项目报告

2015011308 计 53 唐适之

关于项目的编译、运行方法，请参见 README.md。

小组人员分工

仅计 53 唐适之。

实现的附加功能

1. 可以动态增删的非簇集索引，主索引和非主索引均支持使用超过一个域作为键；
2. 属性域约束，支持“CHECK (<field> in (<candidates>))”；
3. 外键约束；
4. DATE 类型和 VARCHAR 类型。不过 VARCHAR 类型还是实现为定长类型；
5. 查询优化：表中有若干索引时选择使用哪一个；
6. 多表连接时的优化：优化多表连接时查询各表的顺序，使其尽可能利用索引；
7. 聚集查询 SUM、AVG、MIN、MAX；
8. 分组查询子句 GROUP BY；
9. 排序子句 ORDER BY；
10. 两种不同的 I/O 模式：读入 SQL 文件，输出 CSV 文件；或命令行交互式读入，输出易于人类判读的字符表格。

系统设计

概述

以自底向上的视角，系统的骨干可以分为如下若干模块：首先是一个页式文件系统，该模块对下以整页为单位与硬盘交换数据，但对上提供了一个字节级别的读写接口，使别的模块可以将不同内存页中的数据当作普通内存中的数据存取，而不必担心具体的缓存替换。

页式文件系统解决了如何储存字节块的问题，还应解决如何存储不同类型数据的问题。为此，我定义了 INT、FLOAT、CHAR、VARCHAR、DATE 五种不同的类型，使用统一的接口在便于存储的字节块、便于运算的运行时表示和便于 I/O 的字面量之间相互转换。其中 DATE 类型使用 C 语言原生的 strptime 和 localtime 函数实现解析和输出。另一方面，我将内存页特化为两种：BitmapPage 类用于存储位串、ListPage 类用于存储上述不同类型的数据，至此系统实现了在内存页中存储不同类型的数据。

接下来要以内存页为基本元件建立数据库中的表。本项目中表的实现分为三层：TablePages 类管理一个表中所使用的各个内存页；BaseTable 类管理表中的基本查询，索引也实现于此；Table 类为表提供了一个面向字面量的接口，这是由于 SQL 查询中各数据类型

是可以互相转换的，单个表之外统一用字面量表示可以避免类型转换的不便。

有了表以后，就要将表组织成数据库。本项目中，TableMgr 类负责管理多个表，其中各表的表头等元信息存储在单独的系统表中，而系统表的表头则是硬编码的。

最后，SQL 解析模块负责将 SQL 解释成具体的交予 TableMgr 执行的命令，并通过 I/O 模块进行输入输出。

以下重点叙述一些较为复杂的模块。

页式文件系统

由于课程提供的页式文件系统缺陷较多，我重新实现了一个页式文件系统。

首先，我将对页的直接读写抽象成 PageMgr 纯虚类，并实现了 FilePageMgr 子类进行文件上的页读写。这样做的好处是：在对其他单元进行单元测试时，可以用一个只在内存上读写的测试类替换掉 FilePageMgr，以避免测试时大量读写文件。

其次，我联合使用哈希表和链表，用链表存哈希表的键，实现了一个类似 Java 中 LinkedHashMap 的数据结构，用于进行页的 LRU 缓存，并实现了 PageCache 类自动进行缓存替换，使缓存对上层透明。PageCache 提供给上层的接口是迭代器而非裸指针形式的，上层函数可以像使用普通指针一样使用这些迭代器。迭代器每次被调用时，会判断目标地址在不在缓存中，若不在，该地址所在页会被自动加载到缓存，相应被弹出缓存的页也会自动写入文件。

如果迭代器每次判断目标在不在缓存中时都需要查表，开销是极大的。实际测量表明，一次十万行级别的插入操作就会带来上亿次访存。故此处进行了一项优化：所有访问同一内存页的迭代器共享同一片管理信息空间，该空间包括该页缓存失效与否、该页在缓存中的位置等信息，不同迭代器间的不同仅限于偏移量。这样当一个迭代器更新了目标在缓存中的位置后，其他迭代器可以立即知悉，不必再次查表。

表内的数据页组织（TablePages 类）

一个表的主要信息都以各种上文所述的 ListPage 页承载，这些 ListPage 页储存在“数据库名.表名.data.db”文件中。但是多个 ListPage 页在不断申请和释放中会产生碎片，为了管理 ListPage 的使用情况，还需要一个或多个 BitmapPage 页记录哪些 ListPage 是空闲（可被申请）的，这个 BitmapPage 页储存在“数据库名.表名.freelist.db”文件中。

ListPage 十分灵活。首先，它是分槽的，每个槽可以储存若干不同类型的列，可以管理空值，也可以将某些列配置为非空。其次，每个 ListPage 各留有 4 个保留字段，分别用于存储其中元素的个数、上一个页 ID、下一个页 ID 和该页的标记。所以不同 ListPage 可以被标记为不同类型，也可以被串成链表使用。每个表既用 ListPage 储存具体数据，也用 ListPage 储存索引，为了在同一个文件中混合使用这些页时散而不乱，这些 ListPage 会被标记为如下 5 类之一：

1. RECORD，表示该页用于储存具体数据，页中每一个槽存储一条记录；
2. ENTRY，表示该页用于储存每个索引入口（无主索引则为链表的入口），页中每一个槽存储一个入口页的 ID；
3. REF，表示该页作为非簇集索引的叶节点使用，页中每一个槽存储一项主索引的键；
4. PRIMARY，表示该页作为主索引的非叶节点使用，页中每一个槽存储一个结点的相关信息，即每个子节点的 ID 及其对应的键；

5. NON_CLUSTER，表示该页作为某个非簇集索引的非叶节点使用，具体是哪个非簇集索引还会再做区分，用法同 PRIMARY。

表的插入、删除、查询操作（BaseTable 类）

此类使用下层（TablePages 类）提供的页执行一个表中具体的插入、删除、查询操作，并返回给上层。我将表的修改操作视作删除后再插入，所以放在更上层实现。

一． 索引

索引就是实现在了此模块中的。不同情况下索引的实现具体如下：

1. 如果没有主索引，各记录页链接成链表，非簇集索引（如有）中存储相应记录的页号。直接查询记录时，遍历链表即可；通过非簇集索引查询记录时，先找到其所在页，然后遍历改页查询；
2. 如果有主索引，各记录以主索引 B+树叶节点的形式存在，非簇集索引（如有）中存储相应记录的主键。无论是主索引还是非主索引，其叶节点之间也会链接成链表。直接查询记录时，遍历链表即可；通过主索引查询记录时，直接在主索引树上查询；通过非簇集索引查询记录时，先找到其主键，然后通过主索引查询。

新建非簇集索引时，先更新上文所述的 ENTRY 页注册索引入口，然后将所有记录的相关信息插入到索引树中。删除非簇集索引时，先递归地删除索引数，然后更新 ENTRY 页删除入口。

索引的实现本质上就是操作这些数据页，并管理它们之间的链接关系，实现索引的复杂之处不于在实现 B+树，而在于把具有拓扑结构的 B+树存储在线性的页上。索引的具体原理与课上所述无异，在此不再赘述。

二．查询和插入的优化

进行表的操作时，还需考虑一些优化问题。

首先，当表中有多个索引存在时，需要考虑选用哪个索引进行查询才是最优的。本系统选择的策略如下：对于每个索引，统计该索引的键中能被用上的域个数。而某个域f_i能被用上，当且仅当以下条件均满足：

1. 查询条件中对f_i进行了限制，且这些限制至少包含“=”、“<”、“>”、“<=”、“>=”之一。即，“!=”、“IS NOT NULL”、“IS NULL”限制是无法使用索引的；
2. 如果$i > 0$，查询条件中对f_{i-1}也进行了限制，且这些限制必须包含“=”限制。即，“<”、“>”、“<=”、“>=”限制是无法级联使用索引的。

选择键中能被用上域的个数最多的索引，即是最优的。

其次，使用数据库的一种典型情况是先插入再查询，新插入的数据主键往往比已插入的数据主键更大。使用本系统进行插入时，如果要一次性插入多行，系统会先对要插入的行按主键进行从小到大排序，这样每插入一行，新插入行的主键就是最大的。

更新主索引时需要在 B+树中逐层递归，对于遇到的每个非叶节点，都要在其所有子节点中找出继续递归的入口，此时即判断是否为上述情况，即新插入的键比结点中已有的键都大，若是就直接沿最右侧子节点递归，否则才执行二分查询。这样，对于典型的数据库使用情况，将插入一条记录的复杂度从$O(n \log n \log m)$优化为$O(n \log n)$，其中n是已有总记录数，m是每页承载的记录数。

系统管理模块（TableMgr 类）

以上的若干类都要使用一些元信息进行初始化，例如某表包含哪些列、哪些索引等。为解决此问题，我实现了若干元信息已知的系统表，分别用于储存数据库名、表名、列、主索引、非簇集索引、外键约束、属性域约束的信息。当需要使用某表时，需要先以系统表中的信息初始化该表对应的 Table 对象。系统管理模块的作用就是管理这些元信息、维护 Table 对象，这样就可以实现数据库、表和索引的增删操作。

一．输入检查

系统管理模块的一大功能就是对输入进行检查，发现不合法操作时尽早报错，保证输入不合法指令后各表数据均不受影响。假若在进行具体表操作时才发现错误，将导致难以将表的状态恢复到进行操作之前。不合法操作可分为两大类，一是外键约束和属性域约束，这类约束有复杂但明确的行为，规定了何种情况合法、何种情况不合法；另一类是输入错误，例如访问了不存在的表或域、类型错误、违反非空限制等，此类错误种类很多，需要耐心排查。

对于外键约束的检查分为三种情况：

1. 插入时，应检查当前表有没有参照其他表的约束，若有，应确保当前表的外键在被参照表的主键中存在；
2. 删除时，应检查有没有其他表参照当前表的约束，若有，应确保当前表的主键在参照表的外键中不存在；
3. 更新时，先检查当前表有没有参照其他表的约束，若有，应确保当前表外键的新值在被参照表的主键中存在；然后，若当前表的主键发生了变化，还应检查有没有其他表参照当前表的约束，若有，应确保当前表主键的旧值在参照表的外键中不存在。

对于属性域约束，则只需在插入和删除时确保各域的新值满足属性域约束即可。

对于如何种类繁多输入错误，此处则不再详述。

本项目中，所有不合法操作均以异常的形式定义在 exception 文件夹中，这些异常均继承 C++ STL 的 runtime_error 异常，以利用其错误信息机制，方便向用户提供完善的错误信息。

二．多表查询

由于系统管理模块联合了多个表，此模块还负责处理多表的连接查询。优化多表查询的性能本质上就是优化多表查询中查询各表的顺序。本项目中，此顺序交由专门用于处理此问题的 Optimizer 类计算，详见本文“优化多表查询”一节。

每一条多表连接查询的查询条件可以分为两个集合C和D，其中$C_{i,j}$表示表T_i和T_j之间的限制条件，D_i表示表T_i自身的限制条件。系统管理模块在进行查询时，需要维护一个结果集$Result$，$Result$的初值为一条空记录，即$Result = \{\emptyset\}$。假设 Optimizer 类得出的表的顺序是T_0~T_{n-1}，程序遍历T_0~T_{n-1}。对于遍历到的每个表T_i，再遍历$Result$集中的每条记录$Result_j$，然后对T_i进行查询，查询的条件是$\bigcup_{k=0}^{i-1}$ *关于*$Result_j$ *特化*$(C_{k,i} \cup C_{i,k}) \cup D_i$，其中“关于$Result_j$特化”意为用$Result_j$中$T_k$的结果代入$C_{k,i} \cup C_{i,k}$。将$Result_j$中原有的信息并入查询结果集内每一条记录中，将新的结果集记为$Result'_j$，更新$Result$为$Result \leftarrow \bigcup_j Result'_j$。然后进行下一轮迭代。

优化多表查询（Optimizer 类）

如上文所述，设$|T| = n$，平均$|Result| = m$，平均$|T_i| = p$。考虑多表连接的典型使用情

况下，表间限制条件$C_{i,j}$以“=”限制为主，各轮迭代间$|Result|$变化较小。按上文的查询方式，共需进行$O(nm)$次表查询。如果查询表T_i使用了T_i的索引，单次查询复杂度为$O(\log p)$；如果查询表T_i没有使用了T_i的索引，单次查询复杂度为$O(p)$。所以优化目标即尽量使用各表的索引。

将各表视作一张有向图$G=(V,E)$，其中$V=T$，$E=\{T_i \to T_j | C_{i,j} \cup C_{j,i}$ *可用T_j的某索引*$\}$。要做的即是找出查询序列T_0~T_{n-1}尽量使得$\forall i, \exists j < i, T_j \to T_i \in E$。受求有向无环图（DAG）拓扑序的算法启发，Optimizer 类使用的算法是：每次找出入度最小的T_i添加进结果序列，然后删除T_i所有的出边，继续迭代直到所有表被添加进结果序列，这样即可找出相对较优的结果。对于典型的多表连接查询使用情况，图G都是链状的，此算法可以保证求出最优解，即对每个表进行查询时都能使用其索引。这样总的查询复杂度为$O(nm\log p)$。

聚集查询（Aggregate 类）

所有聚集函数，无论是 MIN、MAX、SUM 还是 AVG，都可视作一个状态机。其求值都可分为两个步骤：先是不断输入所聚集的记录，改变内部状态；再是进行输出。只要定义了这两步，以及内部状态的初值，即可实现一个聚集函数。各聚集函数可归纳如下：

函数名	内部状态初值	输入值x后的操作	输出y时的操作
$\mathrm{SUM}(x)$	$x_0=0$	$x_0 \leftarrow x_0+x$	$y=x_0$
$\mathrm{AVG}(x)$	$x_0=0, n=0$	$x_0 \leftarrow x_0+x, n \leftarrow n+1$	$y=x_0/n$
$\mathrm{MIN}(x)$	$x_0=+\infty$	$x_0 \leftarrow \min(x_0,x)$	$y=x_0$
$\mathrm{MAX}(x)$	$x_0=-\infty$	$x_0 \leftarrow \max(x_0,x)$	$y=x_0$
x（无函数）	$x_0=NULL$	$x_0 \leftarrow x$	$y=x_0$

Aggregate 类中对于每个聚集函数均定义了其“读入值x后的操作”和“输出y时的操作”。实际求值时，先向 Aggregate 类输入每个记录中所涉及域的值，然后从 Aggregate 类获取输出即可。唯一需要注意的是，若输入为空值则需忽略。

注：由于作业说明中只提到了 SUM、AVG、MIN、MAX 这四个聚集函数，所以我没有实现 COUNT 函数，但 COUNT 函数也完全符合上面的分析，很容易实现。由于作业检查时标准中提到了 COUNT，特作此说明。

结果排序及分组聚集查询

由于本系统中所用到的各个数据类型都已定义了比较其值大小的逻辑，实现结果排序时很简单的，使用 C++ STL 中的排序函数即可。

另一方面，分组聚集查询完全可以基于排序实现。按分组中所依据的域进行排序后，结果集里属于不同分组的记录自然就分开了。接着即可使用上文“聚集查询”一节的方法对每个组分别进行聚集函数的求值。

此外，由于结果排序和分组聚集查询是有可能同时出现在同一条查询语句中的，所以“分组聚集查询”不能破坏“结果排序”的结果。要么先进行“分组聚集查询”的排序，再进行“结果排序”的排序；要么进行“分组聚集查询”的排序时使用稳定排序（stable sort），本系统使用的是后一种方法。

注：由于作业说明中只提到了 ORDER BY 而没提到 ORDER BY DESC，所以我只实现了

一个方向的排序，若要实现逆序，只需修改一下比较函数即可。由于作业检查时标准中提到了 ORDER BY DESC，特作此说明。

SQL 解析模块

我利用 ANTLR 4 实现了 SQL 解析模块。实现中，我对部分文法进行了改进，与作业要求的《文法规则》中规定的有少许不同，在此特别列出：

1. 数字字面量现支持小数和负数；
2. 字符串字面量现支持“\'”和“\\”转义；
3. 主索引和非簇集索引现均支持使用超过一个域作为键；
4. 现允许定义空表；
5. INT 类型现可指定长度（用于指明输出表格中相应列的宽度）；
6. 现允许在查询、删除、修改中省略 where 从句。

I/O 模块

本系统支持两种不同的 I/O 模式：一，读入 SQL 文件，输出 CSV 文件；二，命令行交互式读入，输出易于人类判读的字符表格。这增强了整个系统的用户友好性。程序利用 Linux 的 isatty 机制判断 stdin 是文件还是终端，若为文件则进入模式一，若为终端则进入模式二。

在模式一中，整个文件被用作 SQL 解析模块的输入，如果发现了语法错误，则直接放弃执行整个文件。对于查询的结果，每行输出一条记录，记录中不同的域以逗号分隔（即 CSV 格式），不同查询间以空行分隔。

在模式二中，系统不断接受用户输入的行，如果用户的输入不以分号结尾，则提醒用户接着输入（命令提示符会发生变化）；如果用户的输入以分号结尾，则连同之前还未处理的部分一并交由 SQL 解析模块处理。如果发现了语法错误，不退出程序，而是允许用户接着输入。对于查询的结果，首先计算各列所需的宽度（宽度受标题宽度、此列最长结果宽度及 INT 类型中用户指定的宽度影响），然后用 ASCII 字符画成表格输出。

不管是模式一还是模式二，查询结果均输出到 stdout，错误信息和提示信息均输出到 stderr，这样如果用户将 stdout 重定向到文件，文件中将只记录查询结果，而错误信息和提示信息还会出现在屏幕上。

测试

本项目依托 Google Mock / Google Test 框架编写了单元测试共 155 条，覆盖各个数据类型、文件系统、特化的内存页、单表操作、系统（多表）管理、SQL 解析器、I/O，全部通过。本系统的所有功能当然都是能正常使用的。对于许多非法输入，本系统也有一定地健壮性。

在单元测试中，我着重测试了难以通过直接运行程序复现的情况，例如：对于涉及内存页缓存替换的测试，我通过减少缓存大小，通过少量数据页测试了内存页的替换；对于涉及索引的测试，我通过增加每条记录的大小，通过少量记录测试了索引中具有较多结点的情况。对于系统中索引等较为复杂的部分，我还根据 GNU 工具链中原生的代码覆盖率分析工具 gcov 的指引，补测了一些一开始没被覆盖到的边界情况。

此外，我还使用随机数据对系统进行了性能测试，并使用 GNU 工具链中原生的 gprof 性能分析工具分析各函数耗时，结果发现大量的时间花在了以字符串为键的 unordered_map（哈希表）上。发现此问题后，我将许多冗余哈希表替换成列表，并将另外某些哈希表改成了懒惰求值的形式（实现了 LazyMap 类并替换之）。此外，对于一处十分耗时的冗余列表赋值，我也将其改成了懒惰求值的形式（实现了 VectorRef 类并替换之）。

注：我曾在简要报告中提到我的系统有性能问题，但其实并没有。当初产生误判是因为性能分析器本身太慢了。

使用的第三方工具

1. ANTLR 4 文法解析器生成工具。即一个类似 Lex/YACC 的工具，用于生成 SQL 解析器；
2. Google Mock / Google Test 单元测试框架。仅用于测试，不包含于产品程序中。

本项目的 GitHub 仓库

https://github.com/roastduck/db