multibuffer	多缓冲叉积操作的迭代器，进行块嵌套循环连接
IndexSelectScan	TBase.index.query	索引选择操作的迭代器，执行利用索引的扫描
IndexJoinScan	TBase.index.query	索引连接操作的迭代器，内关系利用索引来取记录

6.2.2　执行器的实现

讨论一些关键类的实现：

一. Scan 接口

Scan 是一个迭代器接口，该接口的定义如图 6-2 所示，它符合一个标准迭代器接口。不过迭代器的打开操作 open 函数是在 Plan 类中提供的。Plan 类中的 open 函数打开一个 Scan 对象，不过在此之前要先打开该 Scan 的所有子 Scan 对象。

```
public interface Scan
{
    void beforeFirst();
    bool next();
    void close();
    Constant getVal(string fldname);
    int getInt(string fldname);
    string getString(string fldname);
    bool hasField(string fldname);
}
```

图 6-2　　Scan 接口

void beforeFirst()：将游标置于该 Scan 的初始位置，即该计划的输出记录的初始位置。

bool next()：判断该迭代器是否还可以取得记录，如果有则返回 true，否则返回 false。执行 next 的时候，移动游标的位置到符合条件的记录。

void close()：关闭迭代器。

Constant getVal(string fldname)：获得游标指定的记录的 fldname 字段的值。

int getInt(string fldname) ：获得游标指定的记录的 fldname 字段的整型值。

string getString(string fldname)：获得游标指定的记录的 fldname 字段的字符串值。

bool hasField(string fldname)：判断该迭代器游标所指的记录是否包含字段 fldname。

二. UpdateScan 类

这是可更新操作的迭代器接口类。有些操作是可以对记录进行修改的，这类的 Scan 就继承自 UpdateScan。UpdateScan 的定义代码如图 6-3 所示。

```
public interface UpdateScan : Scan
{
    void setVal(string fldname, Constant val);
    void setInt(string fldname, int val);
    void setString(string fldname, string val);
    void insert();
    void delete();
    RID getRid();
    void moveToRid(RID RID);
}
```

图 6-3　　UpdateScan 代码

void setVal(string fldname, Constant val)：把当前记录的 fldname 字段的值设置为 val。

void setInt(string fldname, int val)：把当前记录的 fldname 字段的值设置为整型值 val。

void setString(string fldname, string val)：把当前记录的 fldname 字段的值设置为字符串值 val。

void insert()：将当前位置插入一个条记录。

void delete()：删除当前记录。

RID getRid()：返回当前记录的记录号。

void moveToRid(RID RID)：把游标移动到指定记录号的记录。

三. ProductScan 类

ProductScan 为叉积操作的迭代器类，对于外关系的每条记录与内关系的所有记录进行叉积操作，其类如图 6-4.

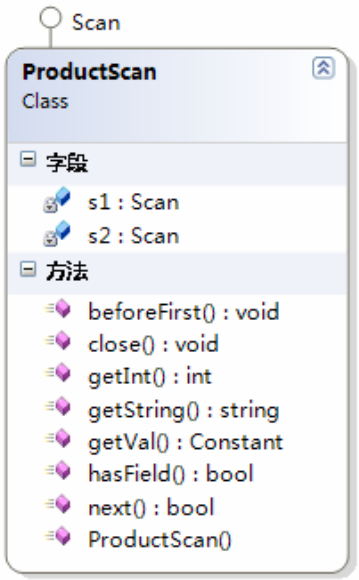

图 6-4　　ProductScan 类

字段 s1 为外关系迭代器。字段 s2 为内关系迭代器。

分析一下该类的几个关键方法：

public bool next()：该方法的定义如图 6-5 所示。假如 s2 还有数据的那么就返回真。否则的话，证明内关系已经读完，那么就需要将外关系的游标往后移一位，将内关系的游标重新移动到首位。

```
public bool next()
{
    if (s2.next())
        return true;
    else
    {
        s2.beforeFirst();
        return s2.next() && s1.next();
    }
}
```

图 6-5　　next 代码

public Constant getVal(string fldname)：返回当前记录的 fldname 字段的值。如果该字段在 s1 中话，就调用 s1 的 getVal，否则调用 s2 的 getVal。

public void close() 关闭迭代器，首先关闭 s1 和 s2.

6.3 执行器的测试

采用黑盒测试的方法，将程序首先整合到一起，然后执行预先设定的 SQL 语句，观察执行的结果与预期的结果是否一致。通过多次测试和修改，现已工作正常。

第七章　结论

7.1　总结

DBMS 的应用越来越广泛，成为了现代信息技术中不可缺少的部分。如今商业数据库系统百花齐放，虽然有许多的 DBMS，其中不乏开源的，但是真正可以用于教学的数据库系统却特别少，尤其是.NET 平台上更是一个空白。为了解决这个问题，我们设计实现了 TBase。

论文首先简要介绍了 TBase 的功能和模块（存储管理，事务管理和查询处理），并详细分析了 TBase 的查询处理子系统。

对于查询处理子系统的介绍，首先从宏观上阐述了查询处理的原理和执行的流程，以及 TBase 查询处理所包含的模块（元数据管理，语法分析，优化器和执行器）及每个模块的大体功能。紧接着分为四个章节对每个模块的需求，设计，实现和测试进行了详细的分析。

总之，TBase 对于.NET 平台下数据库的教学是十分有意义的，它为数据库的研究提供了一个相对容易读懂的源码，这对我们理解数据库的原理，设计和实现都是十分有帮助的。

7.2　未来工作

毕竟 TBase 主要目的是用于教学，提供一个功能简单，但是模块完整的 DBMS。所以还存在很大的可扩展性。

首先，对于 SQL 语法的支持。TBase 并没有提供对于聚集操作符，集合操作，Group by 和 Having 语句的支持。所以可以扩展 TBase 来支持这些语法。

其次，可以扩展一些操作符的实现，比如连接操作，TBase 并没有支持排序归并连接，未来应该扩展这个功能。

第三，对于优化器，TBase 目前采用的是一种比较简单的贪心策略。未来可以扩展用动态规划或者是别的性能能好，结果更优的算法来改进优化器产生执行计划。

参考文献

[1]Joseph M. Hellerstein and Michael Stonebraker．Anatomy of a Database System[A]．Readings in Database Systems 4th edition[C]．London，England ：2005．42-96．

[2]Sumit Ganguly，Waqar Hasan，and Ravi Krishnamurthy．Query Optimization for Parallel Execution[A]．In Proceedings of the ACM SIGMOD International Conference on Management of Data[C]．San Diego： 1992．9-18．

[3]Minos N. Garofalakis and Yannis E. Ioannidis．Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources[A]．In Proc. 23rd International Conference on Very Large Data Bases (VLDB)[C]． Athens，Greece： 1997．296-305．

[4]G. Graefe．Encapsulation of Parallelism in the Volcano Query Processing System[A]．In Proc. ACM-SIGMOD International Conference on Management of Data[C]． Atlantic City：1990．102-111．

[5]Goetz Graefe．The Cascades Framework for Query Optimization[J]．IEEE Data Engineering Bulletin，1995,18(3)：19-29．

[6]罗摩克里希纳(Raghu Ramakrishnan)，格尔基(Johannes Gehrke)．数据库管理系统原理与设计（第三版）[M]．北京： 清华大学出版社， 2004．1-295．

[7]加西亚(Hector Garcia-Molina)，沃尔曼(Jeffrey D. Ullman)，威德姆(Jennifer D. Widom)．数据库系统实现[M]．北京：机械工业出版社，2001．1-238．

[8]Shimin Chen，Phillip B. Gibbons and Todd C. Mowry．Improving Index Performance through Prefetching[A]． In Proc. ACM SIGMOD International Conference on Management of Data[C]，2001．

[9]David J. DeWitt，Robert H. Gerber，Goetz Graefe *et al*． GAMMA - A High Performance Dataflow Database Machine[A]．In Twelfth International Conference on Very Large Data Bases (VLDB)[C]．Kyoto,Japan：1986．228-237．

[10]David J. DeWitt，Randy H. Katz，Frank Olken *et al*．Implementation Techniques for Main Memory Database Systems[A]．In Proc. ACM-SIGMOD International Conference on Management of Data[C]．Boston,MA：1984．1-8．

[11]G. Graefe，R. Bunker，and S. Cooper．Hash joins and hash teams in Microsoft SQL Server[A]．In Proceedings of 24th International Conference on Very Large Data Bases (VLDB)[C]．1993．24-27．

[12] Kristin P. Bennett，Michael C. Ferris，and Yannis E．Ioannidis. A Genetic Algorithm for Database Query Optimization[A]．In Proceedings of the 4th International Conference on Genetic Algorithms[C]．San Diego：July 1991．400-407．

[13] G. Graefe．Query Evaluation Techniques for Large Databases[J]．Computing Surveys，1993，25 (2)： 73-170．

[14] Wei Hong and Michael Stonebraker．Optimization of Parallel Query Execution Plans in XPRS[A]．In Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS)[C]．Miami Beach：1991．218-225．

[15] Alon Y. Levy，Inderpal Singh Mumick，and Yehoshua Sagiv．Query Optimization by Predicate Move-Around[A]．In Proc. 20th International Conference on Very Large Data Bases[C]． Santiago：1994．96-107．

[16] Michael Steinbrunn，Guido Moerkotte，and Alfons Kemper．Heuristic and Randomized Optimization for the Join Ordering Problem[J]．*VLDB Journal*，1997，6(3)：191-208．

致　　谢

本篇论文的写作过程得到了老师，同学的大量帮助。

首先感谢张坤龙老师，在张坤龙老师的指导下，我确定毕业论文的方向，然后一步一步完成我的毕业设计，这期间受到了张老师很多亲自指导和鼓励。在跟随张老师的一年多时间里，我学到了许多数据库方面的知识，从刚开始的《数据库管理系统原理与设计（第三版）》的学习到后来开源数据库的源码阅读，张老师都是不辞劳苦的检查我们的学习成果，并为我们答疑解惑。后来做毕设期间，张老师也对我们的数据库的设计和实现提出了许多宝贵的意见和指导。

感谢杨晓科和韩志攀同学，我们是一起奋斗的“战友”，分别负责 TBase 的三个子系统。我们在一起学习数据库知识，阅读源代码和完成毕业设计，彼此都给予对方很大的精神上的鼓励和知识上的帮助。在完成毕设的阶段，我们更是经常在一起讨论 TBase 的设计和实现，确定 TBase 的接口，然后在 TBase 的整合方面也是互相帮助，克服种种困难，终于完成了一个完整的数据库管理系统。