Московский авиационный институт
(государственный технический университет)

Курсовой проект
по курсу

«Архитектура ЭВМ, системное программное обеспечение»

2 семестр
Задание 6

Выполнил: студент гр. 08-102
Ридли Михаил

Проверил: Макаров Н.К.

Москва, 2011 год

Постановка задачи

Разработать последовательную структуру данных для представления простейшей базы данных (здесь и далее – БД) на файлах СП Си. Составить программу генерации внешнего нетекстового файла заданной структуры, содержащего минимальный представительный набор данных (не менее 10 записей). Распечатать содержимое файла в виде таблицы. Выполнить над файлом заданное действие и распечатать результат. Действие по выборке данных из файла оформить в виде отдельной программы.

База данных содержит сведения о составе комплектующих личных ПЭВМ в студенческой группе: фамилия владельца, число и тип процессоров, объём памяти, тип видеоконтроллера и объём видеопамяти; тип, число и ёмкость винчестеров, операционная система. В соответствии с вариантом требуется составить списки мультимедийных и бездисковых компьютеров.

Использование реализации

Задача естественным образом распадается на несколько частей, выполняющих непересекающиеся действия. Эти части выполнены в виде отдельных исполняемых файлов:

Исполняемый файл	Назначение
generate	Наполнение БД ранее заготовленными образцами
input	Пользовательское наполнение БД
print	Вывод содержимого БД
execute	Вывод содержимого БД, удовлетворяющего условию варианта

Если программа представлена исходным кодом, её нужно собрать с помощью команды `make`. После этого она готова к применению.

Типичный сценарий использования программы – выполнить `generate` или `input` для заполнения БД, после чего изучить её содержимое с помощью `print` и `execute`.

Организация исходного кода

Исходный код программы распадается на две категории: реализацию СУБД и управляющую часть, исполняющую задачу при помощи интерфейса СУБД.

Дерево файлов	Значение
./db	Реализация СУБД
api.h	Интерфейс СУБД
clause.h clause.c	Предикаты (для `SELECT`)
io.h io.c	Отображение таблиц БД на файловую систему
iterator.h iterator.c	Итераторы для обхода БД
table.h table.c	Отношения БД в виде таблиц
tuple.h tuple.c	Элементы отношений (строки таблицы, известные как кортежи)
adders.h adders.c	Процедуры наполнения БД
execute.c	Вывод записей, удовлетворяющих заданию
generate.c	Наполнение БД образцами записей
input.c	Осуществление пользовательского ввода
print.c	Вывод содержимого БД
printers.h	Макросы для вывода содержимого БД
structure.h structure.c	Структура БД, указанная в задании

Управляющая часть расположена в корне проекта, а реализация СУБД – в подкаталоге `db`. Управляющая часть получает доступ к интерфейсу СУБД через файл `db/api.h`.

Для управления сборкой проекта используется `Makefile`, запускаемый из основного каталога проекта. Он служит автоматизации рутинных действий, таких как сборка проекта, упаковка, разные виды очистки каталога от образованных в процессе сборки файлов и так далее.

Реализация СУБД

Общая картина

Как и следует из названия, NanoSQL – крошечная СУБД, реализующая интерфейс подмножества языка запросов SQL.

При реализации SQL-СУРБД принято работать с командами, записанными на SQL. В NanoSQL используется более легковесный подход: программисту предоставляется API, похожий на SQL. К примеру, для создания таблицы используется функция, объявленная как `createTable(table *table, const char *name, tuple_type *scheme)`.

В сущности, данные в NanoSQL хранятся в списочных структурах, а для их отбора используется нечто вроде `filter` из функционального программирования.

В файлах `tuple` (здесь и далее подразумевается раскрытие до `tuple.h` и `tuple.c`) хранится описание кортежей (`tuple_type`). Кортежи задают структуру строк таблицы, но не конкретные наборы элементов.

Структура `iterator`, объявленная в одноимённых файлах, представляет последовательность кортежей и организует навигацию по строкам таблицы.

В свою очередь, файлы `clause` предоставляют предикаты для `SELECT`, такие как проверка равенства или положительности полей, и составные предикаты вроде `AND`.

Реализация операций ввода-вывода хранится в файлах `io`. Кортеж – это просто динамический массив `size` байтов. Это значит, что кортежи можно напрямую записывать и считывать из файла. Здесь же находится отладочная функция `printTable(table *)`, с помощью которой удобно исследовать структуру таблицы.

Для более детального исследования кода СУБД достаточно пояснить содержимое заголовочных файлов, поскольку файлы реализации включают главным образом тривиальные, рутинные и безынтересные алгоритмы.

db/api.h – интерфейс СУБД

Создан как интерфейс СУБД NanoSQL, подключаемый внешними программами. Выполняет подключение заголовочных файлов `tuple.h`, `table.h`, `iterator.h`, `io.h`.

db/tuple.h – кортежи (строки таблицы)

Реализует кортежи `tuple_type`, состоящие из массива полей `field *fields`, их количества size_t `numFields` и размера кортежа в байтах size_t `size`.

Каждое поле характеризуется именем `char name[64]`, типом поля `field_type type` (принимающего значения `INTEGER`, `LOGICAL`, `VARCHAR` перечисления

field_type), байтовым смещением поля size_t offset, размером поля в байтах size.

Примечательно, что tuple_type – это не набор конкретных numFields значений, а схема, описывающая структуру таблицы.

byte *elem(byte *data, const field *field)

Возвращает адрес байта, отстоящего от data на field->offset байт (иначе говоря, указатель на элемент field->offset массива data). Используется для записи данных в таблицу (см. adders.c).

void copyTuple(tuple_type *type, byte *dest, byte *src)

Копирование кортежа размером type->size байт из src в dest.

void createScheme(tuple_type *scheme)

Инициализация пустой схемы scheme для последующего наполнения полей.

void destroyScheme(tuple_type *scheme)

Освобождение памяти от кортежа scheme.

void addField(tuple_type *scheme, const char *name, field_type type, size_t size)

Добавление в кортеж scheme поля типа type, занимающего size байт памяти, под именем name. Функции приходится перераспределять память под новую схему и увеличивать её numFields и size.

table.h – отношения БД в виде таблицы

Для представления таблиц задана структура table, содержащая массив кортежей byte *tuples, их количество size_t numTuples, количество выделенной для кортежей таблицы памяти size_t allocated, имя таблицы char name[64].

void createTable(table *table, const char *name, tuple_type *scheme)

Создание таблицы table с именем name, чьи строки заданы схемой scheme.

Под хранение кортежей в функции изначально выделяется 1*(scheme->size) байт памяти, где вместо единицы может быть любая константа, оптимальная для решаемой задачи.

void destroyTable(table *table)

Высвобождение памяти, выделенной под таблицу.

byte *insertTuple(table *table)

Функция возвращает указатель на первый свободный байт в памяти, выделенной под кортежи таблицы.

Если памяти недостаточно (количество кортежей равно количеству выделенной под них памяти), то выделяется в два раза больше памяти, чем было выделено ранее. Это позволяет свести к минимуму количество перераспределений памяти, хотя для некоторых задач уместнее другие алгоритмы (к примеру, логарифмическое увеличение выделения памяти).

void deleteTuple(table *table, byte *tuple)

Удаление из таблицы `table` строки, начинающейся в памяти с позиции `tuple`. Из-за последовательной схемы хранения это требует перемещения участков памяти.

iterator.h – итераторы (последовательности кортежей) для обхода БД

Существуют 4 типа итераторов, определяемых в перечислении `iterator_type`: `TABLE` (обход таблицы), `JOIN` (обход нескольких таблиц), `SELECT` (обход с фильтрацией), `FIXED` (конкретная ячейка).

Структура iterator задаётся типом итератора `iterator_type type`, признаком наличия итерируемых данных `notEOI`, кортежем `union tuples`, за счёт которого `iterator` реализует списочное поведение.

В объединении `tuples` поддерживаются все четыре вида итераторов в одноимённых полях: `table` (указатель на таблицу `table *table` и местоположение первого байта текущей строки `byte *tuple`), `join` (массив итераторов `iterator *args` и их количество `size_t numArgs`), `select` (указатель на итератор `iterator *from` и на предикат `clause *clause`), `fixed` (`iterator *from`).

Ясно, что итератор – это обыкновенная последовательность кортежей: каждый итератор содержит кортеж и указатель на другой итератор.

void createTableIterator(iterator *iter, table *table)

Создаёт итератор `iter` для обхода таблицы `table`.

void createJoinIterator(iterator *iter, int numTables, table *tables)

Создаёт итератор `iter` по декартову произведению `numTables` таблиц `tables`. Для этого нужно помнить, в какой таблице в последний раз производился перебор строк, при необходимости переходя к более «ранней» таблице.

void createJoinIteratorI(iterator *iter, int numIters, iterator *from)

Создаёт итератор `iter` по декартову произведению `numIters` итераторов `from`. Это более общая версия многотабличного итератора, создаваемого функцией `createJoinIterator`. Пример такого подхода находится в макросе `printComputer` из `printers.h`: для вывода всех винчестеров, ЦПУ и видеокарт, принадлежащих компьютеру, используется произведение итератора `FIXED`, указывающего на конкретный компьютер, и итератора `TABLE`, проходящего по таблице винчестеров/ЦПУ/ видеокарт.

void createSelectIterator(iterator *iter, iterator *from, clause *clause)

Создаёт итератор `iter`, обходящий только те кортежи итератора `from`, которые удовлетворяют предикату `clause`. Это делается построчным проходом `from` до тех пор, пока строка не будет удовлетворять условию.

void createFixedIterator(iterator *iter, iterator *from)

Создаёт итератор `iter`, всегда указывающий на строку, на которую итератор `from` ссылался в момент вызова функции.

void freeIterator(iterator *iter)

Высвобождение памяти, занимаемой итератором `iter`.

bool first(iterator *iter)

Устанавливает итератор `iter` в начальную позицию.

Функция отдельно разбирает все четыре случая, соответствующих разным категориям итераторов: к примеру, для JOIN она выставляет итераторы, участвующие в слиянии, в начальную позицию, а для SELECT возвращает первый элемент, удовлетворяющий некоторому условию.

bool next(iterator *iter)

Передвигает итератор `iter` на следующую позицию.

В случае с `JOIN`-итератором на одну позицию передвигается последний итератор, и в случае его конца – на одну позицию подвигается предпоследний итератор из `iter->tuples->args` и так далее.

byte *get(iterator *iter, const char *tableName, const field *field)

Возвращает указатель на первый байт памяти поля `field` в строке, на которую указывает итератор `iter`. Причём всё это выполняется для таблицы под названием `tableName`.

bool test(iterator *iter, clause *clause)

Проверка соответствия предикату `clause` строки, на которую указывает итератор `iter`. Функция используется для реализации `SELECT`-итератора.

int selectCount(iterator *iter)

Возвращает количество строк, которые можно обойти итератором `iter`.

clause.h – предикаты для SELECT

В NanoSQL реализовано только 5 предикатов: проверка поля на равенство FEQ, проверка поля на равенство нулю FZERO, проверка значения в поле на положительность FPOS, бинарный предикат AND, соответствующий логическому «и». Все эти предикаты заданы в перечислении `clause_type`.

Предикаты представляются структурой `struct_clause`, имеющей поле `clause_type type` и `union data`.

Объединение union_data имеет три структурные интерпретации своего значения: `binarySimple`, `unarySimple`, `binaryComplex`.

Структура `binarySimple` состоит из имени первой таблицы `char table1[64]`, названия поля в первой таблице `field *arg1`, имени второй таблицы `char table2[64]` и названия поля во второй таблице `field *arg2`. Пример такого предиката – «`arg1` больше `arg2`?».

`unarySimple` аналогична `binarySimple`, но является унарной. Пример – проверка поля на неравенство нулю.

`binaryComplex` – предикат, оперирующий предикатами `struct tag_clause *clause1` и `struct tag_clause *clause2`. Пример – логическое «и», истинное лишь тогда, когда оба предиката истинны.

Важно отметить, что непосредственная проверка истинностного значения реализована не в файлах `clause`, а в функции `test` из `iterator.c`, которая используется в функции `next`.

void createEqClause(clause *result, const char *arg1table, field *arg1,

const char *arg2table, field *arg2)

Создание предиката `result`, истинного в случае равенства значения полей `arg1` и `arg2` из таблиц `arg1table` и `arg2table` соответственно.

void createZeroClause(clause *result, const char *argTable, field *arg)

Создание предиката `result`, истинного в случае равенства нулю поля `arg` таблицы `argTable`.

void createPosClause(clause *result, const char *argTable, field *arg)

Создание предиката `result`, истинного в случае, если значение поля `arg` таблицы `argTable` положительно.

void createAndClause(clause *result, clause *arg1, clause *arg2)

Создание предиката `result`, истинного только тогда, когда одновременно истинны предикаты `arg1` и `arg2`.

void freeClause(clause *clause)

Освобождение памяти, занимаемой предикатом `clause`.

io.h – отображение таблиц БД на файловую систему

Содержимое этого файла и `io.c` определяет способ хранения базы данных в файловой системе.

void printTable(table *table)

Вывод сведений о таблице `table`: имя таблицы, количество полей кортежа, его размер в байтах, число строк, заданных этим кортежем. Для каждого поля выводится его байтовое смещение, имя, тип, количество элементов этого типа в поле.

void saveTable(table *table, FILE *f)

Запись таблицы `table` в файл `f`. Изначально поведение функции похоже на поведение printTable. Как только записаны характеристики таблицы и полей, в файл выводятся строки таблицы и их количество.

void loadTable(table *table, FILE *f)

Функция загружает таблицу из файла `f` в `table`, выполняя действия, обратные действиям saveTable: инициализацию схемы, считывание характеристик таблицы и полей, добавление полей к схеме, создание таблицы, её наполнение строками в соответствии со схемой.

Управляющая программа

structure.h – структура БД

В файле заданы схемы (кортежи) `tuple_type *hdd, *cpu, *video, *computer`, по которым впоследствии буду созданы таблицы table `hddt, cput, videot, computert`.

void createStructure()

Функция создаёт структуру БД и помещает её в перечисленные выше глобальные переменные.

БД имеет следующую структуру.

Таблица	Имя	Поле	Тип элемента	Количество
hddt	HDD	Capacity	INTEGER	1
		Type	VARCHAR	5
		ComputerID	INTEGER	1
cput	CPU	Frequency	INTEGER	1
		Manufacturer	VARCHAR	32
		ComputerID	INTEGER	1
videot	Video	Memory	INTEGER	1
		Type	VARCHAR	4
		ComputerID	INTEGER	1
computert	Computer	ID	INTEGER	1
		RAM	INTEGER	1
		Monitor	LOGICAL	1
		NumHDD	INTEGER	1
		NumCPU	INTEGER	1
		NumVideo	INTEGER	1
		OS	VARCHAR	32
		Owner	VARCHAR	32

adders.h – процедуры наполнения БД

Функции этого файла позволяют наполнять таблицы содержимым в соответствии с указанной схемой, абстрагируя низкоуровневые детали реализации.

void addHDD(table *hddt, tuple_type *hdd, const int capacity, const char *type, const int computer)

Добавление винчестера в таблицу `hddt` по схеме `hdd`. Винчестер имеет ёмкость `capacity`, тип `type` и предназначается для компьютера с идентификационным номером `computer`.

void addCPU(table *cput, tuple_type *cpu, const int frequency, const char *manufacturer, const int computer)

Добавление ЦПУ в таблицу `cput` по схеме `cpu`. ЦПУ имеет тактовую частоту `frequency`, произведено `manufacturer` и предназначается для компьютера с идентификационным номером `computer`.

void addVideo(table *videot, tuple_type *video, const int memory, const char *type, const int computer)

Добавление видеокарты в таблицу `videot` по схеме `video`. Видеокарта имеет ёмкость видеопамяти `memory`, тип `type` и предназначается для компьютера с идентификационным номером `computer`.

void addComputer(table *compt, tuple_type *comp, const int id, const int ram, const bool monitor, const int hdd, const int cpu, const int video, const char *os, const char *owner)

Добавление компьютера в таблицу `compt` по схеме `comp`. Компьютер имеет идентификационный номер `id`, `ram` Мб оперативной памяти, имеет или нет монитор (`monitor`), имеет `hdd` винчестеров, `cpu` процессоров, `video` видеокарт, операционную систему `os` и владельца `owner`.

printers.h – макросы для вывода содержимого БД

Файл состоит из нескольких макросов, раскрывающихся в инструкции для вывода чего-либо. Почти каждый макрос принимает итератор и некий объект, значения некоторых полей которого он выводит.

printOS(iter, comp)

Печать названия операционной системы компьютера `comp`.

printID(iter, comp)

Печать идентификационного номера компьютера `comp`.

printMonitor(iter, comp)

Печать признака наличия монитора у компьютера `comp`.
printOwner(iter, comp)

Печать фамилии владельца компьютера `comp`.
printQtyCPU(iter, comp)

Печать количества ЦПУ компьютера `comp`.
printCPU(iter, cpu)

Печать сведений о процессоре `cpu`.

printRAM(iter, comp)

Печать ёмкости оперативной памяти компьютера `comp`.
printQtyGPU(iter, comp)

Печать количества видеокарт в компьютере `comp`.

printGPU(iter, video)

Печать сведений о видеокарте `video`.
printQtyHDD(iter, comp)

Печать количества винчестеров компьютера `comp`.
printHDD(iter, hdd)

Печать сведений о винчестере `hdd`.
printComputer(selector, computer, hdd, video, cpu)

Печать сведений о компьютере `computer`. `hdd`, `video`, `cpu` – схемы винчестера, видеокарты, ЦПУ соответственно.

generate.c – наполнение БД образцами записей

Этот файл – один из тех четырёх, что должны при компиляции порождать одноимённые исполняемые файлы. Программа, образовываемая этим файлом, создаёт небольшую БД, заполняя её некоторыми примерами записей.

Его функциональность тривиальна: с помощью `createStructure()` генерируется структура БД, после чего `createData()` создаёт записи и записывает их в файл.

void createData()

Функция создаёт тестовые данные в глобальных переменных `structure.h`, открывает файл db.bin, записывает в него все таблицы, освобождает от них память, закрывает файл db.bin.

input.c – осуществление пользовательского ввода

Этот файл, порождающий исполняемый файл `input`, по своей сути поход на `generate.c` с тем лишь отличием, что сведения о компьютерах вводятся пользователем.

promptData(FILE *in, FILE *out)

Функция читает данные из `in`, выводя запросы в `out`. Её алгоритм очень прост: информация об отдельных компьютерах считывается во внешнем цикле. Если есть совокупности записей неизвестной заранее длины, то эти записи вводятся в цикле.

print.c – вывод содержимого БД

Исполняемый файл `print`, полученный после компиляции `print.c`, выводит всю БД в понятном пользователю виде.

void printData()

Печатает сведения обо всех компьютерах, записи которых хранятся в `db.bin`. Вывод осуществляется с помощью итератора, пробегающего всю таблицу `computert` и печатающую характеристики каждого компьютера, при необходимости обращаясь к вспомогательным таблицам.

execute.c – вывод записей, удовлетворяющих заданию

Программа `execute` считывает данные из файла `db.bin`, после чего выводит список мультимедийных компьютеров (тех, у которых есть хотя бы одна видеокарта и монитор) и список бездисковых компьютеров (тех, у которых отсутствует винчестер).

void processData()

Алгоритм печати сведений только о тех компьютерах, которые удовлетворяют условию задачи, очевиден: достаточно создать предикат положительности поля количества видеокарт и значения поля наличия монитора, затем соединив их предикатом `AND`. После этого остаётся воспользоваться `createSelectIterator()` и пройтись по полученному итератору.

Аналогично выводятся бездисковые компьютеры: для этого используется `createZeroClause()`, сравнивающее поле количества винчестеров с нулём.

Выводы

Несмотря на концептуальную пригодность, у СУБД NanoSQL есть существенные недостатки, в основном обусловленные смещением баланса удобства в сторону разработчика СУБД, а не её пользователя.

Среди них особенно примечательна скудная функциональность: реализовано всего 3 типа полей (когда в реальности их более 15), лишь несколько конструкций из всего многообразия доступных в SQL, не реализован разбор SQL-синтаксиса и так далее.

Существуют и значительные технические недостатки. К ним можно отнести, к примеру, хранение базы данных в памяти и отображение всей БД на один файл. Очевидно, это не очень хорошо, когда файл БД весит 102 Гб и не помещается в память. Однако для конкретной задачи это приемлемо: в условии указано, что БД заполнено списком компьютеров студентов учебной группы, которая едва ли может насчитывать даже 100 человек.

Разумеется, в проекте наличествуют и иные неудобства: к примеру, в printers.h доступ к значениям полей осуществляется по индексу поля, что несколько неудобно. Это легко исправляется введением функции `searchField(table *table, char *fieldName)`, находящей поле по имени и возвращающей его индекс в кортеже, однако особой необходимости в этом не было.

С другой стороны, у проекта есть и значительные преимущества, такие как достигаемая за счёт NanoSQL гибкость и большая наглядность: в сущности, для реализации других вариантов задачи достаточно переопределить структуру, функции ввода-вывода и содержимое файла `execute.c`.

Безусловно, реализация NanoSQL оказалась полезным занятием, и преподнесла урок повышения удобства за счёт слияния воедино данных и алгоритмов их обработки.

Использованные источники

Керниган, Ритчи, Язык программирования Си
ISO/IEC 9899:201x, Programming languages – C [C99]
Зайцев, Методическое руководство «Практикум по циклу дисциплин «Информатика», часть 2»